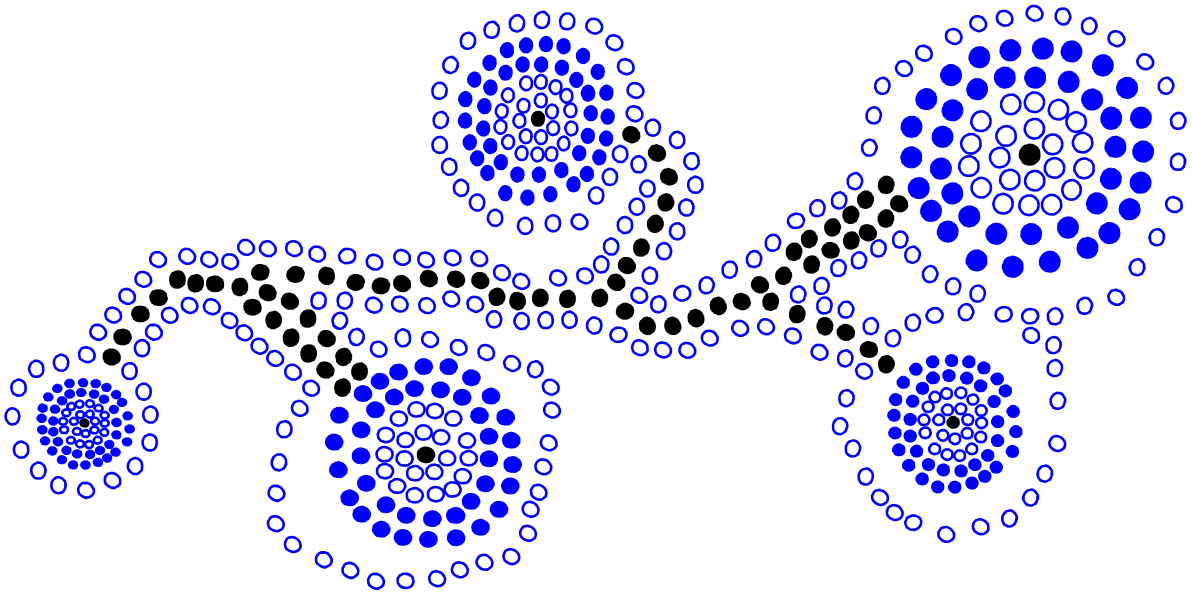
FILES À PRIORITÉ

- Application boursière (motivation)
- Le TAD file à priorité (*Priority Queue*)
- Réalisation d'une file à priorité avec une séquence
- Le tri (*sorting*)
- Problèmes liés au tri



Application boursière

- Nous nous concentrerons sur la vente d'un seul titre, Akamai Technologies, fondée en 1998 par des professeurs et des étudiants du MIT (200 employés, 20 milliards de dollars en capital action)
- Les investisseurs font des ***commandes*** qui comprennent trois items (***action***, ***prix***, ***quantité***), où ***action*** est un ***achat*** ou une ***vente***, ***prix*** est le pire prix que vous êtes prêt à débourser (achat) ou à accepter (vente), et ***quantité*** est le nombre d'actions
- À l'équilibre, toutes les commandes d'achat (***offres***) ont des prix plus bas que toutes les commandes de ventes (***demandes***)
- Une ***cote de niveau 1*** donne l'offre la plus haute et la demande la plus basse (telles que fournies par les sites financiers populaires et les courtiers ou *e-brokers*)
- Une ***cote de niveau 2*** donne toutes les offres et les demandes pour certains seuils de prix (Island ECN sur le Web et cotes pour agents professionnels (*traders*))
- Une ***transaction*** survient lorsqu'une nouvelle commande peut être jumelée à une ou plusieurs commandes existantes, ce qui résulte en une série de transactions de ***suppression***.
- Les commandes peuvent être ***annulées*** à tout moment.

Structures de données pour le marché boursier

- Pour chaque titre, nous conservons deux structures, la première pour les offres et la seconde pour les demandes
- Les opérations qui doivent être supportées:

Action	***Structure Offre***	***Structure Demande***
faire une commande	***insert***(prix,quantité)	***insert***(prix,quantité)
obtenir une cote de niveau 1	***min***()	***max***()
effectuer la transaction	***removeMin***()	***removeMax***()
annuler	***remove***(commande)	***remove***(commande)

- Ces structures de données sont appelées ***files à priorité***.
- Les files à priorité de la bourse NASDAQ supportent en moyenne un volume de transaction quotidien de 1 milliard d'actions (50 milliards de dollars)

Clés et relations d'ordre total

- Une file à priorité (*Priority Queue*) classe ses éléments par ***clé*** avec une relation ***d'ordre total***

- Clés:
  - Chaque élément a sa propre clé
  - Les clés ne sont pas nécessairement uniques

- Relation d'ordre total
  - Dénotée par $\leq$
  - **Réflexive:** $k \leq k$
  - **Antisymétrique:** si $k_1 \leq k_2$ et $k_2 \leq k_1$, alors $k_1 \leq k_2$
  - **Transitive:** si $k_1 \leq k_2$ et $k_2 \leq k_3$, alors $k_1 \leq k_3$

- Une file à priorité supporte ces méthodes fondamentales sur des paires clé-élément:
  - min()
  - insertItem(k, e)
  - removeMin()

Tri par file à priorité

- Une file à priorité P peut être utilisée pour trier une séquence S:
  - en insérant les éléments de S dans P avec une suite d'opérations insertItem(e, e)
  - en retirant les éléments de P en ordre croissant et en les remettant dans S avec une suite d'opérations removeMin()

Algorithm PriorityQueueSort(S, P):
Entrée: Séquence S contenant n éléments, avec une relation d'ordre total, et une file à priorité P qui compare les clés avec cette même relation
Sortie: Séquence S triée à l'aide de la relation d'ordre total

while !S.isEmpty() **do**
  $e \leftarrow S$.removeFirst()
  P.insertItem(e, e)
while P is not empty **do**
  $e \leftarrow P$.removeMin()
  S.insertLast(e)

Le TAD File à priorité

- Une file à priorité *P* supporte les méthodes suivantes:

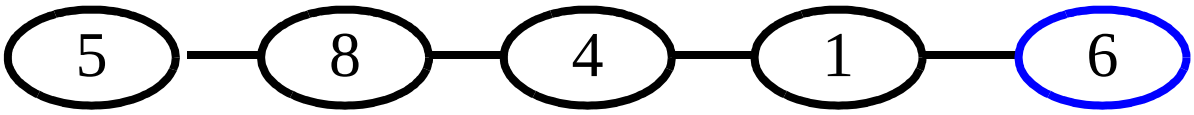
  - size():
    Retourne le nombre d'éléments dans *P*
  - isEmpty():
    Vérifie si *P* est vide
  - insertItem(*k*,*e*):
    Insère un nouvel élément *e* avec sa clé *k* dans *P*
  - minElement():
    Retourne (mais ne retire pas) un élément de *P* à la plus petite clé; une erreur survient si *P* est vide
  - minKey():
    Retourne la plus petite clé de *P*; une erreur survient si *P* est vide
  - removeMin():
    Retire et retourne un élément de *P* à la plus petite clé; une erreur survient si *P* est vide.

Comparateurs

- Patron de conception (*Comparator*)
- La forme la plus générale et la plus réutilisable de file à priorité utilise des objets appelés **comparateurs.**
- Les comparateurs sont externes aux clés à comparer et permettent de comparer deux objets.
- Quand la file à priorité a besoin de comparer deux clés, elle utilise le comparateur qui lui a été fourni.
- Ainsi, une file à priorité peut être suffisamment générale pour contenir n'importe quel objet.
- Le TAD **Comparateur** inclut:
  - isLessThan(*a*, *b*)
  - isLessThanOrEqualTo(*a*,*b*)
  - isEqualTo(*a*, *b*)
  - isGreaterThan(*a*,*b*)
  - isGreaterThanOrEqualTo(*a*,*b*)
  - isComparable(*a*)

Réalisation avec séquence non-triée

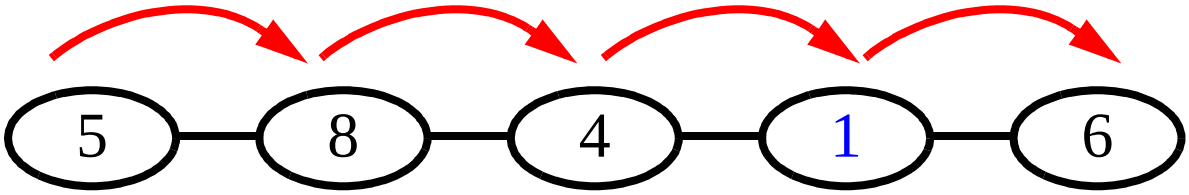
- Essayons de réaliser une file à priorité avec une séquence non-triée *S*.
- Les éléments de *S* sont composés de *k*, la clé, et de *e*, l'élément.
- Nous pouvons réaliser insertItem() en utilisant insertLast() sur les séquences. Le temps d'exécution sera alors ***O***(1).



- Cependant, comme nous insérons toujours à la fin, sans tenir compte de la valeur de la clé, notre séquence n'est pas ordonnée.

Réalisation avec séquence non-triée (suite)

- Ainsi, pour les méthodes telles minElement(), minKey(), et removeMin(), nous devons ***regarder tous les éléments*** de *S*. La complexité du pire des cas est $\boldsymbol{O}(n)$.



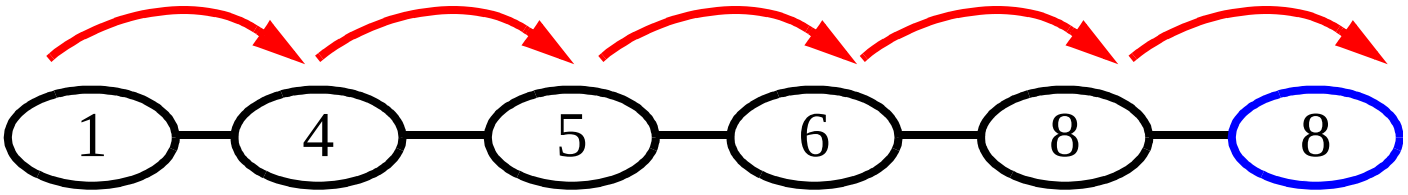
- Sommaire des performances

insertItem	$\boldsymbol{O}(1)$
minKey, minElement	$\boldsymbol{O}(n)$
removeMin	$\boldsymbol{O}(n)$

Réalisation avec séquence triée

- Une autre réalisation possible utilise une séquence *S*, triée par ordre croissant de clés.
- minElement(), minKey(), et removeMin() deviennent alors ***O***(1)

- Cependant, pour réaliser insertItem(), nous devons maintenant parcourir la séquence entière ***dans le pire des cas***. Ainsi, insertItem() s'exécute en un temps $O(n)$



- Sommaire des performances

insertItem	***O*(*n*)**
minKey, minElement	***O*(1)**
removeMin	***O*(1)**

Réalisation avec séquence triée (suite)

```
public class SequenceSimplePriorityQueue
implements SimplePriorityQueue {
   //Implementation of a priority queue
   using a sorted sequence
  protected Sequence seq = new NodeSequence();
  protected Comparator comp;


// auxiliary methods
  protected Object key (Position pos) {
     return ((Item)pos.element()).key();
   }// note casting here


  protected Object element (Position pos) {
     return ((Item)pos.element()).element();
   }// casting here too


   // methods of the SimplePriorityQueue ADT
  public SequenceSimplePriorityQueue (Comparator c) {
     comp = c; }
  public int size () {return seq.size(); }
```

...suite à la page suivante...

Réalisation avec séquence triée (suite)

```
public void insertItem (Object k, Object e) throws
InvalidKeyException {
    if (!comp.isComparable(k)) {
       throw new InvalidKeyException("The key is not valid");
    }
    else {
       if (seq.isEmpty()) {
          //if the sequence is empty, this is the
          seq.insertFirst(new Item(k,e));//first item
       }
       else { //check if it fits right at the end
          if (comp.isGreaterThan(k,key(seq.last()))) {
             seq.insertAfter(seq.last(),new Item(k,e));
          }
          else {
             //we have to find the right place for k.
              Position curr = seq.first();
             while (comp.isGreaterThan(k,key(curr))) {
                curr = seq.after(curr);
             }
             seq.insertBefore(curr,new Item(k,e));
          }
       }
 }
```

...suite à la page suivante...

Réalisation avec séquence triée (suite)

```
    public Object minElement () throws
    EmptyContainerException {
    if (seq.isEmpty()) {
          throw new EmptyContainerException("The priority
          queue is empty");
    }
    else {
          return element(seq.first());
    }



    public boolean isEmpty () {
          return seq.isEmpty();
    }
}
```

Tri par sélection

- Le tri par sélection est une variation du tri par file à priorité (*PriorityQueueSort*) qui utilise une ***séquence non-triée*** pour réaliser la file à priorité *P*.
- **Phase 1**, l'insertion d'un item dans *P* est ***O***(1)
- **Phase 2**, le retrait d'un item de *P* prend un temps proportionnel au nombre d'éléments présents dans *P*

	Séquence *S*	File à priorité *P*
Entrée	(7, 4, 8, 2, 5, 3, 9)	()
Phase 1:		
(a)	(4, 8, 2, 5, 3, 9)	(7)
(b)	(8, 2, 5, 3, 9)	(7, 4)
...	...	...
(g)	()	(7, 4, 8, 2, 5, 3, ,9)
Phase 2:		
(a)	(2)	(7, 4, 8, 5, 3, 9)
(b)	(2, 3)	(7, 4, 8, 5, 9)
(c)	(2, 3, 4)	(7, 8, 5, 9)
(d)	(2, 3, 4, 5)	(7, 8, 9)
(e)	(2, 3, 4, 5, 7)	(8, 9)
(f)	(2, 3, 4, 5, 7, 8)	(9)
(g)	(2, 3, 4, 5, 7, 8, 9)	()

Tri par sélection (suite)

- Comme vous pouvez le constater, la phase 2 est le goulot d'étranglement. La première opération removeMin est $\boldsymbol{O}(n)$, la seconde $\boldsymbol{O}(n-1)$, et ainsi de suite jusqu'à la dernière, qui est $\boldsymbol{O}(1)$.
- Le temps total nécessaire à la phase 2 est:

$$O(n + (n-1) + \ldots + 2 + 1) \equiv O\left(\sum_{i=1}^{n} i\right)$$

- Et comme:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

- Le temps d'exécution de la phase 2 est donc $\boldsymbol{O}(n^2)$. Ainsi, la complexité temporelle de l'algorithme est $\boldsymbol{O}(n^2)$.

Tri par insertion

- Le tri par insertion résulte de l'utilisation d'un tri par file à priorité où la file est réalisée avec ***séquence triée***.

	Séquence *S*	File à priorité *P*
Entrée	(7, 4, 8, 2, 5, 3, 9)	()
Phase 1:		
(a)	(4, 8, 2, 5, 3, 9)	(7)
(b)	(8, 2, 5, 3, 9)	(4, 7)
(c)	(2, 5, 3, 9)	(4, 7, 8)
(d)	(5, 3, 9)	(2, 4, 7, 8)
(e)	(3, 9)	(2, 4, 5, 7, 8)
(f)	(9)	(2, 3, 4, 5, 7, 8)
(g)	()	(2, 3, 4, 5, 7, 8, 9)
Phase 2:		
(a)	(2)	(3, 4, 5, 7, 8, 9)
(b)	(2, 3)	(4, 5, 7, 8, 9)
...	...	...
(g)	(2, 3, 4, 5, 7, 8, 9)	()

Tri par insertion (suite)

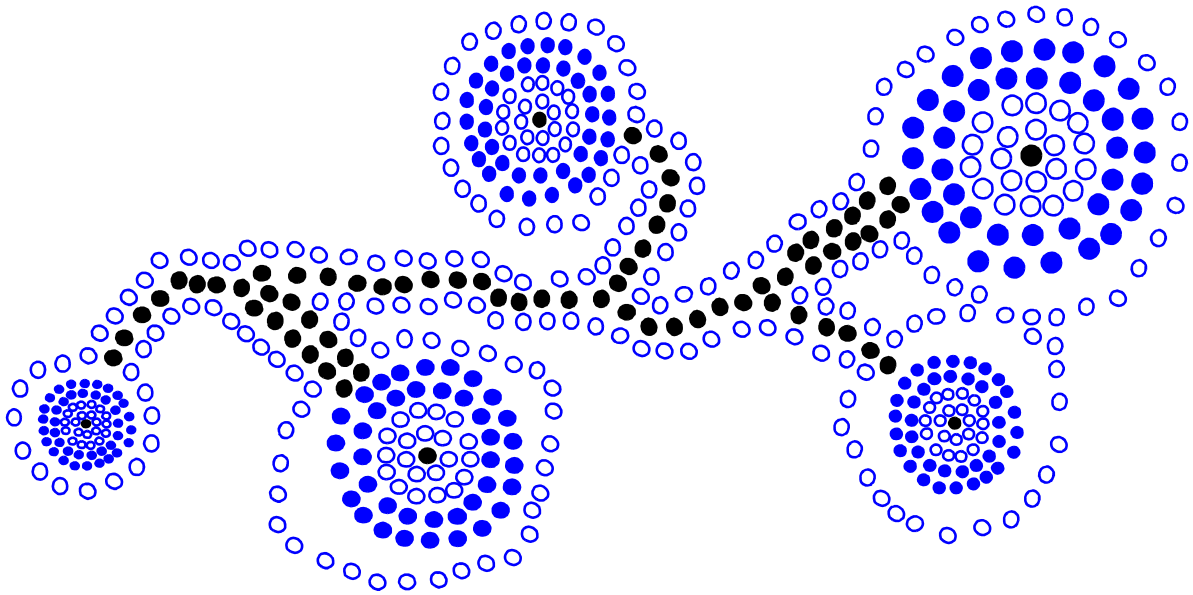
- Nous améliorons ainsi la phase 2, qui est $\boldsymbol{O}(n)$.
- Cependant, la phase 1 devient maintenant le goulot d'étranglement. Le premier insertItem est $\boldsymbol{O}(1)$, le second $\boldsymbol{O}(2)$, jusqu'au dernier qui lui est $\boldsymbol{O}(n)$, pour un temps d'exécution total $\boldsymbol{O}(n^2)$
- Le tri par sélection et le tri par insertion ont tous deux un temps d'exécution $\boldsymbol{O}(n^2)$
- Le tri par sélection va ***toujours*** exécuter un nombre d'opérations proportionnel à n^2, peu importe la séquence d'entrée
- Le temps d'exécution du tri par insertion varie selon la séquence d'entrée
- Aucune n'est une bonne méthode de tri, sauf pour les petites séquences
- Nous cherchons encore la file à priorité ultime...

Le tri

- Maintenant que vous avez une certaine connaissance du tri, parlons-en un peu plus à fond
- Le tri est essentiel parce qu'une ***recherche*** efficace dans une base de données ne peut être faite que si les enregistrements sont triés.
- Certains estiment qu'environ 20% du temps de calcul planétaire est dédié au tri
- Nous observerons qu'il existe un compromis entre "*simplicité*" et *efficacité* des algorithmes de tri:
- Les tris élémentaires vus jusqu'ici, qui étaient simples à comprendre et à réaliser, ont un temps d'exécution $\boldsymbol{O}(n^2)$ (inutilisables pour de grands n)
- Il existe des algorithmes plus sophistiqués $\boldsymbol{O}(n \log n)$
- Comparaison de clés: *comparons-nous la clé entière ou seulement une partie de la clé?*
- Espace requis: *tri à même la structure (**in-place**) versus l'utilisation de structures auxiliaires*
- Stabilité: *un algorithme de tri **stable** conserve l'ordre relatif des clés égales.*

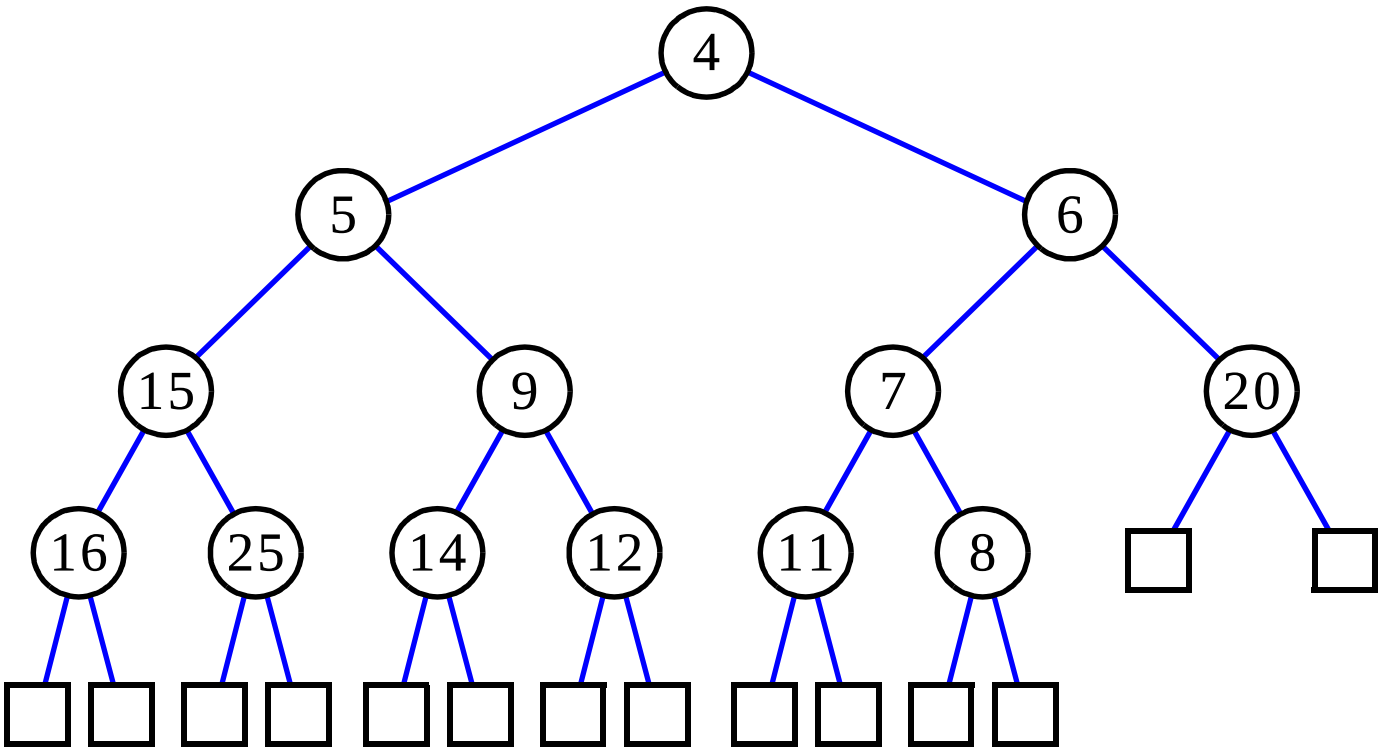
TAS

- Tas (*Heap*)
- Propriétés des tas
- Tri *Heap-Sort*
- Construction ascendante de tas (*Bottom-Up*)
- Repéreurs (*Locator Design Pattern*)



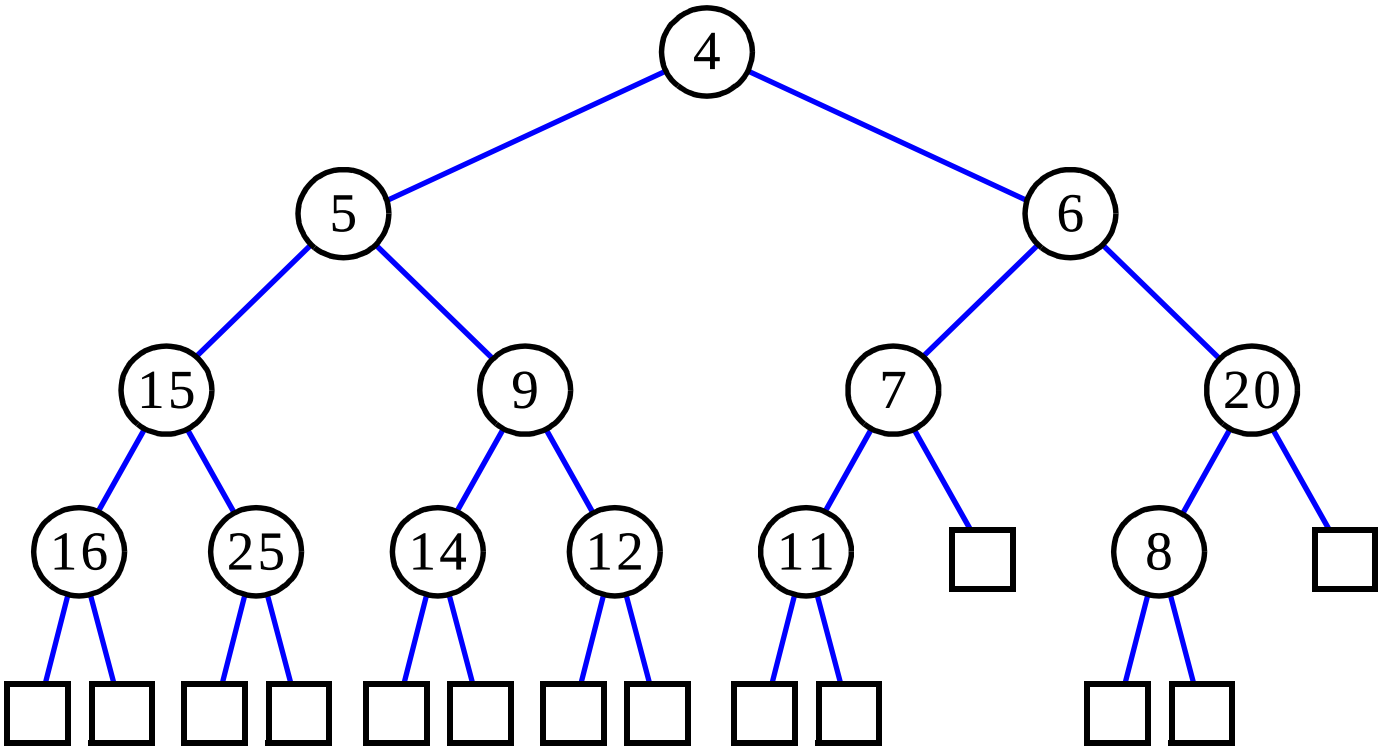
Tas

- Un ***tas*** (*heap*) est un arbre binaire *T* qui emmagasine une collection de clés (ou paires clé-élément) comme nœuds internes et qui satisfait aux deux propriétés suivantes:
  - ***Propriété d'ordre:*** clé(parent) ≤ clé(enfant)
  - ***Propriété structurelle***: tous les niveaux sont pleins, excepté le dernier, ce dernier étant cependant plein à gauche (***arbre binaire complet***)

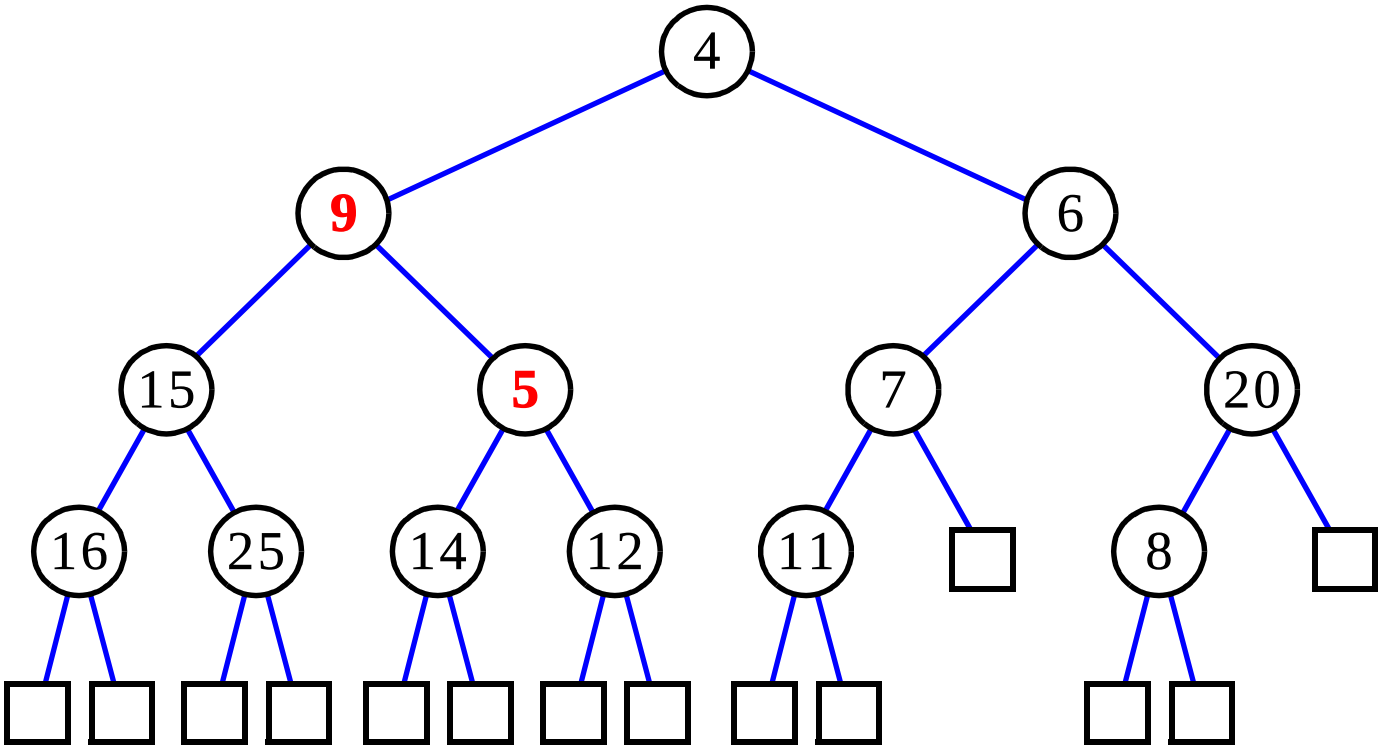


Exemples de non-tas

- le dernier niveau n'est pas plein à gauche



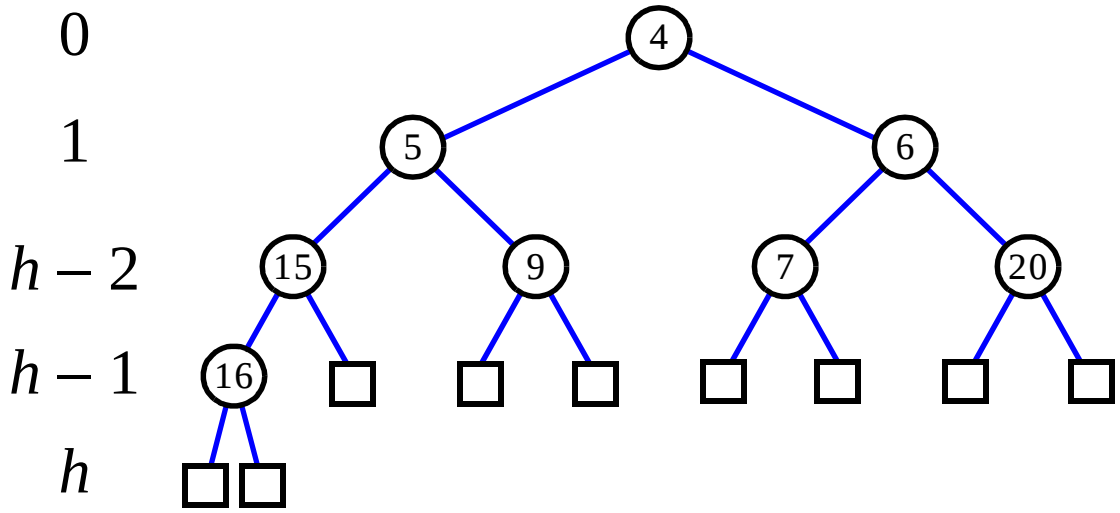
- clé(parent)> clé(enfant)



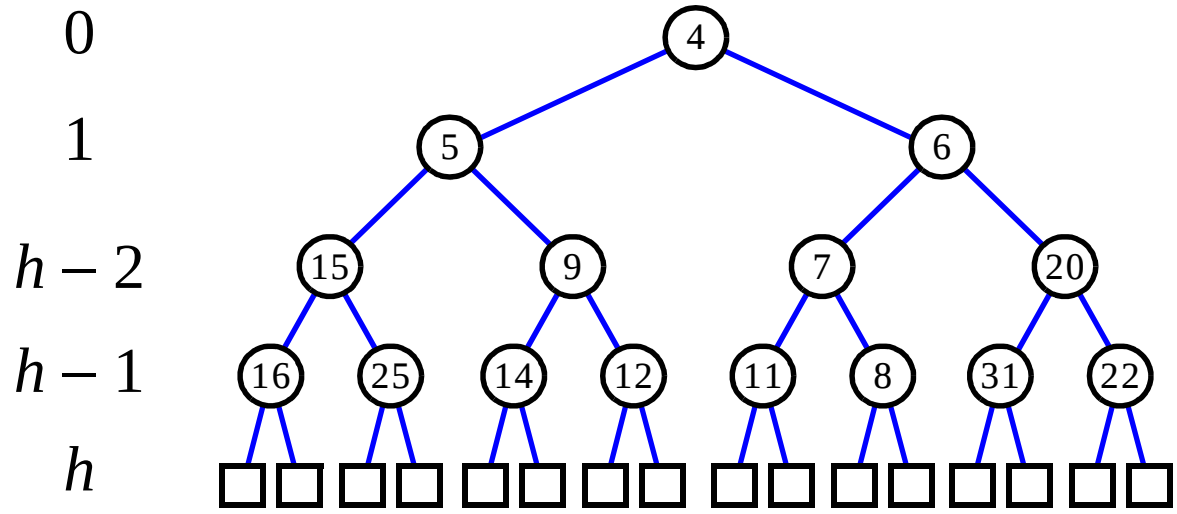
Hauteur d'un tas

Un tas T qui emmagasine n clés a une hauteur $h = \lceil \log(n + 1) \rceil$, qui est $O(\log n)$

- $n \geq 1 + 2 + 4 + \ldots + 2^{h-2} + 1 = 2^{h-1} - 1 + 1 = 2^{h-1}$



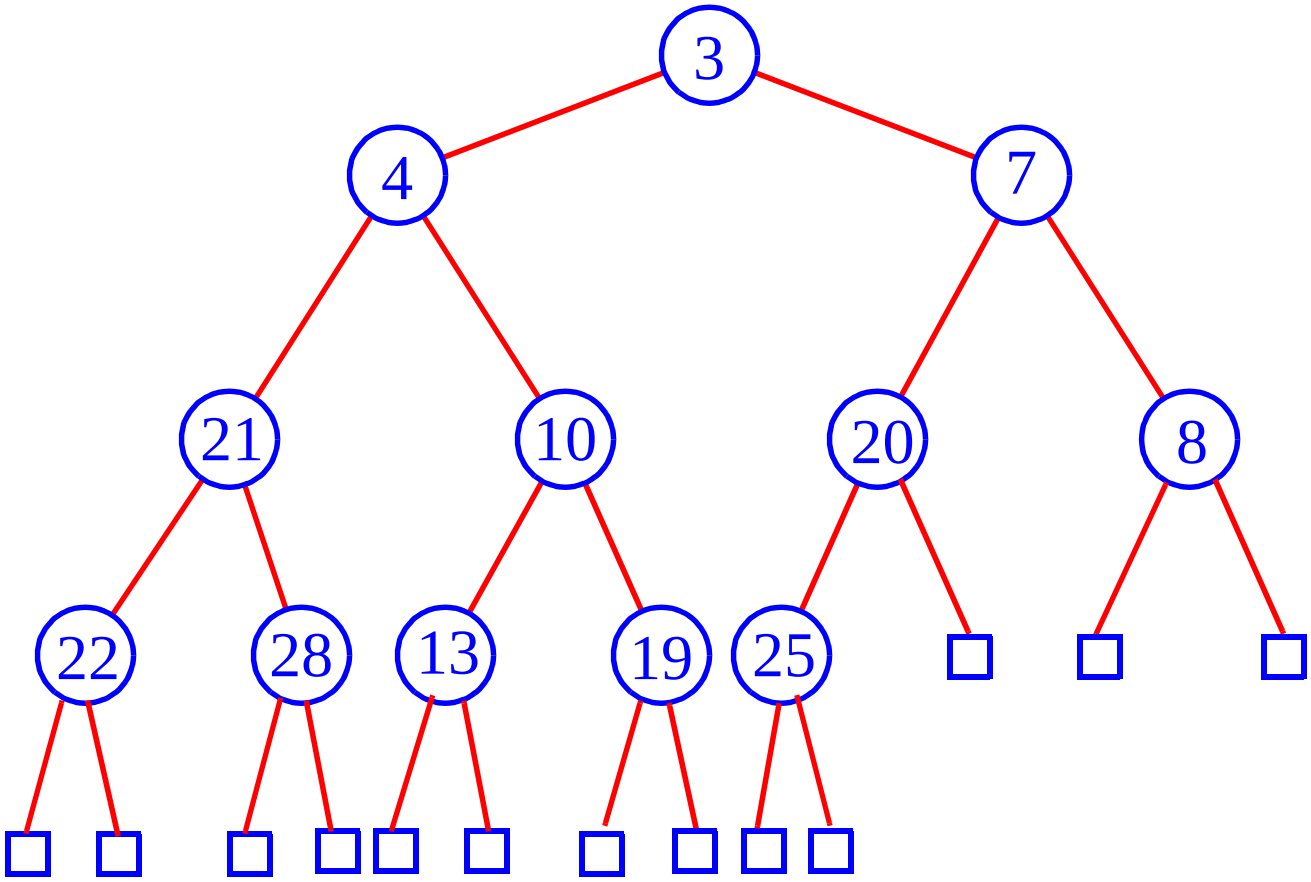
- $n \leq 1 + 2 + 4 + \ldots + 2^{h-1} = 2^{h} - 1$



- Ainsi $2^{h-1} \leq n \leq 2^{h} - 1$
- En calculant le logarithme, nous obtenons $\log (n + 1) \leq h \leq \log n + 1$, et donc $\boldsymbol{h = \lceil \log(n+1) \rceil}$

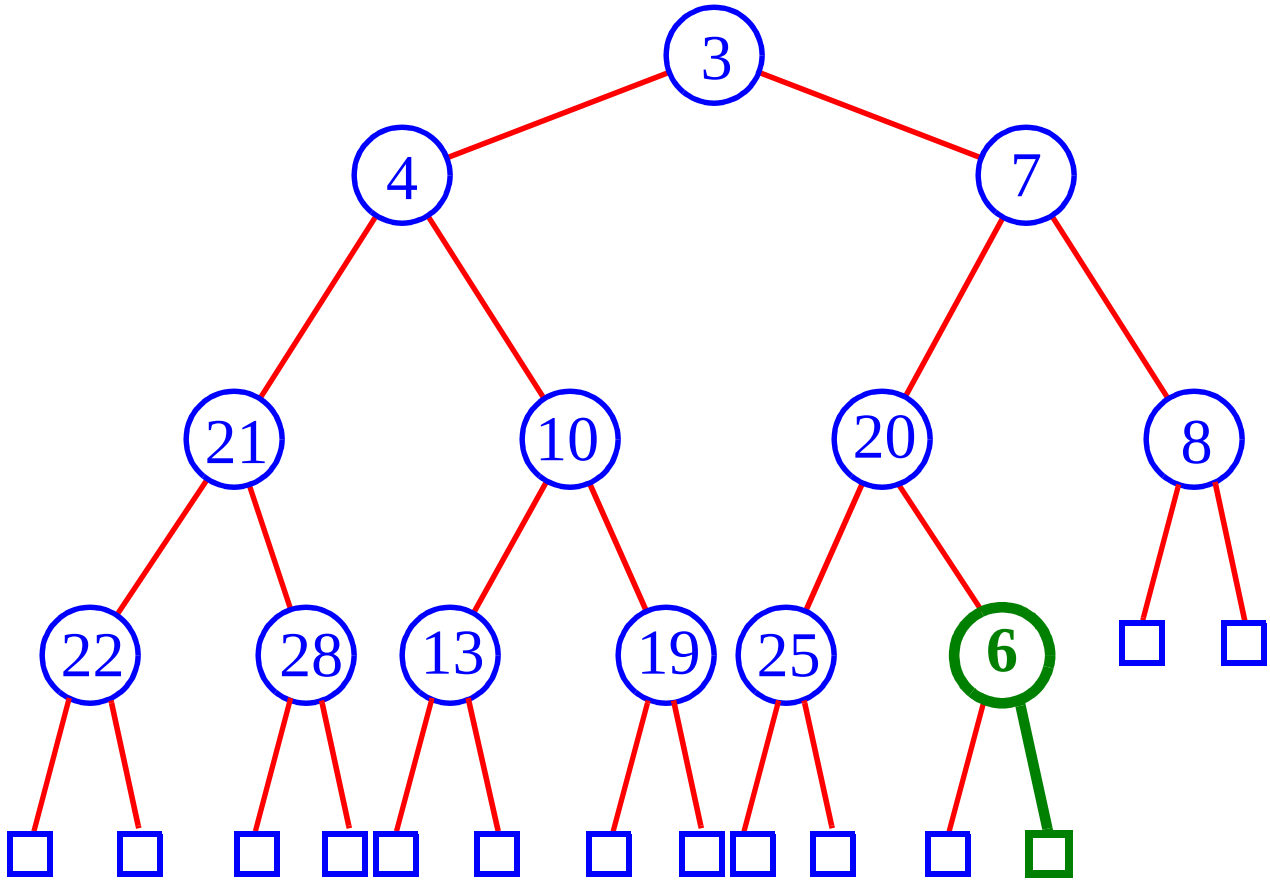
Insertion dans un tas

La clé à insérer est **6**



Insertion dans un tas (suite)

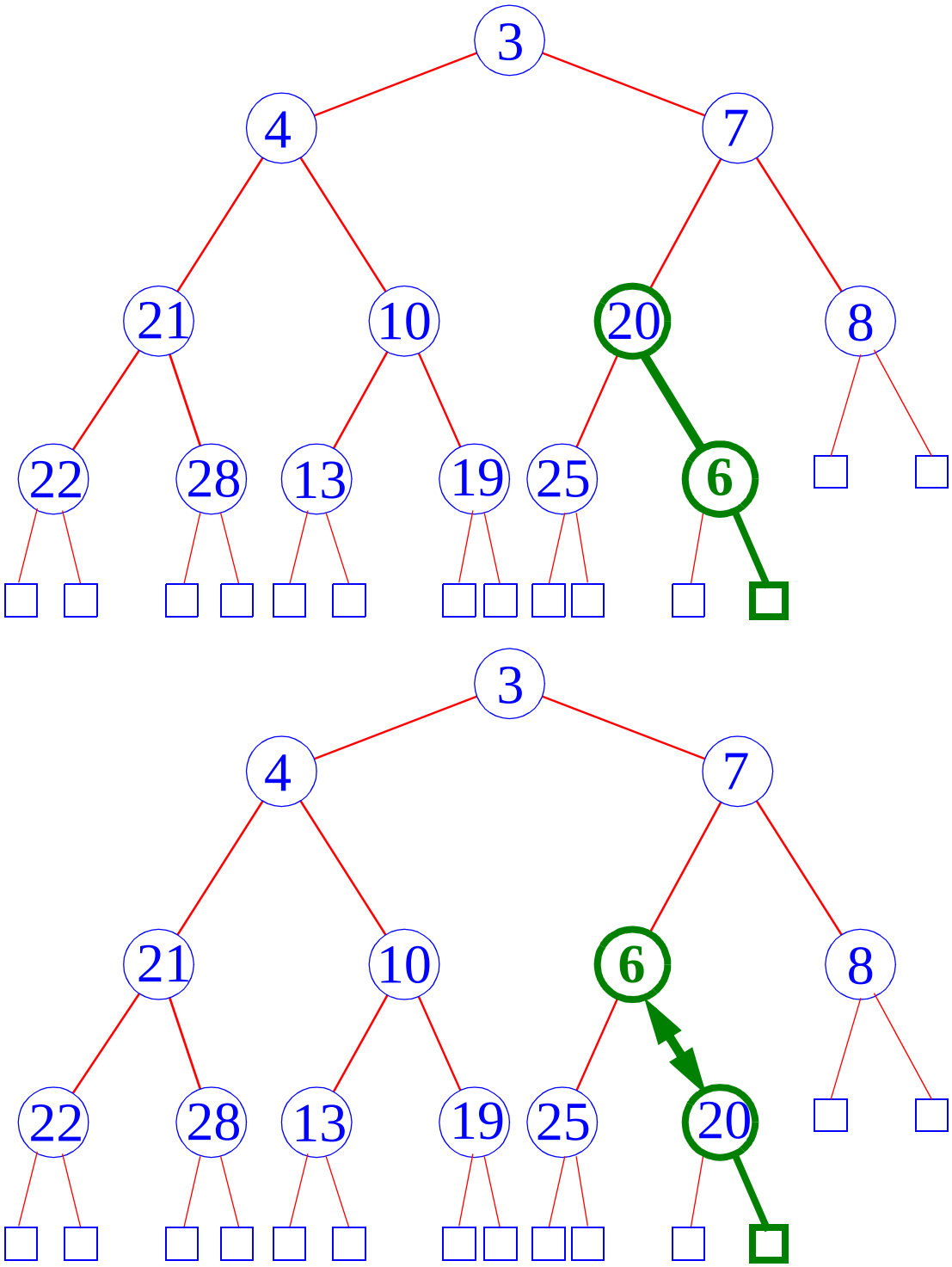
Ajoutez la clé à la ***prochaine position disponible*** dans le tas.



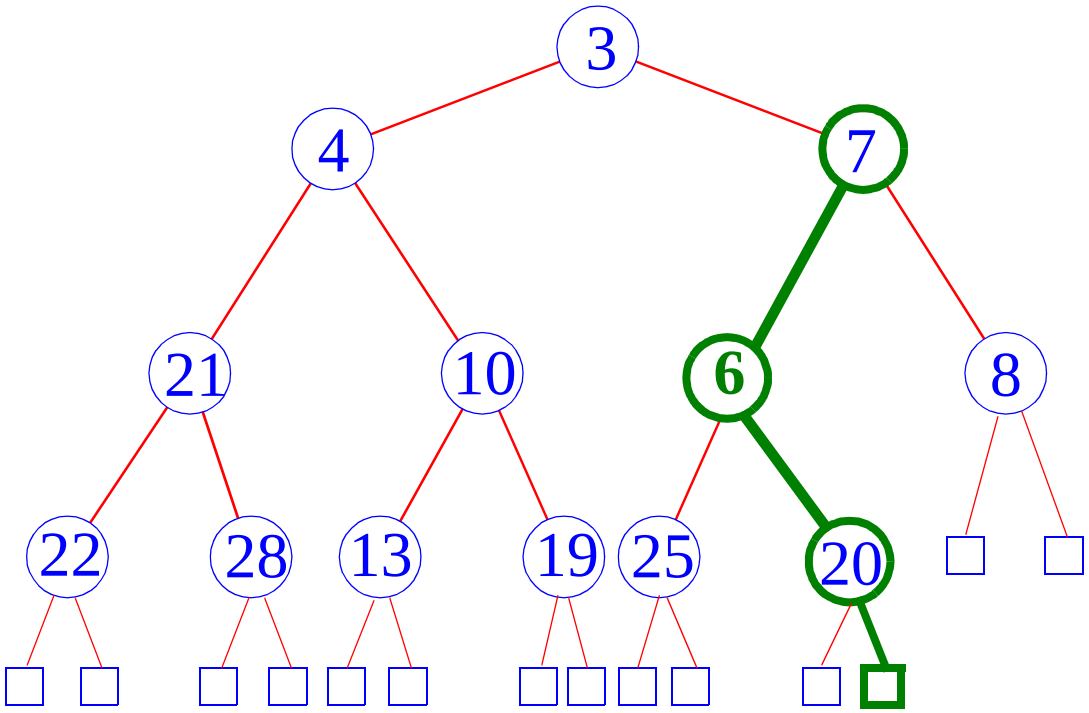
Commencez maintenant la procédure ***Upheap***.

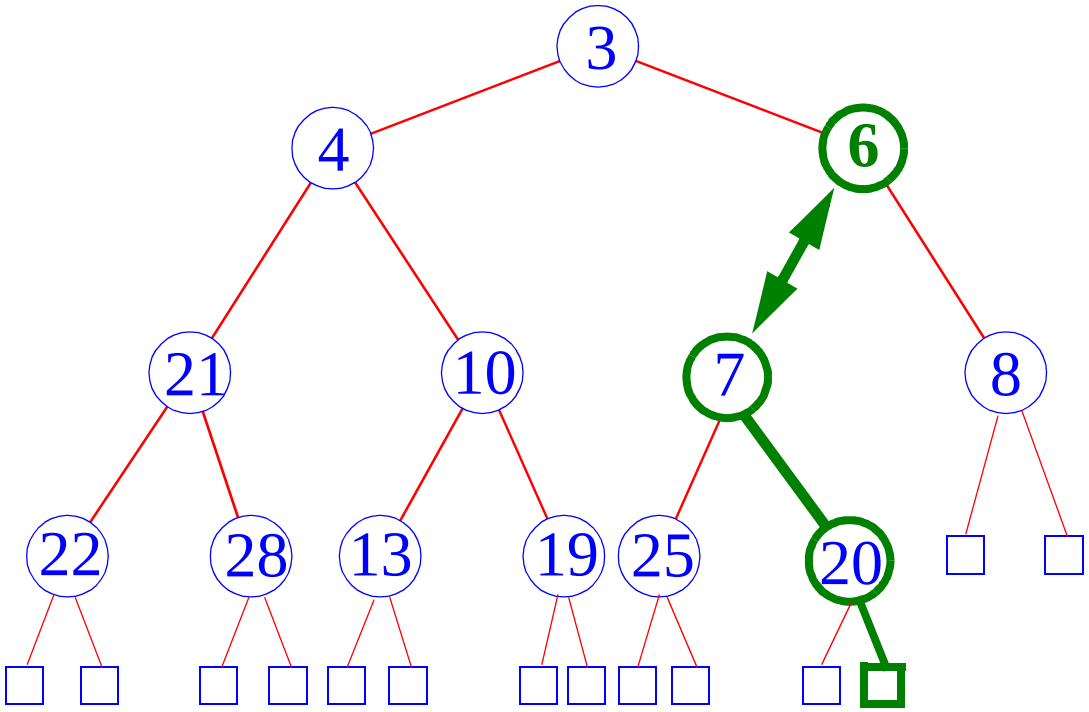
Procédure *Upheap*

- Échangez (*swap*) les clés parent-enfant non-ordonnées

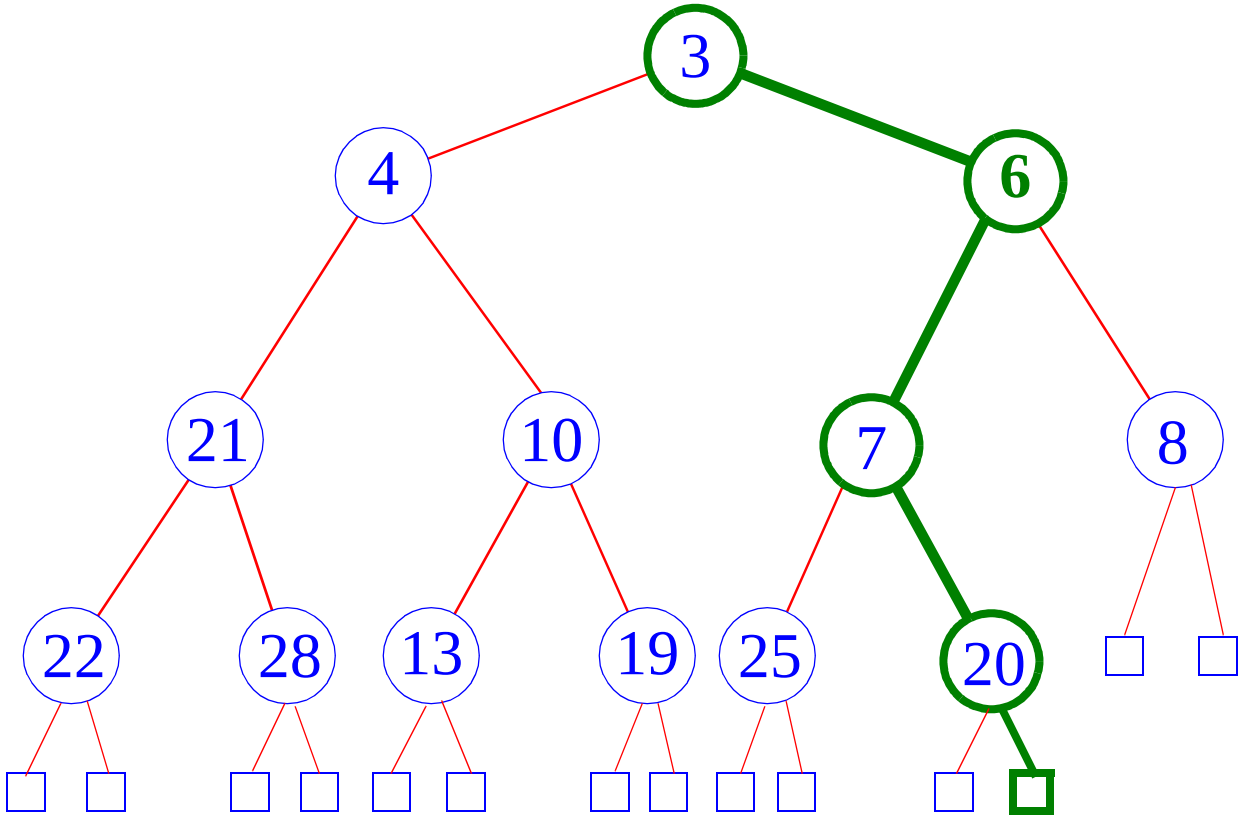


Suite de *Upheap*





Fin de *Upheap*

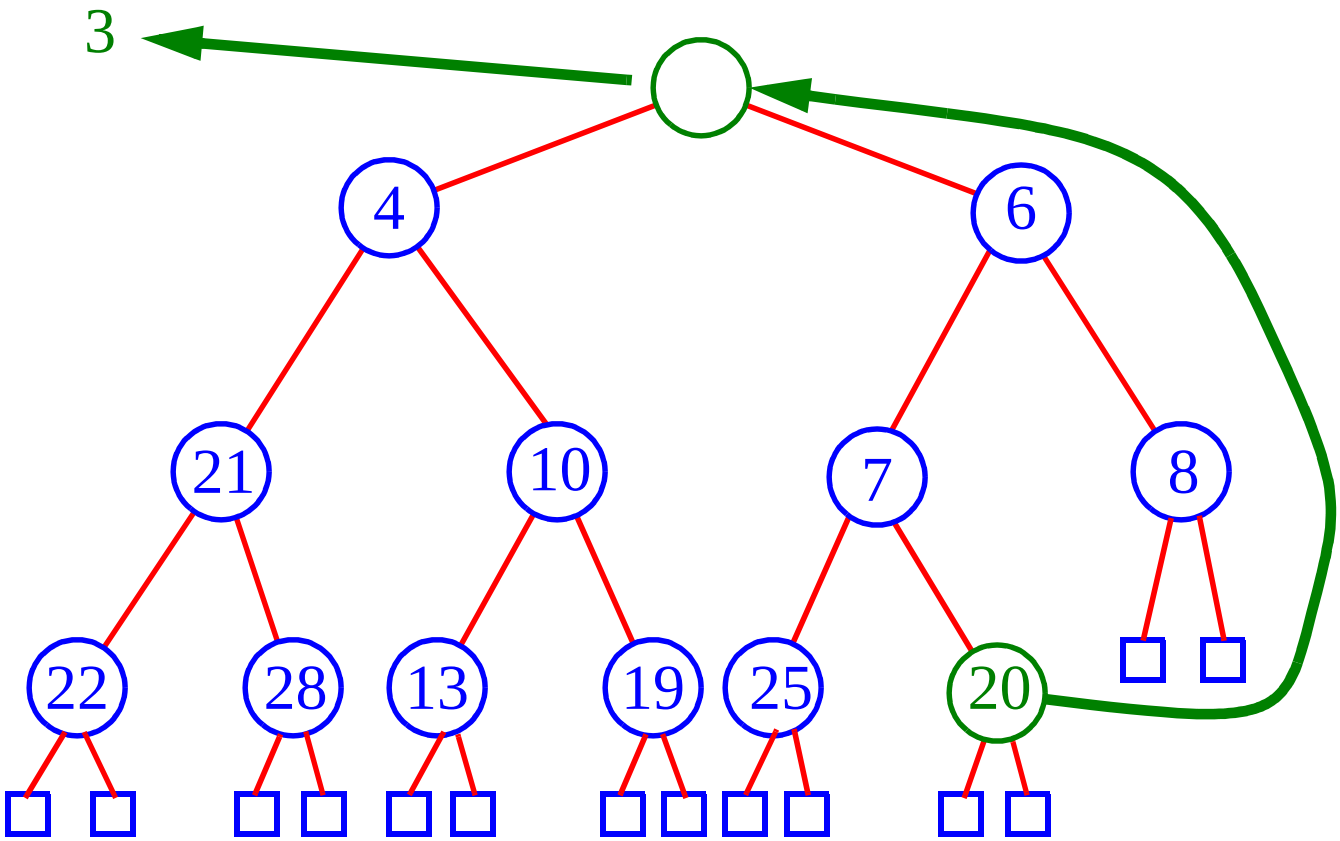


- ***Upheap*** se termine quand la nouvelle clé est plus grande que la clé de son parent **ou** quand le haut du tas est atteint.

- (#échanges total) $\leq$ $(h - 1)$, qui est $O(\log n)$

Suppression dans un tas

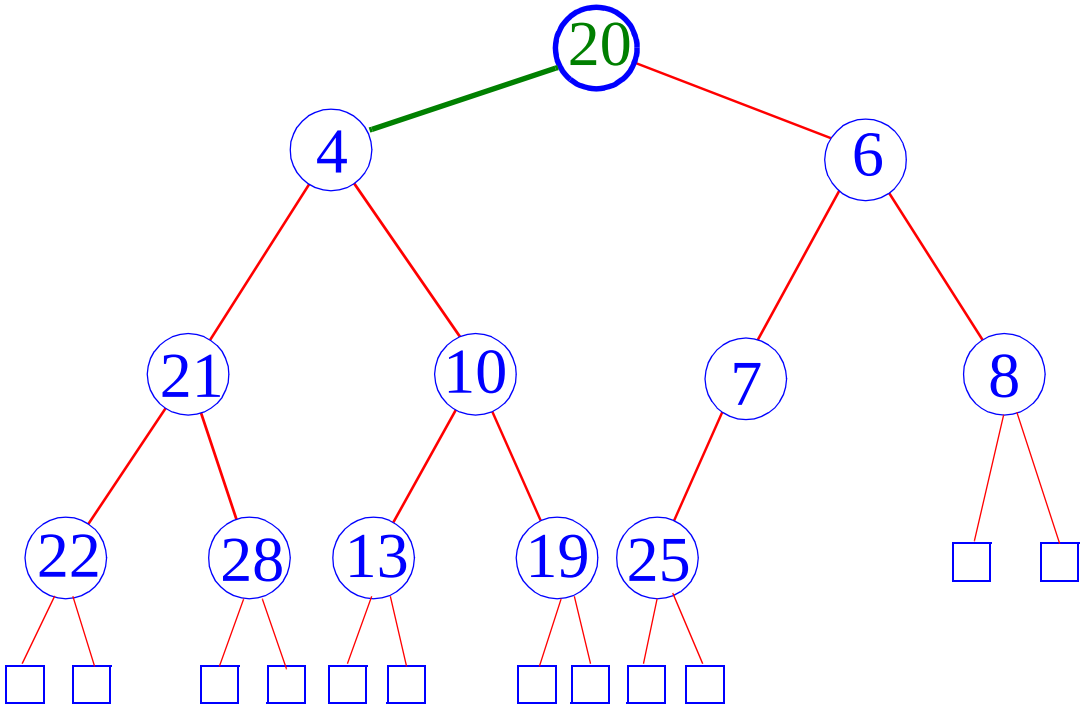
RemoveMin()

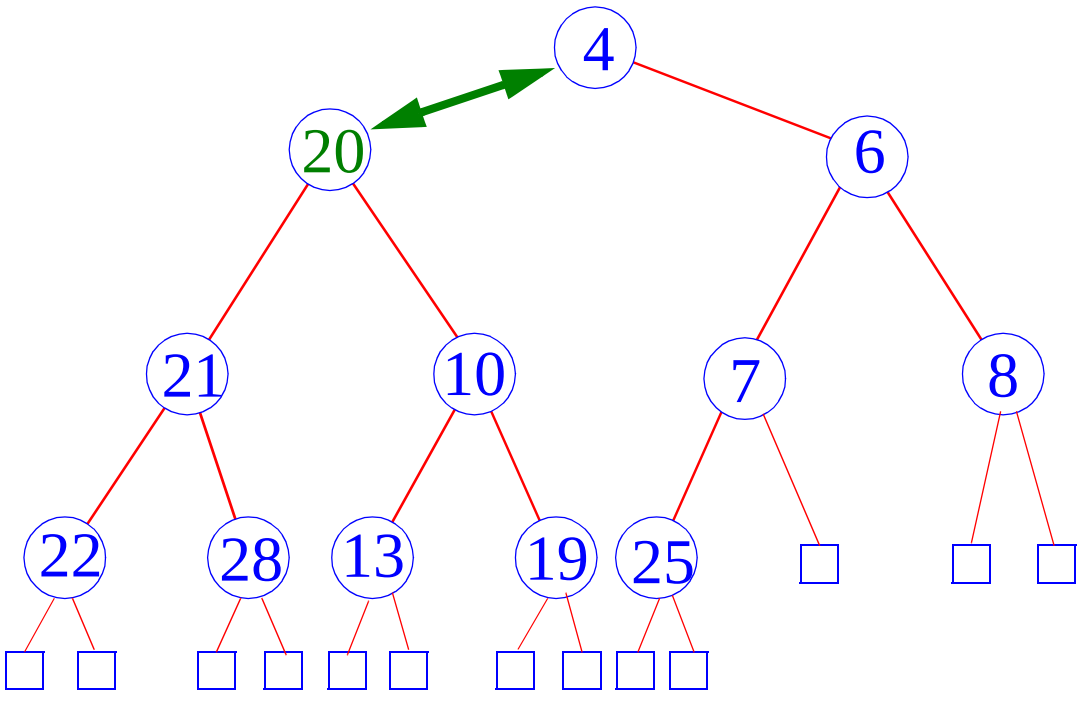


- La suppression de la clé racine laisse un trou
- Nous devons réparer le tas
- Premièrement, remplacez le trou par la toute dernière clé du tas
- Ensuite, appliquez la procédure ***Downheap***

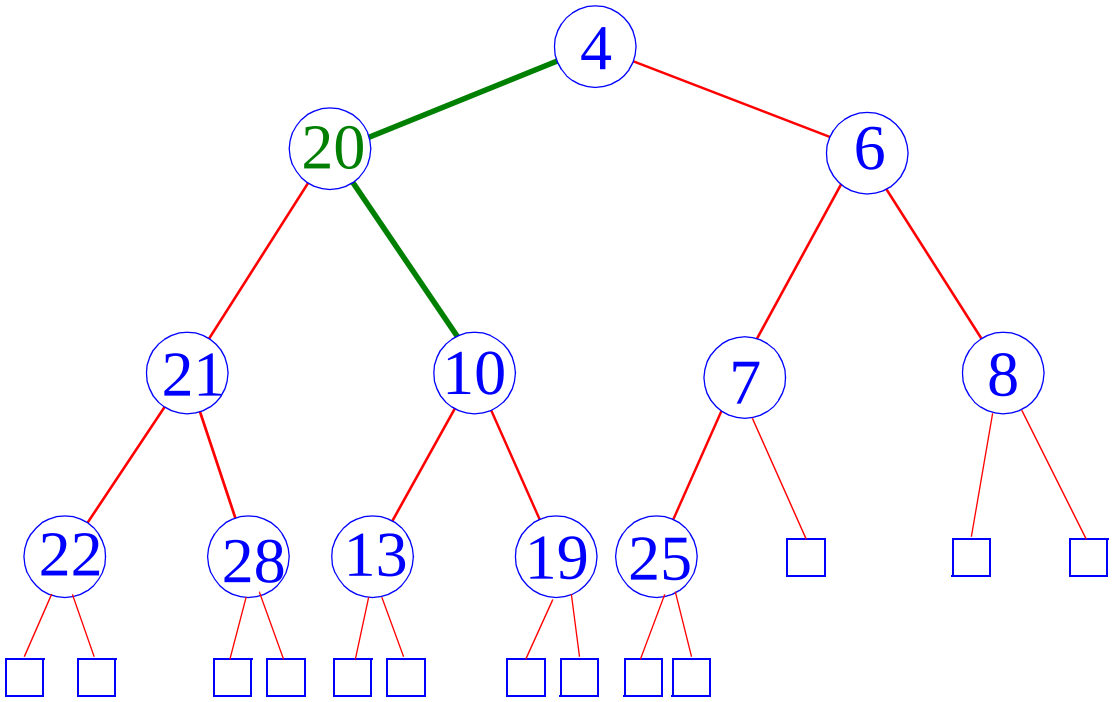
Procédure *Downheap*

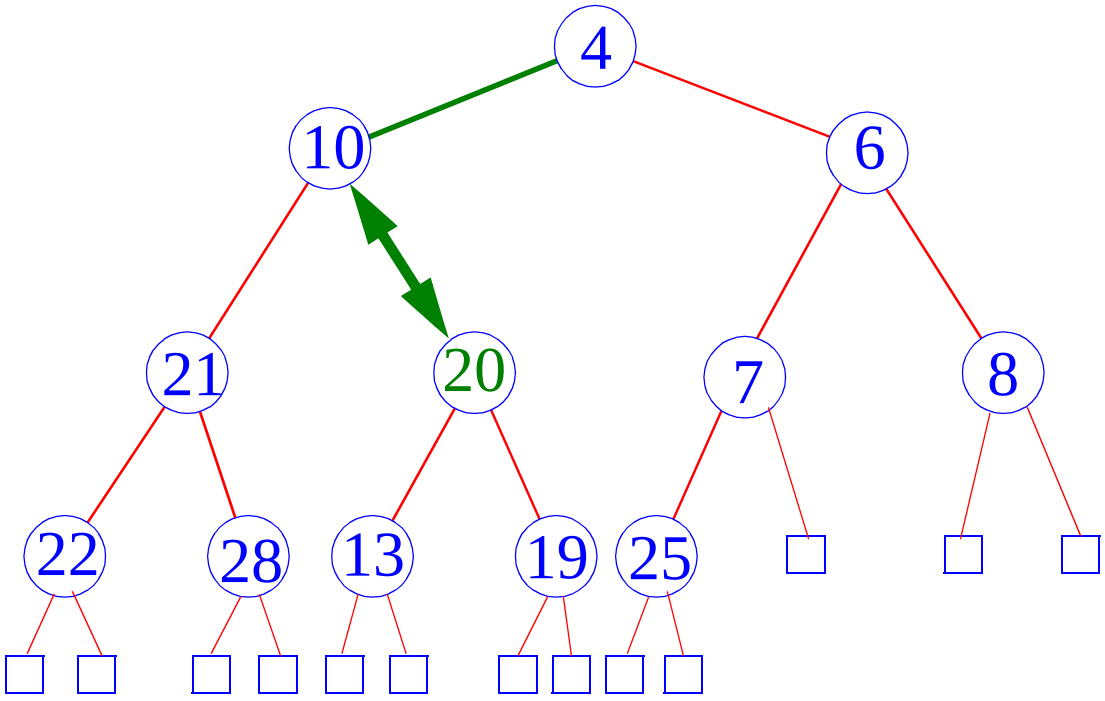
Downheap compare le parent avec son enfant le plus petit. Si cet enfant est plus petit que le parent, alors on les échange l'un pour l'autre.



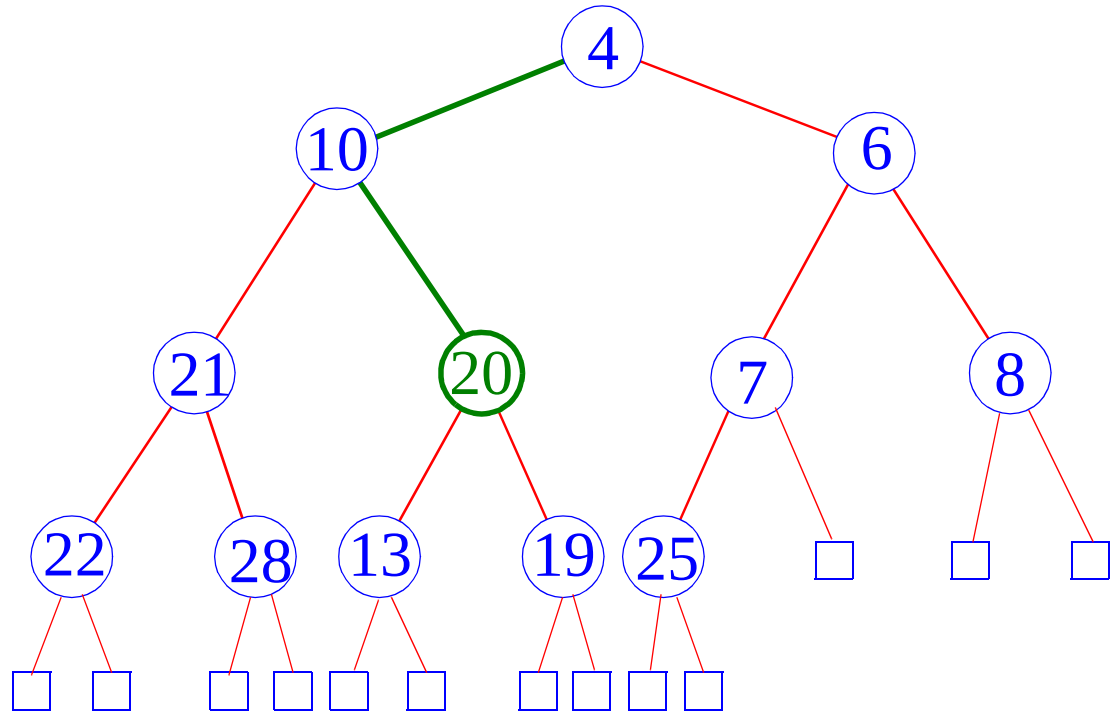


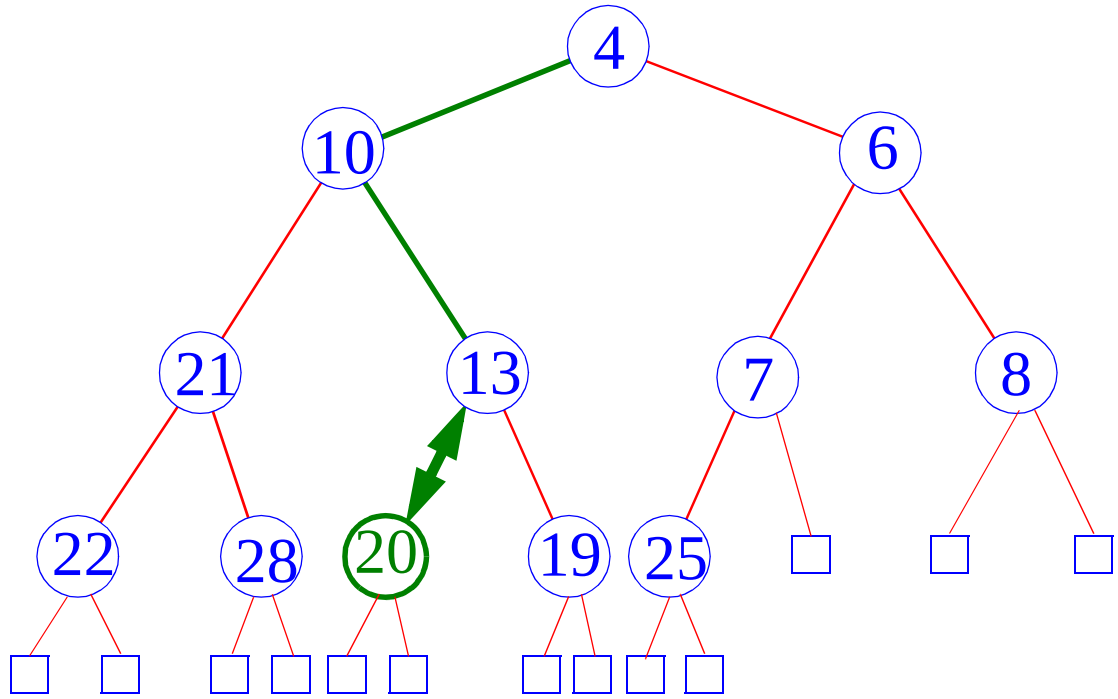
Suite de *Downheap*



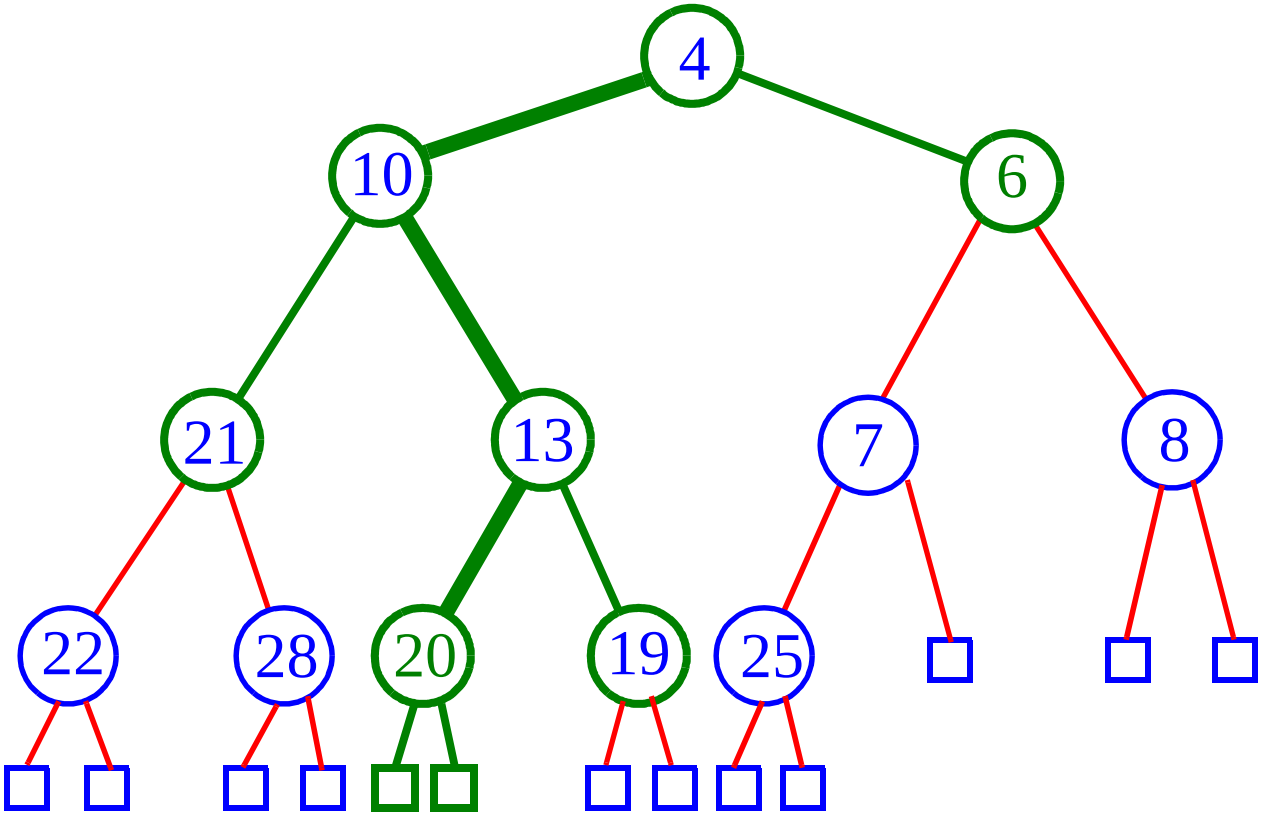


Suite de *Downheap* (2)





Fin de *Downheap*

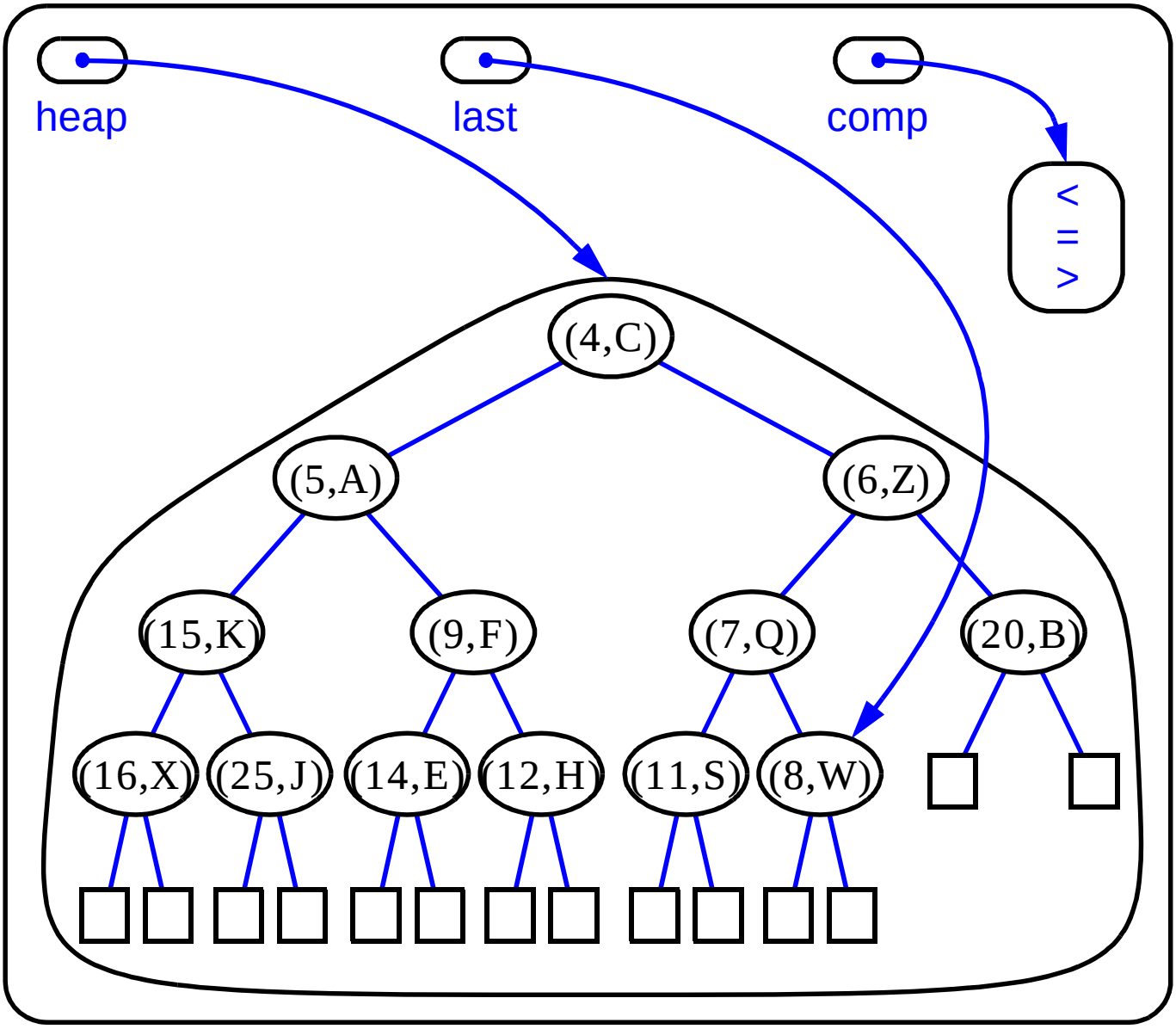


- ***Downheap*** se termine quand la clé est plus grande que les clés de ses deux enfants **ou** quand le bas du tas est atteint.

- (#échanges total) $\leq$ ($h - 1$), qui est $O(\log n)$

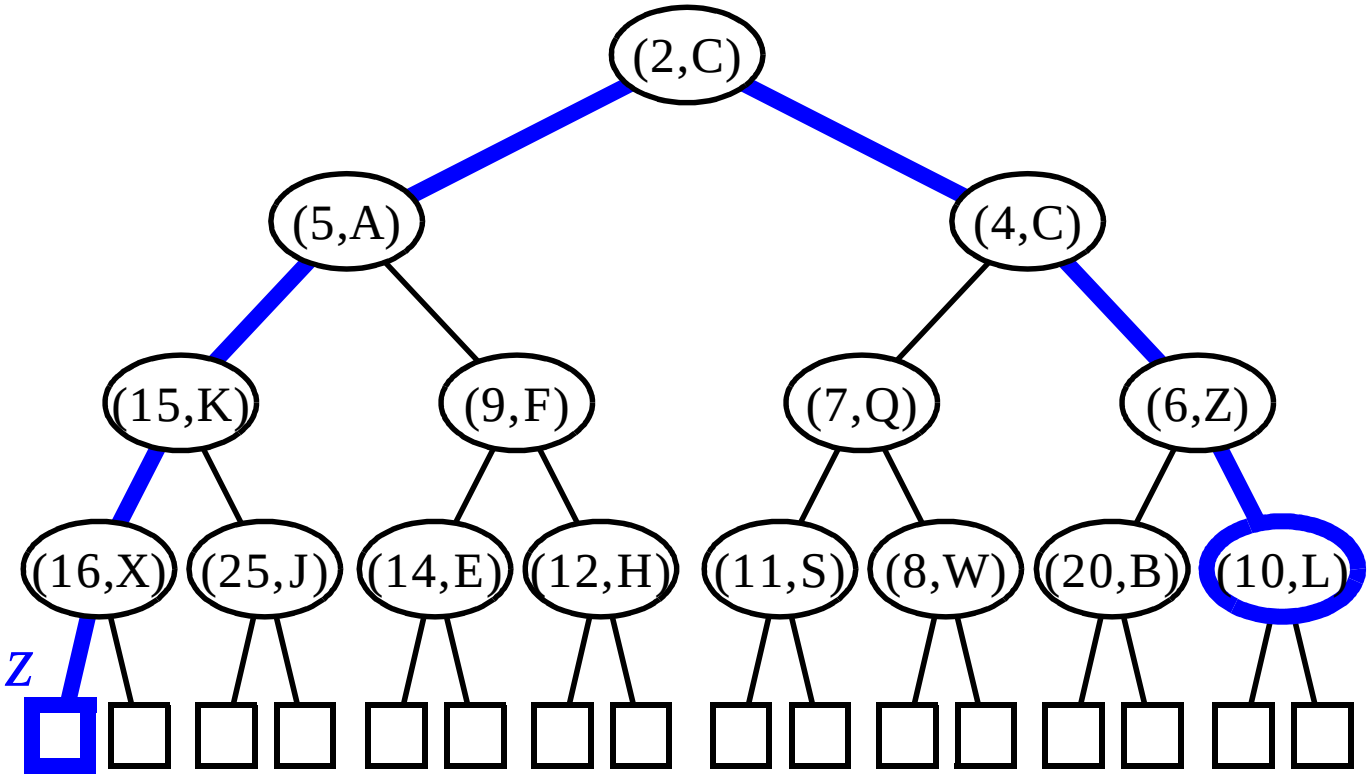
Réalisation d'un tas

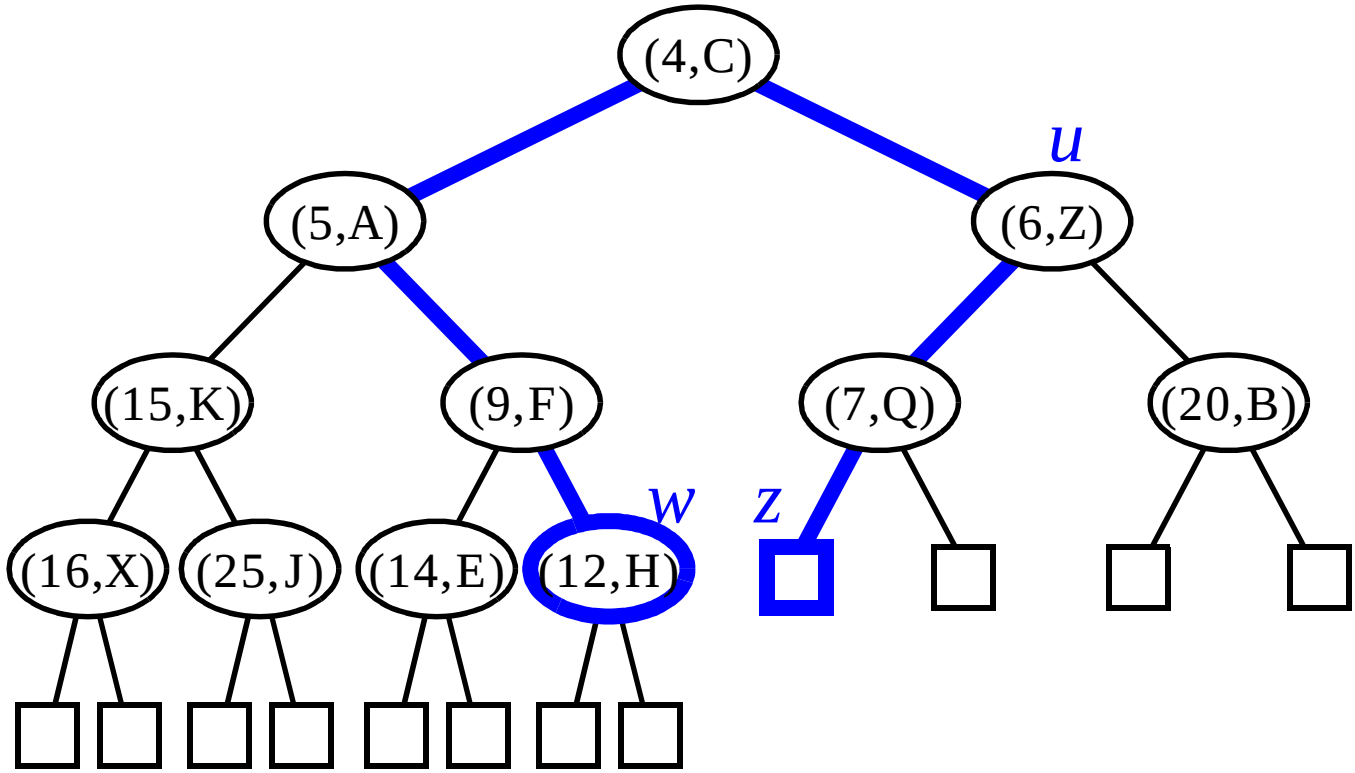
```
public class HeapPriorityQueue implements PriorityQueue
{
   BinaryTree T;
   Position last;
   Comparator comparator;
   ...
}
```



Réalisation d'un tas (suite)

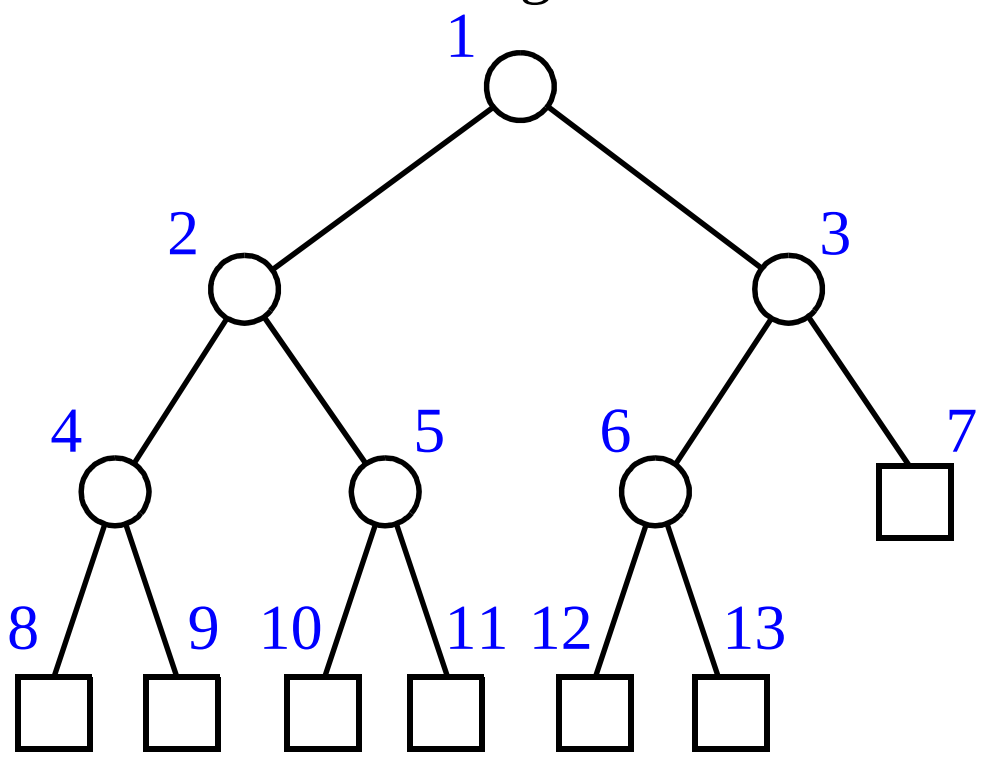
- Deux façons de trouver la position d'insertion *z*:





Réalisation par vecteur (*Vector*)

- Les mises à jour dans l'arbre sous-jacent ne surviennent seulement qu'au "dernier élément".
- Un tas peut être représenté par un vecteur (*vector*), où le nœud au rang *i* a:
  - l'enfant de gauche au rang 2*i* et
  - l'enfant de droite au rang 2*i* + 1



- Les feuilles n'ont pas à être emmagasinées.
- L'insertion et la suppression de clés dans le tas correspondent respectivement à insertLast et à removeLast dans le vecteur.

Tri *Heap-Sort*

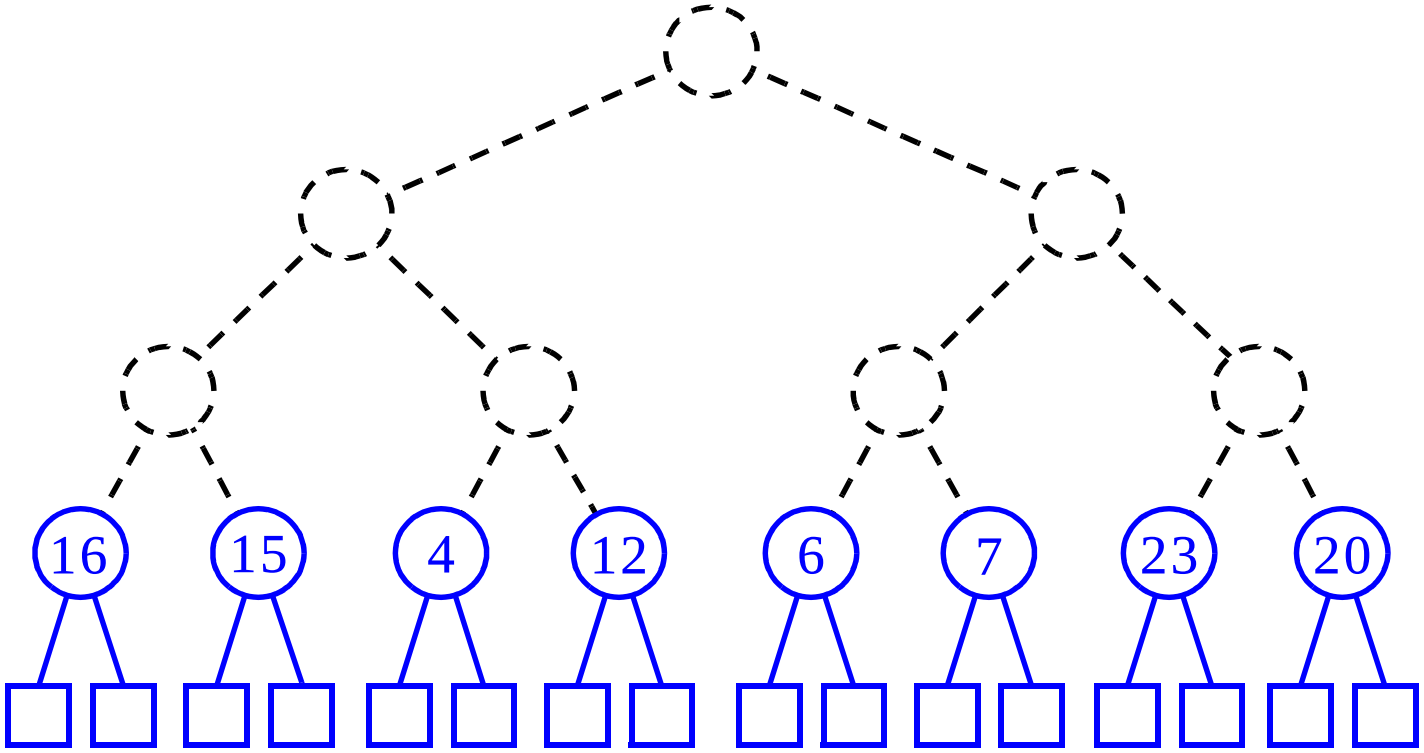
- Toutes les méthodes d'un tas s'exécutent en un temps logarithmique, ou mieux.
- Si nous réalisons le tri *PriorityQueueSort* avec un tas comme file à priorité, insertItem et removeMin prennent alors $O(\log k)$ chacun, où k est le nombre d'éléments dans le tas à un moment donné.
- Nous avons toujours au plus n éléments dans le tas, alors le pire des cas en terme de complexité pour ces méthodes est $O(\log n)$.
- Chaque phase prend donc $O(n \log n)$, et le temps d'exécution de l'algorithme est aussi de $O(n \log n)$.
- Ce tri est connu sous le nom de ***heap-sort***.
- Le temps d'exécution $O(n \log n)$ d'un tri *heap-sort* est bien meilleur que le temps d'exécution $O(n^2)$ d'un tri à bulle, par sélection, ou par insertion.

Tri Heap-Sort *in-place*

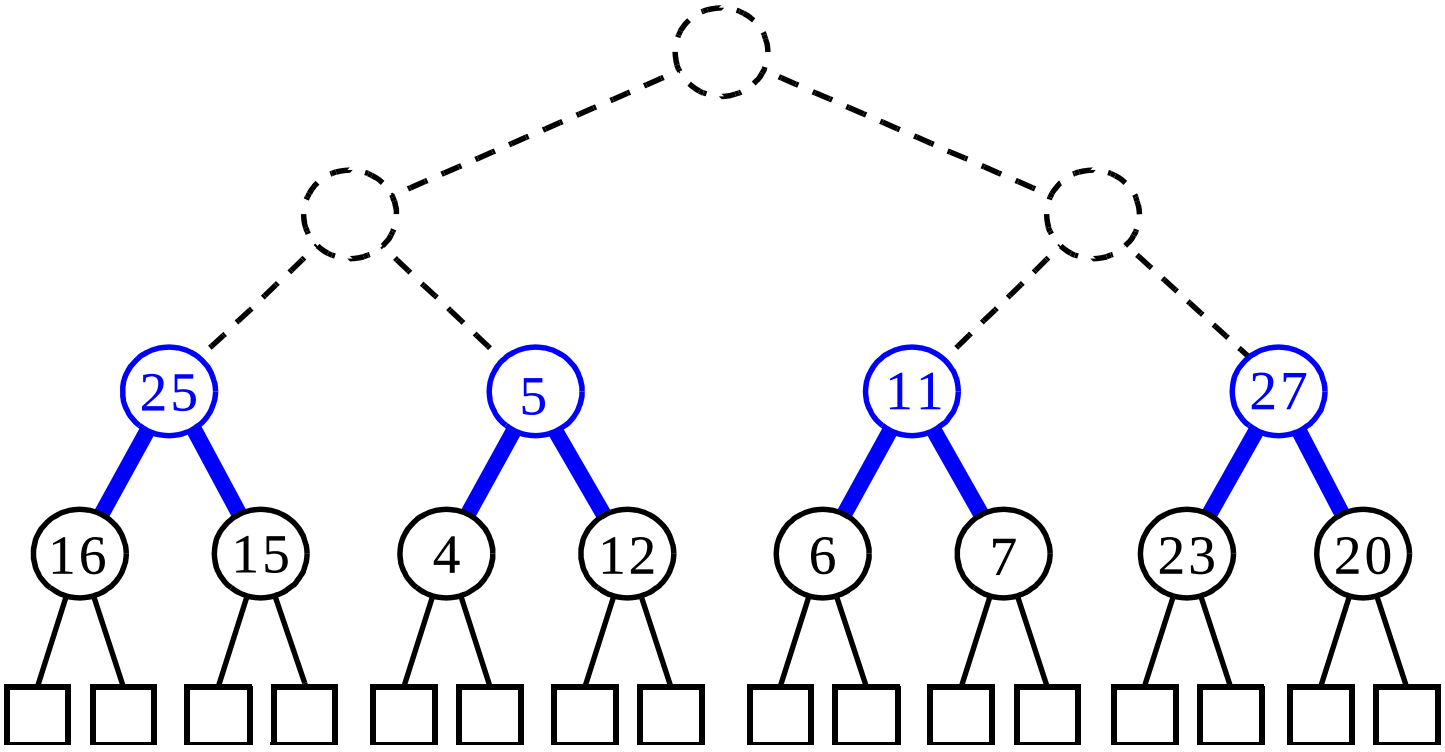
- N'utilise pas de tas (ou d'autre structure) externe.
- Utilise une représentation par vecteur pour contenir le tas. Construction ascendante (*bottom-up*)...

Construction ascendante du tas (1)

- construisez $(n + 1)/2$ tas à un seul élément (trivial)

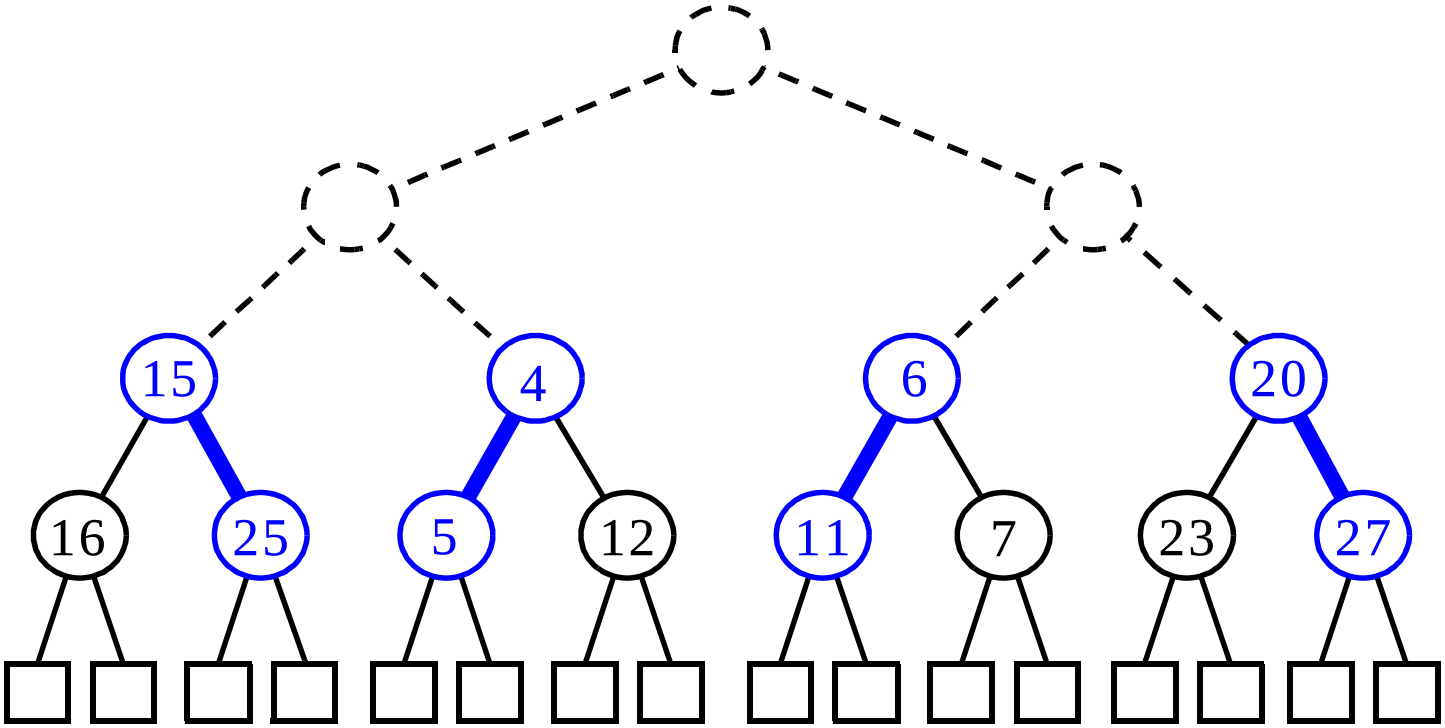


- construisez maintenant des tas à trois éléments

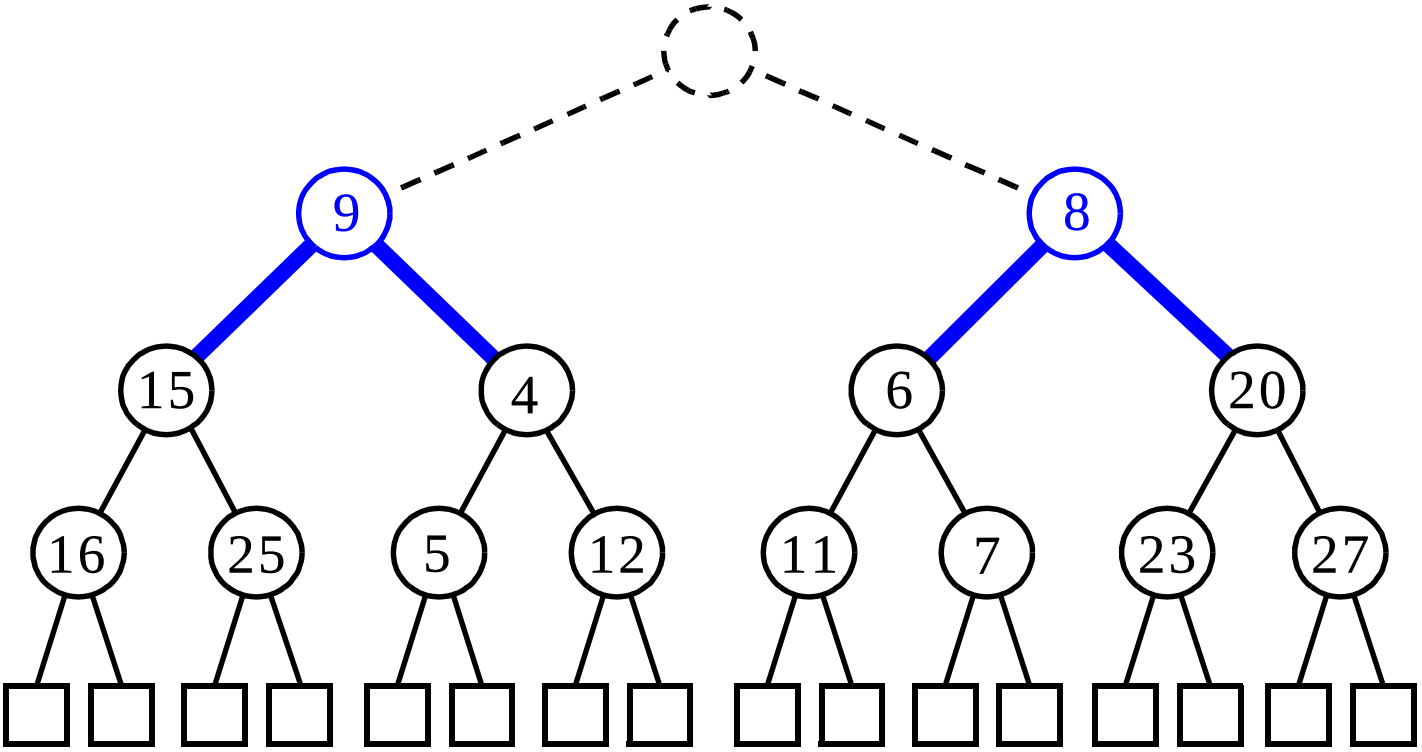


Construction ascendante du tas (2)

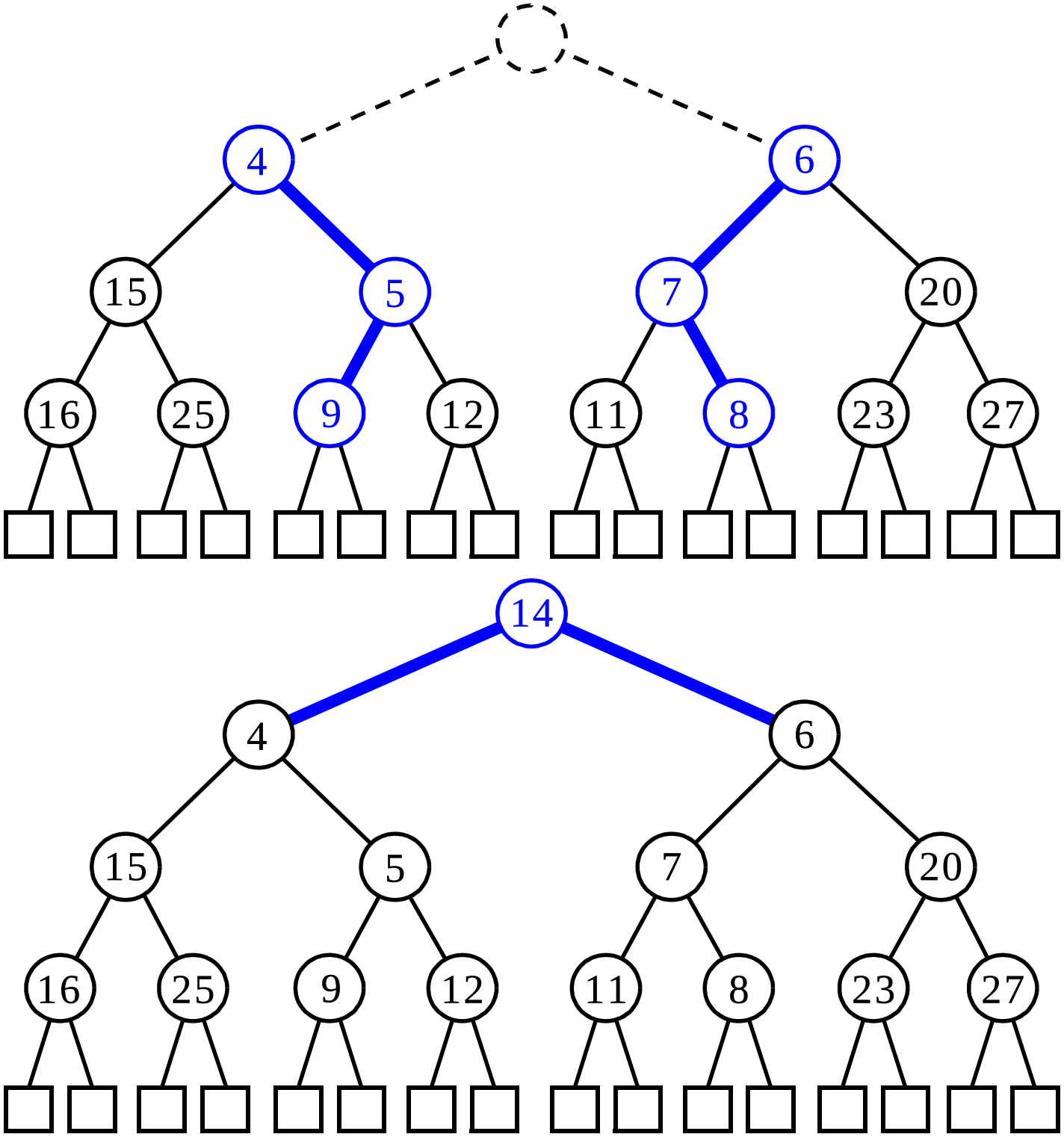
- préservez la propriété d'ordre avec ***downheap***



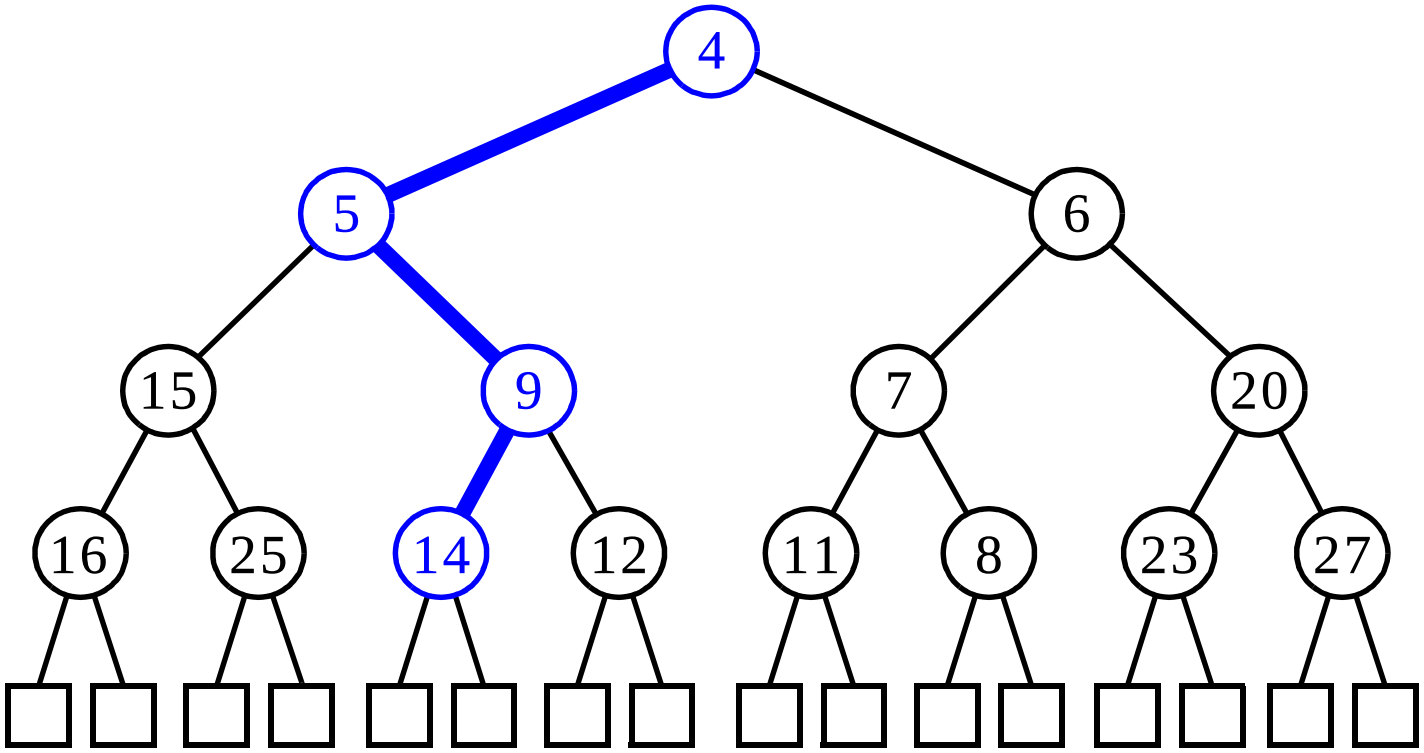
- formez maintenant des tas à 7 éléments



Construction ascendante du tas (3)



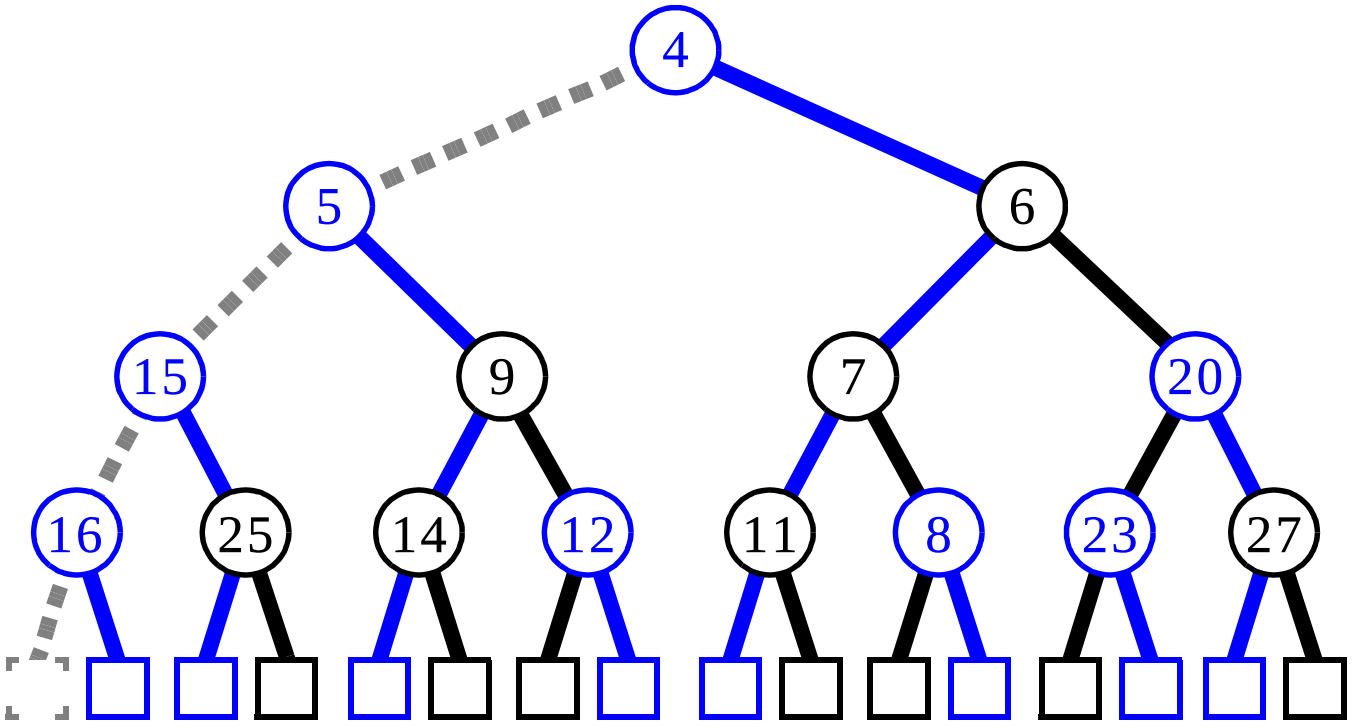
Construction ascendante du tas (4)



Fin!

Analyse de la construction ascendante de tas

- Proposition: la construction ascendante de tas avec n clés a un temps d'exécution $\boldsymbol{O}(n)$.
  - Insérer $(n + 1)/2$ nœuds
  - Insérer $(n + 1)/4$ nœuds et utiliser *downheap*
  - Insérer $(n + 1)/8$ nœuds et utiliser *downheap*
  - ...
  - analyse visuelle:



- n insertions, $n/2$ *downheap* pour un temps d'exécution total d'ordre $\boldsymbol{O}(n)$.

Repéreurs (*Locators*)

- Des repéreurs peuvent être utilisés pour suivre les éléments lorsqu'ils sont déplacés dans un contenant.
- Un repéreur (patron de conception *locator*) suit un élément spécifique, même si cet élément change de position dans son contenant.
- Le TAD locator contient les méthodes fondamentales suivantes:
  - element(): retourne l'élément de l'item associé au locator.
  - key(): retourne la clé de l'item associé au locator.
- À l'aide de repéreurs nous définissons des méthodes additionnelles pour le TAD file à priorité:
  - insert(*k*,*e*): insère (*k*,*e*) dans *P* et retourne son locator
  - min(): retourne le locator de l'élément à la plus petite clé
  - remove(*l*): supprime l'élément au locator *l*
- Dans notre application boursière, nous retournons un repéreur quand une commande est faite. Un repéreur permet de spécifier sans ambiguïté une commande lors d'une annulation.

Positions et Repéreurs

- Vous pourriez être en train de vous demander quelle est la différence entre repéreurs et positions, et pourquoi les distinguer.
- Il est vrai qu'ils ont des méthodes semblables.
- La différence se situe au niveau de leur utilisation primaire.
- Les positions font abstraction de la réalisation spécifique de l'accès aux éléments (indices ou nœuds).
- Les positions sont définies relativement l'une par rapport à l'autre (précédent/prochain, père/enfant).
- Les repéreurs surveillent où se situent les éléments. Dans la réalisation d'un TAD pour repéreurs, un repéreur conserve typiquement la position courante de l'élément.
- Les repéreurs associent les éléments avec leurs clés.

Positions et Repéreurs au travail

- Par exemple, considérez le Service de valet de stationnement CSI2514 (créé par les AE parce qu'ils avaient trop de temps libre).
- Lorsqu'ils ont démarré leur entreprise, André et Daniel décidèrent de créer une structure de données pour déterminer où les voitures sont situées.
- André suggère qu'une ***position*** représente ***l'espace de stationnement*** dans lequel la voiture se trouve.
- Cependant Daniel sait bien que les AE se promènent avec les voitures partout sur le campus et qu'elles ne seront pas toujours stationnées au même endroit.
- Alors ils décident d'installer un ***repéreur*** (un ***appareil sans fil***) dans chaque voiture. Chaque repéreur a un identifiant, qui est inscrit sur le coupon de retour.
- Quand un client demande sa voiture, l'AE active le repéreur, et alors la voiture klaxonne et ses lumières clignotent! Si la voiture est stationnée, André et Daniel sauront où la retrouver dans le stationnement, sinon, l'AE conduisant cette voiture saura qu'il est temps de la rapporter.