Chapitre 2

Structures de données : arbres binaires et graphes

HLIN401 : Algorithmique et complexité

L2 Informatique
Université de Montpellier
2020 – 2021

Introduction

Étude de

- Deux objets informatiques avec algorithmes de base
  - Arbres (binaires)
  - Graphes (non orientés)

Introduction

Étude de

- Deux objets informatiques avec algorithmes de base
  - Arbres (binaires)
  - Graphes (non orientés)
- deux structures de données basées sur les arbres binaires
  - Arbres binaires de recherche (ABR)
  - Tas

Introduction

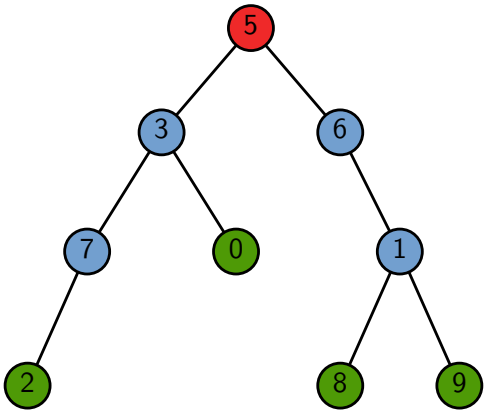
Étude de

- Deux objets informatiques avec algorithmes de base
  - Arbres (binaires)
  - Graphes (non orientés)
- deux structures de données basées sur les arbres binaires
  - Arbres binaires de recherche (ABR)
  - Tas

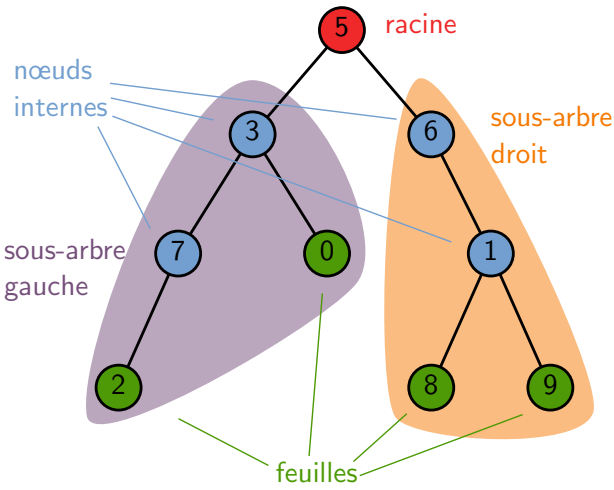
Rappel (HLIN301)

- Tableaux
- Listes (simplement) chaînées
- Piles et files
- Arbres binaires et tas

Définition



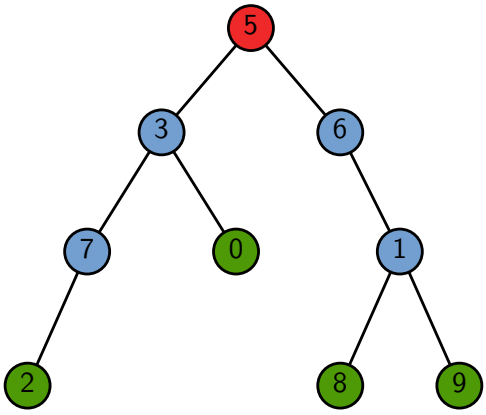
Définition



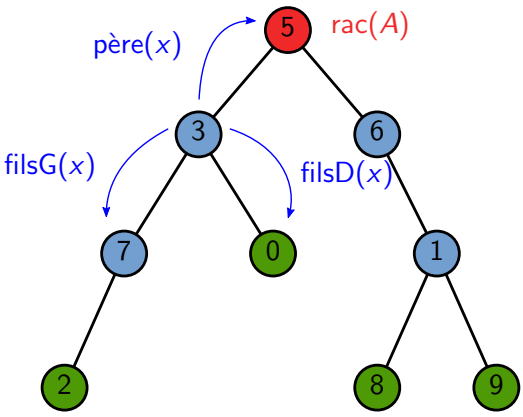
Un **arbre binaire** est défini récursivement :

- 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



Représentation informatique



Un **nœud** x est soit

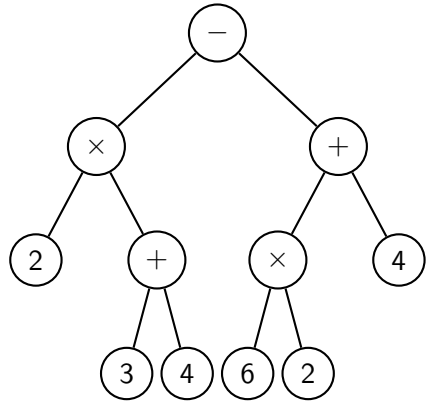
- le **nœud vide**, noté $\emptyset$
- défini par une **valeur** val(x) et **trois pointeurs** vers d'autres nœuds : **père(x), filsG(x), filsD(x)** tels que
  - Si filsG(x) $\neq \emptyset$, père(filsG(x)) $= x$
  - Si filsD(x) $\neq \emptyset$, père(filsD(x)) $= x$

Un **arbre binaire** A est donné par une **racine rac(A)** qui est un nœud tel que père(rac(A)) $= \emptyset$.

Utilité des arbres binaires

- Arbres binaires de recherche
- Tas

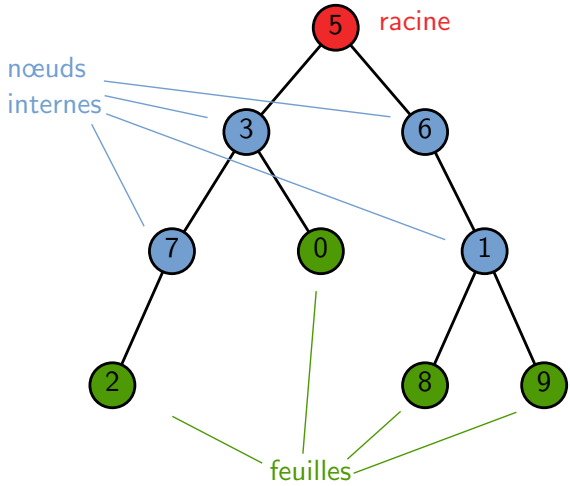
- Analyse syntaxique
- Bases de données
- Partition binaire de l'espace
- Tables de routage
- ...



$$2 \times (3 + 4) - (6 \times 2 + 4)$$

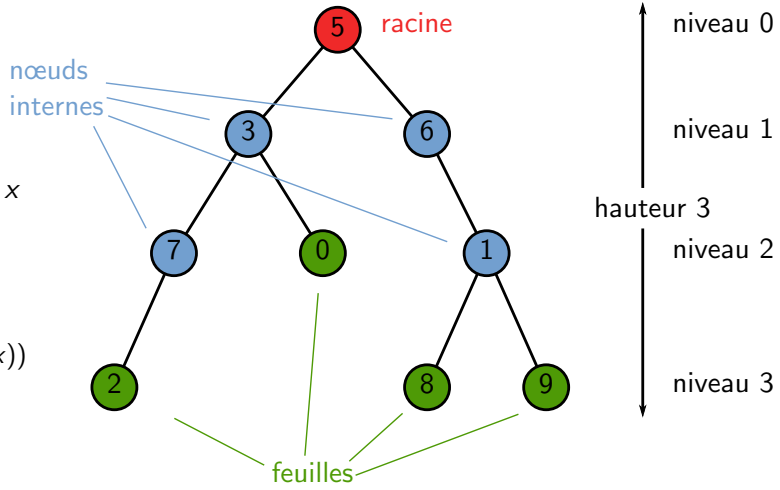
Hauteur et niveaux

- Un nœud x est une **feuille** si $\text{filsG}(x) = \emptyset$ et $\text{filsD}(x) = \emptyset$



Hauteur et niveaux

- 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 $x = \text{rac}(A)$, $h(x) = 0$ ($\Leftrightarrow \text{père}(x) = \emptyset$)
  - Sinon, $h(x) = 1 + h(\text{père}(x))$
- La **hauteur** d'un arbre A est $\boldsymbol{h(A)} = \max\{h(x) : x \in A\}$
- Le $\boldsymbol{k}^{\text{ème}}$ **niveau** de A est $\boldsymbol{N_k} = \{x : h(x) = k\}$



Résultats structurels

Lemme

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

Preuve par récurrence sur k

- $k = 0 : \{x : h(x) = 0\} = \{\text{rac}(A)\}$
- 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

- $k = 0 : \{x : h(x) = 0\} = \{\text{rac}(A)\}$
- 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

- $k = 0 : \{x : h(x) = 0\} = \{\text{rac}(A)\}$
- 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

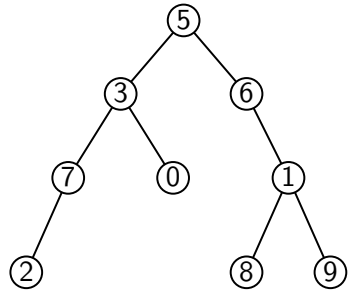
$$\lfloor \log(n(A)) \rfloor \leq h(A) < n(A)$$

Parcours en profondeur d'un arbre binaire

Algorithme : ParcoursInfixe(x)

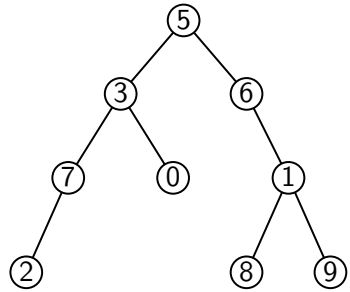
si $x \neq \emptyset$ **:**

- ParcoursInfixe(filsG(x))
- Afficher val(x)
- ParcoursInfixe(filsD(x))



Parcours en profondeur d'un arbre binaire

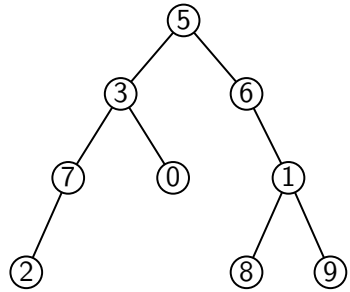
Algorithme : ParcoursInfixe(x)
si $x \neq \emptyset$ **:**
    ParcoursInfixe(filsG(x))
    Afficher val(x)
    ParcoursInfixe(filsD(x))



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

Parcours en profondeur d'un arbre binaire

Algorithme : ParcoursInfixe(x)
si $x \neq \emptyset$ **:**
  ParcoursInfixe(filsG(x))
  Afficher val(x)
  ParcoursInfixe(filsD(x))

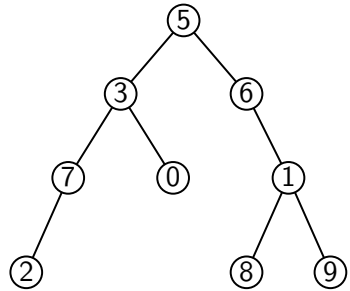


- Affichage : 2 7 3 0 5 6 8 1 9
- Complexité en $O(n(A))$

  Preuve $\mathcal{P}_n$: l'algo. effectue $2n$ appels à lui-même au total
  - $n = 0$: 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, $2n_G$ et $2n_D$ appels à ParcoursInfixe, donc au total $2n_G + 2n_D + 2$ appels. Or $n(A) = n_G + n_D + 1$, d'où le résultat.

Parcours en profondeur d'un arbre binaire

Algorithme : ParcoursInfixe(x)
si $x \neq \emptyset$ **:**
    ParcoursInfixe(filsG(x))
    Afficher val(x)
    ParcoursInfixe(filsD(x))

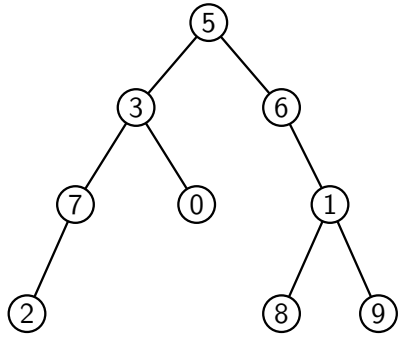


- Affichage : 2 7 3 0 5 6 8 1 9
- Complexité en $O(n(A))$
- Appel de la fonction : ParcoursInfixe(rac(A))
- Variantes : ParcoursPrefixe et ParcoursSuffixe $\rightsquigarrow$ TD

Exemples d'algorithmes

Algorithme : MINIMUM(x)

$m \leftarrow +\infty$
si $x \neq \emptyset$ **:**
    $m_G \leftarrow$ MINIMUM(filsG(x))
    $m_D \leftarrow$ MINIMUM(filsD(x))
    $m \leftarrow \min(m_G, m_D, \text{val}(x))$
renvoyer m



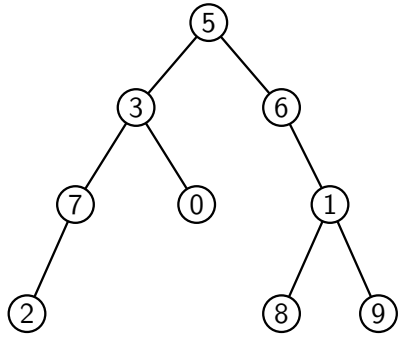
Exemples d'algorithmes

Algorithme : MINIMUM(x)

$m \leftarrow +\infty$
si $x \neq \emptyset$ **:**
    $m_G \leftarrow$ MINIMUM(filsG(x))
    $m_D \leftarrow$ MINIMUM(filsD(x))
    $m \leftarrow \min(m_G, m_D, \text{val}(x))$
renvoyer m

Algorithme : NBNŒUDS(x)

$n \leftarrow 0$
si $x \neq \emptyset$ **:**
    $n_G \leftarrow$ NBNŒUDS(filsG(x))
    $n_D \leftarrow$ NBNŒUDS(filsD(x))
    $n \leftarrow n_G + n_D + 1$
renvoyer n



Algorithme générique sur les arbres binaires

Appel de ALGO(rac(A)) avec

Algorithme : ALGO(x)

res $\leftarrow$ valeur pour l'arbre vide
si $x \neq \emptyset$ **:**
    $\text{res}_G \leftarrow$ ALGO(filsG(x))
    $\text{res}_D \leftarrow$ ALGO(filsD(x))
    res $\leftarrow f(\text{res}, \text{res}_G, \text{res}_D, x)$
renvoyer *res*

Algorithme générique sur les arbres binaires

Appel de ALGO(rac(A)) avec

Algorithme : ALGO(x)

res $\leftarrow$ valeur pour l'arbre vide
si $x \neq \emptyset$ **:**
    $\text{res}_G \leftarrow$ ALGO(filsG(x))
    $\text{res}_D \leftarrow$ ALGO(filsD(x))
    res $\leftarrow f(\text{res}, \text{res}_G, \text{res}_D, x)$
renvoyer *res*

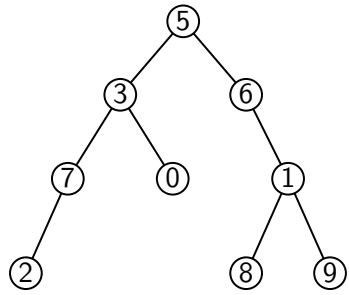
Lemme

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

Parcours en largeur d'un arbre binaire

Algorithme : PARCOURSLARGEUR(x)

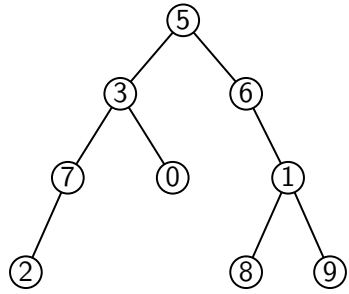
$F \leftarrow$ file vide
si $x \neq \emptyset$ **:** l'ajouter à F
tant que F *est non vide* **:**
    $y \leftarrow$ défiler un élément de F
    Afficher val(y)
    si filsG(y) $\neq \emptyset$ **:** l'ajouter à F
    si filsD(y) $\neq \emptyset$ **:** l'ajouter à F



Parcours en largeur d'un arbre binaire

Algorithme : PARCOURSLARGEUR(x)

$F \leftarrow$ file vide
si $x \neq \emptyset$ **:** l'ajouter à F
tant que F *est non vide* **:**
  $y \leftarrow$ défiler un élément de F
  Afficher val(y)
  si filsG(y) $\neq \emptyset$ **:** l'ajouter à F
  si filsD(y) $\neq \emptyset$ **:** l'ajouter à F

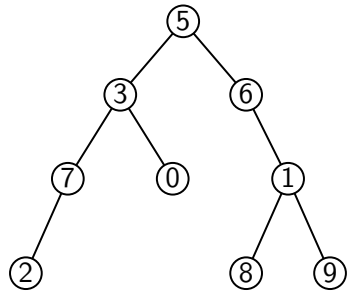


- Affichage : 5 3 6 7 0 1 2 8 9
  - niveau par niveau
  - de gauche à droite

Parcours en largeur d'un arbre binaire

Algorithme : PARCOURSLARGEUR(x)

$F \leftarrow$ file vide
si $x \neq \emptyset$ **:** l'ajouter à F
tant que F *est non vide* **:**
  $y \leftarrow$ défiler un élément de F
  Afficher val(y)
  si filsG(y) $\neq \emptyset$ **:** l'ajouter à F
  si filsD(y) $\neq \emptyset$ **:** l'ajouter à F

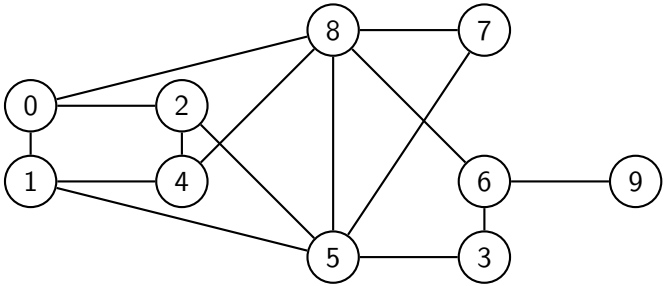


- Affichage : 5 3 6 7 0 1 2 8 9
  - niveau par niveau
  - de gauche à droite
- Complexité en $O(n(A))$

Vocabulaire

Graphe **connexe** constitué de 10 **sommets** et 15 **arêtes** : $G = (\mathbf{S}, \mathbf{A})$

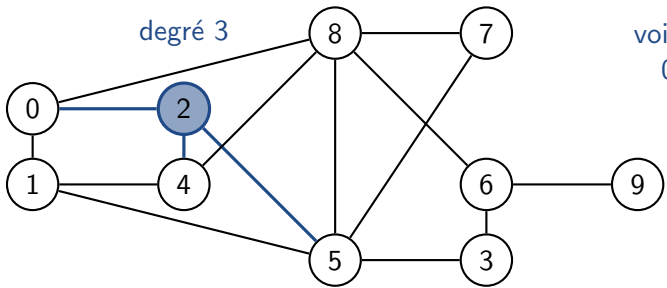
arête : ensemble de deux sommets $\{u, v\} \subset S$, notée souvent uv ou vu



Vocabulaire

Graphe **connexe** constitué de 10 **sommets** et 15 **arêtes** : $G = (\mathbf{S}, \mathbf{A})$

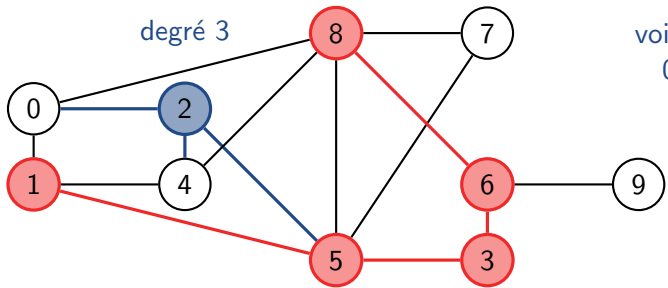
arête : ensemble de deux sommets $\{u, v\} \subset S$, notée souvent uv ou vu



Vocabulaire

Graphe **connexe** constitué de 10 **sommets** et 15 **arêtes** : $G = (\mathbf{S}, \mathbf{A})$

arête : ensemble de deux sommets $\{u, v\} \subset S$, notée souvent uv ou vu

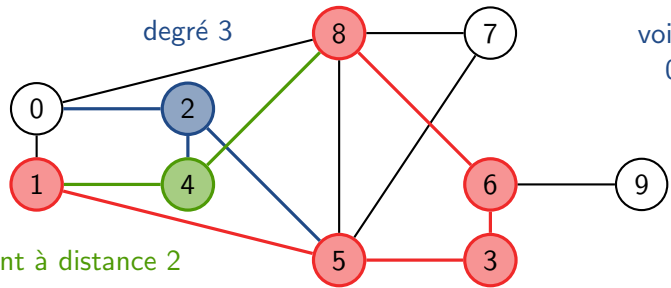


chemin de longueur 4

Vocabulaire

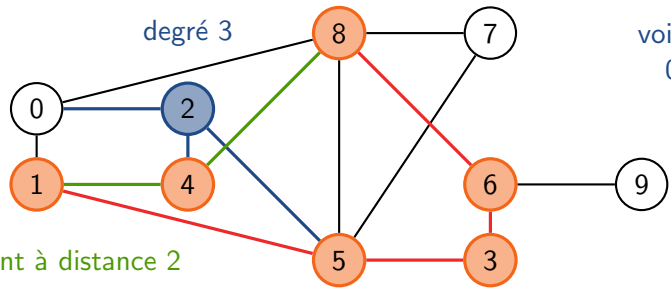
Graphe **connexe** constitué de 10 **sommets** et 15 **arêtes** : $G = (\mathbf{S}, \mathbf{A})$

arête : ensemble de deux sommets $\{u, v\} \subset S$, notée souvent uv ou vu



Vocabulaire

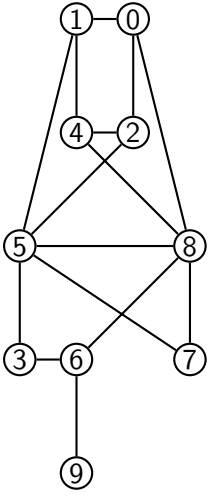
Graphe **connexe** constitué de 10 **sommets** et 15 **arêtes** : $G = (\mathbf{S}, \mathbf{A})$
arête : ensemble de deux sommets $\{u, v\} \subset S$, notée souvent uv ou vu



\+ chemin de longueur 4

= cycle de longueur 6

Représentations informatiques



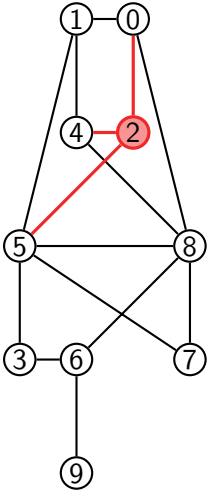
Matrice d'adjacence

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Listes d'adjacence

$$\begin{aligned} 0 &: 1 \to 2 \to 8 \\ 1 &: 0 \to 4 \to 5 \\ 2 &: 0 \to 4 \to 5 \\ 3 &: 5 \to 6 \\ 4 &: 1 \to 2 \to 8 \\ 5 &: 1 \to 2 \to 7 \to 8 \\ 6 &: 3 \to 8 \to 9 \\ 7 &: 5 \to 8 \\ 8 &: 0 \to 4 \to 5 \to 6 \\ 9 &: 6 \end{aligned}$$

Représentations informatiques



Matrice d'adjacence

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

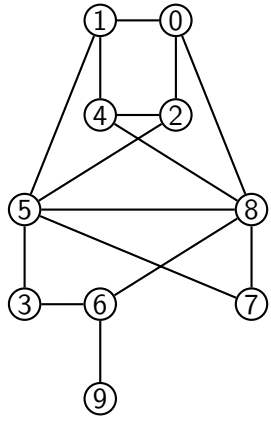
Listes d'adjacence

$$\begin{aligned} 0 &: 1 \to 2 \to 8 \\ 1 &: 0 \to 4 \to 5 \\ 2 &: 0 \to 4 \to 5 \\ 3 &: 5 \to 6 \\ 4 &: 1 \to 2 \to 8 \\ 5 &: 1 \to 2 \to 7 \to 8 \\ 6 &: 3 \to 8 \to 9 \\ 7 &: 5 \to 8 \\ 8 &: 0 \to 4 \to 5 \to 6 \\ 9 &: 6 \end{aligned}$$

Parcours en largeur

Algorithme : PARCOURSLARGEUR(x)

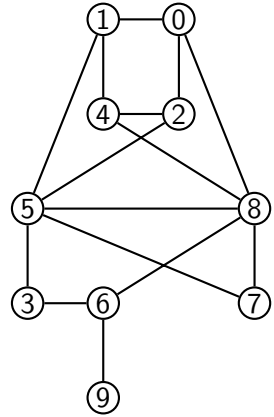
$F \leftarrow$ file vide
si $x \neq \emptyset$ **:** l'ajouter à F
tant que F *est non vide* **:**
    $y \leftarrow$ défiler un élément de F
    Afficher val(y)
    si filsG(y) $\neq \emptyset$ **:** l'ajouter à F
    si filsD(y) $\neq \emptyset$ **:** l'ajouter à F



Parcours en largeur

Algorithme : PARCOURSLARGEUR(G, s)

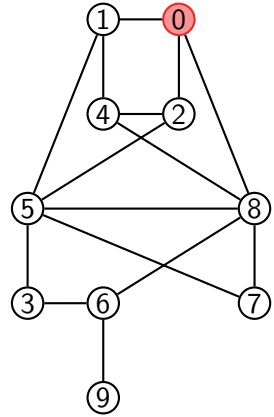
$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v



Parcours en largeur

Algorithme : PARCOURSLARGEUR(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v

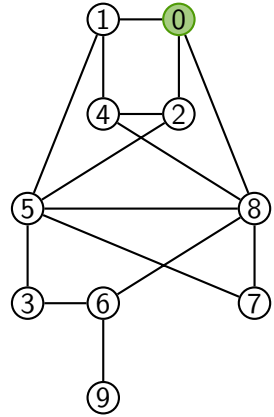


- File : 0
- Affichage :

Parcours en largeur

Algorithme : PARCOURSLARGEUR(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v

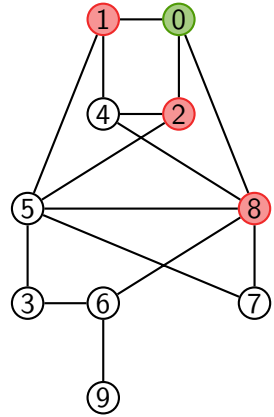


- File :
- Affichage : 0

Parcours en largeur

Algorithme : ParcoursLargeur(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v

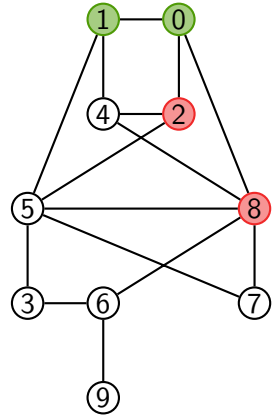


- File : 1 2 8
- Affichage : 0

Parcours en largeur

Algorithme : PARCOURSLARGEUR(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v

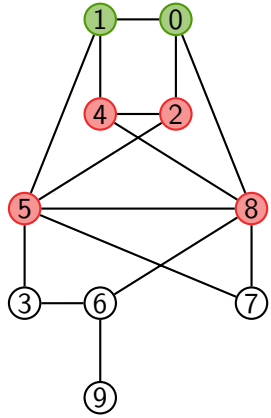


- File : 2 8
- Affichage : 0 1

Parcours en largeur

Algorithme : ParcoursLargeur(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v

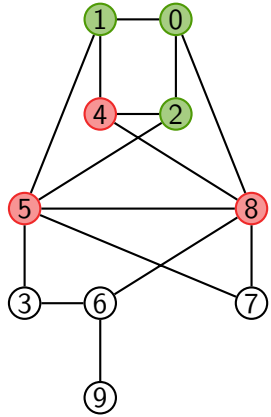


- File : 2 8 4 5
- Affichage : 0 1

Parcours en largeur

Algorithme : ParcoursLargeur(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v

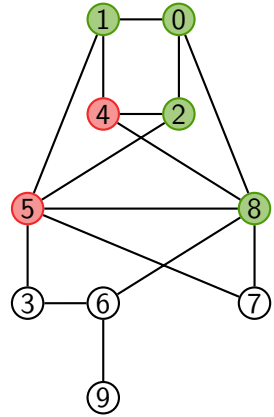


- File : 8 4 5
- Affichage : 0 1 2

Parcours en largeur

Algorithme : ParcoursLargeur(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v

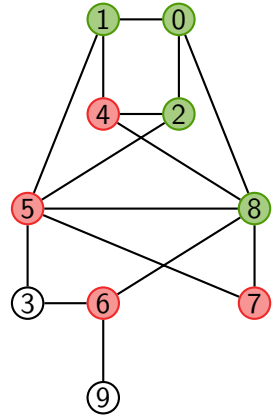


- File : 4 5
- Affichage : 0 1 2 8

Parcours en largeur

Algorithme : PARCOURSLARGEUR(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v

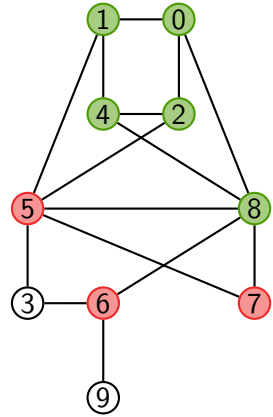


- File : 4 5 6 7
- Affichage : 0 1 2 8

Parcours en largeur

Algorithme : ParcoursLargeur(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v

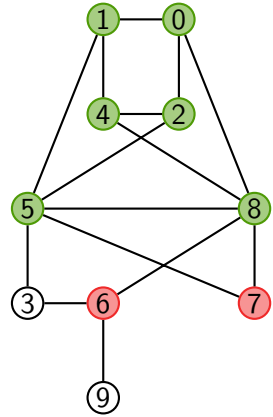


- File : 5 6 7
- Affichage : 0 1 2 8 4

Parcours en largeur

Algorithme : PARCOURSLARGEUR(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v

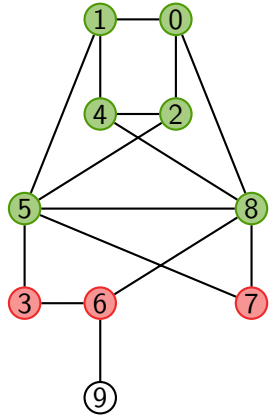


- File : 6 7
- Affichage : 0 1 2 8 4 5

Parcours en largeur

Algorithme : PARCOURSLARGEUR(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v

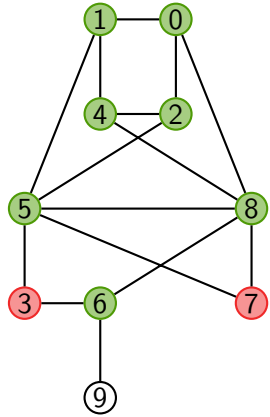


- File : 6 7 3
- Affichage : 0 1 2 8 4 5

Parcours en largeur

Algorithme : PARCOURSLARGEUR(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v

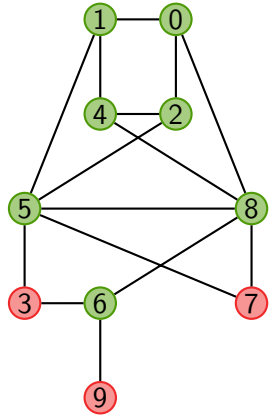


- File : 7 3
- Affichage : 0 1 2 8 4 5 6

Parcours en largeur

Algorithme : PARCOURSLARGEUR(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v

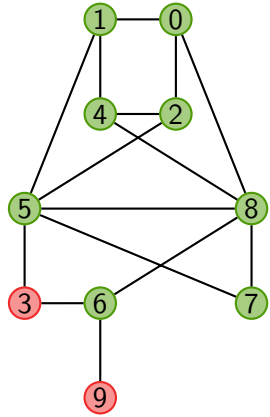


- File : 7 3 9
- Affichage : 0 1 2 8 4 5 6

Parcours en largeur

Algorithme : PARCOURSLARGEUR(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v

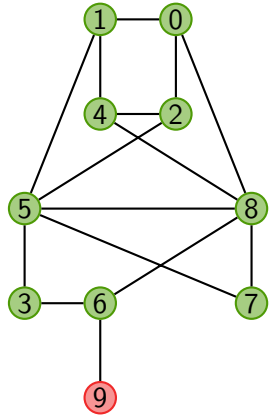


- File : 3 9
- Affichage : 0 1 2 8 4 5 6 7

Parcours en largeur

Algorithme : PARCOURSLARGEUR(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v

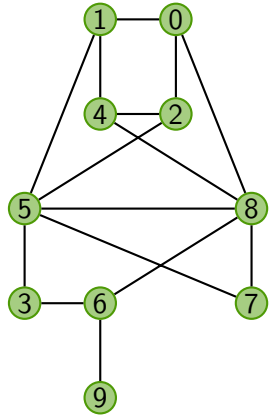


- File : 9
- Affichage : 0 1 2 8 4 5 6 7 3

Parcours en largeur

Algorithme : PARCOURSLARGEUR(G, s)

$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v



- File :
- Affichage : 0 1 2 8 4 5 6 7 3 9

Propriétés du parcours en largeur

Théorème

PARCOURSLARGEUR(G, s) *affiche une fois et une seule chaque sommet de la composante connexe de* s. *Sa complexité est*

- $O(n^2)$ *si le graphe est représenté par matrice d'adjacence*
- $O(m + n)$ *si le graphe est représenté par listes d'adjacence*

où n *est le nombre de sommets et* m *le nombre d'arêtes.*

Propriétés du parcours en largeur

Théorème

PARCOURSLARGEUR(G, s) *affiche une fois et une seule chaque sommet de la composante connexe de s. Sa complexité est*

- $O(n^2)$ *si le graphe est représenté par matrice d'adjacence*
- $O(m+n)$ *si le graphe est représenté par listes d'adjacence*

où n est le nombre de sommets et m le nombre d'arêtes.

Preuve de complexité :

- Matrice : pour chaque sommet, parcours de la ligne correspondante
- Liste : parcours de toutes les listes ; somme des longueurs $= 2m$ ■

Propriétés du parcours en largeur

Théorème

PARCOURSLARGEUR(G, s) *affiche une fois et une seule chaque sommet de la composante connexe de* s*. Sa complexité est*

- $O(n^2)$ *si le graphe est représenté par matrice d'adjacence*
- $O(m + n)$ *si le graphe est représenté par listes d'adjacence*

où n *est le nombre de sommets et* m *le nombre d'arêtes.*

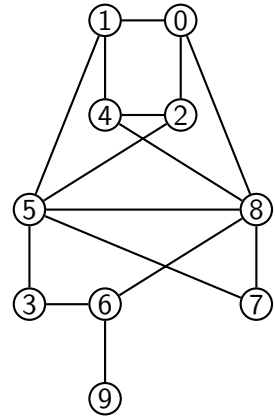
Preuve de correction :

- récurrence sur la distance à s ■

Parcours en profondeur

Algorithme : PARCOURSLARGEUR(G, s)

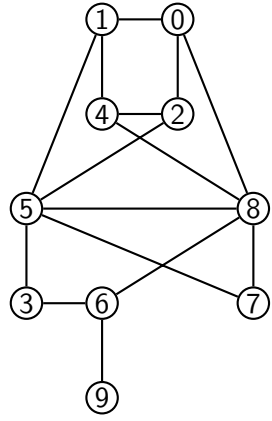
$F \leftarrow$ file vide
Ajouter s à F et marquer s
tant que F *est non vide* **:**
    $u \leftarrow$ défiler un élément de F
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à F et marquer v



Parcours en profondeur

Algorithme : PARCOURSPROFONDEUR(G, s)

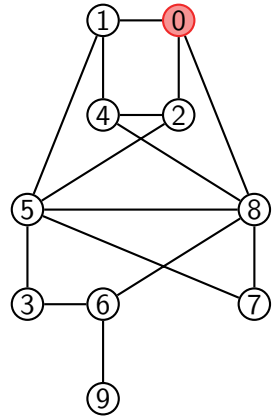
$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v



Parcours en profondeur

Algorithme : ParcoursProfondeur(G, s)

$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v

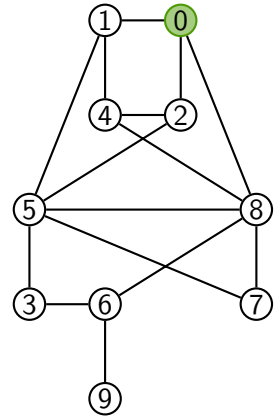


- Pile : 0
- Affichage :

Parcours en profondeur

Algorithme : PARCOURSPROFONDEUR(G, s)

$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v

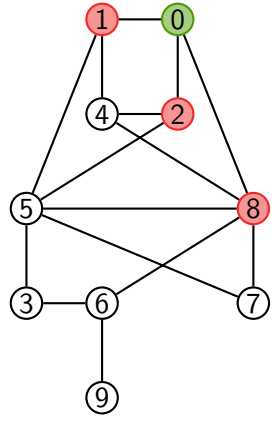


- Pile :
- Affichage : 0

Parcours en profondeur

Algorithme : PARCOURSPROFONDEUR(G, s)

$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v

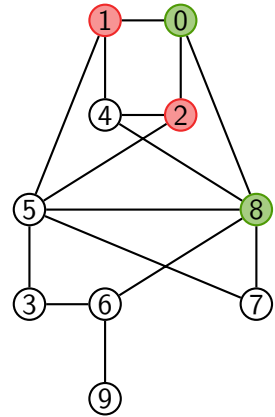


- Pile : 1 2 8
- Affichage : 0

Parcours en profondeur

Algorithme : PARCOURSPROFONDEUR(G, s)

$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v

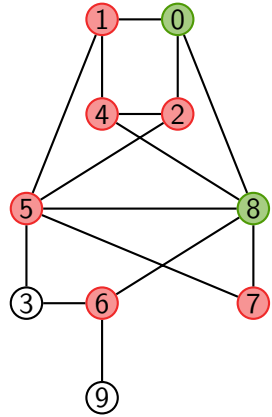


- Pile : 1 2
- Affichage : 0 8

Parcours en profondeur

Algorithme : PARCOURSPROFONDEUR(G, s)

$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v

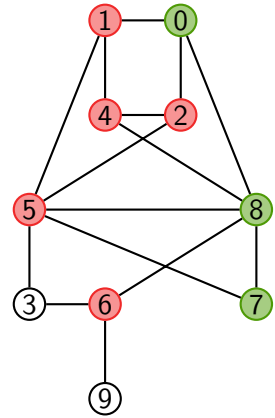


- Pile : 1 2 4 5 6 7
- Affichage : 0 8

Parcours en profondeur

Algorithme : PARCOURSPROFONDEUR(G, s)

$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v

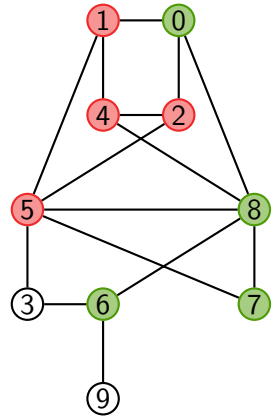


- Pile : 1 2 4 5 6
- Affichage : 0 8 7

Parcours en profondeur

Algorithme : PARCOURSPROFONDEUR(G, s)

$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v

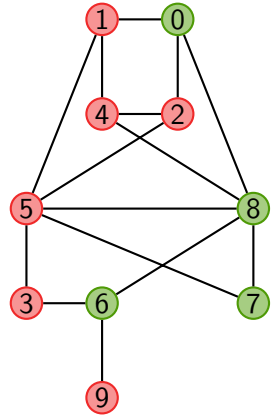


- Pile : 1 2 4 5
- Affichage : 0 8 7 6

Parcours en profondeur

Algorithme : PARCOURSPROFONDEUR(G, s)

$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v

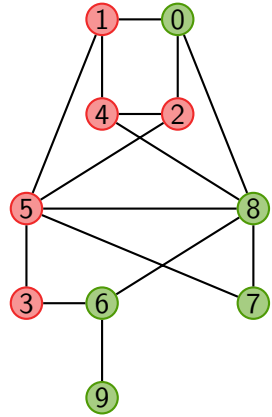


- Pile : 1 2 4 5 3 9
- Affichage : 0 8 7 6

Parcours en profondeur

Algorithme : PARCOURSPROFONDEUR(G, s)

$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v

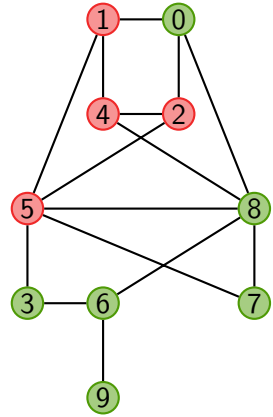


- Pile : 1 2 4 5 3
- Affichage : 0 8 7 6 9

Parcours en profondeur

Algorithme : PARCOURSPROFONDEUR(G, s)

$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v

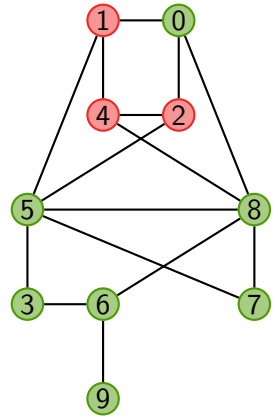


- Pile : 1 2 4 5
- Affichage : 0 8 7 6 9 3

Parcours en profondeur

Algorithme : PARCOURSPROFONDEUR(G, s)

$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v

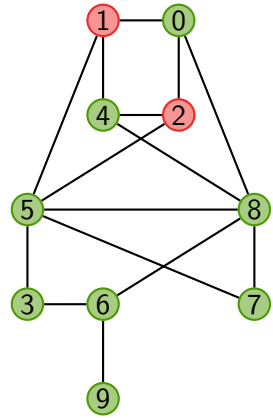


- Pile : 1 2 4
- Affichage : 0 8 7 6 9 3 5

Parcours en profondeur

Algorithme : PARCOURSPROFONDEUR(G, s)

$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v

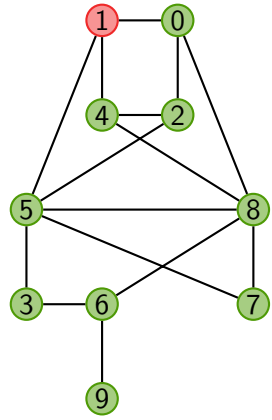


- Pile : 1 2
- Affichage : 0 8 7 6 9 3 5 4

Parcours en profondeur

Algorithme : ParcoursProfondeur(G, s)

$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v

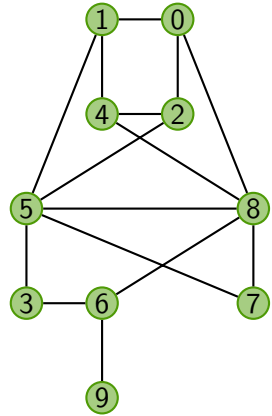


- Pile : 1
- Affichage : 0 8 7 6 9 3 5 4 2

Parcours en profondeur

Algorithme : PARCOURSPROFONDEUR(G, s)

$P \leftarrow$ pile vide
Ajouter s à P et marquer s
tant que P *est non vide* **:**
    $u \leftarrow$ dépiler un élément de P
    Afficher u
    pour *tout voisin non marqué* v *de* u **:**
        Ajouter v à P et marquer v



- Pile :
- Affichage : 0 8 7 6 9 3 5 4 2 1

Propriétés du parcours en profondeur

Théorème

PARCOURSPROFONDEUR(G, s) *affiche une fois et une seule chaque sommet de la composante connexe de* s*. Sa complexité est*

- $O(n^2)$ *si le graphe est représenté par matrice d'adjacence*
- $O(m + n)$ *si le graphe est représenté par listes d'adjacence*

où n *est le nombre de sommets et* m *le nombre d'arêtes.*

Propriétés du parcours en profondeur

Théorème

PARCOURSPROFONDEUR(G, s) *affiche une fois et une seule chaque sommet de la composante connexe de s. Sa complexité est*

- $O(n^2)$ *si le graphe est représenté par matrice d'adjacence*
- $O(m + n)$ *si le graphe est représenté par listes d'adjacence*

où n est le nombre de sommets et m le nombre d'arêtes.

Preuve : Identique au cas du parcours en largeur !

Arbres binaires et graphes

- Un arbre binaire **est** un graphe particulier :
  - connexe — sinon *forêt*
  - sans cycle — pour être un arbre
  - sommets de degrés 1, 2 ou 3 — pour être binaire

Arbres binaires et graphes

- Un arbre binaire **est** un graphe particulier :
  - connexe — sinon *forêt*
  - sans cycle — pour être un arbre
  - sommets de degrés 1, 2 ou 3 — pour être binaire
- Un arbre binaire **n'est pas** un graphe :
  - racine identifiée — sommet de degré 1 ou 2
  - distinction fils gauche / fils droit
  - Définition récursive et représentation informatique bien différentes !

Arbres binaires et graphes

- Un arbre binaire **est** un graphe particulier :
  - connexe — sinon *forêt*
  - sans cycle — pour être un arbre
  - sommets de degrés 1, 2 ou 3 — pour être binaire
- Un arbre binaire **n'est pas** un graphe :
  - racine identifiée — sommet de degré 1 ou 2
  - distinction fils gauche / fils droit
  - Définition récursive et représentation informatique bien différentes !
- Algorithmes de parcours de graphes :
  - besoin de *marquer* les sommets
  - besoin d'une *pile* pour le parcours en profondeur
  - algorithme essentiellement identique pour le parcours en largeur (avec *file*)

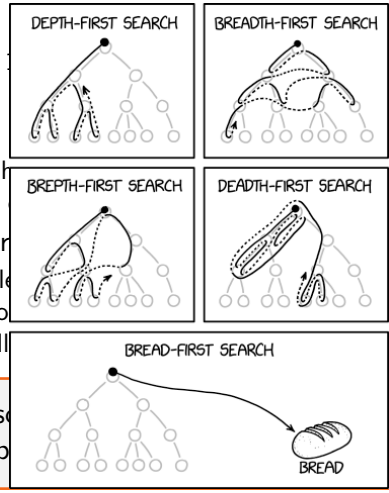
Arbres binaires et graphes

- Un arbre binaire **est** un graphe particulier :
  - connexe — sinon *forêt*
  - sans cycle — pour être un arbre
  - sommets de degrés 1, 2 ou 3 — pour être binaire
- Un arbre binaire **n'est pas** un graphe :
  - racine identifiée — sommet de degré 1 ou 2
  - distinction fils gauche / fils droit
  - Définition récursive et représentation informatique bien différentes !
- Algorithmes de parcours de graphes :
  - besoin de *marquer* les sommets
  - besoin d'une *pile* pour le parcours en profondeur
  - algorithme essentiellement identique pour le parcours en largeur (avec *file*)

Les arbres et les graphes sont deux structures de données proches, mais différentes.
Les algorithmes sur les arbres sont souvent plus simples que sur les graphes.

Arbres binaires et graphes

- Un arbre binaire **est** un graphe particulier :
  - connexe
  - sans cycle
  - sommets de degrés 1
- Un arbre binaire **n'est**
  - racine identifiée
  - distinction fils gauch
  - Définition récursive
- Algorithmes de parcour
  - besoin de *marquer* l
  - besoin d'une *pile* po
  - algorithme essentiell



sinon *forêt*
pour être un arbre
pour être binaire

sommet de degré 1 ou 2

différentes !

en largeur (avec *file*)

Les arbres et les graphes s proches, mais différentes.
Les algorithmes sur les arb ue sur les graphes.

https://xkcd.com/2407

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 $\ll$ bonne $\gg$ 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 $\ll$ bonne $\gg$ 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 $\ll$ bonne $\gg$ 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 $\ll$ bonne $\gg$ 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,

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

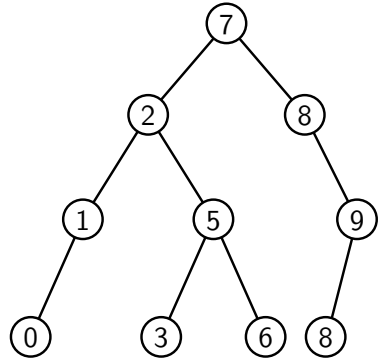
Définition

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,

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



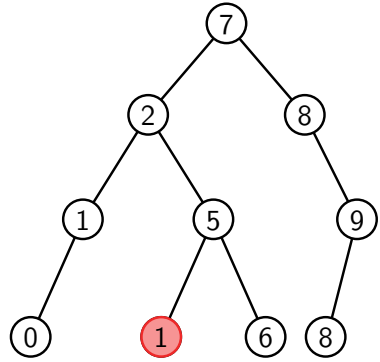
Définition

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,

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



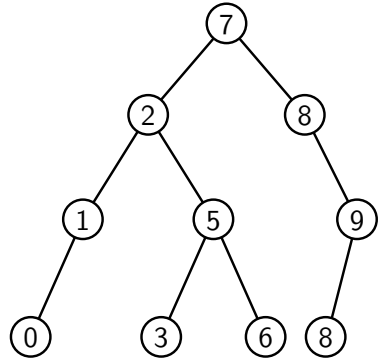
Définition

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,

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

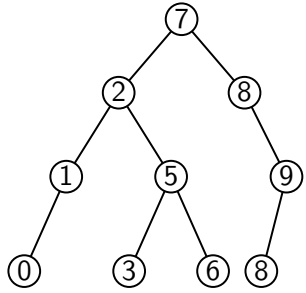


Parcours infixe d'un ABR

Algorithme : ParcoursInfixe(x)

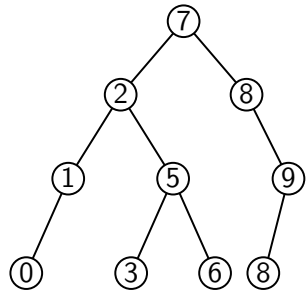
si $x \neq \emptyset$ **:**

- ParcoursInfixe(filsG(x))
- Afficher val(x)
- ParcoursInfixe(filsD(x))



Parcours infixe d'un ABR

Algorithme : ParcoursInfixe(x)
si $x \neq \emptyset$ **:**
    ParcoursInfixe(filsG(x))
    Afficher val(x)
    ParcoursInfixe(filsD(x))



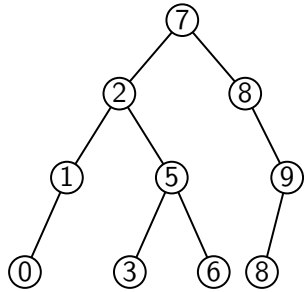
Lemme

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 $\text{val}(y) \leq \text{val}(\text{rac}(A)) \leq \text{val}(z)$ pour $y \in \text{saG}(A)$, $z \in \text{saD}(A)$
ssi A est un ABR

Recherche dans un ABR

Algorithme : RECHERCHER(x, k)
si $x = \emptyset$ **: renvoyer** $\emptyset$
si $\text{val}(x) = k$ **: renvoyer** x
si $\text{val}(x) > k$ **:**
  renvoyer RECHERCHER$(\text{filsG}(x), k)$
renvoyer RECHERCHER$(\text{filsD}(x), k)$



Recherche dans un ABR

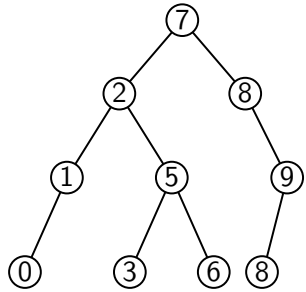
Algorithme : RECHERCHER(x, k)

tant que $x \neq \emptyset$ *et* $\text{val}(x) \neq k$ **:**

  si $k < \text{val}(x)$ **:** $x \leftarrow \text{filsG}(x)$

  sinon : $x \leftarrow \text{filsD}(x)$

renvoyer x



Recherche dans un ABR

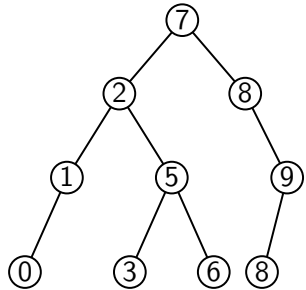
Algorithme : RECHERCHER(x, k)

tant que $x \neq \emptyset$ *et* $\text{val}(x) \neq k$ **:**

  si $k < \text{val}(x)$ **:** $x \leftarrow \text{filsG}(x)$

  sinon : $x \leftarrow \text{filsD}(x)$

renvoyer x



Correction admise

Recherche dans un ABR

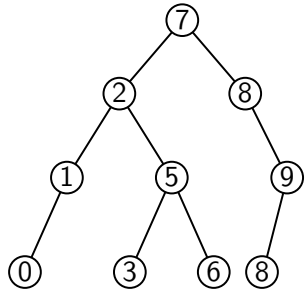
Algorithme : RECHERCHER(x, k)

tant que $x \neq \emptyset$ *et* $\text{val}(x) \neq k$ **:**

  si $k < \text{val}(x)$ **:** $x \leftarrow \text{filsG}(x)$

  sinon : $x \leftarrow \text{filsD}(x)$

renvoyer x



Correction admise

Lemme

RECHERCHER($\text{rac}(A), k$) *a une complexité* $O(h(A))$.

Preuve

- À chaque itération, la hauteur de x augmente de 1 : $\leq h(A)$ itérations
- Chaque itération coûte $O(1)$

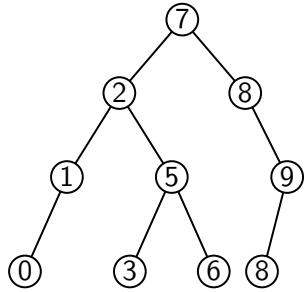
Minimum et successeur

Algorithme : MINIMUM(x)

tant que filsG(x) $\neq \emptyset$ **:**

  $x \leftarrow$ filsG(x)

renvoyer x



Minimum et successeur

Algorithme : MINIMUM(x)

tant que filsG(x) $\neq \emptyset$ **:**
  $x \leftarrow$ filsG(x)

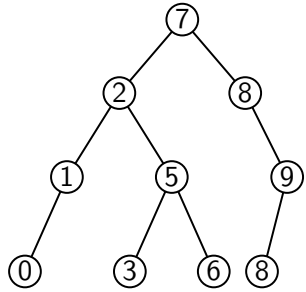
renvoyer x

Algorithme : SUCCESSEUR(x)

si filsD(x) $\neq \emptyset$ **:**
  renvoyer MINIMUM(filsD(x))

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

renvoyer y



Minimum et successeur

Algorithme : MINIMUM(x)

tant que filsG(x) $\neq \emptyset$ **:**
  $x \leftarrow$ filsG(x)

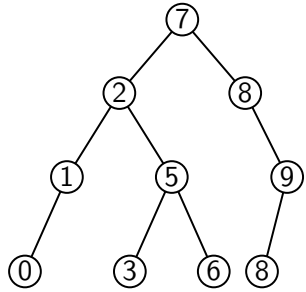
renvoyer x

Algorithme : SUCCESSEUR(x)

si filsD(x) $\neq \emptyset$ **:**
  renvoyer MINIMUM(filsD(x))

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

renvoyer y



SUCCESSEUR(2) = 3

Minimum et successeur

Algorithme : MINIMUM(x)

tant que filsG(x) $\neq \emptyset$ **:**
  $x \leftarrow$ filsG(x)

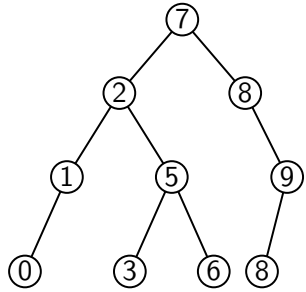
renvoyer x

Algorithme : SUCCESSEUR(x)

si filsD(x) $\neq \emptyset$ **:**
  renvoyer MINIMUM(filsD(x))

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

renvoyer y



SUCCESSEUR(2) = 3
SUCCESSEUR(6) = 7

Minimum et successeur

Algorithme : MINIMUM(x)

tant que filsG(x) $\neq \emptyset$ **:**
  $x \leftarrow$ filsG(x)

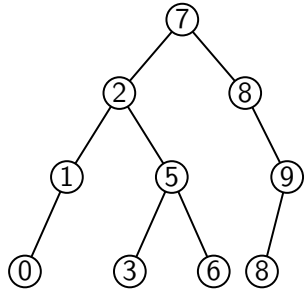
renvoyer x

Algorithme : SUCCESSEUR(x)

si filsD(x) $\neq \emptyset$ **:**
  renvoyer MINIMUM(filsD(x))

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

renvoyer y



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

Minimum et successeur

Algorithme : MINIMUM(x)

tant que filsG(x) $\neq \emptyset$ **:**
  $x \leftarrow$ filsG(x)

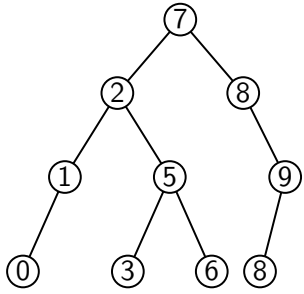
renvoyer x

Algorithme : SUCCESSEUR(x)

si filsD(x) $\neq \emptyset$ **:**
  renvoyer MINIMUM(filsD(x))

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

renvoyer y



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

Lemme

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

Correction de successeur

Algorithme : SUCCESSEUR(x)

si filsD(x) $\neq \emptyset$ **:**
  renvoyer MINIMUM(filsD(x))

$p \leftarrow$ père(x)
tant que $p \neq \emptyset$ *et* $x =$ filsD(p) **:**
  $(x, p) \leftarrow (p,$ père$(x))$

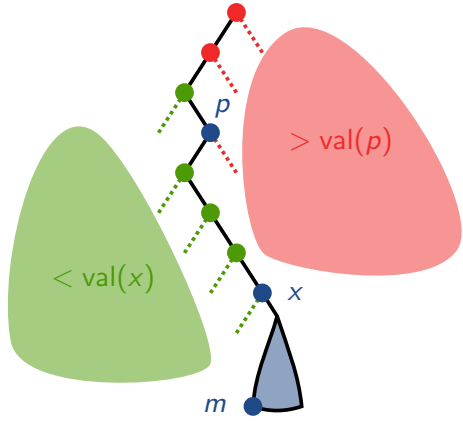
renvoyer p

Lemme

SUCCESSEUR *renvoie un nœud de valeur minimale parmi ceux dont la valeur est* $\geq$ val(x).

Correction de successeur

Algorithme : SUCCESSEUR(x)
si filsD(x) $\neq \emptyset$ **:**
  renvoyer MINIMUM(filsD(x))
$p \leftarrow$ père(x)
tant que $p \neq \emptyset$ *et* $x =$ filsD(p) **:**
  $(x, p) \leftarrow (p,$ père(x))
renvoyer p



Lemme

SUCCESSEUR *renvoie un nœud de valeur minimale parmi ceux dont la valeur est* $\geq$ val(x).

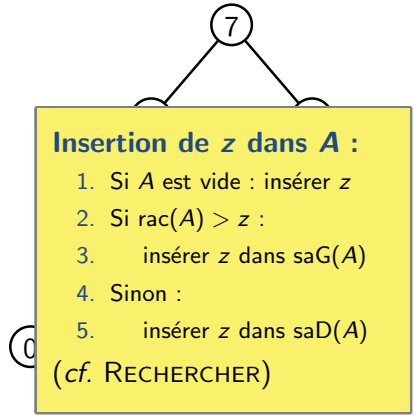
Preuve Supp. les valeurs 2-à-2 distinctes.

- si filsD(x) $\neq \emptyset$ (m existe) : val(x) $<$ val(m) $<$ val(p)
- pour tout *ancêtre* $z \neq p$ de x, deux possibilités :
  - $x \in$ saD(z) $\rightsquigarrow$ val(z) $<$ val(x) et $\forall y \in$ saG(z), val(y) $<$ val(z) $<$ val(x)
  - $p \in$ saG(z) $\rightsquigarrow$ val(z) $>$ val(p) et $\forall y \in$ saD(z), val(y) $>$ val(z) $>$ val(p)

Insertion d'un élément

Algorithme : INSÉRER(A, z)

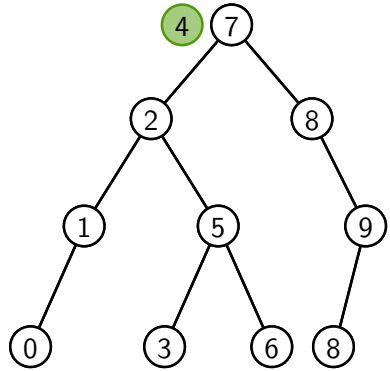
$x \leftarrow \text{rac}(A)$
$p \leftarrow \emptyset$
tant que $x \neq \emptyset$ **:**
    $p \leftarrow x$
    si $\text{val}(z) < \text{val}(x)$ **:** $x \leftarrow \text{filsG}(x)$
    sinon : $x \leftarrow \text{filsD}(x)$
père$(z) \leftarrow p$
si $p = \emptyset$ **:** $\text{rac}(A) \leftarrow z$
sinon si $\text{val}(z) < \text{val}(p)$ **:** $\text{filsG}(p) \leftarrow z$
sinon : $\text{filsD}(p) \leftarrow z$



Insertion d'un élément

Algorithme : INSÉRER(A, z)

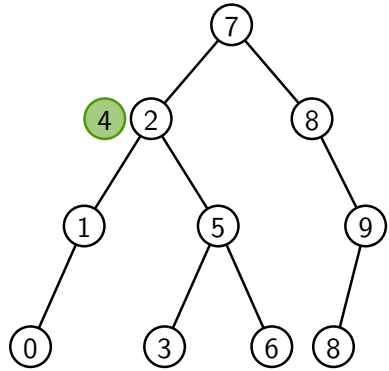
$x \leftarrow \text{rac}(A)$
$p \leftarrow \emptyset$
tant que $x \neq \emptyset$ **:**
    $p \leftarrow x$
    si $\text{val}(z) < \text{val}(x)$ **:** $x \leftarrow \text{filsG}(x)$
    sinon : $x \leftarrow \text{filsD}(x)$
père$(z) \leftarrow p$
si $p = \emptyset$ **:** $\text{rac}(A) \leftarrow z$
sinon si $\text{val}(z) < \text{val}(p)$ **:** $\text{filsG}(p) \leftarrow z$
sinon : $\text{filsD}(p) \leftarrow z$



Insertion d'un élément

Algorithme : INSÉRER(A, z)

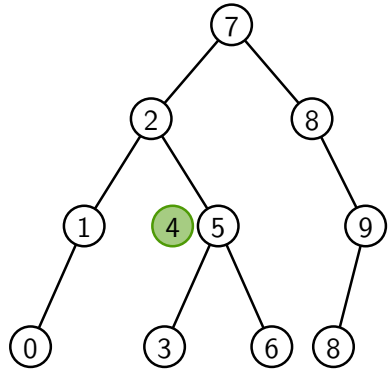
$x \leftarrow \text{rac}(A)$
$p \leftarrow \emptyset$
tant que $x \neq \emptyset$ **:**
    $p \leftarrow x$
    si $\text{val}(z) < \text{val}(x)$ **:** $x \leftarrow \text{filsG}(x)$
    sinon : $x \leftarrow \text{filsD}(x)$
père$(z) \leftarrow p$
si $p = \emptyset$ **:** $\text{rac}(A) \leftarrow z$
sinon si $\text{val}(z) < \text{val}(p)$ **:** $\text{filsG}(p) \leftarrow z$
sinon : $\text{filsD}(p) \leftarrow z$



Insertion d'un élément

Algorithme : INSÉRER(A, z)

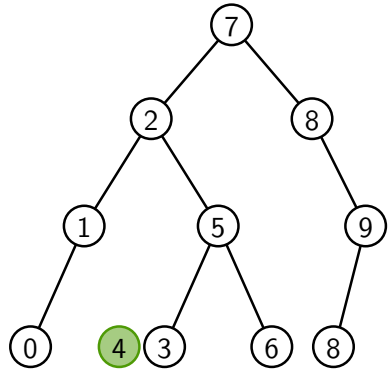
$x \leftarrow \text{rac}(A)$
$p \leftarrow \emptyset$
tant que $x \neq \emptyset$ **:**
    $p \leftarrow x$
    si $\text{val}(z) < \text{val}(x)$ **:** $x \leftarrow \text{filsG}(x)$
    sinon : $x \leftarrow \text{filsD}(x)$
$\text{père}(z) \leftarrow p$
si $p = \emptyset$ **:** $\text{rac}(A) \leftarrow z$
sinon si $\text{val}(z) < \text{val}(p)$ **:** $\text{filsG}(p) \leftarrow z$
sinon : $\text{filsD}(p) \leftarrow z$



Insertion d'un élément

Algorithme : INSÉRER(A, z)

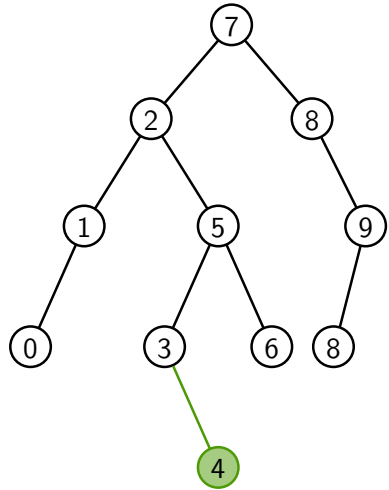
$x \leftarrow \text{rac}(A)$
$p \leftarrow \emptyset$
tant que $x \neq \emptyset$ **:**
    $p \leftarrow x$
    si $\text{val}(z) < \text{val}(x)$ **:** $x \leftarrow \text{filsG}(x)$
    sinon : $x \leftarrow \text{filsD}(x)$
$\text{père}(z) \leftarrow p$
si $p = \emptyset$ **:** $\text{rac}(A) \leftarrow z$
sinon si $\text{val}(z) < \text{val}(p)$ **:** $\text{filsG}(p) \leftarrow z$
sinon : $\text{filsD}(p) \leftarrow z$



Insertion d'un élément

Algorithme : INSÉRER(A, z)

$x \leftarrow \text{rac}(A)$
$p \leftarrow \emptyset$
tant que $x \neq \emptyset$ **:**
    $p \leftarrow x$
    si $\text{val}(z) < \text{val}(x)$ **:** $x \leftarrow \text{filsG}(x)$
    sinon : $x \leftarrow \text{filsD}(x)$
père$(z) \leftarrow p$
si $p = \emptyset$ **:** $\text{rac}(A) \leftarrow z$
sinon si $\text{val}(z) < \text{val}(p)$ **:** $\text{filsG}(p) \leftarrow z$
sinon : $\text{filsD}(p) \leftarrow z$



Suppression d'un élément

Algorithme de remplacement du sous-arbre enraciné en x par celui enraciné en z dans l'arborescence A

- Les 2 pointeurs $x \leftrightarrows \text{père}(x)$ sont remplacés par 2 pointeurs $z \leftrightarrows \text{père}(z)$

Algorithme : REMPLACE(A, x, z)

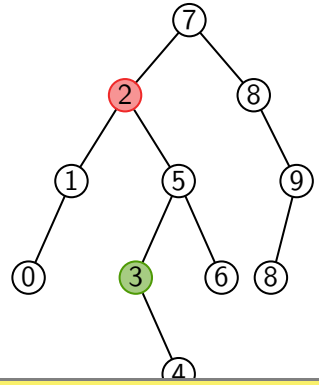
$p \leftarrow \text{père}(x)$
père$(x) \leftarrow \emptyset$
si $p = \emptyset$ **:** rac$(A) \leftarrow z$
sinon si $x = \text{filsG}(p)$ **:** filsG$(p) \leftarrow z$
sinon : filsD$(p) \leftarrow z$

si $z \neq \emptyset$ **:** père$(z) \leftarrow p$

Suppression d'un élément

Algorithme : SUPPRIMER(A, z)

si filsG(z) $= \emptyset$ **:** REMPLACE($A, z,$ filsD(z))
sinon si filsD(z) $= \emptyset$ **:** REMPLACE($A, z,$ filsG(z))
sinon :
  $y \leftarrow$ SUCCESSEUR(z)
  REMPLACE($A, y,$ filsD(y))
  filsD(y) $\leftarrow$ filsD(z) ; filsD(z) $\leftarrow \emptyset$
  filsG(y) $\leftarrow$ filsG(z) ; filsG(z) $\leftarrow \emptyset$
  si filsD(y) $\neq \emptyset$ **:** père(filsD(y)) $= y$
  si filsG(y) $\neq \emptyset$ **:** père(filsG(y)) $= y$
  REMPLACE(A, z, y)

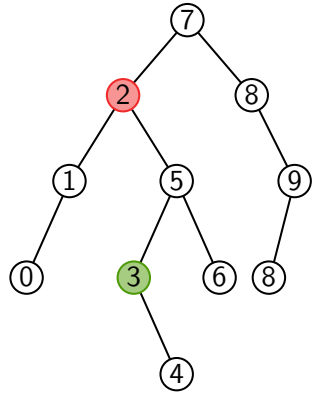


Supprimer z de A :

1. Enlever le successeur y de z de A
2. Déplacer les fils de z comme fils de y
3. Remplacer z par y

Suppression d'un élément

Algorithme : SUPPRIMER(A, z)
si filsG$(z) = \emptyset$ **:** REMPLACE$(A, z,$ filsD$(z))$
sinon si filsD$(z) = \emptyset$ **:** REMPLACE$(A, z,$ filsG$(z))$
sinon :
  $y =$ SUCCESSEUR(z)
  REMPLACE$(A, y,$ filsD$(y))$
  filsD$(y) \leftarrow$ filsD(z) ; filsD$(z) \leftarrow \emptyset$
  filsG$(y) \leftarrow$ filsG(z) ; filsG$(z) \leftarrow \emptyset$
  si filsD$(y) \neq \emptyset$ **:** père(filsD$(y)) = y$
  si filsG$(y) \neq \emptyset$ **:** père(filsG$(y)) = y$
  REMPLACE(A, z, y)



Suppression d'un élément

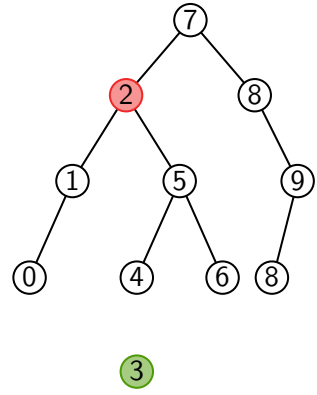
Algorithme : SUPPRIMER(A, z)

si filsG$(z) = \emptyset$ **:** REMPLACE$(A, z,$ filsD$(z))$

sinon si filsD$(z) = \emptyset$ **:** REMPLACE$(A, z,$ filsG$(z))$

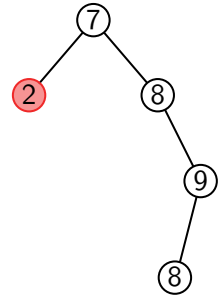
sinon :

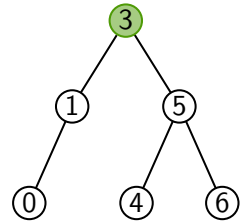
- $y =$ SUCCESSEUR(z)
- REMPLACE$(A, y,$ filsD$(y))$
- filsD$(y) \leftarrow$ filsD(z) ; filsD$(z) \leftarrow \emptyset$
- filsG$(y) \leftarrow$ filsG(z) ; filsG$(z) \leftarrow \emptyset$
- **si** filsD$(y) \neq \emptyset$ **:** père(filsD$(y)) = y$
- **si** filsG$(y) \neq \emptyset$ **:** père(filsG$(y)) = y$
- REMPLACE(A, z, y)



Suppression d'un élément

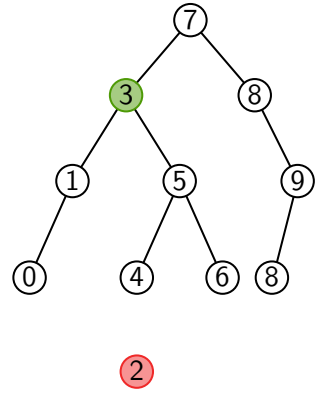
Algorithme : SUPPRIMER(A, z)
si filsG$(z) = \emptyset$ **:** REMPLACE$(A, z,$ filsD$(z))$
sinon si filsD$(z) = \emptyset$ **:** REMPLACE$(A, z,$ filsG$(z))$
sinon :
  $y =$ SUCCESSEUR(z)
  REMPLACE$(A, y,$ filsD$(y))$
  filsD$(y) \leftarrow$ filsD(z) ; filsD$(z) \leftarrow \emptyset$
  filsG$(y) \leftarrow$ filsG(z) ; filsG$(z) \leftarrow \emptyset$
  si filsD$(y) \neq \emptyset$ **:** père(filsD$(y)) = y$
  si filsG$(y) \neq \emptyset$ **:** père(filsG$(y)) = y$
  REMPLACE(A, z, y)





Suppression d'un élément

Algorithme : SUPPRIMER(A, z)
si filsG$(z) = \emptyset$ **:** REMPLACE$(A, z,$ filsD$(z))$
sinon si filsD$(z) = \emptyset$ **:** REMPLACE$(A, z,$ filsG$(z))$
sinon :
  $y =$ SUCCESSEUR(z)
  REMPLACE$(A, y,$ filsD$(y))$
  filsD$(y) \leftarrow$ filsD(z) ; filsD$(z) \leftarrow \emptyset$
  filsG$(y) \leftarrow$ filsG(z) ; filsG$(z) \leftarrow \emptyset$
  si filsD$(y) \neq \emptyset$ **:** père(filsD$(y)) = y$
  si filsG$(y) \neq \emptyset$ **:** père(filsG$(y)) = y$
  REMPLACE(A, z, y)



Correction et complexités

Lemme

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

Correction et complexités

Lemme

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

Lemme

INSÉRER *et* SUPPRIMER *ont une complexité* $O(h(A))$.

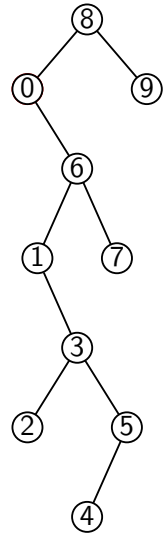
Preuve On parcourt *une branche de l'arbre* pour trouver soit l'endroit où insérer (INSÉRER) soit le successeur (SUPPRIMER) : complexité $O(h(A))$. Le reste est un nombre constant de modifications de pointeurs. ■

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 !



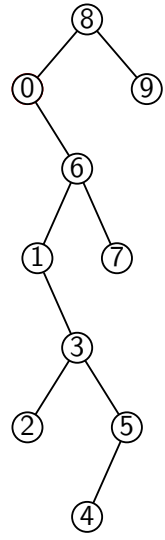
Motivation

Rappel des complexités

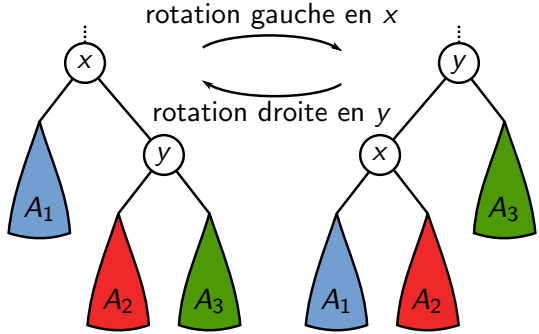
- Insérer et Supprimer : $O(h(A))$
- Minimum et Maximum[1] : $O(h(A))$
- Rechercher : $O(h(A))$
- Successeur et Prédecesseur[1] : $O(h(A))$

[1] Exercice !

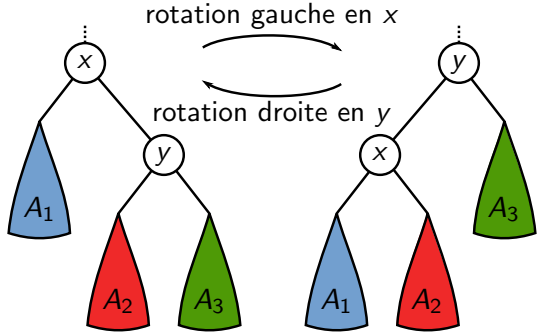
Un ABR est une structure de donnée efficace s'il est **équilibré**, c'est-à-dire si $h(A) = O(\log(n(A)))$.



Outil de base : les rotations



Outil de base : les rotations



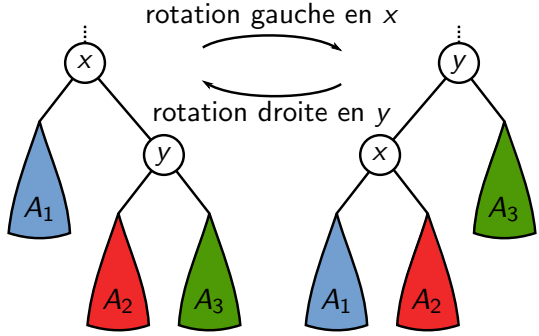
Lemme

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



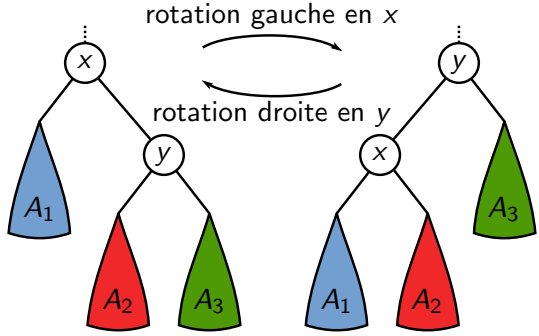
Lemme

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

Outil de base : les rotations

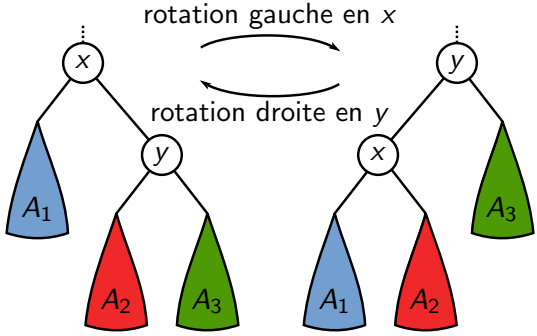


Lemme

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

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

Outil de base : les rotations



Lemme

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

- 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 uniquement 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)
  - EXTRAIREMAX ou EXTRAIREMIN : retire l'élément de priorité maximale ou minimale
  - CHANGERPRIORITÉ : modifie 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)
  - EXTRAIREMAX ou EXTRAIREMIN : retire l'élément de priorité maximale ou minimale
  - CHANGERPRIORITÉ : modifie la priorité d'un élément
- Utilisation de files de priorité
  - Trouver le chemin le plus court entre deux points
    - dans un graphe (Dijkstra) ⇝ partie 2 du cours
    - sur une carte (A*, ...)
  - Répartition de charge entre serveurs, ordonnancement de processus...
  - Priorités aléatoires pour équilibrer des ABR

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)
  - EXTRAIREMAX ou EXTRAIREMIN : retire l'élément de priorité maximale ou minimale
  - CHANGERPRIORITÉ : modifie la priorité d'un élément
- Utilisation de files de priorité
  - Trouver le chemin le plus court entre deux points
    - dans un graphe (Dijkstra)
    - sur une carte (A*, ...)

    ⇝ partie 2 du cours
  - Répartition de charge entre serveurs, ordonnancement de processus...
  - Priorités aléatoires pour équilibrer des ABR

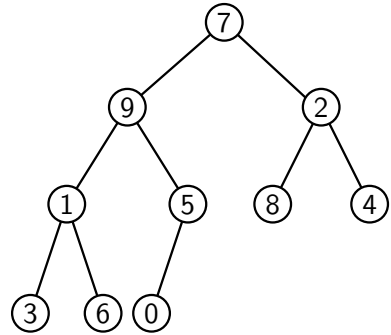
Le tas est **une** structure de donnée permettant d'implanter les files de priorités, mais les autres sont en général des extensions.

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* »

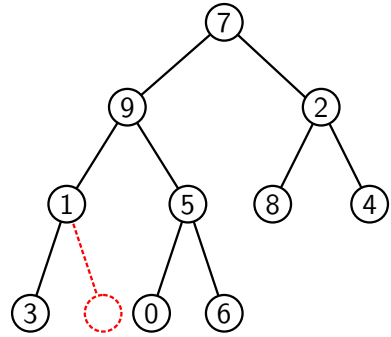


Arbres quasi-complets

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* »

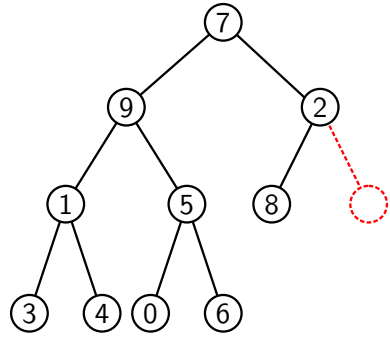


Arbres quasi-complets

Définition

Un arbre binaire est **quasi-complet** si

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

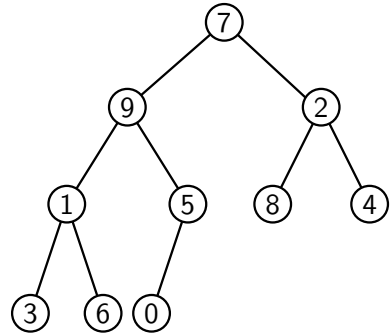


Arbres quasi-complets

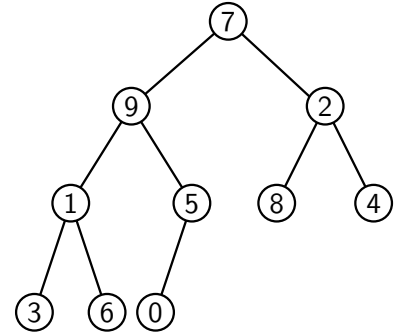
Définition

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, $2^{h(A)} \leq n(A) < 2^{h(A)+1}$.

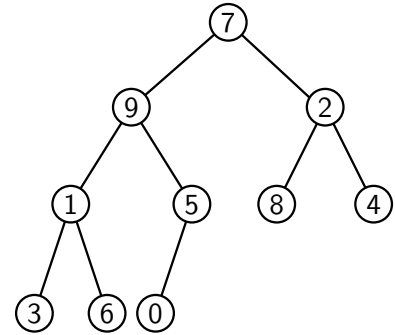
Preuve La borne supérieure est vraie pour tout arbre.

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, $2^{h(A)} \leq n(A) < 2^{h(A)+1}$.

Corollaire

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

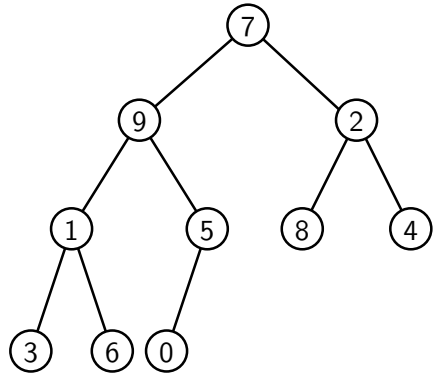
Preuve On a $2^{h(A)} \leq n(A) < 2^{h(A)+1}$ et donc $h(A) \leq \log n(A) < h(A) + 1$. ■

Numérotation des arbres quasi-complets

Définition

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

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

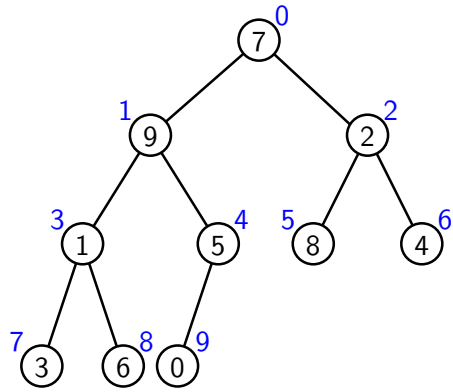


Numérotation des arbres quasi-complets

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 $\text{filsD}(x) \neq \emptyset$, $\text{num}(\text{filsD}(x)) = 2\,\text{num}(x) + 2$

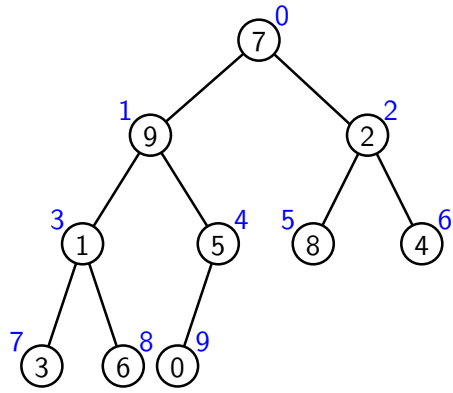


Numérotation des arbres quasi-complets

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 $x \in N_k$, $2^k - 1 \leq \text{num}(x) \leq 2^{k+1} - 2$ (récurrence)
   - $k = 0$: la racine a le numéro 0
   - $2^{k-1} - 1 \leq \text{num}(\text{père}(x)) \leq 2^k - 2$ car $\text{père}(x) \in N_{k-1}$ si $x \in N_k$ (HR)
     Or $2\,\text{num}(\text{père}(x)) + 1 \leq \text{num}(x) \leq 2\,\text{num}(\text{père}(x)) + 2$
     Donc $2 \cdot (2^{k-1} - 1) + 1 \leq \text{num}(x) \leq 2 \cdot (2^k - 2) + 2$

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 $x \in N_k$, $2^k - 1 \leq \text{num}(x) \leq 2^{k+1} - 2$ (récurrence)
2. Si x est le *voisin de gauche* de y, $\text{num}(y) = \text{num}(x) + 1$
   - Si $\text{num}(x) = 2p + 1$, $x = \text{filsG}(\text{père}(x))$. Donc $y = \text{filsD}(\text{père}(x))$ et $\text{num}(y) = 2p + 2$
   - Si $\text{num}(x) = 2p + 2$, $x = \text{filsD}(\text{père}(x))$. Donc $y = \text{filsG}(\text{père}(y))$; et $\text{père}(x)$ est *voisin de gauche* de $\text{père}(y) \rightsquigarrow \text{num}(\text{père}(y)) = \text{num}(\text{père}(x)) + 1$

$$\text{num}(y) = 2\,\text{num}(\text{père}(y)) + 1 = 2\,\text{num}(\text{père}(x) + 1) + 1 = \text{num}(x) + 1$$

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

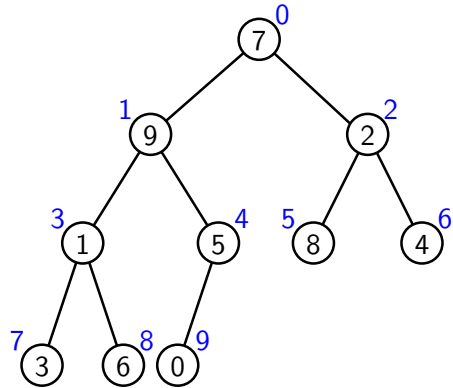
1. Si $x \in N_k$, $2^k - 1 \leq \text{num}(x) \leq 2^{k+1} - 2$ (récurrence)
2. Si x est le *voisin de gauche* de y, $\text{num}(y) = \text{num}(x) + 1$

D'après 1. : numéros 0 à $2^h - 2$ $\iff$ niveaux complets jusqu'à N_{h-1}
D'après 2. : numéros entre $2^h - 1$ et $n(A) - 1$ $\iff$ N_h rempli *par la gauche* ■

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 $\text{num}(x)$.



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

Représentation informatique des arbres quasi-complets

Corollaire

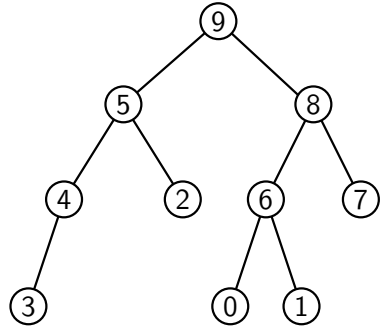
On peut représenter un arbre quasi-complet par un tableau de taille $n(A)$ contenant $\text{val}(x)$ en case $\text{num}(x)$.

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

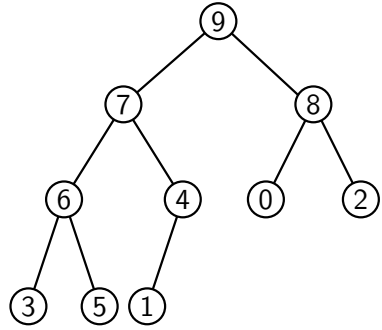
Définition des tas

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



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

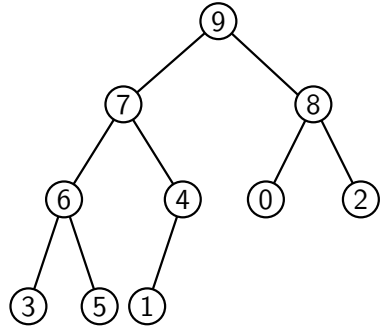
Définition

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 $i \geq 1$, $T_{[\lfloor \frac{i-1}{2} \rfloor]} \geq T_{[i]}$

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 $x \neq \text{rac}(A)$, $\text{val}(\text{père}(x)) \leq \text{val}(x)$



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

Définition

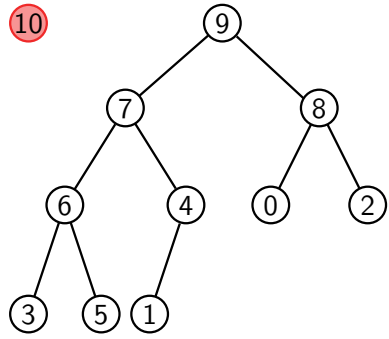
Un **tas max** (resp. **min**) est un **arbre quasi-complet** ayant la propriété de tas **max** (resp. **min**)
Un tableau T est un tas max si pour tout $i \geq 1$, $T_{[\lfloor \frac{i-1}{2} \rfloor]} \geq T_{[i]}$

Remarque

Un arbre binaire peut avoir la propriété de tas max sans être quasi-complet !

Insertion dans un tas max

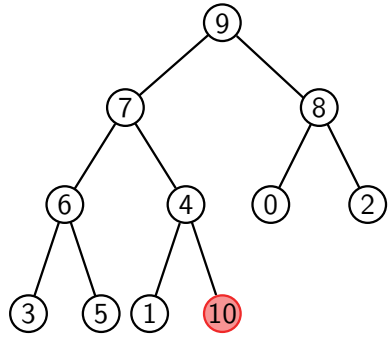
Algorithme : INSÉRER(T, x)
$i \leftarrow n(T)$
Agrandir T d'une case
$T_{[i]} \leftarrow x$
REMONTER(T, i)



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

Insertion dans un tas max

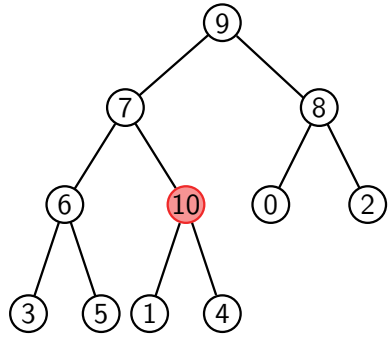
Algorithme : INSÉRER(T, x)
$i \leftarrow n(T)$
Agrandir T d'une case
$T_{[i]} \leftarrow x$
REMONTER(T, i)



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

Insertion dans un tas max

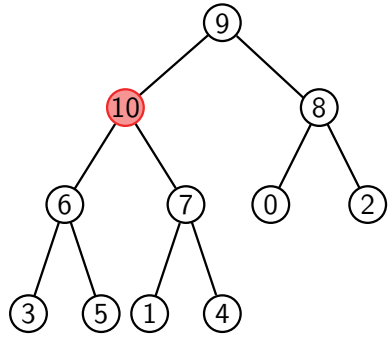
Algorithme : INSÉRER(T, x)
$i \leftarrow n(T)$
Agrandir T d'une case
$T_{[i]} \leftarrow x$
REMONTER(T, i)



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

Insertion dans un tas max

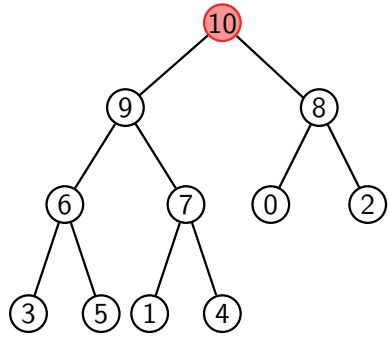
Algorithme : INSÉRER(T, x)
$i \leftarrow n(T)$
Agrandir T d'une case
$T_{[i]} \leftarrow x$
REMONTER(T, i)



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

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)

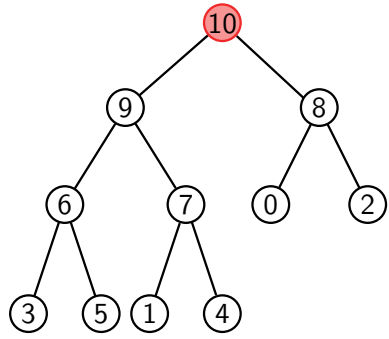


[10, 9, 8, 6, 7, 0, 2, 3, 5, 1, 4]

Insertion dans un tas max

Algorithme : INSÉRER(T, x)
$i \leftarrow n(T)$
Agrandir T d'une case
$T_{[i]} \leftarrow x$
REMONTER(T, i)

Algorithme : REMONTER(T, i)
tant que $i > 0$ *et* $T_{[\text{père}(i)]} < T_{[i]}$ **:**
    Échanger $T_{[i]}$ et $T_{[\text{père}(i)]}$
    $i \leftarrow \text{père}(i)$



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

Complexité et validité de l'insertion

Algorithme : REMONTER(T, i)

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

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

  $i \leftarrow \text{père}(i)$

Lemme

REMONTER(T, i) *a une complexité* $O(\log(n(T)))$.

Preuve $\mathcal{P}_i$: le nombre d'itérations de la boucle est $\leq h(i) = \lfloor \log(i+1) \rfloor$

- à chaque itération, i est remplacé par père(i), et $h(\text{père}(i)) = h(i) - 1$. Donc $h(i)$ diminue de 1 à chaque itération.

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

Complexité et validité de l'insertion

Algorithme : REMONTER(T, i)
tant que $i > 0$ *et* $T_{[\text{père}(i)]} < T_{[i]}$ **:**
  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* REMONTER(T, i).

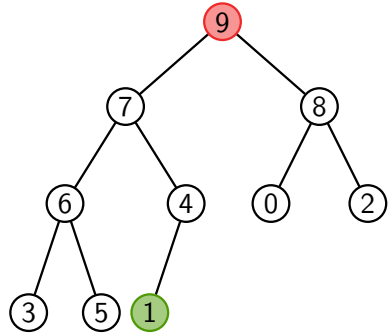
Preuve T_i : sous-arbre enraciné en i

- Invariant : T_i est un tas
  - Initialement, ok car i n'a pas de fils
  - Si l'invariant est satisfait avant une itération, 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
- À la fin : $T_{[i]} \leq T_{[\text{père}(i)]}$ et les nœuds hors de T_i n'ont pas été modifiés, donc T est un tas

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
REMONTER(T, i)
ENTASSER(T, i)
renvoyer x

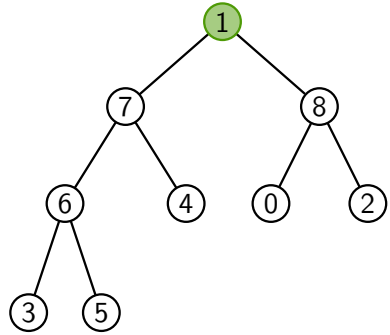


[9, 7, 8, 6, 4, 0, 2, 3, 5, 1]

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
REMONTER(T, i)
ENTASSER(T, i)
renvoyer x

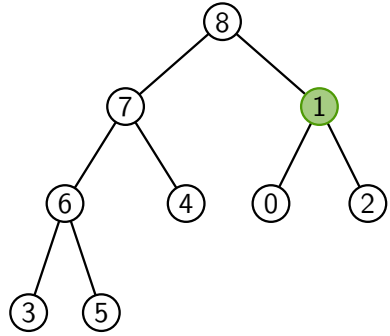


$[1, 7, 8, 6, 4, 0, 2, 3, 5]$

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
REMONTER(T, i)
ENTASSER(T, i)
renvoyer x

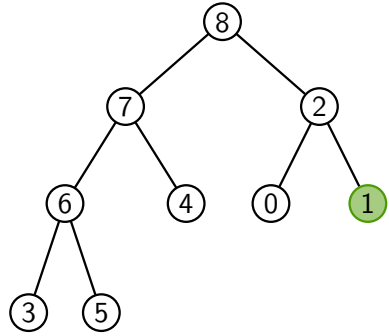


$[8, 7, 1, 6, 4, 0, 2, 3, 5]$

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
REMONTER(T, i)
ENTASSER(T, i)
renvoyer x



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

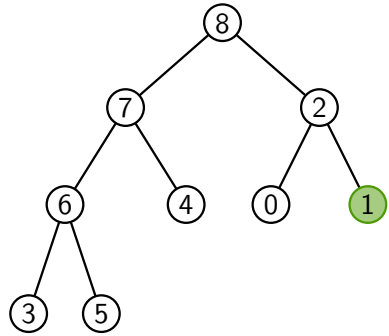
Suppression dans un tas max

Algorithme : SUPPRIMER(T, i)

$x \leftarrow T_{[i]}$
$T_{[i]} \leftarrow T_{[n(T)-1]}$
Réduire T d'une case
REMONTER(T, i)
ENTASSER(T, i)
renvoyer x

Algorithme : ENTASSER(T, i)

tant que filsG$(i) < n(T)$ **:**
  $(m, g, d) \leftarrow (i, \text{filsG}(i), \text{filsD}(i))$
  si $T_{[g]} > T_{[m]}$ **:** $m \leftarrow g$
  si $d < n(T)$ *et* $T_{[d]} > T_{[m]}$ **:** $m \leftarrow d$
  si $m \neq i$ **:** Échanger $T_{[i]}$ et $T_{[m]}$
  sinon : $i \leftarrow n(T)$



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

Complexité et validité de la suppression

Algorithme : ENTASSER(T, i)

tant que filsG$(i) < n(T)$ **:**
  $(m, g, d) \leftarrow (i, \text{filsG}(i), \text{filsD}(i))$
  si $T_{[g]} > T_{[m]}$ **:** $m \leftarrow g$
  si $d < n(T)$ *et* $T_{[d]} > T_{[m]}$ **:** $m \leftarrow d$
  si $m \neq i$ **:** Échanger $T_{[i]}$ et $T_{[m]}$
  sinon : $i \leftarrow n(T)$

Lemme

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)

tant que filsG(i) $< n(T)$:
- $(m, g, d) \leftarrow (i, \text{filsG}(i), \text{filsD}(i))$
- **si** $T_{[g]} > T_{[m]}$ **:** $m \leftarrow g$
- **si** $d < n(T)$ *et* $T_{[d]} > T_{[m]}$ **:** $m \leftarrow d$
- **si** $m \neq i$ **:** Échanger $T_{[i]}$ et $T_{[m]}$
- **sinon :** $i \leftarrow n(T)$

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)

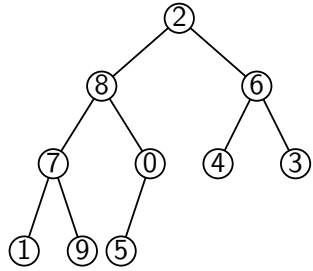
Preuve par récurrence sur $h(T) - h(i)$ (cas de base facile...)

- Si l'algo. s'arrête à la première itération : T_i est déjà un tas
- Sinon, le même algorithme est appliqué avec $i = m$
  - par hypothèse de récurrence, T_m est un tas à la fin de l'algo.
  - 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)

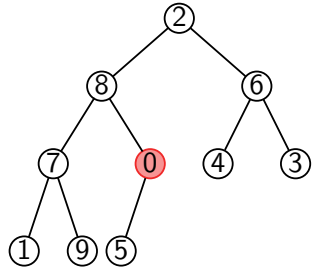
$S \leftarrow$ tableau vide de taille $n(T)$
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 **:**
  ENTASSER(T, i)
pour $i = n(T) - 1$ *à* 0 **:**
  $S_{[i]} \leftarrow$ SUPPRIMER($T, 0$)
renvoyer S



Tri par tas

Algorithme : TRITAS(T)

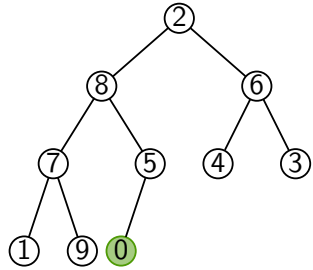
$S \leftarrow$ tableau vide de taille $n(T)$
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 **:**
  ENTASSER(T, i)
pour $i = n(T) - 1$ *à* 0 **:**
  $S_{[i]} \leftarrow$ SUPPRIMER($T, 0$)
renvoyer S



Tri par tas

Algorithme : TRITAS(T)

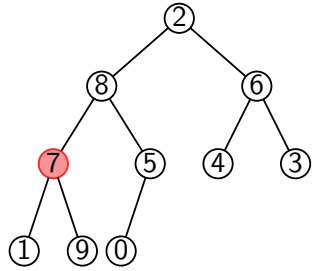
$S \leftarrow$ tableau vide de taille $n(T)$
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 **:**
  ENTASSER(T, i)
pour $i = n(T) - 1$ *à* 0 **:**
  $S_{[i]} \leftarrow$ SUPPRIMER($T, 0$)
renvoyer S



Tri par tas

Algorithme : TRITAS(T)

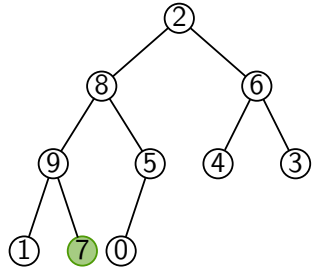
$S \leftarrow$ tableau vide de taille $n(T)$
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 **:**
  ENTASSER(T, i)
pour $i = n(T) - 1$ *à* 0 **:**
  $S_{[i]} \leftarrow$ SUPPRIMER($T, 0$)
renvoyer S



Tri par tas

Algorithme : TRITAS(T)

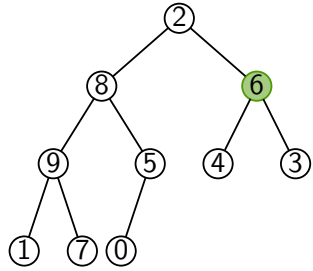
$S \leftarrow$ tableau vide de taille $n(T)$
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 **:**
  ENTASSER(T, i)
pour $i = n(T) - 1$ *à* 0 **:**
  $S_{[i]} \leftarrow$ SUPPRIMER($T, 0$)
renvoyer S



Tri par tas

Algorithme : TRITAS(T)

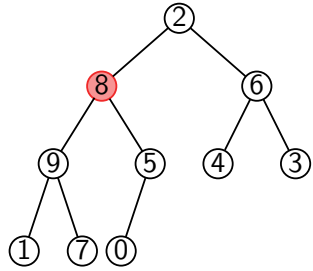
$S \leftarrow$ tableau vide de taille $n(T)$
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 **:**
  ENTASSER(T, i)
pour $i = n(T) - 1$ *à* 0 **:**
  $S_{[i]} \leftarrow$ SUPPRIMER($T, 0$)
renvoyer S



Tri par tas

Algorithme : TRITAS(T)

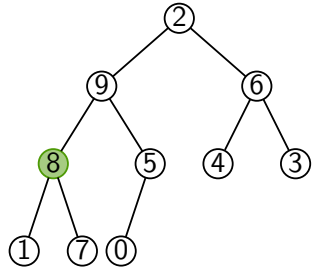
$S \leftarrow$ tableau vide de taille $n(T)$
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 **:**
  ENTASSER(T, i)
pour $i = n(T) - 1$ *à* 0 **:**
  $S_{[i]} \leftarrow$ SUPPRIMER($T, 0$)
renvoyer S



Tri par tas

Algorithme : TRITAS(T)

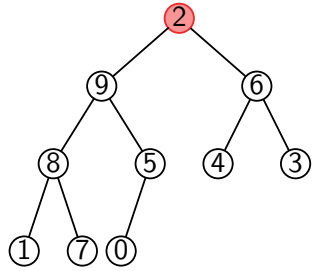
$S \leftarrow$ tableau vide de taille $n(T)$
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 **:**
  ENTASSER(T, i)
pour $i = n(T) - 1$ *à* 0 **:**
  $S_{[i]} \leftarrow$ SUPPRIMER($T, 0$)
renvoyer S



Tri par tas

Algorithme : TRITAS(T)

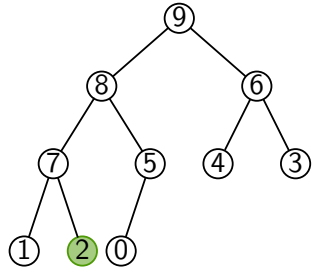
$S \leftarrow$ tableau vide de taille $n(T)$
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 **:**
  ENTASSER(T, i)
pour $i = n(T) - 1$ *à* 0 **:**
  $S_{[i]} \leftarrow$ SUPPRIMER($T, 0$)
renvoyer S



Tri par tas

Algorithme : TRITAS(T)

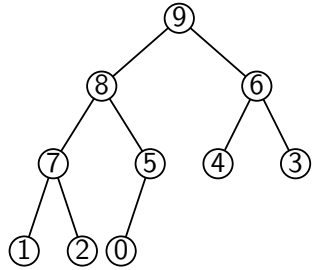
$S \leftarrow$ tableau vide de taille $n(T)$
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 **:**
  ENTASSER(T, i)
pour $i = n(T) - 1$ *à* 0 **:**
  $S_{[i]} \leftarrow$ SUPPRIMER($T, 0$)
renvoyer S



Tri par tas

Algorithme : TRITAS(T)

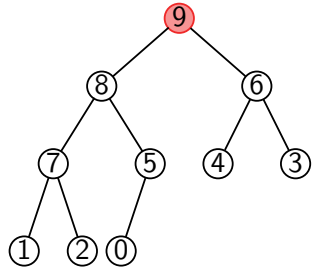
$S \leftarrow$ tableau vide de taille $n(T)$
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 **:**
  ENTASSER(T, i)
pour $i = n(T) - 1$ *à* 0 **:**
  $S_{[i]} \leftarrow$ SUPPRIMER($T, 0$)
renvoyer S



Tri par tas

Algorithme : TRITAS(T)

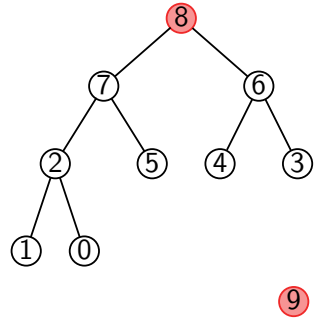
$S \leftarrow$ tableau vide de taille $n(T)$
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 **:**
  ENTASSER(T, i)
pour $i = n(T) - 1$ *à* 0 **:**
  $S_{[i]} \leftarrow$ SUPPRIMER($T, 0$)
renvoyer S



Tri par tas

Algorithme : TRITAS(T)

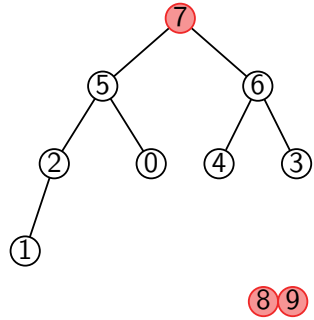
$S \leftarrow$ tableau vide de taille $n(T)$
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 **:**
  ENTASSER(T, i)
pour $i = n(T) - 1$ *à* 0 **:**
  $S_{[i]} \leftarrow$ SUPPRIMER($T, 0$)
renvoyer S



Tri par tas

Algorithme : TRITAS(T)

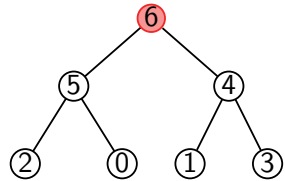
$S \leftarrow$ tableau vide de taille $n(T)$
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 **:**
  ENTASSER(T, i)
pour $i = n(T) - 1$ *à* 0 **:**
  $S_{[i]} \leftarrow$ SUPPRIMER($T, 0$)
renvoyer S



Tri par tas

Algorithme : TRITAS(T)

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

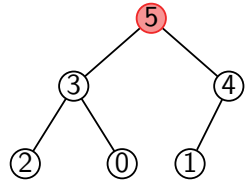


7 8 9

Tri par tas

Algorithme : TRITAS(T)

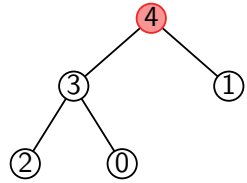
$S \leftarrow$ tableau vide de taille $n(T)$
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 **:**
  ENTASSER(T, i)
pour $i = n(T) - 1$ *à* 0 **:**
  $S_{[i]} \leftarrow$ SUPPRIMER($T, 0$)
renvoyer S



Tri par tas

Algorithme : TRITAS(T)

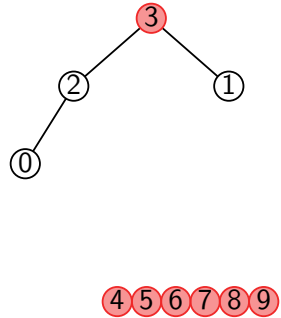
$S \leftarrow$ tableau vide de taille $n(T)$
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 **:**
  ENTASSER(T, i)
pour $i = n(T) - 1$ *à* 0 **:**
  $S_{[i]} \leftarrow$ SUPPRIMER($T, 0$)
renvoyer S



Tri par tas

Algorithme : TRITAS(T)

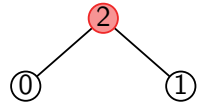
$S \leftarrow$ tableau vide de taille $n(T)$
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 **:**
  ENTASSER(T, i)
pour $i = n(T) - 1$ *à* 0 **:**
  $S_{[i]} \leftarrow$ SUPPRIMER($T, 0$)
renvoyer S



Tri par tas

Algorithme : TRITAS(T)

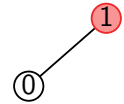
$S \leftarrow$ tableau vide de taille $n(T)$
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 **:**
  ENTASSER(T, i)
pour $i = n(T) - 1$ *à* 0 **:**
  $S_{[i]} \leftarrow$ SUPPRIMER($T, 0$)
renvoyer S



Tri par tas

Algorithme : TRITAS(T)

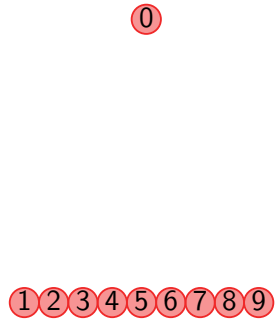
$S \leftarrow$ tableau vide de taille $n(T)$
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 **:**
  ENTASSER(T, i)
pour $i = n(T) - 1$ *à* 0 **:**
  $S_{[i]} \leftarrow$ SUPPRIMER($T, 0$)
renvoyer S



Tri par tas

Algorithme : TRITAS(T)

$S \leftarrow$ tableau vide de taille $n(T)$
pour $i = \lfloor n(T)/2 \rfloor - 1$ *à* 0 **:**
  ENTASSER(T, i)
pour $i = n(T) - 1$ *à* 0 **:**
  $S_{[i]} \leftarrow$ SUPPRIMER($T, 0$)
renvoyer 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 **:**
  ENTASSER(T, i)
pour $i = n(T) - 1$ *à* 0 **:**
  $S_{[i]} \leftarrow$ SUPPRIMER($T, 0$)
renvoyer 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 **:**
  Entasser(T, i)
pour $i = n(T) - 1$ *à* 0 **:**
  $S_{[i]} \leftarrow$ Supprimer($T, 0$)
renvoyer S

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 **:**
  Entasser(T, i)
pour $i = n(T) - 1$ *à* 0 **:**
  $S_{[i]} \leftarrow$ Supprimer($T, 0$)
renvoyer S

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 **:**
  Entasser(T, i)
pour $i = n(T) - 1$ *à* 0 **:**
  $S_{[i]} \leftarrow$ Supprimer($T, 0$)
renvoyer S

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

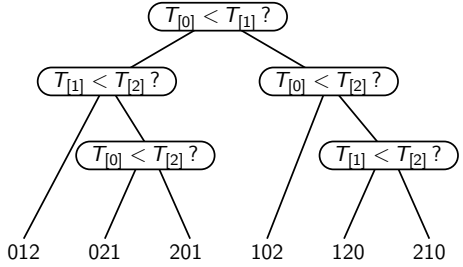
Théorème

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) $\rightsquigarrow$ INSÉRER
- EXTRAIREMAX : retire l'élément de priorité maximale $\rightsquigarrow$ SUPPRIMER
- CHANGERPRIORITÉ : modifie la priorité d'un élément $\rightsquigarrow$ REMONTER/ENTASSER

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

Files de priorité

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

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

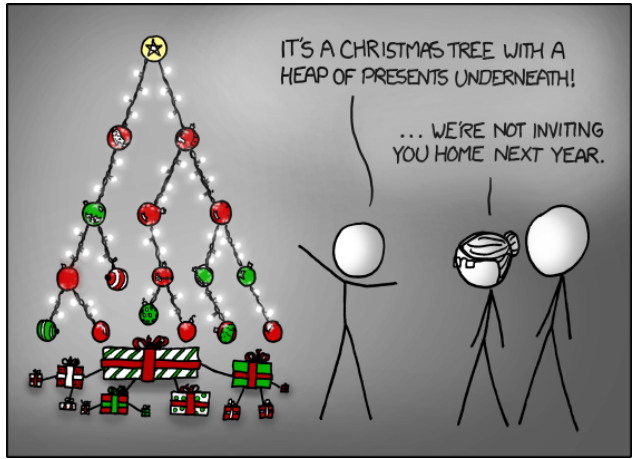
- AJOUTER : ajoute un nouvel élément (avec sa priorité) $\rightsquigarrow$ INSÉRER
- EXTRAIREMIN : retire l'élément de priorité minimale $\rightsquigarrow$ SUPPRIMER
- CHANGERPRIORITÉ : modifie la priorité d'un élément $\rightsquigarrow$ REMONTER/ENTASSER

$\rightsquigarrow$ **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



https://xkcd.com/835/

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?

Applications of binary trees

380

- 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.

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 !

Les graphes en informatique

Un objet (encore plus ?) central en informatique !

- Réseaux sociaux, graphe du web, réseaux biologiques, graphes de connaissance, réseaux de neurones, réseaux routiers, ...

Les graphes en informatique

Un objet (encore plus ?) central en informatique !

- Réseaux sociaux, graphe du web, réseaux biologiques, graphes de connaissance, réseaux de neurones, réseaux routiers, ...
- Algorithmique très riche :
  - Parcours de graphes (détection de cycles, calculs de distances/chemins le plus court, composantes connexes, ...)
  - Algorithmes de flots et coupes (optimisation de réseaux, planification de tâches, ...)
  - ...

Les graphes en informatique

Un objet (encore plus ?) central en informatique !

- Réseaux sociaux, graphe du web, réseaux biologiques, graphes de connaissance, réseaux de neurones, réseaux routiers, ...
- Algorithmique très riche :
  - Parcours de graphes (détection de cycles, calculs de distances/chemins le plus court, composantes connexes, ...)
  - Algorithmes de flots et coupes (optimisation de réseaux, planification de tâches, ...)
  - ...
- Plusieurs algorithmes de graphes dans la suite du cours