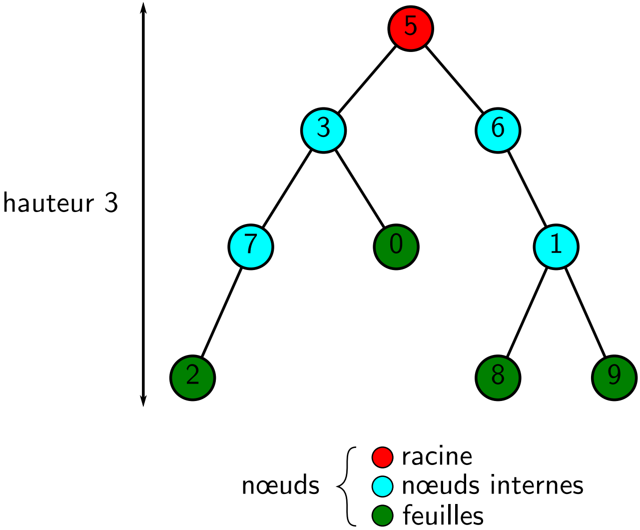
- HAI403I: Algorithme 3, le retour -

Cours 3
Structures de données arborescentes :
arbres binaires de recherche et tas

L2 Informatique
Université de Montpellier

Exemple et vocabulaire



Définition récursive

Un arbre binaire est défini récursivement :

- l'arbre vide $\emptyset$ est un arbre binaire ;
- un arbre non vide est constitué d'une racine, d'un sous-arbre gauche G et d'un sous-arbre droit D qui sont eux-mêmes deux arbres binaires.

Définition récursive

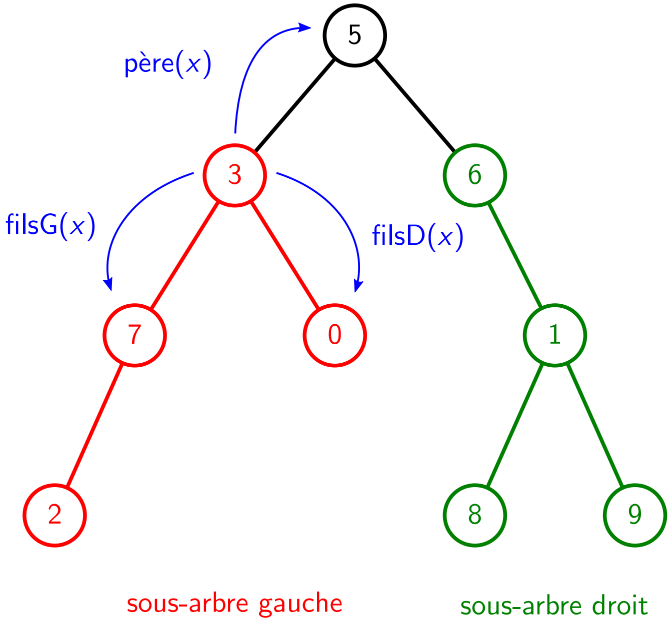
Un arbre binaire est défini récursivement :

- l'arbre vide $\emptyset$ est un arbre binaire ;
- un arbre non vide est constitué d'une racine, d'un sous-arbre gauche G et d'un sous-arbre droit D qui sont eux-mêmes deux arbres binaires.

Représentation informatique

- Un nœud x est soit
  - le nœud vide, noté $\emptyset$
  - défini par une valeur $\text{val}(x)$ et trois liens vers d'autres nœuds : $\text{père}(x)$, $\text{filsG}(x)$, $\text{filsD}(x)$ tels que
    - Si $\text{filsG}(x) \neq \emptyset$, $\text{père}(\text{filsG}(x)) = x$
    - Si $\text{filsD}(x) \neq \emptyset$, $\text{père}(\text{filsD}(x)) = x$
- Un arbre binaire A est donné par une racine $\text{rac}(A)$ qui est un nœud tel que $\text{père}(\text{rac}(A)) = \emptyset$.

Exemple

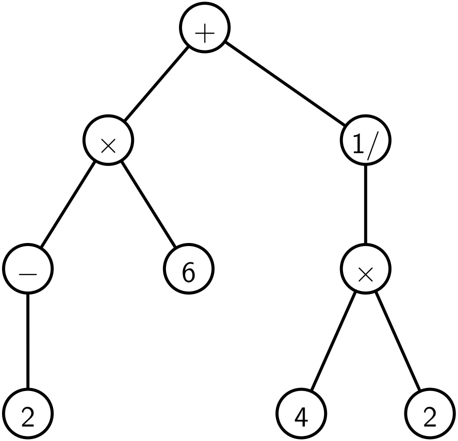


père(x)
5
3
6
filsG(x)
filsD(x)
7
0
1
2
8
9
sous-arbre gauche
sous-arbre droit

Utilité des arbres binaires

- Arbres binaires de recherche
- Tas

- Analyse syntaxique
- Bases de données
- Partition binaire de l'espace
- Tables de routage
- ...



$$(-2 \times 6) + 1/(4 \times 2)$$

Caractéristiques

Définition

- Un nœud x est une feuille si $\text{filsG}(x) = \emptyset$ et $\text{filsD}(x) = \emptyset$
- La hauteur h(x) d'un nœud x dans l'arbre A est définie récursivement par
  - Si $x = \text{rac}(A)$, $h(x) = 0$ $\quad (\Leftrightarrow \text{père}(x) = \emptyset)$
  - Sinon, $h(x) = 1 + h(\text{père}(x))$
- La hauteur d'un arbre A est $\mathrm{h(A)} = \max\{h(x) : x \in A\}$
- Le $\mathrm{k}^{\text{ème}}$ niveau de A est $\mathrm{N_k} = \{x : h(x) = k\}$
- Le sous-arbre gauche (resp. droit) de A est l'arbre dont la racine est le fils gauche (resp. droit) de la racine de A

Résultats structurels

Lemme

$$|N_k| = \#\{x : h(x) = k\} \leq 2^k$$

Preuve par récurrence sur k

- $k = 0 : \{x : h(x) = 0\} = \{\mathrm{rac}(A)\}$
- Chaque nœud de N_{k-1} a au plus 2 fils : Donc $|N_k| \leq 2|N_{k-1}| \leq 2 \cdot 2^{k-1} = 2^k$

Résultats structurels

Lemme

$$|N_k| = \#\{x : h(x) = k\} \leq 2^k$$

Preuve par récurrence sur k

- $k = 0 : \{x : h(x) = 0\} = \{\mathrm{rac}(A)\}$
- Chaque nœud de N_{k-1} a au plus 2 fils : Donc $|N_k| \leq 2|N_{k-1}| \leq 2 \cdot 2^{k-1} = 2^k$

Lemme

$h(A) + 1 \leq n(A) \leq 2^{h(A)+1} - 1$ *où* $n(A) =$ *nombre de nœuds de* A

Preuve $n(A) = \sum_{i=0}^{h(A)} |N_i|$ et $1 \leq |N_i| \leq 2^i$

$\rightsquigarrow h(A) + 1 \leq n(A) \leq \sum_{i=0}^{h(A)} 2^i = 2^{h(A)+1} - 1$

Résultats structurels

Lemme

$$|N_k| = \#\{x : h(x) = k\} \leq 2^k$$

Preuve par récurrence sur k

- $k = 0 : \{x : h(x) = 0\} = \{\mathrm{rac}(A)\}$
- Chaque nœud de N_{k-1} a au plus 2 fils : Donc $|N_k| \leq 2|N_{k-1}| \leq 2 \cdot 2^{k-1} = 2^k$

Lemme

$h(A) + 1 \leq n(A) \leq 2^{h(A)+1} - 1$ *où* $n(A) =$ *nombre de nœuds de* A

Preuve $n(A) = \sum_{i=0}^{h(A)} |N_i|$ et $1 \leq |N_i| \leq 2^i$

$\rightsquigarrow h(A) + 1 \leq n(A) \leq \sum_{i=0}^{h(A)} 2^i = 2^{h(A)+1} - 1$

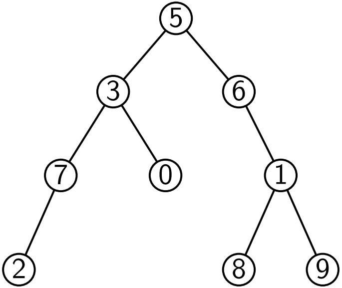
Corollaire

$$\lfloor \log(n(A)) \rfloor \leq h(A) < n(A)$$

Parcours d'un arbre binaire

Algorithme :
PARCOURSINFIXE(x)

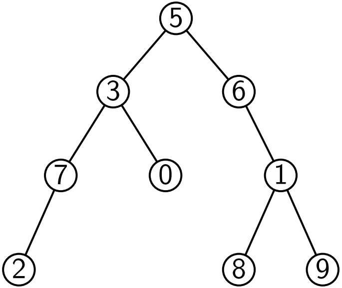
si $x \neq \emptyset$ alors
- PARCOURSINFIXE(filsG(x))
- Afficher val(x)
- PARCOURSINFIXE(filsD(x))



Parcours d'un arbre binaire

Algorithme :
ParcoursInfixe(x)

si $x \neq \emptyset$ alors
    ParcoursInfixe(filsG(x))
    Afficher val(x)
    ParcoursInfixe(filsD(x))

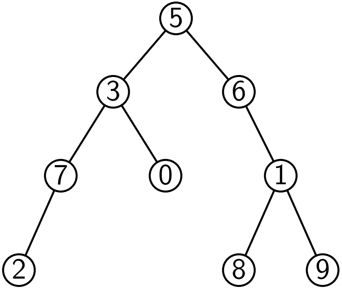


- Affichage : 2 7 3 0 5 6 8 1 9

Parcours d'un arbre binaire

Algorithme :
ParcoursInfixe(x)

si $x \neq \emptyset$ alors
    ParcoursInfixe(filsG(x))
    Afficher val(x)
    ParcoursInfixe(filsD(x))



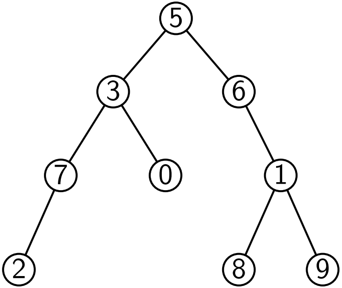
- Affichage : 2 7 3 0 5 6 8 1 9
- Complexité en $O(n(A))$

  Preuve $\mathcal{P}_n$: l'algo. effectue $5n$ appels de fonctions
  - $n = 0$: pas trop dur...
  - Supp. $\mathcal{P}_k$ pour tout $k < n(A)$ et soit n_G et n_D le nb de nœuds dans les sous-arbres gauche et droit. Dans les deux appels récursifs, $5n_G$ et $5n_D$ appels de fonctions, donc au total $5n_G + 5n_D + 5$ appels. Or $n(A) = n_G + n_D + 1$, d'où le résultat.

Parcours d'un arbre binaire

Algorithme :
ParcoursInfixe(x)

si $x \neq \emptyset$ alors
    ParcoursInfixe(filsG(x))
    Afficher val(x)
    ParcoursInfixe(filsD(x))



- Affichage : 2 7 3 0 5 6 8 1 9
- Complexité en $O(n(A))$
- Appel de la fonction : ParcoursInfixe(rac(A))
- Variantes : ParcoursPrefixe et ParcoursSuffixe

Algorithme générique sur les arbres binaires

Appel de ALGO(rac(A)) avec

Algorithme : ALGO(x)

res $\leftarrow$ valeur pour l'arbre vide
si $x \neq \emptyset$ alors
  res$_G$ $\leftarrow$ ALGO(filsG(x))
  res$_D$ $\leftarrow$ ALGO(filsD(x))
  res $\leftarrow$ f(res, res$_G$, res$_D$, x)
retourner *res*

Algorithme générique sur les arbres binaires

Appel de $\textsc{Algo}(\text{rac}(A))$ avec

Algorithme : $\textsc{Algo}(x)$

res $\leftarrow$ valeur pour l'arbre vide
si $x \neq \emptyset$ alors
    $\text{res}_G \leftarrow \textsc{Algo}(\text{filsG}(x))$
    $\text{res}_D \leftarrow \textsc{Algo}(\text{filsD}(x))$
    res $\leftarrow f(\text{res}, \text{res}_G, \text{res}_D, x)$
retourner *res*

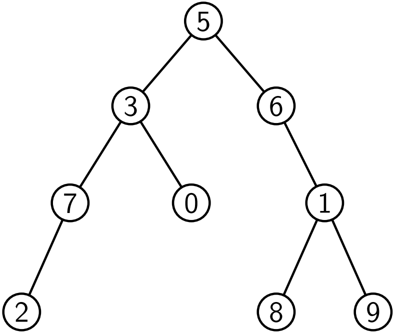
Lemme

L'algorithme générique sur les arbres binaires a une complexité $O(n(A))$ si le calcul de f a une complexité en temps en $O(1)$.

Exemples d'utilisation

Algorithme : $\textsc{Minimum}(x)$

$m \leftarrow +\infty$
si $x \neq \emptyset$ alors
  $m_G \leftarrow \textsc{Minimum}(\text{filsG}(x))$
  $m_D \leftarrow \textsc{Minimum}(\text{filsD}(x))$
  $m \leftarrow \min(m_G, m_D, \text{val}(x))$
retourner m

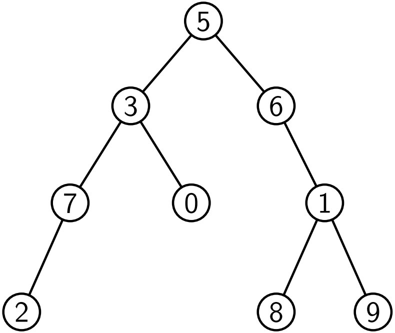


Exemples d'utilisation

Algorithme : $\textsc{Minimum}(x)$

$m \leftarrow +\infty$
si $x \neq \emptyset$ alors
  $m_G \leftarrow \textsc{Minimum}(\text{filsG}(x))$
  $m_D \leftarrow \textsc{Minimum}(\text{filsD}(x))$
  $m \leftarrow \min(m_G, m_D, \text{val}(x))$
retourner m

$$\textsc{Min}(A_5) = \min(5, \textsc{Min}(A_3), \textsc{Min}(A_6))$$

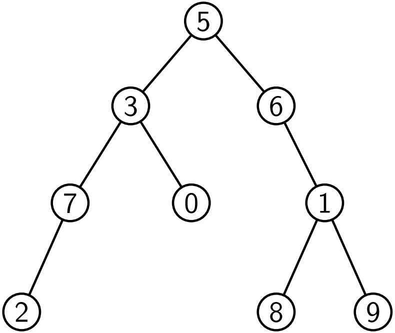


Exemples d'utilisation

Algorithme : $\textsc{Minimum}(x)$

$m \leftarrow +\infty$
si $x \neq \emptyset$ alors
  $m_G \leftarrow \textsc{Minimum}(\text{filsG}(x))$
  $m_D \leftarrow \textsc{Minimum}(\text{filsD}(x))$
  $m \leftarrow \min(m_G, m_D, \text{val}(x))$
retourner m

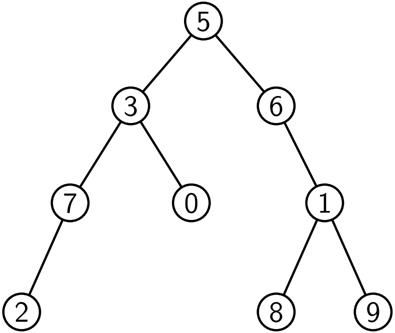
$\textsc{Min}(A_5) = \min(5, \textsc{Min}(A_3), \textsc{Min}(A_6))$
$\textsc{Min}(A_3) = \min(3, \textsc{Min}(A_7), \textsc{Min}(A_0))$



Exemples d'utilisation

Algorithme : $\textsc{Minimum}(x)$

$m \leftarrow +\infty$
si $x \neq \emptyset$ alors
    $m_G \leftarrow \textsc{Minimum}(\text{filsG}(x))$
    $m_D \leftarrow \textsc{Minimum}(\text{filsD}(x))$
    $m \leftarrow \min(m_G, m_D, \text{val}(x))$
retourner m



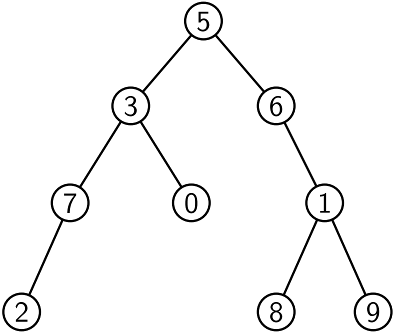
$\textsc{Min}(A_5) = \min(5, \textsc{Min}(A_3), \textsc{Min}(A_6))$
$\textsc{Min}(A_3) = \min(3, \textsc{Min}(A_7), \textsc{Min}(A_0))$
$\textsc{Min}(A_7) = \min(7, \textsc{Min}(A_2), \textsc{Min}(\emptyset))$

Exemples d'utilisation

Algorithme : $\textsc{Minimum}(x)$

$m \leftarrow +\infty$
si $x \neq \emptyset$ alors
  $m_G \leftarrow \textsc{Minimum}(\text{filsG}(x))$
  $m_D \leftarrow \textsc{Minimum}(\text{filsD}(x))$
  $m \leftarrow \min(m_G, m_D, \text{val}(x))$
retourner m

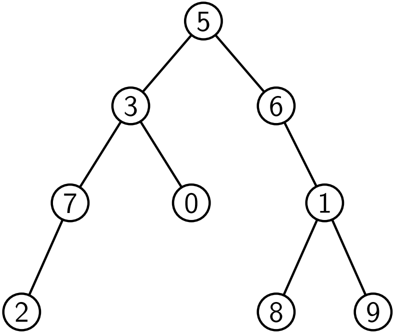
$$\textsc{Min}(A_5) = \min(5, \textsc{Min}(A_3), \textsc{Min}(A_6))$$
$$\textsc{Min}(A_3) = \min(3, \textsc{Min}(A_7), \textsc{Min}(A_0))$$
$$\textsc{Min}(A_7) = \min(7, \textsc{Min}(A_2), \textsc{Min}(\emptyset))$$
$$\textsc{Min}(A_2) = \min(2, \textsc{Min}(\emptyset), \textsc{Min}(\emptyset))$$



Exemples d'utilisation

Algorithme : $\textsc{Minimum}(x)$

$m \leftarrow +\infty$
si $x \neq \emptyset$ alors
  $m_G \leftarrow \textsc{Minimum}(\text{filsG}(x))$
  $m_D \leftarrow \textsc{Minimum}(\text{filsD}(x))$
  $m \leftarrow \min(m_G, m_D, \text{val}(x))$
retourner m

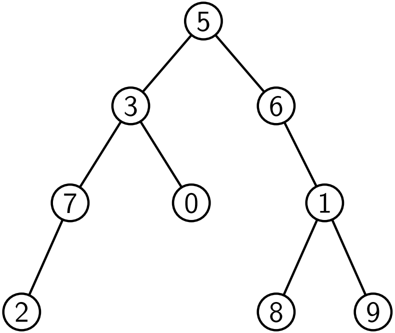


$$\textsc{Min}(A_5) = \min(5, \textsc{Min}(A_3), \textsc{Min}(A_6))$$
$$\textsc{Min}(A_3) = \min(3, \textsc{Min}(A_7), \textsc{Min}(A_0))$$
$$\textsc{Min}(A_7) = \min(7, \textsc{Min}(A_2), \textsc{Min}(\emptyset))$$
$$\textsc{Min}(A_2) = \min(2, \textsc{Min}(\emptyset), \textsc{Min}(\emptyset)) = 2$$

Exemples d'utilisation

Algorithme : $\textsc{Minimum}(x)$

$m \leftarrow +\infty$
si $x \neq \emptyset$ alors
  $m_G \leftarrow \textsc{Minimum}(\text{filsG}(x))$
  $m_D \leftarrow \textsc{Minimum}(\text{filsD}(x))$
  $m \leftarrow \min(m_G, m_D, \text{val}(x))$
retourner m

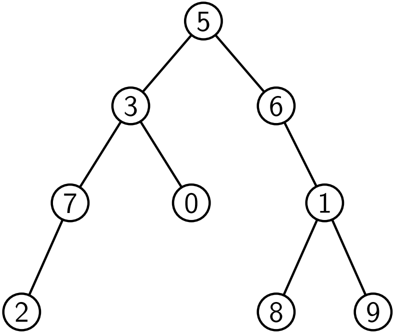


$$\textsc{Min}(A_5) = \min(5, \textsc{Min}(A_3), \textsc{Min}(A_6))$$
$$\textsc{Min}(A_3) = \min(3, \textsc{Min}(A_7), \textsc{Min}(A_0))$$
$$\textsc{Min}(A_7) = \min(7, \textsc{Min}(A_2), \textsc{Min}(\emptyset)) = 2$$
$$\textsc{Min}(A_2) = \min(2, \textsc{Min}(\emptyset), \textsc{Min}(\emptyset)) = 2$$

Exemples d'utilisation

Algorithme : Minimum(x)

$m \leftarrow +\infty$
si $x \neq \emptyset$ alors
  $m_G \leftarrow$ Minimum(filsG(x))
  $m_D \leftarrow$ Minimum(filsD(x))
  $m \leftarrow \min(m_G, m_D, \mathrm{val}(x))$
retourner m

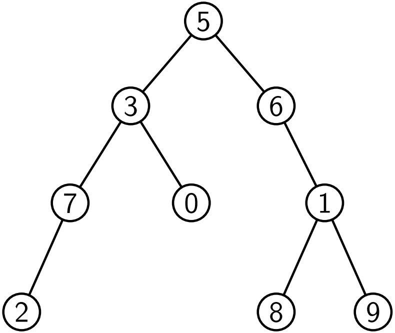


$$\textsc{Min}(A_5) = \min(5, \textsc{Min}(A_3), \textsc{Min}(A_6))$$
$$\textsc{Min}(A_3) = \min(3, \textsc{Min}(A_7), \textsc{Min}(A_0))$$
$$\textsc{Min}(A_7) = \min(7, \textsc{Min}(A_2), \textsc{Min}(\emptyset)) = 2$$
$$\textsc{Min}(A_2) = \min(2, \textsc{Min}(\emptyset), \textsc{Min}(\emptyset)) = 2$$
$$\textsc{Min}(A_0) = \min(0, \textsc{Min}(\emptyset), \textsc{Min}(\emptyset)) = 0$$

Exemples d'utilisation

Algorithme : $\textsc{Minimum}(x)$

$m \leftarrow +\infty$
si $x \neq \emptyset$ alors
  $m_G \leftarrow \textsc{Minimum}(\text{filsG}(x))$
  $m_D \leftarrow \textsc{Minimum}(\text{filsD}(x))$
  $m \leftarrow \min(m_G, m_D, \text{val}(x))$
retourner m

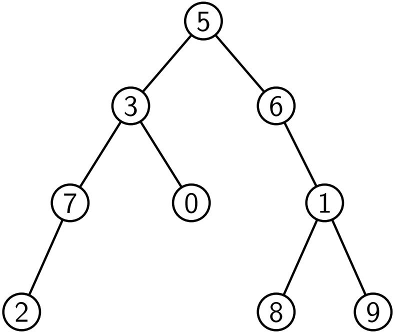


$$\textsc{Min}(A_5) = \min(5, \textsc{Min}(A_3), \textsc{Min}(A_6))$$
$$\textsc{Min}(A_3) = \min(3, \textsc{Min}(A_7), \textsc{Min}(A_0)) = 0$$
$$\textsc{Min}(A_7) = \min(7, \textsc{Min}(A_2), \textsc{Min}(\emptyset)) = 2$$
$$\textsc{Min}(A_2) = \min(2, \textsc{Min}(\emptyset), \textsc{Min}(\emptyset)) = 2$$
$$\textsc{Min}(A_0) = \min(0, \textsc{Min}(\emptyset), \textsc{Min}(\emptyset)) = 0$$

Exemples d'utilisation

Algorithme : $\textsc{Minimum}(x)$

$m \leftarrow +\infty$
si $x \neq \emptyset$ alors
  $m_G \leftarrow \textsc{Minimum}(\text{filsG}(x))$
  $m_D \leftarrow \textsc{Minimum}(\text{filsD}(x))$
  $m \leftarrow \min(m_G, m_D, \text{val}(x))$
retourner m

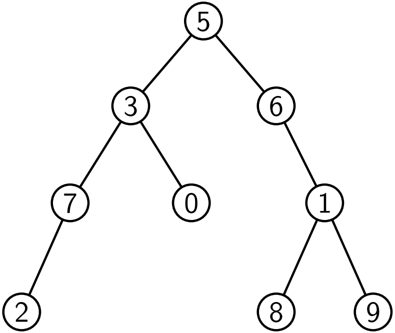


$$\textsc{Min}(A_5) = \min(5, \textsc{Min}(A_3), \textsc{Min}(A_6))$$
$$\textsc{Min}(A_3) = \min(3, \textsc{Min}(A_7), \textsc{Min}(A_0)) = 0$$
$$\textsc{Min}(A_7) = \min(7, \textsc{Min}(A_2), \textsc{Min}(\emptyset)) = 2$$
$$\textsc{Min}(A_2) = \min(2, \textsc{Min}(\emptyset), \textsc{Min}(\emptyset)) = 2$$
$$\textsc{Min}(A_0) = \min(0, \textsc{Min}(\emptyset), \textsc{Min}(\emptyset)) = 0$$
$$\textsc{Min}(A_6) = \cdots = 1$$

Exemples d'utilisation

Algorithme : $\textsc{Minimum}(x)$

$m \leftarrow +\infty$
si $x \neq \emptyset$ alors
  $m_G \leftarrow \textsc{Minimum}(\text{filsG}(x))$
  $m_D \leftarrow \textsc{Minimum}(\text{filsD}(x))$
  $m \leftarrow \min(m_G, m_D, \text{val}(x))$
retourner m



$\textsc{Min}(A_5) = \min(5, \textsc{Min}(A_3), \textsc{Min}(A_6)) = 0$
$\textsc{Min}(A_3) = \min(3, \textsc{Min}(A_7), \textsc{Min}(A_0)) = 0$
$\textsc{Min}(A_7) = \min(7, \textsc{Min}(A_2), \textsc{Min}(\emptyset)) = 2$
$\textsc{Min}(A_2) = \min(2, \textsc{Min}(\emptyset), \textsc{Min}(\emptyset)) = 2$
$\textsc{Min}(A_0) = \min(0, \textsc{Min}(\emptyset), \textsc{Min}(\emptyset)) = 0$
$\textsc{Min}(A_6) = \cdots = 1$

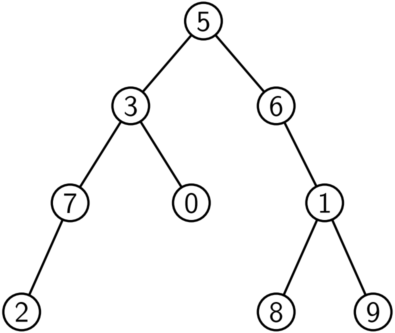
Exemples d'utilisation

Algorithme : $\textsc{Minimum}(x)$

$m \leftarrow +\infty$
si $x \neq \emptyset$ alors
  $m_G \leftarrow \textsc{Minimum}(\text{filsG}(x))$
  $m_D \leftarrow \textsc{Minimum}(\text{filsD}(x))$
  $m \leftarrow \min(m_G, m_D, \text{val}(x))$
retourner m

Algorithme : $\textsc{NbNœuds}(x)$

$n \leftarrow 0$
si $x \neq \emptyset$ alors
  $n_G \leftarrow \textsc{NbNœuds}(\text{filsG}(x))$
  $n_D \leftarrow \textsc{NbNœuds}(\text{filsD}(x))$
  $n \leftarrow n_G + n_D + 1$
retourner n



Objectifs

Stocker un ensemble ordonné de n valeurs avec les opérations :

- INSÉRER et SUPPRIMER
- MINIMUM et MAXIMUM
- RECHERCHER
- SUCCESSEUR et PRÉDÉCESSEUR

$\rightsquigarrow$ toutes ces opérations en « bonne » complexité

Objectifs

Stocker un ensemble ordonné de n valeurs avec les opérations :

- INSÉRER et SUPPRIMER
- MINIMUM et MAXIMUM
- RECHERCHER
- SUCCESSEUR et PRÉDÉCESSEUR

Liste chaînée triée : $O(1)$ pour max/min et succ/pred, $O(n)$ pour le reste

$\rightsquigarrow$ toutes ces opérations en « bonne » complexité

Objectifs

Stocker un ensemble ordonné de n valeurs avec les opérations :

- Insérer et Supprimer
- Minimum et Maximum
- Rechercher
- Successeur et Prédécesseur

Liste chaînée triée : $O(1)$ pour max/min et succ/pred, $O(n)$ pour le reste

$\rightsquigarrow$ toutes ces opérations en « bonne » complexité

Utilisation

- Stockage de données dynamiques
- Base de données (valeurs = identifiant)
- Linux : ordonnancement, mémoire virtuelle, ...

Objectifs

Stocker un ensemble ordonné de n valeurs avec les opérations :

- INSÉRER et SUPPRIMER
- MINIMUM et MAXIMUM
- RECHERCHER
- SUCCESSEUR et PRÉDÉCESSEUR

Liste chaînée triée : $O(1)$ pour max/min et succ/pred, $O(n)$ pour le reste

$\rightsquigarrow$ toutes ces opérations en « bonne » complexité

Utilisation

- Stockage de données dynamiques
- Base de données (valeurs = identifiant)
- Linux : ordonnancement, mémoire virtuelle, ...

Les arbres binaires de recherche sont une structure de donnée remplissant ces objectifs, mais pas la seule !

Définition

Si A est un arbre binaire et $x \in A$, on note

- saG(x) le sous-arbre gauche de x : le s-a de A enraciné en filsG(x)
- saD(x) le sous-arbre droit de x : le s-a de A enraciné en filsD(x)

Définition

Si A est un arbre binaire et $x \in A$, on note

- saG(x) le sous-arbre gauche de x : le s-a de A enraciné en filsG(x)
- saD(x) le sous-arbre droit de x : le s-a de A enraciné en filsD(x)

Un arbre binaire de recherche (ABR) est un arbre binaire tel que pour tout nœud x,

- $\forall y \in \text{saG}(x), \text{val}(y) \leq \text{val}(x)$
- $\forall z \in \text{saD}(x), \text{val}(x) \leq \text{val}(z)$

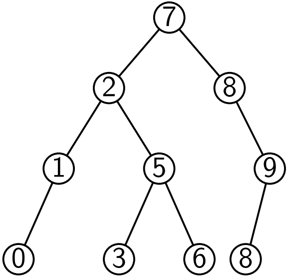
Définition

Si A est un arbre binaire et $x \in A$, on note

- saG(x) le sous-arbre gauche de x : le s-a de A enraciné en filsG(x)
- saD(x) le sous-arbre droit de x : le s-a de A enraciné en filsD(x)

Un arbre binaire de recherche (ABR) est un arbre binaire tel que pour tout nœud x,

- $\forall y \in$ saG(x), val$(y) \leq$ val(x)
- $\forall z \in$ saD(x), val$(x) \leq$ val(z)



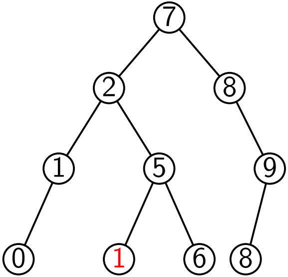
Définition

Si A est un arbre binaire et $x \in A$, on note

- saG(x) le sous-arbre gauche de x : le s-a de A enraciné en filsG(x)
- saD(x) le sous-arbre droit de x : le s-a de A enraciné en filsD(x)

Un arbre binaire de recherche (ABR) est un arbre binaire tel que pour tout nœud x,

- $\forall y \in \text{saG}(x)$, $\text{val}(y) \leq \text{val}(x)$
- $\forall z \in \text{saD}(x)$, $\text{val}(x) \leq \text{val}(z)$



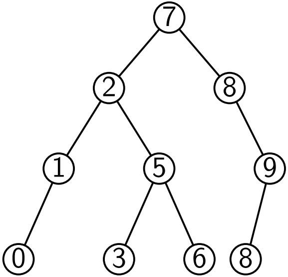
Définition

Si A est un arbre binaire et $x \in A$, on note

- saG(x) le sous-arbre gauche de x : le s-a de A enraciné en filsG(x)
- saD(x) le sous-arbre droit de x : le s-a de A enraciné en filsD(x)

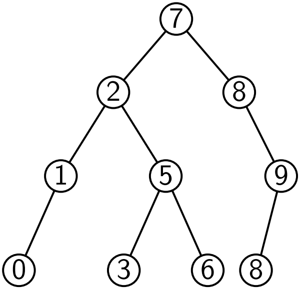
Un arbre binaire de recherche (ABR) est un arbre binaire tel que pour tout nœud x,

- $\forall y \in \text{saG}(x)$, $\text{val}(y) \leq \text{val}(x)$
- $\forall z \in \text{saD}(x)$, $\text{val}(x) \leq \text{val}(z)$



Parcours infixe d'un ABR

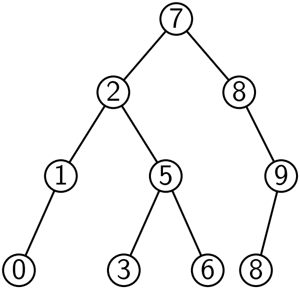
```
Algorithme :
 ParcoursInfixe(x)

si x ≠ ∅ alors
    ParcoursInfixe(filsG(x))
    Afficher val(x)
    ParcoursInfixe(filsD(x))
```



Parcours infixe d'un ABR

Algorithme :
 ParcoursInfixe(x)

si $x \neq \emptyset$ alors
    ParcoursInfixe(filsG(x))
    Afficher val(x)
    ParcoursInfixe(filsD(x))



Lemme

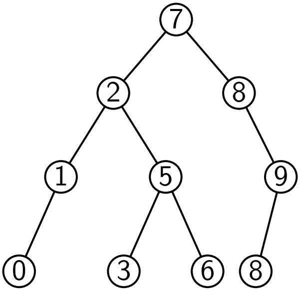
Le parcours infixe d'un arbre binaire A affiche les valeurs de A triées si et seulement si A est un ABR.

Preuve par induction : affichage en ordre $\nearrow$
ssi $\mathrm{val}(y) \leq \mathrm{val}(\mathrm{rac}(A)) \leq \mathrm{val}(z)$ pour $y \in \mathrm{saG}(A)$, $z \in \mathrm{saD}(A)$
ssi A est un ABR

Recherche dans un ABR

Algorithme : $\textsc{Rechercher}(x, k)$

si $x = \emptyset$ alors retourner $\emptyset$
si $\mathrm{val}(x) = k$ alors retourner x
si $\mathrm{val}(x) > k$ alors
    retourner
    $\textit{Rechercher}(\mathrm{filsG}(x), k)$
retourner $\textit{Rechercher}(\mathrm{filsD}(x), k)$

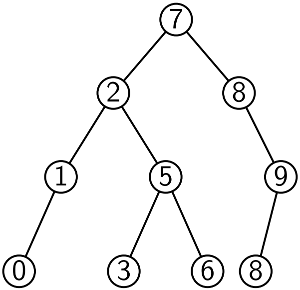


Recherche dans un ABR

Algorithme : RECHERCHER(x, k)

tant que $x \neq \emptyset$ *et* val(x) $\neq k$ faire
- si $k <$ val(x) alors
  - $x \leftarrow$ filsG(x)
- sinon
  - $x \leftarrow$ filsD(x)

retourner x

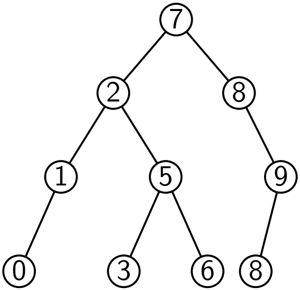


Recherche dans un ABR

Algorithme : RECHERCHER(x, k)

tant que $x \neq \emptyset$ *et* val$(x) \neq k$ faire
- si $k <$ val(x) alors
  - $x \leftarrow$ filsG(x)
- sinon
  - $x \leftarrow$ filsD(x)

retourner x



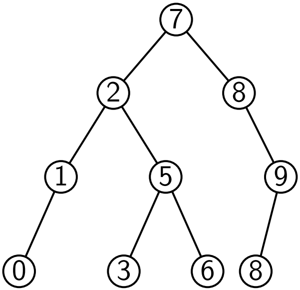
Validité admise

Recherche dans un ABR

Algorithme : $\textsc{Rechercher}(x, k)$

tant que $x \neq \emptyset$ *et* $\mathrm{val}(x) \neq k$ faire
- si $k < \mathrm{val}(x)$ alors
  - $x \leftarrow \mathrm{filsG}(x)$
- sinon
  - $x \leftarrow \mathrm{filsD}(x)$

retourner x



Validité admise

Lemme

$\textsc{Rechercher}(\mathrm{rac}(A))$ a une complexité $O(h(A))$.

Preuve

- À chaque passage dans la boucle, la hauteur de x augmente de 1 : au plus $\leq h(A)$ passages
- Chaque passage coûte $O(1)$

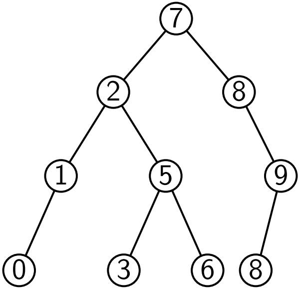
Minimum et successeur

Algorithme : MINIMUM(x)

tant que filsG(x) $\neq \emptyset$ faire

  $x \leftarrow$ filsG(x)

retourner x



Minimum et successeur

Algorithme : MINIMUM(x)

tant que filsG(x) $\neq \emptyset$ faire
  $x \leftarrow$ filsG(x)

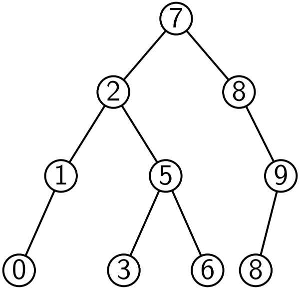
retourner x

Algorithme : SUCCESSEUR(x)

si filsD(x) $\neq \emptyset$ alors
  retourner *MINIMUM*(filsD(x))

$y \leftarrow$ père(x)
tant que $y \neq \emptyset$ *et* $x =$ filsD(y) faire
  $x \leftarrow y$
  $y \leftarrow$ père(x)

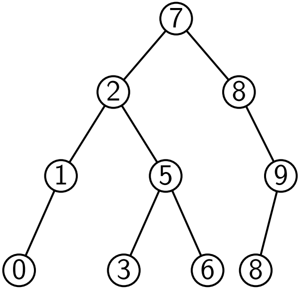
retourner y



Minimum et successeur

```
Algorithme : MINIMUM(x)

tant que filsG(x) ≠ ∅ faire
  └ x ← filsG(x)

retourner x
```

```
Algorithme : SUCCESSEUR(x)

si filsD(x) ≠ ∅ alors
  └ retourner MINIMUM(filsD(x))

y ← père(x)
tant que y ≠ ∅ et x = filsD(y) faire
  │ x ← y
  └ y ← père(x)

retourner y
```



$\textsc{Successeur}(2) = 3$

Minimum et successeur

Algorithme : $\textsc{Minimum}(x)$

tant que $\text{filsG}(x) \neq \emptyset$ faire
- $x \leftarrow \text{filsG}(x)$

retourner x

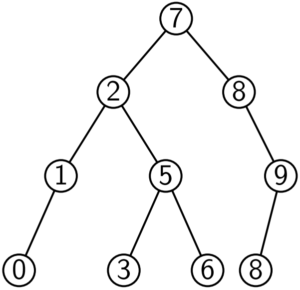
Algorithme : $\textsc{Successeur}(x)$

si $\text{filsD}(x) \neq \emptyset$ alors
- retourner $\textit{Minimum}(\text{filsD}(x))$

$y \leftarrow \text{père}(x)$

tant que $y \neq \emptyset$ *et* $x = \text{filsD}(y)$ faire
- $x \leftarrow y$
- $y \leftarrow \text{père}(x)$

retourner y



$\textsc{Successeur}(2) = 3$

$\textsc{Successeur}(6) = 7$

Minimum et successeur

Algorithme : $\textsc{Minimum}(x)$

tant que $\text{filsG}(x) \neq \emptyset$ faire
- $x \leftarrow \text{filsG}(x)$

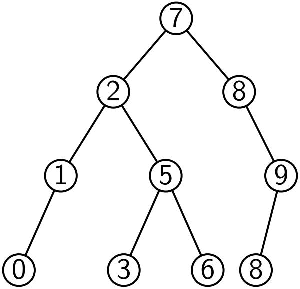
retourner x

Algorithme : $\textsc{Successeur}(x)$

si $\text{filsD}(x) \neq \emptyset$ alors
- retourner $\textit{Minimum}(\text{filsD}(x))$

$y \leftarrow \text{père}(x)$
tant que $y \neq \emptyset$ *et* $x = \text{filsD}(y)$ faire
- $x \leftarrow y$
- $y \leftarrow \text{père}(x)$

retourner y



$\textsc{Successeur}(2) = 3$
$\textsc{Successeur}(6) = 7$
$\textsc{Successeur}(9) = \emptyset$

Minimum et successeur

Algorithme : MINIMUM(x)

tant que filsG(x) $\neq \emptyset$ faire
  $x \leftarrow$ filsG(x)

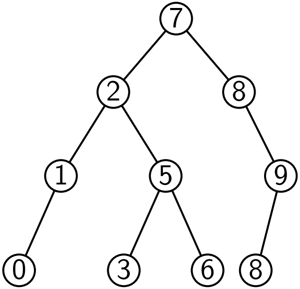
retourner x

Algorithme : SUCCESSEUR(x)

si filsD(x) $\neq \emptyset$ alors
  retourner *MINIMUM*(filsD(x))

$y \leftarrow$ père(x)
tant que $y \neq \emptyset$ *et* $x =$ filsD(y) faire
  $x \leftarrow y$
  $y \leftarrow$ père(x)

retourner y



SUCCESSEUR(2) $= 3$
SUCCESSEUR(6) $= 7$
SUCCESSEUR(9) $= \emptyset$

Lemme
MINIMUM et SUCCESSEUR ont une complexité $O(h(A))$

Validité de successeur

Algorithme : SUCCESSEUR(x)

si filsD(x) $\neq \emptyset$ alors
  retourner *MINIMUM*(filsD(x))

$p \leftarrow$ père(x)
tant que $p \neq \emptyset$ *et* $x =$ filsD(p) faire
  $(x, p) \leftarrow (p, \text{père}(x))$

retourner p

Lemme

SUCCESSEUR renvoie un sommet de plus petite valeur parmi ceux dont la valeur est $\geq \text{val}(x)$.

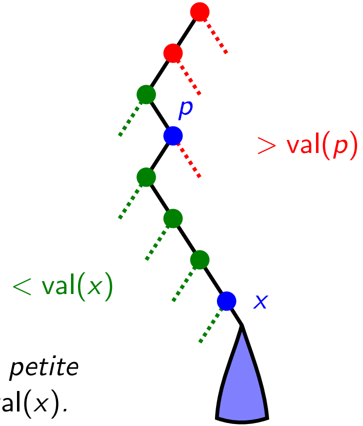
Validité de successeur

Algorithme : $\textsc{Successeur}(x)$

si $\text{filsD}(x) \neq \emptyset$ alors
  retourner $\textit{Minimum}(\text{filsD}(x))$

$p \leftarrow \text{père}(x)$
tant que $p \neq \emptyset$ *et* $x = \text{filsD}(p)$
  faire
  $(x, p) \leftarrow (p, \text{père}(x))$

retourner p



Lemme

Successeur renvoie un sommet de plus petite valeur parmi ceux dont la valeur est $\geq \text{val}(x)$.

Preuve Supp. les valeurs 2-à-2 distinctes. Soit p calculé par l'algo.

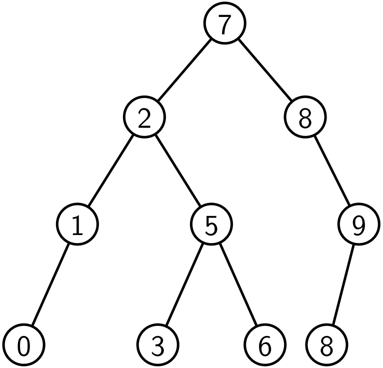
- pour tout $y \in \text{saD}(x)$, $\text{val}(x) < \text{val}(y) < \text{val}(p)$
- pour tout *ancêtre* $z \neq p$ de x, deux possibilités :
  - $x \in \text{saD}(z) \rightsquigarrow \text{val}(z) < \text{val}(x)$ et $\forall y \in \text{saG}(z), \text{val}(y) < \text{val}(x)$
  - $p \in \text{saG}(z) \rightsquigarrow \text{val}(z) > \text{val}(p)$ et $\forall y \in \text{saD}(z), \text{val}(y) > \text{val}(p)$

Insertion d'un élément

Algorithme : $\textsc{Insérer}(A, z)$

$x \leftarrow \mathrm{rac}(A)$
$p \leftarrow \emptyset$
tant que $x \neq \emptyset$ faire
    $p \leftarrow x$
    si $\mathrm{val}(z) < \mathrm{val}(x)$ alors
        $x \leftarrow \mathrm{filsG}(x)$
    sinon $x \leftarrow \mathrm{filsD}(x)$

$\mathrm{père}(z) \leftarrow p$
si $p = \emptyset$ alors $\mathrm{rac}(A) \leftarrow z$
sinon
    si $\mathrm{val}(z) < \mathrm{val}(p)$ alors
        $\mathrm{filsG}(p) \leftarrow z$
    sinon $\mathrm{filsD}(p) \leftarrow z$

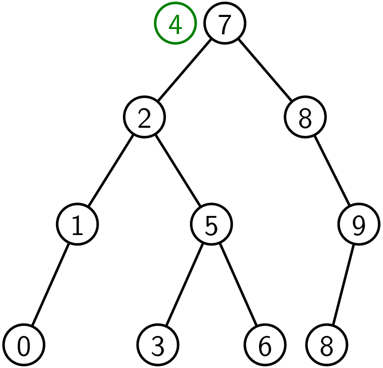


Insertion d'un élément

Algorithme : INSÉRER(A, z)

$x \leftarrow \text{rac}(A)$
$p \leftarrow \emptyset$
tant que $x \neq \emptyset$ faire
    $p \leftarrow x$
    si $\text{val}(z) < \text{val}(x)$ alors
        $x \leftarrow \text{filsG}(x)$
    sinon $x \leftarrow \text{filsD}(x)$

$\text{père}(z) \leftarrow p$
si $p = \emptyset$ alors $\text{rac}(A) \leftarrow z$
sinon
    si $\text{val}(z) < \text{val}(p)$ alors
        $\text{filsG}(p) \leftarrow z$
    sinon $\text{filsD}(p) \leftarrow z$

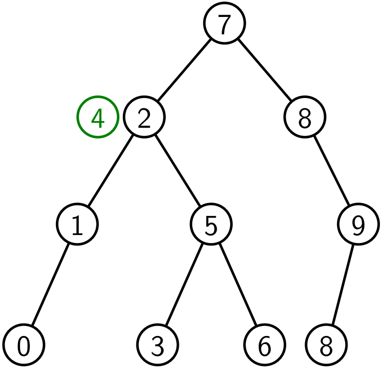


Insertion d'un élément

Algorithme : $\textsc{Insérer}(A, z)$

$x \leftarrow \mathrm{rac}(A)$
$p \leftarrow \emptyset$
tant que $x \neq \emptyset$ faire
    $p \leftarrow x$
    si $\mathrm{val}(z) < \mathrm{val}(x)$ alors
        $x \leftarrow \mathrm{filsG}(x)$
    sinon $x \leftarrow \mathrm{filsD}(x)$

$\mathrm{père}(z) \leftarrow p$
si $p = \emptyset$ alors $\mathrm{rac}(A) \leftarrow z$
sinon
    si $\mathrm{val}(z) < \mathrm{val}(p)$ alors
        $\mathrm{filsG}(p) \leftarrow z$
    sinon $\mathrm{filsD}(p) \leftarrow z$

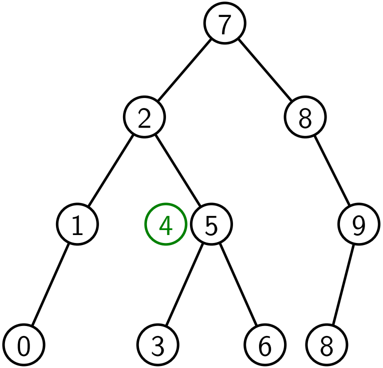


Insertion d'un élément

Algorithme : INSÉRER(A, z)

$x \leftarrow \text{rac}(A)$
$p \leftarrow \emptyset$
tant que $x \neq \emptyset$ faire
    $p \leftarrow x$
    si $\text{val}(z) < \text{val}(x)$ alors
        $x \leftarrow \text{filsG}(x)$
    sinon $x \leftarrow \text{filsD}(x)$

$\text{père}(z) \leftarrow p$
si $p = \emptyset$ alors $\text{rac}(A) \leftarrow z$
sinon
    si $\text{val}(z) < \text{val}(p)$ alors
        $\text{filsG}(p) \leftarrow z$
    sinon $\text{filsD}(p) \leftarrow z$

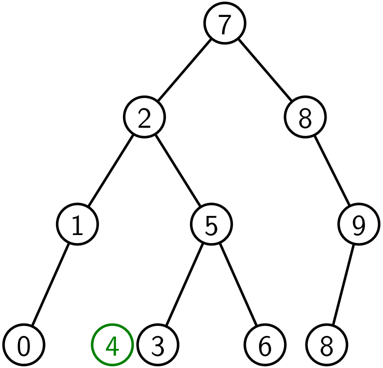


Insertion d'un élément

Algorithme : $\textsc{Insérer}(A, z)$

$x \leftarrow \mathrm{rac}(A)$
$p \leftarrow \emptyset$
tant que $x \neq \emptyset$ faire
- $p \leftarrow x$
- si $\mathrm{val}(z) < \mathrm{val}(x)$ alors
  - $x \leftarrow \mathrm{filsG}(x)$
- sinon $x \leftarrow \mathrm{filsD}(x)$

$\mathrm{père}(z) \leftarrow p$
si $p = \emptyset$ alors $\mathrm{rac}(A) \leftarrow z$
sinon
- si $\mathrm{val}(z) < \mathrm{val}(p)$ alors
  - $\mathrm{filsG}(p) \leftarrow z$
- sinon $\mathrm{filsD}(p) \leftarrow z$

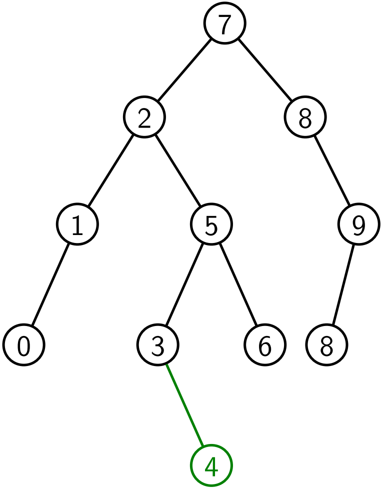


Insertion d'un élément

Algorithme : $\textsc{Insérer}(A, z)$

$x \leftarrow \text{rac}(A)$
$p \leftarrow \emptyset$
tant que $x \neq \emptyset$ faire
  $p \leftarrow x$
  si $\text{val}(z) < \text{val}(x)$ alors
    $x \leftarrow \text{filsG}(x)$
  sinon $x \leftarrow \text{filsD}(x)$

$\text{père}(z) \leftarrow p$
si $p = \emptyset$ alors $\text{rac}(A) \leftarrow z$
sinon
  si $\text{val}(z) < \text{val}(p)$ alors
    $\text{filsG}(p) \leftarrow z$
  sinon $\text{filsD}(p) \leftarrow z$



Suppression d'un élément

- Un petit algorithme de remplacement :
  On remplace le sous-arbre enraciné en x par celui enraciné en z dans l'arborescence A :

> Algorithme : $\textsc{Remplace}(A, x, z)$
>
> $p \leftarrow$ père(x)
> père$(x) \leftarrow \emptyset$
> si $p = \emptyset$ alors
>   rac$(A) \leftarrow z$
> sinon
>   si $x =$ filsG(p) alors
>   filsG$(p) \leftarrow z$
>   sinon
>   filsD$(p) \leftarrow z$
> si $z \neq \emptyset$ alors père$(z) \leftarrow p$;

Suppression d'un élément

Algorithme : $\textsc{Supprimer}(A, z)$

si filsG$(z) = \emptyset$ alors

    $\textsc{Remplace}(A, z, \text{filsD}(z))$

sinon si filsD$(z) = \emptyset$ alors

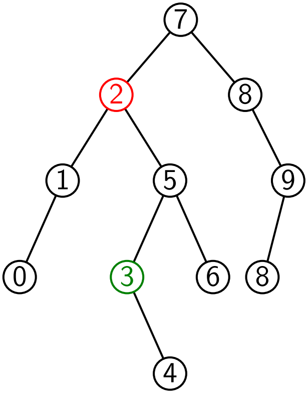
    $\textsc{Remplace}(A, z, \text{filsG}(z))$

sinon

    $y = \textsc{Successeur}(z)$

    $\textsc{Remplace}(A, y, \text{filsD}(y))$

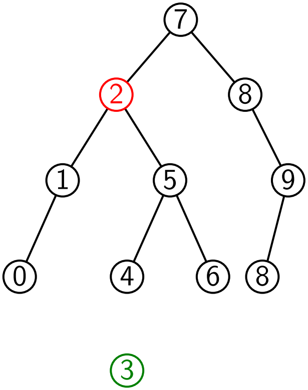
    Remplacer dans A, le nœud z par le nœud y



Suppression d'un élément

Algorithme : $\textsc{Supprimer}(A, z)$

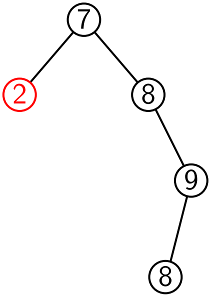
si filsG$(z) = \emptyset$ alors
  $\textsc{Remplace}(A, z, \text{filsD}(z))$
sinon si filsD$(z) = \emptyset$ alors
  $\textsc{Remplace}(A, z, \text{filsG}(z))$
sinon
  $y = \textsc{Successeur}(z)$
  $\textsc{Remplace}(A, y, \text{filsD}(y))$
  Remplacer dans A, le nœud z par le nœud y

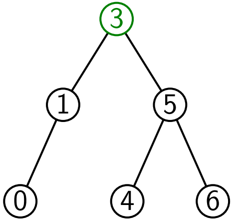


Suppression d'un élément

Algorithme : $\textsc{Supprimer}(A, z)$

si $\text{filsG}(z) = \emptyset$ alors
 $\textsc{Remplace}(A, z, \text{filsD}(z))$
sinon si $\text{filsD}(z) = \emptyset$ alors
 $\textsc{Remplace}(A, z, \text{filsG}(z))$
sinon
 $y = \textsc{Successeur}(z)$
 $\textsc{Remplace}(A, y, \text{filsD}(y))$
 Remplacer dans A, le nœud z par le nœud y





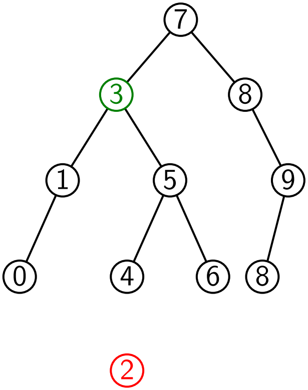
Suppression d'un élément

Algorithme : $\textsc{Supprimer}(A, z)$

si filsG$(z) = \emptyset$ alors
- $\textsc{Remplace}(A, z, \text{filsD}(z))$

sinon si filsD$(z) = \emptyset$ alors
- $\textsc{Remplace}(A, z, \text{filsG}(z))$

sinon
- $y = \textsc{Successeur}(z)$
- $\textsc{Remplace}(A, y, \text{filsD}(y))$
- Remplacer dans A, le nœud z par le nœud y



Validité et complexités

Lemme

Si A est un ABR, il reste un ABR après $\textsc{Supprimer}(A, z)$.

Preuve Le nœud z est remplacé par son successeur y :

- Pour tout $x \in \mathrm{saG}(y)$, $\mathrm{val}(x) \leq \mathrm{val}(z) \leq \mathrm{val}(y)$
- Pour tout $x \in \mathrm{saD}(y)$, $\mathrm{val}(x) \geq \mathrm{val}(y)$ car $y = \min(\mathrm{saD}(z))$

Le reste de l'arbre est inchangé. $\square$

Validité et complexités

Lemme

Si A est un ABR, il reste un ABR après $\textsc{Supprimer}(A, z)$.

Preuve Le nœud z est remplacé par son successeur y :

- Pour tout $x \in \text{saG}(y)$, $\text{val}(x) \leq \text{val}(z) \leq \text{val}(y)$
- Pour tout $x \in \text{saD}(y)$, $\text{val}(x) \geq \text{val}(y)$ car $y = \min(\text{saD}(z))$

Le reste de l'arbre est inchangé. $\square$

Lemme

Insérer et Supprimer ont une complexité $O(h(A))$.

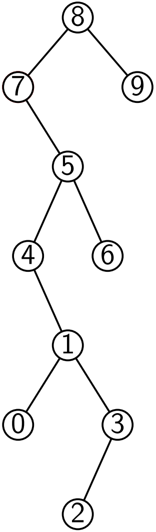
Preuve On parcourt *une branche de l'arbre* pour trouver soit l'endroit où insérer (Insérer) soit le successeur (Supprimer) : complexité $O(h(A))$. Le reste est un nombre constant de modifications de pointeurs. $\square$

Motivation

Rappel des complexités

- INSÉRER et SUPPRIMER : $O(h(A))$
- MINIMUM et MAXIMUM[1] : $O(h(A))$
- RECHERCHER : $O(h(A))$
- SUCCESSEUR et PRÉDECESSEUR[1] : $O(h(A))$

[1] Exercice !



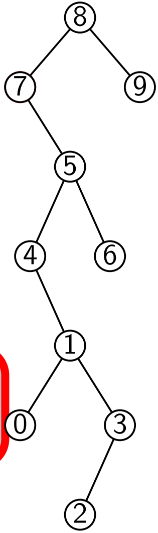
Motivation

Rappel des complexités

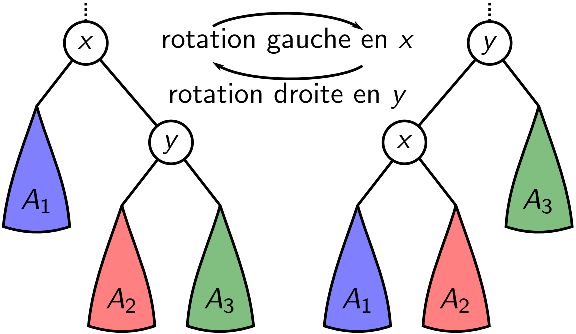
- Insérer et Supprimer : $O(h(A))$
- Minimum et Maximum[1] : $O(h(A))$
- Rechercher : $O(h(A))$
- Successeur et Prédecesseur[1] : $O(h(A))$

[1] Exercice !

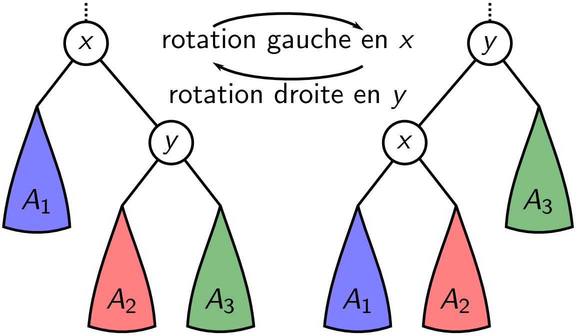
Un ABR est une structure de donnée efficace s'il est équilibré, c'est-à-dire si $h(A) = O(\log(n(A)))$.



Outil de base : les rotations



Outil de base : les rotations



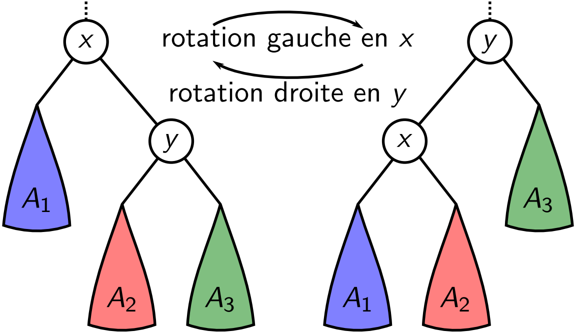
Lemme

Si A est un ABR, il reste un ABR après rotation.

Preuve Les rotations ne modifient que leur sous-arbre.

- Pour tout $z \in A_1$, $\mathrm{val}(z) \leq \mathrm{val}(x) \leq \mathrm{val}(y)$
- Pour tout $z \in A_2$, $\mathrm{val}(x) \leq \mathrm{val}(z) \leq \mathrm{val}(y)$
- Pour tout $z \in A_3$, $\mathrm{val}(x) \leq \mathrm{val}(y) \leq \mathrm{val}(z)$

Outil de base : les rotations



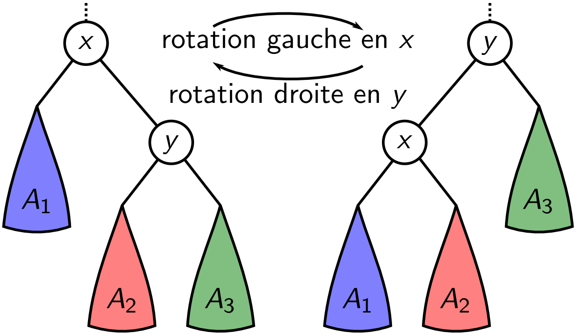
Lemme

Si A est un ABR, il reste un ABR après rotation.

Utilisation

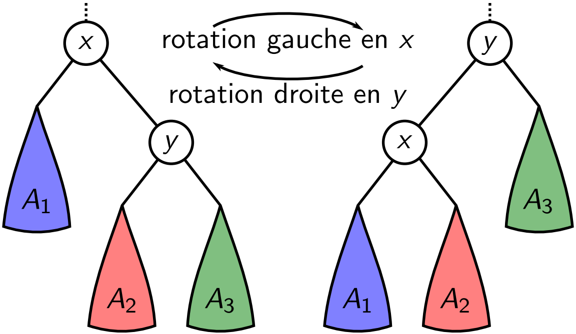
- Augmentation de la hauteur d'un côté, diminution de l'autre
- Opération en temps $O(1)$: quelques pointeurs à changer

Comment équilibrer ?



- Techniques d'équilibrage lors de INSÉRER/SUPPRIMER
  - arbres rouge-noir, AVL, B, déployés, ...
  - Tarbres (ou arbres-tas) : simulent l'insertion en ordre aléatoire

Comment équilibrer ?



- Techniques d'équilibrage lors de INSÉRER/SUPPRIMER
  - arbres rouge-noir, AVL, B, déployés, ...
  - Tarbres (ou arbres-tas) : simulent l'insertion en ordre aléatoire
- Au delà du contenu de ce cours...

Conclusion sur les ABR

- Structure de données pour ensembles ordonnés
- INSÉRER/SUPPRIMER, RECHERCHER, ... : $O(h(A))$
- $\lfloor \log(n(A)) \rfloor \leq h(A) < n(A)$
  - Efficace que si $h(A) = O(\log(n(A)))$
  - Vrai si insertion en ordre aléatoire
  - Techniques d'équilibrage basées sur les rotations

Utilisations principales des tas

- Algorithme du « tri par tas »
- Files de priorité : stockage d'un ensemble d'éléments ayant chacun une priorité, avec les opérations
  - AJOUTER : ajoute un nouvel élément (avec sa priorité)
  - RETIRERMAX : retire l'élément de priorité maximale
  - AUGMENTERPRIORITÉ : augmente la priorité d'un élément
  - DIMINUERPRIORITÉ : diminue la priorité d'un élément

Utilisations principales des tas

- Algorithme du « tri par tas »
- Files de priorité : stockage d'un ensemble d'éléments ayant chacun une priorité, avec les opérations
  - AJOUTER : ajoute un nouvel élément (avec sa priorité)
  - RETIRERMAX : retire l'élément de priorité maximale
  - AUGMENTERPRIORITÉ : augmente la priorité d'un élément
  - DIMINUERPRIORITÉ : diminue la priorité d'un élément
- Utilisation de files de priorité
  - Trouver le chemin le plus court entre deux points
    - dans un graphe (Dijkstra) $\rightsquigarrow$ Chap.6
    - sur une carte (A^*, ...)
  - Répartition de charge entre serveurs, ordonnancement de processus...
  - Priorités aléatoires pour équilibrer des ABR $\rightsquigarrow$ Algo de L3

Utilisations principales des tas

- Algorithme du « tri par tas »
- Files de priorité : stockage d'un ensemble d'éléments ayant chacun une priorité, avec les opérations
  - AJOUTER : ajoute un nouvel élément (avec sa priorité)
  - RETIRERMAX : retire l'élément de priorité maximale
  - AUGMENTERPRIORITÉ : augmente la priorité d'un élément
  - DIMINUERPRIORITÉ : diminue la priorité d'un élément
- Utilisation de files de priorité
  - Trouver le chemin le plus court entre deux points
    - dans un graphe (Dijkstra) $\rightsquigarrow$ Chap.6
    - sur une carte (A^*, ...)
  - Répartition de charge entre serveurs, ordonnancement de processus...
  - Priorités aléatoires pour équilibrer des ABR $\rightsquigarrow$ Algo de L3

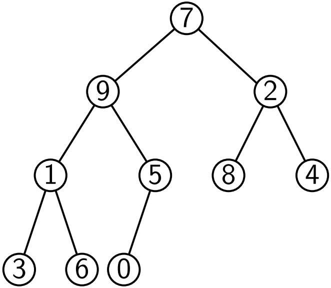
Le tas est une structure de donnée permettant d'implanter les files de priorités, mais les autres sont en général des extensions.

Arbres quasi-complets

Définition

Un arbre binaire est quasi-complet si

- pour tout $k < h(A)$, $|N_k| = 2^k$
- les nœuds de $N_{h(A)}$ sont « *le plus à gauche possible* »

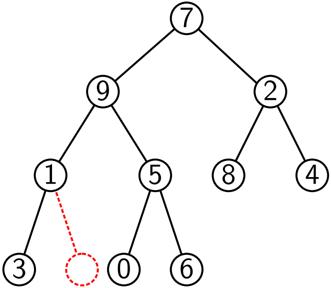


Arbres quasi-complets

Définition

Un arbre binaire est quasi-complet si

- pour tout $k < h(A)$, $|N_k| = 2^k$
- les nœuds de $N_{h(A)}$ sont « *le plus à gauche possible* »

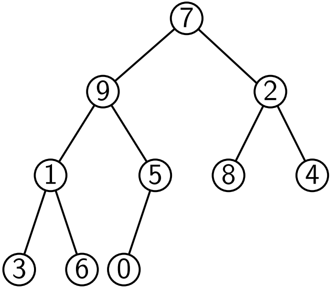


Arbres quasi-complets

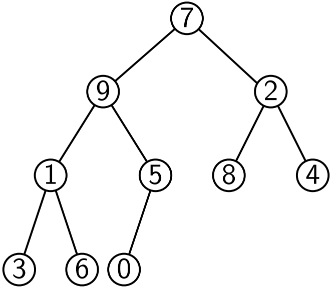
Définition

Un arbre binaire est quasi-complet si

- pour tout $k < h(A)$, $|N_k| = 2^k$
- les nœuds de $N_{h(A)}$ sont « *le plus à gauche possible* »



Arbres quasi-complets



Définition

Un arbre binaire est quasi-complet si

- pour tout $k < h(A)$, $|N_k| = 2^k$
- les nœuds de $N_{h(A)}$ sont « *le plus à gauche possible* »

Lemme

Si A est un arbre quasi-complet, $2^{h(A)} \leq n(A) \leq 2^{h(A)+1} - 1$.

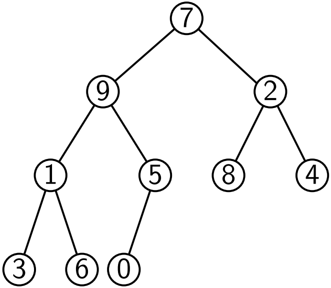
Preuve La borne supérieure est vraie pour tout A.

N_0, ..., $N_{h(A)-1}$ complets et $|N_{h(A)}| \geq 1$

$$\rightsquigarrow n(A) = \sum_{i=0}^{h(A)} |N_i| \geq 1 + \sum_{i=0}^{h(A)-1} 2^i = (2^{h(A)} - 1) + 1$$

(Le + petit arbre quasi-complet est un arbre complet de hauteur $h(A) - 1$, donc de taille $2^{h(A)} - 1$, avec 1 élément au niveau $h(A)$)

Arbres quasi-complets



Définition

Un arbre binaire est quasi-complet si

- pour tout $k < h(A)$, $|N_k| = 2^k$
- les nœuds de $N_{h(A)}$ sont « *le plus à gauche possible* »

Lemme

Si A est un arbre quasi-complet, $2^{h(A)} \leq n(A) \leq 2^{h(A)+1} - 1$.

Corollaire

Si A est un arbre quasi-complet, alors $h(A) = \lfloor \log n(A) \rfloor$.

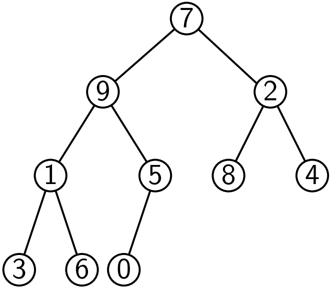
Preuve On a $2^{h(A)} \leq n(A) < 2^{h(A)+1}$ et donc $h(A) \leq \log n(A) < h(A) + 1$. □

Numérotation des arbres quasi-complets

Définition

Pour tout nœud x d'un arbre, soit num(x) son numéro, défini par

- $\mathrm{num}(\mathrm{rac}(A)) = 0$
- si $\mathrm{filsG}(x) \neq \emptyset$, $\mathrm{num}(\mathrm{filsG}(x)) = 2\,\mathrm{num}(x) + 1$
- si $\mathrm{filsD}(x) \neq \emptyset$, $\mathrm{num}(\mathrm{filsD}(x)) = 2\,\mathrm{num}(x) + 2$

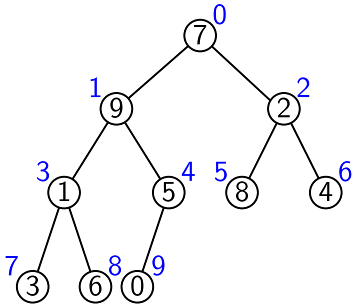


Numérotation des arbres quasi-complets

Définition

Pour tout nœud x d'un arbre, soit num(x) son numéro, défini par

- $\mathrm{num}(\mathrm{rac}(A)) = 0$
- si $\mathrm{filsG}(x) \neq \emptyset$,
  $\mathrm{num}(\mathrm{filsG}(x)) = 2\,\mathrm{num}(x) + 1$
- si $\mathrm{filsD}(x) \neq \emptyset$,
  $\mathrm{num}(\mathrm{filsD}(x)) = 2\,\mathrm{num}(x) + 2$

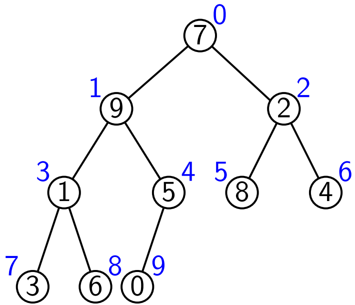


Numérotation des arbres quasi-complets

Définition

Pour tout nœud x d'un arbre, soit num(x) son numéro, défini par

- $\text{num}(\text{rac}(A)) = 0$
- si $\text{filsG}(x) \neq \emptyset$,
  $\text{num}(\text{filsG}(x)) = 2\,\text{num}(x) + 1$
- si $\text{filsD}(x) \neq \emptyset$,
  $\text{num}(\text{filsD}(x)) = 2\,\text{num}(x) + 2$



Numérotation de haut en bas et de gauche à droite
(parcours en largeur)

Propriétés de la numérotation

Lemme

Un arbre binaire est quasi-complet si et seulement si ses nœuds sont numérotés de 0 *à* $n(A) - 1$.

Propriétés de la numérotation

Lemme

Un arbre binaire est quasi-complet si et seulement si ses nœuds sont numérotés de 0 *à* $n(A) - 1$.

Preuve

1. Si $x \in N_k$, $2^k - 1 \leq \mathrm{num}(x) \leq 2^{k+1} - 2$
    - $k = 0$...
    - si $x \in N_k$, $\text{père}(x) \in N_{k-1}$
      $\rightsquigarrow 2^{k-1} - 1 \leq \mathrm{num}(\text{père}(x)) \leq 2^k - 2$
      $\rightsquigarrow 2 \cdot (2^{k-1} - 1) + 1 \leq \mathrm{num}(x) \leq 2 \cdot (2^k - 2) + 2$
      $\rightsquigarrow 2^k - 1 \leq \mathrm{num}(x) \leq 2^{k+1} - 2$

Propriétés de la numérotation

Lemme

Un arbre binaire est quasi-complet si et seulement si ses nœuds sont numérotés de 0 *à* $n(A) - 1$.

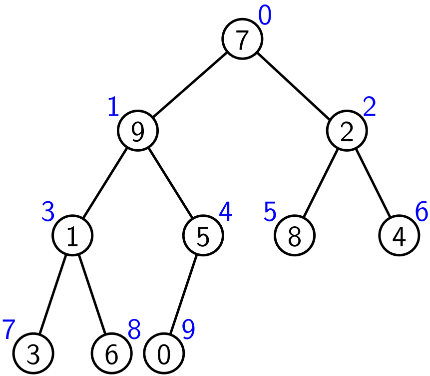
Preuve

1. Si $x \in N_k$, $2^k - 1 \leq \text{num}(x) \leq 2^{k+1} - 2$
   - $k = 0$...
   - si $x \in N_k$, $\text{père}(x) \in N_{k-1}$
     $\rightsquigarrow 2^{k-1} - 1 \leq \text{num}(\text{père}(x)) \leq 2^k - 2$
     $\rightsquigarrow 2 \cdot (2^{k-1} - 1) + 1 \leq \text{num}(x) \leq 2 \cdot (2^k - 2) + 2$
     $\rightsquigarrow 2^k - 1 \leq \text{num}(x) \leq 2^{k+1} - 2$
2. Si x est le *voisin de gauche* de y, $\text{num}(y) = \text{num}(x) + 1$
   - Si $\text{num}(x) = 2p + 1$, $y = \text{filsD}(\text{père}(x))$ donc $\text{num}(y) = 2p + 2$
   - Si $\text{num}(x) = 2p + 2$, $x = \text{filsD}(\text{filsG}(z))$ et $y = \text{filsG}(\text{filsD}(z))$
     $\rightsquigarrow \text{num}(x) = 2 \cdot (2q + 1) + 2 = 4q + 4$
     $\rightsquigarrow \text{num}(y) = 2 \cdot (2q + 2) + 1 = 4q + 5.$

Représentation informatique des arbres quasi-complets

Corollaire

On peut représenter un arbre quasi-complet par un tableau de taille $n(A)$ contenant $\mathrm{val}(x)$ en case $\mathrm{num}(x)$.



$$A = [7, 9, 2, 1, 5, 8, 4, 3, 6]$$

Représentation informatique des arbres quasi-complets

Corollaire

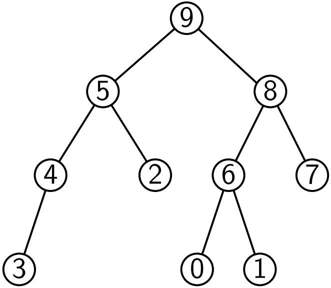
On peut représenter un arbre quasi-complet par un tableau de taille $n(A)$ contenant $\text{val}(x)$ en case $\text{num}(x)$.

On identifie un arbre quasi-complet et le tableau A qui le représente, et un nœud x et son numéro $\text{num}(x)$.

- $\text{rac}(A) = 0$
- $\text{filsG}(i) = 2i + 1$ et $\text{filsD}(i) = 2i + 2$
- $\text{père}(i) = \lfloor (i - 1)/2 \rfloor$
- $\text{val}(i) = A[i]$
- $h(i) = \lfloor \log(i + 1) \rfloor$

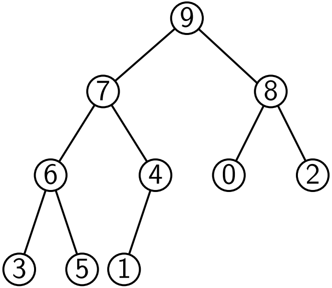
Définition des tas

- Un arbre binaire A a la propriété de tas max si pour tout $x \neq \text{rac}(A)$, $\text{val}(\text{père}(x)) \geq \text{val}(x)$
- Un arbre binaire A a la propriété de tas min si pour tout $x \neq \text{rac}(A)$, $\text{val}(\text{père}(x)) \leq \text{val}(x)$



Définition des tas

- Un arbre binaire A a la propriété de tas max si pour tout $x \neq \mathrm{rac}(A)$, $\mathrm{val}(\text{père}(x)) \geq \mathrm{val}(x)$
- Un arbre binaire A a la propriété de tas min si pour tout $x \neq \mathrm{rac}(A)$, $\mathrm{val}(\text{père}(x)) \leq \mathrm{val}(x)$



$[9, 7, 8, 6, 4, 0, 2, 3, 5, 1]$

Définition

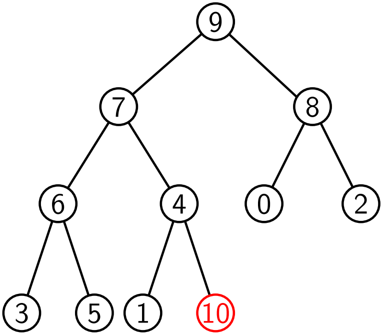
Un tas max (resp. min) est un arbre quasi-complet ayant la propriété de tas max (resp. min)

Un tableau T est un tas max si pour tout $i \geq 1$, $T[\lfloor \frac{i-1}{2} \rfloor] \geq T[i]$

Insertion dans un tas max

Algorithme : $\textsc{Insérer}(T, x)$

$i \leftarrow n(T)$
Agrandir T d'une case
$T[i] \leftarrow x$
$\textsc{Remonter}(T, i)$



$[9, 7, 8, 6, 4, 0, 2, 3, 5, 1, 10]$

Insertion dans un tas max

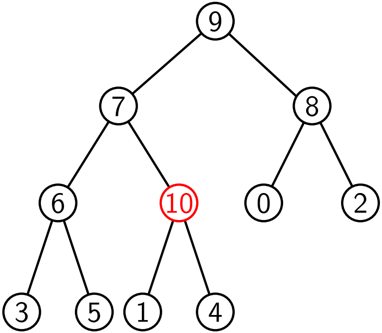
Algorithme : $\textsc{Insérer}(T, x)$

$i \leftarrow n(T)$

Agrandir T d'une case

$T[i] \leftarrow x$

$\textsc{Remonter}(T, i)$



$[9, 7, 8, 6, \textcolor{red}{10}, 0, 2, 3, 5, 1, 4]$

Insertion dans un tas max

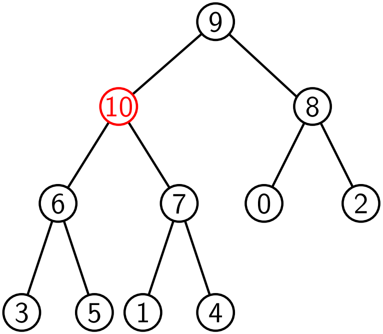
Algorithme : INSÉRER(T, x)

$i \leftarrow n(T)$

Agrandir T d'une case

$T[i] \leftarrow x$

REMONTER(T, i)



$[9, 10, 8, 6, 7, 0, 2, 3, 5, 1, 4]$

Insertion dans un tas max

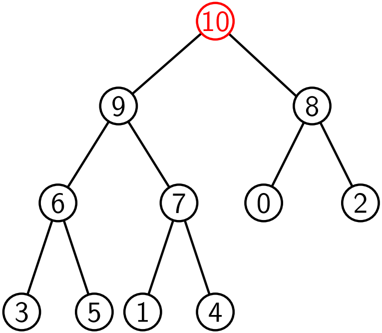
Algorithme : INSÉRER(T, x)

$i \leftarrow n(T)$

Agrandir T d'une case

$T[i] \leftarrow x$

REMONTER(T, i)



$[10, 9, 8, 6, 7, 0, 2, 3, 5, 1, 4]$

Insertion dans un tas max

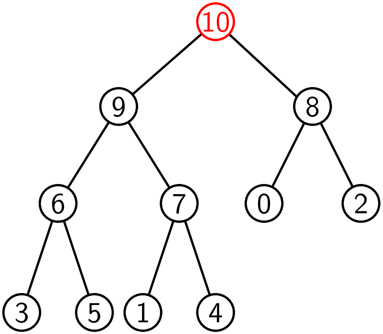
Algorithme : $\textsc{Insérer}(T, x)$

$i \leftarrow n(T)$

Agrandir T d'une case

$T[i] \leftarrow x$

$\textsc{Remonter}(T, i)$



$[10, 9, 8, 6, 7, 0, 2, 3, 5, 1, 4]$

Algorithme : $\textsc{Remonter}(T, i)$

tant que $i > 0$ *et* $T[\text{père}(i)] < T[i]$ faire

  Échanger $T[i]$ et $T[\text{père}(i)]$

  $i \leftarrow \text{père}(i)$

Complexité et validité de l'insertion

Algorithme : $\textsc{Remonter}(T, i)$

tant que $i > 0$ *et* $T[\text{père}(i)] < T[i]$
  faire
    Échanger $T[i]$ et $T[\text{père}(i)]$
    $i \leftarrow \text{père}(i)$

Lemme

$\textsc{Remonter}(T, i)$ a une complexité $O(\log(n(T)))$.

Preuve $\mathcal{P}_i$: le nombre de passage dans la boucle est $\leq h(i)$

- si $i = 0$, ok
- sinon : après le premier passage, i est remplacé par $\text{père}(i)$. Le nombre de passages suivants est $\leq h(\text{père}(i))$ par hypothèse de récurrence, donc le nombre total est $\leq h(\text{père}(i)) + 1 = h(i)$.

$\rightsquigarrow$ Complexité $O(h(T)) = O(\log(n(T)))$

Complexité et validité de l'insertion

Algorithme : $\textsc{Remonter}(T, i)$

tant que $i > 0$ *et* $T[\text{père}(i)] < T[i]$
faire
  Échanger $T[i]$ et $T[\text{père}(i)]$
  $i \leftarrow \text{père}(i)$

Lemme
Si $i = n(T) - 1$ et que T privé de i est un tas, alors T est un tas après $\textsc{Remonter}(T, i)$.

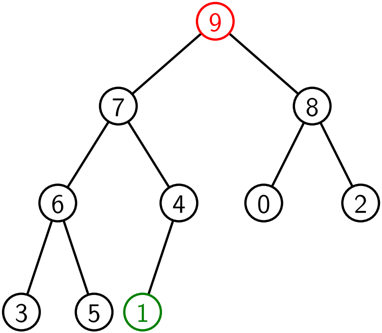
Preuve (idée) T_i : sous-arbre enraciné en i

- utilisation de l'invariant : T_i est un tas
- soit $p = \text{père}(i)$ et f l'autre fils de p s'il existe ; alors $T[i] > T[p] \geq T[f] \rightsquigarrow$ invariant conservé

Suppression dans un tas max

Algorithme : Supprimer(T, i)

$x \leftarrow T[i]$
$T[i] \leftarrow T[n(T) - 1]$
Réduire T d'une case
si (père$(i) \neq \emptyset$ *et* T[père$(i)] < T[i]$)
 alors Remonter(T, i)
sinon Entasser(T, i)
retourner x

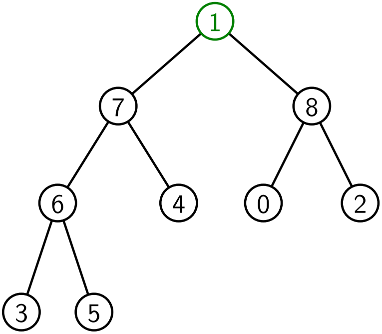


$[9, 7, 8, 6, 4, 0, 2, 3, 5, 1]$

Suppression dans un tas max

Algorithme : $\textsc{Supprimer}(T, i)$

$x \leftarrow T[i]$
$T[i] \leftarrow T[n(T) - 1]$
Réduire T d'une case
si $(\text{père}(i) \neq \emptyset$ *et* $T[\text{père}(i)] < T[i])$
  alors $\textsc{Remonter}(T, i)$
sinon $\textsc{Entasser}(T, i)$
retourner x



$[1, 7, 8, 6, 4, 0, 2, 3, 5]$

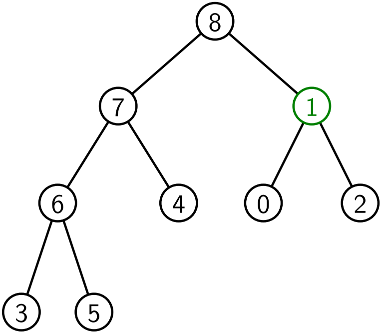
Algorithme : $\textsc{Entasser}(T, i)$

$(m, g, d) \leftarrow (i, \text{filsG}(i), \text{filsD}(i))$
si $g < n(T)$ *et* $T[g] > T[m]$ alors $m \leftarrow g$
si $d < n(T)$ *et* $T[d] > T[m]$ alors $m \leftarrow d$
si $m \neq i$ alors
  Échanger $T[i]$ et $T[m]$
  $\textsc{Entasser}(T, m)$

Suppression dans un tas max

Algorithme : Supprimer(T, i)

$x \leftarrow T[i]$
$T[i] \leftarrow T[n(T) - 1]$
Réduire T d'une case
si (père$(i) \neq \emptyset$ *et* $T[$père$(i)] < T[i]$)
  alors Remonter(T, i)
sinon Entasser(T, i)
retourner x



$[8, 7, 1, 6, 4, 0, 2, 3, 5]$

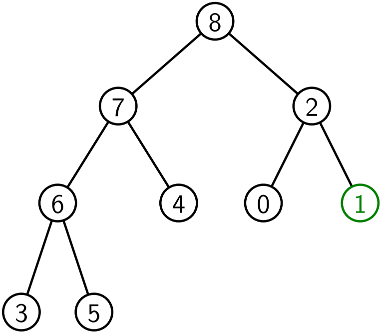
Algorithme : Entasser(T, i)

$(m, g, d) \leftarrow (i, \text{filsG}(i), \text{filsD}(i))$
si $g < n(T)$ *et* $T[g] > T[m]$ alors $m \leftarrow g$
si $d < n(T)$ *et* $T[d] > T[m]$ alors $m \leftarrow d$
si $m \neq i$ alors
  Échanger $T[i]$ et $T[m]$
  Entasser(T, m)

Suppression dans un tas max

Algorithme : $\textsc{Supprimer}(T, i)$

$x \leftarrow T[i]$
$T[i] \leftarrow T[n(T) - 1]$
Réduire T d'une case
si $(\text{père}(i) \neq \emptyset \ \textit{et} \ T[\text{père}(i)] < T[i])$
  alors $\textsc{Remonter}(T, i)$
sinon $\textsc{Entasser}(T, i)$
retourner x



$[8, 7, 2, 6, 4, 0, 1, 3, 5]$

Algorithme : $\textsc{Entasser}(T, i)$

$(m, g, d) \leftarrow (i, \text{filsG}(i), \text{filsD}(i))$
si $g < n(T)$ *et* $T[g] > T[m]$ alors $m \leftarrow g$
si $d < n(T)$ *et* $T[d] > T[m]$ alors $m \leftarrow d$
si $m \neq i$ alors
  Échanger $T[i]$ et $T[m]$
  $\textsc{Entasser}(T, m)$

Complexité et validité de la suppression

Algorithme : $\textsc{Entasser}(T, i)$

$(m, g, d) \leftarrow (i, \text{filsG}(i), \text{filsD}(i))$
si $g < n(T)$ *et* $T[g] > T[m]$ alors $m \leftarrow g$
si $d < n(T)$ *et* $T[d] > T[m]$ alors $m \leftarrow d$
si $m \neq i$ alors
  Échanger $T[i]$ et $T[m]$
  $\textsc{Entasser}(T, m)$

Lemme

$\textsc{Entasser}(T, i)$ a une complexité $O(\log(n(T)))$

Preuve $\mathcal{P}_i$: le nombre d'appels récursifs est $\leq h(T) - h(i)$

Récurrence *descendante* sur $h(i)$:

- Si $h(i) = h(T)$, aucun appel récursif donc ok
- Sinon ≤ 1 appel récursif sur un fils de hauteur $h(i) + 1 \rightsquigarrow$ nombre total d'appels récursif $\leq 1 + [h(T) - (h(i) + 1)]$ par hypothèse de récurrence

$\rightsquigarrow$ Complexité $O(h(T)) = O(\log(n(T)))$

Complexité et validité de la suppression

Algorithme : ENTASSER(T, i)

$(m, g, d) \leftarrow (i, \text{filsG}(i), \text{filsD}(i))$
si $g < n(T)$ *et* $T[g] > T[m]$ alors $m \leftarrow g$
si $d < n(T)$ *et* $T[d] > T[m]$ alors $m \leftarrow d$
si $m \neq i$ alors
    Échanger $T[i]$ et $T[m]$
    ENTASSER(T, m)

Lemme

Si les sous-arbres gauche et droit de i sont des tas, l'arbre enraciné en i est un tas après ENTASSER(T, i)

Preuve par récurrence sur $h(T) - h(i)$ (cas de base facile...)

- par hypothèse, T_m est un tas après l'appel récursif
- l'autre sous-arbre de i est un tas car non modifié
- $T[i] \geq T[g]$ et $T[i] \geq T[d]$ grâce à l'échange

Tri par tas

Algorithme : TriTas(T)

$S \leftarrow$ tableau vide de taille $n(T)$

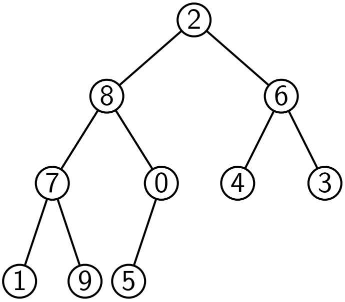
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 faire

  Entasser(T, i)

pour $i = n(T) - 1$ *à* 0 faire

  $S[i] \leftarrow$ Supprimer($T, 0$)

retourner S



Tri par tas

Algorithme : TriTas(T)

$S \leftarrow$ tableau vide de taille $n(T)$

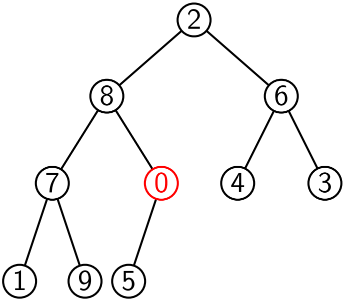
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 faire

  Entasser(T, i)

pour $i = n(T) - 1$ *à* 0 faire

  $S[i] \leftarrow$ Supprimer($T, 0$)

retourner S



Tri par tas

Algorithme : TRITAS(T)

$S \leftarrow$ tableau vide de taille $n(T)$

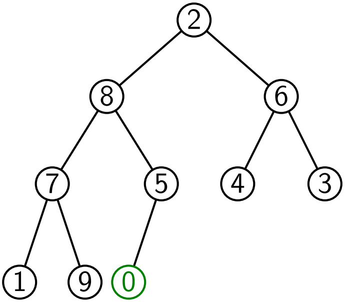
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 faire

  ENTASSER(T, i)

pour $i = n(T) - 1$ *à* 0 faire

  $S[i] \leftarrow$ SUPPRIMER($T, 0$)

retourner S



Tri par tas

Algorithme : TriTas(T)

$S \leftarrow$ tableau vide de taille $n(T)$

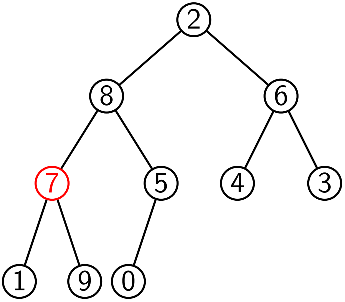
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 faire

  Entasser(T, i)

pour $i = n(T) - 1$ *à* 0 faire

  $S[i] \leftarrow$ Supprimer($T, 0$)

retourner S



Tri par tas

Algorithme : $\textsc{TriTas}(T)$

$S \leftarrow$ tableau vide de taille $n(T)$

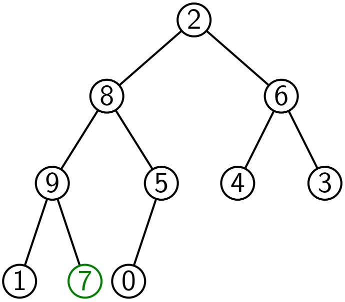
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 faire

  $\textsc{Entasser}(T, i)$

pour $i = n(T) - 1$ *à* 0 faire

  $S[i] \leftarrow \textsc{Supprimer}(T, 0)$

retourner S



Tri par tas

Algorithme : TRITAS(T)

$S \leftarrow$ tableau vide de taille $n(T)$

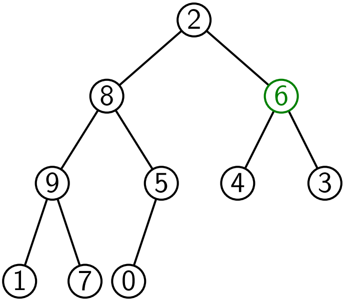
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 faire

  ENTASSER(T, i)

pour $i = n(T) - 1$ *à* 0 faire

  $S[i] \leftarrow$ SUPPRIMER($T, 0$)

retourner S



Tri par tas

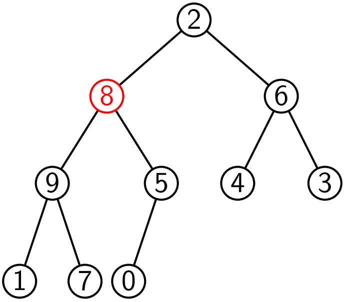
Algorithme : TRITAS(T)

$S \leftarrow$ tableau vide de taille $n(T)$

pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 faire
- ENTASSER(T, i)

pour $i = n(T) - 1$ *à* 0 faire
- $S[i] \leftarrow$ SUPPRIMER($T, 0$)

retourner S



Tri par tas

Algorithme : TriTas(T)

$S \leftarrow$ tableau vide de taille $n(T)$

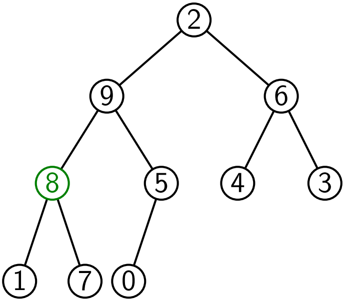
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 faire

  Entasser(T, i)

pour $i = n(T) - 1$ *à* 0 faire

  $S[i] \leftarrow$ Supprimer($T, 0$)

retourner S



Tri par tas

Algorithme : $\textsc{TriTas}(T)$

$S \leftarrow$ tableau vide de taille $n(T)$

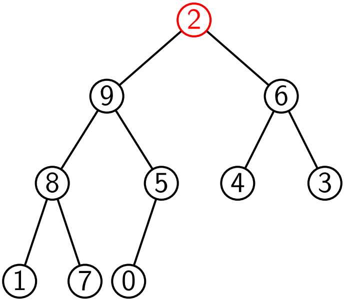
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 faire

  $\textsc{Entasser}(T, i)$

pour $i = n(T) - 1$ *à* 0 faire

  $S[i] \leftarrow \textsc{Supprimer}(T, 0)$

retourner S



Tri par tas

Algorithme : TriTas(T)

$S \leftarrow$ tableau vide de taille $n(T)$

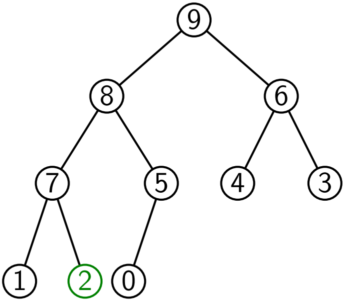
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 faire

  Entasser(T, i)

pour $i = n(T) - 1$ *à* 0 faire

  $S[i] \leftarrow$ Supprimer($T, 0$)

retourner S



Tri par tas

Algorithme : TRITAS(T)

$S \leftarrow$ tableau vide de taille $n(T)$

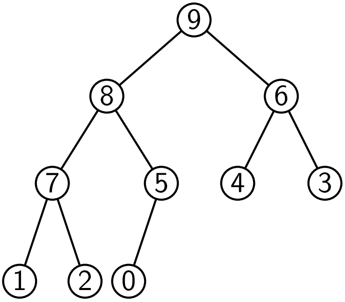
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 faire

  ENTASSER(T, i)

pour $i = n(T) - 1$ *à* 0 faire

  $S[i] \leftarrow$ SUPPRIMER($T, 0$)

retourner S



Tri par tas

Algorithme : TriTas(T)

$S \leftarrow$ tableau vide de taille $n(T)$

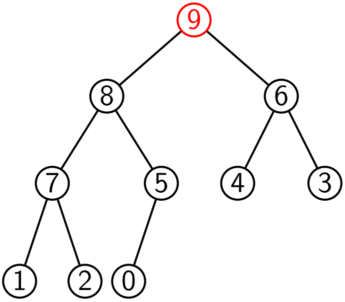
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 faire

  Entasser(T, i)

pour $i = n(T) - 1$ *à* 0 faire

  $S[i] \leftarrow$ Supprimer($T, 0$)

retourner S



Tri par tas

Algorithme : TRITAS(T)

$S \leftarrow$ tableau vide de taille $n(T)$

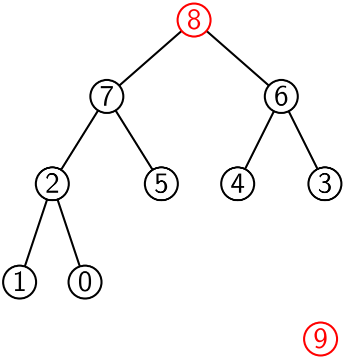
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 faire

  ENTASSER(T, i)

pour $i = n(T) - 1$ *à* 0 faire

  $S[i] \leftarrow$ SUPPRIMER($T, 0$)

retourner S



Tri par tas

Algorithme : TriTas(T)

$S \leftarrow$ tableau vide de taille $n(T)$

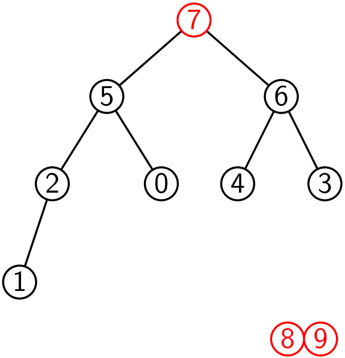
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 faire

  Entasser(T, i)

pour $i = n(T) - 1$ *à* 0 faire

  $S[i] \leftarrow$ Supprimer($T, 0$)

retourner S



Tri par tas

Algorithme : TriTas(T)

$S \leftarrow$ tableau vide de taille $n(T)$

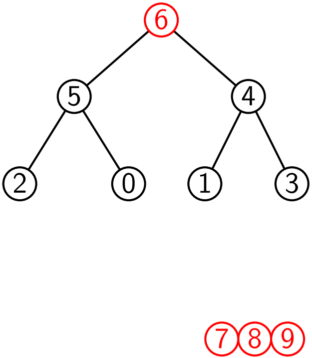
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 faire

  Entasser(T, i)

pour $i = n(T) - 1$ *à* 0 faire

  $S[i] \leftarrow$ Supprimer($T, 0$)

retourner S



Tri par tas

Algorithme : $\textsc{TriTas}(T)$

$S \leftarrow$ tableau vide de taille $n(T)$

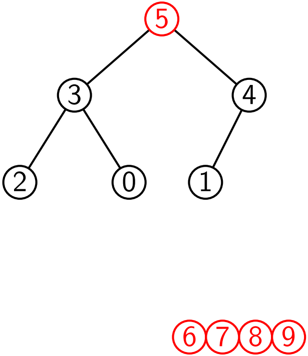
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 faire

  $\textsc{Entasser}(T, i)$

pour $i = n(T) - 1$ *à* 0 faire

  $S[i] \leftarrow \textsc{Supprimer}(T, 0)$

retourner S



Tri par tas

Algorithme : TRITAS(T)

$S \leftarrow$ tableau vide de taille $n(T)$

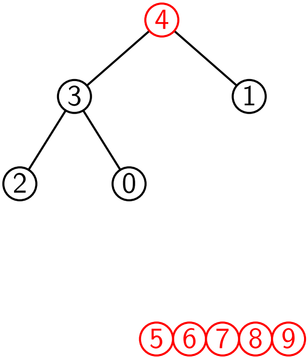
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 faire

  ENTASSER(T, i)

pour $i = n(T) - 1$ *à* 0 faire

  $S[i] \leftarrow$ SUPPRIMER($T, 0$)

retourner S



Tri par tas

Algorithme : TRITAS(T)

$S \leftarrow$ tableau vide de taille $n(T)$

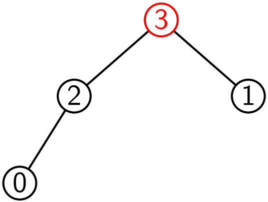
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 faire

  ENTASSER(T, i)

pour $i = n(T) - 1$ *à* 0 faire

  $S[i] \leftarrow$ SUPPRIMER($T, 0$)

retourner S



Tri par tas

Algorithme : TRITAS(T)

$S \leftarrow$ tableau vide de taille $n(T)$

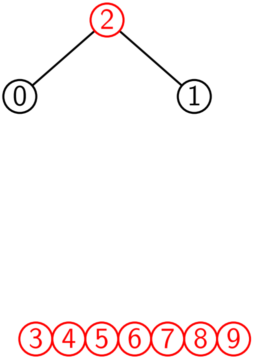
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 faire

  ENTASSER(T, i)

pour $i = n(T) - 1$ *à* 0 faire

  $S[i] \leftarrow$ SUPPRIMER($T, 0$)

retourner S



Tri par tas

Algorithme : TriTas(T)

$S \leftarrow$ tableau vide de taille $n(T)$

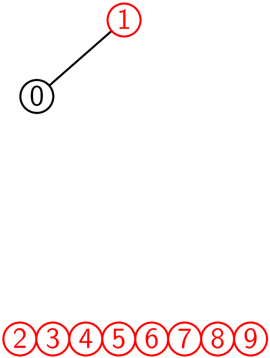
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 faire

  Entasser(T, i)

pour $i = n(T) - 1$ *à* 0 faire

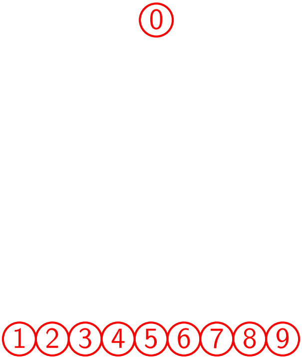
  $S[i] \leftarrow$ Supprimer($T, 0$)

retourner S



Tri par tas

```
Algorithme : TRITAS(T)
S ← tableau vide de taille n(T)
pour i = ⌊n(T)/2⌋ − 1 à 0 faire
    ENTASSER(T, i)
pour i = n(T) − 1 à 0 faire
    S[i] ← SUPPRIMER(T, 0)
retourner S
```



Tri par tas

Algorithme : TriTas(T)

$S \leftarrow$ tableau vide de taille $n(T)$

pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 **faire**

  Entasser(T, i)

pour $i = n(T) - 1$ *à* 0 **faire**

  $S[i] \leftarrow$ Supprimer($T, 0$)

retourner S

0 1 2 3 4 5 6 7 8 9

Tri par tas

Algorithme : TriTas(T)

$S \leftarrow$ tableau vide de taille $n(T)$

pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 faire

  Entasser(T, i)

pour $i = n(T) - 1$ *à* 0 faire

  $S[i] \leftarrow$ Supprimer($T, 0$)

retourner S

0 1 2 3 4 5 6 7 8 9

Lemme

Si T est un tableau quelconque, TriTas renvoie le tableau T trié. Sa complexité est $O(n \log n)$.

Tri par tas

Algorithme : TRITAS(T)

$S \leftarrow$ tableau vide de taille $n(T)$
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 faire
  ENTASSER(T, i)
pour $i = n(T) - 1$ *à* 0 faire
  $S[i] \leftarrow$ SUPPRIMER($T, 0$)
retourner S

0 1 2 3 4 5 6 7 8 9

Lemme

Si T est un tableau quelconque, TRITAS renvoie le tableau T trié. Sa complexité est $O(n \log n)$.

Preuve

- $O(n)$ appels à ENTASSER et SUPPRIMER $\rightsquigarrow O(n \log n)$
- Correction : si $i \geq \lfloor n(T)/2 \rfloor$, i est une feuille

Tri par tas

Algorithme : TRITAS(T)

$S \leftarrow$ tableau vide de taille $n(T)$

pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 faire

  ENTASSER(T, i)

pour $i = n(T) - 1$ *à* 0 faire

  $S[i] \leftarrow$ SUPPRIMER($T, 0$)

retourner S

0 1 2 3 4 5 6 7 8 9

Lemme

Si T est un tableau quelconque, TRITAS renvoie le tableau T trié. Sa complexité est $O(n \log n)$.

Remarque

Possibilité de tri *en place* car on remplit S par la fin $\rightsquigarrow$ TD

Borne inférieure pour le tri

Théorème

Un algorithme de tri ne faisant que des comparaisons a une complexité $\Omega(n \log n)$

Borne inférieure pour le tri

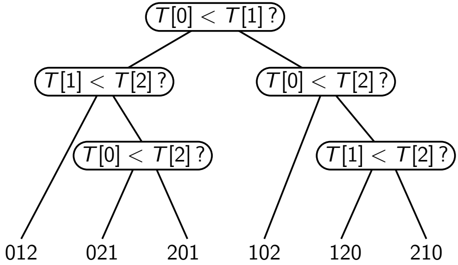
Théorème

Un algorithme de tri ne faisant que des comparaisons a une complexité $\Omega(n \log n)$

Preuve au tableau, basée sur l'arbre de décision :

- Nœuds : comparaisons entre deux entrées du tableau
- Feuilles : toutes les permutations de n éléments

$\rightsquigarrow$ arbre à $n!$ feuilles, donc de hauteur $\geq \lfloor \log(n!) \rfloor = \Omega(n \log n)$



Files de priorité

Stockage d'un ensemble d'éléments x ayant chacun une priorité p_x

Files de priorité

Stockage d'un ensemble d'éléments x ayant chacun une priorité p_x

Tas max de couples (x, p_x) qui vérifie la propriété de tas pour les priorités

Files de priorité

Stockage d'un ensemble d'éléments x ayant chacun une priorité p_x

> Tas max de couples (x, p_x) qui vérifie la propriété de tas pour les priorités

- AJOUTER : ajoute un nouvel élément (avec sa priorité)
  - Algorithme INSÉRER
- RETIRERMAX : retire l'élément de priorité maximale
  - Algorithme SUPPRIMER (en $i = 0$)
- AUGMENTERPRIORITÉ : augmente la priorité d'un élément
  - Algorithme : changer p_x en p'_x puis REMONTER
- DIMINUERPRIORITÉ : diminue la priorité d'un élément
  - Algorithme : changer p_x en p'_x puis ENTASSER

$\rightsquigarrow$ opérations en complexité $O(\log n)$

Conclusion sur les tas

- Structure de données pour conserver un ordre de priorité
- Arbre binaire quasi-complet :
  - représentation en tableau
  - arbre équilibré $\rightsquigarrow$ hauteur $O(\log n)$
- Insérer et Supprimer : $O(\log n)$
- Utilisations :
  - Tri par tas : $O(n \log n)$
  - Files de priorités

Conclusion

Les arbres binaires en informatique

- Représentation structurée de l'information
  - arbres binaires de recherche
  - tas
  - ...

Les arbres binaires en informatique

stackoverflow

Search...

Home

PUBLIC

Stack Overflow

Tags

Users

Jobs

Teams
Q&A for work

Learn More

What are the applications of binary trees?

380

Applications of binary trees

- Binary Search Tree - Used in *many* search applications where data is constantly entering/leaving, such as the `map` and `set` objects in many languages' libraries.
- Binary Space Partition - Used in almost every 3D video game to determine what objects need to be rendered.
- Binary Tries - Used in almost every high-bandwidth router for storing router-tables.
- Hash Trees - used in p2p programs and specialized image-signatures in which a hash needs to be verified, but the whole file is not available.
- Heaps - Used in implementing efficient priority-queues, which in turn are used for scheduling processes in many operating systems, Quality-of-Service in routers, and A* *(path-finding algorithm used in AI applications, including robotics and video games)*. Also used in heap-sort.
- Huffman Coding Tree (Chip Uni) - used in compression algorithms, such as those used by the .jpeg and .mp3 file-formats.
- GGM Trees - Used in cryptographic applications to generate a tree of pseudo-random numbers.
- Syntax Tree - Constructed by compilers and (implicitly) calculators to parse expressions.
- Treap - Randomized data structure used in wireless networking and memory allocation.
- T-tree - Though most databases use some form of B-tree to store data on the drive, databases which keep all (most) their data in memory often use T-trees to do so.

modifié depuis https://stackoverflow.com/a/2200588

Les arbres binaires en informatique

- Représentation structurée de l'information
  - arbres binaires de recherche
  - tas
  - ...
- Raisonnement informatique
  - Arbre de récursion (analyse des algorithmes récursifs)
  - Arbre de décision (borne inférieure sur le tri)
  - ...

Les arbres binaires en informatique

- Représentation structurée de l'information
  - arbres binaires de recherche
  - tas
  - ...
- Raisonnement informatique
  - Arbre de récursion (analyse des algorithmes récursifs)
  - Arbre de décision (borne inférieure sur le tri)
  - ...
- Pourquoi binaires ?
  - Arbres ternaires, ..., d-aires
  - Arbres avec nombre quelconque (non constant) de fils

Les arbres binaires en informatique

- Représentation structurée de l'information
  - arbres binaires de recherche
  - tas
  - ...
- Raisonnement informatique
  - Arbre de récursion (analyse des algorithmes récursifs)
  - Arbre de décision (borne inférieure sur le tri)
  - ...
- Pourquoi binaires ?
  - Arbres ternaires, ..., d-aires
  - Arbres avec nombre quelconque (non constant) de fils

Les arbres sont un des objets centraux de l'informatique !