HEAP. COADĂ DE PRIORITATE

1 Heap

Definiţie: Un heap binar este un arbore binar complet - fieacare nod intern are exact doi descendenţi, cu excepţia 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 - memorat cu ajutorul unui tablou unidimensional (vector) - *array*. În plus există o ordonare a cheilor într-un heap binar, determinată de tipul heap-ului.

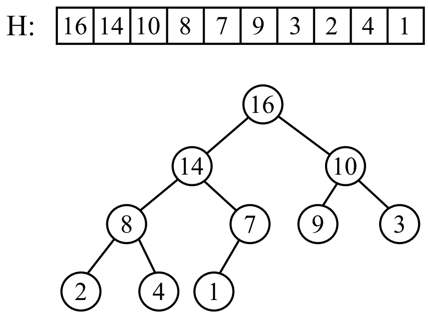

Figura 1: Exemplu de heap-max

1. **Max-heap**: informaţia din fiecare nod este mai mare decât informaţia din oricare descendent al său. Maximul din heap se află în rădăcină (fig. 1).
2. **Min-heap**: informaţia din fiecare nod este mai mică decât informaţia din oricare descendent al său. Minimul din heap se află în rădăcină.

Pentru implementare se poate utiliza structura heap H în care $size$ - numărul de chei din Heap. Rădăcina heap-ului se află pe prima poziţie a heapului $H[0]$. Pentru fiecare nod aflat pe poziţia i:

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

Observaţii:

- Înălţimea arborelui care reprezintă heap-ul este $\log_2 n$ unde $n = H.size$ este numărul de noduri din heap.
- Complexitatea operaţiilor de bază este proporţională cu înălţimea arborelui, adică $O(\log_2 n)$.

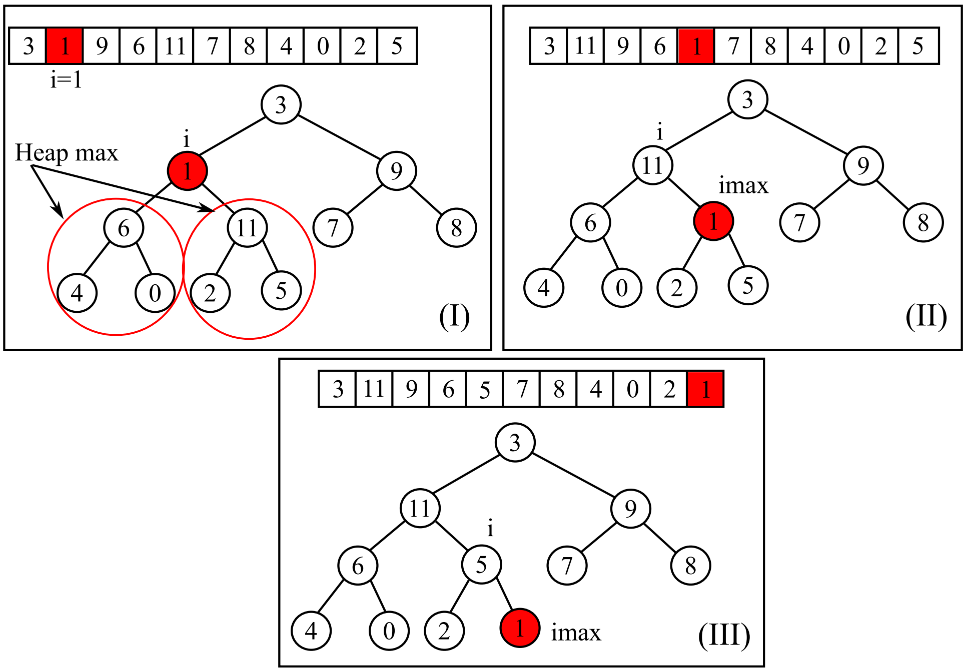

Figura 2: Funcţia MAX_HEAP porneşte în acest exemplu de la nodul i marcat cu roşu în fig. (I). Cei doi subarbori ai săi sunt heap-max. Funcţia are două apeluri recursive, ilustrate în etapele (II) şi (III).

Construcţia unui heap - max

Considerând un vector de elemente, pentru transformarea acestuia într-un heap-max sunt necesare două etape, reprezentate prin funcţiile:

- **MAX_HEAP**(H, i): în care H heap cu $H.size$ elemente şi i indicele din vector a nodului de la se începe. Premiza este aceea că subarborii nodului i sunt heap-max şi doar informaţia din nodul i strică eventual această proprietate. Funcţia **MAX_HEAP** reface proprietatea de heap-max. Această funcţie se întâlneşte în literatură şi sub denumirea *Sift-Down*.
- **CONSTR_HEAP**(H): pe baza funcţiei **MAX_HEAP** se construieşte heap-ul.

Algoritm 1: MAX-HEAP

Intrare: Un heap H cu numărul de elemente $size$, poziţia 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-HEAP(H, $imax$)
sfarsit_daca

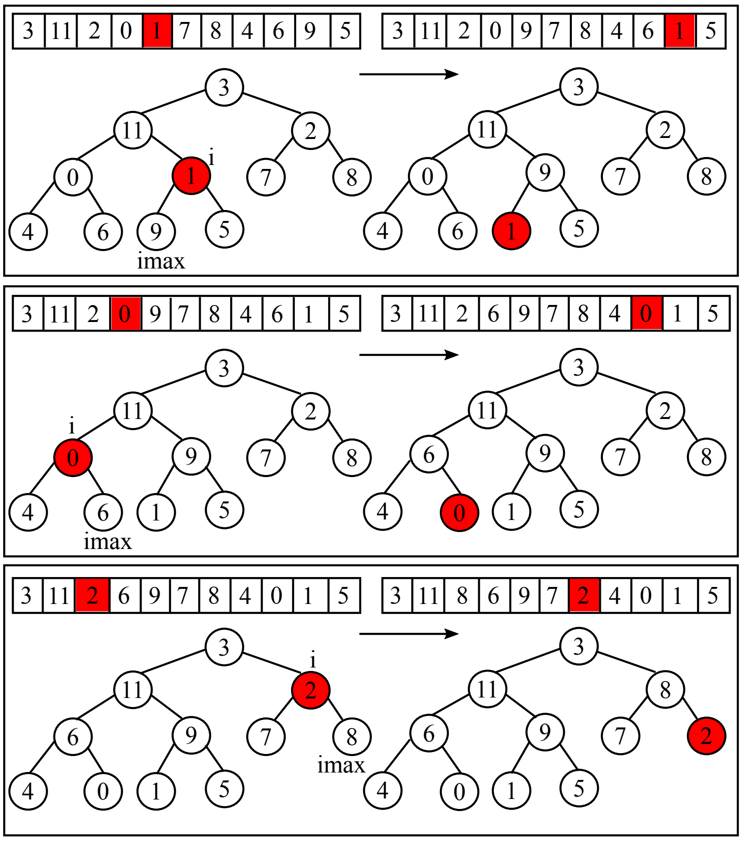

Figura 3: Funcţia CONSTR_HEAP porneşte în acest exemplu de la nodul de pe poziţia a 5-a în heap-ul H, primul nod din dreapta vectorului ce are descendenţi. Apoi se continuă cu nodurile de poziţiile 4 şi 3.

În figurile 3 şi 4 este ilustrată funcţionarea acestui algoritm.

Complexitate: Algoritmul porneşte de la nodul i şi ajunge în cel mai defavorabil caz la o frunză, deci complexitatea depinde de înălţimea arborelui care este $h = log_2 n$. Deci $T(n) = O(log_2 n)$.

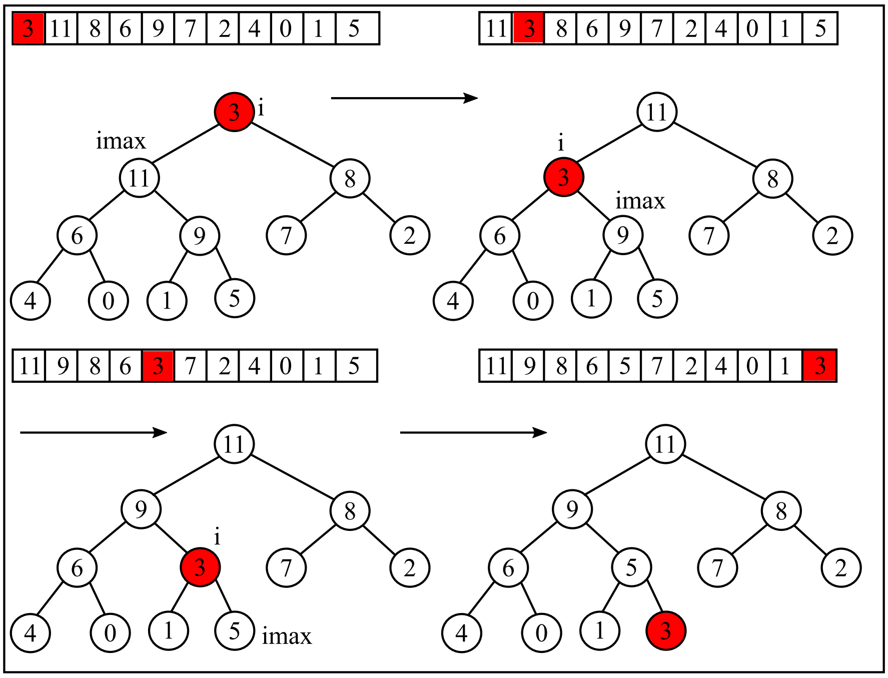

Figura 4: Funcţia MAX_HEAP continuă de la poziţia 2, la care însă nu sunt necesare modificări. Nodul cu cheia 11 satisface proprietatea de max-heap relativ la descendenţii săi, astfel ca se trece la nodul de pe poziţia 1. În final se obţine un heap-max.

Se observă faptul că frunzele sunt heap-max. Atunci e suficient să pornim de la primul nod intern - pornind de la dreapta heap-ului H, adică primul care are cel puţin un descendent. Nodul respectiv se află pe poziţia $H.size/2 - 1$, iar copiii săi sunt frunze, deci heap-max. Se aplică funcţia de MAX_HEAP pentru nodul $H.size/2 - 1$, după care se trece la nodul precedent din vector şi aşa mai departe, până la rădăcină.

Algoritm 2: CONSTR_HEAP

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

În figura 4 este ilustrat procedeul de construcţie a unui heap max.

Complexitate: fiecare apel al funcţiei MAX_HEAP are complexitatea $O(\log_2 n)$.

CONSTR_HEAP efectuează $O(n)$ atfel de apeluri. Deci $T(n)$, unde prin T am notat complexitatea, pentru funcţia CONSTR_HEAP este mărginită superior de $n \log_2 n$. Se demonstrează în literatură faptul că funcţia are complexitate liniară, adică $T(n) = O(n)$,

$n = H.size$.

2 Algoritmul de sortare *HeapSort*

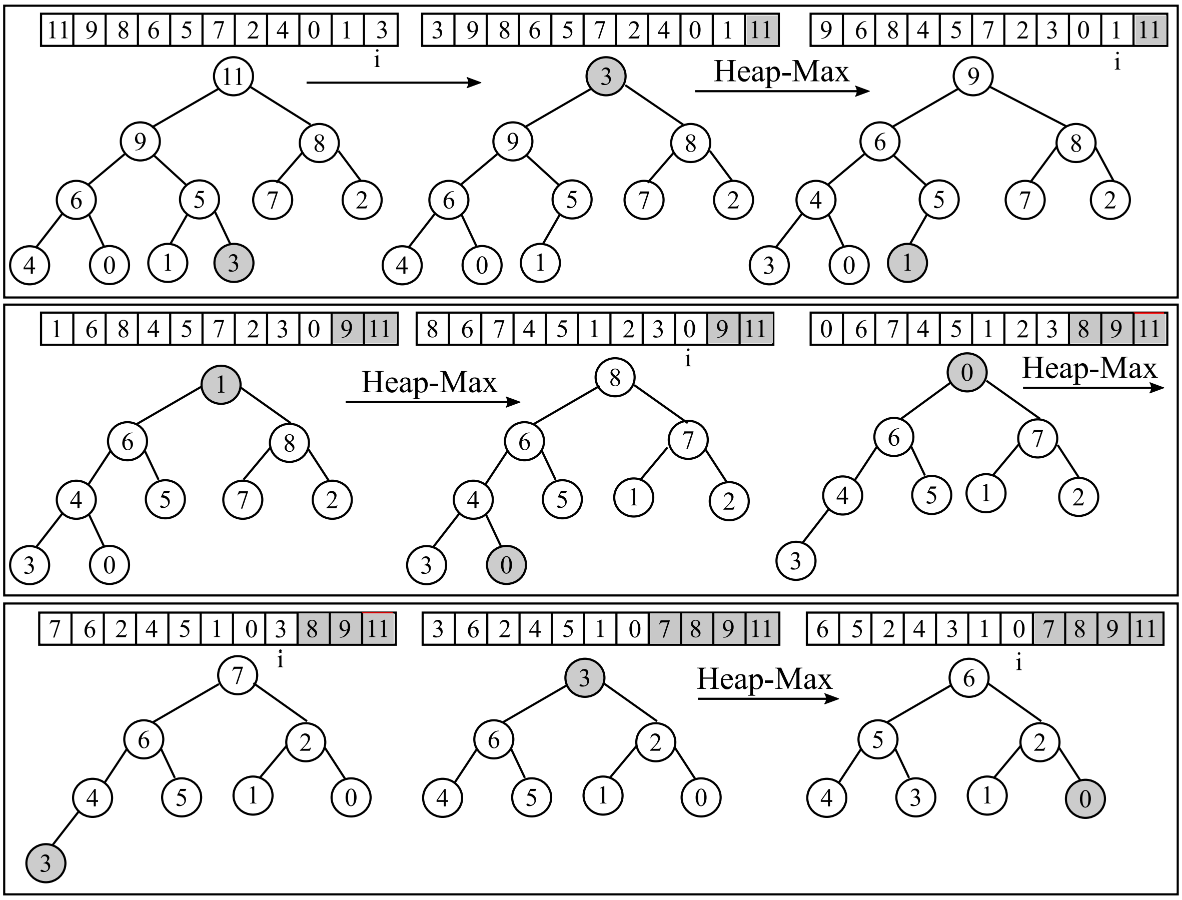

Figura 5: Algoritmul Heapsort.

Se observă faptul că într-un heap-max elementul maxim este plasat în rădăcina arborelui corespunzător, deci pe prima poziţie din vectorul date. În vectorul sortat crescător, elementul maxim trebuie să se afle pe ultima poziţie. Astfel vectorul poate fi sortat prin următorul algoritm: se interschimbă primul element din vectorul heap-ului cu ultimul. Se reduce dimensiunea heap-ului, apoi se reface proprietatea de heap-max aplicând funcţia MAX_HEAP începând din vârful heap-ului. Procedura se reia pentru acest heap redus.

Algoritm 3: Heapsort

Intrare: Un vector v cu n

Construieşte heap-ul H cu funcţia 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

În figurile 5 şi 6 este ilustrată funcţionarea algoritmului Heapsort.

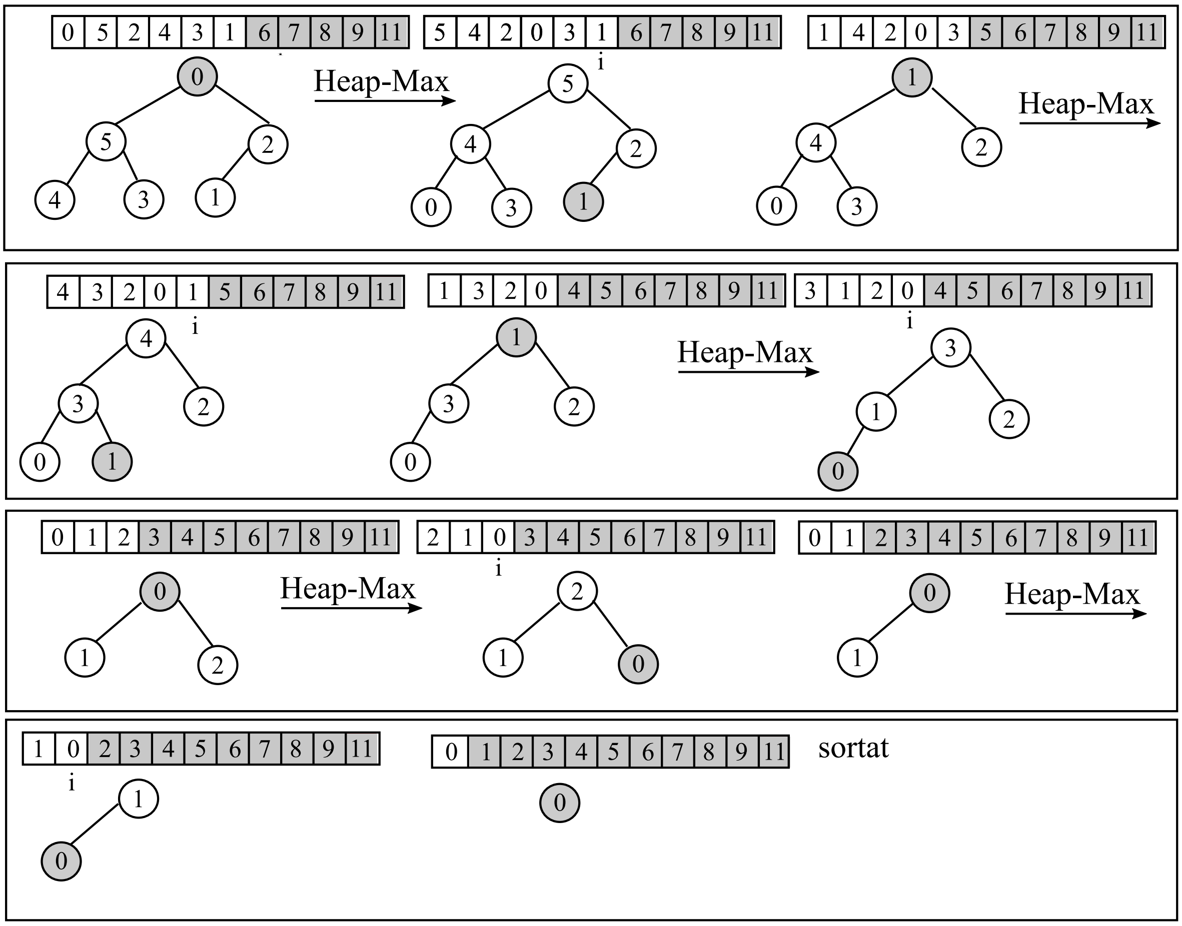

Figura 6: Algoritmul Heapsort.

Complexitate:

- apelul funcţiei CONSTR_HEAP este $O(n)$
- apelul pentru MAX_HEAP este $O(\log_2 n)$
- funcţia MAX_HEAP se apelează de $n-1$ ori

Rezultă $T(n) = O(n \log_2 n)$.

3 Cozi de prioritate

Definiţie: O coadă de prioritate este o structură de date utilizată pentru păstrarea unei mulţimi 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 - şi cozi de min-prioritate - heap-min.

În continuare vom considera cozi cu max-prioritate.

Operaţii în cozi de prioritate (max-heap):

- Inserţia unui element nou
- Determinarea elementului de prioritatea maximă
- Extragerea elementului de prioritate maximă
- Creşterea priorităţii unui element

Utilizare: Cozile de prioritate pot fi utilizate pentru gestionarea proceselor pe un calculator. La fiecare moment se execută procesul de prioritate maximă. Procese noi se pot insera în coadă.
Alt exemplu de aplicaţie este în implementarea unor algoritmi de căutare informată (Dijkstra, A^*).

1. Determinarea elementului cu prioritate maximă: acesta se află în nodul rădăcină al heap-max:

   ```
   CP_MAX(H)
   RETURN H[0]
   ```

   Complexitate: $O(1)$

2. Extragerea maximului: Se plasează ultimul element din heap pe prima poziţie, se scade dimensiunea heap-ului cu o unitate iar apoi se reface heap-ul prin apelul MAX_HEAP.

   Algoritm 4: PRIORITY-EXTRACT-MAX

   Intrare: un heap cu câmpurile H şi $size$
   $H[0] \leftarrow H[H.size - 1]$
   $H.size \leftarrow H.size - 1$
   Max-Heapfy(H, 0)

   Complexitate: $O(\log_2 n)$ - se aplică o singură dată MAX_HEAP.

În figura 7 este prezentat un exemplu pentru extragerea maximului.

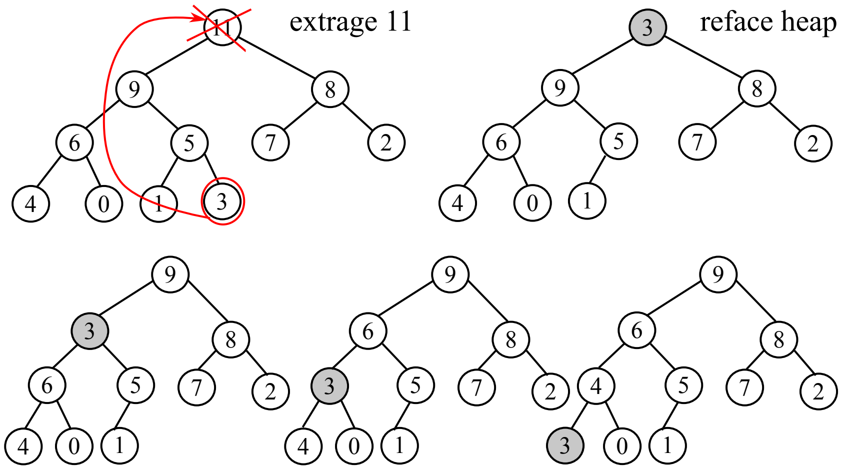

Figura 7: Extragerea maximului.

3. Creşterea priorităţii elementului de pe poziţia i. Prin această modificare este posibil să se strice proprieatea de heap-max, deoarece s-ar putea ca noua valoare a elementului de pe poziţia i să fie mai mare decât a părintelui său. Pentru aceasta se merge din părinte în părinte către rădăcină, până se găseşte o poziţie potrivită pentru noua valoare. Algoritmul următor este cunoscut în literatură şi sub denumirea de *Sift-Up*.

Algoritm 5: INCREASE-PRIORITY

Intrare: poziţia 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

Complexitate: $O(\log_2 n)$ - se porneşte de la o frunză către rădăcină şi deci complexitatea depinde de înălţimea arborelui.

În figura 8 este prezentat un exemplu pentru creşterea priorităţii.

4. Inserţia unui element nou într-un max-heap: se măreşte dimensiunea heap-ului, se plasează noul element pe ultima poziţie cu prioritatea considerată 0 şi apoi se aplică funcţia CP_CRESTE_PRIORITATE pentru acest nou element cu valoarea priorităţii asociate.

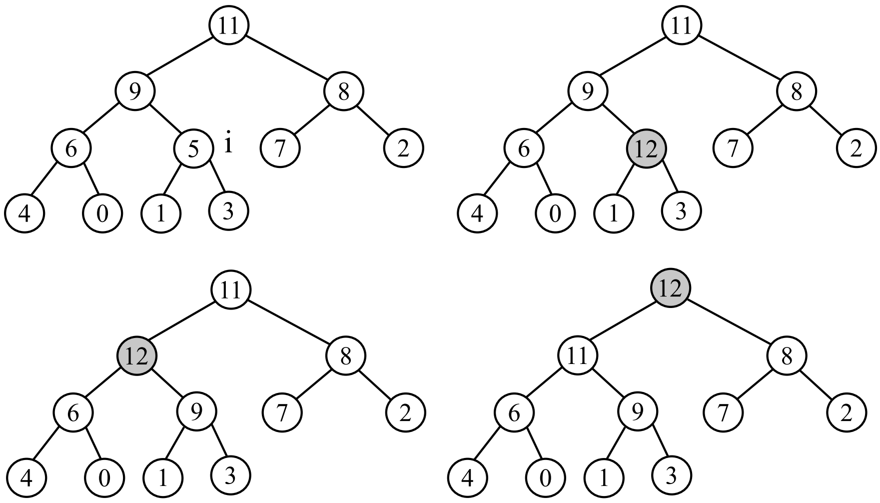

Figura 8: Creşterea priorităţii nodului cu cheia 5 la valoarea 12.

Algoritm 6: 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: $O(\log_2 n)$ - se porneşte de la o frunză către rădăcină şi deci complexitatea depinde de înălţimea arborelui.

Observaţii:

- Pe un max-heap se pot implementa cu complexiatea $O(\log_2 n)$ operaţii cu cozi de prioritate.
- Construcţia unui heap-max, care s-a făcu prin apelarea funcţiei MAX_HEAP, se poate realiza şi prin inserţii succesive ale nodurilor în heap.