ІНФОРМАТИКА ТА ПРОГРАМУВАННЯ

Тема 14. Рекурсивні структури даних

Статичні та динамічні структури даних

- У попередніх темах ми вже розглянули різні типи даних.
- Їх можна поділити на статичні та динамічні з точки зору використання пам’яті комп’ютера.
- Пам’ять для **статичних структур даних** виділяється один раз перед (або під час) виконанням програми, а її обсяг не може бути змінений.
  - Прикладом таких статичних типів у Python є, зокрема, дійсний тип даних.
- **Динамічні структури даних** можуть змінювати обсяг виділеної для них пам’яті у процесі виконання програми.
  - Прикладами динамічних структур даних у Python є списки та словники.

Статичні та динамічні структури даних.2

- Існує багато задач, в яких розмір даних суттєво залежить від умов, які обчислюються тільки під час виконання програм.
- Для таких задач необхідно мати саме динамічні структури даних.
- Реалізація динамічних структур даних залежить від мови програмування: динамічні структури даних можуть бути вбудовані у мову програмування або можуть бути надані засоби побудов таких структур.
- Найпоширенішим засобом побудови є вказівники.

Рекурсивні структури даних

- Серед динамічних структур даних окремо виділяють рекурсивні структури.
- Аналогічно рекурсивним підпрограмам, **рекурсивні структури даних** у своєму описі посилаються самі на себе.
- Взагалі поняття динамічних структур даних є більш широким, ніж поняття рекурсивних структур даних, але в цій темі ми будемо розглядати саме рекурсивні структури даних, їх визначення, реалізацію та використання.

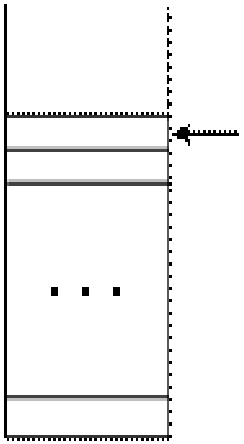
Рекурсивні структури даних.2

- Існує цілий ряд стандартних рекурсивних структур даних: стеки, черги, деки, різноманітні списки, дерева, графи.
- У Python для реалізації рекурсивних структур даних використовують як вбудовані структури (списки, словники), так і аналог вказівників – посилання на об’єкти.
- Для багатьох структур даних найпростішою є реалізація на базі списків Python, які самі є рекурсивною структурою даних.
- Спільним для реалізації всіх рекурсивних структур даних буде використання розглянутих раніше класів та об’єктів.
- Для рекурсивної структури даних визначаються свої операції відношення та інструкції.
- Тому ми можемо вважати, що кожна рекурсивна структура даних є новим типом даних.

14.1 СТЕКИ, ЧЕРГИ ТА ДЕКИ

Стек

- Першою та найбільш простою з рекурсивних структур даних є стек.
- Визначимо стек як:
- 1). Порожній стек.
- 2). Верхівка стеку; стек.
  - Читати це означення треба так: стек – це або порожній стек, або верхівка стеку, за якою слідує стек.



Стек.2

- Стек можна також представити, як сукупність однотипних елементів, в якій ми маємо доступ тільки до верхнього елемента. Цей елемент називають верхівкою стеку.
- Стеки називають ще структурами LIFO (Last In - First Out або Останнім прийшов – першим вийшов) або магазинами через схожість з магазинами стрілецької зброї.

Операції, відношення та інструкції для стеків

- Операції, відношення та інструкції для стеків:
- 1. Почати роботу.
- 2. Чи порожній стек?
- 3. Вштовхнути елемент у стек.
- 4. Виштовхнути верхівку стеку.
  - Дії 1, 3, 4 – інструкції; 2 – відношення.
- “Почати роботу” означає створити порожній стек.
- «Чи порожній стек?» - перевірити, чи є стек порожнім.
- “Вштовхнути елемент у стек” – додати до стеку один елемент, який стає верхівкою стеку.
- “Виштовхнути верхівку стеку” – повернути та видалити верхній елемент. Верхнім стає попередній елемент стеку або стек стає порожнім. Для порожнього стеку ця інструкція повинна давати помилку.

Реалізація стеку

- Для реалізації стеку використаємо список.
- Опишемо клас Stack наступним чином:

```
class Stack:
    '''Реалізує стек на базі списку.
    '''
    def __init__(self):
        '''Створити порожній стек.
        '''
        self._lst = []      #список елементів стеку

    def isempty(self):
        '''Чи порожній стек?.
        '''
        return len(self._lst) == 0
```

Реалізація стеку.2

```
def push(self, data):
    '''Вштовхнути елемент у стек.
    '''
    self._lst.append(data)

def pop(self):
    '''Взяти елемент зі стеку.
    '''
    if self.isempty():
        print('Pop: Стек порожній')
        exit(1)
    data = self._lst.pop()
    return data
```

Реалізація стеку.3

- Цей клас має одне внутрішнє поле _lst – список, який містить елементи стеку, та методи, що реалізують дії над стеком. Ми бачимо, що реалізація стеку на базі списку є дуже простою.
- Звернемо увагу на реалізацію повідомлення про помилку, якщо ми намагаємось взяти елемент з порожнього стеку:

```
if self.isempty():
        print('Pop: Стек порожній')
        exit(1)
```

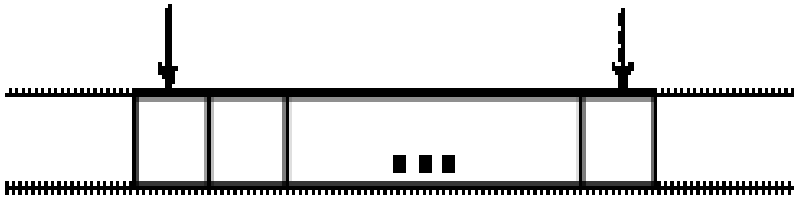
- Функція exit(1) аварійно завершує роботу програми у Python.

Приклад

- Дано послідовність рядків, яка вводиться з клавіатури. Показати цю послідовність у оберненому порядку

Черга

- Черга – ще одна рекурсивна структура даних, яку можна визначити так:
- 1). Порожня черга.
- 2). Перший елемент; черга.
- Чергу можна представити, як сукупність однотипних елементів, в якій ми маємо доступ до кінця черги при додаванні елементів та до початку черги при взятті елементів.



Операції, відношення та інструкції для черг

- 1. Почати роботу.
- 2. Чи порожня черга?
- 3. Додати елемент до кінця черги.
- 4. Взяти елемент з початку черги.
  - Дії 1, 3, 4 – інструкції; 2 – відношення.
- “Почати роботу” означає створити порожню чергу.
- “Додати елемент до кінця черги” – додати до черги один елемент, який стає останнім у черзі.
- “Взяти елемент” – взяти та повернути значення першого елемента. Першим стає попередній елемент черги або черга стає порожньою. Для порожньої черги ця інструкція повинна давати відмову.
- Черги ще називають структурами FIFO (First In - First Out або Першим прийшов – першим вийшов)

Реалізація черги

- Для реалізації черги використаємо список.
- Опишемо клас Queue (англійською - черга) наступним чином:

```
class Queue:
    '''Реалізує чергу на базі списку.
    '''
    def __init__(self):
        '''Створити порожню чергу.
        '''
        self._lst = []                  #список елементів черги

    def isempty(self):
        '''Чи порожня черга?.
        '''
        return len(self._lst) == 0
```

Реалізація черги.2

```
    def add(self, data):
        '''Додати елемент в кінець черги.
        '''
        self._lst.append(data)

    def take(self):
        '''Взяти елемент з початку черги.
        '''
        if self.isempty():
            print('Take: Черга порожня')
            exit(1)
        data = self._lst.pop(0)        #перший елемент черги - це нульовий елемент списку
        return data

    def __del__(self):
        '''Закінчити роботу з чергою.
        '''
        print('Deleting queue')
        del self._lst
```

Реалізація черги.3

- Реалізація черги на базі списку настільки ж нескладна, як і реалізація стеку.
- Звернемо увагу на метод __del__. Він є внутрішнім методом Python та викликається тоді, коли об’єкт класу Queue знищується.
- Такі методи називають **деструкторами**.
- Часто у деструкторах можна не писати код. Ми його написали з метою демонстрації а також з метою вивільнення пам’яті, раніше виділеної під список _lst.

Приклад. Задача «Лічилка»

- По колу розташовано n гравців з номерами від 1 до n. У лічилці m слів. Починають лічити з першого гравця. m-й за ліком вибуває. Потім знову лічать з наступного гравця за вибулим. Знову m-й вибуває. Так продовжують, поки не залишиться жодного гравця. Треба показати послідовність номерів, що вибувають, при заданих n та m.
- Для розв'язання задачі використаємо чергу.
- Опишемо клас Player (Гравець), який містить методи __init__ - створити гравця – та show – показати номер гравця.
- Спочатку до черги додамо n гравців з номерами від 1 до n.
- Потім будемо (m-1) раз перекладати гравця з початку до кінця черги (брати спочатку да додавати до кінця), імітуючи лік. m-го гравця візьмемо спочатку черги та покажемо його номер.
- Будемо повторювати лік, поки черга не спорожніє.

Дек

- Дек називають двостороннім стеком або двосторонньою чергою.

- Визначимо дек:
- 1). Порожній дек.
- 2). Перший елемент; дек.
- 3). Дек; останній елемент.
- Дек можна представити, як сукупність однотипних елементів, в якій ми маємо доступ до початку або кінця деку для додавання або взяття елементів.

Операції, відношення та інструкції для деків

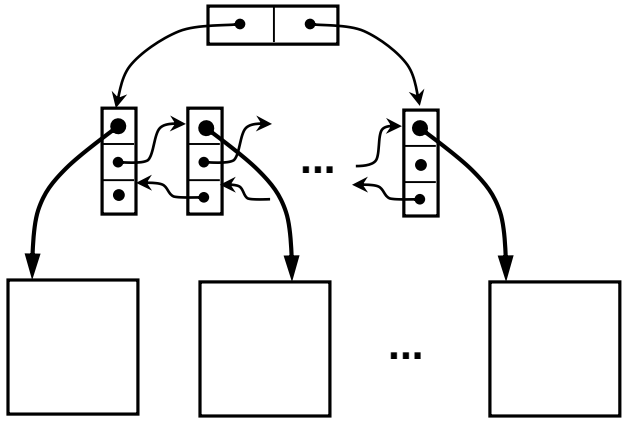
- 1. Почати роботу.
- 2. Чи порожній дек?
- 3. Додати елемент до початку деку.
- 4. Взяти елемент з початку деку.
- 5. Додати елемент до кінця деку.
- 6. Взяти елемент з кінця деку.
  - Дії 1, 3, 4, 5, 6 – інструкції; 2 – відношення.
- “Почати роботу” означає створити порожній дек.
- “Додати елемент до початку деку” – додати до деку один елемент, який стає першим у деку.
- “Взяти елемент з початку деку” – взяти та повернути значення першого елемента. Першим стає наступний елемент деку або дек стає порожнім. Для порожнього деку ця інструкція повинна давати відмову.
- “Додати елемент до кінця деку” – додати до деку один елемент, який стає останнім у деку.
- “Взяти елемент з кінця деку” – взяти та повернути значення останнього елемента. Першим стає попередній елемент деку або дек стає порожнім. Для порожнього деку ця інструкція повинна давати відмову.

Реалізація деку

- Для реалізації деку використаємо посилання на об’єкти.
- При створенні об’єкту Python динамічно виділяє нову пам’ять, а сама змінна є посиланням на початкову адресу виділеного блоку пам’яті.
- Тому ми можемо зв’язати між собою елементи деку за допомогою посилань на об’єкти а також визначити клас для деку в цілому.

Реалізація деку.2

- Опишемо класи _Delem та Deque.
- Клас _Delem є внутрішнім класом модуля та реалізує елемент деку з даними та посиланнями на попередній та наступний елементи(_prev та _next).
- Клас Deque реалізує сам дек як сукупність елементів з визначеними посиланнями. Можемо вважати, що елементи деку зв’язані у ланцюг двома посиланнями: на попередній та наступний елемент. А сам дек містить посилання на перший та останній елементи деку.



Реалізація деку. Клас _Delem

```
class _Delem:
    '''Реалізує елемент деку.
    '''
    def __init__(self, data):
        '''Створити елемент.
        '''
        self._data = data       #дані, що зберігаються у елементі деку
        self._next = None       #посилання на наступний елемент
        self._prev = None       #посилання на попередній елемент
```

Реалізація деку. Клас Deque

```
class Deque:
    '''Реалізує дек без використання списку.
    '''
    def __init__(self):
        '''Створити порожній дек.
        '''
        self._bg = None
        self._en = None

    def isempty(self):
        '''Чи порожній дек?.
        '''
        return self._bg == None and self._en == None
```

Реалізація деку. Клас Deque.2

```
def putbg(self, data):
        '''Додати елемент до початку деку.
        '''
        elem = _Delem(data)          #створюємо новий елемент деку
        elem._next = self._bg       #наступний елемент для нового - це елемент, який є першим
        if not self.isempty():      #якщо додаємо до непорожнього деку
            self._bg._prev = elem   #новий елемент стає попереднім для першого
        else:
            self._en = elem         #якщо додаємо до порожнього деку, новий елемент буде й останнім
        self._bg = elem             #новий елемент стає першим у деку
```

Реалізація деку. Клас Deque.3

```
  def getbg(self):
        '''Взяти елемент з початку деку.
        '''
        if self.isempty():
            print('getbg: Дек порожній')
            exit(1)
        elem = self._bg             #elem - посилання на перший елемент деку
        data = elem._data           #запам'ятовуємо дані для повернення
        self._bg = elem._next       #першим стає наступний елемент деку
        if self._bg == None:        #якщо в деку був 1 елемент
            self._en = None         #дек стає порожнім
        else:
            self._bg._prev = None   #інакше у новому першому елементі посилання на попередній - None
        del elem
        return data
```

Реалізація деку. Клас Deque.4

```
#дії puten та geten повністю симетричні діям putbg та getbg відповідно
    def puten(self, data):
        '''Додати елемент до кінця деку.
        '''
        elem = _Delem(data)
        elem._prev = self._en
        if not self.isempty():
            self._en._next = elem
        else:
            self._bg = elem
        self._en = elem

    def geten(self):
        '''Взяти елемент з кінця деку.
        '''
        if self.isempty():
            print('geten: Дек порожній')
            exit(1)
        elem = self._en
        data = elem._data
        self._en = elem._prev
        if self._en == None:
            self._bg = None
        else:
            self._en._next = None
        del elem
        return data
```

Реалізація деку. Клас Deque.5

```
def __del__(self):
        '''Закінчити роботу з деком.
        '''
        while self._bg != None:          #проходимо по всіх елементах деку
            elem = self._bg              #запам'ятовуємо посилання на елемент
            self._bg = self._bg._next    #переходимо до наступного елементу
            del elem                     #видаляємо елемент
        self._en = None
```

Реалізація деку. Завершення

- Слід відмітити, що дії додавання елемента до кінця деку та взяття елемента з кінця деку (puten та geten) повністю симетричні діям додавання елемента до початку деку та взяття елемента з початку деку (putbg та getbg).
- Їх навіть можна отримати формально, паралельно замінивши всюди у тексті putbg та getbg bg на en, en на bg, next на prev та prev на next.
- Деструктор __del__ для деків є не просто демонстраційним, але й корисний тим, що звільняє пам’ять, виділену не тільки під об’єкт дек, але й під всі його елементи.

Приклад. Задача «Лічилка» з використанням деку

- По колу розташовано n гравців з номерами від 1 до n. У лічилці m слів. Починають лічити з першого гравця. m-й за ліком вибуває. Потім знову лічать з наступного гравця за вибулим. Знову m-й вибуває. Так продовжують, поки не залишиться жодного гравця. Треба показати послідовність номерів, що вибувають, при заданих n та m.
- Розв’язок цієї задачі з використанням деків практично не відрізняється від раніше розглянутого розв’язку з використанням черг.
- Ми тільки використовуємо відповідні методи для деку замість методів для черги.

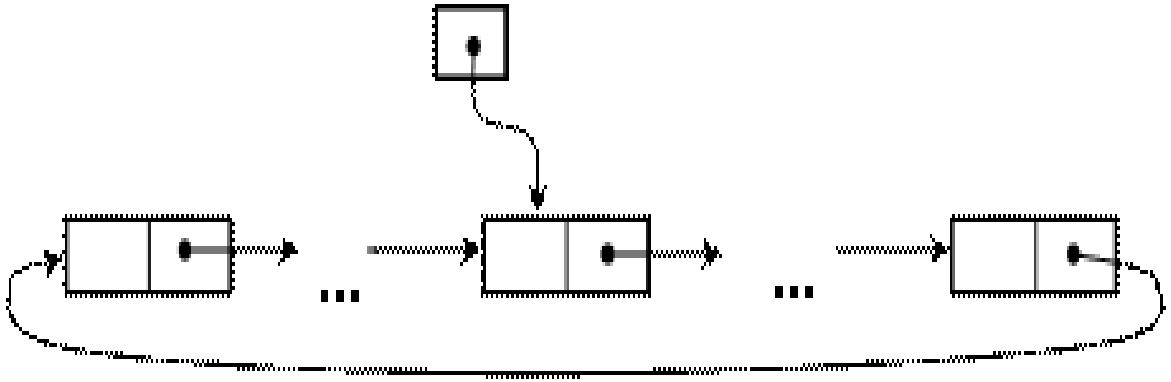
14.2 СПИСКИ

Списки

- Списки також є рекурсивними структурами даних.
- Списки відрізняються від стеків, черг та деків тим, що ми можемо багато разів проходити вздовж списку, отримувати доступ до будь-якого елемента, не змінюючи сам список.
- Список можна визначити так:
- 1). Порожній список.
- 2). Перший елемент; список.
- Є декілька різновидів списків: однозв’язні списки, кільцеві списки, двозв’язні списки.
- Для кожного з цих різновидів списків визначається свій набір операцій, відношень та інструкцій.
- У Python списки є стандартною структурою даних, яку ми розглядали раніше.
- Реалізація списків у Python є специфічною, оскільки ми можемо отримати прямий доступ до довільного елемента списку.
- Списки у Python ближчі до двозв’язних списків. У попередніх темах ми розглядали багато задач, у яких використовували списки. Тому немає потреби їх розглядати окремо.
- Розглянемо натомість один з різновидів списків, який не реалізований у Python: кільцевий список.

Кільцевий список

- Кільцевий список відрізняється від звичайного списку тим, що для кільцевого списку не визначають перший та останній елемент. Всі елементи зв’язані у кільце та відомий лише порядок слідування, а також елемент, який є поточним. Визначимо кільцевий список:
- 1). Порожній список.
- 2). Список; поточний елемент; список.



Набір дій над кільцевими списками

- 1. Почати роботу.
- 2. Довжина списку.
- 3. Перейти до наступного елемента.
- 4. Повернути поточний елемент.
- 5. Оновити поточний елемент.
- 6. Вставити елемент.
- 7. Видалити елемент.
  - Дії 1, 3, 5, 6, 7 – інструкції; 2, 4 - операції.
- Інструкція “Почати роботу” повертає порожній список.
- Операція “ Довжина списку” повертає кількість елементів у списку.
- “Перейти до наступного елемента” – зробити поточним наступний елемент списку. Якщо список порожній, то нічого не робити.
- “Повернути поточний елемент” повертає значення поточного елемента. Список при цьому не змінюється. Якщо список порожній, ця операція повинна давати відмову.
- “Оновити поточний елемент” змінює значення поточного елемента. Якщо список порожній, ця операція повинна давати відмову.
- “Вставити елемент” – вставити новий елемент у список перед поточним.
- “Видалити елемент” – видалити поточний елемент. Поточним стає наступний елемент або список стає порожнім. Якщо список порожній, інструкція повинна давати відмову.

Реалізація кільцевого списку

- Для реалізації кільцевого списку використаємо список Python, у якому будем зберігати елементи кільцевого списку.
- Опишемо клас Rlist, який містить поля _lst – список елементів – та _cur – індекс поточного елемента.
- Цей клас також містить методи, що реалізують дії над кільцевим списком.

Реалізація кільцевого списку.2

```
class Rlist:
    '''Реалізує кільцевий список на базі списку.
    '''
    def __init__(self):
        '''Створити порожній список.
        '''
        self._lst = []                  #список елементів
        self._cur = None                #індекс поточного елемента

    def len(self):
        '''Довжина списку.
        '''
        return len(self._lst)
```

Реалізація кільцевого списку.3

```
def next(self):
    '''Перейти до наступного елемента.
    '''
    l = self.len()
    if l != 0:
        if self._cur == l-1:    #для (l-1) елемента наступним буде нульовий
            self._cur = 0
        else:
            self._cur += 1

def getcurrent(self):
    '''Повернути поточний елемент.
    '''
    if self.len() == 0:
        print('getcurrent: список порожній')
        exit(1)
    data = self._lst[self._cur]
    return data
```

Реалізація кільцевого списку.4

```
def update(self, data):
        '''Оновити поточний елемент.
        '''
        if self.len() == 0:
            print('update: список порожній')
            exit(1)
        self._lst[self._cur] = data

    def insert(self, data):
        '''Вставити елемент перед поточним.
        '''
        if self.len() == 0:            #якщо список порожній
            self._lst.append(data)      #додаємо елемент, він стає поточним
            self._cur = 0
        else:
            self._lst.insert(self._cur,data) #інакше вставляємо елемент перед
поточним
            self._cur += 1                 #щоб поточний елемент не змінився,
треба індекс збільшити на 1
```

Реалізація кільцевого списку.5

```
  def delete(self):
      '''Видалити поточний елемент.
      '''
      if self.len() == 0:
          print('delete: список порожній')
          exit(1)
      l = self.len()
      del self._lst[self._cur]
      if l == 1:                  #якщо список після видалення елемента спорожніє
          self._cur = None
      elif self._cur == l-1:      #якщо поточним був останній елемент списку
          self._cur = 0           #поточним стане елемент з індексом 0
      #else: pass                 якщо поточним був не останній елемент, нічого не робити

   def __del__(self):
      '''Закінчити роботу зі списком.
      '''
      del self._lst
```

Приклад. Гра у відгадування слів

- Реалізувати гру у відгадування слів, яка полягає у наступному.
- По колу розташовані гравці (відгадувачі), яким презентують слово для відгадування.
- Всі літери цього слова спочатку закриті (замінені зірочками, ‘*’).
- Гравці вступають у гру по порядку. Кожен гравець може назвати літеру або слово.
- Якщо гравець називає літеру, а цієї літери, у слові немає, - хід переходить до наступного гравця. Якщо ж така літера у слові є, то всі входження цієї літери у слово відкриваються, а гравцю нараховуються стільки балів, скільки є входжень названої літери у слово. Якщо всі літери слова відкриті, - гравець стає переможцем.
- Якщо гравець називає слово і це слово не дорівнює заданому, то всі бали гравця анулюються, а хід переходить до наступного гравця. Якщо ж слово названо правильно, - гравець отримує стільки балів, скільки є у слові невідгаданих літер, та стає переможцем.
- Переможець отримує премію: стільки балів, скільки літер було у слові.

Приклад. Гра у відгадування слів. Розв'язання

- Для реалізації гри використаємо кільцевий список гравців (відгадувачів).
- Опишемо клас Guesser (Відгадувач), у якому будемо зберігати ім'я гравця та кількість зароблених балів.
- Слово будемо вибирати з текстового файлу наступним чином: знайдемо випадкове місце у файлі. Починаючи з цього місця, прочитаємо 10 рядків файлу, видалимо з них символи-розділювачі, переведемо до нижнього регістру та побудуємо список слів. З цього списку виберемо випадкове слово для відгадування.
- Побудуємо також рядок, який буде містити закриті та вгадані літери вибраного слова (спочатку – всі зірочки).
- Далі гравці будуть називати літери або слова а програма буде аналізувати відповіді та слідувати правилам гри до моменту, поки не буде відгадано задане слово.

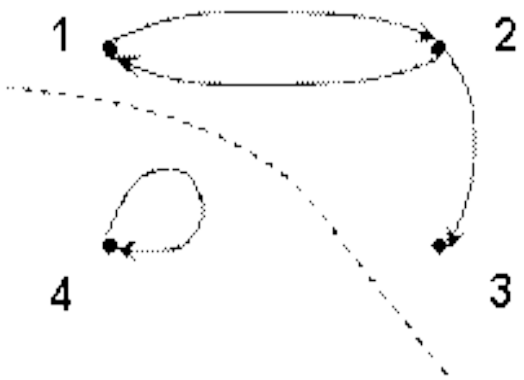
14.3 ДЕРЕВА ТА ГРАФИ

Графи та дерева. Основні поняття

- Стеки, черги та деки, списки є лінійними або одновимірними структурами даних. Дерева та графи є прикладами плоских або двовимірних структур. Дамо декілька означень.
- **Орієнтованим графом** G називають пару множин V, U
- $G = (V, U)$,
- де V – множина вершин, а U – множина дуг. Дуга з'єднує дві вершини графа.
- Далі орієнтовані графи будемо називати просто графами.

Графи та дерева. Основні поняття.2

- Приклад графа зображено на рисунку.



- Цей граф має 4 вершини з номерами від 1 до 4.
- Дуги з’єднують вершини 1 та 2, 2 та 1, 2 та 3, 4 та 4.
- Вершини графа також називають **вузлами**.

Графи та дерева. Основні поняття.3

- Якщо дуга *u* виходить з вершини v_1 та входить у вершину v_2, то кажуть, що v_2 **безпосередньо слідує з** v_1. Позначати це будемо так:
- $v_1 \xrightarrow{u} v_2$ або просто $v_1 \mapsto v_2$
- **Шлях** у графі *G* з вершини v_0 у вершину v_n – це послідовність вершин $v_0, v_1, v_2, \ldots, v_{n-1}, v_n$ така, що

$$v_0 \mapsto v_1; v_1 \mapsto v_2; \ldots; v_{n-1} \mapsto v_n$$

- Шлях будемо позначати $v_0 \to v_n$
- Якщо існує шлях між вершинами v_0 та v_n, кажуть, що v_n **слідує з** v_0 або v_n **є досяжною з** v_0.
- У графі на рисунку вище існують шляхи з 1 до 3, з 1 до 1, з 4 до 4 тощо.
- **Довжина шляху** у графі – це кількість вершин, які входять у шлях.
- Шлях між вершинами v_0 та v_n називається **циклом**, якщо $v_0 = v_n$.
- Граф на рисунку вище має цикли з 1 до 1, з 2 до 2 та з 4 до 4.

Графи та дерева. Основні поняття.4

- Граф *G* називають **незв'язним**, якщо існує розбиття множини вершин *V* на дві множини V_1, V_2 такі, що:

1. $V_1 \cup V_2 = V$, $V_1 \cap V_2 = \varnothing$
2. Для всіх $v_1 \in V_1$, $v_2 \in V_2$ не існує таких u_1, $u_2 \in U$, що $v_1 \xrightarrow{u_1} v_2$ або $v_2 \xrightarrow{u_2} v_1$

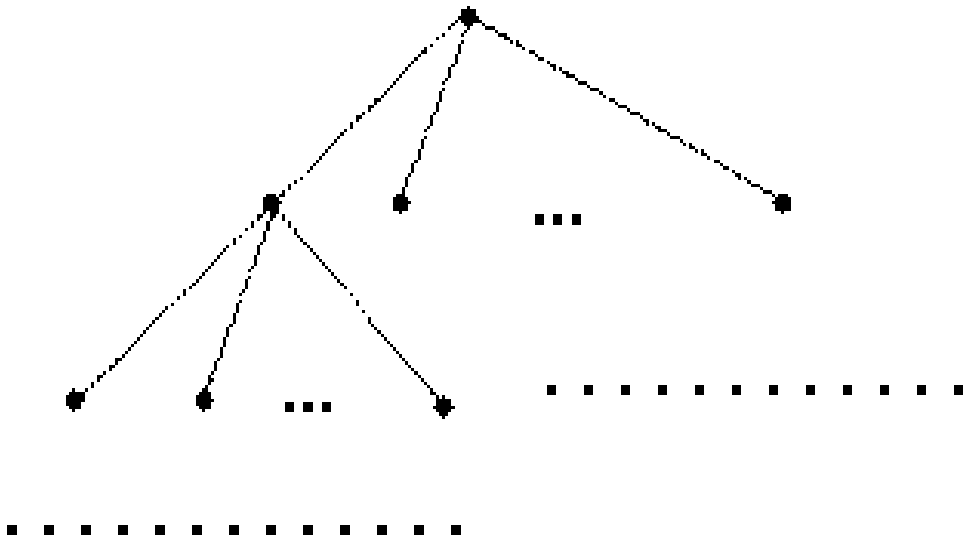
- Іншими словами, граф називають незв'язним, якщо всі його вершини можна розбити на дві підмножини вершин, між яким не проходить жодна дуга.
- Граф на рисунку вище є незв'язним, бо існує розбиття на дві підмножини вершин {1, 2, 3} та {4}, між якими не проходить жодна дуга.
- Граф *G* називають **зв'язним**, якщо він не є незв'язним.

Графи та дерева. Основні поняття.5

- **Напівстепінь входу** вершини *v* графа – це кількість дуг, які входять у дану вершину.
- **Напівстепінь виходу** вершини *v* графа – це кількість дуг, які виходять з даної вершини.
- Вершина з напівстепінню входу 0 називається **джерелом**, а вершина з напівстепінню виходу 0 – **стоком**.
- Часто з вершиною графа пов’язують певні дані. Такі дані називають **навантаженням** вершини.

Графи та дерева. Основні поняття.6

- **Деревом** називають зв’язний граф з одним джерелом та напівстепінню входу всіх вершин не більше 1. Дерева зображують, починаючи від джерела, вниз. Стрілки у дугах, як правило, опускають.

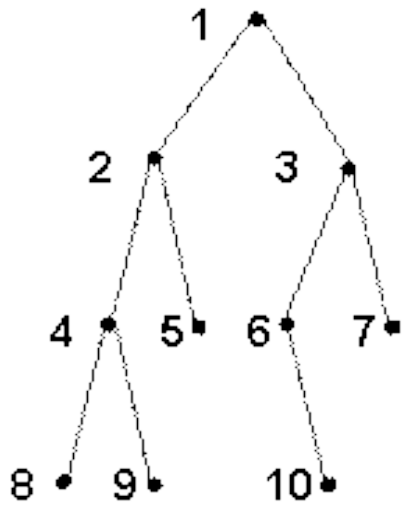


Графи та дерева. Основні поняття.7

- Єдине джерело називають **коренем** дерева.
- Стоки у дереві називають **листям** дерева.
- Будь-який шлях у дереві називають **гілкою** дерева.
- Будь-яка частина дерева, яка сама є деревом, називається **піддеревом**.
- Вершини, які безпосередньо слідують з даної, називаються **синами** даної вершини, а сама ця вершина – їх **батьком**.
- **Синами** називають також не тільки самі вершини, що безпосередньо слідують з даної, але й піддерева, для яких ці вершини є коренями.
- Будь-які два сини однієї вершини називають **братами**.
- Дерево, яке не містить вершин, називається **порожнім** деревом.
- **Висотою** дерева називають довжину найдовшого шляху (найдовшої гілки) у дереві.

Бінарні дерева

- **Бінарним деревом** називається дерево з напівстепінню виходу всіх вершин не більше 2.
- **Впорядкованим бінарним деревом** називається бінарне дерево, кожна вершина якого завжди має 2 сини: лівий син та правий син, які можуть бути порожніми або непорожніми деревами.
- Далі будемо розглядати тільки впорядковані бінарні дерева.



Операції, відношення та інструкції для бінарних дерев

- Бінарні дерева використовують для пошуку та сортування даних, для представлення інформації, обчислення виразів тощо.
- Визначимо операції, відношення та інструкції для бінарних дерев:

1. Почати роботу.
2. Чи порожнє дерево?
3. Створити дерево.
4. Корінь дерева.
5. Лівий син.
6. Правий син.
7. Змінити корінь дерева.
8. Змінити лівого сина.
9. Змінити правого сина.

- Дії 1, 3, 4, 5, 6 – операції; 2 – відношення, 7, 8, 9 - інструкції.

Операції, відношення та інструкції для бінарних дерев.2

- “Почати роботу” повертає порожнє дерево.
- “Створити дерево” – за двома деревами t1, t2 та даними data створити бінарне дерево з коренем з навантаженням data, лівим сином t1 та правим сином t2.
- “Корінь дерева” повертає навантаження кореня дерева. Дерево при цьому не змінюється. Для порожнього дерева ця операція повинна давати відмову.
- “Лівий син” повертає піддерево, яке є лівим сином дерева. Лівий син порожнього дерева за означенням – порожнє дерево.
- “Правий син” повертає піддерево, яке є правим сином дерева. Правий син порожнього дерева за означенням – порожнє дерево.
- “Змінити корінь дерева” – змінити навантаження кореня дерева значенням data. Якщо дерево порожнє, то після цієї інструкції дерево стає таким, що містить одну вершину.
- “Змінити лівого сина” – змінити значення лівого сина дерева значенням t.
- “Змінити правого сина” - змінити значення правого сина дерева значенням t.

Реалізація бінарного дерева

- Бінарні дерева будемо реалізовувати за допомогою посилань на об’єкти.
- Опишемо клас Btree.
- Згідно з описом, бінарне дерево – це об’єкт з полями, що містять навантаження кореня (_data), лівого сина (_ls) та правого сина (_rs).
- Поля _ls та _rs є об’єктами класу Btree.
- Клас Btree також містить методи, що реалізують описані раніше дії над деревами.

Реалізація бінарного дерева.2

```
class Btree:
    '''Реалізує бінарне дерево.
    '''
    def __init__(self):
        '''Створити порожнє дерево.
        '''
        self._data = None       #навантаження кореня дерева
        self._ls = None         #лівий син
        self._rs = None         #правий син

    def isempty(self):
        '''Чи порожнє дерево?.
        '''
        return self._data == None and self._ls == None and self._rs == None
```

Реалізація бінарного дерева.3

```
def maketree(self, data, t1, t2):
    '''Створити дерево.

    Дані у корені - data, лівий син - t1, правий син - t2
    '''
    self._data = data
    self._ls = t1
    self._rs = t2

def root(self):
    '''Корінь дерева.
    '''
    if self.isempty():
        print('root: Дерево порожнє')
        exit(1)
    return self._data
```

Реалізація бінарного дерева.4

```
def leftson(self):
    '''Лівий син.
    '''
    if self.isempty():
        t = self
    else:
        t = self._ls
    return t

def rightson(self):
    '''Правий син.
    '''
    if self.isempty():
        t = self
    else:
        t = self._rs
    return t
```

Реалізація бінарного дерева.5

```
    def updateroot(self, data):
        '''Змінити корінь значенням data.
        '''
        if self.isempty(): #якщо дерево порожнє, додати лівого та правого сина
            self._ls = Btree()
            self._rs = Btree()
        self._data = data

    def updateleft(self, t):
        '''Змінити лівого сина значенням t.
        '''
        self._ls = t

    def updateright(self, t):
        '''Змінити правого сина значенням t.
        '''
        self._rs = t
```

Приклад. Бінарне дерево пошуку

- Бінарним деревом пошуку називається таке бінарне дерево, у якому навантаження кореня більше навантаження будь-якої вершини, що належить лівому сину, та менше навантаження будь-якої вершини, що належить правому сину.
- Побудувати бінарне дерево пошуку за списком рядків та перевірити, чи входять у дерево задані рядки.

- Програма, що розв’язує цю задачу, містить функції побудови списку слів, побудови дерева пошуку за списком слів, пошуку у дереві.
- Побудова дерева пошуку та пошук у дереві використовують одну внутрішню функцію, яка шукає місце, куди можна вставити нове слово w, а також перевіряє, чи є w у дереві.

Графи

- Граф будемо представляти як список вершин.
- При цьому, кожна вершина має унікальний ключ а також навантаження.
- Крім того, для кожної вершини визначено список попередників (вершин, з яких безпосередньо слідує дана вершина) та список наступників (вершин, які безпосередньо слідують з даної).

Операції, відношення та інструкції для графів

1. Створити порожній граф
2. Вершини графа
3. Довжина графа
4. Повернути вершину
5. Повернути дані вершини
6. Повернути список попередників
7. Повернути список наступників
8. Оновити дані вершини
9. Оновити список попередників
10. Оновити список наступників
11. Видалити вершину
12. Оновити (додати) вершину

- Дії 2, 3, 4, 5, 6, 7 – операції;1, 8, 9, 10, 11, 12 - інструкції.

Операції, відношення та інструкції для графів.2

- “Створити порожній граф” повертає порожній граф, що не містить вершин.
- “Вершини графу” повертає список вершин графу (ключів вершин).
- “Довжина графу” повертає кількість вершин у графі.
- “Повернути вершину” повертає вершину графу з заданим ключем.
- “Повернути список попередників” повертає список вершин, з яких безпосередньо слідує вершина з заданим ключем.
- “Повернути список наступників” повертає список вершин, які безпосередньо слідують із вершини з заданим ключем.
- “Оновити дані вершини” – змінити навантаження вершини з заданим ключем значенням data. Якщо такої вершини немає, - відмова.
- “Оновити список попередників” – – змінити список попередників вершини з заданим ключем значенням lst. Якщо такої вершини немає, - відмова..
- “Оновити список наступників” – – змінити список наступників вершини з заданим ключем значенням lst. Якщо такої вершини немає, - відмова..
- “Видалити вершину” – видалити вершину графа з заданим ключем.
- “Оновити (додати) вершину” – оновити або додати (якщо не існує) вершину графа з заданим ключем.

Реалізація графа

- Реалізуємо граф на базі словника.
- Опишемо клас Graph, який містить поле _dct – словник, у якому зберігаються вершини графа, а також методи, що реалізують дії над графом.
- Кожна вершина графа має унікальний ідентифікатор key, а також кортеж (data, predecessors, succeders),
  - де
  - data - дані вершини (навантаження)
  - predecessors - список попередників
  - succeders - список наступників
- У зв’язку з цим, при додаванні, видаленні вершини, зміні списків попередників та наступників треба видалити (або додати) посилання на вершину у списках наступників усіх попередників та у списках попередників усіх наступників цієї вершини.

Реалізація графа.2

```
class Graph:
    '''Реалізує орієнтований граф на базі словника.

    Кожна вершина графу має унікальний ідентифікатор key,
а також трійку     (data, predecessors, succeders),
    де
    data          дані вершини
    predecessors  список попередників
    succeders     список наступників
    '''
    def __init__(self):
        '''Створити порожній граф.
        '''
        self._dct = {}      #_dct - словник, що містить вершини графу

    def nodes(self):
        '''Вершини графу.
        '''
        return self._dct.keys()
```

Реалізація графа.3

```
def __len__(self):
    '''Довжина графу, реалізує len(g).
    '''
    return len(self._dct)

def __getitem__(self, key):
    '''Повернути вершину, реалізує g[key].

    Якщо вершини key немає у графі, повертає None.
    '''
    if key in self._dct:
        value = self._dct[key]
    else:
        value = None
    return value
```

Реалізація графа.4

```
def getdata(self, key):
    '''Повернути дані вершини.

    Якщо вершини key немає у графі, повертає None.
    '''
    if key in self._dct:
        data = self._dct[key][0]
    else:
        data = None
    return data

def getpredecessors(self, key):
    '''Повернути список попередників вершини.

    Якщо вершини key немає у графі, повертає None.
    '''
    if key in self._dct:
        lst = self._dct[key][1]
    else:
        lst = None
    return lst
```

Реалізація графа.5

```
def getsucceders(self, key):
        '''Повернути список наступників вершини.

        Якщо вершини key немає у графі, повертає None. '''
        if key in self._dct:
            lst = self._dct[key][2]
        else:
            lst = None
        return lst

    def setdata(self, key, data):
        '''Оновити дані вершини key значенням data.

        Якщо вершини key немає у графі, видає помилку.'''
        if key in self._dct:
            dt, lp, ls = self._dct[key]     #повертає дані, списки попередників та наступників
            self._dct[key] = (data, lp, ls) #встановлює нове значення вершини з даними data
        else:
            print('setdata: немає вершини', key)
            exit(1)
```

Реалізація графа.6

```
    def setpredecessors(self, key, lst):
        '''Оновити список попередників вершини key значенням lst.

        Якщо вершини key немає у графі, видає помилку.
        '''
        if key in self._dct:
            dt, lp, ls = self._dct[key]     #повертає дані, списки попередників та наступників
            self._removeinpred(key)         #видаляє посилання на вершину key в усіх списках наступників старих попередників вершини
            self._dct[key] = (dt, lst, ls)  #встановлює нове значення вершини з новим списком попередників lst
            self._addinpred(key)            #вставляє посилання на вершину key в усіх списках наступників нових попередників вершини
        else:
            print('setpredecessors: немає вершини', key)
            exit(1)
```

Реалізація графа.7

```
def setsucceders(self, key, lst):
    '''Оновити список наступників вершини key значенням lst.
    Якщо вершини key немає у графі, видає помилку. '''
    if key in self._dct:
        dt, lp, ls = self._dct[key]     #повертає дані, списки попередників та наступників
        self._removeinsucc(key)         #видаляє посилання на вершину key в усіх списках попередників старих наступників вершини
        self._dct[key] = (dt, lp, lst)  #встановлює нове значення вершини з новим списком наступників lst
        self._addinsucc(key)            #вставляє посилання на вершину key в усіх списках попередників нових наступників вершини
    else:
        print('setsucceders: немає вершини', key)
        exit(1)

def _removeinpred(self, key):
    '''Видалити вершину key із списків наступників усіх попередників вершини. '''
    if key in self._dct:
        p = self.getpredecessors(key)   #p - список попередників вершини key
        for k in p:
            lst = self._dct[k][2]       #lst - список наступників вершини k
            lst.remove(key)
```

Реалізація графа.8

```
    def _removeinsucc(self, key):
        '''Видалити вершину key із списків попередників усіх наступників вершини. '''
        if key in self._dct:
            p = self.getsucceders(key)      #p - список наступників вершини key
            for k in p:
                lst = self._dct[k][1]       #lst - список попередників вершини k
                lst.remove(key)

    def _addinpred(self, key):
        '''Додати вершину key до списків наступників усіх попередників вершини. '''
        if key in self._dct:
            p = self.getpredecessors(key)   #p - список попередників вершини key
            for k in p:
                lst = self._dct[k][2]       #lst - список наступників вершини k
                lst.append(key)
```

Реалізація графа.9

```
    def _addinsucc(self, key):
        '''Додатии вершину key до списків попередників усіх наступників вершини. '''
        if key in self._dct:
            p = self.getsucceders(key)      #p - список наступників вершини key
            for k in p:
                lst = self._dct[k][1]       #lst - список попередників вершини k
                lst.append(key)

    def __delitem__(self, key):
        '''Видалити вершину графа key (del x[key]).
           Якщо вершини key немає у графі, видає помилку. '''
        if key in self._dct:
            self._removeinpred(key) #видаляє посилання на вершину key в усіх списках наступників попередників вершини
            self._removeinsucc(key) #видаляє посилання на вершину key в усіх списках попередників наступників вершини
            del self._dct[key]      #видаляє вершину з словника
        else:
            print('__delitem__: немає вершини', key)
            exit(1)
```

Реалізація графа.10

```
    def _addnode(self, key, value):
        '''Додати вершину графа key.

        Якщо вершини key немає у графі, видає помилку.
        '''
        if not key in self._dct:
            self._dct[key] = value  #додає вершину до словника
            self._addinpred(key)    #вставляє посилання на
вершину key в усіх списках наступників попередників
вершини
            self._addinsucc(key)    #вставляє посилання на
вершину key в усіх списках попередників наступників
вершини
        else:
            print('_addnode: вже є вершина', key)
            exit(1)
```

Реалізація графа.11

```
def __setitem__(self, key, value):
        '''Оновити (додати) вершину x[key] = value.

        Якщо вершини key немає у графі, додає її.
        '''
        if not isinstance(value,tuple) or len(value) != 3 \
           or not isinstance(value[1], list)or not
isinstance(value[2], list): #перевірити, чи правильно
передані параметри
            print('x[key] = value: value must be tuple of 3' \
                  ' with lists on second and third place')
            exit(1)
        if key in self._dct:        #якщо вершина key є у графі
            self.__delitem__(key)   #спочатку видалити її
        self._addnode(key, value)   #додати вершину до
графу з новим значенням value
```

Перевизначення операцій

- Якщо звернути увагу на назви методів класу Graph, - зможемо побачити декілька методів які починаються та закінчуються двома підкресленнями ‘__’.
- Ми вже зустрічались з такими методами: конструктором __init__ та деструктором __del__.
- Такі методи називають особливими або магічними.
- Ми не викликаємо їх напряму. Натомість Python викликає ці методи у певних ситуаціях. Так, конструктор викликається під час створення об’єкту, а деструктор, - під час його знищення.
- За допомогою особливих методів у Python можна перевизначити для власного класу практично всі стандартні операції.
- Наприклад, щоб перевизначити для класу операцію ‘+’, треба описати у класі реалізацію метода __add__. Тоді для двох об’єктів цього класу x та y Python буде трактувати x + y як x.__add__(y).
- Неповний перелік операцій для перевизначення та відповідних особливих методів наведено у таблицях нижче.

Перевизначення операцій. Бінарні операції

Операція	Метод
x + other	x.__add__(other)
x - other	x.__sub__(other)
x * other	x.__mul__(other)
x // other	x.__floordiv__(other)
x / other	x.__div__(other)
x % other	x.__mod__(other)
x ** other	x.__pow__(other)

Перевизначення операцій. Присвоєння спеціального виду

Операція	Метод
x += other	x.__iadd__(other)
x -= other	x.__isub__(other)
x *= other	x.__imul__(other)
x /= other	x.__idiv__(other)
x //= other	x.__ifloordiv__(other)
x %= other	x.__imod__(other)
x **= other	x.__ipow__(other)

Перевизначення операцій. Унарні операції

Операція	Метод
- x	x.__neg__()
+ x	x.__pos__()
len(x)	x.__len__()
abs(x)	x.__abs__()
complex(x)	x.__complex__()
int(x)	x.__int__()
long(x)	x.__long__()
float(x)	x.__float__()

Перевизначення операцій. Відношення

Операція	Метод
x < other	x.__lt__(other)
x <= other	x.__le__(other)
x == other	x.__eq__(other)
X != other	x.__ne__(other)
x >= other	x.__ge__(other)
x > other	x.__gt__(other)

Перевизначення операцій. Дії над послідовностями та словниками

Дія	Метод
x[key]	x.__getitem__(key)
x[key] = value	x.__setitem__(key, value)
del x[key]	x.__delitem__(key)
x(other)	x.__call__(other)

- У реалізації графу ми перевизначили три останніх дії а також функцію обчислення довжини len().

Приклад. Перевірка графа на зв'язність

- Дано орієнтовний граф. Треба перевірити чи є він зв'язним.
- Для розв'язання цієї задачі опишемо функції введення графу з текстового файлу спеціального вигляду та власне перевірки графа на зв'язність.
- Для перевірки на зв'язність виберемо довільну (першу) вершину та додамо її у множину досягнутих вершин.
- Утворимо також порожню множину вершин, які переглянуто.
- Далі будемо послідовно для кожної вершини, яку не переглянуто, додавати до множини досягнутих вершин всіх попередників та наступників даної вершини, а саму вершину, - до множини вершин, які переглянуто.
- Так будемо повторювати, поки у множині досягнутих вершин залишаються вершини, які не переглянуто.
- Якщо в результаті множина досягнутих вершин буде рівною множині всіх вершин, то граф є зв'язним.
- Якщо граф не є зв'язним, то щоб отримати розбиття, про яке йшла мова у означенні незв'язності графа, можна просто обчислити різницю множини всіх вершин та множини досягнутих вершин.

Резюме

- Ми розглянули:
  1. Статичні та динамічні структури даних. Рекурсивні структури даних
  2. Стеки деки та черги.
  3. Реалізацію стеку та черги на базі списку.
  4. Реалізацію деку на базі посилань на об’єкти
  5. Списки. Реалізацію кільцевого списку.
  6. Графи та дерева
  7. Реалізацію бінарного дерева на базі посилань на об’єкти
  8. Реалізацію графа на базі словника
  9. Перевизначення операцій

Де прочитати

1. Бублик В.В., Личман В.В., Обвінцев О.В.. Інформатика та програмування. Електронний конспект лекцій, 2003 р.,
2. Марк Лутц, Изучаем Python, 4-е издание, 2010, Символ-Плюс
3. Python 3.4.3 documentation
4. Bruno R. Preiss, Data Structures and Algorithms with Object-Oriented Design Patterns in Python, 2003, http://www.brpreiss.com/books/opus7/
5. http://www.python-course.eu/graphs_python.php
6. http://www.python-course.eu/python3_magic_methods.php
7. http://www.programiz.com/python-programming/operator-overloading