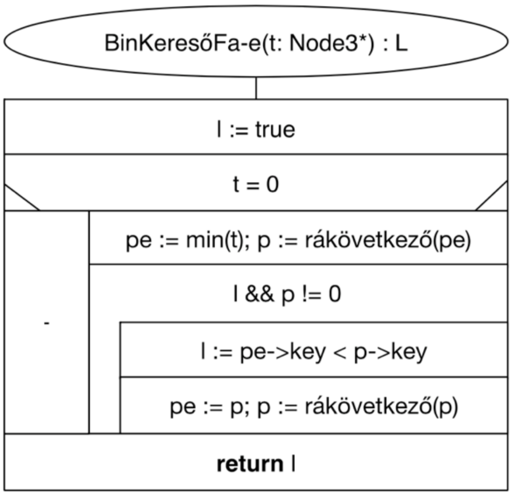
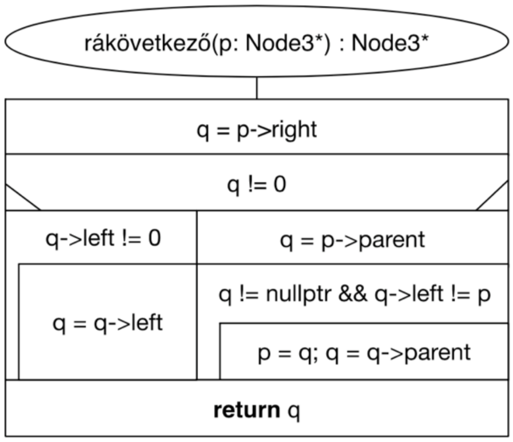
9. Gyakorlat (2019.04.09)

Bináris fák algoritmusai

Feladat: keszítsünk 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 ábrázoljuk **Node3** típussal, azaz szülő pointer is van

Megjegyzés: a rákövetkező elemet megadó algoritmust megtaláljuk a jegyzetben. "**inorder_next**"néven.

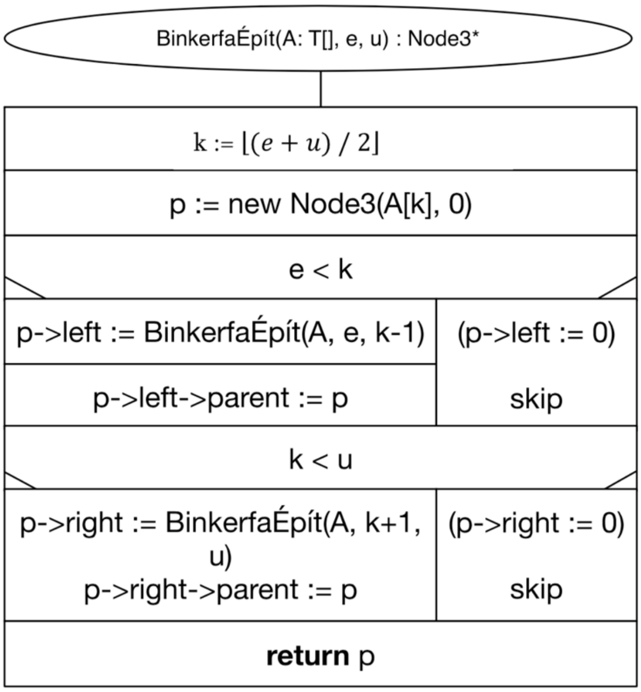




Feladat: a bináris keresőfa alakja nagyon el tud romlani, egyes ágai túl hosszúra nőhetnek (akár listává torzulhat a fa), így elveszíti a benne történő keresés a hatékonyságát. Erre a problémára tudnak megoldást adni az „önkiegyensúlyozó" keresőfák: az AVL fák vagy a piros-fekete fák. Ezek majd Algo2-ből lesznek részletesen, most csak megemlítjük őket.

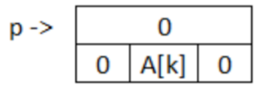
Érdekes viszont a következő ötlet: ha nagyon elromlik a fa alakja, járjuk be inorder bejárással, írjuk egy tömbbe a rekordokat, majd az így kapott szigorúan monoton növekvően rendezett tömbből építsük fel újra a bináris keresőfát úgy, hogy alakja optimális legyen.

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, oly módon, hogy a szülő pointert is beállítjuk.



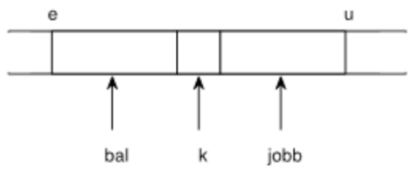
Hívása: **t=BinkerfaÉpít(A,1,A.M)**

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



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:



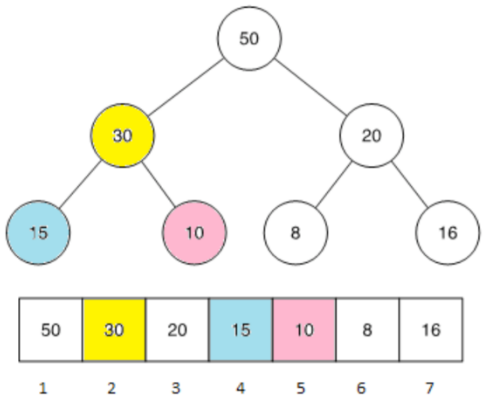
Fontos definíciók!!

- **szigorúan bináris fa:** a fa minden belső pontjának két gyereke van.
- **teljes bináris fa:** olyan szigorúan bináris fa, ahol minden levél azonos szinte helyezkedik el.
- **majdnem teljes bináris fa:** olyan teljes bináris fa, melynek legalsó (levél) szintjéről elhagyhatunk néhány levelet (de nem az összest).
- **majdnem teljes balra tömörített bináris fa:** majdnem teljes bináris fa, de levelek csak a legalsó szintről, a jobb oldalról hiányozhatnak. Ezeket *szintfolytonos* fának 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 a kupac tetején (fa gyökerében) mindig az egyik legnagyobb elem található. Minimum kupac hasonlóan: a szülő kulcs kisebb vagy egyenlő a gyerekei kulcsánál. Gyökérben a legkisebb elem található.

Kupac ábrázolása

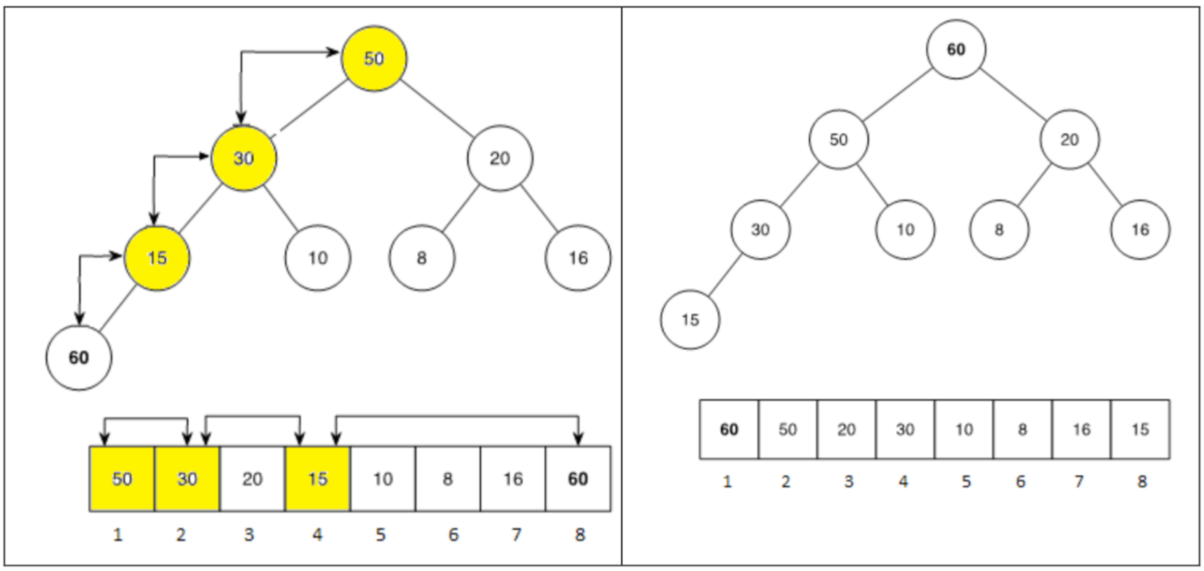
A szintfolytonos bináris fákat, így a kupacokat is tömbbel ábrázoljuk. (Szokás ezt az ábrázolást „*bináris fák aritmetikai ábrázolásának*” is nevezni). A szintfolytonos elhelyezés következménye, hogy a fában való navigálás a tömb indexeinek segítségével történik.

Vizsgált csúcs indexe:	**i**	(ábrán sárga színnel jelölve)
Bal gyerek indexe:	**2*i**	(ábrán kék színnel jelölve)
Jobb gyerek indexe:	**2*i+1**	(ábrán rózsaszínnel jelölve)
Szülő indexe:	$\left\lfloor \frac{i}{2} \right\rfloor$ (i/2 alsó egész része)	



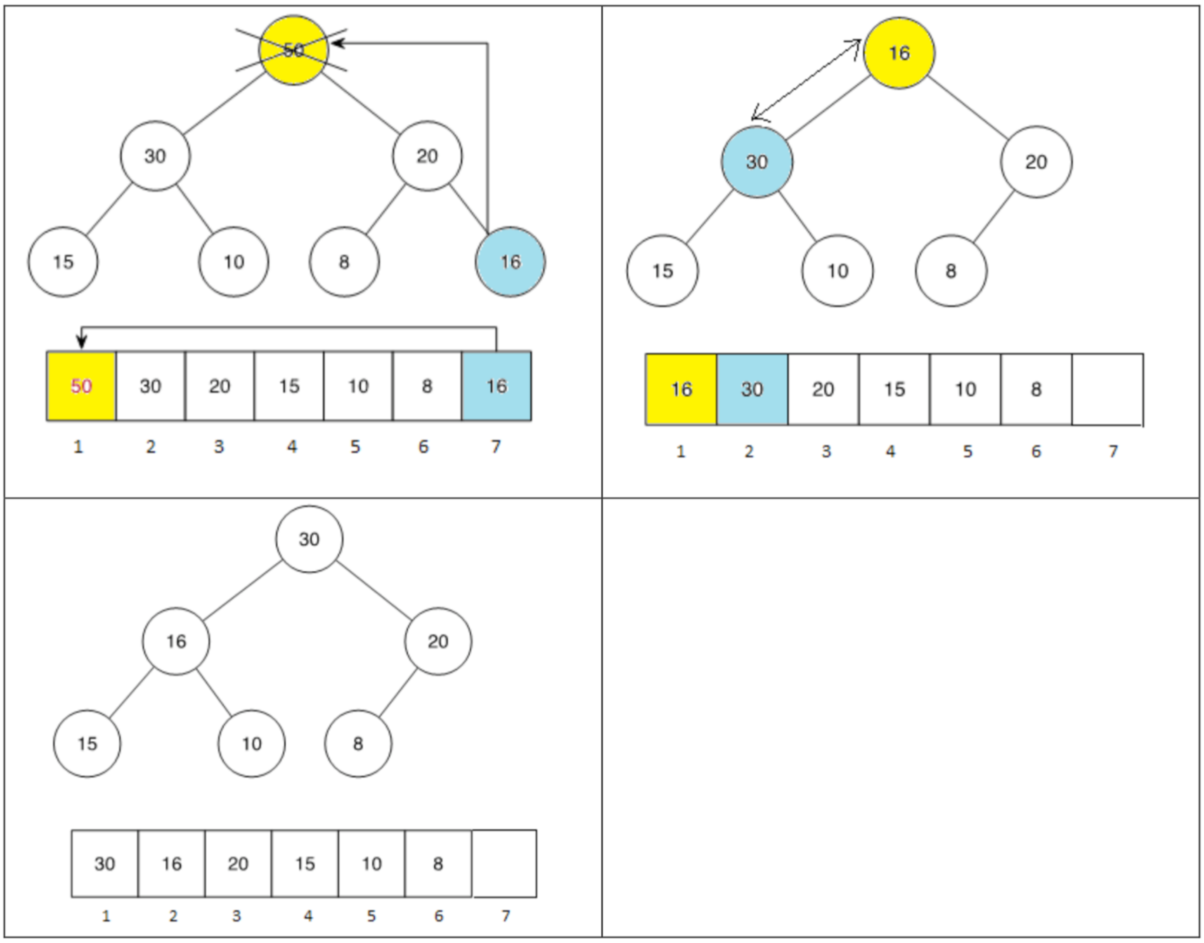
Két fontos művelet: beszúrás, maximum törlése

Elem beszúrása maximum kupacba:



60-as elem beszúrása: mivel a levelek szintje tele van, egy új szint keletkezik, és annak a 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 $\lceil log_2\, n \rceil$.

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 a balra-tömörítettségét). Ezután következik az ún. 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 elem nagyobb vagy egyenlő lesz, mint a nagyobbik gyerek, vagy leérünk a kupac aljára.

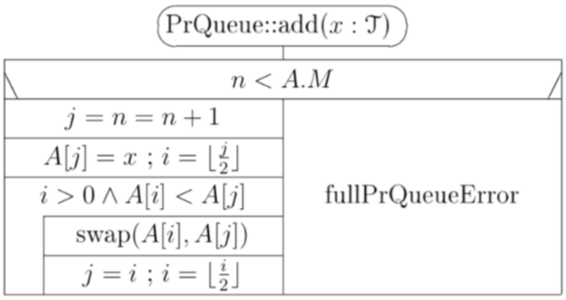
Prioritásos sor

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 ábrázolható kupaccal. Prioritást az elem nagysága adja meg. Itt most a maximum prioritásos sort fogjuk tanulmányozni, amit egy maximum kupaccal ábrázolunk.

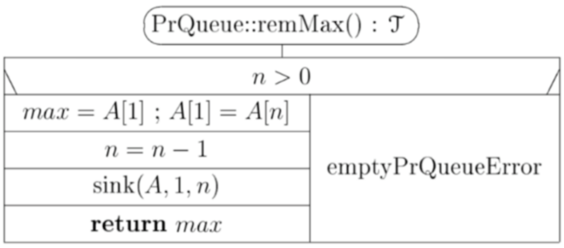
Prioritásos sor UML osztály diagrammja

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** $n == 0$}
+ ~ PrQueue() { **delete** A }
+ setEmpty() {$n = 0$} // reinitialize the priority queue

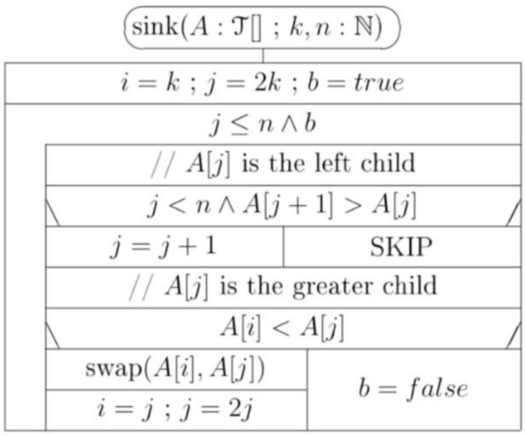
Műveleti kupac esetén:



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.



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	$O(\log n)$	$O(\lg n)$	$\Theta(1)$