Les arbres peuvent comme les tableaux servir de moyen efficace pour stocker des données. Cependant se pose la question de la recherche et de l'insertion d'éléments dans ces derniers.

La recherche ou l'insertion dans un arbre peut toujours être faite en un coup linéaire en la taille de celui-ci (il suffit de le parcourir) mais à l'instar de la recherche par dichotomie dans un tableau trié, nous allons voir qu'il est possible de le faire plus rapidement si celui-ci est en un certain sens **trié** et **équilibré**.

Le caractère trié d'un arbre se traduit par la notion d'arbre binaire de recherche (ABR) sujet de ce chapitre. On parlera rapidement de l'équilibrage sans trop rentrer dans les détails.

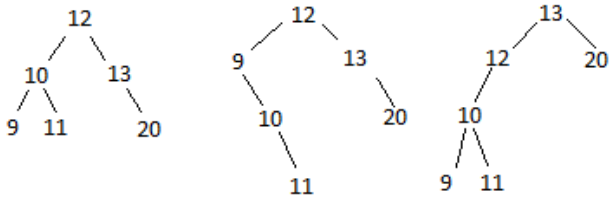
1 Définition et exemples

On considère un type `T` muni d'un opérateur de comparaison $\leq$.

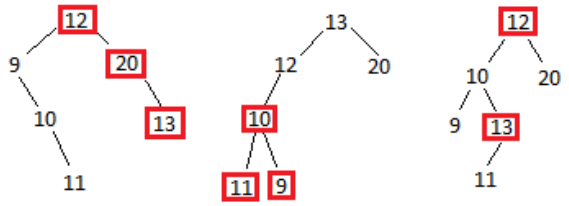
Un **arbre binaire de recherche** (ABR) de type `T` est un arbre binaire dont les valeurs sont de type `T` et dont chaque noeud vérifie les propriétés suivantes :

— chaque valeur contenue dans le sous arbre gauche est plus petite que la valeur du noeud ;
— chaque valeur contenue dans le sous arbre droit est plus grande que la valeur du noeud ;

Exemples :



Contre-exemples :

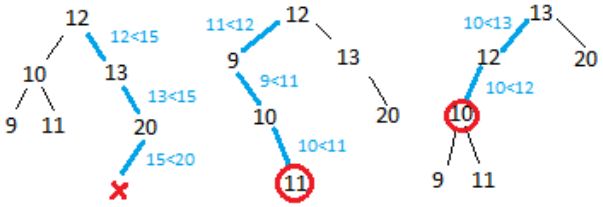


2 Recherche et insertion

Recherche : Étant donné un ABR de type `T`, on dispose de par sa propriété d'un algorithme de recherche simple.

On recherche une valeur `x` dans un ABR `a` :
— si `a` est vide, `x` n'est pas dans `a` ;
— sinon, soit `v` la valeur de la racine de `a` :
  — si `x==v` alors `x` est bien dans `a` ;
  — si `x<v` alors `x` est dans `a` ssi `x` est dans le sous arbre gauche de `a` ;
  — si `v<x` alors `x` est dans `a` ssi `x` est dans le sous arbre droit de `a` ;

Exemples :



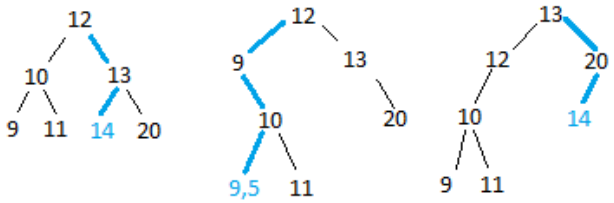
On en déduit la fonction suivante :

```
def recherche_ABR(a, x):
    #recherche de x dans l'ABR a
    if a is None :
        return False
    else :
        v = a.valeur
        if x == v:
            return True
        if x<v:
            return recherche_ABR(a.gauche, x)
        else:
            return recherche_ABR(a.droite, x)
```

Insertion : Étant donné un ABR de type `T` auquel on veut ajouter une valeur `x`, il est important de conserver la propriété des ABR (cf définition).

Insérons un élément `x` dans un ABR `a` :
- si `a` est vide, on renvoie le noeud sans sous arbre et de valeur `x` ;
- sinon, soit `v` la valeur de la racine de `a` :
  - si `x<v` alors `x` doit être inséré dans le sous arbre gauche de `a` ;
  - si `v<x` alors `x` doit être inséré dans le sous arbre droit de `a` ;
  - si `x==v` alors l'insertion dépend du choix de l'implémentation. On pourrait choisir selon les situations de l'insérer à droite par défaut, de l'insérer à gauche par défaut voire même de ne pas l'insérer (pour éviter les doublons).

Exemples :



On en déduit la fonction suivante :

```
def insertion_ABR(a, x):
    #insertion de x dans l'ABR a, droite en cas d'egalite
    if a is None :
        return Noeud(None, x, None)
    else :
        v = a.valeur
        if x<v:
            return Noeud(insertion_ABR(a.gauche, x), v, a.droite)
        else:
            return Noeud(a.gauche, v, insertion_ABR(a.droite, x))
```

3 Complexité et arbres équilibrés

La complexité des algorithmes d'insertion et de recherche dans un ABR est dans le pire cas linéaire en la hauteur de l'arbre. En effet, ces deux algorithmes consistent à suivre un chemin dans l'arbre jusqu'à une feuille.

On a donc une complexité en $\mathcal{O}(h)$.

Cependant, la grandeur pertinente décrivant la quantité de données présentes dans un ABR est plutôt sa taille N. Essayons d'exprimer h en fonction de N en utilisant l'inégalité vue au chapitre A3 :

$$h \leq N \leq 2^h - 1$$

La partie $h \leq N$ peut être utilisée telle quelle. La partie $N \leq 2^h - 1$ donne lieu à l'inégalité :

$$h \leq N \leq 2^h - 1 \leq 2^h$$

En en déduit donc en appliquant $\log_2$ fonction croissante sur $\mathbb{R}_+^*$:

$$\log_2(N) \leq \log_2(2^h) = h$$

Finalement, on obtient l'encadrement suivant :

$$\log_2(N) \leq h \leq N$$

Ce qui nous indique qu'en fonction de la forme de l'arbre, la complexité de nos algorithmes est dans le meilleur cas logarithmique $\mathcal{O}(\log_2(N))$ et dans le pire cas linéaire $\mathcal{O}(N)$.

Pour garantir que ces algorithmes fonctionnent le plus efficacement possible, on s'intéresse donc arbres tels que h soit le plus proche possible de $\log_2(N)$: les **arbres équilibrés**.

Il existe plusieurs notions plus ou moins lâches d'arbres équilibrés :

— on connaît déjà les arbres parfaits, réalisant l'égalité $N = 2^h - 1$;

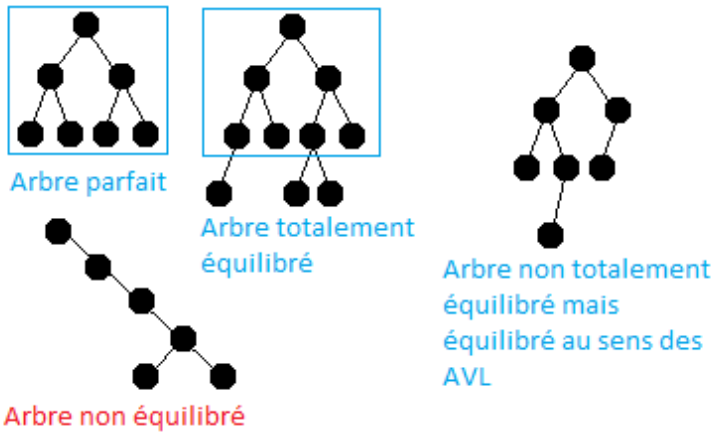
— les arbres totalement équilibrés sont des arbres vérifiant l'inégalité :

$$2^{h-1} \leq N \leq 2^h - 1$$

Ils correspondent à des arbres entièrement remplis jusque la hauteur $h-1$. Cette notion reste assez rigide mais permet de ne pas éliminer les arbres dont le nombre de noeud les empêche arithmétiquement d'être parfaits.

— en pratique, on utilise des critères d'équilibrage partiels, encore un peu moins rigide, par exemple en imposant $h \leq 2log_2(N+1)$.

Exemples :



En pratique, losrqu'on remplit un ABR, on essaie donc en plus de l'insertion de maintenir un certain niveau d'équilibrage de celui-ci afin que la complexité des algorithmes reste logarithmimique.

Les méthodes garantissant cet équilibrage après insertion ne sont pas au programme du lycée. Pour aller plus loin, vous pouvez consulter les pages Wikipedia des deux méthodes classiques :

— les arbres AVL ;

— les arbres rouge-noir.