Heap. Coadă de priorități.

Universitatea "Transilvania" din Brașov

28 martie 2022

Heap binar

Definiție: Un **heap binar** este un arbore binar complet, cu o anumită ordonare a cheilor și anume:

- **Heap-max** - pentru fiecare nod, cheia sa este mai mare decât (sau egală cu) cheia copiilor săi
- **Heap-min** - pentru fiecare nod, cheia sa este mai mică decât (sau egală cu) cheia copiilor săi

Reamintim: Un arbore binar complet - fieacare nod intern are exact doi descendenți, cu excepția eventual a ultimului nod intern de pe penultimul nivel, iar frunzele se află doar pe ultimele două niveluri, frunzele de pe ultimul nivel sunt ordonate de la stânga spre dreapta.

Un heap se stochează de obicei cu ajutorul unui tablou liniar - *array*.

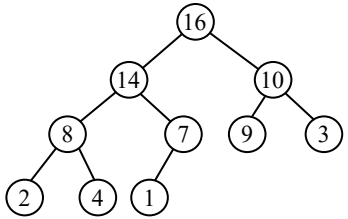
Heap binar

Exemple

Heap max

H1:

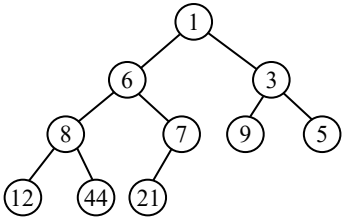
16	14	10	8	7	9	3	2	4	1



Heap min

H2:

1	6	3	8	7	9	5	12	44	21



Heap max - implementare

Folosim o structură ***heap*** cu câmpurile:

- ***H*** - vector care memorează elementele
- ***size*** - variabilă care păstrează numărul de elemente din heap

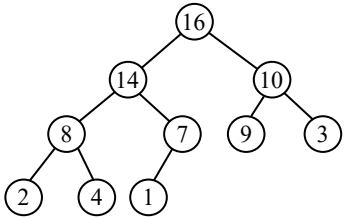
Observație: Pentru fiecare nod i - aflat pe poziția i:

- Părintele se află pe poziția $(i-1)/2$.
- Descendentul stâng se află pe poziția $2 * i + 1$.
- Descendentul drept se află pe poziția $2 * i + 2$.

Heap max

H:

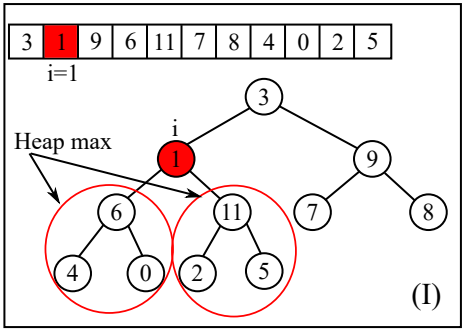
16	14	10	8	7	9	3	2	4	1



Observații:

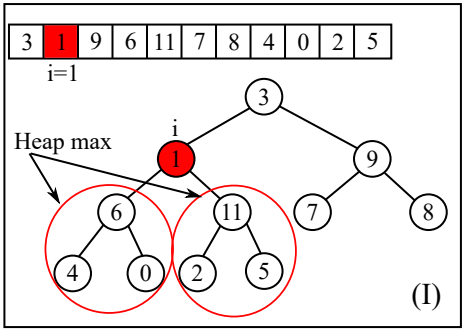
- Înălțimea heap-ului - $\log_2 n$ unde $n = size$ numărul de noduri din heap.
- Complexitatea operațiilor de bază - $O(\log_2 n)$.

Construcția unui heap max



- **Max-Heapfy**: (i): i indicele nodului de la care începe algoritmul. **Premisă**: subarborii nodului i sunt heap-max și doar informația din nodul i strică eventual această proprietate. Funcția **MAX_HEAPFY** reface proprietatea de heap-max.

Construcția unui heap max



- **Max-Heapfy**: (i): i indicele nodului de la care începe algoritmul. **Premisă**: subarborii nodului i sunt heap-max și doar informația din nodul i strică eventual această proprietate. Funcția **MAX_HEAPFY** reface proprietatea de heap-max.
- **Build-Max-Heap**: apelează succesiv funcția **Max-Heapfy**.

Construcția unui heap max

Funcția **Max-Heapfy** idee: - se coboară cheia de pe poziția i pe o poziție corespunzătoare în heap. **Complexitate:** $O(\log_2 n)$

Algoritm 1: MAX-HEAPFY

Intrare: Un heap H cu numărul de elemente *size*, poziția i

$st \leftarrow 2 * i + 1$
$dr \leftarrow 2 * i + 2$
$imax = i$
daca $st < H.size$ *si* $H[st] > H[imax]$ **atunci**
  $imax = st$
sfarsit_daca
daca $dr < H.size$ *si* $H[dr] > H[imax]$ **atunci**
  $imax = dr$
sfarsit_daca
daca $imax \neq i$ **atunci**
  $H[i] \leftrightarrow H[imax]$
  MAX-HEAPFY(H, $imax$)
sfarsit_daca

Construcția unui heap max

Funcția **Build-Max-Heap** - apelează succesiv funcția **Max-Heapfy** pentru i de la $size/2 - 1$ la 0.

Algoritm 2: BUILD-MAX-HEAP

Intrare: Un heap H cu $size$ elemente
pentru $i = size/2 - 1, 0, -1$ **executa**
| MAX-HEAPFY(H, i)
sfarsit_for

Observație: Un heap max se poate construi și prin inserții succesive. Acest lucru va fi ilustrat la coada de priorități.

Construcția unui heap max - complexitate

Complexitatea maximă pentru algoritmul de construcție a unui Heap este $O(n)$.
Demonstrație:

- pentru un nod de înălțime h complexitatea **MAX-HEAPFY** este $O(h)$
- nr max de noduri de înălțime h este $\lceil n/(2^{h+1}) \rceil$
- Deci: timpul maxim pentru construirea heap-ului este

$$\sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{n}{2^{h+1}} O(h) = O\left(n \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^{h+1}} \right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O\left(n \frac{1/2}{(1-1/2)^2} \right) =$$

$$= O(n)$$

Algoritmul *Heapsort*

Idei de bază

- într-un heap-max elementul maxim este plasat în rădăcina, prima poziție din H.

Algoritmul *Heapsort*

Idei de bază

- într-un heap-max elementul maxim este plasat în rădăcina, prima poziție din H.
- în vectorul sortat crescător, elementul maxim trebuie să se afle pe ultima poziție.

Algoritmul *Heapsort*

Idei de bază

- într-un heap-max elementul maxim este plasat în rădăcina, prima poziție din H.
- în vectorul sortat crescător, elementul maxim trebuie să se afle pe ultima poziție.
- dacă interschimbăm elementul de pe prima poziție cu cel de pe ultima și reducem dimensiunea heap-ului cu 1$\Rightarrow$ un heap care se strică la vârf

Algoritmul *Heapsort*

Idei de bază

- într-un heap-max elementul maxim este plasat în rădăcina, prima poziție din H.
- în vectorul sortat crescător, elementul maxim trebuie să se afle pe ultima poziție.
- dacă interschimbăm elementul de pe prima poziție cu cel de pe ultima și reducem dimensiunea heap-ului cu 1$\Rightarrow$ un heap care se strică la vârf
- putem reface heap-ul cu funcția **MAX-HEAPFY**

Algoritmul *Heapsort*

Idei de bază

- într-un heap-max elementul maxim este plasat în rădăcina, prima poziție din H.
- în vectorul sortat crescător, elementul maxim trebuie să se afle pe ultima poziție.
- dacă interschimbăm elementul de pe prima poziție cu cel de pe ultima și reducem dimensiunea heap-ului cu 1$\Rightarrow$ un heap care se strică la vârf
- putem reface heap-ul cu funcția **MAX-HEAPFY**
- aplicăm iterativ acest algoritm pâna se sortează vectorul.

Algoritmul *Heapsort*

Algoritm 3: Heapsort

Intrare: Un vector v cu n

Construiește heap-ul H cu funcția BUILD-MAX-HEAP

$size \leftarrow n$

pentru $i = n-1, 1, -1$ **executa**

    $H[i] \leftrightarrow H[0]$

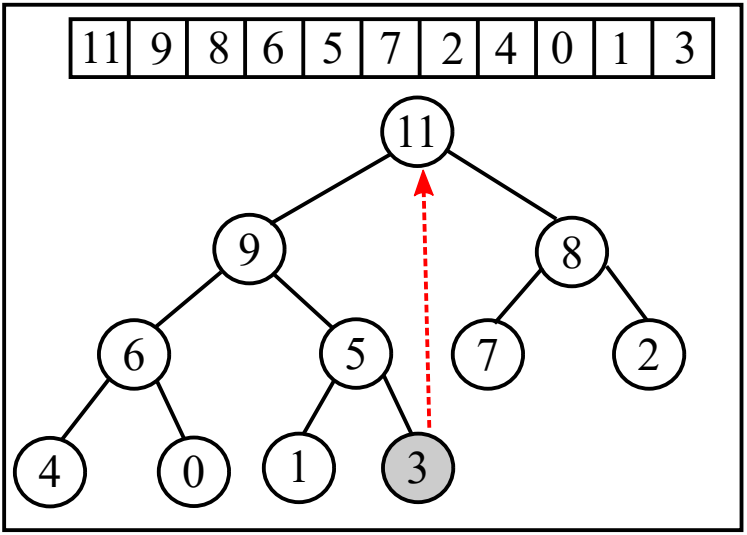
    $H.size \leftarrow H.size - 1$

    Max-Heapfy(H, 0)

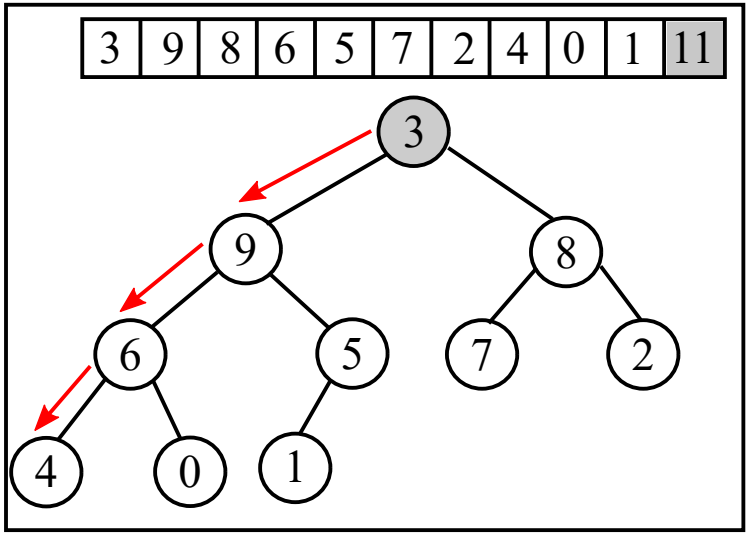
sfarsit_for

Complexitate: $O(n \log_2 n)$

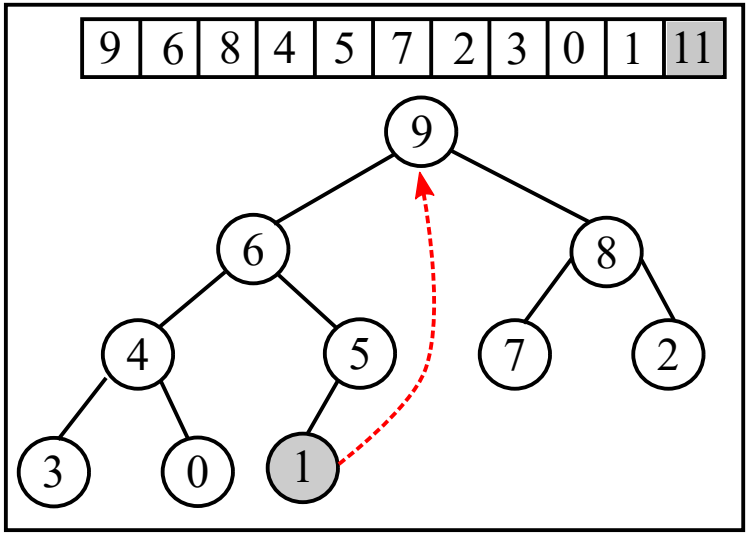
Algoritmul *Heapsort* - Exemplu



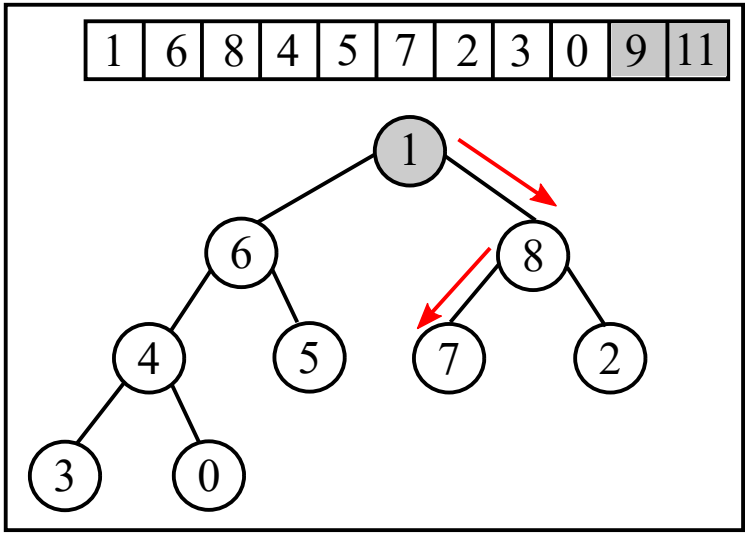
Algoritmul *Heapsort* - Exemplu



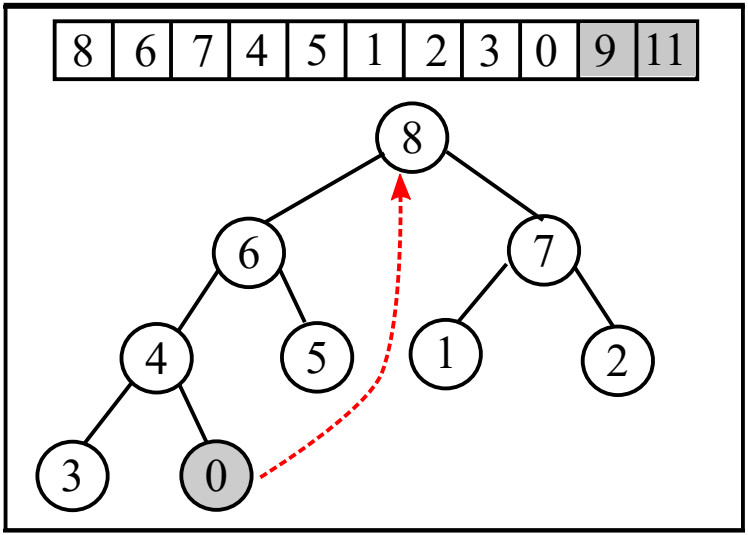
Algoritmul *Heapsort* - Exemplu



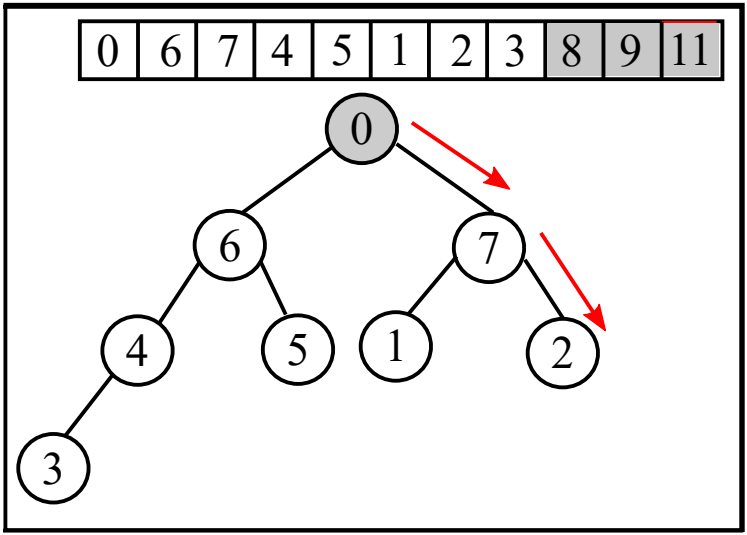
Algoritmul *Heapsort* - Exemplu



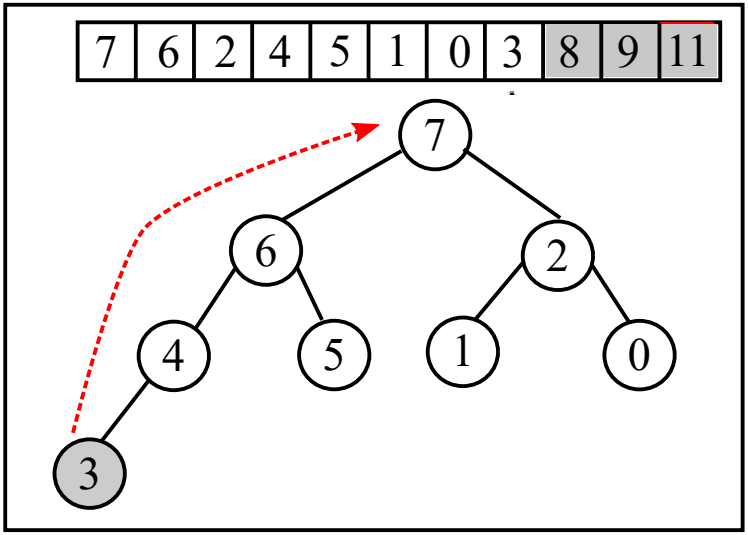
Algoritmul *Heapsort* - Exemplu



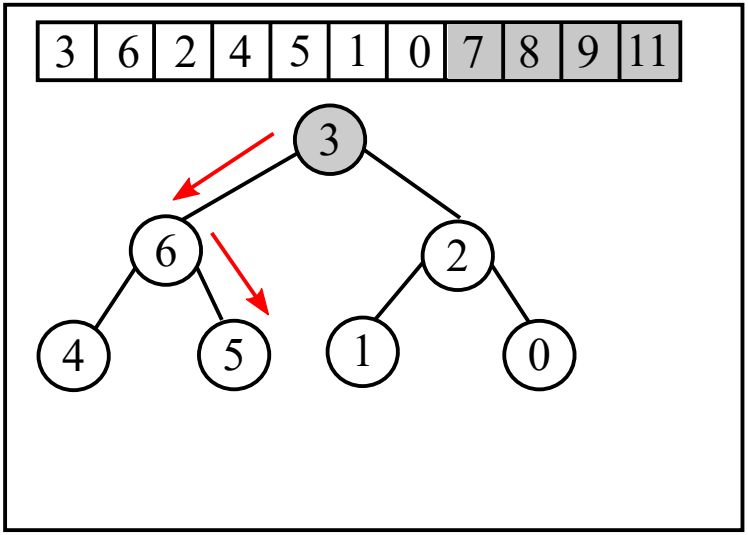
Algoritmul *Heapsort* - Exemplu



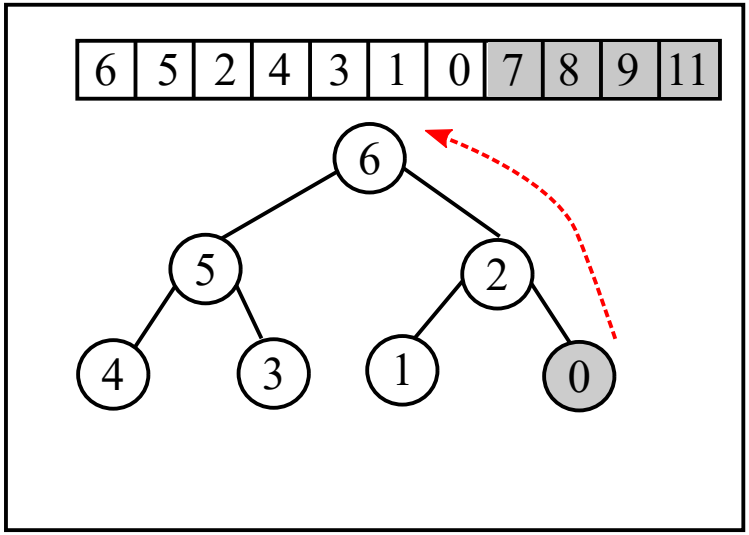
Algoritmul *Heapsort* - Exemplu



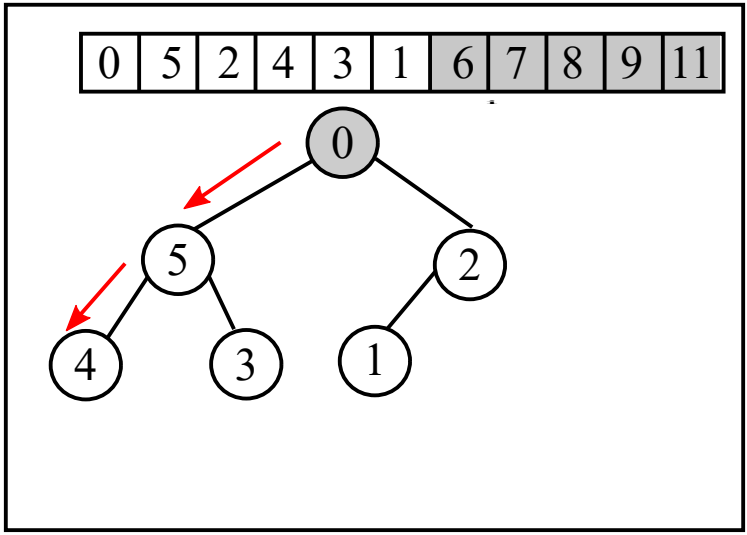
Algoritmul *Heapsort* - Exemplu



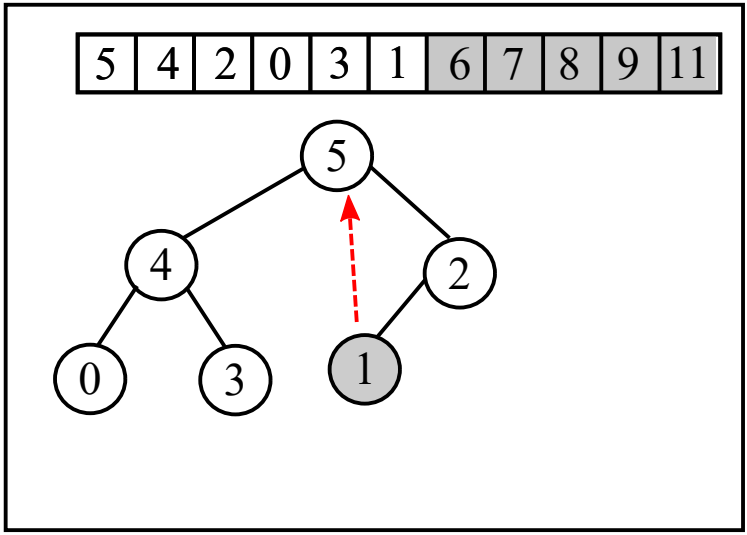
Algoritmul *Heapsort* - Exemplu



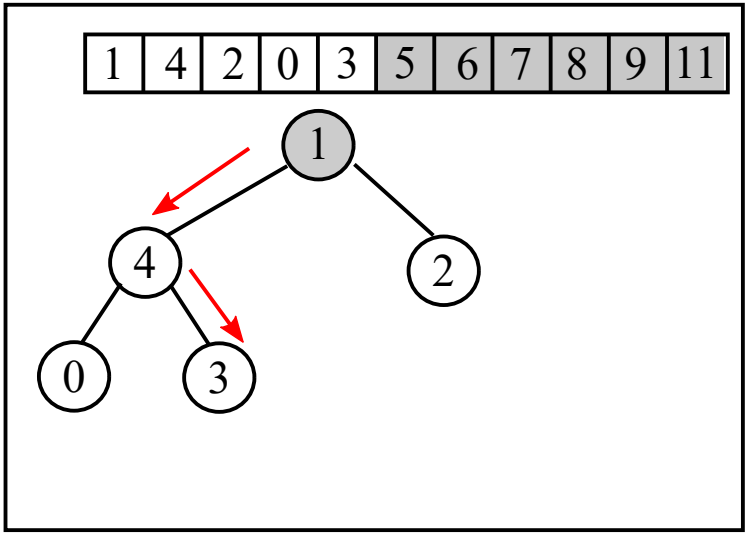
Algoritmul *Heapsort* - Exemplu



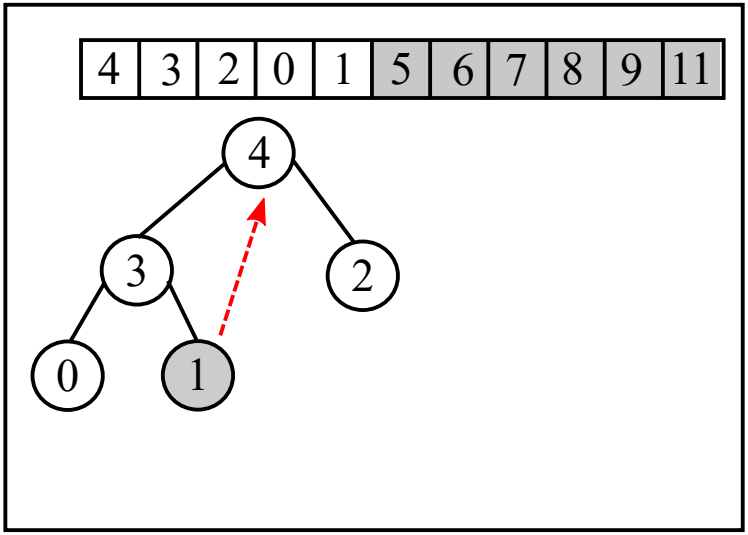
Algoritmul *Heapsort* - Exemplu



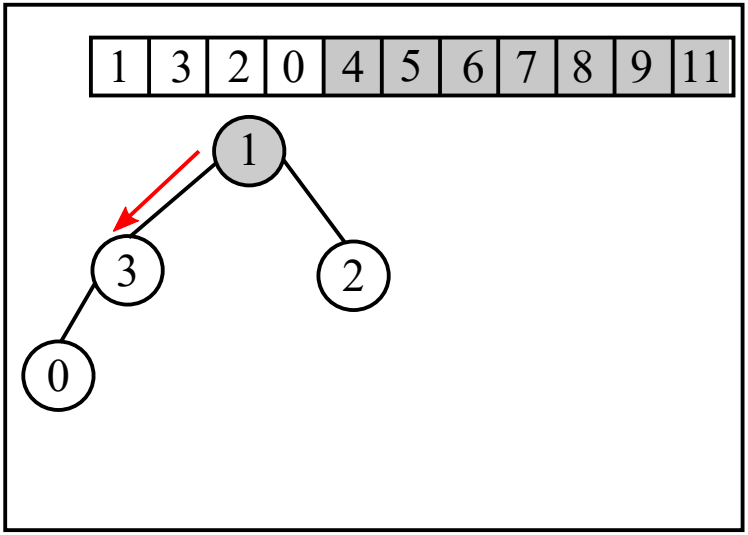
Algoritmul *Heapsort* - Exemplu



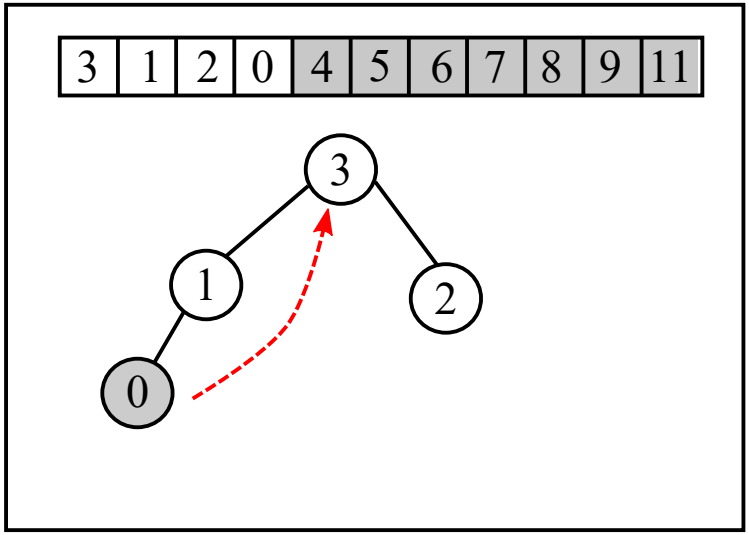
Algoritmul *Heapsort* - Exemplu



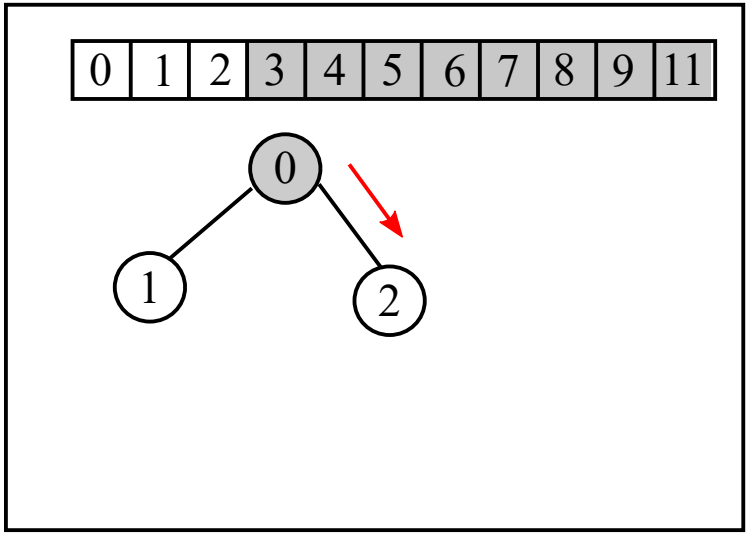
Algoritmul *Heapsort* - Exemplu



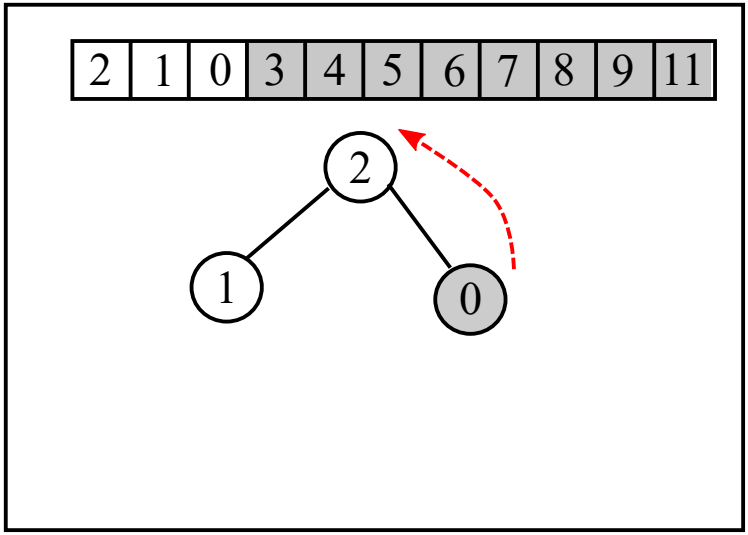
Algoritmul *Heapsort* - Exemplu



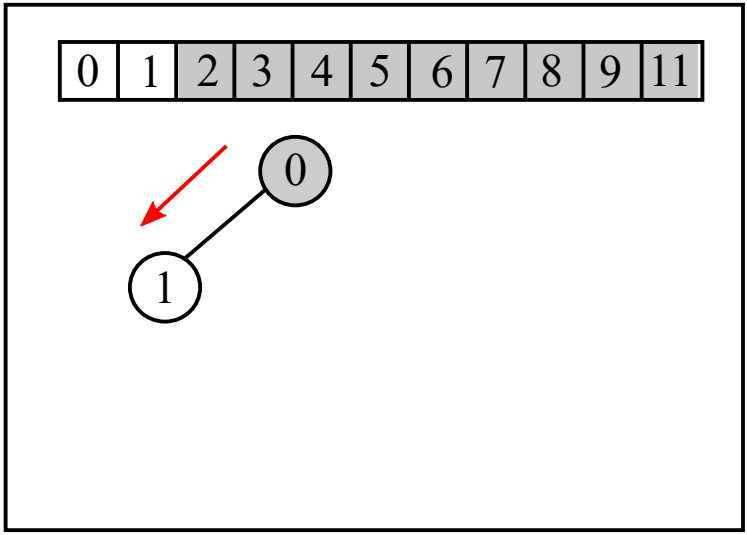
Algoritmul *Heapsort* - Exemplu



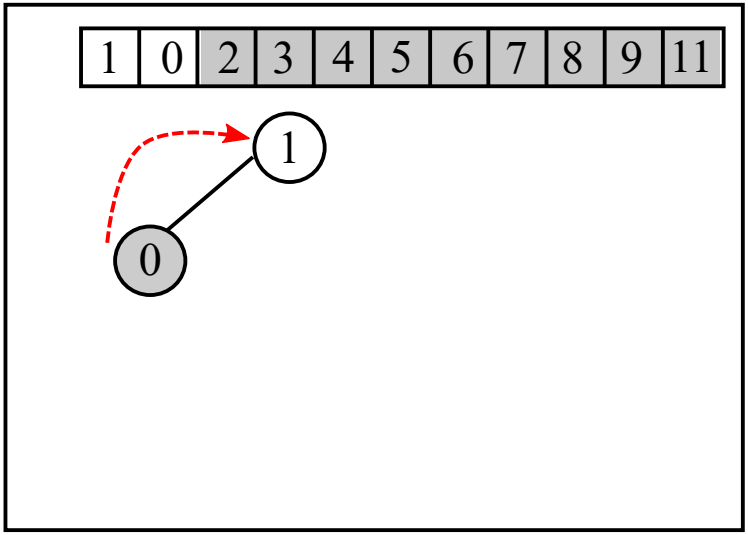
Algoritmul *Heapsort* - Exemplu



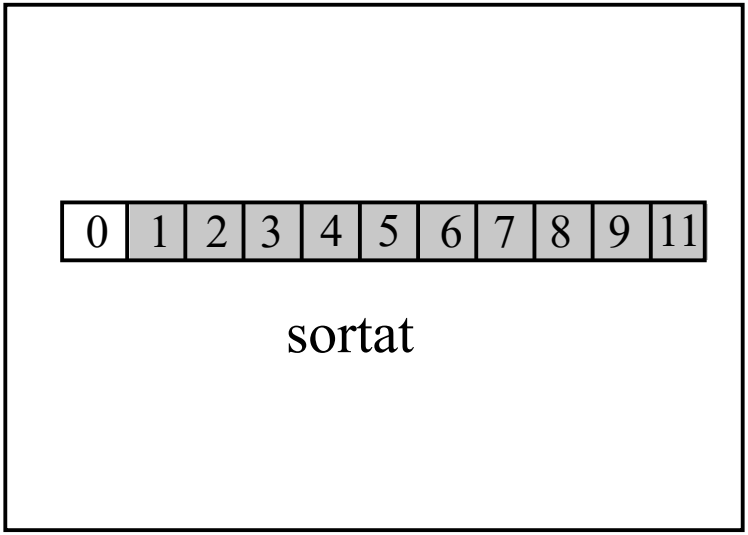
Algoritmul *Heapsort* - Exemplu



Algoritmul *Heapsort* - Exemplu



Algoritmul *Heapsort* - Exemplu



Cozi de prioritate

Definiție: O coadă de prioritate este o structură de date utilizată pentru păstrarea unei mulțimi dinamice de date S în care fiecărui element i se asociază o valoare numită *prioritate*.

Pentru gestionarea eficientă a cozilor de prioritate se utilizează heap-uri.

Există cozi de max-prioritate - heap-max - și cozi de min-prioritate - heap-min.

Cozi de prioritate - Operații

Operații în cozi de prioritate (max-heap):

- Inserția unui element nou
- Determinarea elementului de prioritatea maximă
- Extragerea elementului de prioritate maximă
- Creșterea priorității unui element

Cozi de prioritate - Operații

1. Determinarea elementului cu prioritate maximă - se află pe prima poziție în heap.

Algoritm 4: PRIORITY-MAX

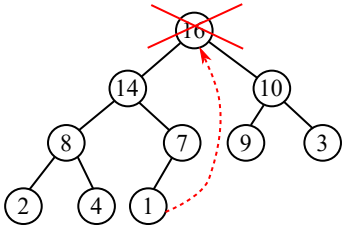
Intrare: un heap cu câmpurile H și *size*
return $H[0]$

Complexitate: $O(1)$

Cozi de prioritate - Operații

H:

16	14	10	8	7	9	3	2	4	1



2. Extragerea maximului

Algoritm 5: PRIORITY-EXTRACT-MAX

Intrare: un heap cu câmpurile H și $size$

$H[0] \leftarrow H[H.size - 1]$

$H.size \leftarrow H.size - 1$

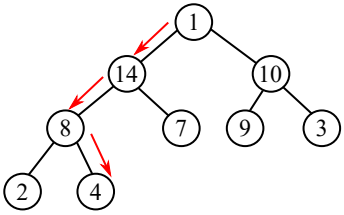
Max-Heapfy(H, 0)

Complexitate: $O(\log_2 n)$

Cozi de prioritate - Operații

H:

1	14	10	8	7	9	3	2	4



2. Extragerea maximului

Algoritm 5: PRIORITY-EXTRACT-MAX

Intrare: un heap cu câmpurile H și $size$

$H[0] \leftarrow H[H.size - 1]$

$H.size \leftarrow H.size - 1$

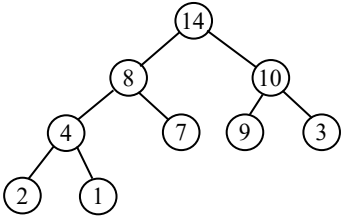
Max-Heapfy(H, 0)

Complexitate: $O(\log_2 n)$

Cozi de prioritate - Operații

H:

14	8	10	4	7	9	3	2	1



2. Extragerea maximului

Algoritm 5: PRIORITY-EXTRACT-MAX

Intrare: un heap cu câmpurile H și $size$

$H[0] \leftarrow H[H.size - 1]$

$H.size \leftarrow H.size - 1$

Max-Heapfy(H, 0)

Complexitate: $O(\log_2 n)$

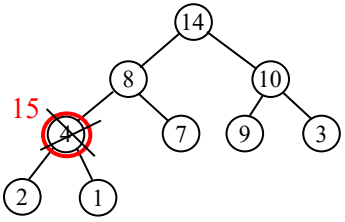
Cozi de prioritate - Operații

3. Creșterea priorității unui element

Creșterea priorității elementului de pe poziția $i \Rightarrow$ posibil să se strice proprieatea de heap-max,$\Rightarrow$ se merge din părinte în părinte către rădăcină, până se găsește o poziție potrivită pentru noua valoare.

H:

14	8	10	4	7	9	3	2	1



Algoritm 6: INCREASE-PRIORITY

Intrare: poziția i, valoarea val cu care se modifică elementul $H[i]$

daca $val > H[i]$ **atunci**
- $H[i] \leftarrow val$
- $p \leftarrow (i-1)/2$
- **cat_timp** $i > 0$ *si* $H[p] < val$ **executa**
  - $H[i] \leftarrow H[p]$
  - $i \leftarrow p$
  - $p \leftarrow (i-1)/2$
- **sfarsit_cat_timp**
- $H[i] \leftarrow val$

sfarsit_daca

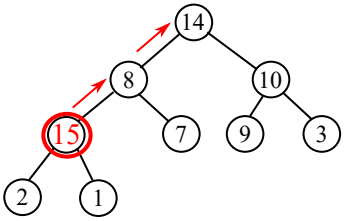
Cozi de prioritate - Operații

3. Creșterea priorității unui element

Creșterea priorității elementului de pe poziția $i \Rightarrow$ posibil să se strice proprieatea de heap-max,$\Rightarrow$ se merge din părinte în părinte către rădăcină, până se găsește o poziție potrivită pentru noua valoare.

H:

14	8	10	15	7	9	3	2	1



Algoritm 6: INCREASE-PRIORITY

Intrare: poziția i, valoarea val cu care se modifică elementul $H[i]$

daca $val > H[i]$ **atunci**
- $H[i] \leftarrow val$
- $p \leftarrow (i-1)/2$
- **cat_timp** $i > 0$ *si* $H[p] < val$ **executa**
  - $H[i] \leftarrow H[p]$
  - $i \leftarrow p$
  - $p \leftarrow (i-1)/2$
- **sfarsit_cat_timp**
- $H[i] \leftarrow val$

sfarsit_daca

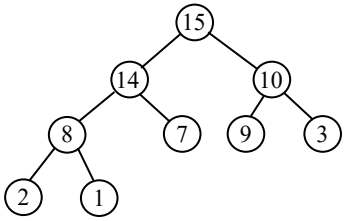
Cozi de prioritate - Operații

3. Creșterea priorității unui element

Creșterea priorității elementului de pe poziția $i \Rightarrow$ posibil să se strice proprieatea de heap-max,$\Rightarrow$ se merge din părinte în părinte către rădăcină, până se găsește o poziție potrivită pentru noua valoare.

H:

15	14	10	8	7	9	3	2	1



Algoritm 6: INCREASE-PRIORITY

Intrare: poziția i, valoarea val cu care se modifică elementul $H[i]$

daca $val > H[i]$ **atunci**
- $H[i] \leftarrow val$
- $p \leftarrow (i-1)/2$
- **cat_timp** $i > 0$ *si* $H[p] < val$ **executa**
  - $H[i] \leftarrow H[p]$
  - $i \leftarrow p$
  - $p \leftarrow (i-1)/2$
- **sfarsit_cat_timp**
- $H[i] \leftarrow val$

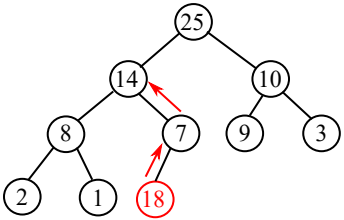
sfarsit_daca

Cozi de prioritate - Operații

Insert(18)

H:

25	14	10	8	7	9	3	2	1	18



4. Inserarea unui element

Inserția: se mărește dimensiunea heap-ului, se plasează noul element pe ultima poziție cu prioritatea considerată 0 și apoi se aplică funcția INCREASE-PRIORITY

Algoritm 7: PRIORITY-INSERT

Intrare: elementul *val*, care se inserează în coadă

$H[H.size] \leftarrow 0$

$H.size \leftarrow H.size + 1$

Increase-Priority(H, H.size -1, val)

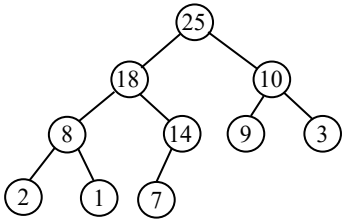
Complexitate: $O(\log_2 n)$

Cozi de prioritate - Operații

Insert(18)

H:

25	18	10	8	14	9	3	2	1	7



4. Inserarea unui element

Inserția: se mărește dimensiunea heap-ului, se plasează noul element pe ultima poziție cu prioritatea considerată 0 și apoi se aplică funcția INCREASE-PRIORITY

Algoritm 7: PRIORITY-INSERT

Intrare: elementul *val*, care se inserează în coadă

$H[H.size] \leftarrow 0$

$H.size \leftarrow H.size + 1$

Increase-Priority(H, H.size -1, val)

Complexitate: $O(\log_2 n)$

Cozi de prioritate

Observații:

- Pe un max-heap se pot implementa cu complexiatea $O(\log_2 n)$ operații cu cozi de prioritate.

Cozi de prioritate

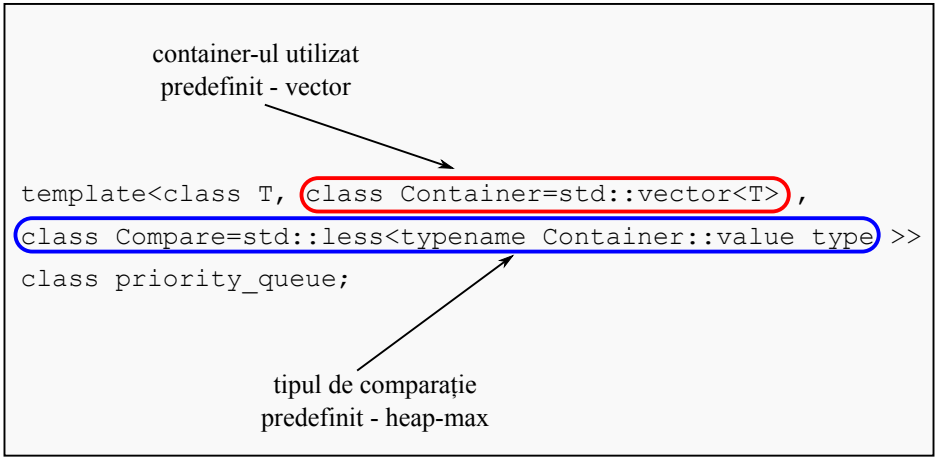
Observații:

- Pe un max-heap se pot implementa cu complexiatea $O(\log_2 n)$ operații cu cozi de prioritate.

- Construcția unui heap-max, care s-a făcu prin apelarea funcției BUILD-MAX-HEAP, se poate realiza și prin inserții succesive ale nodurilor în heap.

STL - Adaptori de containere - coadă de priorități

priority_queue:



STL - Adaptori de containere - coadă de priorităţi

Comaparator:

- un comparator - *Compare* - este definit astfel încât returnează *true* dacă primul argument este "înaintea" celui de al doilea (prin ordinea dată de comparator).

STL - Adaptori de containere - coadă de priorități

Comaparator:

- un comparator - *Compare* - este definit astfel încât returnează *true* dacă primul argument este "înaintea" celui de al doilea (prin ordinea dată de comparator).
- deoarece o coada de priorități max consideră întâi elementele de prioritate mai mare, în vârful cozii se pune al doilea argument din comparator

STL - Adaptori de containere - coadă de priorități

Comaparator:

- un comparator - *Compare* - este definit astfel încât returnează *true* dacă primul argument este "înaintea" celui de al doilea (prin ordinea dată de comparator).
- deoarece o coada de priorități max consideră întâi elementele de prioritate mai mare, în vârful cozii se pune al doilea argument din comparator
- Concluzie:

STL - Adaptori de containere - coadă de priorități

Comaparator:

- un comparator - *Compare* - este definit astfel încât returnează *true* dacă primul argument este "înaintea" celui de al doilea (prin ordinea dată de comparator).
- deoarece o coada de priorități max consideră întâi elementele de prioritate mai mare, în vârful cozii se pune al doilea argument din comparator
- Concluzie:
  - pentru max-heap se folosește comparația $<$ dată prin `std::less<T>`

STL - Adaptori de containere - coadă de priorități

Comaparator:

- un comparator - *Compare* - este definit astfel încât returnează *true* dacă primul argument este "înaintea" celui de al doilea (prin ordinea dată de comparator).
- deoarece o coada de priorități max consideră întâi elementele de prioritate mai mare, în vârful cozii se pune al doilea argument din comparator
- Concluzie:
  - pentru max-heap se folosește comparația $<$ dată prin `std::less<T>`
  - pentru min-heap se folosește comparația $>$ dată prin `std::greater<T>`

STL - Adaptori de containere - coadă de priorități

Comaparator:

- un comparator - *Compare* - este definit astfel încât returnează *true* dacă primul argument este "înaintea" celui de al doilea (prin ordinea dată de comparator).
- deoarece o coada de priorități max consideră întâi elementele de prioritate mai mare, în vârful cozii se pune al doilea argument din comparator
- Concluzie:
  - pentru max-heap se folosește comparația $<$ dată prin `std::less<T>`
  - pentru min-heap se folosește comparația $>$ dată prin `std::greater<T>`
- pentru tipuri de date, pentru care nu există comparație predefinită, se poate defini un comparator propriu.

Adaptori de containere - coadă de priorități

priority_queue are structură de heap deci:

- acces în timp constant la elementul de prioritate maximă

Adaptori de containere - coadă de priorități

priority_queue are structură de heap deci:

- acces în timp constant la elementul de prioritate maximă
- complexitate logaritmică pentru push / pop

Adaptori de containere - coadă de priorități

priority_queue are structură de heap deci:

- acces în timp constant la elementul de prioritate maximă
- complexitate logaritmică pentru push / pop
- containerul predefinit - vector

Adaptori de containere - coadă de priorități

priority_queue: Funcții membre

- Constructor:
  - predefinit: `std::priority_queue<int> first;`
  - de inițializare

```
int v[]= {10,60,50,20};
std::priority_queue<int> second (v,v+4);

std::priority_queue<int, std::vector<int>, std::greater<int> >
third (v,v+4);
```

Adaptori de containere - coadă de priorități

priority_queue: Funcții membre

- **empty** - testează dacă coada este vidă

Adaptori de containere - coadă de priorități

priority_queue: Funcții membre

- **empty** - testează dacă coada este vidă
- **size** - determină câte elemente sunt în coadă

Adaptori de containere - coadă de priorități

priority_queue: Funcții membre

- **empty** - testează dacă coada este vidă
- **size** - determină câte elemente sunt în coadă
- **top** - returnează elementul de prioritate maximă

Adaptori de containere - coadă de priorități

priority_queue: Funcții membre

- **empty** - testează dacă coada este vidă
- **size** - determină câte elemente sunt în coadă
- **top** - returnează elementul de prioritate maximă
- **push** - pune un element în coadă

Adaptori de containere - coadă de priorități

priority_queue: Funcții membre

- **empty** - testează dacă coada este vidă
- **size** - determină câte elemente sunt în coadă
- **top** - returnează elementul de prioritate maximă
- **push** - pune un element în coadă
- **pop** - elimină elementul de prioritate maximă.