ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

ALGORITMUSOK ÉS ADATSZERKEZETEK

Python 6.

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**

2 fős csapatban - videó feltöltés max 5 perc – kiadott feladat

Láncolt lista implementálása

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

HAGYOMÁNYOS PYTHON-LISTÁK HÁTRÁNYAI

Dinamikus tömbök, amelyek összefüggő memóriablokkban tárolják az elemeket.

- Ha a lista nagyobbra nő, mint a kezdeti mérete, akkor **nagyobb** összefüggő **memóriablokkot** kell kiosztania, és a **meglévő elemeket át kell másolnia**.
- **Elemek beszúrásánál és törlésénél** hasonló a helyzet: az összes következő elemet át kell mozgatni más helyre.

Ez **nem hatékony**, és **időigényes** lehet, különösen nagy listák esetén (**memóriaigényes**!).

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

LÁNCOLT LISTÁK ELŐNYEI

Előnyök:

- **dinamikus méretváltoztatás** és **memóriahatékonyság** (nem kell másolgatni)
- könnyű beszúrás és törlés

A láncolt lista **nem összefüggő memóriahelyeken** tárolja az elemeket, hanem **mutatók segítségével kapcsolja össze** azokat.

Dinamikusan növekedhet vagy zsugorodhat újrafelosztás vagy méretváltoztatás nélkül. Az új elemekhez külön-külön rendel memóriát a bővítéskor.

LÁNCOLT LISTÁK HÁTRÁNYAI

Hátrányok:

- **nincs közvetlen hozzáférés** az adatokhoz (szekvenciális haladás van)
- alapesetben **csak lineáris keresés** lehetséges
- a memóriahasználat adatelemenként magasabb a mutató(k) tárolása miatt
- **bonyolultabb használat**, több hibalehetőség

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

LÁNCOLT LISTÁK HASZNÁLATA
GYAKORLATI ALKALMAZHATÓSÁG

A láncolt listák használata akkor célszerű, ha:

- *sokszor kell* elemet beszúrni, illetve törölni
- a *lista mérete gyakran változik*, vagy kiszámíthatatlan
- az adatelemek közvetlen elérése nem követelmény
- az adatelemek nagy méretűek, összetettek

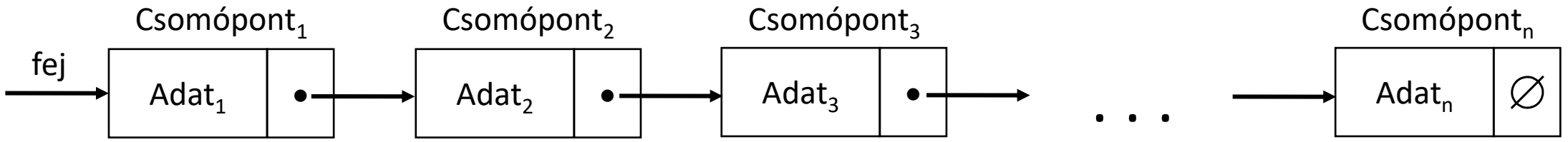
ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

EGYIRÁNYÚ LÁNCOLT LISTA IMPLEMENTÁLÁSA

Az **egyirányú (egyszeresen) láncolt lista** egy **rekurzív adatszerkezet**, önmagára való hivatkozással van definiálva:

- **ahol minden egyes listaelem (Csomópont objektum) tartalmaz egy adatot**
- **és egy referenciát ("mutatót") a következő listaelemre, amely egy ugyanilyen típusú objektum.**

Csak egy irányban - az első csomóponttól (fej) az utolsó csomópontig (vég) - lehet végigmenni rajta. Ha a referencia a következő elemre nem létezik (nem mutat sehova), akkor vagyunk a lista végén. A fej az első elem címét tartalmazza.



ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

NODE (CSOMÓPONT) OSZTÁLY

```
1  class Node:                    # Csomópont osztály
2      def __init__(self,data): # ez a speciális metódus a konstruktor
3          self.data=data       # a self paraméter az osztály aktuális példányára való hivatkozás
4          self.next=None       # a csomóponthoz tartozó adatot a data adattag tartalmazza
5                                # a next adattag a következő csomópont címét tartalmazza, kezdetben
6                                # "nem mutat sehova"
7      def __str__(self):
8          return str(self.data) # a csomópont sztringreprezentációja csak az adatrész
9                                 # sztringreprezentációja
```

A hívásoknál a *self* paramétert **nem kell szerepeltetni**!

A *data* adattagban **bármilyen típusú értéket tárolhatunk**, és az át is adható az *str* függvénynek.

A konstruktor másképp:

```
def __init__(self,data=None,next=None):
    self.data=data
    self.next=next
```

LINKEDLIST OSZTÁLY

```
15  class LinkedList:           # LáncoltLista osztály
16      def __init__(self):     # konstruktor
17          self.head=None      # a lista kezdetben üres, a fej "nem mutat sehova"
18
19      def insertAtBeginning(self,data):   # ElejéreBeszúr művelet
20          new_node=Node(data)             # új csomópont létrehozása a tárolandó adattal
21          new_node.next=self.head         # az új csomópont után következik a teljes eddigi lista
22          self.head=new_node              # az új csomópont lesz a lista új feje
23
24      def printList(self):     # Bejár művelet; az összes adat kiírása
25          tmp=self.head         # tmp a segédcsomópont, amely kezdetben a fej: innen indul a kiírás
26          while tmp:            # egyenértékű: while tmp is not None, vagyis amíg nem érünk a végére
27              print(tmp,end=" ")
28              tmp=tmp.next
29          print()
```

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

FELADAT1 – PYTHON_06_01.PY
LÁNCOLT LISTA KIPRÓBÁLÁSA

Nyissuk meg és egészítsük ki a Python_06_01.py kódot a *#főprogram* megjegyzés után:

- hozzunk létre egy üres listát szavak néven
- szúrjuk be a lista elejére a *lehet* szót
- a lista bejárásával írassuk ki a listát
- szúrjuk be a lista elejére a *könnyű* szót
- szúrjuk be a lista elejére a *Neked* szót
- végül a lista bejárásával írassuk ki a lista elemeit

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

FELADAT2 – PYTHON_06_02.PY

Bővítsük a *LinkedList* osztályunkat, és **írjunk egy rekurzív függvényt**, amely ki tudja írni fordított sorrendben a lista elemeit!

Nyissuk meg, és egészítsük ki a kommenteknek megfelelően a Python_06_02.py kódot, végül teszteljük le!

```
27      def reverseOrder(self): # feltételeznünk kell, hogy véget ér, azaz a lista nem tartalmaz hurkot
28          # leállási feltétel: már nincs több Csomópont, az alapeset az üres lista
29              return
30          # leválasztjuk a fejet
31          tmp_remainder=LinkedList()        # létrehozunk egy segédlistát, amely kezdetben üres
32          # a segédlista fejét a maradék listára állítjuk
33          # kiíratjuk a maradék listát visszafelé
34          print(tmp_head,end=' ')           # kiíratjuk a fejet
```

LINKEDLIST OSZTÁLY (FOLYT. BŐVÍTÉS)

Egy egyirányú lista esetén egy **csomópont beszúrása vagy törlése** akkor hatékony, ha csak a lista elején történik.

Ha pl. a lista végére szeretnénk beszúrni elemet, akkor az elejétől egyesével végig kell lépkedni az elemeken, hogy eljussunk az utolsóhoz, és utána tudjuk fűzni az új elemet.

- Ezért célszerű egy utolsó elemre hivatkozó **végmutató** (*tail*) kezelése is.

Továbbá hasznos lehet egy **számláló** (*counter*), amely mindig az aktuális elemszámot tartalmazza, valamint egy **hibajelző** (*error*), amely azt jelzi, hogy az éppen aktuális műveletnél történt-e valamilyen hiba.

- *Az hibajelző használata kiváltható kivételkezeléssel is, számláló az indexelést helyettesíti - pótolja.*

LINKEDLIST OSZTÁLY

__init__ és insertAtBeginning függvény bővítése

További adattagok: *tail*, *counter*, *error*

```
19  class LinkedList:              # LáncoltLista osztály
20      def __init__(self):        # konstruktor
21          self.head=None         # a lista kezdetben üres, a fej "nem mutat sehova"
22          self.tail=None         # a vég (farok) az utolsó elemre mutat, kezdetben "sehova"
23          self.counter=0         # az elemeket számolja, kezdetben 0
24          self.error=False       # történt-e hiba valamelyik műveletnél
25                                 # (másik lehetőség: kivételkezeléssel)
26
27      def insertAtBeginning(self,data):   # ElejéreBeszúr művelet
28          new_node=Node(data)             # új csomópont létrehozása a tárolandó adattal
29          new_node.next=self.head         # az új csomópont után következik a teljes eddigi lista
30          self.head=new_node              # az új csomópont lesz a lista új feje
31          if self.tail==None:             # ha eddig üres volt a lista, akkor
32              self.tail=self.head         # a tail-t az új, egyetlen elemre kell állítani
33          ++self.counter                  # elemszám növelése 1-gyel
34          self.error=False                # nem történt hiba
```

LINKEDLIST OSZTÁLY

printList függvény módosítása és képernyőtörlés

A *printList* függvény átírásának célja, hogy legyen kicsit „**látványosabb**” az **elemek kiírása**. *Ráadásul korábban az utolsó elem után is kirakott egy szóközt*.

```
36      def printList(self):        # Bejár művelet; az összes adat kiírása
37          tmp=self.head           # tmp a segédcsomópont, amely kezdetben a fej: innen indul a kiírás
38          while tmp:              # egyenértékű: while tmp is not None, vagyis amíg nem érünk a végére
39              print(tmp,end='') # adat kiírása
40              tmp=tmp.next                # következő csomópontra lépés
41              if tmp:                     # ha nem az utolsó csomópont, akkor
42                  print(" --> ",end='')  # kiíratunk utána egy nyilat; így az utolsó után nem lesz
43          print()
```

A kiírás könnyebb olvashatóságáért alkalmazzunk képernyőtörlést! Ehhez szükség van az *os* modul importálására: `1  import os  # csak a képernyőtörlés miatt`

A képernyőtörlés: `48  os.system("cls")  # ez így csak Windows alatt jó`

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

FELADAT3 – PYTHON_06_03.PY

Nyissuk meg, és egészítsük ki a Python_06_03.py kódot az előző két diakockán szereplő újdonságokkal (a kommentek is segítenek):

- az __init__ és az insertAtBeginning függvény bővítése
- a printList függvény módosítása
- képernyőtörlés (importálás és bővítés)

Próbáljuk is ki!

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

LINKEDLIST OSZTÁLY
insertAtEnd függvény – további bővítés

Először nézzük az *insertAtEnd* (*VégéreBeszúr*) függvényt, amelynél nagyon hasznos lesz a végmutató (*tail*). Az üres lista esete külön kezelendő!

```
37      def insertAtEnd(self,data):      # VégéreBeszúr művelet
38          new_node=Node(data)          # új csomópont létrehozása a tárolandó adattal
39          if self.head==None:          # ha üres volt a lista, akkor
40              self.head=new_node       # létrehozunk egy új fejet és véget (a most létrehozott
41              self.tail=self.head      # csomópontra mutatnak)
42          else:                        # ekkor nem üres a lista, ezért a vége után
43              self.tail.next=new_node # kell fűzni az új elemet: a korábbi vég(csomópont) az új
44              self.tail=new_node       # csomópontra mutat, és az új csomópont lesz a lista vége
45          ++self.counter               # elemszám növelése 1-gyel
46          self.error=False             # nem történt hiba
```

LINKEDLIST OSZTÁLY
insertAtPosition függvény lényege

Az *insertAtPosition* (*HelyreBeszúr*) függvény, amely az **n. pozícióra** szúrja be az elemet.

- **Ha a lista üres**, vagy a megadott pozíció legfeljebb 1, akkor a lista elejére fog beszúrni.
- **Ha a pozíció nagyobb**, **mint a lista elemszáma**, akkor pedig a lista végére.

Alternatíva lehetne az *insertAtIndex* függvény, amely index-edik elemként szúrja be az elemet (0-tól indulva).

- *Mindkét függvény megírható úgy is, hogy hiba keletkezzen akkor, ha túl kicsi, vagy túl nagy a pozíció/index.*

LINKEDLIST OSZTÁLY
insertAtPosition függvény kódja

```
48      def insertAtPosition(self,data,n): # HelyreBeszúr művelet: az n. pozícióra szúrja be az elemet
49          new_node=Node(data)
50          if self.head==None or n<=1:    # ha a lista üres vagy a pozíció legfeljebb 1, akkor az
51              new_node.next=self.head    # első helyre szúrja be
52              self.head=new_node
53          else:                          # tmp a segédcsomópont (temporary:ideiglenes)
54              tmp=self.head              # az i mutatja, hogy éppen hányadik pozícióra lehetne
55              i=2                        # beszúrni, vagyis a tmp mindig az (i-1). csomópontra mutat
56              while tmp.next and i<n:    # itt az n legalább 2 (és a listának van legalább 1 eleme)
57                  tmp=tmp.next
58                  i+=1
59              new_node.next=tmp.next     # a ciklus végeztével a tmp után kell beilleszteni az újat
60              tmp.next=new_node          # ha n>=(lista elemszáma+1), akkor is jó: a végére illeszti
61          ++self.counter
62          self.error=False
```

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

FELADAT4 – PYTHON_06_04.PY

Nyissuk meg, és teszteljük a Python_06_04.py kódot a *#főprogram* résznél szereplő kommenteket felhasználva!

LINKEDLIST OSZTÁLY

removeLastNode, removeFirstNode függvény lényege

A következő két függvény a *removeLastNode* (*UtolsótTöröl*) és a *removeFirstNode* (*ElsőtTöröl*).

- *Az eddigiektől eltérően itt már előfordulhat hiba: ha üres listából szeretnénk törölni egy csomópontot.*
- *Meggondolandó az is, hogy mi történik, ha a csomópont törlése után lesz üres a lista.*

Fontos – bár mi ezzel nem foglalkozunk –, hogy a törölt csomópont memóriaterületének felszabadítása megtörténik-e.

LINKEDLIST OSZTÁLY
removeLastNode függvény kódja

```
68      def removeLastNode(self):           # UtolsótTöröl művelet
69          if self.head==None:             # üres listából nem lehet elemet törölni
70              self.error=True
71          else:
72              self.counter-=1             # itt csökkentjük az elemszámot, mert ha
73              if self.counter==0:         # 1 elemű volt a lista, akkor a törlés után üres lista lesz
74                  self.head=None
75                  self.tail=None
76              else:
77                  tmp=self.head
78                  while tmp.next!=self.tail: # a tmp a ciklus végén az utolsó előtti csomópont lesz
79                      tmp=tmp.next
80                  tmp.next=None
81                  self.tail=tmp
82              self.error=False
```

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

FELADAT5 – PYTHON_06_05.PY

Nyissuk meg a Python_06_05.py kódot, és készítsük el *removeFirstNode* függvényt a kommenteket felhasználva!

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

FELADAT6 – PYTHON_06_06.PY

Nyissuk meg, és teszteljük a Python_06_06.py kódot a *#főprogram* résznél szereplő kommenteket felhasználva!

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

TOVÁBBI MŰVELETEK

Még számos művelet megvalósítható (***beadandónak pl***):

- Üres, Üres_e, Hibás_e, IndexnélMódosít, CsomópontUtánTöröl, MegadottElemetTöröl, TeljesListaTöröl stb.
- Bevezetve egy **akt** mutatót, amely mindig az aktuális listaelemre mutat, lehetne:
  - AktTöröl, AktMódosít, AktÉrték, Elsőre, Következőre (az utóbbi 2 mozgatja az akt mutatót), Utolsó_e stb.
- Létrehozható ún. **ciklikus lista** is, ha az utolsó csomópont mutatója visszamutat a legelsőre:
  - self.tail.next=self.head

GYAKORLÓ FELADATOK

1. feladat

Valósítsd meg a *Megszámolás* programozási tételt, azaz bővítsd a *LinkedList* osztályt egy *Counting* függvénnyel, amely megszámolja (majd visszaadja), hogy egy paraméterként kapott adat hányszor szerepel a listában!

2. feladat

Írd át úgy az *insertAtPosition* függvényt, hogy használja az *insertAtBeginning* függvényt!

3. feladat

Írd meg az *insertAtIndex* függvényt hibakezeléssel együtt!

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

KÖSZÖNÖM

A MEGTISZTELŐ FIGYELMET!

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