Algoritmusok és adatszerkezetek gyakorlat – 09

Rendezések

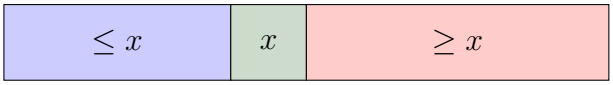
Néhány órával ezelőtt megismerkedtünk már a Merge Sort rendező algoritmussal. A Merge Sort-ról tudjuk, hogy a legrosszabb eset időigénye $O(n \log n)$. Tetszőleges (általános célú) rendezőalgoritmusra a jelenlegi legjobb időigény alsó korlátja $\Omega(n \log n)$. Felmerülhet a kérdés, hogy ha a Merge Sort időigénye éppen ennyi, miért léteznek más algoritmusok is. Eddig csak az időigénnyel foglalkoztunk, viszont a rendezőalgoritmusok tárigényei elég eltérőek lehetnek.

Merge Sort

- **Algoritmus:**
  $MERGESORT(A[1, \ldots, n])$
  1. Ha $n = 1$ *return* A
  2. Rekurzívan alkalmazzuk a lista két felére: $MERGESORT(A[1, \ldots, \lceil n/2 \rceil])$ és $MERGESORT(A[\lceil n/2 \rceil + 1, \ldots, n])$
  3. A 2 listát „fésüljük össze”
- **Időigény:** $O(n \log n)$ (legrosszabb, legjobb és átlagos esetben is)
- **Tárigény:** $O(n)$ plusz tárat igényel, ha az adathalmazt tömbben tároljuk (ugyanannyit, mint a rendezendő input elemszáma) ezért nagy elemszámú tömbben tárolt inputhoz nem ajánlott a használata
- A lehető legjobb választás, ha láncolt listát kell rendezni, mert ekkor csak $O(1)$ a plusz tárigény
- Stabil rendezés, ami azt jelenti, hogy az inputban két egyenlő elem eredeti egymáshoz képesti sorrendje megmarad a rendezés után is

Quick Sort

- **Paradigma:** Divide & Conquer
  1. **Divide:** Osszuk fel a tömböt két résztömbre egy x pivot elem szerint a következőképpen: azok az elemek, amik kisebb egyenlőek a pivotnál legyenek előtte, amik nagyobb egyenlőek nála, legyenek utána.(A pivottal egyenlő elemek bármelyik résztömbbe kerülhetnek.)

  2. **Conquer:** Rekurzívan rendezzük a két résztömböt.
  3. **Combine:** Triviális.

- **Algoritmus:**

```
QuickSort(A, p, r)
  if(p < r)
        q = Partition(A, p, r)
        QuickSort(A, p, q-1)
        QuickSort(A, q+1, r)
```

```
Partition(A, p, q)
  x=A[p]
  i=p
  j:=r
  while(true){
    while(A[j]>=x&&(i<j))
      j:=j-1
    while(A[i]<=x&&(i<j))
      i:=i+1
    if(j<=i) break
    else csere(A[i],A[j])
  }
  csere(A[p],A[j])
  return j
```

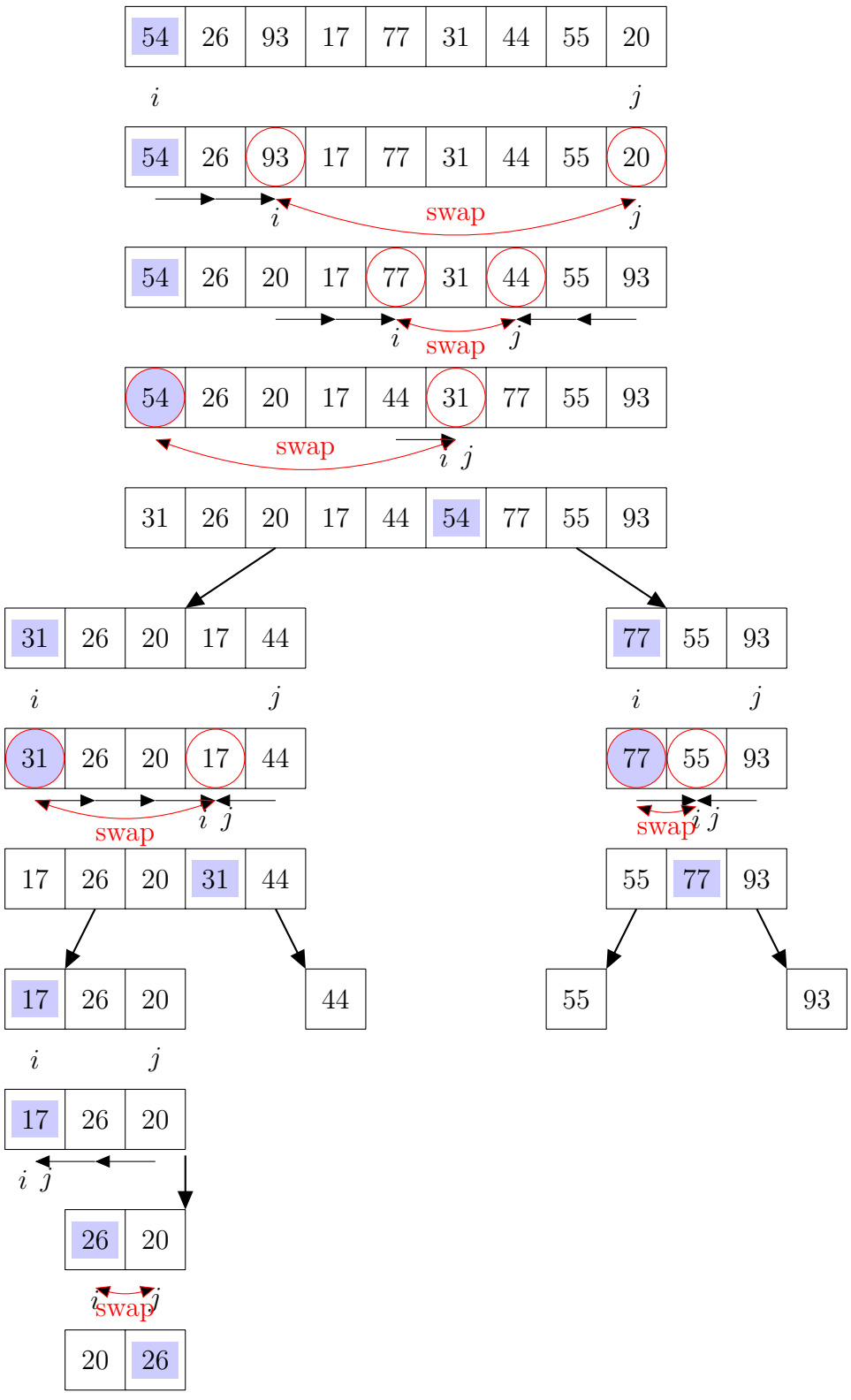
- Meghívása: $QuickSort(A, 1, n)$
- **Időigény:** $O(n^2)$ (legrosszabb eset), $O(n \log n)$ (legjobb és átlagos eset)
- **Tárigény:** $O(\log n)$ plusz tárat igényel
- Nem stabil rendezés
- A gyakorlatban általában kétszer olyan gyors, mint a Merge Sort

Counting Sort

- Akkor használjuk, ha a tömbben legfeljebb k-féle érték szerepel
- **Algoritmus:**
  $CountingSort(A[1, \ldots, n], k)$
  1. Hozzunk létre egy k elemű tömböt és számoljuk bele össze, melyik elemből mennyi van
  2. Módosítsuk úgy az értékeket, hogy minden tömbelem az őt megelőzőek összegét tartalmazza
  3. Tegyünk minden input elemet a helyére a kimenetben úgy, hogy végigmegyünk az inputon és a segédben található indexre tesszük, majd növeljük a számlálóját
- **Időigény:** $O(n + k)$. Akkor érdemes használni, ha $k = O(n)$, mert akkor a futásidő $O(n)$
- **Tárigény:** $O(n + k)$
- Stabil rendezés
- Külső rendezés

1. Feladat Rendezzük az $54, 26, 93, 17, 77, 31, 44, 55, 20$ elemeket tartalmazó tömböt a QuickSort algoritmussal.

Megoldás



2. Feladat Rendezzük a $3, 6, 4, 1, 3, 4, 1, 4$ elemekből álló tömböt Leszámláló rendezéssel.

Megoldás

In:	3	6	4	1	3	4	1	4

Gyűjtsük össze, miből mennyi van. A paraméterek: $n = 8$ és $k = 6$. Vegyünk fel segédnek egy $k + 1$ elemű tömböt.

```
for x in input:
  count[key(x)] += 1
```

C:	0	2	0	2	3	0	1

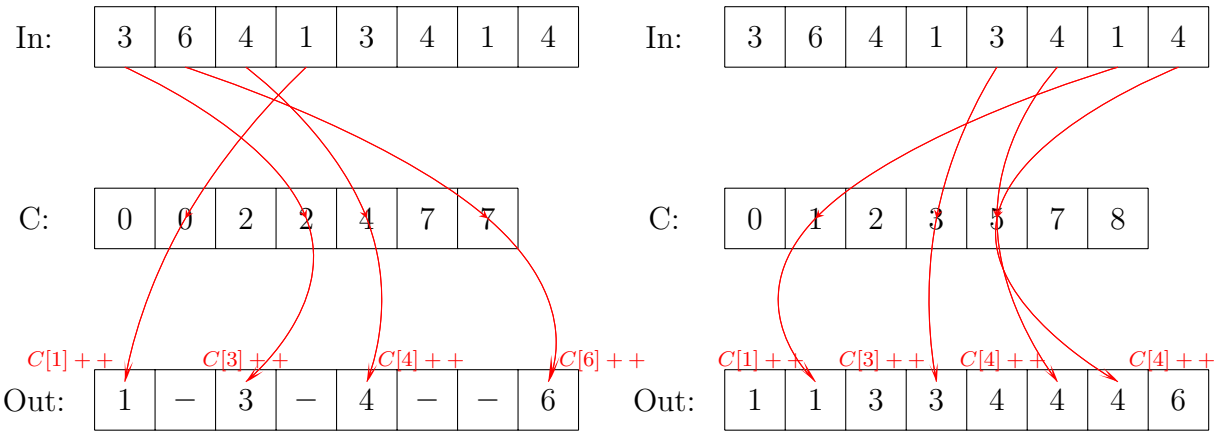
Módosítsuk úgy az értékeket, hogy minden elem a nála <-ek összegét tartalmazza (prefixösszeg tömbbé alakítás):

```
total = 0
for i in range(k+1):    // i = 0, 1, ... k
  oldCount = count[i]
  count[i] = total
  total += oldCount
```

C:	0	0	2	2	4	7	7

Haladjunk végig az inputon és rakjuk arra az indexre, ahol a segédtömbben szerepel, majd növeljük a számláló értékét.

```
for x in input:
  output[count[key(x)]] = x
  count[key(x)] += 1
```



A Counting sort $O(n+k)$ időigénnyel alkalmazható láncolt listák esetén is úgy, hogy a leszámláláshoz ettől függetlenül tömböt használunk.

Kiterjeszthető olyan intervallumokra is, amikor nem 0 az alsó értékhatár. Hogyan módosítanánk a 0-tól eltérő alsó határ kezelésére az algoritmust?

Bináris kupac

- **Bináris fa:** Olyan fa, ahol minden csúcsnak legfeljebb 2 gyereke van
- **Bináris kupac:** majdnem teljes bináris fa, amely minden szintjén teljesen kitöltött kivéve a legalacsonyabb szintet, ahol balról jobbra haladva egy adott csúcsig vannak elemek
- A fát egy tömbben reprezentáljuk, minden elem a szint szerinti bejárás szerinti sorszámának megfelelő eleme a tömbnek.
- A kupacot reprezentáló A tömbhöz két értéket rendelünk: $hossz(A)$ a tömb mérete, $kupacmeret(A)$ a kupac elemeinek száma.
- A kupac gyökere $A[0]$, a szerkezeti kapcsolatok egyszerűen számolhatóak:
  - $A[i]$ bal fia $A[2i+1]$
  - $A[i]$ jobb fia $A[2i+2]$
  - $A[i]$ apja $A[\lfloor (i-1)/2 \rfloor]$
- A kupac minden gyökértől különböző elemére teljesül, hogy az értéke nem lehet nagyobb, mint az apjáé. Ennek következménye, hogy a kupac minden részfájára teljesül, hogy a gyökéreleme maximális.

Heap Sort

- **Algoritmus:**
  $HeapSort(A[1,\ldots,n])$
  1. Építsünk egy maximum kupacot az inputból
  2. Ekkor a legnagyobb elem a kupac gyökerében van. Cseréljük ezt ki a kupac utolsó elemével, és hívjuk meg az eljárást az eggyel kisebb méretű kupacra. Ha a kupactulajdonság sérül az új gyökér miatt, állítsuk helyre.
  3. Ismételjük a fenti lépéseket, amíg a kupac mérete nem lesz 1
- **Időigény:** $O(n \log n)$
- **Tárigény:** $O(1)$
- Nem stabil rendezés
- Helyben rendez

3. Feladat Rendezzük Heap sort algoritmussal az $A = [5, 14, 13, 8, 3, 4, 6, 2]$ tömböt!

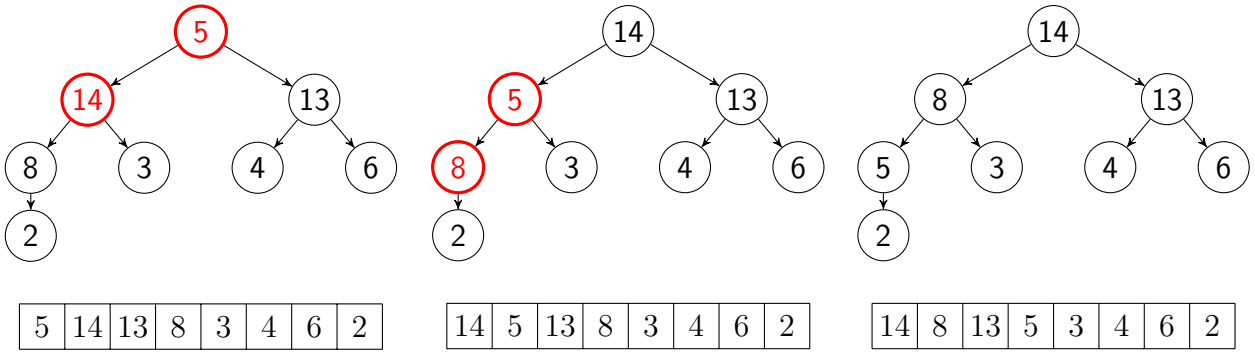
Megoldás

Rendezzük kupacba: Építsük meg a fát. Az $A[0]$ lesz a gyökér. A két fia az $A[2 \cdot 0 + 1] = A[1] = 14$ és $A[2 \cdot 0 + 2] = A[2] = 13$ és így tovább. Pl a 14 gyerekei lesznek az $A[2 \cdot 1 + 1] = A[3] = 8$ és $A[2 \cdot 2 + 2] = A[4] = 3$. Majd állítsuk helyre a kupactulajdonságot a legalsó részfáktól kezdve.

Paraméterek: $hossz(A) = 8$ és $kupacmeret(A) = 8$.

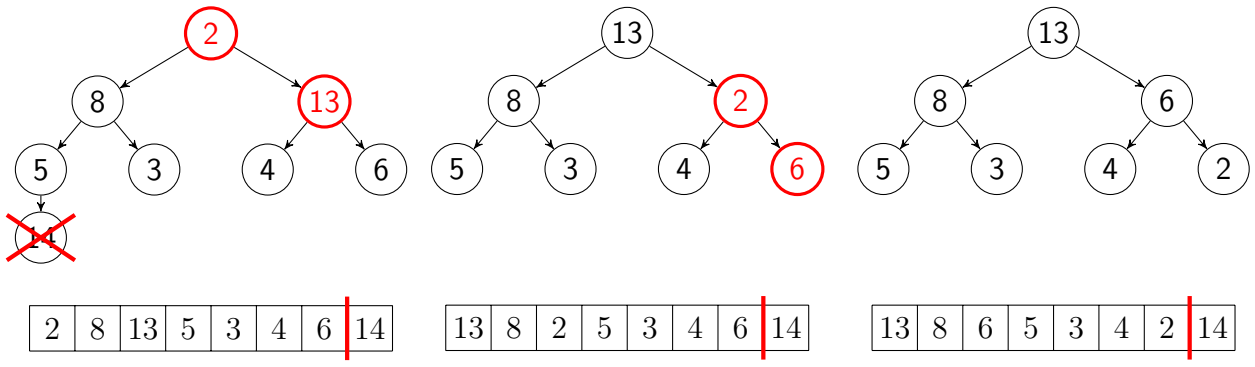
Kupactulajdonság helyreállítása:

- Hasonlítsuk össze a gyökér és a két gyereke értékét
- Ha a gyökér értéke a legnagyobb, nincs további dolgunk
- Ha közülük a legnagyobb a jobb gyerek értéke, akkor cseréljük meg a gyökér és a jobb gyerek értékét, majd rekurzívan állítsuk helyre a jobb részfát
- Ha a bal gyerek értéke a legnagyobb, cseréljük vele, majd rekurzívan állítsuk helyre a bal részfát

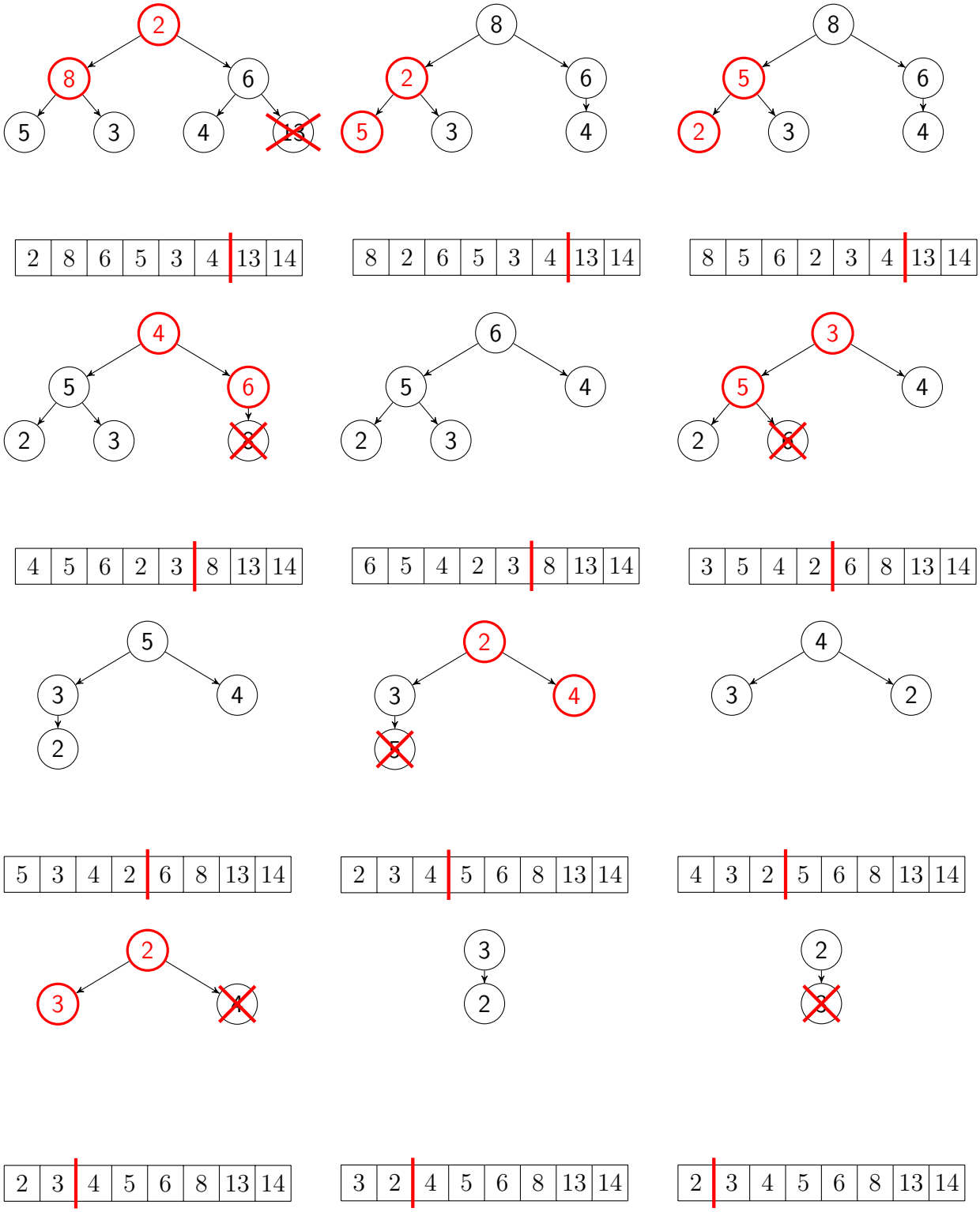


A kapott kupacban minden részfa teljesítette a kupactulajdonságot (hogy a részfájában a gyökér a legnagyobb értékű csúcs). Egyedül a gyökérben lévő elem sértette meg.

Ezután a kupac gyökerét cseréljük ki a kupac utolsó elemével. Így a legnagyobb elem a helyére kerül a tömbben. Töröljük a kupac utolsó elemét. Csökkentsük a kupac méretét eggyel, és állítsuk helyre a kupactulajdonságot az új kupacunkban. $kupacmeret(A) = 7$.



Rakjuk a tömbben a helyére a 13-at, töröljük a kupacból. $kupacmeret(A) = 6$. Állítsuk helyre a kupactulajdonságot.



4. Feladat Döntsd el, melyik rendezőalgoritmust használnád, ha az alábbiakat tudod:

1. Egy tömb 1000000 elemű és $0 - 100$ közötti elemeket tartalmaz.
2. Egy tömb 1000000 elemű és $0 - 1000$ közötti elemeket tartalmaz és helyben szeretnénk rendezni.
3. Egy láncolt lista 100000 elemet tartalmaz. Az elemek intervallumáról nem tudunk semmit.
4. Egy tömb 1000 elemet tartalmaz és az elemek intervalluma $[-200000, 800000]$
5. Egy tömb 10000 elemet tartalmaz, az elemek intervallumáról nincs információnk, viszont fontos, hogy a megegyező elemek az eredeti sorrendjükhöz képest ne változzanak a rendezett listában
6. Egy láncolt lista 1000 elemet tartalmaz. A tárolt adatok intervalluma $[-10, 10]$.

A választ indokold!

Megoldás

1. Mivel a tömb méretéhez képest a benne tárolt elemek intervalluma kicsi, Counting sort-ot érdemes használni

2. Az előbbi szintén igaz, de a Counting sort külső rendezés. Tömbökön jól működik például egy Quick sort.

3. Mivel a tárolt elemekről semmit nem tudunk, láncolt listára a legjobb választás a Merge sort.

4. A tömb méretének négyzete a benne tárolt elemek intervalluma. Ebben az esetben nem éri meg Counting sort-ot használni. Viszont mivel tömbben dolgozunk jól működik a Quick sort vagy a Heap sort.

5. Meg kell tartanunk az egyforma elemek egymáshoz viszonyított eredeti sorrendjét, így csak stabil rendezések jöhetnek szóba. Mivel az elemek intervallumáról nincs információnk, a Merge sortot érdemes alkalmazni.

6. A tárolt elemek intervalluma a lista méretéhez képest kicsi. A Counting sort láncolt listával is $O(n)$ időigényű, tehát szintén használhatjuk.

Érdekesség: ha tudni akarod, hogyan lehet könnyen kiszámolni bizonyos rekurzív algoritmusok (mint pl a MergeSort vagy a FastFourier algoritmus) futásidejét és mi is az a Mester tétel, kattints a nyuszira: