Quadtree

UE 3M101 : projet de recherche supervisé

Thiziri Baiche Charlotte Briquet Serge Durand Kevin Meetooa

5 mai 2019

Table des matières

Introduction

Les arbres, en informatique, sont une structure de donnée classique, aux nombreuses variations et applications. L'objectif de ce rapport est d'étudier et d'implémenter des quadtrees [1]. Pour bien comprendre les quadtrees, nous étudierons d'abord les arbres binaires. Après avoir introduit le vocabulaire de base et les notions spécifiques à ces arbres, nous nous intéresserons en particulier aux arbres binaires de recherche et leur implémentation. En effet, les quadtrees peuvent être vus comme une extension des arbres binaires de recherche. Nous présenterons également des mesures expérimentales de la complexité des méthodes implémentées. Puis nous présenterons les différents types de quadtrees. Enfin, nous illustrerons une application concrète d'un quadtree : la compression d'image.

1. Nous emploierons le terme "quadtree" dans l'intégralité du rapport car il est plus fréquemment utilisé que sa traduction française "arbre quaternaire"

Chapitre 1

Arbres et Arbres Binaires

1.1 Arbres

Un *arbre* est une structure de donnée hiérarchique définie par un nombre fini de nœuds. Chaque *nœud* est composé d'une *étiquette* qui représente sa valeur ou l'information associée, et d'un ensemble d'autres noeuds qui lui sont associés.

Définition 1.1.1 (arbre étiqueté)**.** *Soit E un ensemble fini. On dit qu'un arbre est étiqueté sur E si toutes ses étiquettes appartiennent à E.*

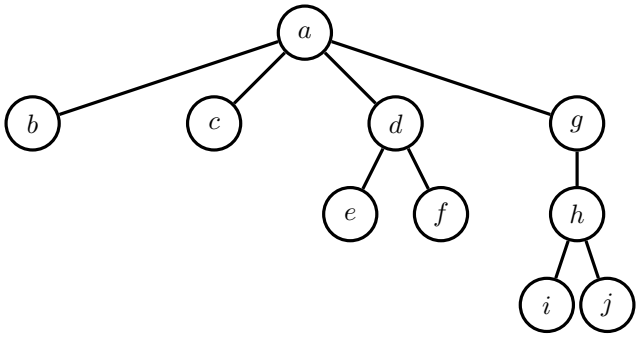


FIGURE 1.1 – Arbre A étiqueté sur $E = \{a, b, c, d, e, f, g, h, i, j\}$

On dit que chaque nœud est relié par une *branche*. On nomme des nœuds *parents*, *fils*, *frères*, *ancêtres* ou *descendants*, les nœuds d'un arbre de manière analogue à un arbre généalogique. Dans un arbre, un noeud a exactement un parent, sauf la *racine* qui n'en a aucun : c'est la particularité de cette structure et ce qui lui donne son nom.On dit qu'un noeud qui n'a aucun fils est une *feuille*. Dans la figure 1.1, la racine de l'arbre est le noeud étiqueté a, les noeuds $\{b, c, e, f, i, j\}$ sont des feuilles.

Définition 1.1.2 (degré)**.** *Soit A un arbre étiqueté sur E.Le degré D d'un nœud est le nombre de fils qu'il possède. Le degré d'un arbre correspond au degré maximal de ses noeuds.*

$$D(A) = max\{D(x), \forall x \in E\}$$

Dans la figure 1.1, le noeud b est de degré 0 ($D(b) = 0$), le noeud g est de degré 1 ($D(g) = 1$), son unique fils est h. Le degré de l'arbre A est $D(A) = 4$.

Définition 1.1.3 (taille)**.** *La taille d'un arbre A est son nombre total de noeuds. Elle est souvent notée $n(A)$.*

Définition 1.1.4 (chemin)**.** *Le chemin ch d'un noeud est une suite de noeuds qu'il faut emprunter pour parcourir l'arbre de la racine au noeud en question. On appelle la longueur L d'un chemin, le nombre de noeuds empruntés :*

$$L(ch(x)) = \#ch(x), \forall x \in E$$

Définition 1.1.5 (hauteur)**.** *La hauteur h d'un arbre A est la longueur du chemin le plus long auquel on ajoute* 1 *car on considère la racine de l'arbre comme de hauteur 1 :*

$$h(A) = max\{L(ch(x)), \forall x \in E\} + 1$$

On parle aussi de hauteur pour un noeud : c'est la longueur du chemin partant de la racine de l'arbre jusqu'à ce noeud. La hauteur d'un noeud est aussi parfois appelée profondeur. Par convention, la hauteur de l'arbre vide est 0. [1].

Dans la figure 1.1, $n(A) = 5$, $ch(f) = \{a, d, f\}$, $L(ch(f)) = 3$, $h(A) = 4$.

On définit un *sous-arbre* comme un autre arbre formé par un sous-ensemble de nœuds et de branches d'un arbre principal. En effet on peut considérer le fils d'un noeud comme la racine d'un nouvel arbre. Dans la figure 1.1, un sous-arbre de A est par exemple, l'arbre formé par le noeud g. Ce sous-arbre A_g est étiqueté sur le sous-ensemble $E_g = \{g, h, i, j\}$

1.2 Arbres Binaires

Définition 1.2.1. *Soit A un arbre. A est un arbre binaire si et seulement si $D(A) = 2$, il est de degré deux, i.e. chaque noeud a au plus deux fils.*

La figure 1.2 représente un arbre binaire, de hauteur 3, de taille 6.

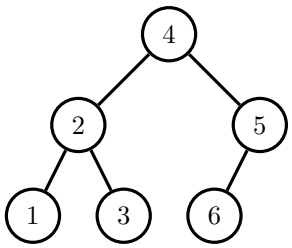


FIGURE 1.2 – Arbre binaire A étiqueté sur les entiers naturels

Un arbre binaire a la particularité que pour un noeud n'ayant qu'un seul fils, la position du fils (s'il est à droite ou à gauche) est importante. En effet, l'arbre de la figure 1.2 n'est pas identique à celui de la figure 1.3. On definit le type de ce dernier :

Définition 1.2.2 (arbres ordonnés)**.** *On dit qu'un arbre étiqueté sur un ensemble muni d'une relation d'ordre totale est ordonné si tous les noeuds de son sous-arbre gauche (resp. droit) ont une étiquette supérieure (resp. inférieure) à celle de chacun de ses enfants s'ils existent.*

Autrement dit, l'étiquette de la racine de chaque sous-arbre est comprise entre l'étiquette de la racine de son sous-arbre gauche et celle de son sous-arbre droit.

1. Il est possible de rencontrer la convention $h(\emptyset) = -1$ et hauteur de la racine $= 0$

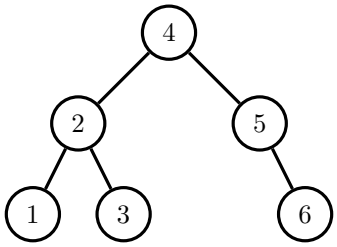


FIGURE 1.3 – Arbre ordonné A étiqueté sur les entiers naturels

L'arbre représenté par la figure 1.3 est donc un arbre ordonné.

Nous verrons dans le chapitre suivant, les arbres binaires de recherche qui sont en fait des arbres binaires ordonnés.

Définition 1.2.3 (tas)**.** *On appelle tas ou arbre tassé un arbre binaire presque complet : tous les niveaux de l'arbre binaire sont remplis, sauf peut-être le dernier qui est éventuellement rempli sur la gauche.*

On parle aussi d'arbre *parfait*. La structure de tas est notamment utilisée pour le tri par tas.

Définition 1.2.4. *On peut trouver des arbres totalement dégénérés, où tous les nœuds sont alignés, ie tous les nœuds ont au plus un fils. On appelle ces arbres, des arbres filiforme.*

Voyons une application d'arbres binaires avant de les définir formellement.

Définition 1.2.5 (arbres d'expression)**.** *Un arbre binaire d'expression est un genre d'arbre binaire utilisé pour représenter, comme son nom l'indique, des expressions. Il existe deux types d'expression qu'un arbre binaire peut représenter : algébrique et booléenne.*

Les feuilles d'un arbre binaire d'expression sont des quantités (constantes ou variables) numériques dans le cas algébrique et "vrai" (T) ou "faux" (F) dans le cas booléen. Les nœuds internes sont des opérateurs : addition ($+$), soustraction ($-$), multiplication ($\times$), division ($\div$) et puissance ($\ldots^{\cdots}$) dans le cas algébrique ou des quantificateurs : "et" ($\wedge$), "ou" ($\vee$) et la négation ($\neg$) dans le cas booléen. Voir figure 1.4 et figure 1.5. Un arbre d'expression est alors évalué en appliquant l'opérateur de la racine au valeurs obtenues en évaluant récursivement les sous-arbres de gauche et de droite, via un parcours suffixe. L'évaluation de l'arbre de la figure 1.4 donne

$$\frac{-8}{x+5} \times 4^2$$

Un arbre binaire qui ne contient pas de nœuds est appelé un arbre vide ou un arbre nul et est noté $\emptyset$. Autrement, un arbre non vide A est défini par un triplet constitué de :

- une clef (ou étiquette de la racine) x
- un sous-arbre gauche G
- un sous-arbre droit D

Les arbres G et D peuvent éventuellement être vides, on dit alors que le fils gauche ou droit est absent ou manquant.

Définition 1.2.6 (arbre binaire)**.** *La définition inductive de l'ensemble $\mathcal{AB}_E$ des arbres binaires étiquetés sur un ensemble fini E est alors :*

Base : $\emptyset \in \mathcal{AB}_E$

Induction : $\forall x \in E, G \in \mathcal{AB}_E, D \in \mathcal{AB}_E : (x, G, D) = A \in \mathcal{AB}_E$

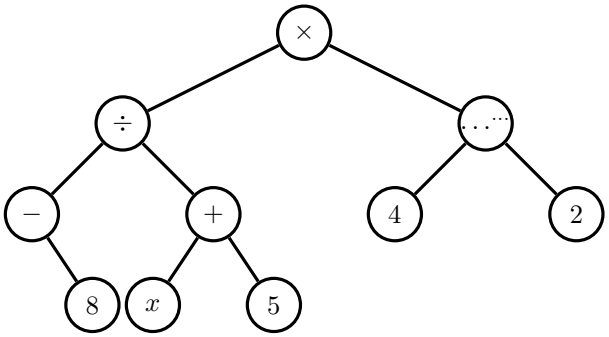


FIGURE 1.4 – Arbre binaire d'expression algébrique

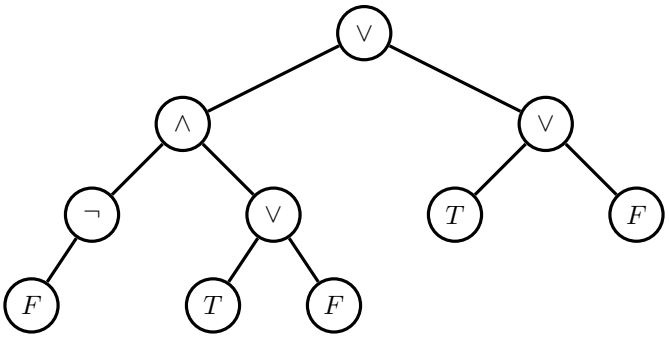


FIGURE 1.5 – Arbre binaire d'expression booléen

1.2.1 Premiers algorithmes

Définition 1.2.7 (Notations de Landau)**.** *Soient f et g des fonctions de $\mathbb{N}$ dans $\mathbb{N}$. On note :*

- $f \in \mathcal{O}(g)$ *("f est un grand O de g") si* $\exists k > 0$ *et* $\exists n_0 \geq 0$ *tels que* $\forall n \geq n_0, f(n) \leq k \cdot g(n)$
- $f \in \Omega(g)$ *("f est un grand oméga de g") si* $\exists k > 0$ *et* $\exists n_0 \geq 0$ *tels que* $\forall n \geq n_0, k \cdot g(n) \leq f(n)$.
- $f \in \Theta(g)$ *("f est un grand thêta de g") si* $\exists k_1, k_2 > 0$ *et* $\exists n_0 \geq 0$ *tels que* $\forall n \geq n_0, k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n)$.

Autrement dit : $f \in \mathcal{O}(g)$ si f est inférieure à g à partir d'un certain rang, à une constante multiplicative près. De plus on remarque que

$$f \in \mathcal{O}(g) \iff g \in \Omega(f)$$

et

$$f \in \Theta(g) \iff f \in \mathcal{O}(g) \text{ et } f \in \Omega(g)$$

Ces notations sont très fréquemment utilisées pour l'analyse de la complexité d'algorithmes.

Calcul de la hauteur et de la taille

Définition 1.2.8 (hauteur)**.** *Soient $x \in E$, G et D des arbres binaires. Soit $A = (x, G, D) \in \mathcal{AB}$. La hauteur est définie par induction comme :*

Base : *la hauteur de l'arbre vide, $h(\emptyset) = 0$*

Induction : *la hauteur de A, $h(A) = 1 + max\{h(G), h(D)\}$*

Définition 1.2.9 (taille). *Soient $x \in E$, G et D des arbres binaires. Soit $A = (x, G, D) \in \mathcal{AB}$. On définit par induction la taille :*

Base : la taille de l'arbre vide, $n(\emptyset) = 0$

Induction : la taille de A, $n(A) = 1 + n(G) + n(D)$

Suivant ces définitions, pour calculer la hauteur d'un arbre on peut utiliser la fonction récursive suivante :

```
def hauteur(self):
        """convention : arbre vide : hauteur 0
                        arbre réduit à un noeud : hauteur 1"""
        if self.clef == None:
            return 0
        if self.gauche is None and self.droit is None:
            return 1
        return max(self.gauche.hauteur(),self.droit.hauteur())+1
```

Théorème 1.2.10 (Complexité, terminaison et correction). *La fonction hauteur(A), implémentée ci-dessus, se termine en $\Theta(n)$ opérations et est correcte, pour tout $A \in \mathcal{AB}$.*

Preuve. Pour le calcul de la complexité, on compte le nombre d'addition $c(n)$ lors de l'appel hauteur(A) sur un arbre A de taille n.

Base : $h(\emptyset) = 0$ par convention.

L'appel retourne 0 directement, il n'y a aucune addition : $c(0) = 0$. La fonction se termine et est correcte.

Induction : par récurrence forte :

Supposons que $\forall k \in [\![0, n-1]\!], c(k) = k$, c'est à dire que l'appel hauteur(A) comporte k additions pour tout arbre A de hauteur k et qu'il se termine et retourne $h(A)$.

Soit $T = (x, T_1, T_2) \in \mathcal{AB}$ tel que $n(T) = n$. On pose $n_1 = n(T_1)$ et $n_2 = n(T_2)$. On a $n_1 + n_2 + 1 = n$ donc $(n_1, n_2) \in [\![0, n-1]\!]^2$.

Puisque T n'est pas vide, on distingue deux cas pour l'appel hauteur(T) provoque deux appels récursifs : hauteur(T_1) et hauteur(T_2) et renvoie 1 additionné au max des valeurs retournées par ces appels. Or par hypothèse de récurrence, les appels hauteur(T_1) et hauteur(T_2) se terminent, renvoient respectivement $h(T_1)$ et $h(T_2)$ et comportent respectivement n_1 et n_2 additions. Donc l'appel hauteur(T) se termine, renvoie $max(h(T_1), h(T_2)) + 1 = h(T)$ et comporte $n_1 + n_2 + 1 = n$ additions.

Conclusion :

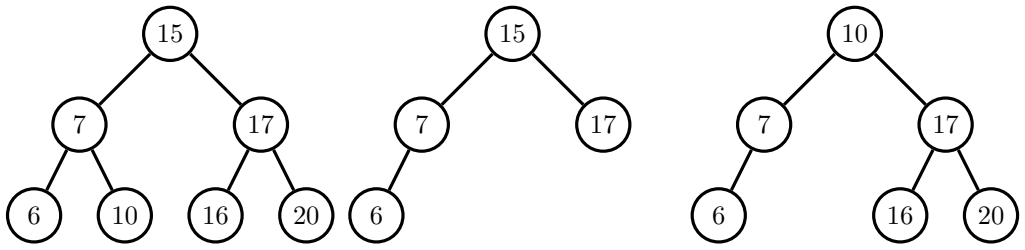
On a montré que la propriété est vraie pour $T = \emptyset$, et que si elle est vraie pour tout $A \in \mathcal{AB}$ tel que $n(A) < n$ alors elle est vraie pour tout $T \in \mathcal{AB}$ tel que $n(T) = n$. En conclusion, la propriété est vraie pour tout arbre binaire de l'ensemble $\mathcal{AB}$. □

En pratique, nous avons dû distinguer le cas où T est réduit à un seul noeud, les appels T.hauteur() ne fonctionnant pas en Python si T est None, mais cela n'a pas d'impact sur la complexité.

La méthode pour calculer la taille d'un arbre peut être implémentée de manière récursive également, avec un résultat et une preuve analogue sur la complexité, le nombre d'additions étant alors $2n$ pour un arbre de taille n.

1.2.2 Relations entre hauteur et taille

On rappelle qu'un arbre *complet* est un arbre dont tout les nœuds internes ont exactement deux fils. C'est un arbre entièrement rempli. Un arbre *parfait* est un arbre dont tout les niveaux sont remplis sauf éventuellement le dernier, en général rempli le plus à gauche. Voir définition 1.2.3. On introduit également la notion d'*arbre H-équilibré* : un arbre dont la différence de profondeur entre deux feuilles est au plus 1. Tous les niveaux sont donc remplis sauf le dernier ici aussi mais sans condition sur la répartition des feuilles du dernier niveau. Voir figure 1.6.



(a) arbre complet (b) arbre parfait (c) arbre H-équilibré

FIGURE 1.6

On remarque que tout arbre $(x, G, D) \in \mathcal{AB}$ est complet si et seulement si $G \in \mathcal{AB}$ et $D \in \mathcal{AB}$ sont complets et de même hauteur.

On a les propriétés suivantes pour les arbres complets :

Proposition 1.2.11 (Relation entre taille et hauteur)**.** *La hauteur d'un arbre complet (h) et sa taille (n) vérifie la relation suivante :*

$$2^h - 1 = n \Leftrightarrow h = log(n+1) \tag{1.1}$$

Preuve. La preuve est immédiate par récurrence sur la hauteur :

Base : $h = 0 \Leftrightarrow$ l'arbre est vide. Or $2^0 - 1 = 1 - 1 = 0$ et il y a bien 0 noeud dans l'arbre vide.

Induction : Supposons que la propriété soit vérifiée pour tout arbre binaire complet de hauteur $h-1$. Soit un arbre $(x, G, D) \in \mathcal{AB}$ complet de hauteur h et de taille n. Alors $G \in \mathcal{AB}$ et $D \in \mathcal{AB}$ sont complets de hauteur $h-1$. Alors :

$$n = n_G + n_D + 1 = 2^{h-1} - 1 + 2^{h-1} - 1 + 1 = 2 \times (2^{h-1}) - 1 = 2^h - 1$$

$$\Leftrightarrow h = log(n+1)$$

La propriété est vraie pour un arbre vide et se propage, elle est donc vérifiée pour tout arbre complet. □

Conséquences : On a

$$log(n) \leq log(n+1) \leq log(n^2) = 2log(n) \text{ donc } log(n+1) = \Theta(log(n))$$

Les algorithmes de complexité linéaire en la hauteur d'un arbre sont donc logarithmiques en la taille de l'arbre pour les arbres complets.

De plus si l'on a une certaine garantie sur la structure d'un arbre, notamment sur son équilibre, on a alors des bornes sur la complexité des fonctions. En effet si T est un arbre H-équilibré de

hauteur h, sa taille, n est comprise entre la taille d'un arbre complet de hauteur $h-1$ et celle d'un arbre complet de hauteur h. Ce qui donne $h-1 \leq log(n-1)$ et $log(n-1) \leq h$. Les algorithmes linéaires en la hauteur sont donc également logarithmiques en la taille pour les arbres H-équilibrés. Pour garantir une certaine efficacité des algorithmes, l'enjeu est donc de construire les arbres les plus équilibrés possible.

Proposition 1.2.12. *Un arbre binaire complet non vide a 2^{h-1} feuilles.*

La preuve est immédiate par récurrence sur la hauteur. On ne la détaille pas, le procédé est le même que précédemment, il suffit de remarquer que pour tout arbre complet non vide $T = (x, G, D)$, $f(T) = f(G) + f(D) = 2 \times (G)$, avec $f(T) =$ le nombre de feuilles de T. On en déduit un corollaire assez intéressant :

Corollaire 1.2.13. *Plus de la moitié des noeuds dans un arbre binaire complet sont des feuilles.*

Preuve. On calcule le rapport entre le nombre de feuilles et le nombre de noeuds. D'après la proposition précédente, ce rapport vaut :

$$\frac{2^{h-1}}{2^h - 1} \geq \frac{2^{h-1}}{2^h} = \frac{1}{2}$$

□

Théorème 1.2.14 (hauteur moyenne d'un noeud)**.** *La profondeur moyenne d'un noeud choisi au hasard est en $\Theta(log(n))$*

Preuve. Par définition d'un arbre binaire complet non vide, il y a 2^{h-1} noeuds à l'étage h de l'arbre, cela est dû au fait que chaque noeud qui n'est pas une feuille possède exactement 2 fils.

Ainsi, il y a 1 noeud à l'étage 1 (la racine), 2 noeuds à l'étage 2 (les 2 fils de la racine), 4 noeuds à l'étage 3 (les 2 fils des 2 noeuds de l'étage 1) et ainsi de suite...

Pour calculer la profondeur moyenne d'un noeud, il suffit de calculer le rapport entre la somme des profondeurs de chaque noeud de l'arbre et le nombre total de noeuds.

Ce rapport vaut :

$$\frac{\sum_{k=0}^{h-1} k \times 2^k}{2^h - 1} = \frac{2^h \times h - 2 \times (2^h - 1)}{2^h - 1} = \frac{(h-1) \times 2^h - 2^h + 2}{2^h - 1}$$

(la valeur de la somme a été calculée à l'ordinateur). Et

$$\frac{(h-1) \times (2^h - 1) + (h-1) - (2^h - 1) + 1}{2^h - 1} = h - 2 + \frac{h}{2^h - 1} \text{ avec } \lim_{h \to \infty} \frac{h}{2^h - 1} = 0$$

Donc la hauteur moyenne d'un noeud vaut $h - 2 = \Theta(log(n))$ (d'après la proposition 1.2.11)

□

Chapitre 2

Arbres Binaires de Recherche

2.1 Définition

Comme vu précédemment, un arbre binaire de recherche est un arbre binaire ordonné. C'est-à-dire qu'il est étiqueté sur un ensemble muni d'un ordre total (typiquement les réels ou les entiers) dans lequel tout nœud a une étiquette supérieure à celles de son sous-arbre gauche et inférieure à celles de son sous arbre-droit. Nous emploierons l'abréviation ABR ou BST pour *binary search tree* pour évoquer un arbre binaire de recherche.

Définition 2.1.1 (arbre binaire de recherche)**.** *La définition par induction de l'ensemble $\mathcal{ABR}_E$ des arbres binaires de recherche étiquetés sur un ensemble fini E est :*

Base : $\emptyset \in \mathcal{ABR}_\mathcal{E}$

Induction : Soient $x \in E, G \in \mathcal{ABR}_\mathcal{E}, D \in \mathcal{ABR}_\mathcal{E}$.

Si :

- $\forall (x_1, G_1, D_1) \in G, x_1 < x$
- $\forall (x_2, G_2, D_2) \in D, x_2 > x$

alors $(x, G, D) \in \mathcal{ABR}_\mathcal{E}$

2.2 Implémentation

Pour implémenter un ABR, deux méthodes s'offrent à nous : utiliser une structure récursive ou bien représenter l'arbre par une liste. Cette section à pour but de comparer ces deux méthodes.

2.2.1 Structure récursive

Une représentation qui semble la plus naturelle est d'implémenter une structure avec auto-référence. En effet en s'inspirant de la définition inductive, un arbre binaire peut être vu comme une structure contenant essentiellement 3 champs : une clef et deux arbres binaires, ses sous-arbres de gauche et de droite.

En terme d'implémentation cela dépend du langage utilisé. En C cela passera par la création d'un type spécifique "arbre" avec en variable des pointeurs vers des variables de ce même type, en plus d'une variable clef de type entier par exemple. En suivant le paradigme de la programmation objet (typiquement en Java ou en Python), nous allons créer une classe arbre ayant deux attributs de type arbre en plus de l'attribut clef (et d'autres attributs si nécessaires). La souplesse de

Python sur les types rend la création de la classe particulièrement aisée mais il faudra être vigilant quant à son utilisation.

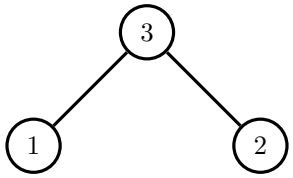
On pourra déclarer la classe comme ceci :

```
class ArbreBinaire():
    def __init__(self,clef,gauche = None ,droite = None):
        """ Anything x ArbreBinaire x ArbreBinaire -> ArbreBinaire
        Constructeur d'arbre"""
        self.clef = clef
        self.gauche = gauche
        self.droite = droite
```

Exemple d'instanciation :

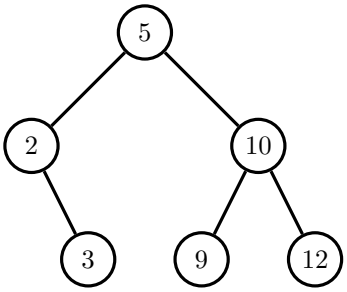
```
Feuille1 = ArbreBinaire(1)
Feuille2 = ArbreBinaire(2)
Arbre = ArbreBinaire(3,Feuille1,Feuille2)
```

L'arbre créé est :



2.2.2 Liste

On peut aussi représenter l'arbre sous forme de liste. L'idée est d'avoir la racine au début de la liste, le fils gauche à l'index 1, le fils droite à l'index 2 et ainsi de suite. Il s'agit du parcours en largeur d'un arbre (voir section 2.3.1) Pour un nœud à l'indice i, on trouve son fils gauche à l'indice $2i$ de la liste, et le droit à l'indice $2i+1$. On peut donc également retrouver le père d'un nœud facilement : pour un nœud d'index i, le père est à l'index $\lfloor \frac{i-1}{2} \rfloor$ (on prend la partie entière dans la division par 2). On fera attention à ne pas sauter les nœuds vides, il faut bien les représenter dans la liste par un caractère spécial en fonction de ce que l'arbre contient comme type de donnée. Par exemple mettre *False* pour un arbre contenant des entiers éventuellement nuls n'est pas une bonne solution, le test *False == 0* renvoyant *True* en Python (et beaucoup d'autres langages). Par exemple l'arbre suivant :



sera représenté par la liste : $[5, 2, 10, \text{"}vide\text{"}, 3, 9, 12]$

Remarquons qu'idéalement, il faudrait utiliser une structure de base qui combine les avantages de la liste chaînée et des tableaux : nous avons besoins d'accéder aux éléments à une position précise en temps constant si possible, mais la structure doit être de taille flexible pour l'agrandir ou la réduire.

2.2.3 Comparaison

Énumérons les avantages et les inconvénients de ces deux méthodes d'implémentation.
Les avantages de la structure récursive :

- lisibilité du code
- gain mémoire pour les arbres avec des nœuds vides : on n'insère que les nœuds non vide dans l'arbre
- plus grande souplesse dans le type de donnée représentée dans le nœud

Les avantages de la représentation en liste :

- gain de mémoire pour les arbres pleins ou parfait.
- simplicité de la mise en place, notamment pour les langages implémentant déjà une structure de liste / tableau (en Python on dispose par exemple déjà des primitives *append, pop, len* etc).
- accès immédiat au parent.

Les inconvénients se déduisent des avantages : la structure récursive est plus lourde à mettre en place, et peut être lourde en mémoire pour des arbres pleins. Quant à la représentation en liste, le code sera peut être moins lisible et surtout de la mémoire est gaspillée lorsque l'arbre contient des nœuds vide.

Dans certains cas, la liste est particulièrement appropriée : par exemple, pour la représentation des tas (définition 1.2.3) et le tri par tas où le but est d'ordonner une liste.

Dans la suite du rapport, nous utiliserons la représentation la plus adaptée à la situation. En outre, la représentation en structure récursive via la programmation orientée objet du langage Python sera employée plus fréquemment.

2.3 Parcours et affichage

Dans cette section, nous allons décrire différents parcours d'arbres. Voici l'arbre sur lequel nous travaillons :

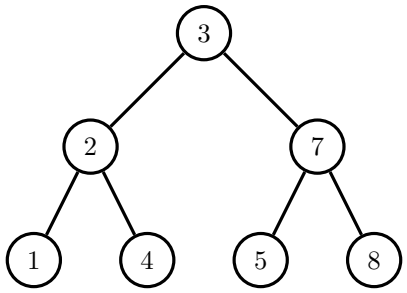


FIGURE 2.1 – Arbre binaire de recherche A

Pour réaliser les différents parcours décrits ci-dessous, on peut définir un arbre par un dictionnaire qui associe chaque nœud à ses fils. Par exemple, l'arbre A est définit par :

```
A = {3:[2, 7], 2:[1, 4], 7:[5, 8], 1:[], 4:[], 5:[], 8:[]}
```

C'est cette méthode qui est utilisée pour décrire un arbre dans le cas du parcours en largeur. Cependant, on peut également définir l'arbre A grâce à la classe ArbreBinaire définie plus haut :

```
A = ArbreBinaire(3)
A.gauche = ArbreBinaire(2)
A.droite = ArbreBinaire(7)
A.gauche.gauche = ArbreBinaire(1)
A.gauche.droite = ArbreBinaire(4)
A.droite.gauche = ArbreBinaire(5)
A.droite.droite = ArbreBinaire(8)
```

On utilisera cette implémentation pour les parcours en profondeur.

2.3.1 Parcours en largeur

Le principe d'un parcours en largeur est de lister les nœuds de l'arbre niveau par niveau en commençant par les nœuds de niveau 1 puis les nœuds de niveau 2 et ainsi de suite. Dans chaque niveau, les nœuds sont parcourus de gauche à droite. Le parcours en largeur de l'arbre étudié (figure 2.1 est donc $[3, 2, 7, 1, 4, 5, 8]$. Contrairement aux parcours suivant, le parcours en largeur s'implémente naturellement par un algorithme itératif. L'algorithme d'un tel parcours se fait à l'aide d'une file (premier entré, premier sorti). On enfile d'abord la racine puis on utilise une boucle tant que la file n'est pas vide. On défile la racine que l'on traite (typiquement par un affichage mais cela peut être un autre traitement), on enfile les fils de la racine en commençant par le fils gauche et on répète la boucle. Et ainsi de suite jusqu'à ce que la file soit vide.

```
def parcours_largeur(arbre, racine):
    liste = [racine]
    parcours = [racine]
    while liste :
        x = liste.pop(0)
        for fils in arbre[x]:
            if fils in liste :
                continue
            parcours.append(fils)
            liste.append(fils)
    return parcours
```

Pour les arbres la complexité d'un tel parcours est en $\Theta(n)$: on fait exactement n tours de boucle (on rappelle que n représente la taille de l'arbre, ie la quantité de nœuds qu'il contient). Remarquons que cet algorithme peut être utilisé sur n'importe quel type d'arbre, et même sur des graphes en général, avec quelques modifications pour ne pas repasser sur le même nœud et avoir une boucle infinie dans le cas de graphe cyclique par exemple.

2.3.2 Parcours en profondeur

Le principe d'un parcours en profondeur est de lister les nœuds de l'arbre récursivement à partir de la racine puis les sous-arbres gauches et droits de cette racine et ainsi de suite pour

la totalité de l'arbre. Il existe plusieurs types de parcours en profondeur : Infixe, Suffixe (ou Postfixe) et Préfixe.

Infixe Le parcours infixe consiste à lister les nœuds en partant du sous-arbre gauche puis remonter à sa racine et enfin parcourir le sous-arbre droit. Le parcours infixe de l'arbre A est donc $[1, 2, 4, 3, 5, 7, 8]$.

```
def parcours_infixe(arbre):
    if arbre:
        parcours_infixe(arbre.gauche)
        print(arbre.clef)
        parcours_infixe(arbre.droite)
```

Suffixe Le parcours suffixe consiste quant à lui à lister les nœuds depuis le sous-arbre gauche puis le sous-arbre droit et enfin remonter à la racine. Le parcours suffixe de l'arbre A est donc $[1, 4, 2, 5, 8, 7, 3]$.

```
def parcours_suffixe(arbre):
     if arbre:
        parcours_suffixe(arbre.gauche)
        parcours_suffixe(arbre.droite)
        print(arbre.clef)
```

Préfixe Le parcours préfixe, finalement, consiste à lister les nœuds en commençant par la racine puis le sous-arbre gauche et enfin le sous-arbre droit. Le parcours préfixe de l'arbre A est donc $[3, 2, 1, 4, 7, 5, 8]$.

```
def parcours_prefixe(arbre):
    if arbre:
        print(arbre.clef)
        parcours_prefixe(arbre.gauche)
        parcours_prefixe(arbre.droite)
```

Dans les trois cas la complexité est également en $\Theta(n)$, on passe exactement une fois sur chaque nœud. Ces parcours sont applicables aux arbres binaires en général, de recherche ou non, comme par exemple pour évaluer un arbre d'expression arithmétique.

2.4 Méthodes principales spécifiques aux ABR

2.4.1 Recherche

La recherche se fait de manière récursive. L'algorithme est assez simple, il suffit de comparer la valeur qu'on recherche avec la clef de la racine et voir sur quel côté continuer la recherche. Il y a 4 cas à traiter :

1. la valeur recherchée se trouve à la racine, dans ce cas on renvoie directement la racine
2. la valeur recherchée est plus petite que la racine ce qui signifie qu'elle est forcément dans le sous-arbre gauche, on fait donc un appel récursif sur ce dernier

3. la valeur recherchée est plus grande que la racine, on fait donc un appel récursif sur le sous-arbre de droite
4. et finalement si on atteint une feuille et que sa clé n'est pas l'élément recherché alors l'élément ne se trouve pas dans l'arbre

On crée alors la fonction récursive suivante :

```
    def recherche(self,x):
        if self.clef == None:
            return False

        if x==clef:
            return self

        if x<self.clef:
            if self.gauche is not None:
                return self.gauche.recherche(x)
            else :
                return False
        elif x > self.clef:
            if self.droit is not None:
                return self.droit.recherche(x)
            else:
                return False
```

Nous rédigeons une preuve détaillée pour la complexité , la terminaison et la correction de cette fonction.

Complexité Le nombre dopérations de la recherche dépend de la hauteur de larbre. Nous étudions ici le nombre de comparaisons :

- Meilleur cas : l'élément cherché est à la racine. La fonction fait un retour immédiat, l'opération est en temps constant (1 comparaison)
- Pire cas : l'élément cherché est dans la feuille de plus grande profondeur (ie de profondeur h = la hauteur de l'arbre). On fait exactement $h-1$ appels récursifs après l'appel initial donc h comparaisons, jusqu'à arriver à la feuille

La recherche est donc en $\Omega(1)$ et $\mathcal{O}(h)$, c'est-à-dire linéaire en la hauteur dans le pire cas. Nous avons vu que $h = \Theta(log(n))$ pour les arbres complets ou équilibrés. Dans ce cas la recherche est donc en $\mathcal{O}(log(n))$. Elle est en $\mathcal{O}(n)$ dans le cas d'arbres filiformes (voir définition 1.2.4). Nous étudierons plus finement la complexité moyenne en fonction de la taille par la suite.

Terminaison et Correction Montrons que l'appel $T.recherche(x)$ se termine et est correct pour tout arbre de recherche T étiqueté sur un ensemble E et pour tout $x \in E$. On fait une preuve par induction sur la hauteur.

Base : Si $h = 0$, alors l'arbre est vide et la fonction renvoie $False$ quelque soit x, ce qui est le résultat attendu.

Induction : Supposons que la fonction se termine et est correcte pour tout arbre de hauteur $p \in [\![0, h]\!]$ et pour tout élément $x \in E$.

Soit $T = (y, G, D) \in \mathcal{ABR}$ un arbre étiqueté sur E de hauteur $h+1$ et soit $x \in E$.

Lors de l'appel $T.recherche(x)$, on distingue trois cas :

1. $y = x$ alors la fonction se termine et renvoie $True$, ce qui est le retour valide.
2. $x < y$ on a alors un appel récursif $G.recherche(x)$. Or on sait que x ne peut pas être dans D car T est un arbre binaire de recherche. De plus par hypothèse de récurrence, l'appel

$G.recherche(x)$ se termine et renvoie $True$ si x est dans G, $False$ sinon. Donc l'appel $T.recherche(x)$ se termine et est valide.

3. $x > y$ dans ce cas, on sait que x ne peut pas être dans G, puis que $D.recherche(x)$ se termine et est valide : l'appel $T.recherche(x)$ se termine et est valide également.

Dans les trois cas l'appel $T.recherche(x)$ se termine et est valide.

Conclusion : On a montré que l'appel $T.recherche(x)$ se termine et est valide pour l'arbre vide puis que s'il est valide et se termine pour tout arbre de hauteur $p \leq h$ alors il est valide et se termine pour tout arbre de hauteur $h+1$, donc l'appel $T.recherche(x)$ se termine et est valide pour tout arbre binaire de recherche.

Remarque : Le cas de l'arbre vide peut se rencontrer de deux manières, un appel directement sur un arbre vide et à la fin de la récursion lorsqu'on arrive aux feuilles. Les deux cas sont distincts dans notre implémentation. En effet, si T est de type $None$ on ne peut pas faire l'appel $T.recherche(x)$, c'est pour cela que l'on fait une vérification avant de faire les appels récursifs. Cependant si T a été initialisé comme un arbre contenant une seul clef vide ($T = Arbre(None)$), l'appel ne provoque pas de problème pour Python et il faut prévoir ce cas, c'est le but de la ligne 2.

Pour les méthodes suivantes la complexité, la terminaison et la correction peuvent être prouvées de manière tout à fait similaire, nous ne détaillons pas ces preuves.

2.4.2 Insertion

Pour l'insertion, on procède de manière analogue à la recherche : pour insérer un élément dans un arbre, il faut savoir dans quelle partie de l'arbre il doit appartenir pour que l'arbre reste un arbre binaire de recherche. Pour cela il suffit de comparer la valeur à insérer au nœud courant. On distingue trois cas :

1. les valeurs sont égales : on fait un retour, il n'y a rien à insérer. On rappelle que dans un arbre binaire de recherche, les étiquettes des nœuds sont toutes distinctes.
2. la valeur à insérer est plus petite que le nœud courant. On a alors deux possibilités :
   - le sous-arbre gauche est vide : on procède à l'insertion en créant un nœud avec pour étiquette la valeur à insérer, et ses sous-arbres gauche et droit vides.
   - le sous-arbre gauche n'est pas vide : on fait un appel récursif sur cet arbre.
3. si elle est plus grande on procède comme dans le cas précédent sur le sous-arbre droit.

On remarque que dans cette version, on insère toujours au niveau des feuilles, si insertion il y a. L'algorithme est simple, mais il n'est pas optimal sur la structure de l'arbre. On ne cherche pas à garantir une structure équilibrée. Nous verrons par la suite qu'il est possible de le faire en introduisant la rotation d'arbre (voir section 3.1). La complexité est en $\mathcal{O}(h)$ avec h la hauteur de l'arbre, comme pour la recherche. La preuve est analogue.

2.4.3 Suppression

La fonction de suppression se fait de manière récursive. Elle est assez simple mais il faut faire attention à ne pas perdre des informations ou à modifier la structure d'arbre, ici aussi on distingue 3 cas. Soit x l'élément que l'on veut supprimer :

1. si x est la racine de notre arbre, c'est assez subtil. Il faut à nouveau traiter trois cas :
   - si x n'a pas de fils, on le supprime directement.
   - si x a un seul fils, on remplace x par son unique fils.

- si x a deux fils, on échange x soit par le nœud maximal du sous-arbre gauche (qui correspond au fils le plus à droite du sous-arbre gauche), soit par le nœud minimal du sous-arbre droit (qui correspond au fils le plus à gauche du sous-arbre droit), puis on supprime le nœud maximal ou minimal en faisant un appel récursif. Pour notre implémentation on choisit le nœud minimal du sous-arbre gauche pour remplacer x.

2. si x est plus petit que la racine, on fait un appel récursif sur le sous-arbre gauche
3. si x est plus grand, on fait un appel récursif sur le sous-arbre droit.

La figure 2.2 représente les étapes d'une suppression du nœud racine.

Une suppression comporte donc un parcours d'une unique branche, un éventuel échange de valeur, puis la suppression d'un nœud possédant au maximum 1 sous arbre. Les deux dernières opérations peuvent être implémentées en $\Theta(1)$. La suppression a donc, comme les autres opérations, un coût maximal en $\mathcal{O}(h)$.

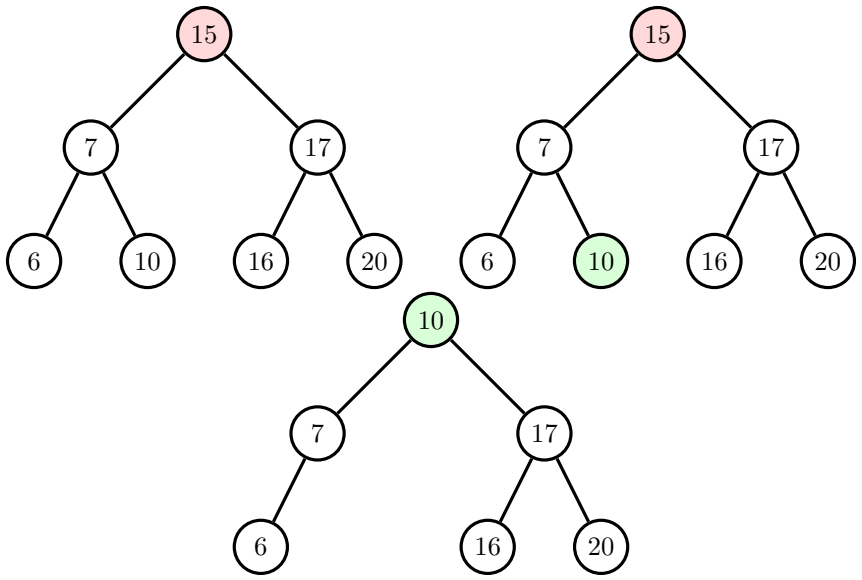


FIGURE 2.2 – Exemple de suppression de la racine : nœud 15

2.5 Hauteur moyenne et autres résultats sur la structure des ABR

Nous avons vu que les complexités des méthodes de recherche, d'insertion et de suppression, pour un arbre A de hauteur h sont en $\mathcal{O}(h)$. Ces méthodes sur une structure de type tableau de taille n sont en $\mathcal{O}(n)$, le cas le plus défavorable étant atteint lorsque l'on doit comparer avec tout les éléments du tableau. L'arbre binaire de recherche n'a donc d'intérêt que si sa hauteur est significativement plus faible que sa taille. Pour les arbres complets, ou quasi-complets, c'est le cas puisque $h = \mathcal{O}(log(n))$. Mais qu'en est-il pour des arbres généré aléatoirement ?

Théorème 2.5.1 (borne sur la hauteur moyenne en fonction de la taille)**.** *Soit $n > 0$, on note $\mathcal{ABR}_n$ l'ensemble des arbres binaire de recherche de taille n, (ou de manière équivalente l'ensemble des ABR étiquetés sur $\{1, ..., n\}$) et X_n la variable aléatoire de $\mathcal{ABR}_\backslash$ dans $\mathbb{N}^*$ qui*

à un arbre de ABR_n associe sa hauteur. $(X_n)_{n\in\mathbb{N}^*}$ est donc une suite de variables aléatoires discrètes. Soit $E(X_n)$ l'espérance de X_n, on a :

$$E(X_n) = \mathcal{O}(log(n)).$$

Ou encore pour n assez grand la hauteur d'un arbre de taille n est inférieure à $Klog(n)$ avec $K > 0$ constant.

Preuve. [1] On reprend les notations du théorème et on pose $Y_n = 2^{X_n}$. Soit $T \in \mathcal{ABR}_n$, note $T = (x, G, D)$. Soit $i \in \{1, ...n\}$ fixé. On a :

$$\mathbb{P}(\{x = i\}) = \frac{1}{n}.$$

Il y a en effet n choix possibles pour la valeur de la racine.

La taille de G, notée $n(G)$ est donc $i - 1$, et $n(D) = n - i$. Alors, puisque $h(T) = 1 + max\{h(G), h(D)\}$, on a :

$$Y_n = 2max\{Y_{i-1}, Y_{n-i}\}$$

Puis :

$$\begin{aligned} E(Y_n) &= E[2\max(Y_{i-1}, Y_{n-i})] \\ &= 2\sum_{i=1}^{n} \mathbb{P}(i)E[\max(Y_{i-1}, Y_{n-i})] \\ &= \frac{2}{n}\sum_{i=1}^{n} E[\max(Y_{i-1}, Y_{n-i})] \end{aligned}$$

Mais $E[\max(Y_{i-1}, Y_{n-i})] \leq E(Y_{i-1}) + E(Y_{n-i})$. Donc :

$$\begin{aligned} E(Y_n) &\leq \frac{2}{n}\sum_{i=1}^{n} E(Y_{i-1}) + \frac{2}{n}\sum_{i=1}^{n} E(Y_{n-i}) \\ &= \frac{2}{n}\sum_{i=0}^{n-1} E(Y_i) + \frac{2}{n}\sum_{i=0}^{n-1} E(Y_i) \\ &= \frac{4}{n}\sum_{i=0}^{n-1} E(Y_i) \end{aligned} \tag{2.1}$$

Puis on montre par récurrence :

$$E(Y_n) \leq \frac{1}{4}\binom{n+3}{3} \tag{2.2}$$

Avec $\binom{n+3}{3} = \frac{(n+3)!}{3!n!}$ le coefficient binomial 3 parmi n+3.

Pour $n = 1$ la seule hauteur possible est 1 et on a $\frac{1}{4}\binom{4}{3} = 1$, l'égalité 2.2 : est donc vérifiée.

Supposons l'égalité vérifiée pour tout $k \in \{0, ...n - 1\}$.

1. La preuve est issue en grande partie de [CLRS09] p.300

On a

$$E[Y_n] \leq \frac{4}{n} \sum_{i=0}^{n-1} E[Y_i] \quad \text{(résultat 2.1)}$$

Donc

$$E[Y_n] \leq \frac{1}{n} \sum_{i=0}^{n-1} \binom{i+3}{3} \quad \text{hypothèse de récurrence.}$$

Montrons que

$$\sum_{i=0}^{n-1} \binom{i+3}{3} = \binom{n+3}{4}. \tag{2.3}$$

En effet :

$$\begin{aligned}
\sum_{i=0}^{n-1} \binom{i+3}{3} &= \sum_{i=0}^{n-1} \frac{(i+3)(i+2)(i+1)}{6} \\
&= \frac{1}{6} \sum_{i=0}^{n-1} i^3 + 6i^2 + 11i + 6 \\
&= \frac{1}{6} \left(\frac{(n-1)^2 n^2}{4} + \frac{6(n-1)n(2n-1)}{6} + \frac{11n(n-1)}{2} + 6n \right) \\
&= \frac{n(n+1)(n+2)(n+3)}{24} \\
&= \binom{n+3}{4}.
\end{aligned}$$

Et alors on obtient :

$$\begin{aligned}
E[Y_n] &\leq \frac{1}{n} \binom{n+3}{4} \\
&= \frac{1}{n} \cdot \frac{(n+3)!}{4!(n-1)!} \\
&= \frac{1}{4} \cdot \frac{(n+3)!}{3!n!} \\
&= \frac{1}{4} \cdot \binom{n+3}{3}.
\end{aligned}$$

Nous avons donc une borne supérieure pour Y_n. Mais par convexité de $x \mapsto 2^x$ et l'inégalité de Jensen on a :

$$2^{E[X_n]} \leq E[2^{X_n}] = E[Y_n].$$

D'où

$$\begin{aligned}
2^{E[X_n]} &\leq \frac{1}{4} \cdot \binom{n+3}{3} \\
&= \frac{1}{4} \cdot \frac{(n+3)(n+2)(n+1)}{6} \\
&= \mathcal{O}(n^3)
\end{aligned}$$

Puis en passant au logarithme des deux côtés on obtient :

$$E[X_n] = \mathcal{O}(log(n)). \tag{2.4}$$

Ce que l'on voulait démontrer. □

Il y a un résultat plus précis, et plus dur à montrer, sur la valeur asymptotique de $E(X_n)$, et celle de $Var(X_n)$, la variance de la variable aléatoire X_n : la hauteur moyenne d'un arbre binaire de recherche tend en probabilité vers $\approx 4.31 \cdot ln(n)$ et la variance de la hauteur tend vers 0 lorsque n tend vers l'infini. Plus précisément :

Théorème 2.5.2 (limite de la hauteur moyenne et variance)**.** *Soit $n > 0$, on note $\mathcal{ABR}_n$ l'ensemble des arbres binaire de recherche de taille n, (ou de manière équivalente l'ensemble des ABR étiquetés sur $\{1, ..., n\}$) et X_n la variable aléatoire de $\mathcal{ABR}_\setminus$ dans $\mathbb{N}^*$ qui à un arbre de ABR_n associe sa hauteur. Soit $E(X_n)$ l'espérance de X_n, et $Var(X_n)$ sa variance. On a :*

$$\frac{E(X_n)}{ln(n)} \xrightarrow[n\to\infty]{\mathbb{P}} \alpha. \tag{2.5}$$

Où $\alpha \approx 4.31107$ est l'unique solution supérieure à 2 de $x \cdot ln(\frac{2e}{x}) = 1$.
Et

$$Var(X_n) = \mathcal{O}(1). \tag{2.6}$$

Preuve. Les preuves font appel à des notions trop avancées pour les auteurs de ce rapport. On renvoie le lecteur à [Dev86] pour la preuve de 2.5 et [Ree03] ainsi que [Drm03] pour celle de 2.6. □

Ces résultats montrent que si les méthodes classiques peuvent nécessiter de l'ordre de n opérations dans les cas les plus défavorables, en moyenne elle seront de l'ordre de $log(n)$ opération, avec une variance d'autant plus faible que n est grand, et les ABR ont donc bien des avantages pratiques.

Si toutefois on a besoin de garanties sur la structure de l'arbre, notamment que la hauteur d'un arbre soit logarithmique en sa taille dans tout les cas, il existe des structures de données basées sur les arbres binaires de recherche qui permettent de maintenir un arbre équilibré à chaque manipulation. Nous avons choisi ici d'implémenter les AVL.

Chapitre 3

AVL

Les AVL sont un type d'arbre binaire de recherche, créés par G. Adelson-Velskii et E. M. Landis [AVL62]. L'idée est de rééquilibrer l'arbre à chaque opération changeant sa structure : l'insertion et la suppression. L'opération de base permettant le rééquilibrage de l'arbre est la rotation.

3.1 Rotations

L'image 3.1 illustre le principe des rotations.

La rotation permet de remonter un noeud et d'en descendre un autre, sans changer l'ordre des éléments de l'arbre. Dans l'image 3.1 le parcours préfixe est A, P, B, Q, C dans les deux cas. Le principe est simple : il suffit d'échanger les noeuds concernés à l'aide d'un pivot.

Le pseudo-code d'un algorithme de rotation gauche est (tiré de [Wik19b]) :

Algorithme 1 rotationGauche(T)

$pivot = T.droit$
$T.droit = pivot.gauche$
$pivot.gauche = T$
$T = pivot$

La rotation droite fonctionne de manière symétrique.

La complexité d'une telle opération est constante : elle ne contient aucune boucle, aucun appel récursif, uniquement quatre assignations de variables.

3.2 Rééquilibrage

Définition 3.2.1 (coefficient d'équilibre)**.** *Soit (x, G, D) un arbre binaire. On définit son coefficient d'équilibre comme la quantité $h(D) - h(G)$.*

On rappelle qu'un arbre est dit équilibré si son coefficient d'équilibre est compris entre -1 et 1 inclus.

Nous aurons souvent besoin de calculer ce coefficient. Nous avons vu (1.2.10) que le calcul de la hauteur d'un arbre de taille n nécessite n opérations. Pour ne pas alourdir la complexité des méthodes, nous ajoutons un attribut *hauteur* à nos arbres, de sorte que l'on ait accès à la

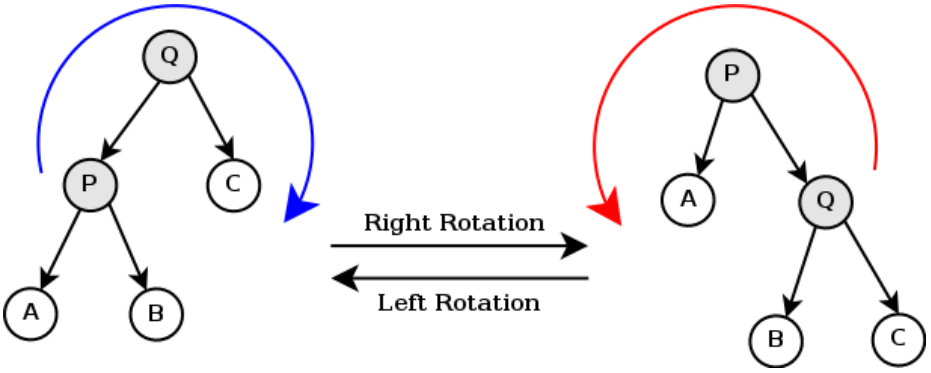


FIGURE 3.1 – Les deux rotations simples d'arbre binaire

Source : Wikipedia : Ramasamy [CC BY-SA 3.0 (http ://creativecommons.org/licenses/by-sa/3.0/)]

hauteur d'un noeud en temps constant. Le calcul du coefficient d'équilibre est donc en temps constant.

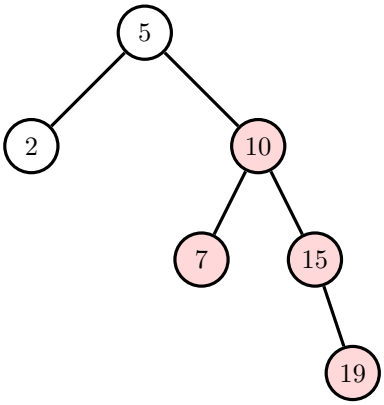
Grâce aux rotations, nous implémentons ensuite une fonction qui rééquilibre un arbre quasi-équilibré. Par quasi-équilibré, on désigne un arbre dont les sous-arbres sont eux-mêmes équilibrés et tel que son coefficient d'équilibre est compris entre -2 et 2.

Nous illustrons la procédure dans le cas où le déséquilibre est du côté droit. Soit $T = (x, G, D)$ un arbre tel que $h(D) > h(G)$. Il faut distinguer deux cas en fonction de la structure de D.

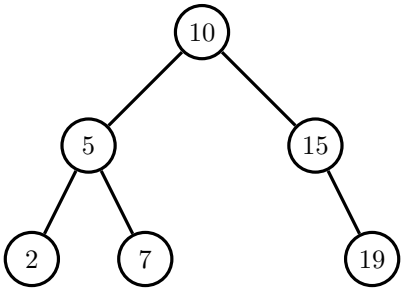
cas 1 : Dans le cas 1, le sous-arbre droit de D est au moins aussi grand (en hauteur) que le sous-arbre gauche de D. Ce cas est illustré sur la figure 3.2, les nœuds de D sont en rouge. Il suffit alors d'une simple rotation gauche pour équilibrer l'arbre. Dans 3.2, le sous-arbre droit de D est même plus grand que le sous-arbre gauche de D, et l'arbre obtenu a un coefficient d'équilibre nul. Si les deux sous-arbres de D ont la même hauteur, l'arbre reste équilibré après une simple rotation, le coefficient d'équilibre étant alors -1. Dans 3.2, cela correspond à la situation où le nœud 7 aurait un ou deux fils. Après la rotation le noeud 7 garde ses fils.

cas 2 : Dans le cas 2, le sous-arbre droit de D est strictement plus petit que le sous-arbre gauche de D. Une rotation simple ne suffit pas, comme l'illustre la figure 3.3b. Dans ce cas la solution est d'abord de faire une rotation droite uniquement sur D, pour se ramener alors au cas 1 et finir par une rotation gauche.

Le cas du déséquilibre du côté gauche est traité de la même façon. Avec ces principes, nous avons implémenté une fonction rééquilibrage, qui faisant appel aux routines $coeffEquilibre(T)$, $rotationGauche(T)$ et $rotationDroite(T)$, le pseudo-code est : 2.

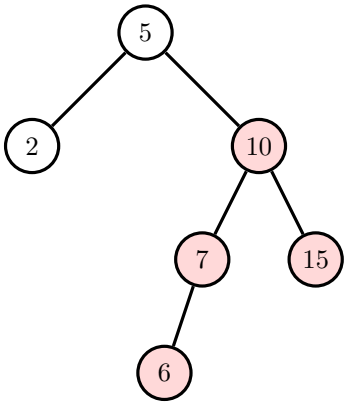


(a) arbre avant rotation

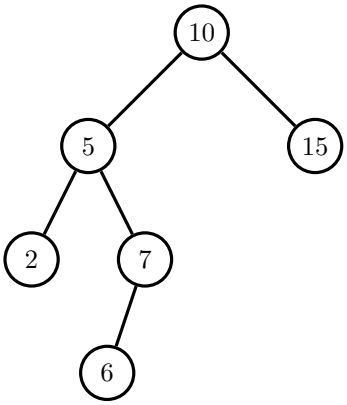


(b) arbre après rotation

FIGURE 3.2 – cas 1



(a) arbre avant rotation



(b) arbre après rotation gauche directe

FIGURE 3.3 – cas 2

Algorithme 2 reequilibrage(T)

if $|coeffEquilibre(T)| \leq 1$ **then**
  return T
else if $coeffEquilibre(T) > 1$ **then**
  if $coeffEquilibre(T.droit) \geq 0$ **then**
    return $rotationGauche(T)$
  else
    $T.droit = rotationDroite(T.droit)$
    return $rotationGauche(T)$
  end if
else
  if $coeffEquilibre(T.gauche) \leq 0$ **then**
    return $rotationDroite(T)$
  else
    $T.gauche = rotationGauche(T.gauche)$
    return $rotationDroite(T)$
  end if
end if

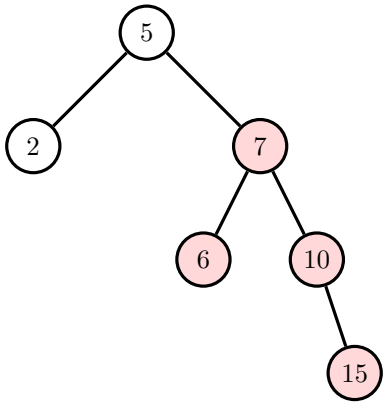


FIGURE 3.4 – cas 2 : arbre après rotation droite de D

Cet algorithme effectue un nombre constant d'opération : il fait au plus deux rotations qui sont elles mêmes en temps constant.

3.3 Insertion et suppression

Remarquons déjà que la recherche fonctionne exactement comme pour un ABR classique puisqu'elle n'induit aucune modification de la structure de l'arbre.

Pour implémenter l'insertion, l'idée est donc de d'abord procéder à l'insertion comme pour un arbre classique, puis de rééquilibrer chaque nœud traversé.

Pour cela deux approches sont possibles : on rajoute un attribut *parent* aux noeuds dans l'implémentation, ce qui a un coût pour l'espace mémoire, ou bien on peut implémenter une version itérative de l'insertion et conserver tous les noeuds traversés dans une liste. C'est cette deuxième approche que l'on choisit.

L'idée est de procéder en trois étapes :

1. On identifie la position du nœud dans lequel on va devoir insérer l'élément à l'aide d'une boucle *while* et d'un nœud temporaire : c'est la version itérative de la recherche. Tant que le nœud temporaire n'est pas vide, on continue à parcourir l'arbre vers la gauche ou vers la droite en fonction de la valeur du nœud courant et de l'élément à insérer. On stocke au passage tous les noeuds ainsi traversés dans une liste de type LIFO (dernier entré, premier sorti). Quand une feuille a été atteinte, on sort de la boucle et on obtient donc le dernier noeud non vide, que l'on note n_i
2. on procède alors à l'insertion : l'élément x est inséré dans n_i, on met à jour la hauteur de n_i.
3. On parcourt ensuite la liste des parents obtenue à la première étape en la vidant au passage. Pour chaque parent, on fait d'abord une mise à jour de la hauteur via la formule classique 1.2.8 puis on fait le rééquilibrage grâce à la fonction rééquilibrage (2). Puisque la liste utilisée est de type LIFO, on commence bien par le sous-arbre le plus profond pour finir à la racine de l'arbre.

Voilà notre implémentation de cette fonction :

```
def insertion(self,x):
        T_x = ArbreAVL(x)
        if self.clef is None:
            self.clef = x
            return self

        else:
            temp = self
            parents = []
            while(temp is not None):
                parents.append(temp)
                pere_insertion = temp
                if T_x.clef == temp.clef:
                    return self
                if T_x.clef > temp.clef:
                    temp = temp.droit
                else:
                    temp = temp.gauche
            #on fait l'insertion, en mettant à jour la hauteur :
            if T_x.clef > pere_insertion.clef :
                pere_insertion.droit = T_x
                if pere_insertion.gauche is None:
                    pere_insertion.hauteur = 1
            else:
                pere_insertion.gauche = T_x
                if pere_insertion.droit is None:
                    pere_insertion.hauteur = 1

            #on repasse sur tout les parents pour rééquilibrer:
            while(len(parents)>0):
                y = parents.pop()
                y.hauteur = y.updateHauteur()
                y = y.reequilibrageNoeud()
            return y
```

La fonction $updateHauteur()$ est un simple calcul de hauteur basée sur la def :hauteur par induction, en temps constant grâce à l'accès direct à la hauteur des sous-arbres gauche et droit.

La complexité de l'insertion est donc toujours en $\mathcal{O}(h)$, pour un arbre de hauteur h.

En effet, dans la première boucle *while*, on fait au maximum h comparaisons, dans le cas où l'élément doit être inséré dans le nœud le plus profond. La deuxième étape n'est qu'une opération d'affectation. La troisième étape consiste à faire autant de rééquilibrage que de nœuds parcourus, c'est-à-dire h rééquilibrages au plus. Les rééquilibrages étant effectués en temps constant, on obtient une complexité finale de $2\mathcal{O}(h) = \mathcal{O}(h)$ dans le pire des cas.

Pour la suppression, l'idée de l'algorithme est identique. Les étapes 1 et 3 sont les mêmes, seule l'étape 2 diffère : on supprime le noeud qui contient la feuille comme pour un ABR classique, avec les mêmes distinctions de cas vues en 2.4.3. Cette méthode a également une complexité en $\mathcal{O}(h)$

Même si la complexité est de l'ordre de la hauteur, comme pour les ABR, on remarque que pour ces deux méthodes il peut y avoir un certain nombre d'opérations supplémentaires par rapport à un ABR classique : les nœuds sur le chemin jusqu'au nœud concerné doivent être parcourus deux fois et on doit faire jusqu'à deux rotations pour chacun de ces noeuds.

3.4 Relations entre hauteur et taille

Nous avons vu que le coût de l'insertion et la suppression peut être plus élevé pour les AVL que les ABR. Que gagne-t-on alors avec cette nouvelle structure ?

Comme nous l'avons vu pour tout arbre AVL $T = (x, G, D)$ on a $|h(D) - h(G)| \leq 1$. En fait, cette propriété d'équilibre de l'arbre permet d'avoir un encadrement de la hauteur d'un AVL, valable dans tout les cas, même pour si les éléments sont insérés en ordre décroissant.

Théorème 3.4.1 (hauteur d'un AVL)**.** *Soit $\mathcal{AVL}_n$ l'ensemble des arbres AVL étiquetés sur $\{1, ..., n\}$. Pour tout $T \in \mathcal{AVL}_n$ on a :*

$$h(T) \leq 1,44 \cdot log(n). \tag{3.1}$$

Preuve. [1] On commence par renverser le problème : plutôt que s'intéresser à la hauteur maximale d'un arbre de taille n donnée, on étudie la taille minimale (c'est-à-dire le nombre minimal de noeuds) d'un arbre de hauteur h fixée.

Soit $T_h = (x, G, D)$ l'arbre de hauteur h de taille $n(T_h)$ minimale pour cette hauteur. Alors, sans perte de généralité, on peut supposer $h(G) = h - 1$ et $h(D) = h - 2$, avec G et D également de taille minimale pour un arbre AVL de hauteur $h-1$ et $h-2$. En notant $G = T_{h-1}$ et $D = T_{h-2}$, la taille de T_h vérifie la relation de récurrence :

$$\begin{aligned} & n(T_h) = n(T_{h-1}) + n(T_{h-2}) + 1 \\ \iff \quad & n(T_h) + 1 = n(T_{h-1}) + 1 + n(T_{h-2}) + 1 \end{aligned} \tag{3.2}$$

De plus $n(T_0) = 0$, $n(T_1) = 1$, $n(T_2) = 2$.

On remarque la ressemblance avec la suite de Fibonacci : pour $h > 0$ le nombre $n(T_h) + 1$ est le (h+3)-ième entier de la suite de Fibonacci. La résolution de la récurrence, après simplification puisque l'on cherche juste une borne inférieure, donne :

$$n(T_h) \geq \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^{h+3}. \tag{3.3}$$

Puis en prenant le logarithme et en exprimant h en fonction de $n(T_h)$ on obtient :

$$h(T) \leq 1,44 log(n(T_h)). \tag{3.4}$$

□

Les arbres AVL ont donc bien un intérêt pratique : ils garantissent une complexité en $\mathcal{O}(log(n))$ dans tous les cas, y compris le plus défavorable, et pas seulement en moyenne.

Nous faisons dans le chapitre suivant une étude expérimentale comparative sur nos implémentations des ABR et des AVL.

1. La preuve est tirée de [Wei, p. 13]

Chapitre 4

Étude expérimentale

4.1 Structure des arbres

Commençons par étudier quelques statistiques sur les hauteurs d'arbres pour une taille n fixée à l'aide de notre implémentation en Python.

Pour $n > 0$ fixé, on génère 1000 listes contenant les entiers de 1 à n dans un ordre aléatoire. On construit ensuite 1000 arbres à partir de ces listes. On calcule la hauteur de chacun de ces arbres générés aléatoirement. On affiche enfin la courbe avec les hauteurs obtenues en abscisses et le nombre d'arbres pour une hauteur donnée en ordonnée, ainsi que diverses informations statistiques. On indique également sur le graphique la valeur du rapport $K = \frac{\text{hauteur moyenne}}{log(n)}$.

Le résultat pour $n = 1000$ pour des arbres binaires est représenté par la figure 4.1. Cela confirme les résultats théoriques énoncés par le théorème 2.5.1. On remarque que sur 1000 arbres de taille 1000 la hauteur maximale est 30, les valeurs extrêmes semblent donc assez rares. On obtient $K \approx 3.2$ ce qui est relativement loin de la constante théorique. Cependant quelques tests sur des tailles plus grandes montre que ce rapport augmente bien avec n, mais lentement. Au vu du coût des tests, nous ne pouvons pas faire de graphique intéressant sur l'évolution du rapport en fonction de la taille, qui semblerait soit quasi constant.

Nous avons fait le même test pour notre implémentation des AVL, toujours pour 1000 arbres dont le résultat est à la figure 4.2. La courbe est moins intéressante : il y a en effet beaucoup moins de hauteurs possibles. Pour 1000 arbres de taille 1000 on a trouvé trois hauteurs : 11, 12 et 13, avec un seul arbre de hauteur 11. La borne théorique semble en tout cas respectée : $log(1000) \times 1.44 \approx 14.3 \geq 13$.

4.2 Complexités des méthodes

Pour mesurer la complexité des méthodes nous avons implémenté une fonction qui prend en paramètre deux entiers, n et k, et le nom d'une des trois fonctions (recherche, insertion, suppression). Nous commençons ensuite par construire un arbre de taille 1000, puis nous appelons la fonction à tester sur l'arbre pour k éléments aléatoires. Nous mesurons le temps d'exécution avec la fonction *timeit* et nous divisons ce temps par k afin de lisser les valeurs extrêmes éventuelles, dûes au fonctionnement de la machine lors du test. Nous augmentons ensuite de plus en plus vite la taille de l'arbre jusqu'à atteindre la taille maximale.

Remarquons que ce protocole pose un problème pour l'insertion et la suppression : si on fait ces tests sur un arbre initialement de taille t, il devient de taille $t+k$ (ou $t-k$ pour la suppression)

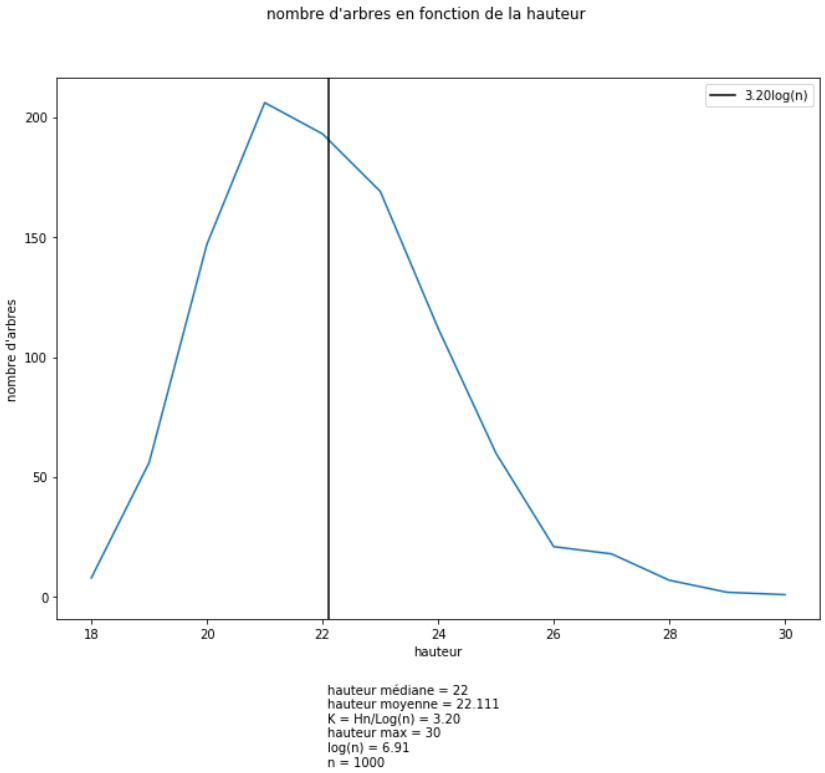


FIGURE 4.1 – nombre d'ABR en fonction de la hauteur pour n = 1000

après les k premiers tests. Cependant puisque nous nous intéressons au logarithme de la taille et puisque nous faisons typiquement 500 tests par arbre, de taille 1000 à 50 000, nous estimons que l'impact de ce changement de taille peut être négligé.

Une fois les mesures effectuées nous utilisons l'outil de régression polynomiale pour calculer une courbe de la forme $y = a \cdot log(x) + b$ que l'on affiche pour comparer à nos mesures. Notons que l'on mesure le temps d'exécution, exprimé en μ secondes, et non pas le nombre d'opération. Les coefficients trouvés n'ont donc pas de raison de s'approcher des résultats théoriques. Le résultat pour la fonction recherche est représenté par les figures 4.4 et 4.3. Nous laissons les autres fonctions en annexe. Pour toutes les fonctions nous retrouvons bien une complexité logarithmique par rapport à la taille des arbres.

Nous avons comparé les fonctions sur les ABR et AVL en recherchant ou insérant le même élément sur des arbres générés par la même liste d'éléments à chaque taille. Il n'y a pas de différence notables, les courbes obtenues sont très proches. Les figures sont consultables en annexe.

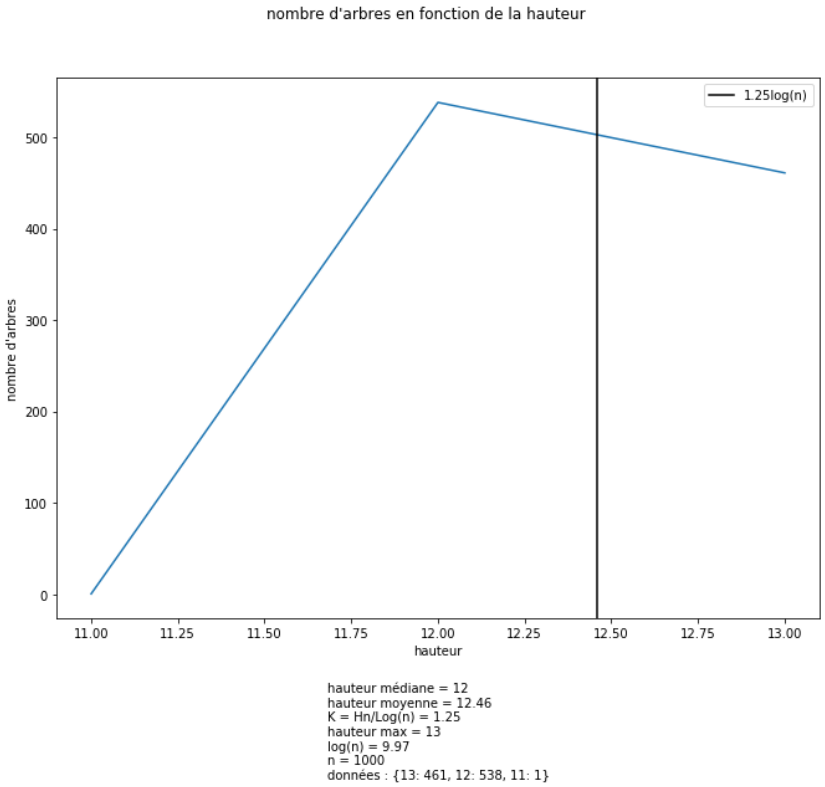


FIGURE 4.2 – nombre d'AVL en fonction de la hauteur pour n = 1000

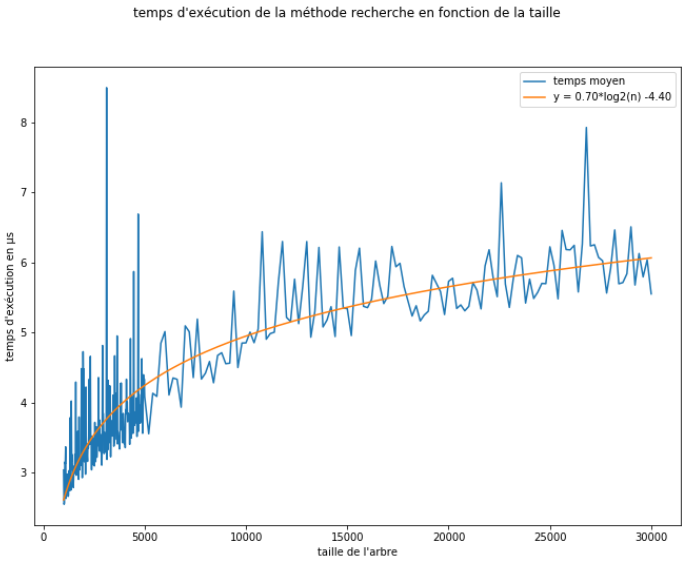


FIGURE 4.3 – mesure de la complexité temporelle de la fonction recherche sur des AVL

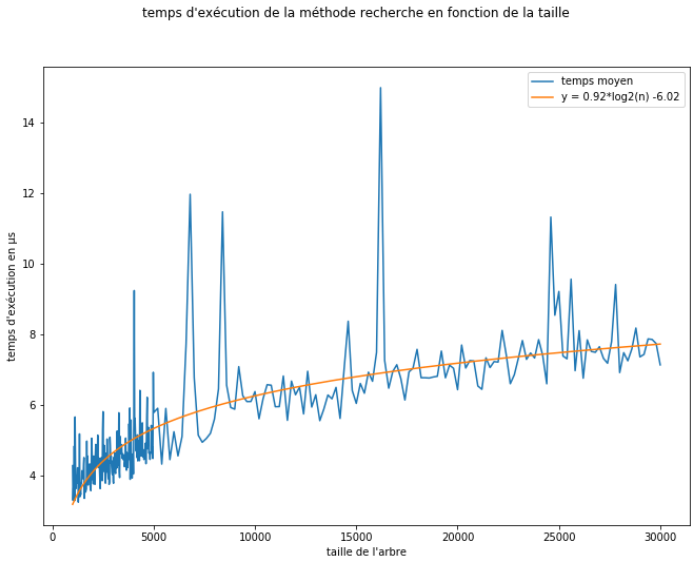


FIGURE 4.4 – mesure de la complexité temporelle de la fonction recherche sur des ABR

Chapitre 5

Quadtree Point

Suite à l'étude des arbres binaires de recherche, nous pouvons commencer à parler de Quadtree. Nous commençerons par décrire les quadtrees dits "point" [1], qui peuvent être vus comme une extension directe des arbres binaires de recherche. [2]

5.1 Présentation et premières méthodes

5.1.1 Structure du quadtree point

L'idée de base d'un quadtree point est la même que celle d'un arbre binaire de recherche : on souhaite "ranger" des données dans un arbre qui préservera toujours certaines propriétés telles que la recherche d'un élément soit la plus efficace possible, i.e. en ne parcourant qu'une certaine partie de l'arbre. Pour le quadtree on s'intéresse à des éléments ayant deux degrés de comparaison. Une manière de se le représenter est de penser à des éléments sur une carte, et on les classe selon leur position respective, en deux dimensions : est-ce que l'élément A est au Nord-Ouest, Nord-Est, Sud-Est ou Sud-Ouest de l'élément B ? On pourra utiliser des quadtrees pour des applications qui ne sont pas strictement géographiques, mais il sera toujours question d'objets à deux dimensions, comme pour le traitement d'une image, chaque pixel ayant une abscisse et une ordonnée.

Le vocabulaire général des arbres s'applique toujours : taille, hauteur, feuilles, noeuds internes etc. Le noeud d'un quadtree point est donc constitué au minimum des éléments suivants :

- deux coordonnées, nous les noterons x et y pour la suite, comme dans le plan euclidien classique
- un identifiant, que l'on notera plus tard "ID" correspondant à la clef ou l'étiquette pour reprendre le vocabulaire des arbres binaires de recherche
- quatre fils, éventuellement vides, chaque fils représentant un quadrant, on ne parlera alors plus de fils gauche ou de fils droit, mais de quadrant Nord-Ouest, Nord-Est, Sud-Est et Sud-Ouest

L'identifiant est indispensable si les coordonnées sont réelles ou si l'on autorise à ce que deux éléments aient la même position. En effet lors de la recherche et d'autres méthodes basées dessus, le test d'égalité entre éléments devra porter sur l'égalité des identifiants et non sur l'égalité des

1. On reprend la terminologie de l'article Wikipedia, [Wik19a] qui est aussi celle de [Sam90]
2. Nous nous sommes appuyés principalement sur les références suivantes pour ce chapitre :[Sam05], [FLB74], [420, p. 36]

coordonnées x et y. Il peut éventuellement contenir d'autres éléments, comme des propriétés sur les points.

Les coordonnées sont en général des réels (ou des entiers). Il faut bien entendu pouvoir les comparer. Au niveau de l'implémentation, nous allons créer une classe spécifique "Point" qui comporte les coordonnées et l'identifiant. Un noeud de quadtree a alors une clef de type Point et quatre sous-arbres.

Définition 5.1.1 (Quadtree Point)**.** *On suppose ici que les coordonnées sont réelles et on note les quatre quadrants pour leur dénomination anglaise, à savoir North-West (NW), North-East (NE), South-East (SE) et South-West (SW). La définition par induction de l'ensemble $\mathcal{QP}$ des quadtrees point :*

Base : $\emptyset \in \mathcal{QP}$

Induction : Soient ID l'identifiant d'un point de coordonnées $x, y \in \mathbb{R}$ et $NW, SW, SE, NE \in \mathcal{QP}$.

Si :

- $\forall (ID_1, NW_1, SW_1, SE_1, NE_1) \in NW, x_1 \leq x \text{ et } y_1 > y$
- $\forall (ID_2, NW_2, SW_2, SE_2, NE_2) \in SW, x_2 < x \text{ et } y_2 \leq y$
- $\forall (ID_3, NW_3, SW_3, SE_3, NE_3) \in SE, x_3 \geq x \text{ et } y_3 < y$
- $\forall (ID_4, NW_4, SW_4, SE_4, N3_4) \in NE, x_4 > x \text{ et } y_4 \geq y$

alors $(ID, NW, SW, SE, NE) \in \mathcal{QP}$

Pour qu'il n'y ait pas d'ambiguïté, il faut énoncer des règles de priorités si un point est situé exactement sur l'un des axes, d'où l'alternance entre les inégalités strictes et larges. Les règles choisies arbitrairement sont représentées par la figure 5.1

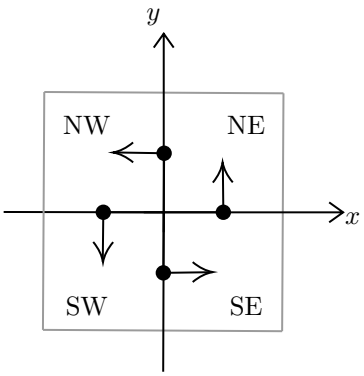


FIGURE 5.1 – Règles pour la localisation des points situés sur une frontière entre deux quadrants

Comme illustré par la figure 5.1, on peut considérer que pour chaque noeud d'un quadtree, il y a un plan euclidien dont le noeud courant, le point de référence, est l'origine $(0, 0)$.

5.1.2 Implémentation

On définit la première classe et la fonction d'égalité, comme annoncées plus haut :

```
class Point() :
    def __init__(self,ID,x,y) :
        self.ID = ID
        self.x = x
        self.y = y
```

```
def is_equal(self, point) :
    return (self.ID == point.ID)
```

Comme vu précédemment, il est crucial de pouvoir déterminer la position dun point en fonction dun autre. Évidemment, si un point est égal à un autre, il n'est dans aucun de ses quadrants.

Par exemple, pour savoir si un point est au Nord-Ouest (NW) d'un autre :

```
def is_NW(self, point):
    """ fonction qui renvoie si le point entré en paramètre se situe au nord-ouest
     du self
    a.is_NW(b) = True --> le point b est au nord-ouest du point a
    """
    return (self.x >= point.x and self.y < point.y)
```

Il en va de même pour les autres quadrants en modifiant les inégalités (larges ou strictes) selon la figure 5.1.

On implémente maintenant la classe "quadtree" :

```
class QuadtreePoint() :
    def __init__(self, point, NW, SW, SE, NE)
        self.point = point
        self.NW = NW
        self.SW = SW
        self.SE = SE
        self.NE = NE
```

5.1.3 Premières méthodes

Hauteur et taille Les définitions de la hauteur et de la taille d'un quadtree sont les mêmes que pour les arbres classiques. Par exemple la définition inductive de la hauteur est :

Base : $h(\emptyset) = 0$, $n(\emptyset) = 0$

Induction : Soit $T = (ID, NW, SW, SE, NE) \in \mathcal{QP}$, tel que $T \neq \emptyset$. La hauteur du quadtree T est donnée par :

$$h(T) = max(h(NW), h(SW), h(SE), h(NE)) + 1$$

Et la taille du quadtree T est donnée par :

$$n(T) = n(NW) + n(SW) + n(SE) + n(NE) + 1$$

Et on a alors des fonctions récursives analogues à celles des ABR pour le calcul de la taille et de la hauteur. La complexité est également en $\Theta(n)$ avec n la taille la taille du quadtree. Les constantes ne sont toutefois pas les mêmes : pour la taille, on sera en $4\Theta(n)$, toujours en comptant le nombre d'additions.

Parcours en largeur On l'implémente de la même manière que pour les arbres binaires avec un algorithme itératif et une file (premier entré, premier sorti). La complexité est aussi en $\Theta(n)$.

Parcours en profondeur On implémente les parcours en profondeur en suivant la même logique que pour les ABR, avec des appels récursifs. Il y a cependant cinq parcours possibles en fonction de l'ordre des appels récursifs, car trois type d'infixe :

- Point / NW / SW / SE / NE (parcours préfixe)
- NW / Point / SW / SE / NE (parcours infixe 1)
- NW / SW / Point / SE / NE (parcours infixe 2)
- NW / SW / SE / Point / NE (parcours infixe 3)
- NW / SW / SE / NE / Point (parcours suffixe ou postfixe)

La complexité est toujours en $\Theta(n)$.

5.2 Méthodes principales

Les méthodes de recherche et d'insertion fonctionnent comme pour les ABR, la suppression est quant à elle propre aux quadtrees. Les sources sont [Sam90] (à partir de la page 48) et [FLB74].

5.2.1 Recherche

La recherche s'implémente de manière très analogue à celle des ABR. La différence réside uniquement dans le nombre de fils qui est doublé. On procède à nouveau en quatre étapes :

- on compare avec le point racine, si l'égalité est vérifiée, on le renvoie directement
- sinon, on teste si la valeur recherchée est dans le quadrant NW, si elle l'est, on fait donc un appel récursif sur ce dernier
- on répète l'étape précédente pour chaque quadrant, jusqu'à atteindre la valeur recherchée
- et finalement si on atteint un point n'ayant pas de quadrant et que son identifiant n'est pas l'élément recherché alors l'élément ne se trouve pas dans le quadtree

```
1       def recherche(self, x):
2           """ recherche d'un point x dans le quadtree """
3           if self.point.ID is None:
4               return False
5
6           if self.point.is_equal(x):
7               return self
8           else:
9               if self.point.is_NW(x):
10                  if self.NW is not None:
11                      return self.NW.recherche(x)
12                  else:
13                      return False
14              elif self.point.is_SW(x):
15                  if self.SW is not None:
16                      return self.SW.recherche(x)
17                  else:
18                      return False
19              elif self.point.is_SE(x):
20                  if self.SE is not None:
21                      return self.SE.recherche(x)
22                  else:
23                      return False
```

```
24                 elif self.point.is_NE(x):
25                     if self.NE is not None:
26                         return self.NE.recherche(x)
27                     else:
28                         return False
```

5.2.2 Insertion

Ici aussi, l'implémentation est très proche de celle des ABR. Pour insérer un nouveau point dans un quadtree, il faut d'abord savoir où il doit se situer. On commence donc par chercher le futur emplacement du nouveau point à insérer, en réalisant des tests sur les quadrants en fonction des coordonnées de ce dernier. Puis on crée un quadtree auquel on attribut le point à insérer comme racine et des quadrants vides.

```
1      def insertion(self,x):
2          """ insertion d'un point x dans le quadtree """
3          if self.point is not None:
4              if self.point.is_NW(x):
5                  if self.NW is None:
6                      self.NW = Quadtree(x)
7                  else:
8                      self.NW.insertion(x)
9              elif self.point.is_SW(x):
10                 if self.SW is None:
11                     self.SW = Quadtree(x)
12                 else:
13                     self.SW.insertion(x)
14             elif self.point.is_SE(x):
15                 if self.SE is None:
16                     self.SE = Quadtree(x)
17                 else:
18                     self.SE.insertion(x)
19             elif self.point.is_NE(x):
20                 if self.NE is None:
21                     self.NE = Quadtree(x)
22                 else:
23                     self.NE.insertion(x)
24         else:
25             self.point = x
```

Comme pour les ABR, la complexité de la recherche et de l'insertion est en $\mathcal{O}(h)$. Dans le pire cas, il faut donc de l'ordre de n opérations pour un arbre de taille n. Cependant, il y a des résultats analogues à 1.2.11 pour les quadtrees, nous renvoyons le lecteur à [Sam05, p. 30] pour plus de détails. La complexité de ces fonctions dans le cas moyen est alors de l'ordre de $\mathcal{O}(log_4(n))$.

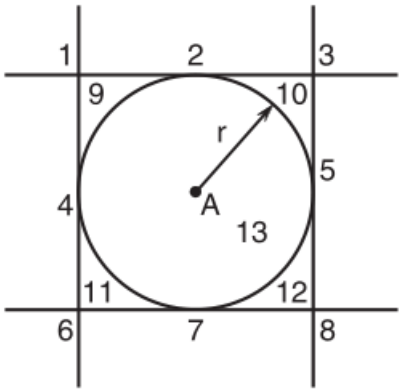
5.2.3 Suppression

La suppression n'est pas aussi simple que pour les arbres binaires. Si la première étape est identique avec une recherche récursive jusqu'à trouver l'élément à supprimer, la gestion des fils du noeud à supprimer peut s'avérer complexe. En effet, on ne peut plus choisir un fils qui prend simplement la place du noeud supprimé, la conservation de la propriété de quadtree n'est pas garantie. Une première approche serait de réinsérer tous les fils après la suppression du fils. La

complexité n'est donc plus de l'ordre de la hauteur, dans le pire cas on doit faire $n-2$ insertions et on obtient une complexité en $\mathcal{O}(nlog(n))$.

5.3 Recherche dans une zone

Une méthode illustrant l'intérêt du quadtree est la recherche dans une zone circulaire. En effet, en suivant les cas décrits dans la figure 5.2, on peut éliminer une grande partie de l'arbre en fonction de la localisation de la zone. On peut ensuite envisager une fonction de recherche des voisins d'un point donné dans une zone circulaire autour de ce point. Il est possible aussi de faire des recherches dans d'autres types de zones, comme les zones rectangulaires, polygones convexe etc. [3]



Problem: Find all nodes within radius r of point A

Solution: If the root is in region *I* (*I* =1…13), then continue to search in the quadrant specified by *I*

1. SE
2. SE,SW
3. SW
4. SE,NE
5. SW,NW
6. NE
7. NE, NW
8. NW
9. All but NW
10. All but NE
11. All but SW
12. All but SE
13. All

FIGURE 5.2 – Recherche dans une zone circulaire : régions éliminables

3. On trouvera plus de détails sur la recherche dans un quadtree point dans [Sam05, p. 36].

Chapitre 6

Quadtree region

6.1 Pésentation des quadtrees région

6.1.1 Description

Les quadtrees région sont basés sur la subdivision d'une région donnée en quatre rectangles égaux à chaque niveau. La région est décomposée récursivement en quatre blocs rectangulaires jusqu'à ce que tous les blocs soient occupés par un objet ou qu'ils soient vides. Chaque nœud de larbre correspond à un quadrant de l'ensemble de données. On a deux types de nœuds :

- Les nœuds intérieurs qui ont quatre fils, où chacun représente un quart de son nœud parent (avec la direction géographique : NW, NE, SW, SE) et tous les quadrants du même niveau ont la même taille.
- Les nœuds feuilles qui correspondent aux blocs qui ne nécessitent pas de subdivision, soit ils sont vide, soit on ne peut plus les diviser.

Ce type d'arbre est utilisé quand on a des données à densité (par exemple les nuages de point), ou plus généralement des données ayant une surface comme les images, les cartes géographiques.

6.1.2 Critère pour l'arrêt de la subdivision d'un quadtree

Toute technique de décomposition spatiale commence par englober le domaine dans une boîte, dont la taille dépend de l'application. Une fois cette construction effectuée, la boîte est ensuite récursivement découpée en régions égales. Le processus est itéré jusqu'à trouver une critère d'arrêt qui dépend spécifiquement de l'application envisagée. Le choix d'un critère a une incidence notable sur l'allure générale de la structure d'arbre (profondeur, équilibrage, etc...).

On peut prendre par exemple comme critère d'arrêt :

- La profondeur maximale de l'arbre, qui va lui être un paramètre, donc on continue le découpage tant qu'on n'a pas atteint la profondeur maximale de l'arbre.
- Le nombre maximal d'objets que peut contenir un quadrant, qu'on note maxObjet, qui va aussi être un paramètre de la fonction de découpage, ce qui fait qu'un bloc est décomposé tant qu'il contient plus d'objet que maxObjet (avec maxObjet >=1).

6.1.3 Exemples d'application des quadtrees région

- La segmentation d'image, qui consiste à regrouper les pixels ayant les même propriétés en régions homogènes. Si on considère une image binaire (en noir et blanc), le critère d'arrêt

du découpage est si la zone contient à la fois des pixels noirs et des pixels blancs. Si oui, il faut la redécouper, sinon la zone est considéré homogène, il faut donc s'arrêter.
La longueur d'un côté est statique et peut être trouvée par la formule : $L = W/2^p$ où :
- L est la longueur d'un quadrant
- W est la longueur de côté de la région donnée
- p est la profondeur, qui est le nombre de niveau par lesquels un nœud est séparé de la racine (la profondeur de la racine est 0).

- La recherche dans un nuage de point-région :
Ce genre d'application utilise des quadtrees que l'on appelle les quadtrees Point-Région (PR). Dans ce qui suit, on se concentrera sur ce type de quadtree.
Définition 6.1.1 (Quadtree PR)**.** *Les quadtrees PR (Point-Région) sont des quadtrees dont les données sont des points, et sont toujours stockées dans les feuilles. On subdivise les quadrants jusqu'à avoir un seul point dans chaque quadrant, ainsi un point est associé au plus petit bloc du quadtree.*

Théorème 6.1.2 (Le nombre de nœuds dans un quadtree)**.** *Si un quadtree est de profondeur p, contenant n points avec un point par région alors le nombre de noeud vaut $O((d+1)n)$.*

Preuve. On renvoie le lecteur à [MdBS00] pour une preuve de ce résultat. □

Théorème 6.1.3 (La profondeur d'un quadtree)**.** *La profondeur d'un quadtree PR est au plus égale au $log(w/c)+3/2$, où c est la plus petite distance entre deux points quelconques du quadtree et w est la longueur du côté du carré initial.*

Preuve. La preuve de ce résultat est également exposée dans [MdBS00]. □

6.2 Fonction pour les quadtrees PR

6.2.1 Recherche

C'est la fonction la plus importante, car toutes les autres fonctions l'utilisent. Elle est intuitive, il faut simplement chercher récursivement dans quel quadrant se trouve un point donné. Pour la complexité on a :
- meilleur cas : le point se trouve dans un quadtrant de profondeur 1 la complexité vaut $\Theta(1)$.
- pire cas : le point cherché se trouve dans le nœud le plus profond(sa profondeur est égale à la profondeur h de l'arbre), alors la complexité est en $\mathcal{O}(h)$.

6.2.2 Insertion

insertion(P) :
- chercher le quadrant qui va contenir le point P
- si ce quadrant est vide, ajouter P
- s'il contient déjà un point Q , subdiviser le quadrant jusqu'à ce que P et Q soient dans deux quadrant différents puis insérer P dans le bon endroit.

La forme de l'arbre est entièrement indépendante de l'ordre dans lequel les éléments sont ajoutés. la fonction d'insertion consiste à faire une recherche du quadrant où il faut insérer le point puis ajouter le point à ce quadrant. Cette dernière étape est en $\Theta(1)$, donc l'étape la plus important est la recherche. Donc au final la complexité de fonction d'insertion est la même que la fonction de recherche, soit $\Theta(1)$ dans le meilleur cas et $\mathcal{O}(h)$ dans le pire cas.

6.2.3 Suppression

suppression(P) :
- chercher le quadrant qui contient P
- supprimer le point
- si le père contient un seul enfant après la suppression, il doit être remplacé par son unique fils, répéter jusqu'à ce que le père ait au moins deux fils.

complexité : pareil que la fonction d'insertion, la fonction de suppression fait d'abord une recherche du point puis sa suppression qui est en $\Theta(1)$, donc sa complexité est : $\Theta(1)$ dans le meilleur cas et $\mathcal{O}(h)$ dans le pire cas.

6.2.4 Comparaison

Les quadtree régions dont les données sont des points :
- repose sur une décomposition régulière de l'espace
- • les données sont stockées dans les feuilles
- \+ suppression simple
- \+ forme indépendante de l'ordre d'insertion

Les quadtrees point
- \+ permet aux rectangles de sétendre à l'infini dans certaines directions
- • les données se trouvent les nœuds internes
- \+ ont souvent moins de nœuds
- suppression plus difficile
- la forme dépend de l'ordre d'insertion

6.3 Application : compression d'image

6.3.1 Algorithme

Une des applications possibles du Quadtree région est la compression d'image. La compression d'image que nous avons implémentée est un algorithme qui charge une image dans un Quadtree implémenté sous forme de *liste* : on insère d'abord tous les pixels qui correspondront aux feuilles du quadtree et on insère ensuite les quadrants composés des pixels que nous venons d'insérer.

L'implémentation sous forme de liste pour le quadtree est analogue à l'implémentation des ABR sous forme de liste (voir 2.2.2). Dans un ABR, pour un nœud à l'indice i, le fils gauche était à l'indice $2i$ de la liste, et le droit à l'indice $2i + 1$. Dans un quadtree, pour un nœud à l'indice i, ses quatre fils seront respectivement aux indices $4i + 1, 4i + 2, 4i + 3, 4i + 4$

L'implémentation sous forme de liste a été choisie pour sa simplicité d'implémentation comparée à une implémentation classique.

Pour déterminer le nombre de quadrants, on divise l'image en 4 parts égales jusqu'à que ces parts fassent un pixel. Le quadtree est ainsi rempli par des quadrants et des pixels. La classe Pixel utilise l'implémentation classique :
- Un tableau de 3 entiers correspondant au codage RGB du pixel
- Deux points permettant de déterminer la position du pixel (car on regroupera plus tard les pixels en zones de plusieurs pixels ce qui nécessitera l'utilisation de deux points pour obtenir la position de la zone)

Principe de l'algorithme :

- On calcule le nombre de pixels de l'image qui correspondra au nombre de feuilles du quadtree
- On calcule le nombre de quadrants
- On ajoute tous les pixels dans le quadtree
- On insère les quadrants par taille croissante dans le quadtree : la couleur d'un quadrant correspond à la moyenne arithmétique des pixels qui le composent.
- On peut ensuite reconstruire l'image initiale en utilisant le quadtree selon un niveau de compression choisi par l'utilisateur.

6.3.2 Illustrations

Image simple

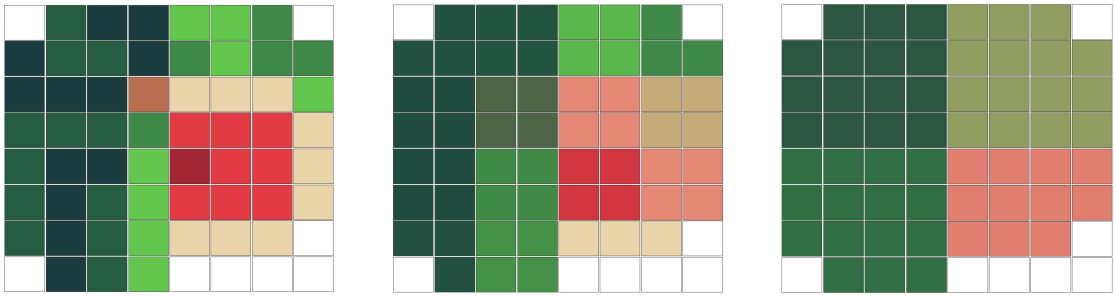


FIGURE 6.1 – De gauche à droite : Image originale, première étape de la compression, deuxième étape de la compression

La figure 6.1 est un premier exemple du fonctionnement de l'algorithme. L'image initiale est une image simple de 8×8 pixels représentant une pastèque.

La deuxième image correspond à la première étape de la compression. On regroupe les pixels en zones de 2×2 pixels. La couleur d'une zone correspond à la moyenne des couleurs des 4 pixels constituant la zone.

Enfin, la troisième image correspond à la deuxième étape de la compression. On regroupe les zones déjà existantes en zones de 4×4 pixels en utilisant le même procédé que précédemment.

La figure 6.2 représente le quadtree utilisé pour la compression de l'image de la figure 6.1. Les feuilles de l'arbre (3ème étage) sont les pixels. L'étage juste au-dessus (2ème étage) correspond aux zones de 4×4 pixels. Enfin, le 1er étage correspond aux zones de 8×8 pixels.

Il y a 64 feuilles dans l'arbre (correspondant aux 64 pixels), 16 quadrants de taille 2×2 au 2ème étage et 4 quadrants de taille 4×4 au 1er étage.

La figure 6.3 illustre la façon dont sont remplies les cases de la liste. Les feuilles (pixels) sont stockées à la fin du tableau. Les quadrants de taille 2×2 sont ensuite insérés juste avant les feuilles. Enfin, les quadrants de taille 4×4 sont placés avant les quadrants de taille 2×2 au début du tableau.

Remarque 1. *La reconstruction de l'image compressée utilise une partie plus ou moins grande de la liste selon la précision voulue.*

Ainsi, en reconstruisant l'image uniquement avec les 4 premières cases de la liste, on obtient la troisième image de la figure 6.1.

Cependant, en utilisant les cases 4 à 19 de la liste, on obtient la deuxième image de la figure 6.1.

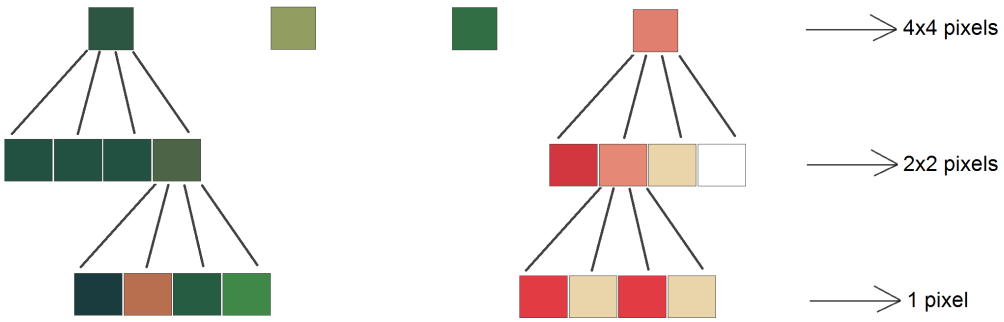


FIGURE 6.2 – Représentation du Quadtree modélisant la figure 6.1

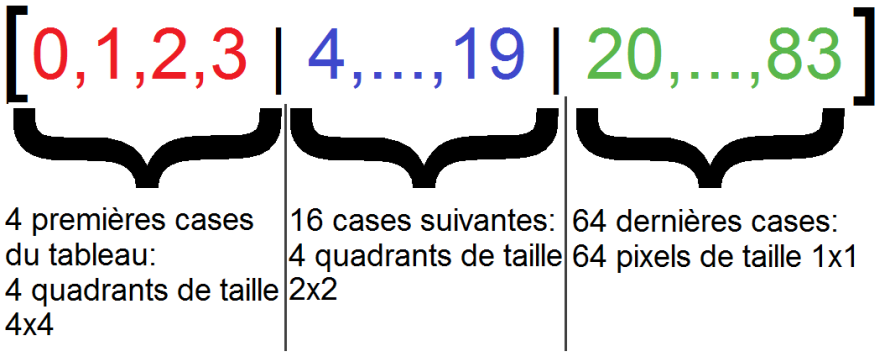


FIGURE 6.3 – Représentation de la liste modélisant le quadtree de la figure 6.2

Enfin, si l'on reconstruit l'image en utilisant les cases 20 à 83, alors l'image obtenue est exactement l'image de départ, ce qui est cohérent car on reconstruit l'image avec tous les pixels qui la composent.

Photographies

FIGURE 6.4 – Image non compressée, 135 ko

FIGURE 6.5 – Image compressée, 62 ko

FIGURE 6.6 – De gauche à droite : Image originale, image compressée, image fortement compressée

La figure 6.4 est une image non compressée. La figure 6.5 est la même image mais cette fois-ci compressée. Après compression, la taille de l'image a été réduite de plus de la moitié de sa taille originale, la perte de qualité est d'ailleurs assez visible.

La figure 6.6 montre une image originale de 135 ko ainsi que deux images compressées :

- Une image compressée de 111 ko
- Une image fortement compressée de 62 ko

L'image du milieu montre qu'il est possible de garder une bonne qualité d'image après compression mais le gain de taille est alors minime.

Conclusion

Cette étude a permis d'établir les avantages des arbres binaires de recherche et des quadtrees pour le stockage et la manipulation de données que l'on peut classer soit par une relation d'ordre, soit par une relation de type position géographique. Nous avons vu, par une analyse théorique ainsi que par des tests expérimentaux, que les méthodes classiques comme la recherche, l'insertion et la suppression d'éléments sont en moyenne bien plus efficaces que pour une structure classique, comme une liste. L'implémentation des AVL permet d'avoir même une structure donnant des garanties sur les complexités de ces méthodes. Nous avons aussi étudié un exemple d'application concrète utilisant les quadtrees : la compression d'image. Notre travail sur les quadtrees peut être prolongé par une étude des octrees, utiles pour des données spatiales, ou les kd-trees pour une généralisation des quadtrees. Il serait également intéressant d'étudier les méthodes autour de la recherche des plus proches voisins sur ce type de structure. [1].

1. Pour approfondir ces questions nous recommandons de commencer par consulter [Sam05]

Annexe

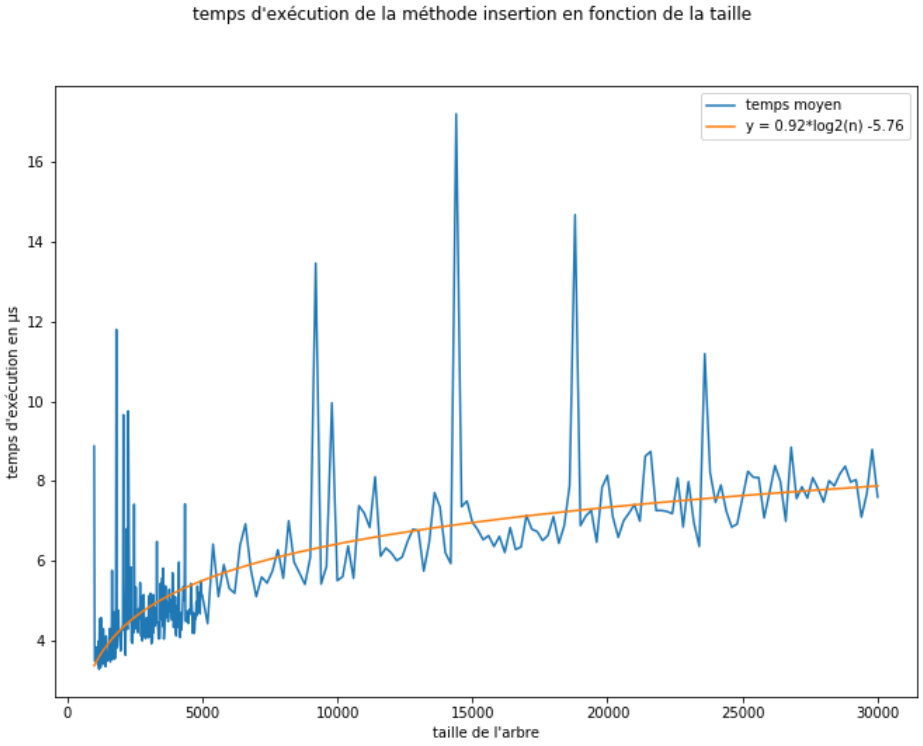


FIGURE 6.7 – mesure de la complexité temporelle de la fonction insertion sur des ABR

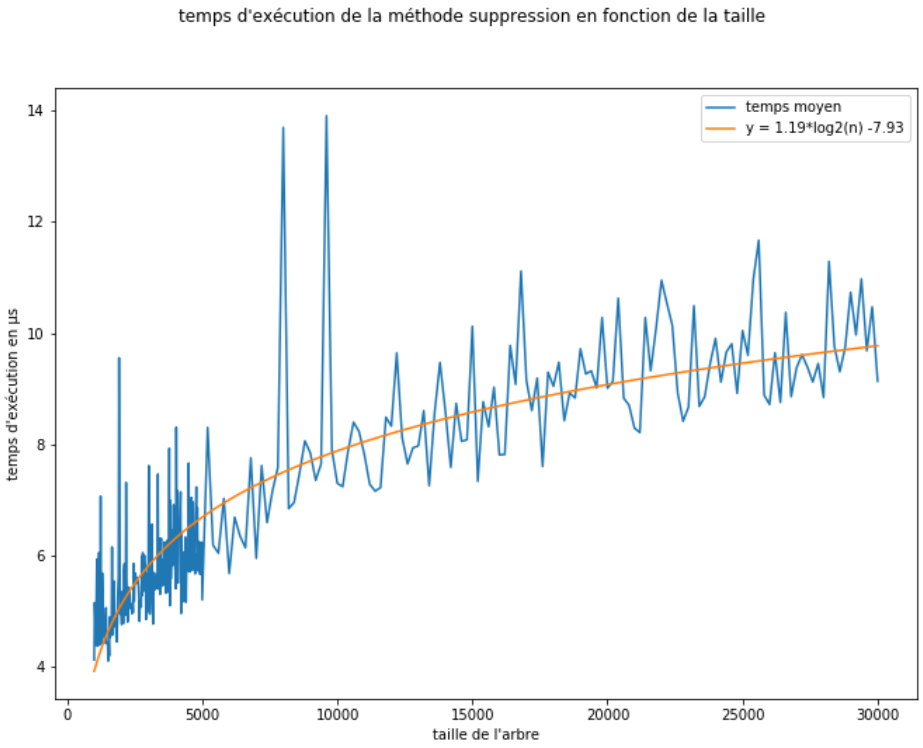


FIGURE 6.8 – mesure de la complexité temporelle de la fonction suppression sur des ABR

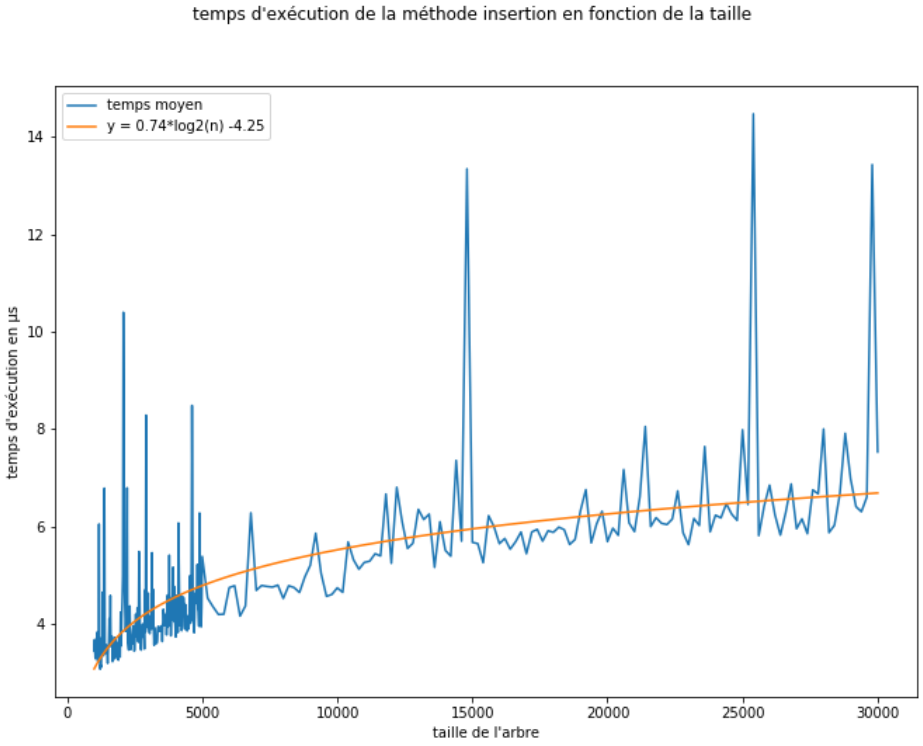


FIGURE 6.9 – mesure de la complexité temporelle de la fonction insertion sur des AVL

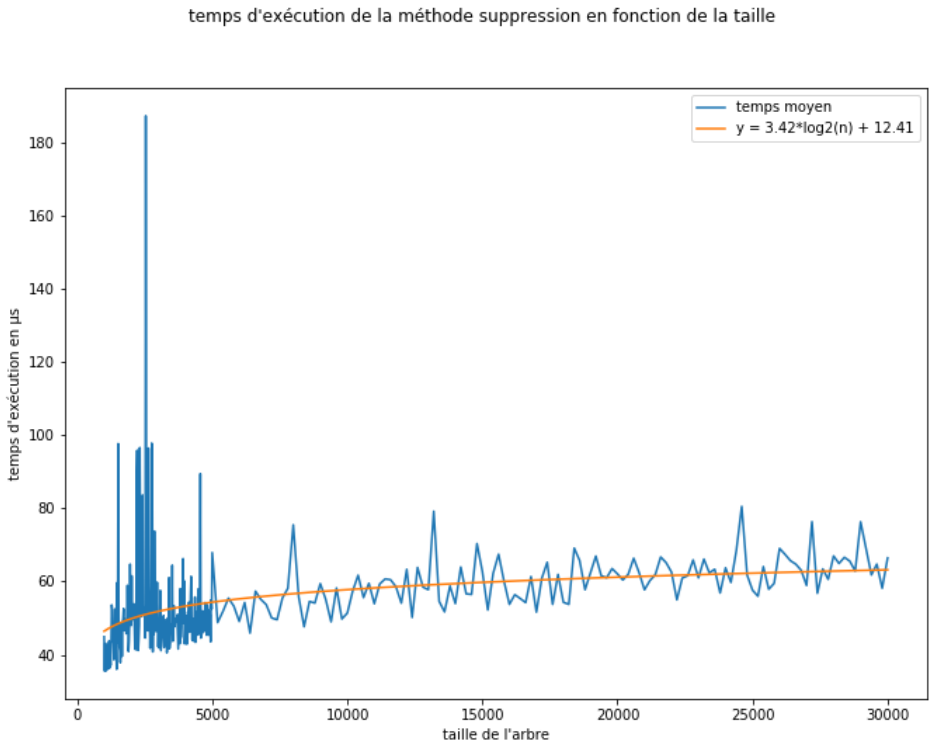


FIGURE 6.10 – mesure de la complexité temporelle de la fonction suppression sur des AVL

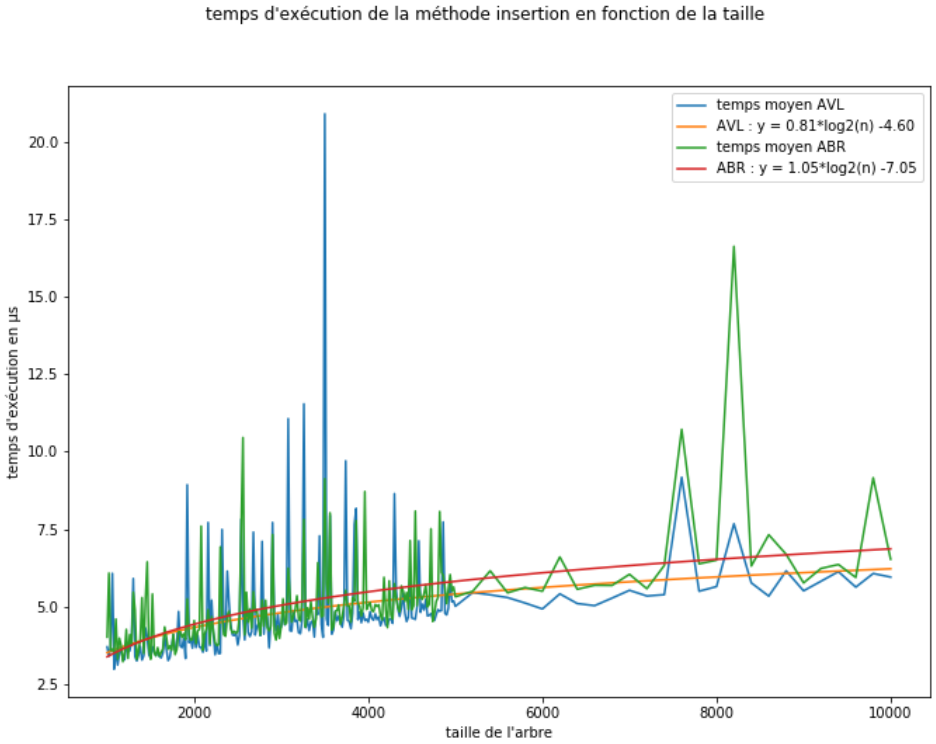


FIGURE 6.11 – comparaison complexité temporelle de la fonction insertion sur les ABR et AVL

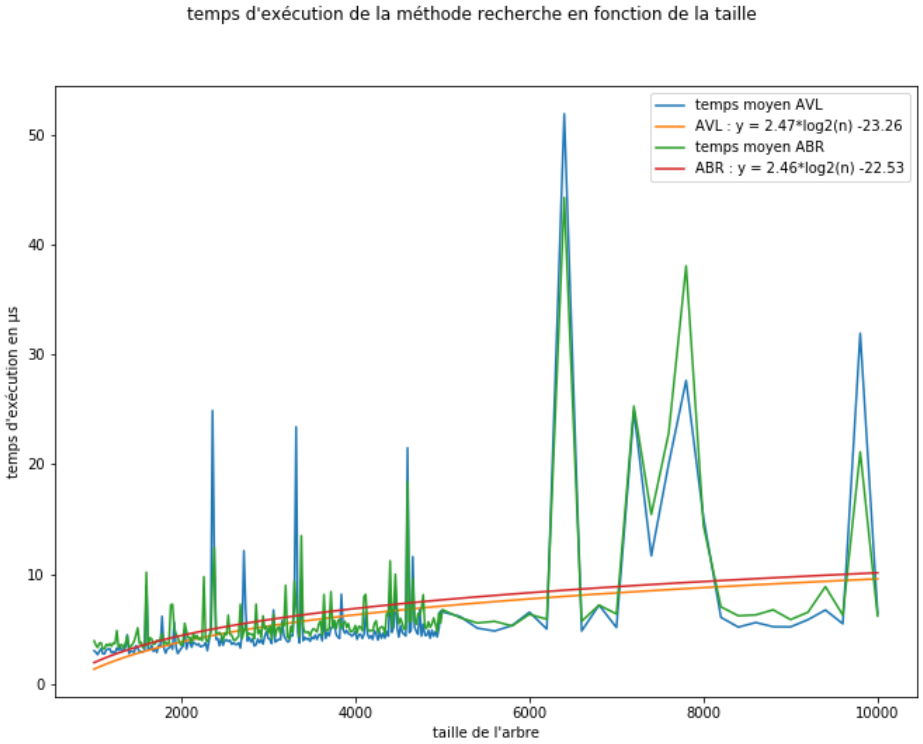


FIGURE 6.12 – comparaison complexité temporelle de la fonction recherche sur les ABR et AVL

Bibliographie

[420] CMSC 420. Quadtree. https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/quadtrees.pdf.

[AVL62] George M Adel'son-Vel'skii and Evgenii Mikhailovich Landis. An algorithm for organization of information. *Russian Academy of Sciences*, 146(2) :263–266, 1962.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[Dev86] Luc Devroye. A note on the height of binary search trees. *Journal of the ACM (JACM)*, 33(3) :489–498, 1986.

[Drm03] Michael Drmota. An analytic approach to the height of binary search trees ii. *Journal of the ACM (JACM)*, 50(3) :333–374, 2003.

[eeI17] equipe enseignante 2I001. Velo-libre. http://www-licence.ufr-info-p6.jussieu.fr/lmd/licence/2018/ue/2I001-2018oct/public/Annales/2I001_2017oct_S1_corrige.pdf, 2017. [Online ; accessed 27-March-2019].

[FLB74] Raphael Finkel and Jon Louis Bentley. Quad trees : A data structure for retrieval on composite keys. *Acta Inf.*, 4 :1–9, 03 1974.

[LD] Bruce Reed Luc Devroye. On the variance of the height of random binary search trees. https://pdfs.semanticscholar.org/ba5d/51962a444d5611c50612d5f07c1c8e1f33b1.pdf.

[MB] Anton Dignös Michael Böhlen. Improving the performance of region quadtrees. https://www.ifi.uzh.ch/dam/jcr:ffffffff-96c1-007c-ffff-fffff2d50548/ReportWolfensbergerFA.pdf.

[McQ] McQuain. Pr quadtrees. http://courses.cs.vt.edu/cs3114/Summer11/Notes/T06.PRQuadTrees.pdf.

[MdBS00] Mark Overmars Mark de Berg, Marc van Kreveld and Otfried Schwarzkopf. *Computational Geometry*. 3rd edition, 2000.

[Qua] Quadtrees. http://personal.us.es/almar/cg/09quadtrees.pdf.

[Ree03] Bruce Reed. The height of a random binary search tree. *Journal of the ACM (JACM)*, 50(3) :306–332, 2003.

[Sam90] Hanan Samet. *The design and analysis of spatial data structures*, volume 85. Addison-Wesley Reading, MA, 1990.

[Sam05] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[Sta11] Richard P. Stanley. *Enumerative Combinatorics : Volume 1*. Cambridge University Press, New York, NY, USA, 2nd edition, 2011.

[Vin] Jean-Marc Vincent. Heapsort. http://mescal.imag.fr/membres/jean-marc.vincent/index.html/ProTer/Algorithmes-Classiques/Heapsort.pdf.

[Wei] Stewart Weiss. Csci software design and analysis 3, chapter 4 trees, part 2. http://www.compsci.hunter.cuny.edu/~sweiss/course_materials/csci335/lecture_notes/chapter04.2.pdf.

[Wik19a] Wikipedia. Quadtree — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Quadtree&oldid=883293662, 2019. [Online ; accessed 27-March-2019].

[Wik19b] Wikipedia. Rotation d'un arbre binaire de recherche — Wikipedia, the free encyclopedia. https://fr.wikipedia.org/wiki/Rotation_d'un_arbre_binaire_de_recherche, 2019. [Online ; accessed 16-March-2019].

[Wik19c] Wikipedia. Tree (graph theory) — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Tree%20(graph%20theory)&oldid=883778328, 2019. [Online ; accessed 04-March-2019].

[Wik19d] Wikipedia. Tri par tas — Wikipedia, the free encyclopedia. https://fr.wikipedia.org/wiki/Tri_par_tas, 2019. [Online ; accessed 16-March-2019].