Kőrösi Gábor

# Algoritmusok és adatszerkezetek a gyakorlatban

Jelen tananyag a Szegedi Tudományegyetemen készült az Európai Unió támogatásával.

Projekt azonosító: EFOP-3.4.3-16-2016-00014



Szegedi Tudományegyetem
Cím: 6720 Szeged, Dugonics tér 13.
www.u-szeged.hu
www.szechenyi2020.hu

# Elemi adatszerkezetek

## Összefoglaló

Ahogyan azt az előző fejezetekben már láthattuk, különböző módszerekkel hatékonyabbá tehetjük az egyes algoritmusok futási költségeit (memóriaigény, processzoridő stb.). Azonban azt is észrevehettük, hogy több esetben is a megoldás felé vezető úton (a tömbök folyamatos pakolásával), ideiglenes változók hadát kellett felhasználnunk. Ennek elkerüléséhez az elemi adatszerkezeteket célszerű használni. A soron következő tananyagrészben megvizsgáljuk, hogy hogyan lehet bonyolult adatstruktúrákat könnyen kezelhető adatszerkezetekkel ábrázolni. Láthatjuk majd, hogy ezt mutatók felhasználásával fogjuk orvosolni, valamint olyan új „adatstruktúrákat" fogunk bevezetni, mint a vermek, a sorok, a láncolt listák, és a gyökeres fák.

**Lecke fejezetei:**

- A sor és a verem – Olvasó (10 perc)
- Kupac – Olvasó (15 perc)
- Láncolt listák – Olvasó (20 perc)
- Gyakorló feladatok – Gyakorlati (50 perc)

**Téma típusa:** Gyakorlati
**Olvasási és gyakorlási idő:** 90 perc

## A sor és a verem *(10 perc)*

A sor (queue) olyan összetett, absztrakt adatszerkezet, amely dinamikus mérettel rendelkezik. Adathozzáférési stratégiájuk a FIFO, vagyis First In First Out – elsőként berakott elemet lehet először kivenni.
Ez azt jelenti, hogy az adatokat csakis ugyanolyan sorrendben tudjuk kivenni, mint amilyen sorrendbe elhelyeztük őket a sorban.

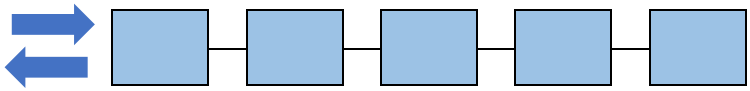
Egy sor jellemző műveletei:

- inicializálás: a sor alapállapotba hozása (a sor alapállapotban üres, nulla elemet tárol)
- sorba (enqueue): egy új elem elhelyezése a sorban
- sorból (dequeue): a sorban legelőször elhelyezett elem értékének visszanyerése, az elem eltávolítása a sorból
- a sor üres-e: megállapítani, hogy a sorban van-e elem (üres sorból nem lehet elemet kiolvasni)
- a sor tele van-e: csak akkor használjuk, ha a sor által tárolható elem számának van felső korlátja. Ezt meg kell vizsgálni, mivel tele lévő sorba már nem lehet további adatokat elhelyezni.

```
def isFull(Q):                      # tele van mar a sorunk
    return len(Q) == max_elem
def isEmpty(Q):                     # van-e meg elem a sorban
    return len(Q) == 0
def EnQueue(Q, elem):               # rakjunk elemet a sorba
    if isFull(Q):
        print("Full")
        return Q
    Q.append(elem)
    return Q
def DeQueue(Q):                     # vegyunk ki elemet a sorbol
    if isEmpty(Q):
        print("Empty")
        return []
    print(Q[:1])
    Q = Q[1:]
    return Q
```

A sort (queue) olyan feladatok megoldásánál célszerű használni, ahol az adatok hozzáférése FIFO kialakítású. Ilyen például a nyomtató várakozási listája, de ezt az absztrakt adatszerkezetet fogjuk használni fák és a gráfok szélességi bejárásánál is.

A verem (stack, LIFO – last in, first out) olyan lineáris absztrakt adatszerkezet, amelyben új elemet a verem elejéhez adunk hozzá (push), és a feldolgozandókat is az elejéről vesszük el (pop). A verem absztrakt adatszerkezet, a LIFO-tulajdonság következtében, hasonlóan működik, mint egy földbe ásott verem, amelynek csak a tetején lehet a terményeket betenni és kivenni is. A veremből az elemeket éppen fordított sorrendben vesszük ki, mint ahogy betettük.



Egy verem jellemző műveletei:

- inicializálás: a verem alapállapotba hozása (a verem alapállapotban üres, nulla elemet tárol)
- behelyezés (push): egy új elem elhelyezése a veremben
- kiolvasás (pop): a veremben legutoljára elhelyezett elem értékének visszanyerése, az elem eltávolítása a veremből
- a verem üres-e: megállapítani, hogy a veremben van-e elem (üres veremből nem lehet elemet kiolvasni)
- a verem tele van-e: csak akkor használjuk, ha a verem által tárolható elem számának van felső korlátja. Ezt meg kell vizsgálni, mert tele lévő verembe már nem lehet további adatokat elhelyezni.

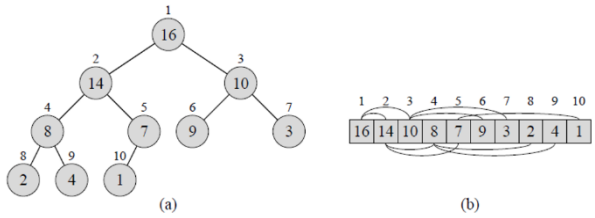
```
def isFull(S):                          # tele van mar a vermunk
    return len(S) == max_elem
def isEmpty(S):                         # ures-e mar vermunk
    return len(S) == 0
def push(S, elem):                      # rakjunk elemet a verembe
    if isFull(S):
        print("Full")
        return S
    S.append(elem)
    return S
def pop(S):                             # vegyunk ki elemet a verembol
    if isEmpty(S):
        print("Empty")
        return []
    print(S[-1])
    S = S[:-1]
    return S
```

## *Kupac (15 perc)*

A prioritási sor egy absztrakt adatszerkezet, melyben az elemeket prioritásuk szerint tároljuk. Itt az elemeket nem a beszúrás ideje alapján érjük el, mint a sor és veremnél, hanem mindig a legnagyobb prioritású elemet tudjuk, nagyon hatékonyan, kiolvasni. A kupac (heap) egy prioritási sor megvalósítás, ahol az elemek struktúrák, és az egyik mezőjük a prioritás (ha egyéb jellemzőjük lenne, azt is külön tárolnánk).

A kupac olyan véges elemsokaság, amely az alábbi tulajdonságokkal rendelkezik:

- Minden elemnek legfeljebb két rákövetkezője (leszármazottja) lehet. Azaz bináris fának tekinthető.
- Minden eleme kisebb (vagy egyenlő) a közvetlen leszármazottjainál.
- A kupac balról folytonos, azaz ha nem teljes a fa, akkor csak a legutolsó szintből hiányozhatnak elemek, de azok is csak a szint jobb széléről.




A kupac műveletei:

- beszúr: beszúr egy elemet
- min/max: a legkisebb/legnagyobb prioritású elemet adja vissza (pl. int-ek esetén minimumot/maximumot) (Fontos: a kupcunk egyidőben csak minimum vagy maxiumumkupac lehet!)

Nézzük meg, hogy ez hogyan festene Python kódban:

Az insert függvényünk az elemeket a kupac utolsó sorának a jobb oldalára helyezi, majd ha kell, akkor felfelé tolja, mindaddig, amíg a megfelelő helyre nem kerül. Ezt megismétli minden egyes elemmel, és így építi fel a kupacot.

```
def Insert(elem):
    global heap
    heap = np.append(heap,elem)      #hozzaadjuk az uj elemet a kupachoz
    akt = len(heap)-1
    if((akt%2)!=0):                  #hova toljuk fel az elemet?
        csere = (int(akt/2))
    else:
        csere = (int((akt-1)/2))
    while(heap[csere] < elem):       # ameddig rossz helyen van az elem, felfele toljuk
        temp = heap[akt]
        heap[akt] = heap[csere]
        heap[csere] = temp
        akt = csere
        if((akt%2)!=0):              #hova toljuk fel az elemet?
            csere = (int(akt/2))
        else:
            csere = (int((akt-1)/2))
```

A max függvényünk a kupac elején lévő elemet (legnagyobb elem) veszi ki kupacból. A „kivevés”, vagyis törlés a következőképpen zajlik: a kupac max elemét (0 elem) kicseréli a legutolsóval, majd törli azt. Ezek után az új 0. elemünket lefelé nyomjuk a kupacban, amíg az el nem éri a megfelelő helyet (felette csak nagyobbak vannak, alatta csak kisebbek vagy ugyanakkora értékűek)

```
def Max():
    global heap
    temp = heap[0]
    heap[0] = heap[len(heap)-1]
    heap[len(heap)-1] = temp
    heap = heap[:-1]

    akt = 0
    for i in range(0,int(len(heap)/2)):
```

```
        jobb = akt*2+2
        bal = akt*2+1
        if (bal<(len(heap))):
            if (heap[bal] > heap[akt]):
                temp = heap[akt]
                heap[akt] = heap[bal]
                heap[bal] = temp
                akt=bal
        if (jobb<(len(heap))):
            if (heap[jobb] > heap[akt]):
                temp = heap[akt]
                heap[akt] = heap[jobb]
                heap[jobb] = temp
                akt=jobb
heap = []
Insert(1)
Insert(5)
Insert(3)
Insert(6)
Insert(2)
Insert(10)
print(heap)
```

```
[10.  5.  6.  1.  2.  3.]
```

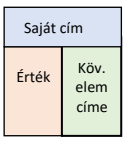
## *Láncolt listák (20 perc)*

Lista absztrakt adatszerkezet, melyben az adatok lineáris sorrendben követik egymást, és egy kulcs többször is előfordulhat. A lista megvalósítható közvetlen eléréssel tömbök segítségével. Az így kialakított listánkban adatok összefüggő memóriaterületen helyezkednek el, minden index közvetlen elérésű, azaz közvetlenül olvasható/írható. A listát emellett ábrázolhatjuk láncolt listás megvalósítással is. A láncolt lista olyan absztrakt adatszerkezet, amelyben az adatok, a tömbökhöz hasonlóan, lineáris sorrendben követik egymást. Míg a tömböknél a lineáris sorrendet a tömbindexek határozzák meg, a láncolt listákban ezt mutatók valósítják meg. Ezt a megoldást főleg akkor érdemes használnunk (például tömb helyett), ha nem tudjuk előre meghatározni mennyi elemünk lesz. Az így létrejött elemünk mindig csak annyi memóriát foglal le, amennyire szükségünk van, hiszen a memóriát a program futása alatt bővíthetjük és fel is szabadíthatjuk. A láncolt listát gyakran használjuk az olyan problémáknál, mint a bináris fák vagy a gráfok bejárása, hiszen a konstans helyfoglalás helyett képes a használata közben átméretezni magát.

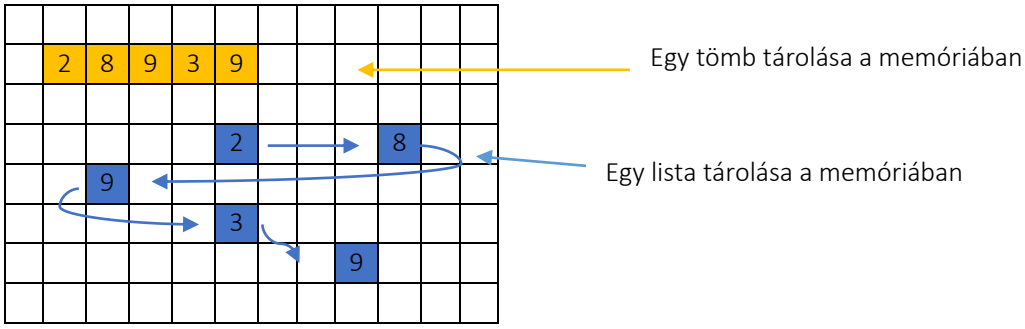
A továbbiakban a teljesség igénye nélkül az egyirányú láncolt listákkal foglalkozunk.

*A láncolt listák felépítése*

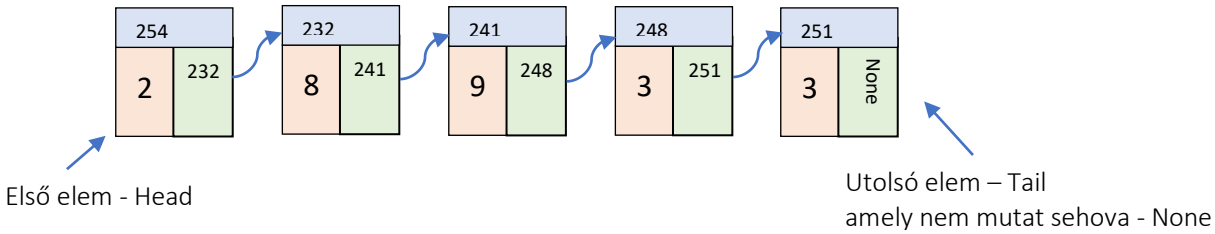
A láncolt listák minden előnye (és hátránya) magából a felépítéséből adódik, amely egy önhivatkozó struktúrából áll, ami eltárolja az elem értékét, és a rákövetkező elem címét, melyet vizuálisan így ábrázolunk:




Míg egy tömb egymást követő adatstruktúrával rendelkezik (a memóriában is), addig a láncolt lista egy látszólag véletlen sorrendben tárolt, de jó sorrendben összekapcsolt absztrakt adatszerkezetként képzelhető el. Például, ha a 2, 8, 9, 3, 9 számokat egy tömbben és egy listában tárolnánk, az valahogy így festene:




A lista megvalósításához egy olyan önhivatkozó adatstruktúrát alkalmazunk, amely az értéke mellett eltárolja a következő elem memóriacímét is. Egy ilyen elemet NODE-nak nevezünk. A lista első elemétől tudunk elindulni, és egész addig lépkedhetünk, amíg a lista következő eleme már nem mutat sehova (a lista utolsó eleme nem mutat sehova – None)




Az egyirányú lista építése relatív egyszerű feladat, azonban a beszúrás és a törlés esetén oda kell figyelni, hiszen, ha valahol megszakítjuk a listánkat, akkor azzal elveszítjük a lista egy részét vagy egészét. A listák esetén használt műveletek:

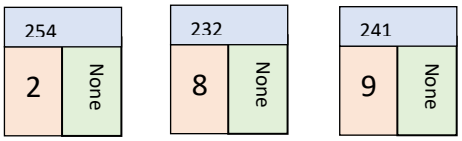
- lista építése
- elem beszúrása a listába
- elem törlése a listából
- elem keresése a listában

*Lista építése*

A következő kódban a lista létrehozását szemléltetjük:

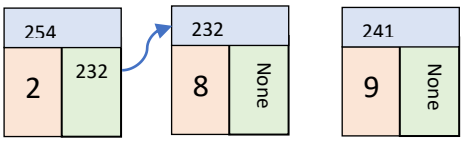
```
# Node osztaly
class Node:
    # a node objektum inicializalasa
    def __init__(self, data):
        self.data = data  # a lista elem erteke
        self.next = None  # a lista elem kovetkezo erteke
class LinkedList:
    def __init__(self):
        self.head = None
# ures lista letrehozasa
llist = LinkedList()
llist.head = Node(1)   #a lista elso elemenek hozzaadasa
masodik = Node(2)      #egy uj node letrehozasa
harmadik = Node(3)     #egy uj node letrehozasa
```

A fenti programban létrehoztuk a lista osztályunkat, továbbá beállítottuk a lista fejét (első elemét), és két új elemnek foglaltuk helyet a memóriában. A szemléltetés kedvéért ezek az elemek most még nincsenek összekötve, csak „szabadon lebegnek” szétszórva a memóriában, mely valahogy így néz ki:



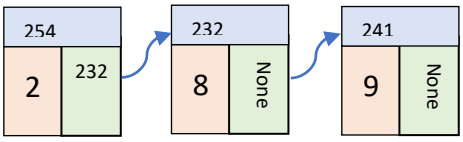

A következő lépésben a lista fejéhez csatoljuk a *masodik* nevű elemünket.

```
llist.head.next = masodik  #hozzacsatoltunk egy elemet a listahoz
```




Az utolsó lépésben a lista második eleméhez csatoljuk a *harmadik* nevű elemünket.

```
masodik.next = harmadik  #hozzacsatoltunk egy elemet a listahoz
```

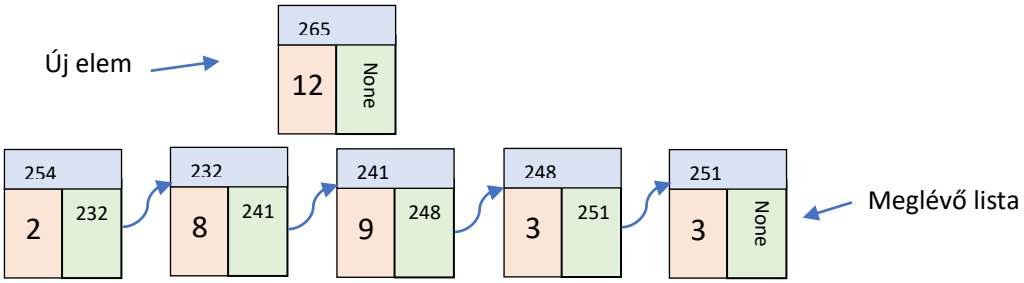



Ez a folyamat természetesen egy program esetén nem így „manuálisan”, hanem ciklussal történik.

*Elem beszúrása a listába*

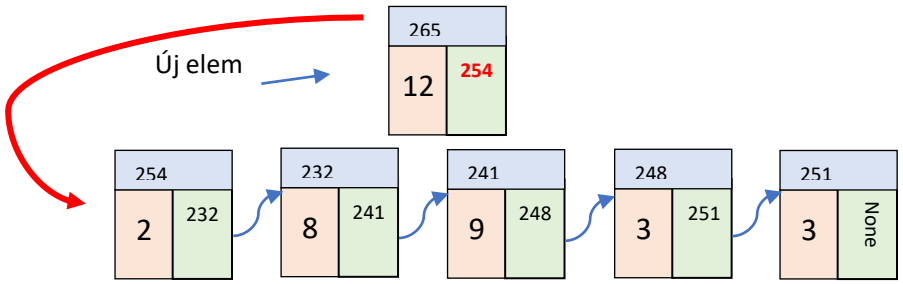
Egy listába több helyen is beszúrhatunk elemeket, ennek egyik legkézenfekvőbb része amikor a lista végére szúrunk be elemet.
Ennek lépései:

- lefoglaljuk/létrehozzuk az új elemet,
- ha a listánk üres, akkor csak ez az elem kerül a listába,
- egyébként megkeressük az utolsó elemet,
- az utolsó elem „következő” mutatóját beállítjuk az új elem címére, az új elemét pedig NONE-ra
- utolsó elemet éppen arról ismerjük meg, hogy a benne lévő következő pointer értéke NONE




*Elem beszúrása a lista elejére*

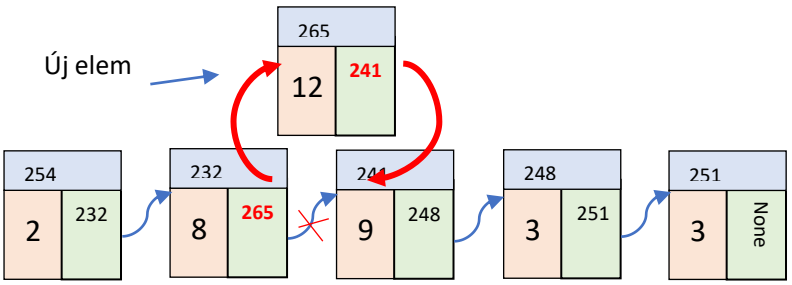
Készítsünk egy függvényt, amely beszúr egy elemet a lista elejére!




```
def push(self, uj_adat):
    # 1-2. lepes: foglaljunk helyet a memoriaban az uj elemnek
    new_node = Node(uj_adat)
    # 3. lepes: az uj elem kovetkezo eleme a lista feje legyen
    new_node.next = self.head
    # 4. lepes: a lista fejenek cime ezentul az uj elem legyen
    self.head = new_node
```

*Elem beszúrása egy adott elem után*

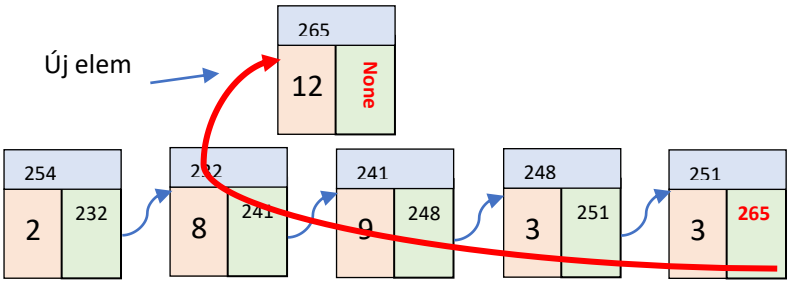
Készítsünk egy függvényt, amely beszúr egy elemet az adott node után.




```
def insertAfter(self, elozo_node, uj_adat):
    # 1. lepes: Nezzuk meg, hogy az elozo node letezik-e
    if elozo_node is None:
        print ("A keresett elem nincs a listaban")
        return
    #  2-3. lepes: Hozzuk letre az uj valtozot es tegyuk bele az uj adatot
    uj_node = Node(uj_adat)
    # 4. lepes: Az uj elem kovetkezo eleme legyen az elozo Node kovetkezojenek cime
    uj_node.next = elozo_node.next
    # 5. lepes: az elozo elem kovetkezoje az uj node legyen
    elozo_node.next = uj_node
```

*Elem beszúrása a lista végére*

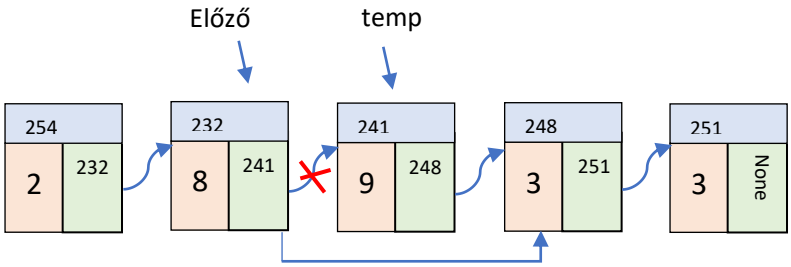
Készítsünk egy függvényt, amely beszúr egy elemet a lista végére.



```
def append(self, uj_adat):
   # 1-2. lepes: Hozzuk letre az uj valtozot es tegyuk bele az uj adatot
   uj_node = Node(uj_adat)
   # 3. lepes: Ha a lista ures, akkor az uj elemunk legyen az uj head
   if self.head is None:
        self.head = uj_node
        return
   # 4. lepes: kulonben menjunk el a lista vegere
   utolso= self.head
   while (utolso.next):
       utolso = utolso.next
   # 5. lepes: az utolso elem kovetkezoje az uj elem legyen
   utolso.next =  uj_node
```

*Elem törlése a listából*

Készítsünk egy függvényt, amely egy kulcs(érték) alapján megkeres és kitöröl egy elemet a listából.




```
def deleteNode(self, kulcs):
        # 1. lepes: eltaroljuk a lista fejet
        temp = self.head
        # 2. lepes: ha a lista feje tartalmazza az erteket, akkor toroljuk ki
        if (temp is not None):
            if (temp.data == kulcs):
                self.head = temp.next
                temp = None
                return
        # 3. lepes: keressuk meg a torolni kivant elemet, es ezalatt vegig taroljuk el az elozo elem kovetkezo
cimet
            if temp.data == kulcs:
                break
            elozo = temp
            temp = temp.next
        # 4. lepes: ha a keresett ertek nincs a listaban
        if(temp == None):
            return
        # 5. lepes: toroljuk az elem kapcsolatat a listabol
        elozo.next = temp.next
        temp = None
```

## *Gyakorló feladatok (45 perc)*

1. A kupac használatával *O(N log N)*-es rendezés készíthető. Készítsünk programot a kupacrendezés bemutatására.

```
def kupacosit(arr, n, i):
    legnagyobb = i  # a legnagyobb a gyokerelem
    l = 2 * i + 1     # bal = 2*i + 1
```

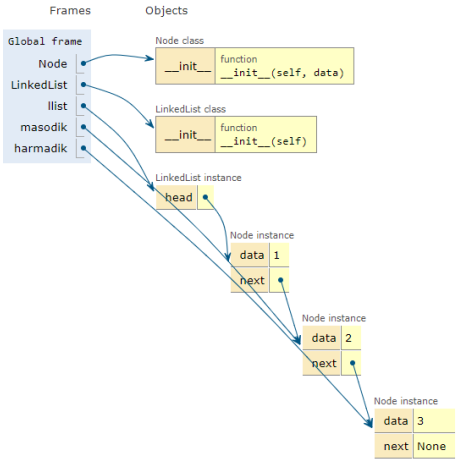
```
    r = 2 * i + 2     # jobb = 2*i + 2

    # Nezzuk meg a baloldali reszfa gyokerelemet
    # es hogy ez nagyobb-e mit a lagnagyobb lelem
    if l < n and arr[i] < arr[l]:
        legnagyobb = l

    # Nezzuk meg a jobboldali reszfa gyokerelemet
    # es hogy ez nagyobb-e mit a lagnagyobb lelem
    if r < n and arr[legnagyobb] < arr[r]:
        legnagyobb = r

    # cserljuk ki a gyokerelem erteket amennyiben szukseges
    if legnagyobb != i:
        arr[i],arr[legnagyobb] = arr[legnagyobb],arr[i]  # csere

        # kupacositsuknk
        kupacosit(arr, n, legnagyobb)

# rendezzuk a tombot
def kupacRendez(arr):
    n = len(arr)

    # epitsunk egy max kupacot
    # mivel az utolso szulo pozicioja ((n//2)-1) kezdhetjuk itt az epitest.
    for i in range(n // 2 - 1, -1, -1):
        kupacosit(arr, n, i)

    # egyesselev vegyuk ki az elemeket
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]   # swap
        kupacosit(arr, i, 0)


arr = [ 12, 11, 13, 5, 6, 7]
kupacRendez(arr)
n = len(arr)
print ("A rendezett tomb:")
for i in range(n):
    print ("%d" %arr[i]),

# a kod Mohit Kumra forraskodja alapjan keszult
```

```
A rendezett tomb:
5
6
7
11
12
13
```

2. Módosítsuk úgy a programot, hogy az elemek csökkenő sorrendebe rendeződnek.
3. Mutasd meg, hogy a kupacrendezés tényleg *O(N log N)* időben fut le.
4. Készíts egy programot mely tömb és láncolt lista felszállásával szemlélteti a következő műveletek időigényét:
   - Beszúrás a lista elejére
   - Beszúrás a lista végére
   - Beszúrás a lista közepébe
   - Törlés a lista elejéről
   - Törlés a lista véréréről
   - Törlés a lista közepéről

5. Vizualizáld a kódot a http://pythontutor.com/visualize.html segítségével. Pl.:

```
# Node osztaly
class Node:
    # a node objektum inicializalasa
    def __init__(self, data):
        self.data = data  # a lista elem erteke
        self.next = None  # a lista elem kovetkezo erteke
class LinkedList:
    def __init__(self):
        self.head = None
# ures lista letrehozasa
llist = LinkedList()
llist.head = Node(1)    #a lista elso elemenek hozzaadasa
masodik = Node(2)       #egy uj node letrehozasa
harmadik = Node(3)      #egy uj node letrehozasa
llist.head.next = masodik  #hozzacsatoltunk egy elemet a listahoz
masodik.next = harmadik  #hozzacsatoltunk egy elemet a listahoz
```




## *Ajánlott kitekintő anyag (146 perc)*

Adatszerkezetek **(11 perc)** – Videó magyar nyelven
Listák **(26 perc)** – Videó magyar nyelven
Verem és a Sor **(18 perc)** – Videó magyar nyelven
Prioritási sor és Kupac **(29 perc)** – Videó magyar nyelven
Implementációs gondolatok **(8 perc)** – Videó magyar nyelven
Verem, Sor, Láncolt Lista **(20 perc)** - Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein - ÚJ ALGORITMUSOK link 188-194 oldal
Kupac **(10 perc)** – Youtube link angol nyelven
Láncolt lista **(18 perc)** – Youtube link angol nyelven
Verem és a Sor **(6 perc)** – Youtube link angol nyelven