Module EA4 – Éléments d'Algorithmique

Dominique Poulalhon
dominique.poulalhon@liafa.univ-paris-diderot.fr

Université Paris Diderot
L2 Informatique
Année universitaire 2014-2015

Remarques sur l'interro

Exercice 1

attention à l'ordre des produits : $\sigma\tau : i \overset{\tau}{\longmapsto} \tau(i) \overset{\sigma}{\longmapsto} \sigma(\tau(i))$

Exercice 2

- $\Theta \equiv (O \text{ et } \Omega)$
- le log écrase tout : $n \notin \Theta(n^2)$, mais $\log n \in \Theta(\log(n^2))$
- $2^n << n! << n^n$, mais $\log(n!) \in \Theta(\log(n^n)) = \Theta(n \log n)$

Exercice 3

- il faut lire l'énoncé
- quand un algo sur une liste est plus coûteux qu'un tri (*i.e.* $\Omega(n \log n)$, mais pas $\Theta(n \log n)$), il faut toujours comparer avec un algo commençant par un tri

Remarques sur l'interro

Exercice 4

- (bis) il faut lire l'énoncé
- penser à la dichotomie

Exercice 5 (et 6)

- revoir l'algorithme de fusion – ce n'est pas une succession d'insertions
- on peut faire plein d'autres choses par un parcours en parallèle de deux listes triées

Remarques sur l'interro

Exercice 7

revoir l'algorithme de sélection rapide – le seul moyen d'obtenir un algorithme linéaire (même seulement en moyenne) tout en faisant des (nombreaux) passages successifs sur une liste, c'est d'en jeter des bouts à chaque étape

Exercice 8

revoir l'algorithme de suppression – il en existe de nombreuses variantes, mais si vous n'utilisez pas celle du cours, il faut la décrire (et s'assurer qu'elle est bien aussi efficace)

Hauteur d'un ABR

Rappels de la semaine dernière :

Lemme

la hauteur $h(A)$ *d'un arbre binaire* A *à* n *sommets vérifie :*

$$\log n \leqslant h(A) \leqslant n - 1$$

Théorème

la hauteur moyenne d'un ABR construit par l'insertion des entiers $1, \ldots, n$ *dans un ordre aléatoire choisi uniformément est en* $\Theta(\log n)$.

Comparaison avec les représentations par liste

	tableau		liste chaînée		ABR
	non trié	trié	non triée	triée	
recherche	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(h)$
insertion	$+\Theta(1)$	$\Theta(n)$	$+\Theta(1)$	$+\Theta(1)$	$\Theta(h)$
suppression	$\Theta(n)$	$\Theta(n)$	$+\Theta(1)$	$+\Theta(1)$	$\Theta(h)$
minimum	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$	$\Theta(h)$

Comparaison avec les représentations par liste

	tableau		liste chaînée		ABR
	non trié	trié	non triée	triée	(en moyenne)
recherche	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$
insertion	$+\Theta(1)$	$\Theta(n)$	$+\Theta(1)$	$+\Theta(1)$	$\Theta(\log n)$
suppression	$\Theta(n)$	$\Theta(n)$	$+\Theta(1)$	$+\Theta(1)$	$\Theta(\log n)$
minimum	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$	$\Theta(\log n)$

Comparaison avec les représentations par liste

	tableau		liste chaînée		ABR
	non trié	trié	non triée	triée	(en moyenne)
recherche	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$
insertion	$+\Theta(1)$	$\Theta(n)$	$+\Theta(1)$	$+\Theta(1)$	$\Theta(\log n)$
suppression	$\Theta(n)$	$\Theta(n)$	$+\Theta(1)$	$+\Theta(1)$	$\Theta(\log n)$
minimum	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$	$\Theta(\log n)$
sélection	$\Theta(kn)$	$\Theta(1)$	$\Theta(kn)$	$\Theta(k)$	??
union	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n)$	??

Corollaire

la construction d'un ABR par insertion successive de n *valeurs a une complexité (en temps)* $\Theta(n \log n)$ *en moyenne*

Corollaire

trier une liste par construction d'un ABR a une complexité $\Theta(n \log n)$ *en moyenne*

Corollaire

la complexité en moyenne de QuickSort est également $\Theta(n \log n)$ *en moyenne*

Assurer une hauteur logarithmique ?

i.e. contraindre les ABR à rester « raisonnablement » équilibrés

- les arbres rouges-noirs
- les arbres AVL
- ...

Assurer une hauteur logarithmique ?

i.e. contraindre les ABR à rester « raisonnablement » équilibrés

- les arbres rouges-noirs
- les arbres AVL
- ...

les arbres rouges-noirs

- sommets rouges ou noirs
- racine noire
- le père d'un sommet rouge est noir
- même nombre de sommets noirs dans chaque branche

Assurer une hauteur logarithmique ?

i.e. contraindre les ABR à rester « raisonnablement » équilibrés

- les arbres rouges-noirs
- les arbres AVL
- ...

les AVL

pour chaque nœud, les hauteurs des deux sous-arbres diffèrent au plus de 1

Assurer une hauteur logarithmique ?

i.e. contraindre les ABR à rester « raisonnablement » équilibrés

- les arbres rouges-noirs
- les arbres AVL
- ...

Théorème

la hauteur d'un arbre rouge-noir à n *sommets est au plus* $2 \log n$

la hauteur d'un AVL à n *sommets est au plus* $1.44 \log n$

Assurer une hauteur logarithmique ?

i.e. contraindre les ABR à rester « raisonnablement » équilibrés

- les arbres rouges-noirs
- les arbres AVL
- ...

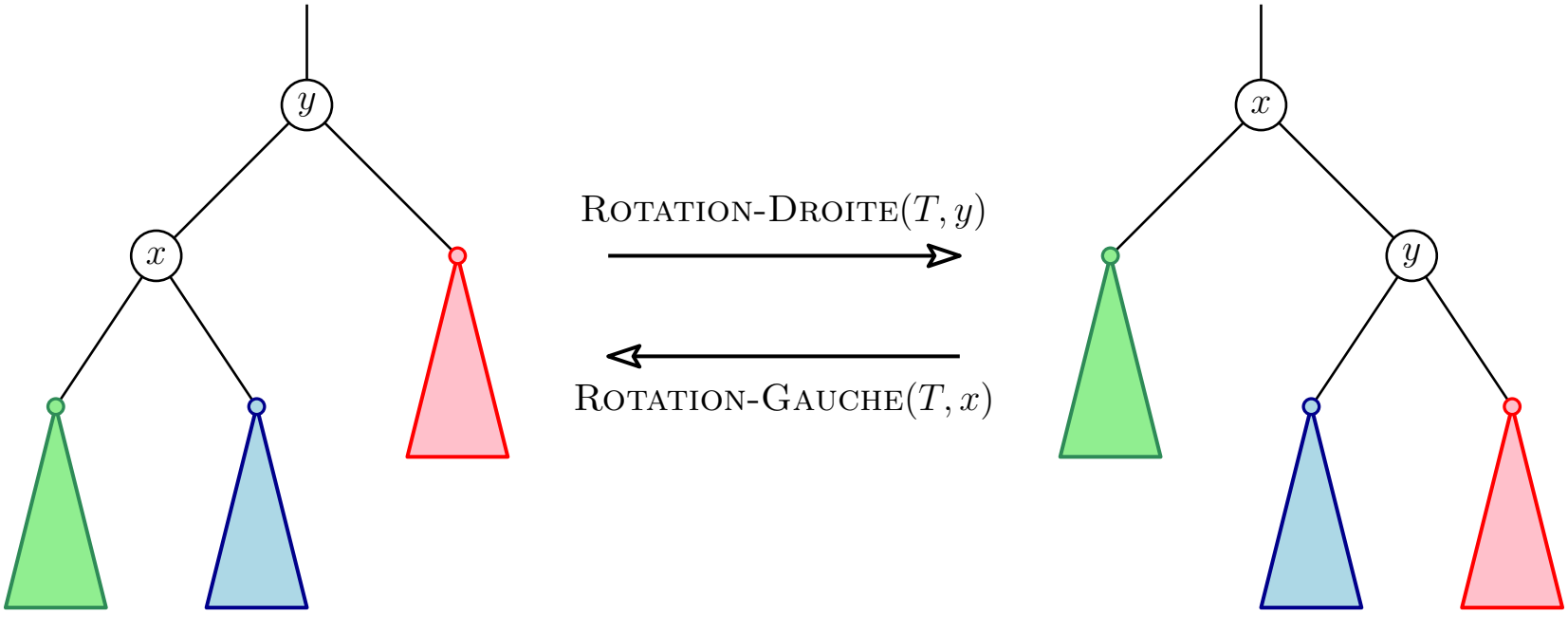
Théorème

la hauteur d'un arbre rouge-noir à n *sommets est au plus* $2 \log n$

la hauteur d'un AVL à n *sommets est au plus* $1.44 \log n$

Inconvénient : les opérations d'insertion et de suppression sont plus complexes

Outils de rééquilibrage : les rotations



D'autres arbres « triés » : les tas

tas-max

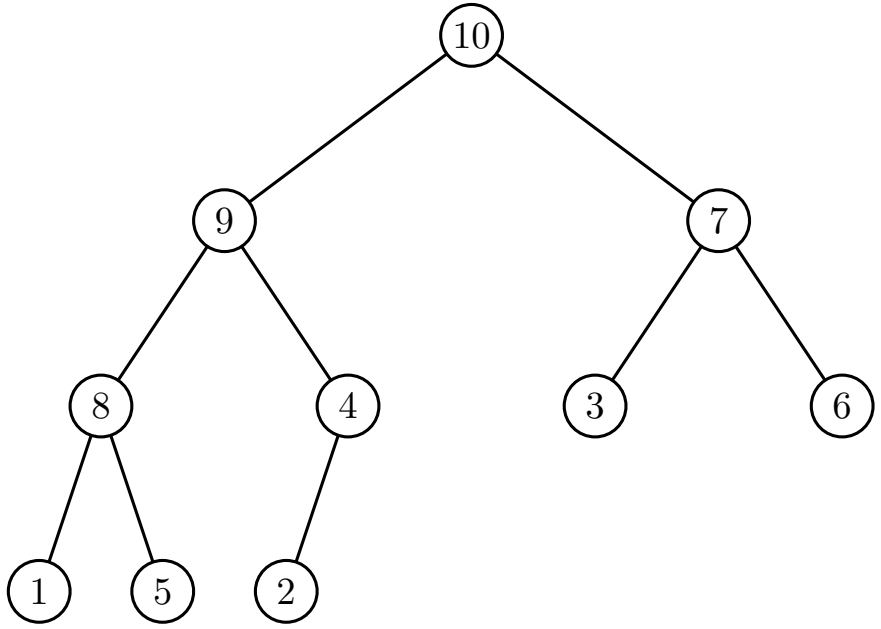
arbre binaire « presque parfait » tel qu'en chaque nœud, l'étiquette est supérieure à celles de ses fils

D'autres arbres « triés » : les tas

tas-max

arbre binaire « presque parfait » tel qu'en chaque nœud, l'étiquette est supérieure à celles de ses fils

arbre binaire presque parfait : dont tous les niveaux sont totalement remplis sauf éventuellement le dernier (qui est rempli depuis la gauche)



Quel est l'intérêt des tas-max ?

accéder en temps constant à l'élément (de priorité) maximal(e)

Quel est l'intérêt des tas-max ?

accéder en temps constant à l'élément (de priorité) maximal(e) – à la racine

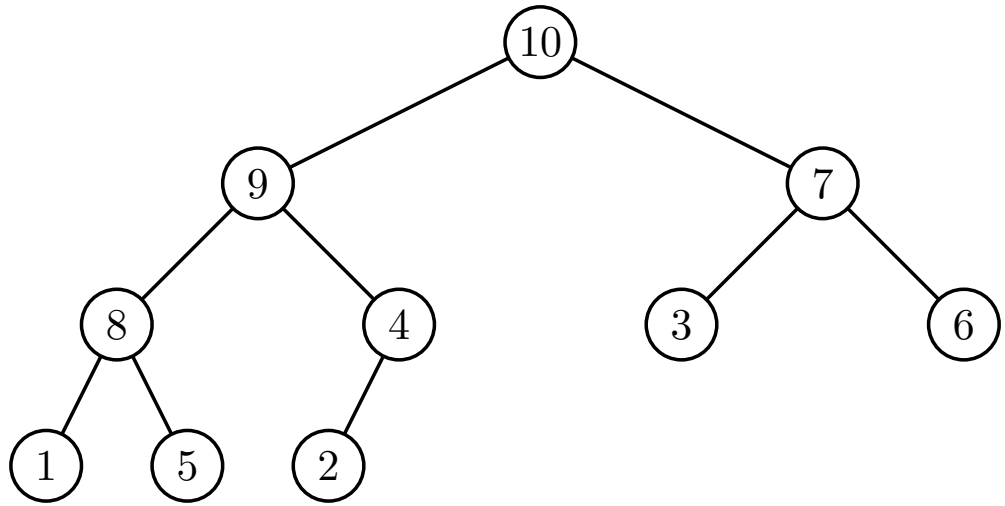
Quel est l'intérêt des tas-max ?

accéder en temps constant à l'élément (de priorité) maximal(e) – à la racine

hauteur optimale : $\log n$ (ou plus exactement $\lfloor \log n \rfloor$)

Quel est l'intérêt des tas-max ?

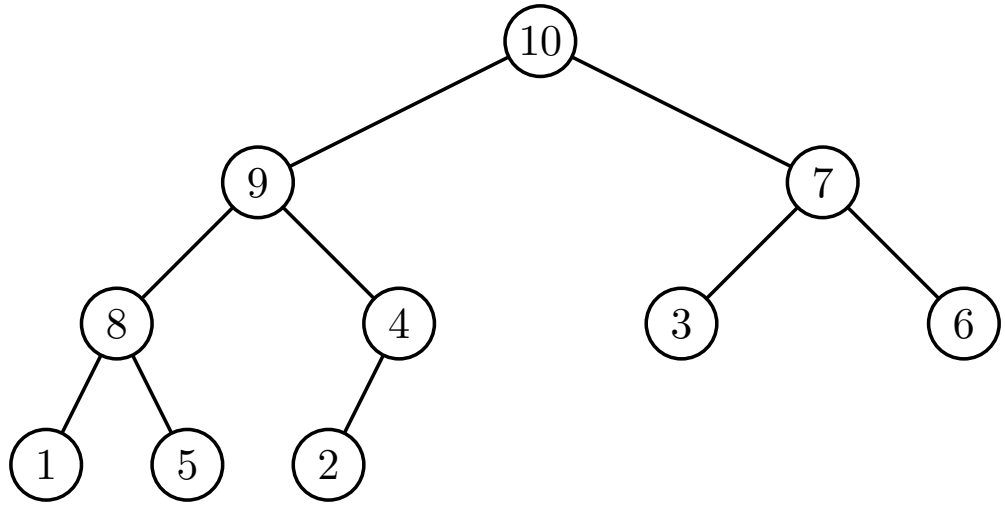
très facile à représenter par un tableau :



Quel est l'intérêt des tas-max ?

très facile à représenter par un tableau :

- stocker les nœuds dans l'ordre du parcours en largeur

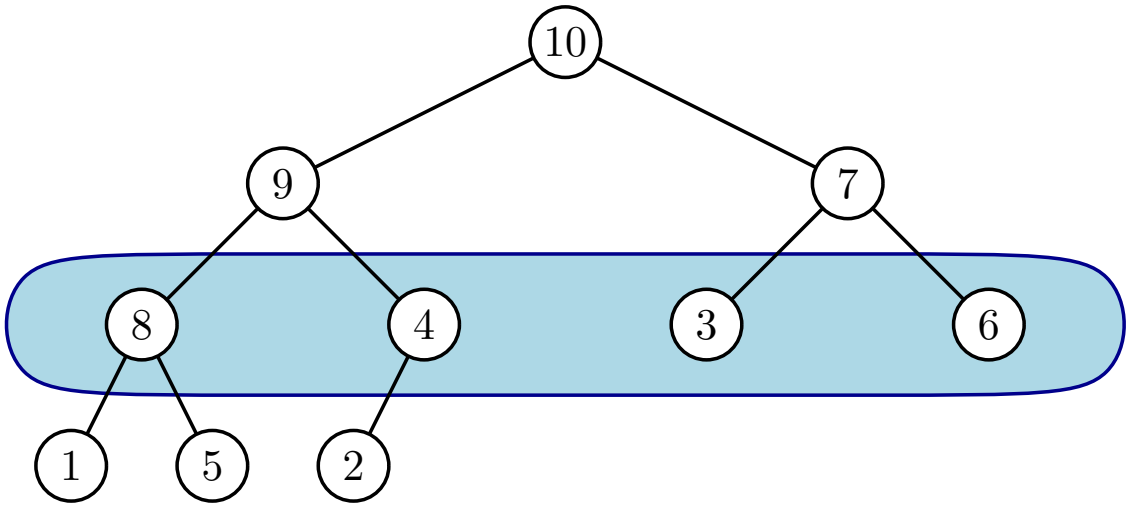


10	9	7	8	4	3	6	1	5	2

Quel est l'intérêt des tas-max ?

très facile à représenter par un tableau :

- stocker les nœuds dans l'ordre du parcours en largeur
- le niveau h est stocké entre les positions 2^h et $2^{h+1} - 1$

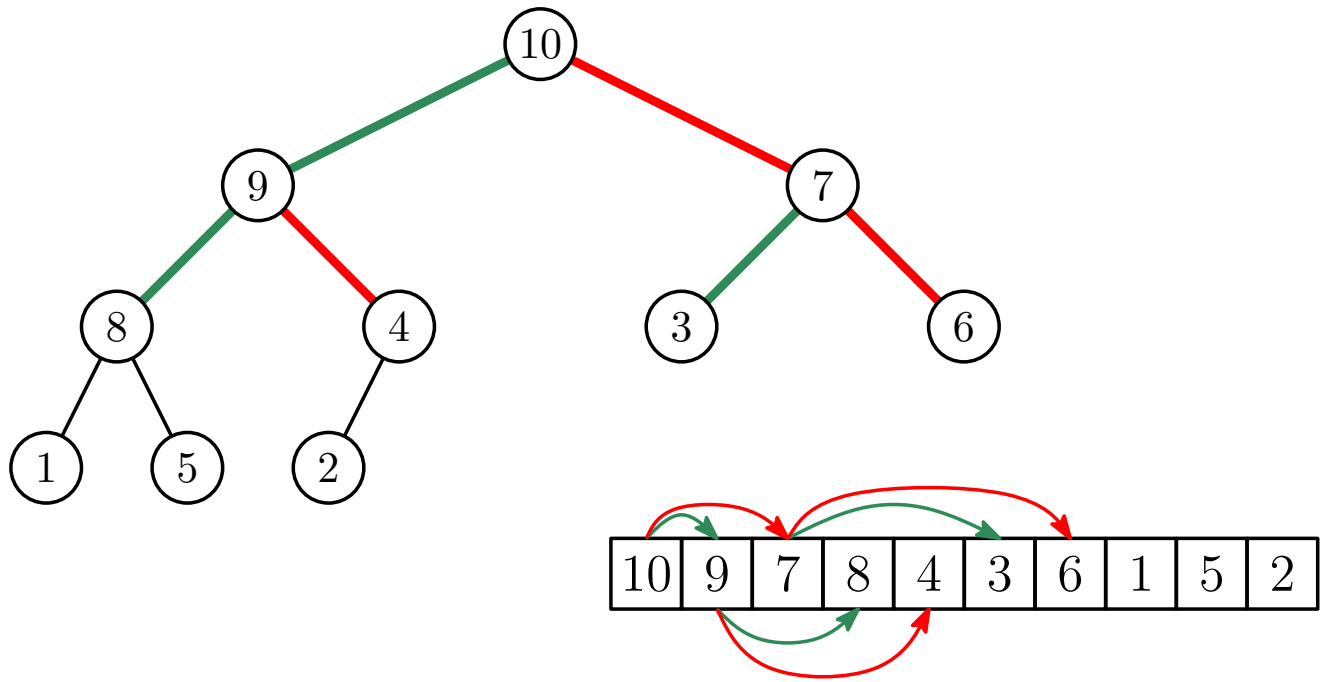


10	9	7	8	4	3	6	1	5	2

Quel est l'intérêt des tas-max ?

très facile à représenter par un tableau :

- stocker les nœuds dans l'ordre du parcours en largeur
- le niveau h est stocké entre les positions 2^h et $2^{h+1} - 1$

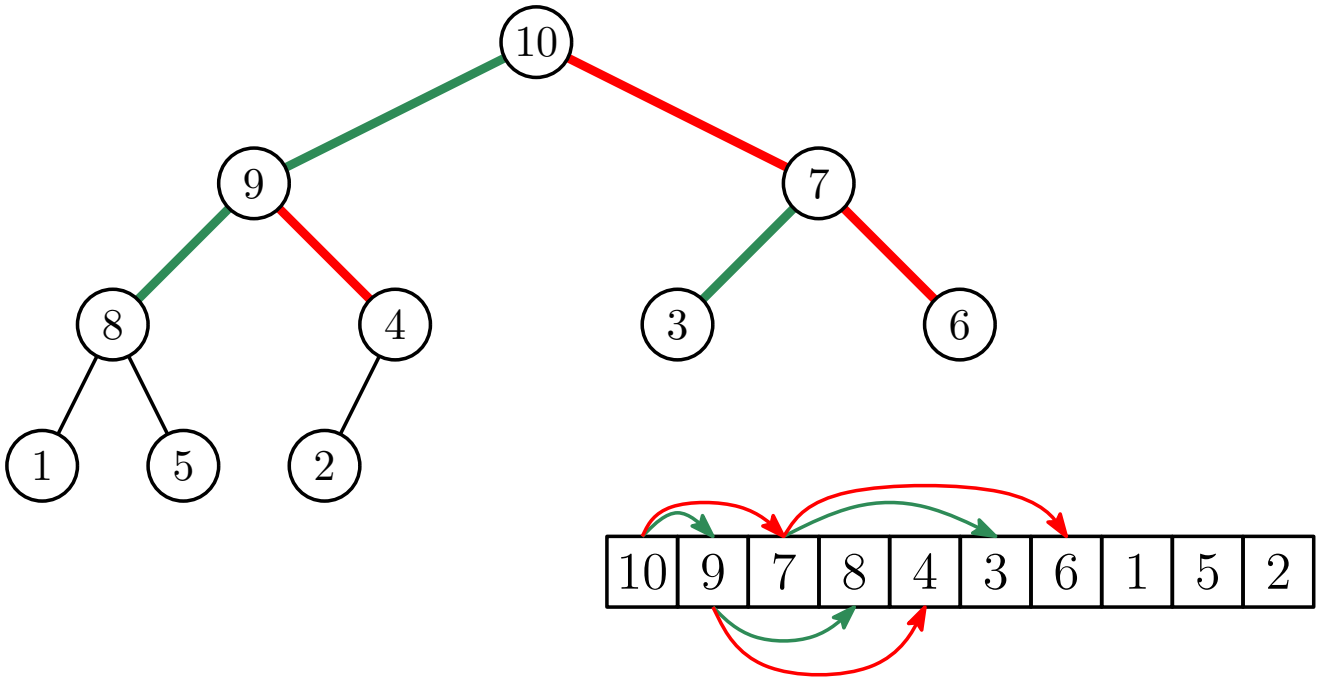


Quel est l'intérêt des tas-max ?

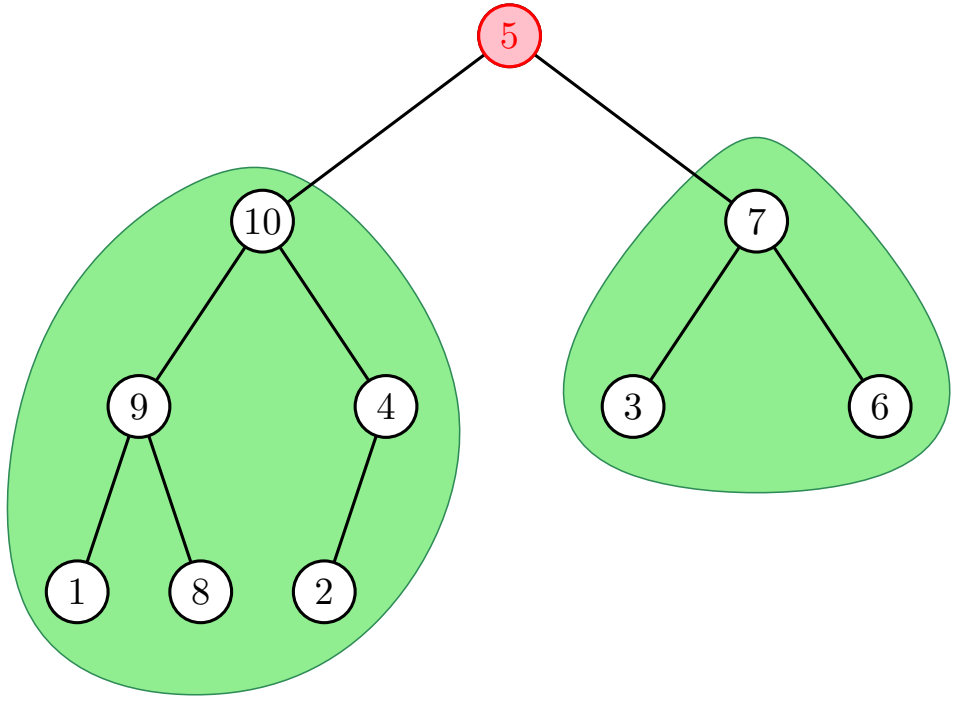
très facile à représenter par un tableau :

- stocker les nœuds dans l'ordre du parcours en largeur
- le niveau h est stocké entre les positions 2^h et $2^{h+1} - 1$
- `pere`$(i) = \lfloor i/2 \rfloor$, `gauche`$(i) = 2i$, `droit`$(i) = 2i + 1$

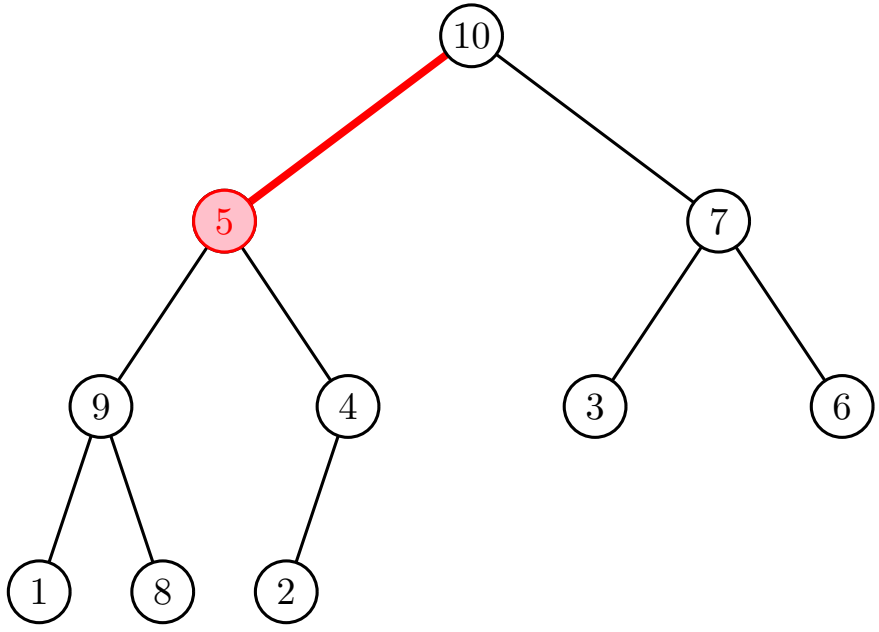
(remarque : les indices commencent à 1)



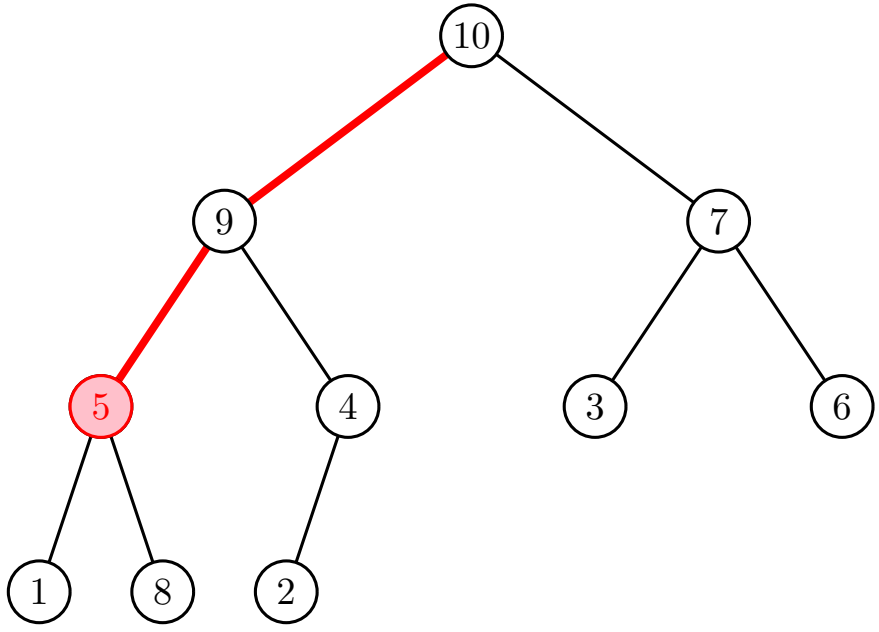
Préserver la propriété de tas-max



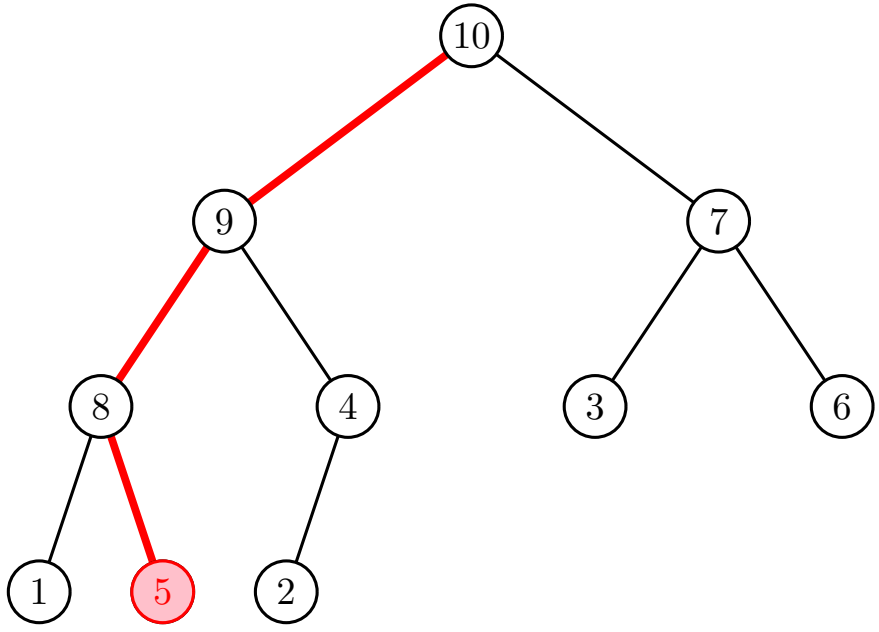
Préserver la propriété de tas-max



Préserver la propriété de tas-max



Préserver la propriété de tas-max



Préserver la propriété de tas-max

Si les sous-arbres du nœud d'indice i sont des tas-max :

```
def entasser_max(T, i) :
  max, l, r = i, gauche(i), droite(i)
  if l < len(T) and T[l] > T[i] : max = l
  if r < len(T) and T[r] > T[max] : max = r
  if max != i :
    T[i], T[max] = T[max], T[i]
    entasser_max(T, max)
```

Complexité : $\Theta(\log n)$ au pire

Transformer un tableau en tas-max

remarque : les feuilles sont des tas-max

Transformer un tableau en tas-max

remarque : les feuilles sont des tas-max

```
def creer_tas_max(T) :
  for i in range(len(T)//2, 0, -1) : # parcours à l'envers
    entasser_max(T, i)
```

Complexité : $\Theta(n)$ dans tous les cas

Démonstration : `entasser_max()` est appelé une fois sur chaque nœud, et a une complexité en $\Theta(h)$ pour chaque nœud (racine d'un sous-arbre) de hauteur h (donc de profondeur $\log n - h$).

La complexité totale est donc (de l'ordre de) la somme de hauteurs des nœuds.

Dans la suite, on donne deux preuves du fait que cette somme est en $\Theta(n)$.

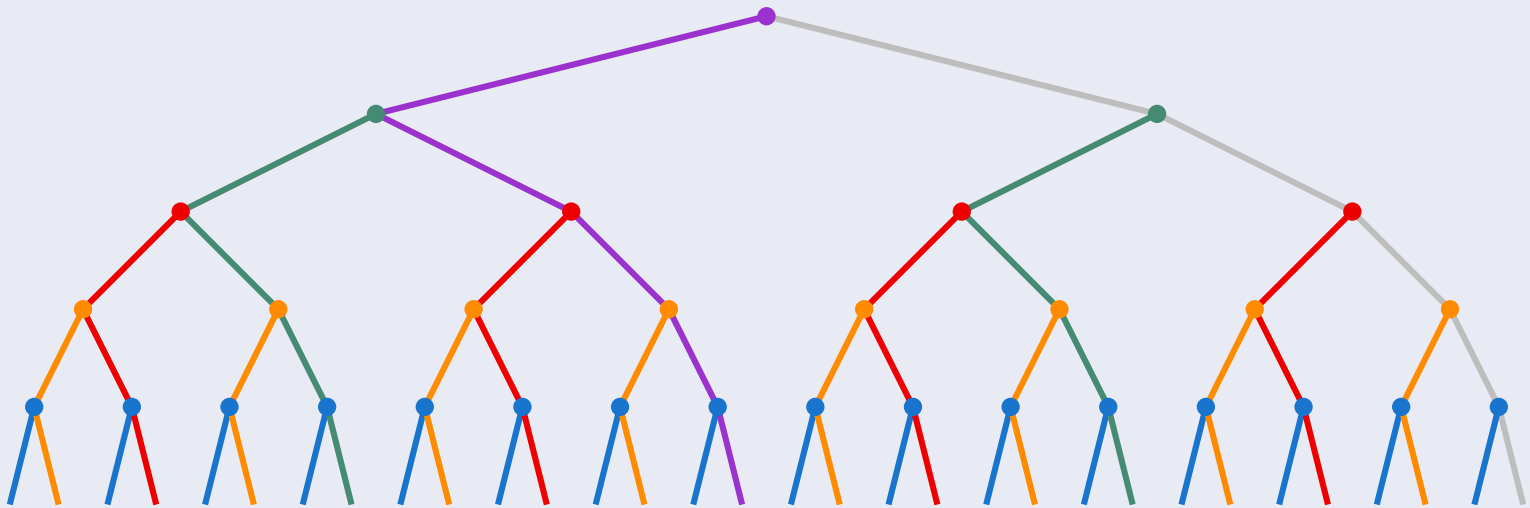
Preuve 1, par le calcul : comme le nombre de nœuds de hauteur h vaut $\frac{n}{2^h}$, on obtient :

$$C(n) = \sum_{h=0}^{\log n} \frac{n}{2^h} h \leqslant n \sum_{h=0}^{+\infty} \frac{h}{2^h}$$

et on peut montrer que cette dernière somme converge vers 2. □

Somme des hauteurs des nœuds

Preuve 2, combinatoirement : à chaque nœud, on associe la branche descendante « *gauche* – *droite* – *droite* – *droite* – $\cdots$ – *droite* » : un nœud de hauteur h est ainsi associé à un chemin dans l'arbre de longueur h, donc la somme des hauteurs est égale à la somme des longueurs de ces chemins.



(*code-couleur : une couleur par niveau, le nœud et le chemin correspondant sont colorés de la même couleur*)

Inversement, chaque arête de l'arbre est associée à exactement un nœud, sauf celles qui appartiennent à la branche droite issue de la racine (en gris sur le dessin). La somme des longueurs des chemins est donc inférieure au nombre total d'arêtes de l'arbre, qui est lui-même égal (à 1 près) au nombre total de sommets (nœuds + feuilles) de l'arbre. □

Trier avec un tas-max

```
def tri_par_tas(T) :
  creer_tas_max(T)
  for i in range(len(T)-1,0, -1) :
    T[1], T[i] = T[i], T[1]
    entasser_max(T, 1, i)
  return T

def entasser_max(T, i, borne = None) :
  if borne == None : borne = len(T)
  # ... comme la première version
  if max != i :
    T[i], T[max] = T[max], T[i]
    entasser_max(T, max, borne)
```

Complexité : $\Theta(n \log n)$ au pire

Implémenter une file de priorité

structure destinée à gérer les priorités, par exemple pour l'ordonnancement de tâches sur un ordinateur

opérations supportées

- `insertion(F, x)`
- `maximum(F)`
- `extraction_max(F)`
- `augmenter_priorité(F, x, k)`

Implémenter une file de priorité

structure destinée à gérer les priorités, par exemple pour l'ordonnancement de tâches sur un ordinateur

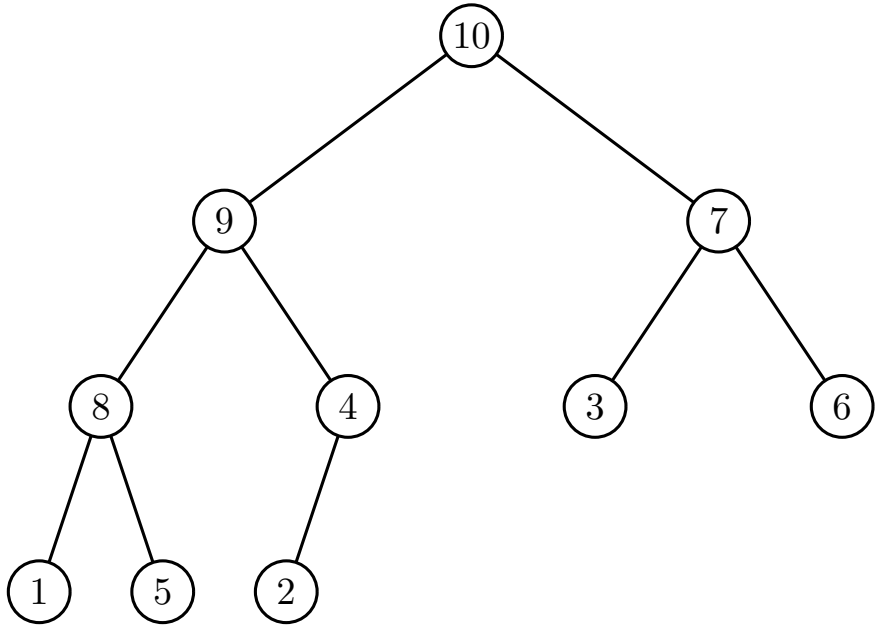
opérations supportées

- `insertion(F, x)`
- `maximum(F)`
- `extraction_max(F)`
- `augmenter_priorité(F, x, k)`

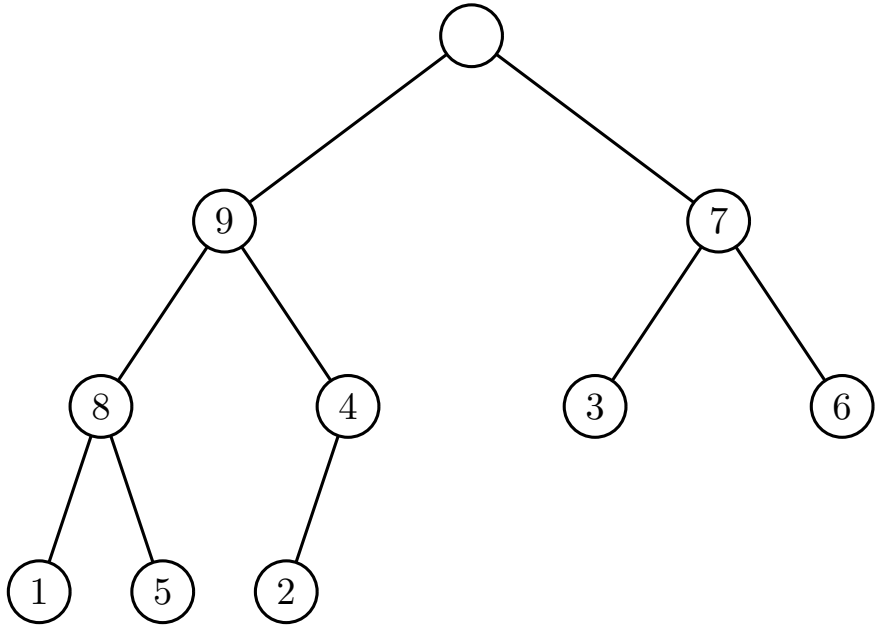
les tas-max sont particulièrement bien adaptés :

- recherche du maximum en temps constant
- les autres opérations se font en temps $\Theta(\log n)$

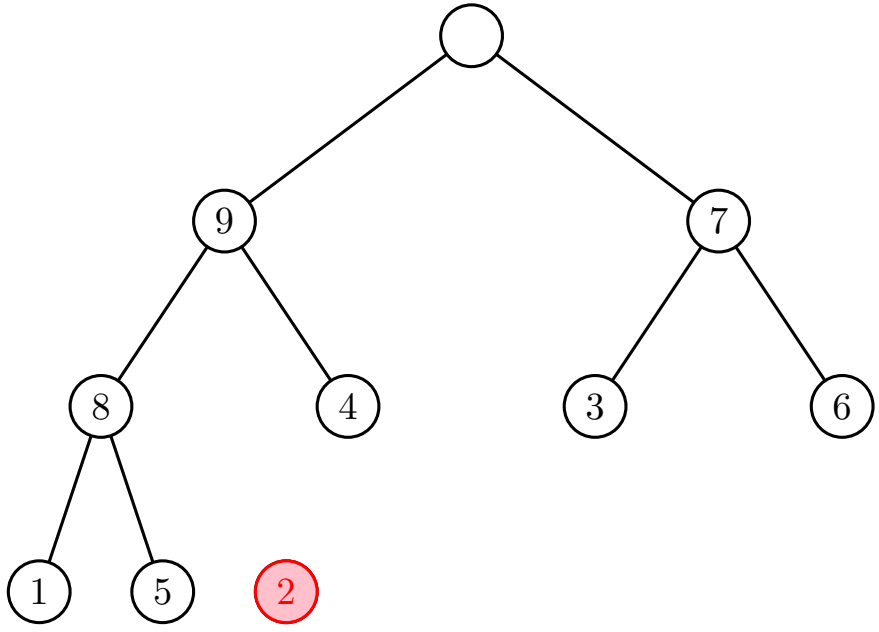
Extraction du maximum



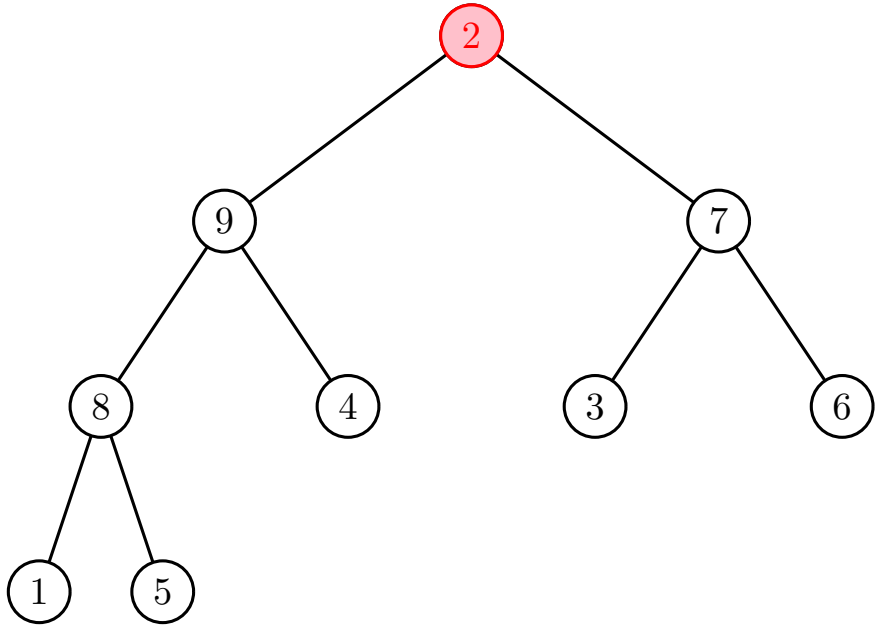
Extraction du maximum



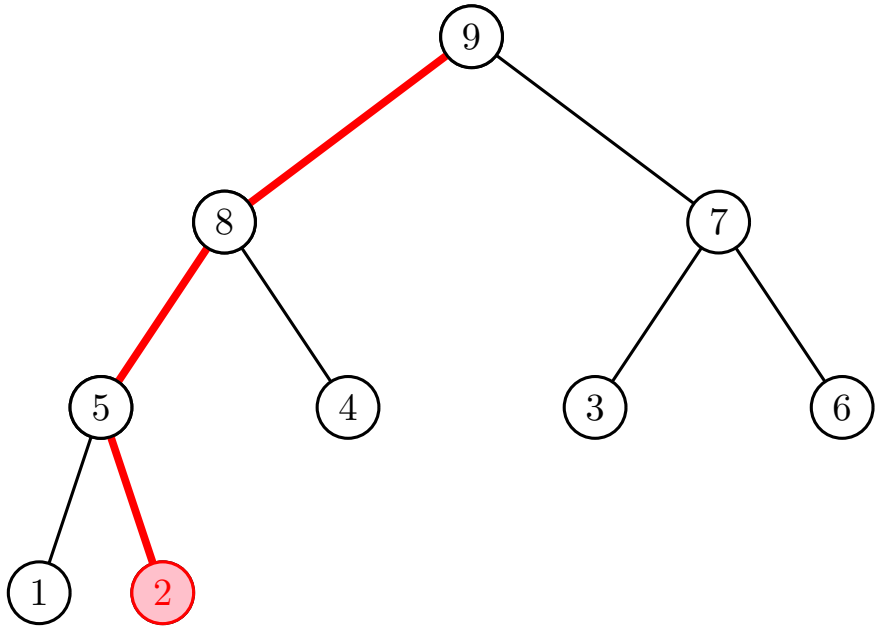
Extraction du maximum



Extraction du maximum



Extraction du maximum

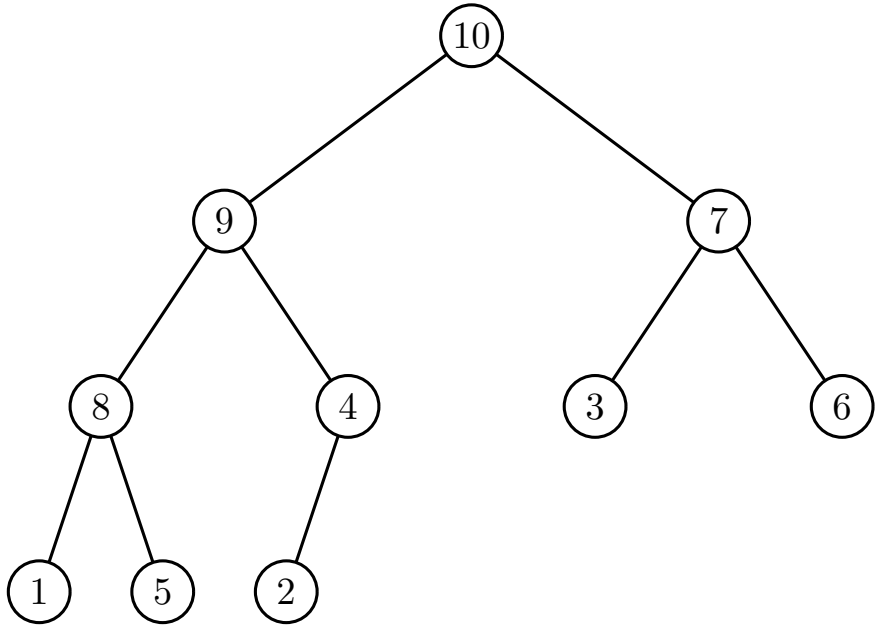


Extraction du maximum

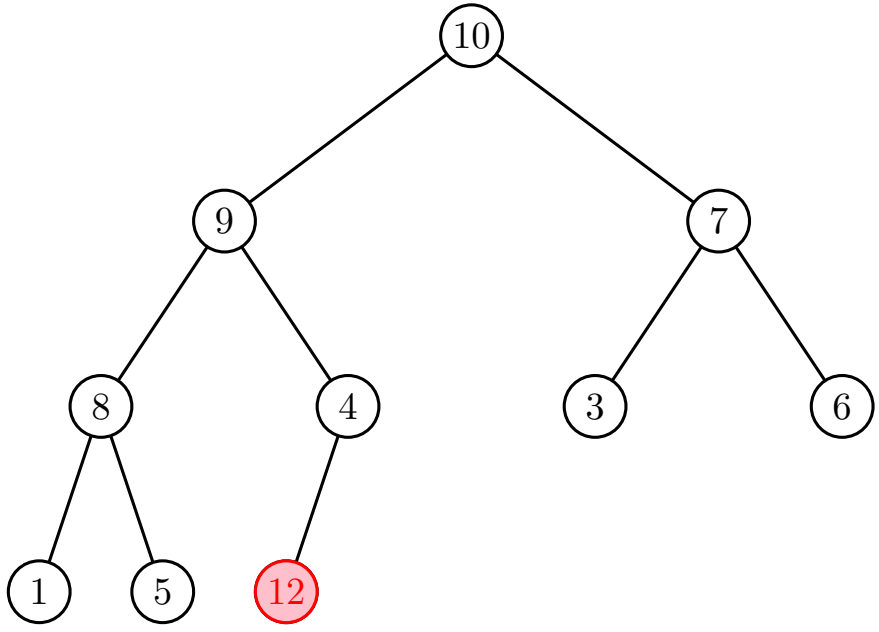
```
def extraction_maximum(F) :
  max = F[1]
  F[1] = F.pop()  # déplacement et redimensionnement du tableau
  entasser_max(F, 1)
  return max
```

Complexité : $\Theta(\log n)$ au pire

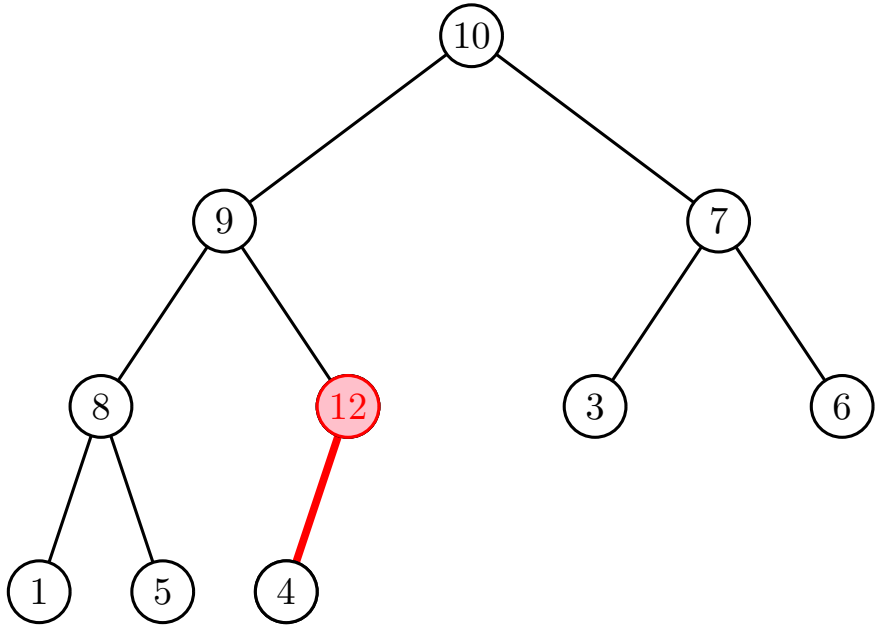
Augmenter une priorité



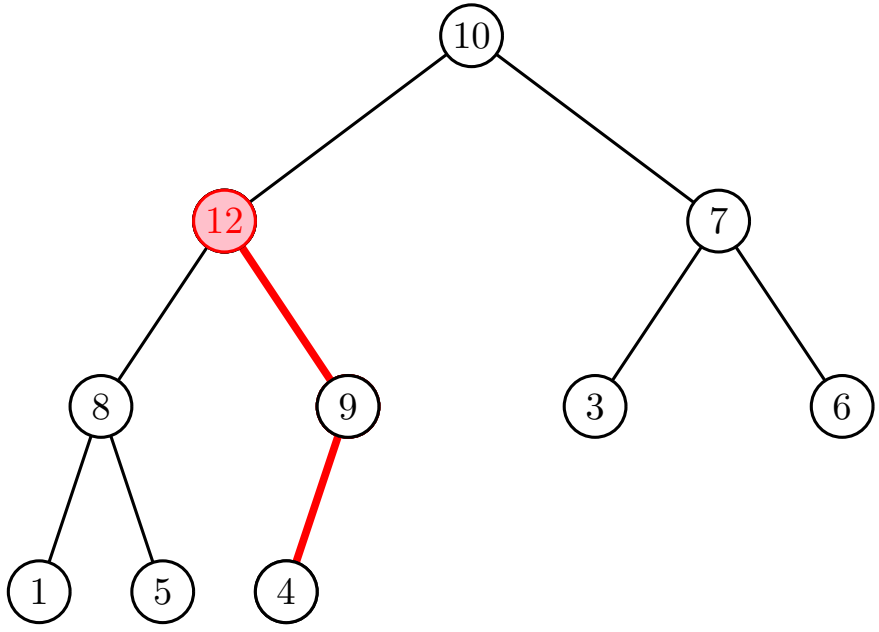
Augmenter une priorité



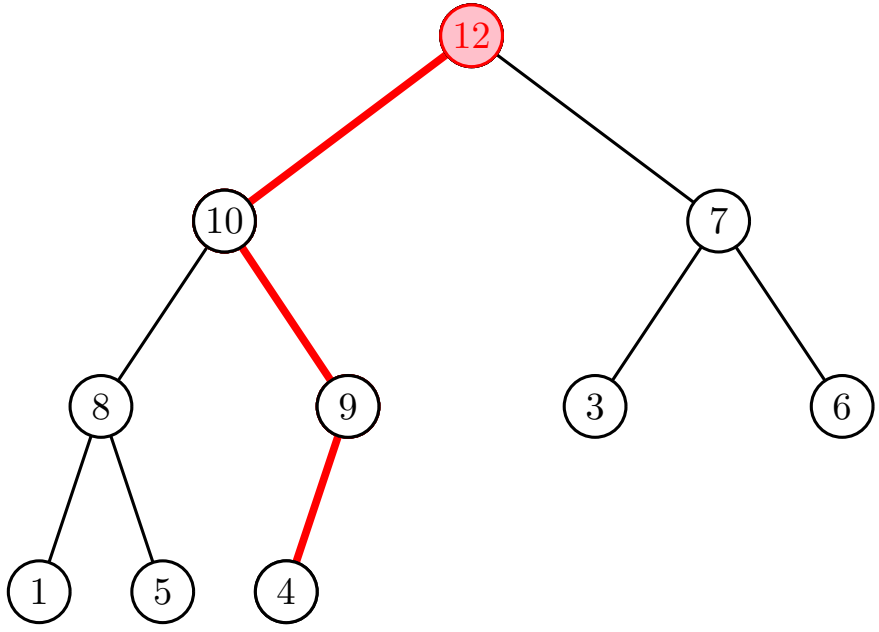
Augmenter une priorité



Augmenter une priorité



Augmenter une priorité



Augmenter une priorité

```
def augmenter_cle(F, i, cle) :
  if cle < F[i] : return
  F[i] = cle
  while i > 1 and F[pere(i)] < F[i] :
    F[i], F[pere(i)] = F[pere(i)], F[i]
    i = pere(i)
```

Complexité : $\Theta(\log n)$ au pire

Insérer une clé

```
def inserer_cle(F, cle) :
  F.append(MIN) # valeur inférieure à toutes les clés
  augmenter_cle(F, len(F)-1, cle)
```

Complexité : $\Theta(\log n)$ au pire