ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

ALGORITMUSOK ÉS ADATSZERKEZETEK

Python 5. hét

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

FÉLÉVES TEMATIKA

Oktatási hét	Témakör - Gyakorlat
1	Bevezetés a Python programozási nyelvbe, alapvető szintaxis és kifejezések, változók, típusok és alapvető műveletek, alapvető be- és kimenet, feltételes elágazások (if, elif, else)
2	Ciklusok (for, while), listák és egyszerű iterációk
3	Függvények és modulok, függvények definiálása és hívása, paraméterek és visszatérési értékek, beépített modulok használata, fájlkezelés, kivételkezelés
4	Algoritmusok és optimalizálási stratégiák rekurziós példán keresztül (brute force, dinamikus programozás, mohó algoritmusok, visszalépéses keresés)
5	Adatszerkezetek I. - Lista, tömb, sor, verem, listaműveletek, queue és stack implementációja statikusan (tömb) és dinamikusan (lista), gyakorlati feladatok
6	Adatszerkezetek II. – Halmaz és szótár, halmazműveletek és szótárműveletek, gyakorlati feladatok
7	**Zárthelyi dolgozat 1.**
8	Adatszerkezetek III. – Láncolt lista implementálása
9	Adatszerkezetek IV. – Bináris keresőfa implementálása
10	Gráfok alapjai, reprezentációik, gráfalgoritmusok implementálása
11	*Rektori szünet*
12	***Féléves beadandó feladat leadása és bemutatása***
13	**Zárthelyi dolgozat 2.**
14	**Félév zárása, javító/pótló zárthelyi dolgozat**

Algoritmusok és optimalizálási stratégiák

mohó algoritmus

MOHÓ ALGORITMUS

- Egy olyan megközelítés, amely egy probléma megoldása során mindig a pillanatnyi legjobb, optimális választást hajtja végre *(a helyi optimalizáció a végén globális optimalizációhoz vezethet).*
- A mohó algoritmus **lépésről lépésre hoz döntéseket**, és minden lépésben a legjobbnak tűnő (mohó) lehetőséget választja.
- Jellemzői:
  - **Lokális optimalizáció:** Minden lépésben az éppen elérhető legjobb lehetőséget választja.
  - **Heurisztikus:** Nem mindig garantálja az optimális megoldást, de gyorsan működik.
  - **Egyszerűség:** Az implementáció általában egyszerű, mivel nincs szükség a teljes megoldási tér vizsgálatára.

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

HÁTIZSÁK PROBLÉMA – MOHÓ ALGORITMUSSAL

- Oldjuk meg a klasszikus hátizsákproblémát egy egyszerű mohó algoritmussal.

- **Ne feledjük:**
  - A mohó algoritmus mindig a lokálisan legjobb döntést hozza.
  - Ebben az esetben mindig a legnagyobb érték/súly arányú tárgyat válasszuk, amíg van hely a hátizsákban.

PYTHON_04_02.PY

A feladat:

- Képzeld el, hogy egy automatából szeretnél pénzt váltani. Az automata csak bizonyos értékű érméket ad ki (1000, 500, 200, 100, 50, 20, 10, 5, 2, 1 forintos).
- Írj egy programot, ami megmondja, hogy milyen érméket és mennyi darabot kell kiadnia az automatának, hogy pontosan az általad megadott összeget kapd vissza.
- **Segítség (import math – math.floor() használata):**
  - *Az automata mindig a legnagyobb értékű érmével kezdi, amit még ki tud adni.*
  - *Amíg a visszaváltandó összeg nagyobb, mint az aktuális érme értéke, addig mindig az adott érmét adja ki.*
  - *Ha már nem fér bele a nagyobb érme, akkor a következő legnagyobb értékűre vált.*
  - *Amikor az összeg nulla lesz, a program kiírja, hogy milyen érmékből és mennyi darabból állt a visszaváltás.*
- *Egészítsd ki a kódot a leírás alapján.*

Lista

listaműveletek

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

LISTA [ ] – EMLÉKEZTETŐ

- Sokoldalú eszközök az adatok tárolására és kezelésére.
- Dinamikus adattárolást tesz lehetővé.
- Számos beépített függvény áll rendelkezésre a listák manipulálására.
- **Néhány fontosabb beépített függvény:**
  - **len**(list): Visszaadja a lista elemeinek számát.
  - list.**sort**(): A listát növekvő sorrendbe rendezi.
  - list.**reverse**(): Megfordítja a lista elemeinek sorrendjét.
  - list.**append**(elem): Hozzáad egy elemet a lista végéhez.
  - list.**insert**(index, elem): Beszúr egy elemet egy adott indexű pozícióra.
  - list.**remove**(elem): Törli az első olyan elemet a listából, amely megegyezik az adott értékkel.
  - list.**pop**(index): Törli és visszaadja az adott indexű elemet.

PYTHON_04_03.PY

```python
# Hozz létre egy listát a kedvenc gyümölcseiddel!
gyumolcsok = []

# Kérj be egy gyümölcs nevét a felhasználótól, és ellenőrizd, hogy benne van-e a listában!
print("Írj be egy gyümölcsnevet: ")
benne_van = False
print("Benne van." if benne_van else "Nincs benne.")

# Másold le a listát egy másik listába!
# Adj hozzá egy elemet az új listához, amit a felhasználó ír be! Nézd meg, változott-e az eredeti lista!
# Hogy lehet elérni, hogy az eredeti lista változzon/ne változzon?

# Rendezd a listát aszerint csökkenő sorrendbe, hogy hányszor szerepel benne az a betű!

# Készíts egy listát, ami tartalmazza a gyümölcs lista és az alábbi lista metszetét!
tobb_gyumolcs = ["szőlő", "barack", "alma", "banán", "cseresznye", "citrom", "ananász", "szilva"]

# Fűzd össze a két listát!

# Távolítsd el a listából a duplikált elemeket!

# Rendezd a listát, majd fűzd össze egy sztringgé a következő elválasztóval.
koz = " -- "
```

Lista

mint sor és verem

SOROK (QUEUE)

- **FIFO** (First In First Out) **elv**
- Egy sorban az első elem kerül hozzáadásra, és az első elem kerül eltávolításra.
- Pythonban a listákat a következőképpen használhatjuk sorokként:
  - **Hozzáadás** (enqueue): Az .**append**() metódus segítségével adhatunk hozzá elemet a lista végéhez.
  - **Eltávolítás** (dequeue): A .**pop**(0) metódus segítségével távolíthatjuk el az első elemet a listából.

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

VEREM (STACK)

- Egy veremben az elemek úgy kerülnek hozzáadásra és eltávolításra, mint egy könyvkupac: az utolsó elem kerül hozzáadásra, és az utolsó elem kerül eltávolításra.
- Ezt a viselkedést **LIFO** (Last In, First Out) **elvnek** nevezzük.
- Pythonban a listákat a következőképpen használhatjuk vermeként:
  - **Hozzáadás** (push): Az **.append**() metódus segítségével adhatunk hozzá elemet a lista végéhez.
  - **Eltávolítás** (pop): A **.pop**() metódus segítségével távolíthatjuk el az utolsó elemet a listából.

PYTHON_04_04.PY

```python
# Pythonban a listát tudjuk használni sorként és veremként is.
# Sor (Queue) beépített listával
queue = []

# Adjunk hozzá elemeket a sorhoz! (enqueue) - 1,2,3 értékeket
print("Sor:", queue)  # [1, 2, 3]

# Távolítsunk el egy elemet a sor elejéről! (pop)
dequeued_item =
print("Kivett elem:", dequeued_item)  # 1
print("A sor kivétel után:", queue)  # [2, 3]

# Írjuk ki az első elemet! (peek)
print("Első elem:")
```

PYTHON_04_04.PY

```
15
16  # Verem (Stack) beépített listával
17  stack = []
18
19  # Adjunk hozzá elemeket a veremhez (append) - 1,2,3
20  print("Verem:", stack)  # [1, 2, 3]
21
22  # Távolítsunk egy elemet a verem tetejéről (pop)
23  popped_item =
24  print("Kivett elem:", popped_item)  # 3
25  print("Verem kivétel után:", stack)  # [1, 2]
26
27  # Nézzük meg a verem legfelső elemét (peek)
28  print("Következő elem:")  # 2
```

Dinamikus és statikus megvalósítás

sor

DINAMIKUS SOR ADATSTRUKTÚRA IMPLEMENTÁLÁSA

A Pythonban a listák segítségével egyszerűen és hatékonyan implementálhatunk dinamikus sorokat és vermeket. Ezek az adatstruktúrák számos algoritmus és alkalmazás alapját képezik, és a rugalmasságuk miatt széles körben használhatók.

Mire használható?

- Feladatütemezés:
  - A feladatokat egy sorba helyezhetjük, és a rendszer mindig az első feladatot dolgozza fel.
- Adatfeldolgozás:
  - Ha nagy mennyiségű adatot kell feldolgozni, sorba állíthatjuk az adatokat, és egyenként dolgozhatunk fel rajtuk.
- Szimulációk:
  - Szimulálhatunk olyan rendszereket, ahol az események sorrendben történnek (pl. egy üzenetsor).
- Operációs rendszerek:
  - Az operációs rendszerekben is számos helyen találkozhatunk sorokkal (pl. nyomtatási sorok, hálózati csomagok sorozata).

_PYTHON_04_05_SOR_DYNAMIC.PY

```python
class DynamicQueue: # a dinamikus sor adatstruktúra reprezentálása  2 usages
    def __init__(self):
        self.queue = [] # inicializálunk egy üres listát, amely a sor elemeit fogja tárolni

    def __len__(self):
        # visszaadja a self.queue lista hosszát, ami a sorban lévő elemek számát jelenti
        return len(self.queue)

    # Ez a metódus hozzáad egy elemet a sor végéhez
    def enqueue(self, item):  9 usages
        # A self.queue.append(item) hozzáfűzi az item elemet a self.queue listához
        self.queue.append(item)

    # a metódus eltávolítja és visszaadja a sor elejéről az első elemet
    def dequeue(self):  4 usages
        # ellenőrzi, hogy a sor üres-e
        # Ha igen, kivételt dob, mivel nem lehet eltávolítani elemet egy üres sorból
        if not self.queue:
            raise Exception("Queue is empty")
        # Ha a sor nem üres, a self.queue.pop(0) eltávolítja és visszaadja a lista első elemét
        return self.queue.pop(0)
```

_PYTHON_04_05_SOR_DYNAMIC.PY

```
22
23      # visszaadja a sor elejéről az első elemet anélkül, hogy eltávolítaná
24      def peek(self):  1 usage
25          # ellenőrzi, hogy a sor üres-e
26          # Ha igen, kivételt dob
27          if not self.queue:
28              raise Exception("Queue is empty")
29          # Ha a sor nem üres, a self.queue[0] visszaadja a lista első elemét
30          return self.queue[0]
31
32      def is_empty(self): # ellenőrzi, hogy a sor üres-e.
33          return len(self.queue) == 0  # True, ha a sor üres
34
35      # speciális metódus - a print() függvénnyel kiírjuk a sor tartalmát
36      def __str__(self):
37          return str(self.queue) # visszaadja a self.queue lista szöveges reprezentációját
38
```

_PYTHON_04_05_SOR_DYNAMIC.PY

```
39
40  # Példa használat
41  dynamic_queue = DynamicQueue() # objektum létrehozása
42
43  dynamic_queue.enqueue(1) # elemeket adunk a sorhoz
44  dynamic_queue.enqueue(2)
45  dynamic_queue.enqueue(3)
46  dynamic_queue.enqueue(4)
47  dynamic_queue.enqueue(5)
48  print(dynamic_queue)  # [1, 2, 3, 4, 5]
49
50  print(dynamic_queue.dequeue())  # 1 - elem eltávolítása
51  dynamic_queue.enqueue(6)
52  dynamic_queue.enqueue(7)
53
54  print(dynamic_queue.peek())  # 2 - első elem lekérdezése
55  print(dynamic_queue.dequeue())  # 2
56  print(dynamic_queue.dequeue())  # 3
57  print(dynamic_queue)  # [4, 5, 6]
58
```

```
[1, 2, 3, 4, 5]
1
2
2
3
[4, 5, 6, 7]
```

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

STATIKUS VEREM/SOR ADATSTRUKTÚRA IMPLEMENTÁLÁSA

- Előre definiált, fix méretű adatszerkezet.
- A létrehozáskor megadott méretnél nem lehet több elemet tárolni.
- Általában tömbökkel implementálják.
- Pythonban **miért nem szokás** statikus vermet/sort implementálni?
  - **Dinamikus típusozás**: dinamikusan típusozott nyelv, ami azt jelenti, hogy a változók típusa futásidőben kerül meghatározásra.
  - **Beépített listák**: listái alapvetően dinamikus tömbök, amelyek rugalmasak és könnyen használhatók.
  - **Memóriakezelés**: automatikusan kezeli a memóriát, így nem kell aggódnunk a memória allokációja és felszabadítása miatt.

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

_PYTHON_04_05_VEREM_STATIC.PY

```python
class StaticStack:  3 usages
    def __init__(self, capacity):
        self.capacity = capacity # Beállítja a verem kapacitását (maximum tárolható elemek száma)
        # Létrehoz egy listát a verem elemeinek tárolására
        # A lista elemei kezdetben None értékkel inicializálódnak
        self.stack = [None] * capacity
        # A verem tetejét jelző index kezdeti értéke -1
        # Mivel a verem még üres, a tetejére mutató index -1-re van állítva
        self.top = -1

    def __len__(self):
        # a lista indexelése 0-tól kezdődik, ezért 1-et adunk hozzá,
        # hogy megkapjuk a veremben lévő elemek tényleges számát
        return self.top + 1
```

_PYTHON_04_05_VEREM_STATIC.PY

```
15
16      def push(self, item):  8 usages
17          # Ellenőrzi, hogy a verem tele van-e
18          if self.top == self.capacity - 1:
19              raise Exception("Stack is full")
20          self.top += 1
21          self.stack[self.top] = item # hozzáadja az elemet a verem tetejére
22
23      # eltávolítja és visszaadja az elemet a verem tetejéről (LIFO)
24      def pop(self):  2 usages
25          if self.top == -1: # Ellenőrzi, hogy a verem üres-e
26              raise Exception("Stack is empty")
27          item = self.stack[self.top] # Lementi az eltávolítandó elemet egy változóba
28          self.stack[self.top] = None # memóriahasználat optimalizálás
29          self.top -= 1 # top-1, a verem új legfelső elemére mutat
30          return item # Visszaadja az eltávolított elemet
```

_PYTHON_04_05_VEREM_STATIC.PY

```
31
32      # megvizsgáljuk a verem tetején lévő elemet anélkül, hogy eltávolítanánk
33      def peek(self):  1 usage
34          if self.top == -1:
35              raise Exception("Stack is empty")
36          return self.stack[self.top] # ha nem üres a verem tetején lévő értéket adja vissza
37
38      def is_empty(self): # ha üres a verem akkor True értéket ad vissza
39          return self.top == -1
40
41      def __str__(self): # szöveges megjelenítés
42          # Visszaadja a self.stack lista azon részét,
43          # amely az aktuálisan használt elemeket tartalmazza (0-tól self.top-ig)
44          return str(self.stack[:self.top + 1])
```

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

_PYTHON_04_05_VEREM_STATIC.PY

```
45
46
47  # Példa használat
48  static_stack = StaticStack(5) # létrehozás 5 elemmel
49  static_stack.push(1) # elemek hozzáadása
50  static_stack.push(2)
51  static_stack.push(3)
52  static_stack.push(4)
53  static_stack.push(5)
54  print(static_stack)  # [1, 2, 3, 4, 5]
55  print(static_stack.pop())  # 5 - eltávolítja a felső elemet
56  print(static_stack.peek())  # 4 - megnézi a felső elemet
57  static_stack.push(6) # elem hozzáadása, azután már kivételt dobna, többet nem enged
58  print(static_stack)  # [1, 2, 3, 4, 6]
59  static_stack.push(7) # hiba, kivételkezelés
60  print(static_stack)
```

```
Traceback (most recent call last): Explain with AI
  File "C:\Progs\Python\PyCharm 2024\pythonProject\Python_04\_python_04_05_verem_static.py", line 59, in <module>
    static_stack.push(7)
  File "C:\Progs\Python\PyCharm 2024\pythonProject\Python_04\_python_04_05_verem_static.py", line 19, in push
    raise Exception("Stack is full")
Exception: Stack is full
[1, 2, 3, 4, 5]
5
4
[1, 2, 3, 4, 6]
```

EGYÉB

- **Más implementációk:**
  - A sorokat és vermeket más adatstruktúrákkal is implementálhatjuk (pl. láncolt listák, kettős láncolt listák), de a listák a legegyszerűbb és leggyakrabban használt megoldás Pythonban.
- **Komplexebb adatstruktúrák:**
  - A sorok és vermek alapvető építőelemei lehetnek más, összetettebb adatstruktúráknak, például fákon, gráfokon vagy prioritási sorokon.

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

KÖSZÖNÖM

A MEGTISZTELŐ FIGYELMET!

Módné Takács Judit *modne.t.judit@amk.uni-obuda.hu*