# Quadtree

Tout le monde

February 21, 2019

# Contents



# 1 Implémentation : comparaison entre structure récursive et représentation en liste

## 1.1 Structure récursive

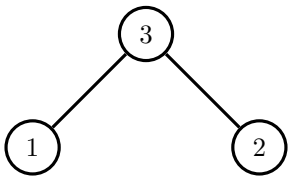
Une représentation qui semble la plus naturelle est d'implémenter une structure avec auto-référence. En effet en s'inspirant de la définition inductive on peut voir un arbre binaire comme une structure contenant essentiellement 3 champs : une clef et deux arbres binaires, ses sous-arbres gauche et droit.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

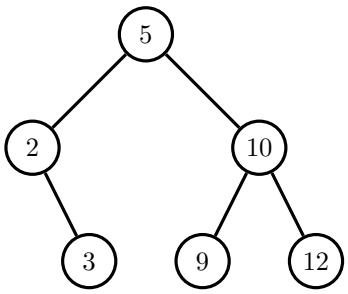
```
1 Feuille1 = ArbreBinaire(1)
2 Feuille2 = ArbreBinaire(2)
3 Arbre = ArbreBinaire(3,Feuille1,Feuille2)
```

L'arbre créé est :




## 1.2 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 : $[5, 2, 10, "vide", 3, 9, 12]$

Remarquons qu'idéalement il faut utiliser une structure 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.

## 1.3 Comparaison

### 1.3.1 Avantages et inconvénients

Pour la structure récursive :

- Lisibilité du code
- Gain mémoire pour les arbres avec des noeuds vides : on n'insère que les noeuds non vide dans l'abre
- Plus grande souplesse dans le type de donnée représentée dans le noeud

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 noeuds vide.