Partie 1. Structures de données

2. Structures de données linéaires

Bruno Grenet

Université Grenoble Alpes – IM²AG
L3 Mathématiques et Informatique
UE Algorithmique

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

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

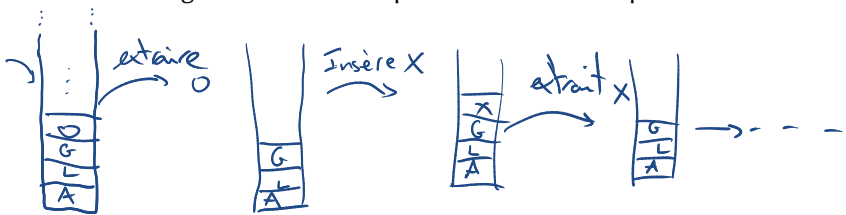
Pile : « dernier arrivé premier servi »

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 :
  - PileVide() : nouvelle pile vide
  - estVide(P)
  - Empiler(e, P) : ajout de e en haut de la pile P
  - Dépiler(P) : renvoie l'élément en haut de P et le supprime de P

dynamique

Implantation à base de liste

- PileVide $\rightsquigarrow$ NouvelleListe ; estVide $\rightsquigarrow$ estVide ; Empiler $\rightsquigarrow$ Ajout — $O(1)$
- Dépiler(P): $e \leftarrow$ Tête(P) ; $P \leftarrow$ Queue(P) ; Renvoyer e — $O(1)$

Remarque

- Le TAD Pile est quasiment identique au TAD Liste

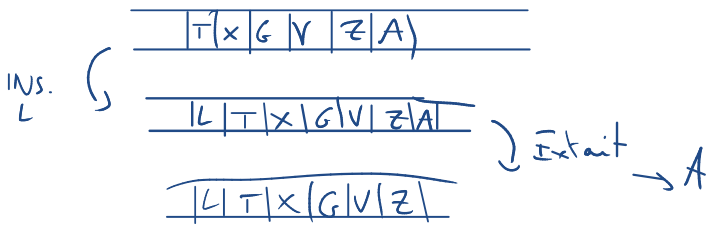
File : « premier arrivé premier servi »

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 :
  - FileVide() : nouvelle file vide — dynamique
  - estVide(F)
  - Enfiler(e, F) : ajout de e en fin de file F
  - Défiler(P) : renvoie l'élément au début de F et le supprime de F

Implantation à base de liste

- FileVide $\rightsquigarrow$ NouvelleListe ; estVide $\rightsquigarrow$ estVide ; Enfiler $\rightsquigarrow$ Ajout — $O(1)$
- Défiler : parcourir la liste pour supprimer et renvoyer son dernier élément — $O(n)$
- Solutions possibles :
  - liste enrichie avec accès aux deux extrémités — $O(1)$
  - en utilisant deux piles — *cf.* TD
- Remarque : on peut échanger les coûts de Enfiler et Défiler

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 :
  - FPVide() : nouvelle file de priorité vide — dynamique
  - estVide(F)
  - Insérer(e, p, F) : insertion de e avec priorité p dans F
  - Extraire(F) : renvoie un élément de priorité maximale et le supprime de F

Implantation à base de liste

- FPVide $\rightsquigarrow$ NouvelleListe, estVide $\rightsquigarrow$ estVide — $O(1)$
- Liste non ordonnée :
  - Insérer(e, p, F) : Ajout($(e, p), F$) — $O(1)$
  - Extraire(F) : chercher l'élément de priorité maximale, et le supprimer — $O(n)$
- Liste ordonnée par priorités:
  - Insérer(e, P, F) : trouver le bon emplacement où insérer — $O(n)$
  - Extraire(F) : renvoyer la tête de F — $O(1)$

Implantations 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 implanter 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 $n \leq N$
- 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

Implantations de Piles

PileVide() : créer un tableau T de taille N, et définir $n = 0$

estVide(P) : tester si $n = 0$

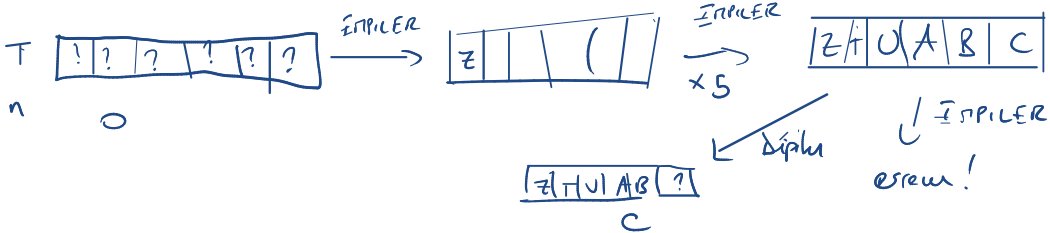
Empiler(e, P) : si $n < N$, $T_{[n]} \leftarrow e$, $n \leftarrow n + 1$; sinon *erreur (pile pleine)*

Dépiler(P) : $n \leftarrow n - 1$; renvoyer $T_{[n]}$

$P = (T, n)$

Remarque

- Toutes les opérations en $O(1)$



Files basées sur des tableaux

Implantations de Files

FileVide() : créer un tableau T de taille N et définir $(d, f) = (0, 0)$

$F = (T, d, f)$

estVide(F) : tester si $f - d = 0$

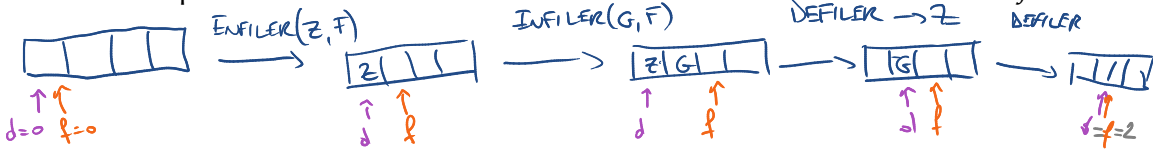
Enfiler(e, F) : Si $f < N$, $T_{[f]} \leftarrow e, f \leftarrow f + 1$; sinon *erreur (file pleine)*

Défiler(F) : $d \leftarrow d + 1$; renvoyer $T_{[d-1]}$

Remarques

- Toutes les opérations sont en $O(1)$!
- File : 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

Implantations à base de liste

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

File : Implantation directe inefficace : Enfiler ou Défiler en $O(n)$

Avec un tad Liste enrichi : complexité $O(1)$

File de priorité : Inefficace : Insérer ou Extraire en $O(n)$

Implantations à base de tableau

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

Implantations à base de liste

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

File : Implantation directe inefficace : Enfiler ou Défiler en $O(n)$

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

File de priorité : Inefficace : Insérer ou Extraire en $O(n)$

Implantations à base de tableau

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

Table des matières

Arbres quasi-complets

Définition

Un arbre binaire de hauteur h est quasi-complet si

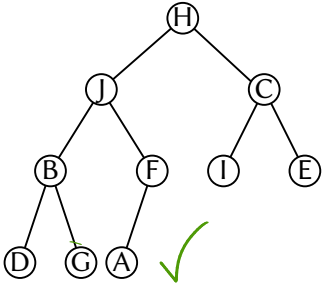
- l'arbre possède 2^k nœuds de hauteur k pour tout $k < h$
- les nœuds de hauteur h sont « *le plus à gauche possible* »

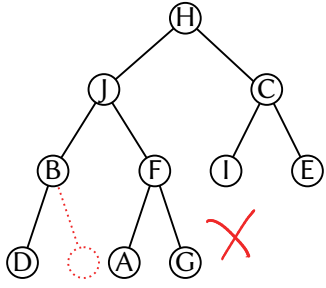
$n = 1 + 2 + 4 + \cdots + 2^{h-1} + k$

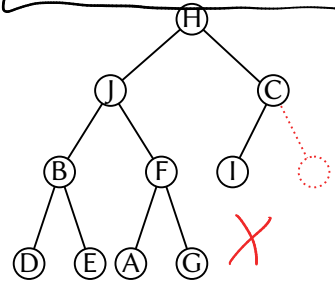
avec $1 \leq k \leq 2^h$

$2^h \leq n \leq 2^{h+1} - 1$

$h \leq \log n < h+1$



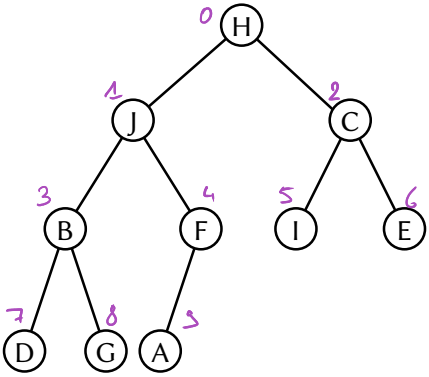




Propriété

$h = \lfloor \log n \rfloor$ 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 $\textsc{num}(x)$:

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

Propriété

- $\textsc{num}(\textsc{EnfantG}(x)) = 2\textsc{num}(x) + 1$
- $\textsc{num}(\textsc{EnfantD}(x)) = 2\textsc{num}(x) + 2$

Remarque

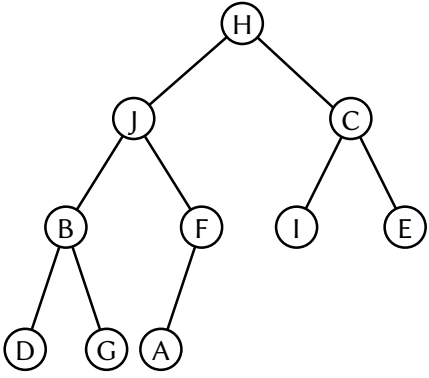
Le numéro correspond à l'ordre du parcours en largeur

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)
- $T_{[\text{NUM}(x)]}$ contient le nœud x



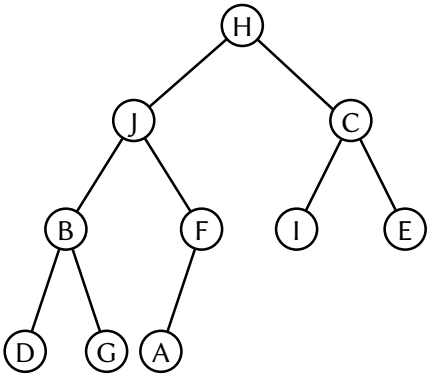
$\mathcal{A} = [H, J, C, B, F, I, E, D, G, A]$

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)
- $T_{[\text{NUM}(x)]}$ contient le nœud x



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

Dans la suite :

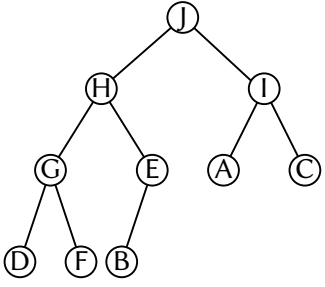
- arbre quasi-complet = tableau
- nœud x identifié par son indice $\text{NUM}(x)$

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

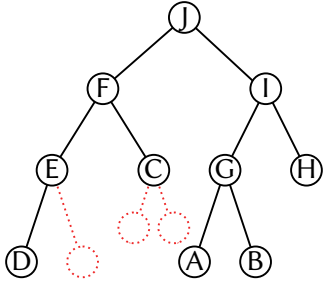
Définition d'un tas

Définition

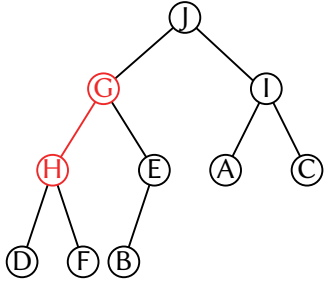
- Un arbre binaire est tassé si la valeur d'un nœud est $\leq$ à celle de son parent
- Un tas est un arbre binaire quasi-complet tassé
  - Un tableau T est un tas si pour tout $i \geq 1$, $T_{[i]} \leq T_{[\lfloor \frac{i-1}{2} \rfloor]}$



Tas [J,H,I,G,E,A,C,D,F,B]



arbre binaire tassé non quasi-complet



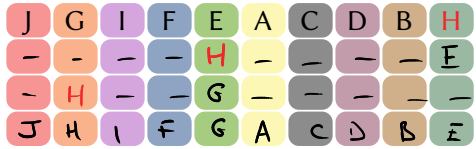
arbre quasi-complet mais non tassé

[J,G,I,H,E,A,C,D,F,B]

Opérations de base dans un tas

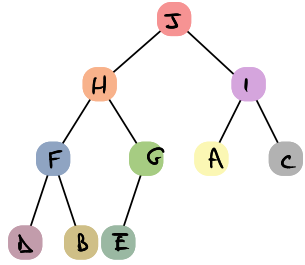
Remonter(T, i) :

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) :

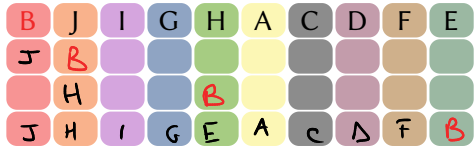
1. Tant que $2i + 1 < n$:
2. $(j, g, d) \leftarrow (i, 2i + 1, 2i + 2)$
3. Si $T_{[g]} > T_{[j]} : j \leftarrow g$
4. Si $d < n$ et $T_{[d]} > T_{[j]} : j \leftarrow d$
5. Si $j \neq i$:
6. Échanger $T_{[i]}$ et $T_{[j]}$
7. $i \leftarrow j$
8. Sinon : sortir de l'algorithme



Opérations de base dans un tas

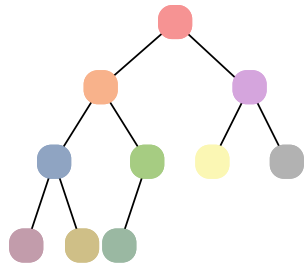
Remonter(T, i) :

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



Implantation d'une file de priorités par un tas

Représentation et opérations de base

- Tableau de taille N fixée à l'avance, nombre n d'éléments stockés
- Chaque case contient un couple (e, p) *(élément, priorité)*
- Propriété de tas pour les priorités $(e_1, p_1) < (e_2, p_2) \iff p_1 < p_2$

FPVide() : créer un tableau T de taille N, et définir $n = 0$ $F = (T, n)$

estVide(F) : tester si $n = 0$

Insérer(e, p, F) :

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

Extraire(F) :

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

Correction et complexité

Théorème

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

Non traité en cours, idée de la preuve :

Insérer est basé sur Remonter et Tasser sur Extraire. On démontre les résultats pour ces algorithmes.

Complexité :

Dans les deux algorithmes, le nombre d'itérations est borné par la hauteur du tas, donc log(n).

Correction :

Dans les deux cas, on montre que seul le nœud en case i pose problème (soit avec son parent pour Remonter, soit avec ses enfants pour Tasser). Quand on s'arrête, soit c'est que le problème n'existe plus, soit qu'on a atteint la racine (Remonter) ou une feuille (Tasser) et que le problème ne peut plus exister.

Bilan des implantations à base de tableau

Avantages et inconvénients

- Limite : taille maximale fixée à l'avance
- Bonnes complexités :
  - Pile : $O(1)$ pour PileVide, estVide, Empiler et Dépiler
  - File : $O(1)$ pour FileVide, estVide, Enfiler et Défiler

File de priorité : $O(1)$ pour FileVide et estVide, $O(\log n)$ pour Insérer et Extraire

Remarques sur les tas

- Implantation la plus classique du TAD File de priorité
- Autres implantations 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	$O(\log n)$

Implantations : 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

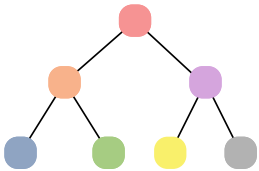
Et en pratique ?

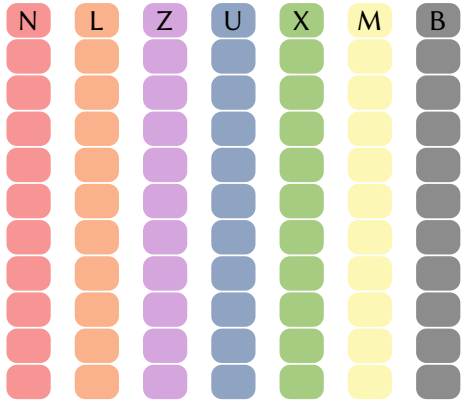
- Structures *très* utilisées à la fois en algorithmique et en programmation
- Ex. des tas : `heapq` (Python), `PriorityQueue` (Java) ou `priority_queue` (C++)

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

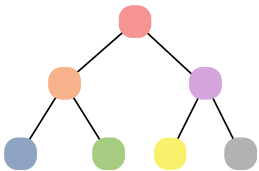


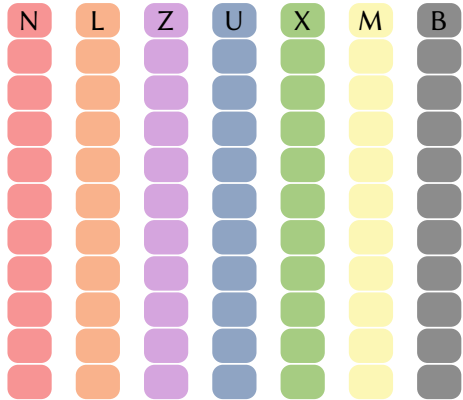


Bonus : le tri par tas

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





Théorème

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