Leçon 901 : Structure de données. Exemples et applications.

Julie Parreaux

2018 - 2019

Références pour la leçon

[1] Beauquier, Berstel et Chretienne, *Éléments d'algorithmique.*
[2] Cormen, *Algorithmique.*
[3] Froidevaux, Gaudel et Soria, *Types de données et algorithmes.*

Développements de la leçon

Les B-arbres

La structure de donnée d'Union-Find

Plan de la leçon

Motivation

Défense

Les structures de données jouent un rôle important dès que on souhaite stoker et manipuler (efficacement) des ensembles de données (qui sont généralement dynamique). La notion de type abstrait apparaît alors pour s'abstraire du langage de programmation. Elle intervient dans la conception des algorithmes qui est longue. On fait des raffinements successifs : la première version se fait indépendamment des structures de données ; la dernière version implémente dans un langage de programmation cette structure de données.

L'utilisation de cette structure abstraite se fait de deux manières différentes : la méthode ascendante qui part du langage et donne une structure abstraire ou la méthode descendante qui part de la structure abstraite et qu'on adapte au langage.

Le choix d'une structure de données est importante. Elle doit s'adapter au besoins et à la "sémantique" de l'algorithme. De plus, en fonction du problème (donc de l'algorithme), certaines structures de données sont plus efficaces que d'autre. On peut alors optimiser la complexité de nos algorithmes.

Remarque importante : Cette leçon n'est pas évidente : on peut très vite tombé dans un catalogue de structure de données sans aucune logique et le rapport du jury n'est pas nécessairement d'une grande aide. Mettre en avant les choix d'implémentations tout au long de la leçon et ne pas oublier de présenter des algorithmes utilisant ces structures. Cette leçon est fourre tout : toutes les notions d'algorithmique au programme (ou presque) peuvent se caser ici. Il y a alors un grand nombre de développements que l'on peut casé ici (on ne les a pas indiqué sur le plan...), il est donc important de soigné son plan.
Cette leçon se prête tout particulièrement aux dessins. Des dessins illustrant les structures de données classiques et leurs opérations pas nécessairement triviale peut s'avérer être intéressant (pensé à prendre le temps de les faire).

Ce qu'en dit le jury

Le mot algorithme ne figure pas dans l'intitulé de cette leçon, même si l'utilisation des structures de données est évidemment fortement liée à des questions algorithmiques. La leçon doit donc être orientée plutôt sur la question du choix d'une structure de données. Le jury attend du candidat qu'il présente différents types abstraits de structures de données en donnant quelques exemples de leur usage avant de s'intéresser au choix de la structure concrète. Le candidat ne peut se limiter à des structures linéaires simples comme des tableaux ou des listes, mais doit présenter également quelques structures plus complexes, reposant par exemple sur des implantations à l'aide d'arbres. Les notions de complexité des opérations usuelles sur la structure de données sont bien sûr essentielles dans cette leçon.

Introduction

Objectif : Comment choisir une bonne structure de données ? (phrase d'accroche)
Motivation : Stoker et manipuler (efficacement) un ensemble de données (dynamiques).

Définition. Un type de données abstrait [1, p.37] est la description d'un ensemble de données et les opérations que l'on peut y appliquer.

Définition. Un type de données concret [1, p.37] est la description d'un format de représentations internes des données en machine.

Définition. Une structure de données pour un type abstrait [1, p.37] est la donnée d'un type concret et des implémentations des différentes fonctions associées.

Tout au long de la leçon nous allons faire un va et vient entre la notion de type de donné abstrait et structure de données. Nous étudions notamment quelques types concrets pour les types abstraits que l'on présente.

1 Un type abstrait général (le type abstrait de base)

Nous commençons par donner un type abstrait de base correspondant à la structure "optimale" que nous voulons construire. Elle contient toutes les opérations de base que nous souhaitons réaliser sur une structure de données afin de manipuler ces données. Nous allons ensuite données trois structures de données pour ce type abstrait.

1.1 Les opérations souhaitées

Une liste d'opérations : create, union, find, insert, remove, find-earliest, find-min.

Remarque : on présente dans cette section des structures de données implémentant ce type abstrait. Dans la suite de la leçon nous allons également nous intéresser des structures de données qui implémentent qu'une partie de ces fonctions pour répondre à des besoins précis.

1.2 La table dynamique : une structure de données contiguë

Type concret	Espace mémoire contiguë de taille fixe Entier conservant le nombre de données	
Implémentation	insert	Si le tableau est plein, création un nouveau qui vaut deux fois la taille du premier ; sinon ajout de l'élément dans la première case vide
	delete	Mettre le dernier élément sur l'élément supprimé
	find-earliest	Impossible
	union	Création d'un tableau de la taille des deux contenant les mêmes valeurs
	Le reste des opérations se fait en parcourant le tableau séquentiellement	

1.3 La liste (l'apparition des pointeurs)

Type concret	une liste chaînée (à l'aide de pointeurs)	
Implémentation	insert	Insertion le nouvel élément en tête
	delete	Manipulation de pointeurs (un cran avant car le prédécesseur va pointer sur le successeur)
	find-earliest	Premier élément de la liste
	union	le dernier élément d'une liste va pointer sur le 1er élément de la 2ième
	Le reste des opérations se fait en parcourant séquentiellement les pointeurs	

Proposition (Comparaison des complexité selon le type concret choisi)**.** *On note n le nombre d'éléments dans la structure et m le nombre d'éléments dans la deuxième structure si besoin.*

Opération	*Tableau*		*Liste*	
find	$O(n)$	*(Au pire : parcourir tout le tableau)*	$O(n)$	*(Au pire : parcourir tout la liste)*
create	$O(n)$	*(Réserver n places mémoires)*	$O(1)$	*(Réserver une place mémoire)*
union	$O(n+m)$	*(Construire un nouveau tableau de taille $n+m$ et recopier les deux précédents dans celui-ci)*	$O(1)$	*(Connexion de la première à la dernière (un pointeur sur le dernier élément))*
insert	$O(n)$ $O(1)$	*(au pire : créer tableau de taille $2n$)* *(complexité amortie)*	$O(1)$	*(au pire : rajout d'un pointeur)*
delete	$O(n)$	*(find et recopie des suivants)*	$O(n)$	*(find et pointeurs)*
find-min	$O(n)$	*(parcourir le tableau)*	$O(n)$	*(parcourir la liste)*
find-earliest	*impossible*		$O(1)$	*(le premier élément de la liste)*

Dans la suite de la leçon nous réduisons le nombre d'opérations dans les structures de données afin de les spécialiser.

2 Les structures de données d'ordonnancement

Les structures de données d'ordonnancement nous donne des structures qui nous permettent de ranger les données selon un certains ordre : de les trier.

2.1 La pile et la file : des structures séquentielles

Type abstrait	**Pile** **File**	insert, create, find-earliest, delete-ealiest insert, create, find-oldest, delete-oldest
Application	parcours en profondeurs de graphe (algorithme 1) [1, p.102] complexité : $O(m+n)$ (m le nombre de sommets et n le nombre d'arcs de G)	
Type concret	une liste doublement chaînée	
Implémentation	insert delete find	on insère le nouvel élément en tête supprimer le premier ou le dernier élément de la liste donner le premier ou le dernier élément de la liste doublement chaînée

Algorithm 1 Parcours en profondeur d'un graphe G à l'aide d'une pile

```
1:  function PARCOURS-PROFONDEUR(G, s)
2:      create S                                   ▷ S est une file
3:      while S ≠ ∅ do
4:          (p, v) = find-oldest S
5:          delete-oldest S
6:          if v est non marqué then
7:              Marquer v
8:              parent[v] ← p
9:              for (w, v) ∈ G do
10:                 insert (v, w)
11:             end for
12:         end if
13:     end while
14: end function
```

Proposition. *L'ensemble des opérations définies sur la pile ou la file s'effectue en $O(1)$ à l'aide de la représentation par liste doublement chaînée.*

2.2 La file de priorité : une généralisation

Type abstrait	**Pile**	insert, create, find-max, delete-max
Applications	algorithme de Dijkstra (algorithme 2) [2, p.600] complexité dépend du type concret choisi tri par tas (pour tout valeurs on trouve le max et on l'enlève)	
Type concret	tas binaire vu comme un tableau [2, p.140]	
Implémentation [2, p.140]	insert	algorithme 3
	delete	algorithme 4
	find-max	prendre la racine
	create	insertion une à une des données dans un tas vide

Remarque. On peut faire une file de priorité analogue en min. De plus, l'ordre sur les clé est général puisqu'il détermine le min ou la max de cette structure de données.

Algorithm 2 Algorithme de Dijkstra

```
1:  function DIJKSTRA(G, w, s)
2:      unique-initialisation(G, s)        ▷ Initiale tous les poids de tous les sommets vers s
3:      E = ∅                              ▷ E est un ensemble
4:      F = G.S                            ▷ F est une file de priorité
5:      while F ≠ ∅ do
6:          u = find-min F
7:          E ← E ∪ {u}
8:          for (u, v) ∈ G.E do
9:              relâcher (u, v, w)         ▷ amélioration de l'estimation du sommet vers s
10:         end for
11:     end while
12: end function
```

Algorithm 3 Insertion dans une file de priorité

```
1: Insertion la plus à gauche possible de x
2: while x n'est pas racine et x > pere(x) do
3:     échange x et pere(x)
4: end while
```

Algorithm 4 Suppression dans une file de priorité

```
1: On retire la racine
2: On échange avec la plus base des valeurs x
3: while x a des fils et x < max fils(x) do
4:     échange x et max fils(x)
5: end while
```

Définition. Un tas binaire est un arbre dont les descendant d'un nœud sont tous supérieurs ou égaux à ce nœud.

Proposition (Comparaison des complexités en fonctions des types concrets)**.** *On note n est le nombre de données.*

type concret	*create*	*insert*	*find-max*	*delete-max*
liste	$O(n)$	$O(1)$	$O(n)$	$O(1)$
liste triée	$O(n \log n)$	$O(n)$	$O(1)$	$O(1)$
tas binaire	$O(n \log n)$	$O(\log n)$	$O(1)$	$O(\log n)$
tas binomiale	$O(n \log n)$	$O(\log n)$	$O(1)$	$O(\log n)$
tas de Fibonacci	$O(n)$	$O(1)$	$O(1)$	$O(\log n)$ *amortie*

2.3 Le graphe : des relations d'ordres partiels

Type abstrait	relationnel de test principal donnant s'il y a une connexion entre 2 sommets. Il est non-orienté si le test est symétrique et orienté sinon.
Application	tri topologique (algorithme 5) [2, p.567] ; complexité : $O(\|V\| + \|E\|)$
Types concrets	liste d'adjacence $O(1)$ l'ajout/suppression et $O(n)$ test matrice d'adjacence $O(1)$ test et ajout/suppression $O(n)$

Algorithm 5 Tri topologique

entrée : $G = (V, E)$; sortie : L une liste contenant tous les sommets de V

1: Pour chaque sommet on fait un parcours en profondeur pour calculer les dates de fin de traitement
2: A chaque fin de traitement, insérer l'élément dans L
3: Renvoie L

Remarque. Le graphe induit un ordre non total (on a un ordre partiel) dans les données.

3 Les structures de recherche : dictionnaires

Les structures de recherche et en particulier le dictionnaire sont des structures facilitant la recherche dans un ensemble de données. Pour cela, elles vont les trier de manière à faciliter la recherche. On cherche à optimiser la fonction de recherche et les fonctions insertion, suppression ne sont pas forcément optimiser.
Type abstrait : **dictionnaire** : insert, delete, find

3.1 Les types concrets arborescents

Définition. Un arbre binaire de recherche [2, p.267] est un arbre binaire est un arbre dont les valeurs dans le sous-arbre droit d'un noeud dont inférieures à celui-ci et dont les valeurs dans le sous-arbre gauche sont supérieures.

Implémentation : Quelque soit le type concret, find consiste à descendre dans l'arbre tant qu'on n'a pas trouver la valeur que l'on recherche : on va à droite si la valeur recherchée est plus grande que le noeud courant et à gauche sinon. Pour insert et delete cela dépendant de la structure de données (c'est à ce moment que nous maintenons la structure ordonnée afin d'optimiser la recherche selon nos critères).

Proposition. *Dans le pire cas, toutes ses opérations se fond en $O(h)$ où h est la hauteur de l'arbre.*

Les arbres de recherche équilibré : AVL [2, p.310]

insert et delete	analogue au find mais on peut le déséquilibrer
Rééquilibrage	rotations gauche et droite (Figures 1 et 2)
Complexité	$\Theta(\log n)$

Les arbres optimaux [2, p.368]. Lorsque les recherches sur les données ne sont pas uniforme, un arbre non équilibré qui laisse accessible les clés les plus cherchées près de la racine. Un type concret est alors un arbre optimal construit aléatoirement suivant la fréquence de la recherche.

Proposition. *La création d'un arbre se fait en $\Theta(n^3)$ où n est le nombre de clés.*

3.2 Un type concret séquentiel : la table de hachage [2, p.235]

Définition. Une table de hachage est une généralisation d'un tableau où la case de stockage d'une donnée est donnée par une fonction souvent surjective.

Définition. Une collision entre deux données implique qu leur valeurs de hachage (par la fonction) sont égales.

Type concret	tableau de la taille de l'espace d'arriver de la fonction de hachage (souvent plus petit que celui de départ) et une liste simplement chaînée pour chacune de ces cellules	
Implémentation	insert	calcul du hasch et insertion dans la bonne liste
	delete	calcul du hasch et suppression dans la liste correspondante
	find	calcul du hasch et recherche dans la liste correspondante

Les problématiques autours du hachage parfait ne sont pas nécessaire dans cette leçon même si elles ne sont pas hors sujet. Maintenant, c'est une notion à garder en tête dès que l'on parle de hachage (même si le terme n'apparaît pas dans la leçon).

3.3 Stoker et manipuler des données en grands nombres : la base de données relationnelle

Objectif : On souhaite une structure de données qui permet de stocker et de manipuler des données en grand nombre efficacement (ces données ne peuvent pas être entièrement stockées sur un disque dur). Elle doit donc gérer un stockage externe efficace.

Dans ce contexte, on cherche à minimiser le nombre d'accès au disque car cet accès est vraiment très lent (facteur d'un million).

Type abstrait	**Base de données relationnelle** create, find, insert, delete
Application	recherche sur des données stockées sur un disque externe (algorithme 11)
Types concrets	B-arbres

Définition. Un B-arbre T possède les propriétés suivantes :

- chaque noeud possède les attributs suivants :
  - $x.n$ le nombre de clés ;
  - $x.cle_1 \leq \cdots \leq x.cle_n$ les n clés ;
  - $x.feuille$ qui indique si le noeud est une feuille ;
  - $x.c_i$ les $n+1$ pointeurs vers ses fils (s'il n'est pas une feuille) ;
- si k_i est stocké dans l'arbre de racine $x.c_i$ alors $x.c_{i-1} \leq k_i \leq x.c_i$;

— les feuilles sont toutes à la même hauteur ;

— t est le degré minimal de l'arbre T : $t-1 \leq x.n \leq 2t-1$ pour tout noeud de l'arbre sauf la racine (dans ce cas on a $1 \leq racine.n \leq 2t-1$).

Proposition. *Toutes les opérations sur un B-arbre (sauf* create*) s'effectuent en* $O(\log_t n)$ *où* t *est le degré minimal de l'arbre et* n *le nombre de données.*

Dans la propriété on garde le t car dans la vrai vie le t est très grand (de l'ordre du millier) ce qui nous donne une hauteur de l'arbre très petit (de l'ordre de l'unité). Donc il influence grandement la constante.

Dans cette leçon, il n'est pas nécessaire de parler d'arbre B+ mais il peut être intéressant de les garder en tête. Ils viennent généraliser les B-arbre dans le sens où ils permettent d'effectuer les opérations de recherche d'intervalles efficacement. Pour cela, on les construit comme les B-arbres sauf qu'on envoie une copie des clés présentes dans l'arbre dans ses feuilles.

4 Partitionner un ensemble : Union-Find [2, p.519]

La structure d'union-find est une structure permettant de représenter un ensemble de données à l'aide d'une partition. C'est une des rares structures de données qui cherche à optimiser l'union de deux structures et la recherche. Notons que la structure de donnée générale présentée en début de leçon met des gyrophares sur cette structure de données.

Objectif : On veut une structure de données constituées d'un ensemble de données permettant de savoir (efficacement) à quelle structures (vues comme des classes d'équivalences) chaque données intervient et unifier ces structures.

Type abstrait	**Union-find** create, find, union
Application	algorithme de Kruskal (les arbres couvrants de poids minimal) (algorithme 6) complexité $O(\|E\| \log \|V\|)$ optimal (avec structure union-find optimale)
Types concrets	listes chaînées et union par rang forêts et union par rang, compression de chemin

Hypothèse : La structure union-find ne touche pas aux données : l'ensemble des données admettent un pointeur (et réciproquement) vers leur copie dans union-find.

Proposition. *Une séquence de* m *opérations* find, create *et* union *telle qu'il y ait* n *opérations* create *se fait en* $O(n+m)$ *pour le type concret liste chaînée avec union par rang.*

Proposition. *Une séquence de* m *opérations* find, create *et* union *telle qu'il y ait* n *opérations* create *se fait en* $O(m\alpha(n))$ *pour le type concret forêt avec union par rang et compression de chemin.*

Quelques notions importantes

La file de priorité

Définition. Une file de priorité [2, p.149] est une structure de données permettant la gestion d'un ensemble S via leur clé. On a les opérations suivante : Insérer, Max, Extraire-Max et Augmenter-Clé.

Remarque. On a les même en MIN

Algorithm 6 Algorithme de Kruskal pour les arbres couvrant minimaux.

entrée : $G = (V, E), w$; sortie : L un ensemble contenant les arêtes formant l'arbre couvrant minimal

```
1: L ← ∅
2: for v ∈ V do
3:     create(v)
4: end for
5: Trier les arêtes de E par ordre croissant de poids
6: for (u, v) ∈ E pris par ordre croissant do
7:     if find(u) ≠ find(v) then
8:         L ← L ∪ {(u, v)}
9:         union(u, v)
10:    end if
11: end for
12: Renvoie L
```

Représentation : le tas binaire

Algorithm 7 Fonction permettant de donner le maximum d'une file de priorité.

```
1: function MAX(A)        ▷ A : file de priorité
2:     A[1]
3: end function
```

Algorithm 8 Fonction permettant d'insérer un nouvel élément dans une file de priorité.

```
1: function MAX(A, cle)   ▷ A : file de priorité ; cle : clé à insérer
2:     A.taille ← A.taille + 1
3:     A[A.taille] ← −∞
4:     Augmenter-Clé(A, A.taille, cle)
5: end function
```

Algorithm 9 Fonction permettant de retirer le maximum d'une file de priorité.

```
1: function EXTRAIRE-MAX(A)   ▷ A : file de priorité
2:     if A.taille < 1 then
3:         Erreur
4:     end if
5:     max ← A[1]
6:     A[1] ← A[A.taille]
7:     A.taille ← A.taille − 1
8:     Entasser(A, 1)
9:     Renvoie max
10: end function
```

Algorithm 10 Fonction permettant d'augmenter la clé d'un élément dans une file de priorité.

```
1: function AUGMENTER-CLÉ(A, i, cle)   ▷ A : file de priorité ; i : position de l'élément ; cle : clé à augmenter
2:     if cle < A[i] then
3:         Erreur
4:     end if
5:     A[i] ← cle
6:     while i > 1 et A[Parent(i)] < A[i] do
7:         Échanger A[i] et A[Parent(i)]
8:         i ← Parent(i)
9:     end while
10: end function
```

Les fonctions ainsi définies pour implémenter les opérations de la file de priorité sont : Max en $O(1)$, Extraire-Max en $O(\log n)$ (appel Entasser), Augmenter-Clé en $O(\log n)$ (descend dans l'arbre) et insérer en $O(\log n)$ (appel Augmenter-Clé).

Amélioration : tas de Fibonacci [2, p.467] Une amélioration possible du tas binaire est le tas de Fibonacci qui est une amélioration d'un tas fusionnable.

Opération	Tas binaire (pire cas)	Tas de Fibonacci (amortie)
Créer	$\Theta(1)$	$\Theta(1)$
Insérer	$\Theta(\log n)$	$\Theta(1)$
Min	$\Theta(1)$	$\Theta(1)$
Extraire-Min	$\Theta(\log n)$	$\Theta(\log n)$
Union	$\Theta(n)$	$\Theta(1)$
Diminuer-Clé	$\Theta(\log n)$	$\Theta(1)$
Supprimer	$\Theta(\log n)$	$\Theta(\log n)$

TABLE 1 – Comparaison des complexités en fonction de la structure de tas (certaines sont admises).

Définition. Un tas fusionnable est une structure de données implémentant les cinq opérations suivantes : Créer, Insérer, Min, Extraire-Min et Union.

Un tas de Fibonacci est un tas fusionnable muni des deux opérations suivantes : Diminuer-Clé et Supprimer. C'est un ensemble d'arbre ordonnés enraciné en tas min (chacun des tas est un tas min). Chaque nœud est relié à ses enfants, ses parents et ses frères avec une liste doublement chaînée circulaire. On ajoute également un pointeur qui indique quel est l'élément minimal du tas (nécessairement une racine). Les opérations de ce tas sont alors implémentées comme suit :

Insérer On insère le nouveau nœud comme une racine d'un nouvel arbre. On vérifie ensuite qu'il n'est pas le plus petit.

Min On donne l'élément pointé par le pointeur sur l'élément minimal.

Union On concatène les deux listes de racines. Ensuite on regarde lequel des deux minimum est réellement le minimum du tas.

Extraire-Max On transforme les fils de l'élément minimal en racine que l'on supprime. On appelle alors une fonction de consolidation qui relie les racines de degré égal jusqu'à obtention d'une unique racine par degré. On trouve deux racines et on en fait un tas : la plus grande racine vient se greffer sur l'autre. C'est lors de l'extraction du minimum que l'on diminue la taille de la liste des racines de notre tas.

Applications des files de priorité Dans la pratique, les files de priorité servent à beaucoup de chose. Elles améliorent des algorithmes qui utilise en grand nombre l'extraction d'un minimum ou de la suppression dans un ensemble de données. Dans le cas de graphes denses qui appelle à la diminution de clé (exemple de problème de flots), elles sont très utilisées. Elles sont, notamment, le support d'algorithmes de calcul d'arbre couvrant minimal (algorithme de Prim [2, p.586]), de recherche de chemin le plus cours (algorithme de Dijstra [2, p.609]). Cependant, elles sont absolument inefficace pour la recherche.

AVL

Référence : [3, p.224]
Historiquement : introduit en 1960 par Adelson-Velskii et Landis
Ils sont une des implémentations possibles d'AVL équilibrés.

Définition des arbres H-équilibré

FIGURE 1 – Une rotation droite

Définition. On dit qu'un arbre est H-équilibré si en tout nœud de l'arbre, les hauteurs des fils gauches et droits diffèrent d'au plus 1. Si on appelle la différence entre les hauteurs du fils gauche et du fils droit, alors ce nombre appartient à l'ensemble $\{-1; 0; 1\}$.

Définition. Un AVL est un ABR H-équilibré.

Proposition. *Tout arbre H-équilibré ayant n nœuds a une hauteur h vérifiant :*

$$\log_2(n+1) \leq h+1 \leq 1.44 \log_2(n+2)$$

Idée de la preuve. La borne inférieure : arbre complet.

La borne supérieure : arbre "déséquilibré" : si on enlève une feuille quelconque alors soit cet arbre n'et plus H-équilibré soit sa hauteur diminue de 1. On obtient alors une récurrence qui est la suite de Fibonacci. □

Ajout dans un AVL L'ajout d'un élément (comme la suppression ou la recherche) s'effectue de la même manière que dans un ABR. Il nous faut alors toujours $O(logn)$ comparaisons. Cependant, l'ajout (ou la suppression) d'un élément peut déséquilibrer l'AVL : il nous faut donc le rééquilibré.

Principe du rééquilibrage : Soit $T =< r, G, D >$ un AVL et supposons que nous ajoutons un élément x dans G (on a le même raisonnement qi on ajout x dans D) et que G reste un AVL (on a une construction inductive). De plus, la hauteur du sous-arbre G augmente alors de 1 (sinon, on a rien à faire).

1. Si le déséquilibre de T valait 0, maintenant il vaut 1.
2. Si le déséquilibre de T valait -1, maintenant il vaut 0.
3. Si le déséquilibre de T valait 1, T est déséquilibré, il faut alors le rééquilibrer. Nous avons alors deux opérations pour le rééquilibre : la rotation à droite (on ajoute x dans le fils droit de G) (Figure 1), la rotation à gauche (on ajoute x dans le fils gauche de G) (Figure 2).

Proposition. *Toute adjonction dans un AVL nécessite au plus une rotation pour le rééquilibrer*

Idée de la preuve. Les rotations ne peuvent être effectué que sur le chemin de x.

Les rotations ne modifient pas la hauteur de l'arbre sur laquelle elle est effectuée. Donc une rotation est effectuée sur le dernier nœud nécessitant un rééquilibrage et on a gagné. □

Complexité dans le pire cas : $\Theta(\log n)$
Complexité en moyenne : ? Empiriquement : une rotation toutes les deux adjonctions.

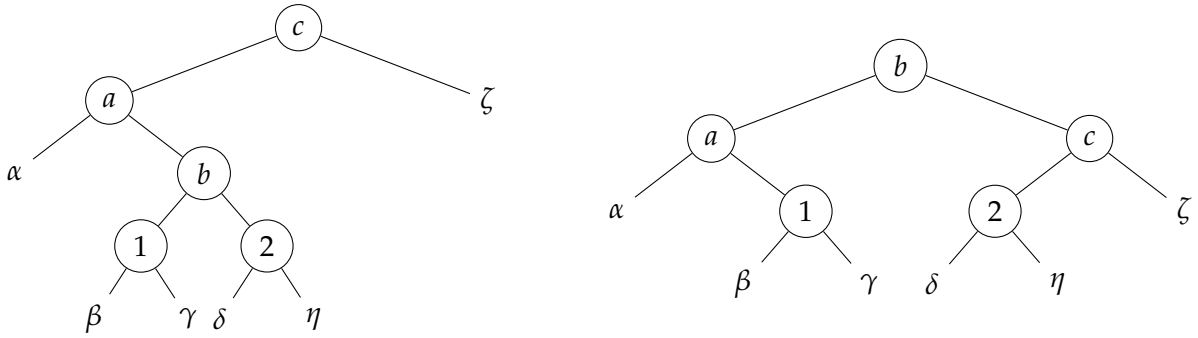


FIGURE 2 – Une rotation gauche

Suppression dans un AVL La suppression dans un AVL se fait de la même manière que dans un AVL : on remplace l'élément supprimé (quand ce n'est pas une feuille) par le plus grand élément de son fils gauche ou par le plus petit élément de son fils droit. Ceci peut entraîner un déséquilibre de l'arbre. On utilise donc les même rotations que dans le cas de l'adjonction : rotation droite (Figure 1) et rotation gauche (Figure 2). Cependant contrairement à l'adjonction, on peut avoir besoin d'effectuer ces rotations jusqu'à la racine.

Complexité dans le pire cas : $\Theta(\log n)$

Complexité en moyenne : ? Empiriquement : une rotation toutes les cinq suppressions.

Arbres rouge-noirs

Référence : [2, p.287]

Les arbres rouge-noir sont des arbres approximativement équilibrée car en rajoutant un unique bit (sa couleur) dont on contrôle sa valeur nous pouvons garantir qu'aucun des chemins n'est plus de deux fois plus long que n'importe quel autre chemin.

Définition des arbres rouge-noir On ajout alors l'attribue COULEUR qui peut prendre les valeurs ROUGE et NOIR, à chacun des nœuds de l'arbre.

Définition. Un arbre rouge-noir est un arbre vérifiant les propriétés rouge-noir suivantes :

1. chaque nœud est soit rouge, soit noir ;
2. la racine est noire ;
3. chaque feuille est noire ;
4. si un nœud est rouge, alors ses deux enfants sont noirs ;
5. pour chaque nœud, tous les chemins simples reliant le nœud à des feuilles situées plus bas contiennent le même nombre de nœuds noirs.

On peut alors définir la notion de hauteur noire (notée *bh*) d'un nœud qui représente le nombre de nœuds noirs dans un chemin simple du nœud.

Remarque. Lors de l'implémentation, nous utilisons souvent une sentinelle (qui possède déjà la couleur noire) pour représenter toutes les feuilles.

Lemme. *Un arbre rouge-noir ayant n nœuds internes a une hauteur au plus égale à* $2\log(n+1)$.

Idée de la preuve. On montre par récurrence sur la hauteur d'un nœud x que le sous-arbre enraciné en x contient au moins $2^{bh(x)} - 1$ nœuds internes.

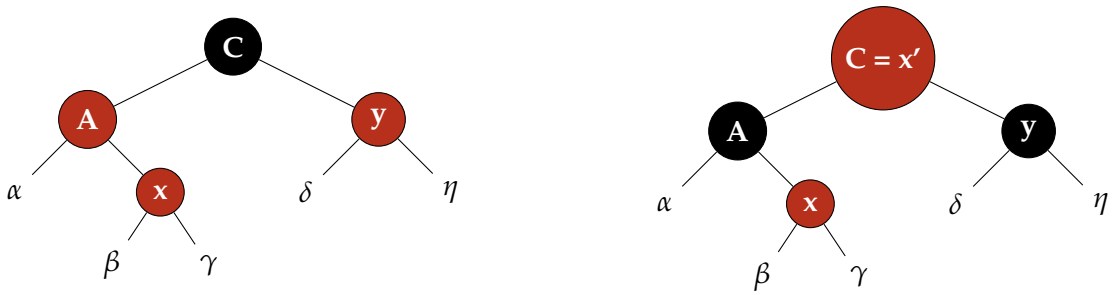


FIGURE 3 – Résolution du cas 1 de l'insertion, lorsque x est fils gauche (le cas à droite se traite de manière analogue)

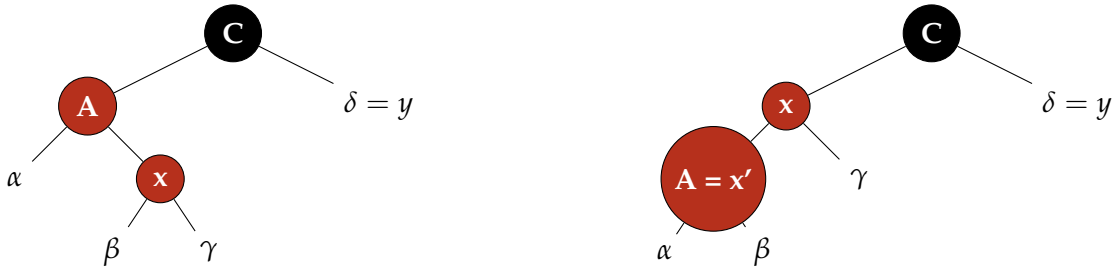


FIGURE 4 – Passage du cas 2 de l'insertion au cas 3 de l'insertion.

On remarque ensuite que d'après la propriété 4 (dans la définition) que $bh(r) \geq \frac{h}{2}$ où r est la racine de l'arbre et h est sa hauteur.

On obtient le résultat en passant au log. □

Conséquences : On peut effectuer les actions d'adjonctions, de suppressions et de recherche en $O(\log n)$.

Rotations Comme pour les AVL, insérer ou supprimer un élément dans cet arbre peut modifier sa structure. Pour cela, on utilise la même rotation droite (Figure 1). Cependant, on définit la rotation gauche comme la réciproque de la rotation droite (Figure 1).

Insertion dans un arbre rouge-noir Pour insérer un nouveau élément dans l'arbre, on le descend au feuille (ses fils pointent sur la sentinelle) et on met alors sa couleur à ROUGE. Cette action risque alors de violer les propriétés rouge-noir. Il nous faut donc les corriger.

On commence par remarqué que seules les propriétés 2 (quand x devient la racine) et 5 (quand le parent de x est également rouge) peuvent être violées. On va alors étudier ces différents cas.

Cas 1 : l'oncle de x, y, est ROUGE Le cas 1 (Figure 3) se produit lorsque que x et son parent A sont tout deux ROUGE. Comme A est ROUGE, on peut colorier son parent C (qui est NOIR) en ROUGE. Dans ce cas, A et y sont à lors tour colorier en NOIR. On applique alors récursivement la procédure à C qui devient le nouveau x, x'.

Cas 2 : l'oncle de x, y est NOIR et x est un enfant de droite. Dans le cas 2 (comme dans le cas 3), y est toujours noir. Les deux cas ne diffère que par le côté d'insertion de x. On va alors utiliser la rotation gauche sur le parent de x, A pour se ramener au cas 3 (Figure 4).

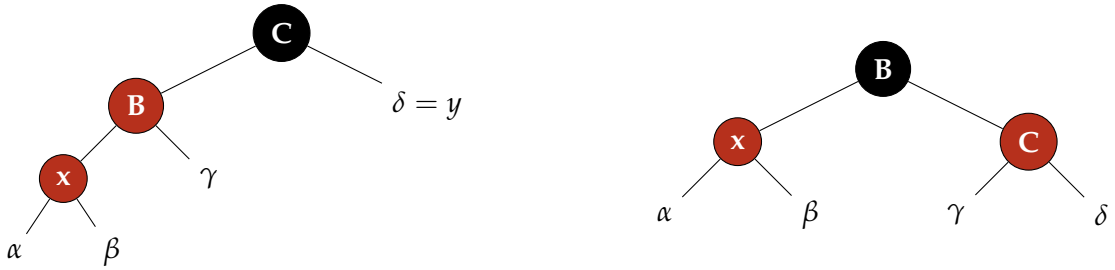


FIGURE 5 – Résolution du cas 3 de l'insertion.

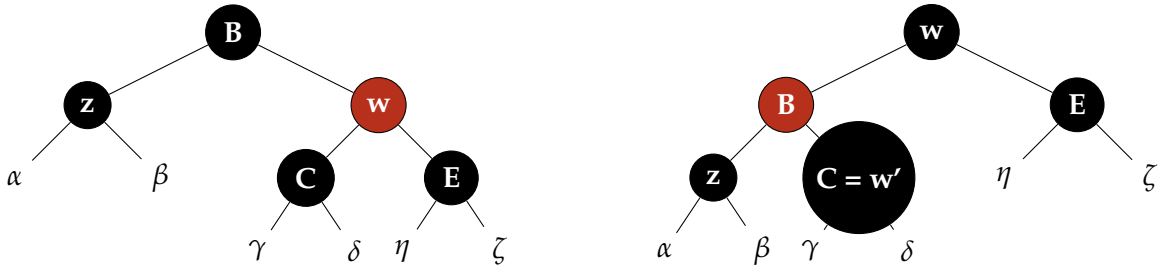


FIGURE 6 – Résolution du cas 1 de la suppression.

Cas 3 : l'oncle de x, y est NOIR et x est un enfant de gauche. Le cas 3 arrive lorsque y est NOIR et que x a été insérer à gauche. On commence alors par effectuer une rotation droite sur le parent du parent de x (qui est B), C. Ensuite, on met B (qui est alors la nouvelle racine du sous-arbre) à NOIR et C prend alors la couleur ROUGE. On ne crée aucune violation donc on n'a pas besoin de réitérer la correction.

Complexité : On n'effectue jamais plus de deux rotations. On a donc une complexité en $O(\log n)$ comme annoncé.

Suppression dans un arbre rouge-noir La suppression d'un nœud x dans un arbre rouge-noir est potentiellement plus complexe que l'insertion. Comme l'insertion d'un nœud, la suppression peut violé les propriétés de l'arbre qu'il faut alors rétablir.

Les différences avec la suppression dans un arbre :

— Le nœud y est supprimé ou déplacé dans l'arbre. Lorsque y est déplacé, il prend la place de x (sans condition).

— Comme la couleur de y risque de changer (quand y prend la place de x, il conserve également la couleur de x), nous conservons en mémoire sa couleur d'origine. En effet si la couleur de y est NOIR nous pouvons avoir un problème.

— Le nœud z prend alors la place de y et le parent de z reste toujours le parent originel de y.

Comme lors de l'insertion, nous risquons de violer les propriétés rouge-noir. Nous devons alors les corriger. l'idée fondamentale de cette correction est de conserver le nombre de nœuds NOIR. Nous allons présenter les différents cas de corrections.

Cas 1 : le frère de z, w est ROUGE. Le cas 1 se produit lorsque le frère de z, w est ROUGE. Comme w doit avoir des enfants NOIR (et que son père B est nécessairement NOIR), on peut permuter les couleurs de B et de w. On finit alors par une rotation gauche (Figure 6).

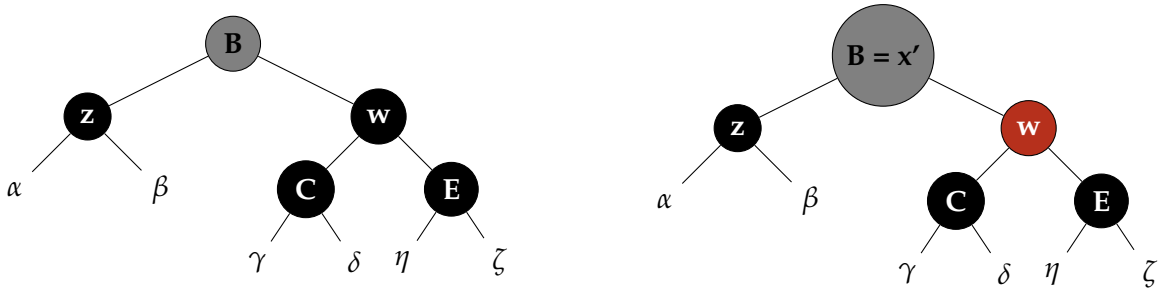


FIGURE 7 – Résolution du cas 2 de la suppression (le nœud gris peut être ROUGE ou NOIR mais il conserve sa couleur durant la transformation).

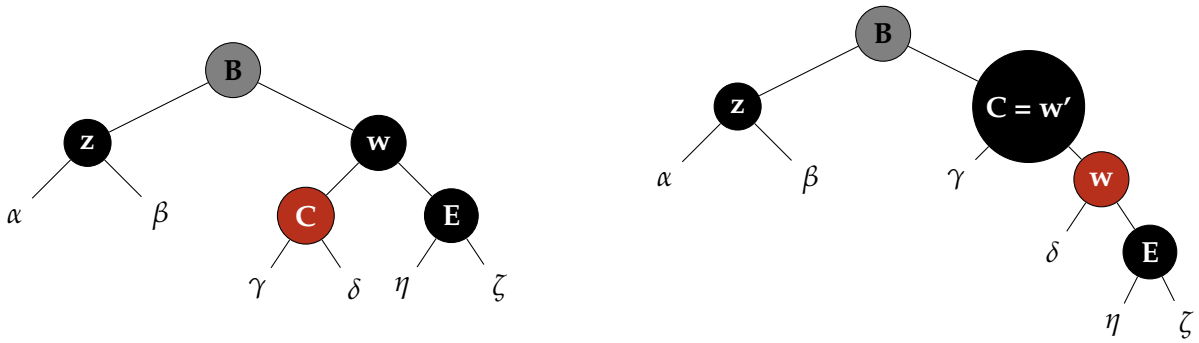


FIGURE 8 – Résolution du cas 3 de la suppression (le nœud gris peut être ROUGE ou NOIR mais il conserve sa couleur durant la transformation).

Cas 2 : le frère de z, w est NOIR et les deux enfants de w sont NOIRs. Le cas 2 se produit lorsque le frère de z, w est NOIR et les deux enfants de w sont NOIRs. On enlève alors un NOIR à x et à w ce qui laisse w à ROUGE. Pour compenser le passage de w à rouge, son parent B peut doit devenir ROUGE alors qu'il pouvait être des deux couleurs. Il faut donc rappliqué l'opération à B qui devient le nouveau x, x' (Figure 7).

Cas 3 : le frère de z, w est NOIR et l'enfant gauche de w est ROUGE et l'enfant droit de w est NOIR. Le cas 3 se produit quand le frère de z, w est NOIR et le fils gauche de w, C, est ROUGE et le fils droit de w, E est NOIR. On va chercher à se ramener au cas 4. Comme E est NOIR, on peut permuter la couleur de w (NOIR) avec celle de C, (ROUGE). Puis on effectuer une rotation droite sur w (sans violer les propriétés de l'arbre) ce qui nous ramène au cas 4 (Figure 8).

Cas 4 : le frère de z, w est NOIR et l'enfant droit de w est ROUGE. Le cas 4 se produit lorsque le frère de z, w est NOIR et le fils droit de w, E, est ROUGE. En modifiant les couleurs : E et le parent de w, B, deviennent NOIR et w prend la couleur de B ; on ne violent pas les propriétés rouge-noir. En effectuant une rotation gauche sur B, on supprime le noir supplémentaire. On obtient alors un arbre correct (Figure 9).

Complexité : Quelque soit le cas que l'on prend, on fait un nombre constant de changement de couleur et on effectue au plus trois rotations. La suppression reste donc en $O(\log n)$ comme annoncé.

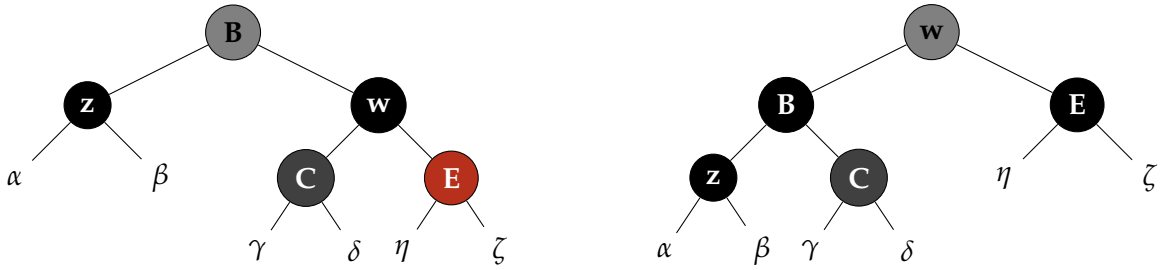


FIGURE 9 – Résolution du cas 4 de la suppression (les nœuds gris peut être ROUGE ou NOIR mais ils conservent leur couleur durant la transformation). Dans ce cas, le nouveau x est la racine de l'arbre.

B-arbres

Motivations : Minimiser le nombre d'action sur un disque externe (c'est ce qui prend le plus de temps, en ms).

Un B-arbre est un arbre de la recherche avec une ramification importante et une hauteur plutôt faible. En pratique un noeud de notre arbre est une page de notre disque externe.

Hypothèses : on se donne deux opérations atomiques : ECRITURE-DISQUE et LECTURE-DISQUE (on ne peut pas les découper en sous fonctions, elles travaillent uniquement sur une page du disque dur). On veut montrer l'optimalité de la structure de donnée face à l'utilisation de ces deux opérations.

Définition d'un B-arbre et exemple On donne maintenant la définition d'un B-arbre. La figure 10 nous donne un exemple d'un tel arbre.

Définition. Un B-arbre est un arbre T possédant les propriétés suivantes :

1. Chaque noeud x contient les attributs ci-après :
   (a) $x.n$ le nombre de clés conservées dans le noeud (le noeud est alors d'arité $n+1$) ;
   (b) les $x.n$ clés $x.cle_1, \dots x.cle_n$ stockées dans un ordre croissant (structure de donnée associée : liste) ;
   (c) un booléen, $x.feuille$ marquant si le noeud est une feuille ;
   (d) les $n+1$ pointeurs des fils, $\{x.c_1, \dots x.c_{n+1}\}$;
2. Les clés et les valeurs dans les arbres fils vérifient la propriété suivante : $k_0 \leq x.cle_1 \leq \dots \leq x.cle_n \leq k_n$;
3. toutes les feuilles ont la même profondeur, qui est la hauteur h de l'arbre ;
4. Le nombre de clé d'un noeud est contenu entre $t-1 \leq n \leq 2t-1$. (Pour la racine on n'a que la borne supérieure, la borne inférieure est 1 : on demande qu'il y ait au moins une clé dans la racine) où t est le degré minimal du B-arbre.

Remarque. Dans la vrai vie, t est choisi en fonction du nombre de tuples que peut contenir une page du disque.

Théorème. *Si $n \geq 1$, alors pour tout B-arbre T à n clés de hauteur h et de degré minimal $t \geq 2$, $h \leq \log_2 \frac{n+1}{2}$.*

Démonstration. On montre par récurrence sur la hauteur de l'arbre que

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} = 2t^h - 1$$

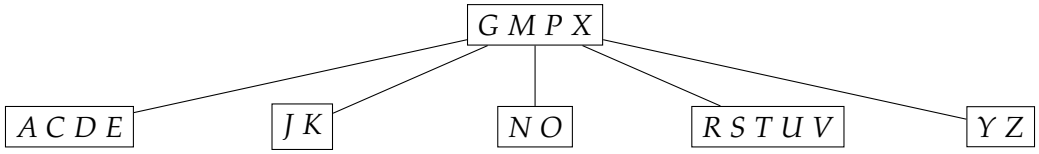


FIGURE 10 – Exemple d'un B-arbre de degré minimal $t = 3$.

D'où $t^h \leq \frac{(n+1)}{2}$ et on conclut en passant au $\log_t$. □

Hypothèses :

— La racine du B-arbre se trouve toujours en mémoire principal : on n'a pas de LIRE-DISQUE ; cependant, il faudra effectuer un ÉCRITURE-DISQUE lors de la modification de la racine.

— Tout noeud passé en paramètre devra subir un LIRE-DISQUE.

Recherche dans un B-arbre La recherche dans un B-arbre est analogue à la recherche dans un ABR mais dans ce cas on prend une décision sur les n clés de l'arbre, on cherche donc à placer la clé cherchée dans un des intervalles définis par les n clés.

L'algorithme prend en paramètre l'élément recherché k et le noeud courant x. On retourne alors le noeud x et la place de k dans le noeud x, si k est dans l'arbre. Sinon, on retourne NIL.

Algorithm 11 Recherche dans un B-arbre

```
1:  function RECHERCHE-BARBRE(x, k)   ▷ x a subit une lecture dans le disque LIRE-DISQUE
2:      i ← 1
3:      while i ≤ x.n et k > x.cle_i do        ▷ Recherche de l'intervalle dans lequel se trouve k
4:          i ← i + 1
5:          if i ≤ x.n et k = x.key_i then                              ▷ On a trouvé k
6:              Retourne (x, i)
7:          else                                                        ▷ On n'a pas trouvé k
8:              if x.feuille then                                       ▷ On a fini l'arbre et k n'y est pas
9:                  Retourne NIL
10:             else    ▷ On peut continuer à descendre dans l'arbre : récursivité sur le fils de x
11:                 LIRE-DISQUE(x.c_i)
12:                 Retourne RECHERCHE-BARBRE(x.c_i, k)
13:             end if
14:         end if
15:     end while
16: end function
```

Complexité : $O(log_t n) = O(h)$ lecture-écriture sur le disque (on ne fait que des lectures) où h est la hauteur et n le nombre de nœuds. Comme $n < 2t$, la boucle **while** s'effectue en $O(t)$, le temps processeur total est $O(th) = O(t \log n)$.

Insertion dans un B-arbre L'insertion dans un B-arbre est plus délicate que dans une ABR. Comme dans un ABR, on commence par chercher la feuille sur laquelle on doit insérer cette nouvelle clé. Cependant, on ne peut pas juste créer une nouvelle feuille (on obtient un arbre illicite). On est alors obliger d'insérer l'élément dans un noeud déjà existant. Comme on ne peut pas insérer la clé dans un noeud plein, on introduit la fonction PARTAGE qui sépare le

noeud plein en deux sous noeud distincts autours de la valeur médiane qui remonte dans le noeud père (à qui il faut peut-être appliquer PARTAGE). Cette fonction PARTAGE, nous permet, contrairement aux AVL ou arbre rouge-noir, de faire grandir l'arbre vers le haut à l'aide de la fonction PARTAGE. Nous conservons alors un équilibre "parfait".

Principe de l'algorithme : On descend alors dans l'arbre pour chercher l'emplacement de la nouvelle clé, de manière analogue à la fonction RECHERCHE-BARBRE. On souhaite alors insérer la clé dans la feuille que l'on trouve. Dans le cas où cette feuille est pleine, nous devons appliqué la fonction PARTAGE qui conduit à insérer une clé dans le noeud père. Dans le pire des cas, on est alors amené à remonter entièrement l'arbre avec cette fonction PARTAGE. En implémentant intelligemment la fonction INSERTION-BARBRE, nous pouvons faire qu'une descente dans l'arbre. Lors de la descente de l'arbre nous allons appliquer PARTAGE à tous les nœuds pleins que nous parcourons. On effectue la fonction PARTAGE avant toutes opérations d'insertion dans l'arbre (la parité du nombre de clé nous permet de séparer le noeud plus facilement).

La fonction PARTAGE qui effectue un copié-collé des bon pointeurs, prend en paramètre un noeud x (en mémoire vive) et un paramètre i qui indique le noeud que l'on doit séparer ($x.c_i$). Elle modifie alors x en lui ajoutant la valeur médiane de $x.c_i$, $x.c_i$ en lui retirant la moitié de ces valeurs et créer un nouveau noeud z fils de x contenant les valeurs restantes.

Algorithm 12 Partage un noeud plein dans un B-arbre

1: **function** PARTAGE(x, i) ▷ x a une lecture dans le disque LIRE-DISQUE
2: $z \leftarrow$ ALLOUER-NOEUD() ▷ z va récupérer les valeurs les plus grandes et devenir un enfant de x
3: $y \leftarrow x.c_i$ ▷ Récupère les plus petites valeurs
4: $z.feuille \leftarrow y.feuille$; $z.n \leftarrow t-1$
5: **for** $j = 1$ à $t-1$ **do** ▷ Fabrication du noeud z : partage de ces clés
6: $z.cle_j \leftarrow y.cle_{j+t}$
7: **end for**
8: **if** non $y.feuille$ **then** ▷ Fabrication du noeud z : partage de ces fils
9: **for** $j = 1$ à t **do**
10: $z.c_j \leftarrow y.c_{j+1}$
11: **end for**
12: **end if**
13: $y.n \leftarrow t-1$
14: **for** $j = x.n+1$ décrois jusqu'à $i+1$ **do** ▷ Modification de x pour la nouvelle valeur : fils
15: $x.c_{j+1} \leftarrow x.c_j$
16: **end for**
17: $x.c_{i+1} \leftarrow z$
18: **for** $j = x.n$ décrois jusqu'à i **do** ▷ Modification de x pour la nouvelle valeur : clés
19: $x.cle_{j+1} \leftarrow x.cle_j$
20: **end for**
21: $x.cle_i \leftarrow y.cle_t$
22: $x.n \leftarrow x.n+1$
23: ÉCRIRE-DISQUE(x) ; ÉCRIRE-DISQUE(y) ; ÉCRIRE-DISQUE(z) ▷ Partage du résultat
24: **end function**

Complexité : $O(1)$ en lecture-écriture sur le disque (on ne fait que trois écritures et deux lecture et une allocation qui utilise ALLOUER-NOEUD : alloue une nouvelle page sur le disque en $O(1)$.). Par les deux boucles **for** en $\Theta(t)$, le temps processeur total est $\Theta(t)$.

On définit la fonction INSERTION-INCOMPLET-BARBRE (une sous fonction de INSERER-

BARBRE) qui insère k dans un noeud d'un B-arbre de racine x, supposé non plein. L'utilisation de cette fonction dans l'insertion globale garantie l'hypothèse.

Algorithm 13 Insertion dans un noeud non-plein d'un B-arbre

```
1:  function INSERTION-INCOMPLET-BARBRE(x, k)        ▷ x a une lecture dans le disque
    LIRE-DISQUE
2:      if x.feuille then                            ▷ Il faut qu'on insère k dans x
3:          while i ≥ n et k < x.cle_i do
4:              x.cle_{i+1} ← x.cle_i ; i ← i − 1
5:          end while
6:          x.cle_{i+1} ← k ; x.n ← x.n + 1
7:          ÉCRIRE-DISQUE(x)
8:      else                                         ▷ k doit être insérer dans un fils de x ; récursivité
9:          while i ≥ n et k < x.cle_i do
10:             i ← i − 1
11:         end while
12:         i ← i + 1
13:         ÉCRIRE-DISQUE(x.c_i)
14:         if x.c_i.n = 2t − 1 then
15:             PARTAGE(x, i)
16:             if k > x.cle_i then
17:                 i ← i + 1
18:             end if
19:         end if
20:         INSERTION-INCOMPLET-BARBRE(x.c_i, k)
21:     end if
22: end function
```

Complexité : $O(log_t n) = O(h)$ lecture-écriture sur le disque (on ne fait que $O(1)$ de lecture-écriture entre deux appels récursif) où h est la hauteur et n le nombre de nœuds. Comme $n < 2t$, les boucles **while** s'effectue en $O(t)$, le temps processeur total est $O(th) = O(t \log n)$.

On définit la fonction INSERTION-BARBRE qui insère k dans B-arbre T. On utilise PARTAGE et INSERTION-INCOMPLET-BARBRE pour assurer que la récursivité ne descendent jamais sur un noeud plein. Cela nous assure qu'on a toujours une unique descente dans l'arbre.

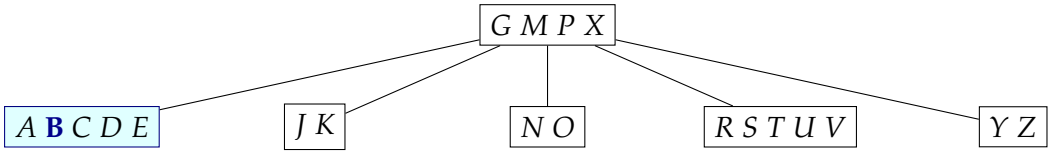
Complexité : $O(log_t n) = O(h)$ lecture-écriture sur le disque (on ne fait qu'une seule descente de l'arbre) où h est la hauteur et n le nombre de nœuds. Le temps processeur total est $O(th) = O(t \log n)$.

Suppression dans un B-arbre La suppression dans un B-arbre est plus délicate que l'insertion car nous sommes amenés à potentiellement modifier tous les nœuds internes (et plus seulement les feuilles). De manière analogue à l'insertion, la suppression peut construire un arbre avec un noeud illégal (ne effet, il peut devenir trop petit). Un algorithme naïf pourrait avoir tendance à rebrousser chemin quand un noeud devient trop petit (et à faire deux plusieurs aller-retour dans l'arbre). Nous allons présenter un algorithme qui se fait en une seule passe et ne remonte pas dans l'arbre (sauf lorsque nous devons rétrécir la hauteur de l'arbre). Dans le cas où un noeud se retrouverait sans clé alors il est supprimé et son unique fils devient la racine de l'arbre (la hauteur de l'arbre diminue de 1). L'arbre rétrécie par le haut (comme il grandit).

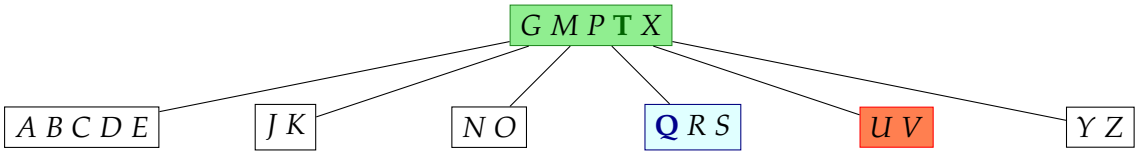
Principe de la suppression : On cherche la clé que l'on souhaite supprimer. Comme pour

Algorithm 14 Insertion d'un élément dans un B-arbre

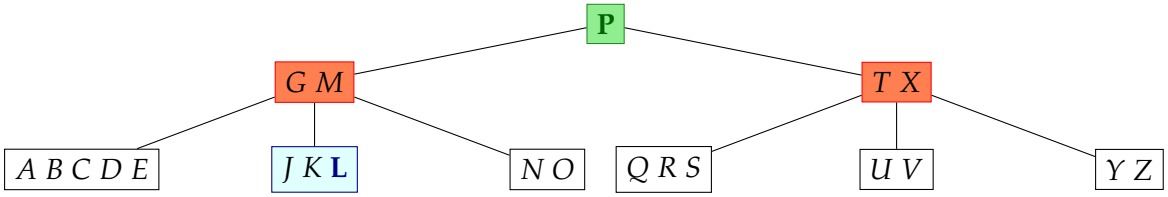
```
1:  function INSERTION-BARBRE(T, k)          ▷ La racine de l'arbre est en mémoire vive
2:      r ← T.racine
3:      if r.n = 2t − 1 then                                  ▷ La racine est pleine
4:          s ← ALLOUER-NOEUD()
5:          T.racine ← s ; s.feuille ← FAUX
6:          x.n ← 0 ; x.c_1 ← r
7:          PARTAGE(s, 1)
8:          INSERTION-INCOMPLET-BARBRE(s, k)
9:      else
10:         INSERTION-INCOMPLET-BARBRE(r, k)
11:     end if
12: end function
```



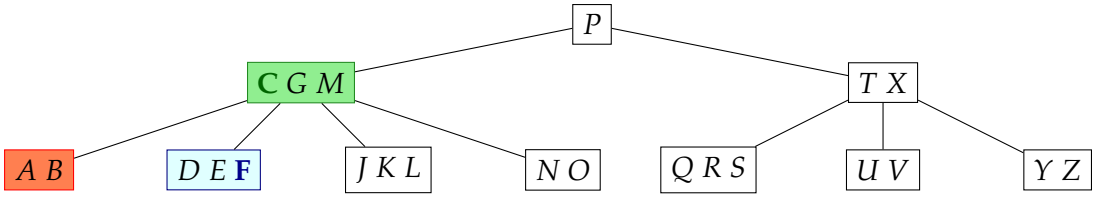
(a) Insertion de la lettre B dans le B-arbre (Figure 10). Le noeud en bleu contient cette nouvelle clé.



(b) Insertion de la lettre Q dans le B-arbre précédent. Le noeud bleu contient cette nouvelle clé, le noeud vert est le noeud qui a hérité de clé suite à l'action de PARTAGE et le noeud rouge est le nouveau noeud crée suite à PARTAGE pour lequel on n'a pas manipulé ces valeurs.



(c) Insertion de la lettre L dans le B-arbre précédent. Le noeud bleu contient cette nouvelle clé. Comme la racine était pleine, nous lui avons appliqué l'opération PARTAGE : le noeud vert est le noeud (la nouvelle racine) qui a hérité de clé suite médiane fait grandir l'arbre vers le haut. Les nœuds rouges sont les deux fils crées suite à PARTAGE pour lequel on n'a pas manipulé leurs valeurs.



(d) Insertion de la lettre L dans le B-arbre précédent. Le noeud bleu contient cette nouvelle clé. Comme la racine était pleine, nous lui avons appliqué l'opération PARTAGE : le noeud vert est le noeud (la nouvelle racine) qui a hérité de clé suite médiane fait grandir l'arbre vers le haut. Les nœuds rouges sont les deux fils crées suite à PARTAGE pour lequel on n'a pas manipulé leurs valeurs.

FIGURE 11 – Insertions dans un B-arbre.

l'insertion, nous nous assurons que nous pouvons descendre dans l'arbre et supprimer la clé en toute légalité.

1. Si la clé k est dans le noeud x qui est une feuille : on supprime k.
2. Si la clé k est dans le noeud x qui n'est pas une feuille :
   (a) Si l'enfant y qui précède k a au moins t clés, on cherche le prédécesseur de k, k' (max dans la liste des clés de y). On supprime récursivement k' dans y et on remplace k par k'. Permet de conserver l'arité du noeud x.
   (b) Si l'enfant y qui précède k a $t-1$ clés, on examine symétriquement le fils suivant z. Si z a au moins t clés, on cherche le successeur de k, k' (min dans la liste des clés de z). On supprime récursivement k' dans z et on remplace k par k'. Permet de conserver l'arité du noeud x.
   (c) Si y et z ont $t-1$ clés, on fusionne k dans le contenu de z et on fait tout passer (en copiant) dans y qui contient $2t-1$ clés (x perd son pointeur vers z et k). On libère z et on supprime récursivement k dans y.
3. Si la clé k n'est pas dans le noeud x, on cherche le sous arbre $x.c_i$ qui devrait contenir k (si il est dans l'arbre). Si $x.c_i.n = t-1$ on exécute un des deux sous-cas suivants (selon les besoin) pour garantir que l'on descend dans un noeud à au moins t clés. On applique alors la récursivité sur l'enfant approprié de x.
   (a) Si $x.c_i$ n'a que $t-1$ clé mais qu'un de ces frères immédiats, y à au moins t clés, on bascule une des clés de y (min ou max en fonction du côté) dans x et on bascule la clé supplémentaire de x dans $x.c_i$. Puis on reconnecte les pointeur la où il faut.
   (b) Si $x.c_i$ et ses frères immédiats n'ont que $t-1$ clés, nous fusionnons les deux frères en descendant la bonne clé de x (devient la clé médiane du noeud).

Complexité : $O(log_t n) = O(h)$ lecture-écriture sur le disque (on ne fait que $O(1)$ de lecture-écriture entre deux appels récursif) où h est la hauteur et n le nombre de nœuds. Comme $n < 2t$, les boucles **while** s'effectue en $O(t)$, le temps processeur total est $O(th) = O(t \log n)$.

Les arbres splay

Les arbres splay sont une implémentation possible pour les arbres de recherche optimaux. Lorsqu'un élément est recherché, il est placé à la racine (recherche auto-adaptative sur un arbre). Ce sont donc des abres binaires de recherche qui permettent d'atteindre l'élément cherché le plus rapidement possible. Ils sont basés sur une opération principale SPLAY qui consiste à faire remonter un noeud à la racine par rotation successive. Ils ont été inventé par Daniel Sleator et Robert Tarjan en 1985.

Opérations : Les arbres splays permettent de faire les opérations suivantes en $O(h)$:

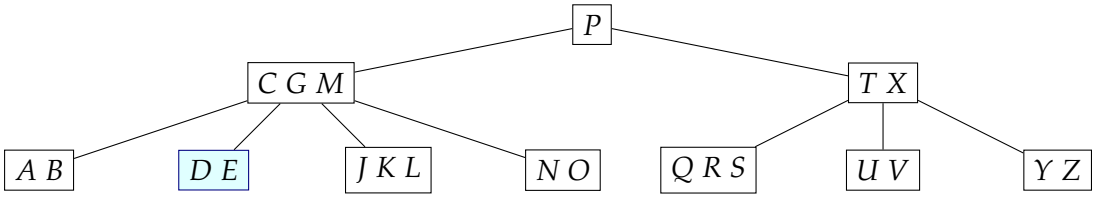
— RECHERCHE(i, S) détermine si l'élément i est un élément de l'arbre S ;

— INSERTION(i, S) ajoute l'élément i dans S s'il n'est pas déjà dedans ;

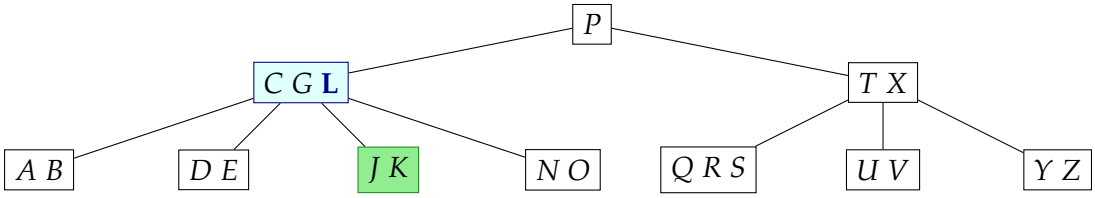
— SUPPRESSION(i, S) supprime l'élément i de S s'il est présent ;

— UNION(S, S') fusionne S et S' dans un unique arbre splay avec comme hypothèse $\forall x \in S, \forall y \in S', x < y$;

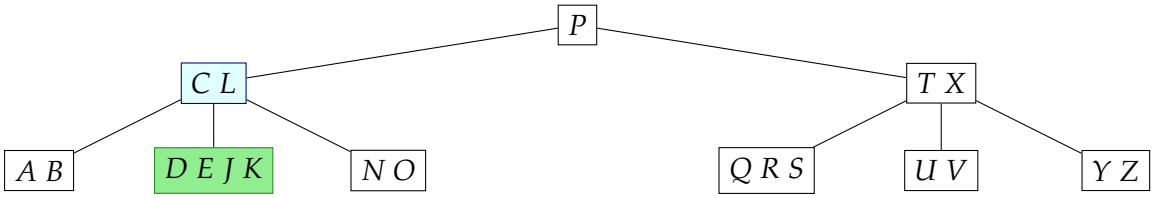
— PIVOT(i, S) sépare l'arbre splay en deux sous-arbre S' et S'' tels que $x \leq i \leq y$ avec $x \in S'$ et $y \in S''$.



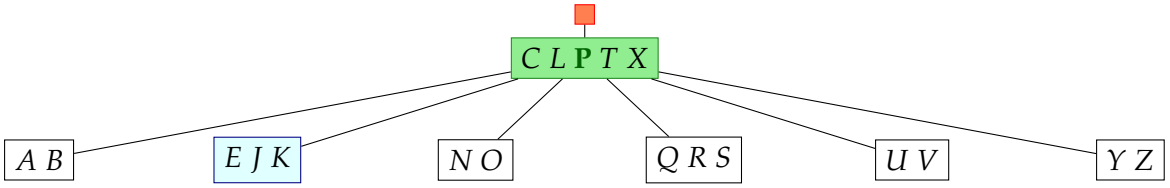
(a) Suppression de la lettre F (**cas 1**) dans le B-arbre (Figure 11). Le noeud en bleu contenait la clé supprimée, c'est le noeud que nous avons manipulé.



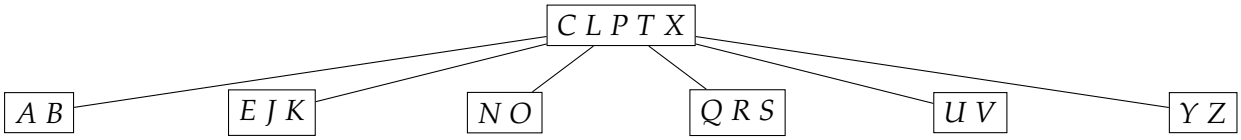
(b) Suppression de la lettre M dans le B-arbre précédent (**cas 2a**). Le noeud bleu contenait la clé supprimé. On lui a ajouter une nouvelle clé issue du noeud vert afin qu'il reste légal.



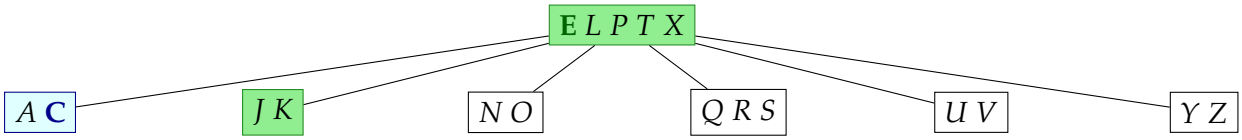
(c) Suppression de la lettre G dans le B-arbre précédent (**cas 2c**). Le noeud bleu contenait la clé supprimé. Le noeud vert est le résultat (légal) de la fusion des deux fils de la clé enlevée : on ne pouvait pas appliquer une rotation comme précédemment.



(d) Suppression de la lettre D dans le B-arbre précédent (**cas 3b**). Comme on ne peut pas descendre dans la récursivité (car le noeud *C L* ne contient que deux clés) nous avons fusionner les deux fils de la racines en rajoutant la clé *P*. Le noeud vert est le résultat de cette fusion. Le noeud bleu contenait la clé supprimé. Le noeud rouge quant à lui est le noeud de la racine (vide).



(e) Suppression de la racine vide. Dans ce cas la hauteur de l'arbre est diminue de 1.



(f) Suppression de la lettre D dans le B-arbre précédent (**cas 3a**). Comme on ne peut pas descendre dans la récursivité (car le noeud *A B* ne contient que deux clés) mais nous ne pouvons pas fusionner les deux fils de la racine car le fils gauche contient trois clés. Les nœuds verts est le résultat d'une rotation des clés. Le noeud bleu contenait la clé supprimé.

FIGURE 12 – Suppressions dans un B-arbre.

FIGURE 13 – Les rotations à la base de l'opération SPLAY : de droite à gauche, $rotation(x)$ et de gauche à droite, $rotation(y)$.

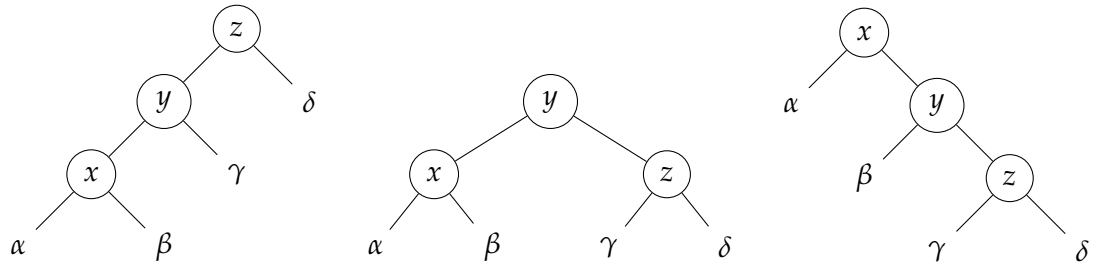


FIGURE 14 – Les rotations à la base de l'opération SPLAY : l'opération $zig - zig(x)$.

Ces opérations se base sur l'opération SPLAY(i, S) qui réorganise l'arbre S en mettant i à la racine de l'arbre S si $i \in S$ et $\max\{k \in S, k < i\}$ sinon.

— RECHERCHE(i, S) : SPLAY(i, S) et on teste la racine ;

— INSERTION(i, S) : SPLAY(i, S) et on insère i à la racine ;

— SUPPRESSION(i, S) : SPLAY(i, S), on supprime i et on applique UNION aux deux fils ;

— UNION(S, S') : SPLAY($+\infty$, S) et on insère S' comme fils droit ;

— PIVOT(i, S) : SPLAY(i, S) et on récupère les deux fils.

L'opération SPLAY se fait à l'aide d'une opération élémentaire assez classique dans les ABR (rotation, Figure 13).

Cette opération fait bien remonter x vers la racine mais il faut faire un peu plus attention pour conserver l'équilibre de l'arbre. On veut donc faire remonter le noeud x à la racine selon les trois cas suivant :

1. Si x est fils de la racine : on a $zig(x) = rotation(x)$ (Figure 13) ;
2. Si x et son père y sont tous les deux fils gauches (ou droits) : $zig - zig(x)$ est l'application successives de $rotation(y)$ ou $rotation(x)$ (Figure 14) ;
3. Si x et son père y sont des fils de côtés différents (droite-gauche ou gauche-droite) : $zig - zag(x)$ est l'application $rotation(x)$ deux fois (Figure 15).

Complexité : La hauteur de l'arbre est dans le pire cas en $O(n)$ (très peu probable), en moyenne elle est de $O(n \log n)$)

Références

[1] D. Beauquier, J. Berstel, and P. Chrétienne. *Eléments d'algorithmique*. Manuels informatiques Masson. Masson, 1992.

[2] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Algorithmique, 3ème édition*. Dunod, 2010.

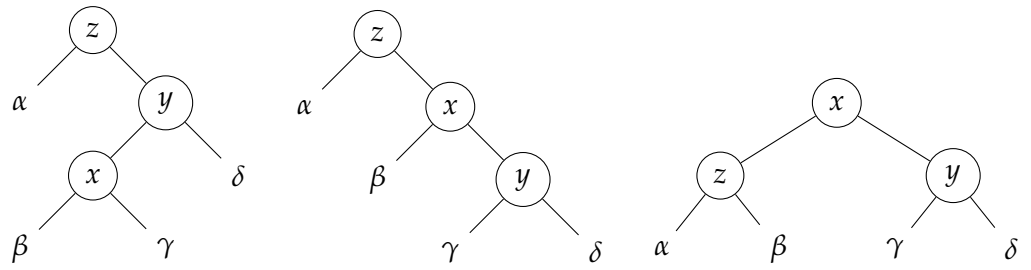


FIGURE 15 – Les rotations à la base de l'opération SPLAY : l'opération $zig - zag(x)$.

[3] C. Froidevaux, M.C. Gaudel, and M. Soria. *Types de données et algorithmes*. McGraw-Hill, 1990.