9. Gyakorlat
Hallgatói segédanyag

Témák:

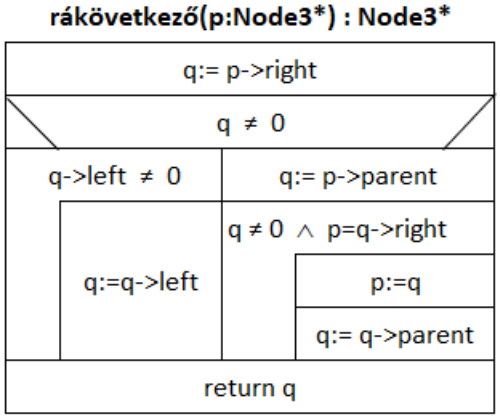
- bináris keresőfákhoz kapcsolódó algoritmusok,
- kupac fogalma, és két fontos művelete a kupac tulajdonság fenntartására: süllyesztés és emelés,
- prioritásos sor megvalósítása kupaccal,
- kupacrendezés.

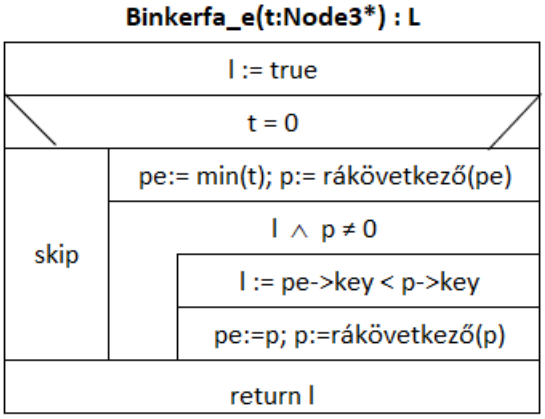
Bináris fák algoritmusai

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. Működése: ha az adott csúcsnak a jobb részfája nemüres, akkor a rákövetkező ennek a részfának a legkisebb kulcsa, azaz jobbra lépünk, majd „leszaladunk" a legbaloldalibb levélhez. Ha a jobb részfája üres, akkor az „ősök" között kell a rákövetkezőt keresni: addig megyünk a szülő pointereken felfelé, amíg elérjük az első olyan szülőt, akinek a kulcsa nagyobb, mint ahol épp járunk. Mivel a kulcsösszehasonlítás költséges lehet, a fa alakjából találjuk ki, melyik a megfelelő szülő: az első olyan, akinek bal részfájában vagyunk! Ha a legnagyobb kulcsú elemtől indítjuk a rákövetkező műveletet, akkor nem fogunk megfelelő szülőt találni, ezért ilyenkor azt is figyelni kell, hogy a szülőre mutató pointer nem NULL értékű.

Megoldás: A „rákövetkező" eljárás segítségével könnyen megoldhatjuk a feladatot. Két pointert vezetünk végig a fa csúcsain: *pe* és *p* pointereket, *p* mindig *pe* rákövetkezőjére mutat. Az algoritmus végig ellenőrzi, hogy *pe->key < p->key* fennáll-e. *(Optimista eldöntés programozási tétel, egy felsorolóra támaszkodva.)* Az algoritmusban a *pe* pointert a fa legkisebb kulcsú eleméről kell indítani, ezt most nem részletezzük, min(t) hívással jelezzük, min(t) lényegében a rákövetkező eljárás bal ágának felel meg.

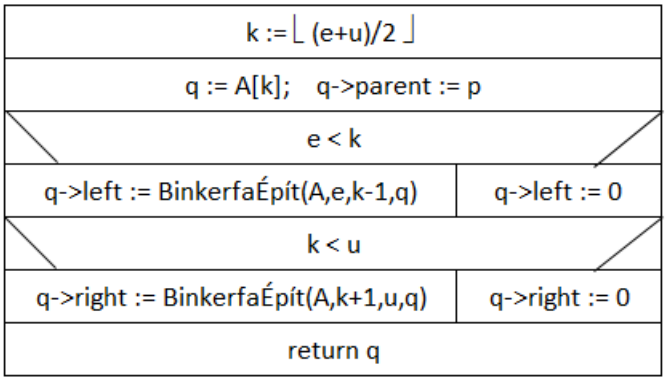




Feladat: a bináris keresőfa alakja nagyon el tud romlani, egyes ágai túl hosszúra nőhetnek (akár listává torzulhat a fa), így elveszíti a benne történő keresés a hatékonyságát. Erre a problémára tudnak megoldást adni az „önkiegyensúlyozó” bináris keresőfák, például az AVL fák vagy a piros-fekete fák. 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 csúcsok címeit, majd az így kapott, a csúcsok kulcsai szerint 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 eleme által hivatkozott csúcsot, 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:Node3*[]; e,u:N; p:Node3*) : Node3*



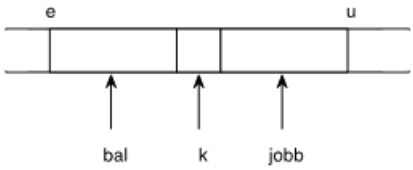
$e \le u$

A p pointer a szülő csúcsra mutat, vagy *NULL*, ha nincs szülő csúcs.

Ha e=k, vagy k=u, akkor a (*q) csúcsnak üres lesz a bal, illetve a jobb részfája: a pointert *NULL*-ra állítjuk.

Hívása: **t:=BinkerfaÉpít(A,0,A.length-1,0)**
(Feltesszük, hogy **A.length ≥ 1**.)

A felezés ábrája:

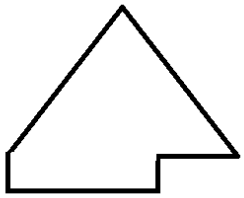


Kupac fogalma

A kupac fogalmához ismételjük át az előadáson tanult 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.

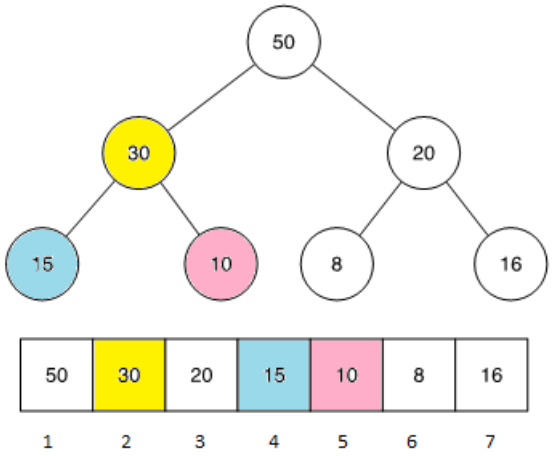
- **kupac:** *maximum* vagy *minimum* kupac lehet. Maximum kupac: egy majdnem teljes, balra tömörített bináris fa, melynek minden belső pontjára teljesül, hogy a belső pont kulcsa nagyobb vagy egyenlő a gyerekei kulcsánál.
  Így kupac tetején (a fa gyökerében) mindig az egyik legnagyobb elem található.
  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:

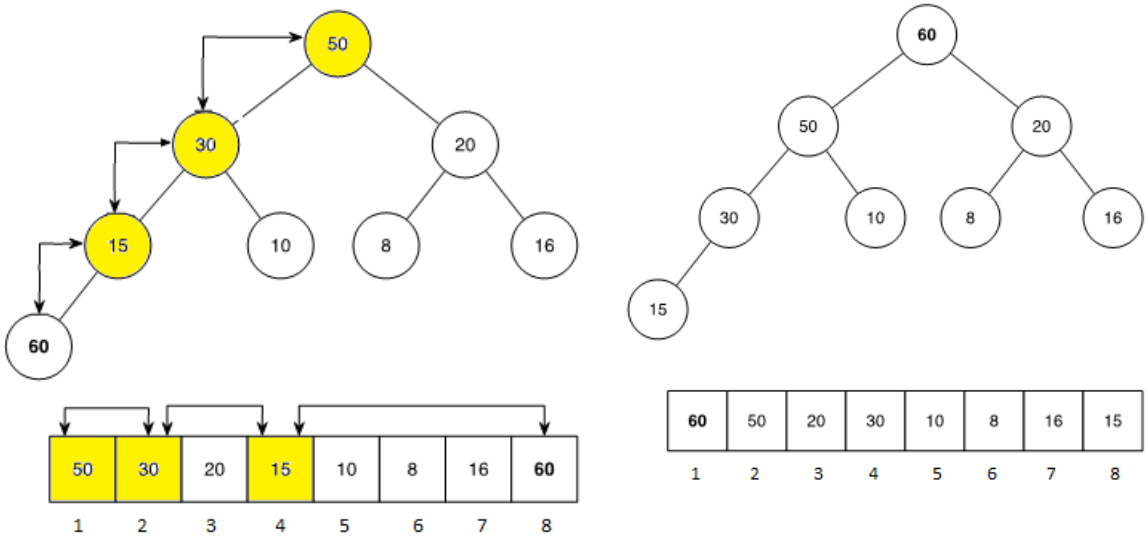
Legyen a csúcs indexe:	*i*
Bal gyerekének indexe:	*2*i*
Jobb gyerekének indexe:	*2*i+1*
Szülő indexe:	$\left\lfloor \frac{i}{2} \right\rfloor$ (i/2 alsó egész része)



Kupac két fontos művelete: beszúrás, maximum törlése

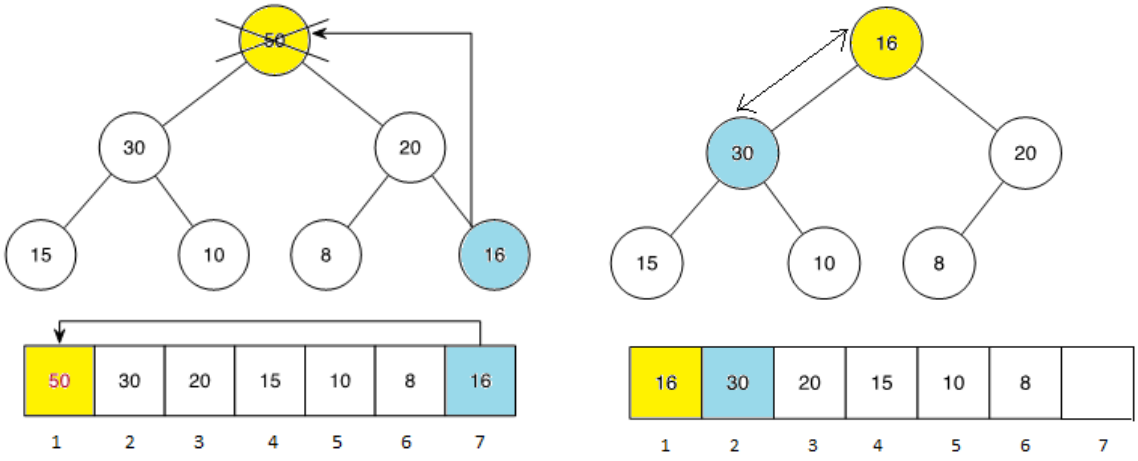
- A műveleteknek fenn kell tartania a kupac tulajdonságait: az alakjára, és a benne tárolt kulcsokra vonatkozó feltételeket.
- A kupac általában nem teljesen foglalja el az őt ábrázoló tömböt: 1 től egy adott indexű elemig tárolja a kupac elemeit, utána „üres” helyek vannak, melyet elfoglalhat a kupac elem beszúrásakor.
- **Beszúrás:** az új elemet a tömbben a kupac utolsó eleme után helyezzük el, így alakja továbbra is balra tömörített lesz, majd az új elemet addig emeljük a gyökér felé, amíg helyre áll a kulcsokra vonatkozó feltétel (szülő nagyobb vagy egyenlő mindkét gyerekénél.)

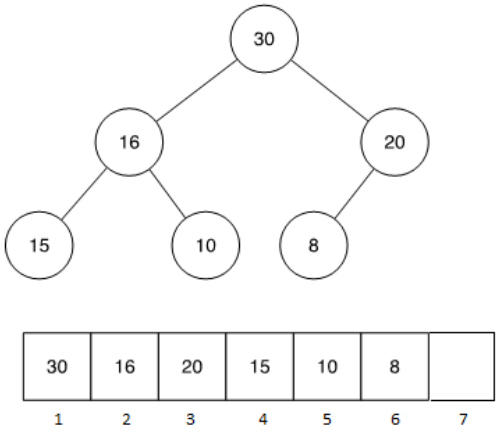
Elem beszúrása a maximum-kupacba



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 $\lfloor \log_2 n \rfloor$.

A maximális kulcsú elem kivétele: gyökér elem törlése





Maximális elem kivétele: a maximális kulcsot eltávolításakor, helyére a fa legalsó szintjének legjobboldalibb levele kerül, azaz a tömbben a kupachoz tartozó legutolsó elem (hogy a fa megtartsa balra-tömörítettségét). 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

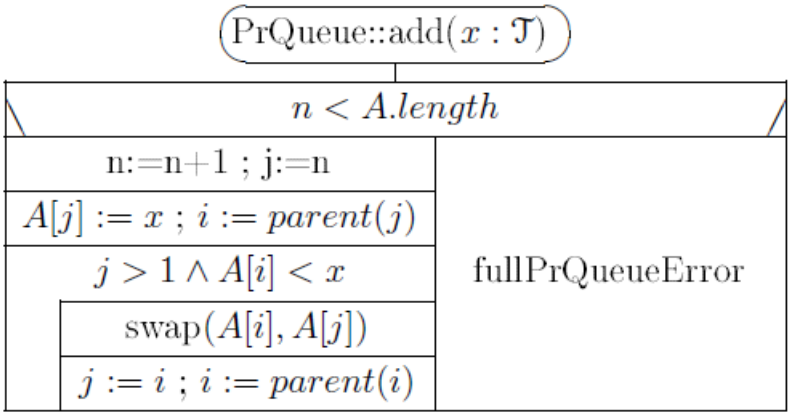
Beszélhetünk maximum-, vagy minimum prioritásos sorról. Maximum prioritásos sor esetén mindig a legnagyobb prioritású elemet tudjuk kivenni, a minimum prioritásos sor esetén pedig a legkisebbet. Mindkettő fajta ábrázolható kupaccal.
Itt most a maximum prioritásos sort fogjuk tanulmányozni, amit egy maximum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

Prioritásos sor UML osztály diagrammja

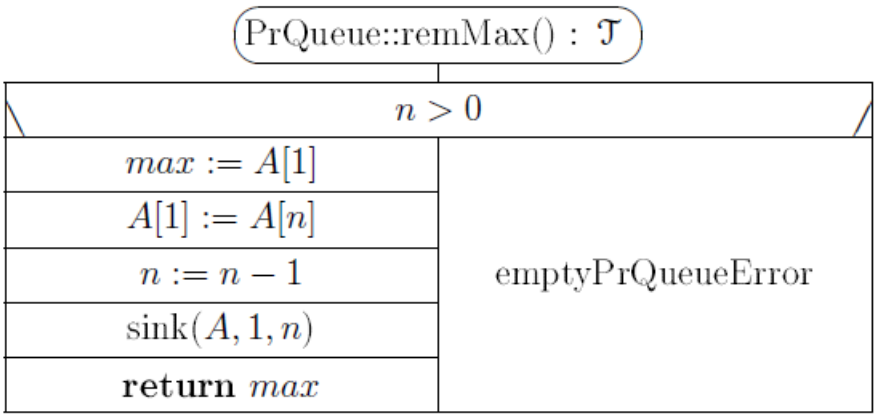
PrQueue
- $A/1 : \mathcal{T}[]$ // $\mathcal{T}$ is some known type - $n : \mathbb{N}$ // $n \in [0..A.length]$ is the actual length of the priority queue

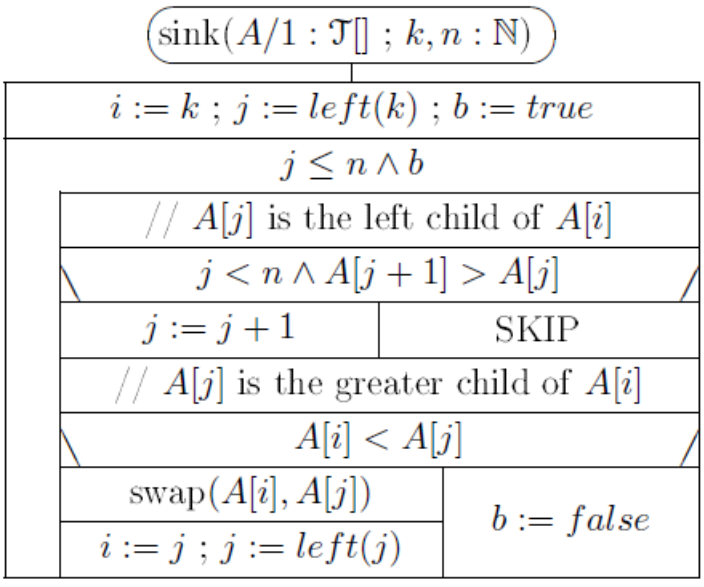
+ PrQueue($m : \mathbb{N}$){ $A :=$ **new** $\mathcal{T}[m]$; $n := 0$ } // create an empty priority queue
+ add($x : \mathcal{T}$) // insert x into the priority queue
+ remMax():$\mathcal{T}$ // remove and return the maximal element of the priority queue
+ max():$\mathcal{T}$ // return the maximal element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

Műveletei kupac esetén (a jegyzetben megtalálhatjuk őket):



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.

Maximum prioritásos sor megvalósításainak összehasonlítása			
	add(x)	**remMax()**	**max()**

rendezetlen tömb *ha a maximális elem indexét nyilvántartjuk*	$\Theta(1)$	$\Theta(\mathrm{n})$	$\Theta(1)$
rendezett tömb *növekvően rendezett*	$\mathrm{O}(\mathrm{n})$	$\Theta(1)$	$\Theta(1)$
maximum kupac	$\mathrm{O}(\log n)$	$\mathrm{O}(\lg n)$	$\Theta(1)$

Feladat:

Szemléltessük a kupaccal ábrázolt prioritásos sor műveleteit, oldjuk meg a következő feladatot:

Egy **prioritásos sort** az A[1..15] elemű tömbben **kupaccal** ábrázoltunk. A tömb tartalma a következő: [40, 26, 27, 21, 14, 15, 2, 9, 6, 3, 8, 10, , ,].

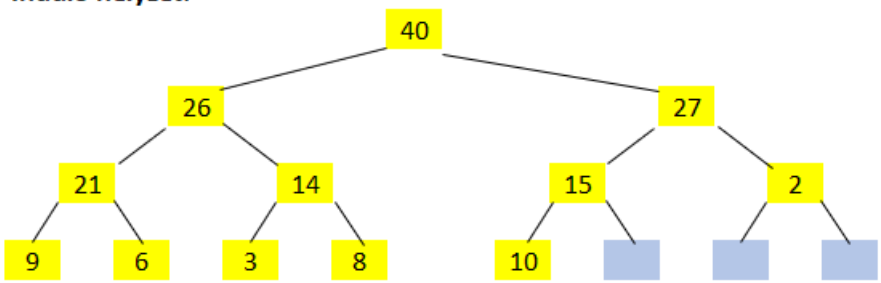
a) Hajtsuk végre a **add(x)** műveletet kétszer egymás után a következő elemekre: 61, 43
b) Hajtsuk végre a **remMax()** műveletet kétszer egymás után az eredeti kupacból kiindulva.

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

Megoldás:

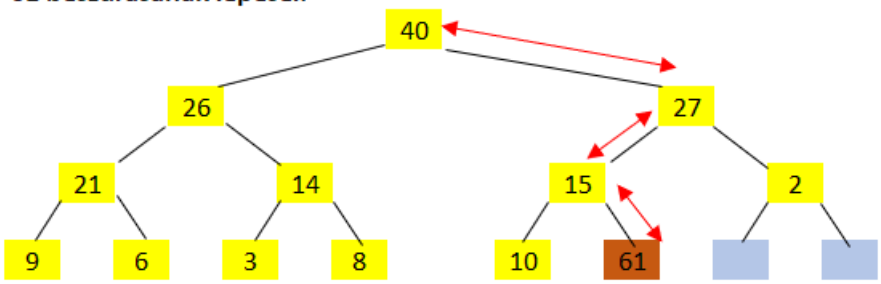
61 beszúrása részletesen szemléltetve:

Induló helyzet:



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
40	26	27	21	14	15	2	9	6	3	8	10			

61 beszúrásának lépései:



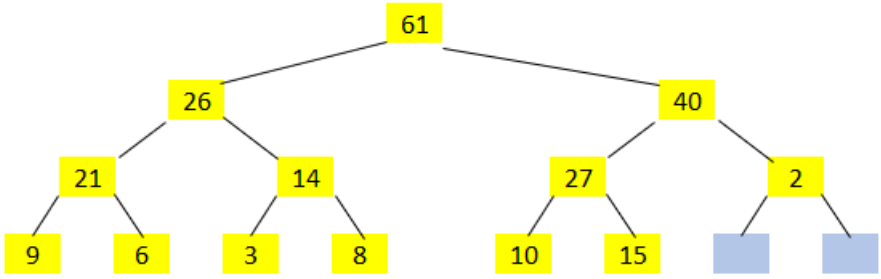
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
40	26	27	21	14	15	2	9	6	3	8	10	61		

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
40	26	27	21	14	61	2	9	6	3	8	10	15		

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
40	26	61	21	14	27	2	9	6	3	8	10	15		

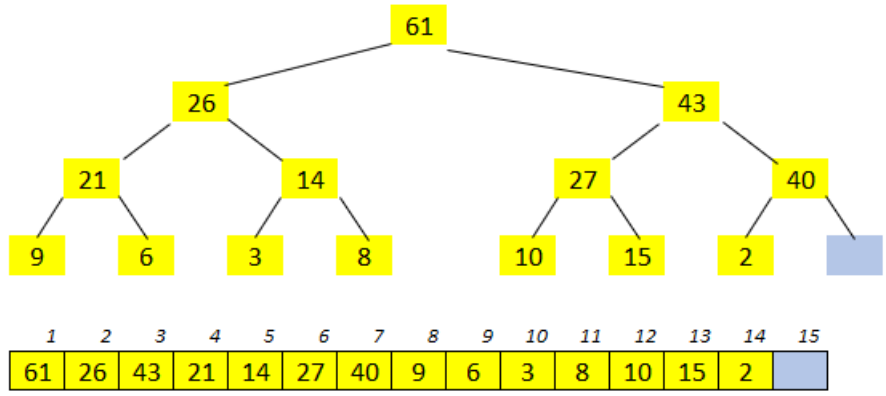
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
61	26	40	21	14	27	2	9	6	3	8	10	15		

Beszúrás után a kupac:



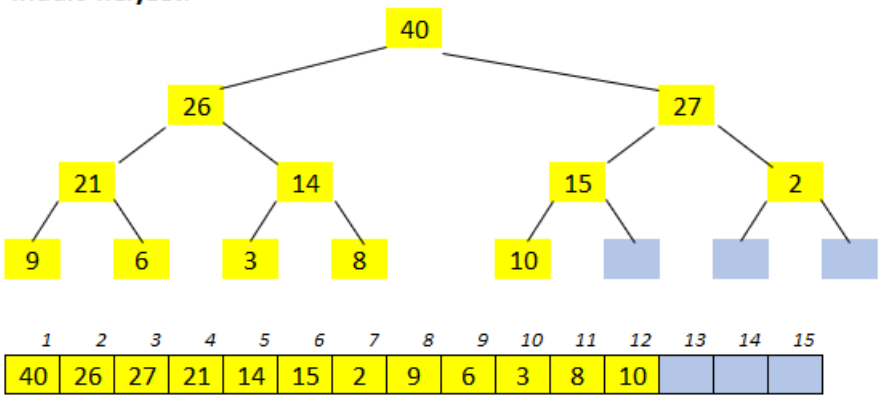
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
61	26	40	21	14	27	2	9	6	3	8	10	15		

43-as kulcs beszúrásának szemléltetését az olvasóra bízzuk, itt csak a végeredményt adjuk meg:

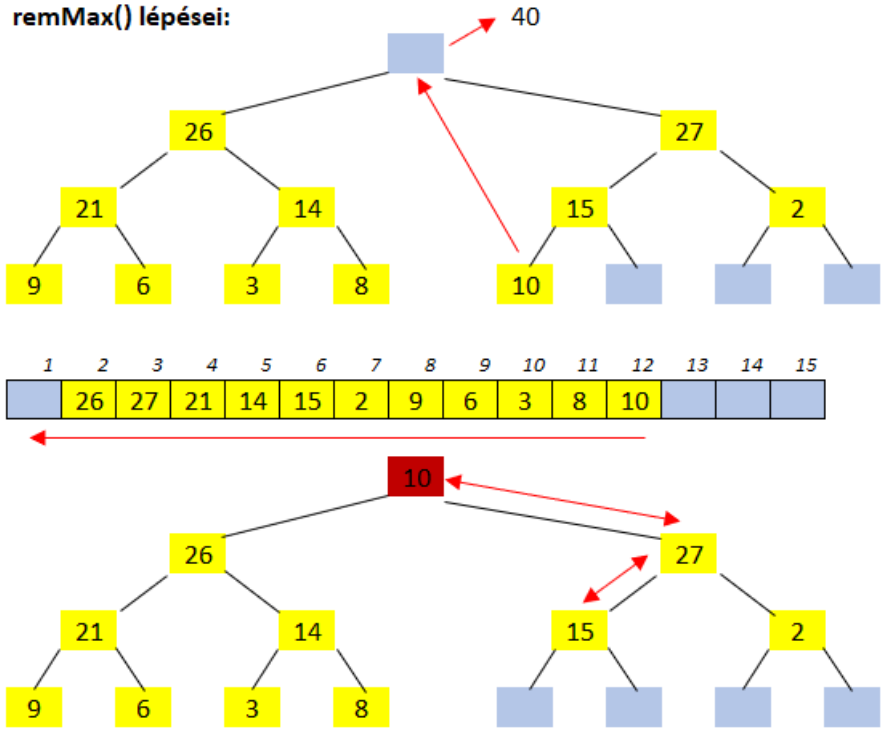


Nézzük meg, hogyan történik a remMax() művelet. Az eredeti kupacra kell kétszer egymás után elvégezni.

Induló helyzet:

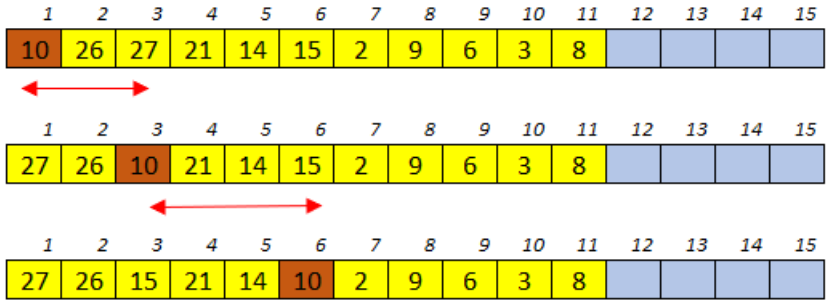


remMax() lépései:

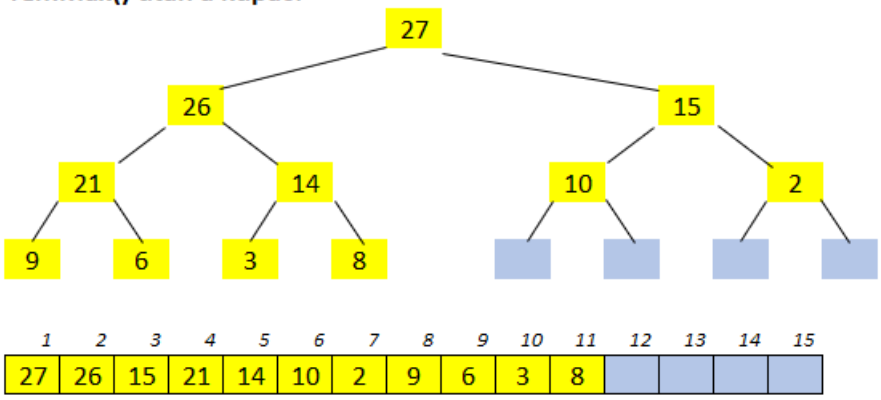


Eltávolítottuk a legnagyobb kulcsú elemet. A kupac tetejére felhelyezzük a kupac elhelyezés szerinti legutolsó elemét, majd azt belesüllyesztjük a kupacba.

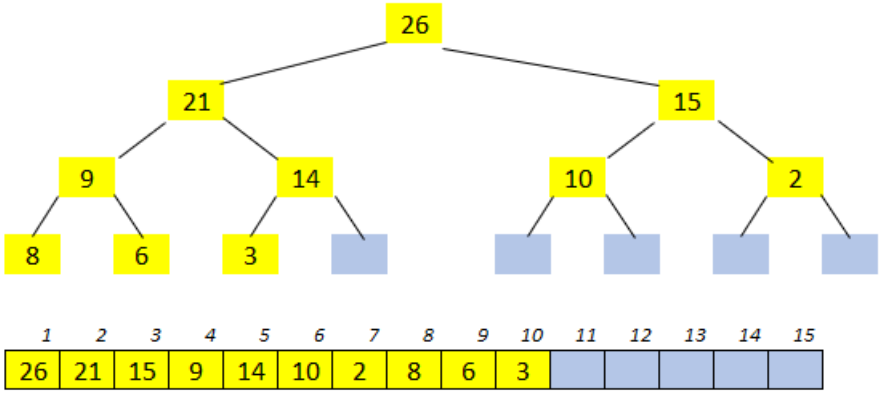
Süllyesztés még egyszer, a tömbben ábrázolva:



remMax() után a kupac:

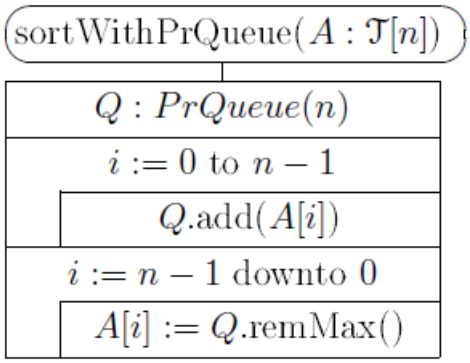


Ezen a kupacon meg kell ismételni a remMax() műveletet. Ennek ábrázolását az olvasóra bízzuk, itt most csak a végeredményt adjuk meg:



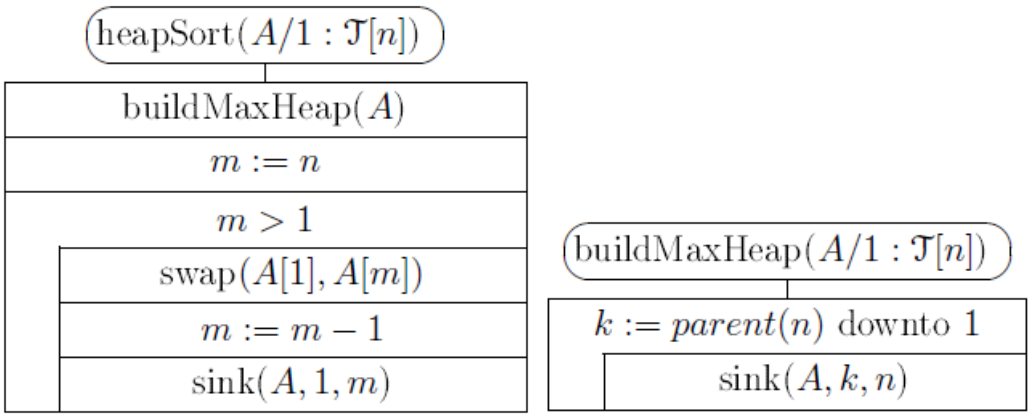
Rendezés elsőbbségi sorral

Egy rendezési ötlet (algoritmusát a jegyzetben megtalálhatjuk): rakjuk be a kulcsokat egy prioritásos sorba, majd onnan kivéve tegyük vissza őket a tömbbe. Vizsgáljuk meg a kapott algoritmust, mennyire hatékony, érdemes-e a gyakorlatban alkalmazni?



*Gyakorlatban nem szokták ezt használni, mert plusz tárigény kell a prioritásos sor miatt, és az egyéb tanult (n*logn)-es rendezők gyorsabbak. Gyakorlatban helyette a kupacrendezést használják, nézzük meg ennek algoritmusát. Ez helyben rendez, és a kezdeti kupacépítés gyorsabb.*

Kupacrendezés (heap sort), algoritmusát a jegyzetben megtalálhatjuk:



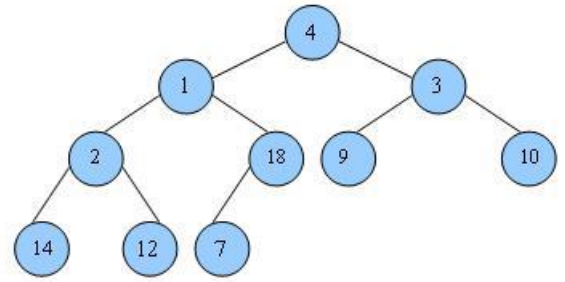
Feladat: szemléltessük a kupacrendezést egy konkrét tömb esetén.

- *Ehhez elsőként le kell játszani a kupac kialakításának meneteit (buildMaxHeap).*
- *Nagyon fontos, hogy alulról felfelé (a legutolsó levél szülőjétől kezdve) süllyesztésekkel alakítsuk ki a kupacot! Az ábrán jelöljük a süllyesztés sorszámát is!*
- *A kész kupacon indul a rendezés második fázisa: A kupac tetején található elemet felcseréljük a kupachoz tartozó utolsó elemmel –így a legnagyobb elem a helyére kerül-, a kupac méretét megadó változót eggyel csökkentjük, majd a kupac tetejére került elemet belesüllyesztjük a kupacba. Ezt addig ismételjük, amíg a kupac mérete 1-re nem csökken.*

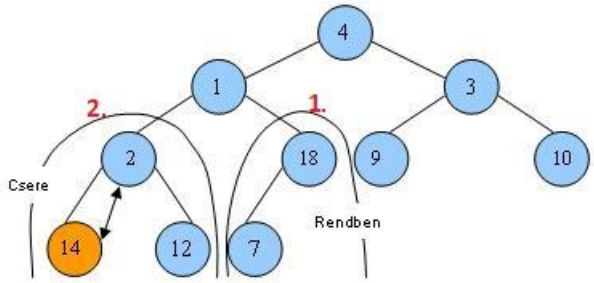
Mutassuk be a kupacrendezést a következő tömbön:
[4, 1, 3, 2, 18, 9, 10, 14, 12, 7]

Kupac kialakításának bemutatása, a buildMaxHeap algoritmus menete:

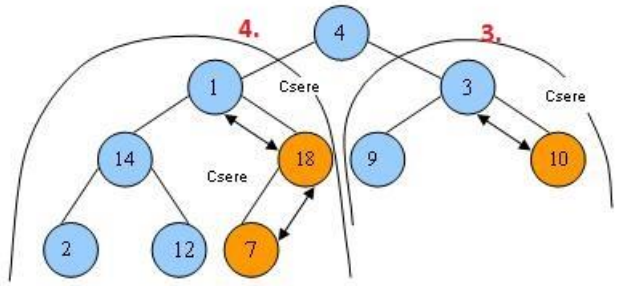
A tömb a következő szintfolytonosan tárolt bináris fát tartalmazza:



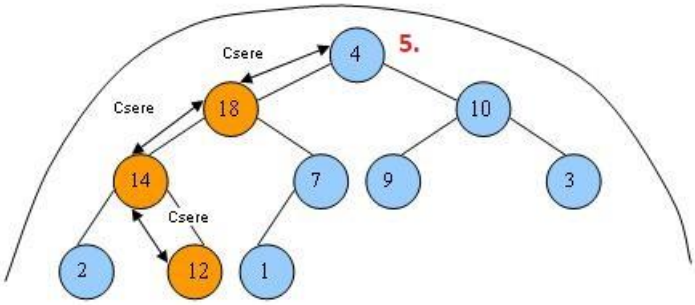
A hetes szülője a 18, ott kezdődik a kupaccá alakítás, majd a 2-es elem következik (félkövér, piros számok jelzik a süllyesztés sorszámát):



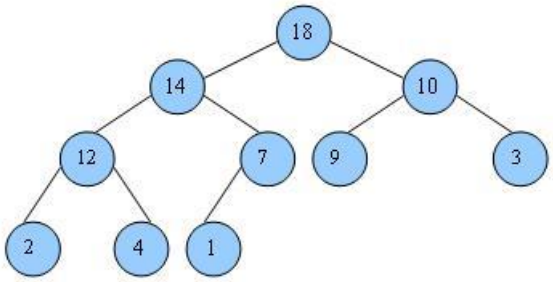
Eggyel feljebbi szinten folytatódik az algoritmus:



Majd felérve a kupac tetejére az utolsó süllyesztés:



A kész kupac:



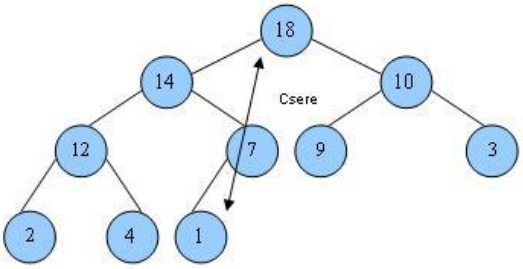
Az utolsó süllyesztés tömbön szemléltetve:

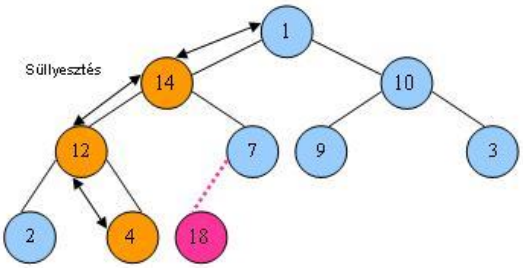
1	2	3	4	5	6	7	8	9	10
4	18	10	14	7	9	3	2	12	1
18	4	10	14	7	9	3	2	12	1
18	14	10	4	7	9	3	2	12	1
18	14	10	12	7	9	3	2	4	1

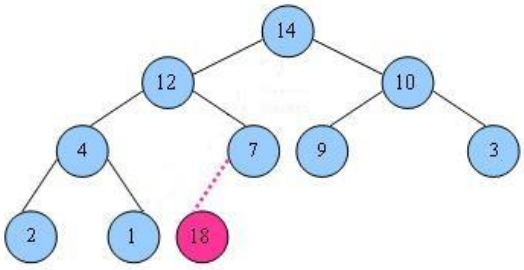
- *sárga az aktuális elem,*
- *kék a bal gyerek,*
- *zöld a jobb gyerek,*
- *a nagyobbik gyerek piros keretes,*
- *a cserét nyíl ábrázolja.*

Rendezés folytatása

Nem fogjuk végig lejátszani, itt csak az első két menetet részletezzük. Ne feledjük, hogy a kupac egy tömbben elhelyezett bináris fa. Így valójában a tömb elemeit cserélgetjük rendezés közben!



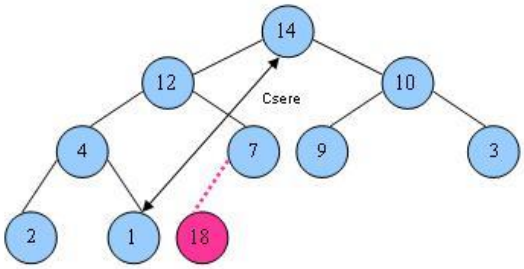


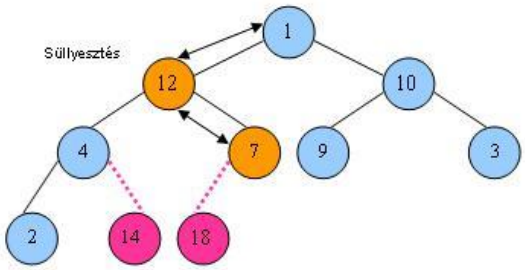


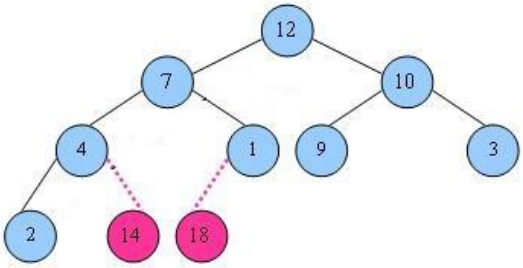
Első menet vége, 18 helyére került és a kupac helyre van állítva. A második legnagyobb elem, a 14 a kupac tetejére került. A 18-as elem a tömbben van, de már nem tartozik a kupachoz!

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

Javaslat: önállóan szemléltessük még a következő két menetet!

Feladat: A prioritásos sornál látott add() művelet adja az ötletet, hogy a kezdőkupac kialakítására a következő algoritmus is működik: a tömb első eleme: A[1] egy egyelemű kupac. A 2..A.length elemeket rendre szúrjuk be a kupacba. Ekkor az elemek úgy kerülnek a helyükre, hogy addig emeljük őket a kupacban felfelé, amíg nagyobbak a szülőjüknél. Miért nem így történik a kezdőkupac kialakítása a kupacrendezés algoritmusában? Indokoljuk, miért kevésbé hatékony ez az algoritmus!

Megoldás vázlata: Mind a süllyesztés, mind az emelés $O(\log_2 m)$ hatékonyságú, ahol m a kupac méretét adja meg, mégis a kupac kialakításának műveletigénye nem ugyanaz a két algoritmus szerint. Nyilvánvaló, hogy minél nagyobb az m, annál hosszabb úton történik a süllyesztés, illetve a fent vázolt megoldásnál az emelés. Csakhogy a süllyesztésen alapuló módszernél, amikor m már „kezd nagy lenni”, azaz közelítünk a kupac teteje felé, egyre kevesebb a süllyesztendő elem – a kupac felső szintjeiről indított süllyesztést egyre kevesebb elemre kell meghívni, így az egyre növekvő „hosszú” úton kevés elemnek kell a helyét megkeresni. Más a helyzet azonban az emelésen alapuló algoritmus esetén. A kupac alsó szintjein egyre több és több elem van – ha egy teljes bináris fára gondolunk, az elemek fele a legalsó szinten van – és ezt a nagyszámú elemet kell a gyökérig tartó „hosszú” úton a helyére emelni.
A jegyzetben megtaláljuk annak bizonyítását, hogy a buildMaxHeap eljárás műveletigénye lineáris, azaz: $\Theta(n)$. Megmutatható, hogy az emeléses módszerrel ez O(n log n) –re (legrosszabb esetben θ (n log n) –re) nő. Ezért nem szabad ezt a módszert választani a kezdőkupac kialakítására.

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 paraméterként megkapjuk az emelendő elemet, majd az elem helyét, a lyukat visszük felfelé a kupacban, azáltal, hogy a szülőt, ha kisebb, mint az emelendő elem, betesszük a lyukba. Így a szülő helyére kerül a lyuk. Ezt folytatjuk, amíg kell, majd végül a lyukba betesszük a paraméterben kapott elemet. Ezzel a módszerrel a süllyesztés is hatékonyabbá tehető.

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.