

# Compte-rendu d'Algorithmique

## Les Arbres Binaires de Recherche (ABR)

### Introduction

La définition d'un type abstrait consiste à rappeler les opérations qu'on peut effectuer sur un ensemble de données pour manipuler ce type abstrait. En général, un type abstrait est composé de cinq champs, on cite par exemple les opérations, les pré-conditions et les axiomes qui contiennent un tas d'axiomes qui expliquent le comportement des opérations décrivant un type abstrait.

Un exemple des types abstraits dans lequel on va s'intéresser dans notre projet c'est les arbres binaires de recherche (ABR).

Un arbre binaire de recherche est un arbre dont les sommets ont des valeurs comparables et vérifiant :

- Chaque sommet a au plus deux fils (arbre binaire).
- Pour chaque sommet (s), les sommets dans le sous-arbre gauche de (s) ont tous une valeur inférieure ou égale à celle de (s) et ceux dans le sous-arbre droit ont tous une valeur strictement supérieure à celle de (s).

Les ABR réalisent de manière efficace les opérations de recherche, insertion et suppression.

Dans ce rapport nous allons donc décrire ces opérations (insertion, suppression, recherche).

# Opérations

## I. Insertion des éléments :

### I.1) Principe

Le principe d'insertion d'un nœud peut s'exprimer ainsi :

- Si la racine est non nul alors
  - Si la clé est plus petite que le nœud n, on remplace le nœud n par son fils gauche et on fait appel à cet l'algorithme d'une façon récursif sur ce nouvel nœud n.
  - Sinon si la clé est égale au nœud on n'insère rien puisque l'élément existe déjà.
  - Sinon on remplace le nœud n par son fils droit et on fait appel à cet l'algorithme d'une façon récursif à partir de ce nouvel nœud n.
- Sinon on insère directement.

### I.2) Algorithme

VARIABLES : racine (Nœud), cle (entiers) et valeur (chaîne de caractères)

Entrée : un arbre d'éléments

sortie : le même arbre d'éléments (Arbre) avec un élément inséré ou pas dans le cas où l'élément existe déjà dans l'arbre

DEBUT

INSERER(racine, cle, valeur) :

```

    SI racine est non nul
        SI cle < racine.cle
            SI racine.gauche est non nul
                inserer(racine.gauche, cle, valeur)
            SINON
                créer un nœud nouveauNoeud contenant (cle, valeur)
                nouveauNoeud.pere → racine
                racine.gauche → nouveauNoeud
            FINSI
        SINON SI cle > racine.cle
            SI racine.droit est non nul
                inserer(racine.droit, cle, valeur)
            SINON
                créer un nœud nouveauNoeud contenant (cle, valeur)
                nouveauNoeud.pere → racine
                racine.droit → nouveauNoeud
            FINSI
        SINON SI cle égale à racine.cle
            afficher « la clé existe déjà »
        FINSI
    SINON
        créer un nœud nouveauNoeud contenant (cle, valeur)
        arbre.racine → nouveauNoeud
    FINSI

```

FIN

## II. Affichage d'un arbre :

### II.1) Principe

- Si le nœud est non nul :
  - Appel récursif de la méthode d'affichage sur le fils gauche du nœud actuel.
  - Affichage de la clé, la valeur et le père du nœud actuel
  - Appel récursif de la méthode d'affichage sur le fils droit du nœud actuel

Cette méthode permet d'afficher l'arbre depuis le fils le plus à gauche jusqu'au fils le plus à droite.

On a ajouté l'affichage du père pour chaque nœud afin de mieux comprendre l'architecture de l'arbre.

### II.2) Algorithme

VARIABLES : racine (Nœud)

Entrée : un arbre d'éléments

sortie : l'affichage des éléments de l'arbre

DEBUT

AFFICHER(racine) :

```

    SI racine est non nul
        afficher(racine.gauche)
        SI racine.pere est non nul
            on affiche le nœud et la clé de son père

        SINON
            on affiche le nœud seul
    FINSI
    afficher(racine.droit)

```

FINSI

FIN

## III. Rechercher un élément :

### III.1) Principe

La recherche d'un nœud peut être définie de manière récursive, Cette méthode devra nous indiquer si un élément avec une clé et une valeur est présent dans l'arbre.

- Si la clé recherchée est celle de la racine alors la recherche est terminée on a trouvée la clé et on retourne la racine.
- Sinon si la clé recherchée est plus petite que celle de la racine la recherche se fera à gauche, on remplace le nœud racine par son nœud fils de gauche et que l'on relance la procédure de recherche à partir de ce nouveau nœud racine.
- Sinon si la clé recherchée est plus grande que celle de la racine la recherche se fera à droite. On remplace racine par son nœud fils droit et que l'on relance la procédure de recherche.
- Sinon la clé n'existe pas et on ne retourne rien.

La récursivité ne sera interrompue que si l'élément recherché est trouvé.

Si l'arbre est parcouru en totalité sans trouver l'élément recherché, la méthode retourne nul

### III.2) Algorithme

VARIABLES : racine (Nœud), cle (entiers)

Entrée : un arbre d'éléments

sortie : le même arbre sans changement

résultat : retourner le nœud entré en paramètre si présent

DEBUT

RECHERCHER(racine, cle) :

```

    SI racine est non nul
        SI cle < racine.cle
            retourner rechercher(racine.gauche, cle)

        SINON SI cle > racine.cle
            retourner rechercher(racine.gauche, cle)

        SINON
            retourner nil

    FINSI
    SINON
        retourner nil
    FINSI

```

FIN

## IV. Suppression d'un élément :

### IV.1) Principe

La suppression d'un élément dans un arbre est une opération plus complexe, Elle se fait selon le principe suivant :

- Si l'arbre est non vide

- Si le nœud est une feuille (il n'a pas de fils), alors on l'élimine directement en supprimant la liaison du nœud avec son père.
- Sinon si le nœud possède un seul fils, on remplace le nœud par son fils, on obtient directement un arbre binaire de recherche.
- Sinon le nœud possède deux fils, plusieurs solutions sont possibles. La méthode consiste à remplacer le nœud qu'on souhaite supprimer par une valeur qui est déjà dans l'arbre, il faut remplacer la valeur de ce nœud, soit avec la plus grande valeur à gauche, soit avec la plus petite valeur à droite.

## IV.2) Algorithme

VARIABLES : racine (Nœud), cle (entiers) et valeur (chaîne de caractères)

Entrée : un arbre d'éléments

sortie : le même arbre d'éléments (Arbre) avec un élément supprimé ou pas dans le cas où l'élément n'existe pas dans l'arbre

DEBUT

SUPPRIMER(racine, cle, valeur) :

```

    SI racine est non nul
        SI cle < racine.cle
            supprimer(racine.droit, cle, valeur)
        SINON SI cle > racine.cle
            supprimer(racine.droit, cle, valeur)

        SINON SI cle égale à racine.cle
            SI racine.gauche est nul et racine.droit est nul
                SI racine.pere.cle > cle
                    casser le lien avec le fils gauche de racine.pere
                SINON
                    casser le lien avec le fils droit de racine.pere
            FINSI
        SINON SI racine.gauche est nul
            SI racine.pere.cle > cle
                mettre le fils droit du racine à la place du fils
                gauche du père du racine
            SINON
                mettre le fils droit du racine à la place du fils
                droit du père du racine
            FINSI
        SINON SI racine.droit est nul
            SI racine.pere.cle > cle
                mettre le fils gauche du racine à la place du fils
                gauche du père du racine
            SINON
                mettre le fils gauche du racine à la place du fils
                droit du père du racine
            FINSI
        SINON
            récupérer le nœud du dernier fils gauche dans un variable tmp
            récupérer la clé du tmp dans un variable clef
            recupere l'élément du tmp dans un variable element
            supprimer(racine, clef, valeur)
            mettre la variable element dans racine

    FINSI

```

FINSI

FIN

## V. Calculer la hauteur d'un arbre

### V.1) Principe

La hauteur d'un arbre est le nombre maximum de sommets sur un chemin allant de la racine à une feuille.

On peut définir la hauteur de manière récursive : la hauteur d'un arbre est le maximum des hauteurs de toutes les branches. C'est à partir de cette définition que nous pourrions exprimer un algorithme de calcul de la hauteur de l'arbre.

Un arbre vide est de hauteur 0

Un arbre non vide a pour hauteur le plus grands chemin qui lie le racine avec une feuille.

### V.2) Algorithme

VARIABLES : racine (Nœud), hauteurDroite, hauteurGauche (entier)

Entrée : un arbre d'éléments

sortie : le même arbre d'éléments

résultat : retourner la hauteur de l'arbre

DEBUT

HAUTEUR(racine) :

```

    SI racine est non nul
        hauteurGauche → hauteur(racine.gauche)
        hauteurDroite → hauteur(racine.droite)

        retourner 1 + max( hauteurGauche, hauteurDroite)
    SINON
        retourner -1
    FINSI

```

FIN

## Temps moyen d'exécution pour l'arbre binaire

### I. Mesure des temps d'exécution des opérations

Dans la partie test de notre projet, on a travaillé sur 3 étapes entre insertion et suppression

La première étape : insertion de 10000 éléments à partir d'un tableau d'éléments générés aléatoirement. Le temps d'exécution pour cette étape est de 296 millisecondes.

La deuxième étape : insertion de 1000 éléments à partir d'un tableau d'éléments générés aléatoirement. Le temps d'exécution pour cette étape est de 33 millisecondes.

La troisième étape : suppression de 1000 éléments à partir du tableau ASUPPRIMER qu'on a créé en utilisant le tableau d'insertion. Le temps d'exécution pour cette étape est de 15 millisecondes

## **II. Le temps moyen pour chaque opération**

En utilisant les résultats des temps d'exécution des étapes précédentes, on en déduit que :

Le temps moyen pour une insertion :  $33 / 1000 = 0,033$  ms

Le temps moyen pour une suppression :  $15 / 1000 = 0,015$  ms

## **Temps moyen d'exécution pour la liste chaînée**

### **I. Mesure des temps d'exécution des opérations**

Dans la partie test de notre projet, on a travaillé sur 3 étapes entre insertion et suppression

La première étape : insertion de 10000 éléments en utilisant la méthode ajouteTete() à partir d'un tableau d'éléments générés aléatoirement. Le temps d'exécution pour cette étape est de 7 millisecondes.

La deuxième étape : insertion de 1000 éléments en utilisant la méthode ajouteTete() à partir d'un tableau d'éléments générés aléatoirement. Le temps d'exécution pour cette étape est de 1 millisecondes.

La troisième étape : suppression de 1000 éléments en utilisant la méthode retireTete() à partir du tableau ASUPPRIMER qu'on a créé en utilisant le tableau d'insertion. Le temps d'exécution pour cette étape est de <1 millisecondes

## **II. Le temps moyen pour chaque opération**

En utilisant les résultats des temps d'exécution des étapes précédentes, on en déduit que :

Le temps moyen pour une insertion :  $1 / 1000 = 0,001$  ms

Le temps moyen pour une suppression :  $< 0,001$  ms

The screenshot shows an IDE with a project named 'Liste Chaînée'. The code in 'Main.java' defines a linked list structure and performs a series of operations. The output in the 'Run' console shows the results of these operations, including the insertion and deletion of 1000 elements and the calculation of average execution times for these operations.

```

tempFin = System.currentTimeMillis();
float temps2 = (tempsFin - tempsDebut);
float tempsMoyenInsertion = temps2 / 1000;
// -> temps d'arrivée

==>860272 -> [TFP] ==>96768 -> [QDV] ==>671827 -> [ZCJ] ==>17756 -> [TPE] ==>774571 -> [QIK] ==>388403 -> [JOY] ==>989085 -> [IEH] ==>943111 -> [XCE] ==>72593 -> [YWB]
==>478586 -> [HMK] ==>89083 -> [LVV] ==>128452 -> [SRV] ==>993861 -> [DGR] ==>617360 -> [CQJ] ==>486159 -> [ZDI] ==>546469 -> [PBV] ==>128816 -> [FEE] ==>756961 -> [BLS]
==>319034 -> [UAF] ==>171938 -> [DSZ] ==>71517 -> [ORP] ==>270487 -> [BRV] ==>530983 -> [ZAY] ==>286741 -> [SCC] ==>570416 -> [RCE] ==>435068 -> [MQB] ==>594828 -> [NPU]
==>342489 -> [BRL] ==>709782 -> [WGL] ==>964498 -> [ZYN] ==>733334 -> [JYE] ==>349956 -> [QJP] ==>770657 -> [DLA] ==>3812 -> [MWN] ==>970408 -> [LFD] ==>285323 -> [SCA]
==>39866 -> [QHR] ==>849728 -> [VFI] ==>366344 -> [GEG] ==>277116 -> [UXD] ==>822986 -> [NYD] ==>210773 -> [MKM] ==>388838 -> [XUL] ==>281055 -> [MYP] ==>589245 -> [SEX]
==>632759 -> [TOL] ==>579042 -> [BLV] ==>814830 -> [YQL] ==>934301 -> [LYZ] ==>737882 -> [NAH] ==>178650 -> [FYF] ==>166874 -> [XMF] ==>436896 -> [HUH] ==>532972 -> [WEI]
==>172356 -> [FKG] ==>129784 -> [HUR] ==>991703 -> [HWO] ==>517988 -> [TXW] ==>412863 -> [ACJ] ==>444744 -> [JHH] ==>143212 -> [CIL] ==>334887 -> [PHP] ==>93961 -> [PNB]
==>878557 -> [CDP] ==>308510 -> [XEC] ==>666674 -> [IDL] ==>272588 -> [EMN] ==>345332 -> [QFR] ==>190355 -> [AGC] ==>924202 -> [UTD] ==>77251 -> [FWW] ==>97251 -> [RBQ]
==>813778 -> [SGJ] ==>608632 -> [GDE] ==>756699 -> [RBH] ==>185386 -> [MJC] ==>586804 -> [TFD] ==>87657 -> [NUA] ==>377984 -> [IFJ] ==>951791 -> [FFE] ==>377981 -> [ISN]
==>396731 -> [TSV] ==>756574 -> [ZRJ] ==>812762 -> [WIH] ==>142788 -> [VIC] ==>102154 -> [XZV] ==>219848 -> [IZZ] ==>583865 -> [ZIR] ==>645468 -> [ODY] ==>780513 -> [AIU]
==>626177 -> [XLK] ==>174967 -> [EES] ==>246790 -> [HYH] ==>108837 -> [BAB] ==>952541 -> [CIZ] ==>44765 -> [MKE] ==>494644 -> [EZC] ==>567140 -> [QSV] ==>185611 -> [SOV]
==>321138 -> [NNO] ==>505828 -> [ICG] ==>378372 -> [TVB] ==>756666 -> [HAT] ==>548694 -> [YPK] ==>811879 -> [OMX] ==>759311 -> [CAT] ==>899294 -> [XLQ] ==>49666 -> [LAS]
==>886788 -> [TXO] ==>911458 -> [AQN] ==>281238 -> [LBO] ==>525388 -> [DZB] ==>465757 -> [OLT] ==>697790 -> nil

Etape 2 : Insertion 1000 éléments supplémentaires :

Etape 3 : Suppression de 1000 elements
Opération effectuée pour la fonction insérer() de 10000 en: 8.0 millisecondes.
Opération effectuée pour la fonction insérer() de 1000 en: 1.0 millisecondes.
Opération effectuée pour la fonction supprimer() de 1000 en: 0.0 millisecondes.

-----
le temps moyen pour la fonction insérer() en: 8.0E-4 millisecondes
le temps moyen pour la fonction insérer() en: 0.001 millisecondes.
le temps moyen pour la fonction supprimer() en: 0.0 millisecondes.

Process finished with exit code 0
  
```

Illustration 1: Le test pour les listes chaînée

## Comparaison entre une liste chaînée et un arbre binaire de recherche :

D'après les résultats obtenus en testant sur les arbres binaires de recherche et les listes chaînées, on remarque que les temps d'exécution pour les opérations effectuées sur ABR sont pratiquement grands par rapport aux temps d'exécution de celles de la liste chaîne, en plus, il y a une cohérence entre le temps d'exécution d'une part, et la hauteur pour les arbres et la longueur pour les listes d'autre part. Ainsi plus l'arbre aura une hauteur élevée ou la liste aura une grande longueur, plus l'opération mettra de temps à s'exécuter.