```
# 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.

77 → 57 → 26 → 17 → **32**

верхівка стеку

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**.

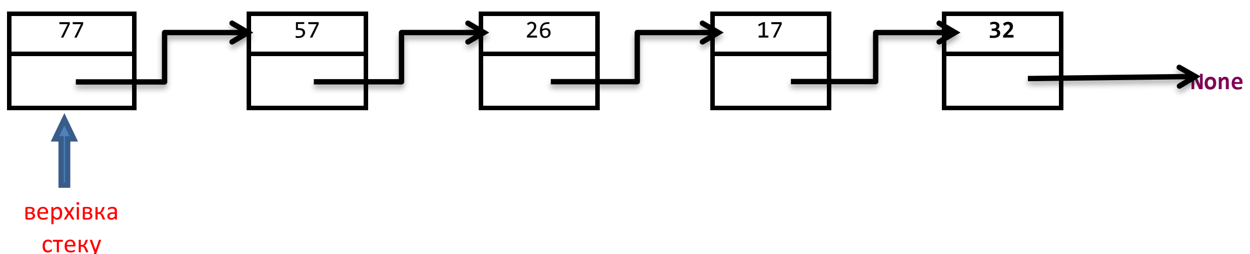


Рисунок 5.1.3. Рекурсивна реалізація стеку.

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 струтктури. 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

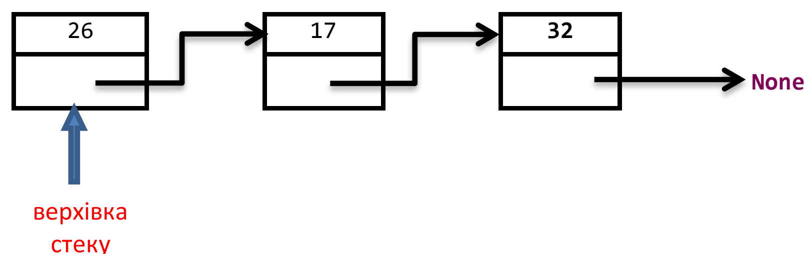
Лістинг 5.1.2. Продовження. Реалізація стеку як рекурсивної струтктури.

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

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

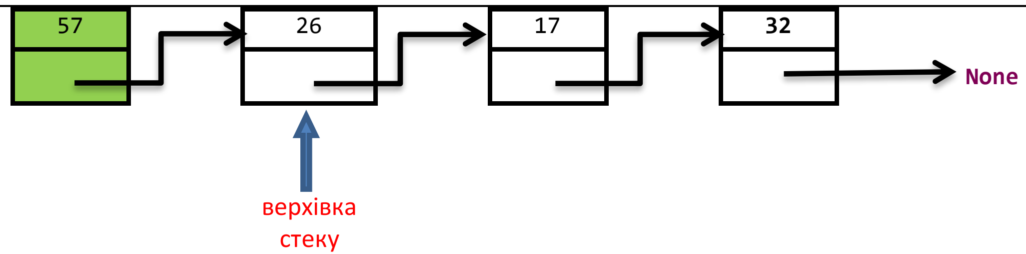
[3] Аллокація (allocation) у програмуванні, процес динамічного виділення пам’яті для розміщення даних об’єктів. Реаллокація (reallocation) – зміна розташування даних об’єкта у пам’яті.

```
    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 місить 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
```

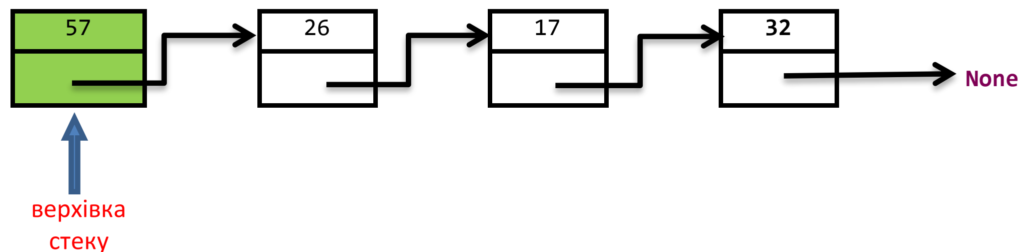
Пояснимо для прикладу схематично роботу методу `push()`. Припустимо у стек додано елементи (32, 17, 26)



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:

$$z = a_{n-1}a_{n-2} \dots a_1 a_0 \qquad (5.1.1)$$

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

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

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)

$$z = a_{n-1}a_{n-2} \dots a_1 a_0 \qquad (5.1.3)$$

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:

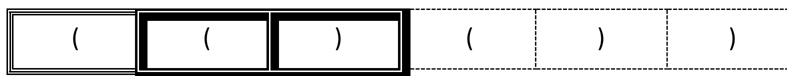
$$4\big((x+1)*(y+2)-2\big)$$

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

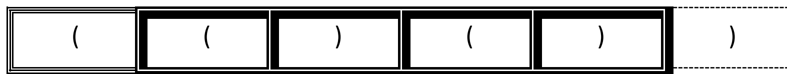
Дужкова послідовність	Аналіз
(( )( ))	правильна
(( )( )( )( ))	правильна
(((( ))))	правильна
)( )(	не правильна
(((( )	не правильна
( ))))	не правильна

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.

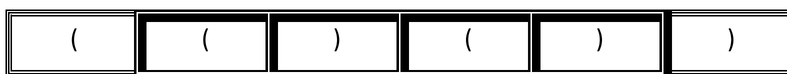
Для прикладу, розглянемо першу дужкову послідовність з вищенаведеної таблиці. Після аналізу перших трьох дужок, зустрівся контейнер (утворений другою і третьою дужкою), який можна виключити з аналізу.



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 пріоритет
    "-": 1,  # Оператор віднімання та його пріоритет
    "*": 2,  # Оператор множення та його пріоритет
    "/": 2,  # Оператор ділення та його пріоритет
}

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 обереннений 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).

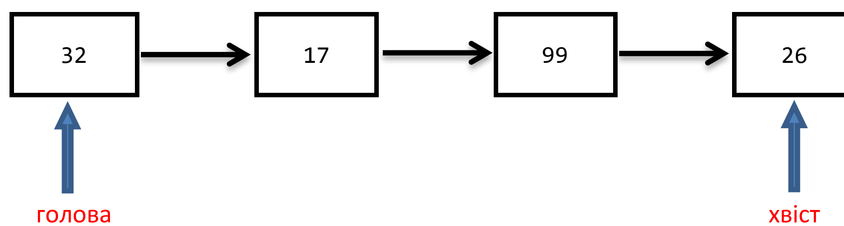


Рисунок 5.2.1. Черга, має початок та кінець.

Елемент, що додається до черги, опиняється у її хвості.

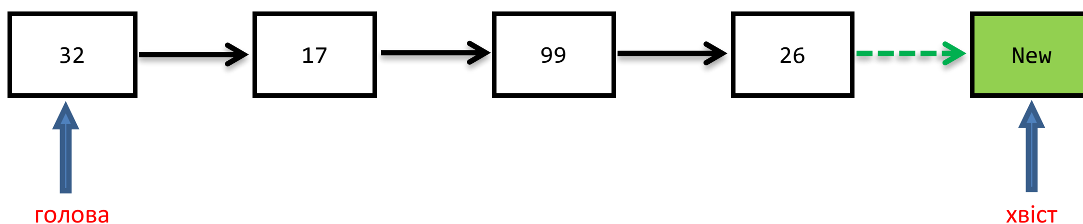


Рисунок 5.2.2. Додавання елемента «New» в кінець черги

Елемент, що береться (тобто видаляється) з черги, розташований у її голові.

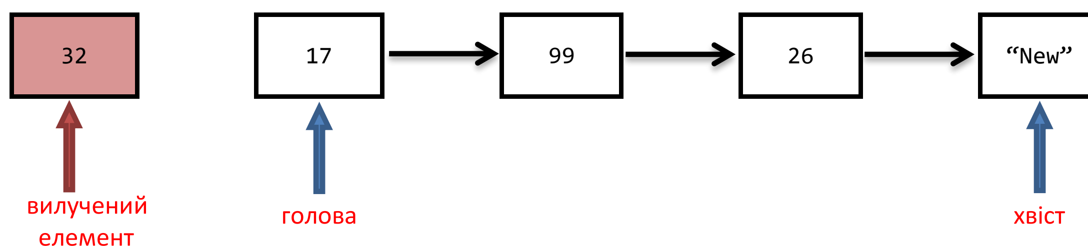


Рисунок 5.2.3. Вилучення першого елемента «32» з черги

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 здійснюються 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.

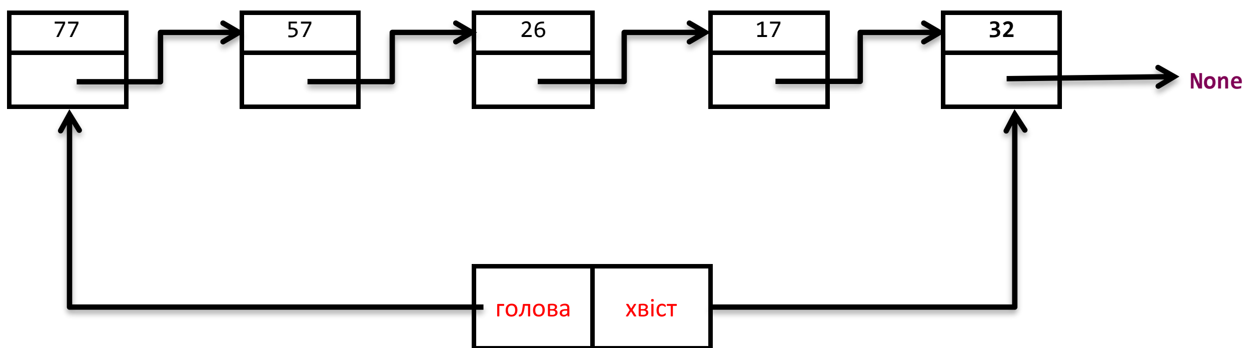


Рисунок 5.2.4. Рекурсивна реалізація черги.

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 структктури. 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 структктури.

```
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.

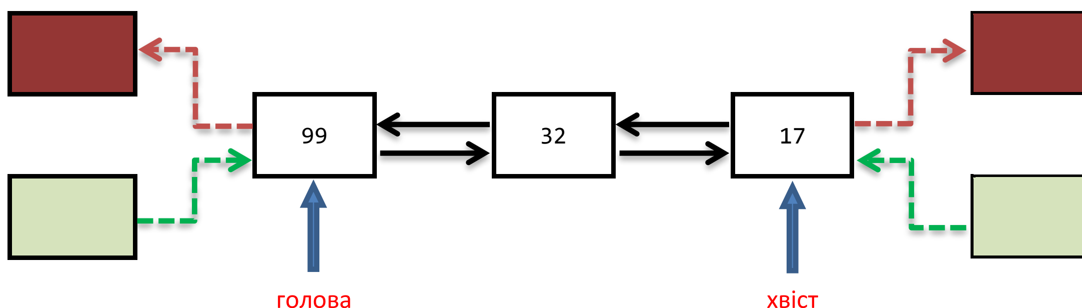


Рисунок 5.2.5. Дек елементів. Схематично позначено можливість додавання та вилучення елементів як у початок, так і в кінець структури даних.

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.

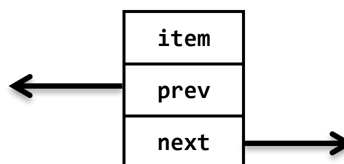


Рисунок 5.2.6. Вузол деку – містить посилання на попередній та наступний елементи деку.

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
```

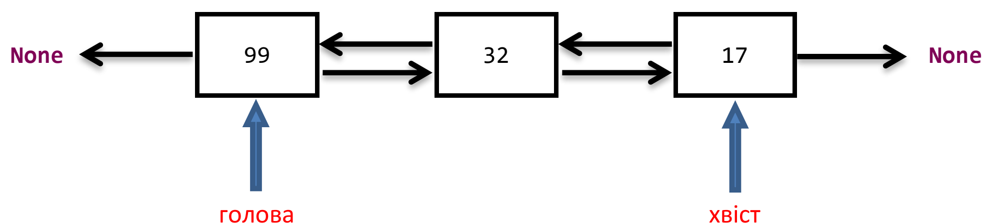
Використовуючи цей клас опишемо клас Deque, що є реалізацією двосторонньої черги як рекурсивної структури даних.

Лістинг 5.2.4. Продовження. Реалізація деку як рекурсивної структктури.

```
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
```

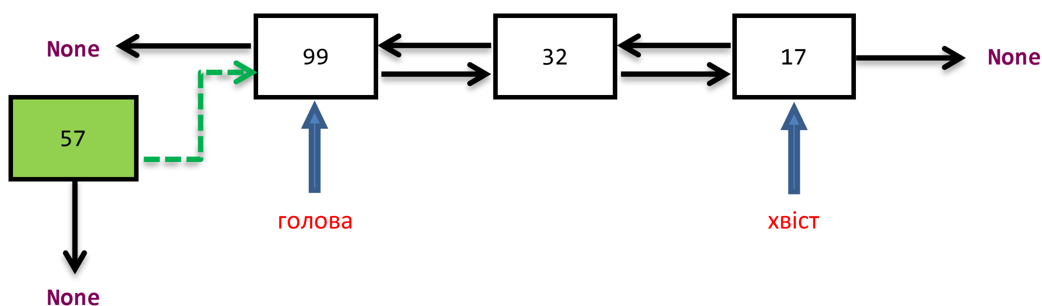
Пояснимо операцію **appendleft()**. Інші операції пропонуємо розібрати читачу самостійно. Отже, розглянемо дек, зображений нижче.



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.

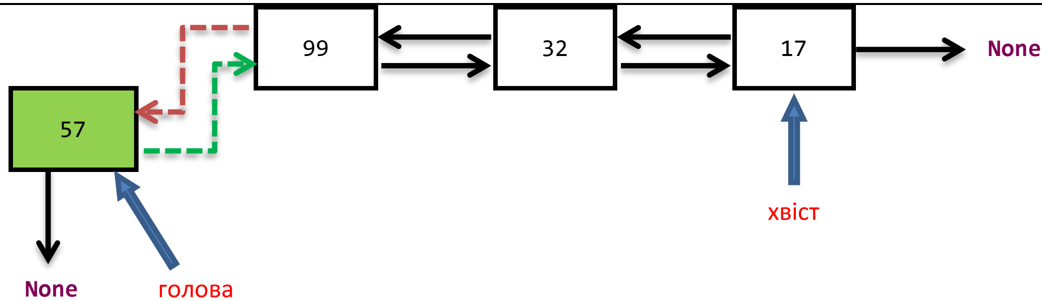


Рисунок 5.2.7. Дек елементів. Додавання елементу у голову деку.

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.

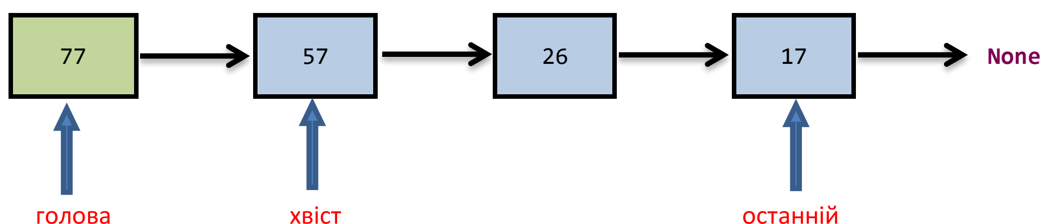


Рисунок 5.3.1. Класичний список

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):
        """ Поверате 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.

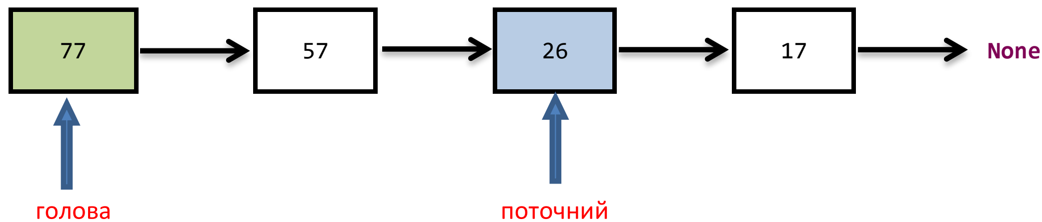


Рисунок 5.3.2. Список з поточним елементом

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 список
        :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.

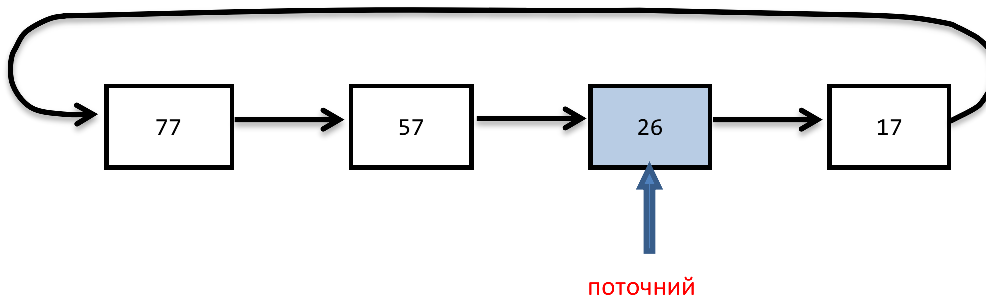


Рисунок 5.3.3. Кільцевий список

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 список
        :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.

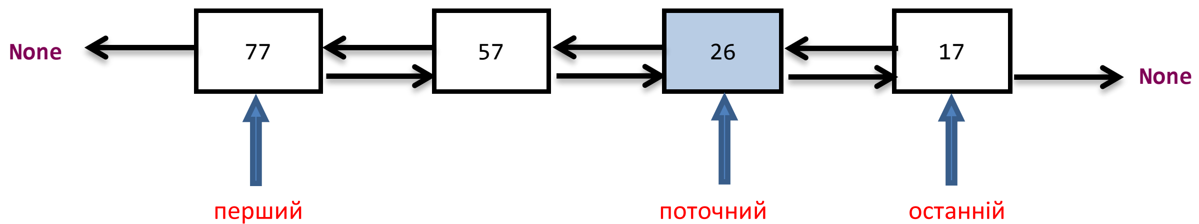


Рисунок 5.3.4. Двозв'язний список

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
```

```python
    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:)».

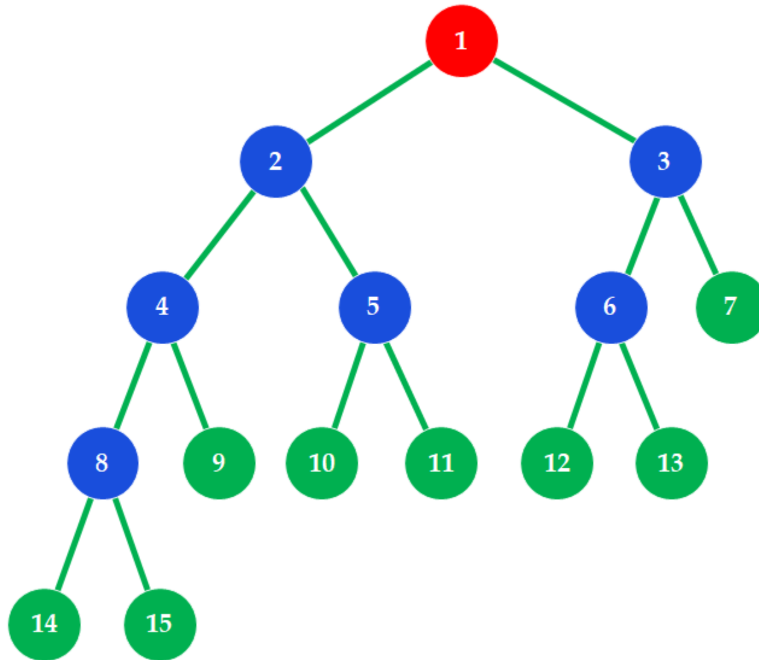


Рисунок 6.1.1. Приклад дерева.

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.