ANSAMBLU (HEAP)

- Este o **structură de date** eficientă pentru memorarea *cozilor cu priorități*
- Tipuri de ansamblu: **binar**, binomial, Fibonacci, *leftist heaps*, *skew heaps*, etc.
- Vom prezenta, în cele ce urmează, structura de **ansamblu binar**.
  - În cartea Cormen, Leiserson & Rivest din docs și în directorul Curs14/docs găsiți documentații despre alte tipuri de ansambluri.

DEFINIȚIE. Un ***ansamblu binar*** $(a_1, a_2, \ldots, a_n)$ este un vector (ale cărui elemente sunt de tip comparabil, **TElement=TComparabil**) care poate fi vizualizat sub forma unui arbore binar având ***structură de ansamblu*** și care verifică ***proprietatea de ansamblu***.

Structură de ansamblu – arborele binar (sub forma căruia poate fi vizualizat ansamblul) este *aproape plin* (dacă toate nivelurile acestuia sunt complete, exceptând ultimul nivel care este plin de la stânga la dreapta).

Exemplu Vectorul **9, 5, 4, -3, -2, 2, 3, -7, -5, -4** poate fi vizualizat sub forma arborelui binar din Figura 1. Acesta are structură de *ansamblu*, fiind *aproape plin*.

- elementele vectorului sunt dispuse pe niveluri, în ordine, începând cu primul element.

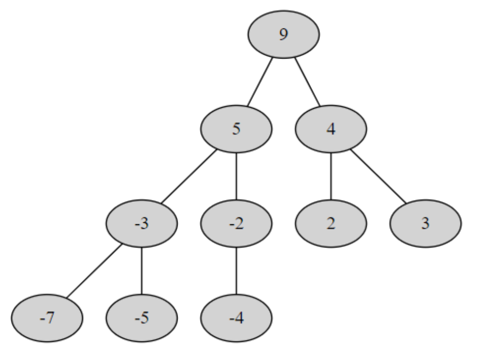


Figura 1. Vectorul **9, 5, 4, -3, -2, 2, 3, -7, -5, -4** vizualizat sub forma unui arbore cu structură de ansamblu

Observații: Pe baza vizualizării vectorului $a_1, a_2, \ldots, a_n$ sub forma unui arbore binar aproape plin (ca în Figura 1) deducem următoarele

- a_1 este elementul din rădăcina arborelui
- a_i are fiul stâng $a_{2 \cdot i}$ dacă $2 \cdot i \leq n$ și fiul drept $a_{2 \cdot i+1}$ dacă $2 \cdot i + 1 \leq n$
- a_i are părintele $a_{[i/2]}$

Proprietatea de ansamblu constă în verificarea următoarelor condiții (impuse între valorile nodurilor din arborele asociat și valorile din descendenți – stâng și drept)

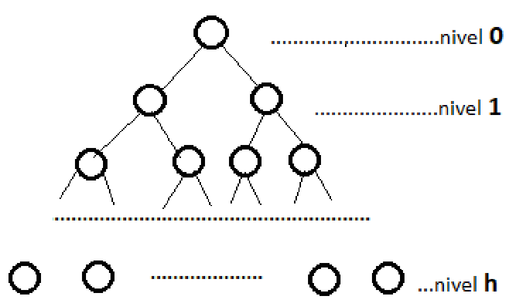
1) $a_i \geq a_{2i}$ $\forall i$, dacă $2 \cdot i \leq n$
2) $a_i \geq a_{2i+1}$ $\forall i$ dacă $2 \cdot i + 1 \leq n$

Observații

- Dacă relația de ordine din inegalitățile 1) și 2) (de mai sus) este „≥", atunci *heap*-ul se numește **max-heap**;
  - Vectorul indicat în **Exemplu** verifică proprietatea de ansamblu și este un **max-heap** (la fiecare nod din arbore, valoarea elementului este mai mare sau egală cu cea a descendenților).
- Dacă relația de ordine din inegalitățile 1) și 2) (de mai sus) este „≤", atunci *heap*-ul se numește **min-heap**;
- Relația de orine („≥", „≤") poate fi generalizată la o relație de ordine $\mathfrak{R}$ oarecare
- Ansambul binar este, în general, memorat ***secvențial*** folosind un vector (dinamic), fără a fi necesară memorarea înlănțuită - legături între elemente (ex. pointeri).

Datorită **proprietății de ansamblu** (relațiile pe care le satisfac elementele acestuia), următoarele afirmații sunt adevărate într-un ansamblu $a_1, a_2, \ldots, a_n$.

- a_1 este cel mai **mare** element din ansamblu dacă $\mathfrak{R}$="≥"
- Dacă $\mathfrak{R}$="≥", atunci pe orice drum de la rădăcină la un nod, elementele sunt ordonate descrescător.
- **Înălțimea** unui heap cu *n* elemente este $\theta(\log_2 n)$. Ca urmare, timpul de execuție a operațiilor specifice va fi $O(\log_2 n)$.
  - **Înălțimea** este definită ca lungimea drumului de la rădăcină la o frunză (nod care nu mai are descendenți – toate frunzele, într-un ansamblu, sunt pe ultimul nivel). În exemplul din Figura 1, înălțimea este 3.
  - În cazul în care arborele binar asociat ansamblului ar fi plin (toate nivelurile ar fi pline), ca în figura de mai jos, iar *h* este înălțimea ansamblului, observăm următoarele:



pe nivelul *i* în arbore sunt 2^i noduri $\Rightarrow n=1+2+\ldots+2^h$

$\Rightarrow n=2^{h+1}-1$

$\Rightarrow h=\log_2(n+1)-1 \in \theta(\log_2 n)$

Sunt 2 operații specifice pe ansamblu:
- **adăugare** element (astfel încât să se păstreze proprietatea de ansamblu)
- **ștergere** element (se șterge elementul maxim dacă $\mathfrak{R}$="≥", cel din vârful ansamblului).

- Pp. în continuare $\mathfrak{R}$="≥".
- Pp. în cele ce urmează că elementele din vectorul care memorează ansamblul sunt indexate de la 1.

Pentru reprezentarea ansamblului vom folosi o structură care memorează vectorul corespunzător.

Ansamblu
Max: Intreg {capacitatea maximă de memorare}
n: Intreg {nr.de elemente din ansamblu}
e: TElement[1..*n*] {elementele din ansamblu}

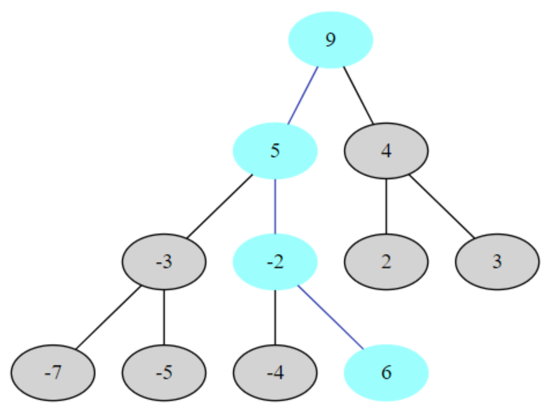
Vom discuta, în cele ce urmează, cele două operații.

Adăugare

Presupunem că în ansamblul **9, 5, 4, -3, -2, 2, 3, -7, -5, -4** (vizualizat în Figura 1) dorim să adăugăm valoarea **6.** Adăugarea presupune următoarele

- Adăugăm valoarea **6** la finalul ansamblului (vectorului)
- Restabilim proprietatea de ansamblu, posibil alterată în urma adăugării elementului.

În exemplul nostru, prin adăugarea lui **6** la finalul ansamblului obținem



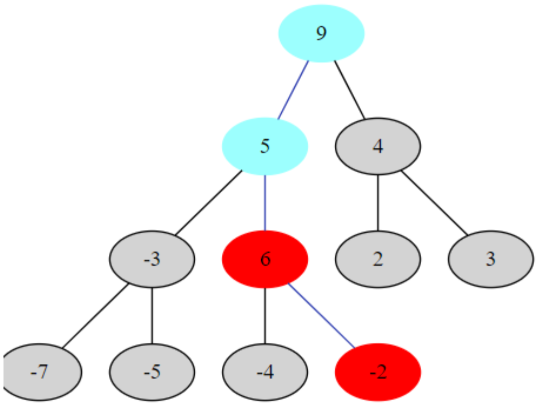
Observăm că proprietatea de ansamblu este alterată doar pe drumul de la elementul adăugat până la rădăcină (6, -2, 5, 9). Practic, va trebui să căutăm loc pentru **6** printre ascendenții săi (marcați pe figură), să îl **urcăm** în ansamblu, până când va fi verificată proprietatea de ansamblu. Modificările necesare pentru a reface proprietatea de ansamblu sunt următoarele:

Pas 1. Comparăm 6 cu părintele său (-2). 6 este mai mare decât -2, înseamnă că îl coborâm pe -2 în locul lui 6 și continuăm să îl urcăm pe 6.

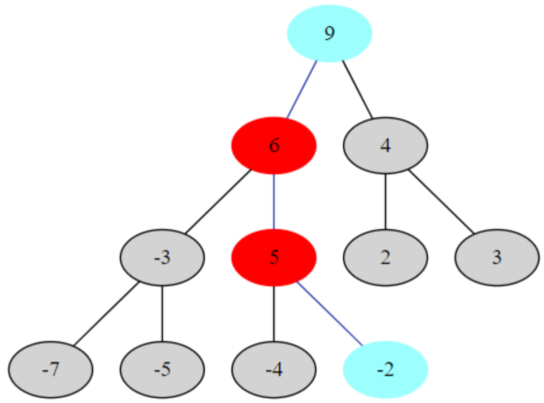
Pas 2. Comparăm 6 cu 5. 6 este mai mare decât 5, înseamnă că îl coborâm pe 5 în locul lui 6 și continuăm să îl urcăm pe 6.

Pas 3. Comparăm pe 6 cu 9. 6 $\leq$9 (părintele), înseamnă că oprim procesul iterativ, am găsit loc pentru 6. **STOP**

Pas 1



Pas2



Prin urmare, ansamblul rezultat în urma adăugării lui **6** în ansamblul **9, 5, 4, -3, -2, 2, 3, -7, -5, -4** este **9, 6, 4, -3, 5, 2, 3, -7, -5, -4, -2** (corespunzător arborelui din dreapta).

Din procesul descris anterior, observăm faptul că numărul maxim de pași ai structurii iterative pentru urcarea lui **6** în ansamblu este *h* (înălțimea arborelui). Rezultă faptul că operația de **adăugare** are complexitatea timp $O(log_2 n)$.

Subalgoritmul de adăugare este descris în Pseudocod mai jos.

Subalgoritmul ADAUGĂ (*a*, *e*) este {complexitate timp $O(log_2 n)$ }
{pre: *a*: Ansamblu, *a* nu e plin, *e*:TElement }
{post: *a* rămâne ansamblu după adăugare}
    a.n←*a*.n+1
    a.e[*a*.n] ←*e*
    URCĂ(*a*, *a*.n) {restabilește proprietatea de ansamblu posibil alterată}
sfADAUGĂ

<u>Obs.</u> În subalgoritmul anterior, nu s-a verificat la adăugare dacă ansamblul e plin. La implementare se poate redimensiona vectorul dacă se observă că se depășește capacitatea maximă alocată.

Subalgoritmul URCĂ (*a*, *i*) este { complexitate timp $O(log_2 n)$ }
{urcă elementul de pe poziția *i* spre rădăcină până va fi satisfăcută proprietatea de ansamblu}
{pre: *a* ansamblu nevid, elem. de pe poziția *i* a fost actualizat}
{post: *a* este ansamblu}
    e←*a*.e[*i*] {elementul de urcat}
    k ←*i* {poziția unde va fi pus elementul *e* }
    p ← [*k*/2] {părintele lui *k*}
    {căutăm o poziție pentru *e* printre strămoșii lui}
    Câttimp (*p* ≥1) și (*a*.e[*p*]<*e*) executǎ
        a.e[*k*] ←*a*.e[*p*] {strămoșii mai mici decât *e* sunt coborâți}
        k ←*p*
        p ← [*p*/2]
    sfCâtTimp

{s-a gasit pozitia *k* pe care poate fi adăugat *e*}
a.e[*k*] ←*e*
sfURCĂ

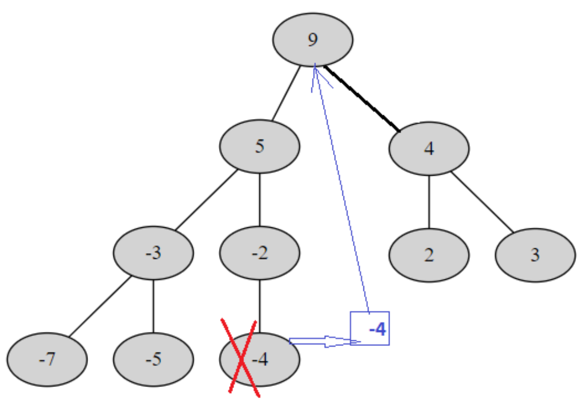
Ilustrăm, în tabelul de mai jos, execuția pas cu pas a algoritmului **URCĂ,** în cazul adăugării valorii 6 (exemplul anterior).

e	*k*	*p*	*a*.e[*p*]	*a*.e[*p*] < *e*	Modificări
6	11	5	-2	-2 < 6, **DA**	*a*.e[11] ← -2
	5	2	5	5 < 6, **DA**	*a*.e[5] ← 5
	2	1	9	9 < 6, **NU**	*a*.e[2] ← 6

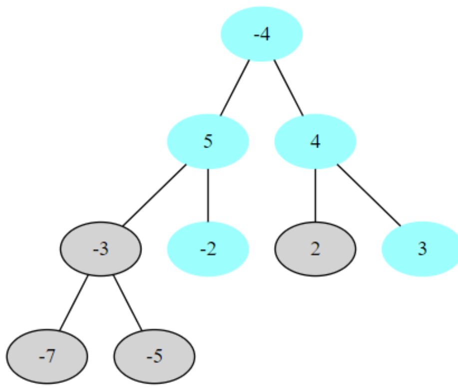
Ștergere

Presupunem că dorim să ștergem elementul maxim (**9**) din ansamblul **9, 5, 4, -3, -2, 2, 3, -7, -5, -4** (vizualizat în Figura 1). După cum menționam anterior, ștergerea din ansamblu este prespecificată, se șterge doar primul element din ansamblu (memorat în rădăcina arborelui asociat).

Ștergerea presupune următoarele
- Ultimul element din ansamblu (cel de finalul vectorului, -4 în exemplul nostru), îl mutăm în locul rădăcinii.
- Restabilim proprietatea de ansamblu, posibil alterată în urma modificării elementului din vârful ansamblului.



În exemplul nostru, prin mutarea lui **-4** în rădăcină, obținem

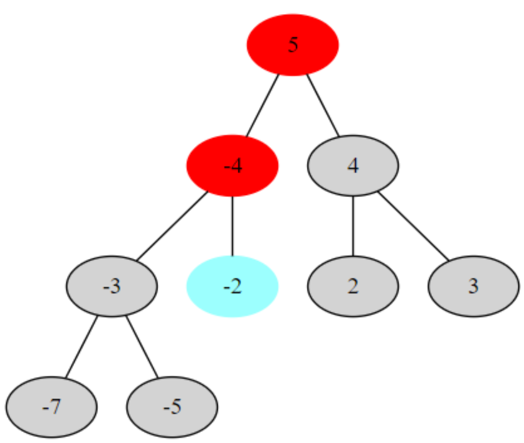


Observăm că proprietatea de ansamblu este alterată pe două drumuri (-4, 5, -2) și (-4, 4, -3). Practic, va trebui să căutăm loc pentru **-4** printre descendenții săi (marcați pe figură), să îl **coborâm** în ansamblu până când va fi verificată proprietatea de ansamblu. De asemenea, observăm că este suficient să refacem proprietatea de ansamblu pe direcția descendentului maxim. Modificările necesare pentru a reface proprietatea de ansamblu sunt următoarele:

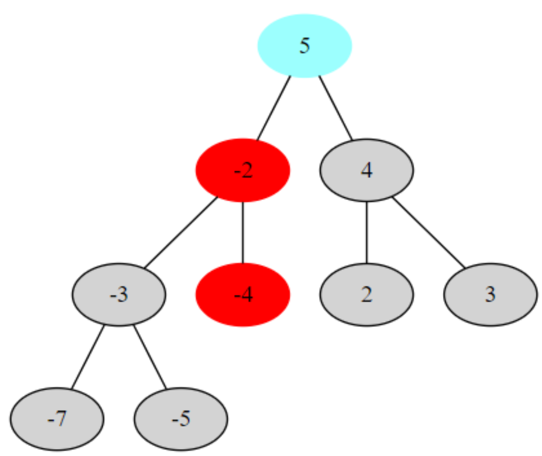
Pas 1. Comparăm pe -4 cu descendentul său maxim (5). Părintele (-4) este mai mic decât descendentul, înseamnă că îl urcăm pe 5 în locul lui -4 și continuăm coborârea lui -4.

Pas 2. Comparăm -4 cu -2. -4 este mai mic decât -2, înseamnă că îl urcăm pe -2 în locul lui -4. Nu mai avem unde să coborâm, îl vom pune pe -4 în locul lui -2. **STOP**

Pas 1



Pas2



Prin urmare, ansamblul rezultat în urma ștergerii (valorii **9)** din ansamblul **9, 5, 4, -3, -2, 2, 3, -7, -5, -4** este **5, -2, 4, -3, -4, 2, 3, -7, -5** (corespunzător arborelui din dreapta).

Din procesul descris anterior, observăm faptul că numărul maxim de pași ai structurii iterative pentru coborârea lui **9** în ansamblu este *h* (înălțimea arborelui). Rezultă faptul că operația de **ștergere** are complexitatea timp $O(log_2 n)$.

Subalgoritmul de ștergere este descris în Pseudocod mai jos.

Subalgoritmul ȘTERGE (*a*, *e*) este {complexitate timp $O(log_2 n)$ }
{pre: *a*:Ansamblu, *a* nu e vid}
{post: *e*:TElement este elementul maxim și e șters, *a* rămâne ansamblu după ștergere}
e← *a*.e[1] {elementul maxim}
a.e[1] ←*a*.e[*a*.n]
a.n←*a*.n-1
COBOARĂ(*a*, 1) {restabilește proprietatea de ansamblu posibil alterată}
sfSTERGE

Subalgoritmul COBOARĂ (*a*, *poz*) este {complexitate timp $O(log_2 n)$ }
{coboară elementul de pe poziția *poz* printre descendenți până va fi satisfăcută proprietatea de ansamblu}
{pre: *a* ansamblu nevid, elem. de pe poziția *poz* a fost actualizat}
{post: *a* este ansamblu}
e←*a*.e[*poz*] {elementul de mutat}
i ←*poz* {poziția unde va fi pus elementul *e* }
j ←2 · *poz* {fiul stâng al lui *i*}
{căutăm o poziție pentru *e* printre descendenți. Descendenții mai mari decât *e* urcă un nivel în arbore }
câttimp (*j* ≤ *a.n*) execută { *i* are fiu stâng}
dacă (*j* < *a.n*) atunci { *i* are și fiu drept? Dacă da, îl luăm pe cel mai mare dintre ei}
dacă *a*.e[*j*] < *a*.e[*j*+1] atunci
j ←*j*+1
sfdacă
sfdacă
dacă *a*.e[*j*] ≤ *e* atunci {cel mai mare fiu este mai mic sau egal cu *e*, atunci STOP}
j ←*a.n*+1
altfel
a.e[*i*] ←*a*.e[*j*] {fiul *j* urcă}
i ←*j*
j ←2 · *i*
Sfdacă
Sfcâttimp
a.e[*i*] ←*e* {pun elementul înapoi în structură}
sfCOBOARĂ

Ilustrăm, în tabelul de mai jos, execuția pas cu pas a algoritmului **COBOARĂ,** în cazul ștergerii (valorii 9).

e	*i*	*j*	*a*.e[*j*] < *a*.e[*j*+1]	*a*.e[*j*] ≤ *e*	Modificări
-4	1	2	5 < 4, **NU**	5 ≤ -4, **NU**	*a*.e[1] ← 5
	2	4 5	-3 < -2, **DA**	-2 ≤ -4, **NU**	*a*.e[2] ← -2
	5	10 (*j* > *a.n*=9, STOP)			*a*.e[5] ← -4

Aplicații ale structurii de ansamblu

1. **Ansamblul** este cea mai potrivită structură de date pentru memorarea elementelor unei **Cozi cu Priorități** (CP): operațiile *adaugă*, *element* (accesare element), *șterge* au complexitate $\boldsymbol{O(\log_2 n)}$.

   Analizăm, comparativ, următoarele structuri de date pentru reprezentarea unei CP folosind

   - **Vector dinamic** ordonat, în care elementul cel mai prioritar este ultimul.
   - **Listă simplu înlănțuită** ordonată, în care elementul cel mai prioritar este primul.
   - **Listă dublu înlănțuită** ordonată, în care elementul cel mai prioritar este primul sau ultimul (nu contează).
   - **Ansamblu**, cu elementul cel mai prioritar primul (în rădăcină)

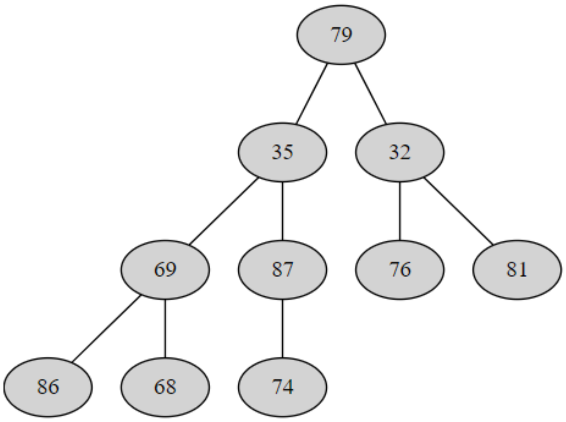
Structura de date	**adăugare**	**ștergere**	**accesare**
Vector dinamic ordonat	$O(n)$	θ(1) amortizat (dacă e cu redimensionare)	θ(1)
LSIO	$O(n)$	θ(1)	θ(1)
LDIO	$O(n)$	θ(1)	θ(1)
Ansamblu	$\boldsymbol{O(\log_2 n)}$	$\boldsymbol{O(\log_2 n)}$	θ(1)

2. **HEAPSORT.** Sortarea unui vector cu *n* elemente folosind un ansamblu. Complexitate timp $O(n \log_2 n)$
- se poate *in place*, fără memorarea suplimentară a ansamblului.
   - **Folosind un ansamblu auxiliar** (*out of place*, spațiu suplimentar de memorare θ(*n*)), ideea este următoarea:
     - Se iau, pe rând, elementele din vector și se adaugă într-un ansamblu ⇒ $O(n \log_2 n)$
       - se poate arăta că timpul necesar pentru construcția unui heap cu *n* elemente este O(*n*) (a se vedea Observația 2)
     - Se aplică de *n* ori ștergerea din ansamblul auxiliar și rezultă elementele în ordine ⇒ $O(n \log_2 n)$

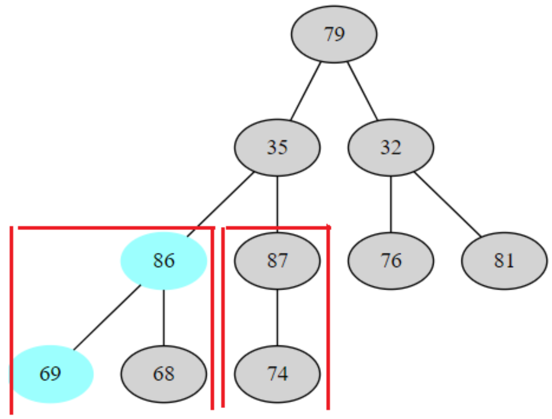
     __**Exemplu**__ Fie vectorul 1, 5, 3, 9, 7. Vrem să îl sortăm descrescător. Construim un **max-heap** cu elementele sale ⇒ ansamblul 9, 7, 3, 1, 5. Apoi scoatem toate elementele din heap și rezultă 9, 7, 5, 3, 1
   - **Fără a folosi un ansamblu auxiliar** (*in place*). Ideea: restructurăm vectorul încât să devină ansamblu (să fie satisfăcută proprietatea de ansamblu la orice nivel în arborele asociat).
     - se încearcă refacerea proprietății de ansamblu de jos în sus (de la frunze spre rădăcină), pornind de la nodurile de înălțime *h*-1, apoi *h*-2,....până se ajunge la înălțime 0 (întregul ansamblu).
       - această operație are complexitate timp $O(n \log_2 n)$.
     - Se aplică de *n* ori ștergerea din ansamblul auxiliar și rezultă elementele în ordine ⇒ $O(n \log_2 n)$

__**Exemplu**__ Fie vectorul 79, 35, 32, 69, 87, 76, 81, 86, 68, 74. Ilustrăm, mai jos, modul în care este restructurat ansamblul

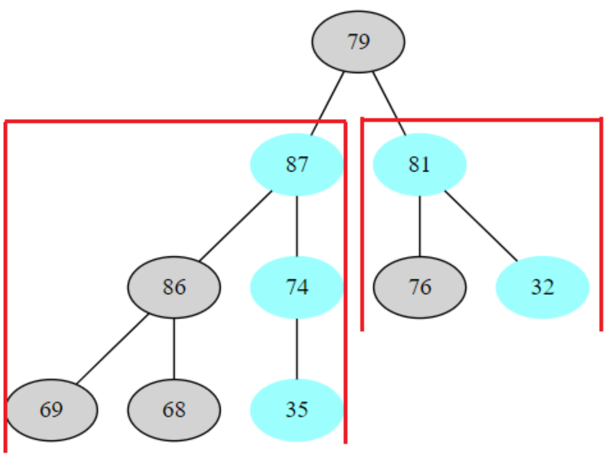
Inițial



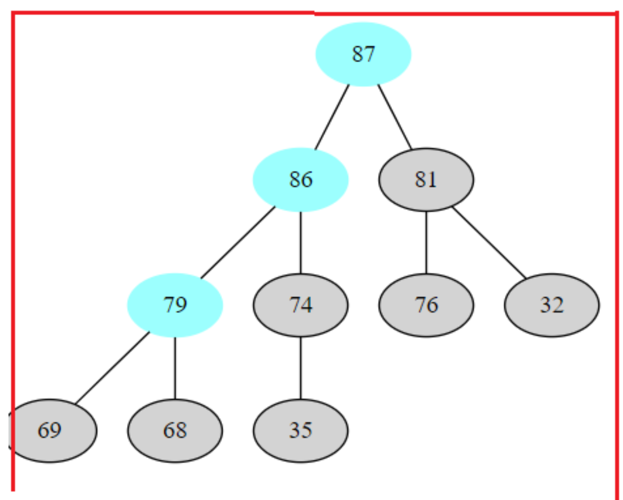
Pas 1



Pas 2



Pas 3



Se observă că ansamblul final (obținut după Pasul 3) **87, 86, 81, 79, 74, 76, 32, 69, 68, 35** reprezintă un **max-heap**

Observații

1. Se poate demonstra (prin inducție) că un ansamblu binar cu *n* elemente are cel mult $[n/(2^{h+1})]$ noduri de înălțime *h*.
2. Se poate demonstra (pe baza 1) că un ansamblu binar se poate construi în $O(n)$ dintr-un vector cu *n* elemente.
3. Reunirea (interclasarea) a două ansambluri binare cu *n* și *m* elemente se poate face în $O(n+m)$.

PROBLEME

1. Fie ansamblul <1, 2, 4, 2, 5>. Aplicați de două ori operația de ștergere.
2. Generalizați relația ”≥” la o relație de ordine $\mathcal{R}$ oarecare și implementați operațiile specifice.
3. Care este cel mai mic, respectiv cel mai mare număr de elemente dintr-un ansamblu având înălțimea *h*?

4. Arătați că un ansamblu având n elemente are înălțimea $[\log_2 n]$
5. Arătați că în orice subarbore al unui ansamblu rădăcina subarborelui conține cea mai mare valoare care aparține în acel arbore (dacă $\mathcal{R}$=”≥”).
6. Dacă $\mathcal{R}$=”≥”, unde se poate afla cel mai mic element al unui ansamblu, presupunând că toate elementele sunt distincte?
7. Este vectorul în care elementele se succed în ordine descrescătoare un ansamblu?
8. Este secvența <23, 17, 14, 6, 13, 10, 15, 7, 12> un ansamblu?
9. Să se generalizare ansamblul binar la un ansamblu *ternar* sau *cuaternar* (în loc de 2 descendenți sunt 3, respectiv 4).
10. Să se implementeze, folosind un ansamblu binar, un container **CP*k*** similar cu **Coada cu priorități**, exceptând faptul că vrem să accesăm și să ștergem **al *k*-lea cel mai prioritar element** în raport cu o relație de ordine $\Re$ între priorități (dacă $\Re$=≤, atunci elementul cel mai prioritar este **minimul**). ***Indicație*: *pentru determinarea elementului cel mai prioritar dintre k elemente, se va folosi tot un ansamblu binar*.**
11. Găsiți un algoritm $O(n \cdot log_2 k)$ pentru a interclasa k liste ordonate, unde n este numărul total de elemente din listele de intrare**.** Reprezentarea listelor este ascunsă, acestea se parcurg folosind iteratori.

Indicație

- generalizăm ideea de la interclasarea a două liste ordonate
- în fiecare dintre cele k liste, avem un element curent (indicat de un iterator)

  a). pentru a extrage minimul/maximul dintre cele k liste, folosim un ansamblu ⇒ $O(\log_2 k)$
  - în lista din care a fost găsit minimul/maxim, deplasăm iteratorul
  - elementul indicat de iterator, îl adăugăm în heap

  b) repetăm pasul a) de n ori (pentru numărul de elemente din toate listele) ⇒ $O(n \cdot log_2 k)$

EXAMEN

Exemple grilă

1. Numărul de pași efectuat într-o căutare binară a unui element într-un vector ordonat cu n elemente este

   a) $O(\log_2 n)$ b) $\theta(\log_2 n)$ c) $\theta(n)$ d) $O(n)$ e) $O(\text{sqrt}(n))$

2. Căutarea binară a unui element într-un vector ordonat cu n elemente se execută în $O(k)$. Cea mai mica valoare a lui k este

   a) $\log_2 n$ b) n c) $\text{sqrt}(n)$