Bevezetés, tömb adatszerkezet

Dr. Szeghalmy Szilvia
Debreceni Egyetem, Informatikai Kar

Miről fogunk tanulni a gyakorlaton?

- Milyen alapvető megoldások léteznek az adatok memóriabeli tárolására?
- Milyen műveletek elvégzését támogatják az egyes módszerek?
- Hogyan lehet megvalósítani ezeket a műveleteket?

- Mire lesz ez jó?
  - Megfelelő adatszerkezet, algoritmus

    => hatékonyabb programok

    => hibalehetőségek csökkentése

    bizonyos műveletek adatszerkezetek szintjén nem támogatottak

Egyéb tudnivalók

- Az algoritmusok megadására a Python nyelvet fogjuk használni
- A Python csak leíró nyelvként szerep, algoritmusokról tanulunk =>
  a beépített függvények használatát általában kerülni fogjuk

- ZH írásnál fontos tudni:
  - **algoritmus megadást várunk el** (összegzés algoritmusa vs. sum függvény)
  - a rendszer (általában) bármilyen jellegű jó megoldást elfogad
  - a ZH-t követő eredmény tájékoztató jellegű lesz, ugyanis a megoldásokat átnézzük és nullázzuk azon feladatoknál a pontszámot, amelyek nem tesznek eleget a fenti követelményeknek

  - Részletek az e-learningben.

Tömb adatszerkezet

Tömb

- Tulajdonságok
  - Homogén: csak azonos típusú elemeket tartalmazhat
  - Statikus (***Létezik dinamikus tömb** is, de ha az adatszerkezetek órán jelző nélkül hallod vagy látod a tömb szót, akkor mindig a statikus adatszerkezetre gondolj.*)
  - Asszociatív (számunkra: elemei egyesével címezhetők)

- Reprezentáció: **folytonos**

  értsd: az adatelemek a memóriában egy tömbben helyezkednek el

0	1	2	…	N-1
5	8	0	…	3

Művelet: Létrehozás

- Létrehozáskor eldől
  - a típus (milyen típusú adatelemeket tárolunk)
  - a dimenziószám
  - dimenziónként az indextartomány

- Megvalósítás Pythonban (csak példák, részletek Programozás1 órán)

```
import numpy as np
t = np.empty( 4, int )          # 1D-s tömb, 4 elemű, egész szám típusú elemekkel
t2 = np.array([10, 5, 2])       # 1D-s tömb, 3 elemű, egész szám típusú elemekkel
t3 = np.zeros(10, np.uint8)     # 1D-s tömb, 10 elemű, egész szám típusú elemekkel
m = np.empty( (5, 3), float )   # 2D-s tömb (mátrix), 5 sor, 3 oszlop, valós típus
```

Megj.: A konkrét megoldás mindig függ a programnyelvtől. Pythonban (C-ben, C++-ban, C#-ban, stb. az indextartomány 0-val indul => a méretet megadva eldől az indextartomány).

Megj.2: A *t2* tömb létrehozásánál a típus meghatározása a felsorolt elemek alapján történik. Ha nem hiszed, akkor a létrehozás után futtasd a *print(t2.dtype)* parancsot.

Megj.3: Az *empty* a memória foglalásánál nem törli ki az ott lévő értékeket => a *t* és *m* tömbök memóriaszeméttel vannak töltve. A *zeros* függvény 0 értékekkel feltöltött tömböt hoz létre.

Műveletek: Elérés

- **Közvetlen elérés** dimenziónként egy-egy index használatával

- A folytonos tárolási módot és a homogenitást kihasználva:
  - 1D-s eset: i. elem elérése

    ```
    tömb_kezdőcíme + i * elem_méret
    ```
  - 2D-s eset: i. sor, j. oszlop

    ```
    mátrix_kezdőcíme + i * elem_méret * oszlopok_száma + j * elem_méret
    ```

  …

- Python:

```
…
t = np.array([3, 4, 6])
print(t[0], t[2])  # A 3-as és a 6-os értékek elérése és megjelenítése

m = np.array([[20, 30],[50, 20],[11, 45]])
print( m[0, 0], m[2, 1] )  # A 20-as és a 45-ös értékek megjelenítése
```

Módosító műveletek

```
# LÉTREHOZÁS:
t = np.empty(5, int)
```

Csere (egy elem felülírása): van, közvetlen elérés alapján

pl.:

```
t[2] = 20  # a 2. indexű elem cseréje (felülírása)
```

Bővítés:

statikus adatszerkezet => fizikai (memóriafoglalással járó) bővítés nincs

Törlés:

statikus adatszerkezet => fizikai (memóriafelszabadítással járó) törlés nincs

Logikai bővítés és törlés (a lefoglalt memória mérete nem változik ezen műveletek hatására):

a) a még üres helyeket speciális jel jelzi

bővítés: az első üres helyre (ha van hely)

törlés: a törlendő elemet cseréljük a speciális jelre

Index	0	1	2	3	4
Elemek	5	0	None	3	None

b) az értékes elemek folytonosan történő tárolásával és azok számának (*db*) tárolásával

bővítés: a *db* helyen (ha van hely)

törlés: az utolsó értékes elemmel felülírjuk a törlendőt

csökkentjük az *db* értékét

Index	0	1	2	3	4
Elemek	5	3	10	5	2
db: 3					

Műveletek: Bejárás

Az adatszerkezet elemeit pontosan egyszer érjük el.

Feladat: Járjuk be az *n* elemű egy dimenziós *t* tömb elemeit és jelenítsük meg azokat a képernyőn!

```
def bejaras(t, n):  # t egy n elemű 1D-s tömb
    for i in range(n):
        print(t[i])   # t[i] az i. indexen lévő tömbelem
```

Megj.: ezen a gyakorlaton igyekszünk az algoritmusra koncentrálni, a technikai elemek visszaszorításával

Pythonban természetesen nem lenne szükség az elemszám átadására (tömb mérete: len(t), t.shape[0])

Megj.2: mivel a statikus tömb adatszerkezetet tárgyaljuk mindig feltételezni fogjuk, hogy legalább egy eleme van a tömbnek (n>=1).

Műveletek: Lineáris (teljes) keresés

Feladat: Keressük meg egy *n* elemű egy dimenziós *t* tömbben a *mit* értékű elemet. A függvény adja vissza az elem **indexét** (tömbbéli pozícióját), vagy ha az elem nem létezik, akkor (-1)-es értéket.

Alapelv:

- Járjuk be az adatszerkezetet és minden egyes elemre ellenőrizzük, hogy egyezik-e a keresett értékkel
- Ha egyezést találtunk a keresésnek sikeresen vége.
- Ha bejártuk az összes elemet és nem volt egyezés, akkor a keresésnek sikertelenül vége.

```
def kereses(t, n, mit):
    for i in range(n):
        if t[i] == mit:
            return i  # sikeresen vége
    return -1 # sikertelenül vége
```

Index	0	1	2	3	4
Elemek	50	20	30	60	24

Megj.: Ha a keresett elem többször szerepel, az első előfordulást adja vissza.

Megj.: Ez a fajta keresés mindig használható, ha az adatszerkezetet be tudod járni és az elemek között értelmezett az egyenlőségvizsgálat.

Műveletek: Lineáris (teljes) keresés

Lássuk tisztábban a szükséges lépéseket:

```
def kereses(t, n, mit):  # n >= 1
    i = 0
    while i < n:
        if t[i] == mit:
           return i
        i += 1
    return -1
```

Index	0	1	2	3	4
Elemek	50	20	30	60	24

Hányszor hajtódik végre az **i<n** ellenőrzés? (Tekintsük ezt egy lépésnek.)

Legjobb eset: A keresett elem a tömb legelső eleme => 1 (nagy) lépésben megtaláljuk

Legrosszabb eset: Ha olyan elemet keresünk, mely nincs a tömbben => n+1 lépés szükséges

A tömbben előforduló elemek megtalálásához átlagosan (n+1)/2 lépést teszünk meg.

Műveletigény

A legrosszabb esetek egyike: Tegyük fel, hogy a 120-as értéket keressük (mit = 120), ami nincs a tömbben.

Lássuk, hogy melyik utasítás hányszor fut le:

Index	0	1	…	3	4
Elemek	50	20	…	60	24

```
def kereses(t, n, mit):
    i = 0           # csak 1x fut le, elhanyagolható az időigény
    while i < n:    # n+1, de a +1-et is itt nyugodtan elhanyagolhatjuk
        if t[i] == mit:  # n-szer fut le, minden elemet ellenőrzünk
            return i     # soha nem fut le
        i += 1           # n-szer fut le
    return -1      # csak 1x fut le, elhanyagolható
```

A legrosszabb eset műveletigénye: ~3n

Az eljárás **lineáris idejű,** vagyis a bemenet hosszával (egyenesen) arányos a műveletigény.

Megj.: Az algoritmusok komplexitásáról részletesen előadáson lesz szó.

Teljes keresés javítása (strázsa elem)

Csak olyan adatszerkezetekben használjuk, ahol a bővítés és törlés "olcsó" művelet

szúrjuk be a keresett elemet az adatszerkezet végére (strázsa)

végezzük el az elem keresését (az elemszámmal nem törődve)

vegyük ki a strázsa elemet

Index	0	1	2	3	4
Elemek	50	20	30	60	24

Tegyük fel, pl. hogy **dinamikus** tömbünk van!

```
def kereses(t, n, mit):
    t.append(mit) # strázsa beszúrása
    i = 0
    while t[i] != mit:
        i += 1
    del t[n] # strázsa törlése
    return i if i<n else -1   # az eredmény visszaadása
```

Mit nyertünk vele? Az **i<n** feltétel már nem része a ciklusnak, mindig csak egy alkalommal fut le a korábbi n+1 alkalom helyett. Megfelelően nagy n esetén megérheti a strázsa elem beszúrásával és törlésével járó plusz munka.

Műveletek: Lineáris keresés rendezett elemekre

Feladat: Keressük meg egy *n* elemű *t* tömbben a *mit* értékű elemet. A függvény adja vissza az elem tömbbéli pozícióját, vagy ha az elem nem létezik, akkor (-1)-es értéket. **A *t* tömb elemei monoton nem csökkenő sorrendben állnak**. Használjuk ki a rendezettséget a keresés során!

Index	0	1	2	3	4
Elemek	10	20	20	60	80

```
def kereses(t, n, mit):
    for i in range(n):
        if t[i] == mit:
            return i # a keresett elem indexe
        if t[i] > mit:  # találtunk egy a keresettnél nagyobb elemet
            return -1

    return -1  # nincs meg az elem
```

Megj.1: Fordított irányú rendezettség esetén a t[i] < mit feltételt használd.

Megj.2: A while ciklusos változatba hasonló módon beépítheted.

Megj.3: A strázsa módszernél a keresett elemet vagy tőle nagyobb (fordított irányú rendezettségnél tőle kisebb) értéket is választhatsz strázsának.

Művelet: Bináris keresés

- Igényei:
  - elemek közti hasonlítás (<, >, ==)
  - közvetlen elérés
  - rendezettség

- A "Gondoltam egy számot!" játéknál használható legjobb stratégiával analóg módon folyik a keresés.

Művelet: Bináris keresés

A 78-as érték keresése

- A keresési tartomány kezdetben az egész tömb.
- Nem üres => ellenőrizzük, a keresési tartomány középső elemének és a keresett elemnek a viszonyát.
- A **keresett elem nagyobb, mint a középső elem** => A **keresési tartomány** jobb oldali részén keresünk tovább.

Index	0.	1.	2.	3.	4.	5.	6.
Elemek	15	20	20	42	78	86	95

Művelet: Bináris keresés

A 78-as érték keresése

- A keresési tartomány 4., 5. és 6. elemet tartalmazza.
- Nem üres => ellenőrizzük a keresési tartomány középső elemének és a keresett elemnek a viszonyát.
- A **keresett elem kisebb, mint a középső elem** => A **keresési tartomány** bal oldali részén keresünk tovább.

Index	0.	1.	2.	3.	4.	5.	6.
Elemek	15	20	20	42	78	86	95

Művelet: Bináris keresés

A 78-as érték keresése

- A keresési tartomány 4. elemet tartalmazza.
- Nem üres => ellenőrizzük a keresési tartomány középső elemének és a keresett elemnek a viszonyát.
- A **keresett elem egyenlő a középső elemmel** => A keresésnek sikeresen vége

Index	0.	1.	2.	3.	4.	5.	6.
Elemek	15	20	20	42	78	86	95

Művelet: Bináris keresés

A 16-os érték keresése

- A keresési tartomány kezdetben az egész tömb.
- Nem üres => ellenőrizzük, a keresési tartomány középső elemének és a keresett elemnek a viszonyát.
- A **keresett elem kisebb, mint a középső elem** => A **keresési tartomány** bal oldali részén keresünk tovább.

Index	0.	1.	2.	3.	4.	5.	6.
Elemek	15	20	20	42	78	86	95

Művelet: Bináris keresés

A 16-os érték keresése

- A keresési tartomány 0., 1. és 2. elemet tartalmazza.
- Nem üres => ellenőrizzük, a keresési tartomány középső elemének és a keresett elemnek a viszonyát.
- A **keresett elem kisebb, mint a középső elem** => A **keresési tartomány** bal oldali részén keresünk tovább.

Index	0.	1.	2.	3.	4.	5.	6.
Elemek	15	20	20	42	78	86	95

Művelet: Bináris keresés

A 16-os érték keresése

- A keresési tartomány a 0. elemet tartalmazza.
- Nem üres => ellenőrizzük, a keresési tartomány középső elemének és a keresett elemnek a viszonyát.
- A **keresett elem nagyobb, mint a középső elem** => A **keresési tartomány** jobb oldali részén keresünk tovább.

Index	0.	1.	2.	3.	4.	5.	6.
Elemek	15	20	20	42	78	86	95

Művelet: Bináris keresés

A 16-os érték keresése

- A keresési tartomány üres => A 16-os érték nem található a tömbben.

Index	0.	1.	2.	3.	4.	5.	6.
Elemek	15	20	20	42	78	86	95

Műveletek: Bináris keresés

Feladat: Írj függvényt, mely egy monoton nem csökkenő sorrendbe rendezett, *n* elemű *t* tömböt és egy keresett értéket (mit) kap meg paraméterként. A függvény adja vissza a keresett elem indexét, vagy ha nem szerepel, akkor (-1)-es értéket.

Index	0.	1.	2.	3.	4.	5.	6.
Elemek	15	20	20	42	78	86	95

```
def binaris_kereses(t, n, mit):
    ah = 0                      # a keresési tartomány alsó határa
    fh = n-1                    # a keresési tartomány felső határa
    while ah<=fh:               # amíg nem üres a tartomány
        k = (ah+fh)//2          # a középső elem indexe (egész szám lehet csak => // )
        if t[k] == mit:         # a középső elem a keresett elem-e?
            return k            # igen, visszaadjuk a tömbbeli pozícióját
        if t[k] > mit:          # a középső elem nagyobb, mint a keresett
            fh = k-1            # szűkül a keresési tartomány a bal oldali részre
        else:
            ah = k+1            # szűkül a keresési tartomány a jobb oldali részre
    return -1                   # a keresési tartomány üres => nincs meg az elem
```

Műveletek: Rendezés (közvetlen kiválasztásos)

Rendezzük közvetlen kiválasztással növekvő sorrendbe az alábbi tömb elemeit!

Hasonlított elemek indexe	i	j		
Elemek	**50**	**20**	10	5

A t[i] > t[j] => cseréljük ki őket és lépjünk a j változóval.

Hasonlított elemek indexe	i		j	
Elemek	**20**	50	**10**	5

t[i] > t[j] => cseréljük ki őket és lépjünk a j változóval.

Hasonlított elemek indexe	i			j
Elemek	**10**	50	20	**5**

t[i] > t[j] => cseréljük ki őket (a j-vel végigértünk, lépünk az i-vel => a vizsgálandó rész csökken)

Hasonlított elemek indexe		i	j	
Elemek	5	**50**	**20**	10

Műveletek: Rendezés (közvetlen kiválasztásos)

Hasonlított elemek indexe		i	j	
Elemek	5	**50**	**20**	10

t[i] > t[j] => cseréljük ki őket és lépjünk a j változóval

Hasonlított elemek indexe		i		j
Elemek	5	**20**	50	**10**

t[i] > t[j] => cseréljük ki őket (a j-vel végigértünk, lépünk az i-vel => a vizsgálandó rész csökken)

Hasonlított elemek indexe			i	j
Elemek	5	10	**50**	**20**

t[i] > t[j] => cseréljük ki őket (a j-vel végigértünk, lépünk az i-vel => a vizsgálandó rész csökken)

Hasonlított elemek indexe				i
Elemek	5	10	20	50

A vizsgálandó rész egyetlen elemű, tehát rendezett, a rendezésnek vége.

Megj.: A legrosszabb esetre láttunk példát. Általában nem kell minden lépésben cserélni.

Műveletek: Rendezés (közvetlen kiválasztásos)

Feladat: Írj void függvényt (eljárást), mely egy *n* elemű *t* tömböt kap meg paraméterként és helyben rendezi azt a közvetlen kiválasztásos rendezéssel növekvő sorrendbe.

```
def rendezes(t, n):
    for i in range(n-1): # mert 1 elemet már nem kell rendezni
         for j in range(i+1, n): # az i. elemtől kezdődő résztömbbel foglalkozunk
              if t[i] > t[j]:  # i<j => ha t[i] > t[j], akkor rossz a sorrend
                  tmp = t[i]   # az i. és a j. tömbelem megcserélése
                  t[i] = t[j]
                  t[j] = tmp
```

#két elem felcserélése Pythonos módon: t[i], t[j] = t[j], t[i]

Alapvető algoritmusok: összegzés

Írj függvényt, mely meghatározza az *n* elemű *t* tömb elemeinek összegét!

```
def osszeg(t, n):
    s = 0
    for i in range(n):
        s += t[i]
    return s
```

Alapvető algoritmusok: számlálás

Írj függvényt, mely meghatározza, hogy hányszor fordul elő érték a tömbben. A függvény a tömböt (*t*), annak méretét (*n*) és a keresett értéket (*x*) kapja meg paraméterként.

```
def szamlal(t, n, x):
      db = 0   # érvényes elem indexe! (statikus adszerk, n>0)
      for i in range(n):
         if t[i] == x:
            db += 1
      return db
```

Alapvető algoritmusok: minimum keresés

Írj függvényt, mely paraméterként kap egy *n* elemű *t* tömböt és visszaadja a legkisebb értéket.

Index	0.	1.	2.	3.	4.	5.	6.
Elemek	95	60	20	28	78	86	5

```
import math
def minhely_kereso(t, n):
    min = math.inf # ettől biztos lesz kisebb elem az adatszerkezetben
    for i in range(n):
       if min > t[i]:
           min = t[i]
    return min
```

Megj.: n>=1 esetén megszokott a t[0] kezdőérték alkalmazása is. Ilyenkor a range(1, n)-re változhat.

Alapvető algoritmusok: minimumhely

Írj függvényt, mely paraméterként kap egy *n* elemű *t* tömböt és visszaadja a legkisebb elem indexét. Több minimum érték esetén a tömbben előrébb álló indexével térjen vissza.

Index	0.	1.	2.	3.	4.	5.	6.
Elemek	95	60	20	28	78	86	5

```
import math
def minhely_kereso(t, n):
    min = math.inf # ettől biztos lesz kisebb elem az adatszerkezetben
    minhely = -1   # érvénytelen
    for i in range(n):
        if min > t[i]:
            min = t[i]
            minhely = i
    return minhely
```

Alapvető algoritmusok: minimumhely II.

Írj függvényt, mely paraméterként kap egy *n* elemű *t* tömböt és visszaadja a legkisebb elem indexét. Több minimum érték esetén a tömbben előrébb álló indexével térjen vissza.

Index	0.	1.	2.	3.	4.	5.	6.
Elemek	95	60	20	28	78	86	5

```
def minhely_kereso(t, n):
      minhely = 0   # érvényes elem indexe! (statikus adszerk, n>0)
      for i in range(n):
         if t[minhely] > t[i]:
            minhely = i
      return minhely
```

Alapvető algoritmusok: maximumhely keresés

Analóg módon kereshető a legnagyobb elem tömbbeli pozíciója is:

```
def maxhely _kereso(t, n):
      maxhely = 0   # érvényes elem indexe! (statikus adszerk, n>0)
      for i in range(n):
         if t[maxhely] < t[i]:
            maxhely = i
      return maxhely
```