Heap-uri MIN-MAX

Mihai Scorțaru

În cadrul acestui articol vom prezenta o altă structură de date derivată din heap-urile binare. Principala diferență dintre heap-urile Min-Max și heap-urile binare este faptul că, la fel ca și heap-urile de intervale, heap-urile Min-Max permit determinarea atât a elementului minim, cât și a elementului maxim. Diferența dintre heap-urile Min-Max și heap-urile de intervale constă în faptul că nu este modificată structura elementelor heap-ului (fiecare element conține un singur element), ci doar proprietatea de heap.

Un *heap Min-Max* este o coadă de priorități cu două capete în care relația de ordonare dintre elemente este bazată pe ordinea min-max.

Cel mai mic element al structurii se va afla în rădăcina arborelui care reprezintă *heap*-ul. Valorile corespunzătoare elementelor de pe nivelurile pare vor fi mai mici sau egale cu valorile corespunzătoare descendenților lor direcți, iar valorile corespunzătoare elementelor de pe nivelurile impare vor fi mai mari sau egale cu valorile corespunzătoare descendenților lor direcți.

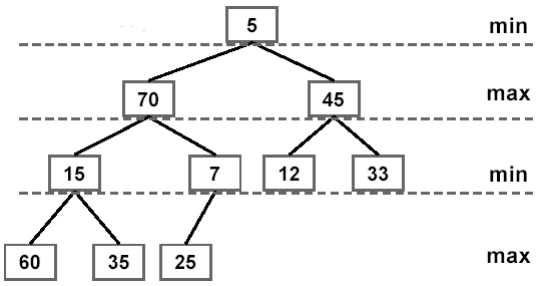

Figura 1: Heap Min-Max

Nivelurile pare sunt uneori numite ***niveluri de minim*** sau ***niveluri min***, iar nivelurile impare sunt numite ***niveluri de maxim*** sau ***niveluri max***. Am considerat că numerotarea nivelurilor începe de la 0.

În cazul în care numerotarea începe de la 1, atunci nivelurile pare vor fi niveluri de maxim, iar nivelurile impare vor fi niveluri de minim.

Structura unui *heap Min-Max* este prezentată în figura 1.

Se observă că elementul maxim este păstrat în unul dintre fiii rădăcinii arborelui care reprezintă *heap*-ul *Min-Max.*

Există și noțiunea de *heap Max-Min* care reprezintă o structură de date asemănătoare în care cel mai mare element se află în rădăcina arborelui, valorile corespunzătoare elementelor de pe nivelurile pare vor fi mai mari sau egale cu valorile corespunzătoare descendenților lor direcți, iar valorile corespunzătoare elementelor de pe nivelurile impare vor fi mai mici sau egale cu valorile corespunzătoare descendenților lor direcți. Am considerat din nou că numerotarea nivelurilor începe de la 0.

Structura de date

Un *heap Min-Max* care conține n elemente poate fi păstrat cu ajutorul unui vector ale căror elemente sunt accesate folosind indici cuprinși între 1 și n.

Elementul de pe poziția i a vectorului va corespunde unui element aflat pe nivelul $[\log_2 i]$ al arborelui care reprezintă *heap-ul Min-Max*.

La fel ca și în cazul *heap*-urilor binare, fiii unui element aflat pe poziția i se vor afla pe pozițiile $2 \cdot i$ și $2 \cdot i + 1$, iar părintele unui element aflat pe poziția i se va afla pe poziția $[i / 2]$.

Diagrama Hasse

O diagramă *Hasse* este o diagramă care prezintă relația de ordine implicită într-o anumită structură.

Pentru *heap*-urile binare diagrama *Hasse* este dată chiar de *heap*-ul însuși.

Pentru *heap-urile Min-Max* diagrama *Hasse* este mult mai complexă. Diagrama *Hasse* corespunzătoare *heap-ului Min-Max* din figura 1 este prezentată în figura 2.

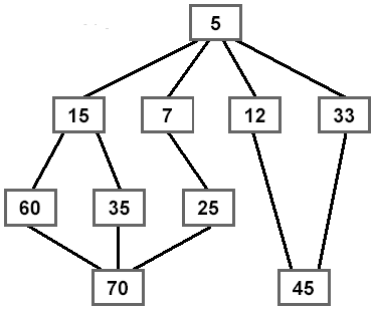

Figura 2: Diagrama Hasse

Se observă că avem o structură în care fiecare element are cel mult patru fii, un element poate avea până la patru părinți, și valoarea corespunzătoare unui element este mai mică sau egală cu valorile corespunzătoare descendenților săi direcți.

Operații

Structura de *heap Min-Max* permite executarea rapidă a următoarelor operații:

- transformarea unui vector de elemente într-un *heap Min-Max*;
- adăugarea unui element;
- determinarea elementului minim;
- determinarea elementului maxim;
- eliminarea elementului minim;
- eliminarea elementului maxim.

Construirea unui heap Min-Max

Vom prezenta acum modalitatea prin care poate fi transformat un vector cu *n* elemente într-un *heap Min-Max*.

Pentru a construi un astfel de *heap* elementele vectorului sunt parcurse de la dreapta la stânga.

Pentru fiecare element se va verifica dacă el se află pe un nivel de minim sau pe un nivel de maxim.

Vom prezenta în continuare situația în care elementul se află pe un nivel de minim.

Subarborii ai căror rădăcini sunt date de fiii nodului curent sunt ordonați corect deoarece au fost creați la pașii anteriori, dar întregul subarbore care are ca rădăcină nodul curent ar putea să nu aibă structura corectă. În această ultimă situație elementul trebuie înlocuit cu unul dintre fii sau unul dintre fiii fiilor.

Datorită proprietății de *heap* descrise, valoarea elementul trebuie să fie mai mică sau egală cu valorile fiilor săi și, de asemenea, mai mică sau egală cu valorile fiilor fiilor săi.

Dacă această proprietate nu este respectată, va fi identificată cea mai mică valoare corespunzătoare unuia dintre cele șase elemente (fiii și fiii fiilor) și se va realiza o interschimbare a conținutului elementelor.

În cazul în care elementul a fost înlocuit cu un fiu al fiilor, procedeul va continua.

Situația este asemănătoare pentru elemente aflate pe nivelurile de maxim.

În figura 3 este exemplificat "traseul" unui element aflat pe un nivel de minim care nu are valoarea corectă (valoarea 55 înlocuiește valoarea 7, iar apoi valoarea 25).

"Traseul" unui element aflat pe un nivel de maxim este exemplificat în figura 4 (valoarea 6 înlocuiește valoarea 60 și apoi valoarea 15).

Prezentăm în cele ce urmează versiunile în pseudocod ale algoritmului general și a subalgoritmilor utilizați de acesta.

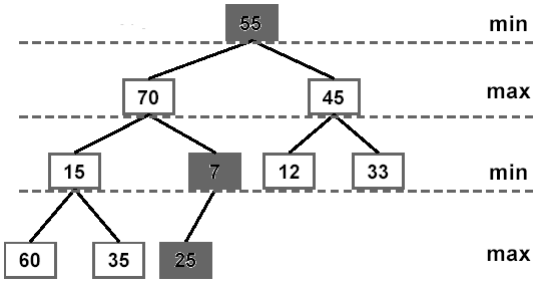

Figura 3: Coborârea unui element de pe un nivel de minim

```
algoritm CreareHeap(a, n)
// transformarea unui vector într-un
// heap Min-Max
  pentru i ← [n / 2], 1 pas -1
                              execută
    // verificarea tipului nivelului
    dacă [log2 i] este număr par
                              atunci
      CoboarăMin(a, i, n)
    altfel
      CoboarăMax(a, i, n)
    sfârșit dacă
  sfârșit pentru
sfârșit algoritm

subalgoritm CoboarăMin(a, i, n)
// coborârea unui element până când
// acesta ajunge în poziția curentă
  m ← Min(a, i, n)
  // se verifică pe ce nivel se află
  // poziția m
  niv ← [m / i]
  // niv = 1 → nu există fii sau nu
  // există valori mai mici
  dacă niv = 1 atunci
    returnează
  sfârșit dacă
  // niv = 2 → nu există fii ai fiilor
  // sau valoarea minimă corespunde
  // unuia dintre fii; există posibili-
  // tatea să avem niv = 3 dacă
  // valoarea minimă corespunde
  // celui de-al doilea fiu al rădăcinii
  dacă niv = 2 sau niv = 3 atunci
    interschimbă am și ai
  sfârșit dacă
  // niv = 4 → există fii ai fiilor
  // și valoarea minimă corespunde
  // unuia dintre fiii fiilor
  dacă niv ≥ 4 atunci
    interschimbă am și ai
    dacă am > a[m/2] atunci
      interschimbă am și a[m/2]
    sfârșit dacă
    CoboarăMin(a, m, n)
  sfârșit dacă
sfârșit subalgoritm

subalgoritm Min(a, i, n)
  m ← i
  // verificarea existenței primului
  // fiu
  dacă 2·i ≤ n atunci
    dacă am < a2·i atunci
      m ← 2·i
    sfârșit dacă
  altfel
    returnează m
  sfârșit dacă
  // verificarea existenței celui de-al
  // doilea fiu
  dacă 2·i + 1 ≤ n atunci
    dacă am < a2·i+1 atunci
      m ← 2·i + 1
    sfârșit dacă
  altfel
    returnează m
  sfârșit dacă
  // verificarea existenței primului
  // fiu al primului fiu
  dacă 4·i ≤ n atunci
    dacă am < a4·i atunci
      m ← 4·i
    sfârșit dacă
  altfel
    returnează m
  sfârșit dacă
```

```
    // verificarea existenței celui de-al
    // doilea fiu al primului fiu
    dacă 4·i + 1 ≤ n atunci
      dacă a_m < a_{4·i+1} atunci
        m ← 4·i + 1
      sfârșit dacă
    altfel
      returnează m
    sfârșit dacă
    // verificarea existenței primului
    // fiu al celui de-al doilea fiu
    dacă 4·i + 2 ≤ n atunci
      dacă a_m < a_{4·i+2} atunci
        m ← 4·i + 2
      sfârșit dacă
    altfel
      returnează m
    sfârșit dacă
    // verificarea existenței celui de-al
    // doilea fiu al celui de-al doilea fiu
    dacă 4·i + 3 ≤ n atunci
      dacă a_m < a_{4·i+3} atunci
        m ← 4·i + 3
      sfârșit dacă
    sfârșit dacă
    returnează m
sfârșit subalgoritm

subalgoritm CoboarăMax(a, i, n)
// coborârea unui element aflat pe
// un nivel de maxim până când
// acesta ajunge în poziția curentă
  m ← Max(a, i, n)
  //  se verifică pe ce nivel se află
  //  poziția m
  niv ← [m / i]
  // niv = 1 → nu există fii sau nu
  // există valori mai mari
  dacă niv = 1 atunci
    returnează
  sfârșit dacă
  // niv = 2 → nu există fii ai fiilor
  // sau valoarea maximă
  // corespunde unuia dintre fii
  // există posibilitatea să avem
  // niv = 3 dacă valoarea maximă
  // corespunde celui de-al doilea
  // fiu al rădăcinii
  dacă niv = 2 sau niv = 3 atunci
    interschimbă a_m și a_i
  sfârșit dacă
  // niv = 4 → există fii ai fiilor
  // și valoarea maximă corespunde
  // unuia dintre fiii fiilor
  dacă niv ≥ 4 atunci
    interschimbă a_m și a_i
    dacă a_m < a_{[m/2]} atunci
      interschimbă a_m și a_{[m/2]}
    sfârșit dacă
    CoboarăMin(a, m, n)
  sfârșit dacă
sfârșit subalgoritm

subalgoritm Max(a, i, n)
  m ← i
  // verificarea existenței primului
  // fiu
  dacă 2·i ≤ n atunci
    dacă a_m > a_{2·i} atunci
      m ← 2·i
    sfârșit dacă
  altfel
    returnează m
  sfârșit dacă
  // verificarea existenței celui de-al
  // doilea fiu
  dacă 2·i + 1 ≤ n atunci
    dacă a_m > a_{2·i+1} atunci
      m ← 2·i + 1
    sfârșit dacă
  altfel
    returnează m
  sfârșit dacă
```

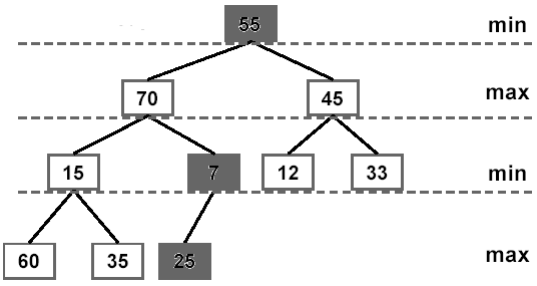

Figura 4: Coborârea unui element de pe un nivel de maxim

```
  // verificarea existenței primului
  // fiu al primului fiu
  dacă 4·i ≤ n atunci
    dacă a_m > a_{4·i} atunci
      m ← 4·i
    sfârșit dacă
  altfel
    returnează m
  sfârșit dacă
  // verificarea existenței celui de-al
  // doilea fiu al primului fiu
  dacă 4·i + 1 ≤ n atunci
    dacă a_m > a_{4·i+1} atunci
      m ← 4·i + 1
    sfârșit dacă
  altfel
    returnează m
  sfârșit dacă
  // verificarea existenței primului
  // fiu al celui de-al doilea fiu
  dacă 4·i + 2 ≤ n atunci
    dacă a_m > a_{4·i+2} atunci
      m ← 4·i + 2
    sfârșit dacă
  altfel
    returnează m
  sfârșit dacă
  // verificarea existenței celui de-al
  // doilea fiu al celui de-al doilea fiu
  dacă 4·i + 3 ≤ n atunci
    dacă a_m > a_{4·i+3} atunci
      m ← 4·i + 3
    sfârșit dacă
  sfârșit dacă
  returnează m
sfârșit subalgoritm
```

Determinarea minimului

Rădăcina arborelui care reprezintă *heap-ul Min-Max* conține întotdeauna valoarea minimă.

Așadar, pentru a determina valoarea minimă este suficient să returnăm primul element al vectorului cu ajutorul căruia păstrăm *heap-ul Min-Max*.

Evident, va trebui să verificăm dacă există un prim element al vectorului (*heap-ul Min-Max* nu poate fi vid).

Prezentăm în cele ce urmează varianta în pseudocod a algoritmului.

```
algoritm Minim(a, n)
  dacă n > 0 atunci
    returnează a_1
  altfel
    raportează o eroare
  sfârșit dacă
sfârșit algoritm
```

Determinarea maximului

Valoarea maximului corespunde întotdeauna unuia dintre cei doi fii ai rădăcinii.

Așadar, pentru a determina valoarea maximă este suficient să determinăm căruia dintre cei doi fii ai rădăcinii arborelui care reprezintă

heap-ul Min-Max îi corespunde o valoare mai mare.

Poate apărea situația în care avem un singur element. În acest caz rădăcina conține atât elementul minim, cât și elementul maxim.

Există și situația în care avem două elemente. În acest caz valoarea maximă corespunde unicului fiu al rădăcinii.

Evident, va trebui să verificăm și dacă există cel puțin un element al vectorului (*heap-ul Min-Max* nu poate fi vid).

Prezentăm în cele ce urmează varianta în pseudocod a algoritmului descris.

```
algoritm Maxim(a, n)
  dacă n > 0 atunci
    dacă n = 1 atunci
      returnează a1
    altfel
      dacă n = 2 atunci
        returnează a2
      altfel
        dacă a2 > a3 atunci
          returnează a2
        altfel
          returnează a3
        sfârșit dacă
      sfârșit dacă
    sfârșit dacă
  altfel
    raportează o eroare
  sfârșit dacă
sfârșit algoritm
```

Eliminarea minimului

Pentru a elimina dintr-un *heap Min-Max* cel mai mic element vom urma o procedură asemănătoare cu cea folosită în cazul *heap*-urilor binare.

Practic, vom înlocui primul element al vectorului folosit pentru a păstra *heap-ul Min-Max* cu ultimul element al acestui vector și apoi vom "coborî în arbore" noul prim element.

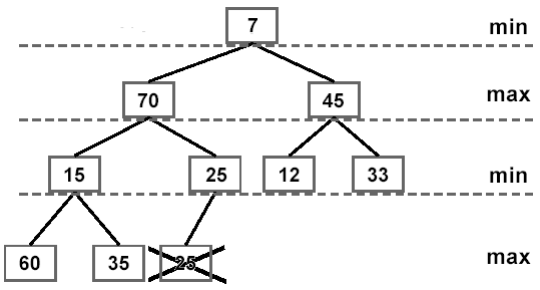

Figura 5: Eliminarea minimului

Așadar, va fi utilizat subalgoritmul `CoboarăMin` descris anterior. Versiunea în pseudocod a algoritmului de eliminare a elementului minim al unui *heap Min-Max* este prezentată în cele ce urmează.

```
algoritm ElimināMinim(a, n)
  dacă n > 0 atunci
    a1 ← an
    n ← n - 1
    CoboarăMin(a, 1, n)
  altfel
    raportează o eroare
  sfârșit dacă
sfârșit algoritm
```

În figura 5 este prezentată structura obținută prin eliminarea elementului minim al *heap-ului Min-Max* din figura 1.

Valoarea `5` (corespunzătoare rădăcinii) este înlocuită de valoarea `25`, iar apoi valorile `25` și `7` sunt interschimbate.

Eliminarea maximului

Pentru a elimina dintr-un *heap Min-Max* cel mai mare element vom folosi un algoritm asemănător.

Mai întâi vom determina poziția elementului maxim și apoi îl vom înlocui cu ultimul element al vectorului folosit pentru a păstra *heap-ul Min-Max*. Ulterior elementul care a înlocuit maximul este "coborât în arbore".

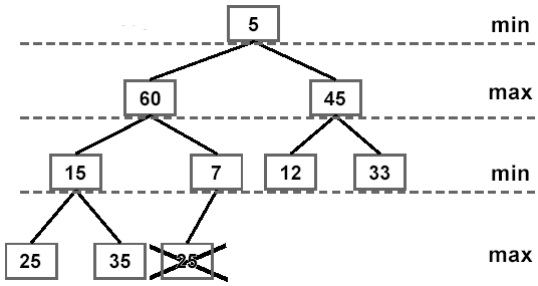

Figura 6: Eliminarea maximului

De data aceasta va fi utilizat subalgoritmul `CoboarăMax`. S-ar putea crede că am avea nevoie și de subalgoritmul `CoboarăMin` în cazul în care *heap-ul Min-Max* ar conține un singur element, caz în care ar trebui să "coborâm" rădăcina. Totuși, în această situație *heap-ul Min-Max* devine vid, deci subalgoritmul nu este neapărat necesar.

```
algoritm ElimināMaxim(a, n)
  dacă n > 0 atunci
    dacă n = 1 atunci
      n ← 0
    altfel
      dacă n = 2 atunci
        n ← 1
      altfel
        dacă a2 > a3 atunci
          a2 ← an
          n ← n - 1
          CoboarăMax(a, 2, n)
        altfel
          a3 ← an
          n ← n - 1
          CoboarăMax(a, 2, n)
        sfârșit dacă
      sfârșit dacă
    sfârșit dacă
  altfel
    raportează o eroare
  sfârșit dacă
sfârșit algoritm
```

În figura 6 este prezentată structura obținută prin eliminarea elementului maxim al *heap-ului Min-Max* din figura 1.

Valoarea `70` (corespunzătoare primului fiu al rădăcinii) este înlocuită de valoarea `25`, iar apoi valorile `25` și `60` sunt interschimbate.

Adăugarea unui element

Pentru a adăuga un nou element într-un *heap Min-Max* este suficient să îl adăugăm la sfârșitul vectorului și să efectuăm interschimbările necesare pentru ca proprietatea de *heap* să fie respectată.

Un algoritm eficient pentru adăugarea unui element poate fi dedus studiind diagrama *Hasse* a unui *heap Min-Max* (vezi figura 2).

Mai întâi observăm că elementul adăugat devine o frunză a arborelui care reprezintă *heap-ul Min-Max* și că în diagrama *Hasse* frunzele se află pe linia din mijlocul diagramei (sau în apropierea acesteia dacă arborele nu este complet).

În momentul adăugării unui element acesta va trebui să "urce în diagramă" sau să "coboare în diagramă" până în momentul în care ordinea va fi restabilită.

Mai întâi vom stabili dacă elementul va "urca" sau va "coborî" în diagramă. "Urcarea în diagramă" corespunde unei "urcări în arbore" pe nivelurile de minim iar "coborârea în diagramă" corespunde unei "urcări în arbore" pe nivelurile de maxim.

După ce am stabilit dacă vom "urca în arbore" pe nivelurile de minim sau pe nivelurile de maxim, este suficient să realizăm comparări între elementul curent și părintele părintelui său.

La început vom determina dacă frunza se află pe un nivel de minim sau pe un nivel de maxim. Apoi vom verifica dacă se respectă proprietatea de *heap* pentru frunză și părintele ei. În cazul în care nu este respectată proprietatea vom realiza o interschimbare. Apoi vom urca în arbore fie pe nivelurile de maxim, fie pe cele de minim.

Prezentăm în continuare versiunea în pseudocod a algoritmului și a subalgoritmilor utilizați în cadrul acestuia.

```
algoritm Adaugă(a, x, n)
  n ← n + 1
  a_n ← x
  // verificarea tipului nivelului
  dacă [log2 i] este număr par
                                atunci
  // verificarea existenței părintelui
    dacă n > 1 atunci
    // verificarea respectării ordinii
    // pentru noul element și
    // părintele acestuia
      dacă a_n > a_[n/2] atunci
        interschimbă a_n și a_[n/2]
        UrcăMax(a, [n / 2], n)
      altfel
        UrcăMin(a, n, n)
      sfârșit dacă
    sfârșit dacă
  altfel
  // verificarea existenței părintelui
    dacă n > 1 atunci
    // verificarea respectării ordinii
    // pentru noul element și
    // părintele acestuia
      dacă a_n < a_[n/2] atunci
        interschimbă a_n și a_[n/2]
        UrcăMin(a, [n / 2], n)
      altfel
        UrcăMax(a, n, n)
      sfârșit dacă
    sfârșit dacă
  sfârșit dacă
sfârșit algoritm
```

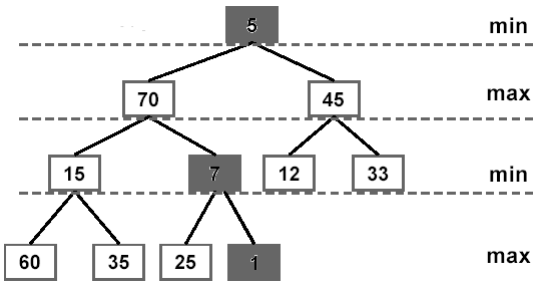

Figura 7: Adăugarea unui element

```
subalgoritm UrcăMin(a, i, n)
// verificarea existenței părintelui
// părintelui
  dacă i > 4 atunci
  // verificarea respectării ordinii
  // pentru elementul curent și
  // părintele părintelui acestuia
    dacă a_i < a_[i/4] atunci
      interschimbă a_i și a_[i/4]
      UrcăMin(a, [i / 4], i)
    sfârșit dacă
  sfârșit dacă
sfârșit subalgoritm

subalgoritm UrcăMax(a, i, n)
// verificarea existenței părintelui
// părintelui
  dacă i > 4 atunci
  // verificarea respectării ordinii
  // pentru elementul curent și
  // părintele părintelui acestuia
    dacă a_i > a_[i/4] atunci
      interschimbă a_i și a_[i/4]
      UrcăMax(a, [i / 4], i)
    sfârșit dacă
  sfârșit dacă
sfârșit subalgoritm
```

Procesul de adăugare a unui nou element este ilustrat în figura 7. Figura prezintă traseul elementului cu valoarea `1` care este adăugat în *heap-ul Min-Max*.

Analiza complexității

Vom compara în cele ce urmează ordinele de complexitate ale diferitelor operații pentru *heap-urile Min-Max* și cele binare.

Trebuie să menționăm faptul că ordinele de complexitate sunt aceleași, cu două excepții: determinarea și eliminarea maximului.

Pentru aceste operații, în cazul *heap-urilor Min-Max* avem ordinele de complexitate $O(1)$, respectiv $O(\log n)$, în timp ce în cazul *heap-urilor* binare ordinul de complexitate al acestor operații este $O(n)$ deoarece implică traversarea întregului *heap*.

Pentru transformarea unui vector într-un *heap* binar sunt necesare $2 \cdot n$ operații elementare în timp ce pentru *heap-urile Min-Max* numărul acestora este de aproximativ $7 \cdot n / 3$.

Pentru eliminarea elementului minim în cazul *heap*-urilor binare sunt necesare aproximativ $2 \cdot \log_2 n$ operații elementare, iar pentru *heap-urile Min-Max* numărul acestora este de aproximativ $2.5 \cdot \log_2 n$.

Pentru eliminarea elementului maxim în cazul *heap*-urilor binare sunt necesare aproximativ $0.5 \cdot n + \log_2 n$ operații elementare, iar pentru *heap-urile Min-Max* numărul acestora este, la fel ca și în cazul eliminării minimului, de aproximativ $2.5 \cdot \log_2 n$.

Pentru inserarea unui element într-un *heap* binar sunt necesare aproximativ $\log_2 (n + 1)$ operații elementare, iar pentru *heap-urile Min-Max* numărul acestora este de aproximativ $0.5 \cdot \log_2 (n + 1)$.