Module EA4 – Éléments d'Algorithmique

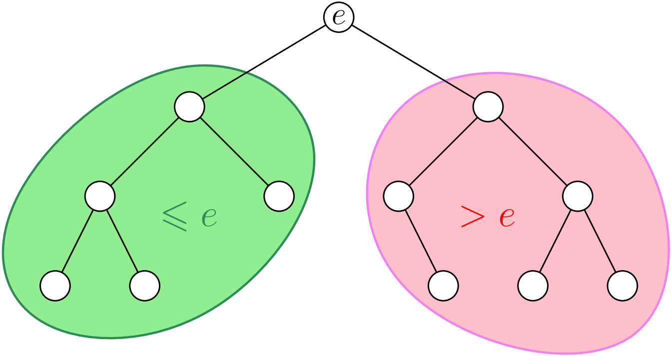
Dominique Poulalhon
dominique.poulalhon@liafa.univ-paris-diderot.fr

Université Paris Diderot
L2 Informatique
Année universitaire 2014-2015

les arbres binaires de recherche (ABR)

en chaque nœud, l'étiquette est comprise entre

- les étiquettes du sous-arbre gauche (plus petites) et
- celles du sous-arbre droit (plus grandes)



Théorème

la liste triée des éléments d'un ABR de taille n *peut être obtenue en temps* $\Theta(n)$ *par un parcours en profondeur infixe.*

Théorème

la recherche, l'ajout et la suppression d'un élément dans un ABR de hauteur h *se font en temps* $\Theta(h)$ *au pire.*

Corollaire

la construction d'un ABR de taille n *a un coût* $O(nh)$, *si* h *est la hauteur de l'arbre obtenu.*

Hauteur d'un ABR

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

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

Hauteur d'un ABR

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

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

chaque arbre binaire à n sommets admet un unique étiquetage par $\{1, \ldots, n\}$ respectant les contraintes d'un ABR la hauteur $h(A)$ d'un arbre binaire A à n sommets vérifie :

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

chaque arbre binaire à n sommets admet un unique étiquetage par $\{1,\ldots,n\}$ respectant les contraintes d'un ABR

Théorème (admis)

la hauteur moyenne d'un arbre binaire choisi uniformément parmi les arbres binaires à n nœuds est en $\Theta(\sqrt{n})$.

Hauteur d'un ABR

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

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

chaque arbre binaire à n sommets admet un unique étiquetage par $\{1, \ldots, n\}$ respectant les contraintes d'un ABR

Théorème (admis)

la hauteur moyenne d'un arbre binaire choisi uniformément parmi les arbres binaires à n nœuds est en $\Theta(\sqrt{n})$.

et pourtant...

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	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n)$	? ?

Comment assurer un coût logarithmique ?

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

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

Comment assurer un coût 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

Comment assurer un coût 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

Comment assurer un coût 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 $1.44 \log n$

Comment assurer un coût logarithmique ?

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

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

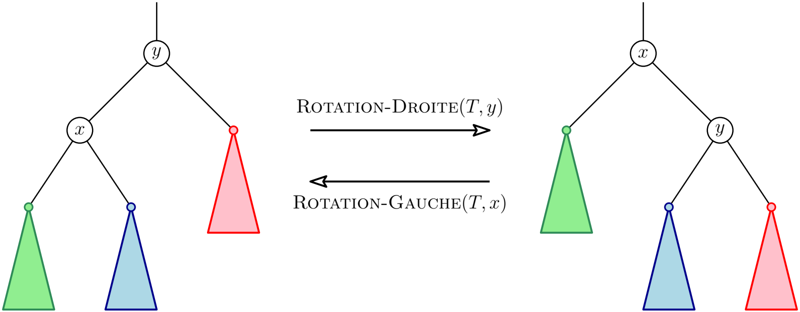
Théorème

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

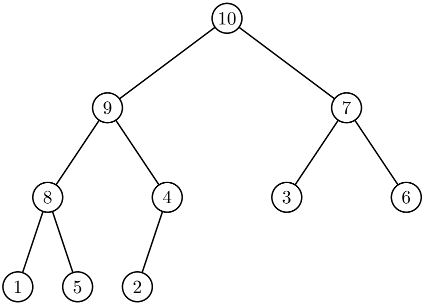
arbre binaire « presque parfait » tel qu'en chaque nœud, l'étiquette est supérieure à celles de ses fils

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

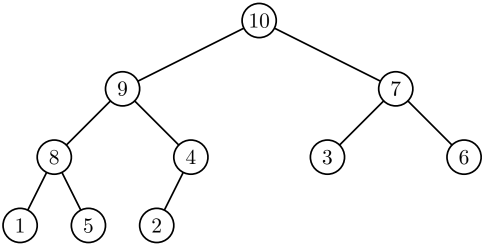
Quel est l'intérêt des tas-max ?

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 ?

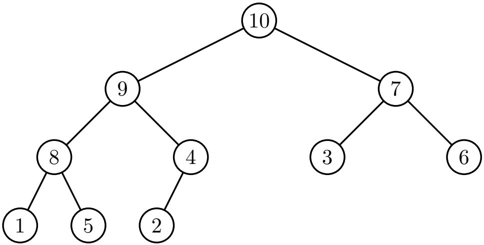
très facile à représenter par un tableau :



Quel est l'intérêt des tas-max ?

très facile à représenter par un tableau :

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

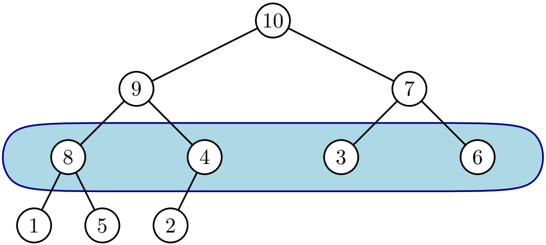


10	9	7	8	4	3	6	1	5	2

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 $2^{h+1} - 1$

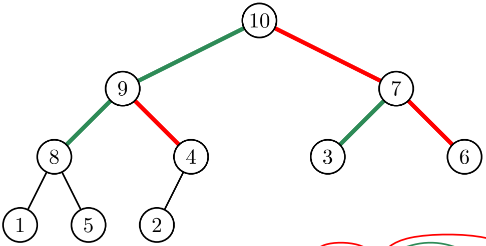


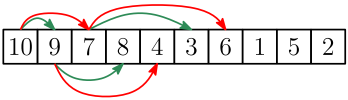
10	9	7	8	4	3	6	1	5	2

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 $2^{h+1} - 1$

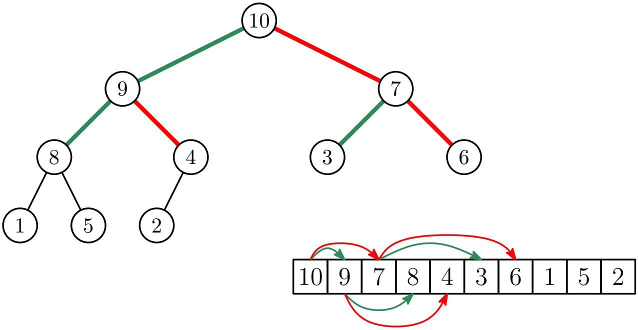




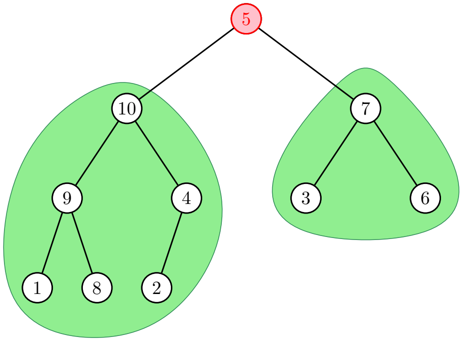
Quel est l'intérêt des tas-max ?

très facile à représenter par un tableau :

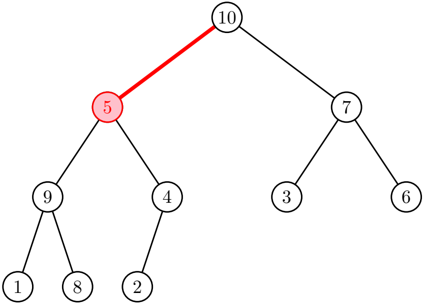
- stocker les nœuds dans l'ordre du parcours en largeur
- le niveau h est stocké entre les positions 2^h et $2^{h+1} - 1$
- `pere`$(i) = \lfloor i/2 \rfloor$, `gauche`$(i) = 2i$, `droit`$(i) = 2i + 1$



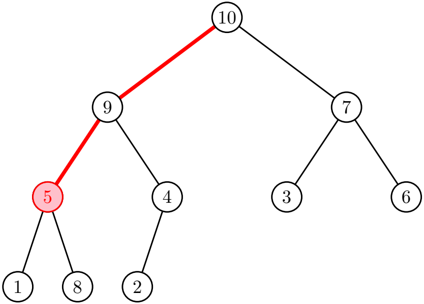
Comment préserver la propriété de tas-max ?



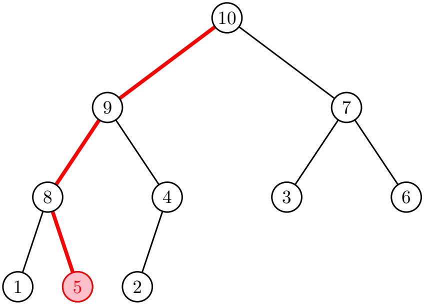
Comment préserver la propriété de tas-max ?



Comment préserver la propriété de tas-max ?



Comment préserver la propriété de tas-max ?



Comment 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

```
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

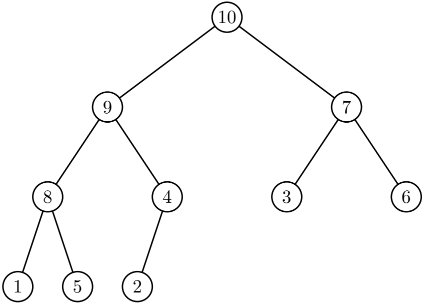
opérations supportées

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

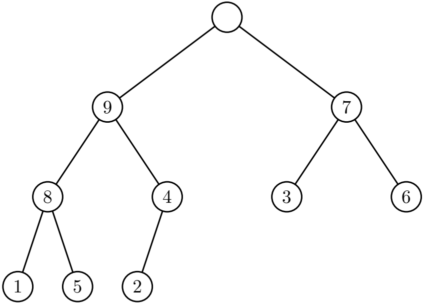
les tas-max sont particulièrement bien adaptés :

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

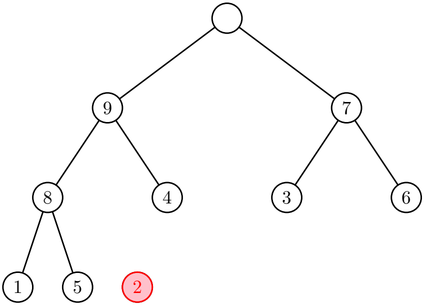
Extraction du maximum



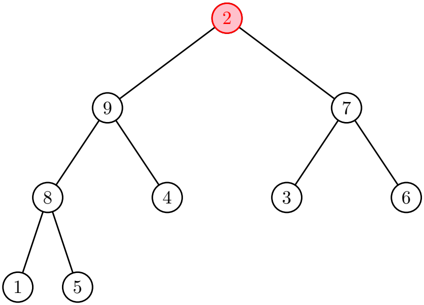
Extraction du maximum



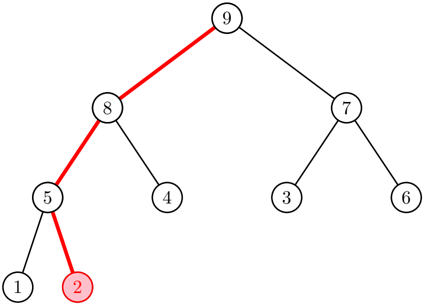
Extraction du maximum



Extraction du maximum



Extraction du maximum

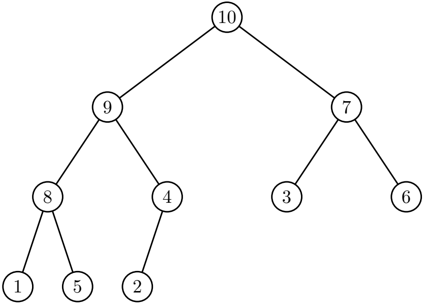


Extraction du maximum

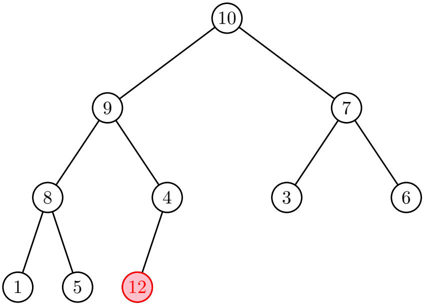
```
def extraction_maximum(F) :
    max = F[1]
    F[1] = F.pop()  # déplacement et redimensionnement du tableau
    entasser_max(F, 1)
    return max
```

Complexité : $\Theta(\log n)$ au pire

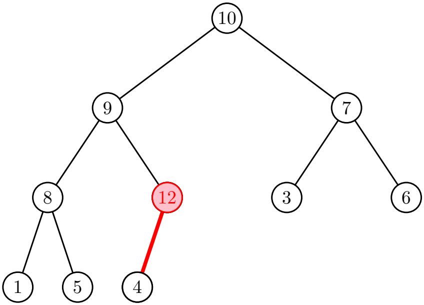
Augmenter une priorité



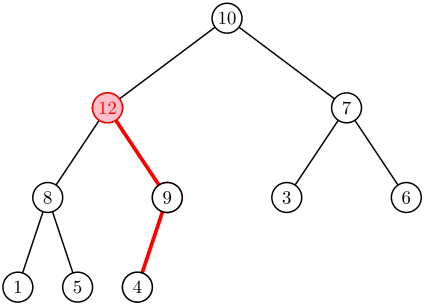
Augmenter une priorité



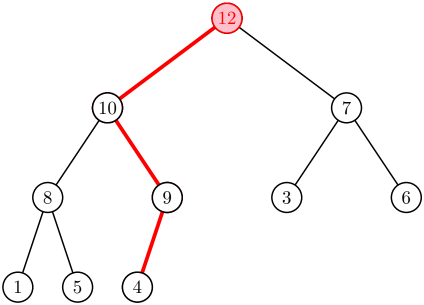
Augmenter une priorité



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