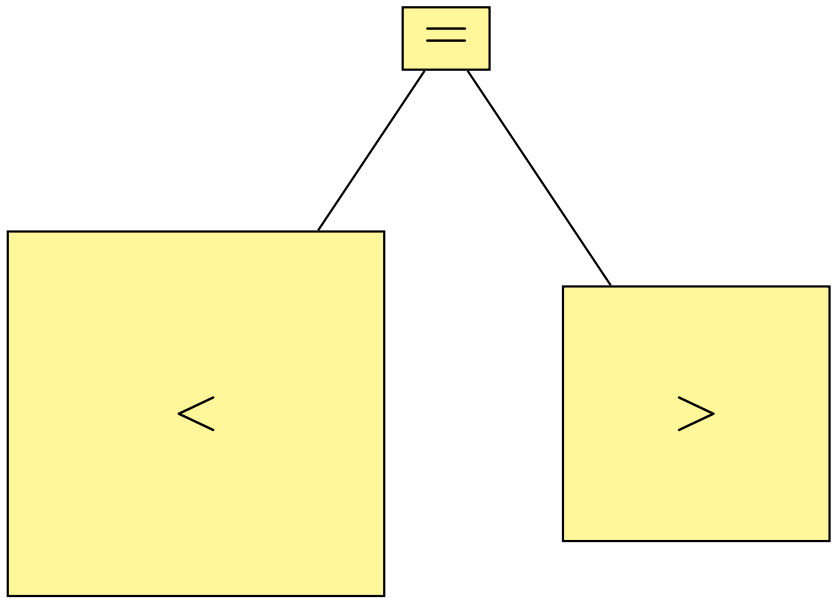
- Diccionaris
  - Arbres binaris de cerca
- Cues de prioritat
  - Heaps
  - Heapsort
- Problemes

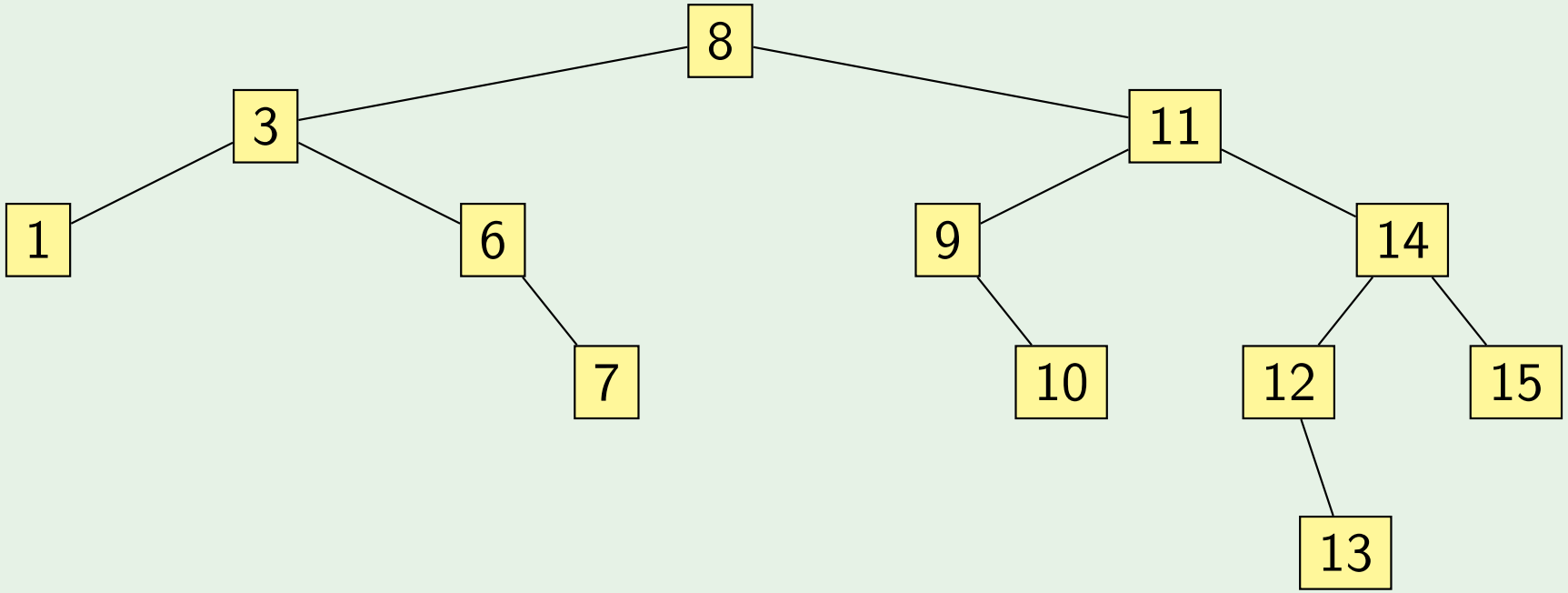
- Un diccionari D sobre un conjunt K de claus i un conjunt V de valors és un conjunt finit de parells (k, v) amb $k \in K$ i $v \in V$ on no hi ha dos parells diferents amb la mateixa clau.
  - Diccionari estàtic (creació, consulta).
  - Diccionari semidinàmic (creació, inserció, modificació, consulta).
  - Diccionari dinàmic (creació, inserció, modificació, eliminació, consulta).
- crear() crea un diccionari D buit.
- inserir(D, k, v) insereix el parell (k, v) en el diccionari D.
- eliminar(D, k) elimina el parell $(k, \ldots)$ del diccionari D; dóna error si no hi ha cap parell amb clau k a D.
- consultar(D, k) retorna el valor v associat a la clau k en el diccionari D; dóna error si no hi ha cap parell amb clau k a D.
- pertany(D, k) retorna *true* si i només si el diccionari D conté un parell amb clau k.

- Un arbre binari de cerca T és un arbre binari que és buit o bé compleix:
  - els subarbres esquerre i dret són arbres binaris de cerca
  - $\text{clau}(x) < \text{clau}(\text{arrel}(T)) < \text{clau}(y)$ per a tot element x del subarbre esquerre de T i tot element y del subarbre dret de T



- Un recorregut en inordre d'un arbre binari de cerca n'enumera els elements per ordre ascendent de clau.

Exemple (Arbre binari de cerca)



Exemple (Recorregut en inordre d'un arbre binari de cerca)

$$[1, 3, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]$$

- **type** bst **is pointer to** node
- **type** node **is**
  record
    info : elem
    left : bst
    right : bst
  end record
- **type** elem **is**
  record
    clau : $\cdots$
    valor : $\cdots$
  end record
- **type** dicc **is** bst

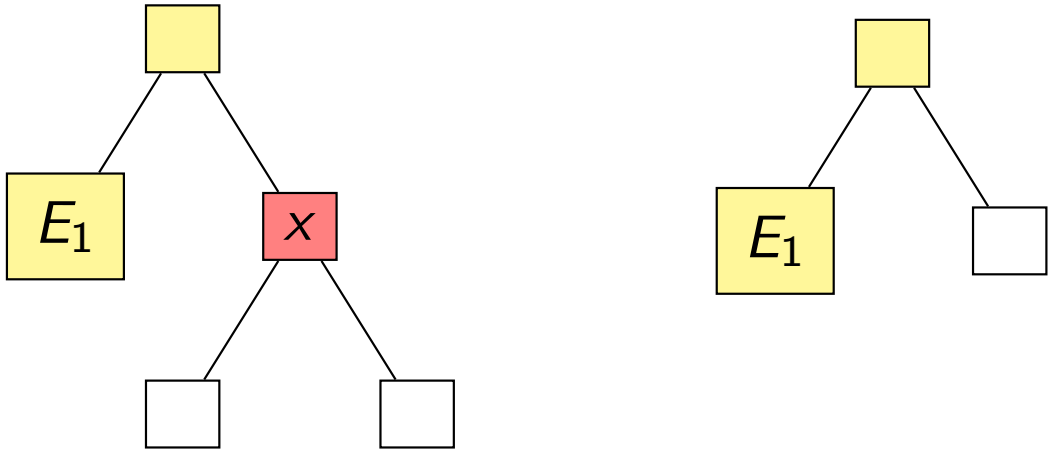
- **procedure** consultar (D: dicc; k: clau; **s** v: valor; **s** *trobat*: bool)

```
elem x
if D = null then
    trobat := false
else
    x := D → info
    if x.clau > k then
        consultar(D → left, k, v, trobat)
    else
        if x.clau < k then
            consultar(D → right, k, v, trobat)
        else
            trobat := true
            v := x.valor
```

- **function** inserir (D: dicc; k: clau; v: valor)
  if $D =$ **null then**
  pointer to node $p :=$ **new** (node)
  if $p =$ **null then**
  error
  $p \rightarrow$ info $:= \langle k, v \rangle$
  $p \rightarrow$ left $:=$ **null**; $p \rightarrow$ right $:=$ **null**
  return p
  else
  elem $x := D \rightarrow$ info
  if x.clau $> k$ **then**
  $D \rightarrow$ left $:=$ inserir$(D \rightarrow$ left$, k, v)$
  else
  if x.clau $< k$ **then**
  $D \rightarrow$ right $:=$ inserir$(D \rightarrow$ right$, k, v)$
  else
  $D \rightarrow$ info $:= \langle k, v \rangle$
  return D

- Per eliminar un element d'un arbre binari de cerca, cal distingir diversos casos.

– Tots dos subarbres del node que conté l'element a eliminar són buits.



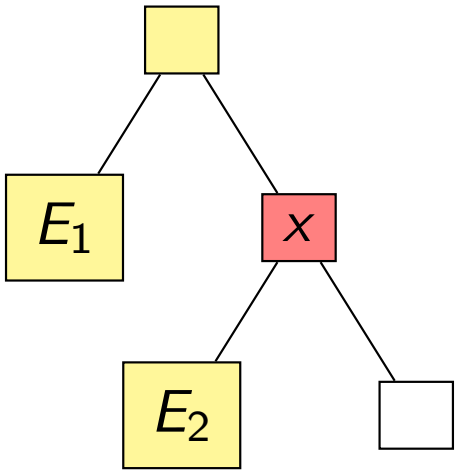
– Un dels subarbres del node que conté l'element a eliminar és buit.

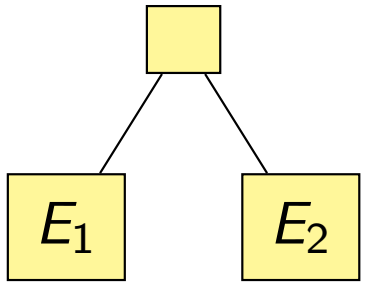
– Cap dels subarbres del node que conté l'element a eliminar no és buit.

- Per eliminar un element d'un arbre binari de cerca, cal distingir diversos casos.

– Tots dos subarbres del node que conté l'element a eliminar són buits.

– Un dels subarbres del node que conté l'element a eliminar és buit.





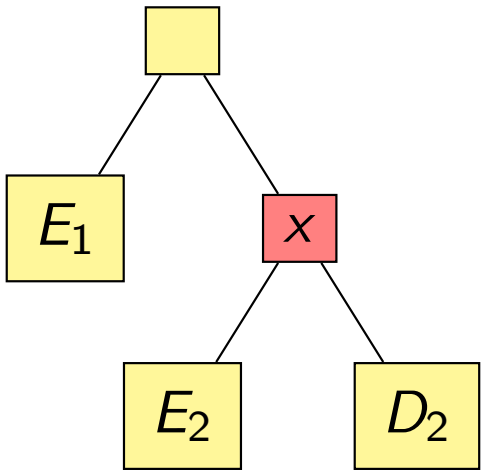
– Cap dels subarbres del node que conté l'element a eliminar no és buit.

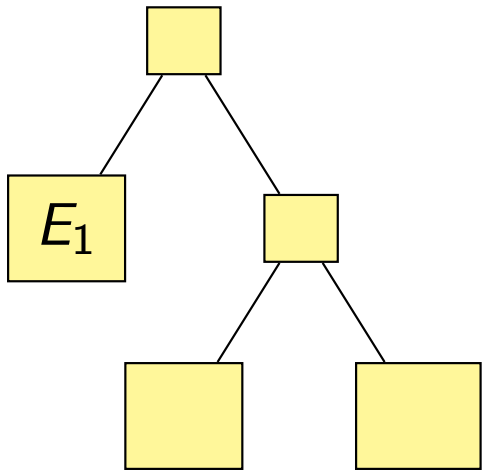
- Per eliminar un element d'un arbre binari de cerca, cal distingir diversos casos.

– Tots dos subarbres del node que conté l'element a eliminar són buits.

– Un dels subarbres del node que conté l'element a eliminar és buit.

– Cap dels subarbres del node que conté l'element a eliminar no és buit.

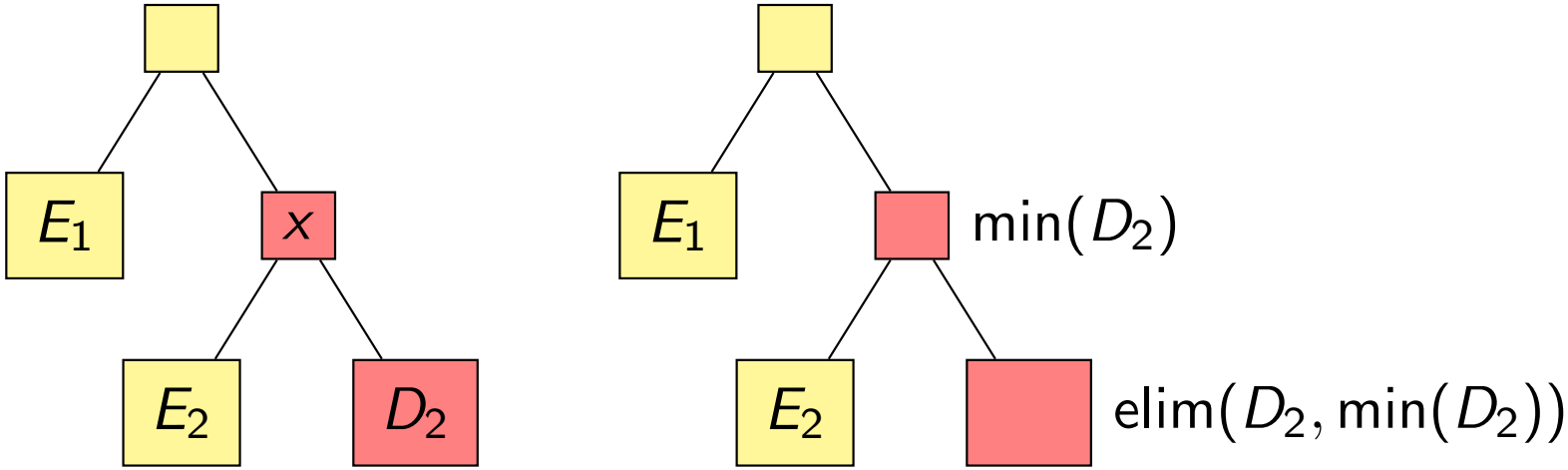




- **procedure** elim (D: dicc; k: clau)
  if buit(D) **then**
  return D
  else
  if $D \rightarrow$ info.clau $= k$ **then**
  if buit($D \rightarrow$ left) **then**
  return $D \rightarrow$ right
  else if buit($D \rightarrow$ right) **then**
  return $D \rightarrow$ left
  else
  ⟪cas general⟫
  else
  if $D \rightarrow$ info.clau $> k$ **then**
  return plantar($D \rightarrow$ info, elim($D \rightarrow$ left, k), $D \rightarrow$ right)
  else
  return plantar($D \rightarrow$ info, $D \rightarrow$ left, elim($D \rightarrow$ right, k))

- ⟨cas general⟩ ≡

  return plantar(min($D \to$ right),
  $D \to$ left,
  elim($D \to$ right, min($D \to$ right)))



- **function** min (D: dicc)
  if buit($D \to$ left) **then**
  return $D \to$ info
  else
  return min($D \to$ left)

Exemple (Donades dues claus k_1, k_2 amb $k_1 < k_2$, doneu una operació per tornar una llista ordenada de tots els elements la clau k dels quals sigui compresa entre les claus donades, és a dir, amb $k_1 \leqslant k \leqslant k_2$.)

procedure rang (D : dicc; k_1, k_2 : clau; L : llista(elem))
  if $D \neq$ **null then**
    if $k_1 \leqslant k$ **then**
      rang($D \rightarrow$ left, k_1, k_2, L)
    if $k_1 \leqslant k$ **and** $k \leqslant k_2$ **then**
      afegir_final($L, D \rightarrow$ info)
    if $k \leqslant k_2$ **then**
      rang($D \rightarrow$ right, k_1, k_2, L)

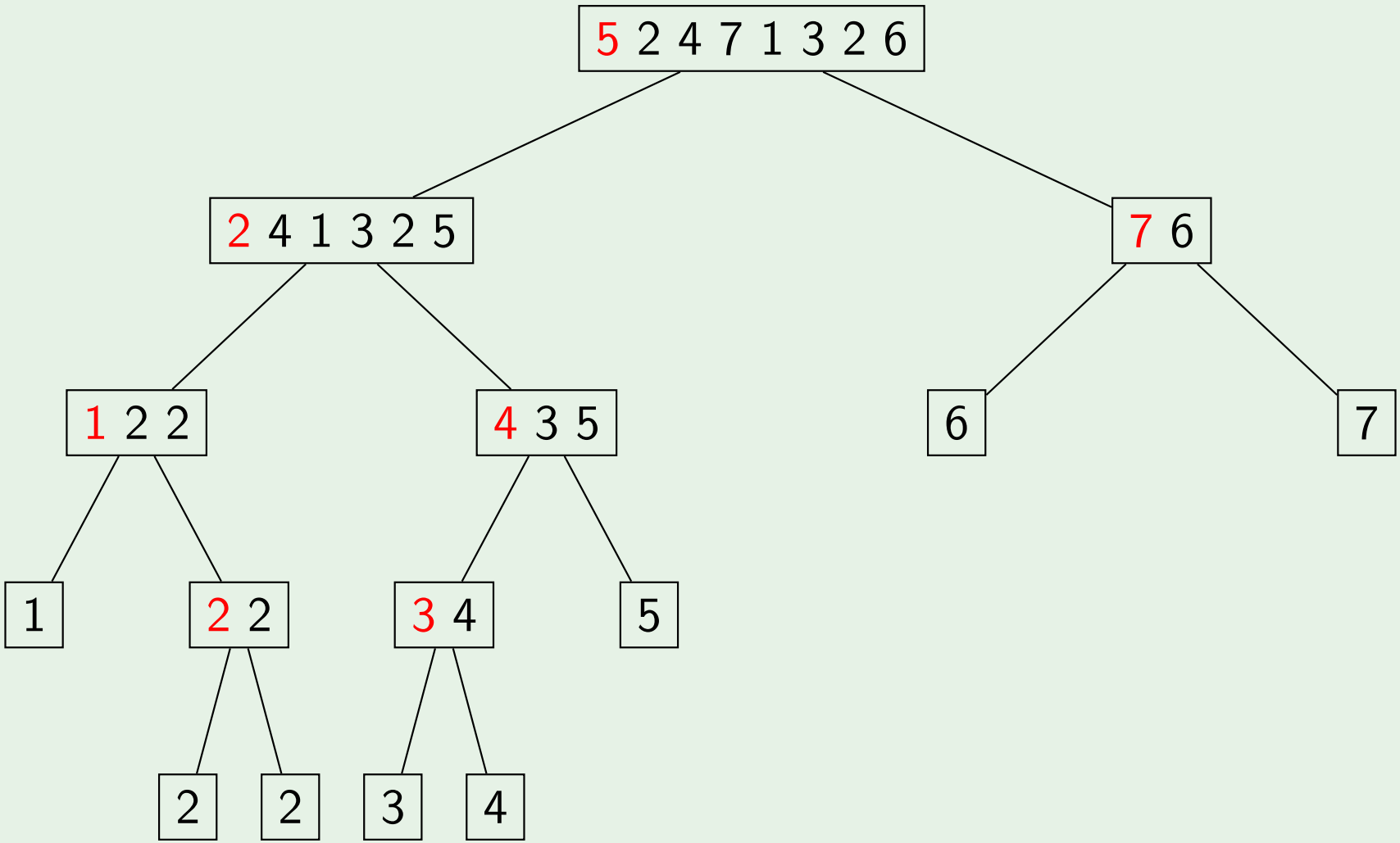
- L'algorisme quicksort és equivalent al procediment d'ordenació següent.

– Inserir tots els elements del vector a ordenar en un arbre binari de cerca inicialment buit.

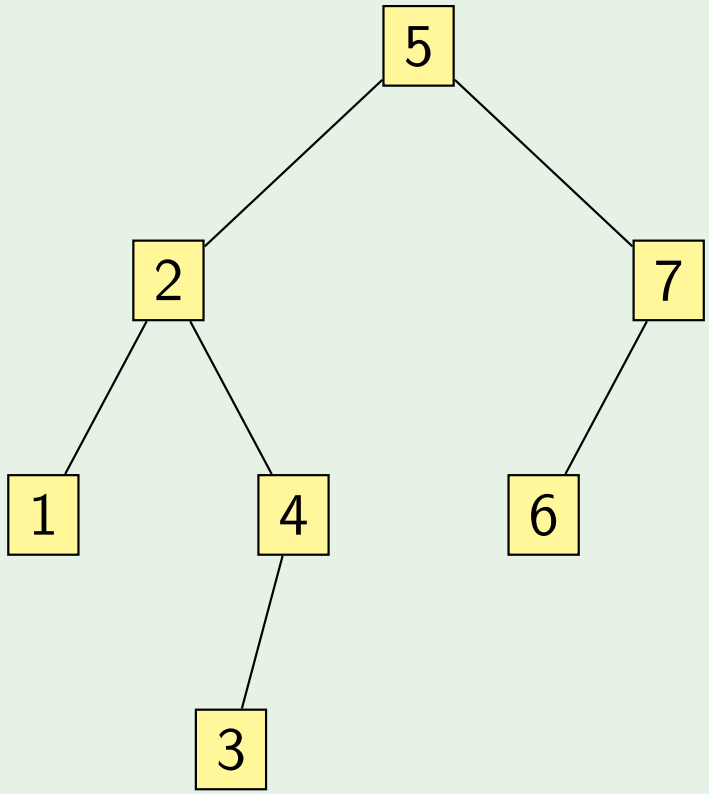
– Construir una seqüència de elements mitjançant un recorregut en inordre de l'arbre binari de cerca resultant.

- De fet, es pot associar un arbre binari de cerca a cada execució del procediment *quicksort*, on l'arrel de l'arbre és l'element pivot p inicial i els subarbres esquerre i dret corresponen a les execucions recursives del procediment *quicksort* sobre els subvectors $A[\ell..p]$ i $A[p+1..u]$, respectivament.
- Òbviament, l'avantatge del procediment *quicksort* rau en el fet d'efectuar l'ordenació sobre el mateix vector, intercanviant-hi elements, sense el cost addicional de construir i mantenir una estructura de dades addicional per a l'arbre binari de cerca.

Exemple (Ordenació de la llista $\langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$ amb quicksort.)



Exemple (Ordenació de la llista $\langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$ per inserció en un arbre binari de cerca.)

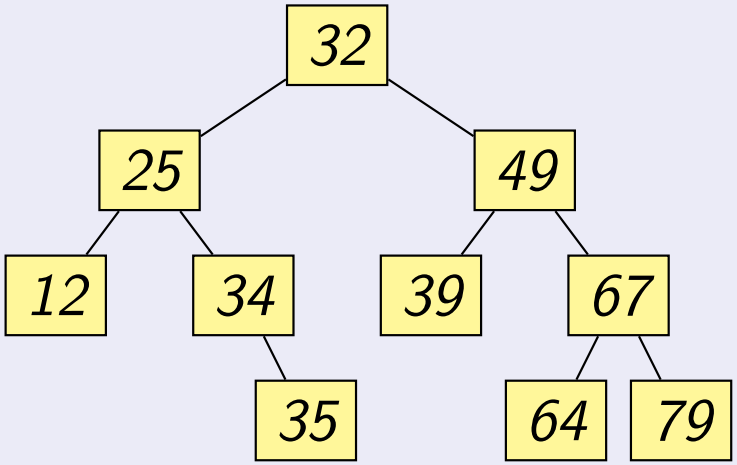


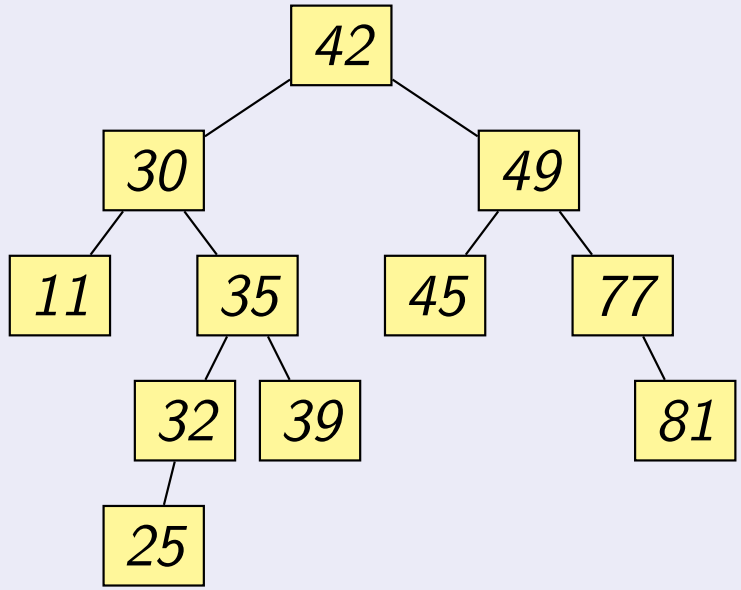
Els problemes marcats amb una estrelleta ($\star$) són més difícils. Si no es diu el contrari i us cal, considereu que tots els logaritmes són en base 2.

- 3.7
- 3.8
- 3.9
- 3.10
- 3.11
- 3.12
- 3.13
- 3.14
- 3.15
- 3.16
- 3.17
- 3.18
- 3.19
- 3.20
- 3.21
- 3.22
- 3.23
- 3.24

Problema (3.7)

Digueu si els arbres següents són arbres binaris de cerca o no i perquè.





Problema (3.8)

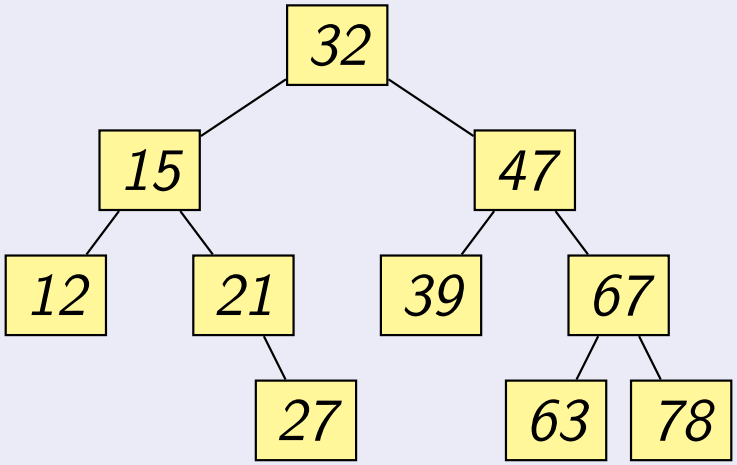
Partint d'un arbre binari de cerca buit, inseriu amb l'algorisme clàssic, l'una rera l'altra, la seqüència de claus $32, 15, 47, 67, 78, 39, 63, 21, 12, 27$.

Problema (3.9)

Partint d'un arbre binari de cerca buit, inseriu amb l'algorisme clàssic, l'una rera l'altra, la seqüència de claus $12, 15, 21, 27, 32, 39, 47, 63, 67, 78$.

Problema (3.10)

Partint de l'arbre binari de cerca següent, elimineu les claus $63, 21, 15, 32$ *l'una rera l'altra. Expliqueu quin algorisme heu usat per eliminar les claus.*



Problema (3.11)

Expliqueu si és cert o no que en un arbre binari de cerca no buit, l'element màxim pot tenir fill esquerre però no pot tenir fill dret.

Problema (3.12)

Demostreu que el recorregut en inordre d'un arbre binari de cerca visita els elements en ordre creixent.

Per als problemes següents, utilitzeu aquesta definició de tipus per als arbres binaris de cerca:

```
struct node {
  Elem x;   // Informacio en el node
  node* fe; // Punter al fill esquerre
  node* fd; // Punter al fill dret

  node (Elem x, node* fe, node* fdr) {
    this->x = x;  this->fe = fe;  this->fd = fd;
  }
  ~node () {
    delete fe;  delete fd;
  }
};

typedef node* abc; // Un ABC es denota per un punter
                   // a la seva arrel (null si es buit)
```

Problema (3.13)

Implementeu una operació `abc crear()` *que retorni un arbre binari de cerca buit. Quin és el seu cost?*

Problema (3.14)

Implementeu una operació `bool hi_es(abc a, Elem x)` *que indiqui si l'element* `x` *és a l'arbre binari de cerca* `a`*. Quin és el seu cost?*

Problema (3.15)

Implementeu una operació `void afegir(abc& a, Elem x)` *que afegeixi l'element* `x` *a l'arbre binari de cerca* `a`*. Quin és el seu cost?*

Problema (3.16)

Implementeu una operació `void treure(abc& a, Elem x)` *que tregui l'element* `x` *de l'arbre binari de cerca* `a`. *Quin és el seu cost?*

Problema (3.17)

Implementeu una operació `Elem minim(abc a)` *que retorni l'element més petit de l'arbre binari de cerca no buit* `a`*. Quin és el seu cost?*

Problema (3.18)

Implementeu una operació `Elem maxim(abc a)` *que retorni l'element més gran de l'arbre binari de cerca no buit* `a`*. Quin és el seu cost?*

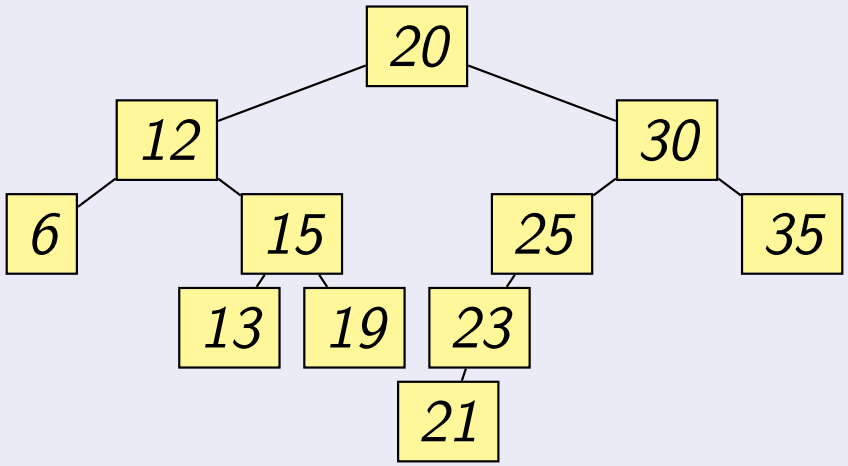
Problema (3.19)

La fusió de dos arbres binaris de cerca és un arbre binari de cerca que conté tots els elements de `a1` *i* `a2`*. Implementeu una operació* `abc fusio(abc& a1, abc& a2)` *que retorni la fusió dels arbres binaris de cerca* `a1` *i* `a2` *(*`a1` *i* `a2` *han de quedar buits després de la crida). Quin és el seu cost?*

Problema (3.20)

Dissenyeu una funció `list<Elem> entre(abc a, Elem x1, Elem x2)` *que, donat un arbre binari de cerca* `a` *i dos elements* `x1` *i* `x2` *amb* `x1` $\leqslant$ `x2`*, retorni una llista amb tots els elements de* `a` *que es trobin entre* `x1` *i* `x2` *en ordre decreixent.*

Per exemple, donat l'arbre



i els valors 18 i 26, cal retornar la llista $\langle 25, 23, 21, 20, 19 \rangle$.

Calculeu el cost del vostre algorisme en el cas pitjor.

Problema (3.21)

Implementeu una funció `int menors(abc a, Elem x)` *que retorni el nombre d'elements de l'arbre binari de cerca* `a` *que són estrictament inferiors a* `x`*. Quin és el seu cost?*

Problema (3.22)

Dibuixeu l'arbre binari de cerca el recorregut en preordre del qual és 8, 5, 2, 1, 4, 3, 6, 7, 11, 9, 13, 12.

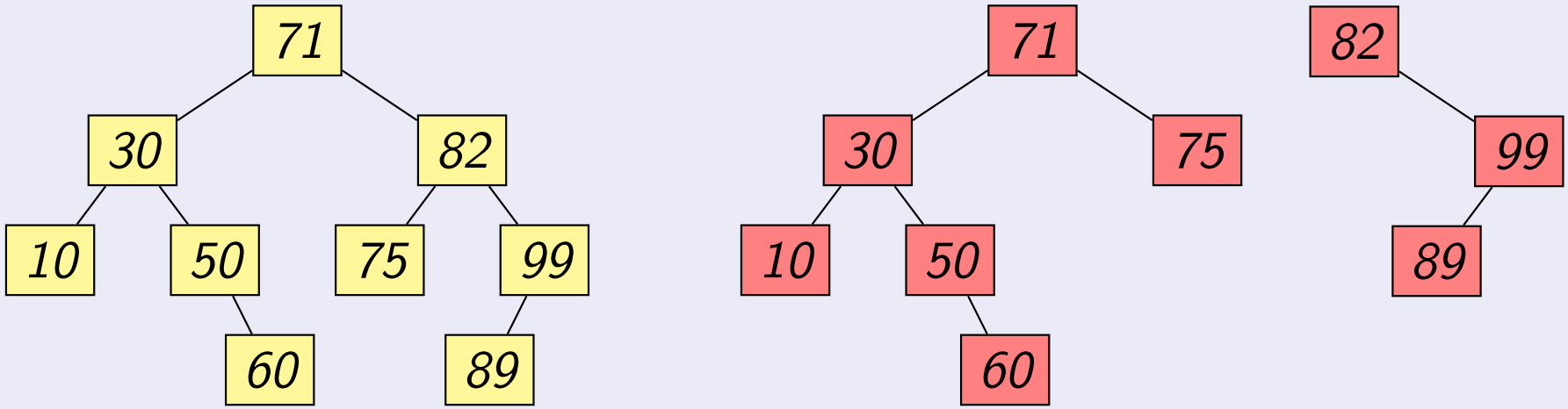
Problema (3.23)

Implementeu la funció `abc reconstrueix(vector<Elem> pre)` *que reconstrueix un arbre binari de cerca a partir del seu recorregut en preordre emmagatzemat en un vector* `pre`*. Quin és el seu cost?*

Problema (3.24)

Implementeu una funció
`void separa(abc& a, Elem x, abc& le, abc& gt)` *que donat un arbre binari de cerca* `a`*, retorni dos arbres binaris de cerca* `le` *i* `gt` *on* `le` *conté tots els elements de* `a` *que són més petits o iguals que* `x` *i* `gt` *conté tots els elements de* `a` *que són més grans que* `x`*. L'arbre original* `a` *ha de quedar destruït.*

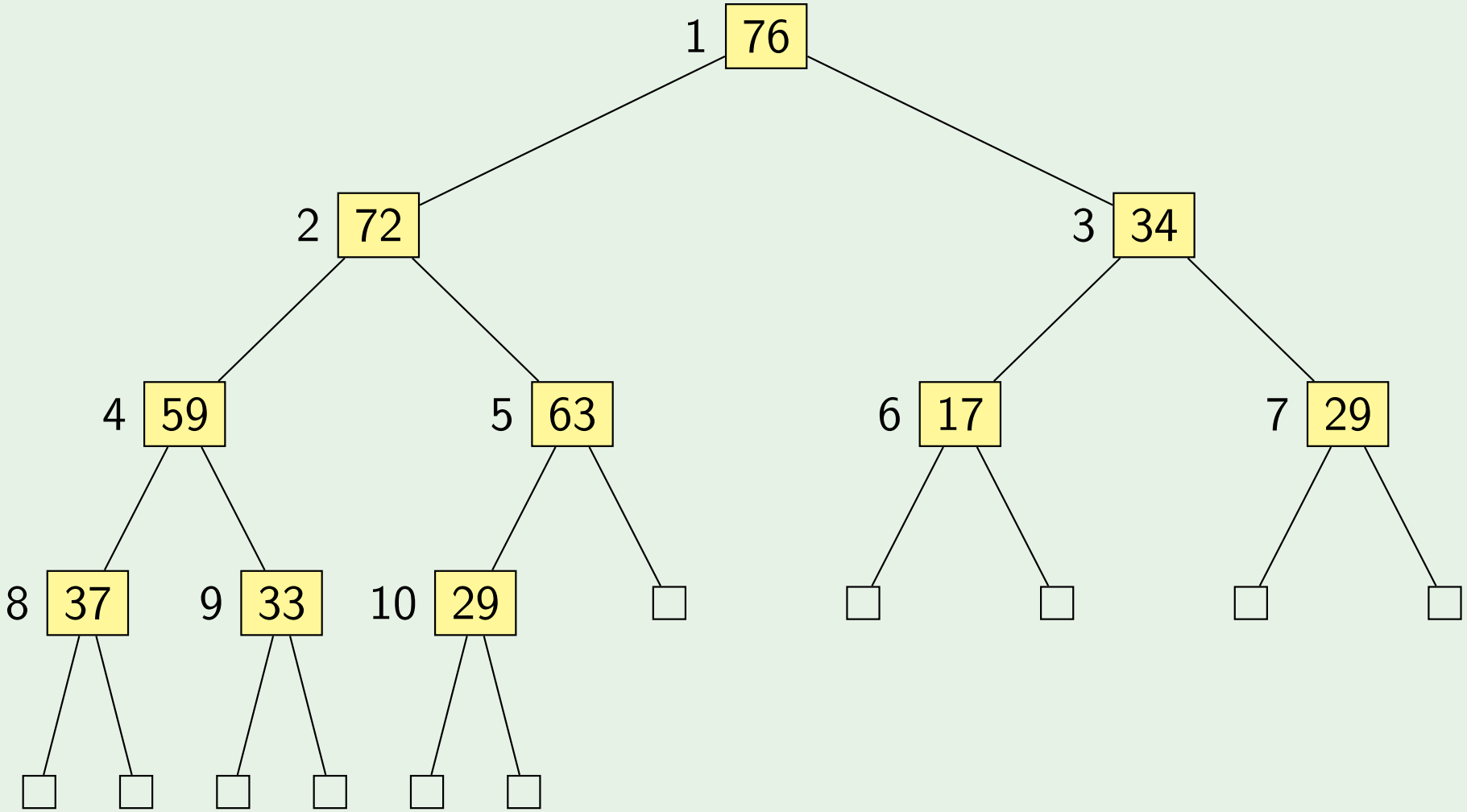
Per exemple, donat l'arbre binari de cerca `a` *de l'esquerra i l'element* $x = 75$*, els arbres* `le` *i* `gt` *són els de la dreta.*



- Una cua de prioritat és una seqüència d'elements amb un valor associat, anomenat la *prioritat* de l'element, en què les eliminacions i consultes només són de l'element de prioritat mínima.
- crear() crea una cua de prioritat Q buida.
- inserir(Q, e, p) insereix el (nou) element e amb prioritat p en la cua de prioritat Q.
- min(Q) retorna l'element de prioritat màxima de la cua de prioritat Q; dóna error si Q és buida.
- elim_min(Q) elimina l'element de prioritat màxima de la cua de prioritat Q; dóna error si Q és buida.
- buit(Q) retorna *true* si i només si la cua de prioritat Q és buida.

- Un monticle (*heap*) és un arbre binari quasi-complet.
  - Totes les fulles són als dos últims nivells de l'arbre.
  - En l'antepenúltim nivell hi ha com a molt un node interior amb un únic fill (esquerre) i tots els nodes a la seva dreta són nodes interiors sense fills.
  - La prioritat de l'element en un node qualsevol és més gran o igual (*max-heap*) o menor o igual (*min-heap*) que la prioritat dels elements en els seus fills esquerre i dret.
- L'element de prioritat màxima (mínima) es troba a l'arrel.
- Un heap de n elements té altura $h = \lceil \log_2(n+1) \rceil$.

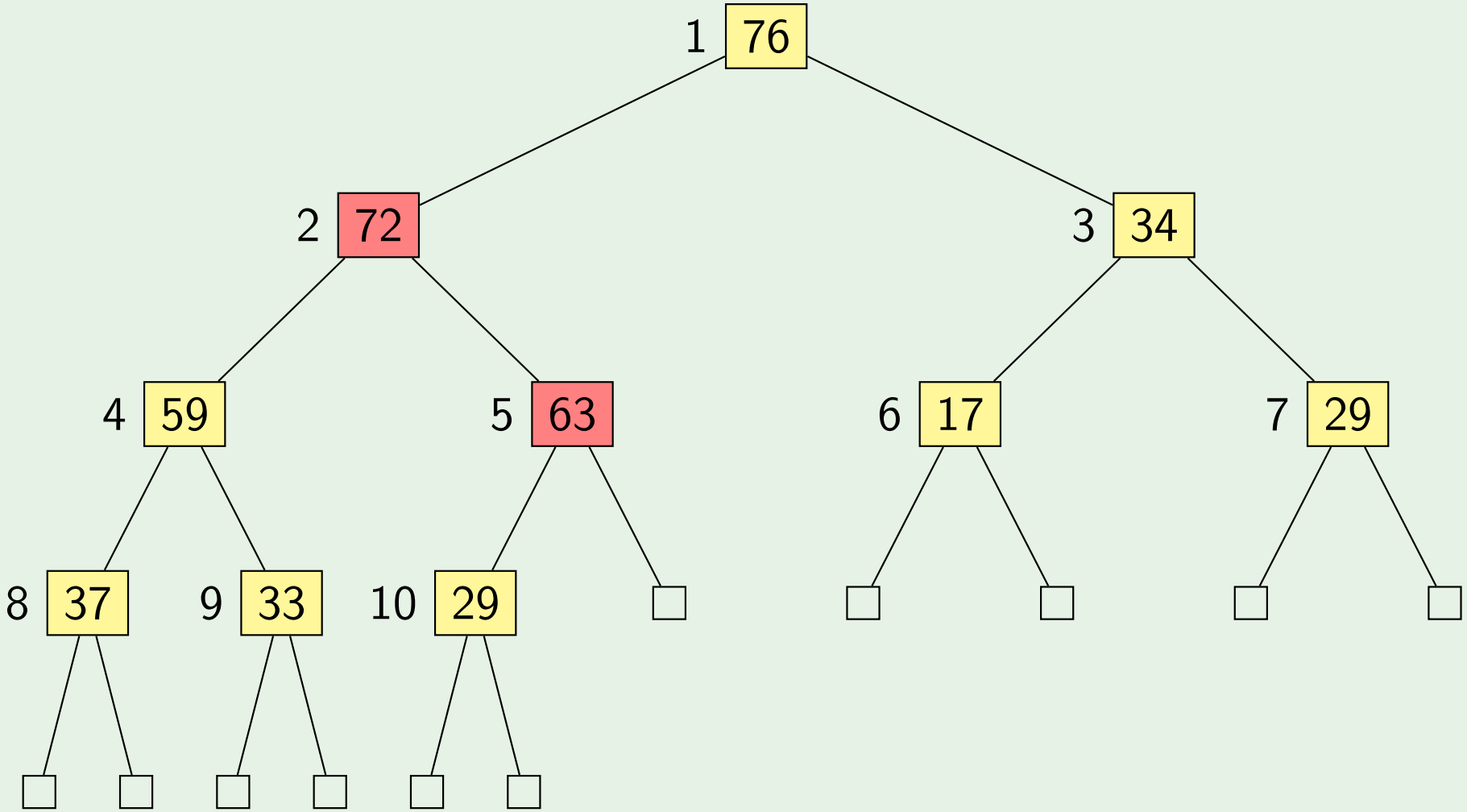
Exemple (Max-heap.)



- **const** max
- **type** heap **is**
  record
    h : **array** [1..max] **of** elem
    n : 0..max
  end record
- **type** cuapr **is** heap
- **type** elem **is**
  record
    prio : $\cdots$
    valor : $\cdots$
  end record

- $A[1]$ conté l'arrel.
- $A[2i]$ conté el fill esquerre de $A[i]$ (si $2i \leqslant n$).
- $A[2i+1]$ conté el fill dret de $A[i]$ (si $2i+1 \leqslant n$)
- $A[i \ \textbf{div} \ 2]$ conté el pare de $A[i]$ (si $i \ \textbf{div} \ 2 \geqslant 1$).
- Els elements del heap són situats en posicions consecutives del vector, amb l'arrel en la primera posició, segons un recorregut de l'arbre per nivells.

Exemple (Max-heap.)



- La implementació de les operacions és molt senzilla.
  - ins Crear un nou node amb l'element a inserir com a nou últim element (segons el recorregut per nivells) per després surar-lo (intercanviar un node amb el seu pare, si aquest és més petit que el seu pare) fins haver reestablert l'ordre del heap.
  - min L'element de prioritat mínima es troba a l'arrel.
  - elim_min Situar l'últim element (segons el recorregut per nivells) en l'arrel, per després enfonsar-lo (intercanviar un node amb el més petit dels seus dos fills, si algun d'aquests és més petit que el seu pare) fins haver reestablert l'ordre del heap.
- Les operacions d'inserció i eliminació es poden implementar amb cost $\Theta(\log n)$.

- **procedure** crear (**sort** Q: cuapr)
  $Q.n := 0$
- **function** min (Q: cuapr)
  if $Q.n = 0$ **then**
  error
  else
  return $Q.h[1]$
- **function** buit (Q: cuapr)
  return $Q.n = 0$

- **procedure** ins (Q: cuapr; x: elem)
  if $Q.n = \max$ **then**
  error
  else
  $Q.n := Q.n + 1$
  $Q.h[Q.n] := x$
  surar$(Q.h, Q.n)$
- **procedure** surar (h: **array** [1..max] **of** elem; j: nat)
  nat $k := j$ **div** 2
  if $k \geqslant 1$ **and** $h[j]$.prio $<$ $h[k]$.prio **then**
  $h[j] \leftrightarrow h[k]$
  surar(h, k)

- **procedure** elim_min (Q: cuapr)
  if $Q.n = 0$ **then**
    error
  else
    $Q.h[1] := Q.h[Q.n]$
    $Q.n := Q.n - 1$
    enfonsar$(Q.h, Q.n, 1)$
- **procedure** enfonsar (h: **array** [1..max] **of** elem; n: nat; j: nat)
  nat $k := 2 \times j$
  if $k < n$ **and** $h[k]$.prio $> h[k+1]$.prio **then**
    $k := k + 1$
  if $k \leqslant n$ **and** $h[j]$.prio $> h[k]$.prio **then**
    $h[j] \leftrightarrow h[k]$
    enfonsar(h, n, k)

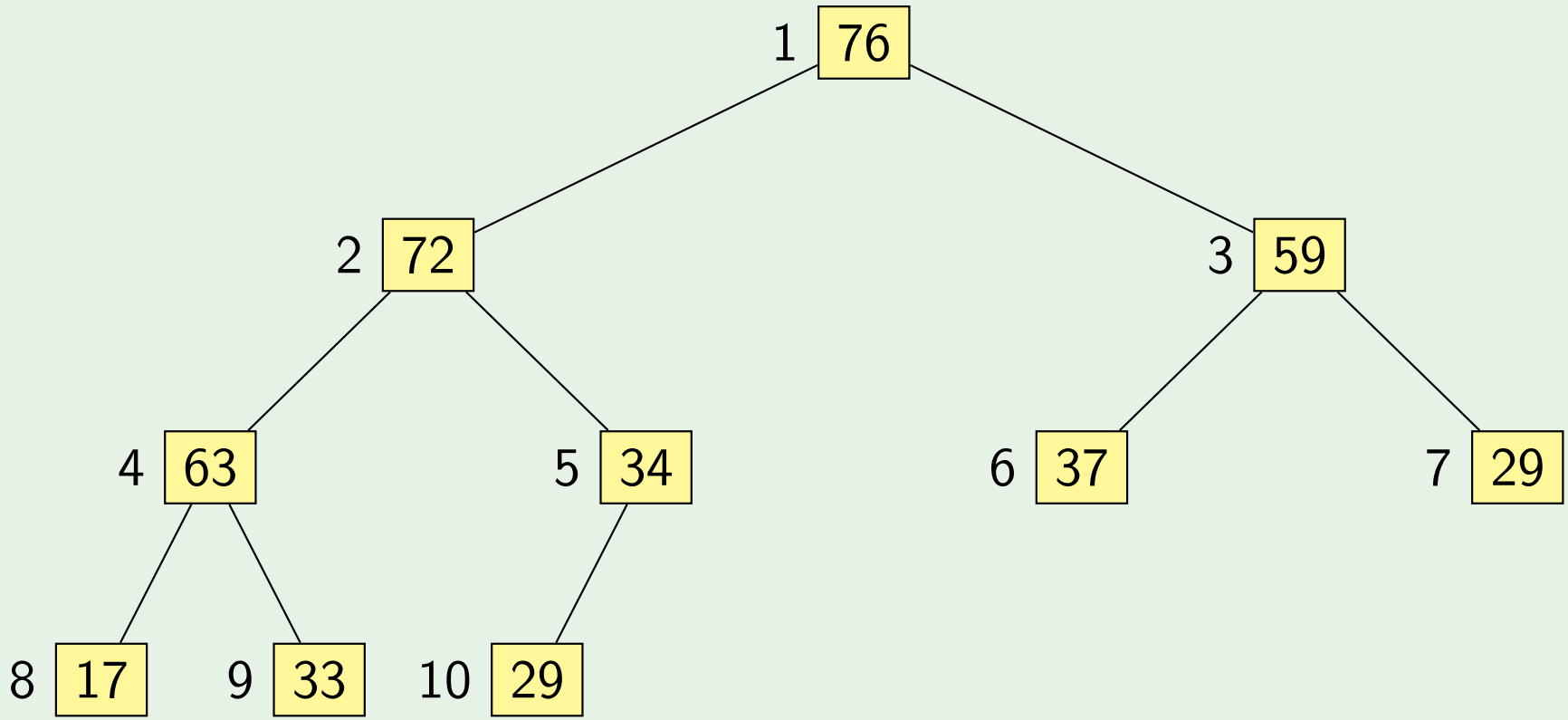
- Una aplicació òbvia de les cues de prioritat és la ordenació.
- ```
  crear(Q)
  for i := 1 to n do
      ins(Q, prio[i], valor[i])
  for i := 1 to n do
      elem x := min(Q); elim_min(Q)
      prio[i] := x.prio
      valor[i] := x.valor
  ```
- La implementació de les cues de prioritat mitjançant heaps permet una ordenació més eficient.
- Heapsort és un algorisme d'ordenació de cost temporal $\Theta(n \log n)$ en el cas pitjor i també en el cas mitjà, sobre un vector de n elements.
- Els factors constants amagats rere la notació asimptòtica són més grans que en l'algorisme quicksort, però també té també l'avantatge d'ordenar sobre el mateix vector, sense necessitat de cap vector auxiliar.
  ```

- **type** heap **is array** [1..max] **of** elem
- **procedure** heapsort (A: heap; n: nat)
  crear_heap(A, n)
  nat $i := n$
  while $i \geqslant 2$ **do**
    $A[1] \leftrightarrow A[i]$
    enfonsar($A, i-1, 1$)
    $i := i - 1$
- Creació top-down del heap inicial (seqüència de n insercions en un heap inicialment buit). $T(n) = \Theta(n \log n)$
- Creació bottom-up del heap inicial. $B(n) = \Theta(n)$
- **procedure** crear_heap (A: heap; n: nat)
  nat $i := n$ **div** 2
  while $i \geqslant 1$ **do**
    enfonsar(A, n, i)
    $i := i - 1$

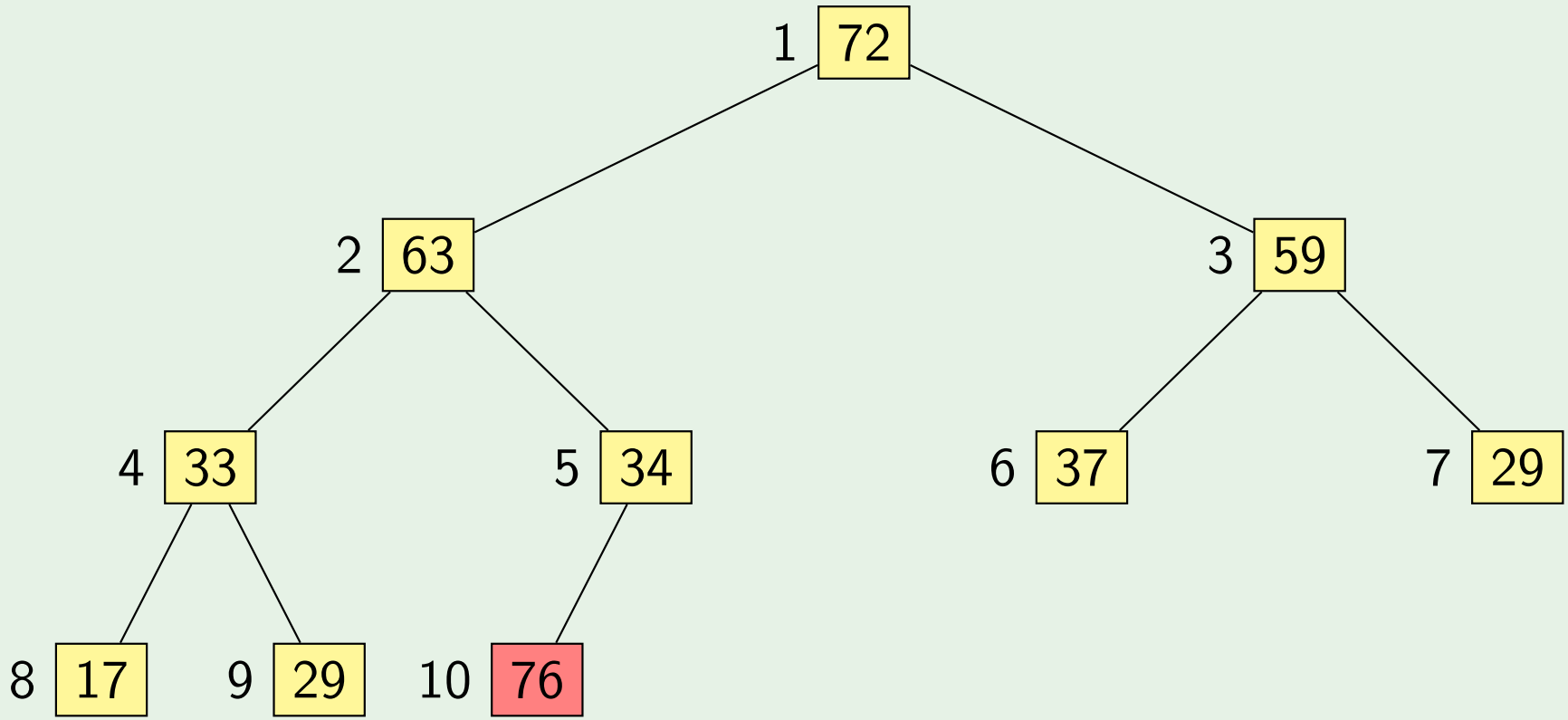
- Creació bottom-up del heap inicial. $B(n) = \Theta(n)$
- **procedure** crear_heap (A: heap; n: nat)
  nat $i := n$ **div** 2
  while $i \geqslant 1$ **do**
    enfonsar(A, n, i)
    $i := i - 1$
- Per a $n = 2^k - 1$,

$$B(n) = \sum_{i=1}^{k} 2^{k-i-1} \cdot i = 2^k - k - 1 = n - k = O(n)\,.$$

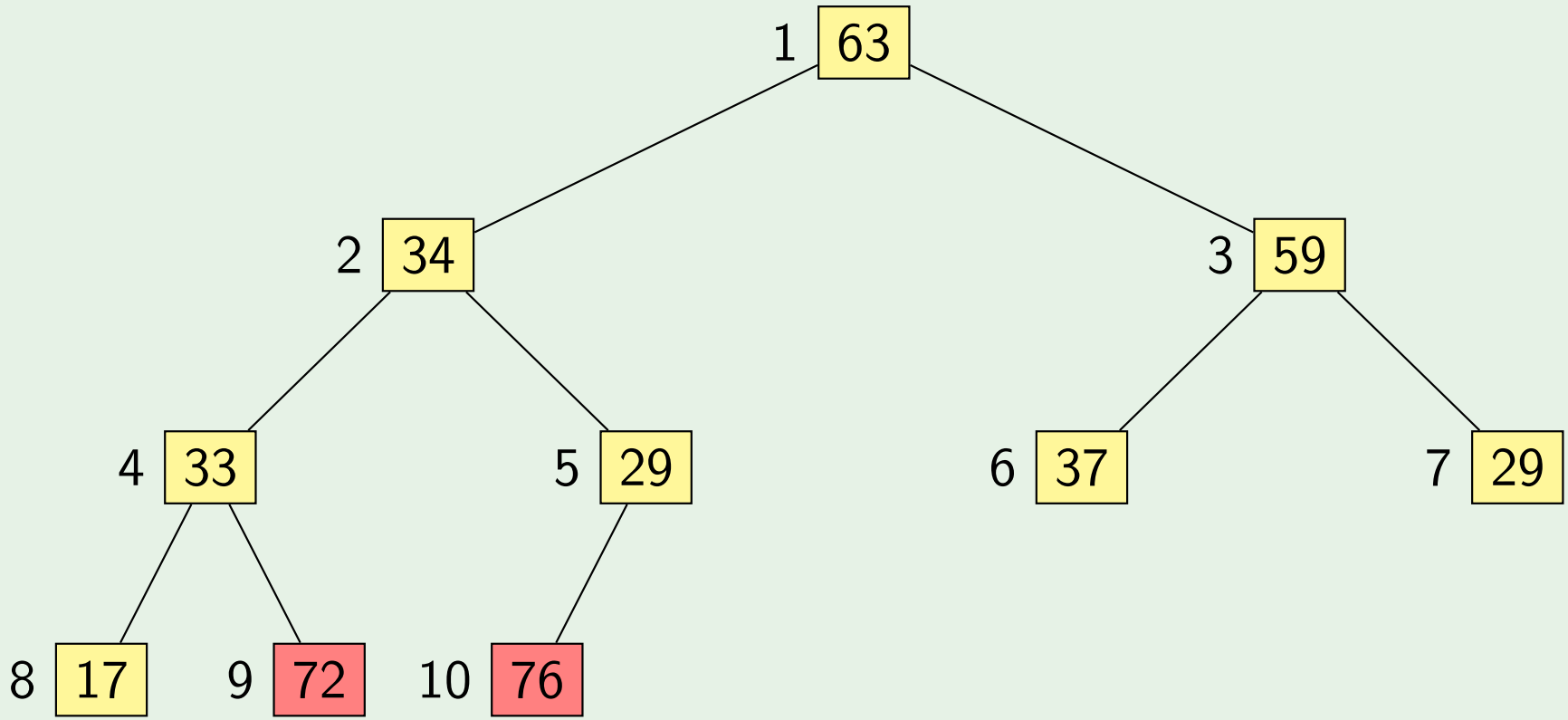
Exemple (Max-heap inicial.)



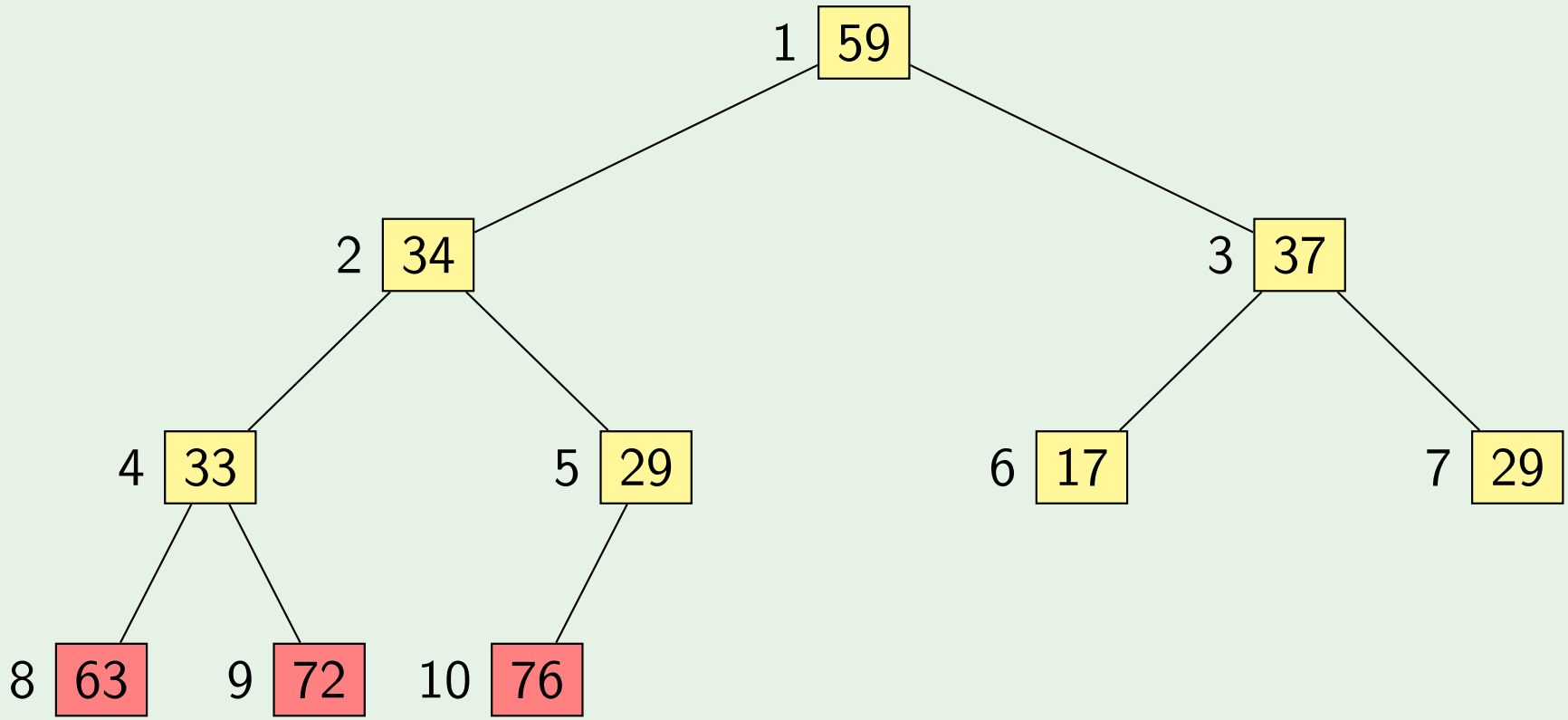
Exemple (Resultat de la iteració amb $i = 10$.)



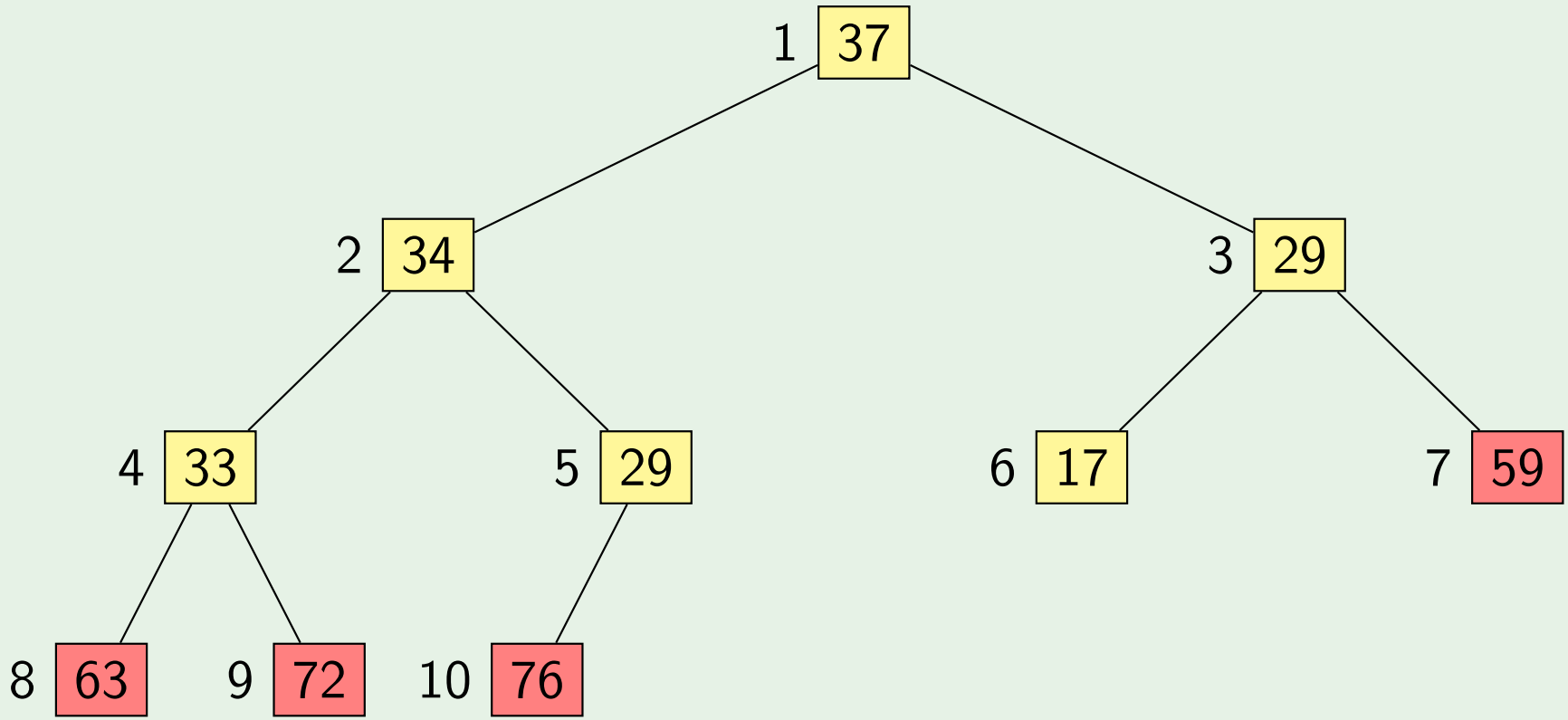
Exemple (Resultat de la iteració amb $i = 9$.)



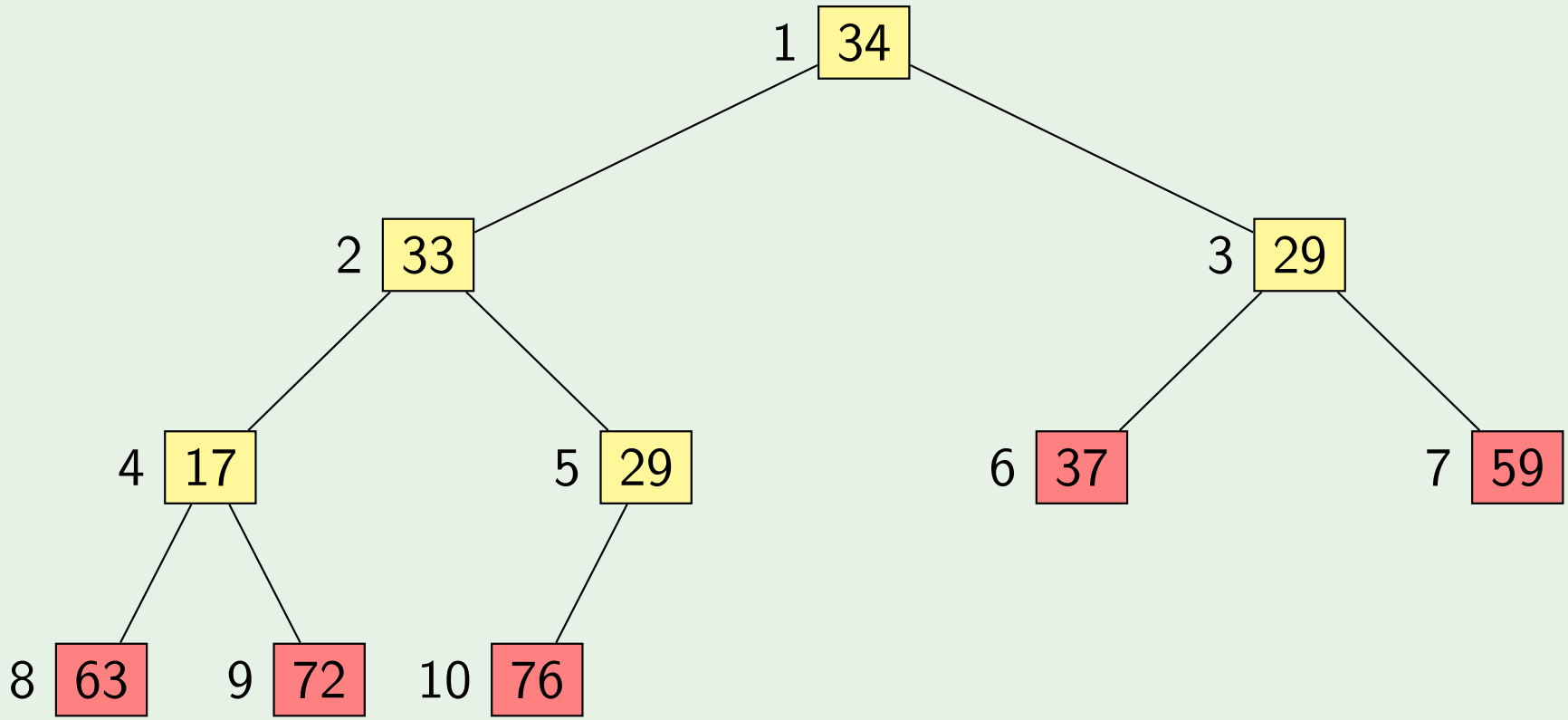
Exemple (Resultat de la iteració amb $i = 8$.)



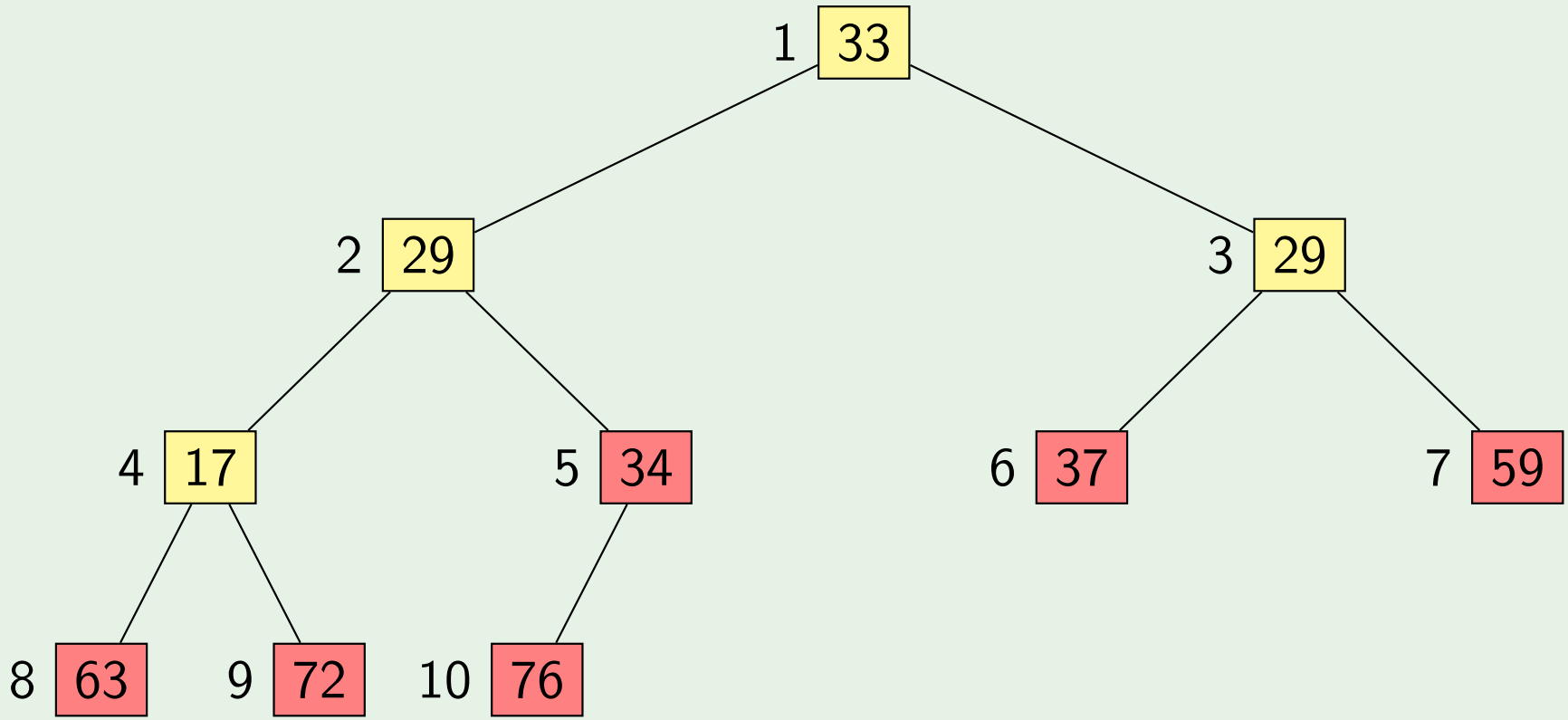
Exemple (Resultat de la iteració amb $i = 7$.)



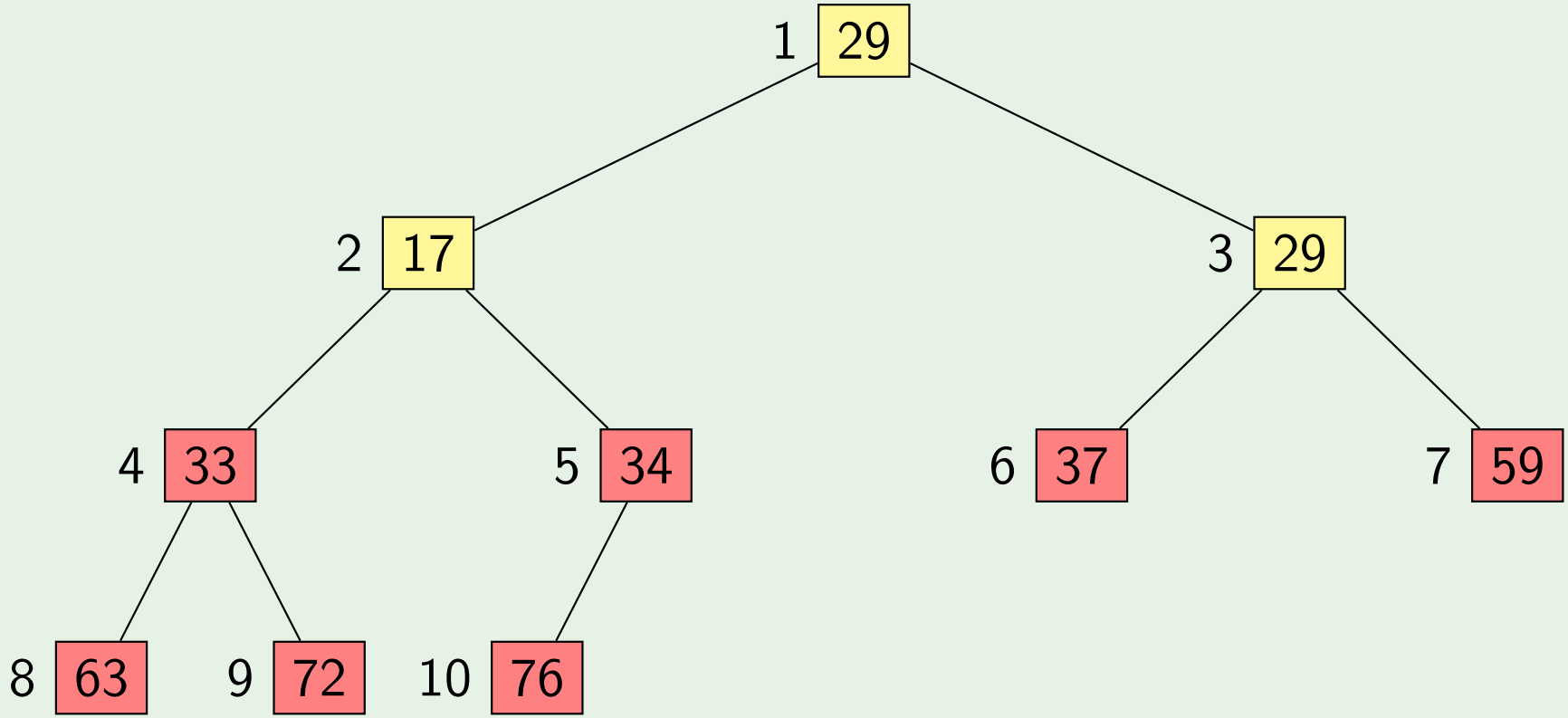
Exemple (Resultat de la iteració amb $i = 6$.)



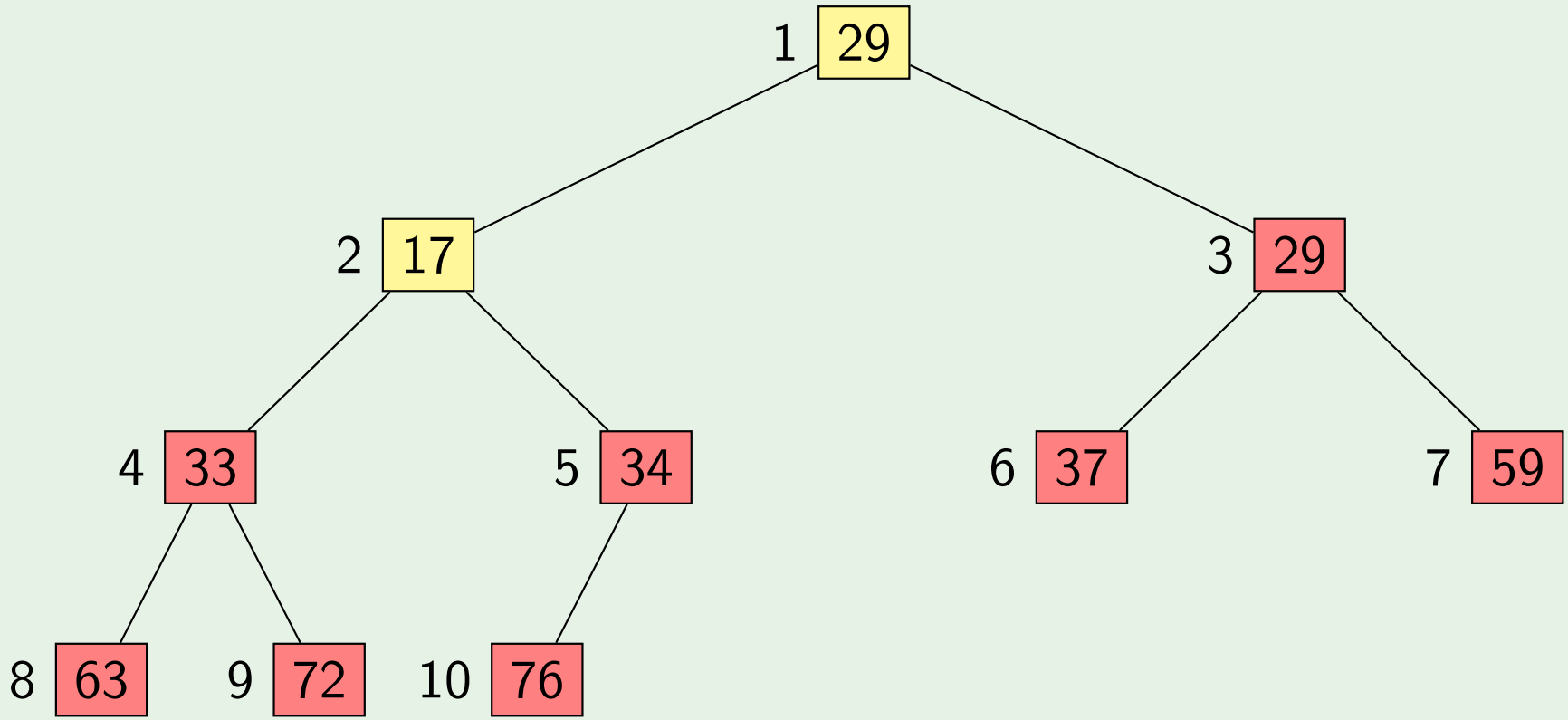
Exemple (Resultat de la iteració amb $i = 5$.)



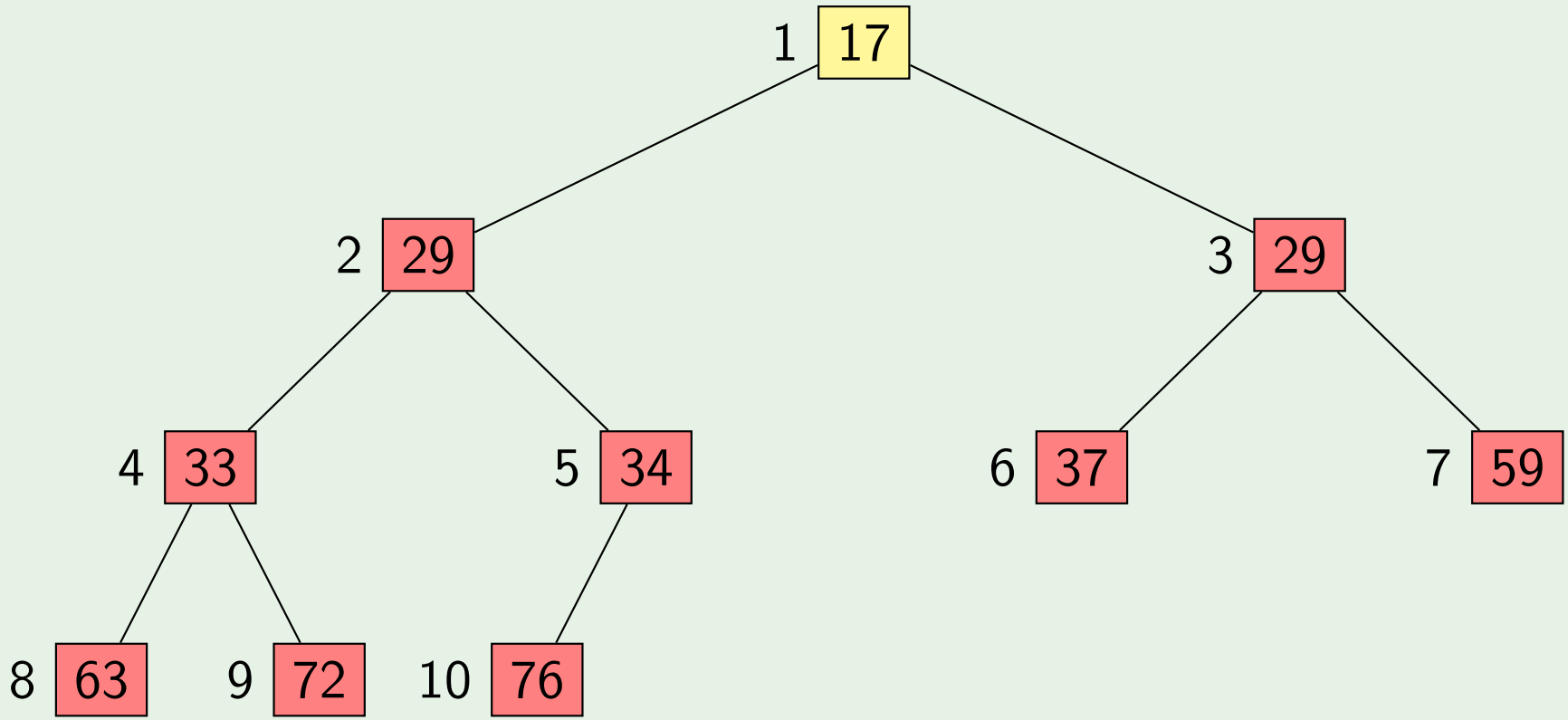
Exemple (Resultat de la iteració amb $i = 4$.)



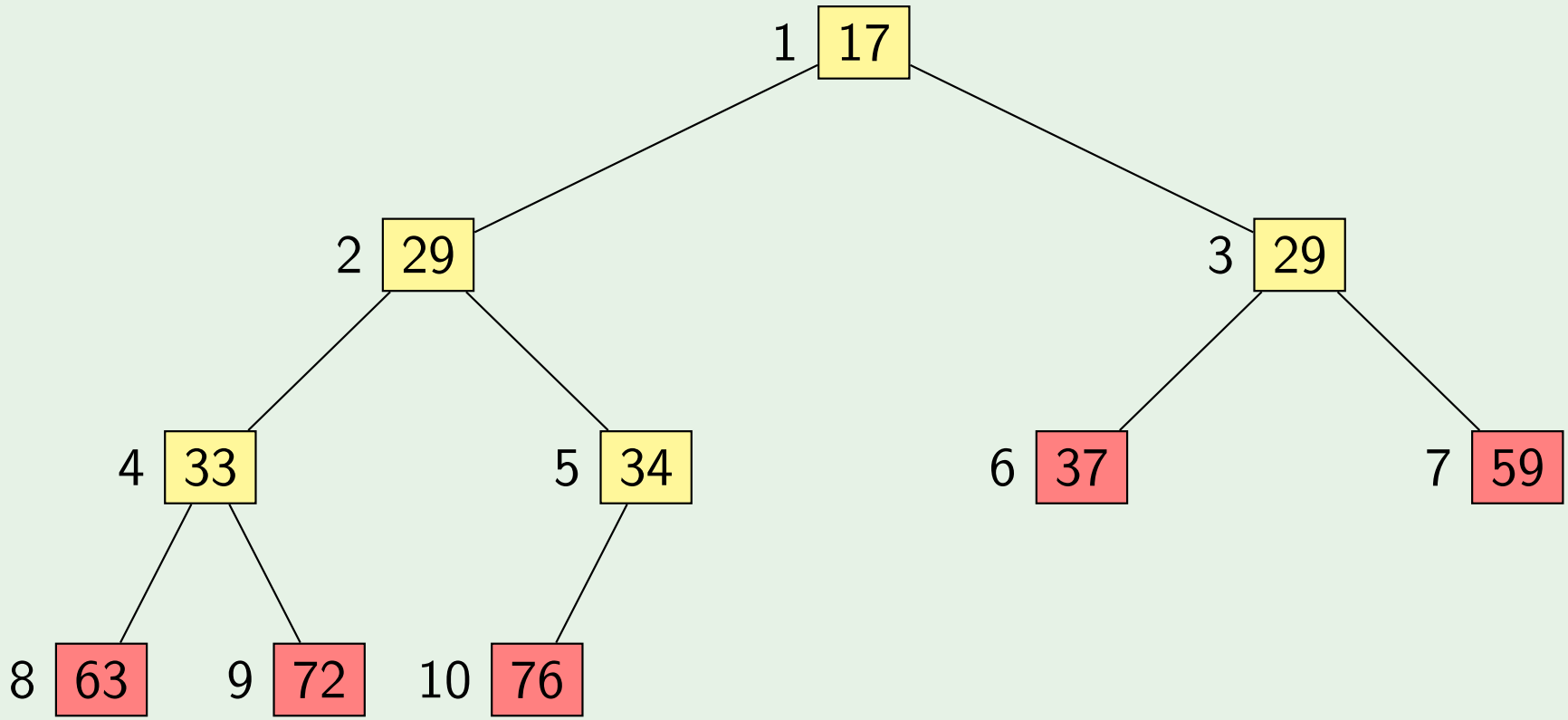
Exemple (Resultat de la iteració amb $i = 3$.)



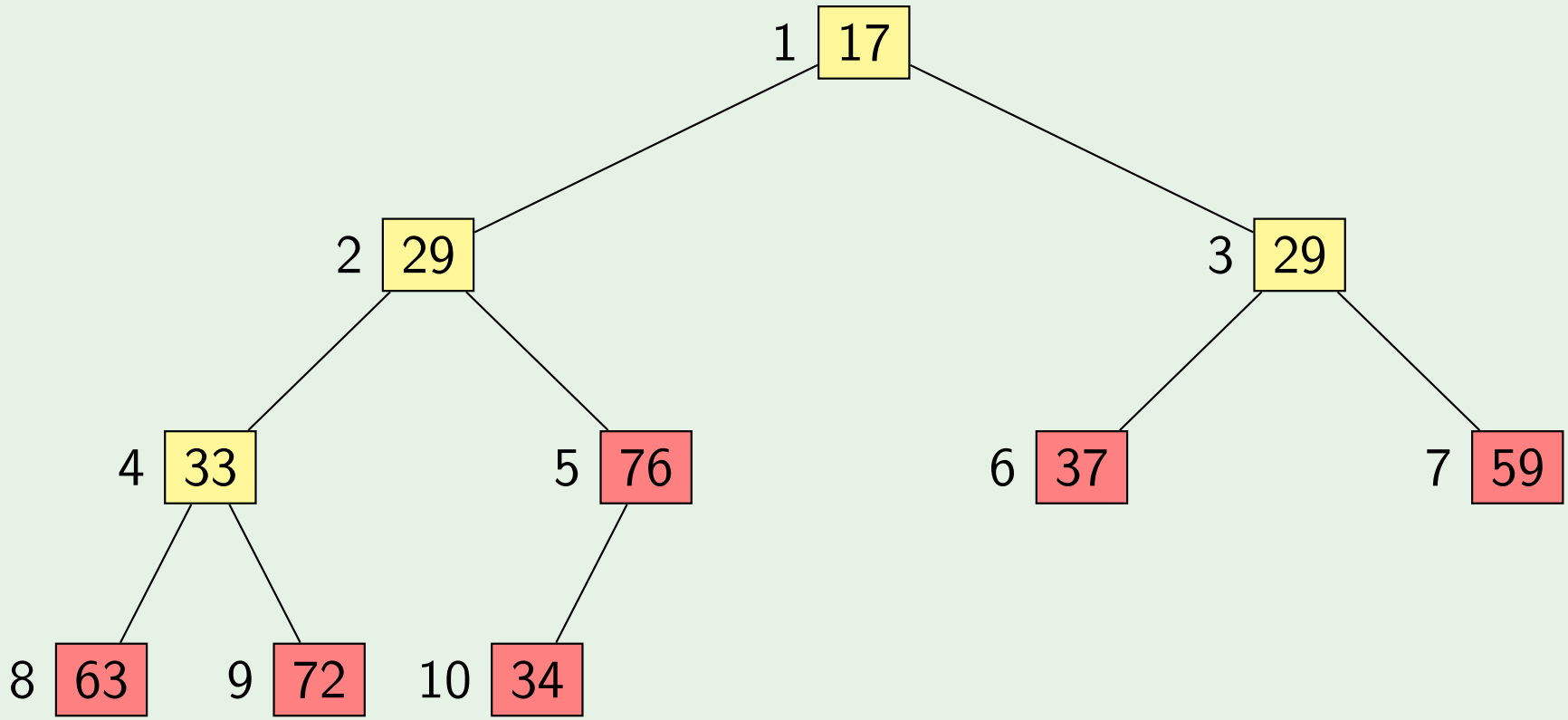
Exemple (Resultat de la iteració amb $i = 2$.)



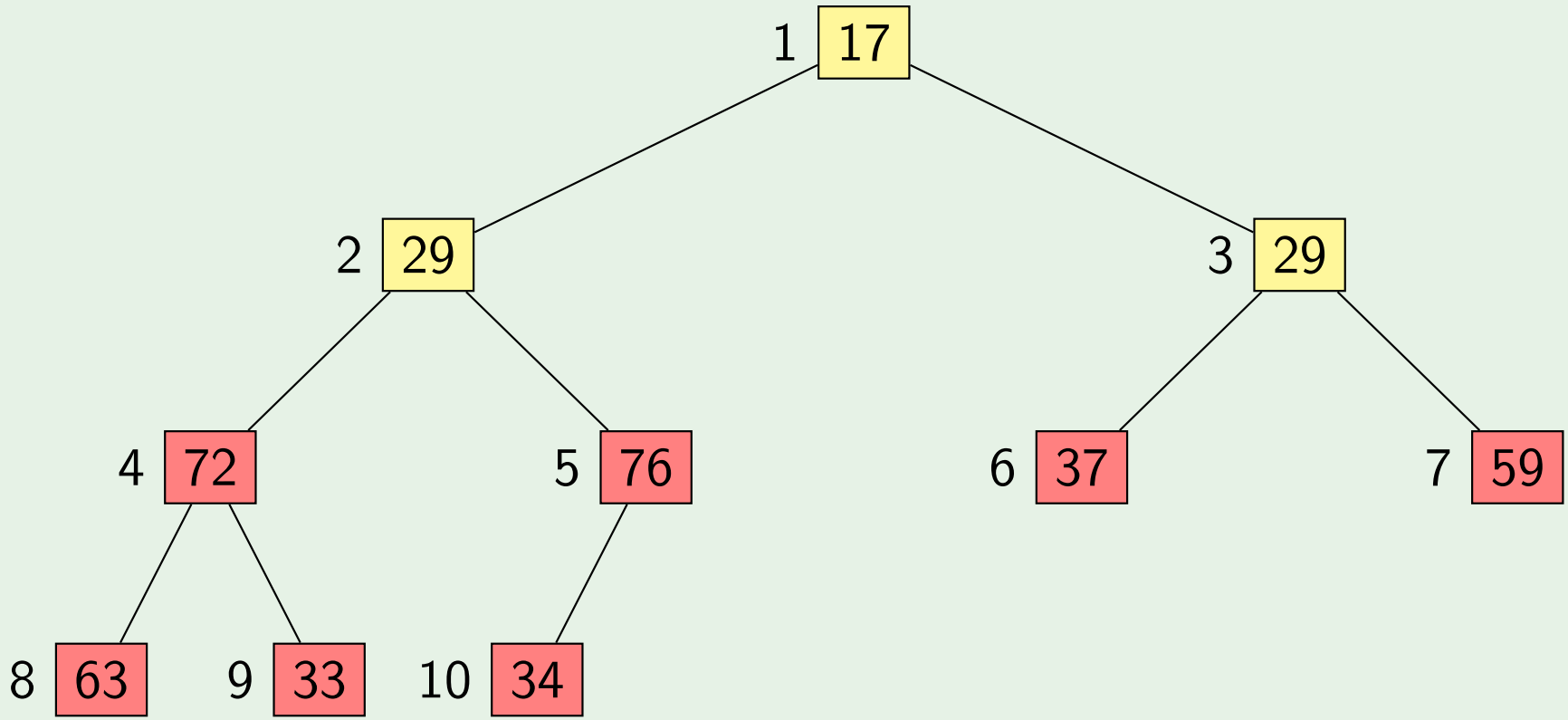
Exemple (Creació bottom-up del max-heap inicial. Els elements indicats satisfan la propietat de heap.)



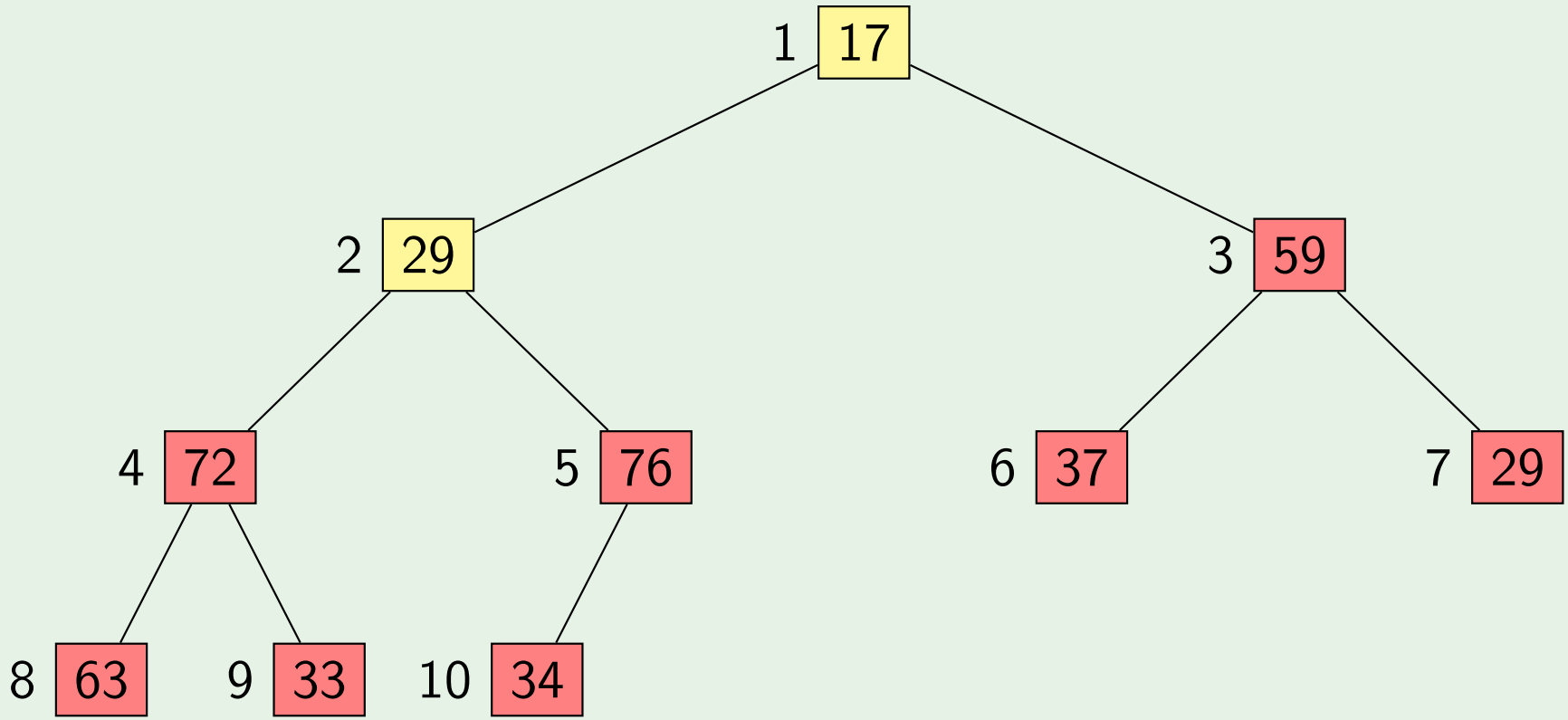
Exemple (Creació bottom-up del max-heap inicial. Resultat de enfonsar($A, 10, 5$).)



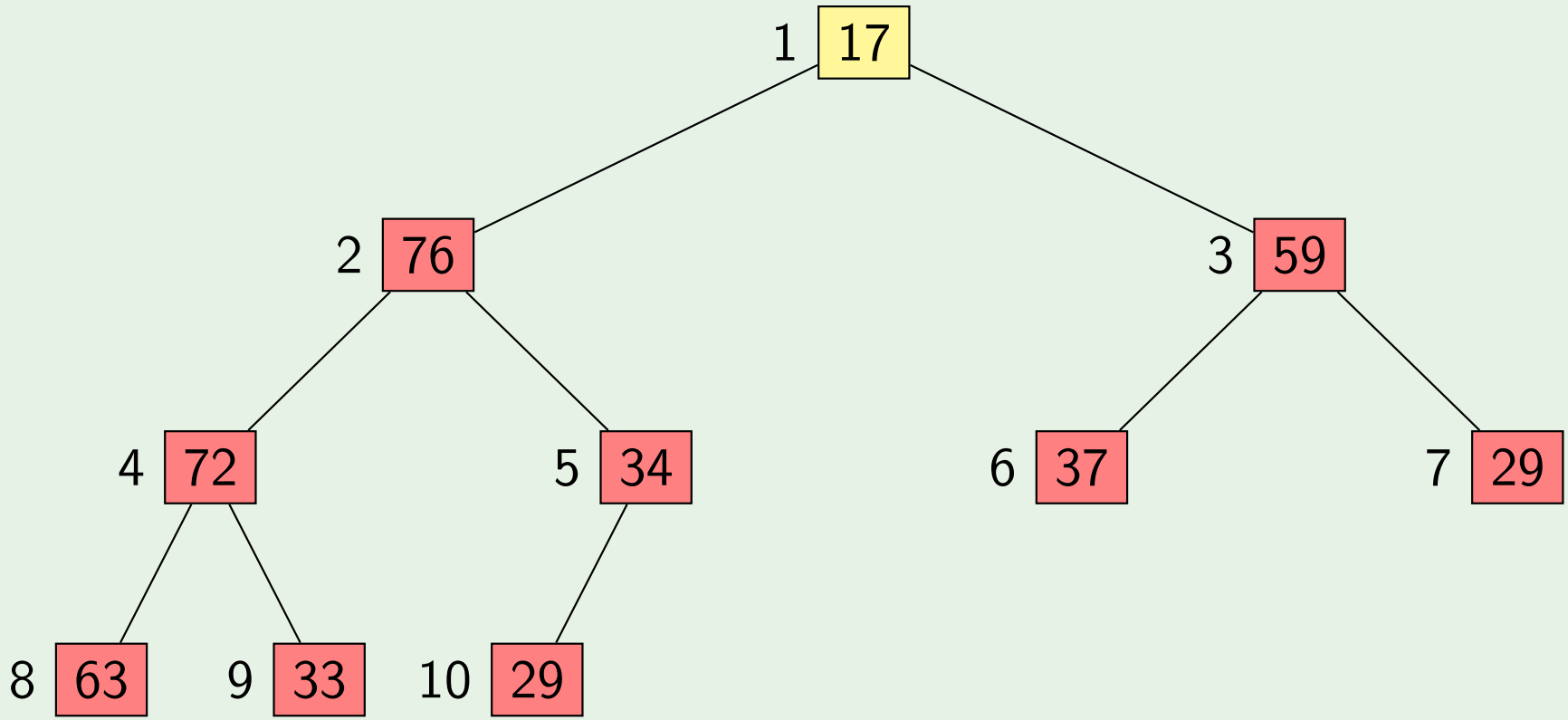
Exemple (Creació bottom-up del max-heap inicial. Resultat de enfonsar($A, 10, 4$).)



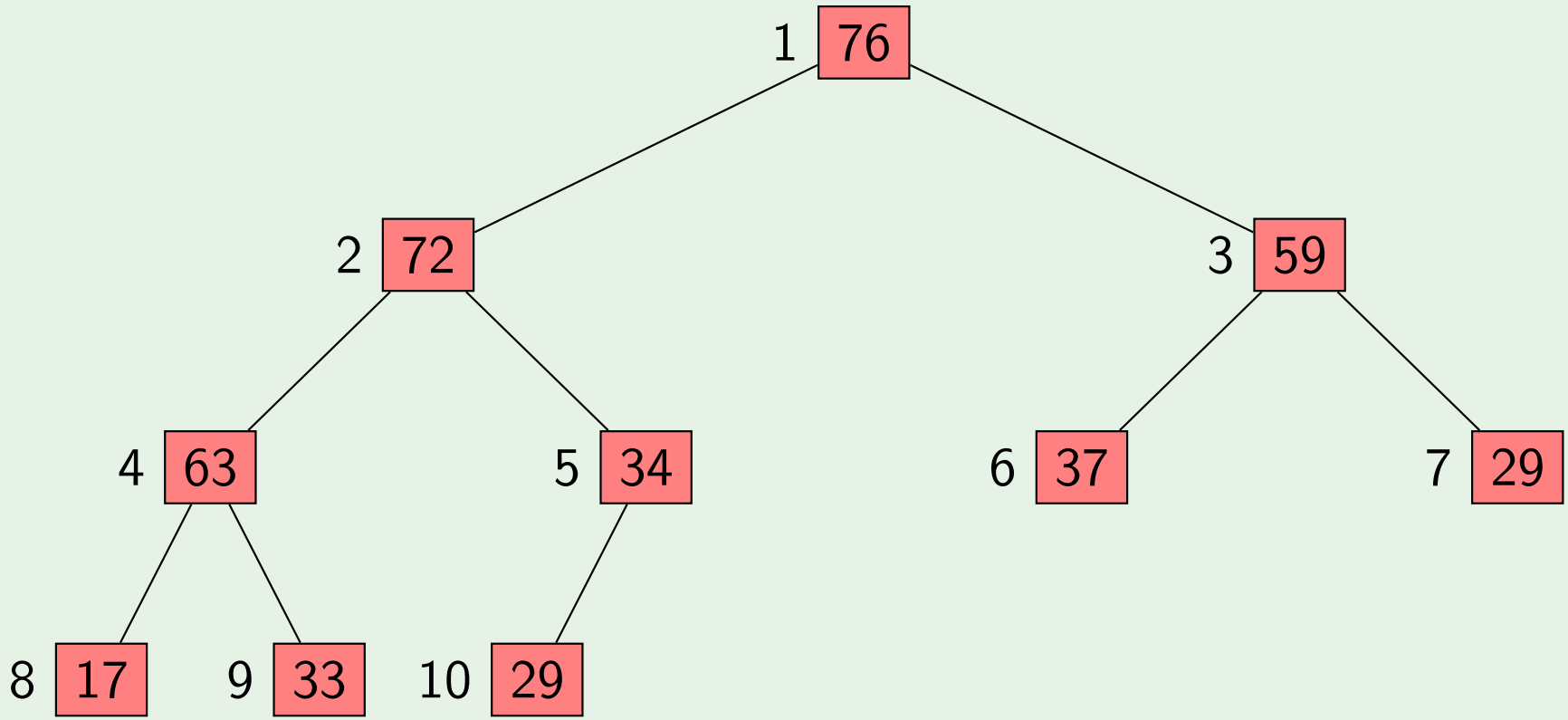
Exemple (Creació bottom-up del max-heap inicial. Resultat de enfonsar($A, 10, 3$).)



Exemple (Creació bottom-up del max-heap inicial. Resultat de enfonsar($A, 10, 2$).)



Exemple (Creació bottom-up del max-heap inicial. Resultat de enfonsar($A, 10, 1$).)

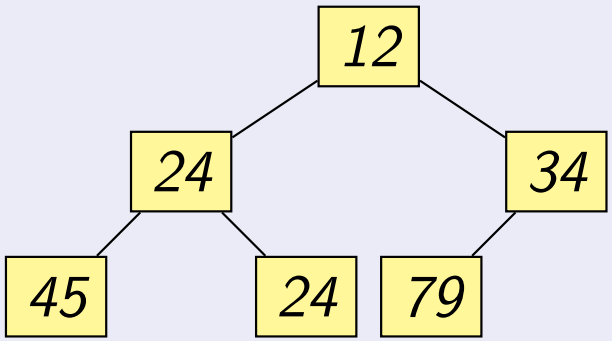


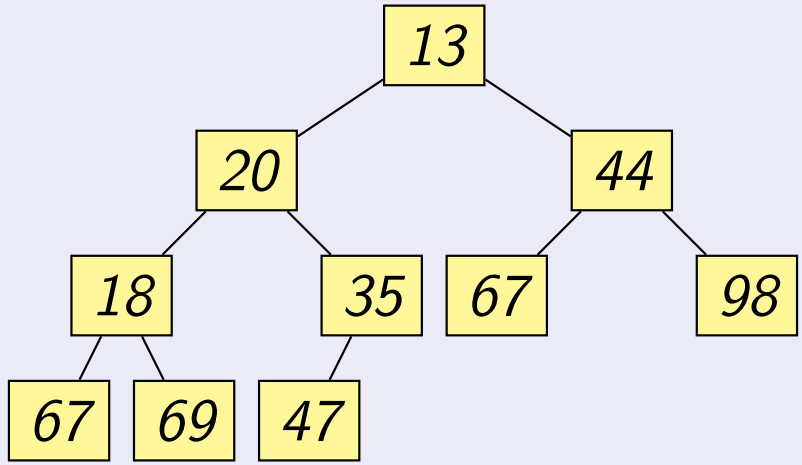
Els problemes marcats amb una estrelleta ($\star$) són més difícils. Si no es diu el contrari i us cal, considereu que tots els logaritmes són en base 2.

- 4.1
- 4.2
- 4.3 (un o dos)
- 4.4 (un o dos)
- 4.5 (un o dos)
- 4.6 (un o dos)
- 4.7
- 4.8 (un o dos)
- 4.9 (un o dos)
- 4.10
- 4.11
- 4.12
- 4.17
- 4.18
- 4.19

Problema (4.1)

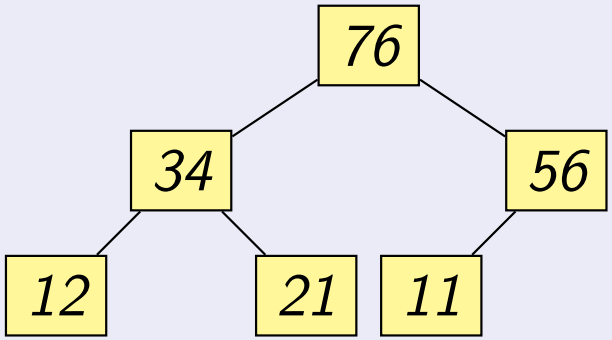
Digueu si els arbres binaris següents són min-heaps o no i perquè.

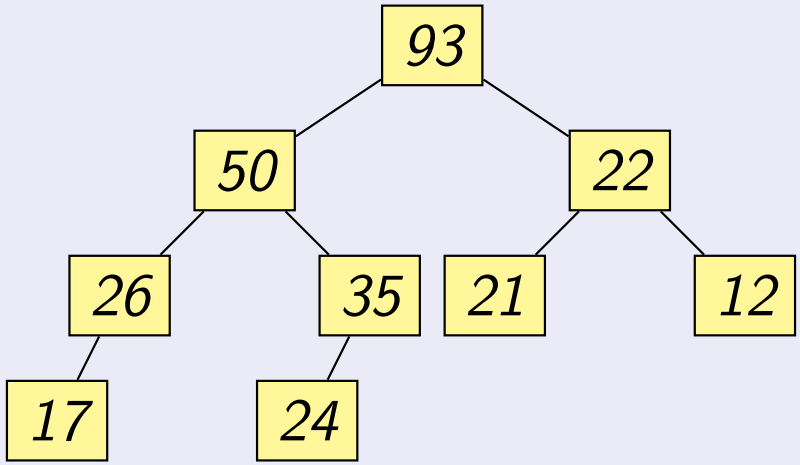




Problema (4.2)

Digueu si els arbres binaris següents són max-heaps o no i perquè.





Problema (4.3)

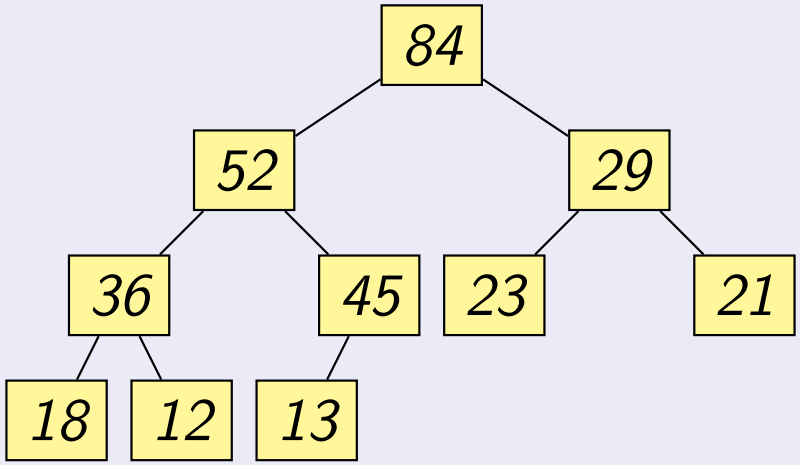
Partint d'un min-heap buit, inseriu successivament les claus 45, 67, 23, 46, 89, 65, 12, 34, 98, 76.

Problema (4.4)

Partint d'un max-heap buit, inseriu successivament les claus 45, 67, 23, 46, 89, 65, 12, 34, 98, 76.

Problema (4.5)

Partint del max-heap següent, elimineu successivament l'element màxim fins que quedi buit.



Problema (4.6)

Donat el min-heap següent implementat en un taula,

7	*11*	*9*	*23*	*41*	*27*	*12*	*29*

dibuixeu l'arbre representat per la taula i, a continuació, dibuixeu l'evolució dels continguts del taula i de l'arbre representat, en aplicar successivament les operacions d'inserir l'element 3 i eliminar el mínim.

Problema (4.7)

Considereu la definició de tipus següent per a arbres binaris:

```
struct node {
  node* fe;
  node* fd;
  int x;
};

typedef arbin node*;
```

Implementeu una funció `bool min_heap (arbin a)` *que indiqui si l'arbre binari* `a` *és un min-heap. Quin és el seu cost?*

Problema (4.8)

Implementeu una funció `bool max_heap (arbin a)` *que indiqui si l'arbre binari* `a` *és un max-heap. Quin és el seu cost?*

Problema (4.9)

Convertiu la taula següent en un min-heap tot aplicant l'algorisme de construcció de heaps de dalt cap a baix.

45	*53*	*27*	*21*	*11*	*97*	*34*	*78*

Problema (4.10)

Convertiu la taula següent en un min-heap tot aplicant l'algorisme de construcció de heaps de baix cap a dalt.

45	*53*	*27*	*21*	*11*	*97*	*34*	*78*

Problema (4.11)

Valideu o desmentiu les afirmacions següents:

1. *"Una taula amb els elements ordenats de menor a major és un min-heap."*
2. *"Inserir en un max-heap amb n elements té cost $\Theta(\log n)$ en el cas pitjor."*
3. *"Trobar l'element màxim en un min-heap té cost $O(\sqrt{n})$."*
4. *"Eliminar el màxim d'una cua de prioritats de n elements amb prioritats diferents en un max-heap en taula té cost $\Theta(1)$ en el cas millor."*

Problema (4.12)

En un heap k-ari amb $k \geqslant 2$, tot node té com a màxim k fills (els heaps habituals són doncs heaps 2-aris). Doneu una representació espacialment eficient d'un heap k-ari en una taula, tot descrivint la implementació de les operacions que permeten anar d'un element al seu pare i els seus k fills.

Problema (4.17)

És l'algorisme heapsort estable?

Problema (4.18)

La STL requereix que l'operació `sort()` *ha de tenir cost* $O(n \log n)$ *en el cas pitjor i ha de ser in-situ. En canvi, l'operació* `stable_sort()` *ha de tenir cost* $O(n \log n)$ *en el cas pitjor i ha de ser estable. Quins algorismes d'ordenació podeu utilitzar per implementar cadascuna d'aquestes operacions?*

Busqueu quin algorisme d'ordenació utilitza la implementació de l'algorisme `sort` *i de* `stable_sort` *de la STL al vostre ordinador.*

Problema (4.19)

Expliqueu per què no se solen implementar les cues de prioritat amb arbres equilibrats.