

Algorithmique — 1. Structures de données

2. Arbres binaires et tas

Bruno Grenet



<https://www-verimag.imag.fr/~grenetb/Algorithmique.html>

Université Grenoble Alpes – IM²AG
L3 Mathématiques et Informatique

Table des matières

1. Arbres binaires (rappels)

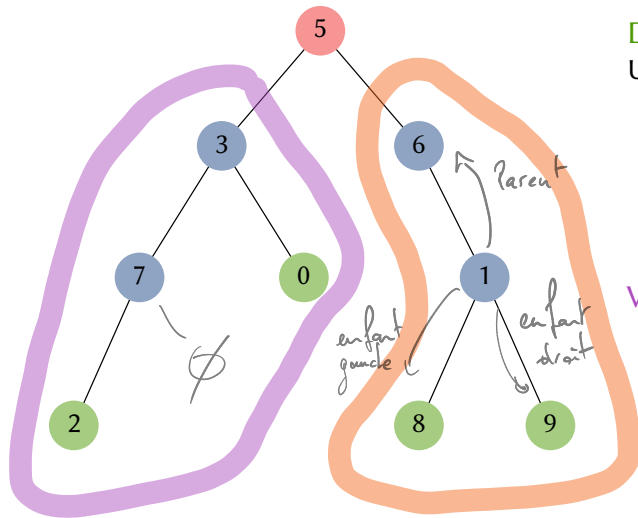
2. Tas

Table des matières

1. Arbres binaires (rappels)

2. Tas

Vocabulaire



Définition

Un **arbre binaire** est défini récursivement :

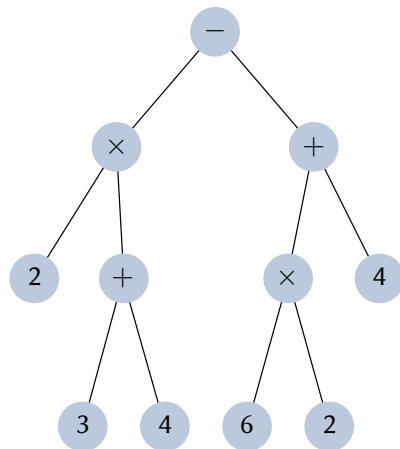
- ▶ soit l'arbre vide $\emptyset$
- ▶ soit constitué de 3 éléments :
 - ▶ la **racine** *donnée*
 - ▶ le **sous-arbre gauche** *arbre binaire*
 - ▶ le **sous-arbre droit** *arbre binaire*

Vocabulaire

- ▶ Nœud : **racine**, **nœud interne**, **feuille**
- ▶ Parent, enfants gauche et droit

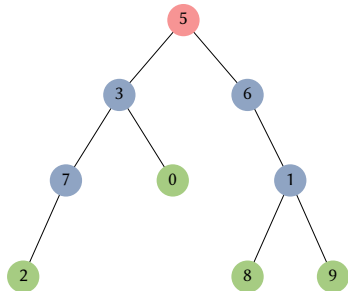
Utilité des arbres binaires

- ▶ Arbres binaires de recherche
- ▶ Tas
- ▶ Analyse syntaxique
- ▶ Bases de données
- ▶ Partition binaire de l'espace
- ▶ Tables de routage
- ▶ ...



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

Hauteur et niveaux



Définition

- ▶ **Hauteur** $h(x)$ d'un nœud x dans $\mathcal{A}$:
 - ▶ 0 si x est la racine de $\mathcal{A}$
 - ▶ $1 + h(p)$ sinon, où p est le parent de x
- ▶ **Hauteur** d'un arbre $\mathcal{A}$:
 $h(\mathcal{A}) = \max\{h(x) : x \in \mathcal{A}\}$

Lemme

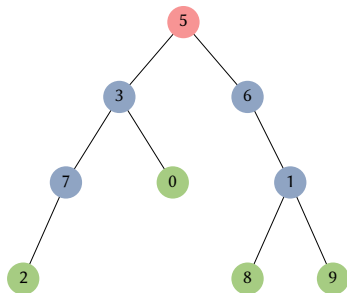
$$\#\{x : h(x) = k\} \leq 2^k$$

Preuve

- $k=0$: $\{\text{nœud à hauteur } 0\} = \{\text{racine}\} : \#\{x : h(x) = 0\} = 1 \leq 2^0$
- $k \geq 0$: On suppose $\#\{x : h(x) = k\} \leq 2^k$. Chaque nœud à hauteur k possède ≤ 2 enfants. Donc $\#\{x : h(x) = k+1\} \leq 2 \times 2^k = 2^{k+1}$

Conclusion Le principe de récurrence permet de conclure.

Hauteur et niveaux



Définition

- **Hauteur** $h(x)$ d'un nœud x dans $\mathcal{A}$:
 - 0 si x est la racine de $\mathcal{A}$
 - $1 + h(p)$ sinon, où p est le parent de x
- **Hauteur** d'un arbre $\mathcal{A}$:
 $h(\mathcal{A}) = \max\{h(x) : x \in \mathcal{A}\}$

Lemme

$$1 \leq \#\{x : h(x) = k\} \leq 2^k \quad (\text{si } k \leq h(\mathcal{A}))$$

Preuve

$$n = \sum_{k=0}^{h(\mathcal{A})} \#\{x : h(x) = k\}$$

• $n \geq \sum_{k=0}^{h(\mathcal{A})} 1 = h(\mathcal{A}) + 1$

• $n \leq \sum_{k=0}^{h(\mathcal{A})} 2^k = 2^{h(\mathcal{A})+1} - 1 < 2^{h(\mathcal{A})+1}$

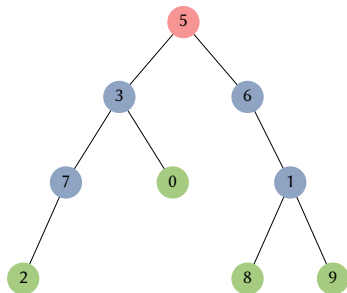
Lemme

$$1 + h(\mathcal{A}) \leq n < 2^{1+h(\mathcal{A})} \quad \text{où } n = \#\mathcal{A}$$

$S = \sum_{k=0}^n 2^k$: en Linéaire

$$\begin{array}{r} 1 \\ + 10 \\ + 100 \\ \vdots \\ + 10 \dots 0 \\ \hline 1 \dots 1 \end{array}$$

Hauteur et niveaux



Preuve

Définition

- ▶ **Hauteur** $h(x)$ d'un nœud x dans $\mathcal{A}$:
 - ▶ 0 si x est la racine de $\mathcal{A}$
 - ▶ $1 + h(p)$ sinon, où p est le parent de x
- ▶ **Hauteur** d'un arbre $\mathcal{A}$:
$$h(\mathcal{A}) = \max\{h(x) : x \in \mathcal{A}\}$$

Lemme

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

Lemme

$$1 + h(\mathcal{A}) \leq n < 2^{1+h(\mathcal{A})} \quad \text{où } n = \#\mathcal{A}$$

Corollaire

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

Un TAD « arbre binaire »

Opérations

- ▶ $\text{NVARBRE}()$: nouvel arbre vide
- ▶ $\text{NVARBRE}(r, \mathcal{G}, \mathcal{D})$: nouvel arbre de racine r , et d'enfants $\mathcal{G}$ et $\mathcal{D}$
- ▶ $\text{ESTVIDE}(\mathcal{A})$: test
- ▶ $\text{RACINE}(\mathcal{A}), \text{ENFANTG}(\mathcal{A}), \text{ENFANTD}(\mathcal{A})$: racine et enfants *uniquement si* $\neg \text{ESTVIDE}(\mathcal{A})$

dynamique

Complexités (hypothèse)

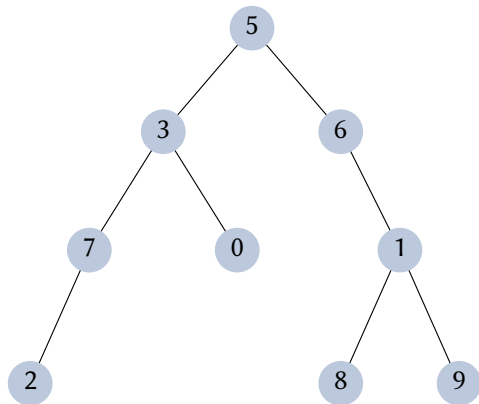
- ▶ Taille proportionnelle au nombre d'éléments
- ▶ Opérations en $O(1)$

Remarques

- ▶ Souvent : type des données à préciser
- ▶ TAD proche de l'implantation pratique
- ▶ Opération parfois ajoutée : $\text{PARENT}(\mathcal{A})$ en $O(1)$

nœuds et pointeurs

L'outil de base : le parcours en profondeur



PARCOURSINFIXE($\mathcal{A}$) :

1. Si NON(ESTVIDE($\mathcal{A}$)) :
2. PARCOURSINFIXE(ENFANTG($\mathcal{A}$))
3. *Traiter* RACINE($\mathcal{A}$)
4. PARCOURSINFIXE(ENFANTD($\mathcal{A}$))

Correction

Chaque nœud est traité une fois et une seule

Preuve par induction

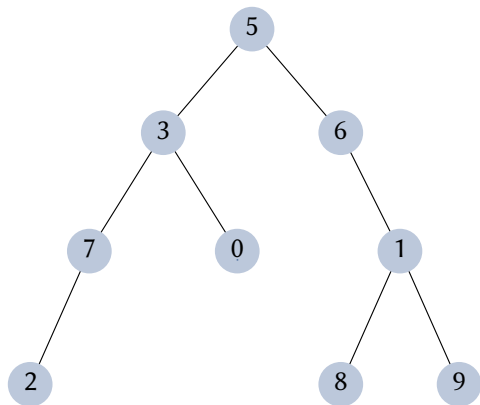
- OK pour un arbre vide

- Soit $\mathcal{A}$ un arbre et G, D ses enfants

On suppose le résultat pour G et D.

Alors chaque nœud de G et de D est traité une unique fois, et la racine aussi. Donc OK.

L'outil de base : le parcours en profondeur



PARCOURSINFIXE($\mathcal{A}$) :

1. Si NON(ESTVIDE($\mathcal{A}$)) :
2. PARCOURSINFIXE(ENFANTG($\mathcal{A}$))
3. *Traiter* RACINE($\mathcal{A}$)
4. PARCOURSINFIXE(ENFANTD($\mathcal{A}$))

Correction

Chaque nœud est traité une fois et une seule

Complexité

$O(n)$ appels à *Traiter*

$n = \#\mathcal{A}$

Ordre de traitement

2 7 3 0 5 6 8 1 9

Deux variantes

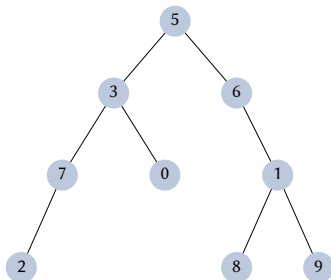
PARCOURSPRÉFIXE($\mathcal{A}$):

1. Si NON(ESTVIDE($\mathcal{A}$)):
2. *Traiter* RACINE($\mathcal{A}$)
3. PARCOURSPRÉFIXE(ENFANTG($\mathcal{A}$))
4. PARCOURSPRÉFIXE(ENFANTD($\mathcal{A}$))

PARCOURSPOSTFIXE($\mathcal{A}$):

1. Si NON(ESTVIDE($\mathcal{A}$)):
2. PARCOURSPOSTFIXE(ENFANTG($\mathcal{A}$))
3. PARCOURSPOSTFIXE(ENFANTD($\mathcal{A}$))
4. *Traiter* RACINE($\mathcal{A}$)

5 3 7 2 0 6 1 8 9

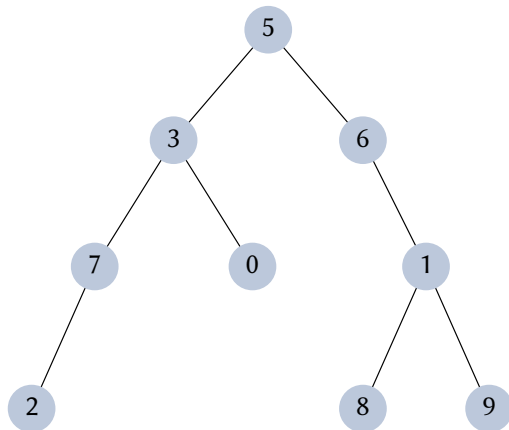


2 7 0 3 8 9 1 6 5

Exemples d'algorithmes

MINIMUM($\mathcal{A}$):

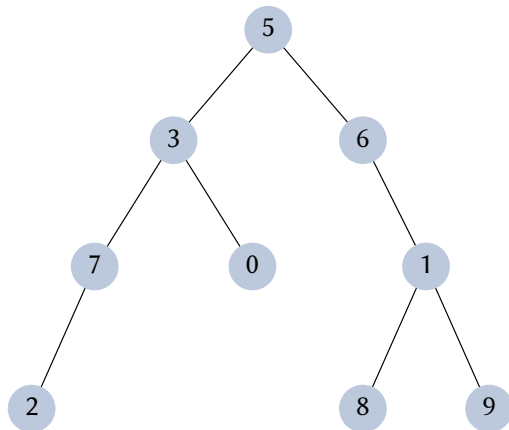
1. $m \leftarrow +\infty$
2. Si NON(ESTVIDE($\mathcal{A}$)):
3. $m_G \leftarrow \text{MINIMUM}(\text{ENFANTG}(\mathcal{A}))$
4. $m_D \leftarrow \text{MINIMUM}(\text{ENFANTD}(\mathcal{A}))$
5. $m \leftarrow \min(m_G, m_D, \text{RACINE}(\mathcal{A}))$
6. Renvoyer m



Exemples d'algorithmes

$\text{NBNEUDS}(\mathcal{A})$:

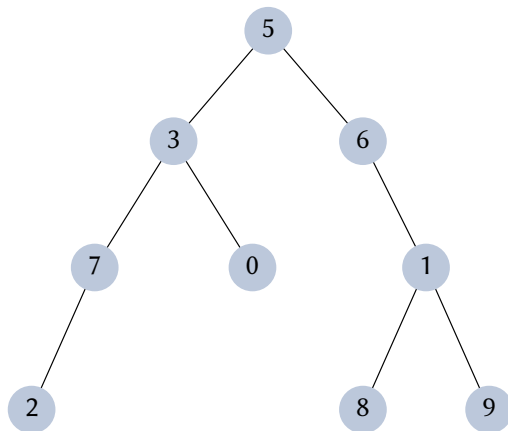
1. $n \leftarrow 0$
2. Si $\text{NON}(\text{ESTVIDE}(\mathcal{A}))$:
3. $n_G \leftarrow \text{NBNEUDS}(\text{ENFANTG}(\mathcal{A}))$
4. $n_D \leftarrow \text{NBNEUDS}(\text{ENFANTD}(\mathcal{A}))$
5. $n \leftarrow n_G + n_D + 1$
6. Renvoyer n



Exemples d'algorithmes

HAUTEUR($\mathcal{A}$):

1. $h \leftarrow -1$
2. Si NON(ESTVIDE($\mathcal{A}$)):
3. $h_G \leftarrow \text{HAUTEUR}(\text{ENFANTG}(\mathcal{A}))$
4. $h_D \leftarrow \text{HAUTEUR}(\text{ENFANTD}(\mathcal{A}))$
5. $h \leftarrow 1 + \max(h_G, h_D)$
6. Renvoyer h



Bilan sur les arbres binaires

Une structure de base en algorithmique

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

Bilan sur les arbres binaires



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.
- [Syntax Tree](#) - Constructed by compilers and (implicitly) calculators to parse expressions.
- [Treap](#) - Randomized data structure used in wireless networking and memory allocation.
- [B-tree](#) - Though most databases use some form of B-tree to store data on the drive, databases

Bilan sur les arbres binaires

Une structure de base en algorithmique

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

Arbres binaires comme TAD de base

- ▶ Utilisé dans ce cours pour construire d'autres TAD
- ▶ Proche des implantations pratiques en programmation

À réviser si vous n'êtes pas à l'aise !

- ▶ Exemples :
 - ▶ compter le nombre de feuilles
 - ▶ recherche d'un élément particulier

Table des matières

1. Arbres binaires (rappels)

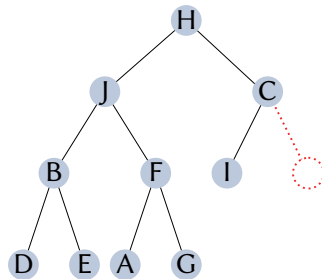
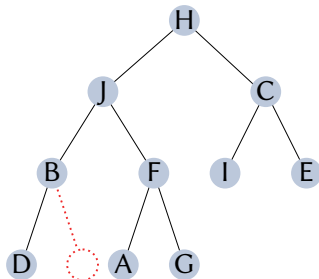
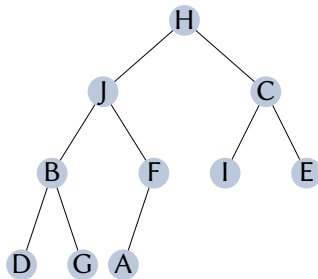
2. Tas

Arbres quasi-complets

Définition

Un arbre binaire de hauteur h est **quasi-complet** si

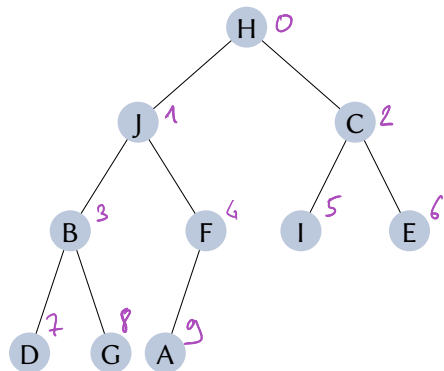
- ▶ l'arbre possède 2^k nœuds de hauteur k pour tout $k < h$
- ▶ les nœuds de hauteur h sont « *le plus à gauche possible* »



Propriété

$h = \lfloor \log n \rfloor$ où n = nombre de nœuds et h = hauteur

Parcours en largeur et numérotation des arbres quasi-complets



Définition

On attribue à chaque nœud x un **numéro** n_x :

- ▶ la racine a le numéro 0
- ▶ on numérote de haut en bas et de gauche à droite

Propriété

- ▶ $n_{\text{ENFANTG}(x)} = 2n_x + 1$
- ▶ $n_{\text{ENFANTD}(x)} = 2n_x + 2$

Remarque

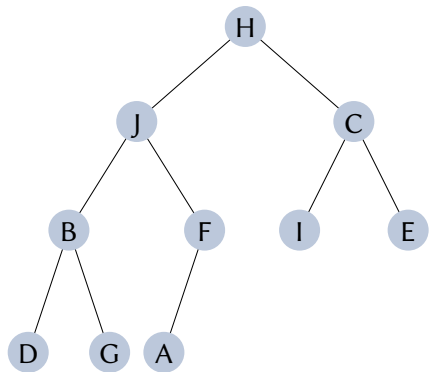
Le numéro correspond à l'ordre du **parcours en largeur**

Arbres quasi-complets et tableaux

Remarque fondamentale

On peut représenter un arbre quasi-complet dans un tableau T :

- ▶ T possède n cases (=nombre de nœuds)
- ▶ $T_{[n_x]}$ contient le nœud x



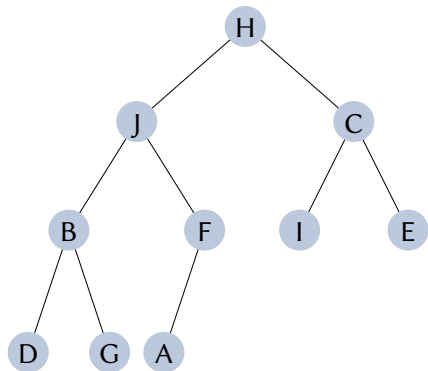
$\mathcal{A} = [H, J, C, B, F, I, E, D, G, A]$

Arbres quasi-complets et tableaux

Remarque fondamentale

On peut représenter un arbre quasi-complet dans un tableau T :

- ▶ T possède n cases (=nombres de nœuds)
- ▶ $T_{[n_x]}$ contient le nœud x



$\mathcal{A} = [H, J, C, B, F, I, E, D, G, A]$

Dans la suite :

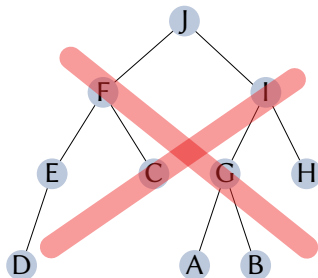
- ▶ arbre quasi-complet $\equiv$ tableau
- ▶ nœud x identifié par son indice n_x

- ▶ $RACINE(\mathcal{A}) = 0$
- ▶ $ENFANTG(i) = 2i + 1$ et $ENFANTD(i) = 2i + 2$
- ▶ $PARENT(i) = \lfloor (i - 1) / 2 \rfloor$

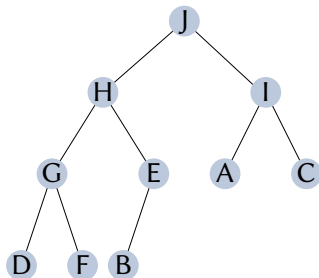
Définition d'un tas

Définition

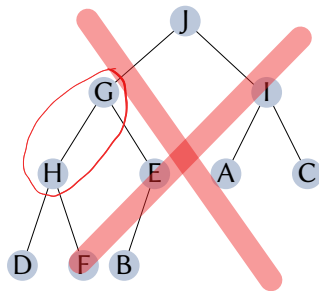
- Un arbre binaire est **décroissant** si la valeur d'un nœud est $\leq$ à celle de son parent
- Un **tas** est un arbre binaire **quasi-complet et décroissant**



décroissant
non quasi-complet



décroissant
quasi-complet



non décroissant
quasi-complet

Lemme

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

Opérations de base dans un tas

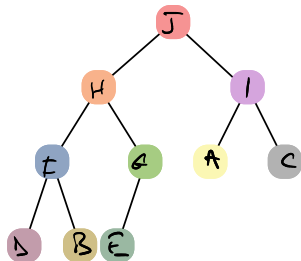
MONTER(T, i):

1. Tant que $i > 0$ et $T_{[\text{PARENT}(i)]} < T_{[i]}$:
2. Échanger $T_{[i]}$ et $T_{[\text{PARENT}(i)]}$
3. $i \leftarrow \text{PARENT}(i)$

DESCENDRE(T, i, n):

1. Tant que $2i + 1 < n$:
2. $(j, g, d) \leftarrow (i, 2i + 1, 2i + 2)$
3. Si $T_{[g]} > T_{[j]} : j \leftarrow g$
4. Si $d < n$ et $T_{[d]} > T_{[j]} : j \leftarrow d$
5. Si $j \neq i$:
6. Échanger $T_{[i]}$ et $T_{[j]}$
7. $i \leftarrow j$
8. Sinon : sortir de l'algorithme

0	1	2	3	4	5	6	7	8	9
J	G	I	F	E	A	C	D	B	H
				H					E
	H			G					F
J	H	I	F	G	A	C	D	B	E



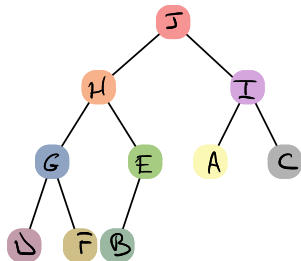
Opérations de base dans un tas

MONTER(T, i):

1. Tant que $i > 0$ et $T_{[\text{PARENT}(i)]} < T_{[i]}$:
2. Échanger $T_{[i]}$ et $T_{[\text{PARENT}(i)]}$
3. $i \leftarrow \text{PARENT}(i)$

DESCENDRE(T, i, n):

1. Tant que $2i + 1 < n$:
2. $(j, g, d) \leftarrow (i, 2i + 1, 2i + 2)$
3. Si $T_{[g]} > T_{[j]} : j \leftarrow g$
4. Si $d < n$ et $T_{[d]} > T_{[j]} : j \leftarrow d$
5. Si $j \neq i$:
6. Échanger $T_{[i]}$ et $T_{[j]}$
7. $i \leftarrow j$
8. Sinon : sortir de l'algorithme



Réalisation d'une file de priorité par un tas

Représentation et opérations de base

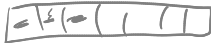
- ▶ Tableau de **taille N fixée à l'avance**, nombre n d'éléments stockés $F = (T, n)$
- ▶ Chaque case contient un couple (x, p) (élément, priorité)
- ▶ Propriété de tas pour les priorités : $(x_1, p_1) < (x_2, p_2) \iff p_1 < p_2$

NVFILEPRIORITÉ():

1. $T \leftarrow \text{NVTABLEAU}(N)$
2. renvoyer $(T, 0)$

INSÉRER(F, x, p):

1. $(T, n) \leftarrow F; N \leftarrow \text{TAILLE}(T)$
2. Si $n = N$: erreur (file pleine)
3. $T[n] \leftarrow (x, p)$
4. $n \leftarrow n + 1$
5. MONTER($T, n - 1$)



ESTVIDE(F):

1. renvoyer « $n = 0$? »

EXTRAIRE(F):

1. $(T, n) \leftarrow F; N \leftarrow \text{TAILLE}(T)$
2. $(x, p) \leftarrow T[0]$
3. $T[0] \leftarrow T[n-1]$
4. $n \leftarrow n - 1$
5. DESCENDRE($T, 0, n$)
6. Renvoyer x



Correction et complexité

Théorème

INSÉRER et EXTRAIRE ont une complexité $O(\log n)$ et T conserve la structure de tas

Complexité

- Complexité de INSÉRER = complexité de MONTER.

Dans MONTER, boucle Tant que qui divise n par 2 à chaque étape

$\Rightarrow O(\log n)$ itérations de la boucle.

- Pareil pour EXTRAIRE avec DESCENDRE.

Correction : invariants

- INSÉRER: le seul endroit possible où il n'y pas décroissance est $T[i]$ vs $T[\text{parent}(i)]$

- EXTRAIRE: _____
 $T[i]$ vs $T[\text{enfant}(i)]$
ou $T[\text{enfant}(i)]$

Bilan des réalisations basées sur des tableaux

Avantages et inconvénients

- ▶ Limite : taille maximale fixée à l'avance
- ▶ Bonnes complexités :
 - Pile : $O(1)$ pour NVPILE, ESTVIDE, EMPILER et DÉPILER
 - File : $O(1)$ pour NVFILE, ESTVIDE, ENFILER et DÉFILER

File de priorité : $O(1)$ pour NVFILEPRIORITÉ et ESTVIDE, $O(\log n)$ pour INSÉRER et EXTRAIRE

Remarques sur les tas

- ▶ Réalisation la plus classique du TAD File de priorité
- ▶ Autres réalisations plus efficaces
 - ▶ Tas de Fibonacci : INSÉRER en $O(1)$, EXTRAIRE en $O(\log n)$
 - ▶ Impossible d'avoir $O(1)$ pour les deux
- ▶ À la base du *tri par tas*
- ▶ La structure d'arbre binaire est *implicite*

pas le TAD arbre binaire

Le tri par tas

TRI-TAS :

1. Transformer le tableau en un tas
2. Transformer le tas en tableau trié

1. Tableau vers tas

- ▶ DESCENDRE les nœuds qui ont un enfant supérieur
- ▶ commencer par les nœuds les plus bas
- ▶ les feuilles peuvent être ignorées

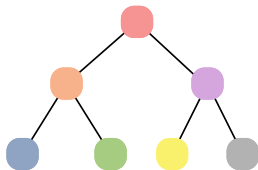
2. Tas vers tableau trié

- ▶ élément maximal en case $T_{[0]}$ (car décroissant)
- ▶ EXTRAIRE le max et le mettre en case $T_{[n-1]}$ maintenant libre

L'algorithme

TRITAS(T):

1. $n \leftarrow \#T$
2. Pour $i = \lfloor n/2 \rfloor - 1$ à 0:
3. DESCENDRE(T, i, n)
4. Pour $i = n - 1$ à 0:
5. $T[i] \leftarrow \text{EXTRAIRE}((T, i + 1))$



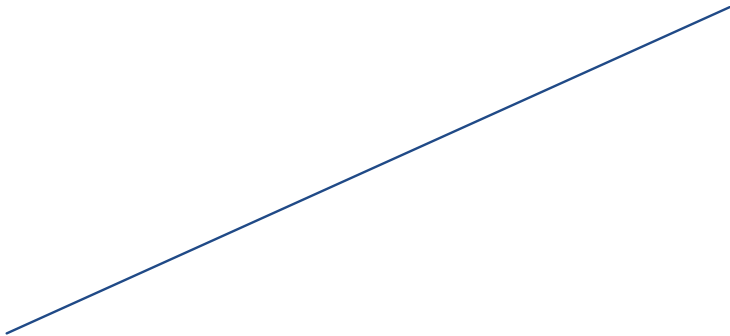
N	L	M	U	X	Z	B
		Z			π	
	X			L		
Z	X	N	U	L	π	B
B	X	N	U	L	π	Z
X	U		B			
π					X	
U	π					
L						
N		L				
B			N			
π	B	L				
L		π				
B	L	π	N	U	X	Z

Analyse du tri par tas

Théorème

TriTas trie le tableau T en faisant $O(n \log n)$ comparaisons

Preuve



Bilan sur les tas

Tas

- ▶ Une réalisation du TAD file de priorité
 - ▶ Bonne complexité $O(\log n)$
 - ▶ Structure statique $\rightarrow$ perte de place
- ▶ Structure de données *très* utilisée théorie et pratique
 - ▶ `heapq` (Python), `PriorityQueue` (Java), `priority_queue` (C++)
 - ▶ Améliorations : tas de Fibonacci, tas binomial, ...
- ▶ File de priorité enrichie
 - ▶ Modification des priorités
 - ▶ Files *fusionables*

Tri par tas

- ▶ Tri par comparaisons en $O(n \log n)$
 - ▶ théorie : optimal
 - ▶ pratique : moins rapide que le *tri rapide*
- ▶ Tri en place, mais non stable

cours 7