Algorithmique — 1. Structures de données

2. Structures de données linéaires

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

Les structures de données *linéaires*

Définition informelle

- Ensemble de données rangé séquentiellement
- Chaque élément a une position, les autres étant *avant* ou *après*
- Aucune autre hiérarchie entre les éléments

Exemples et contre-exemples

- Liste, tableau $\rightarrow$ structures linéaires
- Arbre binaire $\rightarrow$ structure non linéaire

Plusieurs structures linéaires

- Comment insérer / extraire des éléments
- Comment déterminer les positions

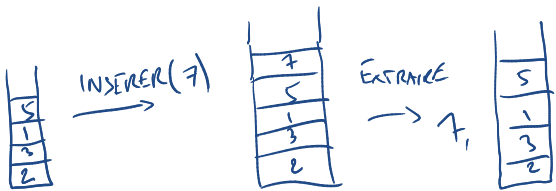
Pile : « dernier arrivé premier servi »

Description informelle

- Analogies : pile d'assiette, tas de cartes, …
- Insertion et extraction en *haut de la pile* uniquement
- Politique LIFO : *Last In First Out*

Utilité

- Pile d'appel de fonctions, opérations *annulables* dans un logiciel, …
- Exemples d'algorithmes :
  - Parcours d'arbres / graphes avec retour en arrière
  - Parenthésage / évaluation d'expressions mathématiques



Le TAD Pile

Définition

- Ensemble de données accessibles avec la politique LIFO
- Opérations :
  - nvPile() : nouvelle pile vide
  - estVide(P)
  - empiler(P, x) : ajout de x en haut de la pile P
  - dépiler(P) : renvoie l'élément en haut de P et le supprime de P

dynamique

Réalisation à base de liste

- nvPile $\rightsquigarrow$ nvListe ; estVide $\rightsquigarrow$ estVide ; empiler $\rightsquigarrow$ ajout — $O(1)$
- dépiler(P): — $O(1)$
  1. $x \leftarrow$ tête(P)
  2. $P \leftarrow$ queue(P)
  3. Renvoyer x

Remarque

- Le TAD Pile est quasiment identique au TAD Liste

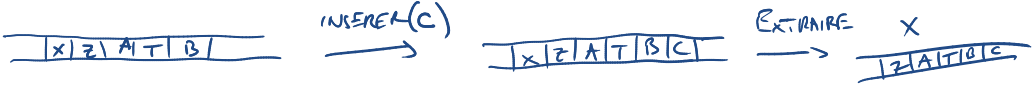
File : « premier arrivé premier servi »

Description informelle

- Analogie : file d'attente à la caisse d'un magasin
- Insertion à la fin de la file et extraction au début
- Politique FIFO : *First In First Out*

Utilité

- Files d'attente informatique *imprimante, service web, …*
- Exemple d'algorithme : parcours d'arbres / graphes en largeur



Le TAD File

Définition

- Ensemble de données accessibles avec la politique FIFO
- Opérations :
  - nvFile() : nouvelle file vide
  - estVide(F)
  - enfiler(F, x) : ajout de x en fin de file F
  - défiler(P) : renvoie l'élément au début de F et le supprime de F

dynamique

Réalisation à base de liste

- nvFile $\rightsquigarrow$ nvListe ; estVide $\rightsquigarrow$ estVide ; défiler $\rightsquigarrow$ *idem* dépiler — $O(1)$
- enfiler(F, x) : — $O(n)$
  1. Si estVide(F) : ajout(F, x)
  2. Sinon :
  3. $\quad Q \leftarrow$ queue(F)
  4. $\quad$ enfiler(Q, x)
  5. $\quad$ ajout(Q, tête(F))
  6. $\quad F \leftarrow Q$

TAD File : réalisation plus efficace ?

Remarque

- Choix effectué : tête de liste = début de file
  - DÉFILER efficace : $O(1)$
  - ENFILER inefficace : $O(n)$ *parcours de toute la liste*
- Choix symétrique : tête de liste = fin de file
  - ENFILER$(F, x) \rightsquigarrow$ AJOUT(F, x) : $O(1)$
  - DÉFILER inefficace : parcours de toute la liste en $O(n)$

Difficulté

- Nécessité d'accès aux deux extrémités de la liste

Solutions possibles

- Utiliser un TAD liste enrichi, avec accès aux deux extrémités (*liste doublement chaînée*)
- Réalisation basée sur deux piles *cf.* TD
- Réalisation basée sur les tableaux *cf.* diapo 12

File de priorité : « plus prioritaire premier servi »

Description informelle

- Analogie : embarquement dans un avion *Business class*, familles, etc.
- Chaque élément inséré a une priorité, on extrait toujours *le plus prioritaire*

Utilité

- Ordonnancement des processus dans un système d'exploitation
- Exemples d'algorithmes :
  - algorithmes de tri tri par tas
  - compression de données
  - recherche de plus court chemins

Le TAD File de priorité

Définition

- Ensemble de données accessibles avec une politique de priorité
- Opérations :
  - nvFilePriorité() : nouvelle file de priorité vide — dynamique
  - estVide(F)
  - insérer(F, x, p) : insertion de x avec priorité p dans F
  - extraire(F) : renvoie un élément de priorité maximale et le supprime de F

Réalisation à base de liste

- nvFilePriorité $\rightsquigarrow$ nvListe, estVide $\rightsquigarrow$ estVide — $O(1)$
- Liste non ordonnée :
  - insérer(F, x, p) : ajout($F, (x, p)$) — $O(1)$
  - extraire(F) : chercher l'élément de priorité maximale, et le supprimer — $O(n)$
- Liste ordonnée par priorités:
  - insérer(F, x, p) : trouver le bon emplacement où insérer — $O(n)$
  - extraire(F) : renvoyer tête(F) — $O(1)$

Réalisations à base de tableau

Principes

Objectif : Simplifier les opérations en utilisant un tableau sous-jacent

Avantage : accès en temps $O(1)$ à n'importe quel élément

Inconvénient : TAD *statique* pour réaliser des TAD *dynamiques*

Solution : Utilisation d'un tableau de taille fixée

- Perte de place $\rightarrow$ tableau plus grand que le strict nécessaire
- Perte de généralité $\rightarrow$ taille maximale fixée à l'avance

Idée générale

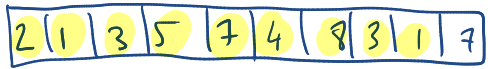
- Un tableau de taille N pour stocker n éléments $\quad n \leq N$
- Information en plus pour savoir où sont les éléments dans le tableau
  - Pile : cases 0 à $n-1$
  - File : cases contigües, pas forcément au début
  - File de priorité : à voir…

Piles basées sur des tableaux

Réalisation de Pile

- nvPile() :
  1. $T \leftarrow$ nvTableau(N)
  2. renvoyer $(T, 0)$
- estVide(P) :
  1. renvoyer « $n = 0$? »
- empiler(P, x) :
  1. si $n < N$:
  2. $\quad T_{[n]} \leftarrow x$
  3. $\quad n \leftarrow n + 1$
  4. sinon : *erreur (pile pleine)*
- dépiler(P) :
  1. $n \leftarrow n - 1$
  2. renvoyer $T_{[n]}$

$P = (T, n)$



Remarque

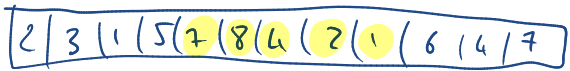
- Toutes les opérations en $O(1)$

Files basées sur des tableaux

Réalisation de File

- NVFILE() :
  1. $T \leftarrow$ NVTABLEAU(N)
  2. renvoyer $(T, 0, 0)$
- ESTVIDE(F) :
  1. renvoyer « $f - d = 0$? »
- ENFILER(F, x) :
  1. si $f < N$:
  2. $\quad T_{[f]} \leftarrow x$
  3. $\quad f \leftarrow f + 1$
  4. sinon : *erreur (file pleine)*
- DÉFILER(F) :
  1. $d \leftarrow d + 1$
  2. renvoyer $T_{[d-1]}$

$F = (T, d, f)$



Remarque

- Toutes les opérations sont en $O(1)$
- On peut utiliser le tableau de manière *circulaire*

cf. TD

Bilan sur les piles, files, files à priorité

Similarités et différences

- *Syntaxiquement* très proches — ajout / suppression
- *Sémantiquement* différentes
  - Remarque : Pile/File peuvent être vues comme de Files de priorité

Réalisations à base de liste

Pile : Quasiment une liste, opérations en $O(1)$

File : Réalisation directe inefficace → ENFILER ou DÉFILER en $O(n)$

Avec un TAD Liste enrichi → complexité $O(1)$

File de priorité : Inefficace → INSÉRER ou EXTRAIRE en $O(n)$

Réalisations à base de tableau

- Inconvénient majeur : taille fixée à l'avance
- Avantage : opérations de Pile et File en $O(1)$
- À résoudre : réalisation des files de priorité

Bilan sur les piles, files, files à priorité

Similarités et différences

- *Syntaxiquement* très proches — ajout / suppression
- *Sémantiquement* différentes
  - Remarque : Pile/File peuvent être vues comme de Files de priorité

Réalisations à base de liste

Pile : Quasiment une liste, opérations en $O(1)$

File : Réalisation directe inefficace → ENFILER ou DÉFILER en $O(n)$

Avec un TAD Liste enrichi → complexité $O(1)$

File de priorité : Inefficace → INSÉRER ou EXTRAIRE en $O(n)$

Réalisations à base de tableau

- Inconvénient majeur : taille fixée à l'avance — tableaux *dynamiques* (cours 3)
- Avantage : opérations de Pile et File en $O(1)$
- À résoudre : réalisation des files de priorité — *tas*

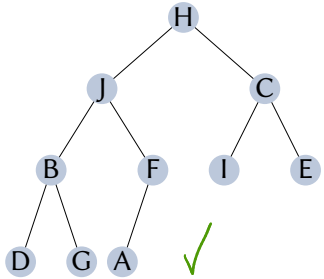
Table des matières

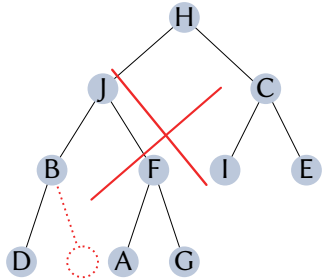
Arbres quasi-complets

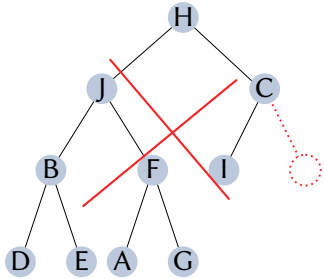
Définition

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* »



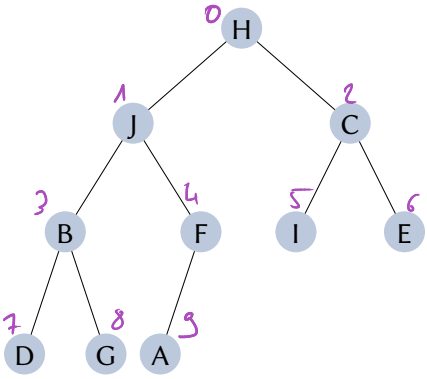




Propriété

$h = \lfloor \log n \rfloor$ 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

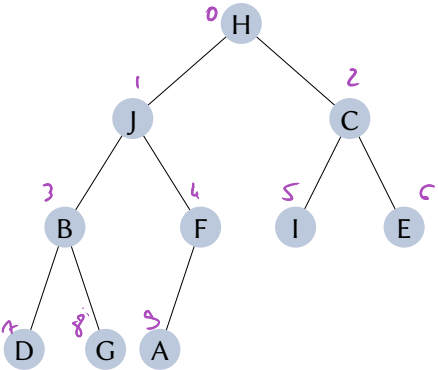
Le numéro correspond à l'ordre du parcours en largeur

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



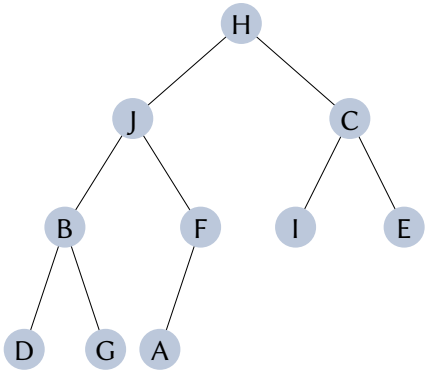
$\mathcal{A} = [H, J, C, B, F, I, E, D, G, A]$

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

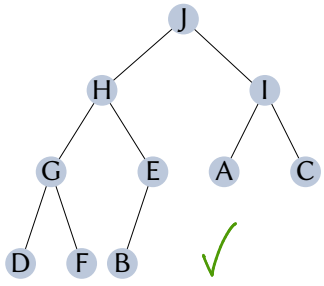
- $\textsc{racine}(\mathcal{A}) = 0$
- $\textsc{enfantG}(i) = 2i + 1$ et $\textsc{enfantD}(i) = 2i + 2$
- $\textsc{parent}(i) = \lfloor (i-1)/2 \rfloor$

Définition d'un tas

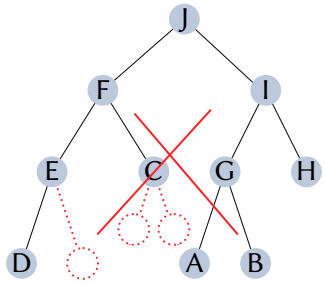
Définition

partiellement ordonné

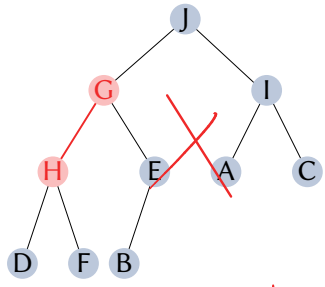
- Un arbre binaire est **tassé** si la valeur d'un nœud est $\leq$ à celle de son parent
- Un **tas** est un arbre binaire **quasi-complet et tassé**



Tas



tassé
non-quasi-complet



quasi-complet
non tassé

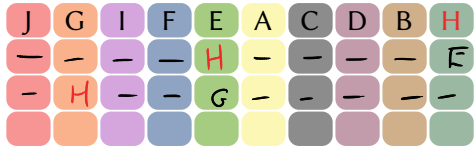
Lemme

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

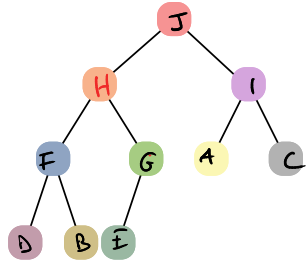
REMONTER(T, i) :

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



TASSER(T, i, n) :

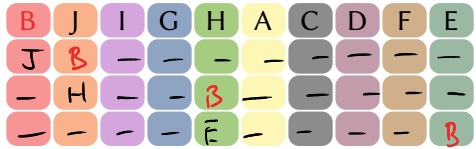
1. Tant que $2i + 1 < n$:
2. $(j, g, d) \leftarrow (i, 2i + 1, 2i + 2)$
3. Si $T_{[g]} > T_{[j]}$: $j \leftarrow g$
4. Si $d < n$ et $T_{[d]} > T_{[j]}$: $j \leftarrow d$
5. Si $j \neq i$:
6. Échanger $T_{[i]}$ et $T_{[j]}$
7. $i \leftarrow j$
8. Sinon : sortir de l'algorithme



Opérations de base dans un tas

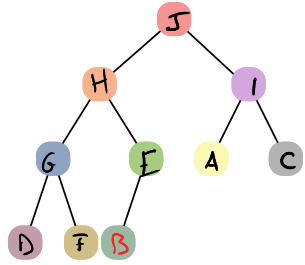
REMONTER(T, i) :

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



TASSER(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. $T \leftarrow$ nvTableau(N)
2. renvoyer $(T, 0)$

insérer(F, x, p) :

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

estVide(F) :

1. renvoyer « $n = 0$? »

extraire(F) :

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

Correction et complexité

Théorème

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

Complexité : - Un tas T a une hauteur $\lfloor \log n \rfloor$

- INSERER → REMONTER
  EXTRAIRE → TASSER
  } complexité Θ(hauteur de T)

Correction : par invariants

REMONTER : le sous-arbre dont (x,p) est racine est un tas pour les priorités

TASSER : les enfants G et D de l'élément tassé sont deux tas

Bilan des réalisations à base de tableau

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*

Conclusion

Trois structures linéaires

Pile :	Insertion et extraction en haut de pile (LIFO)	$O(1)$
File :	Insertion en fin de File, extraction en début (FIFO)	$O(1)$
File de priorité :	Insertion quelconque, extraction par priorité	$O(\log n)$

Réalisations : liste ou tableau ?

Liste : structure dynamique, adaptée pour les piles et éventuellement files

Tableau : structure statique, mais plus efficace en particulier pour les files de priorité en pratique, meilleure localité mémoire

- Meilleur des deux mondes $\rightarrow$ tableau dynamique

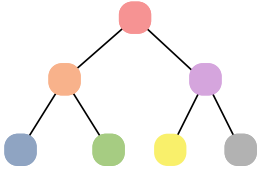
Et en pratique ?

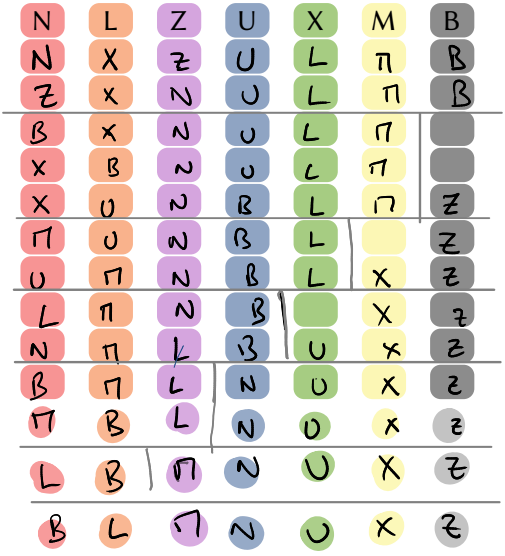
- Structures *très* utilisées à la fois en algorithmique et en programmation
- Ex. des tas : `heapq` (Python), `PriorityQueue` (Java) ou `priority_queue` (C++)

Bonus : le tri par tas

TriTas(T) :

1. $n \leftarrow \#T$
2. Pour $i = \lfloor n/2 \rfloor - 1$ à 0 :
3. tasser(T, i, n)
4. Pour $i = n - 1$ à 0 :
5. $T_{[i]} \leftarrow$ extraire($(T, i + 1)$)
6. Renvoyer T





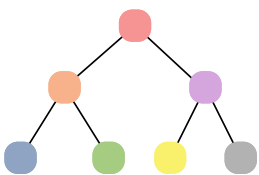
Bonus : le tri par tas

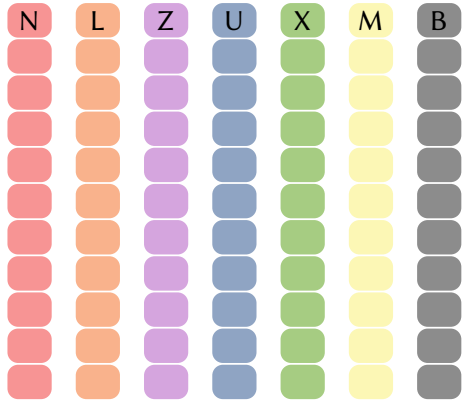
TriTas(T) :

1. $n \leftarrow \#T$
2. Pour $i = \lfloor n/2 \rfloor - 1$ à 0 :
3. tasser(T, i, n)
4. Pour $i = n - 1$ à 0 :
5. $T_{[i]} \leftarrow$ extraire($(T, i+1)$)
6. Renvoyer T

$\frac{n}{2} \times \Theta(\log n)$

$n \times \Theta(\log n)$





Théorème

Le tri par tas trie le tableau T en faisant $O(n \log n)$ comparaisons

- Après 2-3, T est un tas
- 4-5 : invariant → $T_{[0,i-1]}$ est un tas et $T_{[i,n-1]}$ est trié et $T[j] < T[k]$ si $\begin{cases} j \leq i \\ k > i \end{cases}$