Quadtree 3M101

Thiziri Baiche Charlotte Briquet Serge Durand Kevin Meetooa

March 17, 2019

Contents

Introduction

Les arbres sont une structure de donnée classique, aux nombreuses variations et applications. Notre objectif pour ce projet est d'étudier et d'implémenter des quadtree [1], avec plusieurs applications concrètes. Pour bien comprendre les quadtree nous devons d'abord étudier les arbres binaires. Après avoir introduit les notions et le vocabulaire de base sur les arbres binaires nous étudierons en particulier les arbres binaires de recherche avec une première implémentation. En effet les quadtree peuvent être vu comme une extension des arbres binaires de recherche. Nous présentons également des mesures expérimentales de la complexité des méthodes implémentées.

[1] nous utiliserons le terme quadtree dans le reste du rapport, préféré au terme "arbre quaternaire", pas très joli...

Chapter 1

Les Arbres Binaires

1.1 Description et vocabulaire des arbres binaires

1.1.1 Définitions générales sur les arbres

Un *arbre* est une structure de donnée hiérarchique définie par un nombre fini de nœuds. Chaque *nœud* est composé d'une *clef* (appelée également *étiquette*) qui représente sa valeur ou l'information associée, et d'un ensemble de références vers d'autres noeuds. On dirat que l'arbre est *étiqueté* sur l'ensemble E si ses étiquettes appartiennent à l'ensemble E. On dit que chaque nœud est relié par une *branche*. On nomme des nœuds *parents*, *enfants* ou *fils*, *frères*, *ancêtres* ou *descendants*, les nœuds d'un arbre de manière analogue à un arbre généalogique. Dans un arbre, binaire ou non, un noeud a exactement un parent, sauf la *racine* qui n'en a aucun : c'est la particularité de cette structure et ce qui donne le nom d'arbre.

On dit qu'un noeud qui n'a aucun fils est une *feuille*. Le *degré* d'un nœud est défini par le nombre de fils qu'il possède. Le degré maximal correspond au degré de l'arbre. Un arbre **binaire** est donc un arbre de degré deux, c'est à dire que chaque noeud a au plus deux fils. La *taille* d'un arbre est son nombre total de nœuds. Le *chemin* d'un nœud est une suite de nœuds qu'il faut emprunter pour parcourir l'arbre de la racine au nœud en question. On appelle la *longueur* d'un chemin, le nombre de nœuds empruntés.

La *hauteur* (ou profondeur) d'un arbre est la longueur du chemin le plus long. On considère la racine de l'arbre comme de niveau 1 puis à chaque génération le niveau augmente de 1. 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, un sous-arbre donc. Voilà quelques exemples d'arbres :

1.1.2 Les arbres ordonnés

On dit qu'un arbre étiqueté sur un ensemble muni d'une relation d'ordre totale est *ordonné* si tous ses nœuds ont une étiquette supérieure ou égale à celle de chacun de ses enfants s'ils existent. Ainsi, l'étiquette de la racine a la valeur maximale. Pour tout chemin de l'arbre, les étiquettes se succèdent dans un ordre décroissant.

On appelle *tas* ou *arbre tassé* un arbre binaire ordonné 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.

1.1.3 Les arbres d'expression arithmétiques

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) numérique 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. 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.

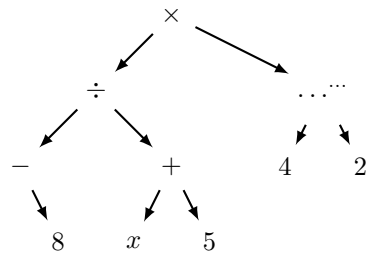


Figure 1.1: Exemple d'arbre binaire d'expression algébrique

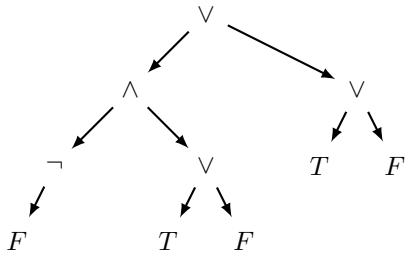


Figure 1.2: Exemple d'arbre binaire d'expression booléen

1.2 Particularité des arbres binaires

On donne la définition inductive de l'ensemble des arbres binaires étiqueté sur un ensemble E, qu'on note AB:

Base :

$\emptyset \in AB$

Induction :

$\forall x \in E, G \in AB, D \in AB : (x, G, D) \in AB$

Un arbre binaire est ainsi un triplet constitué de :

- un nœud racine
- un sous-arbre gauche
- un sous-arbre droit

éventuellement vides.

Un arbre binaire qui ne contient pas de nœuds est appelé un arbre vide ou un arbre nul. Si le sous-arbre de gauche n'est pas vide, sa racine est appelée le fils gauche de la racine de l'arbre entier. Il en va de même pour le sous-arbre de droite. Par exemple, dans l'arbre (a), le nœud « 2 » est le fils de gauche de la racine de l'arbre, il est aussi la racine du sous-arbre de gauche. Si un sous-arbre est un arbre nul, on dit alors que le fils est absent ou manquant.

1.2.1 Premiers algorithmes

Notations de Landau

Pour la description des algorithmes et l'analyse de leur complexité on utilisera les notations usuelles (de Landau). On les rappelle ici : f et g désignent des fonctions de $\mathbb{N}$ dans $\mathbb{N}$.

- $f \in \mathcal{O}(g)$ si $\exists D > 0$ et $n_0 \geq 0$ tels que $\forall n \geq n_0, f(n) \leq Dg(n)$
- $f \in \Omega(g)$ si $\exists C > 0$ et $n_0 \geq 0$ tels que $\forall n \geq n_0, Cg(n) \leq f(n)$.

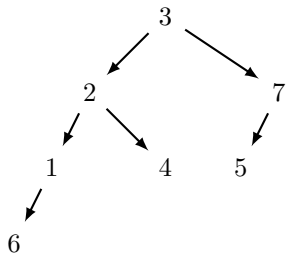


Figure 1.3: Arbre (a)

- $f \in \Theta(g)$ si $\exists C > 0, D > 0$ et $n_0 \geq 0$ tels que $\forall n \geq n_0, Cg(n) \leq f(n) \leq Dg(n)$.

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

calcul de la hauteur et de la taille

Donnons d'abord des définitions inductives de la hauteur et de la taille d'un arbre. Nous notons $h(T)$ la hauteur de l'arbre T, et $n(T)$ sa taille. On a :

Base :

$h(\emptyset) = 0$
$n(\emptyset) = 0$

Induction :

$\forall x \in E, G \in AB, D \in AB$:
$h((x, G, D)) = max(h(G), h(D)) + 1$
$n((x, G, D)) = n(G) + n(D) + 1$

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

```
def hauteur(self):
        """convention : arbre vide de 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
```

Complexité, terminaison et correction : On fait une preuve détaillée pour la terminaison, la correction et la complexité de cette méthode. Pour le calcul de la complexité on compte le nombre d'addition. On pose $c(n) =$ le nombre d'addition lors de l'appel $hauteur(T)$ sur un arbre T de taille n. On montre par induction sur l'ensemble AB que $c(n) = n$ pour tout $T \in AB$ tel que $n(T) = n$, et aussi que la fonction se termine et est correcte.

Base :

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

Induction :

On fait une 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 $T = (x, T_1, T_2)$ 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 + le 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 $A \in AB$ tel que $n(A) < n$ alors elle est vraie pour tout $T \in AB$ tel que $n(T) = n$. En conclusion la propriété est vraie pour tout arbre binaire.

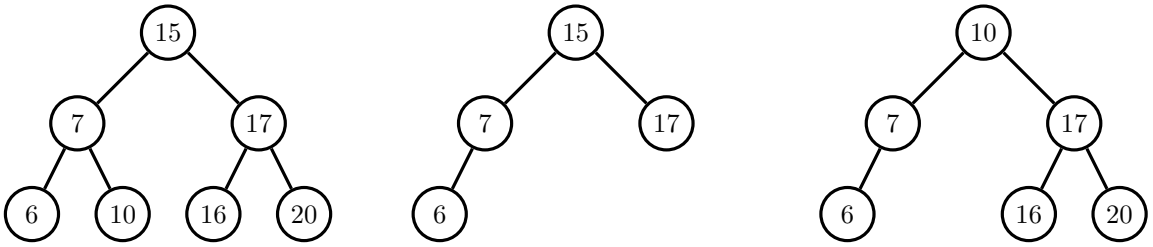
En pratique on a dû distinguer le cas où T est réduit à un noeud, les appels T.hauteur() ne fonctionnant pas en Python si T est None, mais cela n'a pas d'impact sur la complexité.

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.

1.3 Relations entre hauteur et taille

On rappelle qu'un arbre complet est un arbre dont tout les nœuds internes ont 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. On introduit également la notion d' *arbre H-équilibré* : arbre dont la différence de profondeur entre deux feuilles est au plus 1. Tout les niveaux sont donc rempli sauf le dernier ici aussi, sans condition sur la répartition des feuilles du dernier niveau.

Illustration :



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

On remarque que tout arbre binaire (x, G, D) est complet si et seulement si G et D sont complets et de même hauteur.

On a les propriétés suivantes pour les arbres complets :

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

$$2^h - 1 = n \tag{1.1}$$

$$\Leftrightarrow \quad h = log(n+1) \tag{1.2}$$

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 nœud dans l'arbre vide.

Induction : Supposons que la propriété soit vérifiée pour tout arbre binaire complet de hauteur $h-1$. Soit (x, G, D) un arbre binaire complet de hauteur h et de taille n. Alors G et D sont des arbres binaires complets de hauteur $h-1$. Alors $n = n_G + n_D + 1 = 2^{h-1} - 1 + 2^{h-1} - 1 + 1 = 2 * (2^{h-1}) - 1 = 2^h - 1$. La propriété est vraie pour un arbre vide et se propage, elle est donc vérifiée pour tout arbre complet. Il suffit ensuite de prendre le logarithme de chaque côté de (1) pour avoir la deuxième égalité. □

Conséquences : On a $log(n) \leq log(n+1) \leq log(n^2) = 2log(n)$ donc $log(n+1) = \Theta(log(n))$. **Les algorithmes de complexité linéaire en la hauteur d'un arbre sont donc logarithmique en la taille de l'arbre pour les arbres complets**.

De plus si 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 linéaire en la hauteur sont donc également logarithmique en la taille pour les arbres H-équilibré. Pour garantir une certaine efficacité des algorithmes l'enjeu est donc de construire des arbres le plus équilibré possible.

Proposition. *Un ABR 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 $T = (x, G, D)$, $f(T) = f(G) + f(D) = 2 * f(G)$, avec $f(T) =$ le nombre de feuilles de T. On en déduit un corollaire assez intéressant:

Corollaire. *Plus de la moitié des noeuds dans un ABR 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 (hauteur moyenne d'un noeud)**.** *La profondeur moyenne d'un noeud choisi au hasard est en $\Theta(log(n))$*

Preuve. Par définition d'un ABR 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 de noeuds total.

Ce rapport vaut : $\frac{\sum_{k=0}^{h-1} k * 2^k}{2^h - 1} = \frac{2^h * h - 2 * (2^h - 1)}{2^h - 1} = \frac{(h-1) * 2^h - 2^h + 2}{2^h - 1}$ (la valeur de la somme a été calculée à l'ordinateur)

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

Donc la hauteur moyenne d'un noeud vaut $h - 1 = \Theta(log(n))$ (d'après un théorème précédent) □

Chapter 2

Les Arbres Binaires de recherche

2.1 Définition

Un arbre binaire de recherche est un arbre binaire étiqueté sur un ensemble muni d'un ordre total (typiquement les réels ou les entiers) dans lequel tout noeud a une étiquette supérieure à celles de son sous arbre gauche et inférieure à celles de son sous arbre droit. L'abréviation ABR ou BST pour *binary search tree* est fréquemment utilisée. On donne aussi la définition inductive suivante :

Base :

$\emptyset \in ABR$

Induction :

$\forall x \in E, G \in ABR, D \in ABR$, 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 $(x, G, D) \in ABR$

2.2 Parcours et affichage

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], 1:[6], 4:[], 5:[], 6:[]}
```

C'est cette méthode qui est utilisée pour décrire un arbre dans le cas du parcours en largeur. Cependant, on peut également définir une classe Arbre telle que :

```
class Arbre:
    def __init__(self,valeur):
        self.gauche = None
        self.droit = None
        self.racine = valeur
# L'arbre (a) est donc défini par :
A = Arbre(3)
A.gauche = Arbre(2)
A.droit = Arbre(7)
A.gauche.gauche = Arbre(1)
A.gauche.droit = Arbre(4)
A.gauche.gauche.gauche = Arbre(6)
A.droit.gauche = Arbre(5)
```

On utilisera cette classe pour les parcours en profondeur.

2.2.1 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 (a) est donc $[3, 2, 7, 1, 4, 5, 6]$. 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. 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 noeud et avoir une boucle infinie dans le cas de graphe cyclique par exemple.

2.2.2 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 $[6, 1, 2, 4, 3, 5, 7]$.

```
def parcours_infixe(arbre):
    if arbre:
        parcours_infixe(arbre.gauche)
        print(arbre.racine)
        parcours_infixe(arbre.droit)
```

- 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 $[6, 1, 4, 2, 5, 7, 3]$.

```
def parcours_suffixe(arbre):
    if arbre:
        parcours_suffixe(arbre.gauche)
        parcours_suffixe(arbre.droit)
        print(arbre.racine)
```

- 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 $[3, 2, 1, 6, 4, 7, 5]$.

```python
def parcours_prefixe(arbre):
    if arbre:
        print(arbre.racine)
        parcours_prefixe(arbre.gauche)
        parcours_prefixe(arbre.droit)
```

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

2.3 Méthodes principales spécifiques aux ABR

2.3.1 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 coté continuer la recherche, il y a 4 cas à traiter:

1. la valeur qu'on recherche se trouve a la racine, dans ce cas on renvoie directement la racine.
2. la valeur qu'on recherche est plus petite que la racine qui veut dire qu'elle est forcement dans le sous-arbre gauche donc on fait un appel récursif sur ce dernier.
3. la valeur est plus grande que la racine on fait donc un appel récursif sur le sous-arbre droit.
4. et finalement si on atteint une feuille et que sa clé n'est pas l'élément qu'on recherche ça veut dire que l'élément ne se trouve pas dans l'arbre.

On peut alors utiliser la fonction récursive suivante:

```python
    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
```

Complexité, terminaison et correction : On fait une preuve détaillée pour la terminaison, la correction et la complexité de cette fonction.

- Complexité : le nombre d'opérations de la recherche dépend de la hauteur de l'arbre. 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. Nous étudierons plus finement la complexité moyenne en fonction de la taille pour le rendu final du projet.

- 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 $T = (y, G, D)$ un ABR 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. $x > y$ ici 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 le 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.

2.3.2 Insertion :

Pour l'insertion on procède de manière analogue à la recherche : pour insérer un élément il faut savoir dans quelle partie de l'arbre le mettre pour que cela reste un arbre binaire de recherche. Pour cela il suffit de comparer la valeur à insérer au noeud 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 noeuds sont distinctes.

2. la valeur à insérer est plus petite que le noeud courant. On a alors deux possibilités :

   - le sous arbre gauche est vide : on procède à l'insertion en créant un noeud avec pour étiquette la valeur à insérer, et des 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 de manière analogue 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, dans le sens où on ne cherche pas à garantir une structure équilibrée. Nous verrons plus tard (c'est prévu pour le rapport final) qu'il est possible de le faire en introduisant la rotation d'arbre. La complexité est en $\mathcal{O}(h)$ avec h la hauteur de l'arbre, comme pour la recherche. La preuve est analogue à la preuve de la méthode recherche.

2.3.3 Suppression :

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

1. si x c'est la racine de notre arbre, c'est un peu subtil, il faut aussi traiter trois cas:
   - si x n'a pas de fils on le supprime directement.
   - si x a un seul fils on le remplace par celui-ci.
   - s'il en a deux on échange le x soit avec le max du sous arbre gauche (qui correspond au fils le plus a droite du sous arbre gauche), soit avec le min du sous arbre droit(qui correspond au fils le plus à gauche du sous arbre droit), puis on supprime le max ou le min d'après ce qu'on a choisit en faisant un appel récursif.
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.

- Complexité :

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 successeur. Les deux dernières opérations peuvent être implémentées en $\Theta(1)$. La suppression a donc, comme les autres opérations, un coût maximal en $\Theta(log_2(n))$ si l'arbre est équilibré.

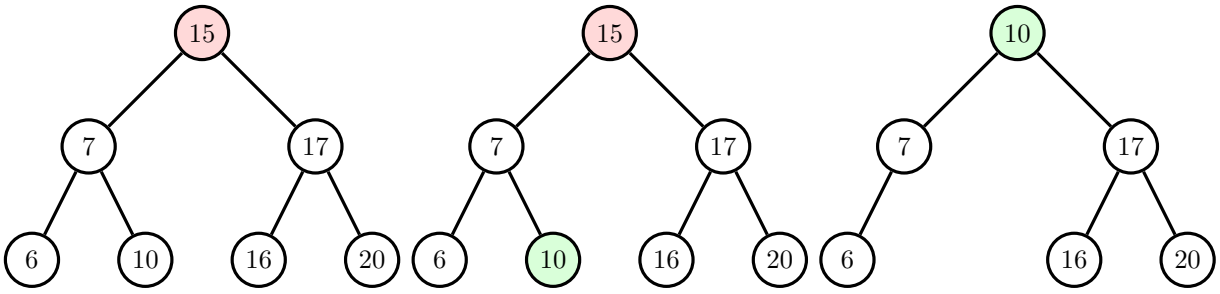


Figure 2.1: Exemple de suppression de la racine : noeud 15

2.4 Complexité des méthodes principales

Dans toute cette section, on notera h la hauteur d'un arbre binaire et n son nombre de noeuds.

2.4.1 Complexité théorique de la fonction de recherche:

Le pire cas pour la fonction de recherche est le cas où l'on doit parcourir l'arbre jusqu'à arriver à une feuille.
Ainsi, la complexité de la fonction de recherche est en O(h) dans le pire cas.

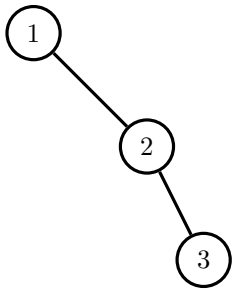
Le cas moyen correspond au cas où l'arbre binaire de recherche est de hauteur logarithmique par rapport à n. En effet, un théorème indique que la hauteur d'un arbre généré aléatoirement sera en moyenne logarithmique par rapport à n. [?]
Ainsi, la complexité de la fonction de recherche est en O($log_2(n)$) dans le cas moyen.

2.4.2 Protocole expérimental:

Pire cas:

Nous avons modélisé le pire cas par des arbres de hauteur égale à leur nombre de sommets.

Graphiquement, cela correspond à des arbres qui ont uniquement des fils gauche ou uniquement des fils droit. Voici un exemple de tel arbre de taille 3:



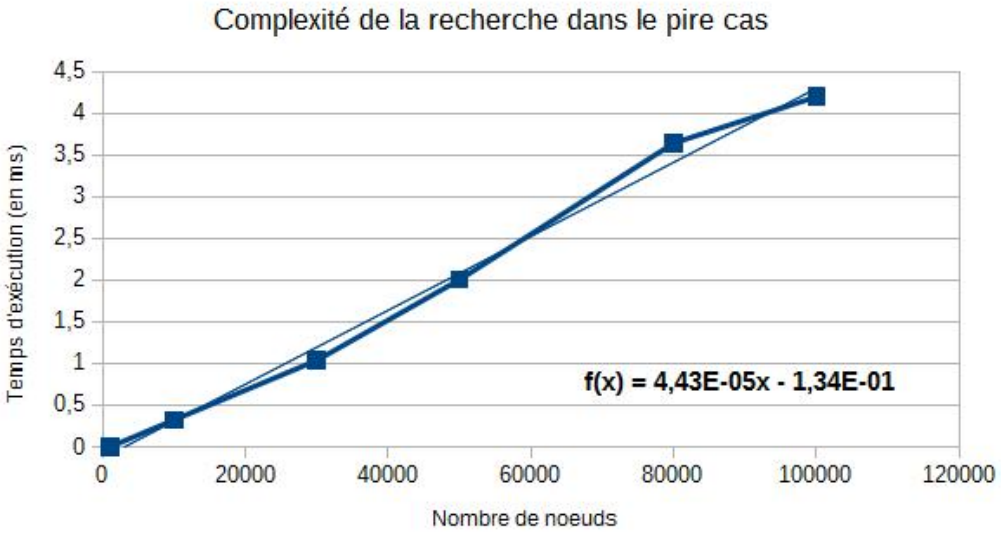
Dans ce cas, nous avons h=n, ainsi, la complexité des fonctions principales devrait être linéaire en n.

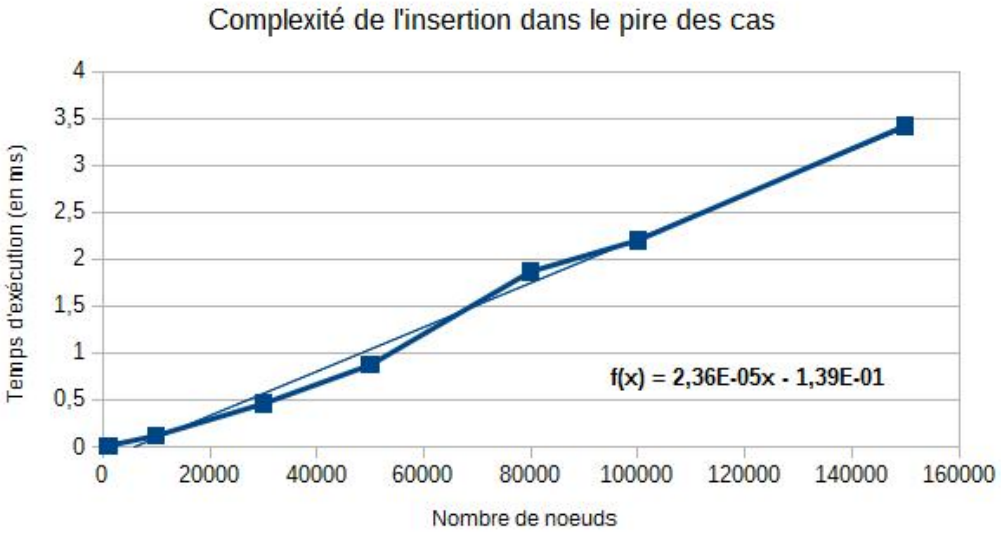
Nous avons utilisé l'implémentation des arbres binaires de recherche en python. Nous avons créé de tels arbres de taille allant de 1000 à 100 000 noeuds. Il nous était impossible d'aller plus haut sans faire crasher python.

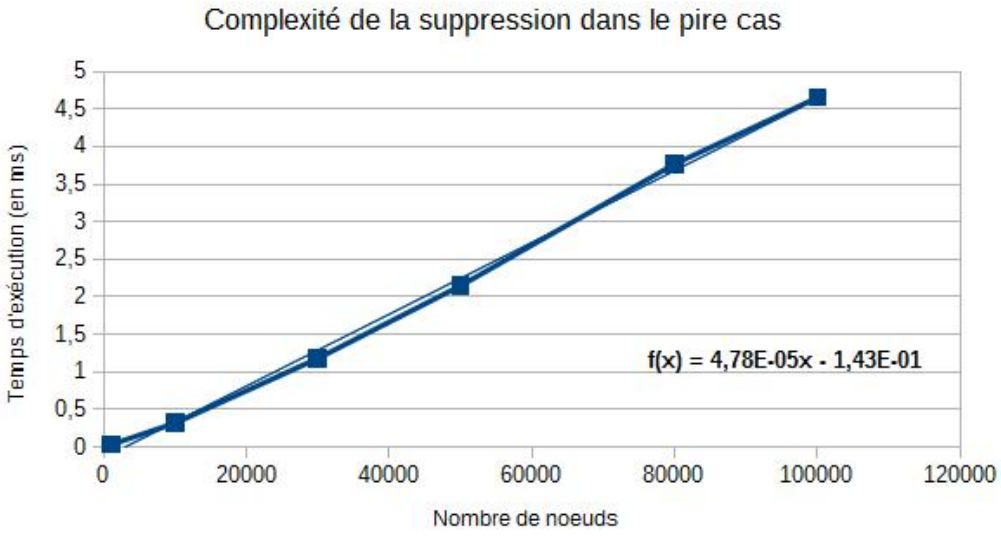
Pour chaque arbre, nous appelons la fonction étudiée sur une feuille de l'arbre (dans ce cas, il s'agit simplement du noeud de plus grande étiquette) et mesurons le temps que cela prend à l'aide du module time.timeit.

Nous répétons cette opération 100 fois et cela nous permet de faire une moyenne pour chaque arbre. Nous pouvons ensuite tracer le temps d'exécution en fonction du nombre de noeuds.

Voici les résultats obtenus :



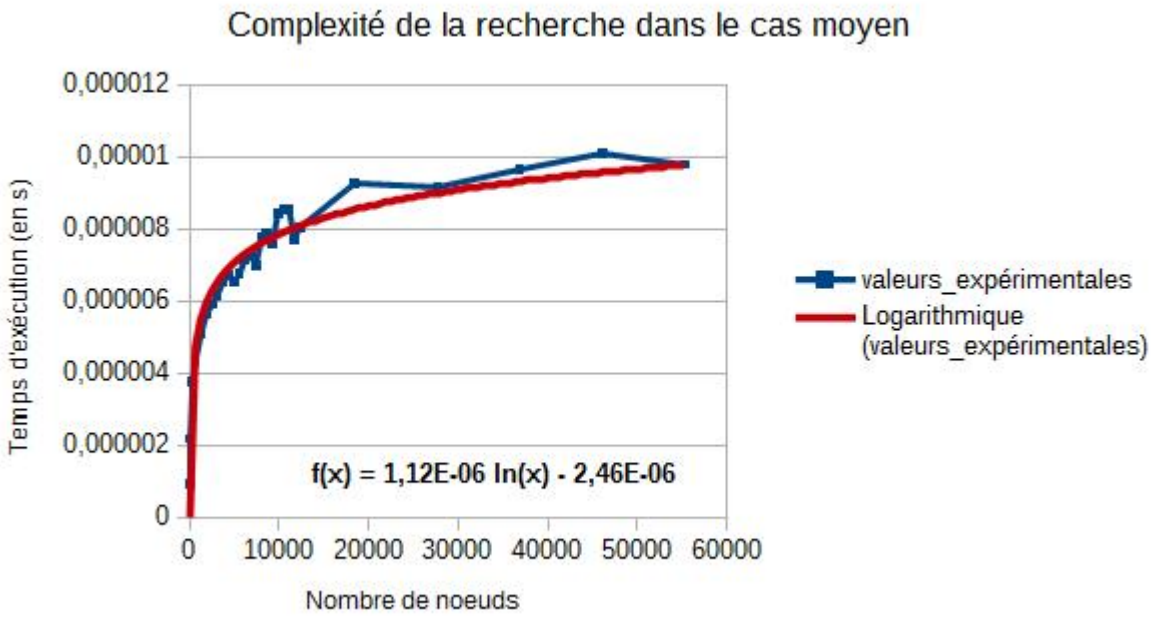


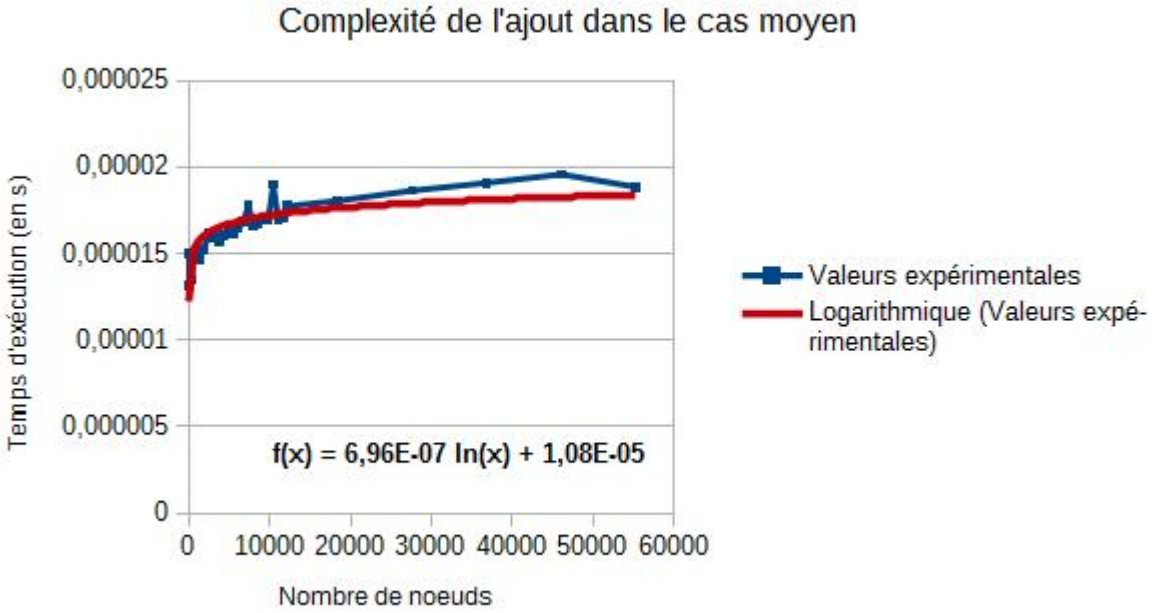


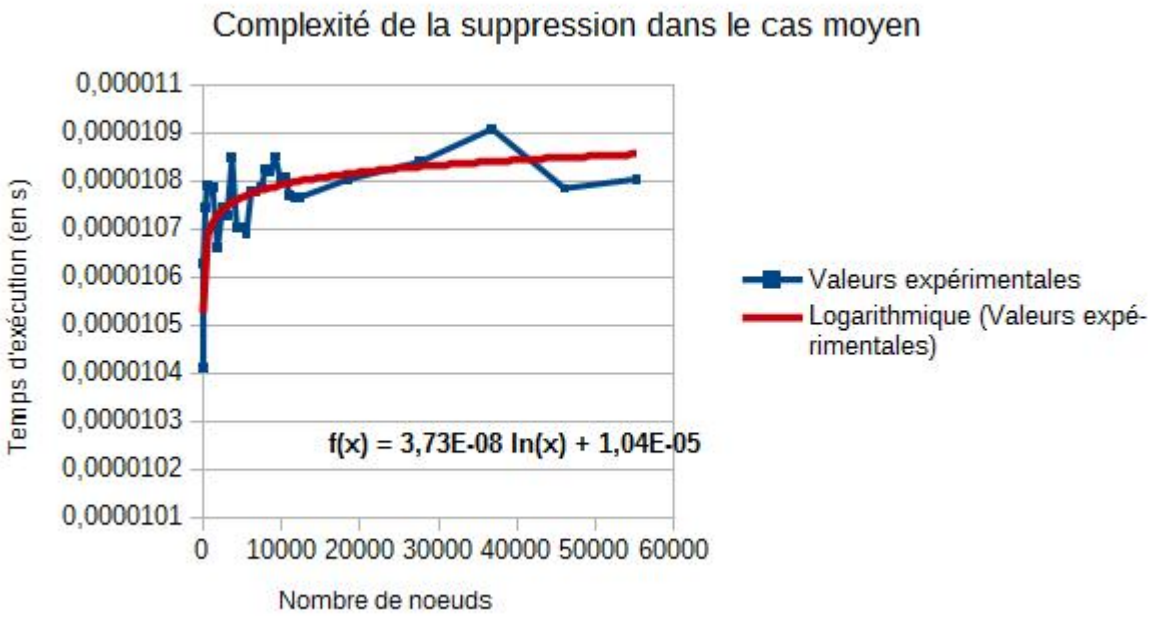
Cas moyen:

Nous avons modélisé le cas moyen par des arbres parfaits car d'après la partie 0.4.2, la hauteur d'un arbre parfait est en $\Theta(log_2(n))$

Voici les résultats que nous avons obtenus en reprenant exactement le même protocole expérimental mais en prenant des arbres parfaits:


Complexité de la recherche dans le cas moyen
Temps d'exécution (en s)
0,000012
0,00001
0,000008
0,000006
0,000004
0,000002
0
f(x) = 1,12E-06 ln(x) - 2,46E-06
0
10000
20000
30000
40000
50000
60000
Nombre de noeuds
valeurs_expérimentales
Logarithmique (valeurs_expérimentales)


Complexité de l'ajout dans le cas moyen
Temps d'exécution (en s)
0,000025
0,00002
0,000015
0,00001
0,000005
0
f(x) = 6,96E-07 ln(x) + 1,08E-05
0
10000
20000
30000
40000
50000
60000
Nombre de noeuds
Valeurs expérimentales
Logarithmique (Valeurs expérimentales)



Les courbes obtenues expérimentalement sont bien logarithmiques. Cependant, nous avons rencontré quelques difficultés dans l'obtention des courbes pour le cas moyen.

La principale difficulté était de réussir à créer des arbres parfaits. Les arbres que nous avons généré pour le cas moyen sont "quasiment" parfaits dans le sens où leur hauteur est faible même s'ils ne sont pas exactement parfaits.

De plus, la création d'arbres parfaits est coûteuse en temps et en mémoire: Il n'est pas possible d'en créer des trop gros sans provoquer un dépassement de capacité de la mémoire. Nous pourrons éventuellement tenter de créer des arbres binaires de grande taille en ajoutant des éléments aux arbres binaires de petite taille mais cela nécessiterait que nous prenions des étiquettes réelles et non plus entières.

Chapter 3

Implémentation : comparaison entre structure récursive et représentation en liste

3.1 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 on peut voir un arbre binaire comme une structure contenant essentiellement 3 champs : une clef et deux arbres binaires, ses sous-arbres gauche et droit.

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), on créé 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 sur l'utilisation.

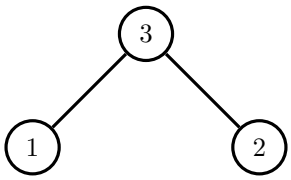
On pourra déclarer la classe comme ceci :

```
1 class ArbreBinaire():
2     def __init__(self,clef,gauche = None ,droite = None):
3         """ Anything x ArbreBinaire x ArbreBinaire -> ArbreBinaire
4         Constructeur d'arbre"""
5         self.clef = clef
6         self.gauche = gauche
7         self.droite = droite
```

Exemple d'instanciation :

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

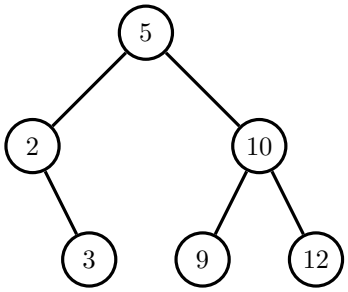
L'arbre créé est :



3.2 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. 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 faut utiliser une structure de base qui combine les avantages de la liste chaînée et des tableaux : nous avons besoins d'accéder à 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.

3.3 Comparaison

3.3.1 Avantages et inconvénients

Pour 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.

Pour 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 : pour la structure récursive elle est plus lourde à mettre en place, ça peut être lourd en mémoire pour des arbres pleins. Pour la liste, le code sera peut être moins lisible et surtout on gâche de la mémoire lorsque l'arbre contient des nœuds vide.

Dans certains cas la liste est particulièrement appropriée par exemple pour la représentation des tas (arbre binaire parfait sur un ensemble totalement ordonnée où les chemins de la racine vers les feuilles sont toujours croissant), utilisés dans le tri par tas où le but est d'ordonner une liste.

Pour notre implémentation nous avons choisi la représentation en structure récursive en utilisant la programmation orientée objet en Python.

Conclusion

Les arbres binaires de recherche sont donc une structure utile pour stocker des données que l'ont peut munir d'une relation d'ordre total, comme des entiers, mais on peut aussi penser à des lettres et donc des mots. En effet dans une telle structure les opérations classiques de recherche, d'insertion et de suppression ont une complexité moyenne logarithmique par rapport au nombre d'élements, les mesures expérimentales ayant confirmé l'étude théorique. Nous verrons par la suite que les quadtree conservent ces propriétés. Nous étudierons les différents types de quadtree (quadtree point et quadtree région notamment) et leurs applications pratiques.

Bibliography

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

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

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

[4] 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].

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