Structuri de Date și Algoritmi
Heap binar

Mihai Nan

Departamentul de Calculatoare
Facultatea de Automatică și Calculatoare
Universitatea POLITEHNICA din București

Anul Universitar 2022–2023

Conținutul cursului

Coadă de prioritate

- Coada de prioritate reprezintă o generalizare a structurii de date coadă în care fiecare element are asociată o prioritate, iar elementele sunt extrase din coadă în ordinea priorității.
- Putem să avem două tipuri:
  1. primul element extras din coada este elementul cu **cea mai mare prioritate** (max-heap);
  2. primul element extras din coada este elementul cu **cea mai mică prioritate** (min-heap);

Observație

În practică există foarte multe aplicații care necesită o astfel de structură.

Coadă de prioritate

- Această structură de date este implementată deja în multe limbaje:
  1. În **C++** avem `priority_queue`
  2. În **Java** avem `PriorityQueue`
  3. În **Python** avem `heapq`
- Există mulți agoritmi care necesită utilizarea unei astfel de structuri.
  1. Algoritmul lui Dijkstra
  2. Algoritmul lui Prim
  3. Algoritmul de compresie Huffman
  4. Algoritmul de sortare HeapSort

Tehnica Greedy

- Strategia Greedy este aplicată, în general, în cazul problemelor de optimizare pentru care atunci când dorim să construim soluția trebuie să determinăm o secvență de acțiuni selectate din mai multe variante posibile.
- Algoritmii bazați pe această tehnică de programare aleg la fiecare pas optimul local în speranța că dacă vom realiza o soluție compusă din optimele locale determinate pentru fiecare pas vom ajunge la o soluție care să îndeplinească criteriul de optim global pentru problema pe care dorim să o rezolvăm.

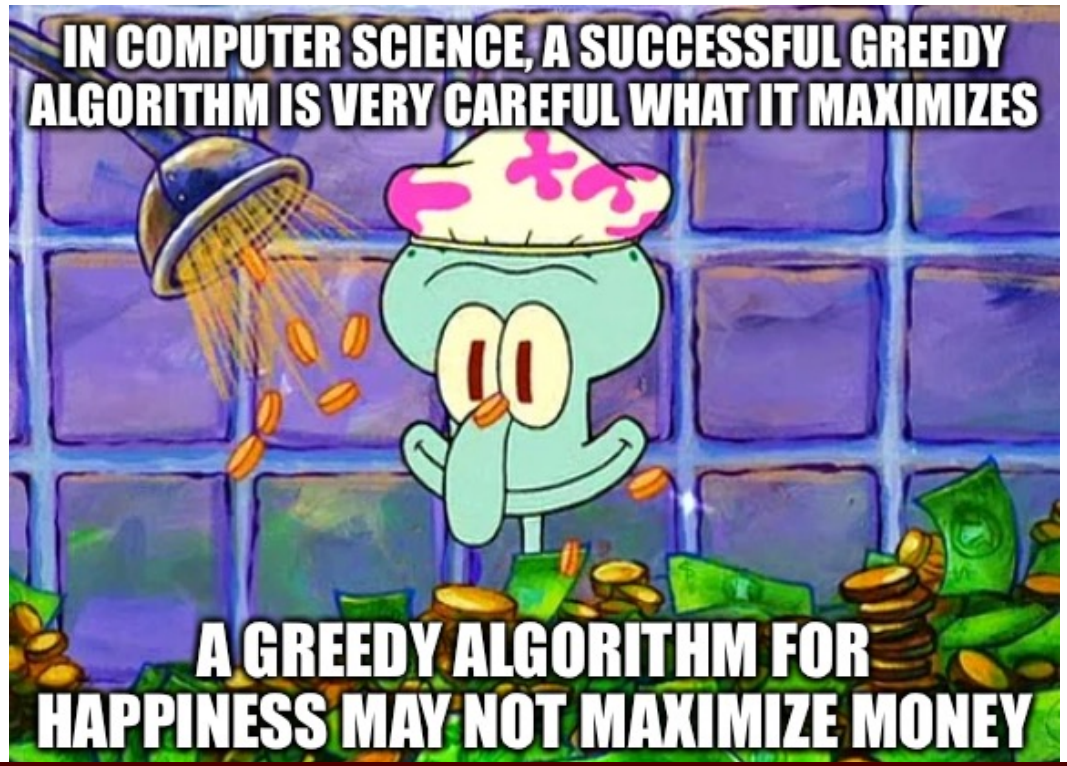
Tehnica Greedy

- Strategia Greedy este aplicată, în general, în cazul problemelor de optimizare pentru care atunci când dorim să construim soluția trebuie să determinăm o secvență de acțiuni selectate din mai multe variante posibile.
- Algoritmii bazați pe această tehnică de programare aleg la fiecare pas optimul local în speranța că dacă vom realiza o soluție compusă din optimele locale determinate pentru fiecare pas vom ajunge la o soluție care să îndeplinească criteriul de optim global pentru problema pe care dorim să o rezolvăm.

Care poate să fie legătura între această tehnică de programare și coada de priorități?

Tehnica Greedy

- Dacă pornim de la pseudocodul care rezolvă o problemă generală prin tehnica Greedy, distingem două variante:
  1. algoritm fără prelucrare inițială a mulțimii A – la fiecare pas se determină care element îndeplinește calitatea de optim local;
  2. algoritm cu prelucrare inițială a mulțimii A – prelucrare care, de fapt, stabilește de la început ordinea în care sunt extrase elementele din A la fiecare pas.



Tehnica Greedy

- Din punct de vedere formal, metoda Greedy se aplică în cazul problemelor pentru care primim o mulțime finită A și dorim să determinăm o submulțime finită S a lui A ($S \subseteq A$) care îndeplinește constrângerile impuse de problemă (reprezintă o soluție validă a problemei) și satisface un criteriu de optim.
- Soluțiile posibile pentru o astfel de problemă se pot construi în mod recursiv pornind de la următoarele două reguli:
  1. $\varnothing$ reprezintă o soluție posibilă a problemei (nu contrazice nicio contrângere impusă de problemă);
  2. Dacă S reprezintă o soluție posibilă a problemei și $S^{'}$ este o submulțime finită a lui S ($S^{'} \subseteq S$), atunci $S^{'}$ este considerată o soluție posibilă a problemei.

Tehnica Greedy – fără prelucrare inițială

```
1:  procedure GREEDY_FĂRĂ_PRELUCRARE_INIȚIALĂ(A)
2:      S ← ∅
3:      n ← |A|
4:      for i = 1 to n do
5:          x ← ALEGE(A)
6:          A ← A \ {x}
7:          if p(S ∪ {x}) = 1 then
8:              S ← S ∪ {x}
9:          end if
10:     end for
11:     return S
12: end procedure
```

Tehnica Greedy

Cum putem alege eficient elementul care îndeplinește criteriul de optim din mulțimea A?

Tehnica Greedy

Cum putem alege eficient elementul care îndeplinește criteriul de optim din mulțimea A?

Ne folosim de o structură de tip coadă de prioritate!

Tehnica Greedy

Cum putem alege eficient elementul care îndeplinește criteriul de optim din mulțimea A?

Ne folosim de o structură de tip coadă de prioritate!

Cum vom utiliza o astfel de structură?

Tehnica Greedy

Cum putem alege eficient elementul care îndeplinește criteriul de optim din mulțimea A?

Ne folosim de o structură de tip coadă de prioritate!

Cum vom utiliza o astfel de structură?

Inițial, inserăm toate elementele din A în coada de prioritate și apoi le vom extrage unul câte unul, în funcție de prioritate.

Coadă de prioritate

Ce modalități de reprezentare am putea folosi pentru coada de prioritate?

Coadă de prioritate

Ce modalități de reprezentare am putea folosi pentru coada de prioritate?

Liste sau vectori

Coadă de prioritate

Ce modalități de reprezentare am putea folosi pentru coada de prioritate?

Liste sau vectori

Considerăm aceste colecții în varianta sortată sau nu?

Coadă de prioritate

Ce modalități de reprezentare am putea folosi pentru coada de prioritate?

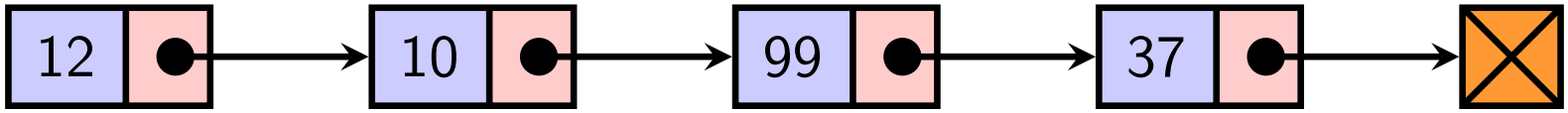
Liste sau vectori

Considerăm aceste colecții în varianta sortată sau nu?

Depinde...

Coadă de prioritate – Listă nesortată

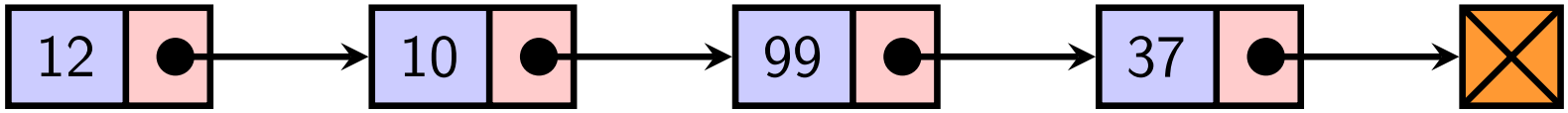
Exemplu



1. Operația de inserare

Coadă de prioritate – Listă nesortată

Exemplu

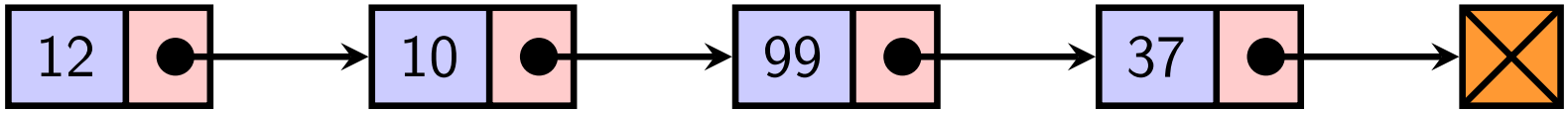


1. Operația de inserare

Adaugă elementul la finalul listei.

Coadă de prioritate – Listă nesortată

Exemplu



1. Operația de inserare

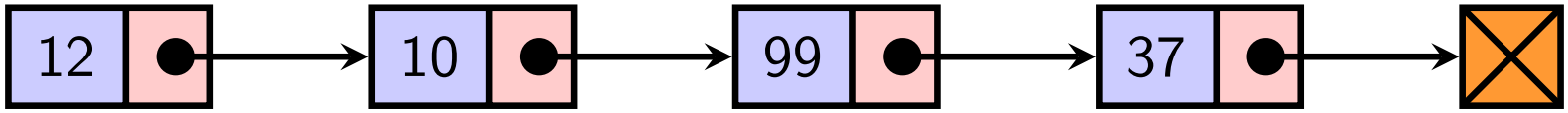
Adaugă elementul la finalul listei.

Complexitate: $O(1)$

2. Operația de extragere a maximului

Coadă de prioritate – Listă nesortată

Exemplu



1. Operația de inserare

Adaugă elementul la finalul listei.

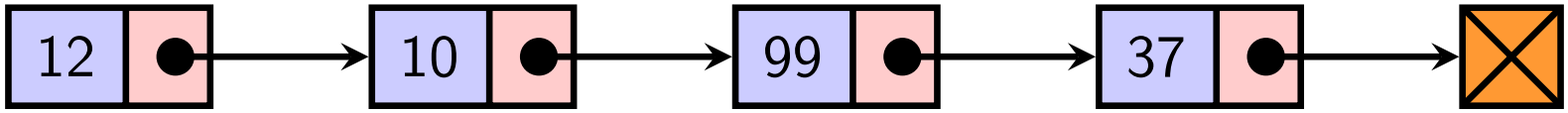
Complexitate: $O(1)$

2. Operația de extragere a maximului

Parcurge lista pentru a găsi maximul.

Coadă de prioritate – Listă nesortată

Exemplu



1. Operația de inserare


Adaugă elementul la finalul listei.

Complexitate: **O(1)**

2. Operația de extragere a maximului


Parcurge lista pentru a găsi maximul.

Complexitate: **O(N)**

Coadă de prioritate – Vector nesortat

Exemplu

1. Operația de inserare

Coadă de prioritate – Vector nesortat

Exemplu

1. Operația de inserare

Adaugă elementul la finalul vectorului.

Coadă de prioritate – Vector nesortat

Exemplu

1. Operația de inserare

Adaugă elementul la finalul vectorului.

Complexitate: $O(1)$

2. Operația de extragere a maximului

Coadă de prioritate – Vector nesortat

Exemplu

1. Operația de inserare

Adaugă elementul la finalul vectorului.

Complexitate: $O(1)$

2. Operația de extragere a maximului

Parcurge vectorul pentru a găsi maximul.

Coadă de prioritate – Vector nesortat

Exemplu

1. Operația de inserare

Adaugă elementul la finalul vectorului.

Complexitate: **O(1)**

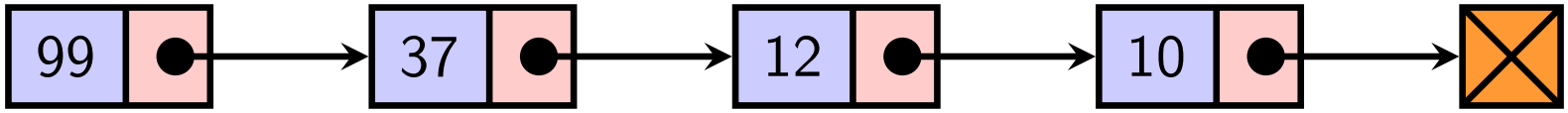
2. Operația de extragere a maximului

Parcurge vectorul pentru a găsi maximul.

Complexitate: **O(N)**

Coadă de prioritate – Listă sortată

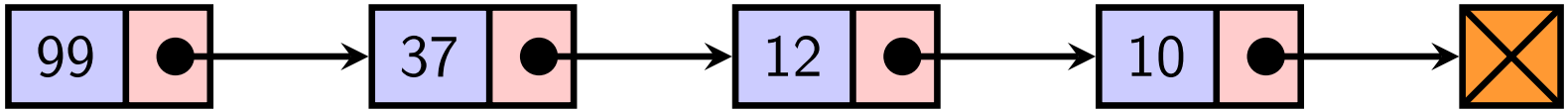
Exemplu



1. Operația de inserare

Coadă de prioritate – Listă sortată

Exemplu

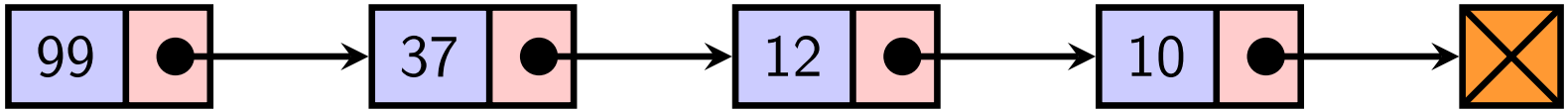


1. Operația de inserare

Găsește poziția pentru element și apoi inserează.

Coadă de prioritate – Listă sortată

Exemplu



1. Operația de inserare

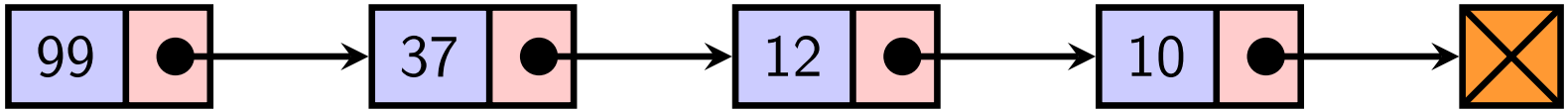
Găsește poziția pentru element și apoi inserează.

Complexitate: **O(N)**

2. Operația de extragere a maximului

Coadă de prioritate – Listă sortată

Exemplu



1. Operația de inserare

Găsește poziția pentru element și apoi inserează.

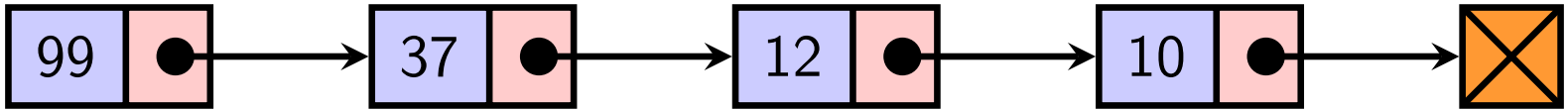
Complexitate: O(N)

2. Operația de extragere a maximului

Extrage primul element din listă.

Coadă de prioritate – Listă sortată

Exemplu



1. Operația de inserare

Găsește poziția pentru element și apoi inserează.

Complexitate: **O(N)**

2. Operația de extragere a maximului

Extrage primul element din listă.

Complexitate: **O(1)**

Coadă de prioritate – Vector sortat

Exemplu

1. Operația de inserare

Coadă de prioritate – Vector sortat

Exemplu

1. Operația de inserare

Aplică algoritmul de căutare binară pentru a determina poziția, deplasează la dreapta cu o poziție elementele aflate după poziția găsită și inserează.

Coadă de prioritate – Vector sortat

Exemplu

1. Operația de inserare

Aplică algoritmul de căutare binară pentru a determina poziția, deplasează la dreapta cu o poziție elementele aflate după poziția găsită și inserează.

Complexitate: $\mathbf{O(\log N) + O(N) = O(N)}$

2. Operația de extragere a maximului

Coadă de prioritate – Vector sortat

Exemplu

1. Operația de inserare

Aplică algoritmul de căutare binară pentru a determina poziția, deplasează la dreapta cu o poziție elementele aflate după poziția găsită și inserează.

Complexitate: $\mathbf{O(\log N) + O(N) = O(N)}$

2. Operația de extragere a maximului

Extrage ultimul element (sortat crescător).

Coadă de prioritate – Vector sortat

Exemplu

1. Operația de inserare

Aplică algoritmul de căutare binară pentru a determina poziția, deplasează la dreapta cu o poziție elementele aflate după poziția găsită și inserează.

Complexitate: $\mathbf{O(\log N) + O(N) = O(N)}$

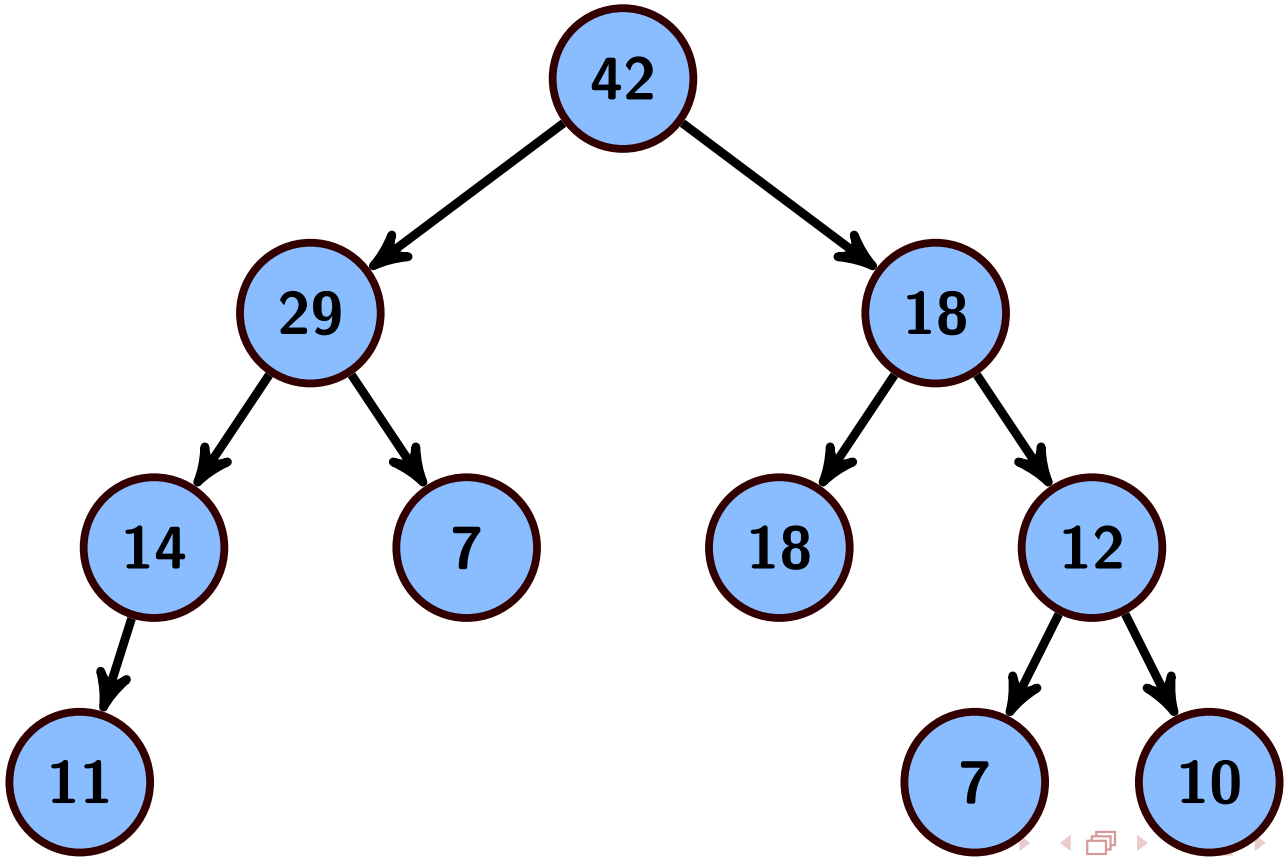
2. Operația de extragere a maximului

Extrage ultimul element (sortat crescător).

Complexitate: $\mathbf{O(1)}$

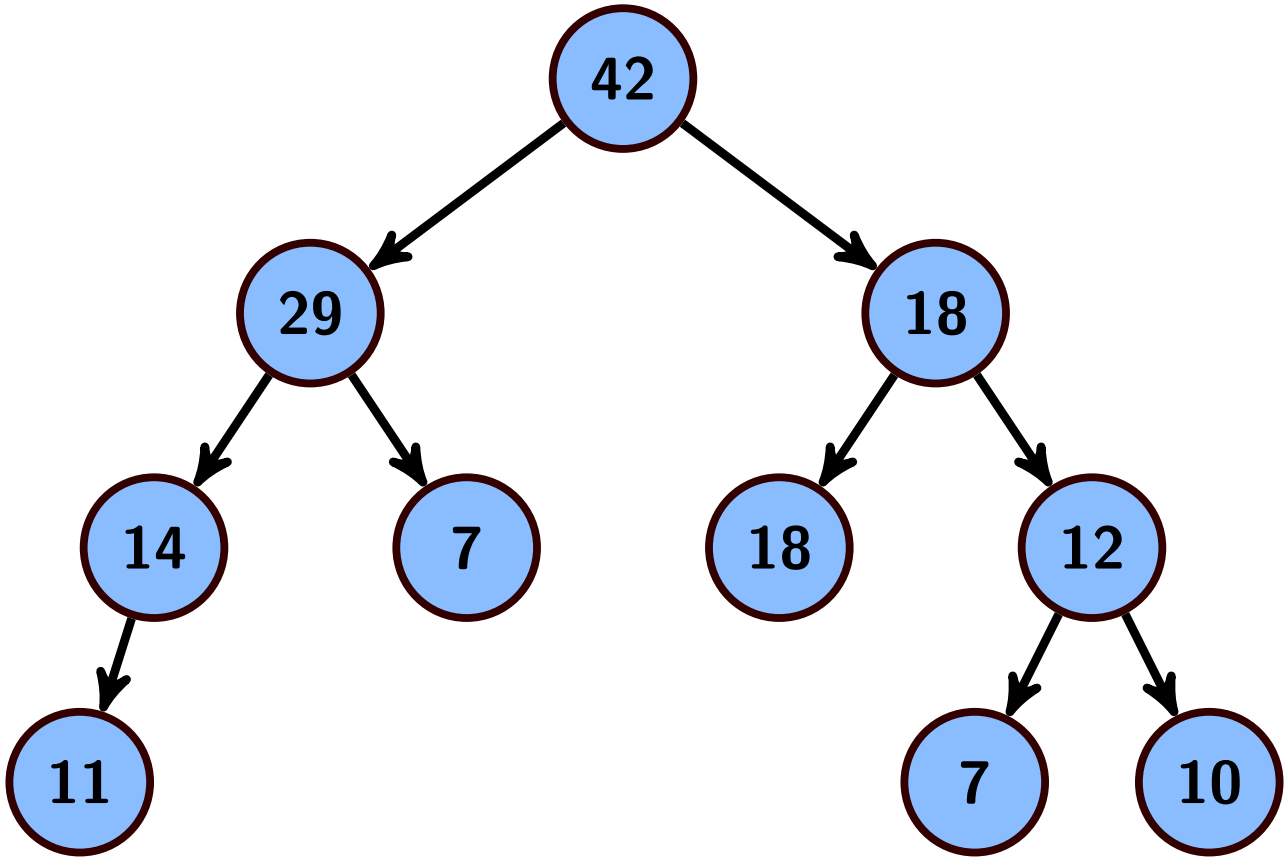
Heap

- Heap-ul este o structură de date care permite implementarea eficientă a operațiilor cu cozi de prioritate.
- Un heap-max binar este un arbore binar cu proprietatea:
  - **pentru orice nod, cheia nodului este mai mare decât cheile din nodurile copii, dacă există copii.**



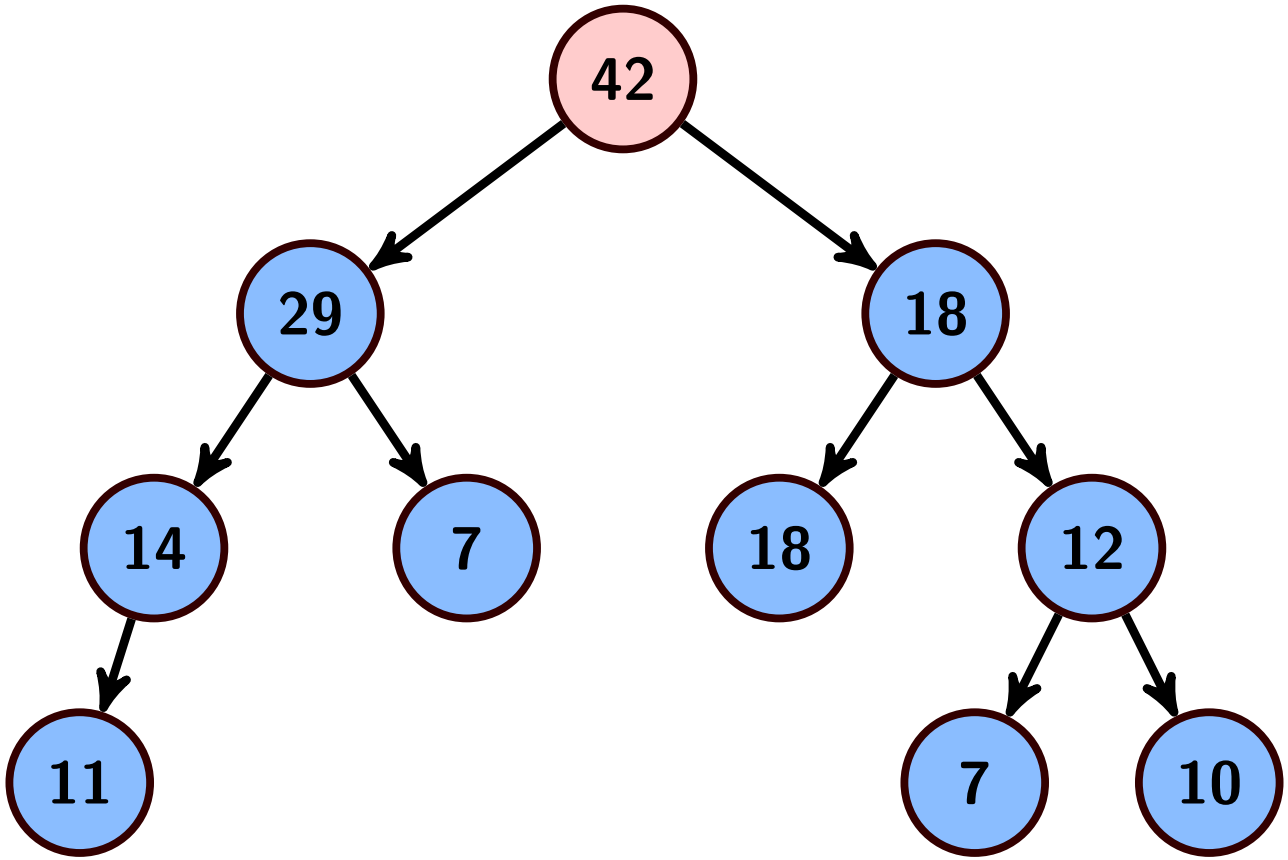
Heap – Determinarea elementului maxim

Cum putem determina elementul maxim?



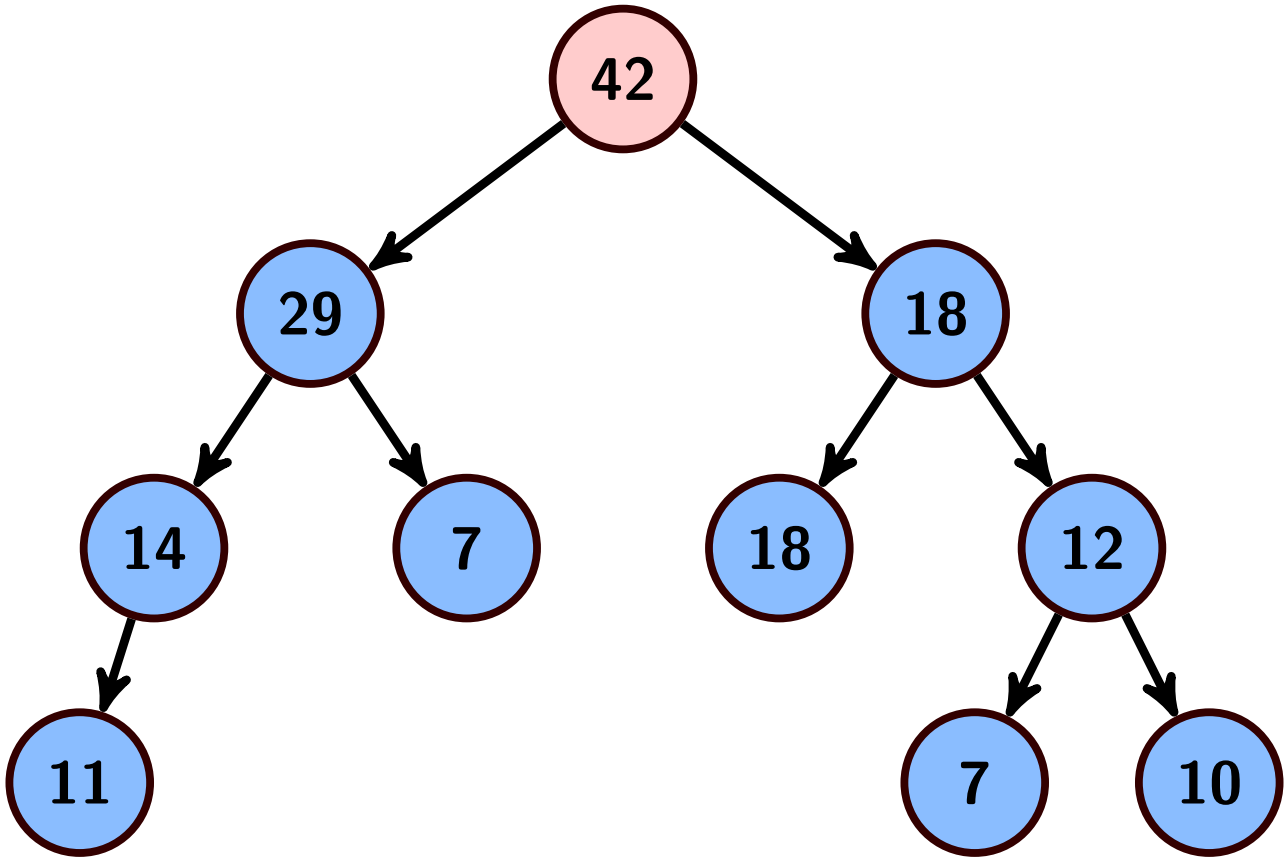
Heap – Determinarea elementului maxim

Rădăcina va conține elementul maxim!



Heap – Determinarea elementului maxim

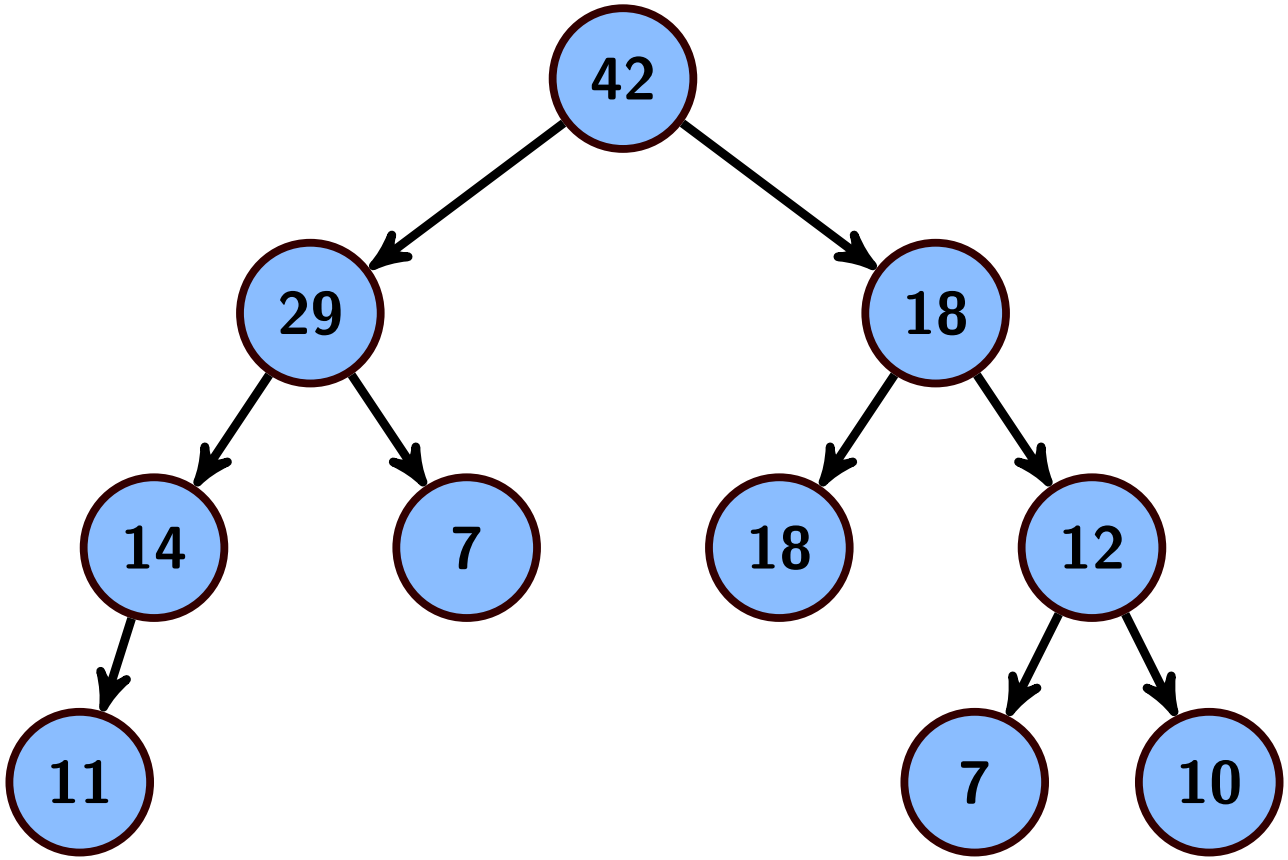
Rădăcina va conține elementul maxim!



Complexitate: $O(1)$

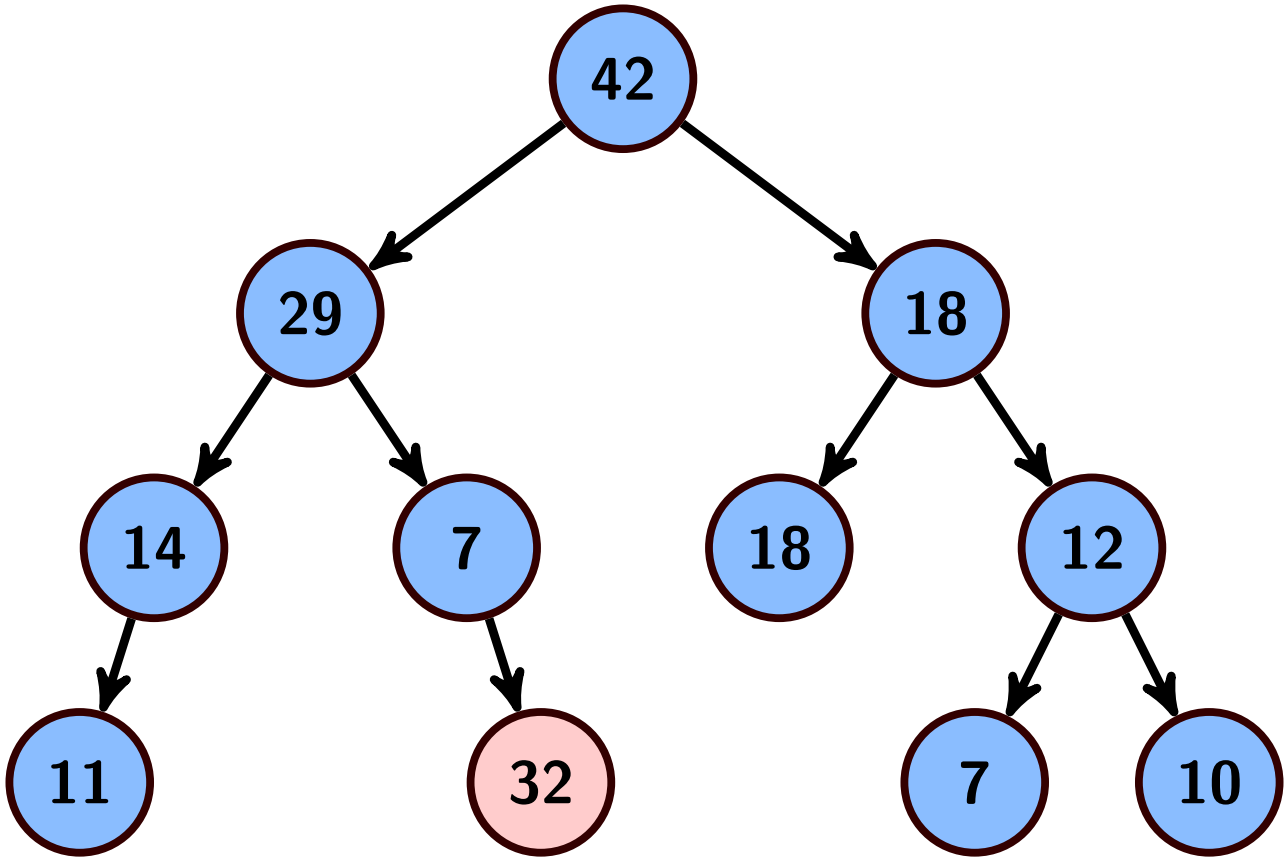
Heap – Inserarea unui element

Cum putem insera un element într-un heap?



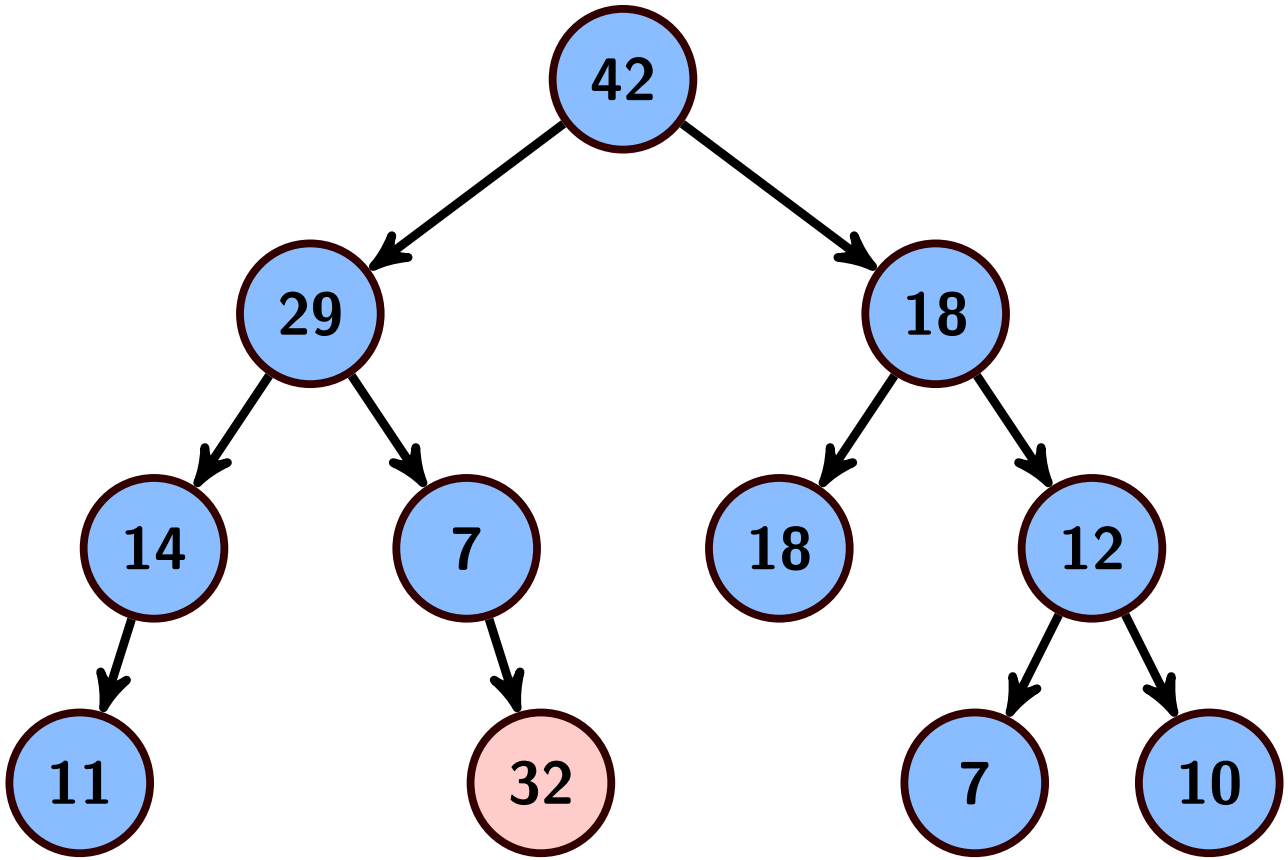
Heap – Inserarea unui element

Adăugăm un copil unui nod frunză!



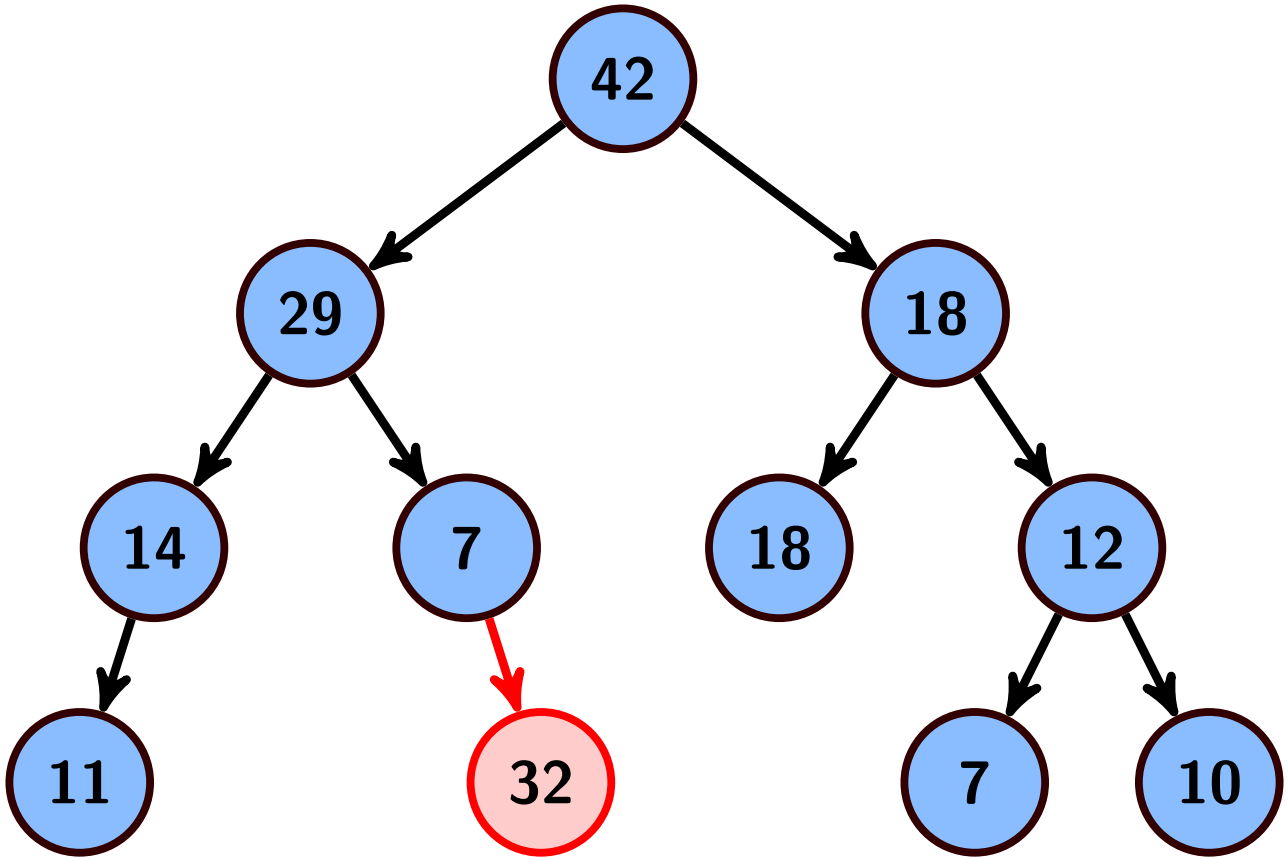
Heap – Inserarea unui element

Arborele obținut poate fi considerat un heap?



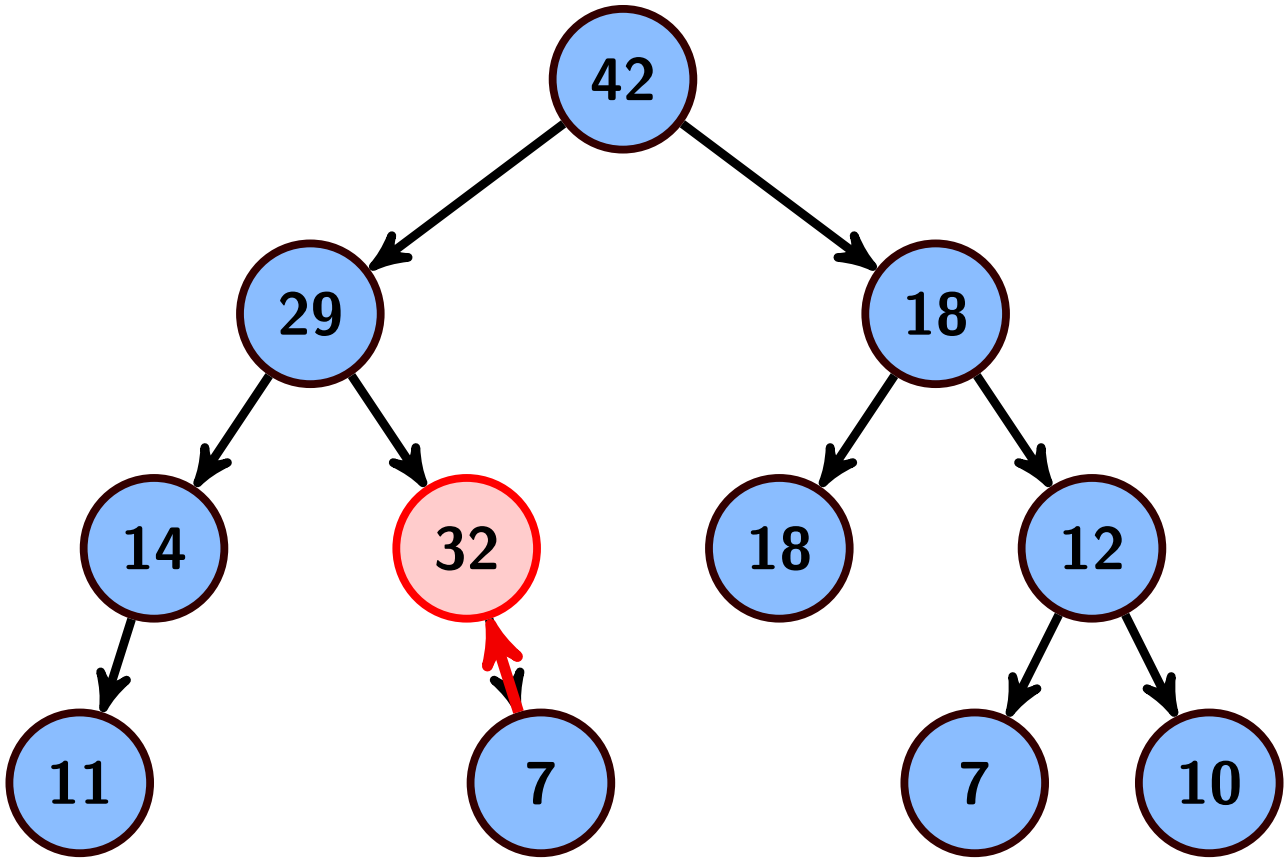
Heap – Inserarea unui element

Este posibil să fie încălcată proprietatea de heap!



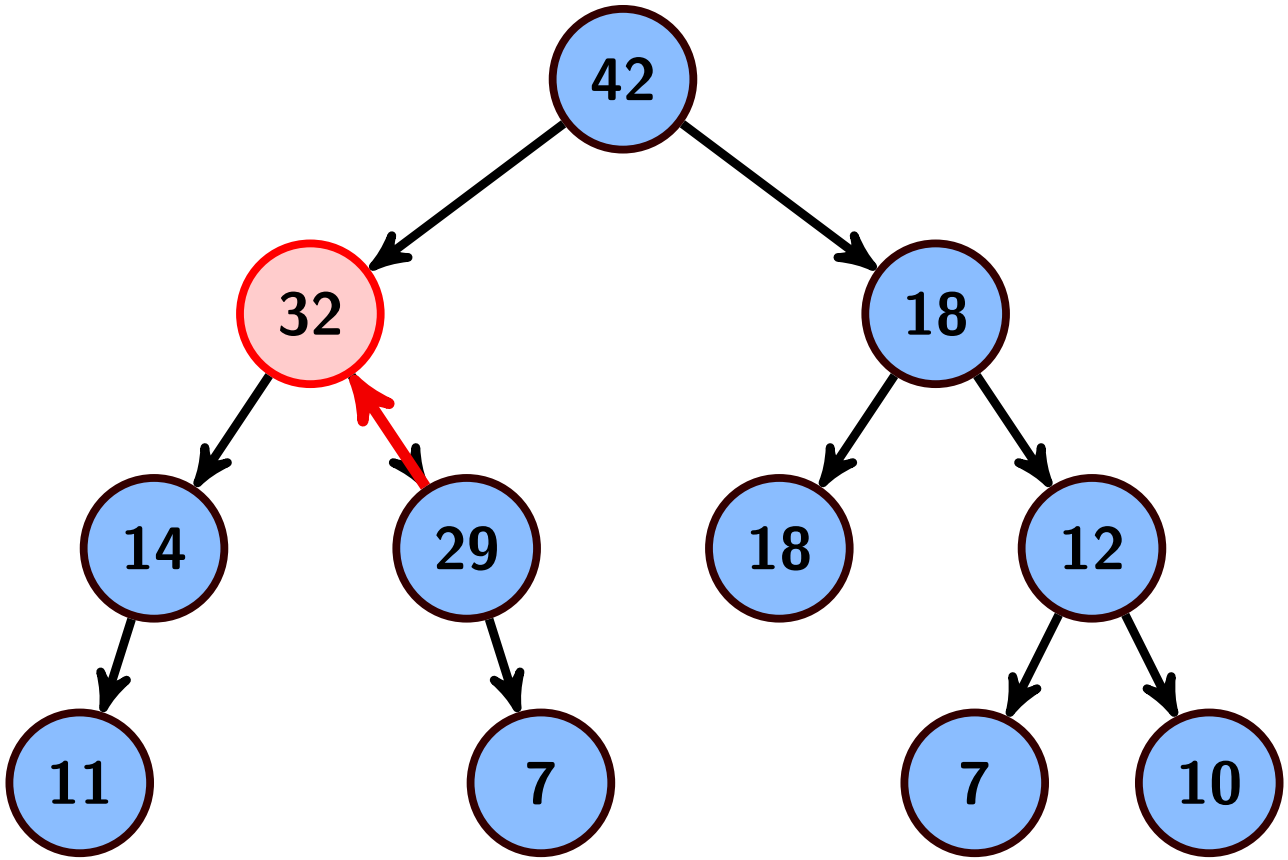
Heap – Inserarea unui element

Aplicăm o operație de *cernere* în sus!



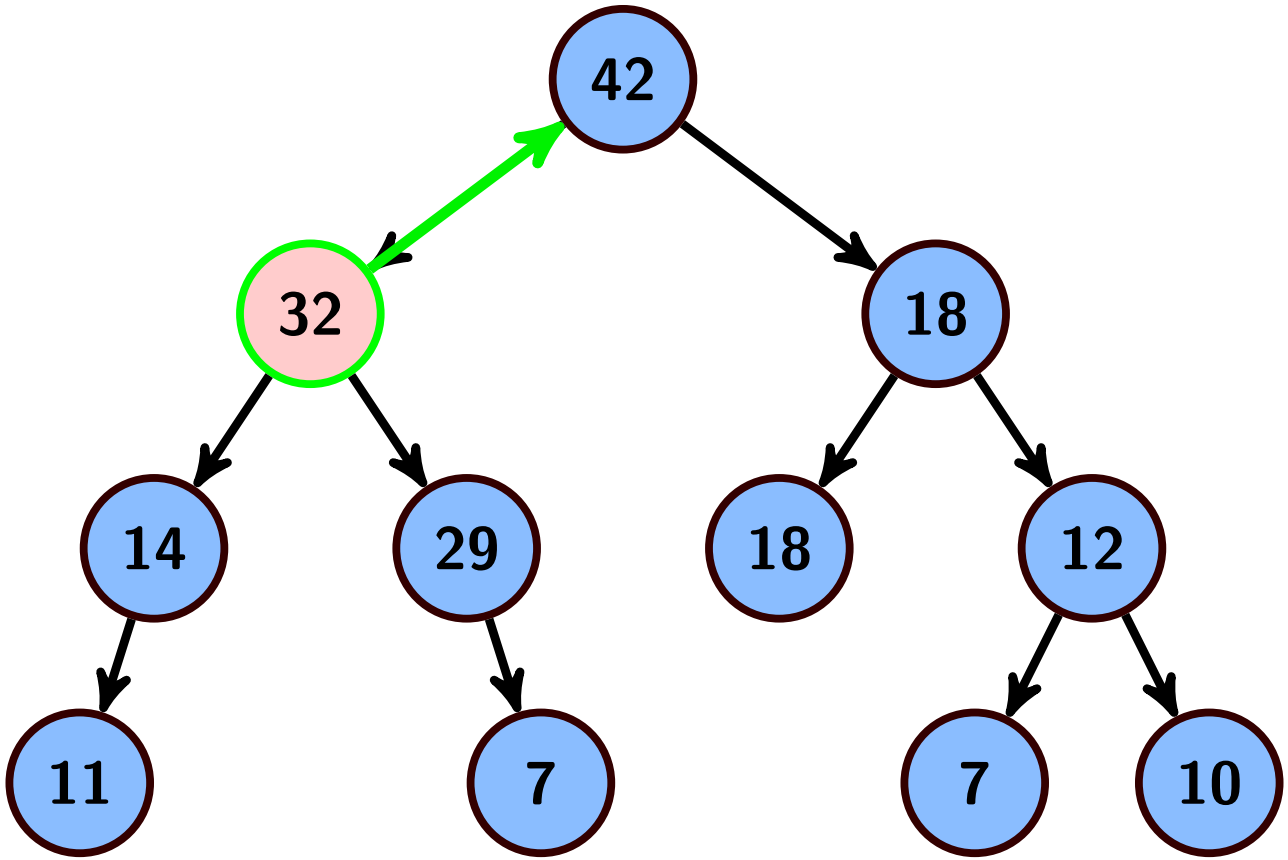
Heap – Inserarea unui element

Aplicăm o operație de *cernere* în sus!



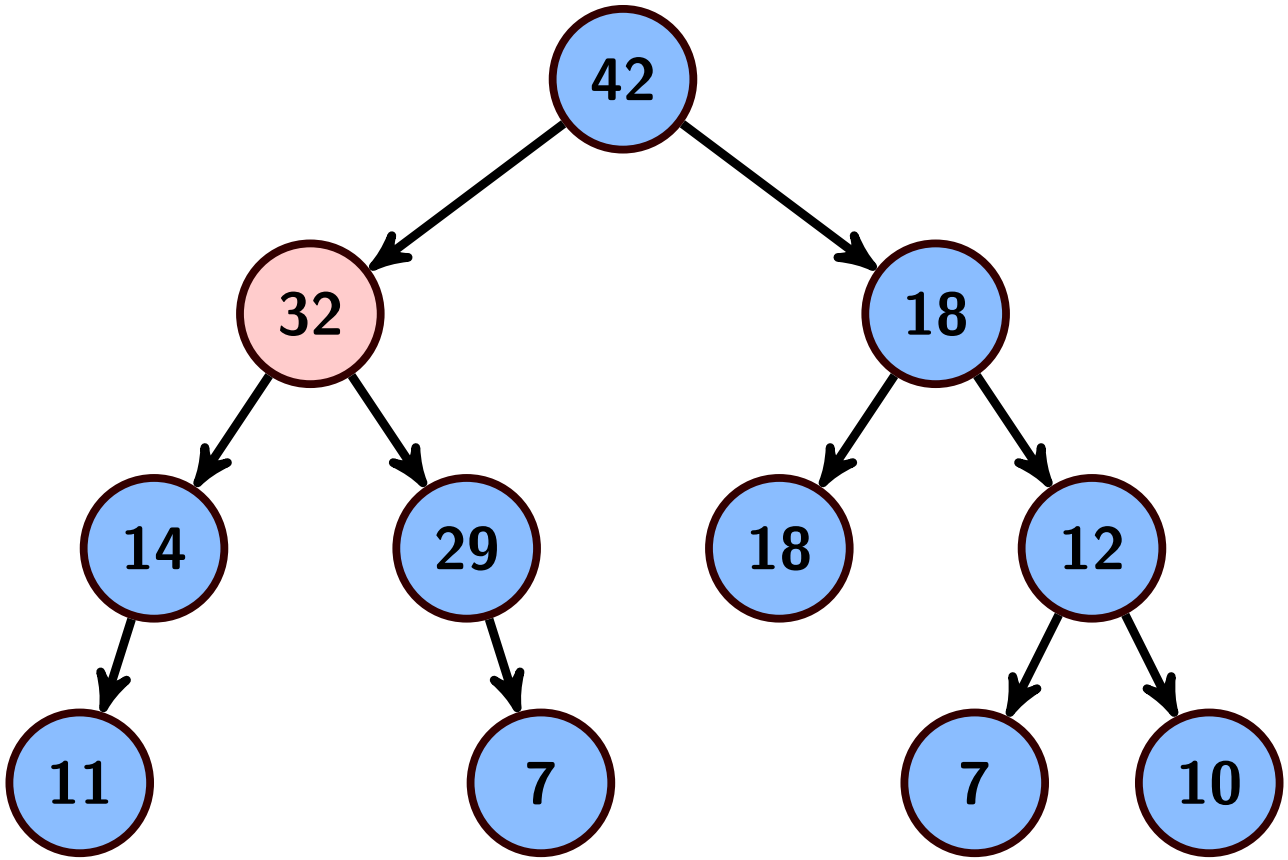
Heap – Inserarea unui element

Aplicăm o operație de *cernere* în sus!



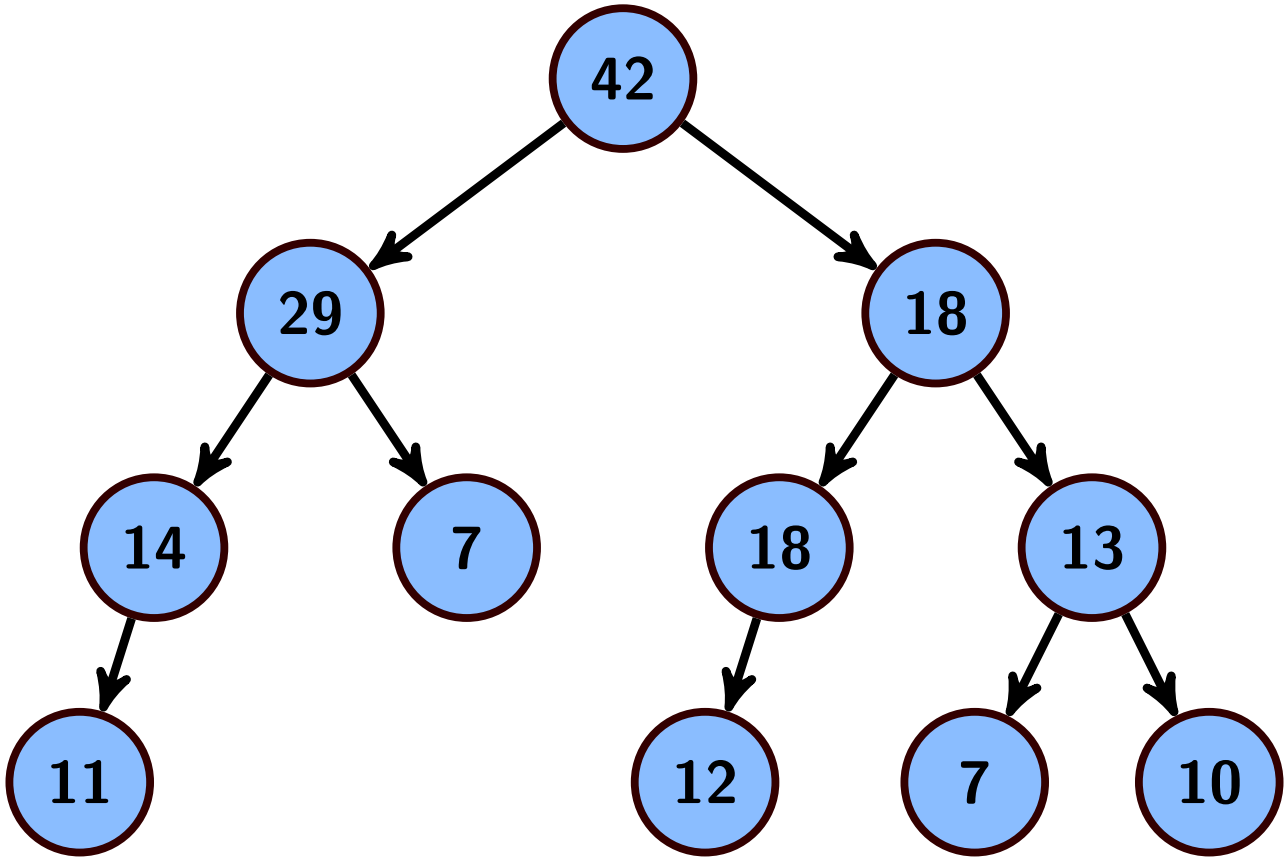
Heap – Inserarea unui element

Aplicăm o operație de *cernere* în sus!



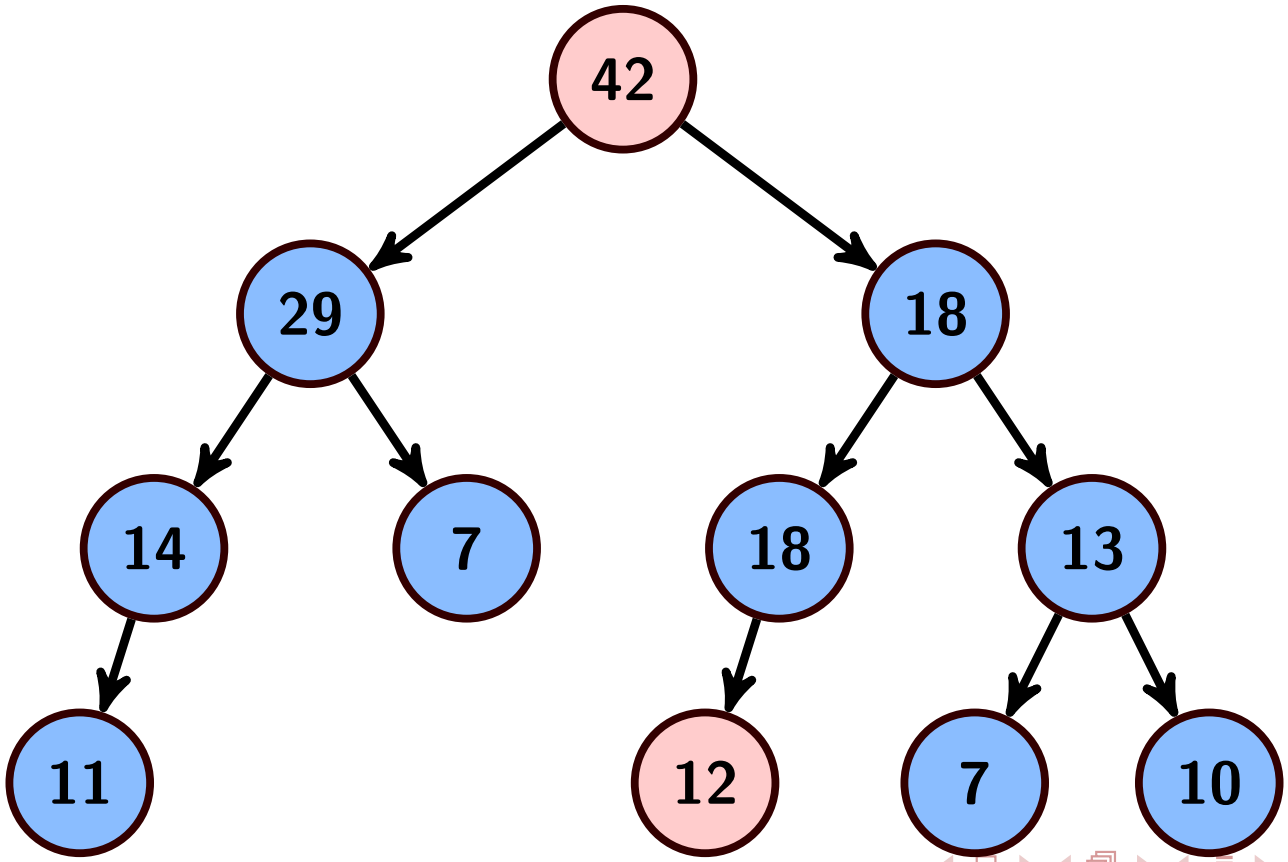
Heap – Eliminarea maximului

Cum putem elimina maximul?



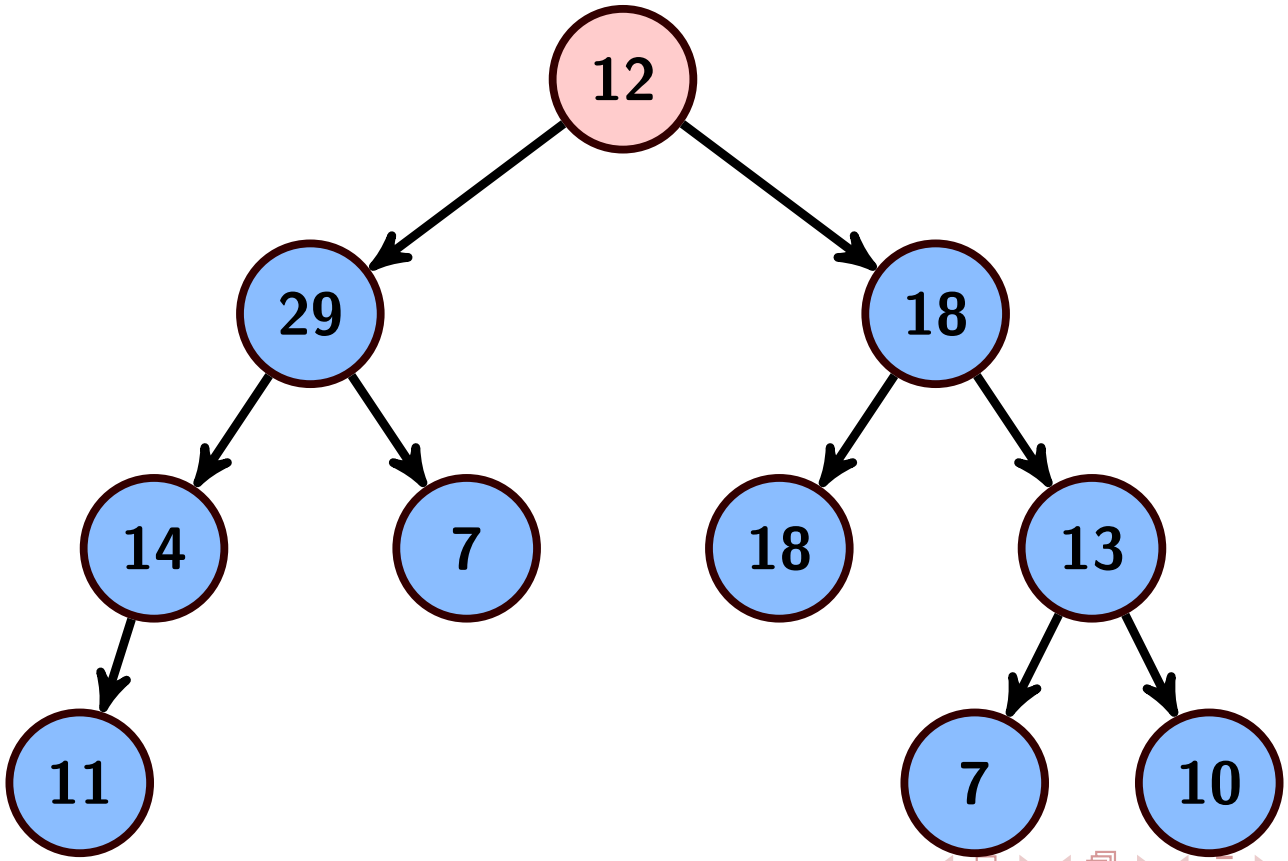
Heap – Eliminarea maximului

Elementul maxim se va afla în rădăcina arborelui și putem pune în loc de rădăcină valoarea unei frunze.



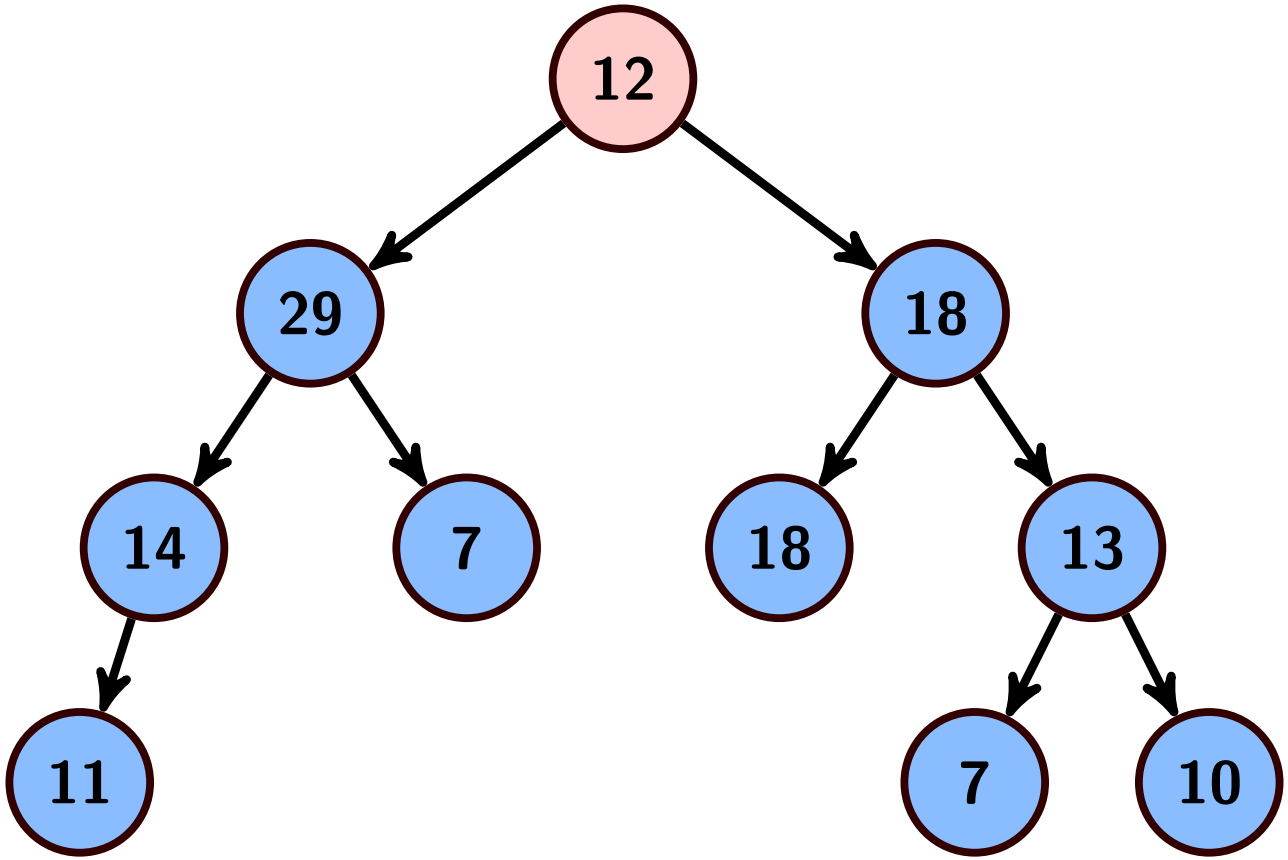
Heap – Eliminarea maximului

Elementul maxim se va afla în rădăcina arborelui și putem pune în loc de rădăcină valoarea unei frunze.



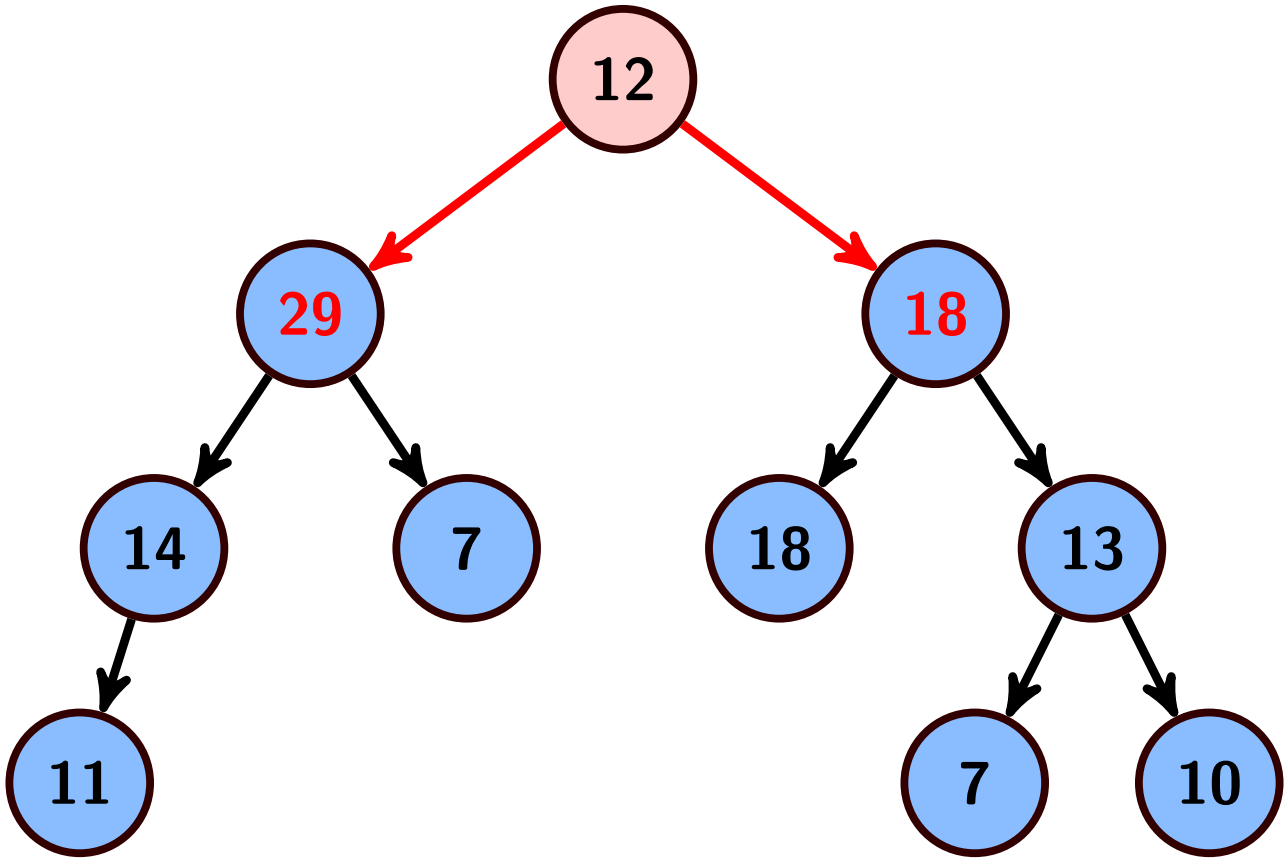
Heap – Eliminarea maximului

Trebuie să verific dacă arborele este un heap.



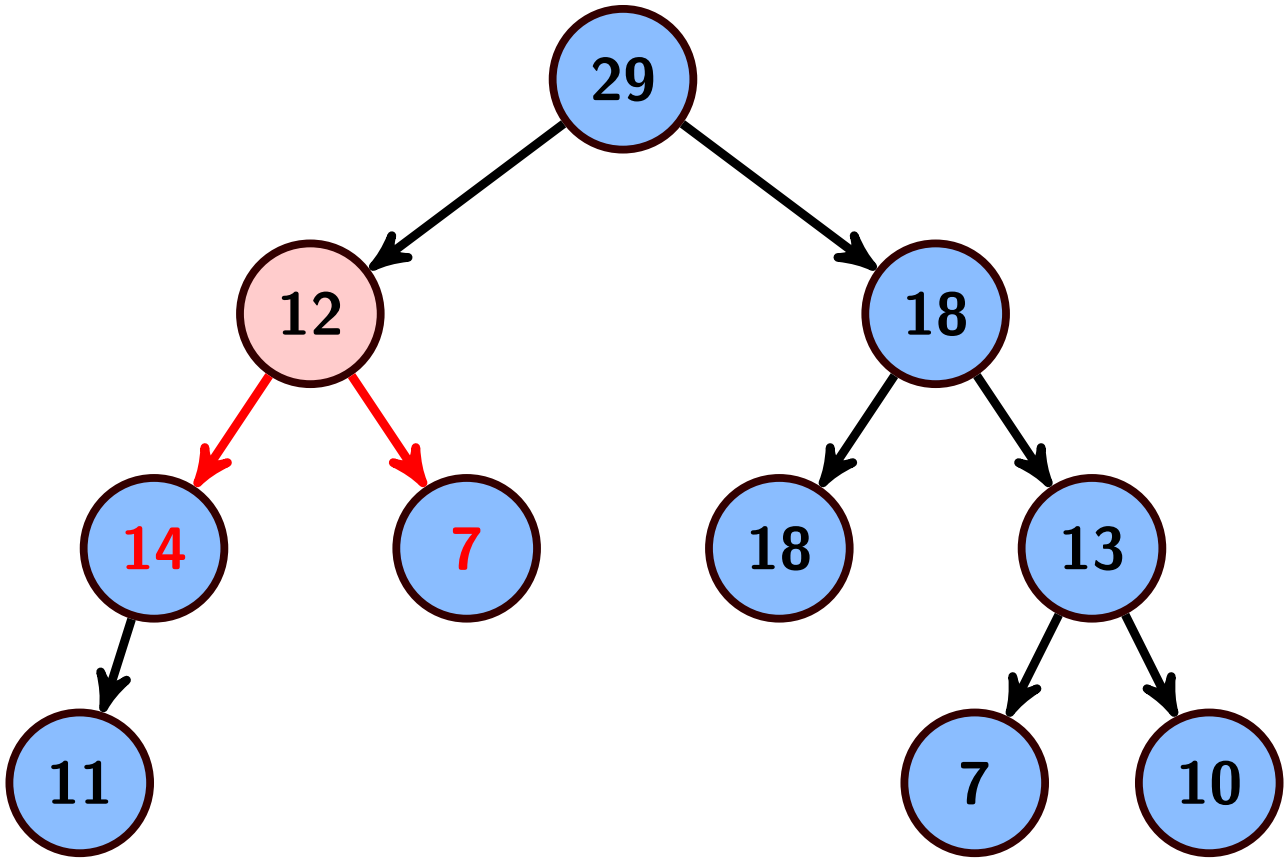
Heap – Eliminarea maximului

Va trebui să aplic operația de *cernere* în jos!



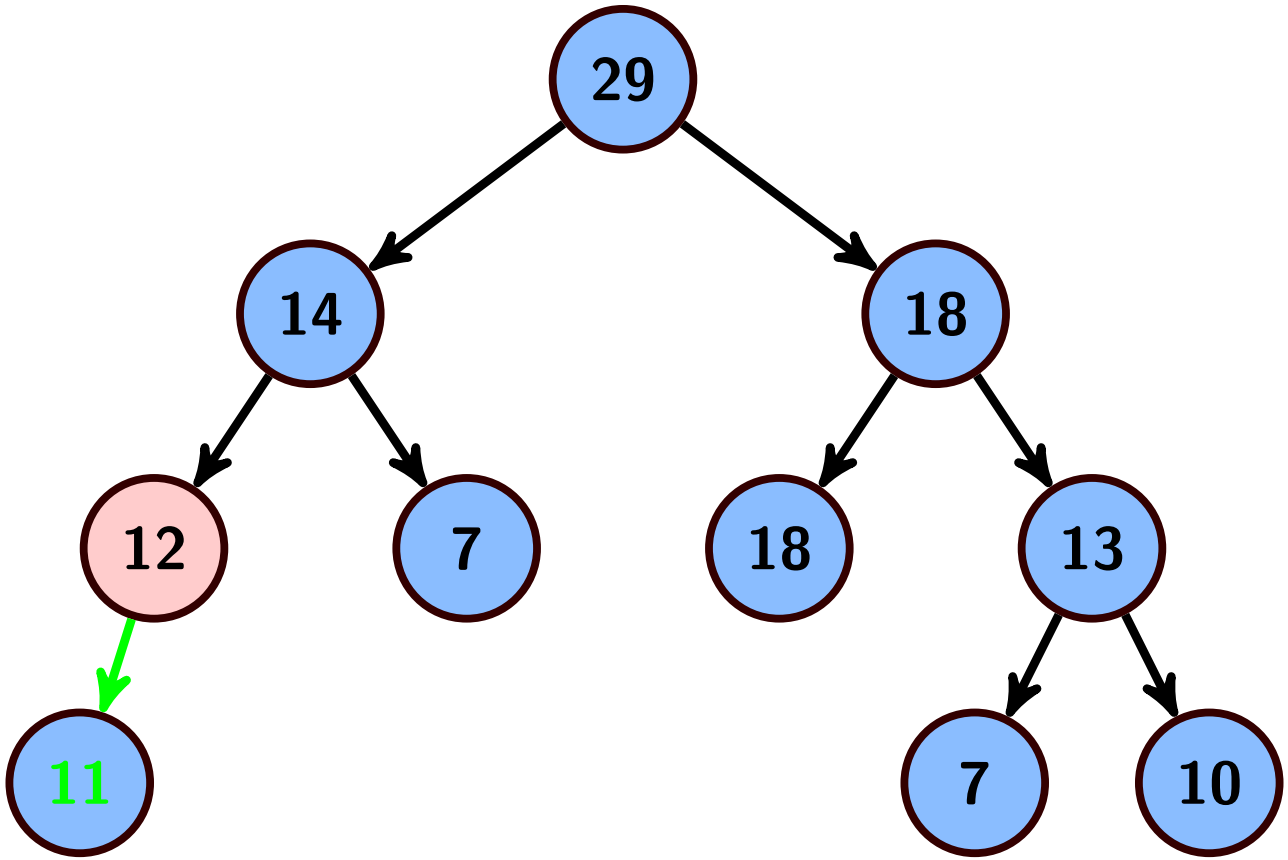
Heap – Eliminarea maximului

Va trebui să aplic operația de *cernere* în jos!



Heap – Eliminarea maximului

Va trebui să aplic operația de *cernere* în jos!

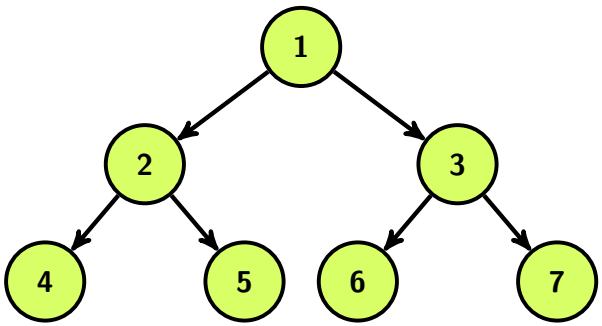


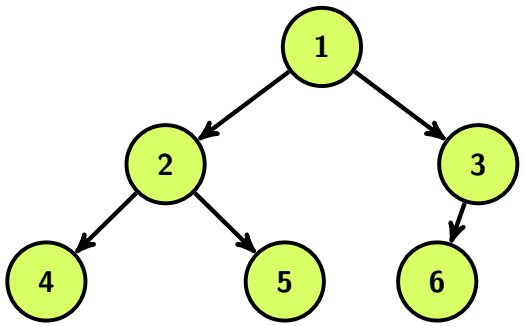
Heap

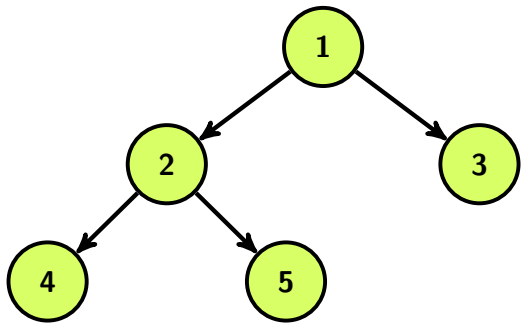
Important

Un heap este reprezentat de multe ori ca un **arbore binar complet**.

Arbore binar complet – un arbore binar care este complet umplut cu posibila excepție a ultimului nivel care este umplut de la stânga la dreapta.





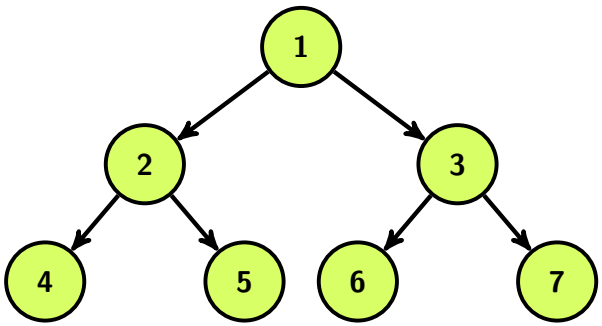


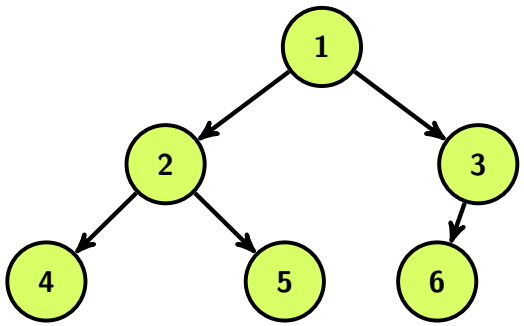
Heap

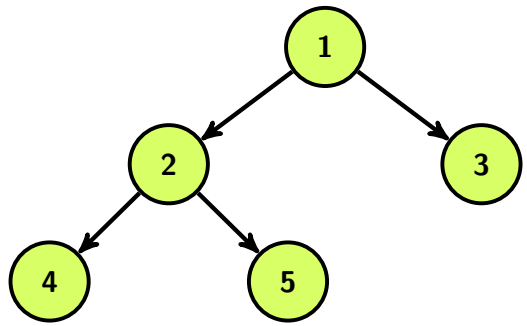
Important

Un heap este reprezentat de multe ori ca un **arbore binar complet**.

Arbore binar complet – un arbore binar care este complet umplut cu posibila excepție a ultimului nivel care este umplut de la stânga la dreapta.





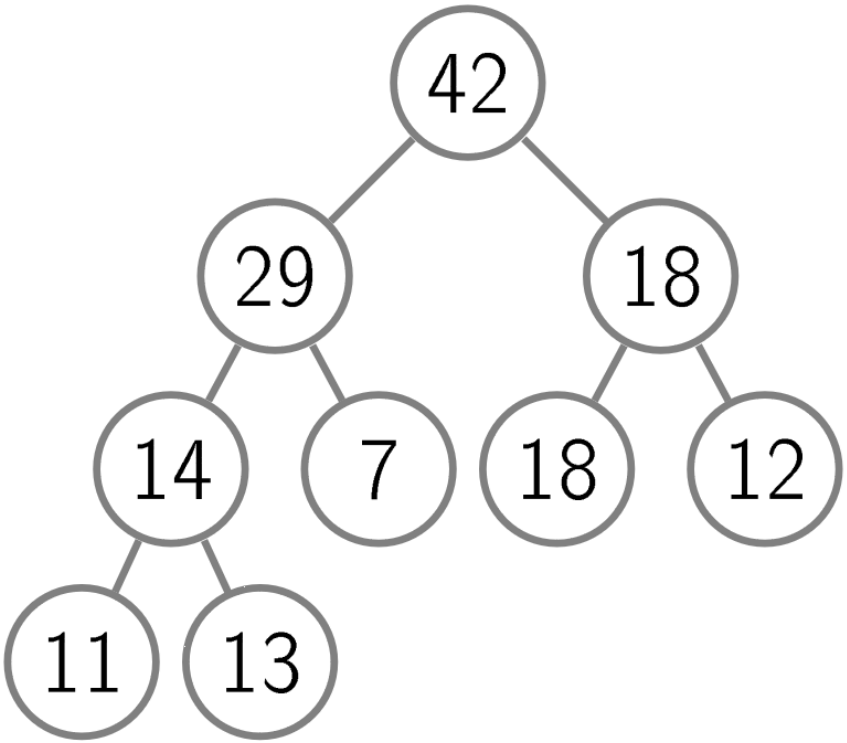


Un arbore binar complet cu N noduri are înălțimea cel mult $O(\log N)$.

Heap

Important

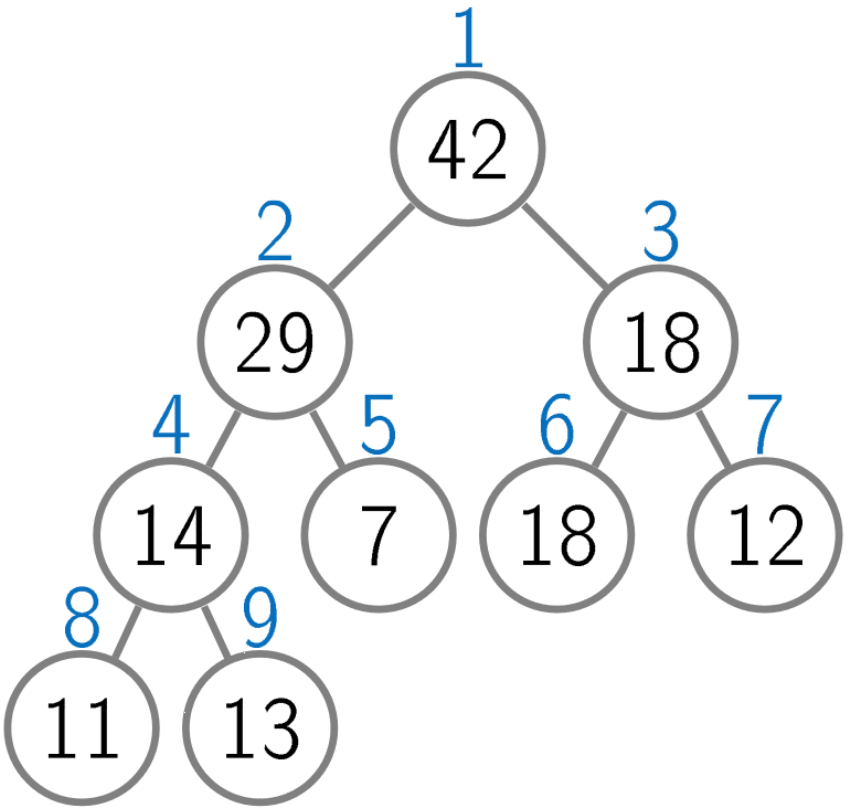
Un alt avantaj al arborilor binari compleți: pot fi reprezentați convenabil ca un vector. Astfel, nu mai avem nevoie de legăturile explicite dintr-un arbore binar.



Heap

Important

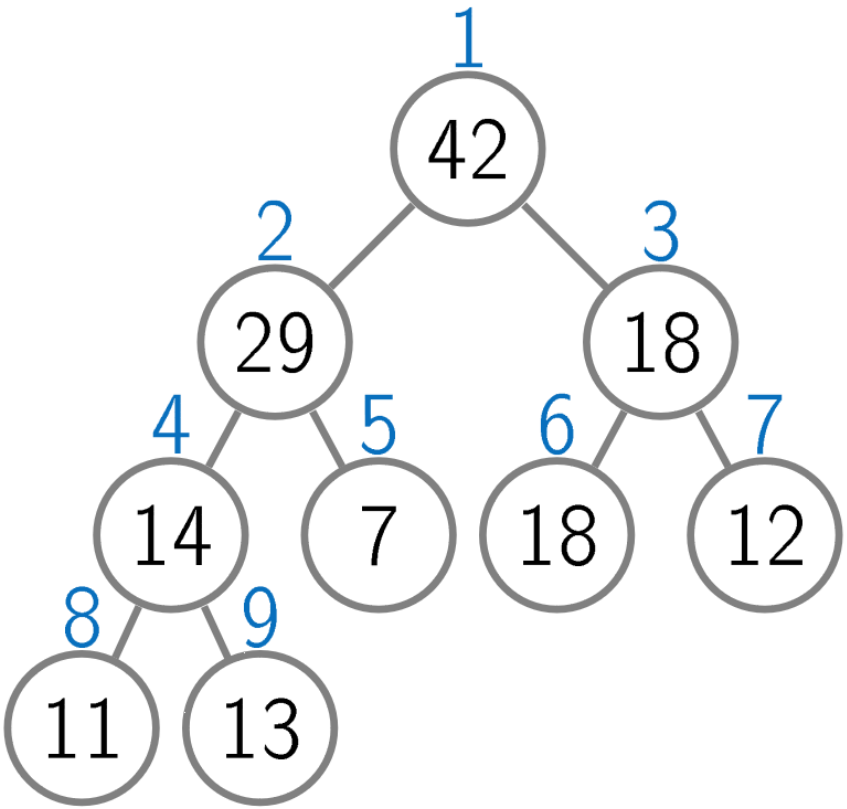
Un alt avantaj al arborilor binari compleți: pot fi reprezentați convenabil ca un vector. Astfel, nu mai avem nevoie de legăturile explicite dintr-un arbore binar.



Heap

Important

Un alt avantaj al arborilor binari compleți: pot fi reprezentați convenabil ca un vector. Astfel, nu mai avem nevoie de legăturile explicite dintr-un arbore binar.



$$\text{parent}(i) = \lfloor \frac{i}{2} \rfloor$$

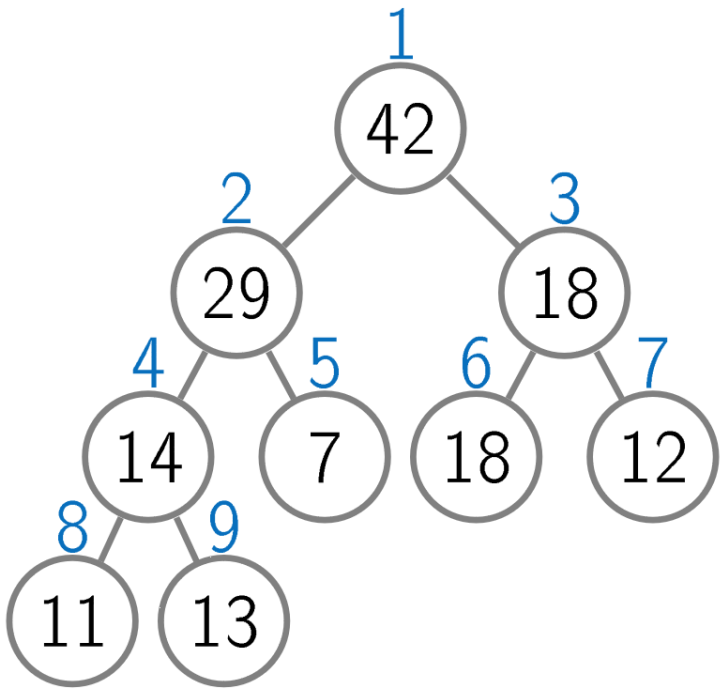
$$\text{leftchild}(i) = 2i$$

$$\text{rightchild}(i) = 2i + 1$$

Heap

Important

Un alt avantaj al arborilor binari compleți: pot fi reprezentați convenabil ca un vector. Astfel, nu mai avem nevoie de legăturile explicite dintr-un arbore binar.



$$\text{parent}(i) = \left\lfloor \frac{i}{2} \right\rfloor$$

$$\text{leftchild}(i) = 2i$$

$$\text{rightchild}(i) = 2i + 1$$

1	2	3	4	5	6	7	8	9
42	29	18	14	7	18	12	11	13

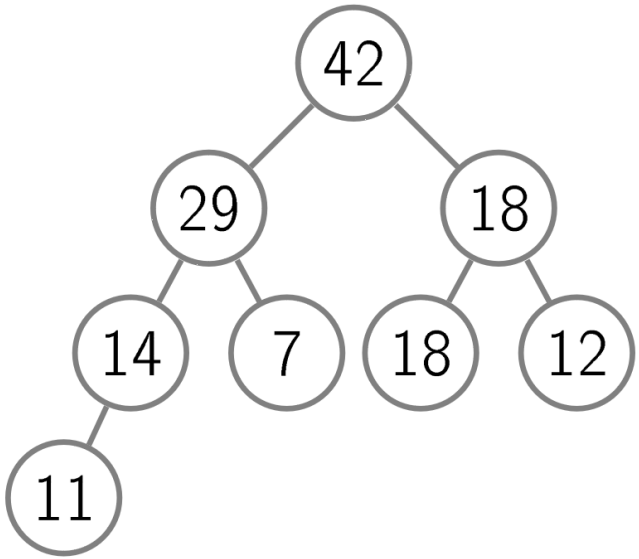
Heap – Inserarea unui element

Cum facem ca heap-ul să rămână **arbore binar complet** după inserare?

Heap – Inserarea unui element

Cum facem ca heap-ul să rămână **arbore binar complet** după inserare?

Inserăm nodul ca frunză în cea mai din stânga poziție disponibilă!



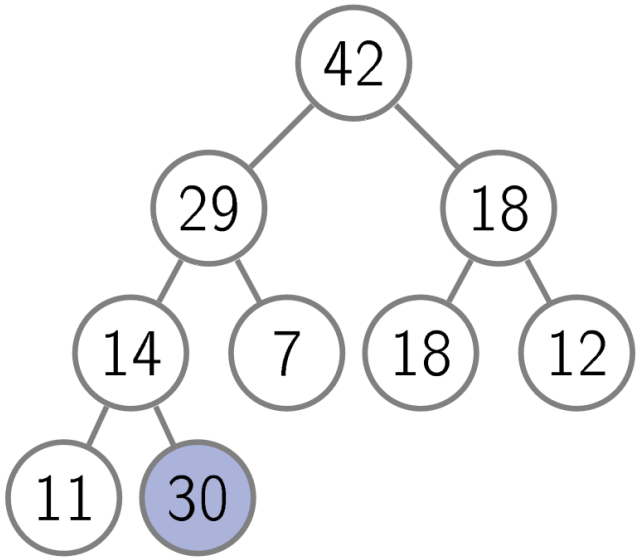
Heap – Inserarea unui element

Cum facem ca heap-ul să rămână **arbore binar complet** după inserare?

Heap – Inserarea unui element

Cum facem ca heap-ul să rămână **arbore binar complet** după inserare?

Inserăm nodul ca frunză în cea mai din stânga poziție disponibilă!



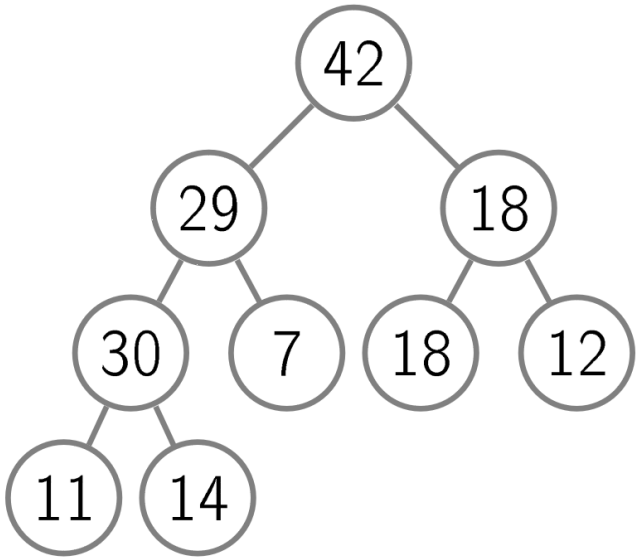
Heap – Inserarea unui element

Cum facem ca heap-ul să rămână **arbore binar complet** după inserare?

Heap – Inserarea unui element

Cum facem ca heap-ul să rămână **arbore binar complet** după inserare?

Inserăm nodul ca frunză în cea mai din stânga poziție disponibilă!



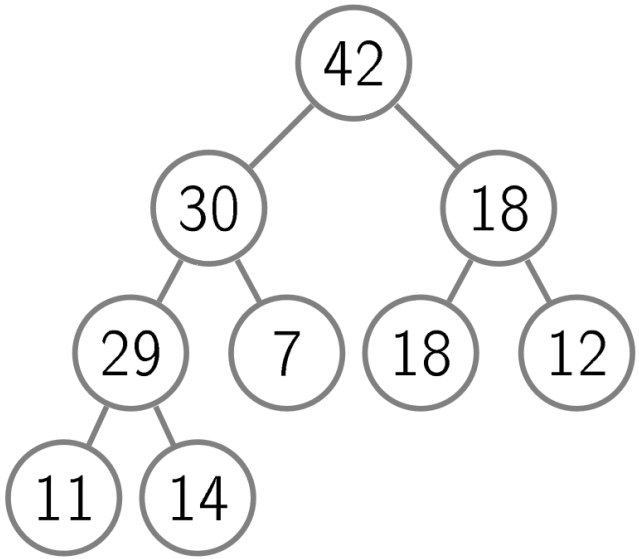
Heap – Inserarea unui element

Cum facem ca heap-ul să rămână **arbore binar complet** după inserare?

Heap – Inserarea unui element

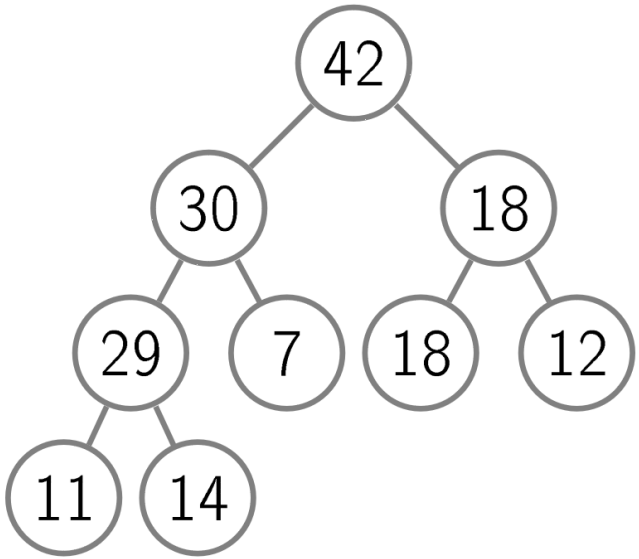
Cum facem ca heap-ul să rămână **arbore binar complet** după inserare?

Inserăm nodul ca frunză în cea mai din stânga poziție disponibilă!



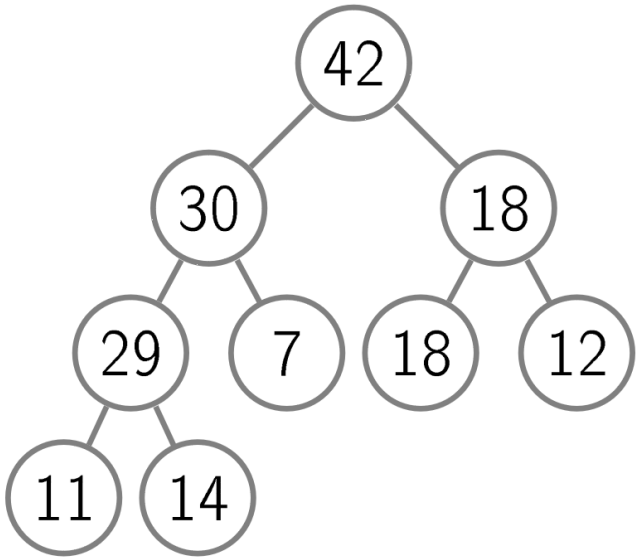
Heap – Eliminarea maximului

Cum facem ca heap-ul să rămână **arbore binar complet** după eliminarea maximului?



Heap – Eliminarea maximului

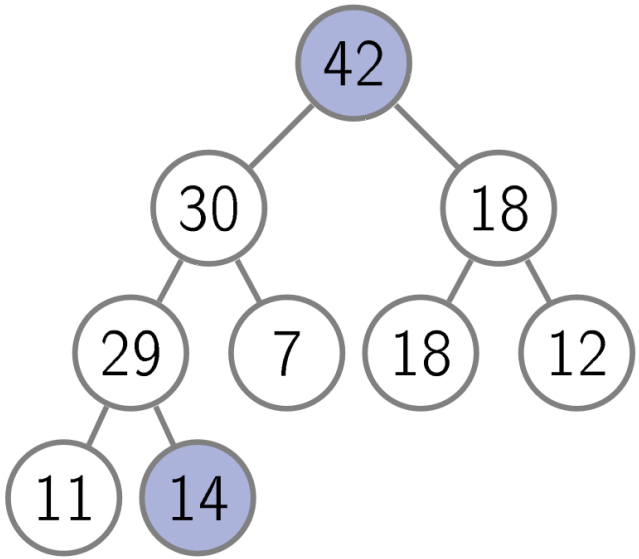
Cum facem ca heap-ul să rămână **arbore binar complet** după eliminarea maximului?



Înlocuim rădăcina cu nodul frunză cel mai din dreapta de pe ultimul nivel!

Heap – Eliminarea maximului

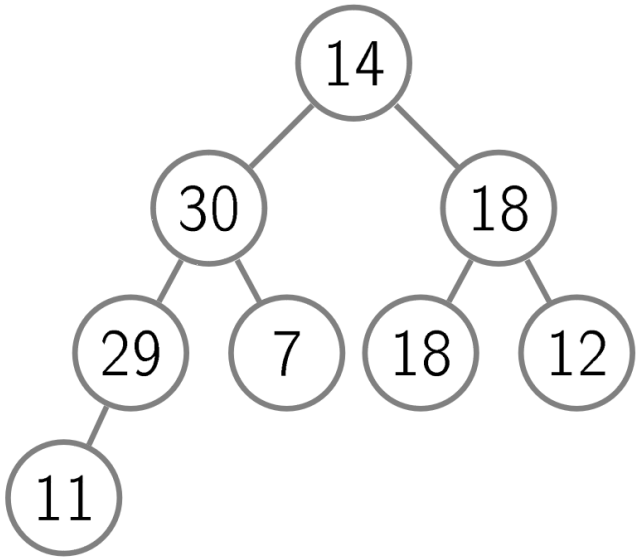
Cum facem ca heap-ul să rămână **arbore binar complet** după eliminarea maximului?



Înlocuim rădăcina cu nodul frunză cel mai din dreapte de pe ultimul nivel!

Heap – Eliminarea maximului

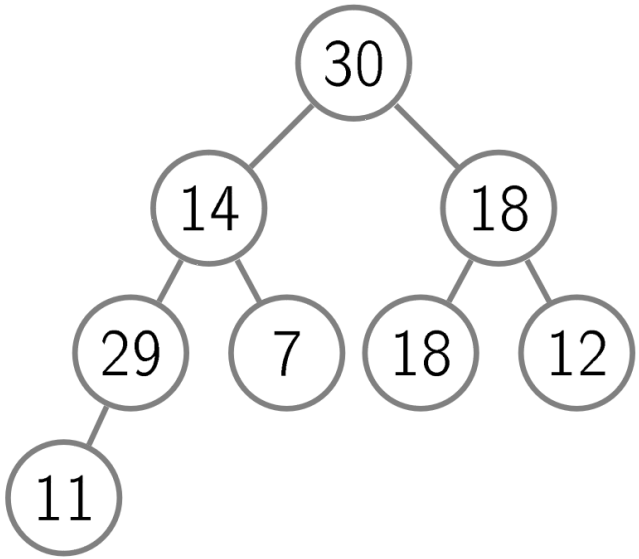
Cum facem ca heap-ul să rămână **arbore binar complet** după eliminarea maximului?



Înlocuim rădăcina cu nodul frunză cel mai din dreapte de pe ultimul nivel!

Heap – Eliminarea maximului

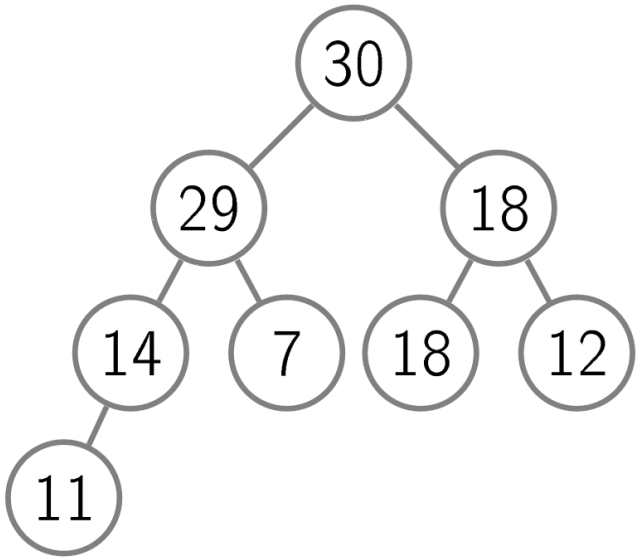
Cum facem ca heap-ul să rămână **arbore binar complet** după eliminarea maximului?



Înlocuim rădăcina cu nodul frunză cel mai din dreapte de pe ultimul nivel!

Heap – Eliminarea maximului

Cum facem ca heap-ul să rămână **arbore binar complet** după eliminarea maximului?



Înlocuim rădăcina cu nodul frunză cel mai din dreapte de pe ultimul nivel!

Implementare Heap

Cum putem defini structura pentru heap?

Implementare Heap

Cum putem defini structura pentru heap?

```
1  typedef int Type;
2
3  typedef struct heap {
4      Type *vector;
5      int size;
6      int capacity;
7      int (*compare_func)(const void*, const void*);
8  } *Heap;
```

Implementare Heap – Indexarea în vector

Parent(i)

return $\lfloor \frac{i-1}{2} \rfloor$

LeftChild(i)

return $2i + 1$

RightChild(i)

return $2i + 2$

Implementare Heap – Funcția de inițializare

```
9  Heap initHeap(int capacity, int (*compare_func) (const
   ↪  void*, const void*))
10 {
11     Heap h = (Heap)malloc(sizeof(struct heap));
12     h->size = 0;
13     h->capacity = capacity;
14     h->vector = malloc(capacity * sizeof(Type));
15     h->compare_func = compare_func;
16     return h;
17 }
```

Implementare Heap – *Cernere* în jos

```
18  Heap siftDown(Heap h, int index) {
19      int maxIndex = index;
20      int l = index * 2 + 1;
21      if(l < h->size && h->compare_func(&h->vector[l],
    ↪   &h->vector[maxIndex]) > 0) maxIndex = l;
22      int r = index * 2 + 2;
23      if(r < h->size && h->compare_func(&h->vector[r],
    ↪   &h->vector[maxIndex]) > 0) maxIndex = r;
24      if(index != maxIndex) {
25          Type aux = h->vector[index];
26          h->vector[index] = h->vector[maxIndex];
27          h->vector[maxIndex] = aux;
28          h = siftDown(h, maxIndex);
29      }
30      return h;
31  }
```

Implementare Heap – *Cernere* în sus

```
32  Heap siftUp(Heap h, int index) {
33      while (index >= 0 &&
    ↪   h->compare_func(&h->vector[(index - 1) / 2],
    ↪   &h->vector[index]) < 0) {
34          Type aux = h->vector[(index - 1) / 2];
35          h->vector[(index - 1) / 2] = h->vector[index];
36          h->vector[index] = aux;
37          index = (index - 1) / 2;
38      }
39      return h;
40  }
```

Implementare Heap – Inserare element

```
41  Heap insertHeap(Heap h, Type element) {
42      if (h->size == h->capacity) {
43          h->capacity *= 2;
44          h->vector = realloc(h->vector, h->capacity *
    ↪  sizeof(Type));
45      }
46      h->vector[h->size] = element;
47      h = siftUp(h, h->size);
48      h->size++;
49      return h;
50  }
```

Implementare Heap – Eliminare maxim

```
32  Type extractMax(Heap h) {
33      Type max;
34      if(h && h->size > 0)
35      {
36          max = h->vector[0];
37          h->vector[0] = h->vector[h->size - 1];
38          h->size--;
39          h = siftDown(h, 0);
40          return max;
41      }
42      exit(1);
43  }
```

Implementare Heap – Dealocare memorie

```
32  Heap freeHeap(Heap h) {
33      if (h != NULL)
34          free(h->vector);
35      free(h);
36      return NULL;
37  }
```

Implementare Heap – Dealocare memorie

```
32  Heap freeHeap(Heap h) {
33      if (h != NULL)
34          free(h->vector);
35      free(h);
36      return NULL;
37  }
```

Funcție de comparație

```
38  int compare(const void *a, const void *b) {
39      int *pa, *pb;
40      pa = (int*) a;
41      pb = (int*) b;
42      if (*pa < *pb)
43          return -1;
44      else if (*pa > *pb)
45          return 1;
46      return 0;
47  }
```

Implementare Heap – Exemplu

```
1  Heap heap = initHeap(10, compare);
2  heap = insertHeap(heap, 10);
3  heap = insertHeap(heap, 5);
4  heap = insertHeap(heap, 6);
5  heap = insertHeap(heap, 22);
6  heap = insertHeap(heap, 18);
7  heap = insertHeap(heap, 94);
8  heap = insertHeap(heap, 63);
9  heap = insertHeap(heap, 99);
10 while (heap->size > 0) {
11     printf("%d ", extractMax(heap));
12 }
13 printf("\n");
```

Implementare Heap – Exemplu

```
1  Heap heap = initHeap(10, compare);
2  heap = insertHeap(heap, 10);
3  heap = insertHeap(heap, 5);
4  heap = insertHeap(heap, 6);
5  heap = insertHeap(heap, 22);
6  heap = insertHeap(heap, 18);
7  heap = insertHeap(heap, 94);
8  heap = insertHeap(heap, 63);
9  heap = insertHeap(heap, 99);
10 while (heap->size > 0) {
11     printf("%d ", extractMax(heap));
12 }
13 printf("\n");
```

```
99 94 63 22 18 10 6 5
```

Schimbarea priorității

Remove(i)

$H[i] \leftarrow \infty$

SiftUp(i)

ExtractMax()

Ștergerea unui nod

ChangePriority(i, p)

$oldp \leftarrow H[i]$

$H[i] \leftarrow p$

if $p > oldp$:

  SiftUp(i)

else:

  SiftDown(i)

Algoritmul HeapSort

HeapSort($A[1 \ldots n]$)

```
create an empty priority queue
for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```

Transformarea unui vector în Heap

BuildHeap($A[1 \ldots n]$)

$size \leftarrow n$

```
for i from ⌊n/2⌋ downto 1:
    SiftDown(i)
```

Algoritmul HeapSort fără memorie auxiliară

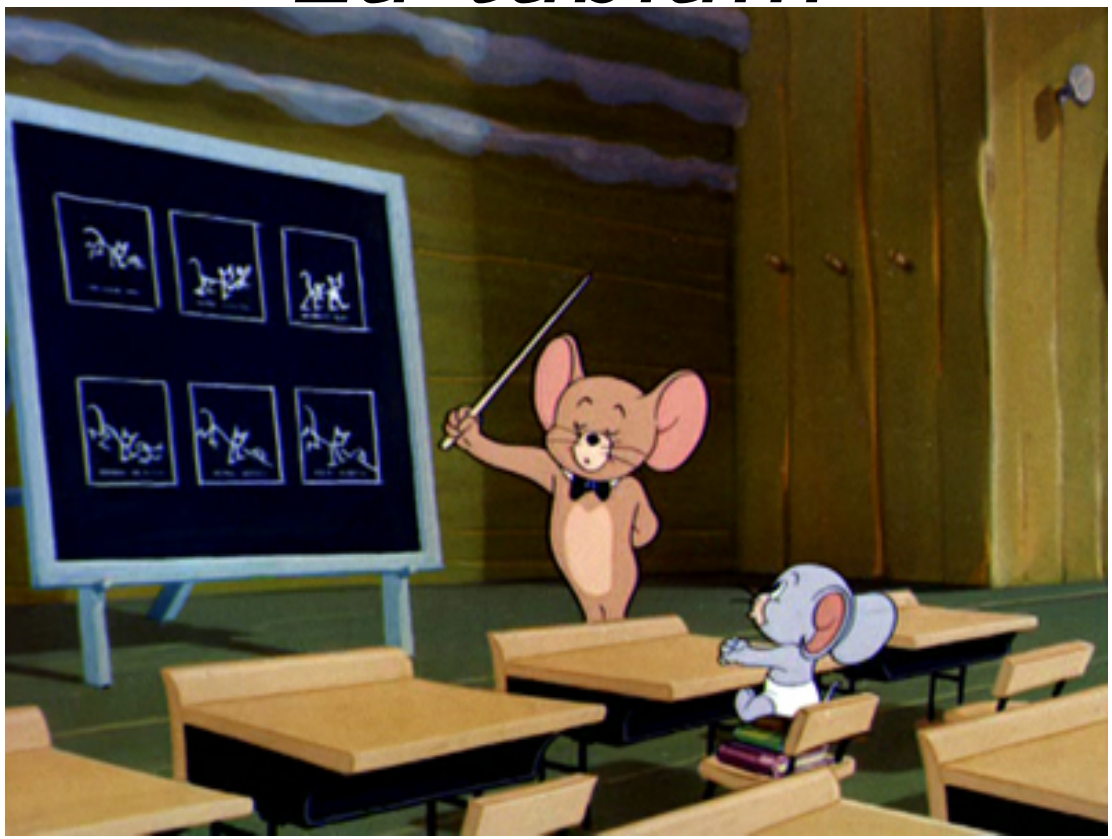
HeapSort($A[1 \ldots n]$)

```
BuildHeap(A)                                   {size = n}
repeat (n − 1) times:
    swap A[1] and A[size]
    size ← size − 1
    SiftDown(1)
```

La tablă...

Decompresia Huffman

La tablă...



Paște Fericit!