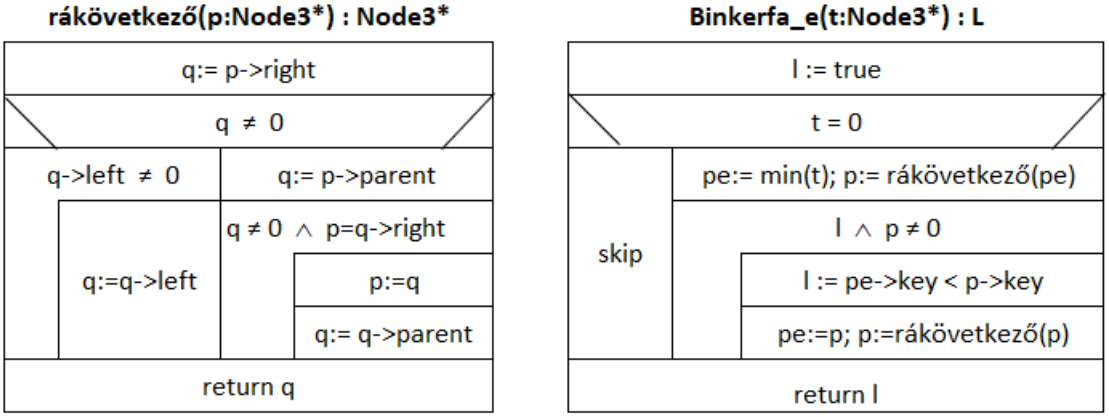
9. Gyakorlat

Témák:

- bináris keresőfákhoz kapcsolódó algoritmusok (ha idő szűkében vagyunk, ezeket kihagyhatjuk, egyedül a „rákövetkező” vizsgaanyag, így azt lehetőleg vegyük az órán),
- kupac fogalma, és két fontos művelete a kupac tulajdonság fenntartására: süllyesztés és emelés,
- prioritásos sor megvalósítása kupaccal,
- kupacrendezés.

Bináris fák algoritmusai

Feladat: készítsünk eljárást mely a bináris keresőfa tetszőleges pontjáról indulva megadja a rendezettség szerinti rákövetkező elemet (ha van), majd ennek segítségével készítsünk iteratív algoritmust, mely eldönti egy bináris fáról, hogy bináris keresőfa-e. A fát láncoltan ábrázoltuk, Node3 típussal, azaz szülő pointer is van.
Megjegyzés: a rákövetkező elemet megadó algoritmust megtaláljuk a jegyzetben: „inorder_next” néven. Ha a teljes feladatot nem is oldjuk meg, ezt mindenképp érdemes elővenni, megbeszélni, egy példán végig nézni.



Feladat: a bináris keresőfa alakja nagyon el tud romlani, egyes ágai túl hosszúra nőhetnek (akár listává torzulhat a fa), így elveszíti a benne történő keresés a hatékonyságát. Erre a problémára tudnak megoldást adni az „önkiegyensúlyozó” bináris keresőfák, például az AVL fák vagy a piros-fekete fák. A következő félévben fogjuk tanulni az AVL fákat.
Érdekes viszont a következő ötlet: ha nagyon elromlik a fa alakja, járjuk be inorder bejárással, írjuk egy tömbbe a rekordokat, majd az így kapott szigorúan monoton növekvően rendezett tömbből építsük fel újra bináris keresőfát úgy, hogy alakja optimális legyen.
Megoldás: építsük fel rekurzívan a következő módon: gyökérnek vegyük a tömb középső elemét, majd a tőle balra és jobbra lévő elemekből hasonló módon építsünk bináris keresőfát, és csatoljuk be a fákat a gyökér alá. Ezt folytatjuk rekurzívan, míg a levelekig nem érünk.
Láttuk, hogy bináris keresőfák esetén szükség lehet a szülő pointerre is, így a fát Node3 elemekből építjük, úgy, hogy a szülő pointert is beállítjuk.

BinkerfaÉpít(A: T[], e:N, u:N) : Node3*

k:=⌊ (e+u)/2 ⌋	
p:= new Node3(A[k],0)	
e < k	
p->left:=BinkerfaÉpít(A,e,k-1)	// p->left:=0 skip
p->left->parent:= p	
k < u	
p->right:=BinkerfaÉpít(A,k+1,u)	// p->right:=0 skip
p->right->parent:= p	
return p	

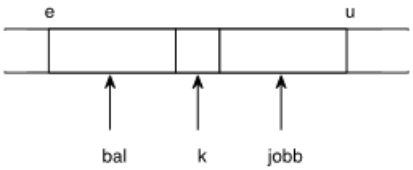
Hívása: **t=BinkerfaÉpít(A,1,A.M)**

Node3 konstruktora olyan elemet hoz létre, melynek kulcsa: A[k],
left és *right* pointerei nullák, a *parent* pointert megadhatjuk, itt most 0-val hívjuk:

p ->

0		
0	A[k]	0

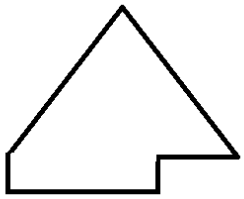
Ha e=k, vagy k=u, akkor a most létrehozott csúcsnak nem lesz bal-, illetve jobb részfája, a pointert nullára állíthatjuk, de felesleges, mert a konstruktor ezt már megtette.
A felezés ábrája:



Kupac fogalma

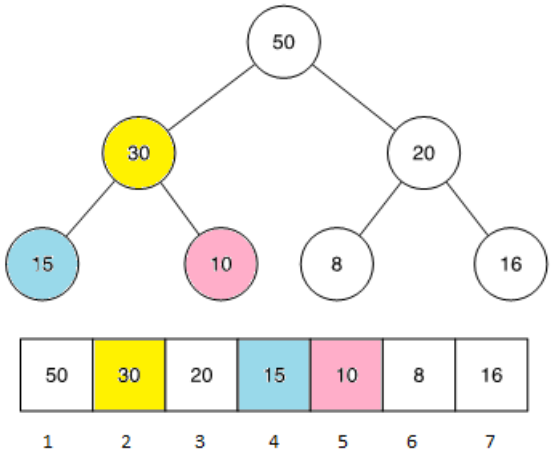
Ismételjük át a hallgatókkal a következő fontos definíciókat:
Definíciók:

- **szigorúan bináris fa:** a fa minden belső pontjának két gyereke van.
- **teljes bináris fa:** olyan szigorúan bináris fa, ahol minden levél azonos szinten helyezkedik el.
- **majdnem teljes bináris fa:** olyan teljes bináris fa, melynek legalsó (levél) szintjéről elhagytunk néhány levelet (de nem az összeset).
- **majdnem teljes balra tömörített bináris fa**: majdnem teljes bináris fa, de levelek csak a legalsó szintről, a jobb oldalról hiányozhatnak. Ezeket *szintfolytonos* fáknak is nevezzük.
- **kupac:** *maximum* vagy *minimum* kupac lehet. Maximum kupac: egy majdnem teljes, balra tömörített bináris fa, melynek minden belső pontjára teljesül, hogy a belső pont kulcsa nagyobb vagy egyenlő a gyerekei kulcsánál.
  Így kupac tetején (a fa gyökerében) mindig az egyik legnagyobb elem található.
  Minimum kupac hasonlóan: a szülő kulcs kisebb vagy egyenlő a gyerekei kulcsánál.



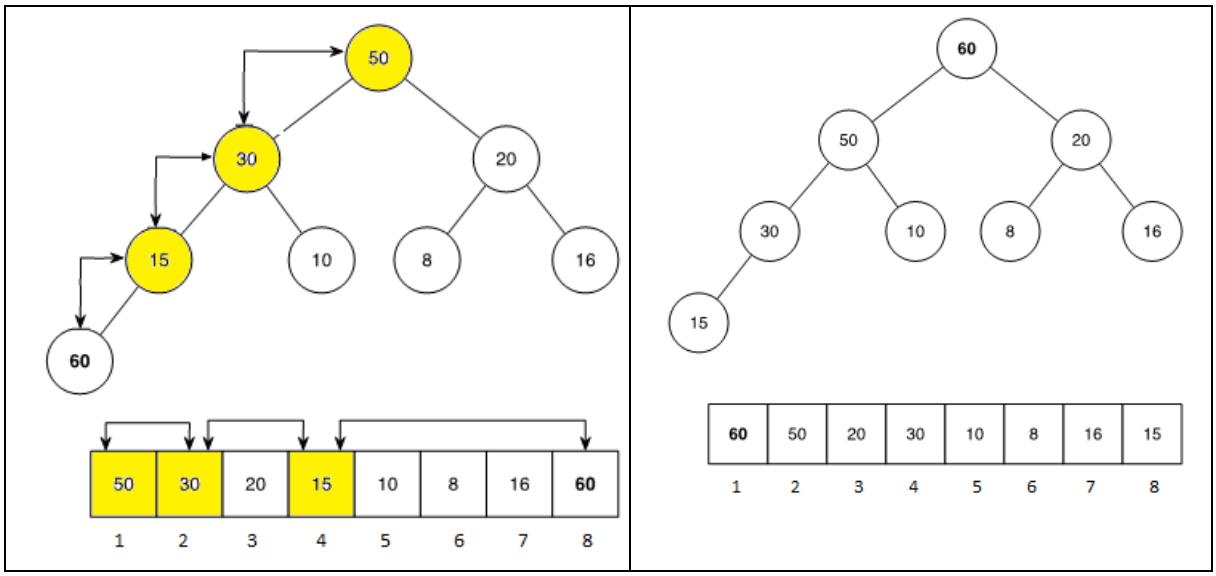
Kupac ábrázolása

A szintfolytonos bináris fákat, így a kupacokat is tömbbel ábrázoljuk. (Szokás ezt az ábrázolást *„bináris fák aritmetikai ábrázolásának”* is nevezni.) A szintfolytonos elhelyezés következménye, hogy a fában való navigálás a tömb indexeinek segítségével történik.
Ha a tömböt 1-től indexeljük, akkor:
Legyen a csúcs indexe: *i*
Bal gyerekének indexe: *2*i*
Jobb gyerekének indexe: *2*i+1*
Szülő indexe: $\left\lfloor \frac{i}{2} \right\rfloor$ (i/2 alsó egész része)



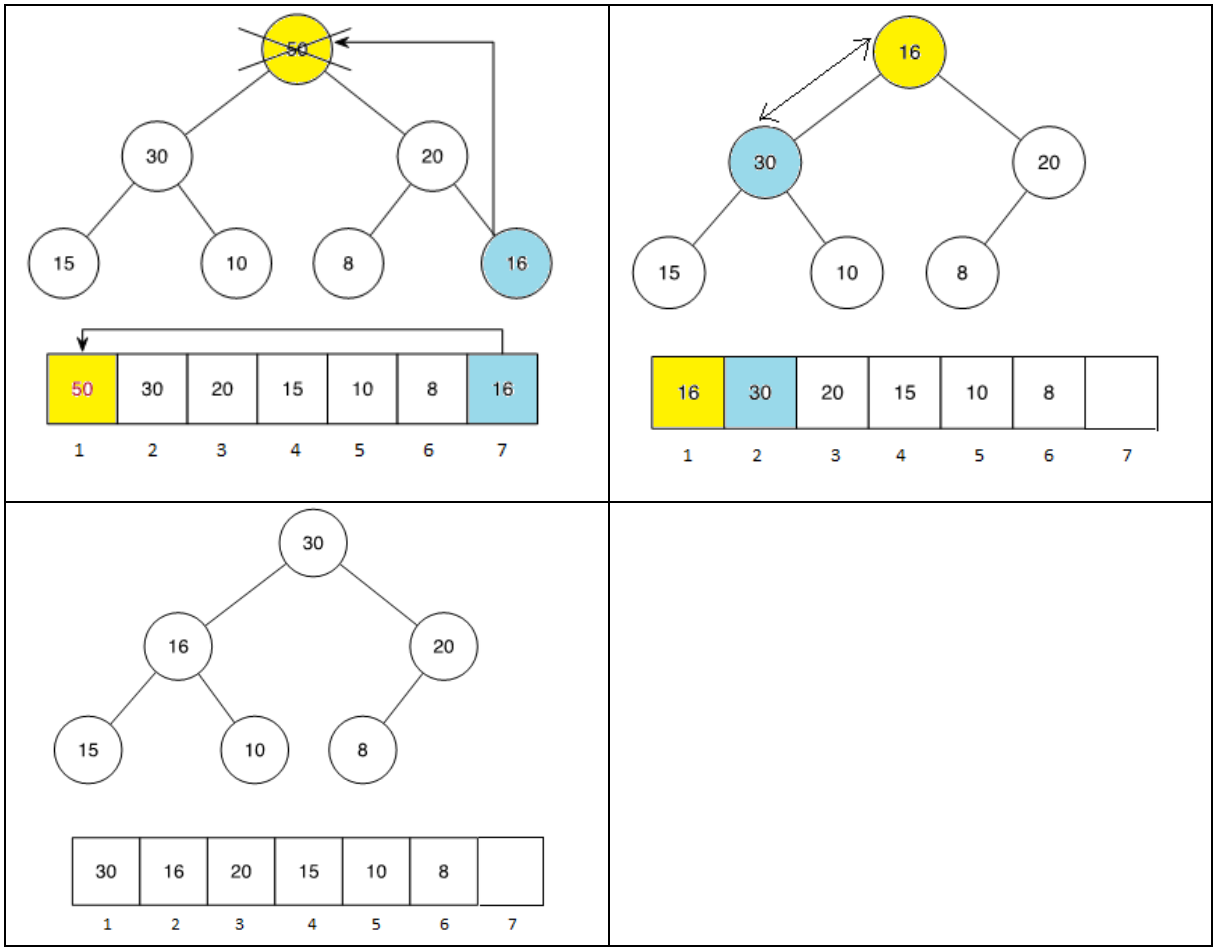
Kupac két fontos művelete: beszúrás, maximum törlése

Elem beszúrása a maximum-kupacba



60-as elem beszúrása: mivel a levelek szintje tele van, egy új szint keletkezik, és annak legbaloldalibb eleme lesz a 60. Majd az új elem addig emelkedik, míg a kupac tulajdonság helyre nem áll, azaz helyet cserél a szülőjével mindaddig, míg a beszúrt elem kulcsa nagyobb, mint a szülőjének kulcsa, vagy fel nem ér a kupac tetejére. Ez legfeljebb annyi cserét jelent, mint a kupac magassága, azaz $\lceil \log_2 n \rceil$.

A maximális kulcsú elem kivétele: gyökér elem törlése



Ha a maximális kulcsot eltávolítjuk, helyére a fa legalsó szintjének legjobboldalibb levele kerül, azaz a tömbben a kupachoz tartozó legutolsó elem (hogy a fa megtartsa balra-tömörítettégét). Ezután következik az úgynevezett süllyesztés: a kulcs addig süllyed lefelé a kupacban, míg kisebb, mint a nagyobbik gyereke (ha két gyereke van). Ezért az algoritmus kiválasztja a nagyobbik gyereket, és ha a süllyesztendő kulcs kisebb nála, akkor helyet cserélnek. A süllyesztés addig tart, míg a süllyesztendő nagyobb vagy egyenlő lesz, mint a nagyobbik gyereke, vagy leérünk a kupac aljára.

Prioritásos sor

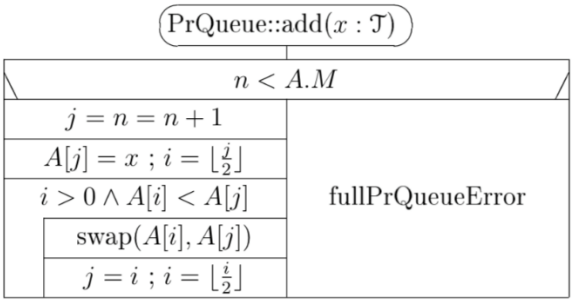
Beszélhetünk maximum-, vagy minimum prioritásos sorról. Maximum prioritásos sor esetén mindig a legnagyobb prioritású elemet tudjuk kivenni, a minimum prioritásos sor esetén pedig a legkisebbet. Mindkettő fajta ábrázolható kupaccal.
Itt most a maximum prioritásos sort fogjuk tanulmányozni, amit egy maximum kupaccal ábrázolunk.

Prioritásos sor UML osztály diagrammja

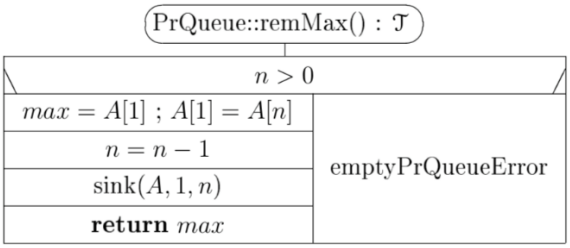
PrQueue
- $A : \mathcal{T}[]$ // $\mathcal{T}$ is some known type - $n : \mathbb{N}$ // $n \in 0..A.M$ is the actual length of the priority queue

+ PrQueue($m : \mathbb{N}$){ $A =$ **new** $\mathcal{T}[m]$; $n = 0$ } // create an empty priority queue
+ add($x : \mathcal{T}$) // insert x into the priority queue
+ remMax():$\mathcal{T}$ // remove and return the maximal element of the priority queue
+ max():$\mathcal{T}$ // return the maximal element of the priority queue
+ isFull() : $\mathbb{B}$ {**return** $n == A.M$}
+ isEmpty() : $\mathbb{B}$ {**return** $n == 0$}
+ $\sim$ PrQueue() { **delete** A }
+ setEmpty() {$n = 0$} // reinitialize the priority queue

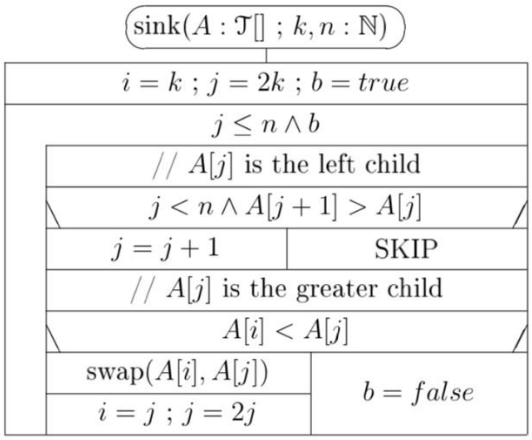
Műveletei kupac esetén:



Az add művelet emeléssel hajtódik végre: ha van még üres hely a tömbben beírjuk az új kulcsot az első szabad helyre, majd emelést hajtunk végre a kupacban.



A maximális elem kivétele után pedig a süllyesztő algoritmus állítja helyre a kupacot.



Az algoritmusok szerepeltek az előadáson, nem kötelező őket felírni!

Maximum prioritásos sor megvalósításainak összehasonlítása			
	add(x)	remMax()	max()
rendezetlen tömb *ha a maximális elem indexét nyilvántartjuk*	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
rendezett tömb *növekvően rendezett*	$O(n)$	$\Theta(1)$	$\Theta(1)$
maximum kupac	$O(\log n)$	$O(\lg n)$	$\Theta(1)$

Feladat:
Szemléltessük a kupaccal ábrázolt prioritásos sor műveleteit, például oldjuk meg a következő feladatot:
Egy **prioritásos sort** az A[1..15] elemű tömbben **kupaccal** ábrázoltunk. A tömb tartalma a következő: [40, 26, 27, 14, 21, 15, 2, 9, 6, 3, 8, 10, , ,].

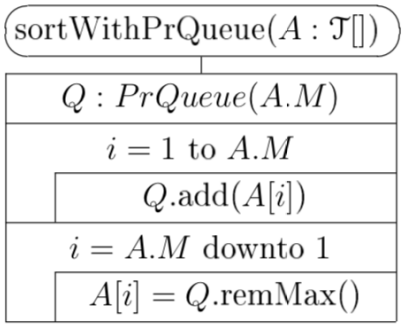
a) Hajtsuk végre a **add(x)** műveletet kétszer egymás után a következő elemekre: 61, 43
b) Hajtsuk végre a **remMax()** műveletet kétszer egymás után az eredeti kupacból kiindulva.

A műveletet szemléltessük a fa alakban lerajzolt kupacon és a tömbös alakban is.

A fenti ábrákhoz hasonlóan mutassuk be a kupacon végrehajtott műveleteket!

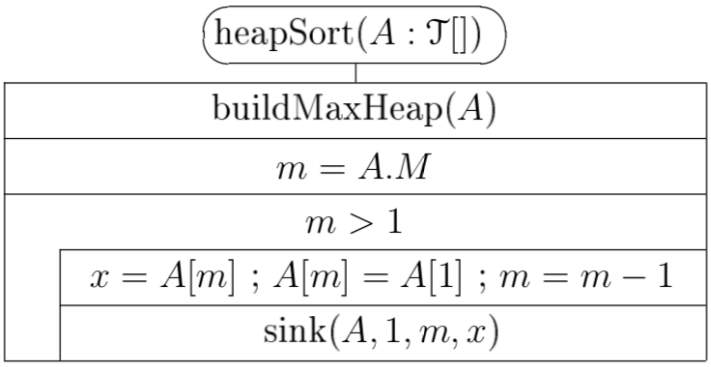
Rendezés elsőbbségi sorral

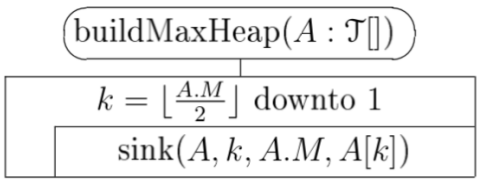
Egy rendezési ötlet: rakjuk be a kulcsokat egy prioritásos sorba, majd onnan kivéve tegyük vissza őket a tömbbe.



Megmutathatjuk a hallgatóknak, de hangsúlyozzuk, hogy nem ez a kupac rendezés, mert az a tapasztalat, hogy összekeverik! Gyakorlatban nem szokták ezt használni, mert plusz tárigény kell a prioritásos sor miatt, és az egyéb tanult n logn-es rendezők gyorsabbak. Az algoritmus kihagyható, nem olyan fontos, mint a most következő: a kupacrendezés.

Kupacrendezés (heap sort)



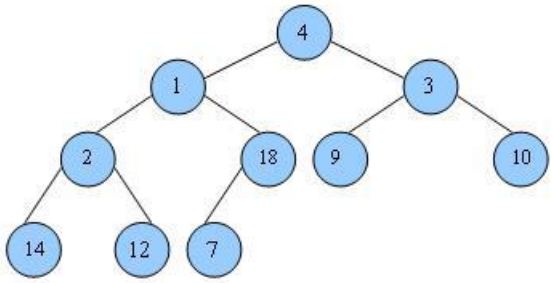


Feladat: a rendezés szemléltetése. Ehhez elsőként le kell játszani a kupac kialakításának meneteit. Nagyon fontos, hogy alulról felfelé (a legutolsó levél szülőjétől kezdve) süllyesztésekkel alakítsuk ki a kupacot! A vizsgán a süllyesztések mellé a sorszámukat is oda kell írni! Egy szinten történő süllyesztéseket lehet egy ábrán bemutatni. Minden szinthez készítsünk új ábrát. A legutolsó süllyesztést kövessük végig tömbben ábrázolva is!

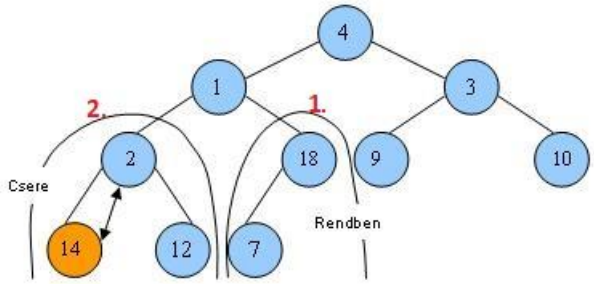
Például:
Mutassa be a kupacrendezést a következő tömbön:
[4, 1, 3, 2, 18, 9, 10, 14, 12, 7]

Kupac kialakításának bemutatása buildMaxHeap algoritmus menete:

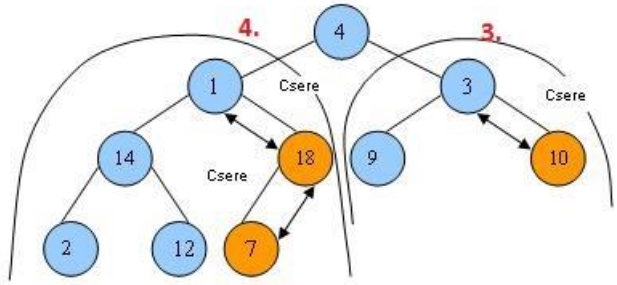
A tömb a következő szintfolytonos fát tartalmazza:



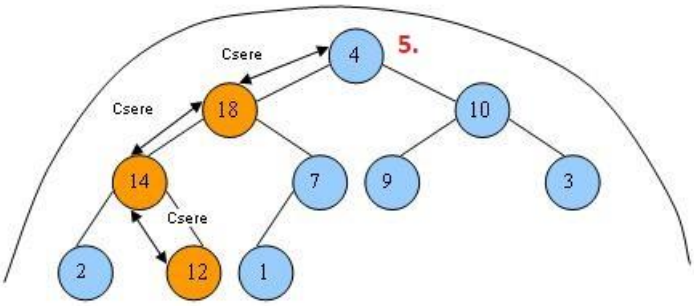
A hetes szülője a 18, ott kezdődik a kupaccá alakítás, majd a 2-es elem következik (félkövér, piros számok jelzik a süllyesztés sorszámát):



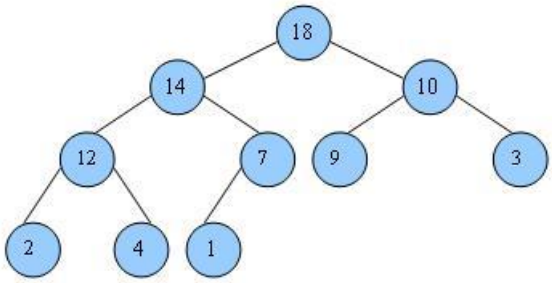
Eggyel feljebbi szinten folytatódik az algoritmus:



Majd felérve a kupac tetejére az utolsó süllyesztés:



A kész kupac:



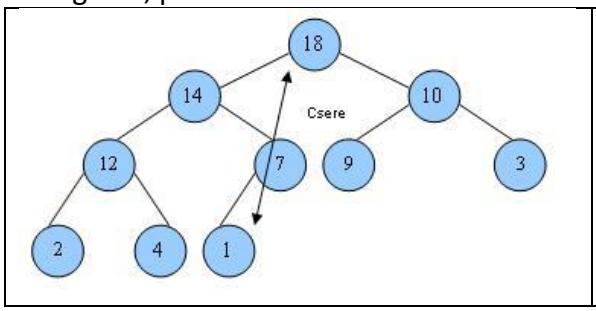
Az utolsó süllyesztés tömbön szemléltetve:

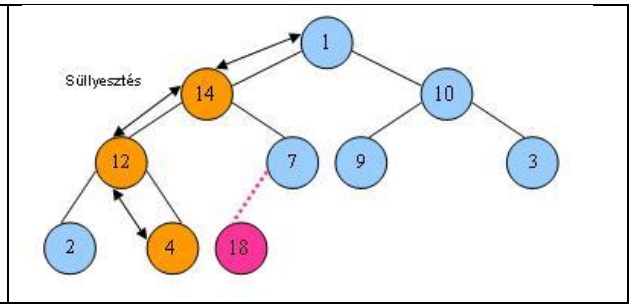
1	2	3	4	5	6	7	8	9	10
4	18	10	14	7	9	3	2	12	1
18	4	10	14	7	9	3	2	12	1
18	14	10	4	7	9	3	2	12	1
18	14	10	12	7	9	3	2	4	1

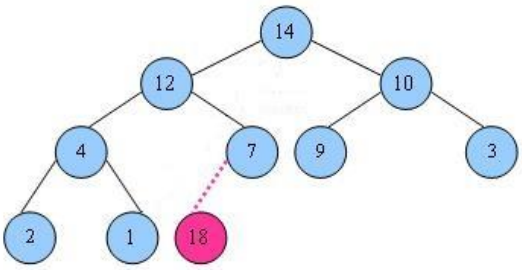
- *sárga az aktuális elem,*
- *kék a bal gyerek,*
- *zöld a jobb gyerek,*
- *a nagyobbik gyerek piros keretes,*
- *a cserét nyíl ábrázolja.*

Rendezés folytatása

Nem kell lejátszani végig, elég bemutatni két-három menetet. Itt is fontos, hogy világos legyen számukra, hogy a rendezés egy tömbön zajlik, tehát érdemes egy-egy menet végén a tömböt is megadni, például:



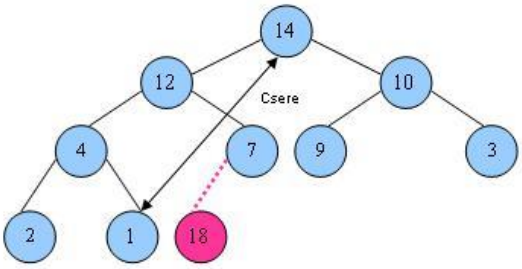


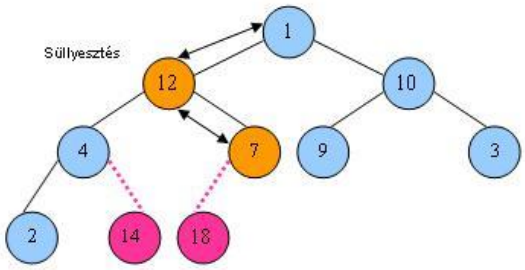


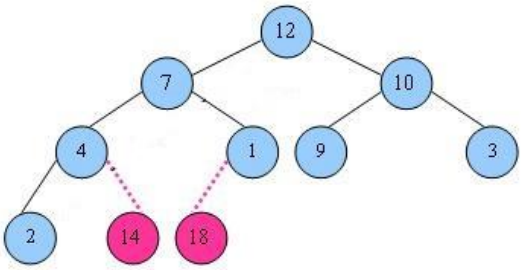
Első menet vége, 18 helyére került és a kupac helyre van állítva. A második legnagyobb elem, a 14 a kupac tetejére került. A 18-as elem a tömbben van, de már nem tartozik a kupachoz!

Tömbben ábrázolva:

1	2	3	4	5	6	7	8	9	10
14	12	10	4	7	9	3	2	1	18





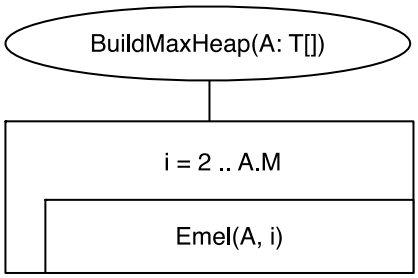


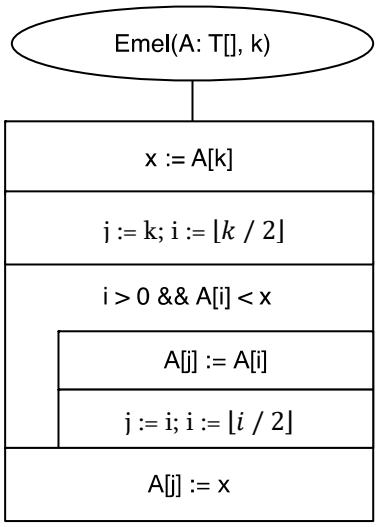
A tömb tartalma:

1	2	3	4	5	6	7	8	9	10
12	7	10	4	1	9	3	2	14	18

Esetleg a következő, harmadik menetet lejátszhatjuk a tömbben, anélkül, hogy a fán lejátszanánk.

ZH-kon, vizsgákon sokszor találkoztunk azzal a hibával, hogy a kezdőkupacot nem az algoritmusnak megfelelően alakítják ki: alulról felfelé haladva süllyesztéseket hívva az aktuális elemre , hanem emelésekkel (hasonlóan a prioritásos sor add műveletéhez) a tömb következő elemét „beemelik” a kupacba:



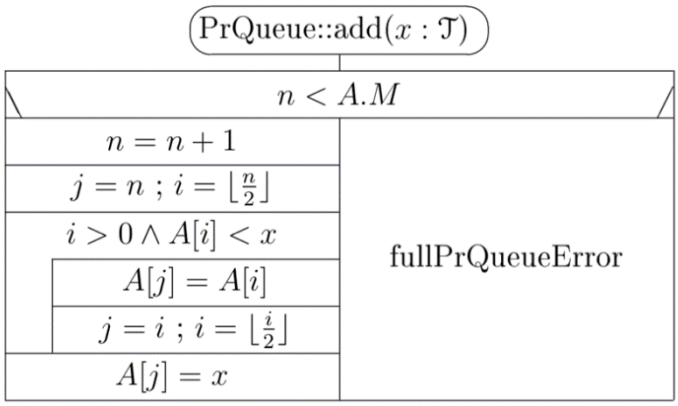


(Ezt így ne írjuk fel, mert megtéveszthet valakit, hogy így néz ki a kupacrendezés!!!!) Viszont az Emel az értékes algoritmus, felhasználható a lejjebb olvasható házi feladatnál.

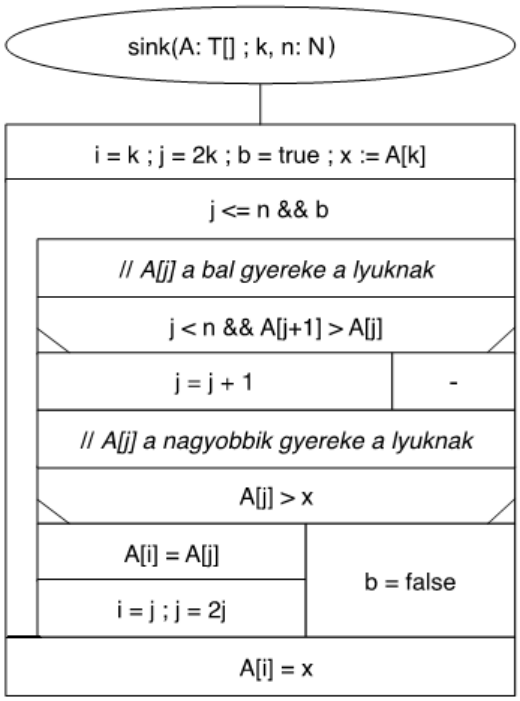
Érdemes megbeszélni, hogy **miért nem ezt használjuk:** süllyesztéssel a kezdőkupac kialakítása ($\Theta(n)$) (lásd jegyzet), ezzel a módszerrel viszont O(n log n)! Ugyanis a süllyesztésen alapuló módszernél, ott, ahol sok elem van (a levelek és levelekhez közeli szinteken), csak kis magasságú kupacba süllyesztünk, és ahogy felfelé haladunk, igaz, hogy egyre magasabb a csonka kupac, amibe süllyesztünk, de egyre kevesebb az elem. A beszúrásos módszernél ez épp fordítva igaz! A levelek és a levelekhez közeli szintek, tehát ahol már sok az elem egy szinten belül, egy magas kupacba emelkednek fel!

Javasolt házi feladat:

A bemutatott műveletek (emelés, és süllyesztés) cseréket használnak. Tudjuk, hogy minden csere három mozgatást jelent. Az algoritmusok hatékonyabbá tehetők, ha nem cseréket végeznek, hanem egy „lyukat" emelnek, vagy süllyesztenek. Például az emelés úgy néz ki, hogy kivesszük egy segédváltozóba az emelendő elemet, majd a lyukat visszük felfelé a kupacban, azáltal, hogy a szülőt, ha kisebb, betesszük a lyukba. Így a szülő helyére kerül a lyuk. (Ezért került az anyagban a fenti Emel nevű algoritmus, mert az pont így csinálja az emelést.) Például a prioritásos sor add művelete ezzel a módszerrel a következőképpen nézne ki.



Készítsék el hasonlóan a süllyesztés algoritmust.



Készült az „Integrált kutatói utánpótlás-képzési program az informatika és számítás-tudomány diszciplináris területein” című EFOP 3.6.3-VEKOP-16-2017-00002 azonosítójú projekt támogatásával.