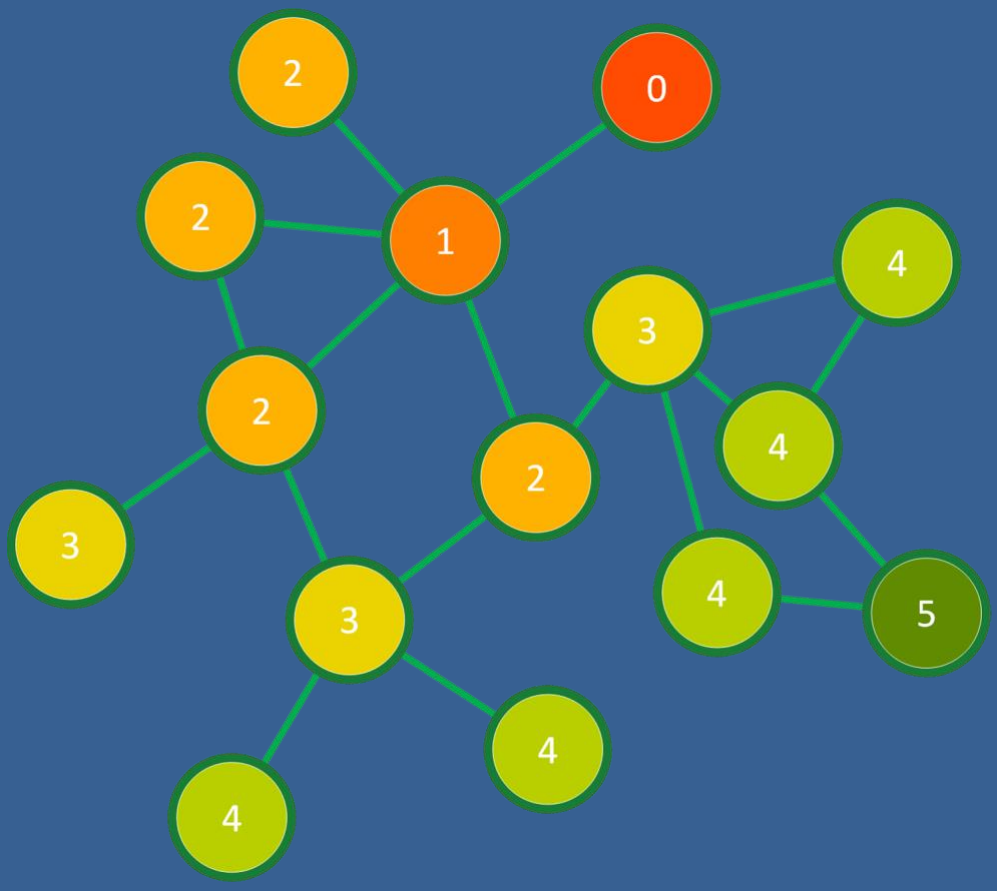
Андрій Креневич

АЛГОРИТМИ І СТРУКТУРИ ДАНИХ

Підручник



КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА

2021

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА

КРЕНЕВИЧ А.П.

АЛГОРИТМИ І СТРУКТУРИ ДАНИХ

Підручник

КИЇВ 2021

УДК 004.438С(075.8); 519.682

Рецензенти:
доктор фіз.-мат. наук Розора І.В.
доктор фіз.-мат. наук Бондаренко Є.В.

Рекомендовано до друку вченою радою
механіко-математичного факультету
(протокол № 19 від 20 травня 2021 року)

Креневич А.П.
Алгоритми і структури даних. Підручник. – К.: ВПЦ "Київський Університет", 2021. – 200 с.

Підручник присвячений вивченню алгоритмів і структур даних у програмуванні. У ньому викладено про рекурентні співвідношення та рекурсію, правила та механізми оцінки складності алгоритмів, алгоритми пошуку та сортування, концепції хешування та хеш-таблиці, лінійні структури даних, дерева, графи та їхнє застосування.

Посібник складено з урахуванням досвіду викладання програмування на механіко-математичному факультеті Київського національного університету імені Тараса Шевченка.

Для студентів університетів та викладачів, що викладають дисципліну «Алгоритми і структури даних».

ЗМІСТ

ВСТУП

Алгоритм – це основа програмування. Він описує послідовність дій, які має виконати програма для розв'язання поставленої задачі. У свою чергу, структури даних дозволяють оптимально розташовувати дані необхідні програмі. Вибір певного алгоритму диктує необхідність використання певного переліку структур даних для оптимального розв'язання задачі. Структури даних не розглядаються у програмуванні як окремі незалежні концепції. Частіше за все їх асоціюють з певними алгоритмами, невід'ємною частиною яких вони є. Тому у програмуванні алгоритми та структури даних – поняття, що завжди стоять поруч.

У цьому підручнику ви познайомитеся з базовими аспектами побудови алгоритмів, основними концепціями та структурами даних. У ньому наведені різноманітні реалізації абстрактних типів даних, починаючи від найпростіших лінійних структур даних, таких як стек, черга чи зв'язний список, закінчуючи деревами та графами, та їхнім застосуванням у різноманітних алгоритмах.

Підручник складається з семи розділів кожен з яких присвячено певній темі дисципліни. У свою чергу, розділи поділені на параграфи, нумерація яких включає номер розділу якому належить параграф. Нумерація всіх об'єктів у посібнику (означень, лістингів програм, прикладів, рисунків тощо) складається з трьох частин: номеру розділу, номеру параграфу та порядкового номеру об'єкта у цьому параграфі. Для відображення фрагментів коду програм, інструкцій та ідентифікаторів, що використовуються при написанні програм, застосовується моноширинний шрифт з підсвічуванням (різними кольорами), аналогічним до того, яке використовується у сучасних інтегрованих середовищах розробки. Ці заходи мають значно полегшити процес сприйняття матеріалу.

Матеріал підручника пояснюється на великій кількості прикладів, більшість з яких супроводжуються детальними описами алгоритмів, а ключові моменти програм пояснені у коментарях. Вихідний код усіх наведених у підручнику програм опублікований на GitHub сторінці автора, звідки при бажанні, читач може завантажити його скориставшись посиланням https://github.com/krenevych/algo. Репозиторій за цим посиланням є проектом, орієнтованим на застосування інтегрованого середовища програмування PyCharm Community, що вільно розповсюджується. Структура репозиторія повністю відповідає структурі підручника. Всі програми розташовані у папці `source`, у якій міститься перелік папок, кожна з яких відповідає певному розділу підручника. Далі кожна папка містить перелік папок, що відповідають параграфам підручника, котрі, в свою чергу, містяться файли з вихідним кодом програм. Сподіваємось така організація прикладів дозволить читачу без особливих труднощів знаходити програми наведені у підручнику.

Матеріал підручника передбачає, що читач володіє мовою програмування Python та парадигмою об'єктно-орієнтованого програмування.

Автор висловлює щиру подяку колегам кафедри математичної фізики за корисні поради, конструктивну критику та допомогу при створенні цього посібника.

Ваші відгуки і побажання щодо змісту підручника
надсилайте на електронну адресу автора:
krenevych@knu.ua

РОЗДІЛ 1. РЕКУРЕНТНІ СПІВВІДНОШЕННЯ ТА РЕКУРСІЯ

□□□
□□

§1.1. Рекурентні співвідношення

Рекурентні співвідношення мають надзвичайно важливе значення для програмування. Вони використовуються у аналізі алгоритмів, наближених обчисленнях, динамічному програмуванні, тощо. Тому вивчення курсу пропонуємо читачу розпочати саме з цього розділу.

1.1.1. Рекурентне співвідношення першого порядку

Нехай $\{a_k : k \geq 0\}$ деяка послідовність дійсних чисел.

Означення 1.1.1. Послідовність $\{a_k : k \geq 0\}$ називається заданою **рекурентним співвідношенням першого порядку**, якщо явно задано її перший член a_0, а кожен наступний член a_k цієї послідовності визначається деякою залежністю через її попередній член a_{k-1}, тобто

$$\begin{cases} a_0 = u \\ a_k = f(n, p,\ a_{k-1}),\ k \geq 1 \end{cases}$$

де u задане (початкове) числове значення, p – деякий сталий параметр або набір параметрів, f – функція, задана аналітично у вигляді арифметичного виразу, що складається з операцій доступних для виконання з допомогою мови програмування (зокрема у нашому випадку Python).

Для прикладу розглянемо послідовність $\{a_k = k! : k \geq 0\}$. Її можна задати рекурентним співвідношенням першого порядку. Дійсно, враховуючи означення факторіалу отримаємо

$$\begin{cases} a_0 = 1 \\ a_k = k a_{k-1},\ k \geq 1 \end{cases}$$

Маючи рекурентне співвідношення можна знайти який завгодно член послідовності. Наприклад, якщо потрібно знайти a_5, то використовуючи рекурентні формули, послідовно від першого члена отримаємо

$$a_0 = 1$$
$$a_1 = 1 \cdot a_0 = 1 \cdot 1 = 1$$
$$a_2 = 2 \cdot a_1 = 2 \cdot 1 = 2$$
$$a_3 = 3 \cdot a_2 = 3 \cdot 2 = 6$$
$$a_4 = 4 \cdot a_3 = 4 \cdot 6 = 24$$
$$a_5 = 5 \cdot a_4 = 5 \cdot 24 = 120$$

З точки зору програмування, послідовність задана рекурентним співвідношенням значно зручніша, ніж задана у явному вигляді. Для обчислення членів послідовностей, заданих рекурентними співвідношеннями, використовують цикли.

Нехай послідовність a_k задана рекурентним співвідношенням

$$\begin{cases} a_0 = u \\ a_k = f(n, p,\ a_{k-1}),\ k \geq 1 \end{cases}$$

Тоді, після виконання коду

```python
a = u
for k in range(1, n + 1):
    a = f(k, p, a)
```

у змінній a буде міститися значення елемента a_n послідовності

Приклад 1.1.1. Для введеного з клавіатури значення n обчислити $n!$

Розв'язок. Як було зазначено раніше послідовність $a_k = k!$ може бути задана рекурентним співвідношенням

$$\begin{cases} a_0 = 1 \\ a_k = k a_{k-1},\ k \geq 1 \end{cases}$$

Тоді, згідно з вищенаведеним алгоритмом, отримаємо

Лістинг 1.1.1. Обчислення факторіалу натурального числа.

```
n = int(input("n = "))
a = 1                          # a = u
for k in range(1, n+1):
    a = k * a                  # a = f(k, p, a)
print ("%d! = %d" % (n, a))    # виводимо на екран результат
```

Приклад 1.1.2. Скласти програму для обчислення елементів послідовності

$$a_k = \frac{x^k}{k!},\ k \geq 0.$$

Розв'язок. Складемо рекурентне співвідношення для заданої послідовності. Легко бачити, що кожен член послідовності a_k є добутком чисел. Враховуючи це, обчислимо частку двох сусідніх членів послідовності. Для $k \geq 1$ отримаємо

$$\frac{a_k}{a_{k-1}} = \frac{x^k}{k!} \cdot \frac{(k-1)!}{x^{k-1}} = \frac{x}{k}$$

Звідки випливає, що для $k \geq 1$

$$a_k = \frac{x}{k} a_{k-1}$$

Отже ми отримали для послідовності a_k рекурентну формулу, у якій кожен член послідовності для всіх $k \geq 1$ визначається через попередній член a_{k-1}. Щоб задати рекурентне співвідношення, залишилося задати перший член a_0. Для цього просто підставимо 0 у вихідну формулу

$$a_0 = \frac{x^0}{0!} = 1.$$

Отже остаточно отримаємо рекурентне співвідношення першого порядку

$$\begin{cases} a_0 = 1 \\ a_k = \frac{x}{k} a_{k-1},\ k \geq 1 \end{cases}$$

Тоді, згідно з вищенаведеним алгоритмом, отримаємо програму

Лістинг 1.1.2.

```
n = int(input("n = "))
x = float(input("x = "))
a = 1
for k in range(1, n+1):
    a = x / k * a       # можна так: a *= x / k
print ("a = ", a)       # виводимо на екран результат
```

Зауважимо, що нумерація членів послідовності інколи починається не з 0, а з деякого натурального числа m, тобто $\{a_k : k \geq m\}$. Припустимо, що рекурентне співвідношення для цієї послідовності має вигляд

$$\begin{cases} a_m = u, \\ a_k = f(k, p, a_{k-1}), \qquad k \geq m+1. \end{cases}$$

Тоді для того, щоб отримати a_n, необхідно замінити наведений вище алгоритм на такий

```
a = u
for k in range(m + 1, n + 1):
    a = f(k, p, a)
```

який, отриманий заміною стартового значення у інструкції **range** на значення m + 1.

Приклад 1.1.3. Скласти програму обчислення суми:

$$S_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}.$$

Розв'язок. Зазначимо, що задане співвідношення має сенс тільки для $n \geq 1$. Складемо рекурентне співвідношення для послідовності $\{S_k : k \geq 1\}$. Помічаємо, що на відміну від попереднього прикладу, кожен член послідовності S_k є сумою елементів вигляду $1/i$, де i змінюється від 1 до k. Отже, для побудови рекурентного співвідношення знайдемо різницю двох сусідніх членів послідовності S_k. Для $k \geq 2$

$$S_k - S_{k-1} = 1/k$$

Підставляючи у вихідне співвідношення $k = 1$, отримаємо $S_1 = 1$. Отже, рекурентне співвідношення для послідовності S_k матиме вигляд:

$$\begin{cases} S_1 = 1 \\ S_k = S_{k-1} + \dfrac{1}{k},\ k \geq 2 \end{cases}$$

Аналогічно до попереднього прикладу, враховуючи, що нумерація членів послідовності починається з 1, а не з нуля, отримаємо програму.

Лістинг 1.1.3. Підрахунок суми

```
n = int(input("n = "))
S = 1
for k in range(2, n + 1):
    S += 1 / k
print("S = ", S)
```

Приклад 1.1.4. Скласти програму обчислення суми

$$S_n = \sum_{i=1}^{n} 2^{n-i} i^2,\ n \geq 1.$$

Розв'язок. Розглянемо послідовність

$$\left\{ S_k = \sum_{i=1}^{k} 2^{k-i} i^2 : k \geq 1 \right\}$$

Складемо для неї рекурентне співвідношення. Підставляючи $k = 1$, отримаємо $S_1 = 1$. Щоб отримати вираз для загального члена, розкриємо суму для $k \geq 2$

$$S_k = 2^k \left(\frac{1^2}{2^1} + \frac{2^2}{2^2} + \ldots + \frac{(k-1)^2}{2^{k-1}} + \frac{k^2}{2^k} \right) =$$
$$= 2 \cdot 2^{k-1} \left(\frac{1^2}{2^1} + \frac{2^2}{2^2} + \ldots + \frac{(k-1)^2}{2^{k-1}} + \frac{k^2}{2^k} \right) =$$
$$2 \cdot 2^{k-1} \left(\frac{1^2}{2^1} + \frac{2^2}{2^2} + \ldots + \frac{(k-1)^2}{2^{k-1}} \right) + 2 \cdot 2^{k-1} \frac{k^2}{2^k} = 2 \cdot S_{k-1} + k^2$$

Отже, рекурентне співвідношення для буде мати вигляд

$$\begin{cases} S_1 = 1, \\ S_k = 2 \cdot S_{k-1} + k^2,\ k \geq 2. \end{cases}$$

і відповідно програма

Лістинг 1.1.4.

```
n = int(input("n = "))
S = 1
```

```
for k in range(2, n + 1):
    S = 2 * S + k ** 2
print("S = ", S)
```

1.1.2. Рекурентні співвідношення старших порядків

Нехай $\{a_k : k \geq 0\}$ деяка послідовність дійсних чисел. m – деяке натуральне число більше за одиницю. Тоді

Означення 1.1.2. Послідовність $\{a_k : k \geq 0\}$ називається заданою рекурентним співвідношенням $\boldsymbol{m}$-го порядку, якщо

$$\begin{cases} a_0 = u,\ a_1 = v, \dots, a_{m-1} = w, \\ a_k = f(n, p,\ a_{k-1}, \dots, a_{k-m},),\ k \geq m \end{cases}$$

де $u, v, \dots, w$ – задані числові сталі, p – деякий сталий параметр або набір параметрів, f – функція, задана аналітично у вигляді арифметичного виразу, що складається з операцій доступних для виконання з допомогою мови програмування.

Найпоширенішим прикладом послідовності заданої рекурентним співвідношенням 2-го порядку є послідовність чисел Фібоначчі. Перші два члени цієї послідовності дорівнюють одиниці, а кожен наступний член є сумою двох попередніх

$$\begin{cases} F_0 = 1,\ F_1 = 1 \\ F_k = F_{k-1} + F_{k-2},\ k \geq 2 \end{cases}$$

Як і у випадку рекурентного співвідношення першого порядку, маючи рекурентне співвідношення можна знайти який завгодно член послідовності.

$$\begin{gathered} F_0 = 1,\ F_1 = 1 \\ F_2 = F_1 + F_0 = 1 + 1 = 2 \\ F_3 = F_2 + F_1 = 2 + 1 = 3 \\ F_4 = F_3 + F_2 = 3 + 2 = 5 \\ F_5 = F_4 + F_3 = 5 + 3 = 8 \\ F_6 = F_5 + F_4 = 5 + 3 = 13 \end{gathered}$$

Для обчислення елементів послідовності, заданої рекурентним співвідношенням вищого порядку, застосовується інший підхід ніж для співвідношень першого порядку.

Алгоритм наведемо на прикладі співвідношення 3-го порядку. Нехай послідовність a_k задана рекурентним співвідношенням

$$\begin{cases} a_0 = u,\ a_1 = v, a_2 = w, \\ a_k = f(k, p,\ a_{k-1},, a_{k-2}, a_{k-3},),\ k \geq 3 \end{cases}$$

Тоді, після виконання коду

```
a3 = u  # a3 - змінна для (k-3)-го члену послідовності
a2 = v  # a2 - змінна для (k-2)-го члену послідовності
a1 = w  # a1 - змінна для (k-1)-го члену послідовності
for k in range(3, n + 1):
    # Обчислення наступного члену
    a = f(k, p, a1, a2, a3)
    # Зміщення змінних для наступних ітерацій
    a3 = a2
    a2 = a3
    a1 = a
```

у змінних a і a1 буде міститися a_n, у змінній a2 – a_{n-1} , а в змінній a3 – a_{n-2}.

Звернемо увагу на той факт, що для обчислення членів послідовності заданої рекурентним співвідношенням першого порядку не потрібно жодних додаткових змінних – лише змінна у якій обчислюється поточний член послідовності. Для рекурентних співвідношень старших порядків, крім змінної, у якій обчислюється поточних член послідовності, необхідні ще додаткові змінні, кількість яких дорівнює порядку рекурентного співвідношення. Наведений вище алгоритм можна дещо спростити, враховуючи особливості мови програмування Python, а саме операцію пакування та розпакування кортежів

```
a3 = u  # a3 - змінна для (k-3)-го члену послідовності
a2 = v  # a2 - змінна для (k-2)-го члену послідовності
a1 = w  # a1 - змінна для (k-1)-го члену послідовності
for k in range(3, n + 1):
    # Обчислення наступного члену та зміщення змінних для наступни ітерацій
    a1, a2, a3 = f(k, p, a1, a2, a3), a1, a2
```

Приклад 1.1.5. Знайти n-й член послідовності Фібоначі

Розв'язок. Як було зазначено раніше послідовність чисел Фібоначчі F_n може бути задана рекурентним співвідношенням

$$\begin{cases} F_0 = 1,\ F_1 = 1 \\ F_k = F_{k-1} + F_{k-2},\ k \geq 2 \end{cases}$$

Оскільки послідовність Фібоначчі задана рекурентним співвідношенням другого порядку, то для того, щоб запрограмувати обчислення її членів, необхідно три змінних. Модифікувавши наведений вище алгоритм для обчислення відповідного члена послідовності заданої рекурентним співвідношенням третього порядку на випадок рекурентного співвідношення другого порядку, отримаємо програму

Лістинг 1.1.5.

```
n = int(input("n = "))
F2 = 1
F1 = 1
for k in range(2, n + 1):
    F = F1 + F2
    F2 = F1
    F1 = F
print("F = ", F)
```

Приклад 1.1.6. Скласти програму для обчислення визначника порядку n:

$$D_n = \begin{vmatrix} 5 & 3 & 0 & 0 & \dots & 0 & 0 \\ 2 & 5 & 3 & 0 & \dots & 0 & 0 \\ 0 & 2 & 5 & 3 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & \dots & 2 & 5 \end{vmatrix}.$$

Розв'язок. Легко обчислити, що

$$D_1 = 5;$$
$$D_2 = \begin{vmatrix} 5 & 3 \\ 2 & 5 \end{vmatrix} = 19.$$

Розкладаючи для всіх $k \geq 3$ визначник D_k по першому рядку отримаємо рекурентне співвідношення

$$D_k = 5D_{k-1} - 6D_{k-2}, k \geq 3.$$

Тоді, згідно з вищенаведеним алгоритмом, програма для знаходження n-го члена послідовності D_k буде мати вигляд

Лістинг 1.1.6.

```
n = int(input("n = "))
D2 = 5   # 1-й член послідовності
D1 = 19  # 2-й член послідовності
for k in range(3, n + 1):
    D1, D2 = 5 * D1 - 6 * D2, D1
print("D_%d = %d" % (n, D1))
```

§1.2. Рекурсія

Означення 1.2.1. Рекурсія – спосіб визначення об'єкту (або методу) попереднім описом одного чи кількох його базових випадків, з наступним описом на їхній основі загального правила побудови об'єкту.

З рекурсією ми часто зустрічаємося у оточуючому житті: рекурсивні зображення, структура рослин та кристалів, рекурсивні розповіді, вірші або жарти.

Означення 1.2.2. Рекурсивна функція – метод визначення функції, при якому функція прямо або неявно викликає сама себе.

Розглянемо вищенаведене означення на прикладі.

Як ми знаємо послідовність чисел Фібоначчі визначається рекурентним співвідношенням другого порядку

$$\begin{cases} F_0 = 1,\ F_1 = 1, \\ F_n = F_{n-1} + F_{n-2},\ n \geq 2. \end{cases}$$

Отже, якщо покласти, що функція $F(n)$ визначає n-те число послідовності Фібоначчі, то з вищенаведеного рекурентного співвідношення отримаємо, що послідовність Фібоначчі може бути визначена рекурсивною функцією:

$$\begin{cases} F(0) = 1,\ F(1) = 1, \\ F(n) = F(n-1) + F(n-2),\ n \geq 2. \end{cases}$$

Початкові значення $F(0) = 1, F(1) = 1$, дуже важливі при визначенні рекурсивної функції. Якби їх не було, то рекурсія б стала нескінченною! Тому, описуючи рекурсивну функцію завжди треба переконуватися, що для всіх допустимих значень аргументів виклик рекурсивної функції завершиться, тобто що рекурсія буде скінченною.

Можна звернути увагу, на те, що на відміну від обчислення елементів послідовності заданої рекурентним співвідношенням, де обчислення відбувається він тривіального (початкового) елементу до шуканого, рекурсивна функція починає обчислення від шуканого.

Кількість вкладених викликів функції називається **глибиною рекурсії**. Наприклад, глибина рекурсії при знаходженні $F(5)$ буде 4.

Опишемо стандартний алгоритм опису рекурсивної функції. Структурно рекурсивна функція на верхньому рівні завжди є розгалуженням, що складається з двох або більше альтернатив, з яких

- принаймні одна є термінальною (припинення рекурсії);
- принаймні одна є рекурсивною (тобто здійснює виклик себе з іншими аргументами).

```
def recursive_func(args):
    if terminal_case:       # Термінальна гілка
       ...
       return trivial_value # Тривіальне значення
    else:                   # Рекурсивна гілка
       ...
       # Виклик функції з іншими аргументами
       return expr(recursive_func(new_args))
```

Приклад 1.2.1. Для прикладу опишемо рекурсивну підпрограму для обчислення чисел Фібоначчі. У програмі цю функцію будемо позначати `Fib(n)`.

Лістинг 1.2.1. Рекурсивна функція для обчислення чисел послідовності Фібоначчі.

```
def Fib(n):
    if n == 0 or n == 1:                # Термінальна гілка
        return 1                        # Тривіальне значення
    else:                               # Рекурсивна гілка
        return Fib(n - 1) + Fib(n - 2)  # Рекурсивний виклик

# Виклик рекурсивної функції
n = int(input("n = "))
print("Fib(%d) = %d" % (n, Fib(n)))
```

Рекурсія використовується, коли можна виділити **самоподібність** задачі. Розглянемо інший класичний приклад, який використовується у навчальній літературі для пояснення рекурсії.

Приклад 1.2.2. Описати рекурсивну функцію для знаходження факторіала натурального числа.

Розв'язок. Очевидно, що функцію $F(n) = n!$ можна задати у такому рекурсивному вигляді

$$\begin{cases} F(0) = 1, \\ F(n) = nF(n-1), \ n \geq 1. \end{cases}$$

Тоді програма, разом з рекурсивною функцією (у програмі будемо позначати її `Fact(n)`) буде мати такий вигляд

Лістинг 1.2.2. Рекурсивна функція для обчислення факторіалу натурального числа.

```
def Fact(n):
    if n == 0 :               # Термінальна гілка
       return 1               # Тривіальне значення
    else:                     # Рекурсивна гілка
       return n * Fact(n - 1) # Рекурсивний виклик

# Виклик рекурсивної функції
n = int(input("n = "))
print("%d! = %d" % (n, Fact(n)))
```

Питання про використання рекурсії дуже суперечливе і неоднозначне. З одного боку, рекурсивна форма як правило значно простіша і наглядніша. З іншого боку, рекомендується уникати рекурсивних алгоритмів, що можуть приводити до занадто великої глибини рекурсії, особливо у випадках, коли такі алгоритми мають очевидну реалізацію у вигляді звичайного циклічного алгоритму. З огляду на це, вищенаведений рекурсивний алгоритм визначення факторіала є прикладом скоріше того, як не треба застосовувати рекурсію.

Теоретично будь-яку рекурсивну функцію можна замінити циклічним алгоритмом (можливо з застосуванням стеку).

РОЗДІЛ 2. СКЛАДНІСТЬ АЛГОРИТМІВ

§2.1. Алгоритм та оцінка його складності

У процесі вирішення різноманітних прикладних задач часто зіштовхуються з проблемою вибору алгоритму. Це часто викликає серйозні труднощі, адже правильний вибір оптимального алгоритму має вирішальне значення. Дійсно, часто правильний з аналітичної точки зору алгоритм дає незадовільний час виконання програми. Інший же алгоритм не можливо реалізувати через вимоги раціонального використання ресурсів комп'ютера (пам'яті, жорсткого диску. тощо).

Для того, щоб зрозуміти, чи підходить алгоритм для розв'язання певної задачі проводять його аналіз.

2.1.1. Алгоритм та його аналіз

Отже, що таке алгоритм і у чому полягає його аналіз?

Означення 2.1.1. **Алгоритмом** називається набір інструкцій, який описує порядок дій виконавця, щоб розв'язати задачу за скінченну кількість дій (за скінченний час).

Алгоритм може бути виражений багатьма способами. Наприклад, він може бути записаний різними мовами у термінах близьких до тих, що ми використовуємо у звичайному житті. Проте нас будуть цікавити алгоритми, які точно записані використовуючи математичний формалізм. Такі алгоритми записуються з використанням деякої мови програмування.

Означення 2.1.2. **Програмою** або **реалізацією алгоритму** називається запис алгоритму за допомогою мови програмування для його виконання комп'ютером.

Після того, як алгоритм реалізований, проводять його аналіз. У результаті цього можна отримати відповідь на питання чи підходить алгоритм для розв'язання поставленої задачі. Аналіз полягає у визначенні та аналізі таких критеріїв як:

1. час роботи програми, як функцію від вхідних даних;
2. загальну кількість пам'яті, що необхідна для даних програми;
3. загальний об'єм програмного коду;
4. чи програма розв'язує коректно (правильно) поставлену задачу;
5. чи стресо-стійка програма, тобто як добре буде поводити себе програма з некоректними вхідними даними.
6. комплексність програми, тобто чи легко програму читати, розуміти та модифікувати.

І якщо останні чотири пункти частіше за все покладаються на зовнішні аналізатори (інженери-тестувальники, автоматичні системи тестування, рецензування коду), то перші два цілковито знаходяться у зоні відповідальності програміста, який створює програму. А відтак у цьому курсі зосередимося саме на них.

Цей розділ здебільшого зосереджений на першому пункті вищенаведеного переліку, а саме на оцінці часу виконання програми.

2.1.2. Час виконання (running time)

Припустимо для розв'язання певної задачі побудований алгоритм. Потрібно оцінити на скільки оптимальним (швидким) є цей алгоритм. Виникає низка цілком закономірних питань:

- Як це зробити, тобто оцінити швидкодію цього алгоритму?
- У яких одиницях цю швидкодію вимірювати?
- Яка врешті буде складність цього алгоритму?

Звичайно більшість читачів, які вперше зіштовхнулися з цією проблематикою, скажуть, що швидкодію алгоритму можна оцінити як «час виконання програми» побудованої за цим алгоритмом і будуть правими. Проте, швидше за все, більшість з них буде вкладати в термін «час» кількість секунд, хвилин чи годин за які виконується програма написана за цим алгоритмом. А ось тут не все так однозначно. Таке означення не є коректним, оскільки воно лише частково характеризує швидкодію алгоритму. Дійсно, ні для кого не секрет, що швидкодія різних комп'ютерів різна і якщо на одному комп'ютері програма буде виконуватися, припустимо 10 секунд, то на комп'ютері, що в 10 разів швидший – 1 секунду. Крім цього, час роботи програми побудованої за алгоритмом буде залежати від мови програмування на якій її написано.

Тому цілком логічним є те, що для оцінки складності алгоритму необхідно використовувати критерій, що не залежить від потужності ЕОМ або мови програмування на якій реалізовано алгоритм.

Робота будь-якої програми, що виконується комп'ютером складається з елементарних операцій, які об'єднуються у блоки для утворення інструкцій мови програмування. До елементарних операцій, у цьому курсі віднесемо операції з набору

- звернення до об'єкту в пам'яті;
- присвоєння;
- елементарні арифметичні операції (додавання, віднімання, множення, ділення, ділення без остачі та остача від ділення) ;
- операції порівняння;
- булеві оператори;
- виклик функції/методу;
- повернення результату функцією;
- звернення за індексом до елементу списку (масиву);
- звернення за ключем до елементу словника.

Як ви можете здогадуватися, час, який витрачає комп'ютер на різні типи інструкцій різний. Наприклад, час інструкцій звернення до об'єкту в пам'яті та присвоєння значення змінній може суттєво відрізнятися. Проте, для дослідження питань цього курсу, нам буде достатнього розглядати спрощену модель комп'ютера, вважаючи, що всі елементарні операції виконуються за однаковий час τ, і без обмежень загальності, можемо вважати, що

$$\tau = 1.$$

Таким чином

Означення 2.1.3. **Часом виконання програми** (eng. running time, time complexity) будемо називати кількість елементарних операцій, які виконує комп'ютер під час виконання програми.

Очевидно, що ця кількість операцій може залежати від вхідних даних (inputs) задачі. Дійсно, наприклад очевидно, що для знаходження визначника матриці розміром 3 треба здійснити значно більше операцій, ніж для знаходження визначника розміром 2.

Час виконання програми будемо позначати

$$T(n)$$

де n – розмір вхідних даних.

Розглянемо приклади.

Приклад 2.1.1. Дано вектор заданої величини n. Оцінимо час виконання програми знаходження розміру цього вектора.

Очевидно, що розмір вектора наперед заданий (і зберігається разом з вектором), тому для його визначення достатньо однієї операції – звернення до пам'яті. Таким чином:

$$T(n) = 1$$

У такому разі кажуть, що алгоритм виконується за сталий час.

Приклад 2.1.2. Знайдемо кількість операцій алгоритму, що обчислює суму компонент заданого вектора розмірності n

$$a = (a_1, \dots, a_n)$$

Функція, що розв'язує цю задачу буде мати вигляд

Лістинг 2.1.1.

```
1   def sumV(a, n):
2       result = a[0]
3       i = 1
4       while i < n:
5           result += a[i]
6           i+=1
7       return result
```

Визначимо час виконання програми. Для цього створимо таблицю рядки якої будуть відповідати рядкам програми

Рядок	Час
2	3
3	2
4	$3 \times n$
5	$5 \times (n-1)$
6	$4 \times (n-1)$
7	2

Зауваження, тут і надалі ми будемо вважати, що операції `+=`, `-=` і подібні виконуються за 4 операції, оскільки їх можна інтерпретувати як

```
i = i + 1
```

При цьому число 1 є літералом, що зберігається у пам'яті. Від так доступ до нього вимагає одну елементарну операцію.

Таким чином час виконання програми буде

$$T(n) = 2 + 3 + 3n + 5(n-1) + 4(n-1) + 2 = 12n - 2$$

Як бачимо «складність» задачі безпосередньо пов'язана із розміром вектора (вхідні дані). Причому цей час виконання є лінійною функцією від розміру вхідних даних. У такому разі кажуть, що алгоритм має лінійну складність, або програма виконується за лінійний час.

Приклад 2.1.3. Знайдемо кількість операцій алгоритму, що обчислює суму компонент квадратної матриці розмірності n.

Для розв'язання цієї задачі можна запропонувати такий алгоритм

Лістинг 2.1.2.

```
1   def sumM(A, n):
2       result = 0
3       i = 0
4       j = 0
5       while i < n:
6           while j < n:
7               result = result + A[i][j]
8               j = j + 1
9           j = 0
10          i += 1
11      return result
```

Аналогічно попередньому прикладу побудуємо таблицю кількості операцій кожного рядка

Рядок	Час
2	2
3	2
4	2
5	$3 \times (n+1)$
6	$(3 \times (n+1)) \times n$
7	$((5+3) \times n) \times n$
8	$(4 \times n) \times n$
9	$2 \times n$
10	$4 \times n$
11	2

Отже, підсумовуючи значення другого стовпчика, отримаємо, що

$$T(n) = 12n^2 + \ldots$$

Час роботи цього алгоритму називають **поліноміальним**, оскільки він оцінюється як поліном від розміру вхідних даних.

Приклад 2.1.4. Оцінимо час роботи алгоритму рекурсивного знаходження F_n - n-го члена послідовності Фібоначчі.

Така підпрограма буде мати вигляд

Лістинг 2.1.3.

```
1   def fib(n):
2       if n <= 1:
3           return 1
4       else:
5           return fib(n - 1) + fib(n - 2)
```

Знову побудуємо таблицю

Рядок	Час	
	$n \leq 1$	$n > 1$
2	3	3
3	2	–
4	–	–
5	–	$1 + 4 \times 2 + 1 + T(n-1) + T(n-2)$

Звідки маємо, що

$$T(0) = 5$$
$$T(1) = 5$$

а, для $n \geq 2$ отримаємо, що складність $T(n)$ описаного алгоритму визначається як рекурентне співвідношення

$$T(n) = T(n-1) + T(n-2) + 10.$$

Звідки маємо, що

$$T(n) \geq F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}} \approx 2^{0.7n}$$

Такий час роботи називають **експоненціальним**. І кажуть, що алгоритм працює за **експоненціальний час**.

Приклад 2.1.5. Оцінимо час роботи алгоритму додавання у стовпчик двох натуральних чисел n та m.

Час роботи такого алгоритму буде пропорційним кількості цифр у «довшому числі». Кількість цифр у числі n не перевищує величину $\log_{10} n$, а у числі m – відповідно величину $\log_{10} m$. Отже час роботи такого алгоритму буде пропорційним такому

$$T(n, m) = \log_{10} \max\{n, m\}$$

Такий час роботи називають **логарифмічним** і, відповідно, кажуть, що алгоритм працює за **логарифмічний час**.

Приклад 2.1.6. Визначимо час виконання програми, для знаходження часткової суми ряду

$$\sum_{i=0}^{\infty} x^i$$

Розглянемо послідовність часткових сум цього ряду

$$S_n = \sum_{i=0}^{n} x^i, n \geq 0.$$

Напишемо програму найпростішим способом, тобто таку, що використовує рекурентне співвідношення

$$S_0 = 0$$
$$S_n = S_{n-1} + x^n, n \geq 1$$

Лістинг 2.1.4.

```
1   def S(x, n):
2       sum = 0
3       i = 0
4       while i <= n:
5           prod = 1
6           j = 0
7           while j < i:
8               prod *= x
9               j += 1
10          sum += prod
11          i += 1
12      return sum
```

та підрахуємо кількість операцій у кожному рядку програми:

Рядок	Час
2	2
3	2
4	$3(n+2)$
5	$2(n+1)$
6	$2(n+1)$
7	$4\sum_{i=0}^{n}(i+1)$
8	$4\sum_{i=0}^{n} i$
9	$4\sum_{i=0}^{n} i$
10	$4(n+1)$
11	$4(n+1)$
12	2

Як відомо

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

Тоді підсумовуючи, отримаємо час виконання програми

$$T(n) = \frac{11}{2}n^2 + \frac{47}{2}n + 24$$

Тепер напишемо програму, що вирішує поставлену вище задачу, проте скористаємося алгоритмом, що базується на рекурентному співвідношенні, що отримується зі схеми Горнера

$$S_0 = 0$$
$$S_n = S_{n-1} * x + 1, n \geq 1$$

Тоді програма буде мати вигляд

Лістинг 2.1.5.

```
1   def S2(x, n):
2       sum = 0
3       i = 0
4       while i <= n:
5           sum = sum * x + 1
6           i += 1
7       return sum
```

і відповідна їй таблиця кількості операцій

Рядок	Час
2	2
3	2
4	$3(n+2)$
5	$6(n+1)$
6	$4(n+1)$
7	2
$\boldsymbol{T(n)}$	$13n+22$

Як бачимо, однаково поставлені задачі можуть бути розв'язані по різному і відповідні їхні алгоритми можуть використовувати різну кількість операцій.

Приклад 2.1.7. Необхідно написати програму піднесення дійсного числа до цілого степеня та оцінити її складність.

Стандартний алгоритм побудований на елементарному рекурентному співвідношенні має вигляд

Лістинг 2.1.6.

```
1   def pow(x, n):
2       result = 1
3       i = 0
4       while i <= n:
5           result = result * x
6           i += 1
7       return result
```

і, очевидно, має лінійний порядок складності n. Проте виникає цілком логічне запитання, чи не можна прискорити цей алгоритм? Не складно помітити, що функцію піднесення до степеня можна визначити рекурсивним чином, а саме

$$\mathrm{pow}(x,n)=\begin{cases}1, & n=0\\ \mathrm{pow}^2\left(x,\frac{n}{2}\right), & n-\text{парне}\\ x\cdot\mathrm{pow}^2\left(x,\frac{n-1}{2}\right), & n-\text{непарне}\end{cases}$$

або, що те ж саме

$$\mathrm{pow}(x,n)=\begin{cases}1, & n=0\\ \mathrm{pow}^2\left(x,\frac{n}{2}\right), & n-\text{парне}\\ x\cdot\mathrm{pow}\left(x^2,\frac{n-1}{2}\right), & n-\text{непарне}\end{cases}$$

Лістинг 2.1.7.

```
1   def pow(x, n):
2       if n == 0:
3           return 1
4       elif n % 2 == 0:
5           return pow(x * x, n / 2)
6       else:
7           return x * pow(x * x, (n - 1) / 2)
```

Складемо таблицю для підрахунку часу виконання програми

Рядок	Час		
	$n=0$	$n>0$ – парне	$n>0$ – непарне
2	3	3	3
3	2	–	–
4	–	5	5
5	–	$10+T(n/2)$	–

7	–	–	$14 + T((n-1)/2)$
$\boldsymbol{T(n)}$	5	$18 + T(n/2)$	$22 + T((n-1)/2)$

$$T(n) = \begin{cases} 5, & n = 0 \\ 18 + T(n/2), & n - \text{парне} \\ 22 + T((n-1)/2), & n - \text{непарне} \end{cases}$$

Спробуємо знайти явний вигляд отриманого рекурентного співвідношення. Припустимо, що $n = 2^k$, для деякого $k > 0$. Тоді для парного n:

$$n/2 = 2^{k-1}$$

Таким чином,

$$T(n) = T(2^k) = 18 + T(2^{k-1}) = \\ = 18 + 18 + T(2^{k-2}) = \cdots = 18j + T(2^{k-j}).$$

Підстановка закінчиться тоді, коли $k = j$. Отже

$$T(2^k) = 18k + T(1) = 18k + 20 + T(0) = 18k + 25.$$

Оскільки $n = 2^k$, то $k = \log_2 n$. І тоді час виконання

$$T(n) = 18 \log_2 n + 25$$

Аналогічно, для непарного n:

$$n + 1 = 2^k \\ T(n) = T(2^k - 1) = 22 + T(2^{k-1} - 1) = \\ = 22 + 22 + T(2^{k-2} - 1) = \cdots = 22j + T(2^{k-j} - 1).$$

Підстановка завершується при $k = j$

$$T(2^k - 1) = 20k + 5$$

або

$$T(n) = 22 \log_2(n+1) + 5$$

Таким чином бачимо, що складність цього алгоритму має логарифмічний порядок.

2.1.3. Найкращий, найгірший та середній час виконання

У попередньому пункті у всіх прикладах знайдений час виконання програм залежав лише від розміру вхідних даних. Проте, для багатьох програм час виконання програми часто залежить не стільки від розміру вхідних даних, скільки від самих цих даних. Щоб у цьому переконатися, розглянемо такий приклад.

Приклад 2.1.8. Визначити чи заданий елемент s міститься у списку a, що містить n елементів.

Оскільки ми нічого не знаємо заздалегідь про елементи списку, то реалізуємо алгоритм послідовного пошуку:

Лістинг 2.1.8.

```
1   def find(a, n, s):
2       i = 0
3       while i < n:
4           if a[i] == s:
5               return True
6           i += 1
7       return False
```

Функція послідовно перебирає усі елементи списку і якщо зустрічає шуканий елемент, то її виконання переривається. Відповідно, і її час виконання залежить не лише від розміру вхідних даних, але і від того, які дані подаються на вхід у функцію. Так, наприклад, якщо список першим елементом містить шуканий елемент, то цикл виконає своє тіло рівно один раз. Якщо ж список взагалі не містить шуканого елементу або він є останнім елементом у списку, то тіло циклу виконається всі n разів. У першому випадку говорять про час виконання **у найкращому випадку**, у другому – **у найгіршому випадку**.

Означення 2.1.4. **Часом виконання програми у найгіршому випадку** (eng. worst-case running time, worst-case time complexity) будемо називати найбільшу кількість елементарних операцій, які виконує комп'ютер під час виконання програми для довільних вхідних даних розміру n.

Означення 2.1.5. **Часом виконання програми у найкращому випадку** (eng. best-case running time, best-case time complexity) будемо називати найменшу кількість елементарних операцій, які виконує комп'ютер під час виконання програми яка досягається для деякого набору вхідних даних розміру n.

Зобразимо таблицю для найкращого та найгіршого випадку нашого прикладу

Рядок	Час	
	у найкращому випадку	у найгіршому випадку
2	2	2
3	3	$3(n+1)$
4	5	$5n$
5	2	–
6	–	$4n$
7	–	2
$\boldsymbol{T(n)}$	12	$12n+7$

Очевидно, що згадані вище випадки є крайніми та не завжди можуть об'єктивно відображати інформацію про швидкодію алгоритму на практиці для конкретних наборів вхідних даних. Тому, крім згаданих випадків, часто розглядають третій – **час виконання в середньому**.

Означення 2.1.6. **Часом виконання програми в середньому** (eng. average-case running time, average-case time complexity) будемо називати усереднену кількість елементарних операцій, які виконує комп'ютер під час виконання програми для всіх наборів вхідних даних розміру n.

Очевидно, що час виконання в середньому час залежить від імовірнісного розподілу вхідних даних і, очевидно, може бути визначеним, якщо ці ймовірнісні характеристики відомі. На практиці час виконання у середньому знайти значно складніше, аніж час виконання у найгіршому випадку, враховуючи математичну складність такої задачі. Тому в основному будемо використовувати час виконання у найгіршому випадку, як характеристику часової складності алгоритму.

Повернемося до розгляду прикладу. Фактично задача визначення часу виконання у середньому у цьому випадку рівнозначна визначенню середньої кількості ітерацій циклу. Припустимо для спрощення, що список складається з різних натуральних чисел з діапазону $[0, n-1]$, а шукане число належить діапазону $[0, n]$. Зроблене припущення означає, що шукане число входить у список не більше ніж один раз, причому ймовірність входження шуканого числа на кожній ітерації циклу є однаковою. Тоді очевидно, що в середньому, кількість ітерацій циклу буде $\frac{(n+1)}{2}$. Доповнимо вищенаведену таблицю колонкою з середнім часом виконання

Рядок	Час виконання		
	у найкращому випадку	у найгіршому випадку	у середньому
2	2	2	2
3	3	$3(n+1)$	$\frac{3(n+1)}{2}$
4	5	$5n$	$\frac{5(n+1)}{2}$
5	2	–	–
6	–	$4n$	$\frac{4(n+1)}{2}$
7	–	2	2
$\boldsymbol{T(n)}$	12	$12n+7$	$6n+10$

Зауважимо, що для іншого набору вхідних даних середній час виконання може суттєво відрізнятися від наведеного вище.

У подальшому, якщо конкретно не зазначено який тип часу виконання розглядається, то будемо вважати, що мається на увазі час виконання у найгіршому випадку.

§2.2. Асимптотична оцінка складності алгоритмів

Спробуємо зрозуміти, чому так важливо оцінювати часову складність алгоритмів. Припустимо, що ми розглядаємо два алгоритми: A і B, для вирішення заданої задачі. Крім того, скажімо, ми зробили ретельний аналіз часів роботи кожного з алгоритмів і визначили їх як $T_A(n)$ та $T_B(n)$ відповідно, де n – розмір вхідних даних. Тоді досить просто порівняти дві функції $T_A(n)$ та $T_B(n)$, щоб визначити, який алгоритм кращий! Очевидно, що якщо для деякого відомого входу n_0 має місце нерівність $T_A(n_0) \leq T_B(n_0)$, то алгоритм A є кращим за B для цього набору вхідних даних. Проте, якщо ми не знаємо наперед нічого про вхідні дані, то сказати який алгоритм буде кращим не завжди просто.

Давайте розглянемо такий приклад. Припустимо, що величина входу деякої задачі є n. Припустимо, що три студенти по різному розв'язали цю задачу.

Нехай алгоритм першого використовує

$$T_1(n) = 345n^3 + 123n^2 + 98n + 15$$

операцій, алгоритм другого

$$T_2(n) = 12n^4 + 1$$

операцій, а алгоритм третього

$$T_3(n) = 3 \cdot 2^n$$

операцій. Який алгоритм кращий? У випадку, якщо розмір входу буде зовсім невеликий, наприклад $n = 4$, то останній алгоритм буде виконуватися за найменшу кількість операцій. Проте вже при $n = 20$ кращим буде другий алгоритм, а от, якщо n буде великим, наприклад 10000, то другий алгоритм буде використовувати значно більше операцій ніж перший, а останній взагалі можна вважати нескінченним.

Тут слід зауважити, що, як правило нікого не цікавить швидкодія алгоритму при невеликих значенням вхідних даних – для вхідних даних малого розміру час виконання будь-якого алгоритму є задовільним з практичної точки зору. Тому при оцінці швидкодії алгоритмів цікавить випадок саме великого розміру вхідних даних. А отже, взагалі кажучи, якщо нічого наперед не відомо про розмір вхідних даних, то алгоритм першого студента можна вважати найкращим.

Під час оцінки швидкодії алгоритму при великих значеннях розміру вхідних даних, як правило цікавить не точна кількість операцій, яку здійснює програма, а порядок цієї кількості операцій. Дійсно, ви можете не знати точно як реалізують на комп'ютері арифметичні операції, умовні оператори тощо. Наприклад, для додавання двох великих цілих чисел може використовуватися не одна операція процесора, а три (якщо розрядність числа більша, ніж розрядність операційної системи). Тому при оцінці швидкодії програми:

1) У формулі кількості операцій, враховують лише доданок, що зростає найшвидше.
2) Сталий множник при цьому доданку встановлюють рівними 1.

Отриману величину називають **порядком складності алгоритму**.

Таким чином алгоритм першого студента має складність порядку n^3, другого n^4, а третього 2^n.

Отже, відповідь на питання, поставлене на початку цього пункту полягає у намаганні передбачити як зростає час виконання програми з ростом розміру вхідних даних. Наприклад, якщо розмір вхідних даних задачі збільшився у 10 рази. У скільки разів тоді повільніше буде працювати програма?

Для оцінки складності програм, залежно від вхідних даних, використовують O, Ω та Θ символіку яка є формалізованим обґрунтуванням порядку складності алгоритму. Основне їхнє призначення це «грубо» оцінити час виконання алгоритму, а також наскільки швидко зростає час роботи алгоритму зі збільшенням розміру вхідних даних. Фактично тут піде мова про **асимптотичну поведінку** функцій часу виконання, що залежать від вхідних даних алгоритму.

2.2.1. O – символіка

Означення та приклади

Нехай є дві функції $f, g: \mathbb{N} \to \mathbb{R}_+$, (які будемо інтерпретувати як час роботи двох різних алгоритмів на різних довжинах вхідних даних).

Означення 2.2.1. Кажуть, що $f = O(g)$, якщо $\exists C > 0$ та $\exists n_0 \geq 0$, що $\forall n \geq n_0$: $f(n) \leq Cg(n)$.

У такому разі також кажуть, що «f зростає не швидше ніж g».

Приклад 2.2.1. Розглянемо функцію $f(n) = 8n + 128$. Очевидно, що $f(n) \geq 0$ для всіх натуральних n. Покажемо, що $f = O(n^2)$. Згідно з означенням, нам треба знайти таке натуральне n_0 і таку сталу $C > 0$, що для всіх $n \geq n_0$

$$8n + 128 \leq Cn^2$$

Не має значення величина сталої – головне чи вона існує. Припустимо, що $C = 1$

$$8n + 128 \leq n^2 \Rightarrow n^2 - 8n - 128 \geq 0$$
$$\Rightarrow (n - 16)(n + 8) \geq 0$$

Очевидно, що остання нерівність виконується при $n \geq 16$. Таким чином, $\exists C = 1$ та $\exists n_0 = 16$, що для всіх $n \geq n_0$

$$8n + 128 \leq n^2$$

Отже,

$$f(n) = O(n^2)$$

Очевидно, що існує безліч значень таких пар C та n_0.

Приклад 2.2.2.

$$345n^3 + 123n^2 + 98n + 15 = O(n^3)$$

Зауваження. У цьому курсі будемо позначати $\log_2 n =: \log n$.

Приклад 2.2.3.

$$n \log n = O(n^2)$$

Зауваження. З того, що $f_1 = O(g)$ і $f_2 = O(g)$ **не випливає**, що

$$f_1 = f_2$$

Дійсно, розглянемо функції $f_1 = n^2$ і $f_2 = n$. легко показати, що $f_1 = O(n^2)$ та $f_2 = O(n^2)$

Властивості

Визначення асимптотичної поведінки функції часто буває дуже клопіткою роботою, якщо користуватися лише означенням. Тому, як правило при оцінці асимптотики функції використовують властивості, деякі з яких наведено нижче.

Будемо вважати, що всі функції, що використовуються нижче є додатнозначними функціями натурального аргументу.

Теорема 2.2.1. Нехай $f_1 = O(g_1)$ і $f_2 = O(g_2)$, тоді

$$f_1 + f_2 = O(\max(g_1, g_2)).$$

Теорема 2.2.2. Нехай $f = f_1 + f_2$, причому

$$\lim_{n\to\infty} \frac{f_2(n)}{f_1(n)} = L,$$

де L деяка стала. Тоді

$$f = O(f_1).$$

Теорема 2.2.3. Нехай $f_1 = O(g_1)$ і $f_2 = O(g_2)$, тоді

$$f_1 \times f_2 = O(g_1 \times g_2).$$

Теорема 2.2.4. Нехай $f_1 = O(g_1)$, тоді для будь-якої функції g_2

$$f_1 \times g_2 = O(g_1 \times g_2).$$

Теорема 2.2.5. (Транзитивність) Нехай $f = O(g)$ і $g = O(h)$. Тоді

$$f = O(h).$$

З доведенням вищенаведених теорем ви можете ознайомитися у підручнику [5, Chapter 3].

Правила визначення асимптотичної поведінки функції

Властивості, наведені вище дозволяють сформулювати прості правила для визначення асимптотичної поведінки функції.

1) Мультиплікативні константи можна опускати, тобто для будь-якої додатної сталої C

$$Cn^3 = O(n^3);$$

2) Нехай $b \geq a$ тоді

$$n^a = O(n^b);$$

3) Нехай $a > 1$. Тоді для будь-якого $k \geq 1$,

$$n^k = O(a^n);$$

4) Для будь-якого $k \geq 1$

$$\log^k n = O(n);$$

5) Якщо функція є сумою кількох функцій, то її асимптотична поведінка визначається доданком, що зростає найшвидше. Наприклад

$$345n^3 + 123n^2 + 98n + 15 = O(n^3).$$

Поширені асимптотичні складності

У таблиці нижче наведені найпоширеніші асимптотичні складності алгоритмів.

Складність	Коментар	Приклади
$O(1)$	Сталий час роботи не залежно від розміру задачі	Пошук у хеш-таблиці
$O(\log\log n)$	Дуже повільне зростання необхідного часу	Очікуваний час роботи інтерполюючого пошуку n елементів
$O(\log n)$	Логарифмічне зростання — подвоєння розміру задачі збільшує час роботи на сталу величину	Швидке обчислення x^n; двійковий пошук у відсортованому масиві з n елементів
$O(n)$	Лінійне зростання — подвоєння розміру задачі подвоїть і необхідний час	Додавання/віднімання чисел з n цифр; лінійний пошук в масиві з n елементів
$O(n\log n)$	Лінеаритмічне зростання — подвоєння розміру задачі збільшить необхідний час трохи більше ніж вдвічі	Сортування злиттям або купою масиву з n елементів.
$O(n^2)$	Квадратичне зростання — подвоєння розміру задачі вчетверо збільшує необхідний час	Елементарні алгоритми сортування масивів з n елементів; Лінійний пошук у квадратній матриці розмірності n.
$O(n^3)$	Кубічне зростання — подвоєння розміру задачі збільшує необхідний час у вісім разів	Звичайне множення матриць
$O(a^n)$	Експоненціальне зростання — збільшення розміру задачі на 1 призводить до a-кратного збільшення необхідного часу; подвоєння	Деякі задачі комівояжера; Алгоритми пошуку повним перебором

	розміру задачі підносить необхідний час у квадрат

2.2.2. Ω – символіка

Крім вищенаведеної «великого О» застосовуються інші типи асимптотик, про які буде розказано далі.

Означення та приклади

Означення 2.2.2. Кажуть, що $f = \Omega(g)$, якщо $g = O(f)$.

У такому разі також кажуть, що «f зростає не повільніше ніж g».

Приклад 2.2.4. Покажемо, що

$$5n^2 - 64n + 256 = \Omega(n^2).$$

Відповідно до означення, нам треба знайти таке натуральне n_0 і таку сталу $C > 0$, що для всіх $n \geq n_0$

$$5n^2 - 64n + 256 \geq Cn^2.$$

Виберемо $C = 1$. Тоді

$$\begin{gathered} 5n^2 - 64n + 256 \geq n^2 \\ \Rightarrow 4n^2 - 64n + 256 \geq 0 \\ \Rightarrow 4(n-8)^2 \geq 0 \end{gathered}$$

Оскільки $(n-8)^2 \geq 0$ для всіх $n \geq 0$, отримуємо, що $n_0 = 1$, що і доводить твердження прикладу.

Приклад 2.2.5. Використовуючи правила, що наведені у попередньому пункті, можна легко бачити, що

$$n^2 = \Omega(n \log^3 n).$$

Ω-оцінка використовується коли потрібно оцінити нижню межу швидкодії алгоритму. Такі оцінки часто використовуються для того, щоб переконатися, що отриманий алгоритм є оптимальним.

Наприклад, нехай у нас є масив з n елементів (які можна порівнювати). Відомим є твердження, що для того, щоб відсортувати такий масив необхідно $\Omega(n \log n)$. Отже, це означає, що не можливо відсортувати такий масив швидше, наприклад за лінійний час.

2.2.3. Θ – символіка

Означення 2.2.3. Кажуть, що $f = \Theta(g)$, якщо $f = O(g)$ та $g = O(f)$.

У такому разі також кажуть, що «f та g мають однаковий порядок росту».

Приклад 2.2.6.

$$\begin{gathered} 345n^3 + 123n^2 + 98n + 15 = \Theta(n^3) \\ \log n = \Theta(\log_{10} n^3) \end{gathered}$$

Зауваження. Досить часто, під час асимптотичної оцінки складності алгоритмів, записують O-оцінку, маючи на увазі Θ-оцінку. Тому, у подальшому, без додаткових обговорень, будемо користуватися O-оцінкою навіть для випадків, якщо має місце Θ-оцінка.

2.2.4. Асимптотичний аналіз алгоритмів

У попередньому параграфі ми навчилися обчислювати час роботи алгоритмів. Ця задача була досить клопіткою навіть у випадку використання спрощеної моделі комп'ютера у якій ми вважали, що кожна елементарна операція виконується за однакову одиницю часу. Проте, для оцінки складності алгоритму у більшості випадків досить оцінки її асимптотичної складності. А відтак потреба у таких детальних обчисленнях кількості операцій відпадає.

Оцінити асимптотичну складність алгоритмів можна використовуючи методи обчислення часу виконання програми та використовуючи правила наведені у цій темі. Здебільшого визначення асимптотичної складності алгоритму переважно зводиться до аналізу циклів і рекурсивних викликів підпрограм.

Як правило, найбільший інтерес щодо дослідження алгоритму з практичної точки зору складає оцінка часу його роботи у найгіршому випадку. Це аргументовано ти що:

1) це дає оцінку швидкодії алгоритму для довільних вхідних даних визначеного розміру;
2) з практичної точки зору, «погані» вхідні дані трапляються досить часто;

3) час роботи в середньому є досить близьким до часу роботи у найгіршому випадку.

Розглянемо ще раз приклад 2.1.6 та оцінимо асимптотичну складність кожного з алгоритмів для запропонованих там. Розширимо таблиці визначання часу виконання колонкою асимптотичної оцінки для кожного з алгоритмів.

Отже, для першого алгоритму таблиця буде мати вигляд

Рядок	Час	Асимптотична оцінка «велике O»
2	2	$O(1)$
3	2	$O(1)$
4	$3(n+2)$	$O(n)$
5	$2(n+1)$	$O(n)$
6	$2(n+1)$	$O(n)$
7	$4\sum_{i=0}^{n}(i+1) = 4\frac{(n+2)(n+1)}{2}$	$O(n^2)$
8	$4\sum_{i=0}^{n} i = 4\frac{n(n+1)}{2}$	$O(n^2)$
9	$4\sum_{i=0}^{n} i = 4\frac{n(n+1)}{2}$	$O(n^2)$
10	$4(n+1)$	$O(n)$
11	$4(n+1)$	$O(n)$
12	2	$O(1)$

Отже, використовуючи правила наведені вище, можемо переконатися, що загальна асимптотика вищенаведеного алгоритму є $O(n^2)$.

Таблиця складності для програми написаної другим способом

Рядок	Час	Асимптотична оцінка
2	2	$O(1)$
3	2	$O(1)$
4	$3(n+2)$	$O(n)$
5	$6(n+1)$	$O(n)$
6	$4(n+1)$	$O(n)$
7	2	$O(1)$
$\boldsymbol{T(n)}$	$13n+22$	$O(n)$

Звідки отримаємо, що її складність має асимптотику $O(n)$.

Як бачимо, для асимптотичної оцінки алгоритму нам фактично не потрібно обчислювати точно час виконання цього алгоритму. Наприклад, якщо певна інструкція знаходиться поза межами циклу, тобто кількість операцій які в ній виконуються не залежить від вхідних даних, то операція виконується за сталий час і асимптотика є $O(1)$. Якщо ж певна операція виконується в циклі, то асимптотика буде напряму залежати від того, скільки разів виконується в циклі операція. Використовуючи такі міркування та скориставшись правилами «великого O», наведеними вище, можна отримати правила для визначення асимптотичної складності алгоритмів.

Правила оцінки алгоритмів

Теорема 2.2.6 (Послідовна композиція). Найгірший час виконання алгоритму, що складається з послідовності виразів

S1
S2
...
Sm

має асимптотичну складність

$$O(\max\{T_1(n), \dots, T_m(n)\}),$$

де $T_1(n), \dots, T_m(n)$ – час виконання відповідно інструкцій `S1`, ..., `Sm`.

Теорема 2.2.7. (Цикл while). Найгірший час виконання алгоритму, що містить цикл while

```
while S1:
    S2
```

має асимптотичну складність

$$O(\max\{T_1(n) \times (I(n)+1), T_2(n) \times I(n)\}),$$

де $T_1(n), T_2(n)$ – час виконання відповідно інструкцій `S1`, `S2`, а $I(n)$ – кількість ітерацій циклу, що виконується у найгіршому випадку.

Теорема 2.2.8. (Цикл for). Найгірший час виконання алгоритму, що містить цикл **`for`**.

```
for i in range(S1):
    S2
```

має асимптотичну складність

$$O(\max\{T_1(n) \times (I(n)+1), T_2(n) \times I(n)\}),$$

де $T_1(n), T_2(n)$ – час виконання відповідно інструкцій `S1`, `S2`, а $I(n)$ – кількість ітерацій циклу, що виконується у найгіршому випадку.

Це правило випливає з того, що цикл по колекції еквівалентний такому циклу **`while`**

```
i = 0
while S1:
    S2
    i += 1
```

Теорема 2.2.9. (Умовний оператор). Найгірший час виконання алгоритму

```
if S1:
    S2
else:
    S3
```

має асимптотичну складність

$$O(\max\{T_1(n), T_2(n), T_3(n)\}),$$

де $T_1(n), T_2(n), T_3(n)$ – час виконання відповідно інструкцій `S1`, `S2`, `S3`.

Наведемо кілька прикладів застосування вищенаведених правил.

Приклад 2.2.7. Знайти асимптотичну складність алгоритму знаходження максимального числа у дійному векторі.

Приклад 2.2.8. Знайти асимптотичну складність алгоритму знаходження максимального числа у матриці.

Приклад 2.2.9. Знайти асимптотичну складність алгоритму обчислення степеня натурального числа, використовуючи ітеративний та рекурсивний алгоритми.

Реальна ситуація

Асимптотичний аналіз алгоритмів на практиці дозволяє оцінити ефективність вашої програми та дати відповідь на питання чи є сенс застосовувати реалізований алгоритм до ваших вхідних даних. Припустимо, що ви реалізували алгоритм двома способами, перший з яких має складність

$$T_1(n) = O(n^2),$$

а другий

$$T_2(n) = O(n^3).$$

На перший погляд може скластися враження, що це не принциповий момент якій з двох реалізацій віддати перевагу – обидві мають поліноміальний час виконання, обидві однаково швидко працюють для ваших тестових даних (звичайно що тестових даних не дуже великого розміру). Проте реальні показники часу виконання програми для різних розмірів вхідних даних можуть переконати в протилежному – потрібно завжди обирати програму порядок складності якої менший. Нижче наведено таблицю, у якій зазначено реальний час виконання програм, що мають різну асимптотичну складність для вхідних даних різного розміру. Таблиця наведена з

розрахунку, що одна елементарна операція виконується за 1нс (одну нано-секунду), і для кожного елементу вхідних даних виконується лише одна елементарна операція.

Складність	$n = 1$	$n = 8$	$n = 1K$	$n = 1024K$
$O(1)$	1 нс	1 нс	1 нс	1 нс
$O(\log n)$	1 нс	3 нс	10 нс	20 нс
$O(n)$	1 нс	8 нс	102 нс	1.05 мс
$O(n \log n)$	1 нс	24 нс	1.02 мс	21 мс
$O(n^2)$	1 нс	64 нс	10.2 мс	18.3 хв
$O(n^3)$	1 нс	512 нс	1.07 с	36.5 р.
$O(2^n)$	1 нс	256 нс	10^{292} р.	10^{10^5} р.

Як бачимо з таблиці, програма, що має складність $O(n^3)$ для вхідних даних розміру 2^{20}, буде виконуватися понад 36 років, у той час як програма зі складністю $O(n^2)$ лише 18 хвилин. Звичайно тут читач може заперечити, апелюючи до того, що можливо його програма не буде оперувати даними таких розмірів і тоді вибір алгоритму не є принциповим. Це дійсно так. Як видно з таблиці, для невеликих розмірів вхідних даних всі вони виконуються за задовільний час. Тут черговий раз важливо наголосити, що коли мова йде про аналіз алгоритму, апріорі передбачається, що алгоритм буде оперувати даними великих розмірів. Більше того, якщо вхідні дані не великого розміру, то розробка складного оптимального алгоритму може себе не виправдати з огляду на витрачений час для його реалізації, відлагодження отриманої програми та подальшу її підтримку. У такому випадку краще обрати алгоритм, що є найпростішим для реалізації.

РОЗДІЛ 3. ПОШУК ТА СОРТУВАННЯ

§3.1. Пошук

3.1.1. Лінійний (послідовний) пошук

Розглянемо колекцію елементів, що зберігаються у простому списку (масиві). У такому разі ми можемо говоримо, що ці елементи є послідовностями, оскільки кожен елемент зберігається на певній позиції, яка однозначно визначається індексом цього елемента (тобто його номером у послідовності). Оскільки значення індексів впорядковані, то ми маємо можливість послідовно проходити по ним. Цей процес породжує найпростішу, з алгоритмічної точки зору, пошукову техніку – **послідовний** (або лінійний) **пошук**.

Означення 3.1.1. **Лінійним** або **послідовним пошуком** називається алгоритм відшукання елемента серед заданого набору шляхом послідовного перебору всіх елементів.

Малюнок нижче демонструє як працює такий пошук. Починаючи з першого елемента в списку, ми послідовно рухаємося по елементах послідовності (у порядку зростання індексів), до тих пір, поки або не знайдемо те, що шукаємо, або не досягнемо останнього елемента. Останнє означає, що послідовність не містить шуканого елемента.

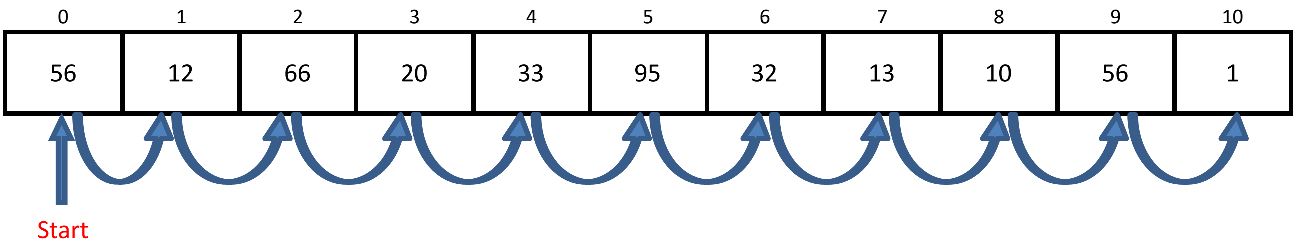


Рисунок 3.1.1. Лінійний пошук по масиву.

Найпростіша реалізація алгоритму лінійного пошуку, результатом якого буде відповідь чи містить колекція array шуканий елемент x буде виглядати таким чином

Лістинг 3.1.1.

```
def linear_search(array, x):
    """ Лінійний пошук у масиві

    :param array: Список елементів
    :param x: Шуканий елемент
    :return: True, якщо шуканий елемент знайдено
    """

    for el in array:
        if el == x:
            return True
    return False
```

Аналіз

Очевидно, що лінійний пошук у найгіршому випадку виконується за лінійний час, тобто $O(n)$. Слід зауважити, що у випадку, якщо шуканий елемент знаходиться на першій позиції, то лінійний пошук здійснюється за сталий час $O(1)$.

Лінійний пошук працює з даними різних типів та не ставить жодних вимог до способу зберігання та розташування даних у колекції – він не вимагає якихось початкових дій по обробці колекції перш ніж розпочати пошук. Проте, він має один суттєвий недолік, який повністю перекреслює його переваги – він занадто повільний, щоб його можна було застосовувати для великих обсягів даних. Дійсно, навряд чи хтось буде користуватися словником у якому слово потрібно шукати послідовно перебираючи всі слова – значно швидше можна знайти потрібне слово користуючись словником у якому слова містяться у лексикографічному порядку. Аналогічно,

зайшовши до супермаркету, покупець не буде послідовно перебирати всі товари – він спочатку Знайде потрібний відділ, скориставшись картою або спитавши у консультанта, а вже потім, використовуючи лінійний пошук серед товарів відділу знайде необхідне. У наступних пунктах цього параграфу розглянемо алгоритми, що дозволяють значно пришвидшити процес пошуку даних.

3.1.2. Бінарний пошук

З використанням бінарного пошуку майже напевно зіштовхувалися всі, що хоч раз користувався словником. Оскільки слова у словнику розташовані у лексикографічному порядку, то щоб знайти потрібне слово, словник відкривається посередині і аналізуючи слова, які містяться на відкритій сторінці, приймається рішення у якій з половин словника міститься шукане слово. Далі процедура повторюється спочатку, але для тієї половини словника, що містить слово. Цей процес повторюється доти, доки не буде знайдено потрібне слово або не буде встановлено, що словник його не містить.

Назва такого методу походить від того, що на кожній ітерації, дані, серед яких проводиться пошук, діляться навпіл. Бінарний пошук буває двох видів: цілочисельний та дійсний. Цілочисельний пошук застосовується для індексованих масивів даних. Дійсний пошук використовується для пошуку аргументу деякої неперервної функції при якому досягається задане значення.

Цілочисельний бінарний пошук

Припустимо, що ми розглядаємо впорядковану колекцію (список) у якій будь-які два елементи можна порівняти, тобто сказати, який з них більший. Якщо ж елементи у цьому списку, розташовані у порядку зростання або спадання то будемо казати, що список впорядкований (за зростанням або спаданням відповідно).

Бінарний пошук базується та тому, що для будь-якого шуканого числа x і для будь-якого індексу масиву i, ми можемо визначити чи для елементу впорядкованого списку a_i має місце співвідношення

$$a_i \leq x.$$

Для прикладу, розглянемо впорядкований за зростанням список цілих чисел зображений нижче.

0	1	2	3	4	5	6	7	8	9	10
1	10	12	13	20	32	33	56	66	71	95

Припустимо потрібно визначити чи містить цей список (тобто знайти у ньому) число 50. Поділимо список навпіл і визначимо який з двох підсписків може містити шуканий елемент.

0	1	2	3	4	5	6	7	8	9	10
1	10	12	13	20	32	33	56	66	71	95

Для цього досить порівняти центральний елемент 32 (індекс $10/2 = 5$) з шуканим елементом 50.

0	1	2	3	4	5	6	7	8	9	10
1	10	12	13	20	32	33	56	66	71	95

Далі потрібно повторити описану послідовність дій для отриманого підсписку. Ця послідовність дій повторюється доти, доки не лишиться один єдиний елемент:

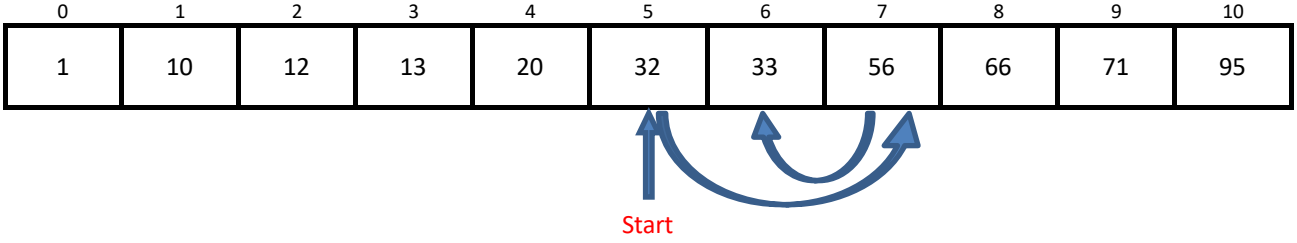


Рисунок 3.1.2. Бінарний пошук по впорядкованому масиву.

На останньому кроці досить порівняти останній отриманий елемент з шуканим. У нашому прикладі, останній елемент 33 не дорівнює шуканому елементу 50, отже список не місить шуканого числа. Як бачимо, для розв'язання задачі пошуку знадобилося лише три кроки. Проаналізуємо скільки кроків алгоритму у загальному випадку необхідно здійснити для пошуку. Як вже було зазначено, основна ідея бінарного пошуку полягає у тому,

що потрібно постійно ділити список навпіл, доки для розгляду не лишиться один останній елемент. Це означає, що алгоритмічна складність алгоритму бінарного пошуку є $O(\log n)$. Таким чином, якщо список складається з $1K$ елементів, то знайти щось у ньому можна всього за 10 кроків.

Як бачимо, ідея алгоритму надзвичайно проста, тому зразу розглянемо його реалізацію на мові Python. Напишемо програму що лише дає відповідь на питання чи містить відсортований список `array` шуканий елемент `x`. На кожній ітерації будемо запам'ятовувати позицію лівого та правого індексів підсписку, що може містити шуканий елемент.

Лістинг 3.1.2. Реалізація бінарного пошуку у масиві.

```
def binary_search(array, x):
    """ Бінарний пошук у масиві

    :param array: Список елементів
    :param x: Шуканий елемент
    :return: True, якщо шуканий елемент знайдено
    """
    left = 0                # Індекс лівого елементу
    right = len(array) - 1  # Індекс правого елементу

    while left < right:
        m = left + (right - left) // 2  # Індекс середнього елементу
        if x > array[m]:
            left = m + 1
        else:
            right = m

    return array[right] == x
```

Як було зазначено вище, така реалізація бінарного пошуку передбачає, що програма буде працювати доти, доки для аналізу не лишиться лише один елемент. Вона не завершиться, якщо на певній ітерації виконання алгоритму трапиться шуканий елемент. Наведемо модифікацію алгоритму бінарного пошуку з такою перевіркою.

Лістинг 3.1.3. Альтернативна реалізація бінарного пошуку у масиві.

```
def binary_search(array, x):
    """ Бінарний пошук у масиві

    :param array: Список елементів
    :param x: Шуканий елемент
    :return: Індекс шуканого елементу
    """
    left = 0                # Індекс лівого елементу
    right = len(array) - 1  # Індекс правого елементу

    while left <= right:
        m = left + (right - left) // 2  # Середина відрізка
        if x > array[m]:
            left = m + 1
        elif x < array[m]:
            right = m - 1
        else:
            return m

    return None
```

Наведена вище функція повертає індекс шуканого елементу, якщо він міститься у масиві, та `None` у іншому разі. При цьому, якщо масив містить кілька однакових елементів, то функція поверне індекс одного з них – того, який першим стане на шляху бінарного пошуку. Проте часто постає задача не лише дати відповідь на питання чи міститься певний елемент у масиві, а скажімо, знайти індекс першого чи останнього входження шуканого елементу до масиву. Тут потрібно зазначити, що якщо масив містить значну кількість однакових елементів, то алгоритмічна складність такого пошуку може зростати до лінійної, при реалізації простого перебору всіх однакових елементів масиву. Тому постає задача модифікувати алгоритм бінарного пошуку таким чином, щоб результатом його виконання була безпосередня відповідь на поставлене запитання.

Спочатку наведемо модифікацію бінарного пошуку, що повертає найперше входження шуканого числа x до впорядкованого за неспаданням масиву чисел array. У разі, якщо шукане число відсутнє у масиві, функція поверне індекс першого елементу, що більший за число x.

Лістинг 3.1.4. Бінарний пошук – найперше входження заданого числа.

```
def bsearch_leftmost(array, x):
    """ Бінарний пошук для відшукання найпершого входження заданого числа

    :param array: Відсортований за неспаданням масив цілих чисел
    :param x:     Шукане число
    :return:      Номер шуканого елемента у масиві
    """
    left = 0            # ліва границя пошуку
    right = len(array)  # права границя пошуку
    while left < right:
        m = left + (right - left) // 2  # середина
        if array[m] < x:
            left = m + 1
        else:
            right = m

    return left
```

Тут слід звернути увагу читача на те, що така модифікація алгоритму, на відміну від двох попередніх, здійснює пошук не на відрізку, а на напівінтервалі – права границя на початку роботи алгоритму вказує не на останній елемент, а на фіктивний елемент, що розташовується після останнього елементу масиву. При цьому, якщо всі елементи масиву array менші за задане число x, то підпрограма поверне індекс саме цього фіктивного елементу. Аналогічні особливості має модифікація бінарного пошуку, що повертає індекс останнього входження заданого числа.

Лістинг 3.1.5. Бінарний пошук – останнє входження заданого числа.

```
def bsearch_rightmost(array, x):
    """ Бінарний пошук для відшукання останнього входження заданого числа

    :param array: Відсортований за неспаданням масив цілих чисел
    :param x:     Шукане число
    :return:      Номер шуканого елемента у масиві
    """
    left = 0            # ліва границя пошуку
    right = len(array)  # права границя пошуку
    while left < right:
        m = left + (right - left) // 2  # Середина
        if array[m] <= x:
            left = m + 1
        else:
            right = m

    return left - 1
```

У випадку, якщо шукане число x відсутнє у заданому масиві array, підпрограма повертає номер останнього елементу, що менший за число x. Як наслідок, якщо всі елементи масиву більші за задане число, то програма поверне значення -1 (мінус один). У подальшому це може породжувати помилки у програмі, що пов'язані з неправильною інтерпретацією такого результату, адже у Python індексом -1 позначається останній елемент масиву.

Наведені вище алгоритми бінарного пошуку, що повертають перше та останнє входження шуканого числа у впорядкованому масиві широко використовуються для різноманітних алгоритмічних задач, наприклад, для швидкого підрахунку кількості елементів масиву, що потрапляють до заданого проміжку.

Дійсний бінарний пошук

Припустимо задано функцію f на відрізку $[a, b]$, що є неспадною (не зростаючою) на області визначення. Припустимо задано дійсне число c. Задача дійсного бінарного пошуку полягає у тому, щоб знайти найменше таке

число $x \in [a, b]$, що $f(x) \geq c$ (або найбільше таке $x \in [a, b]$, що $f(x) \leq c$). Іншими словами, необхідно наближено розв'язати рівняння $f(x) = c$, на проміжку $[a, b]$.

Фактично ідея дійсного бінарного пошуку майже нічим не відрізняється від цілочисельного. Аналогічно до цілочисельного пошуку, будемо на кожній ітерації ділити відрізок $[l, r]$, на якому шукається розв'язок (спочатку $[l, r] = [a, b]$) навпіл і, обчислюючи значення функції у точці поділу $m = (l + r)/2$, визначати у якому з отриманих інтервалів знаходиться розв'язок. Проте, на відміну від цілочисельного бінарного пошуку, постає питання, а коли можна вважати, що ми знайшли розв'язок і що саме вважати розв'язком? Дійсно, як ми знаємо, на будь-якому інтервалі міститься безліч дійсних чисел, а відтак поділ відрізка може відбуватися нескінченно.

Отже, для визначення моменту завершення пошуку використовують три підходи:

1. точність по аргументу;
2. точність по значенню;
3. безпосереднє сусідство двох дійсних чисел.

Розглянемо окремо кожен з них.

Перший підхід пропонує вважати, що пошук завершується у випадку, якщо довжина поточного відрізку $[l, r]$ стає меншою за деяке, наперед визначене додатне число ε

$$r - l < \varepsilon.$$

У цьому випадку розв'язком можна вважати будь-яке значення з останнього інтервалу пошуку – наприклад середину

$$x = \frac{l + r}{2}.$$

Другий підхід пропонує вважати, що пошук завершується у випадку, якщо у середині поточного відрізку $[l, r]$, тобто у точці

$$m = \frac{l + r}{2},$$

функція f має значення, що мало відрізняється від значення c:

$$|f(m) - c| < \varepsilon.$$

При цьому, m вважається розв'язком задачі.

Обидва вищезазначених випадки мають серйозну проблему, яка пов'язана з зображенням дійсних чисел у пам'яті комп'ютера. Може трапитися так, що задана точність ε є такою, що після кількох ітерацій бінарного пошуку, кінці відрізка $[l, r]$ будуть сусідніми дійсними числами. При цьому, очевидно, наступний поділ відрізка навпіл буде породжувати цей же відрізок. Очевидним чином відбудеться зациклення програми.

Читач знайомий з математичним аналізом у цей момент запротестує, апелюючи до того, що з точки зору математики такого не може бути. Дійсно, з математичного аналізу відомо, що дійсні числа щільно заповнюють будь-який відрізок. Проте, на жаль, цього не можливо досягти при моделюванні дійсних чисел у пам'яті комп'ютера – об'єм пам'яті, що виділяється для кожного числа у пам'яті комп'ютера обмежений. Виділеної пам'яті досить для того, щоб змоделювати обмежену множину раціональний чисел, причому цю множину можна впорядкувати за зростанням. Тому, коли мова йде про дійсні числа які обробляються комп'ютером, будемо здебільшого використовувати термін «числа дійсного типу» для того щоб підкреслити, що мова йде про цю множину раціональних чисел, яка моделює дійсні числа. Слід зауважити, що такого зображення дійсних чисел цілком досить для різноманітних математичних задач. Проте з іншого боку, програміст, що має туманне уявлення про зображення чисел у пам'яті комп'ютера приречений на боротьбу з різноманітними помилками, у тому числі із зазначеними вище.

Інша проблема, з якою може зіштовхнутися у процесі реалізації алгоритму бінарного пошуку програміст, це те, що функція f на відрізку $[a, b]$ може дуже швидко зростати (спадати) або дуже повільно зростати (спадати). У першому випадку скоріше за все виникне вищеописана проблема, що кінці відрізка $[l, r]$ на певному етапі перетворяться у два сусідні дійсними числами. У іншому випадку точність знайденого розв'язку буде надзвичайно низькою.

Отже, для того, щоб на практиці подолати описані вище проблеми, пропонуємо використовувати останній третій підхід для визначення моменту завершення пошуку, а саме завершувати пошук, коли кінці відрізка $[l, r]$ будуть сусідніми дійсними числами. Для цього досить перевірити, що середина відрізка $[l, r]$ збігається з одним з кінців l або r.

Реалізуємо алгоритм дійсного бінарного пошуку, що використовує для моменту завершення пошуку останній підхід, а саме безпосереднє сусідство двох дійсних чисел.

Лістинг 3.1.6. Реалізація бінарного пошуку для знаходження розв'язку рівняння.

```
def binary_continuous(f, c, a, b):
    """ Для монотонної на відрізку [a, b] функції f розв'язує рівняння
                    f(x) = c

    :param f: Монотонна функція
    :param c: Шукане значення
    :param a: Ліва межа проміжку на якому здійснюється пошук
    :param b: Права межа проміжку на якому здійснюється пошук
    :return: Розв'язок рівняння
    """
    left = a                  # лівий кінець відрізка
    right = b                 # правий кінець відрізка

    m = (left + right) / 2.0  # середина відрізка [left,right]
    while left != m and m != right:
        if f(m) < c:
            left = m   # [left,right] = [x,right]
        else:
            right = m  # [left,right] = [left,x]

        m = (left + right) / 2.0  # середина відрізка [left,right]

    return left
```

Для перевірки роботи алгоритму знайдемо розв'язок рівняння

$$\tan x - 2x = 0$$

на відрізку $[0.5, 1.5]$. Легко переконатися, що функція записана у лівій частині рівняння є монотонною, а її значення у на лівому та правому кінцях відрізка мають різний знак. Відтак можемо скористатися вищеописаним алгоритмом бінарного пошуку.

Лістинг 3.1.6. (продовження).

```
import math
print(binary_continuous(lambda x: math.tan(x) - 2.0 * x, 0, 0.5, 1.5))
```

Результатом виконання вищенаведеного коду буде

```
1.1655611852072112
```

Бінарний пошук по відповіді

Бінарний пошук можна застосовувати не лише для відшукання елементів впорядкованих масивів чи розв'язання рівнянь з монотонними функціями, але й для розв'язання різноманітних задач де необхідно знайти певне значення.

Для розв'язання таких задач необхідно задати вихідну область пошуку, що визначається лівою l та правою r межами, перша з яких є заздалегідь меншою за відповідь, а права – більшою. Ця область пошуку (тобто відрізок $[l, r]$) фактично і є впорядкованою множиною потенційних розв'язків серед яких потрібно обрати той, що задовольняє умову задачі. Бінарний пошук полягає у тому, що на кожному кроці алгоритму обчислюється значення визначеної умовою задачі характеристики у точці, що є серединою області пошуку і, залежно від отриманого значення, відбувається звуження області пошуку на ліву чи праву її підобласті.

Описаний вище підхід називається бінарним пошуком по відповіді. Розглянемо його детальніше на прикладах.

Приклад 3.1.1. [e-olimp, 5102]. Сьогодні вранці журі вирішило додати у варіант олімпіади ще одну, Дуже Легку Задачу. Відповідальний секретар оргкомітету надрукував її умову в одному екземплярі, і тепер йому потрібно до початку олімпіади встигнути зробити ще n копій. У його розпорядженні є два ксерокси, один з яких копіює аркуш за x секунд, а другий за y. (Дозволяється використовувати як один ксерокс, так і обидва одночасно. Можна копіювати не лише з оригінала, але і з копії.). Допоможіть йому вияснити, який мінімальний час для цього потрібно.

Як було зазначено вище, розв'язання задачі необхідно почати з визначення вихідної області пошуку. Очевидно, що мінімальний час витрачений на друк не може бути меншим за нуль, а максимальний час – час необхідний для того, щоб надрукувати всі копії на одному ксероксі. Тут слід зауважити, що оскільки друк буде відбуватися на двох ксероксах одночасно, а спочатку є наявною лише один екземпляр умови, то першим кроком необхідно зробити ще один екземпляр умови, з якої почнеться друк на другому ксероксі. Цей випадок необхідно виокремити, оскільки він не потраплятиме під загальний підхід. Очевидно, що друк додаткового екземпляру варто здійснювати на швидшому ксероксі і займе цей процес $\min(x, y)$ секунд (важливо не забути врахувати цей час у відповіді). Після цього лишиться надрукувати $n - 1$ екземпляр умови. Отже, областю потенційних розв'язків буде відрізок

$$[l, r] = [0, (n - 1)\max(x, y)].$$

Далі починає працювати бінарний пошук – вираховуємо середину області

$$m = \frac{l + r}{2}$$

та рахуємо скільки повних сторінок можна надрукувати за цей час m використовуючи обидва ксерокси. Якщо кількість сторінок менша за $n - 1$, то змінюємо нижню межу границі області пошуку, інакше – праву.

Лістинг 3.1.7. Бінарний пошук по відповіді.

```
n, x, y = [int(i) for i in input().split()]  # зчитування вхідних даних
x, y = min(x, y), max(x, y)  # впорядковуємо змінні, так щоб x <= y.

l = 0
r = (n - 1) * x
while l < r:
    m = (r + l) // 2     # середина області пошуку
    k = m // x + m // y  # кількість копій за час m використовуючи обидва принтери
    if k < n - 1:
        l = m + 1
    else:
        r = m

print(l + x)  # враховуємо час друку першої копії
```

Бінарний пошук є дуже швидким алгоритмом, проте як ми знаємо для його застосування необхідно, щоб колекція була впорядкована, а впорядкування колекції, з алгоритмічної точки зору, це досить не проста задача і як ми побачимо пізніше, найоптимальніші алгоритми сортування мають асимптотичну складність $O(n \log n)$. А тому застосування бінарного пошуку є виправданим (якщо початково колекція невпорядкована) у випадку, якщо необхідно виконати багаторазовий пошук (принаймні двічі), оскільки операції пошуку передує операція сортування колекції.

3.1.3. Хешування та хеш-таблиці

У попередніх пунктах ми розглянули алгоритми пошуку – лінійний та бінарний. Лінійний пошук дуже повільний – він здійснюється за лінійний час. Бінарний пошук є швидким – працює за логарифмічна час – проте, може виконуватися лише для впорядкованих списків.

Спробуємо піти ще на крок далі: побудуємо таку структуру даних, в якій можна буде здійснювати пошук за сталий час $O(1)$. Ця концепція використовує механізм **хешування**.

Хеш-функції

Означення 3.1.2. **Хешуванням** називається операція перетворення вхідного масиву даних довільної довжини у вихідний бітовий рядок фіксованої довжини за допомогою деякого визначеного алгоритму.

Перетворення вхідного масиву даних відбувається за допомогою спеціальної функції яку називають **хеш-функцією** або **функцією згортки**.

Вхідні дані, на які діє хеш-функція, називаються **ключем** (інколи **повідомленням**). Результат дії хеш-функції на вхідні дані називається **хеш-значенням**, **хеш-кодом** або просто **хешем**. Хеш це натуральне число, яке часто записують у шістнадцятковому вигляді:

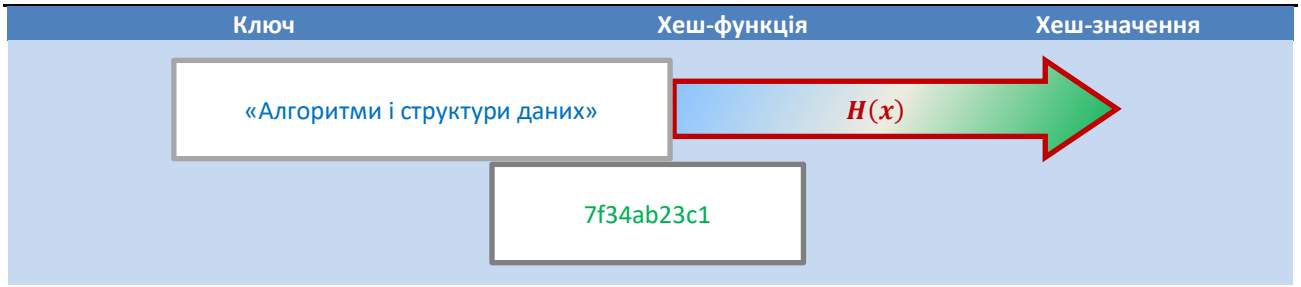


Рисунок 3.1.3 Хешування рядка.

Приклад 3.1.2. Прикладом простої хеш-функції, що діє на натуральні числа може бути, функція обчислення остачі від ділення на деяке число, скажімо 11

$$H(x) = x \,\%\, 11$$

Наведемо кілька прикладів обчислення хешів з допомогою цієї функції $H(x)$

Ключ	Хеш (десяткова система числення)	Хеш (шістнадцяткова система числення)
65	10	a
313222	8	8
12	1	1
777	7	7
5625242255631	9	9
1065	9	9

Як бачимо, ця хеш-функція ставить у відповідність натуральному числу, інколи досить великому, ціле число з проміжку від 0 до 10. З наведеного прикладу добре прослідковуються основні найважливіші характеристики яким має задовольняти хеш-функція.

Характеристики хеш-функції

1) На виході хеш-функції завжди отримуються значення фіксованої довжини:

$$H(65) = 10 \leq 10$$
$$H(5625242255631) = 9 \leq 10$$

   Навіть, якщо на вхід хеш-функції надійдуть дані велетенської довжини, довжина значення на виході не зміниться. Аналогічна ситуація має місце і для випадку, якщо дані на вході хеш-функції будуть малого розміру. Довжина вихідних значень хеш-функції залежить від типу та явного вигляду хеш-функції. Таке співставлення може бути досить корисним для різних типів задач.
2) Для однакових вхідних даних, хеш-функція незмінно повертає один і той же результат

$$H(65) = 10 = H(65)$$

3) Якщо вхідні дані дуже схожі, проте відрізняються хоча б одним бітом, вихідні дані будуть різними, інколи навіть суттєво різними

$$H(65) = 10$$
$$H(66) = 0$$

4) Трапляється ситуація, коли для цілком різних даних, результатом хеш-функції буде одне значення

$$H(5625242255631) = 9$$
$$H(1065) = 9$$

   Така ситуація називається *колізією хеш-функції* і буде детально розглянута нижче.
5) Відновити вхідні дані за їхнім хешем практично неможливо, навіть знаючи явний вигляд самої хеш-функції.
6) Хеш-функція має просту алгоритмічну складність, для того, щоб уникнути зайвого обчислювального навантаження.

Колізії хеш-функції

Якщо уважно подивитися на вищенаведену хеш-функцію можна помітити, що вона може для різних вхідних даних повертати однаковий хеш. Дійсно,

$$H(1) = 1$$
$$H(34) = 1$$
$$H(342) = 1$$

Така ситуація називається колізією хеш-функції.

Означення 3.1.3. **Колізією** хеш-функції (**хеш-конфліктом**) називається випадок, при якому хеш-функція для різних вхідних блоків даних повертає однакові хеш-коди, тобто

$$x \neq y \Rightarrow H(x) = H(y)$$

Колізії існують для більшості хеш-функцій, що застосовуються на практиці. Проте для «хороших» хеш-функцій частота їхнього виникнення повинна бути близькою до теоретичного мінімуму.

Означення 3.1.4. **Ідеальною хеш-функцією** називається функція, що відображає кожен ключ з деякого набору ключів у множину цілих чисел без колізій.

Очевидно, що ідеальну хеш-функцію легко побудувати для випадку, коли множина різних вхідних даних є скінченною. Проте на практиці найчастіше виникає ситуація коли потрібно хешувати дані різного розміру за допомогою хеш-кодів сталої довжини. У такому разі не можливо уникнути колізій.

Види хеш-функцій

Підбір хеш-функції залежить від типу даних які мають хешуватися з її допомогою. Наприклад, для цілих чисел часто використовують операцію остача від ділення, а наприклад, для рядків символів – операції що використовують їхні коди.

Можна навести безліч прикладів, проте розрізняють хороші і погані хеш-функції. Хороша хеш-функція повинна задовольняти таким умовам:

- швидко обчислюватися;
- мінімізувати кількість колізій.

Крім цих властивостей часто від хеш-функцій вимагають виконання деяких додаткових вимог, таких наприклад, як неможливість відновлення вхідних даних за їхнім хешем.

Розглянемо які види хеш-функцій застосовуються практиці.

Хеш-функції на основі ділення

Найпростіший спосіб побудови хеш-функції для ключів з множини натуральних чисел полягає у тому, що у ролі хешу натурального числа x використовується залишок від його ділення на M:

$$H(x) = x \ \% \ M,$$

де M — це кількість всіх можливих хешів. На практиці M зазвичай обирають простим. Такий вибір допомагає ефективно розв'язувати проблеми, які виникають під час колізій.

Хеш-функції на базі множення

Інший спосіб побудови хеш-функцій полягає у виборі деякої цілої константи A, що є взаємно простою з w, де w –кількість можливих варіантів значень у вигляді машинного слова (наприклад для 32-бітних операційних системах $w = 2^{32}$). Тоді хеш-функцію можна визначити як

$$H(x) = \left[M \left[\frac{A}{w} \cdot x\right]\right].$$

У цьому випадку, як правило M обирають як степінь числа 2.

Хеш-функції на основі методу середніх квадратів

Як і попередні два методи побудови хеш-функції, метод середніх квадратів також застосовується до ключів натурального типу. Цей метод полягає у тому, що значення ключа підноситься до квадрату, а далі з отриманого результату виділяється деяка порція цифр. На базі якої і будується хеш-значення.

Приклад 3.1.3. Розглянемо ключ 44. Піднесемо його до квадрату і отримаємо $44^2 = 1{,}936$. Виділимо дві середніх цифри отриманого числа – 93. Знайдене число можна вважати хеш-значенням вхідного елементу.

Проте, частіше за все до отриманого значення застосовують допоміжну хеш-функцію, наприклад $h_1(x) = x \% 11$. Тоді, хеш-значення ключа 44 буде дорівнювати 5.

Згортка

Згорткою називається метод побудови хеш-функції, у якому для обчислення хешу вхідний елемент ділиться на складові частини однакового розміру – фрагменти (можливо, крім останнього, який може мати менший розмір). Кожному фрагменту ставиться у відповідність певне ціле значення (як правило за допомогою деякої допоміжної хеш-функції). Отримані значення підсумовуються. Результуюче хеш-значення обчислюється з отриманої суми за допомогою ще однієї допоміжної хеш-функції.

Отже, нехай K – вхідний ключ, $(a_1, \dots a_l)$ – послідовність фрагментів на які розбито ключ. Тоді хеш-функцію можна визначити у такому вигляді

$$H(K) = \big(h_1(a_1) + \cdots + h_l(a_l)\big) \% M,$$

де $h_i, i = 1, \dots, l$ – послідовність допоміжних хеш-функцій, M – кількість всіх можливих хешів.

Приклад 3.1.4. Знайдемо методом згортки хеш для ключа, що є телефонним номером

$$K = +380 - 44 - 256 - 0540.$$

Візьмемо послідовність його цифр і поділимо її на групи (фрагменти) по два $(38, 04, 42, 56,\ 05, 40)$. Додаючи отримані фрагменти та обчисливши остачу від ділення на 11 від отриманої суми

$$H(K) = (38 +\ 04 +\ 42 +\ 56 + 05 +\ 40) \% 11 = 185 \% 11 = 9$$

отримаємо хеш-значення 9 для вхідного ключа K.

Хеш-функції для рядків

Для побудови хеш-функцій для рядків частіше за все використовується метод згортки. Наприклад, для хешування ключа S, що складається з $l + 1$ символів $S = "c_0c_2 \dots c_l"$, можна запропонувати формулу для обчислення значення хеш-функції у такому вигляді

$$H(S) = \big(h_0(c_0) + h_1(c_1) + \cdots + h_l(c_l)\big) \% M,$$

де $h_i, i = 0, \dots, l$ – послідовність допоміжних хеш-функцій, що діють на символи рядка, M – кількість всіх можливих хешів.

Приклад 3.1.5. Розглянемо для прикладу рядок «hash». У ролі усіх допоміжних хеш-функцій $h_i, i = 0, \dots,3$ будемо використовувати функцію, що повертає код символа:

$$h_i(c_i) = ord(c_i), \qquad i = 0, \dots,3.$$

Сталу M, як і раніше, виберемо 11. Визначимо коди символів, з яких складається рядок:

символ	h	a	s	h
код	104	97	115	104

Просумуємо отримані значення та знайдемо остачу від ділення на 11 для отриманої суми:

$$H("hash") = (104 + 97 + 115 + 104) \% 11 = 420 \% 11 = 2.$$

Таким чином, хеш-значення вхідного рядка «hash» буде 2.

Зауваження. Зазначений у прикладі 3.1.5 метод побудови хеш-функції для рядків не є хорошим, оскільки різні рядки, що складаються з однакових символів, наприклад «hash» та «hsah» будуть мати однакові хеші, що є незадовільним з точки зору властивостей, яким має задовольняти хеш-функції. Цю проблему можна подолати, якщо послідовність допоміжних функцій визначити так, щоб кожна з допоміжних функцій $h_i, i = 0, \dots, l$ мала явну залежність від позиції символа у рядку на який вона діє. Наприклад, можна модифікувати функції h_i, визначені вище, додавши позицію символа у ролі вагового коефіцієнту

$$h_i(c_i) = (i + 1) \cdot ord(c_i), \qquad i = 0, \dots,3.$$

Інший спосіб побудови хеш-функції для рядків методом згортки, що враховують позицію символів у рядку може бути описаний таким чином. Виберемо деяке просте натуральне число N, що не перевищує 255, наприклад 31. Тоді формула побудови хешу для рядка $S = "c_0c_2 \dots c_l"$ буде мати вигляд

$$H(S) = \left(ord(c_0) * N^l + ord(c_1) * N^{l-1} + \cdots + ord(c_{l-1}) * N^1 + ord(c_l)\right) \% M,$$

Таке зображення хеш-функції, при досить великому значенні M дозволяє значно зменшити ймовірність колізій. У вигляді коду ця функція буде мати такий простий вигляд

Лістинг 3.1.8. Реалізація хеш-функції для рядків.

```
N = 31      # Просте число, що не перевищує 255
M = 100007  # Кількість всіх можливих хешів

def H(S):
    h = 0
    for i in range(len(S)):
        h = h * N + ord(S[i])
    return h % M
```

Крім наведених вище методів хешування рядків, використовують інші методи побудови хеш-функцій, наприклад, алгоритм хешування Пірсона [ ]. З ним та іншими методами, ми пропонуємо познайомитися читачу самостійно.

Хеш таблиці

Одним з застосувань хешування є оптимізація пошуку серед набору даних.

Означення 3.1.5. Асоціативний масив (eng. associative array) – це абстрактний тип даних, що дозволяє зберігати дані у вигляді набору пар ключ — значення та доступом до значень за їхніми ключами.

Фактично асоціативний масив це структура, що дозволяє зберігати пари ключ – значення і здійснювати три основні операції:

- додавання нової пари (ключ, значення);
- пошук значень за ключем;
- видалення пари ключ – значення за ключем.

Зауваження. Мова програмування Python має вбудовану структуру даних словник (`dict`), яка реалізує інтерфейс асоціативного масиву.

Одна з реалізацій асоціативного масиву базується на застосуванні хеш-таблиці.

Означення 3.1.6. **Хеш-таблиця** (eng. hash table, hash-map) це структура даних, що реалізує інтерфейс асоціативного масиву, у якій додавання, пошук та видалення даних здійснюється за сталий час $O(1)$.

Реалізація хеш-таблиці здійснюється за допомогою масиву наперед визначеного розміру.

0	1	2	3	4	5	6	7	8	9	10
None	None	None	None	None	None	None	None	None	None	None

Рисунок 3.1.4

Як відомо доступ до елементів у масиві здійснюється за допомогою індексу (тобто номеру елемента), який у випадку реалізації хеш-таблиці називають **слотом**. Отже, хеш-таблиця наведена на рисунку вище має 11 слотів, що мають номери від 0 до 10. Кількість слотів хеш-таблиці називається її розміром.

Спочатку хеш-таблиця не містить ніяких елементів, оскільки кожен з них порожній (у Python будемо ініціалізувати їх значенням **None**).

Виконання операцій з хеш-таблицею починається з обчислення значення хешу для ключа. Отримане значення хешу є індексом у масиві, тобто визначає номер слоту де містяться дані або куди треба вставити дані.

Припустимо, що ключі хеш-таблиці є натуральними числами. Тоді, виберемо хеш-функцію

$$H(x) = x \% 11 \qquad (3.1.1)$$

– остача від ділення ключа на 11. Як ми вже знаємо, число 11 вибрано не випадково – це кількість всіх можливих хешів, що власне і визначає розмір хеш-таблиці. Зауважимо, що як правило, на практиці, операція «остача від ділення» присутня у тій чи іншій формі у всіх хеш-функціях, оскільки хеш-код елементу має обов'язково знаходитися у діапазоні номерів слотів хеш-таблиці.

Припустимо ми хочемо розмістити у хеш-таблиці дані, що мають ключами числа $\{54, 26, 93, 17, 77, 31\}$. Обчислимо їхні хеші

Ключ	Хеш
54	10
26	4
93	5
17	6
77	0
31	9

Нам лишається розмістити вихідні значення за їхніми хеш-кодами у відповідні слоти хеш-таблиці, як показано на рисунку нижче

0	1	2	3	4	5	6	7	8	9	10
77	None	None	None	26	93	17	None	None	31	54

Рисунок 3.1.5

Таким чином, очевидно, що використовуючи хеш-таблиці ми можемо знайти потрібний елемент серед набору за сталий час $O(1)$. Дійсно, для цього нам потрібно лише обчислити хеш для шуканого елементу і звернутися за відповідним індексом до хеш-таблиці. Оскільки кожна з зазначених операцій має складність $O(1)$, то і складність пошуку складає $O(1)$.

При роботі з хеш-таблицями важливу роль відіграє величина, яка називається фактором (коефіцієнтом) заповнення хеш-таблиці. Ця величина визначається як відношення кількості зайнятих слотів (n) до загального числа слотів (N):

$$\lambda = \frac{n}{N}$$

У нашому випадку, після додавання елементів у таблицю, $\lambda = 6/11$. Фактор заповнення хеш-таблиці вважається важливим параметром від якого напряму залежить середній час виконання операцій у хеш-таблиці.

Розв'язання колізій

Зазначена вище техніка коректно працює лише у випадку, якщо кожен елемент претендує на унікальну позицію в хеш-таблиці, тобто різні ключі мають різні хеш-значення. Якщо ж нам, наприклад, потрібно додати у раніше створену хеш-таблицю ключ 44, то виникне колізія. Дійсно, ключ 44 відповідно до нашої хеш-функції має хеш 0. Такий же хеш має ключ 77, який уже міститься у таблиці. Виникає питання: яким чином розмістити у хеш-таблиці новий ключ 44 і чи можливо це взагалі зробити?

Виникнення колізії це досить поширене явище під час процедури хешування. Наприклад, при додаванні у таблицю розміром 365 слотів усього лише 23 елементів, ймовірність колізії вже перевищує 50%[1].

Як ми знаємо, теоретично, уникнути колізій можливо у випадку використання ідеальної хеш-функції. Але, на практиці така ситуація практично неможлива – множина ключів може бути необмеженою, а множина хешів – обмежена, що диктується властивостями хеш-функцій. Таким чином, розв'язання колізій є важливою частиною хешування.

Означення 3.1.7. Систематичний метод розміщення у хеш-таблиці елементів, що мають однакові хеші називається **процесом розв'язанням колізій**.

Існує два основних типи хеш-таблиць

- з відкритою адресацією,
- з ланцюжками,

які розрізняються способом розв'язання колізій у хеш-таблиці.

Хеш-таблиці з відкритою адресацією

У хеш-таблицях з відкритою адресацією під час розв'язання колізій проводиться пошук іншого вільного місця для розміщення елементу. Щоб це зробити, починаючи з оригінальної позиції (тобто зі слоту, номер якого є хешем елементу), будемо переміщуватися по слотах деяким визначеним способом, доти, доки не буде знайдено порожній слот, власне у який і буде записаний елемент. Зауважимо, що можливо буде необхідним повернутися до першого слоту таблиці циклічно, щоб охопити таблицю повністю. Якщо спосіб пошуку вільного слоту полягає

[1] Відомий факт з теорії ймовірності, що називається «парадокс днів народження».

у послідовному відвідуванні всіх слотів, починаючи з оригінального, то такий метод розв'язання колізій називається **лінійним зондуванням**.

Для прикладу, припустимо, необхідно додати по черзі елементи 44, 55, 20 та 13, до раніше створеної хеш-таблиці, заповнення слотів якої зображено на рисунку 3.1.5. Хеш-значення, обчислене за формулою (3**.1.**1) для елементу 44 дорівнює 0. Як бачимо з рисунка 3.1.5, слот з номером 0 зайнятий іншим елементом. Тоді для елементу 44 шукаємо перший вільний слот – він має номер 1. Розміщуємо туди елемент 44:

0	1	2	3	4	5	6	7	8	9	10
77	44	None	None	26	93	17	None	None	31	54

Рисунок 3.1.6

У цьому прикладі, фоном будемо виділяти вставлені у таблицю елементи.

Вставимо тепер елемент 55, хеш якого також дорівнює 0. Вільним слотом після оригінального, буде вже слот з номером 2. Після вставки у слот з номером 2 елемента 55, хеш-таблиця набуде вигляду

0	1	2	3	4	5	6	7	8	9	10
77	44	55	None	26	93	17	None	None	31	54

Рисунок 3.1.7

Для елемента 20, хеш якого дорівнює 9, відповідний слот також зайнятий. Щоб знайти вільний слот методом лінійного зондування, прийдеться почати пошук з початку таблиці, оскільки останній слот таблиці, що має номер 10 також зайнятий. В результаті розміщуємо елемент 20 у слот номер 3.

0	1	2	3	4	5	6	7	8	9	10
77	44	55	20	26	93	17	None	None	31	54

Рисунок 3.1.8

Нарешті, вставка елемента 13 відбудеться аж у позицію 7.

0	1	2	3	4	5	6	7	8	9	10
77	44	55	20	26	93	17	13	None	31	54

Рисунок 3.1.9

Пошук елемента у хеш-таблиці здійснюється аналогічним чином. Припустимо необхідно знайти елемент 77 у таблиці. Його хеш значення буде 0. Звертаючись до слоту з номер нуль виявимо там число 77. Отже шуканий елемент знайдений. Тепер, припустимо, що необхідно знайти у таблиці елемент 55. Його хеш також 0. Проте, слот з номером 0 містить інший елемент. Це не означає, що таблиця не містить елементу 55 – під час заповнення таблиці могла відбутися колізія (що власне і відбулося) і елемент 55 був розміщений у іншому слоті. Починаємо послідовно переглядати всі елементи починаючи з наступної позиції, доки не буде знайдений елемент 55 або не зустрінемо **None**. Останнє буде означати, що таблиця не містить шуканого елементу. Таким чином знаходимо шуканий елементи у позиції 2.

Недоліком лінійного зондування є його схильність до кластеризації – елементи у таблиці групуються. Це означає, що якщо виникає багато колізій з одним хеш-значенням, то слоти, що знаходяться поруч під час лінійного зондування будуть заповнені. Нижче на рисунку, виділено кластер елементів, що мають хеш 0.

0	1	2	3	4	5	6	7	8	9	10
77	**44**	**55**	20	26	93	17	13	None	31	54

Рисунок 3.1.10

Кластеризація в свою чергу впливає на вставку інших елементів – так ключі 20 та 13 у нашому прикладі були вставлені у позиції, що знаходяться далеко від їхніх хеш-зазначень. Останнє, у свою чергу, значно сповільнює пошук елементів у таблиці, фактично перетворюючи пошук по хеш-таблиці у лінійний пошук.

Щоб подолати проблему кластеризації ключів, описану вище, замість лінійного зондування використовують інші способи пошуку вільних слотів у таблиці – послідовний пошук замінюють таким у якому, частину слотів пропускають (навіть якщо вони були вільні). Наприклад, під час колізії переглядають не кожен наступний слот, а через один. Або використовують не сталий крок пропуску, а такий, що змінюється за деяким відомим законом.

При такому пошуку вільного слоту для розміщення нового значення, відбувається більш рівномірний розподіл елементів, що викликають колізію. Це у свою чергу, зменшує кластеризацію.

Загальна назва для такого процесу пошуку іншого вільного слота після колізії – **повторне хешування**. При цьому використовується функція повторного хешування, яка повертає нове хеш-значення для поточного хешу елементу

$$h_{new} = R(h_{curr}), \tag{3.2}$$

тут h_{curr} – поточний хеш, h_{new} – новий хеш. Наприклад, під час звичайного лінійного зондування, функція повторного хешування матиме вигляд

$$h_{new} = (h_{curr} + 1) \ \% \ l_{table},$$

де l_{table} – розмір хеш-таблиці.

Під час зондування з пропуском через один, функція повторного хешування буде

$$h_{new} = (h_{curr} + 2) \ \% \ l_{table},$$

а якщо величина пропуску s, то функція повторного хешування буде мати вигляд

$$h_{new} = (h_{curr} + s) \ \% \ l_{table}.$$

Зауваження. Важливим є те, що величина пропуску s має бути такою, щоб в результаті відвідати усі слоти. Інакше, можливий випадок, що для вставки елементу не знайдеться вільного слоту, притому, що частина таблиці залишиться невикористаною. Найпростішим способом забезпечити виконання цієї умови визначити розмір таблиці простим числом. Саме тому у прикладах наведених вище під час побудови хеш-таблиці використовувалося 11 слотів.

Схема відкритої адресації дозволяє уникнути додаткових витрат пам'яті на розміщення кожного нового елементу, що є її значною перевагою перед іншими способами побудови хеш-таблиць. Проте, недоліком усіх схем відкритої адресації є те, що кількість елементів, які можуть зберігатися у таблиці може досягти кількості слотів у ній. На практиці, навіть з хорошими хеш-функціями, продуктивність серйозно падає, якщо фактор завантаження таблиці наближається до 0.7. Це, в свою чергу, викликає необхідність динамічного збільшення розміру хеш-таблиці з відповідними затратами.

Реалізація класу HashTable з використанням методу лінійного зондування

Реалізація хеш-таблиці вимагає опису таких основних операції:

- `put(key, value)` – додавання пари (ключ, значення). Якщо такий ключ вже існує, то ця операція заміняє старе значення новим;
- `get(key)` – за заданим ключем повертає значення з таблиці або `None`, якщо такий елемент відсутній у таблиці;
- `del` – видаляє пару ключ значення.

Крім цього перевантажують оператори

- `len()` – кількість пар ключ-значення, що містяться у колекції.
- `in` – перевірки входження ключа у колекцію.

Реалізуємо тип даних `HashTable`.

Лістинг 3.1.9. Реалізація хеш-таблиці методом лінійного зондування.

```
class HashTable:
    """Хеш-таблиця у якій колізії розв'язуються методом лінійного зондування."""

    def __init__(self):
        """ Конструктор - ініціалізує таблицю. """
        self.max_size = 11                    # кількість слотів таблиці
        self.current_size = 0                 # поточний розмір таблиці
        self.keys = [None] * self.max_size    # слоти таблиці, що містять ключі
        self.values = [None] * self.max_size  # дані пов'язані зі слотом
```

```
    def hash(self, key):
        """ Повертає хеш для ключа
        :param key: ключ
        :return: хеш ключа """
        return key % self.max_size
    def put(self, key, value):
        """ Додає пару (ключ, значення) до таблиці
        :param key: ключ
        :param value: значення
        :return: None  """
        current = self.hash(key)  # Поточний слот таблиці
        while self.keys[current] != None:
            if self.keys[current] == key:
                self.values[current] = value
                return
            current = (current + 1) % self.max_size
        # якщо ключ у таблиці не знайдений
        self.keys[current] = key      # додаємо ключ
        self.values[current] = value  # додаємо значення
        self.current_size += 1

    def get(self, key):
        """ Повертає значення за ключем
        :param key: ключ
        :return: значення  """
        current = self.hash(key)
        while self.keys[current] != None:
            if self.keys[current] == key:
                return self.values[current]
            current = (current + 1) % self.max_size
        # якщо ключ у таблиці не знайдений
        return None
```

Вправа 3.1.1. Запропонуйте та реалізуйте алгоритм видалення ключів у хеш-таблиці з відкритою адресацією?

Вправа 3.1.2. Запропонуйте алгоритм збільшення слотів хеш-таблиці, що використовує лінійне зондування для розв'язання колізій, для випадку, якщо фактор завантаження таблиці перевищує деякий поріг.

Лістинг 3.1.9. Продовження. Перевантаження спеціальних методів для класу HashTable.

```
class HashTable:
    """Хеш-таблиця у якій колізії розв'язуються методом лінійного зондування."""
    ...

    def __setitem__(self, key, value):
        """ Перевантаження оператора [ ] для запису
        :param key: ключ
        :param value: нове значення
        """
        self.put(key, value)

    def __getitem__(self, key):
        """ Перевантаження оператора [ ] для читання
        :param key: ключ
        :return: значення, що відповідає ключу key
        """
        return self.get(key)

    def __len__(self):
        """ Перевантаження вбудованого метода len()
        :return: Кількість елементів у таблиці.
        """
        return self.current_size

    def __contains__(self, key):
        """ Перевантаження оператора in
        :param key: ключ
        :return: True, якщо ключ міситься у таблиці.
        """
```

```
        return not (self[key] is None)

    def __str__(self):
        """ Перевантаження вбудованого методу str()
        :return: Зображення таблиці у рядковому вигляді
        """
        return str(self.keys) + '\n' + str(self.values) + '\n'
```

Подальша робота зі створеною таблицею нагадує роботу зі звичайним словником:

Лістинг 3.1.9.(продовження)

```
M = HashTable()  # Створюємо таблицю
M.put(55, "zz")   # додаємо пару (56, "zz")
M.put(66, "AA")   # додаємо пару (66, "AA")
M.put(66, "66")   # змінюємо значення за ключем 66
M.put(77, "77")   # додаємо пару (77, "77")

M[56] = "RR"      # M.put(56, "RR")
M[55] = "55"      # M.put(55, "55")

print(M[56])      # print(M.get(56))
print(len(M))
print(62 in M)
```

Хеш-таблиці з ланцюжками

Альтернативним методом розв'язання проблеми колізій є така організація хеш-таблиці у якій кожен слот містить не конкретний елемент, а посилання на колекцію елементів (ланцюжок), що мають одне і теж саме хеш-значення.

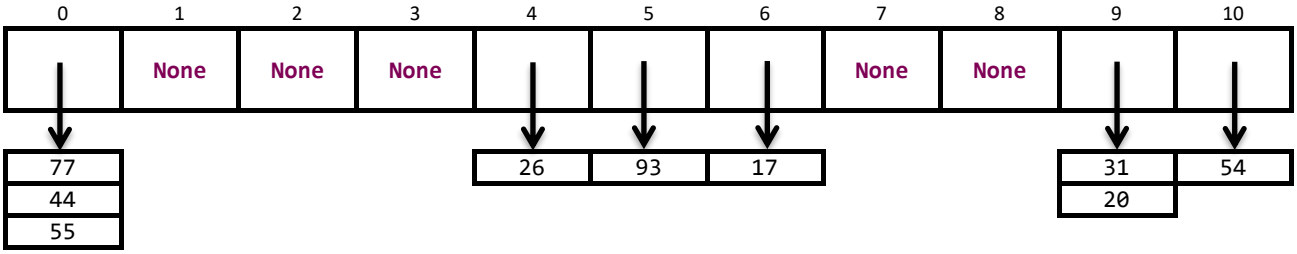


Рисунок 3.1.11. Хеш-таблиця з ланцюжками.

Хороша реалізація хеш-таблиці вимагає щоб пошук елемента у таблиці та вставка нового елементу до хеш-таблиці відбувалися за сталий час. Тому, як правило, вставка нового елементу у таблицю здійснюється як додавання його в кінець або у початок ланцюжка (залежно від організації структури самого ланцюжка). Наприклад, якщо ланцюжок таблиці реалізовано за допомогою стандартного списку Python, то очевидно, що оптимальним буде додавати нові елементи у кінець відповідного ланцюжка. Якщо ж ланцюжок оформлено у вигляді зв'язного списку, що буде розглянуто пізніше, то вставка здійснюється у його початок.

Нижче на рисунку наведено вставку у таблицю елементів 33, 37 та 57 у кінець відповідного ланцюжка.

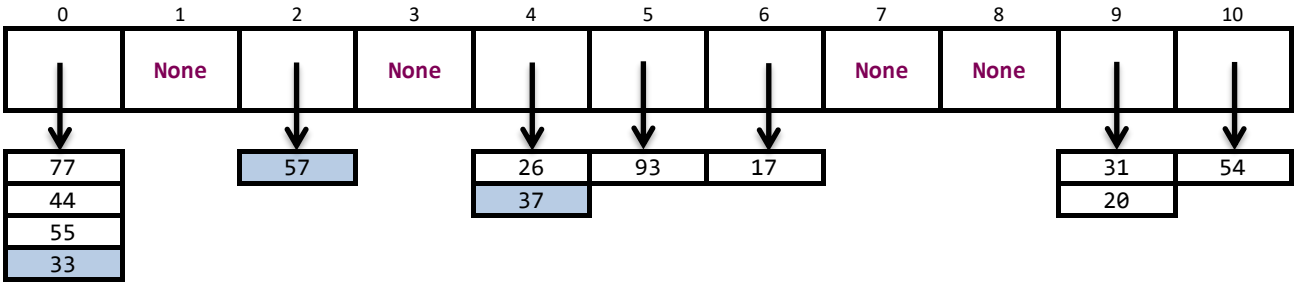


Рисунок 3.1.12. Додавання елементів до хеш-таблиці

Пошук серед елементів по хеш-таблиці здійснюється лінійним пошуком серед елементів відповідного ланцюжка. Очевидно, що чим більше елементів хешуються в один ланцюжок, тим важче знайти елемент у хеш-таблиці.

Реалізація класу HashTable з ланцюжками

Розглянемо реалізацію хеш-таблиці з розв'язанням колізій методом ланцюжків, у якій ланцюжки оформлено у вигляді зв'язного списку[2]. У такому разі описують допоміжний клас Node (Вузол таблиці), який є набором полів:

- key – ключ;
- value – значення, що відповідає ключу;
- next – посилання на наступний вузол таблиці, ключ якої має такий же хеш.

Крім зазначених вище полів, вузол таблиці може містити інші допоміжні поля, наприклад, поле valid, що містить означу чи вузол таблиці чинний (яка використовується для реалізації операції видалення ключів із таблиці).

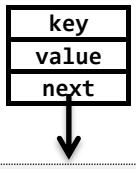


Рисунок 3.1.13. Вузол хеш-таблиці – містить посилання на наступний елемент.

Лістинг 3.1.10. Допоміжний клас Node – вузол таблиці.

```
class Node:
    """ Допоміжний клас вузол таблиці """
    def __init__(self, key: int, value):
        self.key = key      # ключ
        self.value = value  # значення
        self.next = None    # посилання на наступний вузол з таким же хеш-значенням
        self.valid = True   # чи чинний вузол (для операції видалення)
```

Звернемо увагу на те, що конструктор встановлює значення поля next як None для нових вузлів.

Лістинг 3.1.10. Продовження. Реалізація хеш-таблиці.

```
class HashTable:
    """Хеш-таблиця у якій колізії розв'язуються методом ланцюжків."""

    MAX_SIZE = 11                    # кількість слотів таблиці

    def __init__(self):
        """ Конструктор - ініціалізує таблицю. """
        self.slots = [None] * HashTable.MAX_SIZE  # слоти таблиці, що мітять ланцюжки вузлів

    @staticmethod
    def hash(key: int) -> int:
        """ Повертає хеш для ключа
        :param key: ключ
        :return: хеш ключа
        """
        return key % HashTable.MAX_SIZE

    def put(self, key, value):
        """ Додає пару (ключ, значення) до таблиці
        :param key: ключ
        :param value: значення
        """
        hash_key = HashTable.hash(key)  # хеш ключа
        slot = self.slots[hash_key]     # поточний слот таблиці
        while slot != None:
            if slot.key == key:      # якщо ключ у таблиці знайдений
                slot.value = value   # змінюємо значення
                slot.valid = True
                return  # припиняємо роботу методу
            slot = slot.next  # переходмо до наступного елементу у ланцюжку

        # якщо ключ у таблиці не знайдений, додаємо новий вузол
        # з ключем та значенням у початок ланцюжка
        slot = Node(key, value)
```

[2] Детально зв'язні списки викладені у розділі 5 цього підручника.

```
        slot.next = self.slots[hash_key]
        self.slots[hash_key] = slot

    def get(self, key):
        """ Повертає значення за ключем
        :param key: ключ
        :return: значення """
        hash_key = HashTable.hash(key)   # хеш ключа
        slot = self.slots[hash_key]      # поточний слот таблиці
        while slot != None:
            if slot.key == key:    # якщо ключ у таблиці знайдений
                return slot.value  # повертаємо значення, що відповідає ключу
            slot = slot.next       # переходмо до наступного елементу у ланцюжку

        # якщо ключ у таблиці не знайдений
        return None
```

Розуміння принципів роботи методів **get**() та **put**() не повинно викликати у читача труднощів. Дійсно, у цих операціях першим кроком обчислюється хеш-значення ключа, а далі відбувається прохід по ланцюжку доти доки не буде знайдений вузол з відповідним ключем або не буде досягнуто кінця списку. Ситуація в останньому випадку буде означати, що ключ не міститься у таблиці. Розглянемо тепер детальніше операцію вставки нового ключа 33 на прикладі таблиці наведеної на рисунку 3.1.14.

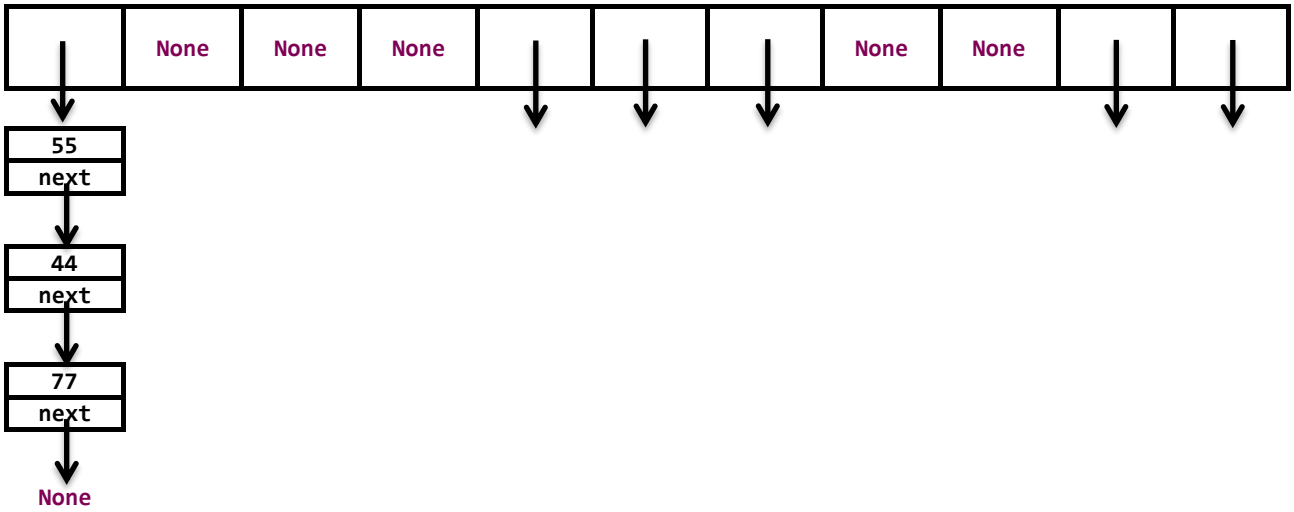


Рисунок 3.1.14. Додавання елементів до хеш-таблиці

Обчисливши хеш для числа 33 (як остачу від ділення ключа на 11), отримаємо, що цей ключ має потрапити у список, на який посилається перший слот таблиці. Пройшовши по списку першого слота переконаємося у тому, що цей ключ не міститься у таблиці. Тоді створюємо новий вузол з ключем 33, поле next якого посилається на найперший елемент списку.

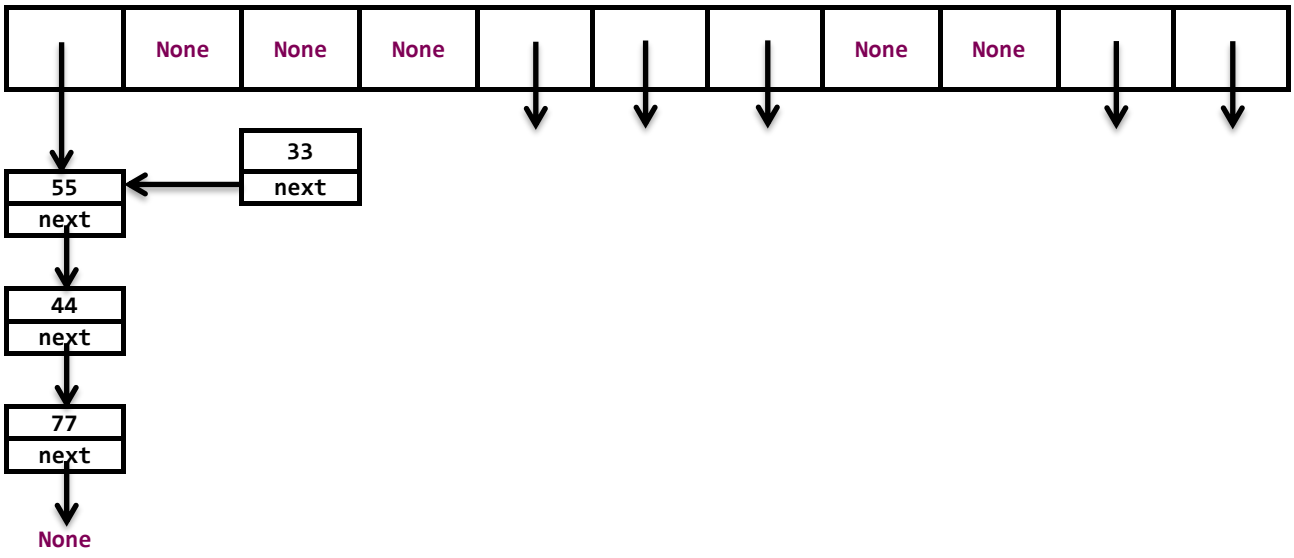


Рисунок 3.1.15. Додавання елементів до хеш-таблиці

Далі лишається лише переставити посилання слота на щойно створений вузол.

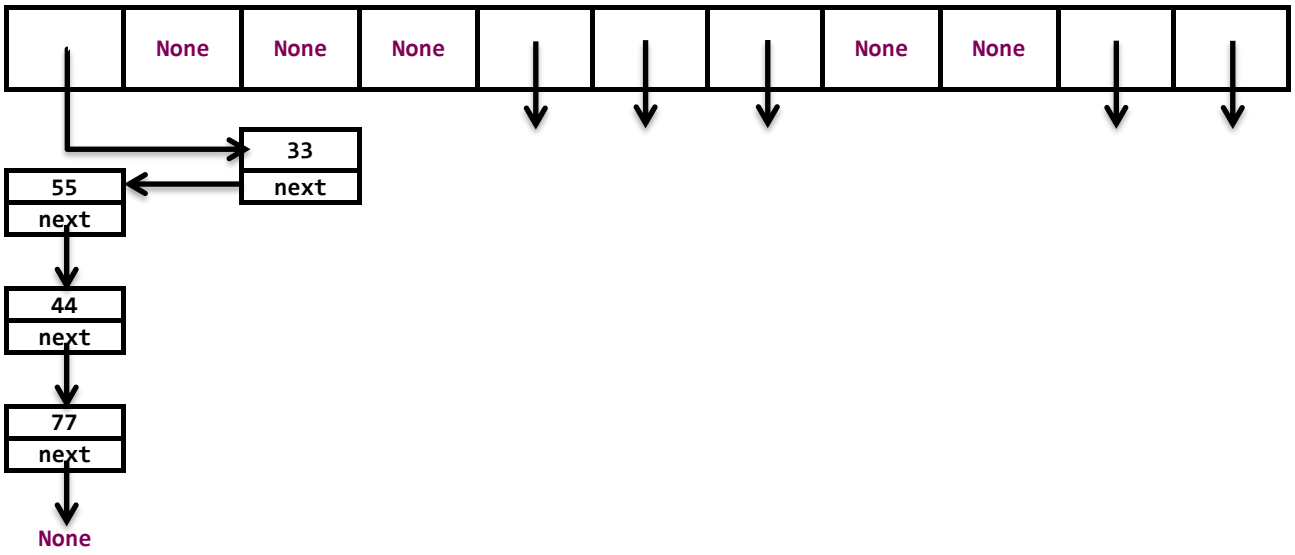


Рисунок 3.1.16. Додавання елементів до хеш-таблиці

Як бачимо при такій реалізації методу `put()` нові ключі вставляються у початок відповідного списку. Такий спосіб вставки нових ключів дозволяє отримати просту для розуміння та компактну для запису реалізацію.

Перевантаження спеціальних методів та використання описаного класу повністю повторює відповідну реалізацію для хеш-таблиці з відкритою адресацією, наведену у лістингу 3.1.9.

§3.2. Сортування

Задачі сортування мають надзвичайно важливе значення з практичної точки зору. Вони часто є першим кроком підготовки даних для їхнього подальшого аналізу. Наприклад, ми вже знаємо, що алгоритми бінарного пошуку, вимагають впорядкованості даних.

Існує безліч класичних алгоритмів сортування. Деякі з них не надто швидкі, проте мають не великі вимоги по пам'яті, інші навпаки – дуже швидкі проте потребують значних ресурсів пам'яті. У цьому параграфі розглянемо кілька основних класичних алгоритмів сортування, що використовуються для сортування послідовностей чисел.

3.2.1. Бульбашкове сортування

Бульбашкове сортування (також, використовується термін *сортування обміном*, *англ. Bubble sort*) є найпростішим, з алгоритмічної точки зору, алгоритмів сортування. Його ідея полягає у тому, що здійснюється кілька проходів по списку, під час кожного з яких порівнюють пари сусідніх елементів. Якщо елементи стоять не правильно, вони міняються місцями. Кожен прохід по списку ставить наступне найбільше значення на його правильну позицію. Розглянемо кілька проходів бульбашкового сортування для списку елементів наведеного нижче.

56	12	66	20	33	95	32	13	10

Рисунок 3.2.1. Не відсортований список елементів

Будемо йти зліва направо. Затіняти, клітини, які розглядаються. Більший з двох елементів будемо позначати напівжирним шрифтом.

12	**56**	66	20	33	95	32	13	10

12	56	**66**	20	33	95	32	13	10

12	56	20	**66**	33	95	32	13	10

12	56	20	33	**66**	95	32	13	10
12	56	20	33	66	**95**	32	13	10
12	56	20	33	66	32	**95**	13	10
12	56	20	33	66	32	13	**95**	10
12	56	20	33	66	32	13	10	**95**
12	56	20	33	66	32	13	10	**95**

Рисунок 3.2.2. Перший прохід по списку.
Найбільший елемет списку займає потрібну позицію.

Як бачимо у результаті першого проходу по списку, найбільший елемент 95 опинився на останній позиції – це його позиція у відсортованому списку і у подальшому вона змінювати вже не буде. Наступний прохід бульбашкового сортування здійснюється для всіх елементів списку крім елемента 95, що стоїть вже на своїй правильній позиції.

12	20	33	56	32	13	10	**66**	95

Рисунок 3.2.3. Два найбільших елементи списку займають свої позиції.

В результаті другого проходу по списку, елемент 66 опиниться на останньому місці серед елементів, що розглядалися на цьому проході. Далі процедура проходу здійснюється аналогічно вищенаведеному, для елементів списку без елементів 66 і 95, що вже займають свої правильні позиції.

12	20	33	32	13	10	56	**66**	95

Рисунок 3.2.4. Третій прохід бульбашкового сортування.

Фактично, в результаті кожного проходу, найбільший елемент, серед елементів по яких здійснюється прохід, опинятиметься (подібно до бульбашки, що підіймається на поверхню рідини) на останній позиції цього підсписку.

Отже, як бачимо, якщо у списку n елементів, то за перший прохід здійснюється $n-1$ операція порівняння. На другій ітерації проводимо вищеописану процедуру для $n-1$ елементів списку (крім останнього, бо він уже знаходиться на своїй позиції). Наступна ітерація проводиться для $n-2$ елементів списку. І так далі, доки не дійдемо до необхідності пройти останні два елементи, що і завершує процедуру сортування.

Реалізація алгоритму

Ідея алгоритму є досить простою, тому наведемо його без додаткових пояснень, крім тих, що виписані як коментарі у вихідному коді алгоритму.

Лістинг 3.2.1. Реалізація алгоритму сортування «Бульбашкою».

```
def bubble_sort(array):
    """ Реалізує алгоритм сортування "Бульбашкою"

    :param array: Масив (список однотипових елементів)
    :return: None
    """
```

```
    n = len(array)
    for pass_num in range(n - 1, 0, -1):
        for i in range(pass_num):
            # Якщо наступний елемент менший за попередній
            if array[i] > array[i + 1]:
                # Міняємо місцями елементи, тобто
                # виштовхуємо більший елемент нагору
                array[i], array[i + 1] = array[i + 1], array[i]
```

Аналіз алгоритму

При аналізі бульбашкового сортування необхідно звернути увагу, що незалежно від початкового порядку елементів, для списку з n елементів буде здійснено $n-1$ прохід. При цьому, на кожному проході буде здійснено $n-j$ операцій порівняння, де j – це номер проходу. Таким чином, очевидно, що складність алгоритму бульбашкового сортування буде $O(n^2)$.

Бульбашкове сортування часто розглядається як найбільш неефективний сортувальний метод, оскільки він повинен переставляти елементи поки не стане відома їх остаточна позиція. Ці "порожні" операції обміну вельми затратні. Однак, оскільки бульбашкове сортування робить прохід по всій несортованій частині списку, воно вміє те, що не можуть більшість сортувальних алгоритмів. Зокрема, якщо під час проходу не було зроблено жодної перестановки, то ми знаємо, що список вже відсортований. Таким чином, алгоритм може бути модифікований, щоб зупинятися раніше, якщо виявляється, що завдання виконане.

3.2.2. Сортування вибором

Алгоритм сортування вибором дуже подібний до бульбашкового сортування, проте є дещо оптимальнішим, оскільки за кожен прохід по списку відбувається лише одна операція перестановки елементів. Його ідея полягає у тому, що на кожному кроці відбувається (лінійний) пошук найбільшого елементу серед невідсортованої частини списку, який переставляється на відповідну позицію (крайню праву/ліву позицію невідсортованої частини списку).

Розглянемо роботу алгоритму на вищенаведеному списку елементів (див Рисунок 3.2.1). Найбільший елемент, як і раніше, будемо виділяти напівжирним шрифтом, а елементи, що міняються місцями – підсвічуваннями. Отже, в результаті першого проходу по списку, знайдено найбільший елемент 95

56	12	66	20	33	**95**	32	13	10

Рисунок 3.2.5. Знайдено найбільший елемент списку 95.

який, міняється місцями з останнім елементом у списку 10:

56	12	66	20	33	10	32	13	**95**

Рисунок 3.2.6. Елементи 10 та 95 міняються місцями.

Як і у випадку бульбашкового сортування, після першого проходу списку, найбільший елемент списку 95 зайняв свою правильну позицію. На другому проході процедура проводиться для елементів списку без елементу 95, результатом чого елементи 66 та 13 поміняються місцями

56	12	13	20	33	10	32	**66**	95

Рисунок 3.2.7. Елементи 66, 95 зайняли свої позиції.

Отже, як і у випадку бульбашкового сортування, після першого проходу списку, найбільший елемент займає потрібне місце. Після другого проходу – на своє місце стає наступний найбільший елемент. Процес продовжується доки всі елементи не займуть потрібні місця. Для цього потрібно здійснити $n-1$ прохід алгоритму.

Реалізація алгоритму

Лістинг 3.2.2. Реалізація алгоритму сортування вибором.

```
def selection_sort(array):
    """ Реалізує алгоритм сортування вибором
    :param array: Масив (список однотипових елементів)
    :return: None
    """
    n = len(array)
    for i in range(n - 1, 0, -1):
        # реалізуємо пошук найбільшого елементу
        maxpos = 0
        for j in range(1, i + 1):
            if array[maxpos] < array[j]:
                maxpos = j
        # Міняємо місцями поточний і найбільший елемент
        array[i], array[maxpos] = array[maxpos], array[i]
```

Аналіз алгоритму

Як в алгоритмі бульбашкового сортування, при сортування вибором для списку з n здійснюється $n-1$ прохід алгоритму по списку, на кожному з яких буде здійснено $n-j$ операцій порівняння, де j – це номер проходу. А отже, хоча за кількістю операцій майже напевно, цей алгоритм оптимальніший, за бульбашкове сортування, проте його асимптотична складність така ж, як у бульбашкового сортування – $O(n^2)$.

3.2.3. Сортування вставкою

Сортування вставкою підтримує відсортованою частина елементів. Кожен же наступний елемент вставляється у потрібну позицію, так щоб підсписок лишався відсортованим. Спочатку вважаємо, що підсписок з одного елементу (що знаходиться на нульовій позиції) відсортований. Далі, кожен наступний елемент, на кожному проході вставляється у відповідну позицію. При цьому, може виникнути ситуація, коли для вставки необхідно зсунути частину елементів списку.

Розглянемо ідею алгоритму на прикладі сортування списку зображеного вище на Рисунку 3.2.1. Починаємо з підсписку, що містить перший елемент списку, який очевидно є відсортованим. Будемо підсвічувати відсортовану частину списку, а елемент, що вставляється напівжирним шрифтом.

56	12	66	20	33	95	32	13	10
12	56	66	20	33	95	32	13	10
12	56	**66**	20	33	95	32	13	10
12	**20**	56	66	33	95	32	13	10
12	20	**33**	56	66	95	32	13	10
12	20	33	56	66	**95**	32	13	10
12	20	**32**	33	56	66	95	13	10

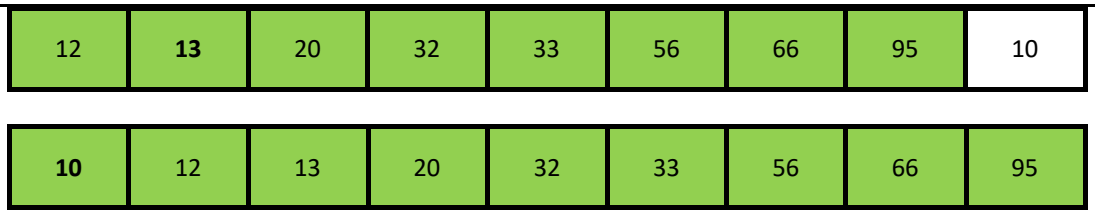
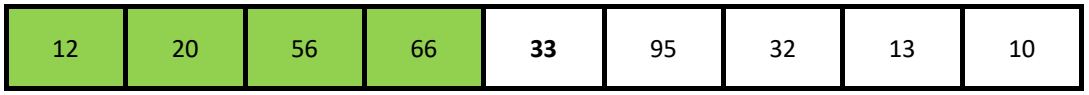


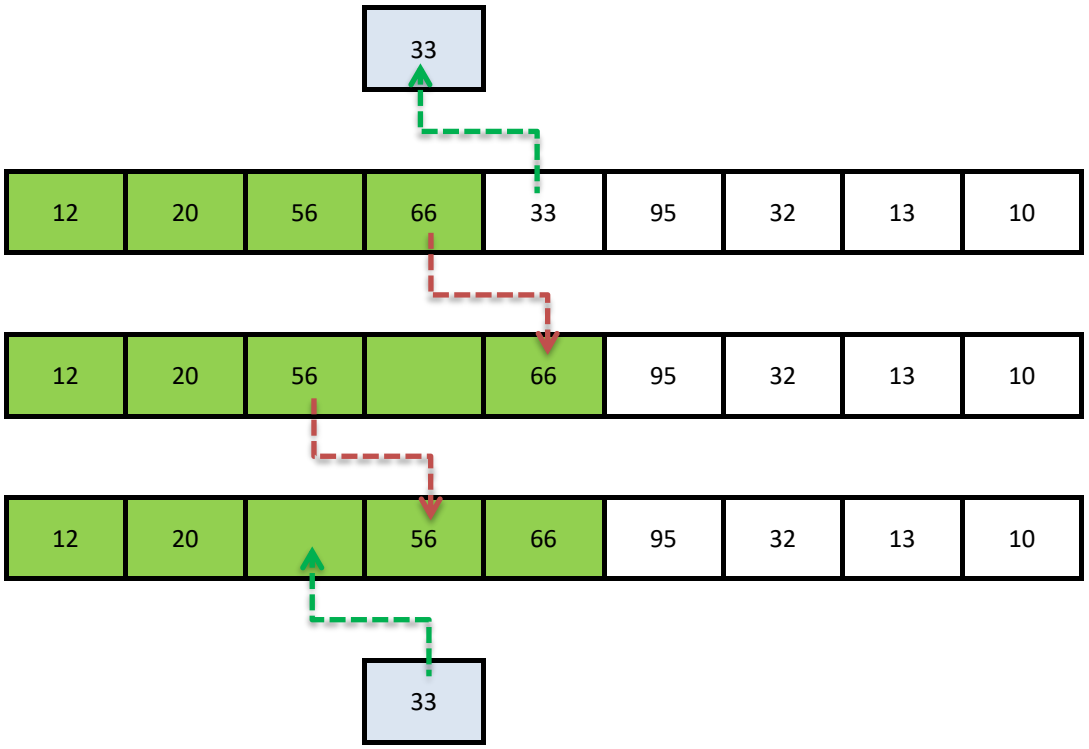
Рисунок 3.2.8. Сортування вставкою

Реалізація алгоритму

Перш ніж навести повну реалізацію алгоритму, розглянемо його фрагмент, що стосується вставки елементу на потрібну позицію. Наприклад, розглянемо 4-й прохід алгоритму по наведеному вище списку, а саме стан списку наведений нижче, під час вставки елементи 33.



Щоб підсписок лишався відсортованим, елемент 33 потрібно вставити на позицію між елементами 20 і 56. Іншими словами, потрібно зсунути вправо елементи 56 і 66, а на місце, що звільнилося вставити елемент 33. Пошук місця вставки і зсув елементів списку будемо здійснювати одночасно. Для цього запам'ятаємо елемент 33. Тоді на його місце (за необхідності) можемо переставити елемент, що розташовується ліворуч.



Отже, напишемо реалізацію алгоритму

Лістинг 3.2.3. Сортування вставкою.

```
def insertion_sort(array):
    """ Реалізує алгоритм сортування вставкою
    :param array: Масив (список однотипових елементів)
    :return: None
    """
    n = len(array)
    for index in range(1, n):
        currentValue = array[index]  # запам'ятовуємо елемент, що необхідно вставити
        position = index             # та його позицію
        # пошук позиції для вставки поточного елемента
        while position > 0:
```

```
            if array[position - 1] > currentValue:
                # зсув елементу масиву вправо
                array[position] = array[position - 1]
            else:
                # знайдено позицію
                break
            position -= 1

        # Вставка поточного елемента у знайдену позицію
        array[position] = currentValue
```

Аналіз алгоритму

Як і у випадках бульбашкового сортування і сортування вибором, алгоритм здійснює $n - 1$ прохід для списку з n елементів. Якщо підсписок у який вставляється елемент складається з i елементів, то у найгіршому випадку для вставки нового елементу потрібно i операцій зсуву.

Отже, як і для розглянутих раніше алгоритмів асимптотична складність алгоритму буде $O(n^2)$. Проте, у найкращому випадку, а це буде якщо список вже відсортований, алгоритм буде виконуватися за $O(n)$.

Зауважимо, що взагалі кажучи, операція зсуву вимагає близько третини від обчислювальної роботи обміну, оскільки здійснюється лише одна операція присвоєння. На практиці сортування вставками має дуже хорошу абсолютну швидкодію у порівнянні з бульбашковим сортуванням або сортуванням вибором.

3.2.4. Сортування злиттям

Попередні алгоритми сортування виконувалися у загальному випадку за квадратичний час $O(n^2)$. Розглянемо алгоритми, що базуються на стратегії «розділяй та владарюй», яка дозволяє значно збільшити швидкодію алгоритмів.

Сортування злиттям – це рекурсивний алгоритм, який можна схематично записати так:

1. Якщо список порожній або складається з одного елементу, то він вважається відсортований.
2. Якщо список складається більше ніж з двох елементів, то він розділяється навпіл, після чого для кожної з половин рекурсивно викликається сортування злиттям. Далі два відсортовані списки об'єднуються (зливаються) у один так, щоб утворений список був відсортованим.

Рисунок нижче демонструє операцію розбиття списку навпіл

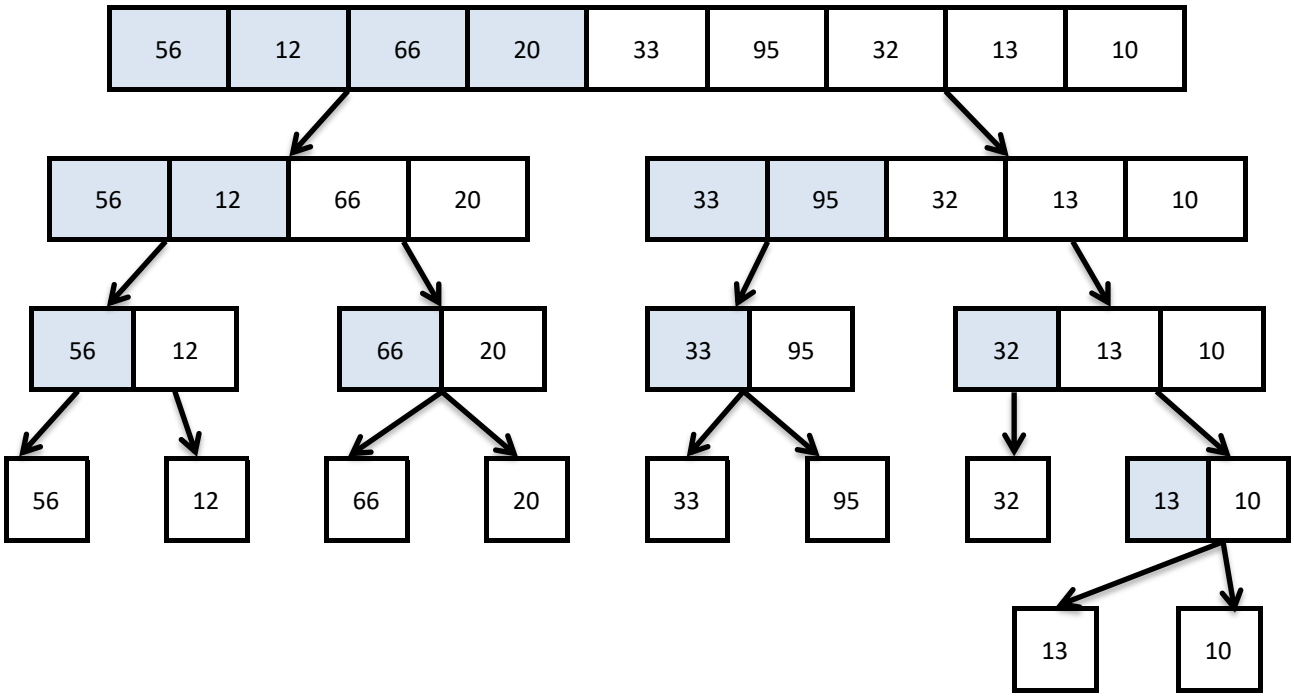


Рисунок 3.2.9. Розбиття списку навпіл

Наступний рисунок демонструє процес злиття вже відсортованих списків.

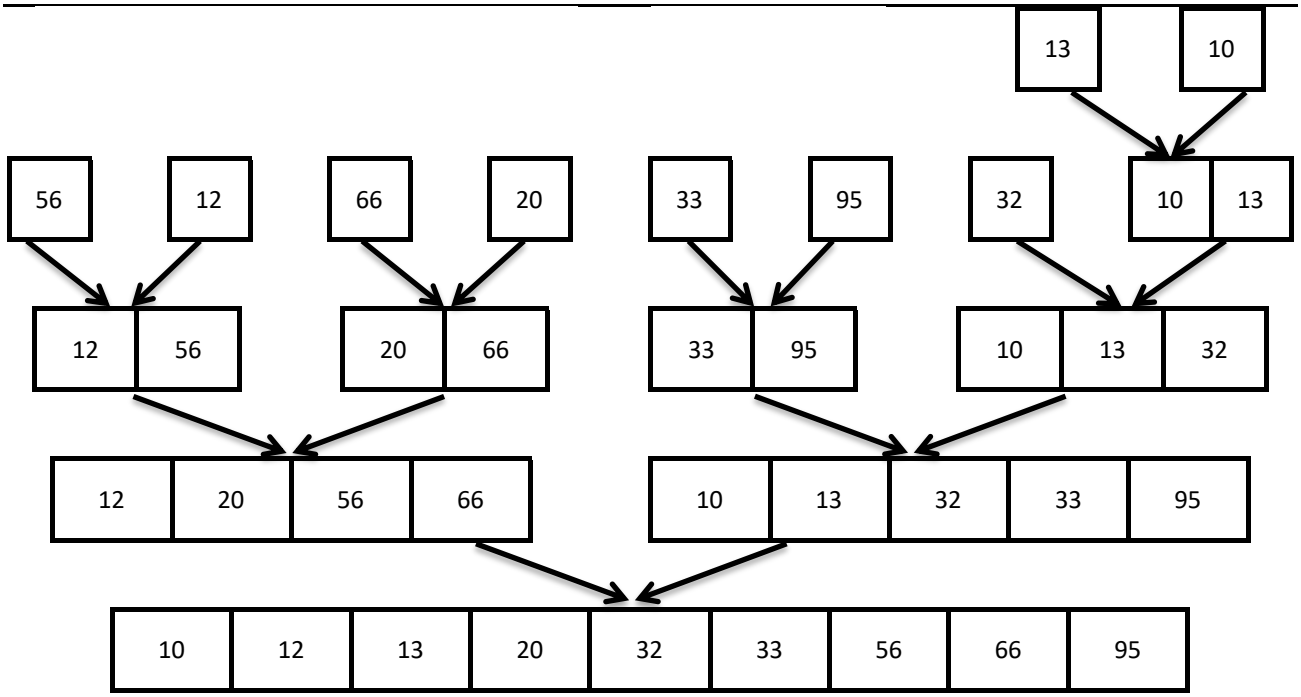


Рисунок 3.2.10. Злиття списків

Реалізація алгоритму

Лістинг 3.2.4. Реалізація алгоритму сортування злиттям

```
def merge_sort(array):
    """ Реалізує алгоритм сортування злиттям

    :param array: Масив (список однотипових елементів)
    :return: None
    """
    print("Splitting ", array)
    if len(array) > 1:
        # Розбиття списку навпіл
        mid = len(array) // 2
        lefthalf = array[:mid]
        righthalf = array[mid:]

        # Рекурсивний виклик сортування
        # для кожної з половин
        merge_sort(lefthalf)
        merge_sort(righthalf)

        # Злиття двох відсортованих списків
        i = 0
        j = 0
        k = 0
        while i < len(lefthalf) and j < len(righthalf):
            if lefthalf[i] < righthalf[j]:
                array[k] = lefthalf[i]
                i += 1
            else:
                array[k] = righthalf[j]
                j += 1
            k += 1

        while i < len(lefthalf):
            array[k] = lefthalf[i]
            i += 1
            k += 1

        while j < len(righthalf):
            array[k] = righthalf[j]
```

```
            j += 1
            k += 1
```

Аналіз алгоритму

На кожній ітерації рекурсивного алгоритму ми ділимо список навпіл. Це означає, що для списку з n елементів буде $\log n$ рекурсивних викликів. Крім цього на кожній ітерації буде здійснюватися операція об'єднання двох списків, асимптотична складність якої є $O(n)$. Отже, можемо зробити висновок, що складність алгоритму сортування злиттям є $O(n \log n)$.

Недоліком алгоритму сортування злиттям є потреба у використанні додаткової пам'яті, яка виникає під час ділення списку навпіл, тобто під час створення двох половинок списку, які використовуються для рекурсивного виклику алгоритму сортування.

3.2.5. Швидке сортування

Алгоритм швидкого сортування, використовується для того, щоб отримати ті ж переваги по швидкості, що і сортування злиттям, не використовуючи при цьому додатку пам'ять. Крім цього, однією з особливостей алгоритму є те, що він використовує значно меншу кількість порівнянь і перестановок елементів, ніж інші алгоритми.

Як і алгоритм сортування злиттям, швидке сортування виконується за час $O(n \log n)$. Проте цей час виконання досягається в середньому, а у окремих випадках, алгоритм швидкого сортування може давати навіть гірші результати за бульбашковий алгоритм або алгоритм вставкою.

Розглянемо ідею алгоритму швидкого сортування на прикладі сортування списку зображеного на Рисунку 3.2.11. На кожній ітерації алгоритму обирається еталонний елемент списку, що називається опорним елементом (який як правило вибирається як центральний елемент списку).

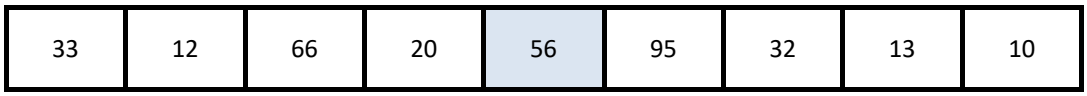


Рисунок 3.2.11. Обираємо елемент 56 опорним елементом на першій ітерації алгоритму

Далі вихідний список поділяється на дві частини (проте без створення нового списку), одна з яких місить всі елементи які менші за опорний, а інша – елементи що більші. Це досягається з допомогою перестановок елементів, так, щоб зліва від опорного елементу знаходилися елементи списку менші за нього, а справа – більші. Для цього виберемо два маркери – лівий і правий, що спочатку будуть вказувати на крайній лівий та крайній правий елементи списку відповідно.

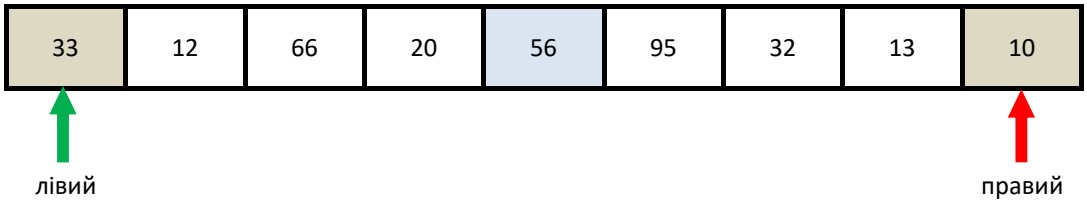


Рисунок 3.2.12. Визначаємо лівий та правий маркери

Спочатку лівий маркер рухається вправо, доки не зустріне елемент, що є більший або рівний за опорний. У нашому випадку це буде елемент 66.

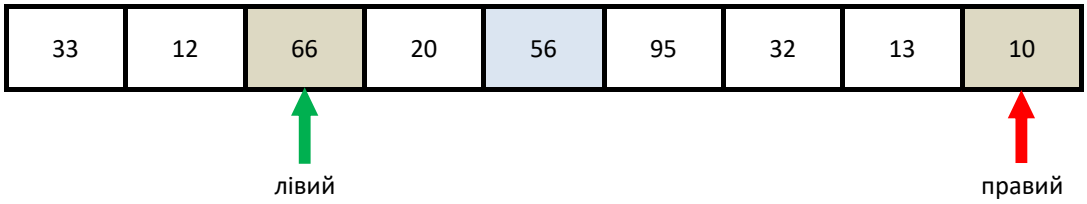


Рисунок 3.2.13. Лівий досягає вказує на елемент 66.

Після чого правий маркер починає рух вліво доки не зустріне число, що є меншим або рівний за опорний елемент. У нашому випадку з самого початку правий маркер вказує на такий елемент. Після того, як обидва маркери зупиняться – елементи на які вони вказують міняються місцями.

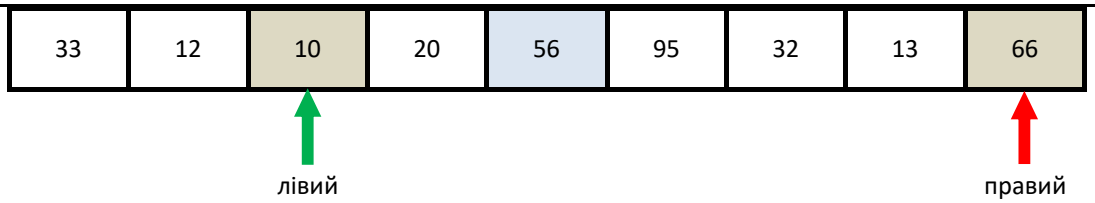


Рисунок 3.2.14. Лівий досягає вказує на елемент 66.

Далі, лівий маркер починає знову рух вправо, поки не знайде елемент більший або рівний за опорний, а правий – вліво поки не знайде елемент менший або рівний за опорний.

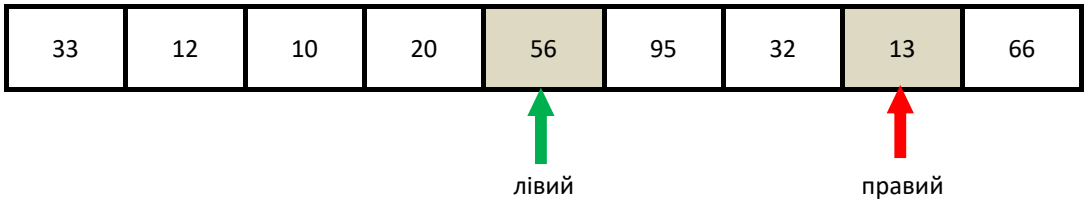


Рисунок 3.2.15. Лівий маркер рухається вправо, а правий вліво.

міняємо їх місцями і знову продовжуємо процедуру змістивши лівий маркер на одну позицію вправо, а правий – на одну позицію вліво.

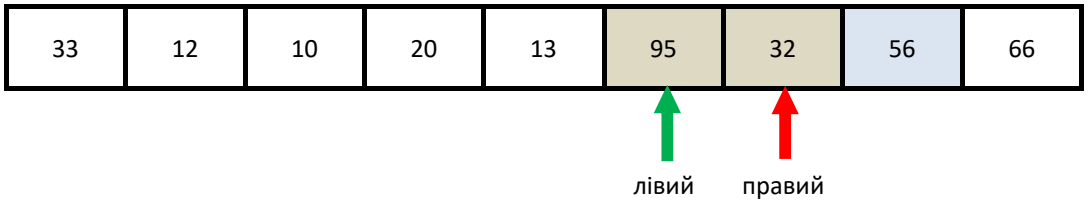


Рисунок 3.2.16. Міняємо місцями елементи на які вказують маркери.

Рух маркерів, разом з описаною вище процедурою, відбувається доти, доки лівий і правий маркери не почнуть вказувати на один і той же елемент або лівий маркер не займе позицію праворуч від правого.

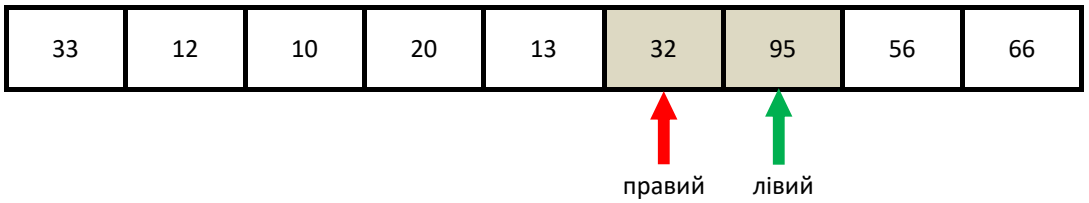


Рисунок 3.2.17. Лівий маркер займає позицію правіше правого маркера.

Якщо уважно подивитися на отриманий в результаті такої процедури список, то можна помітити, що зліва від правого маркера опинилися всі елементи, що менші або рівні за опорний елемента, у той час, як справа від нього – відповідно більші або рівні опорному. Після цього, зазначена операція рекурсивно здійснюється для лівого (від початкового елемента списку до елемента на який вказує правий маркер) і правого (від наступного елемента, що йде за правим маркером до кінця списку) підсписків.

Реалізація алгоритму

Лістинг 3.2.5. Швидке сортування

```
def qsort(array, a, b):
    """ Швидке сортування масиву
        :param array: масив
    :param a: ліва межа сортування
    :param b: права межа сортування
    """

    if a >= b:
        return
```

```
    pivot = array[a + (b - a) // 2]  # опорний елемент
    left = a                         # лівий маркер
    right = b                        # правий маркер

    while True:
        while array[left] < pivot:   # Рухаємося зліва на право, поки не знайдемо
            left += 1                # елемент, що більший або рівний за опорний

        while pivot < array[right]:  # Рухаємося справа на ліво, поки не знайдемо
            right -= 1               # елемент,що менший або рівний за опорний

        if left >= right:  # маркери вказують на один і той де елемент або
            break          # лівий маркер займає позицію праворуч від правого

        array[left], array[right] = array[right], array[left]  # міняємо місцями елементи
        left += 1          #  зміщуємо лівий маркер вправо
        right -= 1         #  зміщуємо правий маркер вліво

    #  рекурсивно повторюємо процедуру для лівого та правого підсписків
    qsort(array, a, right)
    qsort(array, right + 1, b)
```

Аналіз алгоритму

Якщо список складається з n елементів, і якщо точка поділу списку на кожній ітерації рекурсивної функції є приблизно в точкою середини списку, то, очевидно, що складність алгоритму буде близькою до $O(n \log n)$. При цьому алгоритм практично не вимагає додаткової пам'яті. Проте, на жаль, у найгіршому випадку, точка розбиття може бути не посередині, а стрибати зліва на право, роблячи розбиття дуже нерівномірним. У цьому випадку, сортування списку з n елементів розділиться на сортування списків розміром 0 та $n-1$ елемент. Далі 0 та $n-2$ елементи і так далі. Як наслідок, у такому випадку сортування буде здійснюватися за $O(n^2)$. Тому до вибору точки розбиття потрібно підходити системно, аналізуючи дані, що містяться у масиві. Наприклад, можна провести попередній аналіз масиву, що матиме лінійну складність, та знайти його медіану (середній елемент масиву – не плутати з середнім арифметичним). Якщо обирати такий елемент у якості опорного, на кожному виклику рекурсивного методу, то це гарантуватиме, що складність алгоритму буде близькою до $O(n \log n)$.

РОЗДІЛ 4. ПОВНИЙ ПЕРЕБІР

§4.1. Повний перебір

При розв'язанні алгоритмічних задач, найпростішим з концептуальної точки зору (проте не з точки зору реалізації), є метод повного перебору, коли серед множини всіх можливих розв'язків потрібно обрати правильний. Проте, коли мова заходить про оптимальність такого розв'язання, частіше за все побутує думка, що повний перебір є найгіршим з можливих підходів розв'язання. Так, частково це правда – найчастіше повний перебір застосовують у тих випадках, коли або іншого варіанту не лишається і множина всіх можливих розв'язків є не великою, або можна заздалегідь відкинути значну частину хибних розв'язків. Проте, існує широкий спектр алгоритмічних задач, метод повного перебору для яких є досить оптимальним. Раніше ми познайомилися з алгоритмом сортування злиттям, котрий концептуально є нічим іншим, як методом повного перебору і при цьому він є одним із найоптимальніших серед усіх наведених у цьому посібнику алгоритмів сортування масивів. Отже, складність повного перебору залежить від кількості всіх можливих розв'язків задачі, від порядку розгляду цих розв'язків, а також чи можливо відкинути значну частину заздалегідь неправильних варіантів.

Отже,

Означення 4.1.1. **Повний перебір** (англ. brute force, метод «грубої сили») – загальний метод розв'язання математичних задач шляхом перебору усіх можливих потенційних варіантів.

Фактично, повним перебором можна скористатися лише тоді, коли ми маємо справу з дискретною системою, стани якої можуть бути легко проаналізовані.

Загальних методів побудови алгоритмів, що використовують механізм повного перебору, взагалі кажучи, не існує – у кожному конкретному випадку може бути застосований той чи інший підхід. Проте, очевидно, всі вони мають спільну рису – потрібно організувати певним чином послідовний перебір всіх можливих станів системи, не пропустивши жодного. Часто алгоритм такого перебору станів системи виникає сам собою після детального аналізу дискретної системи, що розглядається у задачі.

Приклад 4.1.1 Наприклад, якщо потрібно побудувати всі можливі n-значні числа, що складаються з цифр 0 та 1, то, згадавши про двійкове зображення натуральних чисел, можна застосувати такий алгоритм

Лістинг 4.1.1. Пошук n-значних чисел, що складаються з 0 та 1.

```
n = int(input())  # кількість цифр

l = 1 << n - 1    # найменше n-значне число з нулів та одиниць
r = 1 << n        # найменше (n+1)-значне чило з нулів та одиниць

# послідовність чисел з проміжку [l, r) у двійковій системі числення буде
# будувати всі можливі комбінації n-значних чисел, що складаються з 0 та 1.
for i in range(l, r):
    print(bin(i))
```

Результатом роботи наведеної програми для числа введеного числа 3 буде

```
3
0b100
0b101
0b110
0b111
```

У цьому прикладі, для повного перебору досить було застосувати послідовний перебір натуральних чисел з заданого проміжку. Якщо ж задача полягає у тому, що необхідно побудувати всі можливі комбінації деякої послідовності з повторами елементів, при цьому кількість елементів наперед відома і фіксована, то найпростішим способом організації такого перебору буде використання вкладених циклів, у якому кожен цикл буде перебирати послідовно усі елементи набору:

```
# SET_ELEMENTS - деякий набір елементів, наприклад список [1, 2, 3]
for el_1 in SET_ELEMENTS:
```

```
    for el_2 in SET_ELEMENTS:
        ...
        for el_n in SET_ELEMENTS:
            ...
```

Приклад 4.1.2 Напишемо програму, що визначає кількості тризначних натуральних чисел, сума цифр яких дорівнює n ($n \geq 1$).

Звичайно, можна як і у попередньому прикладі, пробігтися одним циклом по всіх натуральних числах від 100 до 999, розбиваючи на кожному кроці число на сотні, десятки і одиниці. Проте, таке розбиття буде використовувати додаткові операції цілочисельного ділення та остачі від ділення. Тому підемо іншим шляхом, а саме організуємо перебір використовуючи три вкладених цикли – відповідно для перебору сотень, десятків та одиниць.

Лістинг 4.1.2. Пошук всіх тризначних чисел, сума цифр яких дорівнює заданому числу.

```
n = int(input())
counter = 0                     # лічильник кількості чисел
for i in range(1, 10):          # Перебір сотень (від 1, бо інакше число не тризначне)
    for j in range(10):         # Перебір десятків
        for k in range(10):     # Перебір одиниць
            if i + j + k == n:
                counter += 1

print(counter)
```

Хоча цикли часто дають чи не найкращий підхід до повного перебору для широкого класу задач, проте загалом на практиці, найчастіше повний перебір реалізовують використовуючи рекурсивний опис функції. Однією з задач, що розв'язується методом повного перебору є задача відшукання всіх можливих перестановок елементів заданої послідовності. Розв'яжемо її у такій інтерпретації:

Приклад 4.1.3 За заданим натуральним числом n виведемо усі перестановки з цілих чисел від 1 до n.

Для того, щоб краще зрозуміти запропонований нижче алгоритм, випишемо всі можливі комбінації для $n = 1,2,3$. Розглянемо найпростіший випадок $n = 1$. Тобто нам потрібно побудувати всі можливі перестановки елементів послідовності $[1]$. Очевидно, існує лише один варіант такої перестановки – це власне цей один елемент і утворює цю перестановку.

Тепер розглянемо $n = 2$. Як ми знаємо з комбінаторики, таких комбінацій є $2! = 2$. Скористаємося цим фактом, щоб переконатися, що виписано всі комбінації. Очевидно, що всі такі комбінації записуються таким чином

$$1\ 2$$
$$2\ 1$$

тобто дописуванням елементу 2 зліва та справа від числа 1.

Далі, випишемо всі можливі комбінації для числа 3, тобто для набору $[1, 2, 3]$. Кількість різних комбінацій вже буде $3! = 6$.

$$1\ 2\ 3$$
$$1\ 3\ 2$$
$$2\ 1\ 3$$
$$2\ 3\ 1$$
$$3\ 1\ 2$$
$$3\ 2\ 1$$

Очевидно, що ці послідовності було отримано вставкою цифри 3 у всі можливі позиції для послідовностей записаних для попереднього випадку.

Досить часто, при продумуванні алгоритму повного перебору, виписані варіанти для найпростіших випадків наштовхують на ідею алгоритму. Так, якщо уважно придивитися, то, використовуючи отриманий результат, можна продовжити для $n = 4$, вставкою цифри у всі можливі позиції щойно отриманого набору послідовностей. Отже

Лістинг 4.1.3. Пошук всіх перестановок заданої послідовності

```
def sequences(lst : list, k, n):
    """
    :param lst: підсписок перестановок
```

```
    :param k:   елемент для вставки
    :param n:   найбільший елемент послідовності
    """
    if k > n:  # Якщо всі елементи вже вичерпано
        print(*lst)
        return

    # Вставляємо елемент k у всі можливі позиції списку
    # отриманого на попередніх ітераціях
    for pos in range(k):
        lst_next = lst[:]             # Копіюємо список
        lst_next.insert(pos, k)       # Вставляємо елемент
        sequences(lst_next, k + 1, n) # Запускаємо рекурсивно додавання наступного числа.

# Головна програма
if __name__ == "__main__":
    n = int(input())
    lst = []
    sequences(lst, 1, n)
```

Результатом наведеної програми для введеного числа 3 буде

```
3 2 1
2 3 1
2 1 3
3 1 2
1 3 2
1 2 3
```

Приклад 4.1.4 [e-olimp, 2169]. За заданим натуральним числом n виведемо усі перестановки з цілих чисел від 1 до n у лексикографічному порядку.

Програма наведена у попередньому прикладі побудувала всі варіанти. Спробуємо дещо змінити її для того, щоб варіанти будувалися у лексикографічному порядку. Звичайно ніщо нам не заважає відсортувати отриманий список послідовностей, Проте, щоб уникнути зайвих витрат процесорного часу, перепишемо програму, застосувавши інший підхід. А саме: будемо не вставляти новий елемент на всі можливі позиції, а додавати елемент в кінець списку, якщо він ще не міститься у ньому.

Лістинг 4.1.4. Пошук всіх перестановок заданої послідовності. Видення у лексикографічному порядку.

```
def sequences(lst : list, n):
    """
    :param lst: підсписок перестановок
    :param n:   найбільший елемент послідовності
    """

    k = len(lst)  # Визначаємо кількість членів поточної підпослідовності

    if k == n:    # Якщо всі елементи вже вичерпано
        print(*lst)
        return

    for i in range(1, n + 1):
        if i not in lst:            # Якщо елемент i не міститься у підпослідовності
            lst_next = lst[:]       # Копіюємо список
            lst_next.append(i)      # Додаємо до нього елемент i
            sequences(lst_next, n)  # Додавання рекурсивно інших членів послідовності.

# Головна програма
if __name__ == "__main__":

    n = int(input())
    lst = []
    sequences(lst, n)
```

Однією з класичних задач, що розв'язуються повним перебором, є задача про рюкзак. Розв'яжемо її в такій постановці.

Приклад 4.1.5 [e-olimp, 2103]. Вася зібрався у похід з друзями-програмістами і вирішив відповідально підійти до вибору того, що він візьме з собою. У Васі є n речей, які він міг би взяти з собою у рюкзаку. Кожна річ важить 1 кілограм. Речі мають різну "користність" для Васі. Похід очікується досить тривалий, і Вася хотів би носити рюкзак вагою не більше w кілограм. Допоможіть йому визначить максимальну сумарну "корисність" предметів у нього в рюкзаку при вазі рюкзака не більше w кілограм.

Оскільки системи станів задачі є дискретною, її можна розв'язавши, повністю перебравши всі можливі розв'язки. Отже, у нас є n речей, які можна укладати в рюкзак. Для кожного предмета існує 2 варіанти: предмет або кладеться в рюкзак, або ні. Тоді, як ми знаємо, алгоритм, що використовує повний перебір всіх можливих варіантів має складність $O\ (2^n)$. Це дозволяє його використовувати лише для невеликої кількості предметів. З ростом кількості предметів задача стає нерозв'язною даним методом за прийнятний час.

Тоді, рекурсивна функція побудови всіх варіантів та підрахунку поточної вартості рюкзака матиме вигляд

Лістинг 4.1.5.Задача про рюкзак.

```
def max_score(weight, score, num):
    """
    :param weight: поточна вага рюкзака
    :param score:  поточна вартість рюкзака
    :param num:    номер предмета
    """
    global maxScore, W, n   # глобальні змінні:
                            # maxScore – поточна максимальна вага рюкзака,
                            # W – максимальна вага рюкзака, n – кількість предметів

    # досягли максимальної глибини рекурсії
    if weight == W or num >= n:   # якщо вага рюкзака W або речі закінчилися
        if score > maxScore:      # порівнюємо поточну вартість рюкзака
                                  # з поточною максимальною вартістю знайденою раніше
            maxScore = score      # зберігаємо оптимальний розв'язок (при необхідності)
        return

    # будуємо наступні варіанти
    max_score(weight,     score,          num + 1)  # предмет не кладемо у рюкзак
    max_score(weight + 1, score + a[num], num + 1)  # предмет кладемо у рюкзак
```

Виклик цієї функції буде мати вигляд

Лістинг 4.1.5. (продовження)

```
maxScore = 0                          # максимальна вартість рюкзака
W, n = map(int, input().split())      # вага рюкзака і кількість різних речей
a = list(map(int, input().split()))   # список цінностей речей
max_score(0, 0, 0)                    # старт рекурсивної функції
print(maxScore)                       # виведення результату
```

§4.2. Метод гілок та меж

Чи не найбільшою групою задач, які розв'язуються методом повного перебору є задачі дискретної оптимізації. Такі задачі мають скінченну множину допустимих розв'язків, які теоретично можна перебрати і вибрати найкращий, тобто той, що дає мінімум (або максимум) цільової функції. На практиці ж такий перебір може стати нездійсненним навіть для задач, дискретна система станів якої є невеликою. Тому постає задача оптимізації підходу повного перебору. Така оптимізація, завжди використовує властивості задачі, що розв'язується і полягає у тому, щоб так організувати перебір, щоб відкинути значну частину хибних розв'язків. Такі методи перебору називаються методами неявного перебору і, найбільш поширений з них – метод гілок і меж.

Метод гілок і меж базується на двох процедурах – розгалуженні та знаходженні оцінок (меж). Процедура розгалуження полягає у тому, що множина допустимих розв'язків розбивається на підмножини меншого розміру. На кожному кроці елементи такого розбиття аналізуються на предмет того, чи містить дана підмножина оптимальний розв'язок чи ні. Якщо розглядається задача знаходження мінімуму цільової функції, то така перевірка здійснюється шляхом обчислення оцінки знизу для цільової функції на даній підмножині. Якщо оцінка

знизу не менша за рекорд (найкращий, зі знайдених розв'язків), то підмножина може бути відкинута, оскільки вона не містить розв'язку, що кращий за рекорд. Якщо значення цільової функції на знайденому розв'язку є меншим за рекорд, то відбувається зміна рекорду. Якщо всі елементи розбиття вдається відкинути, алгоритм завершає свою роботу, а поточний рекорд є оптимальним розв'язком. Інакше, серед підмножин, що залишилися обирається найперспективніша, наприклад з найменшим значенням нижньої оцінки, для якої проводиться поділ на підмножини. Нові підмножини знову аналізуються. Ця процедура проводиться (рекурсивно) доти, доки не буде знайдено оптимальний розв'язок.

Задача знаходження максимуму цільової майже повністю повторює вищенаведений опис, за виключенням того, що аналізується оцінка зверху.

Для демонстрації алгоритму розглянемо задачу

Приклад 4.2.1 [e-olimp, 3533]. Василько просто у захваті від гри "Вормікс". Він досягнув вже значного рівня, тож може відкривати бій із босом. Щоб відкрити новий бій, йому потрібно набрати не менше, ніж K очок за місії. Відомо, що всього є N місій. Для кожної місії відомо скільки часу триватиме її проходження і скільки за неї буде нараховано очок. Також відомо, що Василько є дуже добрим гравцем, а отже він з легкістю зможе пройти будь-яку місію. На жаль, він немає часу, щоб пройти всі місії, але дуже хоче відкрити бій з босом, тож він хоче дізнатися за який мінімальний проміжок часу він зможе набрати не менше K очок.

Як і у задачі про рюкзак, система станів задачі є дискретною, а отже її можна розв'язати повністю перебравши всі можливі розв'язки. Як і у випадку задачі про рюкзак, у цій задачі кожна місія може бути врахована або не врахована. Проте, оптимізуємо цю задачу скориставшись методом гілок і границь. А саме, будемо враховувати кожну наступну місію, лише у тому разі, якщо сумарний час усіх попередніх місій разом з поточною не перевищує мінімального значення часу на деякому розв'язку (рекорду).

Отже, рекурсивна функція знаходження мінімального часу проходження місій буде мати вигляд

Лістинг 4.2.1. Метод гілок та меж.

```
minTime = 100500   # ініціалізація значення рекорду

def findMinTime(time, score, mission_num):
    """
    :param time:        поточне значення часу
    :param score:       поточний рахунок
    :param mission_num: номер місії
    """
    global minTime  # глобальна змінна, що містить рекорд

    # Термінальна гілка, якщо опрацьовані всі місії
    if mission_num >= N:
        # Якщо значення цільової функції на знайденому розв'язку є меншим за рекорд
        if score >= K and minTime > time:
            minTime = time  # зміна рекорду
        return

    # рекурсивний виклик без урахуванням місії mission_num
    findMinTime(time, score, mission_num + 1)

    nextTime = time + t[mission_num]
    # Якщо оцінка знизу не менша за рекорд, то підмножина може бути відкинута
    if nextTime >= minTime:
        return

    nextScore = score + a[mission_num]  # рахунок з урахування місії mission_num
    # рекурсивний виклик з урахуванням місії mission_num
    findMinTime(nextTime, nextScore, mission_num + 1)
```

Зчитування даних задачі та виклик цієї функції буде мати вигляд

Лістинг 4.2.1. (продовження)

```
maxN = 100
a = [0] * maxN
t = [0] * maxN
```

```
# зчитування даних задачі
N, K = map(int, input().split())
# зчитування вартостей місій та часу їхнього проходження
for i in range(N):
    a[i], t[i] = map(int, input().split())

findMinTime(0, 0, 0)      # старт рекурсивної функції

# Виведення результату
if minTime == 100500:
    print(-1)
else:
    print(minTime)
```

§4.3. Метод «Розділяй і володарюй»

Одним з найважливіших методів, що широко застосовується при проектуванні ефективних алгоритмів є метод, що називається **методом декомпозиції** (або на жаргоні алгоритмістів – метод «розділяй і володарюй»). Цей метод передбачає поділ вихідної задачі на дрібніші задачі, розв'язання яких з алгоритмічної точки зору має меншу складність, на основі розв'язків яких можна легко отримати розв'язок вихідної задачі. При цьому, структура кожної з підзадач є подібною до структури вихідної, що в свою чергу дозволяє далі поділили їх на підзадачі аж доки не дійдемо до підзадач розв'язання яких є тривіальним.

Отже, будь-який алгоритм, що розв'язує задачу методом декомпозиції складається з трьох кроків:

1. Розбиття задачі на кілька простіших незалежних між собою підзадач;
2. Розв'язання кожної з підзадач (явно у тривіальних випадках або рекурсивно);
3. Об'єднання отриманих розв'язків підзадач.

Як бачимо реалізація концепції «розділяй та володарюй» має рекурсивний характер. Враховуючи специфіку методу, оцінка часу виконання завжди буде зображуватися у вигляді рекурентної формули

$$T(n) = aT(n/b) + f(n)$$

де, a – кількість підзадач, n/b – розмір підзадач, $f(n)$ – час, що витрачається на декомпозицію задачі та об'єднання результатів розв'язків підзадач.

Прикладами застосування цього методу є вивчені раніше алгоритми сортування злиттям та бінарний пошук. Іншим прикладом застосування цього методу є підрахунок елементів послідовності чисел Фібоначчі використовуючи рекурентні формули або рекурсивний опис.

Метод «розділяй і володарюй», фактично, є різновидом концепції повного перебору.

РОЗДІЛ 5. ЛІНІЙНІ СТРУКТУРИ ДАНИХ

Ми вже знайомі з деякими структурами даних, такими як списки, словники, множини, які входять до бази Python. Розглянемо інші, більш складні структури даних, без розуміння яких не можна уявити сучасне програмування.

Спочатку розглянемо три простих, проте дуже потужних концепцій – стек, черга, зв'язний список. Ці структури є прикладами колекцій, елементи яких впорядковані в залежності від того, як вони додавалися або видалялися. Доданий, елемент залишається на одному і тому ж місці по відношенню до решти, що були додані у колекцію раніше або пізніше нього. Колекції такого роду часто називають **лінійними структурами даних**.

§5.1. Стек

5.1.1. Означення та приклади

Означення 5.1. **Стек** – це динамічна структура даних із принципом до-ступу до елементів "останнім прийшов – першим пішов" (англ. LIFO – last in, first out). Усі операції у стеку проводяться тільки з одним елементом, що називається **верхівкою стека**, та був доданий до стеку останнім.

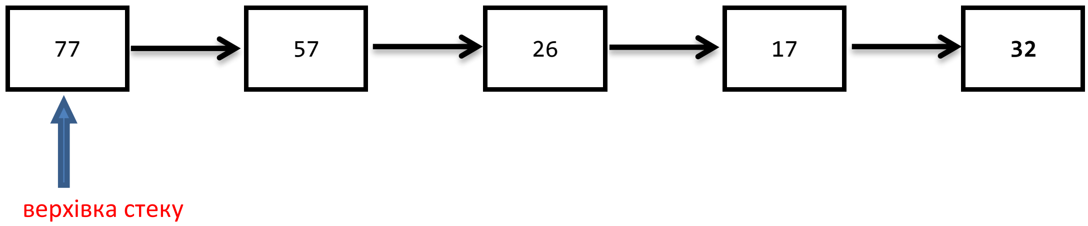


Рисунок 5.1.1. Стек елементів

На рисунку нижче наведено стек книжок. Єдиною книжкою, до якої ми маємо доступ, так щоб не зруйнувати всю колекцію є верхня книга. Щоб отримати доступ до книжок, що знаходяться нижче, потрібно прибрати по одній книжці зі стопки.

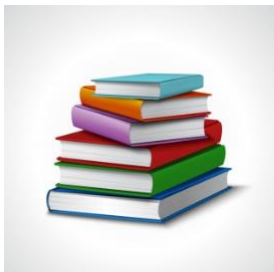


Рисунок 5.1.2. Стопка книжок утворює стек

Базові дії при роботі зі стеком

Для стеку визначають чотири базових операції:

1. Створення порожнього стеку.
2. Операція **`empty`**`()`– визначення, чи є стек порожнім. Повертає булеве значення.
3. Операція **`push`**`(item)` – вштовхнути (додати) елемент `item` до стека. Після цього стек змінюється. Елемент `item` стає верхівкою стеку.
4. Операція **`pop`**`()` – взяти верхівку стека. Ця операція повертає верхівку стеку. Стек при цьому змінюється.

Крім зазначених вище операцій, опціонально визначають інші операції, наприклад, визначають операцію **`top`**`()`, що повертає верхівку стеку, без видалення верхівки зі стеку. Також корисною є операція визначення довжини (тобто кількості елементів) у стеку.

У нижченаведеній таблиці наведено приклад роботи операцій зі стеком. У колонці «Вміст стеку після здійснення операції» напівжирним шрифтом виділено елемент, що є верхівкою стека.

Операція над стеком	Вміст стеку після здійснення операції	Значення, що повертається в результаті здійснення операції
`stack = Stack()`	`[]`	
`stack.empty()`	`[]`	`True`
`stack.push(32)`	`[32]`	
`stack.push(17)`	`[17, 32]`	
`stack.push(26)`	`[26, 17, 32]`	
`len(stack)`	`[26, 17, 32]`	3
`stack.empty()`	`[26, 17, 32]`	`False`
`stack.pop()`	`[17, 32]`	26
`stack.top()`	`[17, 32]`	17

Реалізація стеку на базі вбудованого списку Python

Найпростішу реалізацію стеку у Python можна здійснити на базі вбудованого списку (**`list`**). У такій реалізації елементи стеку розміщують у списку, причому вважається, що верхівка стеку знаходиться в кінці списку. Тоді для додавання та віднімання елементів використовують методи списку **`append()`** та **`pop()`** відповідно.

Опишемо клас `Stack`, що реалізує базові методи роботи зі стеком зазначені у попередньому пункті.

Лістинг 5.1.1. Реалізація стеку на базі вбудованого списку.

```
class Stack:
    """ Клас, що реалізує стек елементів
        на базі вбудованого  списку Python """

    def __init__(self):
        """ Конструктор """
        self.items = []

    def empty(self):
        """ Перевіряє чи стек порожній

        :return: True, якщо стек порожній
        """
        return len(self.items) == 0

    def top(self):
        """ Повертає верхівку стека
            Сам елемент при цьому лишається у стеці

        :return: Верхівку стеку
        """

        if self.empty():
            raise Exception("Stack: 'top' applied to empty container")
        return self.items[-1]

    def __len__(self):
        """ Розмір стеку

        :return: Розмір стеку
        """
        return len(self.items)
```

Для перевірки роботи стеку, створимо новий стек, вштовхнемо туди кілька нових елементів, виштовхнемо елементи зі стеку та виведемо отримані елементи на екран.

Лістинг 5.1.1. Реалізація стеку на базі вбудованого списку (Продовження).

```
if __name__ == "__main__":
    stack = Stack()
    stack.push(10)
    stack.push(11)
```

```
    stack.push(12)
    stack.push(13)

    print(stack.pop())
    print(stack.pop())
    print(stack.pop())
    print(stack.pop())
```

Результатом роботи програми буде

```
13
12
11
10
```

Реалізація стеку як рекурсивної структури даних

Наведена вище реалізація є дещо штучною, та не відображає самої концепції стеку. Дійсно, при бажанні можна легко модифікувати програму так, щоб доступ був не лише до верхівки стеку, а й до будь-якого його елементу. Крім цього, додавання елементів до стеку може здійснюватися не за сталий час, а за лінійний, наприклад, у випадку, якщо для додавання нового елементу до стеку необхідно провести операцію реалокації[3].

Тому більш правильним є зображення стека у вигляді рекурсивної структури. У такому разі кожен елемент стека є структурою, що крім даних, містить посилання на наступний елемент стека. Доступ є лише до елемента, що є верхівкою стека. Останній елемент у стеку посилається на **None**.

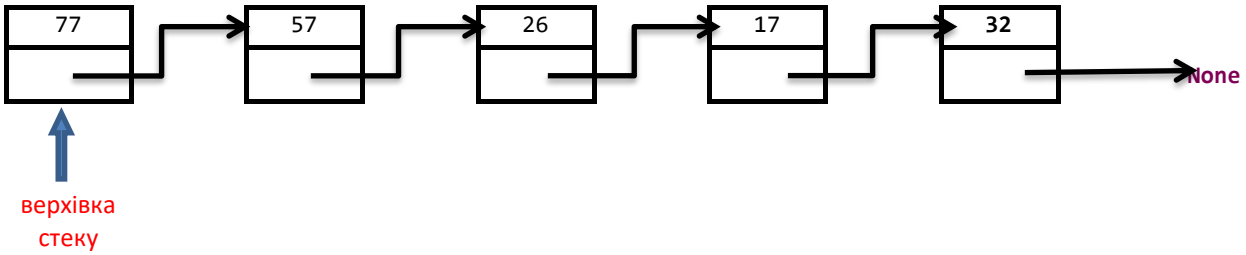


Рисунок 5.1.3. Рекурсивна реалізація стеку.

Отже, опишемо допоміжний клас Node (вузол стека), що містить дані (навантаження вузла) та посилання на наступний вузол стека, тобто об'єкт класу Node.

Лістинг 5.1.2. Реалізація стеку як рекурсивної струткутри. Допоміжний клас Node – Вузол списку.

```
class Node:
    """ Допоміжний клас, що реалізує вузол стеку """

    def __init__(self, item):
        """ Конструктор
        :param item: навантаження вузла
        """
        self.item = item   # створюєм поле для зберігання навантаження
        self.next = None   # посилання на наступний вузол стеку
```

Тоді реалізація класу Stack буде мати такий вигляд

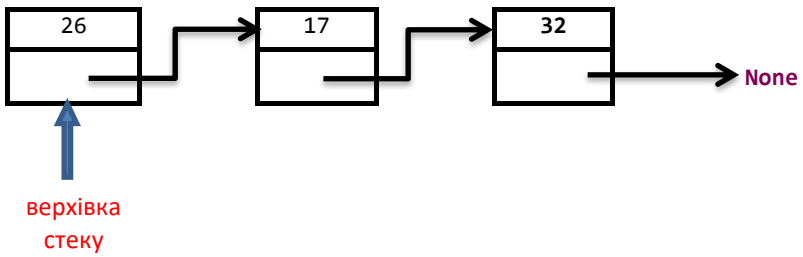
Лістинг 5.1.2. Продовження. Реалізація стеку як рекурсивної струткутри.

```
class Stack:
    """ Клас, що реалізує стек елементів
        як рекурсивну структуру """

    def __init__(self):
        """ Конструктор """
        self.top_node = None  # посилання на верхівку стеку
```

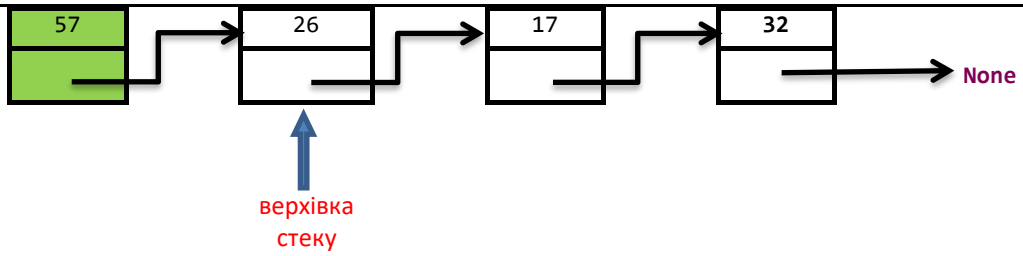
[3] Аллокація (allocation) у програмуванні, процес динамічного виділення пам'яті для розміщення даних об'єктів. Реаллокація (reallocation) – зміна розташування даних об'єкта у пам'яті.

```
    def empty(self):
        """ Перевіряє чи стек порожній

        :return: True, якщо стек порожній
        """
        return self.top_node is None

    def push(self, item):
        """ Додає елемент у стек

        :param item: елемент, що додається у стек
        """

        new_node = Node(item)              # Створюємо новий вузол стеку
        if not self.empty():               # Якщо стек не порожній, то новий вузол
            new_node.next = self.top_node  # має посилатися на поточну верхівку

        self.top_node = new_node  # змінюємо верхівку стека новим вузлом

    def pop(self):
        """ Забирає верхівку стека
            Сам вузол при цьому прибирається зі стеку

        :return: Навантаження верхівки стеку
        """
        if self.empty():   # Якщо стек порожній
            raise Exception("Stack: 'pop' applied to empty container")

        current_top = self.top_node         # запам'ятовуємо поточну верхівку стека
        item = current_top.item             # запам'ятовуємо навантаження верхівки
        self.top_node = self.top_node.next  # замінюємо верхівку стека наступним вузлом
        del current_top  # видаляємо запам'ятований вузол, що містить попередню верхівку
        return item

    def top(self):
        """ Повертає верхівку стека
            Сам вузол при цьому лишається у стеці

        :return: Навантаження верхівки стеку
        """
        if self.empty():
            raise Exception("Stack: 'top' applied to empty container")
        return self.top_node.item
```

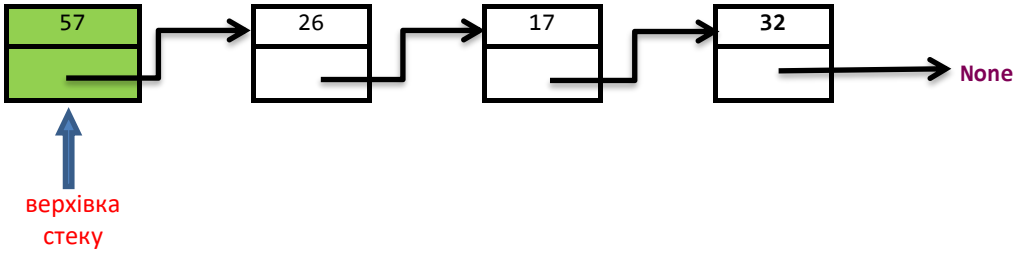
Пояснимо для прикладу схематично роботу методу `push()`. Припустимо у стек додано елементи (32, 17, 26)



Вштовхнемо у стек елемент 57. Для цього створюємо новий вузол стеку, записуємо у нього дані – число 57, а посилання на ступний елемент для нього – верхівка стеку:



Зміщуємо верхівку стека на новий вузол.



Метод `pop()` працює відповідно до метода `push()` у зворотному порядку. Для кращого розуміння реалізації стека як рекурсивної структури, пропонуємо читачу змоделювати роботу метода `pop()` самостійно.

5.1.2. Застосування стеку

Стек у програмуванні має надзвичайно широке застосування, зокрема, у алгоритмах де потрібно дотриматися принципу "останнім прийшов – першим пішов". Напевно чи не найпоширенішим прикладом є інверсія даних, тобто коли послідовність даних потрібно переписати у зворотному порядку. Розглянемо інші класичні застосування стеку, що зустрічаються при розв'язанні реальних задач у інформатиці.

Конвертування чисел з однієї системи числення до іншої

У повсякденному житті, під час різноманітних обчислень, ми використовуємо позиційну десяткову систему числення. У цій системі числення кожне число записується у вигляді послідовності цифр – значень розрядів цього числа – тобто кількості одиниць, десятків, сотень і т.д:

$$z = a_{n-1}a_{n-2} \dots a_1a_0 \tag{5.1.1}$$

де a_0 – цифра у нульовому розряді (кількість одиниць), a_1 – цифра у першому розряді (кількість десятків), і так далі, a_{n-1} – цифра у найстаршому розряді. При цьому такому запису відповідає сума

$$z = \sum_{i=0}^{n-1} a_i \cdot 10^i \tag{5.1.2}$$

тут n – кількість розрядів числа (розрядність), i – номер розряду цифри a_i, починаючи з нульового. При цьому число 10 називається основою десяткової системи числення, а всі цифри, як ми знаємо, задовольняють нерівність

$$0 \le a_i < 10, i = 1, \dots, n-1.$$

Наприклад, число 256 зображується таким чином

$$256 = 2 \cdot 10^2 + 5 \cdot 10^1 + 6 \cdot 10^0$$

тобто число 256 складається з 2 сотень, 5 десятків та 6 одиниць.

Поруч із десятковою системою числень широко застосовуються інші позиційні системи числення, відмінною від числа 10 основою $b \ge 2$. Найуживаніші з них, це системи числення з основами 2, 8 та 16.

У таких системах числення, будь-яке число z також визначається послідовністю значень цифр у відповідних позиціях (розрядах)

$$z = a_{n-1}a_{n-2} \dots a_1a_0 \tag{5.1.3}$$

при цьому, набір "цифр" задовольняє нерівності

$$0 \le a_i \le b-1, i = 1, \dots, n-1.$$

і такому запису відповідає сума

$$z = \sum_{i=0}^{n-1} a_i \cdot b^i \qquad (5.1.4)$$

Часто, щоб дати зрозуміти, що використовується система числення з основою відмінною від 10, для запису числа використовується такий запис

$$(a_{n-1} \dots a_1 a_0)_b$$

де b – основа системи числення. При цьому дужки частіше за все опускають

$$1033_8$$

Під час роботи з комп'ютером ми постійно зіштовхуємося з конвертуванням чисел з однієї системи числення у іншу. Дійсно, нам звичною для використання є десяткова система числення, у той час як комп'ютер може працювати виключно з системою числення з основою 2 – двійковою системою числення. Причини такого підходу базуються на апаратній (фізичній) реалізації комп'ютера і виходить за межі цього посібника. Тому лишаємо це питання читачу для самостійного вивчення.

Звичайно, частіше за все, користувач навіть не підозрює про постійні операції конвертування з десяткової системи числення у двійкову і навпаки, під час використання комп'ютера, оскільки останній це робить автоматично без явного втручання користувача.

Давайте розберемося, як же працює алгоритм конвертування чисел з однієї системи числення у іншу. Спочатку розглянемо алгоритм перетворення з системи числення з основою b, відмінною від 10, у десяткову систему числення. З вищенаведених формул такий алгоритм отримується очевидним чином – досить просто розписати число за формулою (5.1.4) і порахувати за правилами десяткової системи числення. Дійсно, таким чином, наприклад

$$1033_8 = 1 \cdot 8^3 + 0 \cdot 8^2 + 3 \cdot 8^1 + 3 \cdot 8^0 = 539$$

Тепер, спробуємо конвертувати число записане у десятковій системі числення у систему числення з основою $b \geq 2$. Як правило, цей алгоритм викликає дещо більше труднощів у порівнянні з попереднім. Проте, фактично він є нічим інший як зворотною процедурою. І якщо попередня процедура базувалася на операції множення то відповідно ця буде базуватися на діленні.

Для кращого розуміння, спочатку розглянемо алгоритм на прикладі конвертування числа 539 у відповідне число у системі числення з основою 8. Очікується, ми отримаємо число 1033_8. Випишемо послідовно, кілька степенів числа 8, так щоб всі вони будуть менші за вихідне число 539

$$8^0 = 1$$
$$8^1 = 8$$
$$8^2 = 64$$
$$8^3 = 512$$

Тепер будемо послідовно, починаючи з найбільшого записаного степені числа 8, дивитися скільки його цілих частин поміщається у нашому числі. Ця процедура повністю аналогічна до такої якби нас цікавило, скільки у числі десяткової системи числення міститься сотень, десятків та одиниць (наприклад, у числі 256 міститься 2 сотні, 5 десятків та 6 одиниць).

Отже,

539 // 512	=	**1**	mod 27
27 // 64	=	**0**	mod 27
27 // 8	=	**3**	mod 3
3 // 1	=	**3**	mod 0

Як бачимо, ми отримали число 1033_8. Хоча такий алгоритм конвертації числа з однієї системи числення дуже простий і не викликає труднощів з точки зору розуміння, проте на практиці він не застосовується через надмірні алгоритмічні вимоги. Дійсно, наприклад, цей алгоритм вимагає створення списку степенів основи, що накладає додаткові витрати часу і пам'яті. Тому на практиці використовується алгоритм, що фактично, здійснює вищенаведену процедуру лише у зворотному порядку. Отже, будемо послідовно ділити число на 539 на основу системи числення 8 з остачею, далі результат ділення і так далі, поки число не перетвориться у нуль. Запишемо результат у такому вигляді

```
539 // 8 =  67  mod 3
          67 // 8 =  8  mod 3
                     8 // 8 =  1  mod 0
                               1 // 8 =  0  mod 1
```

Давайте подивимося уважно на отриманий результат. Чи помітили ви щось цікаве? Так, дійсно, якщо записати отримані остачі від ділення у порядку зворотному до часу їхнього отримання (ось де приховане використання стеку у цій задачі!), то отримаємо 1033 – запис числа 539 у позиційній вісімковій системі числення.

Лістинг 5.1.3. Конвертування числа.

```
def convert(dec_number, base):
    """ Перетворює задане десяткове число, до заданої системи числення
    :param dec_number: вхідне десяткове число
    :param base:       основа системи числення [2, 16]
    :return:           рядок-число у системи числення з основою base
    """

    assert 2 <= base <= 16  # Перевіряємо основу від ділення

    stack = Stack()  # Використаємо стек, для запису отриманих остач від ділення
    while dec_number > 0:
        stack.push(dec_number % base)
        dec_number //= base

    converted = ""    # Рядок, що містить конвертоване число
    while not stack.empty():
        converted = converted + get_char_digit(stack.pop())

    return converted
```

Вищенаведена функція використовує допоміжну функцію `get_char_digit()` що перетворює число [0, 15] числа у відповідну йому «цифру». Наведемо її код без додаткових пояснень

Лістинг 5.1.3. Продовження. Функція `get_char_digit()`

```
def get_char_digit(digit):
    """ Допоміжний метод, що за числом повертає символ-цифру системи числення
          0 -> '0',..., 9 -> '9', 10 -> 'A',..., 15 -> 'F'
    :param digit: число з проміжку 0,.., 15
    :return: рядок що місить символ-цифру системи числення
    """
    assert digit <= 16
    if digit <= 9:
        str_digit = str(digit)
    else:
        str_digit = chr(ord("A") + digit - 10)

    return str_digit
```

Для перевірки роботи програми запустимо її для деякого набору чисел

Лістинг 5.1.3. Продовження.

```
print(convert(100, 2))      # 100 у двійкову систему числення
print(convert(63, 8))       # 63 у вісімкову систему числення
print(convert(102234, 11))  # 102234 у систему числення з основою 11
print(convert(2286755, 16)) # 2286755 у шістнадцяткову систему числення
```

Збалансовані дужки

Часто, при аналізі арифметичних виразів постає питання про правильну розстановку дужок, що визначають пріоритет арифметичних операцій:

$$4\big((x+1)*(y+2)-2\big)$$

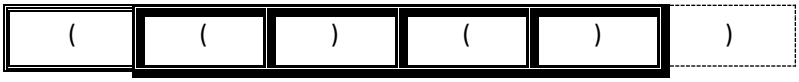
Зараз будемо говорити не про правильність такої розстановки дужок з математичної точки зору, тобто чи правильно розставлені дужки (змінений пріоритет операцій) для отримання виразу, що коректно розв'язує конкретну поставлену задачу. Зараз нас буде цікавити їхня **збалансованість**, тобто чи відповідає кожній відкритій дужці, закрита і пари дужок правильно вкладені одна в іншу. Якщо з виразу прибрати всі символи крім дужок, то вираз, отриманий таким чином (що містить лише дужки) називається дужковою послідовністю. Будемо казати, що дужкова послідовність правильна, якщо дужки у ній збалансовані. У таблиці нижче наведено приклади правильних і не правильних дужкових послідовностей

Дужкова послідовність	Аналіз
(( )( ))	правильна
(( )( )( )( ))	правильна
(((( ))))	правильна
)( )(	не правильна
(((( )	не правильна
( ))))	не правильна

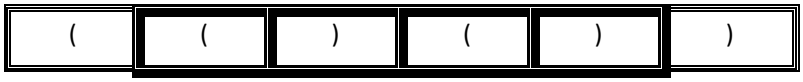
Основна ідея алгоритму буде полягати у тому, що ми будемо читати дужки зліва на право і намагатися співставити відкритій дужці, відповідну закриту дужку. Якщо таке співставлення на деякому кроці буде знайдено, то знайдено контейнер, що містить правильну дужкову підпослідовність, яку можна сміливо прибрати з розгляду (оскільки вона не псує правильність дужкової послідовності). Цей процес продовжується далі, доки не будуть відкинуті всі контейнери або, дійшовши до кінця дужкової послідовності, не буде встановлено, що вона не правильна.

Для прикладу, розглянемо першу дужкову послідовність з вищенаведеної таблиці. Після аналізу перших трьох дужок, зустрівся контейнер (утворений другою і третьою дужкою), який можна виключити з аналізу.

Далі, вилучаємо з розгляду наступний дужковий контейнер, утворений четвертою та п'ятою дужкою



І, нарешті, розглянувши останню дужку, отримуємо дужковий контейнер, що визначається першою і останньою дужкою



Отже, дужкова послідовність є правильною.

Як ми побачили, основна ідея алгоритму полягає у тому, щоб розпізнати дужкові контейнери. Тому, для написання алгоритму, що розв'язує цю задачу ідеально підходить структура даних – стек.

Таким чином, алгоритм буте таким, створивши порожній стек, перебираємо послідовно дужки виразу:

- зустрівши відкриту дужку будемо її запам'ятовувати (вштовхуючи у стек) – це потенційний початок дужкового контейнера.
- щойно зустрічається закрита дужка, намагаємося дістати зі стеку відкриту дужку:
  - якщо це можливо, це означає, що знайдено дужковий контейнер і він вилучається з розгляду;
  - якщо це не можливо (стек порожній), то вихідний вираз утворює не правильну дужкову послідовність, що і завершує аналіз.
- після того, як усі дужки опрацьовані (і, відповідно, вилучено всі правильні контейнери), аналізуємо чи стек порожній:
  - якщо так, то вихідний вираз утворює правильну дужкову послідовність,
  - якщо ні, то вихідний вираз утворює не правильну дужкову послідовність.

Наступний лістинг містить підпрограму, що перевіряє чи задана послідовність дужок є правильною дужковою послідовністю.

Лістинг 5.1.4. Правильна дужкова послідовність.

```
def bracketsChecker(brackets_sequence):
    """ Перевіряє чи brackets_sequence правильна дужкова послідовність

    :param brackets_sequence: дужкова послідовність
    :return: True, якщо brackets_sequence - правильна дужкова послідовність
    """
    s = Stack()              # Створюємо порожній стек
    for bracket in brackets_sequence:
        if bracket == "(":
            s.push(bracket)  # Потенційний початок контейнера
        else:
            if s.empty():
                return False # Дужкова послідовність не правильна
            else:
                s.pop()      # Прибираємо контейнер з розгляду

    return s.empty()

print(bracketsChecker('(()())'))
print(bracketsChecker('(()()()())'))
print(bracketsChecker('((()))'))
print(bracketsChecker(')()('))
print(bracketsChecker('((((('))
print(bracketsChecker('()))))'))
```

Звичайно, що задачу наведену вище (якщо у виразі використовують лише дужки одного виду) можна легко розв'язати без використання стеку. Проте, якщо вираз буде містити дужки різних видів, (наприклад, крім круглих дужок "(" та ")", ще дужки виду "{" , "}", "[" , "]") без використання стеку таку задача буде розв'язати значно складніше. А якщо кількість дужок значна, то альтернативи стеку при розв'язанні такої задачі просто не існує.

Інфіксні та постфіксні арифметичні вирази

Вивчаючи арифметику, ще з початкової школи ми звикли, що для того, щоб записати вираз для суми двох чисел, потрібно записати ці два числа, поставивши між ними оператор суми, наприклад сума чисел 4 і 6 записується, як

$$4 + 6 \tag{5.1.5}$$

Для запису виразу для множення, потрібно діяти подібний чином:

$$4 * 6$$

Такий тип запису арифметичних виразів називається **інфіксним** (далі інфіксна нотація), оскільки оператор знаходиться між операндами. Хоча інфіксна нотація здається очевидною і зручною, проте вона має кілька суттєвих недоліків. Одним з таких недоліків є неоднозначність пов'язана з порядком, у якому виконуються арифметичні операції, якщо вираз містить більше ніж одну операцію. І якщо для деяких операторів жодних проблем не виникне, то для інших зміна порядку обчислення приведе до принципово іншого результату. Дійсно, розглянемо такий класичний приклад арифметичного виразу

$$2 + 2 * 2 \tag{5.1.6}$$

Результат цього арифметичного виразу буде залежати від того, яку операцію виконати раніше. Якщо першою виконати операцію додавання, а потім множення, то результат буде 8, якщо ж навпаки, то 6. Для того, щоб подолати цю неоднозначність у арифметиці є правила, що вказують чіткий порядок виконання дій у виразах. Ці правила базуються на тому, що для всіх арифметичних операторів визначені пріоритети. Операції з вищими пріоритетами виконуються раніше ніж з нижчими. Якщо ж пріоритет операцій однаковий, то дії виконуються послідовно зліва на право.

Як ми знаємо, за правилами арифметики, операція множення має вищий пріоритет ніж додавання. А значить у вищенаведеному прикладі потрібно спочатку виконати операцію множення, а вже потім додавання. Відповідно, правильним результатом обчислення вищенаведеного виразу буде 6.

Для зміни пріоритету арифметичних операцій використовуються дужки. Таким чином, якщо у вищенаведеному виразі потрібно спочатку виконати операцію додавання, то вираз потрібно записати у такому вигляді

$$(2 + 2) * 2$$

Інфіксна нотація для запису арифметичних виразів звична для користувача, тому використовується як домінуюча у повсякденному житті. Навіть досить великий вираз зі значною кількістю арифметичних операцій ми можемо проаналізувати, визначити пріоритетність операцій, розбити вираз на частини та обчислити його крок за кроком.

Проте для комп'ютера інфіксна нотація є складнішою для аналізу. Здебільшого це пов'язано з архітектурою комп'ютера, а саме яким чином проводяться арифметичні операції на процесорі. Тому на низькому рівні він використовує інші нотації – **префіксну** (або польську) або **постфіксну** (або обернену польську). На перший погляд вони можуть здатися неочевидними, проте вони повністю долають проблему неоднозначності результату, що може виникати при використанні інфіксної нотації. При цьому, жодна з них не використовує дужки для зміни пріоритету. Префіксна нотація запису виразів вимагає, щоб усі оператори передували двом операндам на які вони діють, у той час як постфіксна, щоб оператори знаходилися після операндів.

Розглянемо арифметичний вираз (5.1.5). Якщо бінарний оператор додавання поставити на початку операндів, на які він діє

$$+4\ \ 6 \tag{5.1.7}$$

то отримаємо префіксну нотацію, якщо ж оператор поставити після операндів

$$4\ \ 6+ \tag{5.1.8}$$

то буде вже постфіксна нотація.

Вираз

$$a + b * c$$

у префіксній нотації буде мати вигляд

$$+a * bc$$

Оператор множення розташовується безпосередньо перед операндами b та c, що вказує на пріоритет множення перед додаванням. Після цього виконується оператор додавання операнда a з результатом множення. У постфіксній нотації цей же вираз буде мати вигляд

$$abc * +$$

Як бачимо знову порядок операцій відповідає пріоритетам операторів множення та додавання.

Чому ж префіксна і постфіксна форми зручніші для комп'ютера? Давайте розберемося, як же буде проводитися обчислення комп'ютером, наприклад у випадку постфіксної форми. Все надзвичайно просто: скануємо вираз поки не зустрінемо арифметичний оператор. Щойно це відбулося – проводимо обчислення для двох операндів, що знаходяться лівіше оператора. Отриманий результат записується на їхнє місце.

Тоді постає запитання, як же конвертувати вираз до префіксної або постфіксної форми. Оскільки постфіксна нотація є поширенішою, то розглянемо алгоритм конвертування виразу записаного у інфіксній нотації до виразу записаного у постфіксній нотації.

Щоб спростити алгоритм, з метою не відволікати читача зайвими технічними деталями, припустимо інфіксний вираз є рядком токенів розділених символами пропуску. Токенами операторів є символи

```
'+', '-', '*', '/',
```

що відповідають операторам додавання, віднімання, множення та ділення, а також круглі дужки

```
'(', ')'
```

для визначення пріоритетів. Токеном операнда є рядок, що містить число (неперервна послідовність цифр). Нижче наведено приклад такого рядка токенів

```
"25 * ( 3 + 5 )"
```

Тоді алгоритм перетворення інфіксного виразу, записаного списком токенів до постфіксного аналогу буде таким

1. Перетворюємо інфіксний рядок у список токенів (за нашим припущенням це можна здійснити методом **`split()`**)
2. Створюємо порожній стек для зберігання операторів та дужок.
3. Створюємо порожній вихідний список для виразу у постфіксній нотації.

4. Скануємо список токенів зліва направо.
    - Якщо токен є операндом, то додаємо його у кінець вихідного списку.
    - Якщо токен ліва дужка, кладемо її у стек операторів.
    - Якщо токен права дужка, виштовхуємо елементи зі стеку операторів, поки не буде знайдено відповідна ліва дужка. При цьому кожен оператор додається у кінець вихідного списку.
    - Якщо токен є оператором, то вштовхуємо його в стек операторів. При цьому аналізуємо оператор, що у верхівці стеку: якщо він має вищий або такий же пріоритет, то виштовхуємо його і додаємо до вихідного списку.
5. Після завершення сканування вхідного списку, перевіряємо стек операторів – всі оператори, що містяться у ньому слід виштовхнути зі стеку і додати у кінець вихідного списку.

Нижче, наведена реалізація класу `StringCalculator`, що обчислює вираз заданий рядком токенів. Конвертування інфіксного рядка у постфіксну форму здійснює метод **`convert_to_polish()`** , що використовує вищенаведений алгоритм.

Лістинг 5.1.5. Рядковий калькулятор.

```
" Словник операторів, що використовуються у калькуляторі та їхні пріоритети "
OPERATORS = {
    "+": 1,  # Оператор додавання та його пріорітет
    "-": 1,  # Оператор віднімання та його пріорітет
    "*": 2,  # Оператор множення та його пріорітет
    "/": 2,  # Оператор ділення та його пріорітет
}

class StringCalculator:
    """ Клас рядковий калькулятор.

        Обчислює значення арифметичного виразу використовуючи обернений польский запис
    """

    def __init__(self, str_expression):
        """ Конструктор
        :param str_expression: рядок, що містить арифметичний вираз у інфіксному вигляді.
        """
        self.mInfixStr = str_expression  # Поле (рядок), що містить арифметичний вираз
                                         # у інфіксному вигляді
        self.mPostfixList = self.convert_to_polish()  # Поле (список), що містить
                                         # арифметичний вираз у постфіксному вигляді

    def set_expression(self, str_expression):
        """ Задає калькулятору арифметичний вираз
        Для спрощення передбачається, що вхідний параметр містить
        правильний арифметичний вираз у інфіксному вигляді

        :param str_expression: рядок, що містить арифметичний вираз у інфіксному вигляді.
        :return: None
        """
        self.mInfixStr = str_expression
        self.mPostfixList = self.convert_to_polish()

    def convert_to_polish(self):
        """ Конвертує арифметичний вираз з інфіксного у постфіксний вигляд

            Для коректної роботи цього методу, передбачається, що у рядку
            (що містить арифметичний вираз) усі операнди, оператори та дужки
            записуються через символ пропуску, наприклад "25 * ( 3 + 5 )"

        :return: Рядок, що містить арифметичний вираз у постфіксному вигляді
        """
        infix_list = self.mInfixStr.split()  # Розділяємо рядок на токени
        postfix_list = []  # Список, що міститиме вираз у постфіксному вигляді
        stack = Stack()    # Допоміжний стек арифметичних операторів та дужок

        for token in infix_list:    # Ітеруємо по всіх токенах інфіксного виразу
            if token in OPERATORS:  # токен є оператором
                while not stack.empty():
                    prev = stack.top()  # підглянемо попередній оператор зі стеку
```

```
                # Якщо попередній токен є оператором
                # пріорітет якого вищий за пріорітет поточного оператора
                if prev in OPERATORS and OPERATORS[prev] >= OPERATORS[token]:
                    stack.pop()                # Видаляємо його зі стеку операторів
                    postfix_list.append(prev)  # Додаємо його до постфіксного списку
                else:
                    break
            stack.push(token)  # кладемо поточний оператор у стек
        elif token == "(":     # токен є лівою дужкою,
            stack.push(token)  # кладемо його в стек
        elif token == ")":     # якщо токен є правою дужкою,
            it = stack.pop()   # Виштовхуємо елементи зі стеку опертаорів stack
            while it != "(":   # доки не знайдемо відповідну ліву дужку.
                postfix_list.append(it)  # при цьому кожен оператор додаємо до списку
                it = stack.pop()
        else:                            # якщо токен є операндом
            postfix_list.append(token)  # додаємо його у кінець постфіксногосписку.

    while not stack.empty():
        postfix_list.append(stack.pop())

    return postfix_list

@staticmethod
def simple_operation(left, right, operator):
    """ Допоміжний метод, що обчислює значення виразу "left operator right"

    :param left:     лівий операнд
    :param right:    правий операнд
    :param operator: оператор
    :return: значення виразу "left operator right"
    """

    assert operator in OPERATORS

    left = float(left)
    right = float(right)

    if operator == "+":
        return left + right
    elif operator == "-":
        return left - right
    elif operator == "*":
        return left * right
    elif operator == "/":
        return left / right

def calculate_by_polish(self):
    """ Обчилює значення виразу використовуючи оберенений польский запис

    :return: Значення арифметичного виразу
    """
    stack = Stack()
    for token in self.mPostfixList:
        if token in OPERATORS:           # Якщо поточний токен оператор
            right_operand = stack.pop()  # Дістаємо перший елемент зі стеку - він
                                         # відповідає правому операнду
            left__operand = stack.pop()  # Дістаємо другий елемент зі стеку - він
                                         # відповідає лівому операнду
            # Обчислюємо значення простого арифмтемтиного виразу
            res = self.simple_operation(left__operand, right_operand, token)
            stack.push(res)    # Кладемо результат обчислень у стек
        else:                  # Якщо поточний токен є операндом
            stack.push(token)  # Кладемо його у стек

    return stack.pop()
```

Для перевірки роботи програми обчислимо вираз "`25 * ( 3 + 5 )`":

Лістинг 5.5. Продовження. Обчислення виразу "`25 * ( 3 + 5 )`".

```
calc = StringCalculator("25 * ( 3 + 5 )")
print(calc.calculate_by_polish())
```

результатом буде

```
200.0
```

§5.2. Черга. Дек. Пріоритетна черга

5.2.1. Черга

Означення 5.2.1. **Черга** – динамічна структура даних із принципом доступу до елементів "першим прийшов – першим пішов" (англ. FIFO – first in, first out).

У черги є початок (голова) та кінець (хвіст).

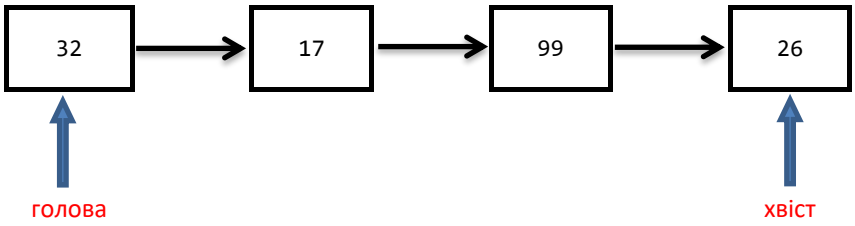


Рисунок 5.2.1. Черга, має початок та кінець.

Елемент, що додається до черги, опиняється у її хвості.

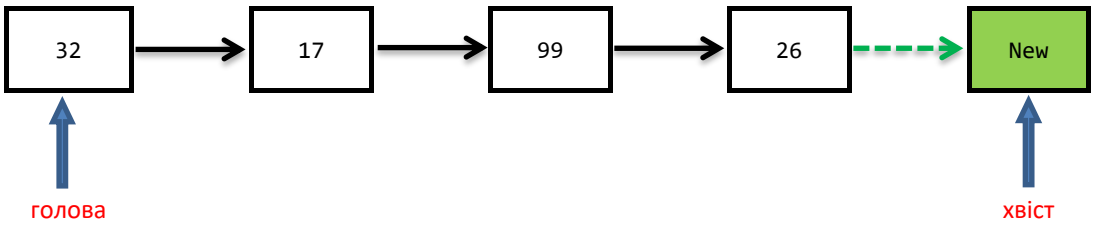


Рисунок 5.2.2. Додавання елемента «New» в кінець черги

Елемент, що береться (тобто видаляється) з черги, розташований у її голові.

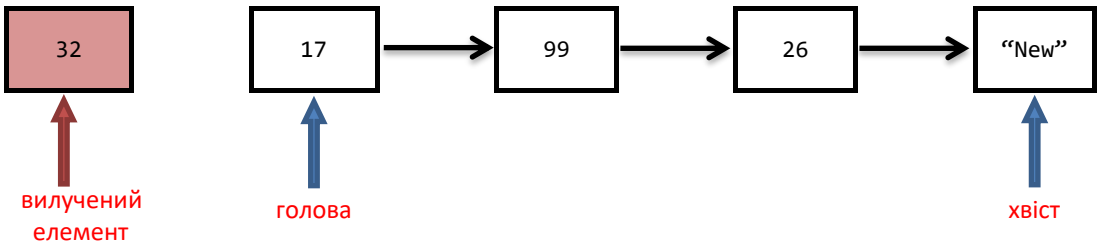


Рисунок 5.2.3. Вилучення першого елемента «32» з черги

Базові операції для роботи з чергою

Класична реалізація черги вимагає реалізувати структуру даних з такими операціями:

1. Створення порожньої черги.
2. Операція **`empty`**`()`– визначення, чи є черга порожньою. Повертає булеве значення.
3. Операція **`enqueue`**`(item)` – додати елемент `item` до кінця черги.
4. Операція **`dequeue`**`()` – взяти елемент з початку черги.

Як і у випадку зі стеком, крім зазначених вище операцій, опціонально визначають інші операції: перегляд елемента в голові та хвості черги без їхнього видалення з черги, розмір черги, тобто кількість елементів у ній, тощо.

У нижченаведеній таблиці наведено приклад роботи операцій із чергою. У колонці «Вміст черги після здійснення операції» напівжирним шрифтом виділено елемент, що є головою черги, а підкресленням – елемент, що знаходиться у її хвості.

Операція над чергою	Вміст черги після здійснення операції	Значення, що повертається в результаті здійснення операції
`queue = Queue()`	`[]`	
`queue.empty()`	`[]`	`True`
`queue.enqueue(32)`	`[32]`	
`queue.enqueue(17)`	`[32, 17]`	
`queue.enqueue(99)`	`[32, 17, 99]`	
`queue.enqueue(26)`	`[32, 17, 99, 26]`	
`queue.enqueue("New")`	`[32, 17, 99, 26, "New"]`	
`len(queue)`	`[32, 17, 99, 26, "New"]`	5
`queue.empty()`	`[32, 17, 99, 26, "New"]`	`False`
`queue.dequeue()`	`[17, 99, 26, "New"]`	32

Реалізація черги на базі вбудованого списку

Як і у випадку зі стеком, спочатку наведемо простішу реалізацію черги, що базується на вбудованому списку. Опишемо клас `Queue` у якому реалізуємо основні операції, а також функцію визначення розміру черги.

Лістинг 5.2.1. Реалізація черги на базі вбудованого списку.

```
class Queue:
    """ Клас, що реалізує чергу елементів
        на базі вбудованого списку Python """

    def __init__(self):
        """ Конструктор """
        self.items = []  # Список елементів черги

    def empty(self):
        """ Перевіряє чи черга порожня

        :return: True, якщо черга порожня
        """
        return len(self.items) == 0

    def enqueue(self, item):
        """ Додає елемент у чергу (у кінець)

        :param item: елемент, що додається
        :return: None
        """
        self.items.append(item)

    def dequeue(self):
        """ Прибирає перший елемент з черги
            Сам елемент при цьому прибирається із черги

        :return: Перший елемент черги
        """
        if self.empty():
            raise Exception("Queue: 'dequeue' applied to empty container")
        return self.items.pop(0)

    def __len__(self):
        """ Розмір черги

        :return: Кількість елементів у черзі
        """
        return len(self.items)
```

Як можна побачити з вищенаведеної реалізації, додавання елементу до черги здійснюється за сталий час $O(1)$. У той час як вилучення елемента з черги здійснюється за час $O(n)$, де n – кількість елементів у черзі. Дійсно, цього вимагає операція **pop(0)** – вилучення першого елементу списку.

Така реалізація не найкращий варіант з практичної точки зору – якщо черга використовується у системі де постійно в черзі знаходиться велика кількість елементів, продуктивність програми буде дуже низькою.

Реалізація черги як рекурсивної структури даних

Наведемо реалізацію черги, у якій операції додавання нового елементу до черги та вилучення елементу з голови здійснюється за сталий час $O(1)$. Така реалізація дуже подібна до рекурсивної реалізації стеку – кожен елемент черги, крім даних, також містить посилання на наступний елемент черги, а для доступу до голови та хвоста черги використовується спеціальний елемент керування.

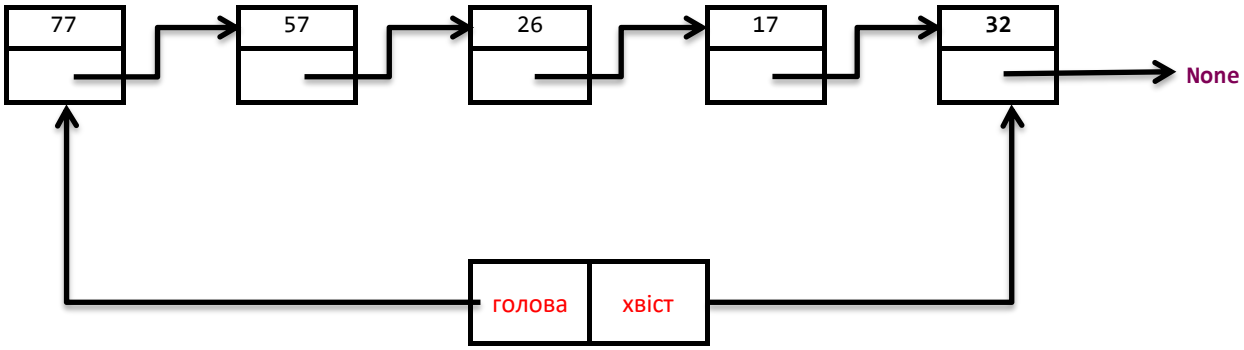


Рисунок 5.2.4. Рекурсивна реалізація черги.

Як у випадку зі стеком, рекурсивна реалізація черги використовує структуру даних – Node (вузол черги), що містить окрім даних посилання на вузол, що містить наступний елемент черги.

Лістинг 5.2.2. Реалізація черги як рекурсивної струтктури. Допоміжний клас Node – Вузол черги.

```
class Node:
    """ Допоміжний клас - вузол черги.

    Вузлол зберігає у собі навантаження - певну інформаційну частину
    (іншу структуру або тип даних) - та посилання на наступний вузол
    """

    def __init__(self, item):
        """ Конструктор

        :param item: навантаження вузла
        """
        self.item = item   # поле для зберігання навантаження
        self.next = None   # посилання на наступний вузол черги
```

Використовуючи вищеописаний клас опишемо клас Queue, що є реалізацією черги як рекурсивної структури даних.

Лістинг 5.2.2. Продовження. Реалізація черги як рекурсивної струтктури.

```
class Queue:
    """ Клас, що реалізує чергу елементів
        як рекурсивну структуру """

    def __init__(self):
        """ Конструктор """
        self.front = None  # Посилання на початок черги
        self.back = None   # Посилання на кінець черги

    def empty(self):
        """ Перевіряє чи черга порожня
```

```
        :return: True, якщо черга порожня
        """
        # Насправді досить перевіряти лише одне з полів front або back
        return self.front is None and self.back is None

    def enqueue(self, item):
        """ Додає елемент у чергу (в кінець)

        :param item: елемент, що додається
        :return: None
        """
        new_node = Node(item)      # Створюємо новий вузол черги
        if self.empty():           # Якщо черга порожня
            self.front = new_node  # новий вузол робимо початком черги
        else:
            self.back.next = new_node  # останній вузол черги посилається на новий вузол

        self.back = new_node       # Останній вузол вказує на новий вузол

    def dequeue(self):
        """ Прибирає перший елемент з черги
            Сам елемент при цьому прибирається із черги

        :return: Навантаження голови черги (перший елемент черги)
        """
        if self.empty():
            raise Exception("Queue: 'dequeue' applied to empty container")

        current_front = self.front       # запам'ятовуємо поточну голову черги
        item = current_front.item        # запам'ятовуємо навантаження першого вузла
        self.front = self.front.next     # замінюємо перший вузол наступним
        del current_front                # видаляємо запам'ятований вузол

        if self.front is None:  # Якщо голова черги стала порожньою
            self.back = None    # Черга порожня = хвіст черги теж порожній
        return item
```

Реалізація методів вищеописаного класу дуже подібна до реалізації методів стеку. Зокрема методи `enqueue()` та `dequeue()` концептуально дуже подібні до реалізації методів стеку `push()` та `pop()` відповідно. Проте, звернемо увагу читача на крайові моменти, а саме ситуацію, коли черга порожня у момент додавання нового елементу або стає порожньою внаслідок вилучення елементу. Пропонуємо читачу проаналізувати вищенаведену реалізацію черги та змоделювати процедури додавання та вилучення елементів у черзі.

5.2.2. Черга з двома кінцями

Означення 5.2.2. Черга з двома кінцями (двостороння черга або дек, deque від англ. double ended queue) – динамічна структура даних, елементи якої можуть додаватись як у початок (голову), так і в кінець (хвіст), і вилучатися як з початку, так і з кінця.

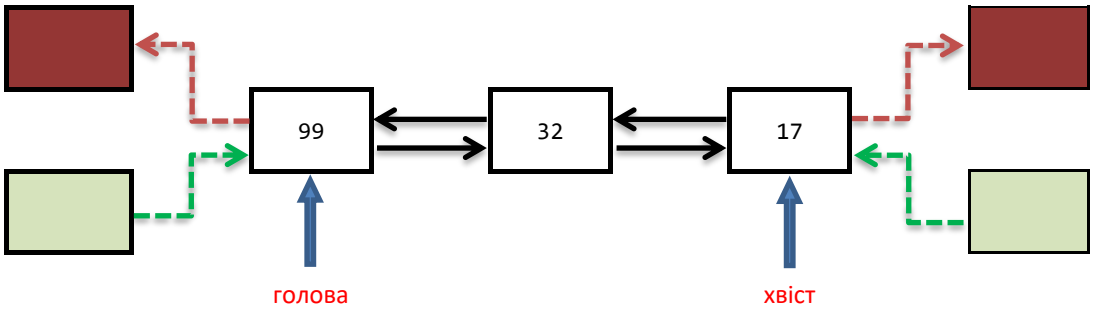


Рисунок 5.2.5. Дек елементів. Схематично позначено можливість додавання та вилучення елементів як у початок, так і в кінець структури даних.

Базові операції для роботи з деком

Класична реалізація деку вимагає опису структури даних, що підтримує такі операції:

1. Створення порожнього деку.

2. Операція **empty**()– визначення, чи є дек порожнім.
3. Операція **appendleft**(item) – додати елемент item до початку дека.
4. Операція **popleft**() – взяти елемент з початку дека.
5. Операція **append**(item) – додати елемент item до кінця дека.
6. Операція **pop**() – взяти елемент з кінця дека.

У нижченаведеній таблиці наведено приклад роботи операцій із двосторонньою чергою. У колонці «Вміст деку після здійснення операції» напівжирним шрифтом виділено елемент, що є головою деку, а підкресленням – елемент, що знаходиться у його хвості.

Операція над деком	Вміст деку після здійснення операції	Значення, що повертається в результаті здійснення операції
deque = **Deque**()	[]	
deque.**empty**()	[]	**True**
deque.**append**(32)	[**32**]	
deque.**append**(17)	[**32**, 17]	
len(deque)	[**32**, 17]	2
deque.**empty**()	[**32**, 17]	**False**
deque.**appendleft**(99)	[**99**, 32, 17]	
deque.**appendleft**(57)	[**57**, 99, 32, 17]	
deque.**pop**()	[**57**, 99, 32]	17
deque.**popleft**()	[**99**, 32]	57

Реалізація на базі вбудованого списку

Опишемо клас Deque, що є реалізацією деку на базі вбудованого списку.

Лістинг 5.2.3. Реалізація деку на базі вбудованого списку.

```
class Deque:
    def __init__(self):
        """ Конструктор деку
        Створює порожній дек.
        """
        self.items = []    # Список елементів деку

    def empty(self):
        """ Перевіряє чи дек порожній

        :return: True, якщо дек порожній
        """
        return len(self.items) == 0

    def append(self, item):
        """ Додає елемент у кінець деку

        :param item: елемент, що додається
        :return: None
        """
        self.items.append(item)

    def pop(self):
        """ Повертає елемент з кінця деку.

        :return: Останній елемент у деку
        """
        if self.empty():
            raise Exception("Deque: 'popBack' applied to empty container")
        return self.items.pop()

    def appendleft(self, item):
        """ Додає елемент до початку деку
```

```
        :param item: елемент, що додається
        :return: None
        """
        self.items.insert(0, item)

    def popleft(self):
        """ Повертає елемент з початку деку.

        :return: Перший елемент у деку
        """
        if self.empty():
            raise Exception("Deque: 'popFront' applied to empty container")
        return self.items.pop(0)

    def __len__(self):
        """ Розмір деку

        :return: Кількість елементів у деку
        """
        return len(self.items)
```

Перевірку роботи деку, описаного вище, можна провести таким чином

Лістинг 5.2.3. Продовження. Виклик деку.

```
if __name__ == "__main__":
    D = Deque()       # Створюємо новий дек
    D.append(32)      # Додаємо елемент 32 у кінець деку
    D.append(17)      # Додаємо елемент 17 у кінець деку
    D.appendleft(99)  # Додаємо елемент 99 у початок деку
    D.appendleft(57)  # Додаємо елемент 57 у початок деку

    print(D.pop())      # Виштовхуємо останній елемент (17) з деку
    print(D.popleft())  # Виштовхуємо перший елемент (57) з деку
```

Реалізація двосторонньої черги як рекурсивної структури

Вищенаведена реалізація деку на базі вбудованого списку хоча і є достатньо простою, проте суперечить загальноприйнятому правилу, про те що додаватися та вилучатися елементи з деку мають за сталий час. Це стосується операцій `appendleft()` та `popleft()`, час виконання яких є лінійним за кількістю елементів у деку. Тому, аналогічно до черги, реалізацію деку здійснюють рекурсивним чином.

Як і у випадку зі стеком, та чергою, дек використовує структуру даних – Node (вузол деку). Проте, на відміну від вищезгаданих структур, вузол деку, окрім посилання на наступний вузол, містить ще й посилання на попередній вузол.

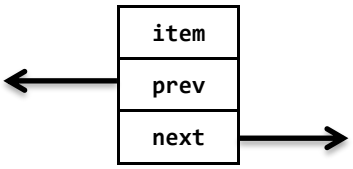


Рисунок 5.2.6. Вузол деку – містить посилання на попередній та наступний елементи деку.

Лістинг 5.2.4. Реалізація деку на базі вбудованого списку. Допоміжний клас Node – Вузол деку.

```
class Node:
    """ Допоміжний клас - вузол деку """

    def __init__(self, item):
        """ Конструктор вузла деку

        :param item: Елемент деку
        """
        self.item = item  # поле, що містить елемент деку
```

```
        self.next = None  # наступний вузол
        self.prev = None  # попередній вузол
```

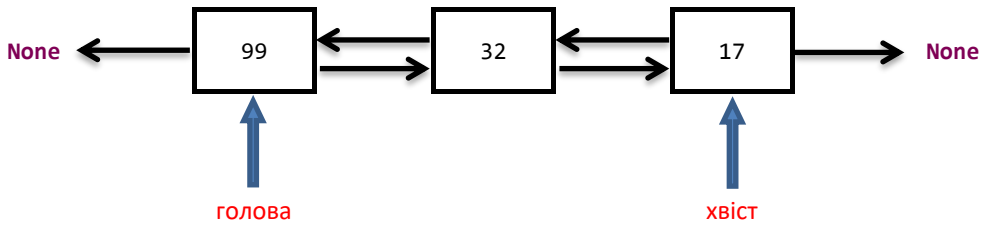
Використовуючи цей клас опишемо клас Deque, що є реалізацією двосторонньої черги як рекурсивної структури даних.

Лістинг 5.2.4. Продовження. Реалізація деку як рекурсивної струткткури.

```
class Deque:
    """ Реалізує дек як рекурсивну структуру. """

    def __init__(self):
        """ Конструктор деку - Створює порожній дек.
        """
        self.front = None  # Посилання на перший елемент деку
        self.back = None   # Посилання на останній елемент деку

    def empty(self):
        """ Перевіряє чи дек порожній

        :return: True, якщо дек порожній
        """
        return self.front is None and self.back is None

    def appendleft(self, item):
        """ Додає елемент до початку деку

        :param item: елемент, що додається
        :return: None
        """
        node = Node(item)           # створюємо новий вузол деку
        node.next = self.front      # наступний вузол для нового - елемент, що є першим
        if not self.empty():        # якщо додаємо до непорожнього деку
            self.front.prev = node  # новий вузол стає попереднім для першого
        else:
            self.back = node  # якщо додаємо до порожнього деку, новий вузол є останнім
        self.front = node     # новий вузол стає першим у деку

    def popleft(self):
        """ Повертає елемент з початку деку.

        :return: Перший елемент у деку
        """
        if self.empty():
            raise Exception('pop_front: Дек порожній')
        node = self.front       # node - перший вузол деку
        item = node.item        # запам'ятовуємо навантаження
        self.front = node.next  # першим стає наступний вузлом деку
        if self.front is None:  # якщо в деку був 1 елемент
            self.back = None    # дек стає порожнім
        else:
            self.front.prev = None  # інакше перший елемент посилається на None
        del node                    # Видаляємо вузол
        return item

    # методи append та pop повністю симетричні appendleft та popleft відповідно
    def append(self, item):
        """ Додає елемент у кінець деку

        :param item: елемент, що додається
        :return: None
        """
        elem = Node(item)
        elem.prev = self.back
        if not self.empty():
            self.back.next = elem
        else:
```

```
            self.front = elem
        self.back = elem

    def pop(self):
        """ Повертає елемент з кінця деку.

        :return: Останній елемент у деку
        """
        if self.empty():
            raise Exception('pop_back: Дек порожній')
        node = self.back
        item = node.item
        self.back = node.prev
        if self.back is None:
            self.front = None
        else:
            self.back.next = None
        del node
        return item

    def __del__(self):
        """ Деструктор - використовується для коректного видалення
            усіх елементів деку у разі видалення самого деку

        :return: None
        """
        while self.front is not None:  # проходимо по всіх елементах деку
            node = self.front  # запам'ятовуємо посилання на елемент
            self.front = self.front.next  # переходимо до наступного елементу
            del node  # видаляємо елемент
        self.back = None
```

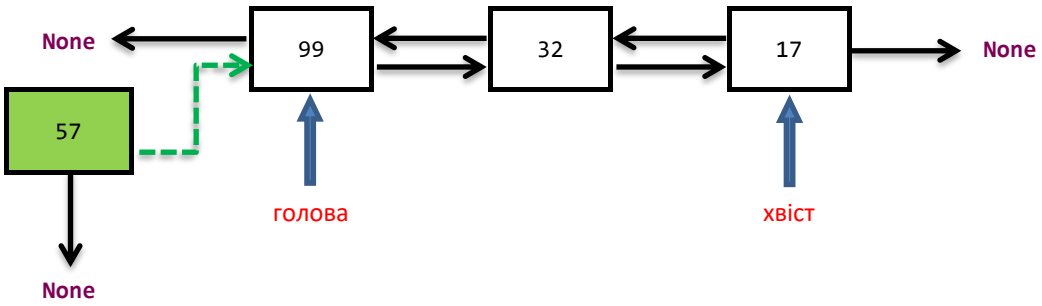
Пояснимо операцію **appendleft()**. Інші операції пропонуємо розібрати читачу самостійно. Отже, розглянемо дек, зображений нижче.



Нехай до нього застосовується операція

```
appendleft(57)  # Додаємо елемент 57 у початок деку
```

Спочатку створюємо новий вузол, у який записуємо число 57, попереднім його вузлом робимо **None**, а наступним – перший елемент деку.



Тепер лишається замінити посилання на попередній елемент для першого елементу деку та змістити голову дека на новий вузол.

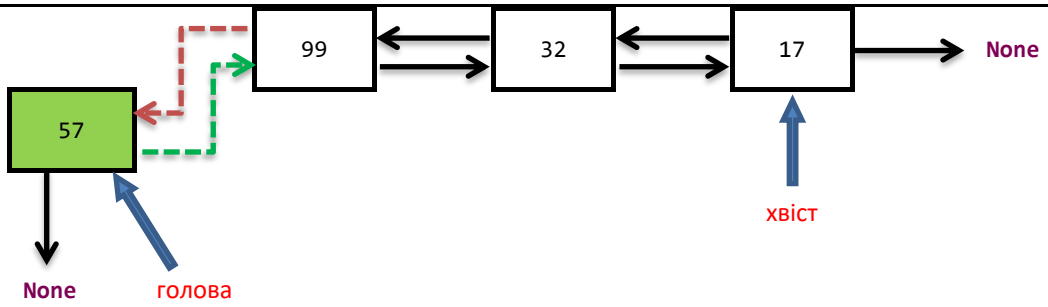


Рисунок 5.2.7. Дек елементів. Додавання елементу у голову деку.

5.2.3. Пріоритетна черга

Пріоритетна черга, взагалі кажучи, не є лінійною структурою даних в повному розумінні цієї концепції, адже для цією структури даних є не принциповим розташування елементу у структурі – важливим є порядок вилучення елементу з черги.

Означення 5.2.3. **Пріоритетна черга** – це динамічна структура даних, що містить елементи разом з додатковою інформацією, що називається ключем. Ключ елементу визначає пріоритет вилучення елементу у порівнянні з іншими елементам, що містяться у черзі.

На наступному рисунку схематично зображено операції додавання нового елементу з ключем 8 до пріоритетної черги та вилучення елементу з найвищим пріоритетом – ключ 2 є найменшим серед ключів елементів що містяться у черзі.

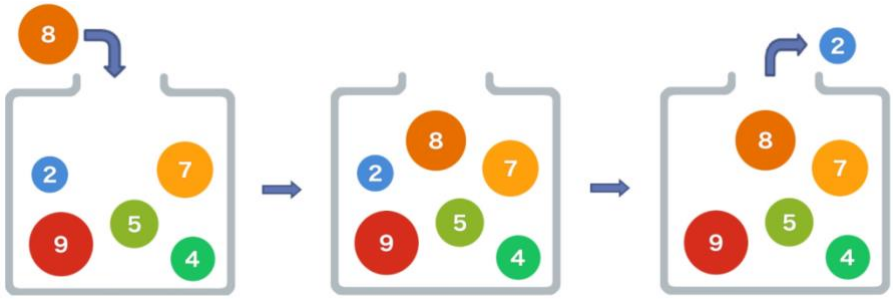


Рисунок 5.2.8. Додавання нового елементу та вилучення елементу з найвищим пріоритетом.

Базові операції для роботи з пріоритетною чергою

Як правило, елементи у пріоритетну чергу додаються у парі з їхнім пріоритетом, тобто у вигляді кортежу `(priority, item)`. Дійсно, ця черга відрізняється від стандартної тим, що елементи вибираються з неї не в порядку якому вони додавалися, а згідно з їхнім пріоритетом. Над пріоритетною чергою допустимі такі операції

1. Створити чергу.
2. Операція **`empty`**`()` – чи порожня черга.
3. Операція **`insert`**`(priority, item)` – вставити пару `(priority, item)` у чергу.
4. Операція **`extractMinimum`**`()` – отримати пару з найвищим пріоритетом. Вважається, що найвищий пріоритет має елемент у якого `priority` є найменшим.

Наведемо приклад реалізації пріоритетної черги, час додавання елементу до якої має порядок $O(1)$, а час отримання елементу $O(n)$.

Лістинг 5.2.5. Реалізація пріоритетної черги на базі вбудованого списку.

```
class PriorityQueue:
    def __init__(self):
        """ Конструктор """
        self.mItems = []   # Список елементів черги, містить пари (ключ, пріоритет)

    def empty(self):
        """ Перевіряє чи чергра порожня

        :return: True, якщо черга порожня
```

```
        """
        return len(self.mItems) == 0

    def insert(self, key, priority):
        """ Додає елемент у чергу разом з його пріоритетом

        :param key: елемент
        :param priority: пріоритет
        :return: None
        """
        self.mItems.append((key, priority))

    def extractMinimum(self):
        """ Повертає елемент з черги, що має найвищий пріоритет

        :return: елемент з черги з найвищим пріоритетом
        """
        if self.empty():
            raise Exception("PriorityQueue: 'extract_minimum' applied to empty container")

        # шукаємо елемент з найвищим пріоритетом
        # у нашому випадку, той елемент для якого значення priority найменша
        minpos = 0
        for i in range(1, len(self.mItems)):
            if self.mItems[minpos][1] > self.mItems[i][1]:
                minpos = i

        return self.mItems.pop(minpos)[0]
```

Зауваження. Реалізація пріоритетної черги, наведена вище, скоріше має ознайомчий характер. Хоча така реалізація є достатньою для розв'язання багатьох прикладних задач, проте, у випадку, великої кількості даних, продуктивність черги може бути незадовільною. Пізніше ми розглянемо реалізацію пріоритетної черги на базі структури даних двійкова купа. Така реалізація дозволяє здійснювати вставку та отримання елементів з черги за логарифмічний час.

§5.3. Списки

5.3.1. Однозв'язний список

Списки відрізняються від стеків, черг та деків тим, що мають можливість безліч разів проходити вздовж списку, отримувати доступ до будь-якого елемента, не змінюючи сам список. Існує декілька різновидів списків: однозв'язні списки, кільцеві списки, двозв'язні списки. Кожен з цих списків вирізняється переліком базових операцій та внутрішньою структурою.

Означення 5.3.1. Класичний (зв'язний або однозв'язний) список – це динамічна структура даних, що складається з елементів (як правило одного типу), пов'язаних між собою у строго визначеному порядку: кожен елемент списку вказує на наступний елемент списку.

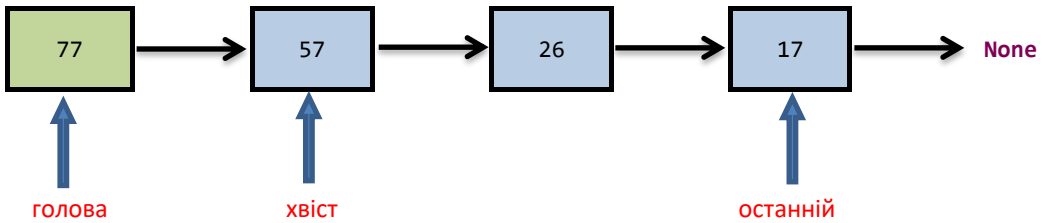


Рисунок 5.3.1. Класичний список

Елемент, на який немає посилання називається **початковим** або **головою** списку. Останній елемент не посилається на жоден елемент списку (тобто посилається на None). **Хвостом** списку будемо називати список, що складається з усіх елементів вихідного списку, крім першого. Фактично хвіст списку це посилання на другий елемент цього списку.

Як бачимо, організація даних однозв'язного списку дуже подібна на таку для черги або стеку з рекурсивною реалізацією. Відмінністю списків від вищезгаданих структур є операції які проводять над ними. У однозв'язному

списку можна пересуватися лише від початку в сторону його кінця, використовуючи операцію отримання хвоста списку.

Базовий набір дій над класичним списком:

Отже, базовий набір дій над однозв'язним списком є таким:

1. Створити список.
2. Операція визначення, чи порожній список.
3. Додати елемент у початок списку.
4. Взяти перший елемент списку (без зміни всього списку).
5. Отримати хвіст списку.

Реалізація зв'язного списку у мові Python

Реалізація зв'язного списку як рекурсивної структури дуже схожа на реалізацію лінійних структур наведених вище. Як у випадку зі стеком та чергою, рекурсивна реалізація використовує структуру даних – Node (вузол), що містить окрім даних посилання на вузол, що містить наступний елемент списку.

Лістинг 5.3.1. Допоміжний клас Вузол списку.

```
class Node:
    """ Допоміжний клас - вузол списку. """

    def __init__(self, item):
        """ Конструктор
        :param item: навантаження вузла
        """
        self.mItem = item   # навантаження вузла
        self.mNext = None   # посилання на наступний вузол списку
```

Реалізація списку вимагає описати методи, що реалізують базовий набір роботи зі списком.

Лістинг 5.3.2.Реалізація зв'язного списку.

```
class LinkedList:

    def __init__(self):
        """ Конструктор - створює порожній зв'язний список """
        self.mFirst = None

    def empty(self):
        """ Перевіряє чи список є порожнім
        :return: True, якщо список порожній
        """
        return self.mFirst is None

    def insert(self, item):
        """ Вставляє заданий елемент у початок списку
        :param item: елемент для вставки
        """
        node = Node(item)         # створюємо новий елемент списку
        node.mNext = self.mFirst  # наступний елемент для нового - це елемент, який є першим
        self.mFirst = node        # новий елемент стає першим у списку

    def head(self):
        """ Повертає навантаження голови списку
        :return: навантаження голови списку або None, якщо список порожній
        """
        if self.empty():
            return None
        else:
            return self.mFirst.mItem

    def tail(self):
        """ Повертає хвіст списку
```

```
        :return: хвіст списку
        """
        if self.empty():
            raise Exception("LinkedList: 'tail' applied to empty container")
        self.mFirst = self.mFirst.mNext
        return self
```

5.3.2. Список із поточним елементом

Означення 5.3.2. Список із поточним елементом – різновид класичного списку – динамічна структура даних, що складається з елементів (як правило одного типу), пов'язаних між собою, і структури керування, що вказує на поточний елемент структури.

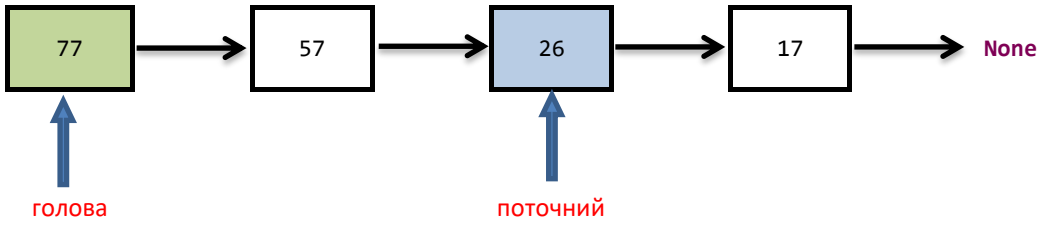


Рисунок 5.3.2. Список з поточним елементом

Аналогічно до однозв'язного списку, у списку з поточним елементом можна пересуватися лише у одному напрямку – під початку до кінця списку. Проте на відміну від однозв'язного списку, де така операція здійснювалася через операцію визначення хвоста списку, тут використовується поточний елемент.

Базовий набір дій над списком з поточним елементом:

1. Почати роботу.
2. Операція визначення, чи порожній список.
3. Зробити поточним перший елемент списку.
4. Перейти до наступного елемента.
5. Отримати поточний елемент (список при цьому не змінюється).
6. Вставити новий елемент у список після поточного.

Реалізація списку з поточним елементом у мові Python

Аналогічно до наведеної вище реалізації зв'язного списку, реалізація списку з поточним елементом використовує допоміжний клас Node, описаний у лістингу 5.3.1.

Лістинг 5.3.3.Реалізація зв'язного списку.

```
class ListWithCurrent:

    def __init__(self):
        """ Конструктор - створює новий порожній список.
        """
        self.mHead = None  # Перший вузол списку
        self.mCurr = None  # Поточний вузол списку

    def empty(self):
        """ Перевіряє чи список порожній

        :return: True, якщо список не містить жодного елемента
        """
        return self.mHead is None

    def reset(self):
        """ Зробити поточний елемент першим."""
        self.mCurr = self.mHead

    def next(self):
        """ Перейти до наступного елемента.
```

```
        Породжує виключення StopIteration, якщо наступний елемент порожній
        :return: None
        """
        if self.empty() or self.mCurr.mNext is None:
            raise StopIteration
        else:
            self.mCurr = self.mCurr.mNext

    def current(self):
        """ Отримати поточний елемент

        :return: Навантаження поточного елементу
        """
        if self.empty():
            return None
        else:
            return self.mCurr.mItem

    def insert(self, item):
        """ Вставити новий елемент у список після поточного

        :param item: елемент, що вставляється у списоск
        :return: None
        """
        node = Node(item)
        if self.empty():
            self.mHead = node
            self.mCurr = node
        else:
            node.mNext = self.mCurr.mNext
            self.mCurr.mNext = node
```

Отже, тепер, для створення списку скористаємося командою:

```
l = ListWithCurrent()
```

а для додавання елементів, відповідно

```
l.insert(11)
l.insert(12)
l.insert(13)
l.insert(14)
l.insert(15)
l.insert(16)
```

Для зручності визначимо у класу спеціальний метод, для зручного доступу до даних елементів, а саме

```
    def __str__(self):
        return str(self.current())
```

Тепер, для виведення поточного елементу досить скористатися командою

```
print(l)
```

Оскільки список, це колекція, опишемо клас-ітератор для послідовного перебору елементів списку.

Лістинг 5.3.4. Ітератор для зв’язного списку.

```
class ListIterator:
    """ Клас Ітератор """
    def __init__(self, lst):
        """ Конструктор ітератора
        :param lst: список
        """
        self.mCursor = lst.mHead  # поточна позиція ітератора у колекції
```

```
    def __next__(self):
        if self.mCursor == None:
            raise StopIteration
        else:
            curr = self.mCursor.mItem
            self.mCursor = self.mCursor.mNext
            return curr
```

Нарешті доповнимо опис класу `ListWithCurrent` з лістингу 5.3.2 методом, що забезпечує підтримку ітераційного протоколу.

Лістинг 5.3.2. Продовження. Реалізація зв'язного списку.

```
class ListWithCurrent:
    ...
    def __iter__(self):
        """ Спеціальний метод, що повертає ітератор для колекції
        :return: Ітератор колекції
        """
        return ListIterator(self)
```

Після цього можемо скористатися звичайним циклом по колекції для виведення всіх елементів списку

```
for el in l:
    print(el)
```

або що те ж саме

```
it = iter(l)
while True:
    try:
        print(next(l))
    except StopIteration:
        break
```

5.3.3. Кільцевий список

Означення 5.3.3. **Кільцевий список** – різновид списку з поточним елементом, для якого не визначено перший та останній елементи. Усі елементи зв'язані у кільце, відомий лише порядок слідування.

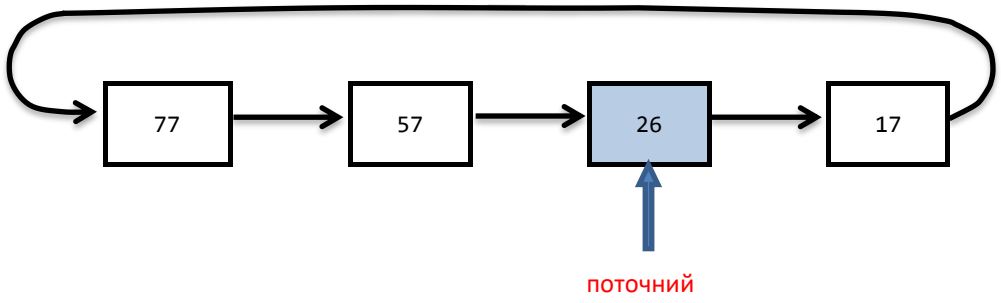


Рисунок 5.3.3. Кільцевий список

Базовий набір дій над кільцевими списками:

1. Почати роботу.
2. Операція визначення, чи порожній список.
3. Перейти до наступного елемента.
4. Отримати поточний елемент (список при цьому не змінюється).
5. Вставити новий елемент у список після поточного.

Реалізація у мові Python

Лістинг 5.3.5. Реалізація кільцевого списку.

```
class CircularLinkedList:

    def __init__(self):
        """ Конструктор - створює новий порожній список.
        """
        self.mCurr = None  # Поточний вузол списку

    def empty(self):
        """ Перевіряє чи список порожній

        :return: True, якщо список не містить жодного елемента
        """
        return self.mCurr is None

    def next(self):
        """ Перейти до наступного елемента.

        Породжує виключення StopIteration, якщо наступний елемент порожній
        :return: None
        """
        if self.empty():
            raise StopIteration
        else:
            self.mCurr = self.mCurr.mNext

    def current(self):
        """ Отримати поточний елемент

        :return: Навантаження поточного елементу
        """
        if self.empty():
            return None
        else:
            return self.mCurr.mItem

    def insert(self, item):
        """ Вставити новий елемент у список перед поточним

        :param item: елемент, що вставляється у спиоск
        :return: None
        """

        node = Node(item)
        if self.empty():
            node.mNext = node
            self.mCurr = node
        else:
            node.mNext = self.mCurr.mNext
            self.mCurr.mNext = node
```

Як бачимо, для усіх наведених вище списків, що базуються на однозв’язних списках серед переліку базових операцій відсутні операції видалення елементів, наприклад, видалення поточного елементу списку. Пояснюється це тим, що такі операції здебільшого вимагають пошуку попереднього елементу списку по відношенню до поточного для зміни посилання на наступний елемент, що, у свою чергу, вимагає проходу по всьому списку. Відповідно складність операції видалення у однозв’язних списках є лінійною, що для багатьох задач неприпустимо. Для списків, що будуть розглянуті нижче операції видалення будуть входити до базового набору операцій, крім того, такі операції будуть мати сталий час виконання.

5.3.4. Двозв'язний список

Означення 5.3.4. **Двобічно зв'язаний** (двозв'язний) **список** – динамічна структура даних, що складається з елементів одного типу, зв'язаних між собою у строго визначеному порядку. При цьому визначено перший та останній елементи у списку, а кожен елемент списку вказує на наступний і попередній елементи у списку. Перший

елемент має попереднім елементом посилання на невизначений елемент **None**. Аналогічно, **None** буде наступним елементом для останнього елементу списку.

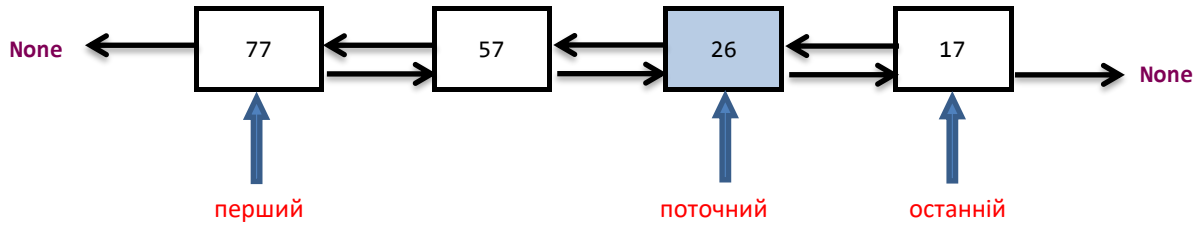


Рисунок 5.3.4. Двозв'язний список

Додатково для двозв’язного списку визначається поточний елемент. Рух по списку здійснюється за рахунок пересування поточного елементу на попередній або наступний елемент списку.

Базовий набір дій над двозв'язними списками:

1. Створити список.
2. Операція визначення, чи порожній список. Повертає булеве значення.
3. Зробити поточними перший елемент списку.
4. Зробити поточними останній елемент списку.
5. Перейти до наступного елемента.
6. Перейти до попереднього елемента.
7. Отримати поточний елемент.
8. Вставити новий елемент перед поточним.
9. Вставити новий елемент після поточного.
10. Видалити поточний елемент.

Реалізація у мові Python

Лістинг 5.3.6. Допоміжний клас Вузол двозв’язного списку.

```
class Node:
    """ Допоміжний клас - вузол двобічно зв'язаного списку """

    def __init__(self, item):
        """ Конструктор вузла

        :param item: Елемент списку
        """
        self.mItem = item  # дані, що пов'язані з вузлом деку
        self.mNext = None  # наступний вузол
        self.mPrev = None  # попередній вузол
```

Лістинг 5.3.7. Реалізація двозв’язного списку.

```
class DoublyLinkedList:
    """ Двобічно зв'язаний список. """

    def __init__(self):
        """ Конструктор списку - створює порожній список.
        """
        self.mFirst = None  # Перший вузол списку
        self.mLast = None   # Останній вузол списку
        self.mCurr = None   # ПОточний вузол списку

    def empty(self):
        """ Перевіряє чи список порожній
        :return: True, якщо список порожній
        """
        return self.mFirst is None
```

```
    def setFirst(self):
        """ Зробити поточними перший елемент списку """
        self.mCurr = self.mFirst

    def setLast(self):
        """ Зробити поточними останній елемент списку """
        self.mCurr = self.mLast

    def next(self):
        """ Перейти до наступного елемента """
        if self.mCurr != self.mLast:
            self.mCurr = self.mCurr.mNext

    def prev(self):
        """    Перейти до попереднього елемента """
        if self.mCurr != self.mFirst:
            self.mCurr = self.mCurr.mPrev

    def current(self):
        """ Отримати поточний елемент
        :return: Навантаження поточного вузла
        """
        if self.mCurr is not None:
            return self.mCurr.mItem
        else:
            return None

    def insertBefore(self, item):
        """ Вставити новий елемент перед поточним
        поточний елемент залишається на місці
        :param item: елемент для вставки у список
        :return: None
        """
        node = Node(item)  # створюємо вузол, для нового елементу списку
        if self.empty():                # вставка у порожній список
            self.mFirst = self.mLast = self.mCurr = node
        else:
            node.mNext = self.mCurr
            if self.mCurr == self.mFirst:  # вставка перед першим елементом
                self.mFirst = node
            else:                          # вставка всередині списку
                node.mPrev = self.mCurr.mPrev
                self.mCurr.mPrev.next = node
            self.mCurr.mPrev = node

    def insertAfter(self, item):
        """ Вставити новий елемент після поточного
        елемент, що був вставлений стає поточним
        :param item: елемент для вставки у список
        :return: None
        """
        node = Node(item)  # створюємо вузол, для нового елементу списку
        if self.empty():  # вставка у порожній список
            self.mFirst = self.mLast = self.mCurr = node
        else:
            node.mPrev = self.mCurr
            if self.mCurr == self.mLast:  # вставка перед першим елементом
                self.mLast = node
            else:                        # вставка всередині списку
                node.mNext = self.mCurr.mNext
                self.mCurr.mNext.prev = node

            self.mCurr.mNext = node
            self.mCurr = node  # елемент, що був вставлений стає поточним

    def remove(self):
        """ Видалити поточний елемент зі списку """
        if self.empty():
            raise Exception("DoublyLinkedList: 'remove' applied to empty list")
```

```
        node = self.mCurr  # Запам'ятовуємо поточний вузол

        if node == self.mFirst:  # якщо поточний вузол перший у списку
            self.mFirst = node.mNext
        else:
            node.mPrev.mNext = node.mNext

        if node == self.mLast:  # якщо поточний вузол останній у списку
            self.mCurr = self.mLast = node.mPrev
        else:
            node.mNext.mPrev = node.mPrev
            self.mCurr = node.mNext

        del node  # видалення вузла
```

РОЗДІЛ 6. ДЕРЕВА

Розглянуті у попередньому розділі структури даних називаються лінійними, оскільки (навіть у випадку пріоритетної черги) без додаткових зусиль, можна визначити чіткий порядок елементів у кожній з цих структури даних, а для кожного елементу такої структури можна визначити його лівого та правого сусіда. Об'єктом розгляду цього та наступного розділів є структури даних для яких лінійний порядок елементів визначити не можна, а кожен елемент матиме не двох (лівого та правого, як в лінійних структурах даних) сусідів, а цілий список сусідів. До таких структур даних належать дерева і графи. Ці структури даних іноді називають плоскими, оскільки, якщо лінійні структури даних можна зобразити у вигляді певного ланцюжка, то такі структури даних часто схематично зображують у вигляді розгалуженої структури на площині. Потреба у застосуванні цих структур даних природньо виникає у багатьох областях математики, комп'ютерних науках та програмуванні.

У цьому розділі розглядаються структури даних дерева. Ми вивчимо їхнє моделювання за допомогою комп'ютера, а також застосування у прикладних задачах.

Дерево це ієрархічна структура деякої сукупності елементів. Зауважимо, що з деревами ми вже неявно зустрічалися у попередніх розділах цього курсу. Наприклад, виклик рекурсивної підпрограми можна зобразити у вигляді дерева.

Термін дерево вибраний не випадкового, адже ця структура даних має багато спільного з деревами рослинного світу – дерево має корінь, гілки та листя. Єдиною суттєвою відмінністю є те, що дерева у програмуванні завжди схематично зображують так, що корінь розташовується вгорі, а гілки йдуть донизу. На рисунку нижче зображено дерево файлової системи операційної системи Windows. Диск «Win10(C:)» є коренем дерева файлової системи. Файли, що знаходяться у папках, це листя, а відношення, що встановлюють яка папка міститься у якій – це гілки дерева файлової системи.

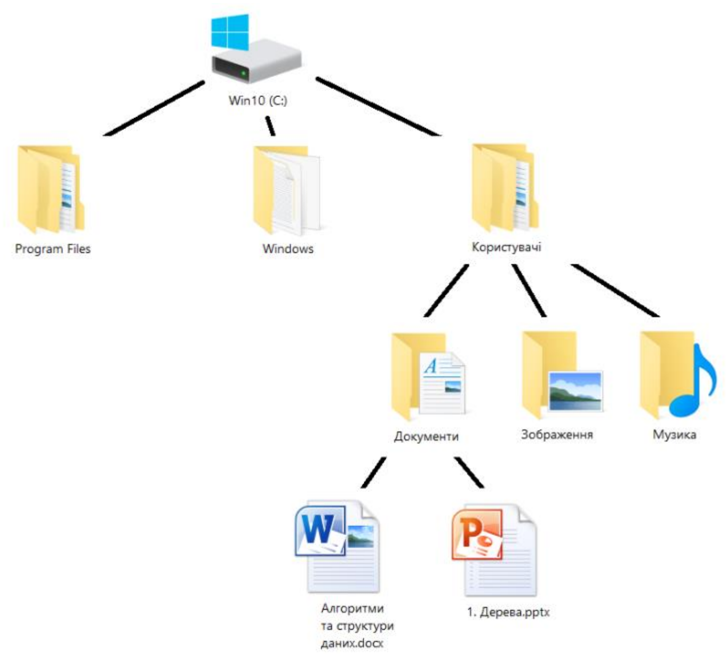


Рисунок 6.0.1. Дерево файлової системи.

§6.1. Означення, приклади та реалізація у Python

6.1.1. Основні означення

Наведемо означення дерева та вивчимо інші означення пов'язані з ним, що будуть траплятися у цьому курсі. Отже

Означення 6.1.1. Дерево – це сукупність вузлів і відношень, що утворює ієрархічну структуру.

Вузол (або вершина) дерева – це математична абстракція, яка моделює абстрактні об'єкти чи об'єкти реального світу. Наприклад, для дерева файлової системи зображеної на рисунку 6.0.1, вузлами є папки і файли

файлової системи. Вузол, як правило, має ім'я, яке називається його **ключем** або **ідентифікатором**. У вищенаведеному прикладі, таким ідентифікатором є повний шлях до файлу або папки (разом з іменем).

Навантаження вузла – це додаткова інформація, що міститься у вузлі дерева та не впливає на його структуру. Для дерева файлової системи зображеної на рисунку 6.0.1 таким навантаженням може бути час створення папки/файлу, розмір тощо.

Відношення визначають зв'язки між вершинами типу батько-дитина. Ці зв'язки називаються **гілками (ребрами) дерева**. Відношення на вищенаведеному дереві файлової системи – це зв'язок, що вказує яка папка/файл у якій міститься. Наприклад, папка «Користувачі» міститься у кореневій папці «Win10(C:)». У цьому випадку кажуть, що папка «Win10(C:)» є **батьком** папки «Користувачі», а відповідно папка «Користувачі» є **дитиною** (сином, дочкою, тощо) папки «Win10(C:)».

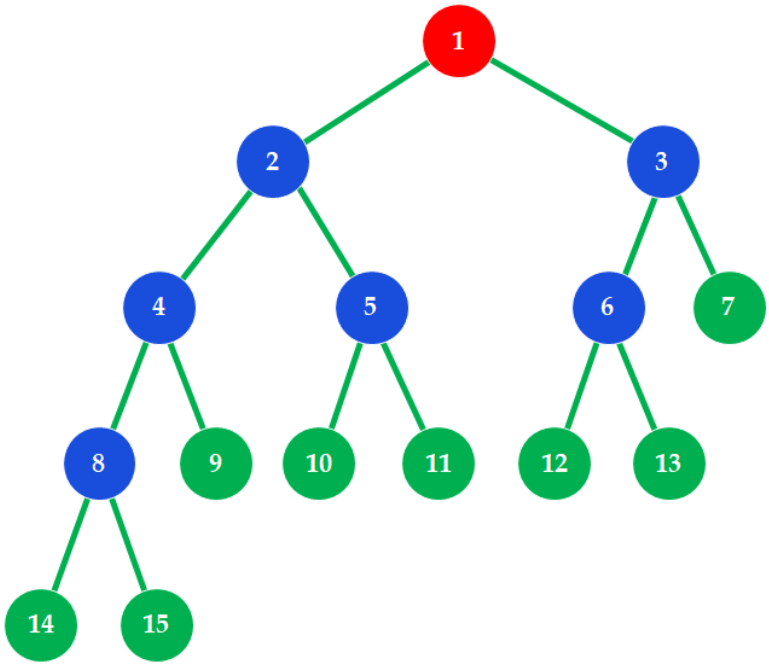


Рисунок 6.1.1. Приклад дерева.

Як ми вже знаємо, вузол, що немає батька називається **коренем дерева**. Вузол, що немає дітей називається **листком дерева**. Дерево, що складається лише з одного кореня називається **пеньком**. Вузол, що не є коренем або листком називається внутрішнім вузлом дерева. На рисунку 6.1.1 зображено абстрактне дерево, вузли якого мають ідентифікаторами числа від 1 до 15. Вузол 1 є коренем цього дерева. Вузли 7, 9, 10, 11, 12, 13, 14, 15 є листками цього дерева. Вузли 2, 3, 4, 5, 6, 8 є внутрішніми вузлами.

Степінь вузла – кількість дітей, що має вузол. Всі вершини дерева на рисунку 6.1.1, крім ти, що є листками мають степінь 2.

Брати (сестри) – вузли одного батька. На рисунку наведеному вище вузли 6 та 7 є братами.

Нащадок – вузол, що досяжний послідовними переходами від батька до дитини.

Піддерево – частина дерева, що утворює дерево. Кожен вузол дерева (разом з усіма його вузлами-нащадками) визначає **піддерево**, для якого цей вузол є коренем. Наприклад, на рисунку 6.1.1 вузол 3 дерева визначає піддерево, до якого входять вузли 6, 7, 12 та 13.

Предок – вузол, досяжний послідовними переходами від дитини до батька. Наприклад, вузол 2 є предком вузла 14, відповідно вузол 14 є нащадком вузла 2. При цьому вузол 2 не є предком вузла 12.

Шлях – послідовність вершин і ребер, що з'єднують вузол з нащадком. На рисунку вище прикладом шляху, що з'єднує вершини 1 та 11 є послідовність 1 – 2 – 5 – 11. **Довжиною шляху** називається кількість ребер у шляху. Очевидно, що довжина шляху між вузлами 1 та 11 дорівнює 3.

Рівень вузла – кількість ребер, що з'єднують вузол з коренем. Корінь дерева знаходиться на нульовому рівні. Для дерева зображеного на рисунку 6.1.1 вузли 2 та 3 знаходяться на першому рівні, а вузол 14 та 15 на 4 рівні.

Висота дерева – кількість ребер найдовшого шляху між коренем і листом. Висота дерева зображеного на рисунку 6.1.1 є 4 – це довжина шляху від кореня до листка 14 (або 15).

Отже, формально дерево можна визначити рекурсивно, таким чином

Означення 6.1.2.

1) Один вузол дерева є деревом. Цей вузол також є коренем цього дерева.

2) Нехай n – вузол, а $T_1, \dots, T_k$ – дерева з коренями відповідно $n_1, \dots, n_k$. Тоді структура утворена таким чином, що для вузла n, вузли $n_1, \dots, n_k$ є його синами також утворює (нове) дерево. При цьому n є коренем нового дерева, а $T_1, \dots, T_k$ – його піддерева.

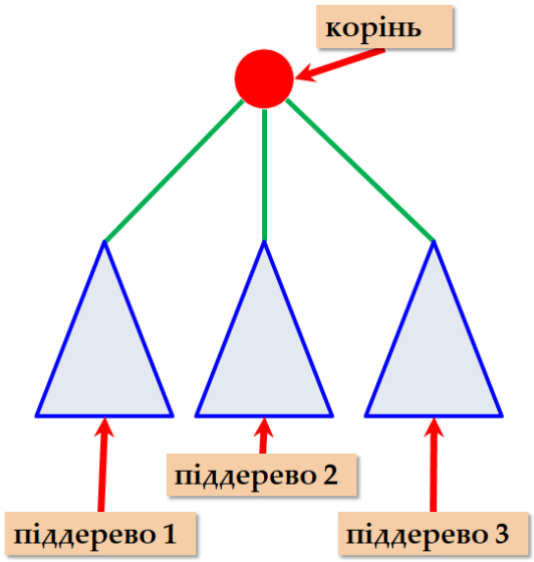


Рисунок 6.1.2. Резурсивне зображення дерева.

Отже, підсумовуючи вищенаведене, можемо сказати, що дерево складається з набору вузлів і набору гілок, що поєднують пари вузлів. При цьому дерево має такі властивості:

- Один з вузлів дерева визначено як його корінь.
- Кожен вузол n (крім кореня) з'єднується гілкою з єдиним іншим вузлом n_p – батьком вузла n.
- Існує лише один шлях, який кожен вузол з'єднується з коренем.

Операції з деревами

Дерева значно складніші структури, у порівнянні з лінійними структурами даних, такими як черга чи стек. Крім того існує багато різних їхніх типів призначених для застосування для тих чи інших практичних задач. Саме тому у мовах програмування не визначеного вбудованих або бібліотечних типів даних, що у загальному випадку реалізують деревовидні структури – у кожній конкретній ситуації деревовидна структура реалізується розробником самостійно, залежно від поставленої задачі. Аналогічно, для дерев в загальному випадку не визначають жодних обов'язкових операцій – намір операцій визначається поставленою задачею і реалізація кожної такої операції повністю покладається на розробника.

Упорядкування вузлів дерева

Дерево називається **впорядкованим**, якщо набір дітей кожного його вузла є впорядкований. Інакше, дерево називається **не впорядкованим**. Діти вузла впорядкованого дерева, як правило впорядковуються зліва на право, якщо інакший порядок окремо не зазначений. Впорядкування дерев використовується для співставлення вузлів, що не пов'язані співвідношенням типу батько-син. Наприклад, на рисунку нижче два впорядкованих дерева вважаються різними, оскільки є різним порядок їхніх дітей: у першому випадку 2, 3, а у другому – 3, 2.

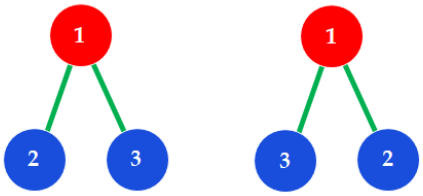


Рисунок 6.1.3. Приклад дерева.

6.1.2. Реалізація у Python

Зображення дерева у вигляді списка списків.

Найпростіший спосіб зображення дерев у Python, використання вбудованих списків (або словників). Ідея такого зображення полягає у тому, що у списку на нульовій позиції будемо зберігати корінь дерева, а на наступних позиціях – послідовність його дітей (тобто послідовність списків, що визначають відповідні піддерева).

Розглянемо дерево зображене на рисунку 6.1.1. За допомогою списків його можна зобразити у пам'яті комп'ютера у такому вигляді

Лістинг 6.1.1. Зображення дерева у вигляді списку списків.

```
tree = [1,           # корінь дерева рівень 0
         [2,           # рівень 1
           [4,           # рівень 2
             [8,           # рівень 3
               [14], [15]]   # рівень 4
             [9]],         # рівень 3
           [5,           # рівень 2
             [10], [11]]],  # рівень 3
         [3,           # рівень 1
           [6,           # рівень 2
             [12], [13]],  # рівень 3
           [7]]]         # рівень 2
```

Тепер, якщо необхідно взяти певний вузол дерева, то можна явно його вибрати використовуючи відповідну індексацію. Наприклад, щоб дістатися до вузла з ключем 11 можна діяти так

```
n = tree[1][2][2][0]  # Звернення до вузла 11
```

Хоча такий звернення до об'єктів дерева є швидким, проте, він є досить громіздким та неочевидним. У наступному пункті розглянемо інший спосіб зображення дерев, який значно зручніший з точки зору наочності і при цьому майже не поступається у швидкодії наведеному вище.

Зображення дерева у вигляді рекурсивної структури.

Спосіб зображення дерев наведений вище є чи не найзручнішим, коли потрібно зберегти або передати дані (чисельні або текстові), що мають деревовидну структуру. Наприклад, один з популярних зараз форматів даних JSON, використовує подібний принцип. Проте, коли мова доходить до обробки таких даних у власній програмі, то часто виявляється, що значно зручніше спочатку перетворити такі дані до іншого вигляду, що має рекурсивну структуру. Таке зображення фактично базується на означення 6.1.2 – деревом буде вузол (кореневий), що містить список вузлів-дітей – кожен з яких є піддеревом. Отже, схематично, вузол дерева буде мати таку структуру

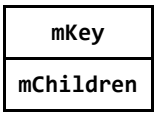


Рисунок 6.1.4. Вузол дерева – містить посилання на список дітей.

Опишемо клас `Tree`, що реалізує такий вузол. Реалізуємо для цього вузла такий набір операцій

1. Створення дерева (виклик конструктора).
2. Метод **`setKey`**`(key)` – встановлює для вершини ключ.
3. Метод **`key`**`()` – повертає ключ поточного вузла.
4. Метод **`addChild`**`(child)` – додає до списку дітей поточного вузла новий вузол `child`.
5. Метод **`removeChild`**`(key)` – видаляє зі списку дітей поточного вузла вузол, що має ключ `key`.
6. Метод **`getChildren`**`()` – повертає список дітей поточного вузла.
7. Метод **`getChild`**`(key)` – повертає вузол, що має ключ `key`, якщо він міститься у списку серед дітей заданого вузла

Для зручності, здійснимо опис класу `Tree` у два кроки:

- Клас `Node`, що містить лише інформацію про ключ вузла та методи роботи з ним.
- Клас `Tree`, що є нащадком класу `Node` і інкапсулює решту методів роботи з деревом.

Лістинг 6.1.2. Опис класу Node.

```
class Node:
    """ Клас, що реалізує вузол дерева """

    def __init__(self, key):
        """ Конструктор - створює вузол дерева
        :param key: ключ вузла, що створюється
        """
        self.mKey = key

    def setKey(self, key):
        """ Встановлює ключ для вузла
        :param key: нове значення ключа
        """
        self.mKey = key

    def key(self):
        """ Повертає ключ вузла
        :return: ключ вузла
        """
        return self.mKey

    def __str__(self):
        """ Повертає ключ вузла.
        :return: рядок, у вигляді "key"
        """
        return str(self.mKey)
```

Тепер, реалізуємо клас Tree, як нащадок класу Node

Лістинг 6.1.3. Зображення дерева у рекурсивному вигляді.

```
class Tree(Node):
    """ Клас, що реалізує структуру даних дерево """

    def __init__(self, key):
        """ Конструктор - створює вузол дерева
        :param key: ключ вузла, що створюється
        """
        super().__init__(key)
        self.mChildren = []

    def addChild(self, child):
        """ Додає до поточного вузла заданий вузол (разом з відповідним піддеревом)
        :param child: вузол (піддерево), що додається
        """
        self.mChildren.append(child)

    def removeChild(self, key):
        """ Видаляє у поточному вузлі вузол-дитину
        :param key: ключ вузла, що видаляється
        :return: False, якщо вузол не містить дитину заданим ключем
        """
        for child in self.mChildren:
            if child.key() == key:
                self.mChildren.remove(child)
                return True
        return False

    def getChild(self, key):
        """ За заданим ключем, повертає вузол зі списку дітей
        :param key: ключ вузла
        :return: знайдений вузол якщо його знайдено, None у іншому випадку
        """
        for child in self.mChildren:
            if child.key() == key:
```

```
                return child
        return None  # якщо ключ не знайдено

    def getChildren(self):
        """
        Повертає список дітей поточного вузла
        :return: Список дітей
        """
        return self.mChildren
```

Хоча описаний вище клас називається Tree, проте фактично його екземплярами є вузли. Для того, щоб створити повноцінне дерево, потрібно створити всі його вузли та задати відповідно ієрархію. Створимо дерево зображене на рисунку 6.1.1. Створювати його будемо знизу вгору - спочатку листя, потім внутрішні вузли, додаючи до них відповідні піддерева. Корінь створимо останнім і додамо до нього відповідні піддерева.

Нижче наведено відповідний код, що містить пояснення у вигляді коментарів.

Лістинг 6.1.3. Продовження. Побудова дерева.

```
def createSampleTree():
    """
    Створювати дерево будемо знизу вгору - спочатку листя,
    потім внутрішні вузли, додаючи до них відповідні піддерева.
    Корінь створимо останнім і додамо до нього відповідні піддерева.
    """

    # Створимо вузли, що є листям дерева
    node7 = Tree(7)
    node9 = Tree(9)
    node10 = Tree(10)
    node11 = Tree(11)
    node12 = Tree(12)
    node13 = Tree(13)
    node14 = Tree(14)
    node15 = Tree(15)

    # Створимо внутрішні вузли дерева та додаємо до них піддерева
    node8 = Tree(8)
    node8.addChild(node14)
    node8.addChild(node15)

    node4 = Tree(4)
    node4.addChild(node8)
    node4.addChild(node9)

    node5 = Tree(5)
    node5.addChild(node10)
    node5.addChild(node11)

    node2 = Tree(2)
    node2.addChild(node4)
    node2.addChild(node5)

    node6 = Tree(6)
    node6.addChild(node12)
    node6.addChild(node13)

    node3 = Tree(3)
    node3.addChild(node6)
    node3.addChild(node7)

    # Створюємо корінь дерева та додаємо до нього відповідні вузли
    root = Tree(1)
    root.addChild(node2)
    root.addChild(node3)

    return root
```

```
# Головна програма – виклик підпрограми, що створює дерево
if __name__ == "__main__":
    tree = createSampleTree()
```

Тепер, щоб дістатися до вузла 11 можна скористатися описаними методами. Отже

```
node11 = tree.getChild(2).getChild(5).getChild(11)
```

І хоча такий спосіб доступу до відповідного вузла є ще більш громіздким у порівнянні з наведеним у попередньому пункті способом, проте він є цілком наочний. Дійсно, ми рухалися по ієрархії вузлів звертаючись до відповідної гілки, поки нарешті не досягнули своєї мети.

У вищенаведеній реалізації дерева, відношення батько-дитина оформлені таким чином, що кожен вузол містить список дітей. Відтак, оскільки список впорядкована колекція, вищенаведена реалізація дерева реалізує впорядковане дерево. Якщо ж постає задача про реалізацію невпорядкованого дерева, то потрібно в реалізації вузла список, що містить послідовність його дітей, замінити невпорядкованою колекцією, наприклад, словником. Ключем у словнику буде ключ вузла, а значенням – відповідний вузол дитини. Нижче наведено реалізацію невпорядкованого дерева, у якій описані зазначені вище операції. При цьому для всіх методів витримано таку ж сигнатуру, як і в реалізації впорядкованого дерева, для сумісності з наведеними нижче реалізаціями алгоритмів обходу дерев.

Лістинг 6.1.4. Зображення дерева у рекурсивному вигляді – невпорядковане дерево.

```
class UnorderedTree(Node):
    """Клас, що реалізує структуру даних невпорядковане дерево """

    def __init__(self, key):
        """ Конструктор - створює вузол дерева
        :param key: ключ вузла, що створюється
        """

        super().__init__(key) # Виклик конструктора батьківського класу
        self.mChildren = {}   # словник дітей вузла, місить пари {ключ: піддерево}

    def addChild(self, child):
        """ Додає до поточного вузла заданий вузол (разом з відповідним піддеревом)
        :param child: вузол (піддерево), що додається
        """
        self.mChildren[child.key()] = child

    def removeChild(self, key):
        """ Видаляє для поточного вузла, вузол-дитину
        :param key: ключ вузла, що видаляється
        :return: False, якщо вузол не містить дитину заданим ключем
        """
        if key in self.mChildren:
            del self.mChildren[key]
            return True
        else:
            return False

    def getChild(self, key):
        """ За заданим ключем, повертає відповідний вузол-дитину
        :param key: ключ вузла
        :return: знайдений вузол якщо його знайдено, None у іншому випадку
        """
        if key in self.mChildren:
            return self.mChildren[key]
        else:
            return None

    def getChildren(self):
        """ Повертає дітей поточного вузла
        :return: Послідовність дітей
```

```
        """
        return self.mChildren.values()
```

Якщо припустити, що дерево зображене на рисунку 6.1.1 є невпорядкованим, то його створення повністю повторює вищенаведений код для впорядкованого дерева, за виключення того, що замість класу `Tree` буде використовуватися щойно описаний клас `UnorderedTree`.

У прикладах, наведених вище, для того, щоб знайти відповідний вузол ми скористалися тим, що знаємо топологію дерева, оскільки вона зображена на відповідному рисунку. Проте, частіше за все дерева змінюються динамічно і програміст не завжди може передбачити їхню топологію навіть для найпростіших випадків. Тому постає задача у відшуканні відповідних вузлів дерева під час роботи програми. У наступному пункті ми познайомимося з такими алгоритмами та їхніми властивостями.

6.1.3. Алгоритми на деревах

При використанні лінійних структур даних, таких як списки чи черги питання опрацювання всіх їхніх елементів було тривіальним оскільки для елементів такої структури чітко визначений порядок – щоб обійти всю структуру достатньо просто "виймати" по одному елементи і проводити над ними бажані операції. З деревами ситуація значно складніша – кожен вузол дерева має цілий список "послідовників", що є його дітьми. Постає питання, як обійти дерево, так, щоб не забути відвідати жоден вузол.

У цьому пункті ми познайомимося з двома базовими алгоритмами обходу дерев:

- Пошук у глибину;
- Пошук у ширину.

Як бачимо назви цих алгоритмів починаються зі слова "пошук". Проте мова йде не лише про пошук у класичному розумінні, наприклад, знайти серед вузлів дерева вершину, що має певні властивості. Власне це загальноприйнятий термін для обходу дерева, при якому відвідуються всі (або частина) вузлів дерева.

Пошук у глибину

Пошук в глибину (англ. Depth-first search, DFS) є одним з найпростіших, з точки зору реалізації, алгоритмів обходу дерев. Його стратегія полягає у тому, щоб, починаючи з кореня дерева, заглиблюватися у дерево наскільки це можливо, перш ніж здійснити перехід до наступної (сусідньої) вершини.

Знову розглянемо дерево, зображене на рисунку 6.1.1. На рисунку 6.1.5 зображено "занурення" у дерево алгоритму, під час відвідування вершин починаючи з кореня на максимальну глибину, вибираючи кожного разу серед списку дітей вузол, що знаходиться лівіше.

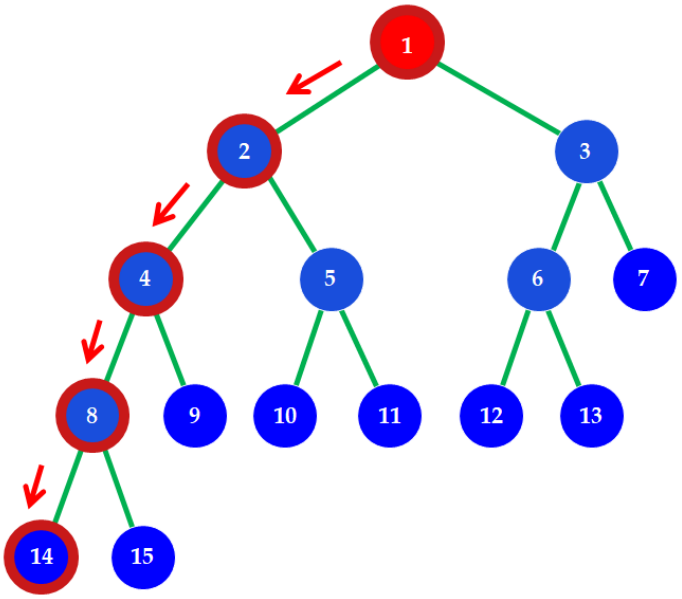


Рисунок 6.1.5. Порядок обходу вершин під час пошуку в глибину.

Таким чином порядок відвідування вершин буде 1 – 4 – 8 – 14. Після того, як буде досягнуто вузол, що немає дітей, відбувається перехід назад до батьківського вузла з якого запускається пошук у глибину для наступного з невідвіданих його дітей – у нашому випадку це буде вершина 15. Процес повторюється доти, доки не будуть відвідані всі вузли дерева.

Наведемо порядок обходу всіх вузлів дерева під час пошуку в глибину. Для наочності зобразимо порядок відвідування на цьому ж дереві, замінивши ключі вершин їхнім порядком відвідування

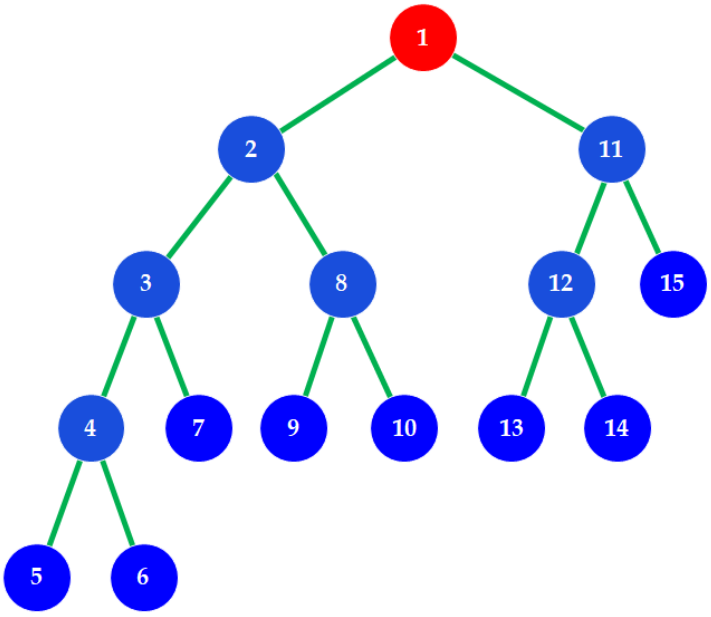


Рисунок 6.1.6. Порядок відвідування вершин під час пошуку в глибину.

Найпростіша реалізація алгоритму пошуку в глибину має рекурсивний вигляд.

Лістинг 6.1.5. Обхід дерева в глибину

```
def DFS(root):
    """ Обхід дерева в глибину
    :param root: корінь дерева з якого починається обхід
    """
    print(root.key(), end=" -> ")  # Опрацьовуємо корінь
    # запускаємо DFS для всіх дітей кореня
    for child in root.getChildren():
        DFS(child)
```

Застосуємо цей алгоритм до нашого дерева, зображеного на рисунку на рисунку 6.1.1. Для його створення скористаємося описаною вище підпрограмою `createSampleTree()`.

Лістинг 6.1.5. Продовження. Застосування обходу в глибину.

```
# Головна програма
if __name__ == "__main__":
    tree = createSampleTree()  # підпрограма створення дерева
    DFS(tree)
```

У результаті роботи програми буде виведено порядок обходу вузлів дерева зображеного на рисунку 6.1.1 під час пошуку в глибину:

```
1 -> 2 -> 4 -> 8 -> 14 -> 15 -> 9 -> 5 -> 10 -> 11 -> 3 -> 6 -> 12 -> 13 -> 7 ->
```

Існує багато модифікацій алгоритму пошуку в глибину залежно від того, у якому порядку опрацьовуються вузли-діти дерева по відношенню до батьківського. У вищенаведеному алгоритмі, ми спочатку опрацьовували корінь, а потім запускали пошук у глибину для його дітей. Проте, можна спочатку запустити процес заглиблення, а вже потім опрацювати корінь.

Лістинг 6.1.6. Обхід дерева в глибину. Спочатку діти, потім корінь.

```
def DFS(root):
    """ Обхід дерева в глибину
    :param root: корінь дерева з якого починається обхід
    """
    # запускаємо DFS для всіх дітей кореня
    for child in root.getChildren():
```

```
        DFS(child)
    print(root.key(), end=" -> ")  # Опрацьовуємо корінь після опрацювання дітей
```

Тоді, результатом виклику цієї підпрограми для нашого дерева буде зовсім інший порядок опрацювання вершин дерева

```
14 -> 15 -> 8 -> 9 -> 4 -> 10 -> 11 -> 5 -> 2 -> 12 -> 13 -> 6 -> 7 -> 3 -> 1 ->
```

Пошук у ширину

Пошук в ширину (англ. breadth-first search, BFS), є одним з базових алгоритмів обходу графів. Цей алгоритм є трохи складнішим за алгоритм пошуку вглибину, оскільки використовує додаткову структуру даних. Проте, за допомогою цього алгоритму можна розв'язувати значно ширший клас задач. Крім того, він є основою інших складніших алгоритмів.

Стратегія пошуку в ширину полягає у тому, щоб проходити вузли дерева по рівнях – спочатку корінь, потім всі вершини на першому рівні, потім всі вершини на другому і так далі. На наступному рисунку зображено порядок обходу вершин дерева під час пошуку в ширину. Для наочності різні рівні цього дерева зображено різними відтінками.

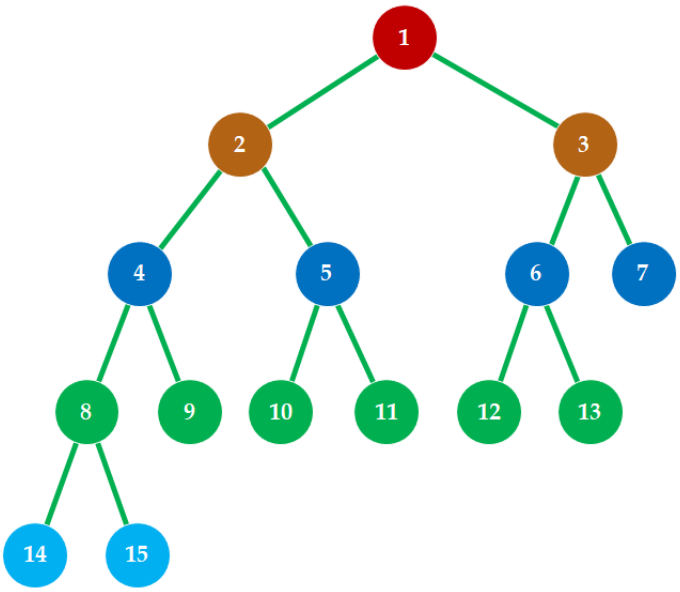


Рисунок 6.1.7. Порядок відвідування вершин під час пошуку в ширину.

Як бачимо, якщо під час обходу вузлів одного рівня, будемо вузли обходити зліва на право, то, конкретно для цього дерева, порядок обходу вузлів всього дерева буде відповідати ключам його вузлів: 1 – 2 – 3 – 4 – 5 – 6 – 7 – 8 – 9 – 10 – 11 – 12 – 13 – 14 – 15.

Як вже було раніше сказано, для послідовного перебору вузлів окремих рівнів дерева, починаючи з кореня, використовується додаткова структура даних – черга, У неї, на початку роботи алгоритму додається корінь. Далі алгоритм полягає у тому, що поки ця черга не порожня, дістаємо з черги елемент, опрацьовуємо його та додаємо до черги всіх його дітей.

Лістинг 6.1.7. Обхід дерева в ширину.

```
def BFS(root):
    """ Обхід дерева в ширину
    :param root: корінь дерева з якого починається обхід
    """
    q = Queue()       # Черга для опрацьованих вузлів
    q.enqueue(root)  # Додаємо у чергу корінь дерева

    while not q.empty():                # Поки черга не порожня
        node = q.dequeue()              # Беремо перший вузол з черги
        print(node.key(), end=" -> ")  # Опрацьовуємо взятий вузол

        # Додаємо в чергу всіх дітей поточного вузла
```

```
        for child in node.getChildren():
            q.enqueue(child)
```

Застосуємо цей алгоритм до дерева, зображеного на рисунку 6.1.1. Як і раніше, для його створення скористаємося описаною раніше підпрограмою `createSampleTree()`.

Лістинг 6.1.7. Продовження. Застосування обходу в ширину.

```
# Головна програма
if __name__ == "__main__":
    tree = createSampleTree()  # підпрограма створення дерева
    BFS(tree)                  # запуск обходу в ширину
```

У результаті роботи програми буде виведено порядок обходу вузлів дерева зображеного на рисунку 6.1.1 під час пошуку в ширину:

```
1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15 ->
```

Зауважимо, що наведений вище алгоритм, як і алгоритм пошуку в глибину, виконується доки не будуть пройдені всі вузли дерева. Проте у випадку, якщо потрібно просто знайти деякий елемент у дереві, програма може здійснювати зайві операції перегляду вузлів (після того, як цільовий елемент знайдено). Перевага цього алгоритму пошуку в глибину, на відміну від наведеного раніше алгоритму пошуку в глибину, полягає у тому, що цей алгоритм легко модифікувати, щоб не здійснювати зайвих операцій. Дійсно, для цього всього лиш потрібно перервати роботу циклу, після того, як цільова вершина знайдена.

Лістинг 6.1.8. Пошук елемента у дереві за ключем використовуючи пошук в ширину.

```
def search(root, elem):
    """ Пошук заданого елемента у дереві
    Використовується пошук в ширину

    :param root: корінь дерева
    :param elem: шуканий елемент
    :return: True, якщо шуканий елемент міститься у дереві
    """

    q = Queue()      # Черга для опрацьованих вузлів
    q.enqueue(root)  # Додаємо у чергу корінь дерева

    while not q.empty():           # Поки черга не порожня
        node = q.dequeue()         # Беремо перший вузол з черги
        if node.key() == elem:     # Якщо елемент знайдено
            return True            # Припиняємо пошук, повертаємо True

        # Додаємо в чергу всіх дітей поточного вузла
        for child in node.getChildren():
            q.enqueue(child)

    return False   # Повертаємо False – дерево не містить шуканого елементу
```

§6.2. Бінарні дерева

6.2.1. Означення та реалізація

У попередньому параграфі ми вивчили, що таке дерева, способи їхнього зображення у комп'ютері та найпростіші алгоритми їхнього обходу. Цей параграф присвячений особливому типу дерев – бінарним деревам.

Означення 6.2.1. Дерево називається **бінарним**, якщо кожен його вузол має не більше ніж двох нащадків.

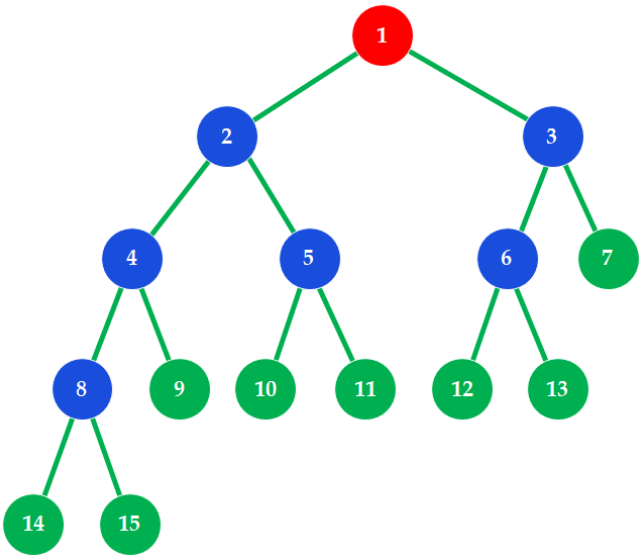


Рисунок 6.2.1. Приклад бінарного дерева.

Чому ці дерева виділяються в окрему групу і вивчають окремо, адже бінарне дерево є лише підвидом дерева. Відповідь на це питання полягає така: двійкові дерева додатково поєднують у собі переваги інших двох структур даних – впорядкованого масиву і зв'язного списку. Пошук у двійковому дереві виконується так само швидко, які у впорядкованому масиві, а операції вставки та видалення елементів майже настільки ж швидко, як у зв'язному списку.

Бінарні дерева вважаються впорядкованими і діти його вузлів називаються спеціальним чином – лівим та правим сином (рисунок 6.2.2). Інша термінологія для бінарних дерев повністю повторює відповідну термінологію попереднього параграфа.

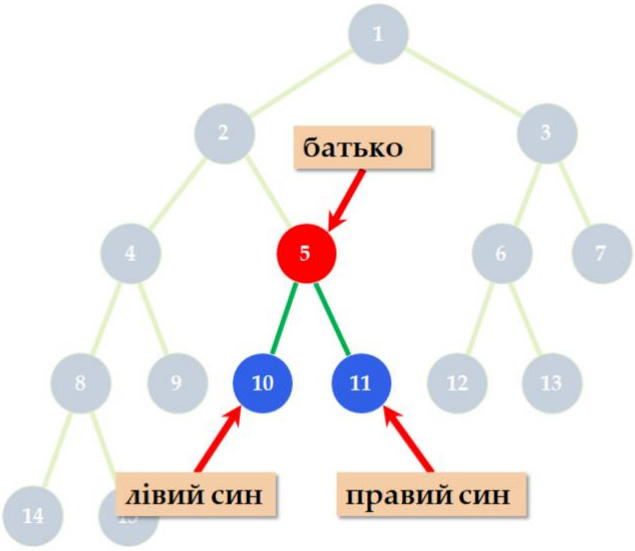


Рисунок 6.2.2. Лівий та правий сини вузла у бінарному дереві.

Кількість дітей вузла бінарного дерева може бути і меншою за два – вузол може мати лише лівого або лише правого сина, а може й взагалі не мати дітей (вузол є листком).

Аналогічно звичайному дереву, бінарне дерево може бути зображене рекурсивним чином

Означення 6.2.2.

1) Один вузол бінарного дерева є бінарним деревом. Цей вузол також є коренем цього дерева.

2) Нехай n – вузол, а T_{left}, T_{right} – дерева з коренями відповідно n_{eft}, n_{right}. Тоді структура утворена таким чином, що для вузла n, вузли n_{eft}, n_{right} є його лівим та правим синами відповідно, також утворює (нове) дерево. При цьому n є коренем нового дерева, а T_{left}, T_{right} – його ліве та праве піддерева.

Графічно це означення можна зобразити так

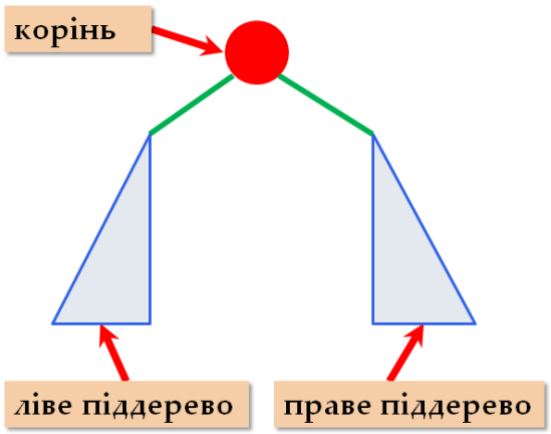


Рисунок 6.2.3. Рекурсивне зображення бінарного дерева.

Наведемо реалізацію бінарного дерева використовуючи рекурсивний підхід. Отже, як і раніше деревом є набір вузів та встановлення зв'язків між ними. Фактично кожен вузол дерева є деревом, що місить навантаження та посилання на двох дітей – лівого та правого – які, у свою чергу, також є деревами (вузлами дерева)

Лістинг 6.2.1. Зображення бінарного дерева у рекурсивному вигляді.

```
class BinaryTree:
    """ Реалізує бінарне дерево."""

    def __init__(self, key):
        """ Конструктор

        Створює новий вузол дерева - корінь та ініціалізує його заданим значенням
        :param key: Навантаження вузла
        """
        self.mKey = key           # ключ кореня дерева
        self.mLeftChild = None    # поле для лівого сина
        self.mRightChild = None   # поле для правого сина
        self.mParent = None       # поле для батька поточного вузла

    def hasLeft(self) -> bool:
        """ Чи містить дерево лівого сина

        :return: True, якщо дерево має лівого сина.
        """
        return self.mLeftChild is not None

    def hasRight(self) -> bool:
        """ Чи містить дерево правого сина

        :return: True, якщо дерево має правого сина.
        """
        return self.mRightChild is not None

    def hasNoChildren(self) -> bool:
        """ Визначає чи має дерево дітей

        :return: True, якщо дерево немає дітей.
        """
        return self.mLeftChild is None and self.mRightChild is None

    def setNode(self, item):
        """ Змінює поточний вузол
        :param item: Нове піддерево або ключ
        """
        if isinstance(item, BinaryTree):          # якщо item є деревом
            self.mKey = item.mKey                 # змінюємо ключ
            self.mLeftChild = item.mLeftChild     # змінюємо ліве піддерево
            self.mRightChild = item.mRightChild   # змінюємо праве піддерево
        else:
```

```
            self.mKey = item

    def setLeft(self, item):
        """ Змінює лівого сина.
        :param item: Навантаження або піддерево
        """
        if isinstance(item, BinaryTree):        # якщо item є деревом
            self.mLeftChild = item              # змінюємо все піддерево
        elif self.hasLeft():                    # якщо дерево містить лівого сина
            self.mLeftChild.setNode(item)       # замінюємо вузол
        else:                                   # якщо дерево немає лівого сина
            self.mLeftChild = BinaryTree(item)  # створюємо дерево з вузлом item
                                                # та робимо його лівим сином
        self.mLeftChild.mParent = self          # встановлення предка для вставленого вузла

    def setRight(self, item):
        """ Змінює правого сина

        :param item: Ключ або піддерево
        """
        if isinstance(item, BinaryTree):         # якщо item є деревом
            self.mRightChild = item              # змінюємо все піддерево
        elif self.hasRight():                    # якщо дерево містить правого сина
            self.mRightChild.setNode(item)       # замінюємо вузол
        else:                                    # якщо дерево немає правого сина
            self.mRightChild = BinaryTree(item)  # створюємо дерево з вузлом item
                                                 # та робимо його правим сином
        self.mRightChild.mParent = self          # встановлення предка для вставленого вузла

    def removeLeft(self):
        """ Видаляє лівого сина """
        self.mLeftChild = None

    def removeRight(self):
        """ Видаляє лівого сина """
        self.mRightChild = None
```

Як і у випадку звичайного дерева, екземпляри класу `BinaryTree` є лише вузлами дерева. Щоб створити повноцінне бінарне дерево, потрібно створити всі його вузли та задати відповідну ієрархію. Підпрограма наведена нижче створює бінарне дерево зображене на рисунку 6.2.1. Звернемо, що наведена вище реалізації не зобов'язує нас створювати окремо вузли, що є листками – можна зразу їх додавати ключами.

Лістинг 6.2.1. Продовження. Побудова бінарного дерева.

```
def createSampleTree():
    """
    Приклад створення бінарного дерева
    """

    # Створимо внутрішні вузли дерева та додаємо до них піддерева
    node8 = BinaryTree(8) # Створення вузла з ключем 8
    node8.setLeft(14)     # Додавання лівого піддерева, додаючи листок 14
    node8.setRight(15)    # Додавання правого піддерева, додаючи листок 15

    node4 = BinaryTree(4) # Створення вузла з ключем 4
    node4.setLeft(node8)  # Додавання лівого піддерева
    node4.setRight(9)     # Додавання правого піддерева, додаючи листок 9

    node5 = BinaryTree(5) # Створення вузла з ключем 5
    node5.setLeft(10)     # Додавання лівого піддерева, додаючи листок 10
    node5.setRight(11)    # Додавання правого піддерева, додаючи листок 11

    node2 = BinaryTree(2)  # Створення вузла з ключем 2
    node2.setLeft(node4)   # Додавання лівого піддерева, додаючи листок
    node2.setRight(node5)  # Додавання правого піддерева, додаючи піддерево
```

```
    node6 = BinaryTree(6)
    node6.setLeft(12)      # Додавання лівого піддерева, додаючи листок 12
    node6.setRight(13)     # Додавання правого піддерева, додаючи листок 13

    node3 = BinaryTree(3)  # Створення вузла з ключем 3
    node3.setLeft(node6)   # Додавання лівого піддерева
    node3.setRight(7)      # Додавання правого піддерева, додаючи листок 7

    # Створюємо корінь дерева та додаємо до нього відповідні вузли
    root = BinaryTree(1)
    root.setLeft(node2)    # Додавання лівого піддерева до кореня
    root.setRight(node3)   # Додавання правого піддерева до кореня

    return root  # Функція повертає корінь створеного дерева
```

6.2.2. Алгоритми на бінарних деревах

Два найпростіших алгоритми для дерев – пошук в глибину і ширину – очевидно також мають місце для бінарних дерев. Їхня реалізація фактично повністю повторює таку для звичайних дерев. Відмінність буде лише у тому, що змість перебору всіх дітей поточного вузла з допомогою циклу, буде вибиратися лівий та правий сини поточного вузла (якщо вузол їх має). Тому, наведемо алгоритми пошуку в глибину та ширину для бінарних дерев без додаткових пояснень.

Лістинг 6.2.2. Пошук у глибину у бінарному дереві.

```
def DFS(root):
    """ Обхід бінарного дерева в глибину

    :param root: Корінь бінарного дерева
    """

    print(root.mKey)            # Опрацьовуємо корінь елемент

    if root.hasLeft():          # якщо дерево має лівого сина
        DFS(root.mLeftChild)    # запускаємо DFS для лівого сина

    if root.hasRight():         # якщо дерево має правого сина
        DFS(root.mRightChild)   # запускаємо DFS для правого сина
```

Лістинг 6.2.3. Пошук у ширину у бінарному дереві.

```
def BFS(root):
    """ Обхід бінарного дерева в ширину

    :param root: Корінь бінарного дерева
    """
    q = Queue()
    q.enqueue(root)  # Додаємо у чергу корінь дерева

    while not q.empty():
        node = q.dequeue()    # Беремо перший елемент з черги
        print(node.mKey)      # Опрацьовуємо взятий елемент

        # Додаємо в чергу лівий і правий сини поточного вузла
        if node.hasLeft():                # якщо поточний вузол має лівого сина
            q.enqueue(node.mLeftChild)    # додаємо у чергу лівого сина
        if node.hasRight():               # якщо поточний вузол має правого сина
            q.enqueue(node.mRightChild)   # додаємо у чергу правого сина
```

У цьому пункті ми розглянули бінарні дерева, навчилися їх створювати та розглянули адаптацію алгоритмів пошуку в глибину та ширину для бінарних дерев. Перейдемо до застосувань бінарних дерев. Як ми побачимо всі ці застосування пов’язані з різними видами пошуку даних. Раніше ми вже розглянули префіксні дерева, що дозволяють реалізувати інтерфейси асоційованих масивів і здійснювати пошук за час, що є пропорційний довжині ключа. Проте, такі дерева не призначені для розв’язання задач пов’язаних з обробкою всіх даних у

масиві, наприклад, у випадку, якщо потрібно швидко знайти (або вилучити) мінімум серед елементів заданого набору, або знайти суму елементів фрагменту заданого відрізку. Саме для таких задач і застосовуються бінарні дерева.

6.2.3. Бінарне дерево пошуку

Одне з найширших застосувань бінарних дерев полягає у способі організації даних для їхнього швидкого пошуку. Наприклад, вони будуть корисними для реалізації таких структур як множини або асоціативні масиви (що гарантують доступ до значень за ключем). Така організація даних отримала назву – **бінарні дерева пошуку**. Положення даних у таких деревах не є принциповим – більший інтерес складає швидкість їхнього відшукання, яка, при правильній організації даних відбувається за логарифмічний час (кількості елементів у структурі).

Розглянемо детальніше означення та приклади бінарних дерев. Нехай на множині ключів визначено операції відношень (у тому числі операції порівняння $<, \leq, >, \geq$). Тоді

Означення 6.2.3. Бінарне дерево називається **бінарним деревом пошуку**, якщо
1) обидва його піддерева – і ліве і праве – є бінарними деревами пошуку;
2) для будь-якого вузла n, значення ключів усіх вузлів лівого піддерева є меншими за значення ключа вузла n;
3) для будь-якого вузла n,значення ключів усіх вузлів правого піддерева є більшими за значення ключа вузла n.

Для бінарних дерев пошуку визначають такі базові операції:
1. Створення дерева пошуку.
2. Операція `search(key)` – пошук елемента у дереві за заданим ключем key.
3. Операція `insert(item)` – вставка елемента `item` у дерево.
4. Також, залежно від поставленої задачі, часто для бінарних дерев пошуку визначають операцію `delete(key)`, що видаляє елемент з ключем key у дереві, якщо такий елемент у ньому міститься.

Як випливає з операцій, наведених вище, бінарні дерева пошуку дозволяють реалізувати абстрактний тип даних – асоціативний масив, подібно до хеш-таблиці.

Нижче на рисунку 6.2.4 зображене бінарне дерево пошуку, у вузлах якого зображені ключі (12, 19, 8, 4, 10, 5, 21, 11, 15 , 9, 1, 14, 16), що є цілими числами.

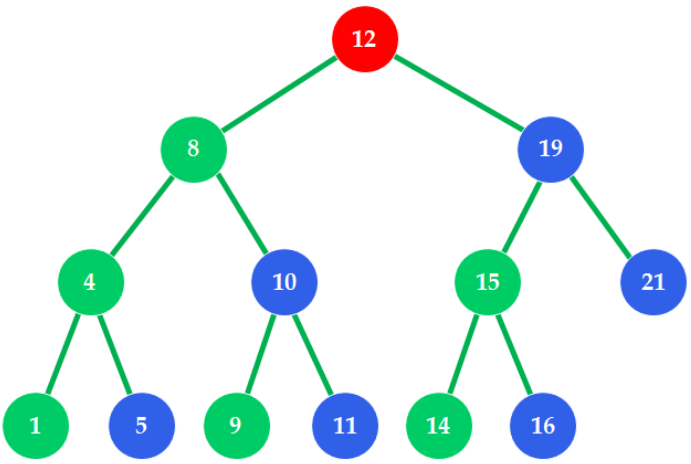


Рисунок 6.2.4. Бінарне дерево пошуку

Розглянемо реалізацію бінарних дерев пошуку. Вона буде базуватися на реалізації бінарного дерева, що наведена у лістингу 6.2.1, та полягає у описі методів для операцій 3-5.

Пошук та вставка вузлів у дерево

Враховуючи вищенаведене означення бінарного дерева, очевидним є алгоритм пошуку вузла з заданим ключем key. Дійсно, досить лише рухатися починаючи від кореня дерева у напрямку його листків, вибираючи у вузлах гілку, залежно від результату порівняння шуканого елементу з ключем key зі значенням ключа відповідного вузла:

- якщо шуканий ключ key більшій за ключ вузла – вибираємо праву гілку;
- якщо шуканий ключ key менший за ключ вузла – вибираємо ліву гілку;
- якщо шуканий ключ key збігається з ключем вузла – елемент знайдено у дереві;
- якщо вузол є листком – дерево пошуку не містить шуканого елементу.

Для прикладу роботи алгоритму розглянемо для вищенаведеного дерева пошук за ключем 9.

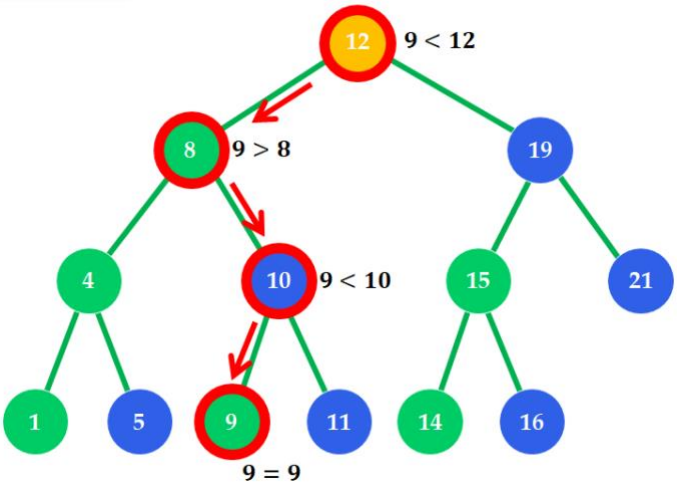


Рисунок 6.2.5. Пошук вузла з ключем 9 у бінарному дереві пошуку

Операція вставки нового вузла з ключем key у дерево дуже подібна на операцію пошуку – фактично вона полягає у пошуку місця для вставки нового листка у дерево. Її алгоритм полягає у такому:

- якщо бінарне дерево пошуку порожнє – коренем дерева встановлюємо новий вузол з ключем key;
- якщо дерево не порожнє, спускаємося по дереву від кореня відповідно до вищенаведеного алгоритму пошуку;
- якщо дерево містить ключ key, припиняємо роботу алгоритму – вставка не потрібна
- якщо дерево не містить ключа key, тобто ми досягли вузла, що не має відповідного сина – додаємо новий вузол з ключем key як відповідного сина дерева.

На рисунку нижче показано додавання у дерево, що зображене на рисунку 6.2.4, нового вузла з ключем 26.

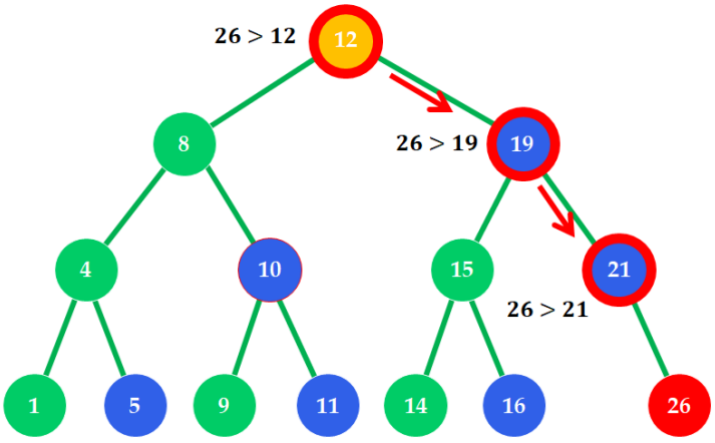


Рисунок 6.2.6. Вставка вузла з ключем 26 у бінарне дерево пошуку

Розглянемо тепер реалізацію операцій пошуку та вставки у бінарними деревами пошуку. Вона буде базуватися на реалізації бінарного дерева, яку наведено вище у лістингу 6.2.1 та використовувати рекурсивну структуру бінарного дерева.

Лістинг 6.2.4. Реалізація бінарного дерева пошуку – пошук та вставка.

```
class SearchTree(BinaryTree):
    """ Клас - Бінарне дерево пошуку.

     Реалізує структуру даних, у якій вставка та пошук елементів здійснюється
     (в середньому) за логарифмічний час. """

    def insert(self, key):
        """ Метод, що реалізує вставку елемента у бінарне дерево

        :param key: ключ, що необхідно вставити
        """
        self._insert_helper(self, key)  # запускаємо вставку елемента key у дерево,
```

```
                                    # починаючи з кореня

    def search(self, key):
        """ Метод, що реалізує пошук елемента item у бінарному дереві

        :param key: Шуканий елемент
        :return: Вузол з ключем key якщо такий елемент міститься у дереві
                 та None - якщо елемент не знайдений.
        """
        return self._search_helper(self, key)  # запускаємо пошук вузла з ключем key у дереві,
                                               # починаючи з кореня

    @staticmethod
    def _insert_helper(root, key):
        """ Допоміжний рекурсиввий метод, для вставки заданого елемента у задане піддерево.

        :param root: корінь піддерева у яке відбувається вставка нового елементу
        :param key: Елемент для вставки
        """

        if root.mKey > key:          # якщо елемент для вставки має міститися у лівому піддереві
            if root.hasLeft():       # якщо дерево має лівого нащадка
                #  запускаємо рекурсивно вставку item у ліве піддерево
                SearchTree._insert_helper(root.mLeftChild, key)
            else:                    # якщо дерево не має лівого нащадка
                root.setLeft(key)    # додаємо item у ролі лівого нащадка
        elif root.mKey < key:        # якщо елемент для вставки має міститися у правому піддереві
            if root.hasRight():      # якщо дерево має правого нащадка
                #  запускаємо рекурсивно вставку item у праве піддерево
                SearchTree._insert_helper(root.mRightChild, key)
            else:                   # якщо дерево не має правого нащадка
                root.setRight(key)  # додаємо item у ролі правого нащадка

    @staticmethod
    def _search_helper(root, key):
        """ Допопоміжний рекурсиввий метод, для пошуку елементу у заданому піддереві.

        Пошук здійснюється проходом в глибину.
        :param root: корінь піддерева у якому здійснюється пошук
        :param key: Шуканий елемент
        :return: посилання на знайдений елемент, якщо елемент міститься у дереві
                 та None - якщо елемент не знайдений.
        """
        if root.mKey == key:  # якщо ключ поточного вузла збігається з шуканим,
            return root  # повертаємо знайдений вузол
        elif key < root.mKey:  # випадок: шуканий елемент може міститися у лівому піддереві
            return SearchTree._search_helper(root.mLeftChild, key) if root.hasLeft() else None
        else:  # випадок: шуканий елемент може міститися у правому піддереві
            return SearchTree._search_helper(root.mRightChild, key) if root.hasRight() else None
```

Як бачимо, наведена вище реалізація бінарного дерева пошуку використовує рекурсивні підпрограми **_insert_helper**() та **_search_helper**() у яких реалізовано рух вниз по вузлах дерева у пошуку необхідного елементу. Ці функції є дуже простими та наглядними з точки зору розуміння алгоритму. Проте, якщо висота дерева є значною, то може виникнути перевищення допустимої глибини вкладення у рекурсію, з відповідними наслідками. Тому нижче наведемо нерекурсивну реалізацію операцій вставки та пошуку у дереві.

Лістинг 6.2.5. Нерекурсивна реалізація операцій пошуку та вставки в бінарне дерево.

```
class SearchTree(BinaryTree):
    """ Клас - Бінарне дерево пошуку.

     Реалізує структуру даних, у якій вставка та пошук елементів здійснюється
     (в середньому) за логарифмічний час. """

    def search(self, key):
        """ Метод, що реалізує пошук елемента item у бінарному дереві
            Не рекурсивна реалізація
        :param key: Шуканий елемент
```

```
        :return: Вузол з ключем key якщо такий елемент міститься у дереві
        та None - якщо елемент не знайдений.
        """
        node = self   # починаємо з кореня
        while node is not None:
            if node.mKey == key:   # якщо ключ поточного вузла збігається з шуканим,
                return node        # повертаємо знайдений вузол
            elif node.mKey > key:  # випадок: шуканий елемент може міститися у лівому піддереві
                node = node.mLeftChild  # рухаємося вниз до лівого нащадка
            else:                  # випадок: шуканий елемент може міститися у правому піддереві
                node = node.mRightChild # рухаємося вниз до правого нащадка
        # якщо опустилися по дереву до листка, то дерево не містить шуканого елемента
        return None

    def insert(self, key):
        """ Метод, що реалізує вставку елемента у бінарне дерево
            Не рекурсивна реалізація

        :param key: ключ, що необхідно вставити
        """
        node = self         # починаємо з кореня
        while True:
            if node.mKey == key:       # якщо ключ поточного вузла збігається з шуканим,
                break                  # вставка не потрібна
            elif node.mKey > key:      # якщо елемент вставки має міститися у лівому піддереві
                if node.hasLeft():     # якщо дерево має лівого нащадка
                    node = node.mLeftChild  # рухаємося вниз до лівого нащадка
                else:                  # якщо дерево не має лівого нащадка
                    node.setLeft(key)  # додаємо key у ролі лівого нащадка
                    break
            elif node.mKey < key:      # якщо елемент вставки має міститися у правому піддереві
                if node.hasRight():    # якщо дерево має правого нащадка
                    node = node.mRightChild # рухаємося вниз до правого нащадка
                else:                  # якщо дерево не має правого нащадка
                    node.setRight(key) # додаємо key у ролі правого нащадка
                    break
```

Видалення вузлів

Нарешті розглянемо алгоритм операцію видалення. Алгоритмічно вона є найскладнішою серед набору базових операцій. Видалення елемента з ключем key, прочитається з пошуку відповідного вузла у дереві.

- якщо бінарне дерево пошуку порожнє або такого елементу у дереві не знайдено, то операція, у найпростішому випадку, породжує виключення;
- якщо відповідний вузол знайдений, аналізуємо наявність у нього дітей:
    a. якщо вузол немає дітей – видаляємо його без будь-яких наслідків.
    b. якщо вузол має лише одного сина (лівого або правого) – заміняємо вузол, що видаляється – сином.
    c. якщо вузол має обох синів, знаходимо у лівому піддереві вузол, з максимальним ключем – переставляємо цей вузол на місце елемента, що видаляється (ця операція передбачає відповідно рекурсивну операцію видалення знайденого вузла з максимальним ключем у лівому піддереві).

Щоб краще зрозуміти, розглянемо графічно всі ці три випадки. Отже, розглянемо знову дерево, що зображене на рисунку 6.2.4 та розглянемо видалення з нього вузла з ключем 9. Спочатку здійснимо пошук вузла з цим ключем.

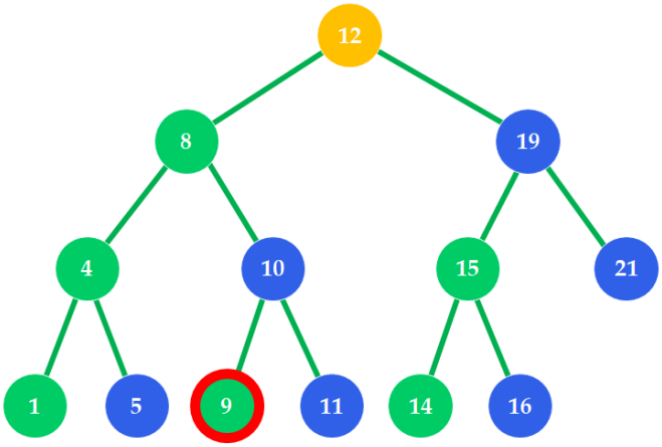


Рисунок 6.2.7. Пошук вузла з ключем 9

Як бачимо цей вузол немає дітей, тому просто видаляємо його з дерева.

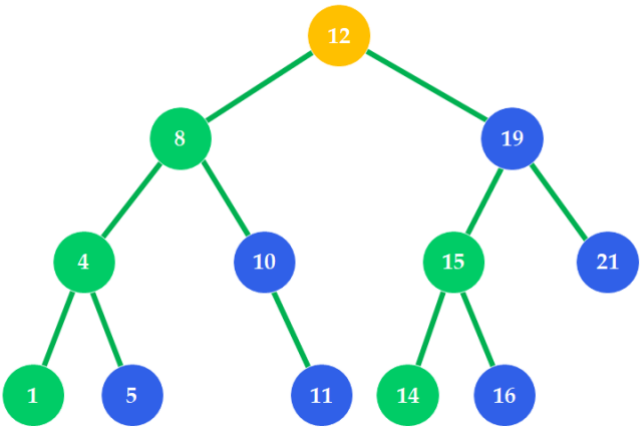


Рисунок 6.2.8. Видалення вузла з ключем 9

Видалимо тепер з отриманого дерева вузол 10. Знову шукаємо його в дереві.

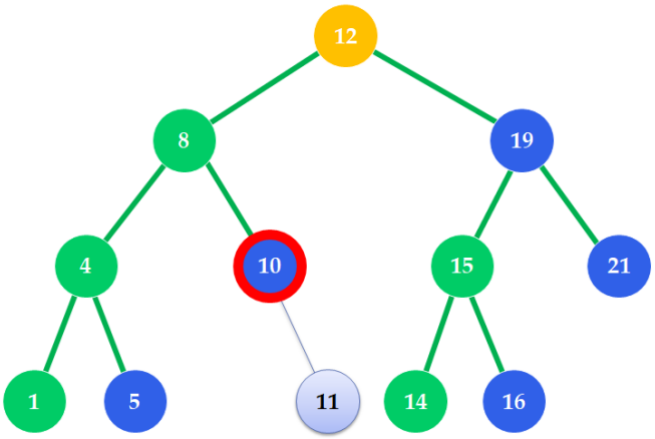


Рисунок 6.2.9. Знайдений вузол 10 має лише правого сина

Знайдений вузол має лише одного правого сина (з ключем 11), тому просто видаляємо вузол 10 та переставляємо на його місце вузол 11.

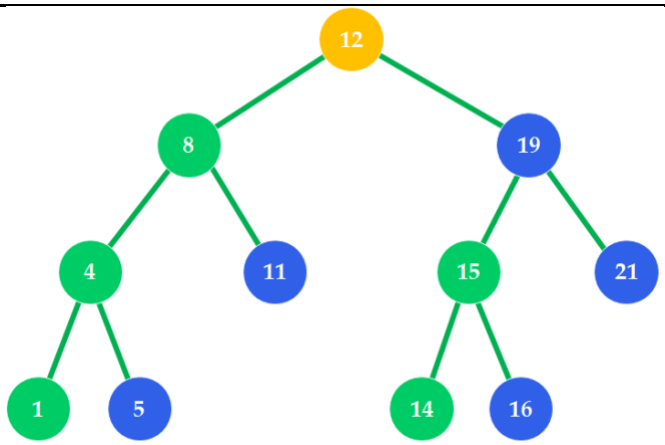


Рисунок 6.2.10. Видалення вузла з ключем 10

Розглянемо тепер третій випадок, коли вузол, що необхідно видалити має обох синів. Отже, видалимо з дерева, отриманого в наслідок останнього видалення вузла з дерева (рисунок 6.2.10), вузол з ключем 8.

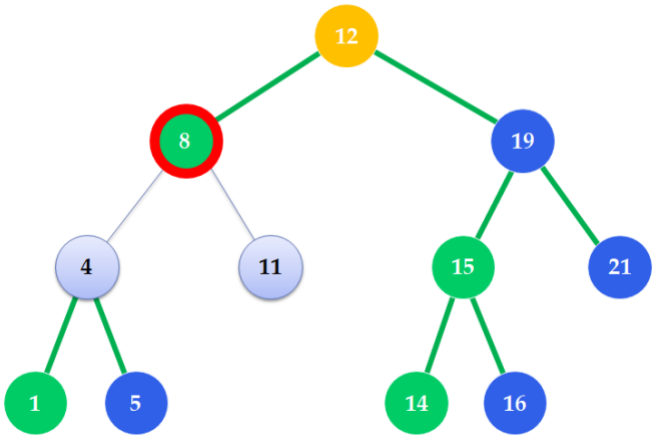


Рисунок 6.2.11. Знайдений вузол 8 має обох дітей.

Знаходимо у лівому піддереві вузол з найбільшим ключем – це вузол 5

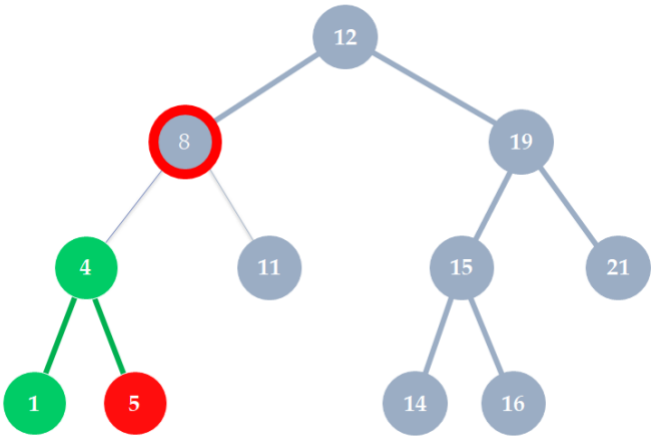


Рисунок 6.2.12. Найбільший елемент лівого піддерева вузла 8 є 5.

Переставляємо його на місце вузла 8, що і завершує видалення вузла.

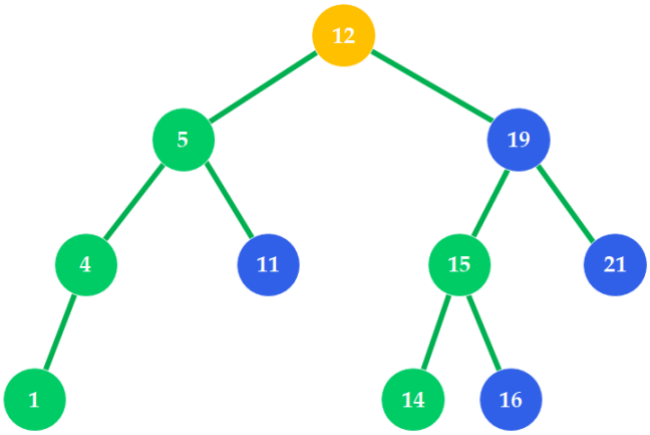


Рисунок 6.2.13. Переставляємо вузол 5 на місце вузла 8.

У лістингу 6.2.6 наведено опис класу SearchTreeWithDelete, що є бінарним деревом пошуку, з успадкованими від класу SearchTree операціями пошуку та видалення елементів, та з підтримкою операції видалення вузлів.

Лістинг 6.2.6. Реалізація бінарного дерева пошуку – видалення.

```
class SearchTreeWithDelete(SearchTree):
    """ Розширення класу бінарного дерева можливістю видаляти елементи """

    def delete(self, key):
        """ Видаляє заданий елемент у бінарному дереві

        :param key: Елемент, який потрібно видалити з бінарного дерева
        """
        self._delete_helper(self, key)

    @staticmethod
    def _search_max(root):
        """ Допоміжний рекурсивний метод пошуку найбільшого вузла у заданому піддереві.

            Згідно з властивостями бінарного дерева пошука, максимальний елемент може бути
            знайдений при проходженні дерева в глиб рухаючись лише по правих нащадках
        :param root: корінь піддерева у якому небхідно знайти найбільший вузол
        :return: знайдений вузол.
        """
        return SearchTreeWithDelete._search_max(root.mRightChild) if root.hasRight() else root

    @staticmethod
    def _delete_helper(root, key):
        """ Допоміжний рекурсивний метод, що видаляє з дерева вузол з
            заданим ключем, якщо такий ключ міститься у дереві.

        :param root: корінь піддерева у якому потрібно видалити заданий елемент
        :param key: Елемент, який потрібно видалити
        """
        node = SearchTreeWithDelete._search_helper(root, key)  # Знаходимо вузол, який треба
                                                               # видалити
        if node is None:  # Якщо шуканий елемент не міститься у дереві, то припиняємо роботу
                                                               # підпрограми
            return
        if node.hasNoChildren():      # Якщо знайдений вузол - листок (немає нащадків)
            if node.mParent is None:  # Якщо предок - корінь всього дерева
                node.mKey = None      # Робимо дерево порожнім
            else:
                if node.mParent.mLeftChild == node:
                    node.mParent.mLeftChild = None  # Видаляєм знайдений елемент
                else:
                    node.mParent.mRightChild = None   # Видаляєм знайдений елемент
        elif node.hasRight() and not node.hasLeft():  # Якщо знайдений вузол має лише одну
```

```
                                                # праву гілку
            node.setNode(node.mRightChild)      # Замінюємо знайдений вузол його правим
                                                # піддіревом
        elif node.hasLeft() and not node.hasRight():  # Якщо знайдений вузол має лише одну
                                                # ліву гілку
            node.setNode(node.mLeftChild)       # Замінюємо знайдений вузол його лівим
                                                # піддіревом
        else:                                   # Якщо знайдений вузол має обидві гілки
            left_max = SearchTreeWithDelete._search_max(node.mLeftChild)  # Знаходимо
                                                # максимальний вузол у лівому піддереві
            left_max_key = left_max.mKey
            node.setNode(left_max_key)          # Замінюємо значення елемета node
                                                # знайденим максимальним
            SearchTreeWithDelete._delete_helper(node.mLeftChild, left_max_key)  # Видалення з
                                                #  лівого піддерева найбільшого елементу
```

Нерекурсивну реалізацію операції видалення пропонуємо здійснити читачу самостійно.

Аналіз дерева пошуку та застосування

Звернемо увагу читача на кілька фактів пов'язаних з бінарними деревами пошуку:

1. Найбільший елемент у бінарному дереві пошуку розташовується у листку, до якого можна дістатися послідовним переходом починаючи від кореня переходячи кожного разу до правого сина вузла. Наприклад для дерева зображеного на рисунку 6.2.4, найбільшим є вузол 21, до якого можна дістатися подолавши шлях 12 – 19 – 21.
2. Найменший елемент у бінарному дереві пошуку розташовується у листку, до якого можна дістатися послідовним переходом починаючи від кореня переходячи кожного разу до лівого сина вузла. Наприклад для дерева зображеного на рисунку 6.2.4, найменшим є вузол 1, до якого можна дістатися подолавши шлях 12 – 8 – 4 – 1.
3. Середній час пошуку, вставки та видалення елементів у бінарне дерево пошуку, є $O(\log n)$, де n – кількість вузлів, що міститься у дереві.
4. Час пошуку, вставки та видалення елементів у бінарне дерево пошуку може регресувати до $O(n)$.

Пункт 3 вищенаведеного переліку може здатися читачу очевидним. Дійсно, проаналізувавши рисунок 6.2.4, стає зрозуміло, що висота дерева є $\log n$. Саме це і пояснює таку асимптотику швидкодії базових операцій. Проте так сталося тому, що додавання вузлів у дерево було рівномірним – по рівнях. Але все може бути по іншому, навіть у випадку побудови дерева з тієї ж множини вузлів. Дісно відсортуємо послідовність ключів (12, 19, 8, 4, 10, 5, 21, 11, 15, 9, 1, 14, 16) що використовувалася при побудові дерева зображеного на рисунку 6.2.4, за зростанням: (1, 4, 5, 8, 9, 10, 11, 12, 14, 15, 16, 19, 21) та побудуємо бінарне дерево пошуку послідовним додавання вузлів цього списку. Результат буде зображений на рисунку нижче

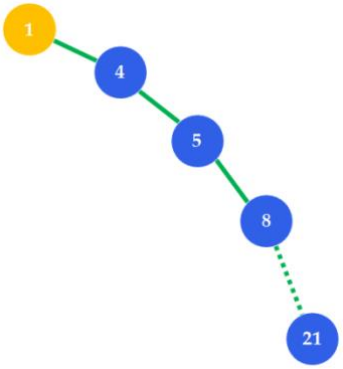


Рисунок 6.2.14. Додавання відсортованої послідовності у бінарне дерево пошуку.

Як бачимо, бінарне дерево фактично виродилося у лінійний список, пошук у якому відбувається за лінійний час. Для уникнення цієї ситуації застосовуються різні підходи балансування дерев під час вставки та видалення елементів, про які буде розказано у наступному пункті.

Отже, одне з основних застосувань бінарних дерев пошуку – реалізація інтерфейсу асоціативного масиву. Проте, на відміну від хеш-таблиці, яка дозволяє здійснювати пошук за сталий час $O(1)$, у бінарному дереві пошуку, відшукання ключа та пов'язаного з ним значення відбувається повільніше, а саме за логарифмічний час $O(\log n)$. Більші витрати часу компенсуються тим, що взагалі кажучи, бінарне дерево пошуку, на відміну від хеш-таблиці не вимагає додаткової пам'яті – використовується лише пам'ять для безпосереднього зберігання пар ключ–значення.

6.2.4. Збалансовані дерева пошуку

Вище ми розглянули побудову бінарного дерева пошуку та виявили, що при незбалансованості його швидкодія регресує до $O(n)$. У цьому пункті розглянемо один зі спеціальних видів бінарних дерев, що автоматично гарантують підтримують себе у збалансованому вигляді.

Означення 6.2.4. **Збалансоване дерево** – бінарне дерево яке автоматично підтримує свою висоту таким чином, щоб кількість рівнів вершин під коренем була мінімальною.

Означення 6.2.5. Бінарне дерево називається **ідеально збалансованим**, якщо для кожної його вершини кількість вершин у лівому та правому піддереві відрізняються не більше ніж на одиницю.

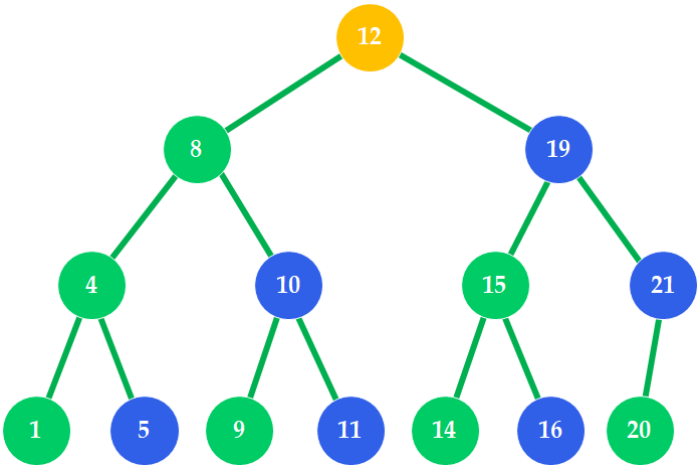


Рисунок 6.2.15. Ідеально збалансоване дерево.

Підтримувати умову ідеальної збалансованості доволі складна задача, тому на практиці використовують менш жорсткі означення збалансованості, наприклад, AVL-збалансованості.

АВЛ-дерева

Означення 6.2.6. Бінарне дерево називається **AVL- збалансованим**, якщо для кожної його вершини висоти лівого та правого піддерев різняться не більше ніж на одиницю.

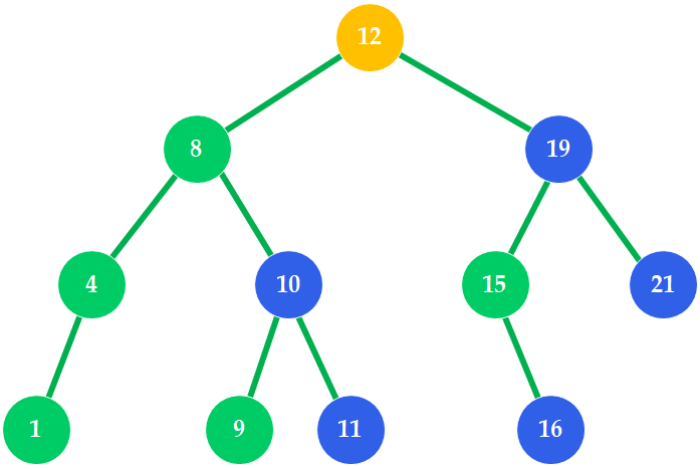


Рисунок 6.2.16. AVL-дерево.

Дерева, що задовольняють наведеній у означенні 6.2.6 умові збалансованості називають **AVL-деревами**[4]. AVL-дерево, як і звичайне бінарне дерево пошуку застосовується для реалізації асоціативного масиву. Проте, на відміну від бінарного дерева, AVL-дерево підтримує збалансованість на кожній з операцій вставки та видалення даних. Щоб реалізувати AVL-дерево, необхідно відстежувати на кожній з операцій вставки та видалення фактор балансу для кожного вузла дерева, спостерігаючи за висотою лівого та правого його (вузла) піддерев:

$$F_B = h(T_L) - h(T_R)$$

[4] Абревіатура утворена першими літерами прізвищ творців AVL-дерев – Георгія Максимовича Адельсон-Вельського та Євгена Михайловича Ландіса.

тут F_B – фактор балансу, $h(T_L)$ та $h(T_R)$ – висоти лівого та правого піддерева відповідно. На рисунку 6.2.17 зображено фактор балансу для всіх вузлів AVL-дерева, зображеного на рисунку 6.2.16.

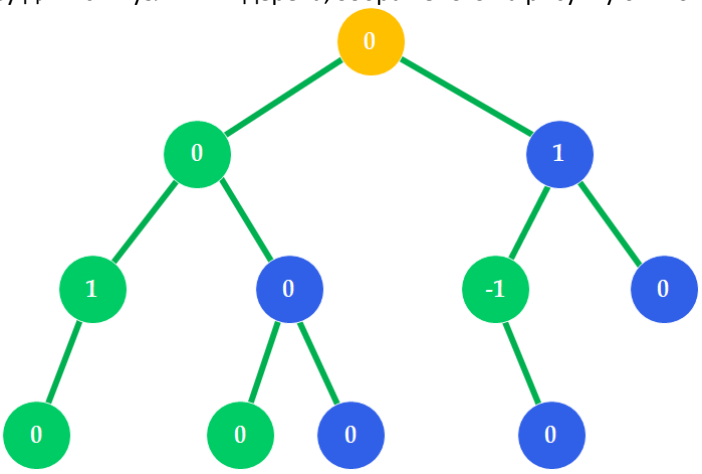


Рисунок 6.2.17. Фактор збалансованості для кожного вузла дерева.

Будемо казати, що дерево у вузлі переважує вліво, якщо фактор балансу у цьому вузлі більший за нуль, та вправо, якщо фактор збалансованості менший за нуль. Згідно з означенням 6.2.6, дерево є AVL–збалансованим (надалі збалансованим), якщо у кожному вузлі його фактор збалансованості дорівнє -1, 0 або 1. Якщо на одній з операцій вставки чи видалення, фактор збалансованості виходить з переліку зазначених значень, то застосовується процедура перебанансування, що повертає дерево до збалансованого стану. Нижче, на рисунку 6.2.16 зображене незбалансоване дерево, що переважає вліво.

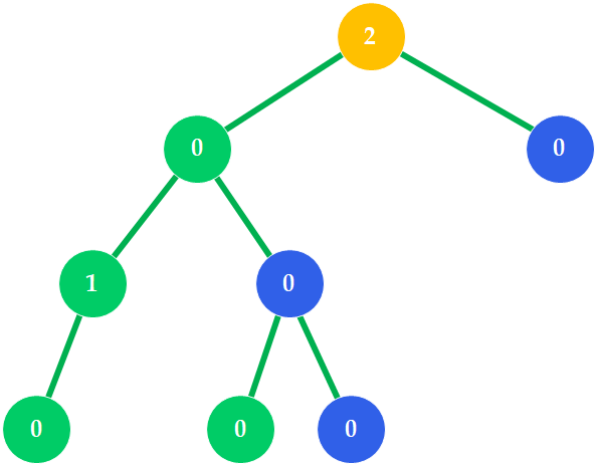


Рисунок 6.2.18. Незбалансоване дерево та фактори балансу у всіх вузлах дерева.

Вставка вузлів у АВЛ-дерево

Реалізація AVL–дерева базується на реалізації бінарного дерева пошуку. Відмінність полягає у тому, що після вставки чи видалення вузла, може вимагатися процедура балансування, якщо фактор балансу у деякому вузлі вийшов за межі переліку –1, 0, 1. Такий вузол будемо називати розбалансованим.

Розглянемо детальніше процедуру балансування під час встравки нового вузла. Оскільки всі ключі вставляються як листки, то, очевидно, фактор балансу вставленого вузла дорівнює нулю і, відповідно, процедуру балансування до нього застосовувати немає сенсу. Проте, необхідно оновити фактор збалансованості його батька. Дійсно, якщо листок – правий син, то фактор балансу зменшиться на одиницю, якщо лівий – то збільшиться на одинцю. Процедура корегування балансу рекурсивно поширюється на предків вставленого вузла аж до самого кореня дерева.

Вона буде мати два таких термінальних випадки:

- рекурсивний виклик досяг кореня дерева;
- фактор балансу батька скоректувався до нуля.

Вправа 6.2.1. Доведіть, що якщо фактор балансу кореня піддерева скориговується до нуля, то баланс його вузла-предка не змінюється.

Опишемо клас `AVLTree`, що є нащадком класу `SearchTree`. Клас `AVLTree` успадкує від предка метод **`insert`**`()` вставки нового вузла у дерево. Цей метод використовує допоміжні методи **`setLeft`**`()` та **`setRight`**`()`, які відповідають за вставку вузла після відшукання необхідної позиції. Замістимо ці методи у класі `AVLTree` новою

реалізацією, яка відрізнятиметься від базової тим, що містить виклик процедури оновлення балансу для батьківського вузла.

Лістинг 6.2.7. Клас AVLTree.

```
class AVLTree(SearchTree):

    def __init__(self, key=None, left=None, right=None):
        super().__init__(key, left, right)
        self.mBalanceFactor = 0
        self.mIsRoot = False

    def setLeft(self, item):
        """ Змінює лівого сина.
        :param item: Навантаження або піддерево
        :return: None
        """
        if isinstance(item, AVLTree):           # якщо item є деревом
            self.mLeftChild = item              # змінюємо все піддерево
        elif self.hasLeft():                    # якщо дерево містить лівого сина
            self.mLeftChild.setNode(item)       # замінюємо вузол
        else:                                   # якщо дерево немає лівого сина
            self.mLeftChild = AVLTree(item)     # створюємо дерево з вузлом item та
                                                # робимо його лівим сином
        self.mLeftChild.mParent = self
        self.updateBalance(self.mLeftChild)     # оновлення балансу для предків вставленого вузла

    def setRight(self, item):
        """ Змінює правого сина
        :param item: Ключ або піддерево
        :return: None
        """
        if isinstance(item, AVLTree):            # якщо item є деревом
            self.mRightChild = item              # змінюємо все піддерево
        elif self.hasRight():                    # якщо дерево містить правого сина
            self.mRightChild.setNode(item)       # замінюємо вузол
        else:                                    # якщо дерево немає правого сина
            self.mRightChild = AVLTree(item)     # створюємо дерево з вузлом item та
                                                 # робимо його правим сином
        self.mRightChild.mParent = self
        self.updateBalance(self.mRightChild)    # оновлення балансу для предків вставленого вузла
```

Для балансування дерева, у розбалансованому вузлі застосовуються операція перебалансування, що змінює зв’язок предок-нащадок у піддереві цього вузла так, щоб фактор балансу для всіх вершин піддерева опинився у переліку –1, 0, 1. Ця операція здійснюється у рекурсивному методі **updateBalance**(), реалізацію якого наведено у наступному лістингу

Лістинг 6.2.7. Продовження. Метод оновлення балансу.

```
    @staticmethod
    def updateBalance(node):

        if node.mBalanceFactor > 1 or node.mBalanceFactor < -1:
            AVLTree.rebalance(node)
            return

        if not node.mParent.mIsRoot:
            if node.isLeftChild():
                node.mParent.mBalanceFactor += 1
            elif node.isRightChild():
                node.mParent.mBalanceFactor -= 1
            if node.mParent.mBalanceFactor != 0:
                AVLTree.updateBalance(node.mParent)
```

На першому кроці метод перевіряє чи втратив вузол баланс настільки, що вимагається процедура перебалансування, що здійснюється у за допомогою статичного методу `rebalance()`, що буде описаний нижче. Якщо перебалансування не потрібне, то просто корегується фактор балансу батька відповідно до роз'яснень наведених вище. Далі, якщо цей фактор балансу відмінний від нуля (див. Вправа 6.2.1), то алгоритм продовжує роботу рекурсивно вгору по дереву, викликаючи `updateBalance()` для батька поточного вузла.

Повернемося тепер до процедури перебалансування вузла, що здійснюється за допомогою статичного методу `rebalance()`. Ця процедура використовує один з чотирьох типів обертань розбалансованої вершини:

1. Мале ліве обертання.

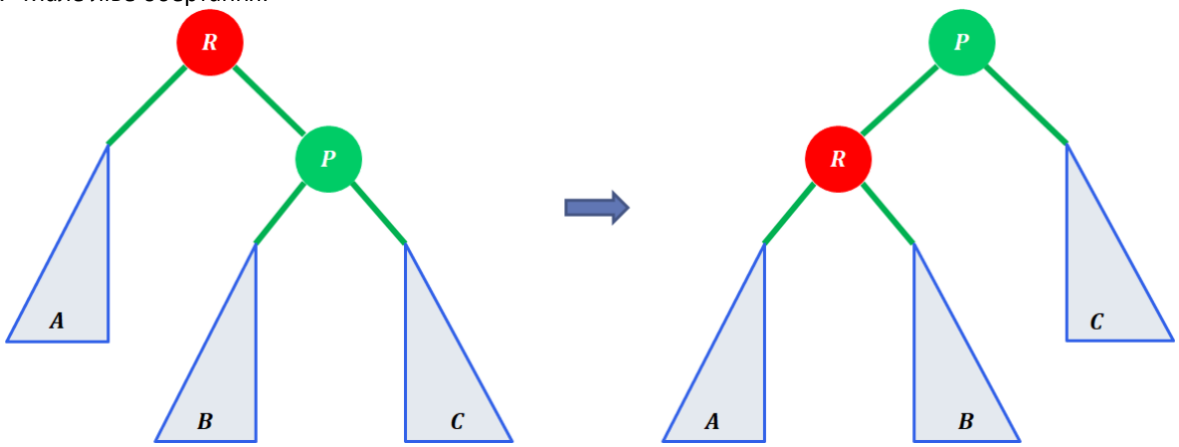


Рисунок 6.2.19. Мале ліве обертання.

Використовується коли

$$h_P - h_A = 2 \\ h_B \leq h_C \qquad (6.2.1)$$

тут h – висота відповідного піддерева. У такому випадку кажуть, що відбувається лівий поворот вершини R навколо вершити P. При цьому, як було зазначено вище, змінюється зв'язок предок-нащадок – піддерево з коренем R під час повороту перетворюється у піддерево з коренем P, відповідно змінюється і точка підвішування піддерева у загальному дереві. Алгоритмічно мале ліве обертання складається з таких кроків:

a. переміщуємо вершину P, так, щоб вона стала коренем піддерева;
b. переміщуємо старий корінь піддерева – вершину R – так щоб вона стала лівим сином для вузла P;
c. переміщуємо старого лівого сина вузла P – піддерево B – так, що він став правим сином вузла R.

2. Мале праве обертання.

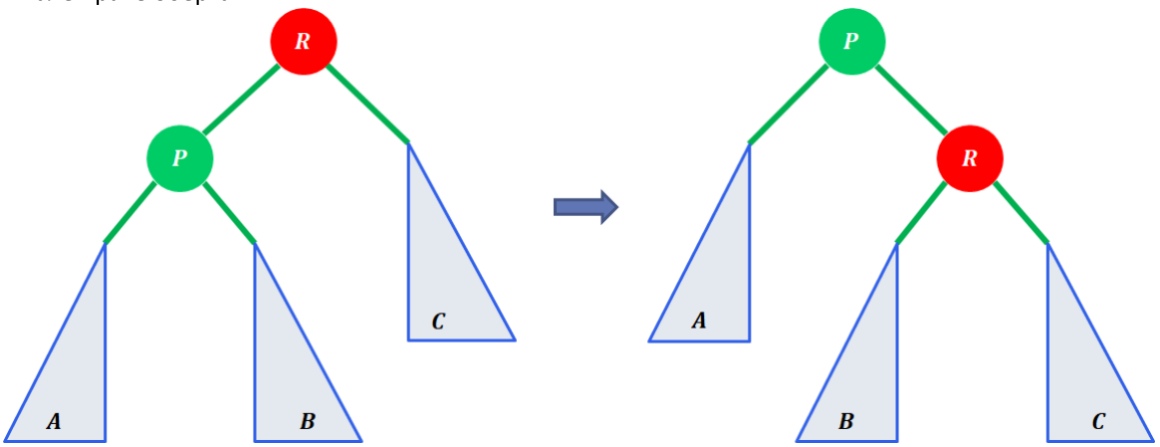


Рисунок 6.2.20. Мале праве обертання.

Використовується коли

$$h_P - h_C = 2 \\ h_B \leq h_A \qquad (6.2.2)$$

Алгоритмічно мале праве обертання складається з таких кроків:

a. переміщуємо вершину P, так, щоб вона стала коренем піддерева;
b. переміщуємо старий корінь піддерева – вершину вершину R – так щоб вона стала правим сином для вузла P;
c. переміщуємо старого правого сина вузла P – піддерево B – так, що він став лівим сином вузла R.

3. Велике ліве обертання.

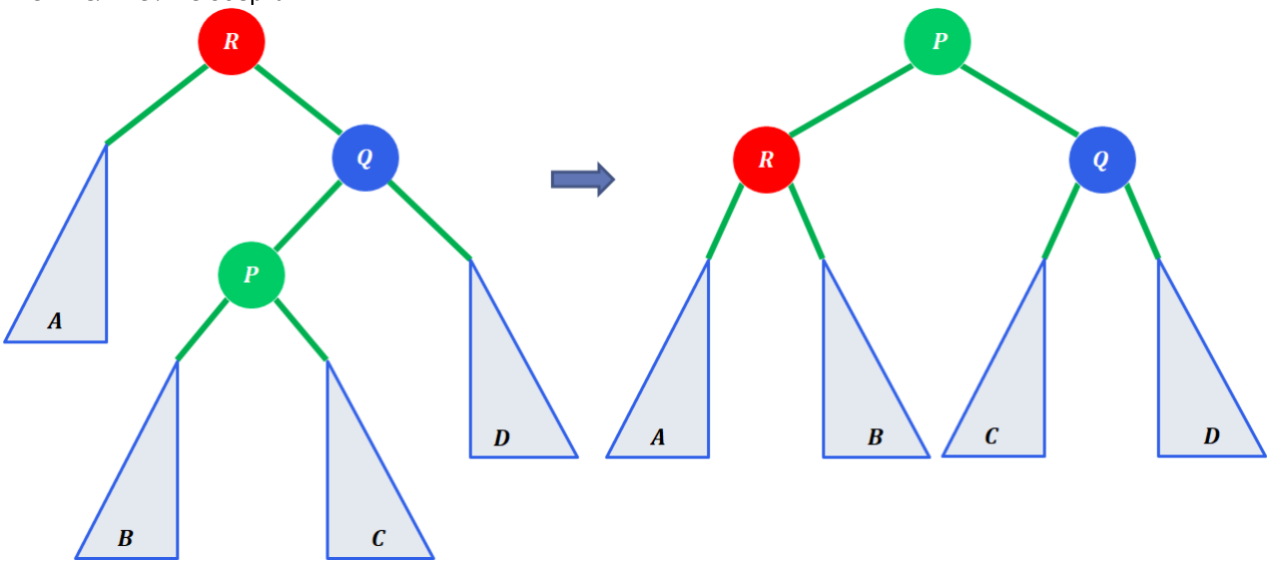


Рисунок 6.2.21.Велике ліве обертання.

Використовується коли

$$h_P - h_A = 2 \\ h_P > h_D \tag{6.2.3}$$

Алгоритмічно велике ліве обертання складається з таких кроків:

a. переміщуємо лівого сина вузла P, так, щоб він став правим сином вузла R;
b. переміщуємо правого сина вузла P, так, щоб він став лівим сином вузла Q;
c. переміщуємо вершину P, так, щоб вона стала коренем піддерева;
d. переміщуємо вузли R та Q, так щоб вони стали відповідно лівим та правим синами вузла P.

Якщо уважно проаналізувати велике ліве обертання, то можна прийти до висновку, що воно є послідовністю малих обертань – малого правого повороту навколо вершини P та малого лівого повороту навколо нової позиції вершини P. Тому, у подальшому, при необхідності здійснити велике ліве обертання, будемо використовувати відповідну послідовність малих обертань.

4. Велике праве обертання.

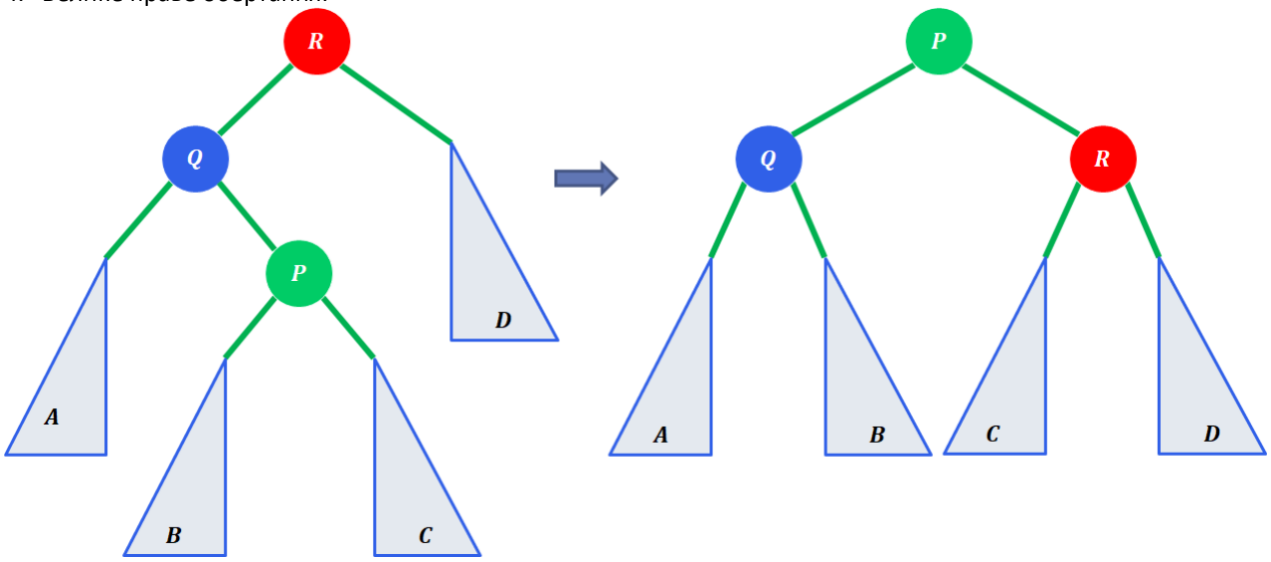


Рисунок 6.2.22.Велике праве обертання.

Використовується коли

$$h_Q - h_D = 2 \\ h_P > h_A \tag{6.2.4}$$

Вправа 6.2.2. Запишіть алгоритм переміщення вершин при великому правому обертанні.

Аналогічно, до великого лівого обертання, велике праве обертання є послідовністю малого лівого обертання навколо вершини Q та малого правого обертання навколо нової позиції вершини Q.

Вправа 6.2.3. Покажіть, що при кожній з операцій обертання, наведених вище, дерево залишається бінарним деревом пошуку.

Вправа 6.2.4. Покажіть, що при кожній з операцій обертання, наведених вище, повна висота дерева зменшується не більше ніж на 1 і не може збільшитися.

Отже, підсумовуючи, операція повороту у певній вершині здійснюються у випадку, якщо дерево у цій вершині є розбалансованим, при цьому якщо дерево переважує вліво, то здійснюється праве обертання, якщо дерево переважує вправо, то відповідно – ліве. При цьому, при лівому обертанні, аналізуються висоти лівого та правого піддерев вершини, що є правим сином розбалансованої вершини – якщо висота правого піддерева більша за висоту лівого то здійснюється велике ліве обертання, у іншому разі – мале. Аналогічно, при правому обертанні, якщо висота правого піддерева вузла, що є лівим сином розбалансованої вершини, більша за висоту лівого її піддерева, то здійснюється велике праве обертання, інакше – мале.

Розглянемо для прикладу дерево зображене нижче на рисунку 6.2.23. Як бачимо це дерево розбалансоване у вершині R і переважується вправо. Отже, здійснимо лівий поворот навколо вершини P. При цьому, як бачимо, оскільки вершина P немає лівого сина (тобто висота лівого піддерева вершини P є меншою за висоту правого піддерева), то для балансування необхідно здійснювати мале ліве обертання.

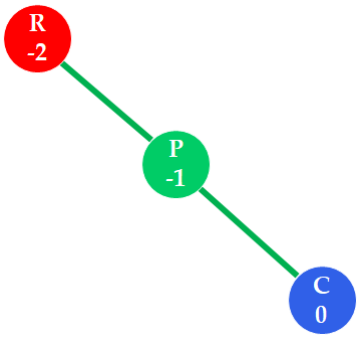


Рисунок 6.2.23.Розбалансоване дерево.

При такому повороті вершина P стане коренем, а вершина R – її лівим сином.

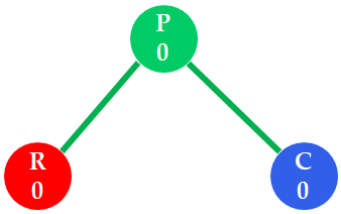


Рисунок 6.2.24.Збалансоване дерево після лівого повороту.

Як бачимо дерево стало збалансованим, оскільки фактори балансу усіх вузлів перетворилися у 0.

Розглянемо інший приклад. Розглянемо дерево зображене нижче на рисунку 6.2.25, яке розбалансоване у вершині R і переважується вліво. Здійснимо правий поворот навколо вершини P. При цьому, оскільки висота правого сина вершини P є меншою за висоту лівого сина, то здійснювати необхідно мале праве обертання.

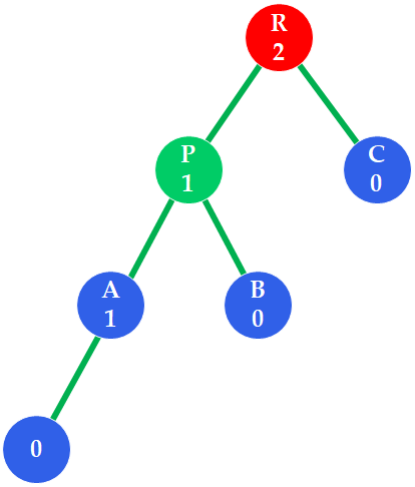


Рисунок 6.2.25.Розбалансоване дерево.

Пересунемо вершину P у корінь дерева, вершину R зробимо правим сином вершини P, а вершину B – лівим сином вершини R.

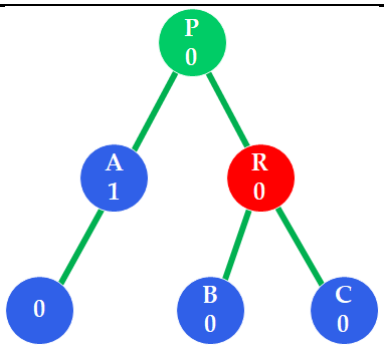


Рисунок 6.2.26. Збалансоване дерево після правого повороту.

Нарешті розглянемо приклад дерева (див рисунок 6.2.27), якому для балансування необхідно здійснити велике обертання. Як бачимо, це дерево розбалансоване у вершині R та, оскільки перевішує вліво, вимагає правого повороту. При цьому, оскільки висота дерева P більша за висоту піддерева A, то вимагається велике обертання.

Вправа 6.2.5. Покажіть, що при малому правому обертанні навколо вершини Q, дерево, зображене на рисунку 6.2.27 залишиться розбалансованим у вершині R.

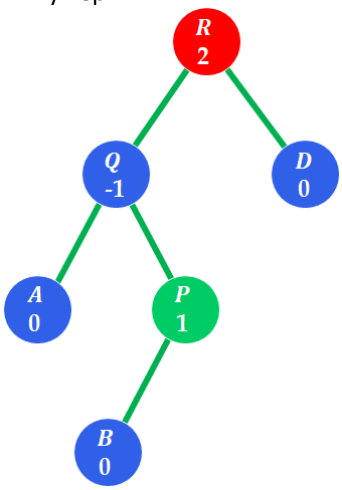


Рисунок 6.2.27. Розбалансоване дерево.

Велике обертання здійснимо у два кроки, спочатку мале ліве обертання навколо вершини P, після чого – мале праве обертання навколо нової позиці вершини P.

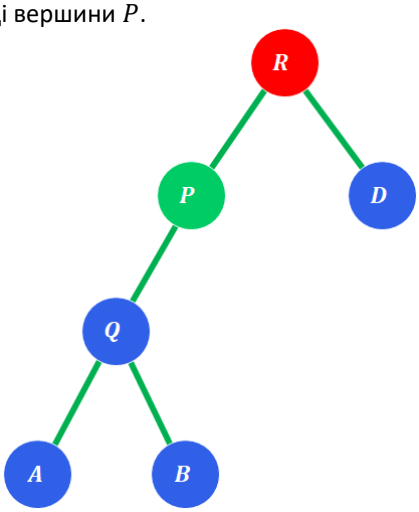


Рисунок 6.2.28. Мале ліве обертання навколо вершини P.

Звернемо увагу читача на те, що на цьому проміжному етапі баланс деяких вершин може збільшитися. Дійсно, баланс вершини P зараз дорівнює 2. Наступний поворот виправить ситуацію з балансом у цій вершині і балансом вершини R. Отже, здійснимо мале праве обертання навколо вершини P.

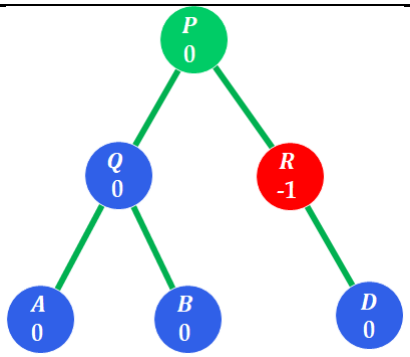


Рисунок 6.2.29. Збалансоване дерево після великого правого обертання.

Як бачимо з рисунка, дерево стало збалансованим в усіх вершинах.

Тепер, нарешті можемо записати код для функції **rebalance**(). Аргументом цієї функції буде вершина, яка втратила баланс. У цій функції спочатку відбувається перевірка яке – ліве чи праве – обертання необхідно провести. Для цього потрібно зрозуміти куди перевішує дерево. Тут, замість формул, що використовують висоти відповідних піддерев, зручніше користуватися фактором балансу. Дійсно, якщо фактор балансу є від'ємним, то дерево перевіщує вправо і вимагається ліве обертання, якщо ж фактор балансу є додатним, то дерево перевішує вліво, а значить вимагається праве обертання. Аналогічно діємо у випадку, якщо необхідно зрозуміти велике чи мале обертання вимагається.

Лістинг 6.2.7. Продовження. Метод перебалансування розбалансованої вершини.

```
    @staticmethod
    def rebalance(node):
        """ Здійснює балансування дерева у розбалансованій вершині.

        :param node: розбалансована вершина дерева
        """
        if node.mBalanceFactor < 0:  # дерево перевішує вправо
            if node.mRightChild.mBalanceFactor > 0:  # необхідне велике обертання
                AVLTree.rotateRight(node.mRightChild)
                AVLTree.rotateLeft(node)
            else:
                AVLTree.rotateLeft(node)
        elif node.mBalanceFactor > 0:  # дерево перевішує вправо
            if node.mLeftChild.mBalanceFactor < 0:  # необхідне велике обертання
                AVLTree.rotateLeft(node.mLeftChild)
                AVLTree.rotateRight(node)
            else:
                AVLTree.rotateRight(node)
```

Як бачимо метод **rebalance**(), використовує статичні методи **rotateLeft**() та **rotateRight**(), які, судячи з назв здійснюють відповідно мале ліве та мале праве обертання вершини.

Лістинг 6.2.7. Продовження. Мале ліве та праве обертрання вершин.

```
    @staticmethod
    def rotateLeft(node):
        """ Здійснює мале ліве обертання заданої вершини
        :param node: вершина, що обертається
        """
        node_parent = node.mParent
        if node.isLeftChild():
            node_parent.mLeftChild = AVLTree.__rotateLeft(node)
        elif node.isRightChild():
            node_parent.mRightChild = AVLTree.__rotateLeft(node)

    @staticmethod
    def rotateRight(node):
```

```
        """ Здійснює праве обертання заданої вершини
        :param node: вершина, що обертається
        """
        node_parent = node.mParent
        if node.isLeftChild():
            node_parent.mLeftChild = AVLTree.__rotateRight(node)
        elif node.isRightChild():
            node_parent.mRightChild = AVLTree.__rotateRight(node)
```

Як бачимо кожен з цих методів визначає, лівим чи правим сином є вузол для свого батька та змінює у батьківського вузла відповідного сина, вузлом, що є результатом роботи допоміжних статичних методів **__rotateLeft**() та **__rotateRight**() у яких власне й відбувається перебудова відповідних піддерев у результаті операцій обертання та обновлення балансу усіх вузлів, які брали участь в операції обертання. І, якщо алгоритм перебудови піддерева фактично має лінійну структуру і детально був розглянутий вище, то питання про те, як можна обновити фактори збалансованості вузлів без повного перерахунку висот нових піддерев є досить таки не тривіальним.

Розглянемо мале ліве обертання зображене на рисунку 6.2.19. Очевидно, що операція обертання змінить фактор збалансованості лише у вершинах R та P, адже операція обертання не змінює структури жодного з піддерев A, B та C.

Позначимо через $oldBal(R)$ та $newBal(R)$ – відповідно фактор збалансованості вершини R до обертання та після відповідно. Тоді очевидно

$$oldBal(R) = h_A - h_P = h_A - (1 + \max(h_B, h_C))$$
$$newBal(R) = h_A - h_B$$

Тоді

$$newBal(R) - oldBal(R) = h_A - h_B - h_A + (1 + \max(h_B, h_C)) =$$
$$= -h_B + 1 + \max(h_B, h_C) = 1 + \max(0,\ h_C - h_B) = 1 - min(0,\ h_B - \ h_C) =$$
$$= 1 - \min(0, oldBal(P))$$

Звідки отримаємо співвідношення між фактором збалансованості до та після обертання

$$newBal(R) = oldBal(R) + 1 - \min(0, oldBal(P))$$

Проведемо подібні викладки для визначення співвідношення між фактором збалансованості до та після обертання для вершини P.

$$oldBal(P) = h_B - h_C,$$
$$newBal(P) = h_R - h_C = 1 + \max(h_A, h_B) - h_C.$$

Тоді

$$newBal(P) - oldBal(P) = 1 + \max(h_A, h_B) - h_C - h_B + h_C =$$
$$= 1 + \max(h_A, h_B) - h_B = 1 + \max(h_A - h_B, 0) =$$
$$= 1 + \max(newBal(R), 0).$$

Отже,

$$newBal(P) = oldBal(P) + 1 + \max(newBal(R), 0)$$

Вправа 6.2.6. Виведіть співвідношення для зміни фактору збалансованості вершин R та P при правому обертанні зображеному на рисунку 6.2.20.

Отже, наведемо код методів **__rotateLeft**() та **__rotateRight**(), який завершує викладення цієї теми.

Лістинг 6.2.7. Продовження. Мале ліве та праве обертрання вершин.

```
@staticmethod
def __rotateLeft(root):
    """ Для піддерева заданим коренем здійснює мале ліве обертання

    :param root: корінь піддерева
    :return: новий корінь піддерева після операції обертання
    """
    pivot = root.mRightChild   # вершина, навколо якої здійснюється обертання
    root.mRightChild = pivot.mLeftChild

    if pivot.mLeftChild:
        pivot.mLeftChild.mParent = root

    pivot.mLeftChild = root
```

```
    node_parent = root.mParent
    root.mParent = pivot
    pivot.mParent = node_parent

    # Оновлення фактору збалансованості
    root.mBalanceFactor = root.mBalanceFactor + 1 - min(0, pivot.mBalanceFactor)
    pivot.mBalanceFactor = pivot.mBalanceFactor + 1 + max(0, root.mBalanceFactor)

    return pivot

@staticmethod
def __rotateRight(root):
    """ Для піддерева заданим коренем здійснює мале праве обертання

    :param root: корінь піддерева
    :return: новий корінь піддерева після операції обертання
    """
    pivot = root.mLeftChild  # вершина, навколо якої здійснюється обертання
    root.mLeftChild = pivot.mRightChild

    if pivot.mRightChild:
        pivot.mRightChild.mParent = root

    pivot.mRightChild = root

    node_parent = root.mParent
    root.mParent = pivot
    pivot.mParent = node_parent

    # Оновлення фактору збалансованості
    root.mBalanceFactor = root.mBalanceFactor - 1 - max(pivot.mBalanceFactor, 0)
    pivot.mBalanceFactor = pivot.mBalanceFactor - 1 + min(root.mBalanceFactor, 0)

    return pivot
```

Видалення вузлів АВЛ-дерева

Як і у випадку вставки нових вузлів до АВЛ-дерева, видалення вузлів АВЛ-дерева базується на відповідній операції видалення вузлів у бінарному дереві пошуку. Відмінність полягає лише в тому, що після операції видалення необхідно оновити фактор балансу у вузлах, які зачепило таке видалення. Здійснювати операцію оновлення балансу буде допоміжний метод

```
def updateBalanceOnDelete(node, came_from_left)
```

що буде детально описаний нижче.

Видалення вузла, у випадку, якщо він має лише одного сина, здійснюється заміною вузла, що видаляється – сином. Відтак замістимо у класі `AVLTree` метод **`setNode()`**, так, щоб фактор балансу переносився у вершину від сина

Лістинг 6.2.7. Продовження. Заміщення методу **`setNode()`**.

```
class AVLTree(SearchTree):
    ...
    def setNode(self, item):
        """ Змінює поточний вузол
        :param item: Нове піддерево або ключ
        """
        super().setNode(item)
        if isinstance(item, AVLTree):         # якщо item є АВЛ-деревом
            self.mBalanceFactor = item.mBalanceFactor
```

Для розвантаження класу `AVLTree` й спрощення сприйняття матеріалу, реалізацію операції видалення та допоміжних операцій здійснимо у нащадку класу `AVLTree`.

Операцію видалення здійснимо засобами двох методів:

- **`_delete_helper()`** – допоміжний рекурсивний метод, що видаляє з дерева вузол з заданим ключем, якщо такий ключ міститься у дереві;
- **`delete()`** – публічний метод, що викликає метод **`_delete_helper()`** для кореня дерева.

Лістинг 6.2.8. Клас AVLTreeWithDelete – АВЛ-дерево з операцією видалення.

```
class AVLTreeWithDelete(AVLTree):
    """ Розширення класу бінарного дерева можливістю видаляти елементи """

    def delete(self, key):
        """ Видаляє заданий елемент у бінарному дереві
        :param key: Елемент, який потрібно видалити з бінарного дерева
        """
        self._delete_helper(self, key)

    @staticmethod
    def _delete_helper(root, key):
        """ Допоміжний рекурсивний метод, що видаляє з дерева вузол з
            заданим ключем, якщо такий ключ міститься у дереві.
        :param root: корінь піддерева у якому потрібно видалити заданий елемент
        :param key: ключ елемента, який потрібно видалити
        """

        node = root.search(key)  # Знаходимо вузол, який треба видалити

        if node is None or node.mIsRoot:  # Якщо шуканий елемент не міститься у дереві,
            return                        # то припиняємо роботу підпрограми

        if node.hasNoChildren():          # Якщо знайдений вузол - листок (немає нащадків)
            isLeft = node.isLeftChild()   # Запам'ятаємо, яким сином по відношенню до
                                          # батьківського є вузол, що видаляється
            node.removeSelfFromParent()   # Видаляємо вузол
            AVLTreeWithDelete.updateBalanceOnDelete(node.mParent, isLeft)  # Оновлюємо баланс
                                                                           # батьківського вузла
        elif node.hasLeft() and not node.hasRight():  # Якщо знайдений вузол має лише ліву гілку
            node.setNode(node.mLeftChild)             # Замінюємо вузол його лівим піддіревом
            AVLTreeWithDelete.updateBalanceOnDelete(node.mParent, node.isLeftChild())

        elif node.hasRight() and not node.hasLeft():  # Якщо знайдений вузол має лише праву гілку
            node.setNode(node.mRightChild)            # Замінюємо вузол його правим піддіревом
            AVLTreeWithDelete.updateBalanceOnDelete(node.mParent, node.isLeftChild())

        else:                                         # Якщо знайдений вузол має обидві гілки
            left_max = AVLTreeWithDelete._search_max(node.mLeftChild)  # Знаходимо максимальний
                                                                       # вузол у лівому піддереві
            left_max_key = left_max.mKey
            node.setNode(left_max_key)   # Замінюємо значення елемета node знайденим максимальним
            AVLTreeWithDelete._delete_helper(node.mLeftChild, left_max_key)  # Видалення з лівого
                                                                             # піддерева найбільшого елементу
```

Як бачимо, метод **_delete_helper**() майже повністю повторює однойменний метод, що наведений у лістингу 6.2.6 з класу SearchTreeWithDelete. Різниця полягає лише у тому, що після операції видалення, явно, або не явно (у випадку якщо вузол, що видаляється має обидві дочірні гілки) викликається статичний метод **updateBalanceOnDelete**() для оновлення балансу. Подібно до методу **updateBalance**(), наведеного у лістингу 6.2.7, що використовувався нами для оновлення балансу під час вставки елементів у дерево, метод **updateBalanceOnDelete**() є рекурсивним методом, задачею якого є оновлення балансу усіх вузлів, починаючи із заданого вузла, рухаючись вгору по дереву до кореня. Проте, звернемо увагу читача, що на відміну від методу **updateBalance**(), рекурсивний виклик буде здійснюватися, якщо фактор балансу скорегувався до нуля, оскільки це значить, що дерево зменшило свою висоту і необхідно провести оновлення балансу для батька поточної вершини.

Лістинг 6.2.8. Продовження. Метод оновлення балансу.

```
class AVLTreeWithDelete(AVLTree):
    ...
    @staticmethod
    def updateBalanceOnDelete(node, came_from_left):
        """ Оновлює баланс для поточно вузла при операції видалення
        :param node: вузол у якому потрібно оновити баланс
```

```
        :param came_from_left: від якого, лівого чи правого сина ми піднялися у вузол
        """

        if node.mIsRoot:
            return

        # оновлюємо баланс вузла залежно від того з якого нащадка ми прийшли
        if came_from_left:            # якщо ми прийшли з лівого сина,
            node.mBalanceFactor -= 1  # то баланс у вузлі зменшується на 1
        else:                         # якщо ми прийшли з правого сина,
            node.mBalanceFactor += 1  # то баланс у вузлі збільшується на 1

        # Якщо після оновлення балансу, вузол розбалансувався
        if node.mBalanceFactor > 1 or node.mBalanceFactor < -1:
            AVLTree.rebalance(node)  # Проводимо балансування вузла
            node = node.mParent      # після балансування змінилася вершина за яку
                                     # була підвішена розбалансована вершина
        if node.mBalanceFactor == 0:
            # Якщо фактор балансу скорегувався до нуля,
            # це значить, що дерево зменшило свою висоту і необхідно провести
            # оновлення балансу для предка поточної вершини
            AVLTreeWithDelete.updateBalanceOnDelete(node.mParent, node.isLeftChild())
```

6.2.5. Двійкова купа та пріоритетна черга

У попередньому розділі ми вивчили структуру даних пріоритетна черга, елементи з якої, на відміну від звичайної черги, вилучаються не у тому порядку у якому додавалися, а залежно від деякого ключа (пріоритету) – чим вищий є пріоритет елемента по відношенню до інших елементів, що містяться у черзі, тим раніше він з неї буде вилучений. Ми навели просту реалізацію такої черги на базі вбудованого списку. Така реалізація має один суттєвий недолік – вона працює дуже повільно. Дійсно, хоча час вставки елемента до неї сталий ($O(1)$), проте час отримання елементу з неї є лінійний ($O(n)$, якщо черга містить n елементів). Існує цілий клас задач, які будуть розглянуті пізніше, що розв'язуються із застосуванням пріоритетної черги. Проте така швидкість роботи пріоритетної черги є незадовільної для їхньої оптимальної роботи. Тому постає природнє запитання як можна реалізовувати пріоритетну чергу, що буде працювати швидше.

Класичний спосіб реалізації пріоритетної черги, час вставки та отримання елементів з якої є швидшим за лінійний є використання структури даних, що називається **двійкова купа**. Ця структура даних дозволяє вставляти та отримувати елементи за логарифмічний час ($O(\log n)$, якщо черга містить n елементів).

Двійкова купа є бінарним деревом, проте її реалізація використовує звичайний лінійний список – як пізніше буде показано для реалізації такої структури не потрібно використання механізмів посилань на дочірні елементи вузла і відповідно для дітей не потрібно посилань на їхніх батьків.

Означення 6.2.7. Двійкова (бінарна) купа (англ. binary heap) – це бінарне дерево, для якого виконуються такі умови:

1) будь-яке її (ліве чи праве) піддерево є двійковою купою;
2) значення ключа будь-якого вузла є не меншим за значення ключів його дітей (структурна властивість купи).
3) глибина усіх листків дерева відрізняється не більше ніж на 1.
4) при додаванні нових вузлів, останній шар (листків) заповнюється зліва направо (без «дірок»).

Враховуючи вищенаведене означення, можна прийти до висновку, що двійкова купа може бути зображена у вигляді піраміди у вершині якої розташовується найменший елемент. Очевидно, з означення випливає, що відповідне твердження має виконуватися для всіх її піддерев. На рисунку 6.2.30 зображено бінарну купу, у вершині якої зберігається найменший елемент 2.

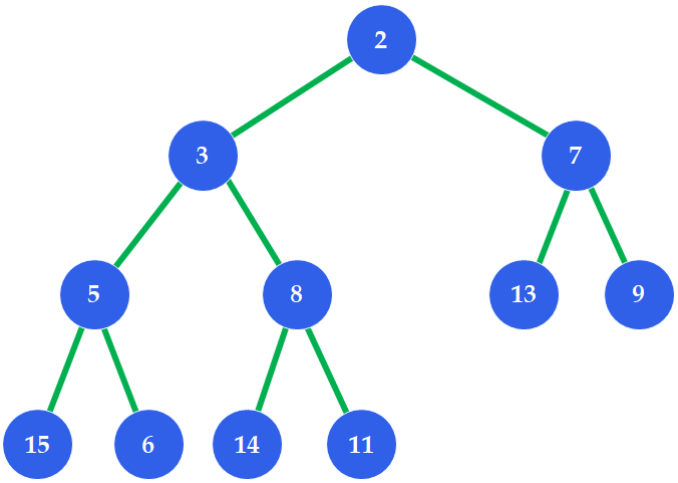


Рисунок 6.2.30. Двійкова купа – найменший елемент знаходиться у корені.

Зауважимо, що у другому пункті означення фразу «не меншим», часто замінюють фразою «не більшим». У такому разі двійкова купа лишається пірамідою у корені якої зберігається найбільший елемент структури.

Операції з двійковою купою

Реалізація двійкової купи передбачає реалізацію таких операцій

1. Створити купу.
2. Операція **`empty`**`()` – чи порожня купа.
3. Операція **`insert`**`(key)` – вставити ключ `key` у купу.
4. Операція **`extract_minimum`**`()` – отримати (з вилученням) з купи найменший ключ.

При цьому операції **`insert`** та **`extract_minimum`** повинні виконуватися за час $O(\log n)$, де n кількість елементів, що міститься у купі.

Крім зазначених вище операцій додатково реалізують ще дві операції, що будуть корисні при реалізації пріоритетної черги:

5. Операція **`update`**`(key, newkey)` – замінити ключ `key` у купі новим ключем `newkey`.
6. Операція **`get_minimum`**`()` – взяти (без вилучення) з купи найменший ключ.

Розглянемо операцію **`insert`**`(key)` на прикладі вставки нового елемента у купу зображену на рисунку 6.2.30. Припустимо, що необхідно вставити елемент з ключем 4. Згідно з означенням, цей елемент необхідно вставити як лівого сина вузла 13 (див рисунок 6.2.31).

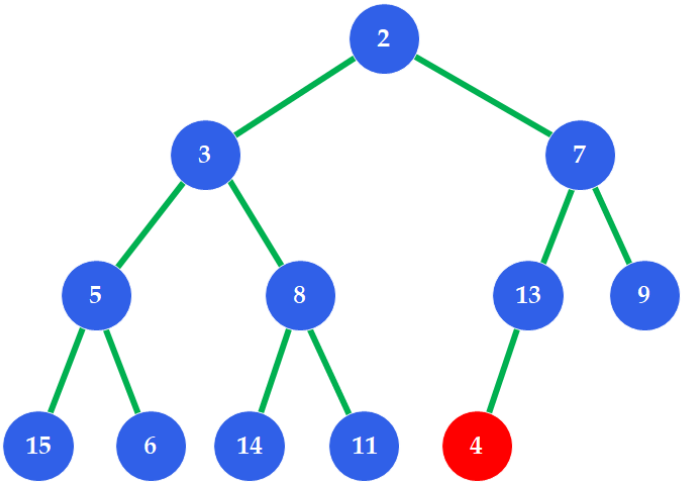


Рисунок 6.2.31. Вставка у купу елемента 4.

Проте, просто здійснити операцію вставки зазначеним способом буде не достатньо. Дійсно така вставка зруйнувала купу як структуру, оскільки порушується пункт 2) означення купи – як бачимо вузол 13 є більшим за вузол 4. Тому після вставки елемента у купу здійснюється операція його «просіювання» вгору. Воно полягає у тому, що вставлений елемент порівнюється з його батьком і якщо він менший за батька – елементи міняються місцями. Ця операція здійснюється доти доки елемент не займе потрібної позиції і, відповідно, не відновиться структурна властивість купи. Таким чином вставлений елемент може «просіюватися» вгору аж до кореня. Отже на першій ітерації міняються місцями вузли 4 та 13.

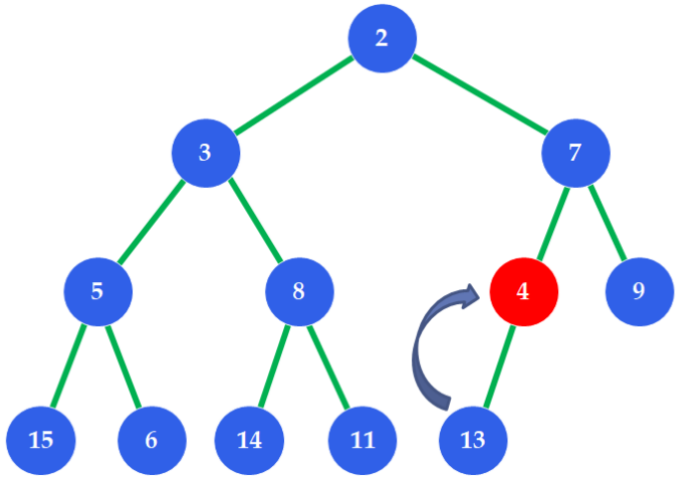


Рисунок 6.2.32. «Просіювання» елемента 4 вгору.

Друга ітерація і вона ж остання для цієї вставки міняє місцями вузли 4 та 7. Як бачимо структурна властивість купи відновилася.

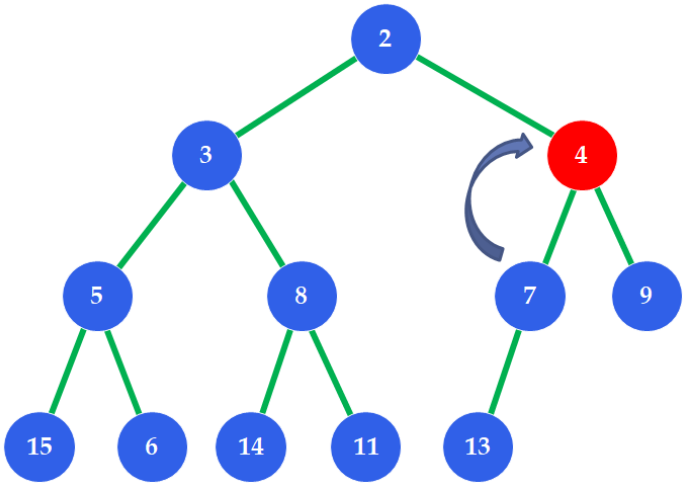


Рисунок 6.2.33. «Просіювання» елемента 4 вгору. Відновлено структурну властивість купи.

Перейдемо до операції **`extract_minimum`**( ) вилучення найменшого елемента. Як ми знаємо з властивостей купи, її найменший елемент знаходиться у корені. Тому операція його відшукання є тривіальною. Проте, просте вилучення цього елемента руйнує купу як деревовидну структуру – двійкова купа розпадається на два незалежних дерева, що є неприпустимо.

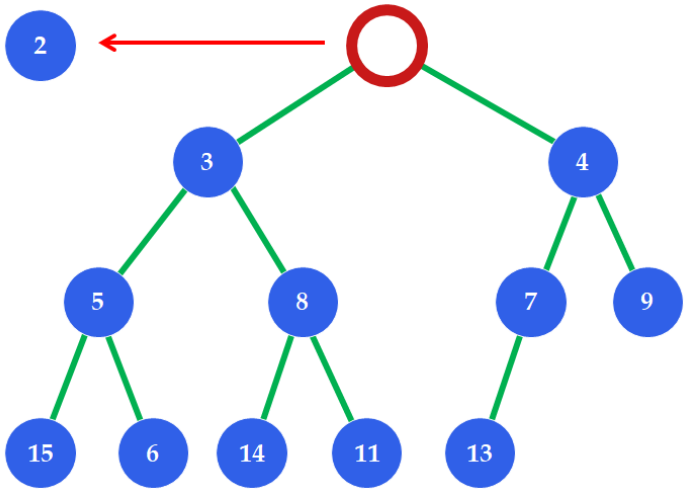


Рисунок 6.2.34. Просте вилучення мінімуму руйнує купу як деревовидну структуру.

Тому постає питання: як видалити корінь дерева та зберегти бінарну купу? Відповідь є очевидною – потрібно заповнити корінь іншим вузлом. Причому також очевидно, що для того, щоб виконувалася умова 4) означення двійкової купи, потрібно переставити на місце кореня вузол, що на останньому рівні займає крайнє праве положення. У нашому випадку це вузол 13:

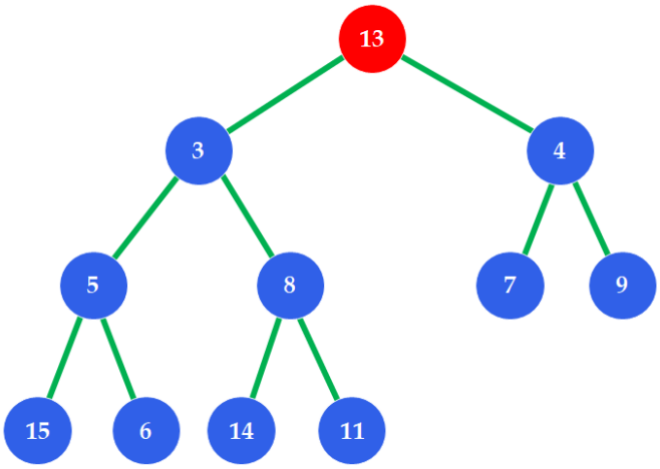


Рисунок 6.2.35. Переставляємо на місце кореня вузол 13.

Очевидно, що після такої перестановки необхідно провести відновлення структурної властивості купи. Здійснюється таке відновлення за допомогою операції «просіювання вниз» вставленого вузла. Вона полягає у тому, щоб, рухаючись від кореня до листків опустити вставлений елемент на таку позицію, щоб структурна властивість. Здійснюється вона у два кроки:

1) Для елемента, що був вставлений шукаємо мінімум з двох його дітей.
2) Якщо цей мінімум менший за батьківський елемент – міняємо їх місцями.

Таким чином, для вузла 13 менший з його двох синів є вузол 3. Він менший за 13, відповідно вузол 13 «просіюється вниз» міняючись місцями з вузлом 3. На наступному кроці, для вузла 13 меншим з його дітей є вузол 5.

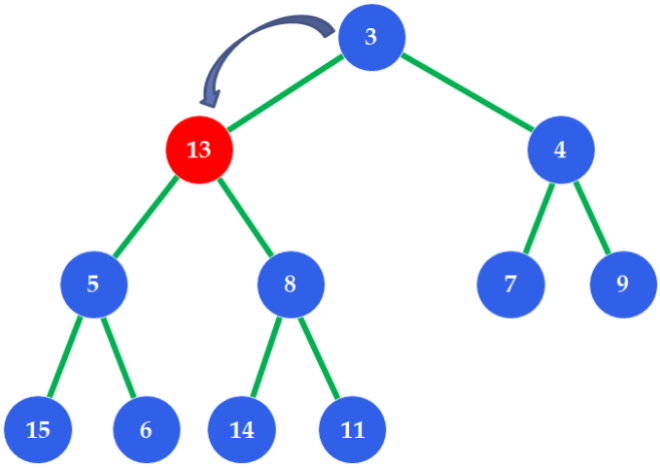


Рисунок 6.2.36. Вузол 13 «просіюється вниз» міняючись місцями з вузлом 3 – меншим з його синів.

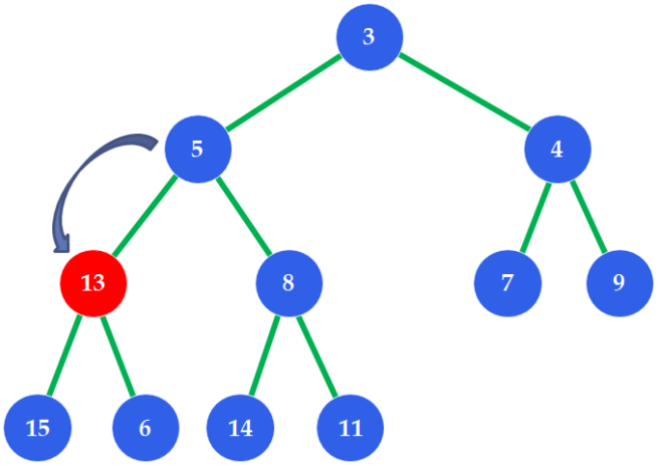


Рисунок 6.2.37. Вузол 13 «просіюється вниз» міняючись місцями з вузлом 5 – меншим з його синів.

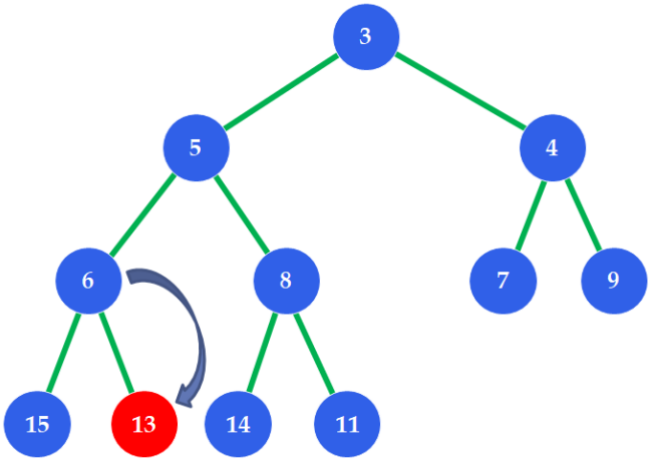


Рисунок 6.2.38. Відновлено структурну властивість купи.

Проаналізувавши наведені вище алгоритми вставки та вилучення елементів для бінарної купи, очевидним є, що якщо купа містить n елементів, то висота бінарного дерева, що її моделює не перевищує $\log n$. Цей факт доводить те, що швидкість роботи цих операцій має складність $O(\log n)$.

Повне та майже повне бінарні дерева.

Перш ніж перейти до реалізації двійкової купи, розглянемо спеціальний вид дерев.

Означення 6.2.8. Бінарне дерево називається повним бінарним деревом, якщо всі його вузли мають рівно двох дітей, крім листків, що розташовуються на однаковій глибині.

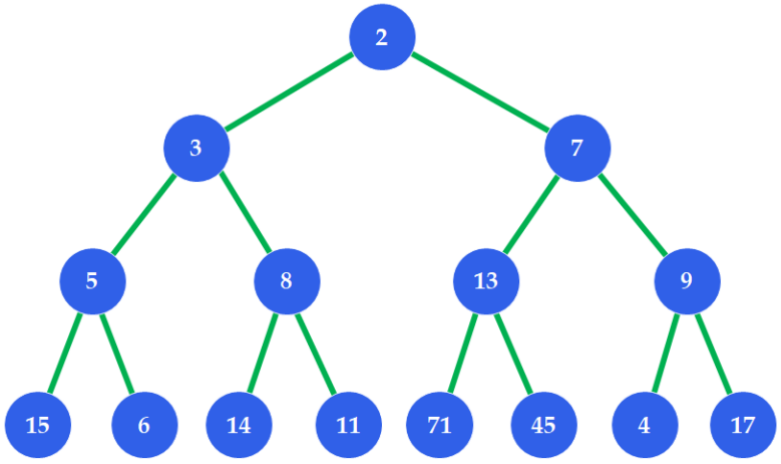


Рисунок 6.2.39. Повне бінарне дерево.

Як бачимо, у повному бінарному дереві всі його рівні повністю заповнені. Власне це і дано назву такому типу дерев.

Розглянемо детальніше властивості такого дерева. Легко бачити, що повне бінарне дерево на нульовому рівні має один вузол – корінь. Його перший рівень має вже два вузли – лівого та правого сина кореня. Очевидно, що другий рівень буде мати в два рази більше вузлів ніж перший – чотири. Продовжуючи такі міркування, можемо прийти до висновку, що n-й рівень повного бінарного дерева матиме 2^n вузлів. Тепер занумеруємо всі вузли повного бінарного дерева (вважаючи, що корінь має номер 1) у порядку пошуку в ширину.

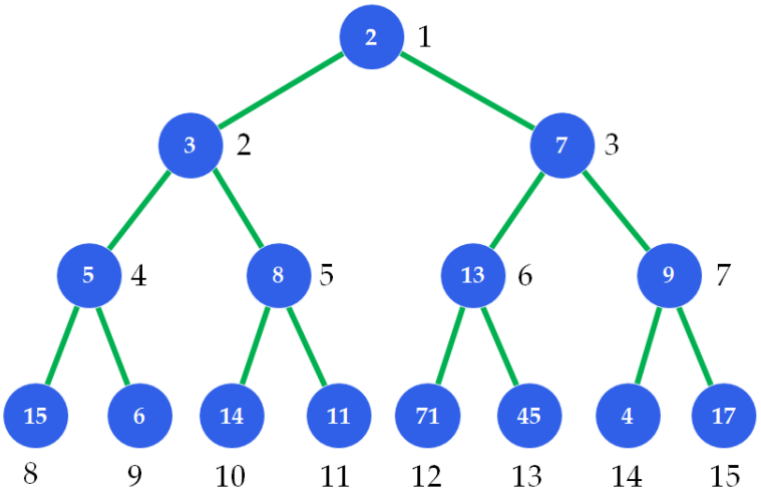


Рисунок 6.2.40. Нумерація вузлів повного бінарного дерева.

Отже, як бачимо з рисунка, на нульовому рівні розташовуються корінь з індексом 1. На першому рівні – вузли з індексами 2 та 3. На другому рівні вузли з номерами що належать проміжку $[4, 7]$, на третьому з проміжку $[8, 15]$. Продовжуючи ці міркування, можемо прийти до висновку, що n-й рівень повного бінарного дерева буде містити вершини індекси яких змінюються в діапазоні $[2^n, 2^{n+1} - 1]$. Крім цього, можна помітити, що якщо деякий вузол має номер (будемо називати його надалі індекс) n, то його діти мають індекси $2n$ та $2n + 1$ для лівого та правого сина відповідно. Батько ж цього вузла матиме індекс, що дорівнює результату цілочисельного ділення n на 2: $[n/2]$. Ця властивість повного бінарного дерева дає змогу реалізувати цю структуру даних без використання рекурсивних підходів (вузлів та посилань на дітей/батьків). Дійсно, для реалізації такої структури можна використати звичайний масив:

	корінь	1-й рівень		2-й рівень				3-й рівень							
–	2	3	7	5	8	13	9	15	6	14	11	71	45	4	17
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Рисунок 6.2.41. Зображення повного дерева за допомогою масиву.

Зауважимо, що при зображенні повного бінарного дерева у вигляді списку, індекси масиву відповідають номерам вузлів дерева починаючи з 1. Нульовий елемент масиву не несе жодного навантаження (фіктивний елемент). Такий підхід використовується для того, щоб уникнути проблеми з множенням на нуль при відшуканні індексів дітей.

Крім повних бінарних дерев використовуються майже повні бінарні дерева.

Означення 6.2.9. Бінарне дерево називається **майже повним бінарним деревом**, якщо дерево утворене в результаті вилучення всіх вузлів останнього рівня є повним бінарним деревом.

Бінарне дерево зображене на рисунку 6.2.30 є майже повним бінарним деревом. Як бачимо, у майже повного бінарного дерева всі рівні крім останнього заповнені повністю. З огляду на вищенаведене означення, можна стверджувати, що двійкова купа є майже повним бінарним деревом. Відповідно її реалізацію зручно здійснювати використовуючи масив (індексований список).

Реалізація двійкової купи

Лістинг 6.2.9. Реалізація структури даних Купа.

```
class Heap:
    """ Клас структура даних Купа """
```

```
    def __init__(self):
        """ Конструктор """
        self.mItems = [0]
        self.mSize = 0

    def empty(self):
        """ Перевіряє чи купа порожня

        :return: True, якщо купа порожня
        """
        return len(self.mItems) == 1

    def insert(self, item):
        """ Вставка елемента в купу
        :param item: - Елемент, що вставляється у купу
        """
        self.mSize += 1
        self.mItems.append(item)  # Вставляємо на останню позицію,
        self.siftUp()             # просіюємо елемент вгору

    def extractMinimum(self):
        """ Повертає мінімальний елемент кучі
        :return: Мінімальний елемент кучі
        """
        root = self.mItems[1] # Запам'ятовуємо значення кореня дерева
        self.mItems[1] = self.mItems[-1]  # Переставляємо на першу позицію
                                          # останній елемент (за номером) у купі
        self.mItems.pop()  # Видаляємо останній (за позицією у масиві) елемент купи
        self.mSize -= 1
        self.siftDown()    # Здійснюємо операцію просіювання вниз, для того,
                           # щоб опустити переставлений елемент на відповідну позицію у купі
        return root        # повертаємо значення кореня, яке було запам'ятовано на початку

    def siftDown(self):
        """ Просіювання вниз """
        i = 1
        while (2 * i) <= self.mSize:
            left = 2 * i
            right = 2 * i + 1
            min_child = self.minChild(left, right)
            if self.mItems[i] > self.mItems[min_child]:
                self.swap(min_child, i)
            else:
                break

            i = min_child

    def siftUp(self):
        """ Дпопоміжний метод просіювання вгору """
        i = len(self.mItems) - 1
        while i > 1:
            parent = i // 2
            if self.mItems[i] < self.mItems[parent]:
                self.swap(parent, i)
            else:
                break

            i = parent

    def swap(self, i, j):
        """ Допоміжний метод для перестановки елементів у купі,
        що знаходяться на заданих позиціях i та j
        :param i: перший індекс
        :param j: другий індекс
        """
        self.mItems[i], self.mItems[j] = self.mItems[j], self.mItems[i]

    def minChild(self, left_child, right_child):
        """ Допоміжна функція знаходження меншого (за значенням) вузла серед нащадків поточного
```

```
        :param left_child: лівий син
        :param right_child: правий син
        :return: менший з двох синів
        """
        if right_child > self.mSize:
            return left_child
        else:
            if self.mItems[left_child] < self.mItems[right_child]:
                return left_child
            else:
                return right_child
```

Застосування двійкової купи

Найпоширенішими застосуваннями двійкової купи є алгоритм швидкого сортування на базі двійкової купи – пірамідальне сортування (англ. heapsort) та реалізація структури даних, розглянутої вище у розділі 5, пріоритетна черга.

Алгоритм сортування на базі купи

Уважний читач може помітити, що застосування двійкової купи дозволяє отримати алгоритм сортування, асимптотичний час роботи якого (у найгіршому випадку, власне які у найкращому) буде складати $O(n \log n)$ для вхідного масиву, що складається з n елементів. Дійсно, для цього досить додати всі елементи вхідного масиву у купу, а потім по одному забрати їх звідти. Проте така реалізація вимагатиме використання додаткової пам'яті для організації роботи двійкової, а також додаткового часу на копіювання елементів спочатку у купу, а потім назад з неї у вхідний масив. Тому наведемо реалізацію алгоритму сортування на базі двійкової купи, що буде модифікувати сам вхідний масив даних.

Алгоритм пірамідального сортування складається з двох кроків. На першому кроці, для вхідного масиву даних здійснюється відновлення структури двійкової купи. Відновлення полягає у тому, що для всіх внутрішніх вузлів дерева двійкової купи здійснюється просіювання вниз. При цьому, у випадку, якщо вхідний масив необхідно відсортувати за зростанням (не спаданням) елементів, то просіювання здійснюється так, щоб найбільший елемент структури знаходився у корені дерева. У реалізації двійкової купи, наведеної вище у лістингу 6.2.9, просіювання здійснювалося так, щоб у вершині піраміди опинився найменший елемент.

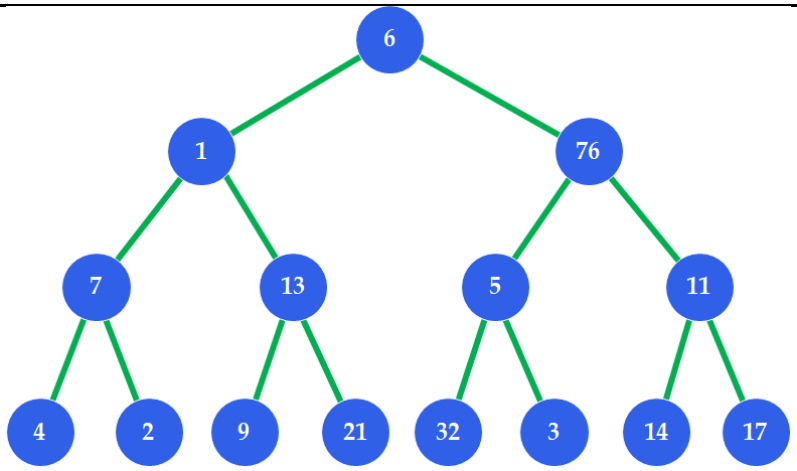
Другий крок алгоритму сортування полягає у тому, що найбільший елемент двійкової купи міняється місцями з останнім елементом масиву – який займає свою позицію у відсортованому масиві. Після цього, для елемента, що опинився у вершині піраміди здійснюється просівання вниз (не зачіпаючи елемент масиву, що опинився на останній позиції). Далі міняємо місцями елемент у вершині з передостаннім елементом масиву та здійснюємо просіювання вниз. Цю процедуру здійснюємо доти, доки всі елементи не займуть свої позиції у відсортованому масиві.

Зауваження. Звернемо увагу читача, що оскільки індекси вхідного масиву починаються з нуля, а не з одиниці, на відміну від наведеної вище у лістингу 6.2.9 реалізації двійкової купи, то, при моделюванні структури даних купа за допомогою цього ж масиву, для поточного вузла i індекси його лівого та правого синів будуть $2i + 1$ та $2i + 2$.

Розглянемо для прикладу масив зображений нижче.

6	1	76	7	13	5	11	4	2	9	21	32	3	14	17

Якщо припускати, що цей масив моделює повне бінарне дерево, то його елементи розміщуються по рівнях так як зображено на рисунку 6.2.42



Корінь	1-й рівень		2-й рівень				3-й рівень							
6	1	76	7	13	5	11	4	2	9	21	32	3	14	17
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Рисунок 6.2.42. Повне бінарне дерево.

Як бачимо, дерево зображене на рисунку 6.2.42 не є бінарною купою, тому для внутрішніх вузлів (тих що не є листками, тобто вузлів з індексами від 0 до 6) здійснюємо просіювання вниз.

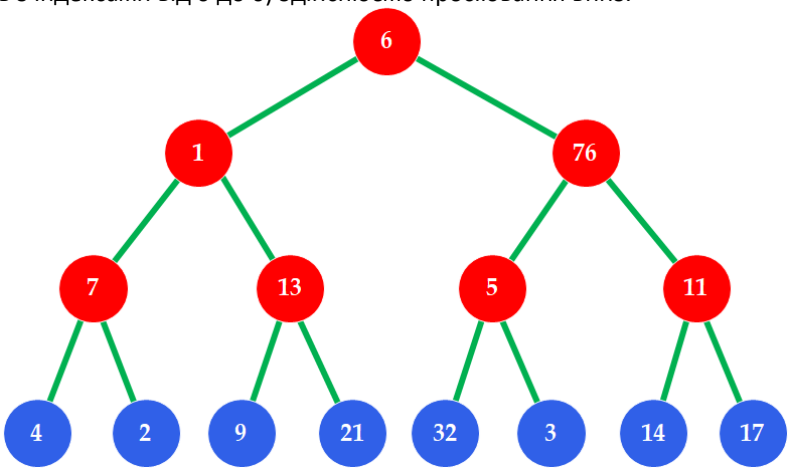
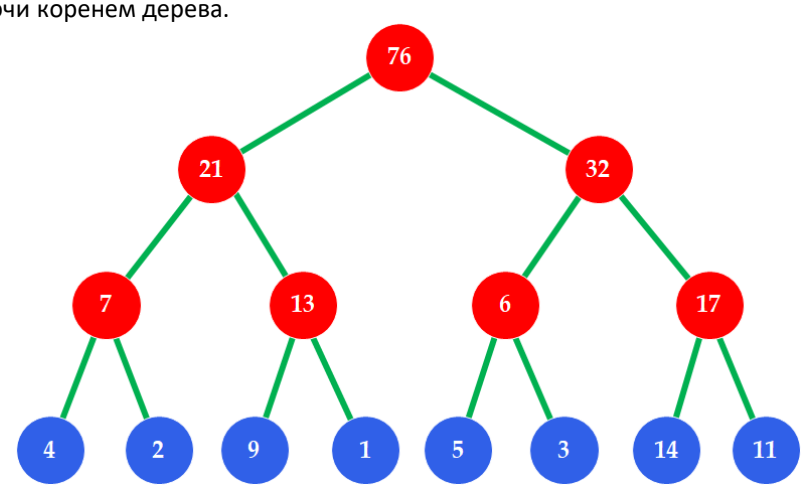


Рисунок 6.2.43. Вузли, що необхідно просіяти вниз.

При цьому раніше будемо здійснювати просіювання вузлів зі старшими індексами – починаючи з вершини з індексом 6, закінчуючи коренем дерева.



Корінь	1-й рівень		2-й рівень				3-й рівень							
76	21	32	7	13	6	17	4	2	9	1	5	3	14	11
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Рисунок 6.2.44. Дерево стає бінарною купою, після просіювання.

Отже, вхідний масив перетворений у бінарну купу, а отже його найбільший елемент знаходиться у корені. Міняємо у масиві місцями 0-й елемент з останнім та виключаємо його з розгляду, як елемент бінарної купи – він зайняв свою позицію у відсортованій частині масиву.

Корінь	1-й рівень		2-й рівень				3-й рівень							
11	21	32	7	13	6	17	4	2	9	1	5	3	14	76
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

При цьому руйнується структура дерева, як бінарної купи – елемент 11 не є найбільшим елементом купи. Здійснюємо просіювання вниз елемента 11. У результаті просіювання, відновлюється структура купи, а її найбільший елемент 32 опиняється у корені.

Корінь	1-й рівень		2-й рівень				3-й рівень							
32	21	17	7	13	6	14	4	2	9	1	5	3	11	76
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Міняємо його місцями з передостаннім елементом

Корінь	1-й рівень		2-й рівень				3-й рівень							
11	21	17	7	13	6	14	4	2	9	1	5	3	32	76
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

та повторюємо процедуру просіювання для елемента, що опинився у корені для відновлення структурної властивості купи

Корінь	1-й рівень		2-й рівень				3-й рівень							
21	13	17	7	11	6	14	4	2	9	1	5	3	32	76
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Повторюючи цю процедуру доти, доки для розгляду у купі не лишиться лише один елемент, що й буде означати, що масив відсортовано.

Наведемо реалізацію алгоритму. Вона буде складатися з двох функцій – власне функції сортування та допоміжної функції просіювання вниз.

Лістинг 6.2.10. Пірамідальне сортування.

```
def heapSort(array):
    """ Пірамідальне сортування вхідного масиву
    :param array: вхідний масив даних
    """

    size = len(array)
    # Відновлення стутуктури двійкової купи для вхідного масиву даних
    # Для всіх внутрішніх вузлів дерева двійкової купи
    # (для вузлів з індексами [0, size / 2] )
    for i in range(size // 2 - 1, -1, -1):
        siftDown(array, i, size)  # здійснюємо просіювання вниз

    # Для всіх внутрішніх вузлів дерева двійкової купи
    for i in range(size - 1, 0, -1):
        # Преставляємо найбільший елемент у кінець
        array[0], array[i] = array[i], array[0]
        # тепер частина масиву з індексами [i, size - 1] відсортована
        # Відновлюємо структуру двійкової купи для частини масиву [0, i - 1]
        siftDown(array, 0, i)  # просіюванням елементу з індексом 0 вниз

def siftDown(array, start, end):
    """ Функція просіювання елементів двійкової купи вниз
    Здійснює просіювання елементів двійкової купи
    у діапазоні [start, end - 1] так, що найбільший елемент
    найбільший елемент опиняється у позиції start
    :param array: Вхідний масив, що моделює двійкову купу
    :param end:   Розмір двійкової купи
    :param start: Початковий індекс
    """
```

```
    while True:
        # Визначаємо лівого та правого синів поточного вузла
        # Індекси масиву починаються від нуля, тому
        left = start * 2 + 1
        right = left + 1

        largest = start
        if left < end and array[left] > array[largest]:
            largest = left
        if right < end and array[right] > array[largest]:
            largest = right
        if largest == start:
            break

        array[start], array[largest] = array[largest], array[start]
        start = largest
```

Звернемо увагу читача, що на відміну від вивчених раніше швидких алгоритмів сортування, алгоритм пірамідального сортування не є рекурсивним та практично взагалі не вимагає додаткової пам'яті для своєї роботи.

Реалізація структури даних пріоритетна черга

На базі двійкової купи легко реалізувати структуру даних пріоритетна черга. Для цього необхідно замість елементів у купі зберігати пари – ключ (ідентифікатор елементу) та його пріоритет. При цьому

Для цього реалізуємо допоміжний клас `PQElement`, що є структурою з двох елементів – ключ (елемент) та його пріоритет.

Лістинг 6.2.11. Елемент пріоритетної черги.

```
INF = sys.maxsize  # Умовна нескінченність

class PQElement:
    """ Клас Елемент пріорітетної черги """

    def __init__(self, key=None, priority=INF):
        """ Конструктор

        :param key: Ключ (ім'я) елементу
        :param priority: Пріорітет
        """
        self.mKey = key
        self.mPriority = priority

    def updatePriority(self, priority):
        """ Змінює пріоритет для елементу

        :param priority: Пріорітет
        :return: None
        """
        self.mPriority = priority

    def key(self):
        """ Повертає ключ елемента

        :return: Ключ елемента
        """
        return self.mKey

    def __le__(self, other):
        """ Перевизначає оператор '<='

        :param other: інший елемент
        :return: True, якщо пріоритет поточного елементу менший або рівний за пріоритет іншого
        """
        return self.mPriority <= other.mPriority
```

```
    def __lt__(self, other):
        """ Перевизначає оператор '<'

        :param other: інший елемент
        :return: True, якщо пріоритет поточного елементу менший за пріоритет іншого
        """
        return self.mPriority < other.mPriority

    def __gt__(self, other):
        """ Перевизначає оператор '>'

        :param other: інший елемент
        :return: True, якщо пріоритет поточного елементу більший за пріоритет іншого
        """
        return self.mPriority > other.mPriority

    def __ge__(self, other):
        """ Перевизначає оператор '>='

        :param other: інший елемент
        :return: True, якщо пріоритет поточного елементу більший або рівний за пріоритет іншого
        """
        return self.mPriority >= other.mPriority
```

Тепер наведемо реалізацію пріоритетної черги без додаткових пояснень (за виключенням тих, що розміщені у коментарях програми).

Лістинг 6.2.12. Пріоритетна черги на базі двійкової купи.

```
class PriorityQueue:
    """ Клас пріоритетна черга на базі структури даних Купа """

    def __init__(self):
        """ Конструктор """

        self.mItems = [PQElement(0, 0)]
        self.mSize = 0
        self.mElementsMap = {}  # Карта індексів (у масиві, що моделює чергу) елеменітів черги.
                                # Є словник з елементів (елемент, індекс)
                                # Використовується для визначення чи міститься елемент у черзі
                                # а також для швидкої зміни пріорітеру елемента у черзі

    def insert(self, key, priority):
        """ вставки пари: (елемент, пріоритет)
        :param key:      елемент
        :param priority: пріоритет
        :return:
        """
        el = PQElement(key, priority)
        self.mSize += 1
        self.mItems.append(el)                  # Вставляємо на останню позицію,
        self.mElementsMap[key] = self.mSize    # S
        self.siftUp()          # просіюємо елемент вгору

    def extractMinimum(self):
        """ Повертає елемент черги з найвищим пріоритетом
        у цій реалізації пріоритетної черги вважається чим меншим є значення
        поля пріоритету тим вищим є пріоритет елементу у черзі.
        :return: Елемент черги з найвищим пріоритетом
        """
        root = self.mItems[1].key()         # Запам'ятовуємо значення кореня дерева
        self.swap(1, -1)  # Переставляємо на першу позицію останній елемент (за номером) у купі
        self.mItems.pop()               # Видаляємо останній (за позицією у масиві) елемент купи
        del self.mElementsMap[root]     # Видаляємо елемент з мапи елементів
        self.mSize -= 1
        self.siftDown()   # Здійснюємо операцію просіювання вниз, для того,
                          # щоб опустити переставлений елемент на відповідну позицію у купі
        return root
```

```
    def swap(self, i, j):
        """ Перевизначення методу батьківського класу обміну місцями елементів
            на позиціях i та j у черзі із простеженням позиції елемента у черзі.
        :param i: перший індекс
        :param j: другий індекс
        :return: None
        """
        pos_i = self.mItems[i].key()  # Поточна позиція елемента i у масиві
        pos_j = self.mItems[j].key()  # Поточна позиція елемента j у масиві
        self.mElementsMap[pos_i] = j
        self.mElementsMap[pos_j] = i
        self.mItems[i], self.mItems[j] = self.mItems[j], self.mItems[i]

    def __contains__(self, item):
        """ Перевизначає оператор 'in'

        :param item: Ключ
        :return: True, якщо ключ міститься у черзі
        """
        return item in self.mElementsMap

    def updatePriority(self, key, priority):
        """ Метод перерахунку пріоритету елемента.
        Працює лише у випадку підвищення пріоритету у черзі, тобто якщо
        значення параметру priority є меншим ніж поточне значення пріоритету
        Працює по принципу, замінюємо пріоритет елемента у черзі та здійснюємо просіювання вгору.

        :param key: Ключ
        :param priority: Новий пріоритет
        :return: True
        """
        i = self.mElementsMap[key]
        self.mItems[i].updatePriority(priority)
        # просіювання вгору для елемента зі зміненим пріоритетом
        while i > 1:
            parent = i // 2
            if self.mItems[i] < self.mItems[parent]:
                self.swap(parent, i)
            else:
                break
            i = parent
        return True
```

Звернемо увагу читача на той факт, що методи,

```
    def empty(self):
    def siftDown(self):
    def siftUp(self):
    def minChild(self, left_child, right_child):
```

які явно або неявно зустрічаються у нашій реалізації пріоритетної черги дослівно повторюють однойменні методи реалізації двійкової купи та були наведені у Лістингу 6.2.9. Тому, ми опустили їхній опис у цьому лістингу.

6.2.6. Дерево відрізків

Нехай задано деякий індексований масив даних $a = \{a_i : i \in [0, n]\}$.

Означення 6.2.10. Дерево відрізків – структура даних, що дозволяє за час $O(\log n)$ знаходити значення деякої цільової асоціативної функції для елементів масиву a на проміжку $[i, j]$, $0 \le i \le j \le n$

$$f(a_i, a_{i+1}, \dots, a_j).$$

Прикладами таких цільових функцій може бути сума, добуток, мінімальне, максимальне значення, найбільший спільний дільник, тощо для елементів масиву на заданому проміжку.

Дерево відрізків – це повне бінарне дерево, у якому кожна вершина відповідає за деякий відрізок у масиві. Корінь дерева відповідає за весь масив, два його сини – за ліву та праву половини масиву і так далі. Кожен вузол дерева, що відповідає за більше ніж один елемент відрізка масиву має двох синів, що відповідають за ліву та праву половини масиву. Листки дерева відповідають за окремі елементи масиву.

Для прикладу, побудуємо дерево відрізків для швидкої обчислення суми елементів масиву зображеного на рисунку 6.2.45.

2	7	6	4	1	3	5	8
0	1	2	3	4	5	6	7

Рисунок 6.2.45. Масив вхідних даних.

Спочатку, створюємо листки дерева, що є елементами масиву.При цьому потрібно врахувати, що оскільки дерево відрізків є повним бінарним деревом, то кількість листків має обов'язково бути степенем двійки. Це необхідно для його коректної побудови. Якщо розмір заданого вхідного масиву не є степенем двійки, то його необхідно доповнитити, фіктивними елементами що не вплинуть на коректну побудову дерева (тобто на значення асоціативної функції на розширеному проміжку), до найближчого спепеня двійки. Наприклад, у нашому випадку, функція повертає суму на заданому проміжку, відповідно, необхідно доповнити масив нулями. У нашому випадку масив складається з восьми елементів, тому його розширення здійснювати не потрібно.

Рисунок 6.2.46. Лиски дерева відріза для масиву, зображеного на рисунку 6.2.45.

На наступному кроці створюємо батьківські вузли у які записуємо суму лівого і правого синів

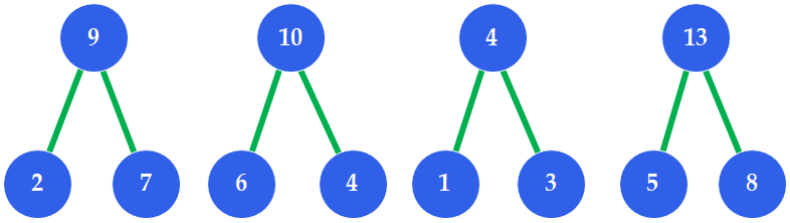


Рисунок 6.2.47. Передостанній рівень дерева відрізків.

Заповнюємо наступний рівень дерева відрізків, як суму значень у вузлах передостаннього рівня

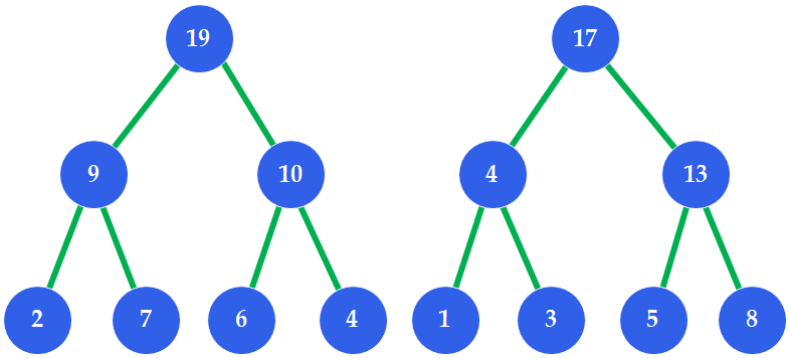


Рисунок 6.2.48.

Нарешті, остаточно, дерево відрізків набуде вигляду

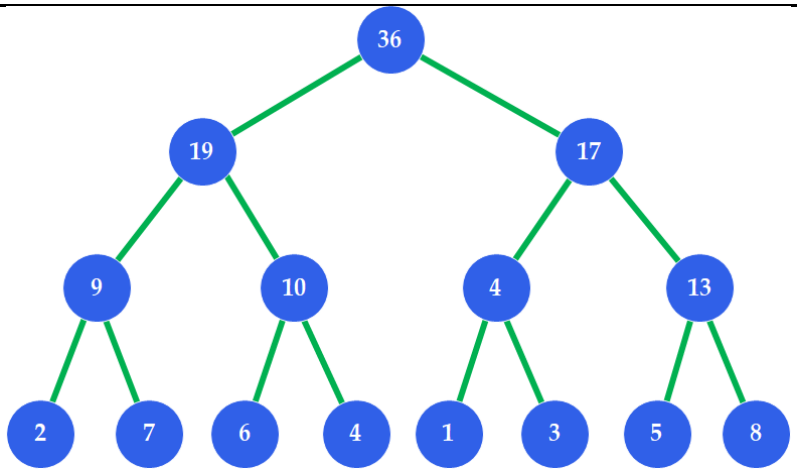


Рисунок 6.2.49. Дерево відрізків для знаходження суми.

Звернемо увагу, що для масиву, що складається з k елементів, його дерево відрізків буде складатися з $2n$ вершин, де $n = 2^p$ – найперший степінь двійки, що $k \leq n$. При цьому висота дерева відрізків буде $\log n$.

Найголовніша властивість дерева відрізків, на якій базуються усі алгоритми для роботи з ним полягає у тому, що довільний неперервний відрізок у масиві з p елементів можна зобразити за допомогою $2 \log p$ вершин дерева відрізків. Дійсно, знову розглянемо дерево відрізків для абстрактної цільової функції для масиву зображеного на рисунку 6.2.45. Зобразимо його на рисунку 6.2.50 таким чином, що у його вузлах будуть міститися усі елементи вхідного масиву, що мають відношення до цього вузла. Припустимо, необхідно визначити значення цільової функції на відрізку [1, 6]. Тоді необхідно використати чотири вузли дерева, що на рисунку 6.2.50 затінені.

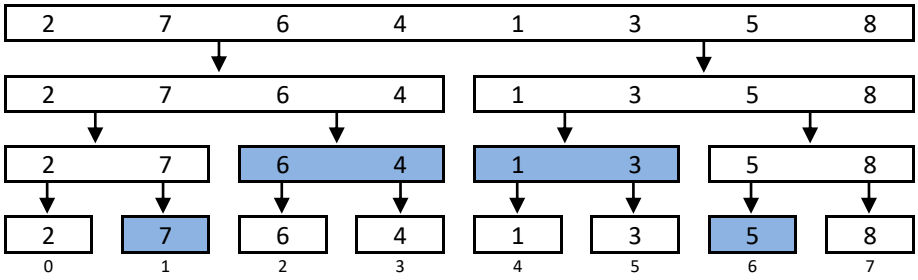


Рисунок 6.2.50. Дерево відрізків.

Розглянемо реалізацію дерева відрізків. Оскільки дерево відрізків є повним бінарним деревом, то, подібно до бінарної купи, для його зображення у пам'яті комп'ютера будемо використовувати звичайний масив з $2n$ елементів. При цьому листки дерева, тобто елементи вихідного масиву будуть зберігатися у правій половині масиву – 0-й елемент вихідного масиву буде міститися у позиції n, 1-й елемент масиву у позиції $n + 1$ і так далі. Внутрішні вузли дерева будуть міститися у позиціях від 2-ї по $n - 1$, а корінь дерева у позиції 1. Нульова позиція масиву, аналогічно до реалізації бінарної купи, лишається незадіяною. Наведемо зображення дерева відрізків для масиву наведеного на рисунку 6.2.45

-	корінь	1-й рівень		2-й рівень				листя дерева – вхідний масив даних							
-	36	19	17	9	10	4	13	2	7	6	4	1	3	5	8
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Операції з деревом відрізків та його реалізація

Реалізація дерева відрізків передбачає реалізацію таких операцій для вхідного масиву даних:

1. Побудова дерева відрізку на базі вхідного масиву даних.
2. Заміна значення одного з елементів вхідного масиву.
3. Повернення результату цільової функції для заданого проміжку вхідного масиву.

Опишемо клас `SegmentTree`. У ролі цільової функції оберемо функцію, що повертає суму елементів масиву заданого проміжку. Нагадаємо, що при такій реалізації повного бінарного дерева, для вузла з індексом i вузли з індексами $2i$ та $2i + 1$ будуть його відповідно лівим і правим синами, а вузол з індексом $[i/2]$ – його батьком. Найпершу операцію – побудову дерева відрізків – будемо здійснювати у конструкторі. Єдиним аргументом конструктора буде вхідний масив даних.

Лістинг 6.2.13. Реалізація дерева відрізків.

```
class SegmentTree:

    def __init__(self, array):
        """ Конструктор – створює бінарне дерево пошуку
        :param array: Вхідний масив елементів
        """
        k = len(array)                    # кількість елементів у масиві даних
        n = (1 << int(log2(k - 1)) + 1)   # доводимо розмірність масиву до степені 2
        self.mItems = 2 * n * [0]         # масив для збереження вузлів дерева

        # записуємо дані вхідного масиму у листки дерева
        for i in range(k):
            self.mItems[n + i] = array[i]

        # записуємо дані для внутнішніх вузлів дерева
        for i in range(n - 1, 0, -1):
            self.mItems[i] = self.mItems[2 * i] + self.mItems[2 * i + 1]

        self.mSize = n    # запам'ятовуємо розмірність масиву
```

Як бачимо побудова дерева відрізків має лінійну асимптотику, від кількості елементів вхідного масиву. Заміна одного зі значень вхідного масиву вимагатиме логарифмічного часу для оновлення дерева відрізків. Дійсно для оновлення всього дерева достатньо пройтися від листка до кореня і перерахувати значення у відповідних внутрішніх вузла та корені всього дерева.

Лістинг 6.2.13. Продовження. Оновлення елементів.

```
    def update(self, pos, x):
        """ Заміняє значення ключа у вхідному масиві
        :param pos: позиція елемента вхідного масиву, що потрібно замінити
        :param x:   нове значення
        """
        pos += self.mSize
        self.mItems[pos] = x  # Змінюємо значення у листі
        # перераховуємо значення у внутрішних вузлах
        i = pos // 2  # Позиція батьківського вузла
        while i > 0:
            self.mItems[i] = self.mItems[2 * i] + self.mItems[2 * i + 1]
            i //= 2
```

Найскладнішою операцією з трьох операцій для дерева відрізків, є операція обчислення значення цільової функції на заданому проміжку. Для обчислення її значення, використаємо два маркери `left` та `right` – відповідно ліву і праву межі заданого відрізка. Далі будемо рухати лівий і правий маркери по дереву вгору, доки вони не зустрінуться. При цьому, якщо лівий маркер є лівим сином свого батька у дереві відрізка, то відповідний відрізок `[left, left + 1]` міститься у батьківському відрізку і буде врахований на вищих рівнях дерева, а якщо лівий маркер є правим сином свого батька, то цей елемент необхідно врахувати окремо (див рисунок 6.2.50). Аналогічною, з точністю до симетрії, буде ситуація з правим маркером. Таким чином, наведемо реалізацію функції обчислення суми на відрізку.

Лістинг 6.2.13. Продовження. Обчислення цільової функції на заданому проміжку.

```
    def sum(self, left, right):
        """ Сума елементів відрізку від left до right вхідного масиву

        :param left: ліва межа відрізку
        :param right: права межа відрізку
        """
        left += self.mSize
        right += self.mSize
```

```
        res = 0

        while left <= right:
            if left % 2 == 1:   # якщо left - правий син свого батька
                res += self.mItems[left]
            if right % 2 == 0:  # якщо right - лівий син свого батька
                res += self.mItems[right]

            # піднімаємося у дереві на рівень вище
            left = (left + 1) // 2
            right = (right - 1) // 2

        return res
```

Модифікації дерев відрізків

Існує багато різних модифікацій дерев відрізків для розв'язання різноманітних задач. Наприклад, для розв'язання деяких задач, вершина, що відповідає за відрізок, повинна містити усі елементи цього відрізка (зокрема, у відсортованому вигляді). Класичною задачею, що використовує таку модифікацію дерева відрізків є задача у якій необхідно швидко відповідати на питання скільки елементів більших за заданий містить заданий проміжок.

Інший тип модифікацій дерева відрізків застосовується коли необхідно змінювати у вхідному масиві зразу групу елементів. Як ми знаємо дерево відрізків дозволяє ефективно, за логарифмічний час оновлювати елемент вхідного масиву. Проте, якщо необхідно оновити не один елемент, а зразу цілу групу, то по-елементне оновлення приведе до регресії швидкодії алгоритму до лог-лінійної ($O(n \log n)$).

Перший тип модифікацій, що дозволяє ефективно оновлювати дерево відрізків застосовується у випадку коли необхідно застосувати одну й ту ж зміну до всіх елементів заданого проміжку, наприклад, у випадку, якщо необхідно додати до всіх елементів вхідного масиву задане число. У цьому разі, щоб здійснювати додавання ефективно, будемо зберігати у кожній вершині дерева відрізків, скільки необхідно додати до всіх чисел цього відрізка (додаткове навантаження). Тоді у випадку, коли надійде запит на отримання значення цільової функції на відрізку, будемо просто додавати до значення цільової функції додаткові навантаження вершин, які зустрінуться на шляху роботи алгоритму.

Інша модифікація дерева відрізків з масовим оновленням елементів застосовується для задач, коли виникає запит на зміну усіх елементів деякого відрізу вхідного масиву заданим значенням. Для таких задач модифікація полякає у такому: На запиті присвоєння будемо рухатися від кореня до листків (!). Нехай у процесі виконання запиту на оновлення даних на відрізку ми опинилися у вершині, що повністю належить цьому проміжку. Відповідно до класичного алгоритму побудови дерева відрізків ми зобов'язані змінити значення у цій вершині і у вершинах її піддерева. Проте складність такої операції, як зазначено вище є неприйнятною ($O(n \log n)$). Замість цього ми змінимо значення лише у самій вершині, не оновлюючи її піддерево та додамо до вершини позначку, що у цієї вершини є, неузгоджена з її піддеревами модифікація. Якщо подальші запити не будуть звертатися до піддерев з неузгодженою модифікацією, то вони будуть виконуватися коректно. У випадку, якщо надійде запит, що за зачіпає піддерево з неузгодженою модифікацією, будемо передавати модифікацію дочірнім вузлам (лише вузлам, не усім підеревам). Після такої операції вузол стає узгодженим, а позначка про неузгодженість переходить до її дочірніх вузлів. Така операція називається проштовхуванням модифікації.

РОЗДІЛ 7. ТЕОРІЯ ГРАФІВ

§7.1. Означення, приклади та реалізація у Python

7.1.1. Означення та приклади

Графи це математичні модель, що зображає сукупність об’єктів та зв’язки між ними. Велика кількість структур, які мають практичну цінність в математиці, інформатиці та інших науках, можуть бути зображені як графи. Наприклад, системи доріг, схема руху метрополітену, множина аеропортів та регулярні рейси літаків між ними, комунікації в інтернеті, тощо можуть бути зображені у вигляді графів. На рисунку 7.1.1 зображено схему руху київського метрополітену, разом зі схемою руху міської електрички, яка є нічим іншим як графом.

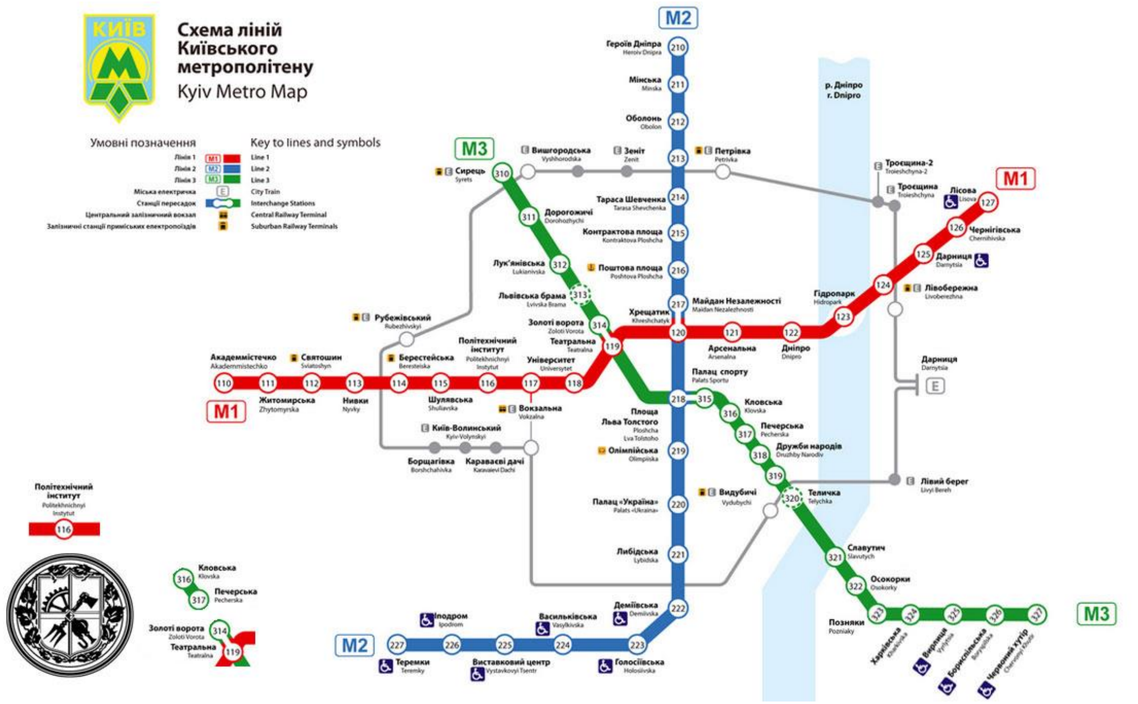


Рисунок 7.1.1. Схема руху є пракладом графу.

Неформально граф – це набір точок (вершин) і ліній (ребер), що ці точки з’єднують.
Отже, наведемо строге означення графа.

> **Означення 7.1.1.** Графом G називається пара
> $$G = (V, E)$$
> де $V = \{v_1, v_2, \dots, v_n\}$ – множина вершин графу, $E = \{e_1, e_2, \dots, e_m\}$ – множина ребер графу.

Вершина (або вузол) графу – це математична абстракція, яка моделює об’єкти. Наприклад, на схемі на рисунку 7.1.1 – вершинами є станції метро та станції міської електрички. Вершина, як правило, має ім’я, яке називається ***ключем*** або ідентифікатором вершини. Ключі використовуються, для того щоб розрізняти вершини. На схемі вище, ключами є назви станцій метро та електрички. Часто при розв’язанні конкретних задач виникає необхідність пов’язати з певною вершиною деякі дані. Ці дані називаються корисним **навантаженням вершини** (або просто навантаженням вершини). У нашому прикладі навантаженням може бути час роботи станції, визначні місця поруч, тощо.

Ребра (або дуги) графу визначають відношення (тобто з’єднання) між парами вершин. Так у нашому прикладі, ребрами будуть відрізки, що сполучають сусідні станції метро, перегони між станціями міської електрички або переходи між станціями з однієї гілки на іншу. Ребра визначаються парою (тобто кортежем) вершин які вони з’єднують.

Кількість вершин у графі називається **порядком графа** та позначається $|V|$, а кількість його ребер – **розміром графу** та позначається $|E|$.

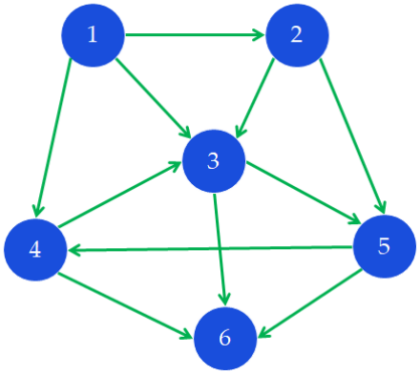


Рисунок 7.1.2. Граф.

На рисунку 7.1.2 наведено граф, що має 6 вершин (а отже порядок графа 6)

$$V = \{1,2, \dots ,6\}$$

та 11 ребер (відповідно розмір графа – 11)

$$E = \{(1,2), (1,3), (1,4), (2,3), (2,5), (3,5), (3,6), (4,3), (4,6), (5,4), (5,6)\}$$

Ребра графу можуть бути **одно**- або **двонаправленими**. Однонаправлене ребро (також використовується термін дуга) графа вказує на односторонній зв'язок пари вершин графу. Прикладами дуг, тобто однонаправлених ребер в реальному житті можуть бути вулиці з одностороннім рухом, що з'єднують перехрестя або транспортні розв'язки (тобто вершини). Дуги позначаються на малюнку відрізками (або дугами) зі стрілками, що визначають дозволений рух від однієї вершини до іншої. Граф, у якого хоча б одне ребро якого є дугою, називається **орієнтованим**. Наведений вище на рисунку 7.1.2 граф є орієнтованим, оскільки всі його ребра є дугами.

Якщо рух по ребру між двома вершинами може здійснюватися у обох напрямках, то ребро називається двонаправленим (у подальшому просто ребро). Граф всі ребра якого двонаправлені називається **неорієнтованим** графом. Наприклад, на схемі метрополітену зображеній на рисунку 7.1.1, всі ребра є двонаправленими, відповідно вона утворює неорієнтований граф.

На схемах, двонаправлені ребра позначаються одним з таких способів

- відрізками (або дугами) зі стрілками на обох кінцях, щоб закцентувати увагу, що ребро у орієнтованому графі є направленим (рисунок 7.1.3);
- просто відрізками (або дугами), що сполучають відповідні вершини, якщо граф є неорієнтованим, (рисунок 7.1.4)

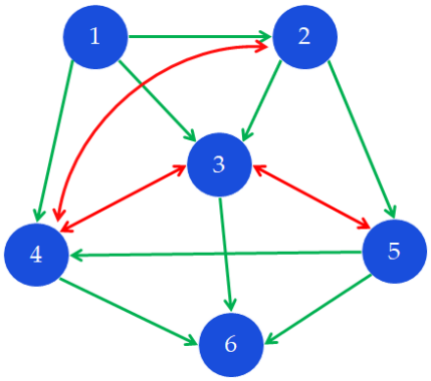


Рисунок 7.1.3. Орієнтований граф, що має три двонаправлені ребра. Двонаправлене ребро (2, 4) позначається дугою зі стрілками на кінцях

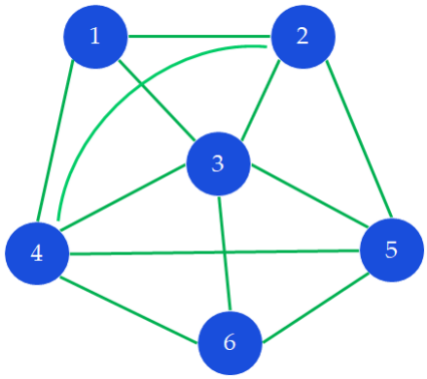


Рисунок 7.1.4. Неорієнтований граф – всі його ребра двонаправлені.

У неорієнтованих графах вершини сполучені ребром називають **суміжними** (або сусідами). Також називають **суміжними ребра**, що мають спільну вершину. Вершина 1 графа зображеному на рисунку 7.1.4 має три суміжні вершини 2, 3 та 4, а для вершини 3 суміжними є всі (крім самої вершини 3) вершини графа.

Для орієнтованих графів, суміжними з вершиною v називаються вершини, що з'єднуються з вершиною v дугою (з початком у точці v). Так, для прикладу, у графі зображеному на рисунку 7.1.3, суміжними з вершиною 1 є вершини 2, 3, та 4. А вершина 6 взагалі немає суміжних, хоча вона є суміжною з вершинами 3, 4 та 5.

Для неорієнтованих графів вважається, що рух з вершини безпосередньо у себе не має сенсу. Для орієнтованих графів такий спосіб допускається. При цьому вершина вважається суміжною собі, що зазначається при описі графа заданням відповідного ребра. Дуги що позначають такий перехід називаються **петлями**.

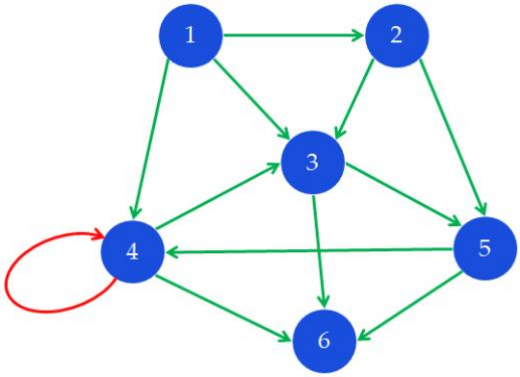


Рисунок 7.1.5. Граф що має петлю у вершині 4.

Степінь вершини у неорієнтованому графі це кількість ребер, що виходить з цієї вершини. Степінь вершини v позначається

$$deg\, v$$

Для графу, зображеного на рисунку 7.1.4

$$deg\, 1 = 3$$
$$deg\, 3 = 5$$

Очевидно, що для неорієнтованих графів степінь вершини дорівнює кількості її сусідів.

Напівстепінь входу вершини v орієнтованого графа – це кількість дуг, які входять у дану вершину. Позначається

$$deg^+ v$$

Для графа зображеного на рисунку 7.1.3

$$deg^+ 1 = 0$$
$$deg^+ 3 = 4$$
$$deg^+ 6 = 3$$

Напівстепінь виходу вершини v орієнтованого графа – це кількість дуг, які виходять з даної вершини.

$$deg^- v$$

Для графа зображеного на рисунку 7.1.3

$$deg^- 1 = 3$$
$$deg^- 3 = 3$$
$$deg^- 6 = 0$$

Вершина з напівстепенем входу 0 називається **джерелом**, а вершина з напівстепенем виходу 0 – **стоком**. Отже, вершина 1 у графі на рисунку 7.1.3 є джерелом, а вершина 6 – відповідно стоком.

Граф G^T, отриманий у результаті зміни напрямів всіх ребер графа G на протилежні називається транспонованим до графа G.

Ребра можуть мати **вагу**. Вага визначає «вартість» переміщення від однієї вершини до іншої. Наприклад, у графі, що моделює схему доріг, що зв'язує міста, вага може визначати відстань між двома містами. На зображеннях графів, вагу вказують над відповідними ребрами або поруч з ними.

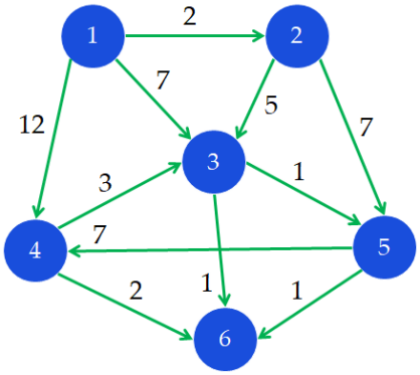


Рисунок 7.1.6. Зважений граф.

Граф, ребра якого мають вагу називається **зваженим**.

Шлях

Означення 7.1.2. Маршрутом в графі називається послідовність вершин і ребер, яка має такі властивості:
1) вона починається і закінчується вершиною;
2) вершини і ребра в ній чергуються;
3) будь-яке ребро цієї послідовності має своїми кінцями дві вершини: що безпосередньо передує йому в цій послідовності і наступну що йде відразу за ним.

Прикладом маршруту у графі зображеному на рисунку 7.1.2 може бути така послідовність

$$1 \to 2 \to 5 \to 4 \to 6$$

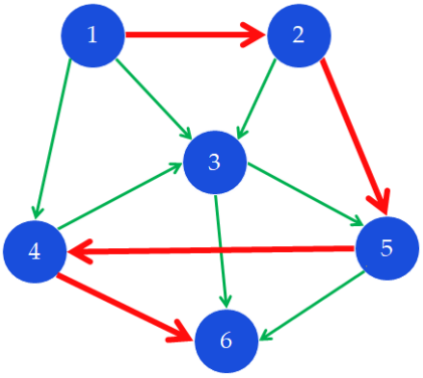


Рисунок 7.1.7. Маршрут у графі.

Перша (тобто вершина 1) і остання (тобто вершина 6) вершини в цій послідовності називаються **початком** і **кінцем** маршруту відповідно.

Шляхом називається такий маршрут, в якому жодне ребро не зустрічається двічі. Очевидно, що вищенаведений маршрут є шляхом.

Маршрут, що веде з вершини $v_i \in V$ у точку $v_j \in V$ будемо позначати

$$v_i \rightsquigarrow v_j.$$

Шлях, що веде з вершини v_i у точку v_j будемо позначати

$$v_i \longmapsto v_j.$$

Кажуть, що вершина v_j досяжна з вершини v_i, якщо існує шлях з вершини v_i у вершину v_j. Якщо такого шляху не існує, то вершина v_j називається недосяжна з вершини v_i.

Довжиною шляху у незваженому графі називається кількість ребер у цьому шляху. Довжина шляху зображеного на рисунку 7.1.7 буде 4. Шлях у (зваженому) графі має довжину, що є сумою ваг усіх ребер, що входять до цього шляху. Вважається, що шлях з вершини у себе в неорієнтованому графі має довжину 0.

Шлях називається **замкненим** або **циклічним**, якщо він починається та закінчується у одній і тій же вершині. Наприклад, у графі зображеному на рисунку 7.1.2 замкненим буде шлях $3 \to 5 \to 4 \to 3$:

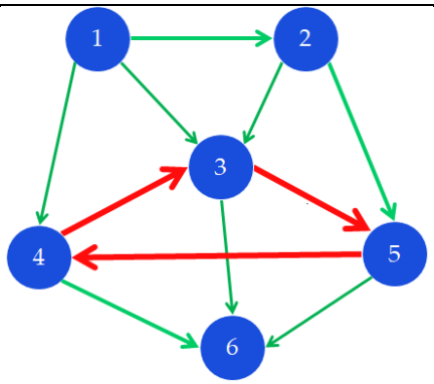


Рисунок 7.1.8. Замкнений шлях у графі.

Не замкнений шлях називається простим, якщо кожна вершина у ньому відвідується лише раз. Шлях зображений на рисунку 7.1.7 є простим Замкнений шлях називається простим, якщо кожна вершина, за виключенням початкової та кінцевої вершини у ньому відвідується лише один раз. Простий циклічний шлях називається **циклом**. Для циклу не є принциповим є яка вершина є його початком і кінцем. Замкнений шлях зображений на рисунку 7.1.8 є циклом.

На рисунку нижче, зображено граф у якому визначено шлях з вершини 1 у вершину 6,

$$1 \to 3 \to 5 \to 4 \to 3 \to 6$$

який не є простим, оскільки вершина 3 відвідується двічі.

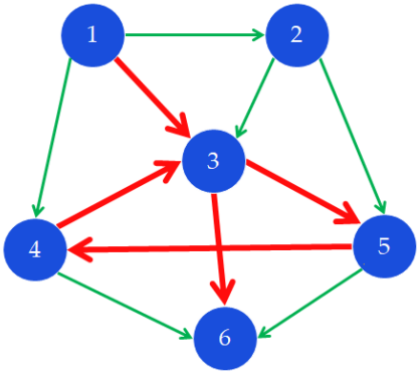


Рисунок 7.1.9. Не простий шлях з вершини 1 у вершину 6 у графі.

Граф, що не містить циклів називається **ациклічним**. Нижче наведено приклади ациклічного графу

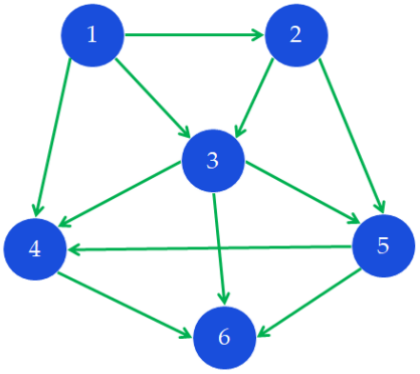


Рисунок 7.1.10. Ациклічний граф.

Зв'язність графів

Не орієнтований граф називається **зв'язним**, якщо у ньому будь-яка вершина є досяжною з будь-якої іншої. Неорієнтований граф на рисунку 7.1.4 є зв'язним. Неорієнтований граф називається **незв'язним**, якщо він не є зв'язним.

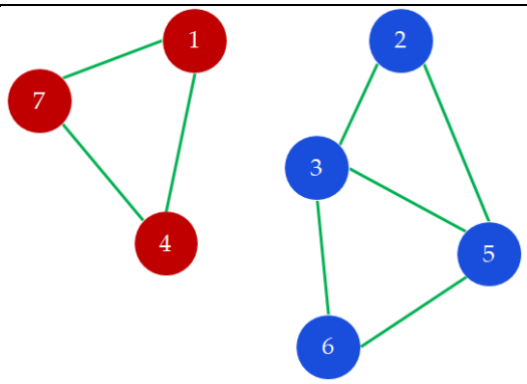


Рисунок 7.1.11. Незв'язний граф.

Бінарне відношення на множині вершин графа, задане як вершина v_j досяжна з вершини v_i є відношенням еквівалентності і відповідно розбиває граф на класи еквівалентності, що називаються **компонентами зв'язності**. Граф зображений на рисунку 7.1.11 має дві компоненти зв'язності: у першу входять вершини 1, 4 та 7, у другу відповідно – 2, 3, 5, 6. Очевидно, що якщо граф має лише одну компоненту зв'язності, то він зв'язний. Обернене твердження також є правильним.

Для орієнтованих графів поняття зв'язності не є коректним. Натомість говорять про сильну, односторонню та слабку зв'язність графів.

Отже, орієнтований граф називається **сильно-зв'язним**, якщо в ньому існує шлях з будь-якої вершини до будь-якої іншої. У такому разі говорять що граф містить одну компоненту сильної зв'язності.

Орієнтований граф називається **односторонньо-зв'язним**, якщо для будь-яких двох його вершин v_i та v_j існує хоча б один зі шляхів або від v_i до v_j чи від v_j до v_i.

Орієнтований граф називається **слабко-зв'язним**, якщо є зв'язним неорієнтований граф, отриманий з нього заміною орієнтованих ребер на неорієнтовані.

Очевидно, що граф зображений на рисунку 7.1.2 не є сильно-зв'язним, оскільки вершина 1 є джерелом і, відповідно, вона не є досяжною з жодної іншої вершини графа. Також очевидною є слабка зв'язність цього графу. Найпростіший (проте не найоптимальніший) спосіб довести, що цей граф є односторонньо-зв'язним, потрібно побудувати шляхи між усіма вершинами цього графа. Пропонуємо читачу самостійно провести цю операцію.

Зв'язок графів та дерев

Між графами та деревами є прямий зв'язок, що полягає у тому, що кожне дерево є орієнтованим зв'язним графом, якщо передбачається рух лише від вузла до синів, або неорієнтованим, якщо додатково допускається рух від синів до батьків. У цьому розділі пізніше будуть наведені алгоритми, перевірки чи є граф деревом, а також побудови по графу кістякового дерева.

Операції з графами

Як і у випадку дерев, для графів взагалі кажучи не визначено обов'язкових операцій – відповідні операції визначаються залежно від поставленої задачі та типу графа, щоб будується для її розв'язання. Наведемо основні операції, що часто визначають для незваженого графу

1. Створення нового порожнього графу;
2. Операція додавання в граф нової вершини;
3. Встановлення навантаження для заданої вершини;
4. Операція додавання в граф нового ребра (для неорієнтованого графу);
5. Операція додавання в граф нової дуги (для орієнтованого графу);
6. Пошук у графі вершини;
7. Отримання списку всіх вершин графа;
8. Перевірка чи входить задана вершина в граф.

Для графів, вершини якого мають додаткове навантаження визначається додатково методи

9. Отримати/встановити навантаження вершини.

Для зважених графів, визначають операції

10. Отримати/встановити вагу ребра.

7.1.2. Реалізація графу на мові Python

Існує два широковідомих підходи для реалізації графів:

- Матриця суміжності
- Список суміжності

Матриця суміжності

Одним з найпростіших способів реалізації графу (тобто його зображення у пам'яті комп'ютера) є використання матриці суміжності – двовимірної матриці, у якій i-й рядок відповідає вершині графа v_i, j-й стовпчик відповідає вершині графа v_j, а елемент матриці на перетині i-го рядка та j-го стовпчика показує, чи існує ребро з вершини v_i до вершини v_j. Нульовий елемент матриці суміжності говорить, що ребро між відповідними вершинами відсутнє. Ненульове значення вказує, що існує ребро з вершини v_i до вершини v_j, причому це значення визначає вагу ребра (v_i, v_j). Для незважених графів цю вагу, як правило встановлюють 1.

На рисунку 7.1.12 показана матриця суміжності для графа, зображеного рисунку 7.1.2.

	1	2	3	4	5	6
1	0	1	1	1	0	0
2	0	0	1	0	1	0
3	0	0	0	0	1	1
4	0	0	1	0	0	1
5	0	0	0	1	0	1
6	0	0	0	0	0	0

Рисунок 7.1.12. Матриця суміжності графа.

Матриця суміжності для зваженого графа зображеного на рисунку 7.1.6 буде відрізнятися тим, що замість одиниць у відповідних клітинах будуть розміщуватися значення ваги відповідних ребер.

	1	2	3	4	5	6
1	0	2	7	12	0	0
2	0	0	5	0	7	0
3	0	0	0	0	1	1
4	0	0	3	0	0	2
5	0	0	0	7	0	1
6	0	0	0	0	0	0

Рисунок 7.1.13. Матриця суміжності зваженого графа.

Матриця суміжності графа для неорієнтованого графа завжди буде симетричною. Дійсно, якщо у орієнтованому графі є ребро що з'єднує вершину v_i з вершиною v_j то відповідно є ребро, що з'єднує вершину v_j з вершиною v_i. Нижче наведено матрицю суміжності для графа зображеного на рисунку 7.1.4.

	1	2	3	4	5	6
1	0	1	1	1	0	0
2	1	0	1	1	1	0
3	1	1	0	1	1	1
4	1	1	1	0	1	1
5	0	1	1	1	0	1
6	0	0	1	1	1	0

Рисунок 7.1.14. Матриця суміжності графа.

Переваги:

- Простота реалізації
- Математична наочність
- Однаково реалізується як зважений так і не зважений граф.

Недоліки

- Проблематично додавати/видаляти вершини;
- Великий об'єм оперативної пам'яті, що використовується даремно;
- Для ідентифікації вершин (без додаткового виділення коду) можуть використовуватися лише послідовні натуральні числа;

- Навантаження вершин треба реалізовувати окремою структурою даних;
- При побудові алгоритмів (наприклад пошуку в глибину чи ширину) пошук сусідів для кожної вершини є лінійним, оскільки вимушує пробігати повністю по відповідному рядку матриці;

Отже, матрицю суміжності для графів рекомендується використовувати для задач, у яких кількість вершин є не значною (максимум 2к вершин), кількість вершин наперед відома та задача не передбачає додавання чи видалення вершин.

Наведемо реалізацію графа з використанням матриці суміжності. Клас `Graph` містить лише конструктор та метод для зв'язування двох вершин ребром.

Лістинг 7.1.1. Реалізація графа – матриця суміжності.

```
class Graph:
    def __init__(self, oriented=False, vertex_number=20):
        """ Конструктор графа

        :param oriented:      Чи орієнтований граф
        :param vertex_number: Кількість вершин у графі
        """
        self.mIsOriented = oriented         # Поле чи орієнтований граф
        self.mVertexNumber = vertex_number  # Лічильник вершин у графі

        # Створюємо матрицю суміжності заповнену нулями
        self.mAdjacentMatrix = []
        for i in range(self.mVertexNumber):
            self.mAdjacentMatrix.append([0] * self.mVertexNumber)

    def addEdge(self, source, destination, weight=1):
        """ Додавання ребра з кінцями в точках source та destination з вагою weight
        :param source:      Перша вернина
        :param destination: Друга вершина
        :param weight:      Вага ребра
        """
        assert 0 <= source < self.mVertexNumber and 0 <= destination < self.mVertexNumber
        self.mAdjacentMatrix[source][destination] = weight

        # Якщо граф є неорієнтованим, то треба встановити зворотній
        # зв'язок з вершини toVert до fromVert
        if not self.mIsOriented:
            self.mAdjacentMatrix[destination][source] = weight
```

Щоб створити граф, зображений на рисунку 7.1.6 скористаємося ланцюжком команд наведених нижче. Зауважимо, що оскільки вершини у нашому графі ідентифікуються натуральними числами від 1 до 6, а елементи списків індексуються від 0, то для того, щоб не перейменовувати вершини у програмі, у коді нижче створюється граф з семи вершин (з фіктивною вершиною з ключем 0).

Лістинг 7.1.1. Продовження. Побудова графа.

```
g = Graph(True, 7)   # Створюємо орієнтований зважений граф.

g.addEdge(1, 2, 2)   # ребро (1, 2) з вагою 2
g.addEdge(1, 3, 7)   # ребро (1, 3) з вагою 7
g.addEdge(1, 4, 12)  # ребро (1, 4) з вагою 12
g.addEdge(2, 3, 5)   # ребро (2, 3) з вагою 5
g.addEdge(2, 5, 7)   # ребро (2, 5) з вагою 7
g.addEdge(3, 5, 1)   # ребро (3, 5) з вагою 1
g.addEdge(3, 6, 1)   # ребро (3, 6) з вагою 1
g.addEdge(4, 3, 3)   # ребро (4, 3) з вагою 3
g.addEdge(4, 6, 4)   # ребро (4, 6) з вагою 4
g.addEdge(5, 4, 7)   # ребро (5, 4) з вагою 7
g.addEdge(5, 6, 1)   # ребро (5, 6) з вагою 1
```

Якщо задача передбачає використання навантаженого графа, то вищенаведений клас треба доповнити полями та методами роботи з даними

Лістинг 7.1.2. Навантажений граф.

```
class GraphWithData(Graph):
    """ Клас нащадок класу Graph
        Містить поле навантаження вершини та методи роботи з ним
    """

    DEFAULT_VERTEX_DATA = 0  # типове навантаження вершини

    def __init__(self, oriented: bool = False, vertices: int = 20):
        """ Конструктор
        :param oriented: Чи орієнтований граф
        :param vertices: Кількість вершин у графі
        """
        super().__init__(oriented, vertices)  # Виклик конструктора батьківського класу

        # Список навантажень вершин
        self.mData = [self.DEFAULT_VERTEX_DATA] * self.mVertexNumber

    def setData(self, vertex, data):
        """ Встановлення навантаження на вершину
        :param vertex: Вершина графа
        :param data:   Навантаження
        """
        assert 0 <= vertex < self.mVertexNumber
        self.mData[vertex] = data

    def data(self, vertex):
        """ Повертає навантаження заданої вершини графу
        :param vertex: Вершина графа
        :return: Навантаження вершини графа
        """
        assert 0 <= vertex < self.mVertexNumber
        return self.mData[vertex]
```

Список суміжності

Оптимальнішим та гнучкішим способом реалізації розрідженого графа є використання списку суміжності. У такому зображенні граф фактично є списком усіх вершин, кожна з яких володіє інформацією про пов'язані з нею вершини.

На рисунку 7.1.15 показаний список суміжності для графа з рисунку 7.1.2.

Вершина	Список суміжних вершин
1	{2, 3, 4}
2	{3, 5}
3	{5, 6}
4	{3, 6}
5	{4, 6}
6	–

Рисунок 7.1.15. Зображення графа використовуючи список суміжності.

У випадку, якщо граф є зваженим, разом з ключем вершини-сусіда потрібно запам’ятовувати вагу ребра, що їх з’єднує. Хоча підхід і називається «список» суміжності, проте у нашій реалізації будемо використовувати словник для швидкового пошуку відповідної вершини серед її сусідів. У цьому словнику у ролі пари ключ-значення будемо використовувати відповідно ключ вершини-сусіда та вагу ребра, що сполучає відповідну вершину з поточною. Крім цього, використання словника дозволить зняти обмеження на іменування ключів вершин – нагадаємо, що у випадку використання матриці суміжності, ключами могли бути лише натуральні числа.

На рисунку нижче зображено список суміжності для зваженого орієнтованого графа, що зображений на рисунку 7.1.6.

Вершина	Список суміжних вершин
1	{2 : 2, 3 : 7, 4 : 12}
2	{3 : 5, 5 : 7}
3	{5 : 1, 6 : 1}
4	{3 : 3, 6 : 4}
5	{4 : 7, 6 : 1}
6	–

Рисунок 7.1.16. Зображення зваженого графа.

У випадку, якщо граф є неорієнтованим, то під час додавання відповідного ребра, що сполучає вершину v_i з вершиною v_j відповідно потрібно додавати ребро, що з'єднує вершину v_j з вершиною v_i. Отже, список суміжності для неорієнтованого графа зображеного на рисунку 7.1.4. буде мати вигляд

Вершина	Список суміжних вершин
1	{2, 3, 4}
2	{1, 3, 4, 5}
3	{1, 2, 4, 5, 6}
4	{1, 2, 3, 5, 6}
5	{2, 3, 4, 6}
6	{3, 4, 5}

Рисунок 7.1.17. Зображення графа використовуючи список суміжності.

Перевагою реалізації графа використовуючи список суміжності є те, що він дозволяє компактно зображувати розріджені графи. Також у списку суміжності легко знайти всі посилання, безпосередньо пов'язані з конкретною вершиною.

Реалізація списку суміжності буде складатися з трьох класів:

1. `VertexBase` – базовий клас для опису вершини графа, що використовується для моделювання вершини як контейнера, що містить пару – ключ та навантаження;
2. `Vertex` – нащадок класу `VertexBase` та реалізує вершину графа, зберігаючи список (словник) вершин, що є суміжними до неї. Для універсальності класу, ребра незваженого графа будемо встановлювати рівними 1.
3. `Graph` – власне є списком суміжності вершин графу. Знову ж таки, щоб зняти обмеження з іменування ключів графа та реалізувати швидкий пошук відповідної вершини за ключем, замість списку будемо використовувати словник, у якому парами ключ-значення будуть відповідно ключ вершин та власне вершина класу `Vertex` (що містить у собі інформацію про усіх своїх сусідів).

Лістинг 7.1.3. Реалізація графа – базовий клас `VertexBase` для опису вершини графа.

```
class VertexBase:
    """ Базовий клас Vertex - вершина.

        Є базовим класом для класу, що описує вершину графа
        Клас містить поля - ключ (ім'я) вершини mKey,
        а також її навантаження (тобто дані) mData. """

    def __init__(self, key):
        """ Конструктор створення вершини
        :param key: Ключ вершини
        """
        self.mKey = key    # Ключ (ім'я) вершини
        self.mData = None  # Навантаження (дані) вершини

    def key(self):
        """ Повертає ключ (ім'я) вершини
        :return: Ключ вершини
```

```
        """
        return self.mKey

    def setData(self, data):
        """ Встановлює навантаження на вершину
        :param data: Навантаження
        :return: None
        """
        self.mData = data

    def data(self):
        """ Повертає навантаження вершини

        :return: Навантаження вершини
        """
        return self.mData
```

Клас `Vertex` буде розширяти клас `VertexBase`, додаючи всі необхідні атрибути та методи, що необхідні будуть для роботи з графами. Тут додане поле `mNeighbors`, яке буде містити список сусідів вершини у вигляді словника, де ключем буде ключ вершини-сусіда, а значенням – значення ваги ребра (дуги у орієнтованому графі), що з'єднує вершину з сусідом. Наприклад, якщо словник `mNeighbors` буде мати вигляд,

$$\{17: 4, 3: 2\}$$

то означає, наша вершина має сусідами вершини 17 та 3 з вагами ребер 4 та 2 відповідно.

Лістинг 7.1.4. Реалізація графа – опис класу `Vertex`.

```
class Vertex(VertexBase):

    def __init__(self, key):
        """ Конструктор створення вершини
        :param key: Ключ вершини
        """
        super().__init__(key)  # Викликаємо конструктор батьківського класу
        self.mNeighbors = {}   # Список сусідів вершини у вигляді пар (ключ: вага_ребра)

    def addNeighbor(self, vertex, weight=1):
        """ Додати сусіда

        Додає ребро, що сполучає поточну вершину з вершиною Vertex з вагою weight
        Vertex може бути або іншою вершиною, тобто об'єктом класу Vertex
        або ключем (ідентифікатором вершини)
        :param vertex: Вершина-сусід або ключ вершини
        :param weight: Вага ребра
        """
        if isinstance(vertex, VertexBase):  # Якщо Vertex - вершина
            self.mNeighbors[vertex.key()] = weight
        else:                               # Якщо Vertex - ім'я (ключ) вершини
            self.mNeighbors[vertex] = weight

    def neighbors(self):
        """ Повертає список ключів всіх сусідів поточної вершини

        :return: Список ключів всіх сусідів вершини
        """
        return self.mNeighbors.keys()

    def weight(self, neighbor):
        """ Повертає вагу ребра, що сполучає поточну вершину та вершину-сусіда

        :param neighbor: Вершина-сусід
        :return: Вага ребра
        """
        if isinstance(neighbor, VertexBase):  # Якщо aNeighbor - вершина (не ім'я)
            return self.mNeighbors[neighbor.key()]
        else:  # Якщо aNeighbor - ім'я (ключ) сусідньої вершини
            return self.mNeighbors[neighbor]
```

Нарешті опишемо клас Graph, що буде використовувати клас Vertex.

Лістинг 7.1.5. Реалізація графа – опис класу Graph.

```
class Graph:
    """ Граф, що задається списком суміжних вершин """

    def __init__(self, oriented=False):
        """ Конструктор графа
        :param oriented: Чи орієнтований граф
        """
        self.mIsOriented = oriented    # Поле чи орієнтований граф
        self.mVertexNumber = 0         # Лічильник вершин у графі
        self.mVertices = {}            # Список (словник) вершин у графі у вигляді
                                       # пар (ключ: вершина)

    def addVertex(self, vertex):
        """ Додає вершину у граф, якщо така вершина не міститься у ньому

        :param vertex: ключ (тобто ім'я) нової вершини
        :return: True, якщо вершина успішно додана
        """

        if vertex in self:  # Якщо вершина міститься у графі, її вже не треба додавати
            return False

        new_vertex = Vertex(vertex)  # створюємо нову вершину з іменем Vertex
        self.mVertices[vertex] = new_vertex  # додаємо цю вершину до списку вершин графу
        self.mVertexNumber += 1              # Збільшуємо лічильник вершин у графі
        return True

    def getVertex(self, vertex):
        """ Повертає вершину графу, якщо така вершина міститься у графі

        :param vertex: ключ (тобто ім'я) вершини
        :return: Вершина графа
        """
        assert vertex in self

        # Визначаємо ключ вершини, якщо це необхідно
        key = vertex.key() if isinstance(vertex, Vertex) else vertex
        return self.mVertices[key]

    def vertices(self):
        """ Повертає список всіх вершин у графі"""
        return self.mVertices

    def addEdge(self, source, destination, weight=1):
        """ Додавання ребра з кінцями в точках source та destination з вагою weight

        :param source:      Перша вернина
        :param destination: Друга вершина
        :param weight:      Вага ребра
        """
        if source not in self:        # Якщо вершина source ще не міститься у графі
            self.addVertex(source)    # додаємо вершину source
        if destination not in self:   # Якщо вершина destination ще не міститься у графі
            self.addVertex(destination)  # додаємо вершину destination

        # Встановлюємо зв'язок (тобто ребро) між вершинами source та destination
        self[source].addNeighbor(destination, weight)

        if not self.mIsOriented:  # Якщо граф неорієнтований, додамо зворотній зв'язок
            self.mVertices[destination].addNeighbor(source, weight)

    def setData(self, vertex, data):
        """ Встановлення навантаження вершини
        :param vertex: ключ вершини або вершина графа
        :param data:   навантаження
```

```
        """
        assert vertex in self # Перевірка чи міститься вершина в графі
        self[vertex].setData(data)

    def getData(self, vertex):
        """ Повертає навантаження вершини

        :param vertex: Вершина або її ключ
        :return: Навантаження вершини
        """
        assert vertex in self  # Перевірка чи міститься вершина в графі
        return self[vertex].data()

    def transpose(self):
        """ Будує граф транспонований до заданого

        :return: Новий граф, що є транспонований до заданого
        """

        g_inv = Graph(self.mIsOriented)
        for vertex in self:
            for neighbor_key in vertex.neighbors():  # Для всіх сусідів поточної вершини
                g_inv.addEdge(neighbor_key, vertex.key())

        return g_inv

    def __contains__(self, vertex):
        """Перевизначення оператора in - перевіряє чи міститься вершина у графі

        :param vertex: Вершина або її ключ
        :return: True, якщо задана вершина міститься у графі
        """
        if isinstance(vertex, Vertex):  # Якщо Vertex - вершина (не ім'я)
            return vertex.key() in self.mVertices
        else:                           # Якщо Vertex - ім'я (ключ) вершини
            return vertex in self.mVertices

    def __iter__(self):
        """ Ітератор для послідовного проходження всіх вершин у графі """
        return iter(self.mVertices.values())

    def __len__(self):
        """ Перевизначення функції len() як кількість вершин у графі

        :return: кількість вершин у графі
        """
        return self.mVertexNumber

    def __getitem__(self, vertex):
        return self.getVertex(vertex)
```

Звернемо увагу читача на те, що у вищенаведеному описі класу визначено додатково декілька методів (**inverse**(), **__contains__**(), **__iter__**(), тощо), що використовуються у інших методах графів або стануть у нагоді пізніше.

Запишемо код, що буде створювати неорієнтований граф, зображений на рисунку 7.1.4. Звернемо увагу читача, що конструктор класу, на відміну від попередньої реалізації графа у вигляді матриці суміжності, не вимагає інформації про початкову кількість вершин у графі. Це пов’язане з тим, що новостворений граф – порожній, тобто не містить жодної вершини. Таким чином, спочатку треба додати у граф вершини і лише потім встановити зв’язки між ними. Причому у граф можна додати будь-яку кількість вершин у будь-який момент і ця операція буде здійснена за сталий час. Останнє є суттєвою перевагою зображення графа у вигляді списку суміжності перед матрицею суміжності у задачах, де розмір та порядок графу можуть часто змінюватися. Для зручності метод додавання нового ребра **addEdge**() здійснює перевірку наявності вершин у графі і додає відповідну вершину, якщо вона відсутня.

Лістинг 7.1.5. Продовження. Побудова графа.

```
g = Graph()  # Створюємо неорієнтований граф

g.addEdge(1, 2)  # ребра (1, 2) та (2, 1)
g.addEdge(1, 3)  # ребра (1, 3) та (1, 3)
g.addEdge(1, 4)  # ребра (1, 4) та (4, 1)
g.addEdge(2, 3)  # ребра (2, 3) та (3, 2)
g.addEdge(2, 4)  # ребра (2, 4) та (4, 2)
g.addEdge(2, 5)  # ребра (2, 5) та (5, 2)
g.addEdge(3, 4)  # ребра (3, 4) та (4, 3)
g.addEdge(3, 5)  # ребра (3, 5) та (5, 3)
g.addEdge(3, 6)  # ребра (3, 6) та (6, 3)
g.addEdge(4, 5)  # ребра (4, 5) та (5, 4)
g.addEdge(4, 6)  # ребра (4, 6) та (6, 4)
g.addEdge(5, 6)  # ребра (5, 6) та (6, 6)
```

§7.2. Алгоритми на графах

7.2.1. Пошук у глибину

З алгоритмом пошуку в глибину ми вже ознайомилися під час вивчення дерев. Розглянемо тепер його розширення для графів.

Алгоритм пошуку в глибину (англ. Depth-first search, DFS) — алгоритм для обходу графа, у якому застосовується стратегія йти, на скільки це можливо, вглиб графа. Проте на відміну від дерев пошук в глибину (власне як і пошук в ширину) має одну особливість – при обході дерев, на відміну від графів, не може трапитися ситуація при якій ми спробуємо відвідати вершину у якій ми вже були. Дійсно, якщо розглядати дерево як граф, то напівстепінь входу кожної вершини не може бути більшим за 1 (0 для кореня і 1 для всіх його вузлів). Останнє означає, що в кожну вершину дерева можна потрапили єдиним шляхом. Іншою суттєвою відмінністю алгоритму пошуку в глибину для графа від аналогічного алгоритму для дерев є те, що ми змушені визначати початкову вершину з якої стартує пошук. Для дерев пошук завжди стартував з кореня дерева.

Нижче наведено порядок обходу графа, зображеного на рисунку 7.1.2. починаючи з вершини 1.

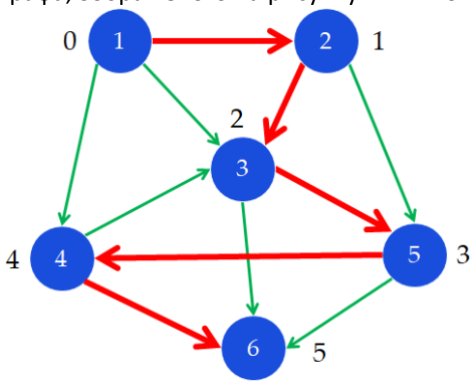


Рисунок 7.2.1. Обхід графа в глибину починаючи з вершини 1.

Вибір початкової вершини під час обходу графа в глибину є важливим з огляду на те, чи стоїть задача обійти всі вершини графа. Якби у попередньому прикладі у ролі початкової вершини вибрали не 1, а іншу вершину, наприклад 2 або 5 то ми б не змогли обійти всі вершини графа – вершина 1 є джерелом, а отже вона не є досяжною з жодної вершини графа. Нижче зображені обходи графа з вершин 2 та 5

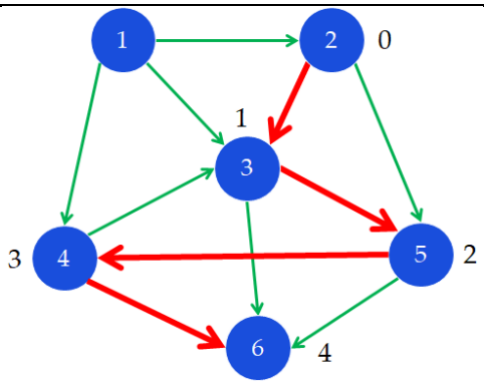


Рисунок 7.2.2. Обхід графа в глибину починаючи з вершини 2.

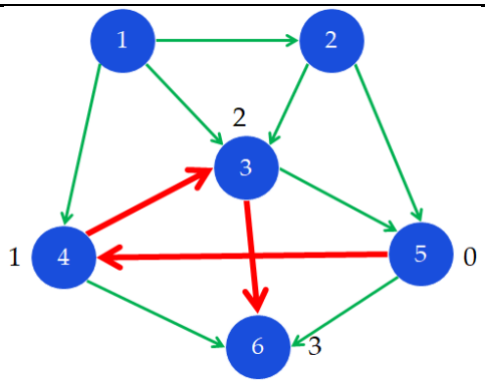


Рисунок 7.2.3. Обхід графа в глибину починаючи з вершини 5.

Алгоритм

Як і у випадку дерев, пошук в глибину реалізується рекурсивно. Ідея рекурсивного алгоритму пошуку в глибину полягає у тому, що починаючи з обраної вершини в графі, потрібно здійснити пошук в глибину для всіх її сусідів. Проте, на відміну від алгоритму для дерев, під час пошуку в глибину для графа, потрібно помічати вже відвідані вершини, щоб уникнути повторного їхнього опрацювання. Інакше, пошук в глибину зациклиться.

Реалізація

Реалізуємо пошук в глибину на Python використовуючи клас `Graph` реалізований вище на базі списку суміжності. Реалізація алгоритму складається з двох підпрограм. Перша, створює список у якому відмічаються відвідані вершини та запускає пошук в глибину. Друга – власне рекурсивний алгоритм обходу графа в глибину.

Лістинг 7.2.1. Пошук в глибину для графа

```
def DFS(graph, start):
    """ Обхід графа в глибину починаючи з заданої вершини

    :param graph: Граф
    :param start: Вершина з якої відбувається запуск обходу в глибину
    :return: Множина відвіданих вершин
    """

    visited = set()    # відвідані вершини
    __dfs_helper(graph, visited, start)  # запускаємо DFS з вершини start
    return visited

def __dfs_helper(graph, visited, start):
    """ Рекурсивний допоміжний метод, що реалізує обхід графа в глибину

    :param graph:   Граф
    :param visited: Відвідані вершин
    :param start:   Вершина з якої відбувається запуск обходу в глибину
    """
    print(start, end=" ")    # Опрацьовуємо елемент на вході
    visited.add(start)       # Помічаємо стартовий елемент як відвіданий
    # для всіх сусідів стартового елементу
    for neighbour in graph[start].neighbors():
        if neighbour not in visited:  # які ще не були відвідані
            __dfs_helper(graph, visited, neighbour)  # запускаємо DFS
```

Щоб перевірити роботу алгоритму створимо граф зображений на рисунку 7.1.2.

Лістинг 7.2.1. Продовження. Побудова графа перевірки роботи алгоритму пошуку в глибину.

```
g = Graph(True)  # Створюємо неорієнтований граф

g.addEdge(1, 2)  # ребро (1, 2)
g.addEdge(1, 3)  # ребро (1, 3)
```

```
g.addEdge(1, 4)  # ребро (1, 4)
g.addEdge(2, 3)  # ребро (2, 3)
g.addEdge(2, 5)  # ребро (2, 5)
g.addEdge(3, 5)  # ребро (3, 5)
g.addEdge(3, 6)  # ребро (3, 6)
g.addEdge(4, 3)  # ребро (4, 3)
g.addEdge(4, 6)  # ребро (4, 6)
g.addEdge(5, 4)  # ребро (5, 4)
g.addEdge(5, 6)  # ребро (5, 6)
```

Запустимо для нього пошук в глибину з вершин 1, 2, 5 та 4.

Лістинг 7.2.1. Продовження. Пошуку в глибину.

```
DFS(g, 1)
print()
DFS(g, 5)
print()
DFS(g, 4)
print()
DFS(g, 2)
```

Пошук в глибину для кожного випадку виведе на екран послідовність обходу вершин у графі:

```
1 2 3 5 4 6
5 4 3 6
4 3 5 6
2 3 5 4 6
```

Зауваження. Наведений вище алгоритм буде працювати доки не будуть пройдені всі вершини у графі, що є досяжними з стартової точки. Його важко модифікувати (а якщо можна, то це не є ефективним), так, щоб алгоритм припиняв свою роботу у випадку відшукання певної вершини, що є метою пошуку, оскільки, принаймні, необхідно вийти з усіх рекурсивних викликів функції DFS. Тому використання алгоритму пошуку в глибину є виправданим, якщо у будь-якому разі необхідно пройти всі вершини, що досяжні з заданої. Отже, алгоритм пошуку в глибину використовується для розв'язання задач, що пов'язані з досяжністю однієї вершини з іншої, причому довжина шляху не є важливою. Відповідно, DFS використовується для знаходження циклів у графі, перевірки чи є граф зв'язним тощо.

Алгоритмічна складність алгоритму пошуку в глибину

Алгоритмічна складність пошуку в глибину залежить від того, як реалізований граф. Так, наприклад, у випадку реалізації графа через матрицю суміжності алгоритмічна складність пошуку в глибину буде $O(n^2)$ якщо граф має n вершин. У випадку реалізації графу через список суміжності, його алгоритмічна складність буде порядку m у середньому, та $O(n+m)$ у найгіршому випадку, де m – кількість ребер у графі.

7.2.2. Пошук у ширину

Аналогічно до пошуку в глибину, з пошуком в ширину ви вже знайомі з попереднього розділу.

Пошук у ширину (англ. breadth-first search, BFS) — алгоритм пошуку на графі, у якому застосовується стратегія послідовного перегляду окремих рівнів графа, починаючи з заданого вузла. Нижче наведено порядок обходу графа, зображеного на рисунку 7.1.4 починаючи з вершини 1.

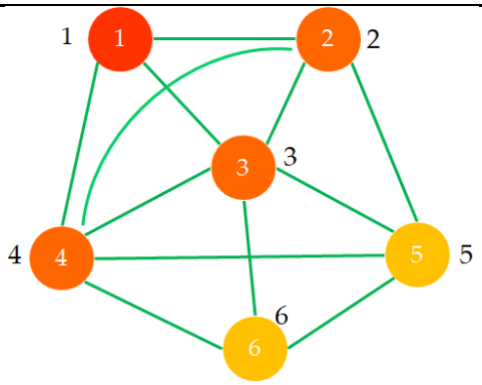


Рисунок 7.2.4. Обхід графа в ширину починаючи з вершини 1.

Алгоритм

Для послідовного перегляду окремих рівнів графа, починаючи з заданого вузла, використовується черга. Спочатку у цю чергу додається стартова вершина графа, яка зразу позначається як така, що відвідана. Далі алгоритм полягає у такому, що поки ця черга не порожня ми вилучаємо з черги черговий елемент, опрацьовуємо його, додаємо до черги всіх його сусідів, що не були до цього відвідані, позначаючи їх відвіданими.

Реалізація

Реалізуємо пошук в ширину на Python використовуючи клас `Graph` реалізований вище на базі списку суміжності.

Лістинг 7.2.2. Пошук в ширину для графа

```
def BFS(graph, start):
    """ Обхід графа в ширину починаючи з заданої вершини

    :param graph: Граф
    :param start: Вершина з якої відбувається запуск обходу в ширину
    :return: Список (множину) відвіданих вершин
    """

    visited = set()     # відвідані вершини

    q = Queue()          # Створюємо чергу
    q.enqueue(start)     # Додаємо у чергу стартову вершину
    visited.add(start)   # та позначаємо її як відвідану

    while not q.empty():       # Поки черга не порожня
        current = q.dequeue()  # Беремо перший елемент з черги

        print(current)         # Опрацьовуємо взятий елемент

        # Додаємо в чергу всіх сусідів поточного елементу
        for neighbour in graph[current].neighbors():
            if neighbour not in visited:  # які ще не були відвідані
                q.enqueue(neighbour)
                visited.add(neighbour)    # Помічаємо як відвідану

    return visited
```

Зауважимо, що наведений вище алгоритм також виконується доки не будуть пройдені всі вершини графа, досяжні зі стартової. Проте, цей алгоритм легко модифікувати, якщо задача поставлена знайти потрібний елемент у графі. Для цього, якщо цільова вершина знайдена достатньо всього лиш перервати роботу циклів.

Алгоритмічна складність

Як і у випадку пошуку в глибину, алгоритмічна складність пошуку в ширину залежить від того, як реалізований граф. У випадку реалізації графа через матрицю суміжності алгоритмічна складність пошуку в ширину буде $O(n^2)$ якщо граф має n вершин. Тоді, як для у випадку реалізації графу через список суміжності, його алгоритмічна складність буде $O(n+m)$ у найгіршому випадку, де m – кількість ребер у графі.

7.2.3. Хвильовий алгоритм

Виявляється пошук в ширину не лише може знайти вершину у графі, якщо вона досяжна з деякої заданої, але і знайти довжину найкоротшого шляху до цієї вершини зі стартової. Найпростіший спосіб здійснити це – застосувати хвильовий алгоритм, який базується на пошуку в ширину.

Алгоритм

Ідея хвильового алгоритму полягає у тому, що ми запускаємо BFS і при цьому для кожної вершини, крім того чи відвідана вона чи ні, ще будемо пам'ятати відстань від стартової точки до неї. Для цього будемо вважати, що відстань від стартової точки до себе є нулем. А далі під час додавання точки до черги (якщо вона ще не була відвідана) будемо встановлювати для неї цю відстань на одиницю більшу ніж для поточної.

Наприклад, для такого неорієнтованого графа зображеного на рисунку 7.1.4, відстань від вершини 1 буде схематично зображувати таким чином

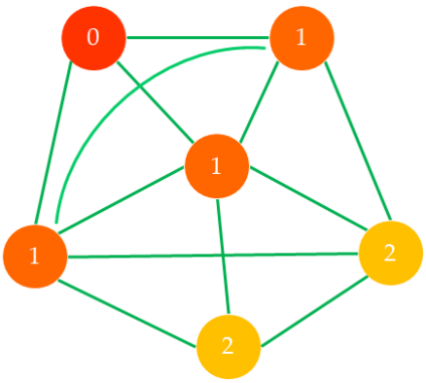


Рисунок 7.2.5. Відстань від вершини 1 до всіх вершин графа.

Звідки можна зробити висновок, що, наприклад, найкоротша відстань від вершини 1 до вершини 5 буде дорівнювати 2.

Реалізація

Наведемо реалізацію хвильового алгоритму. Цей алгоритм є модифікацією вищенаведеного алгоритму пошуку в ширину, що полягає у тому, що будемо запам'ятовувати в окремій структурі не лише чи була відвідана вершина, але й відстань до неї зі стартової точки.

Лістинг 7.2.3. Хвильовий алгоритм

```
def wave(graph, start):
    """ Хвильовий алгоритм, що використовує масив
        відстаней від стартової точки до поточної
        для визначення чи була вже відвідана вершина

    :param graph: Граф
    :param start: Вершина з якої починається обхід
    :return: Список відстаней від стартової вершини до кожної вершини графа
    """
    q = Queue()       # Створюємо чергу
    q.enqueue(start)  # Додаємо у чергу стартову вершину

    # Словник, що для всіх вершин містить відстані від стартової вершини start.
    distances = {start: 0}  # Відстань від стартової точки до себе нуль.

    while not q.empty():
        current = q.dequeue()  # Беремо перший елемент з черги
        # Додаємо в чергу всіх сусідів поточного елементу
        for neighbour in graph[current].neighbors():
            if neighbour not in distances:  # які ще не були відвідані
                q.enqueue(neighbour)
                distances[neighbour] = distances[current] + 1

    return distances
```

Результат роботи алгоритму для графа g зображеного на рисунку 7.1.4 (створеного в лістингу 7.1.5) що стартує з вершини 1

Лістинг 7.2.3. Продовження. Хвильовий алгоритм з вершини 1.

```
distances = wave(g, 1)
print(distances)
```

буде таким

```
{1: 0, 2: 1, 3: 1, 4: 1, 5: 2, 6: 2}
```

7.2.4. Пошук найкоротшого шляху

Вище наведено алгоритм як знайти найкоротшу відстань між двома вершинами у графі. Проте питання про те як знайти сам цей найкоротший шлях поки що залишалося відкритим. Виявляється хвильовий алгоритм досить легко для цього адаптувати.

Алгоритм

Ідея такої модифікації хвильового алгоритму, щоб власне знайти найкоротший шлях полягає у тому, що нам потрібно не запам'ятовувати відстань від стартової точки до заданої, а вершину (тобто маркер) з якої ми прийшли по цьому найкоротшому шляху. Тоді, для відшукання шляху необхідно буде лише відновити шлях від кінцевої точки до початкової по встановлених маркерах.

Для прикладу, розглянемо вищенаведений граф. Запам'ятаємо вершини з яких ми прийшли, як показано на рисунку нижче.

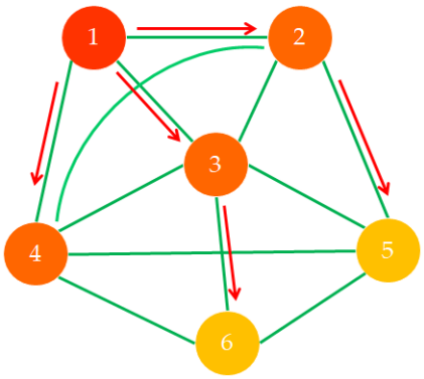


Рисунок 7.2.6. Пошук шляху у графі.

Тепер, як бачимо, наприклад, найкоротший шлях від вершини 1 до вершини 5 можна знайти як:

- у 5 ми прийшли з 2
- у 2 ми прийшли з 1

Таким чином шлях буде виглядати $1 \to 2 \to 5$. Відповідно його довжина буде 2.

Як бачимо для того щоб побудувати шлях, нам треба пройти за маркерами у зворотному порядку. Таку операцію зручно робити за допомогою стеку.

Реалізація алгоритму

Лістинг 7.2.4. Пошук найкоротшого шляху

```
INF = sys.maxsize  # Умовна нескінченність

def waySearch(graph, start, end):
    """ Пошук найкоротшого шляху між двома заданими вершинами графа

    :param graph: Граф
    :param start: Початкова вершина
    :param end:   Кінцева вершина
    :return: список вершин найкоротшого шляху, що сполучає вершини start та end
    """

    assert start != end
```

```
    # Словник, що для кожної вершини (ключ) містить ключ вершини з якої прийшли у поточну
    sources = {start: None}  # Для стартової вершини не визначено звідки в неї прийшли.

    q = Queue()           # Створюємо чергу
    q.enqueue(start)      # Додаємо у чергу стартову вершину

    while not q.empty():
        current = q.dequeue()  # Беремо перший елемент з черги
        # Додаємо в чергу всіх сусідів поточного елементу
        for neighbour in graph[current].neighbors():
            if neighbour not in sources:  # які ще не були відвідані
                q.enqueue(neighbour)
                # при цьому для кожної вершини запам'ятовуємо вершину з якої прийшли
                sources[neighbour] = current

    if end not in sources:  # шляху не існує
        return None

    # будуємо шлях за допомогою стеку
    stack = Stack()
    current = end
    while True:
        stack.push(current)
        if current == start:
            break
        current = sources[current]

    way = []  # Послідовність вершин шляху
    while not stack.empty():
        way.append(stack.pop())

    # Повертаємо шлях
    return way
```

Знову виконаємо цей алгоритм для графа g зображеного на рисунку 7.1.4 (створеного в лістингу 7.2.1), що стартує з вершини 1

Лістинг 7.2.4. Продовження. Пошук найкоротшого шляху з вершини 1 у вершину 5.

```
way = waySearch(g, 1, 5)
print(way)
```

Результатом роботи буде список вершин шляху у першому рядку та його довжина

```
[1, 2, 5]
```

7.2.5. Топологічне сортування

Одним із застосувань орієнтованих ациклічних графів є моделювання різноманітних процесів (технологічних, біологічних, бізнес-процесів, тощо), що складаються з великої кількості модулів, які, часто, виконуються незалежно один від одного та взаємодіють між собою за визначеними алгоритмами. Для такого процесу граф показує, коли має стартувати відповідний модуль, коли можлива взаємодія між різними модулями і у якому порядку. Наприклад, на загальному рівні, технологічний процес виробництва автомобіля, спочатку вимагає його проектування. Далі необхідно виготовити складові частини (причому всі ці складові як правило виготовляються незалежно одна від іншої). Далі необхідно доставити необхідні складові на відповідні підприємства для виготовлення великих вузлів автомобіля (мотору, трансмісії, шасі, кузову, тощо). Завершується процес виготовлення автомобіля велико-вузловим складанням. Напевно читачу зрозуміло, що порядок виробництва є чітко визначеним і, взагалі кажучи, не може бути зміненим. Дійсно, ви не можете вставити мотор у автомобіль, якщо мотор ще не виготовлений. І не можете пофарбувати автомобіль, якщо його кузов все ще знаходиться на етапі виготовлення.

Тепер поглянемо на технологічний процес виготовлення з іншого боку – контролю якості. І нехай цю задачу поставлено перед деякою абстрактною аудиторською компанією. Компанія вирішує, що найкращий спосіб переконатися у якості автомобіля, це проаналізувати повний цикл виробництва одного автомобіля – від виробництва найдрібніших деталей до тестування готового авто. Причому, для того, щоб результат був

об'єктивний стосовно усіх модулів виробництва, компанія вирішила покласти цю задачу на єдиного контролера, який має послідовно перевірити всі етапи. А отже, перед компанією постає задача: у якому порядку має рухатися контролер, відвідуючи модулі виробництва, щоб не пропустити жодного, бодай найдрібнішого, модуля.

Відповідь на це запитання така: потрібно застосувати алгоритм, топологічного сортування графа, що моделює виробництво автомобіля.

Означення 7.2.1. Топологічне сортування — впорядковування вершин ациклічного орієнтованого графа відповідно до часткового порядку, що визначається ребрами цього графу на множині його вершин.

Іншими словами, під топологічним сортуванням ациклічного графа розуміють процес лінійного впорядковування його вершин таким чином, що якщо в графі існує ребро (v,u), то, в упорядкованому списку вершин графа, вершина v передує вершині u.

Для прикладу, розглянемо ациклічний граф зображений на рисунку 7.1.10. Припустимо, що цей граф моделює деякий технологічний процес, наприклад, описаний вище процес виробництва автомобіля. Очевидно, для того, щоб відвідати всі його вершини потрібно рухатися починаючи від вершини 1 у вершину 6, оскільки вершина 1 є джерелом, а вершина 6 – стоком. Причому, вершина 4 не може бути відвідана раніше ніж вершини 3 та 5, від яких вона залежить, а вершина 3, у свою чергу не може бути відвіданою раніше вершини 2. Отже, проаналізувавши залежності одних вершин від інших можемо сказати, що порядок вершин обходу має бути таким:

$$1 \to 2 \to 3 \to 5 \to 4 \to 6$$

Слід зауважити, що в загальному випадку, порядок вершин отриманих в результаті топологічного сортування графа, не єдиний. Наприклад, для графа, зображеного на рисунку 7.2.7 нижче

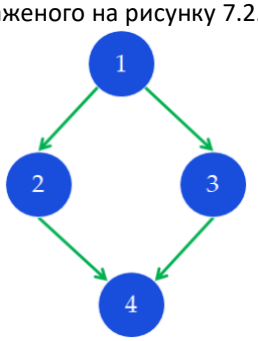


Рисунок 7.2.7. Ациклічний граф для якого існує два різних топологічних сортування.

в результаті топологічного сортування можна отримати дві різних послідовності вершин

$$1 \to 2 \to 3 \to 4 \text{ та } 1 \to 3 \to 2 \to 4$$

Графи зображені на рисунках 7.1.10 та 7.2.7 є досить простими. Для них отримати порядок обходу його вершин відповідно до топологічного сортування не склало труднощів. Проте, якщо граф є складнішим, питання топологічного сортування його вершин вже є нетривіальним і вимагає застосування формалізованих алгоритмів. Найпростіший з цих алгоритмів базується на алгоритмі пошуку в глибину. Фактично він полягає у тому, щоб відсортувати вершини за часом виходу під час пошуку в глибину. Для прикладу, знову розглянемо граф зображений на рисунку 7.1.10. Код його створення наведено нижче

Лістинг 7.2.5. Створення ациклічного орієнтованого графу

```
g = Graph(True)  # Створюємо орієнтований граф

g.addEdge(1, 2)  # ребро (1, 2)
g.addEdge(1, 3)  # ребро (1, 3)
g.addEdge(1, 4)  # ребро (1, 4)
g.addEdge(2, 3)  # ребро (2, 3)
g.addEdge(2, 5)  # ребро (2, 5)
g.addEdge(3, 4)  # ребро (3, 4)
g.addEdge(3, 5)  # ребро (3, 5)
g.addEdge(3, 6)  # ребро (3, 6)
g.addEdge(4, 6)  # ребро (4, 6)
g.addEdge(5, 4)  # ребро (5, 4)
g.addEdge(5, 6)  # ребро (5, 6)
```

Модифікуємо допоміжний метод **__dfs_helper**() з лістингу 7.2.1, таким чином, щоб під час входження у вершину з ключем Key (тобто під час входу у рекурсивну функцію **__dfs_helper**() для вершини Key) виводилось повідомлення

```
-> Key
```

а під час виходу з вершини (тобто виходу з рекурсивної функції **__dfs_helper**()) повідомлення

```
<- Key (вихід)
```

Лістинг 7.2.5. Продовження. Модифікація допоміжного методу пошуку в глибину.

```
def __dfs_helper(graph, visited, start):
    """ Рекурсивний допоміжний метод, що реалізує обхід графа в глибину

    :param graph:   Граф
    :param visited: Відвідані вершин
    :param start:   Вершина з якої відбувається запуск обходу в глибину
    """
    print("->", start)       # входження у вершину з ключем start
    visited.add(start)       # Помічаємо стартовий елемент як відвіданий
    # для всіх сусідів стартового елементу
    for neighbour in graph[start].neighbors():
        if neighbour not in visited:  # які ще не були відвідані
            __dfs_helper(graph, visited, neighbour)  # запускаємо DFS
    print("<-", start, "(вихід)")    # вихід з вершини start
```

Тепер запустимо для цього графа обхід в глибину

Лістинг 7.2.5. Продовження. Запуск пошуку в глибину .

```
DFS(g, 1)  # запуск пошуку в глибину починаючи з вершини 1
```

Результат роботи алгоритму наведено нижче

```
-> 1
-> 2
-> 3
-> 4
-> 6
<- 6 (вихід)
<- 4 (вихід)
-> 5
<- 5 (вихід)
<- 3 (вихід)
<- 2 (вихід)
<- 1 (вихід)
```

Якщо уважно проаналізуємо результат, то побачимо, що порядок виходу з вершин (якщо рахувати з кінця) такий же як і отриманий раніше емпіричним чином, порядок при топологічному сортуванні. Це не випадковість: якщо після закінчення обходу в глибину всіх вершин, записати отриману послідовність у зворотному порядку, то отримаємо топологічне сортування графа.

Доведемо, що отримана таким чином послідовність вершин дійсно є топологічним сортуванням графа. Для цього покажемо, що якщо в графі існує ребро (v, u), то в отриманому списку вершин, вершина v передує вершині u. Доведення проведемо від супротивного. Припустимо що в графі існує ребро (v, u), проте вершина u зустрічається в упорядкованій послідовності раніше вершини v. Останнє означає, що вихід з вершини v відбувся раніше, ніж з вершини u. Відповідно, у момент виходу з вершини v

- або вершина u ще не була досягнута;
- або ще не відбувся вихід з вершини u.

У першому випадку ми повинні були б пройти по ребру (v, u) далі вглиб графа (згідно з алгоритмом пошуку в глибину), але не зробили цього. Другий випадок означає, що пошук в глибину знайшов цикл. Отримали суперечність, яка засвідчує, що отримана послідовність є топологічним сортуванням нашого графа.

Підсумовуючи викладки наведені вище, отримаємо алгоритм топологічного сортування графу на основі пошуку в глибину:

1. Запускаємо пошук в глибину для графа.
2. Зберігаємо вершини в списку у порядку спадання їхніх "моментів" виходу з вершин.

3. Повертаємо упорядкований список як результат топологічного сортування.

Зауважимо, що під час обґрунтування алгоритму ми побічно отримати алгоритм пошуку циклів у орієнтованих графах.

Отже, наведемо алгоритм топологічного сортування, з можливістю виявлення циклів (якщо вони містяться у графі). Доповнимо клас `Vertex`, описаний у лістингу 7.1.4 полями, що будуть позначати на якому етапі знаходиться вершина під час роботи алгоритму пошуку в глибину. Для цього будемо додатково вважати, що вершина додатково має колір:

- Білий (WHITE) – якщо вершина ще не відвідана;
- Сірий (GRAY) – якщо під час пошуку в глибину увійшли у вершину;
- Чорний (BLACK) – якщо під час пошуку в глибину вийшли з вершини.

Лістинг 7.2.6. Розширення класу `Vertex`.

```
WHITE = 0  # Вершина білого кольору - вершина ще не відвідана
GRAY = 1   # Вершина сірого кольору - під час DFS ввійшли у вершину
BLACK = 2  # Вершина чорного кольору - під час DFS вийшли з вершини

class ColorVertex(Vertex):
    """ Допоміжний клас, (нащадок класу Vertex)

        має поле mColor - колір вершини, що
        використовується для алгортму топологічного сортування
    """
    def __init__(self, key):
        super().__init__(key)
        self.mColor = WHITE

    def setColor(self, color):
        self.mColor = color

    def color(self):
        return self.mColor
```

Далі необхідно перевизначити метод додавання вершин у графі, так, щоб використовувався щойно описаний клас. Для цього опишемо клас `ColorGraph`, що є нащадком класу `Graph` з перевизначеним методом **`addVertex()`**.

Лістинг 7.2.6. Продовження. Клас `ColorGraph`, вершини якого екземпляри класу `ColorVertex`.

```
class ColorGraph(Graph):
    """ Граф, що містить вершини ColorVertex"""

    def addVertex(self, vertex):
        """ Додає вершину у граф, якщо така вершина не міститься у ньому

        :param vertex: ключ (тобто ім'я) нової вершини
        :return: True, якщо вершина успішно додана
        """

        if vertex in self:  # Якщо вершина міститься у графі, її вже не треба додавати
            return False

        new_vertex = ColorVertex(vertex)     # створюємо нову вершину з іменем vertex
        self.mVertices[vertex] = new_vertex  # додаємо цю вершину до списку вершин графу
        self.mVertexNumber += 1              # Збільшуємо лічильник вершин у графі
        return True
```

Нарешті наведемо алгоритм

Лістинг 7.2.7. Топологічне сортування.

```
def __dfs_helper(graph, vertex, stack):
    """ Рекурсивний допоміжний метод, що реалізує топологічне
        сортування використовуючи пошук в глибину
```

```
    :param graph:  Граф
    :param vertex: вершина з якої починається пошук в глибину
    :param stack:  поточний стек відсортовних вершин
    :return: None
    """

    if vertex.color() == BLACK:  # вершина повністю опрацьована - вийшли з неї
        return

    if vertex.color() == GRAY:   # Істиність цієї умови означає, що знайдено цикл,
        raise Exception          # подальша робота алгоритму не має сенсу

    # Входимо у вершину
    vertex.setColor(GRAY)       # помічаємо вершину сірим кольором, тобто ввійшли в неї
    for neighbour_key in graph[vertex].neighbors():  # для сусідів стартового елементу
        neighbour = graph[neighbour_key]
        __dfs_helper(graph, neighbour, stack)  # запускаємо DFS

    # Виходимо з вершини
    vertex.setColor(BLACK)    # Помічаємо вершину як опрацьовану (чорним кольором)
    stack.push(vertex.key())  # Вставляємо вершину у список відсортованих вершин

def topological_sorting(graph):
    """ Функція топологічного сортування вершин графа

    :param graph: граф
    :return: список топологічно відсортованих вершин
    """

    stack = Stack()        # Стек, що буде містити відсортовані елементи
    for vertex in graph:   # для всіх вершин графа
        # запускаємо пошук в глибину
        __dfs_helper(graph, vertex, stack)

    # Створюємо список топологічно відсортованих вершин
    sequence = []
    while not stack.empty():
        sequence.append(stack.pop())

    return sequence  # Повертаємо список, що містить відсортовані елементи
```

Звернемо увагу читача, що у функції **topological_sorting()**, пошук в глибину ініціюється з усіх вершин графа. Така необхідність пов'язана з тим, що граф може містити вершини, що є недосяжними з інших вершин. Такий підхід може суттєво відобразиться на швидкодії алгоритму, навіть не зважаючи на те, що для вже опрацьованих вершин рекурсивний метод **__dfs_helper()** буде припиняти роботу вже на самому початку підпрограми. Тому для багатьох прикладних задач є потреба у детальному початковому аналізі вхідного графа з метою оптимізації алгоритму. Наприклад, якщо граф має лише одне джерело і один сток, то природньо запускати пошук в глибину лише з вершини-джерела. Якщо ж граф є сильно-зв'язним, то початкова вершина для алгоритму топологічного сортування взагалі не є принциповою – починаючи з неї будуть пройдені всі вершини графа.

Лістинг 7.2.7. Продовження. Топологічне сортування.

```
g = ColorGraph(True)  # Створюємо орієнтований граф

g.addEdge(1, 2)  # ребро (1, 2)
g.addEdge(1, 3)  # ребро (1, 3)
g.addEdge(1, 4)  # ребро (1, 4)
g.addEdge(2, 3)  # ребро (2, 3)
g.addEdge(2, 5)  # ребро (2, 5)
g.addEdge(3, 4)  # ребро (3, 4)
g.addEdge(3, 5)  # ребро (3, 5)
g.addEdge(3, 6)  # ребро (3, 6)
g.addEdge(4, 6)  # ребро (4, 6)
g.addEdge(5, 4)  # ребро (5, 4)
g.addEdge(5, 6)  # ребро (5, 6)

s = topological_sorting(g)
```

```
print(*s)
```

Результатом роботи алгоритму буде

```
1 2 3 5 4 6
```

7.2.6. Зв'язність графів

Алгоритми, що стосуються зв'язності графів здебільшого ґрунтуються на наведених у цьому параграфі алгоритмах пошуку в глибину або ширину. Наприклад, щоб перевірити чи неорієнтований граф є зв'язним досить запустити пошук (у глибину/ширину) з деякої вершини та перевірити чи всі вершини графа були відвідані.

Лістинг 7.2.8. Перевірка неорієнтованого графа на зв'язність.

```
def __check_connected_helper(graph: Graph, start):
    """ допоміжний метод, який використовуючи пошук в глибину,
    перевіряє чи існує шлях від вершини start до всіх вершин графа

    :param graph: Граф
    :param start: Початкова вершина графа
    :return: True, якщо шлях існує та False, якщо ні
    """

    visited = DFS(graph, start)

    for v in graph.vertices():
        if v not in visited:  # якась з вершин не була відвідана під час обходу в глибину
            return False      # то граф не є зв'язним

    return True  # якщо не знайдено невідвіданих вершин - граф зв'язний


def checkConnected(graph: Graph):
    """ Перевіряє чи є неорієнтований граф зв'язним

    :param graph: Граф
    :return: True, якщо граф є зв'язним та False, якщо ні
    """
    assert not graph.mIsOriented     # Перевіряємо, що граф є не орієнтованим
    return __check_connected_helper(graph, 1)
```

Вищенаведений алгоритм можна модифікувати таким чином, щоб не лише перевіряти чи є граф зв'язним, але й знайти кількість його компонент зв'язності. І тоді, якщо граф матиме одну компоненту зв'язності, то він зв'язний. Для цього, як і раніше будемо використовувати пошук в глибину/ширину, щоб вияснити чи є одні вершини досяжними з інших. При цьому, замість того, щоб просто відмічати чи була вершина відвідана чи ні, будемо додатково зазначати номер компоненти зв'язності до якої належить вершина. Для цього будемо використовувати словник, ключами у якому будуть ключі відповідних вершин, а значеннями – номери компоненти зв'язності, до яких належить відповідна вершина.

Лістинг 7.2.9. Пошук кількості компонент зв'язності графа.

```
def __dfs_helper(graph, visited, start, connected_component):
    """ Рекурсивний допоміжний метод, що реалізує обхід графа в глибину
    При цьому, для відвіданих вершин зберігається інформація
    про їхні компоненти зв'язності

    :param graph:   Граф
    :param visited: Словник відвіданих вершин
    :param start:   Вершина з якої відбувається запуск обходу в глибину
    :param connected_component: Номер поточної компоненти зв'язності
    """
    visited[start] = connected_component   # Помічаємо стартовий елемент як відвіданий
                                           # та запам'ятовуємо його компоненку зв'язності
    # для всіх сусідів стартового елементу
```

```
    for neighbour in graph[start].neighbors():
        if neighbour not in visited:  # які ще не були відвідані
            __dfs_helper(graph, visited, neighbour, connected_component) # запускаємо DFS

def findConnectedComponent(graph: Graph):
    """ Перевіряє чи є неорієнтований граф зв'язним

    :param graph: Граф
    :return: Кількість компонент зв’язності неорієнтованого графа
    """
    assert not graph.mIsOriented  # Перевіряємо, що граф є не орієнтованим

    visited = {}    # Словник відвіданих вершин, містить пари
                    # (вершина: номер_компоненти_зв'язності)

    connected_component = 0   # Кількість компонент зв'язності графа
    for v in graph.vertices():
        if v not in visited:  # Якщо якась з вершин була не відвідана під час обходу
            connected_component += 1  # то з'явилася нова копонента зв'язності графа
            # запускаємо DFS з невідвіданої вершини
            __dfs_helper(graph, visited, v, connected_component)

    print(visited)     # Для тестування програми, виведемо відвідані вершини разом з
                       # номерами знайдених компонент зв'язності, до яких вони належать

    return connected_component
```

Перевірку роботи алгоритму здійснимо для графу, зображеному на рисунку 7.1.11.

Лістинг 7.2.9. Продовження. Пошук кількості компонент зв’язності графа

```
g = Graph()  # Створюємо неорієнтований граф

# Перша компонента зв'язності
g.addEdge(1, 4)  # ребро (1, 4)
g.addEdge(1, 7)  # ребро (1, 7)
g.addEdge(4, 7)  # ребро (4, 7)

# Друга компонента зв'язності
g.addEdge(2, 3)  # ребро (2, 3)
g.addEdge(2, 5)  # ребро (2, 5)
g.addEdge(3, 5)  # ребро (3, 5)
g.addEdge(3, 6)  # ребро (3, 6)
g.addEdge(5, 6)  # ребро (5, 6)

print("Кількість компонент зв'язності: ", findConnectedComponent(g))
```

Результатом роботи програми буде

```
{1: 1, 4: 1, 7: 1, 2: 2, 3: 2, 5: 2, 6: 2}
Кількість компонент зв'язності:  2
```

Як бачимо, вершини 1, 4 та 7 належать до першої компоненти зв’язності, а інші вершини 2, 3, 5 та 6 – до другої.

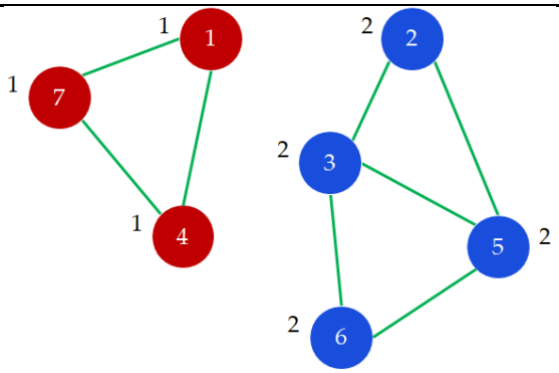


Рисунок 7.2.8. Пошук компонент зв'язності неорієнтованого графа.

Перейдемо тепер до орієнтованих графів. Як вище було зазначено, поняття зв'язності для орієнтованих графів є некоректним і замість нього використовуються інші, адаптовані до орієнтованих графів поняття. Зупинимося детально на дослідженні сильної зв'язності графів. Зокрема нас буде цікавити чи заданий граф є сильно-зв'язним, або скільки компонент сильної зв'язності він має у іншому разі.

Отже, спробуємо описати алгоритм, що дасть відповідь на питання чи є граф сильно-зв'язним. Перше, що може прийти в голову, це діяти згідно з означенням, тобто для всіх пар вершин (v_i, v_j), $i = 1, \dots, n$, знайти чи досяжна вершина v_i з вершини v_j та навпаки. Для цього досить запустити пошук в глибину/ширину з кожної вершини графа. Проте такий алгоритм є не оптимальним. Дійсно, якщо згадати, що алгоритм пошуку в глибину/ширину для графа, що має n вершин та m ребер має складність $O(n+m)$ ($O(n^2)$ у випадку зображення графа за допомогою матриці суміжності), то асимптотична складність запропонованого алгоритму буде $n \cdot O(n+m)$ ($n \cdot O(n^2)$ у випадку зображення графа за допомогою матриці суміжності).

Теорема наведена нижче дозволяє запропонувати алгоритм перевірки чи є граф сильно-зв'язним, причому асимптотична складність цього алгоритму залишиться такою ж як в алгоритмі пошуку в глибину/ширину.

Теорема 7.2.1. Нехай у орієнтованому графі G для довільно-фіксованої вершини v виконують такі твердження:

1) Всі вершини графа є досяжними з вершини v.
2) Вершина v є досяжною з будь-якої вершини графа.

Тоді граф є сильно зв'язним.

Доведення цієї теореми є тривіальним. Дійсно, для її доведення необхідно показати, що існує шлях між двома довільними вершинами графа v_i та v_j $i, j = 1, \dots, n$. З першої умови теореми випливає, що існує шлях від вершини v до вершини v_j:

$$v \rightsquigarrow v_j.$$

Друга умова теореми гарантує, що існує шлях від вершини v_i до вершини v.

$$v_i \rightsquigarrow v.$$

Отже,

$$v_i \rightsquigarrow v \rightsquigarrow v_j.$$

Застосуємо вищенаведену теорему для побудови алгоритму. Першу умову перевірити легко – досить лише запустити пошук у глибину/ширину з будь-якої фіксованої вершини графа v і, по завершенню його роботи, переконатися, що всі вершини протягом його роботи були відвідані. Для перевірки другої умови теореми потрібно побудувати граф G^T, що є транспонованим по відношенню до початкового, далі запустити пошук з вершини v та, по завершенню його роботи, переконатися, що всі вершини були відвідані. Звернемо увагу читача на те, що складність побудови транспонованого графа є такою ж як пошук в глибину/ширину, тому результуюча складність алгоритму залишиться такою ж як у алгоритмах пошуку.

Тепер можемо навести код перевірки чи є граф сильно-зв'язним.

Лістинг 7.2.10. Перевірка чи орієнтований граф є сильно-зв'язним.

```
def checkStrongConnected(graph: Graph):
    """ Перевіряє чи є орієнтований граф сильно зв'язним,
    використовується пошук в глибину
```

```
    :param graph: Граф
    :return: True, якщо граф є сильно зв'язним та False, якщо ні
    """

    assert graph.mIsOriented  # Перевіряємо, що граф є орієнтованим

    # Перевіряємо перше твердження теореми:
    # 1. З деякої заданої фіксованої вершини існує шлях до всіх вершин графа
    if not __check_connected_helper(graph, 1):
        return False

    # Перевіряємо друге твердження теореми:
    # 2. З кожної вершини графа існує шлях до деякої заданої фіксованої.
    graph_inv = graph.transpose()  # Побудова графу, транспонованого до заданого
    return __check_connected_helper(graph_inv, 1)
```

Як бачимо ця функція є дуже простою, оскільки використовує раніше описані функції та методи. Зокрема допоміжна функція `__check_connected_helper()` була описана у лістингу 7.2.8, а метод побудови транспонованого графа до заданого у лістингу 7.1.5.

Завершимо викладку матеріалу цього пункту алгоритмом знаходження компонент сильної зв’язності орієнтованого графа. **Компонентою сильної зв'язності** орієнтованого графу G називається найбільший сильно-зв'язаний підграф. Якщо кожну компонента сильної зв'язності стягнути до однієї вершини, отримаємо орієнтований ациклічний граф, що називається **ущільненням** (або конденсацією від англ. condensation) графа G. Орієнтований граф є ациклічним тоді і лише тоді, коли він не має компонент сильної зв'язності з більш як однією вершиною, бо орієнтований цикл є сильно зв'язним і кожна нетривіальна компонента сильної зв'язності графа містить щонайменше один орієнтований цикл.

Наведемо алгоритм Косараджу пошуку компонент сильної зв’язності графа. Він базується на кількох кроках наведених нижче.

1. Запускаємо пошук в глибину для графа G, та обчислюємо "час" виходу з кожної вершини;
2. Будуємо G^T.
3. Запускаємо пошук в глибину для графа G^T, при цьому в основному циклі пошуку в глибину будемо досліджувати кожну вершину в порядку спадання "часу" її виходу отриманому на кроці 1.
4. Кожне дерево лісу, знайденого на кроці 3, є компонентою сильної зв’язності.

§7.3. Пошуки шляхів у лабіринтах

Теорія графів дозволяє розв’язувати найрізноманітніші задачі, серед яких алгоритми обробки лабіринтів. Окремим класом лабіринтів, є лабіринти, що зображуються двомірною (або тривимірною) прямокутною сіткою у якій кожна клітина має певне (наперед визначене) навантаження – тобто її стан (вільна клітина, стіна, тощо). Причому, вважається, що клітина сітки одночасно може бути лише у одному з кількох визначених наперед станів.

Як правило, всі лабіринти у таких задачах мають клітини, що є

- вільними – клітини лабіринту у які можна пересуватися з інших клітин;
- зайнятими – клітини лабіринту, пересування у які забороняється умовою задачі (стіна, небезпечна ділянка).

На рисунку 7.3.1 зображено двовимірний лабіринт у якому білим кольором позначаються вільні клітини, а сірим – відповідно зайняті. Крім зазначених вище типів клітин, можуть бути й інші, навантаження яких додатково визначається умовою задачі.

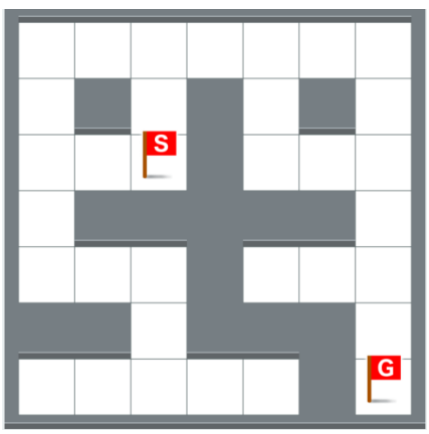


Рисунок 7.3.1. Двовимірний прямокутний лабіринт.

Рух у такому лабіринті за один крок допускається лише з поточної до сусідніх клітин. При цьому рух з поточної до сусідніх клітин, для двовимірного (тобто, плоского лабіринту) може здійснюватися у чотирьох або восьми напрямках (що визначається умовою задачі)

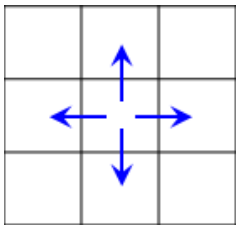


Рисунок 7.3.2. Рух у лабіринті дозволяється у чотирьох напрямках.

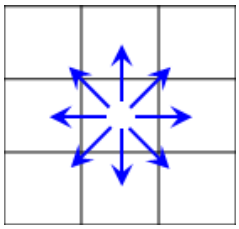


Рисунок 7.3.3. Рух у лабіринті дозволяється у восьми напрямках.

У тривимірному лабіринті напрямків буде відповідно 6 або 26. Пропонуємо читачеві самостійно схематично їх зобразити.

7.3.1. Моделювання лабіринту

Зображення лабіринту за допомогою матриці

Зображення лабіринту у пам'яті комп'ютера звичайно можна провести за допомогою графів (тобто матриці або списку суміжності) як це здійснювалося вище. Проте це не є зручним способом, як з точки зору зручності для подальшого програмування, так і з точки зору швидкодії. Тому у таких задачах доцільно для зображення лабіринту використовувати двовимірні або тривимірні матриці, елементами яких буде навантаження клітини сітки лабіринту. Наприклад, для вищенаведеного на рисунку 7.3.1 лабіринту використаємо таку матрицю

0	0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	1	0
0	1	0	1	0	1	0	1	0
0	1	1	2	0	1	1	1	0
0	1	0	0	0	0	0	1	0
0	1	1	1	0	1	1	1	0
0	0	0	1	0	0	0	1	0
0	1	1	1	1	1	0	3	0
0	0	0	0	0	0	0	0	0

Рисунок 7.3.4. Матриця, що визначає конфігурацію лабіринту.

у якій 0 буде позначати стіну, тобто зайняту клітину, 1 – вільну клітину у яку можливий перехід, 2 – початкову позицію у лабіринті з якої починається рух по лабіринту та 3 – кінцеву точку шляху.

Звернемо увагу читача, що хоча лабіринт на рисунку 7.3.1 має розмірність 7x7, ми використали для зображення лабіринту матрицю 9x9 у якій два додаткових рядки та стовпчики, що заповнені 0 використовуються для того, щоб позначити зовнішні границі лабіринту. Такий підхід не є обов'язковим, проте у подальшому дозволяє спростити написання алгоритму, оскільки зникає необхідність у перевірці умов виходу за межі масиву (тобто за межі лабіринту).

Для прикладу створимо лабіринт, що зображений рисунку 7.3.1 за допомогою матриці, як показано на рисунку 7.3.4. Для початку створимо матрицю розміром дев'ять на дев'ять заповнену нулями. Така матриця моделює лабіринт всі клітини якого є непрохідними.

Лістинг 7.3.1. Створення лабіринту з непрохідними клітинами.

```
N = 7  # рядків сітки лабіринту
M = 7  # стовпчиків сітки лабіринту

maze = []  # Створення порожньої матриці для задавання лабіринту
```

```
for i in range(N + 2):
    row = [0] * (M + 2)  # рядок матриці з M+2 елементів, що складається з 0
    maze.append(row)
```

Далі необхідно зазначити всі вільні клітини, явно вказуючи їх у програмі.

Лістинг 7.3.1. Продовження. Зазначення вільних клітин.

```
maze[1][1] = 1  # вільна клітина лабіринту
maze[1][2] = 1  # вільна клітина лабіринту
maze[1][3] = 1  # вільна клітина лабіринту
......
maze[3][3] = 2  # стартова клітина лабіринту
......
maze[7][7] = 3  # кінцева клітина лабіринту
```

Функція виведення лабіринту на екран, що неодмінно знадобиться принаймні для тестування роботи алгоритму, буде мати вигляд

Лістинг 7.3.2. Виведення лабіринту на екран.

```
def showMaze(maze):
    """ Функція форматованого виведення матриці лабіринту

    :param maze: матриця лабіринту
    :return:
    """
    for row in maze:
        for el in row:
            print("%3s" % (el,), end="")
        print()
```

Зауваження. Тут варто привернути увагу читача, до того факту, що лабіринт задається матрицею, у якій нумерація рядків йде згори до низу. Фактично, з точки зору координатного підходу це означає, що початок системи координат лабіринту знаходиться у лівому верхньому куті лабіринту, а вісь OY направлена донизу.

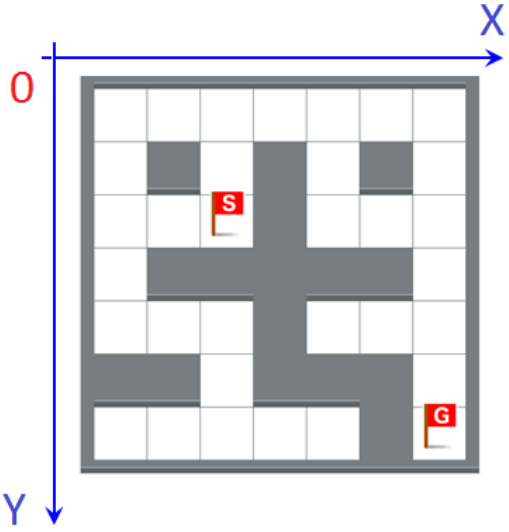


Рисунок 7.3.5. Система координат лабіринту.

Крім цього зазначимо, що оскільки лабіринт задається матрицею `maze`, доступ до елементів у якій за домовленістю здійснюється із зазначенням номера рядка та стовпчика

```
maze[1][2] # клітина лабіринту, що знаходиться у 1-му рядку та 2-му стовпчику
```

то фактично з точки зору координатного підходу, першою задається координата `y`, а вже другою `x`:

```
maze[y][x] # клітина лабіринту, що знаходиться у y-му рядку та x-му стовпчику
```

Щоб зменшити ймовірність виникнення помилок, пов'язаних з неправильним співставленням координат на площині та позицій відповідних елементів у матриці, будемо уникати у цьому параграфі способу ідентифікації клітин лабіринту, як елементів координатної сітки площини, а також позначень, що є похідними від x та y для ідентифікації елементів матриць лабіринтів.

```
maze[i][j] # клітина лабіринту, що знаходиться у i-му рядку та j-му стовпчику
```

Зчитування лабіринту з клавіатури

Очевидно, що можна використовувати інший спосіб задавання типу клітин лабіринту у матриці. Наприклад, якщо матриця лабіринту зчитується з клавіатури по рядках, то можна використати такий спосіб

Лістинг 7.3.3. Введення лабіринту з клавіатури.

```
def inputMaze(N, M):
    """ Функція зчитування лабіринту з клавіатури
    :param N: кількість рядків у лабіринті
    :param M: кількість стовчиків у лабіринті
    :return: матриця лабіринту
    """
    maze = []  # Створення порожньої матриці для задавання лабіринту

    row0 = [0] * (M + 2)      # перший рядок матриці, що визначає верхіню стіну
    maze.append(row0)

    for i in range(N):
        str_row = input()     # Зчитування рядка з клавіатури
        row = list(map(int, str_row.split()))  # Перетворення рядка у список цілих чисел
        row.insert(0, 0)      # додавання лівої "стіни" лабіринту
        row.append(0)         # додавання правої "стіни" лабіринту
        maze.append(row)      # додавання рядка до лабіринту

    rowLast = [0] * (M + 2)  # останній рядок матриці, що визначає нижню стіну
    maze.append(rowLast)

    return maze               # Повертаємо створений лабіринт
```

Тут зчитування лабіринту оформлено у вигляді підпрограми, що вхідними параметрами має кількість рядків та стовпчиків у лабіринті. Підпрограмою самостійно буде додано рядки та стовпчики, що моделюють зовнішні стіни лабіринту. Отже, для лабіринту зображеного на рисунку 7.3.1, виклик цієї функції виглядатиме так

```
maze = inputMaze(7, 7)
```

При цьому користувач має ввести для кожного рядка лабіринту послідовність числових позначень клітин лабіринту, відповідно до домовленості наведеної вище. А саме, послідовність з семи таких рядків

```
1 1 1 1 1 1 1
1 0 1 0 1 0 1
1 1 2 0 1 1 1
1 0 0 0 0 0 1
1 1 1 0 1 1 1
0 0 1 0 0 0 1
1 1 1 1 1 0 3
```

Зчитування лабіринту з текстового файлу

Якщо ж матриця лабіринту зчитується з файлу по рядках (і саме такий спосіб рекомендується використовувати для тестування ваших власних програм), то можна використати код наведений нижче. Як і вище цей код оформлений у вигляді підпрограми, яка першим параметром має ім'я текстового файлу у якому зберігається

лабіринт. Другий параметр – кількість стовпчиків у лабіринті. Кількість рядків лабіринту буде визначатися автоматично, які кількість рядків файлу.

Лістинг 7.3.4. Зчитування лабіринту з файлу.

```
def readMazeFromFile(fileName, M):
    """ Функція зчитування лабіринту з текстового файлу

    :param fileName: ім'я текстового файлу, що містить лабіринт
    :param M: кількість стовпчиків у лабіринті
    :return: матриця лабіринту
    """
    maze = []  # Створення порожньої матриці для задавання лабіринту

    row0 = [0] * (M + 2)  # перший рядок матриці, що визначає верхню стіну
    maze.append(row0)

    # Зчитування лабіринту з файлу
    with open(fileName) as f:
        for str_row in f:
            row = list(map(int, str_row.split()))  # Перетворення рядка у список цілих чисел

            if len(row) == 0:  # Захист від зайвих рядків у кінці файлу
                break

            row.insert(0, 0)   # додавання лівої "стіни" лабіринту
            row.append(0)      # додавання правої "стіни" лабіринту

            maze.append(row)   # додавання рядка до лабіринту

    rowLast = [0] * (M + 2)    # останній рядок матриці, що визначає нижню стіну
    maze.append(rowLast)

    return maze  # Повертаємо створений лабіринт
```

Виклик цієї підпрограми матиме вигляд

```
maze = readMazeFromFile("maze.txt", 7)
```

де, файл `maze.txt` міститься у тій же папці, що і файл програми та має такий вміст

```
1 1 1 1 1 1 1
1 0 1 0 1 0 1
1 1 2 0 1 1 1
1 0 0 0 0 0 1
1 1 1 0 1 1 1
0 0 1 0 0 0 1
1 1 1 1 1 0 3
```

Рух по лабіринту

Для пошуку у лабіринтах використовуються ті ж стандартні алгоритми обходу графів, такі як пошук DFS, BFS, хвильовий алгоритм, тощо. Розглянемо модифікації цих алгоритмів, для двомірних лабіринтів.

Для зручного (з точки зору програмування) пересування по лабіринту з деякої поточної клітини до сусідніх використовується два допоміжних списки

```
di = [0, -1, 0, 1]  # Зміщення по рядках
dj = [-1, 0, 1, 0]  # Зміщення по стовпчиках
```

у випадку якщо рух можливий лише по вертикалі та горизонталі, тобто у напрямках зазначених на рисунку 7.3.2, та списки

```
dj = [-1, -1, 0, 1, 1, 1, 0, -1]  # Зміщення по рядках
di = [0, -1, -1, -1, 0, 1, 1, 1]  # Зміщення по стовпчиках
```

якщо допускається рух по діагоналі (див рисунок 7.3.3). Тоді, щоб відвідати всіх сусідів клітини з координатами `(i,j)` потрібно виконати такий код

```
for k in range(len(dx)):
    process(i+di[k], j+dj[k])
```

де **`process`**`(i,j)` деякий код, що виконується над клітиною лабіринту з координатами `(i,j)`.

7.3.2. Пошук в ширину та хвильовий алгоритм

Пошук в ширину чи хвильовий алгоритм для лабіринтів майже нічим не відрізняється від відповідних стандартних алгоритмів для графів. Проте алгоритм пошуку власне найкоротшого шляху має певну особливість, яка дозволяє значно економити оперативну пам'ять комп'ютера. Отже розглянемо алгоритм пошуку в ширину на прикладі хвильового алгоритму, що дозволить нам вияснити чи досяжна деяка клітина лабіринту з заданої початкової клітини і, якщо так, то яка найкоротша до неї відстань.

Адаптація хвильового алгоритму для лабіринтів базується на використанні допоміжної матриці, що має таку ж розмірність, що і матриця лабіринту. Призначення цієї матриці полягає не лише у тому, щоб зберігати інформацію про відвідані клітини лабіринту під час обходу в ширину, а також відстань від стартової точки лабіринту до поточної. Як правило така матриця ініціалізується значенням -1 для всіх клітин. Цю допоміжну матрицю будемо називати хвильовою матрицею лабіринту.

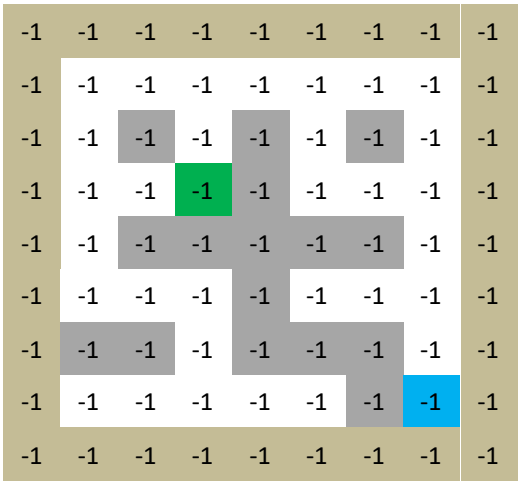


Рисунок 7.3.6. Ініціалізація хвильової матриця.

На рисунку 7.3.6 наведено ініціалізовану хвильову матрицю для лабіринту, що зображений рисунку 7.3.1 перед початком обходу в ширину. Зауважимо, що при цьому немає значення яким чином ініціалізуються клітини хвильової матриці, що відповідають зайнятим клітинам лабіринту.

Для стартової точки, з якої починається пошук у лабіринті, очевидно, у хвильовій матриці встановлюється значення 0. Далі алгоритм повністю повторює наведений вище хвильовий алгоритм, за виключенням того, що у чергу додаються не вершини графу, а координати сусідніх клітин які ще до цього не були відвідані.

Нижче наведено підпрограму, що реалізує хвильовий алгоритм. Вхідними параметрами будуть матриця лабіринту `maze` та початкова клітина пошуку `start`. Після виконання, підпрограма поверне заповнену хвильову матрицю.

Лістинг 7.3.5. Функція що здійснює обхід в ширину та заповнює хвильову матрицю.

```
di = [0, -1, 0, 1]  # Зміщення по рядках
dj = [-1, 0, 1, 0]  # Зміщення по стовпчиках

def wave(maze, start):
    """ функція побудови хвильової матриці для лібіринту зі стартовою точкою start
    :param maze: Матриця лабіринту
    :param start: Стартова точка - кортеж (r, c) - номери рядка та стовпчика відповідно
```

```
    :return: заповнена хвильова матриця
    """
    n = len(maze)     # кількість рядків у матриці maze
    m = len(maze[0])  # кількість стовпчиків у матриці maze

    # створення та ініціалізація хвильової матриці
    # такої ж розмірності, що і матриця лабіринту
    waveMatrix = []
    for i in range(n):
        row = [-1] * m
        waveMatrix.append(row)

    q = Queue()       # Створюємо чергу
    q.enqueue(start)  # Додаємо у чергу координати стартової клітини
    waveMatrix[start[0]][start[1]] = 0  # Відстань від стартової клітини до себе нуль

    while not q.empty():

        current = q.dequeue()  # Беремо перший елемент з черги
        i = current[0]         # координата поточного рядка матриці
        j = current[1]         # координата поточного стовчика матриці

        # Додаємо в чергу всі сусідні клітини
        for k in range (len(dj)):
            i1 = I + di[k]     # координата рядка сусідньої клітини
            j1 = j + dj[k]     # координата стовпчика сусідньої клітини
            # які ще не були відвідані та у які можна пересуватися
            if waveMatrix[i1][j1] == -1 and maze[i1][j1] != 0:
                q.enqueue((i1, j1))
                # Встановлюємо відстань на одиницю більшу ніж для поточної
                waveMatrix[i1][j1] = waveMatrix[i][j] + 1

    # Повертаємо хвильову матрицю, та sources-матрицю
    return waveMatrix
```

Виклик цієї підпрограми для вищезазначеного лабіринту `maze` та початкової клітини `(3,3)` буде таким

```
waveMatrix = wave(maze, (3, 3))
```

результатом виконання якої буде хвильова матриця зображена нижче

-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	4	3	2	3	4	5	6	-1
-1	3	-1	1	-1	5	-1	7	-1
-1	2	1	0	-1	6	7	8	-1
-1	3	-1	-1	-1	-1	-1	9	-1
-1	4	5	6	-1	12	11	10	-1
-1	-1	-1	7	-1	-1	-1	11	-1
-1	10	9	8	9	10	-1	12	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1

Рисунок 7.3.7. Хвильової матриця після роботи хвильового алгоритму.

Як бачимо, по цій матриці легко встановити відстань від початкової клітини `(3,3)` до будь-якої клітини лабіринту, зокрема, відстань до клітини `(7,7)` буде дорівнювати 12.

7.3.3. Відшукання шляху

Для відшукання шляху, під час виконання хвильового алгоритму, як і для графі можна було запам’ятовувати для кожної клітини, координати клітини з якої ми прийшли. Проте, ця процедура виявляється надлишковою,

оскільки маючи хвильову матрицю можна легко відновити шлях. Дійсно, для цього достатньо рухатися, починаючи з кінцевої точки у лабіринті, від сусіда до сусіда, у напрямку, у якому значення хвильової матриці є меншим на одиницю від поточного.

-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	4	3	2	3	4	5	6	-1
-1	3	-1	1	-1	5	-1	7	-1
-1	2	1	0	-1	6	7	8	-1
-1	3	-1	-1	-1	-1	-1	9	-1
-1	4	5	6	-1	12	11	10	-1
-1	-1	-1	7	-1	-1	-1	11	-1
-1	10	9	8	9	10	-1	12	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1

Рисунок 7.3.8. Відновлення шляху за хвильовою матрицею.

Наведемо дві реалізації відшукання шляху у лабіринті. У першій реалізації покажемо шлях за допомогою допоміжної матриці, у такого ж розміру як і матриця лабіринту. У цій матриці початкова точка шляху буде позначена символом "S", кінцева – символом "F", проміжні точки шляху – символом "#", а точки, що не належать шляху – ".".

Лістинг 7.3.6. Відшукання шляху.

```
di = [0, -1, 0, 1]  # Зміщення по рядках
dj = [-1, 0, 1, 0]  # Зміщення по стовпчиках

def findWay(maze, start, end):
    """ Зображує шлях за допомогою матриці

    :param maze: Матриця лібіринту
    :param start: Початкова точка шляху
    :param end: Кінцева точка шляху
    :return: Матриця у якій умовно зображено шлях
    """
    waveMatrix = wave(maze, start)  # Будуємо хвильову матрицю лабіринту

    if waveMatrix[end[0]][end[1]] == -1: # Кінцева точка не досяжна зі стартової - шляху не існує
        print("The way doesn't exist")
        return

    n = len(waveMatrix)     # кількість рядків у матриці maze
    m = len(waveMatrix[0])  # кількість стовпчиків у матриці maze

    # матриця на якій буде показано шлях
    matrix = []
    for i in range(n):
        row = [" . "] * m
        matrix.append(row)

    matrix[end[0]][end[1]] = " F "  # Позначаємо точку кінця шляху буквою F
    current = end                   # Рух починаємо з кінця
    while True:
        if current == start:
            matrix[current[0]][current[1]] = " S "  # Позначаємо точку початку шляху буквою S
            break

        i = current[0]  # координата поточного рядка матриці
        j = current[1]  # координата поточного стовчика матриці

        for k in range(len(dj)):
```

```
            i1 = i + di[k]  # координата рядка сусідньої клітини
            j1 = j + dj[k]  # координата стовпчика сусідньої клітини

            current = None
            if waveMatrix[i1][j1] == waveMatrix[i][j] - 1:
                current = (i1, j1)     # Знайдено клітину з якої ми прийшли у поточну
                matrix[i1][j1] = " # " # Позначаємо проміжну точку шляху
                break

    return matrix
```

Використаємо нашу функцію для відшукання шляху у лабіринті наведеному на рисунку рисунку 7.3.1, такого що починається у клітині з координатами (3, 3) та закінчується клітиною з координатами (7, 7)

```
wayMatrix = findWay(maze, (3, 3), (7, 7))
```

У результаті виведення на екран `wayMatrix` отримаємо схематично зображений шлях

```
 .  .  .  .  .  .  .  .  .
 .  .  .  #  #  #  .  .  .
 .  .  .  #  .  #  .  .  .
 .  .  .  S  .  #  #  #  .
 .  .  .  .  .  .  .  #  .
 .  .  .  .  .  .  .  #  .
 .  .  .  .  .  .  .  #  .
 .  .  .  .  .  .  .  F  .
 .  .  .  .  .  .  .  .  .
```

Інший спосіб побудови шляху буде повертати послідовність клітин. При цьому зауважимо, що оскільки хвильову матрицю ми будемо проходити від точки кінця шляху до її стартової точки, то для відновлення шляху скористаємося стеком у який будемо вштовхувати клітини у ході їхнього відшукання.

Лістинг 7.3.7. Відшукання шляху.

```
di = [0, -1, 0, 1]  # Зміщення по рядках
dj = [-1, 0, 1, 0]  # Зміщення по стовпчиках

def findWay(maze, start, end):
    """ Шукає шлях у лабіринті

    :param maze: Матриця лабіринту
    :param start: Початкова точка шляху
    :param end: Кінцева точка шляху
    :return: Список, клітин шляху
    """
    waveMatrix = wave(maze, start)  # Будуємо хвильову матрицю лабіринту

    if waveMatrix[end[0]][end[1]] == -1: # Кінцева точка не досяжна зі стартової - шляху не існує
        return []

    stack = Stack()  # Будуємо шлях за допомогою стеку
    current = end    # Рух починаємо з кінця
    while True:
        stack.push(current)  # Вштовхуємо у стек поточку клітину шляху
        if current == start: # Якщо поточка вершина шляху є стартовою
            break            # Усі клітини шляху містяться у стеку

        i = current[0]  # координата поточного рядка матриці
        j = current[1]  # координата поточного стовчика матриці

        for k in range (len(dj)):
            i1 = i + di[k]  # координата рядка сусідньої клітини
            j1 = j + dj[k]  # координата стовпчика сусідньої клітини
```

```
            # Шукаємо клітину з якої ми прийшли у поточну
            current = None
            if waveMatrix[i1][j1] == waveMatrix[i][j] - 1:
                current = (i1, j1)
                break

    # Відновлюємо шлях зі стеку
    way = []
    while not stack.empty():
        way.append(stack.pop())

    return way
```

Здійснимо пошук шляху з тими ж вхідними даними, що й у попередньому варіанті відшукання шляху

```
way = findWay(maze, (3, 3), (7, 7))
```

Результатом виведення на екран змінної way буде такий список

```
[(3, 3), (2, 3), (1, 3), (1, 4), (1, 5), (2, 5), (3, 5), (3, 6), (3, 7), (4, 7), (5, 7), (6, 7),
(7, 7)]
```

що відповідає послідовності клітин знайденого шляху.

§7.4. Алгоритми на зважених графах

У випадку, якщо граф зважений, то для пошуку найкоротшого шляху використовувати звичайний алгоритм пошуку в ширину вже не підійде. Дійсно, в зважених графах вага одного з ребер може бути значно більшою за довжину шляху, що складається з цілого переліку ребер. Наприклад, у графі на рисунку 7.4.1, найкоротший шлях з вершини 1 у вершину 6 пройде через вершини 2, 3, 5 та 4:

$$1 \to 2 \to 3 \to 5 \to 4 \to 6$$

Його довжина буде дорівнювати сумі ваг відповідних ребер, тобто 8.

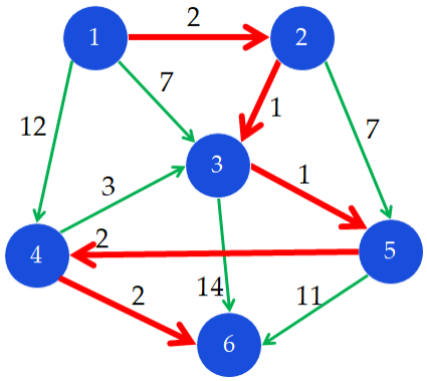


Рисунок 7.4.1. Найкоротший шлях у зваженому графі.

Як ми бачимо цей шлях пройде через 5 ребер, у той час, як класичний алгоритм пошуку в ширину найкоротший шлях з вершини 1 у вершину 6 провів би через два ребра:

$$1 \to 3 \to 6$$

або

$$1 \to 4 \to 6.$$

Перший має вагу 21, другий – 14.

Існує низка алгоритмів, серед яких алгоритми Беллмана—Форда, алгоритм Дейкстри, A* алгоритм, алгоритм Флойда – Уоршелла, що дозволяють відшукати найкоротший шлях у зважених графах. Кожен з алгоритмів має свою специфіку та область застосування. Тому поговоримо про кожен з них окремо.

7.4.1. Алгоритм Беллмана – Форда

Алгоритм Беллмана – Форда це ітеративний алгоритм, що не лише дозволяє знайти найкоротший шлях від однієї вершини до іншої у зваженому графі, але й знайти найкоротші шляхи від однієї вершини зваженого графа до усіх інших його вершин. Цей алгоритм є одним з найпростіших з алгоритмічної точки зору, проте й одним з найповільніших. Його перевагою над іншими алгоритмами є те, що він може відшукати найкоротший шлях у графі, що містить ребра від'ємної ваги, якщо граф не має циклів від'ємної ваги. Крім цього, алгоритм Беллмана – Форда дозволяє досить просто визначити чи містить граф цикл від'ємної ваги.

Опис алгоритму

Отже, пояснимо алгоритм на прикладі знаходження найкоротшого шляху з вершини 1 у вершину 6 для графа зображеного на рисунку 7.4.1. Для кожної вершини встановлюється додаткове навантаження – величина найкоротшого шляху від стартової вершини до поточної вершини графа. До початку роботи алгоритму будемо вважати, що вона є нескінченністю (рисунок 7.4.2).

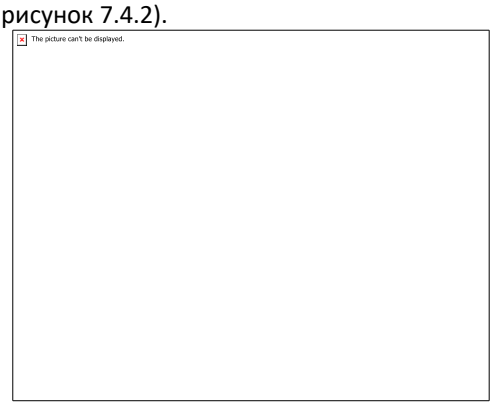


Рисунок 7.4.2. Ініціалізація додаткових навантажень вершин графа.

Далі, встановлюємо відстань від стартової вершини до себе нулем.

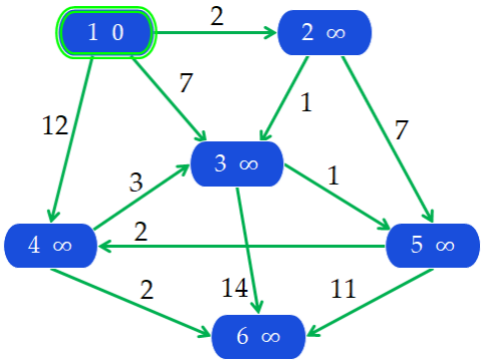


Рисунок 7.4.3. Ініціалізація навантаження стартової вершини графа.

На цьому етапі підготовка завершується, а ми припускаємо, що відстані (по деякому, не оптимальному, шляху) від стартової вершини до всіх вершин графа відомі. Виберемо деяку вершину v. Проаналізуємо як зміниться навантаження її сусідів, якби ми прийшли у них прямуючи з вершини v. Відповідно до припущення ми знаємо відстань від стартової вершини до кожного сусіда вершини v по деякому шляху. Якщо цей шлях є гіршим, ніж шлях через вершину v, то змінюємо навантаження сусіда таким чином, що ми прийшли у нього з вершини v. Описаний процес називається релаксацією вздовж ребра, що сполучає вершину v з її сусідом.

Отже, виберемо вершину 1. У кожному з її сусідів – вершин 2, 3, та 4 відстані від стартової вершини є нескінченність. Якщо ми прийдемо у вершину 2 з вершини 1, то відстань від стартової вершини буде 2 – сума навантаження у вершині 1 та вага ребра $(1,2)$, у вершині 3 – відповідно 7, а у вершині 4 – 12.

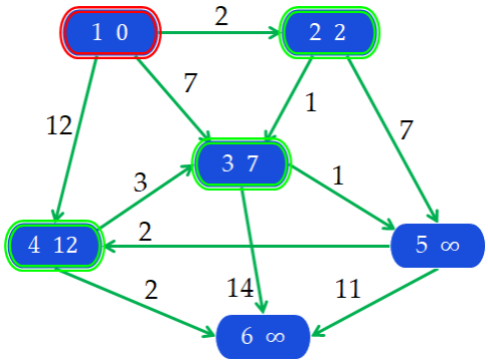


Рисунок 7.4.4. Оновлення навантажень сусідів вершини 1.

Здійснюємо описану процедуру для наступної вершини графа, вершини 2:

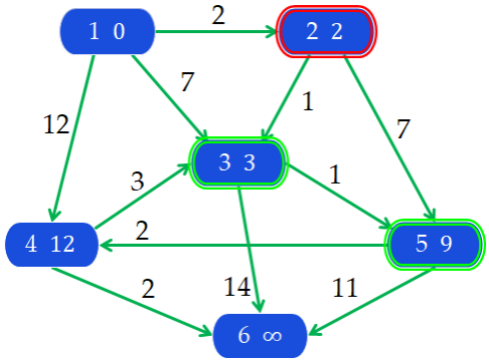


Рисунок 7.4.5. Оновлення навантажень сусідів вершини 2.

Як бачимо, шлях у вершину 3 через вершину 2 буде мати довжину 3, що є краще ніж попереднє значення довжини шляху, отримане при досягненні вершини 3 з вузла 1. Продовжуємо процедуру послідовно для вершин 3, 4, 5 та 6.

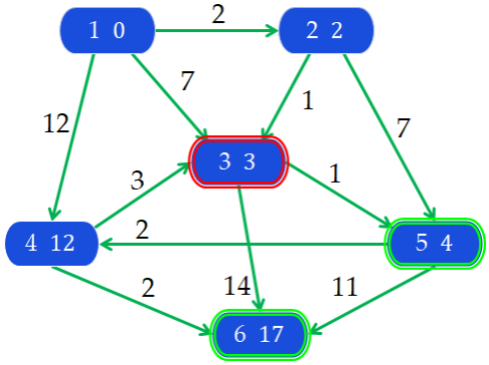


Рисунок 7.4.6. Оновлення навантажень сусідів вершини 3.

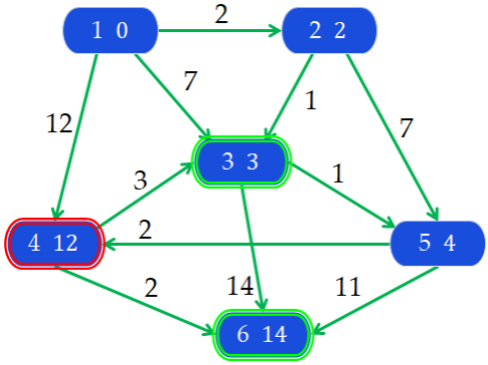


Рисунок 7.4.7. Оновлення навантажень сусідів вершини 4.

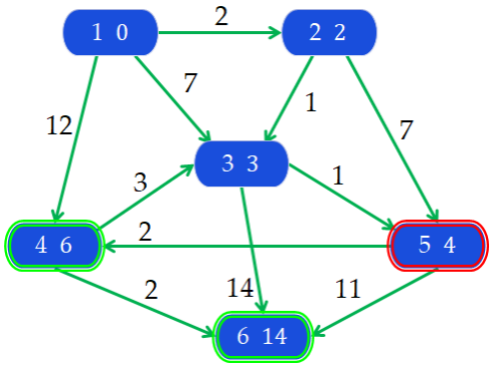


Рисунок 7.4.8. Оновлення навантажень сусідів вершини 5.

З вершини 6 не виходить жодного ребра, тому оновлення навантажень сусідів вершини 6 зображувати не має сенсу. На цьому завершилася перша ітерація алгоритму Беллмана – Форда. Як бачимо, ми не досягли необхідного результату – у вершині 6, очевидно не довжина найкоротшого шляху. І це не дивно, адже відбулася лише перша ітерація алгоритму. Щоб знайти найкоротші шляхи від стартової вершини до всіх вершин графа необхідно здійснити $|V| - 1$ ітерацій алгоритму. Дійсно, найкоротший шлях не може містити більше ніж $|V| - 1$ ребер, оскільки у такому разі він буде містити цикл, який гарантовано можна виключити.

Отже, після 5 ітерацій алгоритму Беллмана – Форда додаткові навантаження, що показують найкоротшу відстань від стартової вершини до вершин графа будуть такими, як зображено на рисунку нижче

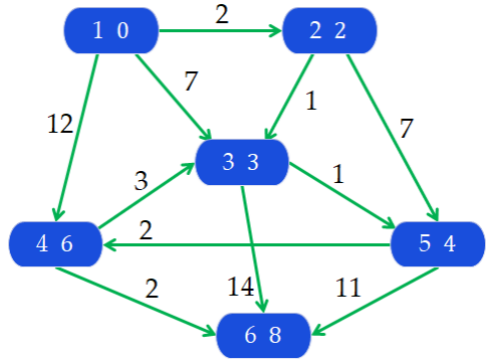


Рисунок 7.4.9. Навантаження графа після завершення роботи алгоритму.

Для відшукання шляху, як і у випадку незважених графів, для кожної вершини будемо запам’ятовувати вершину з якої ми прийшли у поточну.

Реалізація

Для реалізації алгоритму, розширимо клас `Vertex`, який було описано у лістингу 7.1.4, додавши поля для зберігання інформації про довжину найкоротшого шляху від деякої стартової вершини до поточної, а також вершини, з якої ми прийшли у поточну по цьому найкоротшому шляху.

Лістинг 7.4.1. Клас VertexForAlgorithms – навантажений `Vertex`.

```
INF = sys.maxsize  # Умовна нескінченність

class VertexForAlgorithms(Vertex):
    """ Клас, що є розширенням класу для вершини графа

        Використовується у алгоритмах Белмана-Форда, Дейкстри, A*...
        Міститть додаткову технічну інформацію, що необхідна для реалізації щих алгоритмів
     """

    def __init__(self, key):
        """ Конструктор створення вершини

        :param key: Ключ вершини
        """
        super().__init__(key)

        """ Відтань - додаткове навантаження на вершину - величина найкоротшого шляху
```

```
            від деякої фіксованої вершини до поточної вершини графа.
            Використовується для алгоритмів Дейкстри, Белмана-Форда та ін. """
        self.mDistance = INF  # До початку роботи алгоритму вважаємо, що вона є нескінченністю!

        """ Джерело - додаткове навантаження на вершину -
            вершина з якої прийшли по найкорошому шляху
            у поточну вершину на деякому кроці алгоритму """
        self.mSource = None   # До початку роботи алгортиму вважаємо, що вона є невизначеною

    def setDistance(self, distance):
        """ Встановлює відстань для поточної вершини

        :param distance: Нова відстань
        """
        self.mDistance = distance

    def distance(self):
        """ Повератє поточну відстань у вершині

        :return: Відстань у вершині
        """
        return self.mDistance

    def setSource(self, source):
        """ Встанолює джерело для поточної вершини

        :param source: Нове джерело вершини
        :return: None
        """
        self.mSource = source

    def source(self):
        """ Повертає джерело для поточної вершини

        :return: Джерело поточної вершини
        """
        return self.mSource
```

Тепер розширимо клас Graph, вершинами якого будуть екземпляри класу VertexForAlgorithms.

Лістинг 7.4.2. Клас GraphForAlgorithms.

```
class GraphForAlgorithms(Graph):
    """Клас, що є розширенням класу для графа Graph
     Використовується у алгоритмах Дейкстри, Белмана-Форда...
     Міститть додаткову технічну інформацію, що необхідна для цих алгоритмів """

    def addVertex(self, vertex) -> bool:
        """ Додає вершину у граф, якщо така вершина не міститься у ньому

        :param vertex: ключ (тобто ім'я) нової вершини
        :return: True, якщо вершина успішно додана
        """

        if vertex in self:  # Якщо вершина міститься у графі, її вже не треба додавати
            return False

        new_vertex = VertexForAlgorithms(vertex)  # створюємо нову вершину з іменем key
        self.mVertices[vertex] = new_vertex       # додаємо цю вершину до списку вершин графу
        self.mVertexNumber += 1                   # Збільшуємо лічильник вершин у графі
        return True

    def constructWay(self, start, end):
        """ Домопіжний метод, що будує шлях, між двома вершинами у графі
        Може бути застосовами лише після дії алгоритмів пошуку шляху
        (Хвильового, Дейкстри, Беллмана-Форда, тощо) які записують
        допоміжну інформацію у вершини графа.

        :param start: Вершина, що початком шляху
```

```
        :param end: Вершина, що є кінцем шляху
        :return: Кортеж, що містить список вершин - найкоротший шлях,
                 що сполучає вершини start та end та його вагу """

        if self[end].source is None:  # шляху не існує
            return None, INF

        # будуємо шлях за допомогою стеку
        stack = Stack()
        current = end
        while True:
            stack.push(current)
            if current == start:
                break
            current = self[current].source()
            if current is None:
                return None

        way = []  # Послідовність вершин шляху
        while not stack.empty():
            way.append(stack.pop())

        # Повертаємо шлях та його вагу
        return way, self[end].distance()
```

Звернемо увагу читача, що клас GraphForAlgorithms містить метод **constructWay**(), призначення якого відновити шлях у графі після відпрацювання алгоритму Беллмана – Форда. Наведемо тепер реалізацію алгоритму для випадку, якщо граф гарантовано не містить циклів від'ємної довжини.

Лістинг 7.4.3. Алгоритм Беллмана – Форда для графів без циклів від'ємної довжини.

```
def BelmanFord(graph: GraphForAlgorithms, start, end):
    """ Реалізує класичний алгоритм Беллмана-Форда для графів, що немають циклів від'ємної ваги
    :param graph: Граф
    :param start: Стартова вершина
    :param end: Кінцева вершина
    :return: Кортеж, що містить список вершин - найкоротший шлях,
             що сполучає вершини start та end та його вагу """

    # Ініціалізуємо додаткову інформацію у графі для роботи алгоритму
    for vertex in graph:
        vertex.setDistance(INF)  # Відстань для кожної вершини від стартової
                                 # ставиться як нескінченність
        vertex.setSource(None)   # Вершина з якої прийшли по найкорошому шляху невизначена

    # Відстань від першої вершини до неї ж визначається як 0
    graph[start].setDistance(0)

    for i in range(len(graph) - 1):
        for vertex in graph:                          # Для всіх вершин графу
            for neighbor_key in vertex.neighbors():   # Для всіх сусідів поточної вершини
                neighbor = graph[neighbor_key]        # Беремо вершину-сусіда за індексом

                # Обчислюємо потенційну відстань у вершині-сусіді
                newDist = vertex.distance() + vertex.weight(neighbor_key)

                # Якщо потенційна відстань у вершині-сусіді менша за її поточне значення
                if newDist < neighbor.distance():
                    # Змінюємо поточне значення відстані у вершині-сусіді обчисленим
                    neighbor.setDistance(newDist)
                    # Встановлюємо для сусідньої вершини ідентифікатор звідки ми прийшли у неї
                    neighbor.setSource(vertex.key())

    return graph.constructWay(start, end)
```

Аналіз алгоритму

Як було зазначено вище, зовнішній цикл алгоритму наведеного у лістингу 7.4.3, у виконується $|V|-1$ разів. Далі, необхідно пройти по всіх ребрах графа, відтак внутрішній цикл буде мати асимптотичну складність $O(|E|)$, у випадку, якщо граф заданий списком суміжності та $O(|V|^2)$ – якщо граф заданий матрицею суміжності. Крім цього, початкова ініціалізація додаткових навантажень у вершинах графа, що необхідна для роботи алгоритму, вимагає додатково $O(|V|)$ операцій.

Таким чином, підсумовуючи, можемо стверджувати, що складність алгоритму Беллмана – Форда буде:

- $O(|V|\cdot|E|)$, якщо граф заданий списком суміжності;
- $O(|V|^3)$, якщо граф заданий матрицею суміжності.

Додаткові властивості та оптимізації алгоритму

Алгоритм Беллмана – Форда, дозволяє дуже легко визначити, чи має граф цикл від'ємної ваги, що досяжний зі стартової вершини. Дійсно, досить провести зовнішню ітерацію циклу алгоритму не $|V|-1$, а рівно $|V|$ рази. При цьому, якщо при виконанні останньої ітерації довжина найкоротшого шляху до однієї з вершини зменшиться, то граф містить цикл від'ємної ваги, що досяжний зі стартової вершини.

Лістинг 7.4.4. Пошук циклів від'ємної ваги за допомогою алгоритму Беллмана – Форда.

```
def BelmanFordClasical(graph, start):
    """ Класичний алгоритм Беллмана-Форда для графів, що можуть мати цикли від'ємної ваги
        Перевіряє чи має граф цикли від'ємної ваги
    :param graph: Граф
    :param start: Стартова вершина
    :return: False, якщо граф не містить циклів від'ємної ваги
    """

    # Ініціалізуємо додаткову інформацію у графі для роботи алгоритму
    for vertex in graph:
        vertex.setDistance(INF)  # Відстань для кожної вершини від стартової – нескінченність

    # Відстань від першої вершини до неї ж визначається як 0
    graph[start].setDistance(0)
    for i in range(len(graph) - 1):
        for vertex in graph:                                # Для всіх вершин графу
            for neighbor_key in vertex.neighbors():         # Для всіх сусідів поточної вершини
                neighbour = graph[neighbor_key]             # Беремо вершину-сусіда за індексом
                # Обчислюємо потенційну відстань у вершині-сусіді
                newDist = vertex.distance() + vertex.weight(neighbor_key)
                # Якщо потенційна відстань у вершині-сусіді менша за її поточне значення
                if newDist < neighbour.distance():
                    # Змінюємо поточне значення відстані у вершині-сусіді обчисленим
                    neighbour.setDistance(newDist)

    # Перевірка на наявність циклів з від'ємною вагою
    # Тут фактично здійснюється ще одна ітерація циклу і якщо у одній з вершин
    # знайдена відстань зменшиться, то це і означатиме, що знайдено цикл від'ємної ваги.
    for vertex in graph:                                # Для всіх вершин графу
        for neighbor_key in vertex.neighbors():         # Для всіх сусідів поточної вершини
            neighbour = graph[neighbor_key]             # Беремо вершину-сусіда за індексом
            # Обчислюємо потенційну відстань у вершині-сусіді
            newDist = vertex.distance() + vertex.weight(neighbor_key)
            # Якщо потенційна відстань у вершині-сусіді менша за її поточне значення
            if newDist < neighbour.distance():
                return True                             # Знайдено цикл від'ємної ваги.

    return False  # Граф не місіить циклів від'ємної ваги.
```

Алгоритм пошуку циклу від'ємної ваги, наштовхує на ідею оптимізації алгоритму Беллмана – Форда. Можна відстежувати зміни додаткових навантажень у графі (що визначають відстані найкоротших шляхів від стартової вершини). Щойно зміни припиняться, можна припиняти роботу зовнішнього циклу – подальші ітерації вже не потрібні – найкоротші шляхи вже знайдено.

Лістинг 7.4.5. Оптимізація алгоритму Беллмана – Форда.

```
def BelmanFordOptimized(graph: GraphForAlgorithms, start: int, end: int):
    """ Оптимізований алгоритм Беллмана-Форда для графів, що немають циклів від'ємної ваги
        Тут здійснюється перевірка, що якщо на певному кроці роботи алгоритму не відбулося
        жодних змін у дистанціях, в вершинах знайдених на попередньому кроці,
        то алгоритм фактично закінчив свою роботу

    :param graph: Граф
    :param start: Стартова вершина
    :param end: Кінцева вершина
    :return: Кортеж, що містить список вершин - найкоротший шлях,
             що сполучає вершини start та end та його вагу
    """

    # Ініціалізуємо додаткову інформацію у графі для роботи алгоритму
    for vertex in graph:
        vertex.setDistance(INF)  # Відстань для кожної вершини від стартової - нескінченність
        vertex.setSource(None)   # Вершина з якої прийшли по найкорошому шляху невизначена

    # Відстань від першої вершини до неї ж визначається як 0
    graph[start].setDistance(0)

    for i in range(len(graph) - 1):
        # Мітка, що алгоритм ще не закінчив роботу і необхідна принаймні ще одна його ітерація
        isRelaxed = True
        for vertex in graph:                            # Для всіх вершин графу
            for neighbor_key in vertex.neighbors():     # Для всіх сусідів поточної вершини
                neighbour = graph[neighbor_key]         # Беремо вершину-сусіда за ключем
                # Обчислюємо потенційну відстань у вершині-сусіді
                newDist = vertex.distance() + vertex.weight(neighbor_key)
                # Якщо потенційна відстань у вершині-сусіді менша за її поточне значення
                if newDist < neighbour.distance():
                    # Змінюємо поточне значення відстані у вершині-сусіді обчисленим
                    neighbour.setDistance(newDist)
                    # Встановлюємо для сусідньої вершини ідентифікатор звідки ми прийшли у неї
                    neighbour.setSource(vertex.key())
                    # Встановлюємо ознаку, що алгоритм потребує ще принаймні одного проходу
                    isRelaxed = False

        if isRelaxed:  # Якщо на поточному кроці роботи алгоритму у знайдених раніше дистанціях у
                       # графі не відбулося жодних змін
            break      # припиняємо ітерації алгоритму, оскільки всі відстані знайдено

    return graph.constructWay(start, end)
```

7.4.2. Алгоритм Дейкстри

Це одним з алгоритмів, які дозволяють розв’язати задачу пошуку найкоротшого шляху у зваженому графі, є алгоритм Дейкстри. Він названий на честь нідерландського математика Едсгера Дейкстри (нід. Edsger Wybe Dijkstra), який його власне і відкрив. Як і алгоритм Беллмана – Форда, алгоритм Дейкстри дозволяє знайти найкоротший шлях від однієї фіксованої вершини графа до всіх інших його вершин, проте є значно оптимальнішим. Суттєвим обмеженням цього алгоритму у порівнянні з алгоритмом Беллмана – Форда є те, що він працює лише для графів, не містять ребер від’ємної ваги.

Опис алгоритму

Знову розглянемо граф, зображений на рисунку 7.4.1. Як при вивченні алгоритму Беллмана – Форда, пояснимо алгоритм Дейкстри на прикладі знаходження найкоротшого шляху з вершини 1 у вершину 6. Подібно до алгоритму Беллмана – Форда, задіємо додаткову інформацію для всіх вершин графа, а саме навантаження, що визначає величину найкоротшого шляху від стартової вершини до поточної вершини графа. Як і вище, вважається, що відстань від стартової точки до себе є нульовою, а відстані до всіх інших точок графу є нескінченними. На рисунку 7.4.10 – поруч з ідентифікатором вершини зазначається навантаження вершин до початку роботи алгоритму).

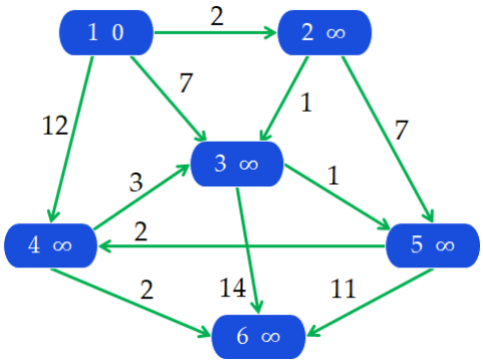


Рисунок 7.4.10. Ініціалізація навантаження вершин графа у алгоритмі Дейкстри.

На цьому етапі підготовка до роботи алгоритму завершується.

Кожен крок алгоритму починається з відшукання вершини з найменшим навантаженням. У нашому випадку, це вершина 1, що має навантаження 0. Ця вершина фіксується – найкоротша відстань до неї вже відома – і у подальшому виключається з розгляду, оскільки і її навантаження уже в подальшому не буде змінюватися.

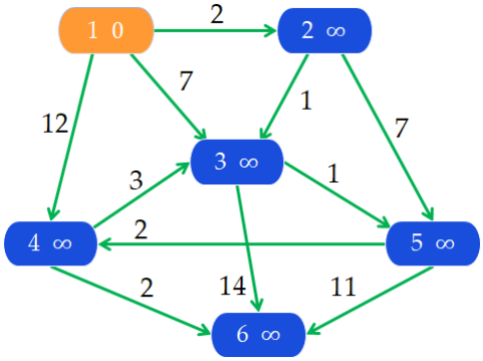


Рисунок 7.4.11. Вершина 1 – фіксована.

Далі відбувається пошук із вершини 1. Подібно до того, як це відбувалося в алгоритмі Беллмана – Форда, ми аналізуємо навантаження у сусідах. У нашому випадку ми можемо з вершини 1 потрапити у вершини 2, 3 та 4, при цьому поточне навантаження всіх трьох вершин є нескінченним.

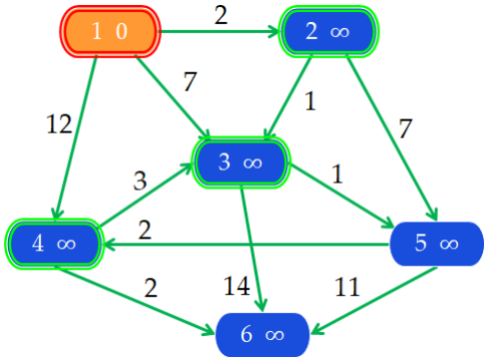


Рисунок 7.4.12. Пошук з вершини 1.

Розглянемо вершину 2. Якщо вважати, що у неї ми прийшли з вершини 1, то це можна зробити зі стартової вершини подолавши відстань 2 – сума навантаження у вершині 1 та ребра, що сполучає вершини 1 та 2. Це краще ніж поточне значення вершини, тому змінюємо навантаження вершини 2 значенням 2. Аналогічно навантаження у вершині 3 зміниться на 7, а у вершині 4 – значенням 12.

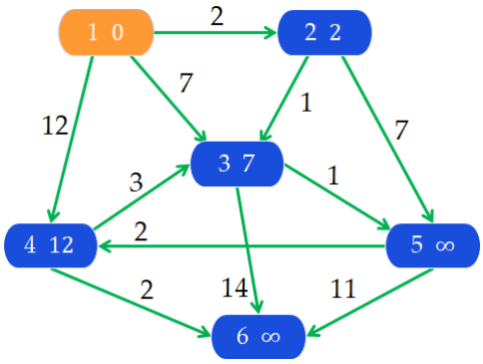


Рисунок 7.4.13. Зміна навантажень у сусідах вершини 1.

Серед усіх не фіксованих вершин, шукаємо вершину з найменшим навантаженням. У нашому випадку це вершина 2 – її навантаження дорівнює 2. Здійснюємо пошук з цієї вершини по усіх сусідах. Вона має два сусіди – вершину 3 та вершину 5.

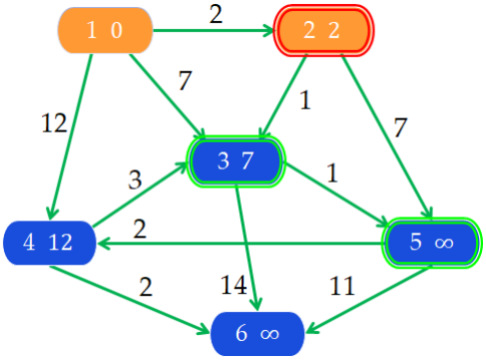


Рисунок 7.4.14. Опрацювання вершини 2.

Якби ми прийшли у вершину 3 з вершини 2, то шлях у вершину 3 дорівнював би 3 – сума навантаження у вершинні 2 та вага ребра, що сполучає вершини 2 та 3. Це значення краще ніж 7 – поточне навантаження вершини 3, яке було отримано якби у вершину 3 ми прийшли безпосередньо з вершини 1. Тому заміняємо навантаження вершини 3 кращим. Аналогічно, навантаження у вершині 5 стає 9.

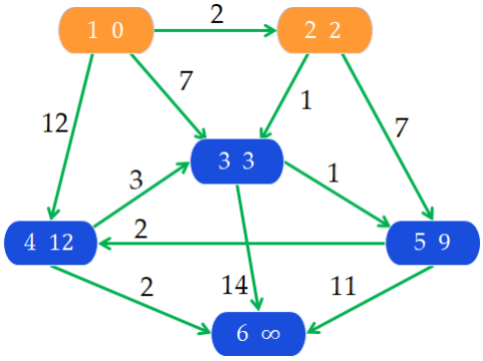


Рисунок 7.4.15. Опрацювання сусідів вершини 2.

Наступна ітерація алгоритму приводить нас до розгляду вершини 3 та її сусідів – вершин 5 та 6.

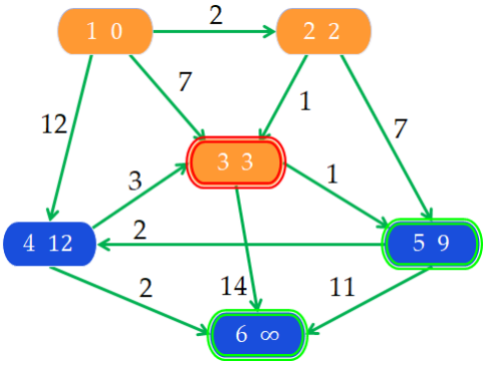


Рисунок 7.4.16. Опрацювання вершини 3.

Очевидно, що для обох її сусідів кращим є якби перехід у них з вершини 3

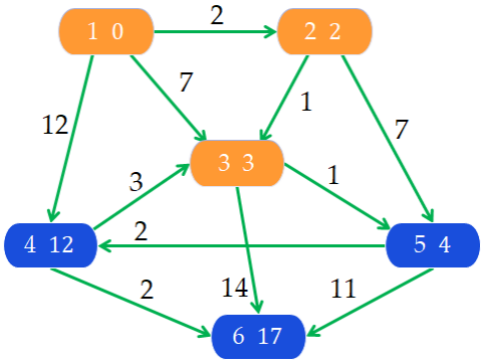


Рисунок 7.4.17. Оновлення навантажень сусідів вершини 3.

Наступною вершиною з найменшим навантаженням буде вершина 5.

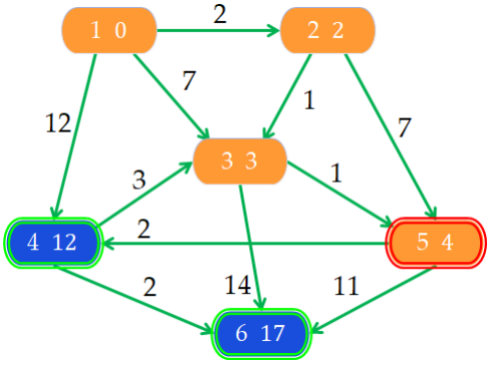


Рисунок 7.4.18. Опрацювання вершини 5.

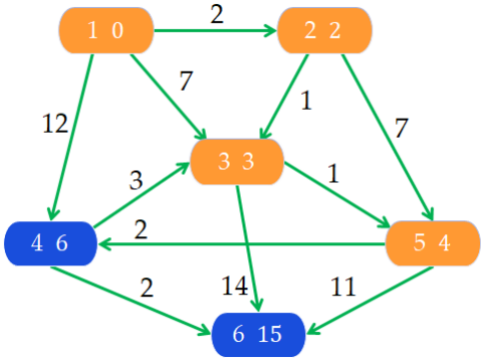


Рисунок 7.4.19. Оновлення навантажень сусідів вершини 5.

Серед вершин, що лишилися нефіксованими у графі – вершин 4 та 6 – вибираємо вершину 4, оскільки її навантаження 6 менше за 15 – навантаження вершини 6.

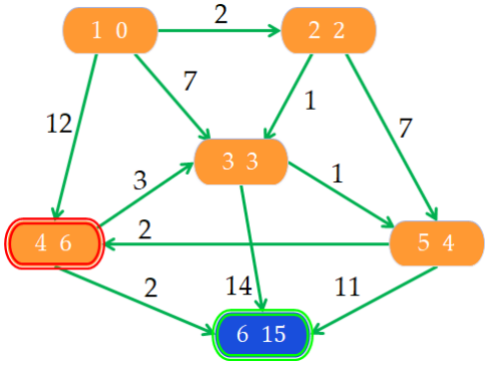


Рисунок 7.4.20. Опрацювання вершини 4.

Опрацювання вершини 4 фактично завершує роботу алгоритму

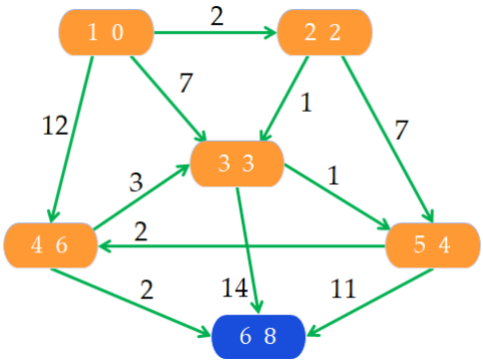


Рисунок 7.4.21. Оновлення навантажень сусідів вершини 4.

оскільки на останньому кроці лишається зафіксувати останню вершину 6.

Як і у випадку використання Алгоритму Беллмана – Форда, ми знайшли не лише довжину найкоротшого шляху від вершини 1 до вершини 6, але й найкоротші відстані від вершини 1 до всіх вершин графу. Наприклад, найкоротша відстань до вершини 5 є 4. При цьому, якщо, запам'ятовувати вершину з якої прийшли у поточну, то легко відтворити й сам шлях.

Реалізація

Алгоритм Дейкстри використовує структуру даних пріоритетну чергу для визначення вершини, що має найменше навантаження на поточному кроці алгоритму.

Лістинг 7.4.6. Реалізація алгоритму Дейкстри.

```
def Dijkstra(graph, start, end):
    """ Реалізує алгоритм Дейкстри

    :param graph: Граф
    :param start: Стартова вершина
    :param end: Кінцева вершина
    :return: Кортеж, що містить список вершин - найкоротший шлях,
             що сполучає вершини start та end та його вагу
    """

    # Ініціалізуємо додаткову інформацію у графі для роботи алгоритму.
    for vertex in graph:
        vertex.setDistance(INF)  # Відстань для кожної вершини від стартової - нескінченність
        vertex.setSource(None)   # Вершина з якої прийшли по найкорошому шляху невизначена

    # Відстань у старотовій вершині (тобто від стартової вершини до себе) визначається як 0
    graph[start].setDistance(0)

    pq = PriorityQueue()              # Створюємо пріоритетну чергу
    pq.insert(start, 0)               # Додаємо у чергу початкову вершину з нульовим пріоритетом

    # Введемо масив, що буде містити ознаку чи фіксована вже вершина.
    # Ініціалізуємо масив значеннями False  (що означає, що вершина не фіксована)
    fixed = [False] * len(graph)

    while not pq.empty():
        vertex_key = pq.extractMinimum()   # Беремо індекс вершини з черги з найвищим пріоритетом
        fixed[vertex_key] = True           # та позначаємо її як фіксавану
        vertex = graph[vertex_key]         # Беремо вершину за індексом

        if vertex_key == end:              # Якщо поточний елемент є шуканим
            break                          # пошук завершено

        for neighbor_key in vertex.neighbors():    # Для всіх сусідів поточної вершини
            if fixed[neighbor_key]:                # які ще не були фіксовані
                continue

            neighbour = graph[neighbor_key]                # Беремо вершину-сусіда за ключем
            # Обчислюємо потенційну відстань у вершині-сусіді
```

```
            newDist = vertex.distance() + vertex.weight(neighbor_key)
            # Якщо потенційна відстань у вершині-сусіді менша за її поточне значення
            if newDist < neighbour.distance():
                # Змінюємо поточне значення відстані у вершині-сусіді обчисленим
                neighbour.setDistance(newDist)
                # Встановлюємо для сусідньої вершини ідентифікатор звідки ми прийшли у неї
                neighbour.setSource(vertex_key)

                if neighbor_key in pq:   # Якщо вершина сусід міститься у черзі
                    # перераховуємо її пріоритет в черзі
                    pq.updatePriority(neighbor_key, newDist)
                else:
                    # або додаємо елемент до черги, якщо його там ще немає.
                    pq.insert(neighbor_key, newDist)

    return graph.constructWay(start, end)    # Повертаємо шлях та його вагу
```

У наведеному лістингу програми передбачається використання графу описаного у лістингу 7.4.2.

Зауважимо, що якщо вага усіх ребер графу однакова, то алгоритм Дейкстри вироджується у звичайний алгоритм пошуку в ширину на незваженому графі.

7.4.3. A* алгоритм

Алгоритм A* подібний до алгоритму Дейкстри і фактично є його удосконаленням. У ньому також для визначення порядку обходу вершин використовується черга з пріоритетом. Проте, на відміну від алгоритму Дейкстри, у якому пріоритет кожної неопрацьованої вершини враховує лише поточну довжину шляху від стартової точки, пріоритет вершин у алгоритмі A* використовує додатково допоміжну функцію (евристику), аби скеровувати напрям пошуку. Таким чином, пріоритет буде визначатися за допомогою функції

$$f(x) = g(x) + h(x)$$

де $g(x)$ – функція, значення якої дорівнюють вартості шляху від початкової вершини до x, (як у алгоритмі Дейкстри), $h(x)$ – евристична функція, яка оцінює вартість шляху від вершини x до кінцевої.

Пояснимо це на прикладі пошуку у лабіринті зображеному на рисунку нижче, де клітини позначені літерами S та F позначають відповідно стартову та кінцеву клітини пошуку.

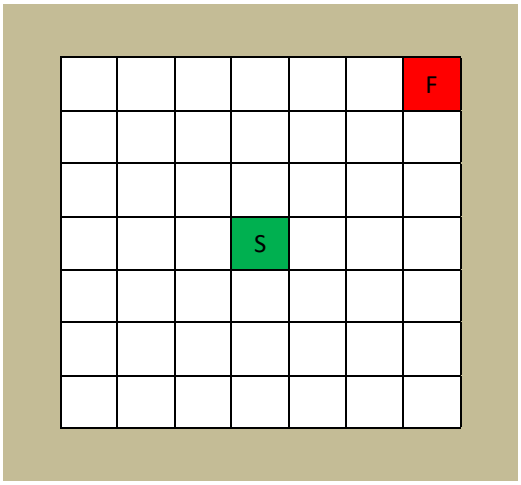


Рисунок 7.4.22. Відновлення шляху за хвильовою матрицею.

Будемо вважати, що рух по лабіринту допускається лише у напрямах зображених на рисунку 7.3.2. та вартість переходу з клітини у сусідню є однаковою для усіх клітин лабіринту. У такому разі, як було зазначено вище, застосування алгоритму Дейкстри вироджується у алгоритм пошуку в ширину. Побудуємо хвильову матрицю для цього лабіринту

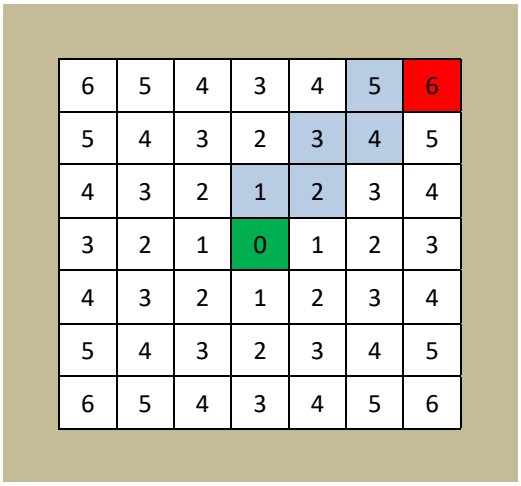


Рисунок 7.4.23. Відновлення шляху за хвильовою матрицею.

Найкоротший шлях було знайдено. Проте, як бачимо, для цього нам знадобилося обійти усі клітини лабіринту, оскільки рух по лабіринту здійснювався у всіх напрямках рівномірно починаючи від стартової вершини. По великому рахунку, рух вниз і вліво не має сенсу, оскільки не він навпаки віддаляє нас від фінішної клітини і мало ймовірно, що рухаючись таким чином ми досягнемо фінішу. Більше того, у нашому випадку ми знаємо, щоб дістатися до фінішу найкоротшим шляхом, у пріоритеті треба зробити рух вгору і вправо. Саме для цього і використовується евристична функція у алгоритмі A*. Наприклад, у нашому прикладі, евристичною функцією може бути відстань (лінійна відстань на площині) від поточної точки до фінішної клітини. Це зробить рух у клітини що розташовуються справа і згори від поточної пріоритетнішим, ніж рух у клітини, що розташовуються зліва та знизу. Відтак, фінішна клітина буде досягнута значно раніше, ніж будуть пройдені всі клітини лабіринту (як у хвильовому алгоритмі), що суттєво пришвидшить відшукання найкоротшого шляху.

Зауважимо, що такий спосіб визначення евристичної функції далеко не єдиний і залежить від поставленої задачі. Важливим є той факт, що значення евристичної функції для всіх вершин графа обраховуються один раз на початку роботи алгоритму, та ніяк не впливають на швидкодію алгоритму. Проте, очевидно, що вибір евристичної функції може бути вирішальним фактором для оптимізації алгоритму A*.

Реалізація

Наведемо реалізацію алгоритму A* на прикладі пошуку найкоротшого шляху на плоскому графі. Для цього нам знадобляться додаткові класи. Спочатку опишемо клас `VertexWithHeuristic` що є розширенням класу `VertexForAlgorithms` описаному у лістингу 7.4.1. У цьому класі будемо вважати, що поле `mData` вершини містить координати вершини на площині.

Лістинг 7.4.7. Вершина графа з евристикою.

```
class VertexWithHeuristic(VertexForAlgorithms):
    """ Клас, що є розширенням графа VertexForAlgorithms
     Використовується у алгоритмі A*.
     Міститть значення евристичної функції (евристику), що необхідна для реалізації алгоритму A*.
     """

    def __init__(self, key):
        """ Конструктор вершини

        :param key: Ключ вершини
        """
        super().__init__(key)
        self.mHeuristicValue = 0   # Значення еврестичної функції (евристика)

    def heuristic(self):
        """ Значення евристики у поточній вершині
        :return: евристику у поточній вершині
        """
```

```
        return self.mHeuristicValue

    def calculateHeuristic(self, other):
        """ Допоміжний метод, що встанолює значення евристичної функції у вершині.
        Вибирається спеціальним чином з аналізу властивостей графу
        (наприклад в залежності від топології графу).
        Тут евристика визначається як геометрична відстань (по прямій) між двома вершинами графа
        :param other: інша вершина графа
        """
        position = self.mData

        if isinstance(other, Vertex):
            dest_position = other.data()
        else:
            dest_position = other

        assert position is not None and dest_position is not None

        self.mHeuristicValue = (((dest_position[0] - position[0]) ** 2) +
                                        ((dest_position[1] - position[1]) ** 2)) ** 0.5
```

Як бачимо, цей клас місить нове поле – значення евристичної функції яка обчислюється, як лінійна відстань між поточною вершиною та деякою іншою вершиною графа. Очевидно, що такою іншою вершиною варто вибрати фінішну вершину графа до якої має знайти найкоротший шлях алгоритм A*.

Тепер опишемо клас `PlainGraph`, вершини якого є екземплярами класу `VertexWithHeuristic`, а вага ребер між вершинами – лінійна відстань між ними.

Лістинг 7.4.8. Граф на площині.

```
class PlainGraph(GraphForAlgorithms):
    """ Граф, вершини якого містять геометричні координати,
        а вага ребер - відстані між ними
        Використовується для тестування алгоритму A*
        """


    def addEdge(self, source, destination, weight=1):
        """ Додавання ребра з кінцями в точках source та destination
        з вагою, що дорівнює геометричній відстані між цими вершинами
        :param source: Перша вершина
        :param destination: Друга вершина
        :param weight: Вага ребра - формальний параметер, що необхідний лише для наслідування
        """
        weight = self.distance(source, destination)
        super().addEdge(source, destination, weight)

    def addVertex(self, vertex) -> bool:
        """ Додає вершину у граф, якщо така вершина не міститься у ньому
        :param vertex: ключ (тобто ім'я) нової вершини
        :return: True, якщо вершина успішно додана
        """
        if vertex in self:  # Якщо вершина міститься у графі, її вже не треба додавати
            return False
        newVertex = VertexWithHeuristic(vertex)  # створюємо нову вершину з іменем key
        self.mVertices[vertex] = newVertex  # додаємо цю вершину до списку вершин графу
        self.mVertexNumber += 1          # Збільшуємо лічильник вершин у графі
        return True

    def distance(self, source, destination):
        """ Визначає геометричну відстань (по прямій лінії) між двома вершинами source
            та destination у графі
        :param source: Перша вершина
        :param destination: Друга вершина
        :return: Геометрична відстань між вершинами source та destination
        """
        source_position = self.getData(source)
        destination_position = self.getData(destination)
```

```
        assert source_position is not None and destination_position is not None
        return (((destination_position[0] - source_position[0]) ** 2)
                    + ((destination_position[1] - source_position[1]) ** 2)) ** 0.5
```

Наведемо тепер алгоритм A*, що майже дослівно буде повторювати алгоритм Дейкстри.

Лістинг 7.4.9. Алгоритм A*.

```
def AStar(graph, start, end):
    """ Функція, що реалізує пошук шляху у графі між двома вершинами використовуючи A* алгоритм
    :param graph: Граф
    :param start: Стартова вершина
    :param end: Кінцева вершина
    :return: Кортеж, що містить список вершин - найкоротший шлях,
             що сполучає вершини start та end та його вагу
    """

    # Ініціалізуємо додаткову інформацію у графі для роботи алгоритму.
    for vertex in graph:         # Для кожної вершини графа
        vertex.setDistance(INF)  # Відстань від стартової - нескінченність
        vertex.setSource(None)   # Вершина з якої прийшли по найкоротшому шляху невизначена
        vertex.calculateHeuristic(graph[end])  # Обчислюємо значення евристичної функції

    # Відстань від першої вершини до неї ж визначається як 0
    graph[start].setDistance(0)

    pq = PriorityQueue()       # Створюємо пріоритетну чергу
    pq.insert(start, 0)        # Додаємо у чергу початкову вершину з нульовим пріоритетом

    # Введемо масив, що буде містити ознаку чи фіксована вже вершина.
    # Ініціалізуємо масив значеннями False (що означає, що вершина не фіксована)
    fixed = [False] * len(graph)

    while not pq.empty():                  # Поки черга не порожня
        vertex_key = pq.extractMinimum()   # Беремо індекс вершини з черги з найвищим пріоритетом
        fixed[vertex_key] = True           # та позначаємо її як фіксавану
        vertex = graph[vertex_key]         # Беремо поточну вершину за індексом

        if vertex_key == end:              # Якщо поточний елемент є шуканим
            break                          # пошук завершено

        for neighbor_key in vertex.neighbors():  # Для всіх сусідів (за ключами) поточної вершини
            if fixed[neighbor_key]:              # які ще не були фіксовані
                continue

            neighbour = graph[neighbor_key]      # Беремо вершину-сусіда за ключем
            # Обчислюємо потенційну відстань у вершині-сусіді
            # newDist = g(x) згідно з алгоритмом
            newDist = vertex.distance() + vertex.weight(neighbor_key)
            # Якщо потенційна відстань у вершині-сусіді менша за її поточне значення
            if newDist < neighbour.distance():
                # Змінюємо поточне значення відстані у вершині-сусіді обчисленим
                neighbour.setDistance(newDist)
                # Встановлюємо для сусідньої вершини ідентифікатор звідки ми прийшли у неї
                neighbour.setSource(vertex_key)
                # Беремо значення евристичної функції у вершині-сусіді.
                h = neighbour.heuristic()
                # f(x) = g(x) + h(x) - обчилюємо новий пріорітет для вершини-сусіда.
                f = newDist + h

                if neighbor_key in pq:  # Якщо вершина сусід міститься у черзі
                    # перераховуємо її пріоритет в черзі
                    pq.updatePriority(neighbor_key, f)
                else:
                    # або додаємо елемент до черги, якщо його там ще немає.
                    pq.insert(neighbor_key, f)
```

```
    return graph.constructWay(start, end)    # Повертаємо шлях та його вагу
```

7.4.4. Побудова кістякового дерева та алгоритм Прима

Означення 7.4.1. Кістяковим (або каркасним) деревом (англ. Spanning tree) зв'язного неорієнтованого графа називається ациклічний зв'язний підграф цього графа, який містить усі його вершини.

Іншими словами, для того щоб утворити каркасне дерево графа, потрібно з вихідного графа видалити максимальну кількість ребер, так, щоб при цьому граф лишався зв'язним. Очевидно, що якщо граф містить n вершин, то кістякове дерево цього графу має містити $n-1$ ребро. На рисунку 7.4.24 зображено граф та його кістякове дерево.

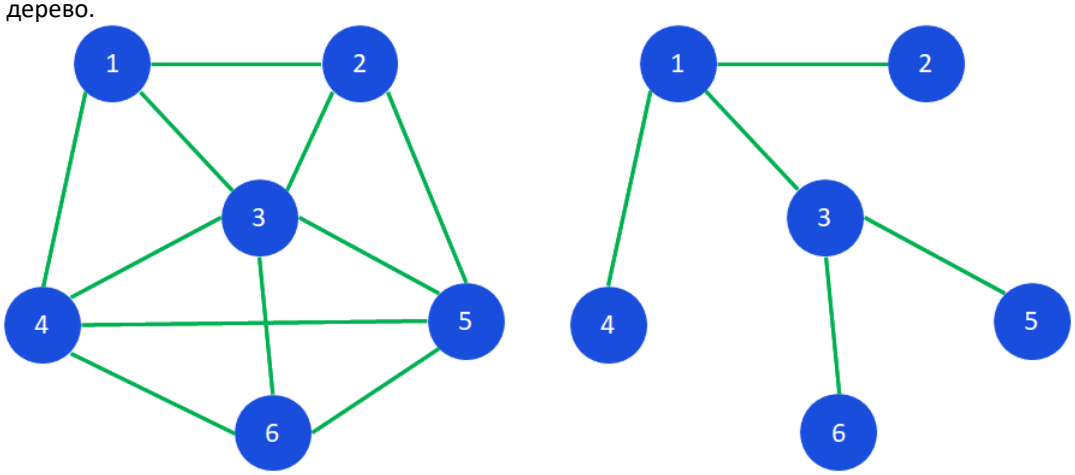


Рисунок 7.4.24. Неорієнтований граф та його кістякове дерево.

Очевидно, що кістякове дерево може бути побудоване різними способами та залежить від вибору алгоритму його побудови. Як правило, для побудови кістякового дерева застосовують один з алгоритмів обходу графа, наприклад пошук в ширину чи глибину. У цьому пункті зосередимося на побудові мінімального кістякового дерева у зваженому графі.

Означення 7.4.2. Мінімальним кістяковим деревом зв'язного неорієнтованого графа називається кістякове дерево цього графа, що має мінімально можливу вагу. Вагою дерева будемо назвати суму ваг його ребер.

На рисунку 7.4.25 зображено зважений неорієнтований граф та його мінімальне кістякове дерево (ребра, що входять до мінімального кістякового дерева на рисунку мають більшу товщину).

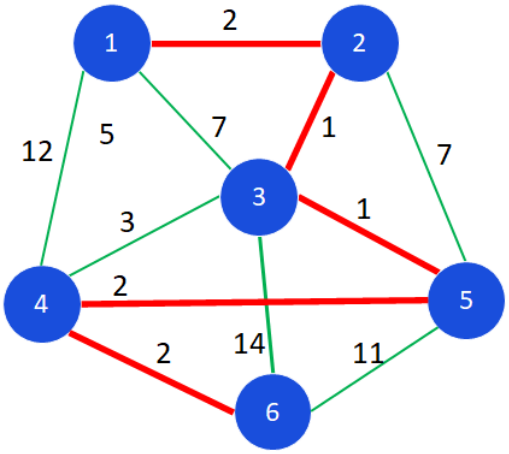


Рисунок 7.4.25. Мінімальне кістякове дерево зваженого графа.

Необхідність побудови мінімального кістякового дерева виникає у багатьох прикладних задачах. Однією з таких, є задача про мінімальну вартість побудови мережі автомобільних доріг, що сполучає набір міст. Якщо відома вартість побудови доріг між містами (між деякими або усіма парами міст), яку трактуватимемо як ребра, а міста – як вузли графа, то така потенційна мережа автомобільних доріг утворює зважений неорієнтований граф. Очевидно, що алгоритм який дозволить побудувати мінімальне кістякове дерево на цьому графі дозволить

визначити пріоритетні дороги, що з'єднають усі міста, і будівництво яких буде коштувати мінімальних затрат для регіону.

Існує кілька алгоритмів побудови кістякового дерева, найвідоміші з яких алгоритм Прима, алгоритм Крускала, алгоритм Борувки. У цьому пункті вивчимо найпростіший з них – алгоритм Прима.

Ідея алгоритму Прима дуже проста і полягає у тому, що, починаючи з деякої вершини, на кожному кроці алгоритму до побудованого на попередніх кроках дерева додається ребро з найменшою вагою. Цей процес відбувається доти, доки дерево не стане містити всі вершини вихідного графа. Звернемо увагу читача на те, що на вхід алгоритм Прима отримує зважений зв'язаний неорієнтований граф.

Розглянемо на прикладі графу зображеного на рисунку 7.4.25 побудову мінімального кістякового дерева. На першому кроці обираємо довільну вершину з якої починається робота алгоритму. Нехай це буде вершина 3.

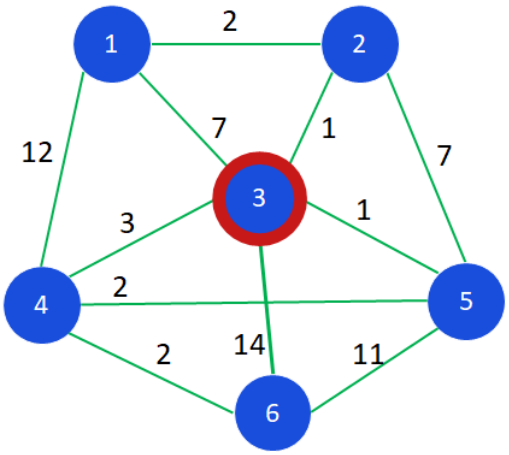


Рисунок 7.4.26. Вибираємо початкову вершину для побудови кістякового дерева.

Тепер, серед сусідів вершини 3 знаходимо вершину, що розташовується "найближче" до неї. Тобто серед ребер, що сполучають вершину 3 з сусідніми вершинами, обираємо те з них, що має найменшу вагу. У нас два таких ребра – кожне з ребер, що сполучають вершину 3 з вершинами 2 та 5 мають вагу 1. Додаємо до дерева одну цих вершин, наприклад, вершину 5.

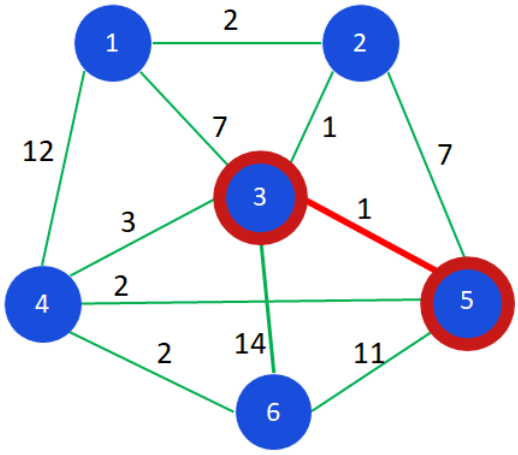


Рисунок 7.4.27. Додаємо дерева вершину 5.

На наступному кроці вибираємо вершину, що розташовується найближче що побудованого на попредніх кроках дерева, тобто знаходимо ребро з найменшою вагою, що виходить з вершин 3 та 5. Таким є ребро, що сполучає вершину 3 з вершиною 2 – його вага 1. Додаємо вершину 2 разом з ребром, що сполучає вершини 3 та 2.

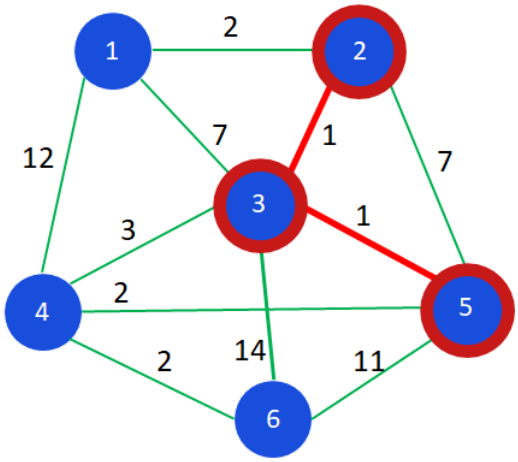


Рисунок 7.4.28. Додаємо дерева вершину 3.

Серед вершин 1, 4 та 6, що наразі не входять до дерева, знову обираємо ту, що знаходиться найближче до дерева. Обидві з вершин 1 та 4 знаходяться на відстані 2 від дерева, тому обираємо довільну з них, наприклад вершину 4.

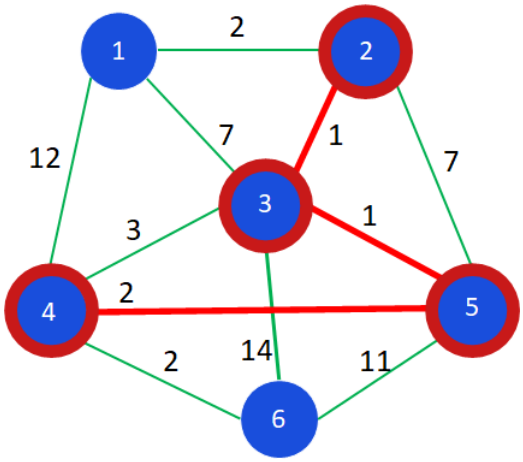


Рисунок 7.4.29. Додаємо дерева вершину 4.

Останні два кроки алгоритму додають вершини 1 та 6 (з однаковим пріоритетом, оскільки обидві вони знаходяться на відстані 2 від побудованого на попередніх кроках кістякового дерева)

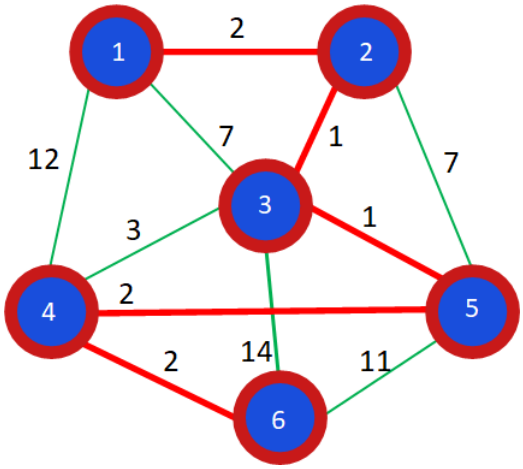


Рисунок 7.4.30. Додавання вершин 1 та 6 завершує побудову кістякового дерева.

Отже, всі вершини були опрацьовані та додані до дерева, що завершує побудову мінімального кістякового дерева.

Наведемо тепер реалізацію алгоритму Прима побудови кістякового дерева.

Лістинг 7.4.10. Реалізація алгоритму Прима побудови кістякового дерева.

```
def Prim(graph):
    """ Реалізує алгоритм Прима побудови каркасного дерева
    :param graph: заданий граф на базі якого будується каркасне дерево
    :return: граф, що є мінімальним каркасним деревом для заданого графа
    """
    assert not graph.mIsOriented
    assert checkConnected(graph)

    start = 0  # Вибираємо довільну точку графа як початкову з якої стартує алгоритм

    # Ініціалізуємо додаткову інформацію у графі для роботи алгоритму.
    # У алгоритмі Прима, ця додаткова інформація визначається у кожній
    # вершині як відстань від неї, до вже побудованого дерева.
    for vertex in graph:
        vertex.setDistance(INF)  # Відстань для кожної вершини ініціалізується як нескінченність
        vertex.setSource(None)   # Вершина з якої прийшли по найкорошому шляху невизначена

    # Відстань у старотовій вершині (тобто від стартової вершини до себе) визначається як 0
    graph[start].setDistance(0)

    pq = PriorityQueue()  # Створюємо пріоритетну чергу
    # Додаємо у чергу з пріоритетом всі вершини графа.
    for vertex in graph:
        pq.insert(vertex.key(), vertex.distance())  # де пріоритет - це відстань у вершині

    while not pq.empty():
        vertex_key = pq.extractMinimum()  # Беремо індекс вершини з черги з найнижчим пріоритетом
        vertex = graph[vertex_key]        # Беремо вершину за індексом
        for neighbor_key in vertex.neighbors():   # Для всіх сусідів поточної вершини
            neighbour = graph[neighbor_key]       # Беремо вершину-сусіда за індексом
            newDist = vertex.weight(neighbor_key) # Визначаємо вагу ребра між вершиною та сусідом
            # Якщо вершина-сусід не додана до каркасного дерева і потенційна відстань у
            # вершині-сусіді менша за її поточне значення
            if neighbor_key in pq and newDist < neighbour.distance():
                # Змінюємо поточне значення відстані у вершині-сусіді обчисленим
                neighbour.setDistance(newDist)
                # Встановлюємо для сусідньої вершини ідентифікатор звідки ми прийшли у неї
                neighbour.setSource(vertex_key)
                pq.updatePriority(neighbor_key, newDist)  # перераховуємо її пріоритет в черзі

    # Будуємо граф, що є каркасним деревом
    spanning_tree = GraphForAlgorithms()
    for vertex in graph:
        destination = vertex.key()
        source = vertex.source()
        if source is None:
            continue
        weight = vertex.weight(source)
        spanning_tree.addEdge(source, destination, weight)

    return spanning_tree
```

Як бачимо, алгоритм Прима дуже подібний на алгоритм Дейкстри відшукання найкоротшого шляху. Пріоритетна черга дозволяє на кожному кроці вибрати вершину, що розташовується найближче до дерева. Результатом роботи алгоритму є побудованих граф, що є кістяковим деревом. Для цього проходяться усі вершини вихідного графа та додаються до нового графу разом з ребрами які були отримані під час виконання алгоритму.

СПИСОК ЛІТЕРАТУРИ ТА ВИКОРИСТАНІ ДЖЕРЕЛА

1. Кнут Д. Э. Искусство программирования. Том 1. Основные алгоритмы = The Art of Computer Programming. Volume 1. Fundamental Algorithms / под ред. С. Г. Тригуб (гл. 1), Ю. Г. Гордиенко (гл. 2) и И. В. Красикова (разд. 2.5 и 2.6). — 3. — Москва: Вильямс, 2002. — Т. 1. — 720 с. — ISBN 5-8459-0080-8.
2. Кнут Д. Э. Искусство программирования. Том 2. Получисленные алгоритмы = The Art of Computer Programming. Volume 2. Seminumerical Algorithms / под ред. Л. Ф. Козаченко (гл. 3, разд. 4.6.4 и 4.7), В. Т. Тертышного (гл. 4) и И. В. Красикова (разд. 4.6). — 3. — Москва: Вильямс, 2001. — Т. 2. — 832 с. — ISBN 5-8459-0081-6.
3. Кнут Д. Э. Искусство программирования. Том 3. Сортировка и поиск = The Art of Computer Programming. Volume 3. Sorting and Searching / под ред. В. Т. Тертышного (гл. 5) и И. В. Красикова (гл. 6). — 2-е изд. — Москва: Вильямс, 2007. — Т. 3. — 832 с. — ISBN 5-8459-0082-1.
4. Кнут Д. Э. Искусство программирования, том 4, А. Комбинаторные алгоритмы, часть 1 = The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1 / под ред. Ю. В. Козаченко. — 1. — Москва: Вильямс, 2013. — Т. 4. — 960 с. — ISBN 978-5-8459-1744-7.
5. Bruno R. Preiss. Data Structures and Algorithms with Object-Oriented Design Patterns in Python / Bruno R. Preiss., 2003. – 566 с.
6. Калиткин Н.Н. Численные методы.– Наука, 1978. — 512 с.
7. Problem Solving with Algorithms and Data Structures using Python [Електронний ресурс] – Режим доступу до ресурсу: https://runestone.academy/runestone/books/published/pythonds/index.html.
8. Вихідний код програм [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/krenevych/algo.
9. Kent D. Lee. Data Structures and Algorithms with Python / Kent D. Lee, Steve Hubbard., – Springer, 2015. – 363 с.
10. Лафоре Р. Структуры данных и алгоритмы в Java. Классика Computers Science. 2-е изд. — СПб.: Питер, 2013. — 704 с.
11. Клейнберг Дж., Тардос Е. Алгоритмы: разработка и применение. Классика Computers Science / Пер. с англ. Е. Матвеева. — СПб.: Питер, 2016. — 800 с.
12. Ахо, Альфред, В., Хопкрофт, Джон, Ульман, Джеффри, Д. Структуры данных и алгоритмы. : Пер. с англ. : М. : Издательский дом "Вильяме", 2003. — 384 с.
13. Коротєєва Т. О. Алгоритми та структури даних: навч. посібник / Т. О. Коротєєва. – Львів: Львівської політехніки, 2014. – 280 с.
14. Креневич, А.П. С у задачах і прикладах : навчальний посібник із дисципліни "Інформатика та програмування" / А.П. Креневич, О.В. Обвінцев. – К. : Видавничо-поліграфічний центр "Київський університет", 2011. – 208 с.
15. E-Olymp [Електронний ресурс] – Режим доступу до ресурсу: www.e-olymp.com.
16. Вирт Н. Алгоритмы + структуры данных = программы.–М.: Мир, 1985. –406 с.
17. The Python Tutorial [Електронний ресурс] – Режим доступу до ресурсу: https://docs.python.org/3/tutorial/index.html.