Arbres binaires de recherche (ABR)

ATTENTION : NE LIRE CE CORRIGE QU'APRES AVOIR FAIT LE TP SUR LES Abrbres Binaires de Recherche

Introduction

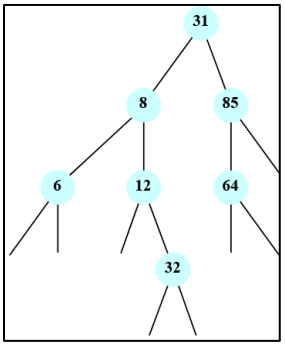
Les arbres binaires de recherche permettent de stocker des données de façon performante tant pour l'insertion que pour la recherche d'éléments.

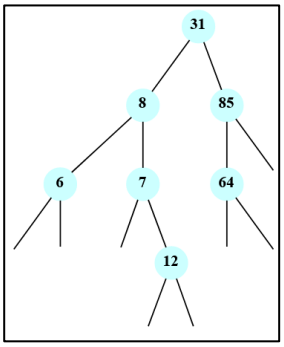
I : Présentation, exemples

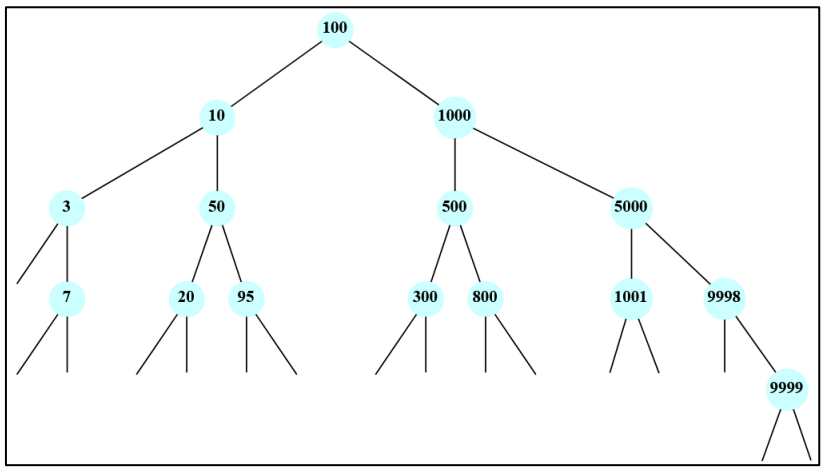
Un arbre binaire de recherche est un arbre binaire étiqueté dont les valeurs peuvent être ordonnées (par exemple des entiers ou des chaînes de caractères) et vérifiant la propriété suivante :
Pour tout nœud de l'arbre binaire :

- toutes les valeurs du sous-arbre gauche sont plus petites que la valeur portée par le nœud,
- toutes les valeurs du sous-arbre droit sont plus grandes que la valeur portée par le nœud.

Exemple et contre-exemples :







L'arbre représenté à droite est un arbre binaire de recherche :

- Pour le nœud de valeur 100, le sous-arbre gauche ne contient que des valeurs inférieures à 100, et le sous-arbre droit ne contient que des valeurs supérieures à 100.
- Pour le nœud de valeur 10, c'est la même chose.
- Pour le nœud de valeur 1000, c'est la même chose.
- etc.

En revanche, les deux de gauche ne sont pas des arbres binaires de recherche (entourer les paires de nœuds qui posent problème).

Propriété : Le parcours infixe d'un arbre binaire de recherche parcourt les valeurs de l'arbre par ordre croissant.

Justification de la propriété : Le parcours infixe consiste à d'abord explorer le sous-arbre gauche (dont les valeurs sont les plus petites) puis à explorer la racine et enfin à explorer le sous-arbre droit (dont les valeurs sont les plus grandes).

Principe de fonctionnement de la recherche et de l'insertion dans les arbres binaires de recherche :
On souhaite insérer la valeur 777 dans l'arbre binaire de recherche ci-dessus à droite. On part du nœud racine 100.
Puisque $777 > 100$, on sait qu'il faudra l'insérer dans le sous-arbre droit dont le nœud vaut 1000.
Puisque $777 < 1000$, on sait qu'il faudra l'insérer dans le sous-arbre gauche dont le nœud vaut 500.
Puisque $777 > 500$, on sait qu'il faudra l'insérer dans le sous-arbre droit dont le nœud vaut 800.
Puisque $777 < 800$, on sait qu'il faudra l'insérer dans le sous-arbre gauche … qui est vide donc on le remplace par un nœud de valeur 777.

On peut naturellement exprimer cela sous forme récursive. Soit e un élément à insérer dans un ABR a :

Si $e \leq a.valeur$ alors insérer e dans $a.gauche$, sinon insérer e dans $a.droit$.

Cet algorithme d'insertion (et ce serait le même principe pour une recherche) a une complexité liée à la hauteur de l'arbre ce qui donne :

> Pour contruire un arbre binaire de recherche il suffit de disposer d'une structure d'arbre binaire vide et d'y insérer au fur et à mesure des valeurs en suivant un algorithme récursif intuitif. Les opérations de recherche d'un élément s'effectuent selon le même principe algorithmique.
>
> En suivant ce principe, insertion et recherche sont en complexité $O(h)$ où h est la hauteur de l'arbre.

II : Algorithme de recherche et d'insertion en langage Python

On suppose qu'un arbre est implémenté grâce à une classe `Noeud` disposant uniquement des attributs `valeur`, `gauche` et `droit`.

Algorithme d'insertion

```
def inserer(element, arbre):
    if arbre == None :
        return Noeud(element, None, None)
    elif element < arbre.valeur :
        return Noeud(arbre.valeur, inserer(element, arbre.gauche), arbre.droit )
    elif element > arbre.valeur :
        return Noeud(arbre.valeur, arbre.gauche, inserer(element, arbre.droit))
    else:
        return arbre
```

Algorithme de recherche

```
def rechercher(element, arbre):
    if arbre == None :
        return False
    elif element < arbre.valeur :
        return rechercher(element, arbre.gauche)
    elif element > arbre.valeur :
        return rechercher(element, arbre.droit)
    else:
        return True
```

Remarque 1 : ici l'insertion d'un élément déjà présent dans le tableau …

Remarque 2 : il peut être utile de disposer d'une fonction permettant de transférer le contenu d'un tableau dans un arbre binaire de recherche (par exemple pour effectuer un tri du tableau : voir en Exercices)

Remarque 3 : Nous avons vu en TP/TD comment encapsuler ces fonctionnalités dans une classe `ABR`.

III : Recherche d'une valeur dans une liste, un dictionnaire ou un ABR

L'efficacité du traitement de données est crucial lorsqu'on manipule des structures de données contenant plusieurs milliers ou millions d'informations, a fortiori lorsque ces manipulations sont extrêmement fréquentes (pensons par exemple à certaines données accessibles via internet qui peuvent être manipulées par des milliers d'utilisateurs).

Ici on s'intéresse à la recherche ou à l'insertion d'un couple (clef, valeur) dans une SDD connaissant sa clef.
Il est temps de faire un point sur ce que nous avons rencontré jusqu'ici :

Structure	Insertion (pire des cas)	Recherche (pire des cas)
Liste	O(1)	O(n)
ABR	O(n)	O(n)
ABR *équilibré*	O(log N)	O(log N)
Dictionnaire (via table de hachage)	O(1) en moyenne	O (1) en moyenne

Remarque : nous ne mentionnons pas le cas des tableaux car en Python les tableaux sont très spécifiques comparés aux autres langages.

La notion d'arbre binaire *équilibré* (voir la fin du TP) n'est pas au programme mais il est impératif de savoir ce que cela signifie puisque ce sont ces arbres *équilibrés* qui justifient très souvent l'utilisation d'arbres binaires. Vous devez donc retenir qu'en assurant une insertion "rusée" des valeurs dans un arbre binaire de recherche, on peut garder une hauteur d'arbre binaire minimale (ou presque).

Dit autrement, pour un arbre binaire de recherche construit sans précaution on a :

$$\log_2 N < h \leq N$$

Puisque insertion et recherche sont en $O(h)$, on a donc dans le pire des cas une complexité en $O(N)$.

A contrario pour un arbre binaire de recherche construit précautionneusement pour le garder équilibré on a :

$$h = O(\log_2 N)$$

Et donc une complexité en $O(h) = O(\log_2 N)$ pour insertion et recherche.