Algorithmique — 1. Structures de données

2. Arbres binaires et tas

Bruno Grenet

https://membres-ljk.imag.fr/Bruno.Grenet/Algorithmique.html

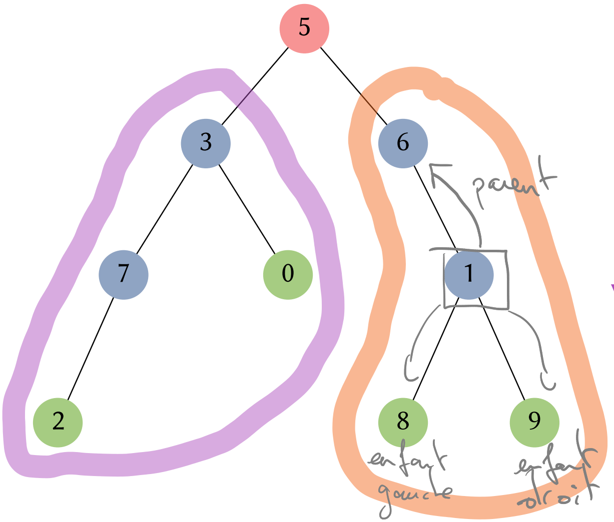
Université Grenoble Alpes – IM²AG

L3 Mathématiques et Informatique

Table des matières

Table des matières

Vocabulaire



Définition

Un arbre binaire est défini récursivement :

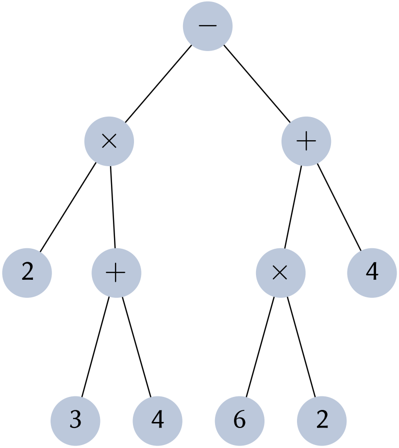
- soit l'arbre vide $\emptyset$
- soit constitué de 3 éléments :
  - la racine *donnée*
  - le sous-arbre gauche *arbre binaire*
  - le sous-arbre droit *arbre binaire*

Vocabulaire

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

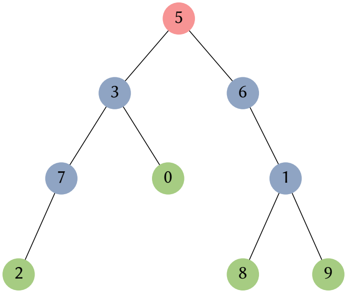
Utilité des arbres binaires

- 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

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

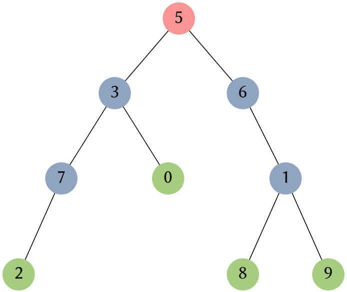
Preuve

par récurrence :

- $h = 0$: le seul nœud de hauteur 0 est la racine. Donc $\#\{x : h(x) = 0\} = 1 \leq 2^0$
- Supp que $\#\{x : h(x) = k\} \leq 2^k$. Tous les nœuds de hauteur $k+1$ ont un parent à hauteur k. Chaque nœud ayant ≤ 2 enfants, $\#\{x : h(x) = k+1\} \leq 2 \times \#\{x : h(x) = k\}$

  $\leq 2 \times 2^k = 2^{k+1}$.

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

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

Lemme

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

Preuve

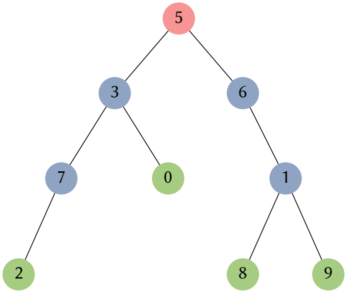
- $n_k = \#\{x : h(x) = k\}$: $1 \leq n_k \leq 2^k$ (si $k \leq h(A)$)
- $n = \sum_{k=0}^{h(A)} n_k$

$$\Rightarrow \underbrace{\sum_{k=0}^{h(A)} 1}_{= h(A) + 1} \leq n \leq \sum_{k=0}^{h(A)} 2^k = S$$

$$S = 1_2 + 10_2 + 100_2 + \dots + 10\dots0_2 = \overbrace{11\dots1_2}^{h(A)+1}$$

$$S + 1 = 1\underbrace{0\dots0}_{h(A)+1}{}_2 = 2^{h(A)+1}$$

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

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

Lemme

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

Corollaire

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

Preuve

$1 + h(A) \leq n \iff h(A) < n$

$n < 2^{1+h(A)} \underset{\log}{\Longrightarrow} \log n < 1 + h(A)$

$\Rightarrow \lfloor \log n \rfloor < 1 + h(A)$

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

Un TAD « arbre binaire »

Opérations

- nvArbre() : nouvel arbre vide — dynamique
- nvArbre$(r, \mathcal{G}, \mathcal{D})$: nouvel arbre de racine r, et d'enfants $\mathcal{G}$ et $\mathcal{D}$
- estVide$(\mathcal{A})$: test
- racine$(\mathcal{A})$, enfantG$(\mathcal{A})$, enfantD$(\mathcal{A})$: racine et enfants *uniquement si* $\neg$*estVide*$(\mathcal{A})$

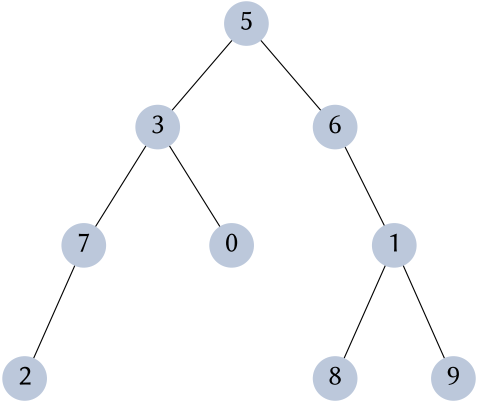
Complexités (hypothèse)

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

Remarques

- Souvent : type des données à préciser
- TAD proche de l'implantation pratique — *nœuds* et *pointeurs*
- Opération parfois ajoutée : parent$(\mathcal{A})$ en $O(1)$

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($\mathcal{A}$))

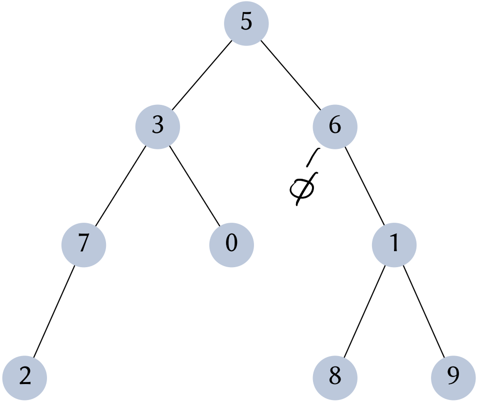
Correction

Chaque nœud est traité une fois et une seule

- Si $\mathcal{A} = \emptyset$: trivial
- Sinon, chaque nœud de ENFANTG($\mathcal{A}$) est traité une fois et une seule par hypothèse d'induction ; idem pour ENFANTD($\mathcal{A}$)

Et la racine est traitée une fois à la ligne 3

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 sommet 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

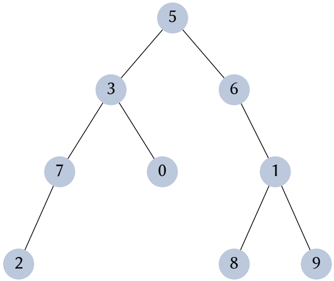
PARCOURSPRÉFIXE($\mathcal{A}$) :

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

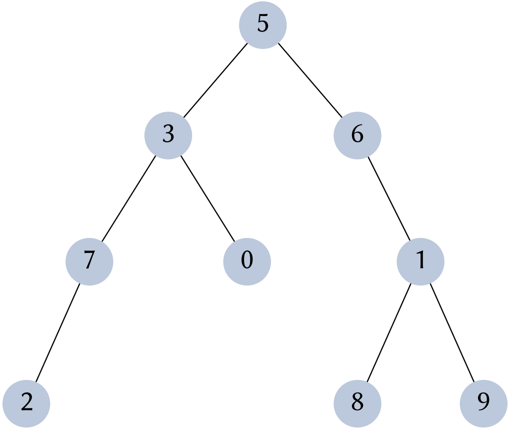
Exemples d'algorithmes

$\textsc{Minimum}(\mathcal{A})$:

1. $m \leftarrow +\infty$
2. Si $\textsc{non}(\textsc{estVide}(\mathcal{A}))$:
3. $\quad m_G \leftarrow \textsc{Minimum}(\textsc{enfantG}(\mathcal{A}))$
4. $\quad m_D \leftarrow \textsc{Minimum}(\textsc{enfantD}(\mathcal{A}))$
5. $\quad m \leftarrow \min(m_G, m_D, \textsc{racine}(\mathcal{A}))$
6. Renvoyer m

$m : \underbrace{2 \quad 0}$

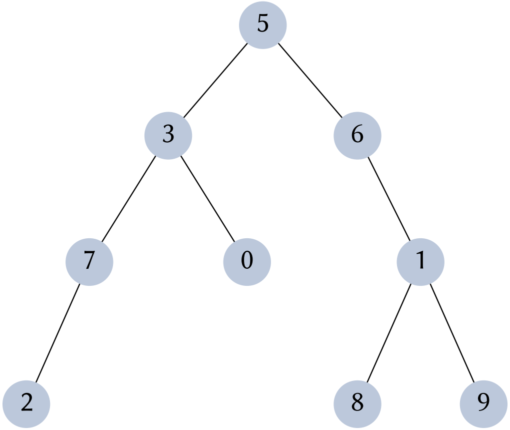
$m_G = 2 \quad m_D = 0$



Exemples d'algorithmes

NbNœuds($\mathcal{A}$) :

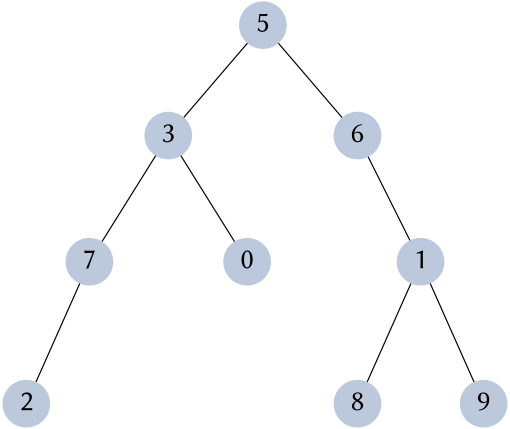
1. $n \leftarrow 0$
2. Si non(estVide($\mathcal{A}$)) :
3. $\quad n_G \leftarrow$ NbNœuds(enfantG($\mathcal{A}$))
4. $\quad n_D \leftarrow$ NbNœuds(enfantD($\mathcal{A}$))
5. $\quad n \leftarrow n_G + n_D + 1$
6. Renvoyer n



Exemples d'algorithmes

HAUTEUR($\mathcal{A}$) :

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

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

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

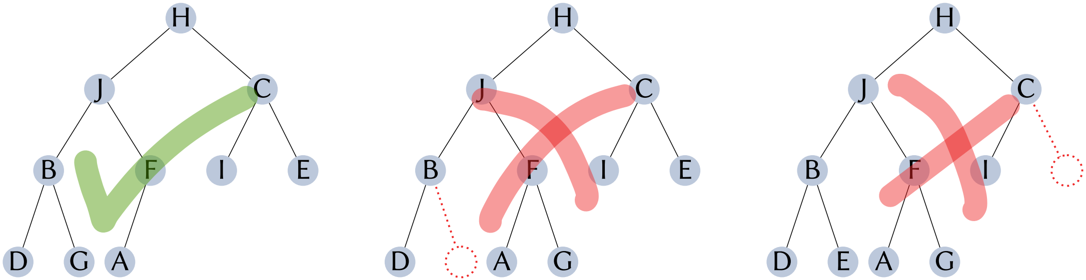
Table des matières

Arbres quasi-complets

Définition

Un arbre binaire de hauteur h est quasi-complet si

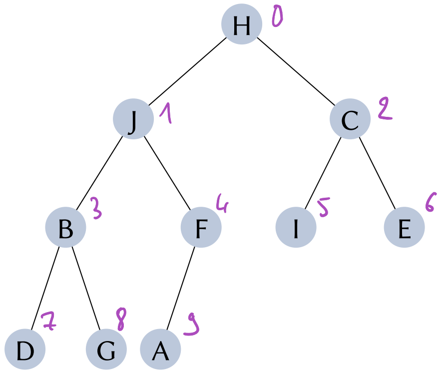
- l'arbre possède 2^k nœuds de hauteur k pour tout $k < h$
- les nœuds de hauteur h sont « *le plus à gauche possible* »



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

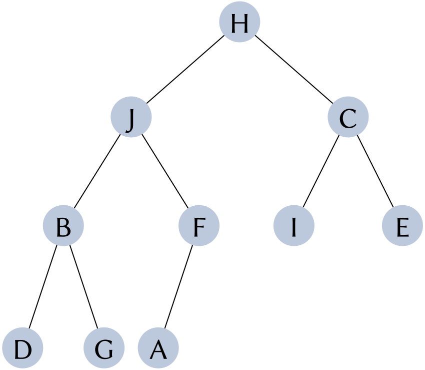
Le numéro correspond à l'ordre du parcours en largeur

Arbres quasi-complets et tableaux

Remarque fondamentale

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

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



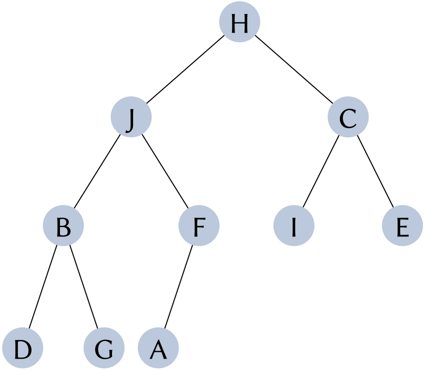
$\mathcal{A} = [H, J, C, B, F, I, E, D, G, A]$

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



$\mathcal{A} =$

Dans la suite :

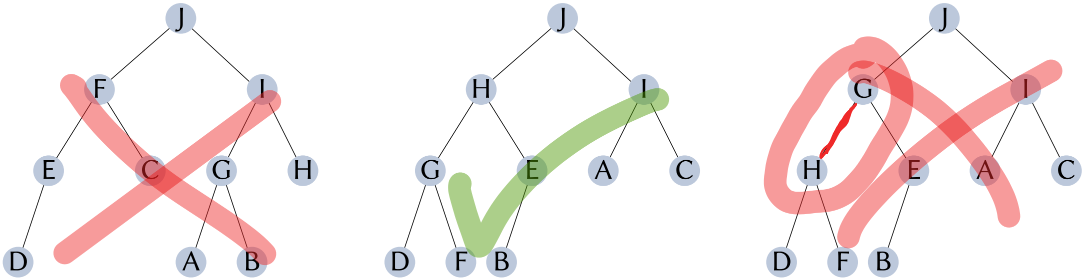
- arbre quasi-complet $\equiv$ tableau
- nœud x identifié par son indice n_x

- $\textsc{racine}(\mathcal{A}) = 0$
- $\textsc{enfant}G(i) = 2i + 1$ et $\textsc{enfant}D(i) = 2i + 2$
- $\textsc{parent}(i) = \lfloor (i-1)/2 \rfloor$

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

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

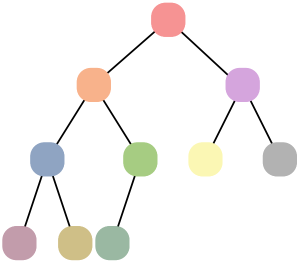
MONTER(T, i) :

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

i										
9	J	G	I	F	E	A	C	D	B	H
4					H					E
1		H			G					E
0	J	H	I	F	G	A	C	D	B	E



Opérations de base dans un tas

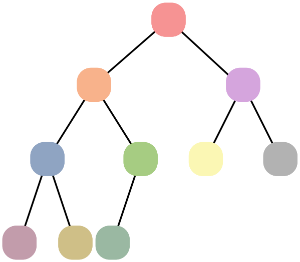
$\textsc{monter}(T, i)$:

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

$\textsc{descendre}(T, i, n)$:

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

i	j										
0	1	B	J	I	G	H	A	C	D	F	E
1	4	J	B								
4	9	J	H			B					
9	X	J	H	I	G	E	A	C	D	F	B



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 : $(x_1, p_1) < (x_2, p_2) \iff p_1 < p_2$

$\textsc{nvFilePriorité}()$:

1. $T \leftarrow \textsc{nvTableau}(N)$
2. renvoyer $(T, 0)$

$\textsc{estVide}(F)$:

1. renvoyer « $n = 0$? »

$\textsc{insérer}(F, x, p)$:

1. $(T, n) \leftarrow F$; $N \leftarrow \textsc{taille}(T)$
2. Si $n = N$: *erreur (file pleine)*
3. $T_{[n]} \leftarrow (x, p)$
4. $n \leftarrow n + 1$
5. $\textsc{monter}(T, n-1)$

$\textsc{extraire}(F)$:

1. $(T, n) \leftarrow F$; $N \leftarrow \textsc{taille}(T)$
2. $(x, p) \leftarrow T_{[0]}$
3. $T_{[0]} \leftarrow T_{[n-1]}$
4. $n \leftarrow n - 1$
5. $\textsc{descendre}(T, 0, n)$
6. Renvoyer x

Correction et complexité

Théorème

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

- Complexité : MONTER et DESCENDRE ont complexité $O(h)$ où h est la hauteur du tas. Or $h = \lfloor \log n \rfloor$ d'où le résultat.

- Correction (uniquement les invariants) :
  - INSERER : invariant = le seul endroit où T n'est pas un arbre décroissant est entre $T[i]$ et $T[\text{PARENT}(i)]$.
  - EXTRAIRE : invariant = le seul endroit où T n'est pas un arbre décroissant est entre $T[i]$ et ses enfants.

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

TriTas :

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

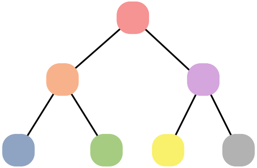
2. Tas vers tableau trié

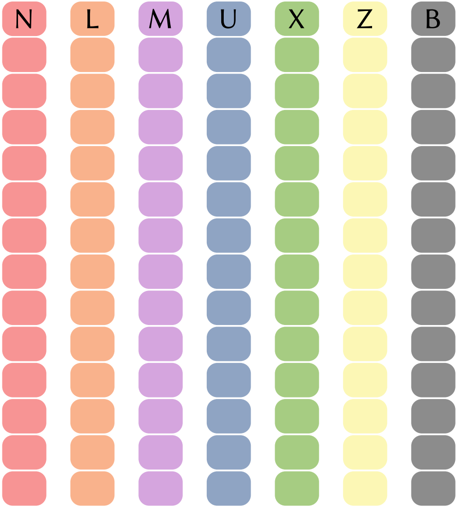
- élément maximal en case $T_{[0]}$ (car décroissant)
- extraire le max et le mettre en case $T_{[n-1]}$ maintenant libre

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. $T_{[i]} \leftarrow$ EXTRAIRE$((T, i + 1))$





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 $\to$ 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* — cours 7
- Tri en place, mais non stable