FRAPPER Colin
N’GUESSAN Jean-Philippe

Conception et analyse d'algorithmes

Implémentation et Manipulation d’AABRR

12/01/2018

Sommaire :

I - Introduction

La structure de données en arbre est une structure particulière, elle permet de représenter des données identiques liées entre elle comme nous avons pu le voir dans le module de conception algorithmique.

L'objectif lié à ce projet était de nous permettre de manipuler les arbres binaires de recherches, les structures de données et les arbres binaires de recherche à rebours.

Pour la réalisation du projet nous avons fait le choix du langage java. Ce choix a été influencé par le faite que nous effectuons notre stage de fin d'année en java et aussi par notre aisance technique sur ce langage.
Par ailleurs les structures d'arbres demandé ici nécessitent une grande utilisation des pointeurs c'est aussi dans cette optique que nous avons estimé judicieux de faire du java au détriment d'un langage de "bas niveau" comme le C.

Notre document présente en premier lieu l'explication détaillé de nos fonctions implémentées ce qui nous permettra de déterminer les complexités de ces dernières.
Un jeu de test est proposé ainsi qu'un lien explicatif du fonctionnement de notre application.

II - Classes implémentées

1. La classe Main

Classe principalement pour tester mais possède également quelques fonctions que je détaillerais plus bas.

2. La classe AABRR

Cette classe représente arbre binaire de recherche avec différentes caractéristiques :

- deux entiers m et M tel que m <= M
- un ABRR (Arbre Binaire de Recherche à Rebours) A',

Attributs :

```
private int min, max ;
private ABRR noeud ;
private AABRR fils_gauche ;
private AABRR fils_droit ;
private AABRR racine ;
```

```
private AABRR parent;
```

3. La classe ABRR

Classe représentant un arbre binaire de recherche à rebours, c'est à dire :

- chaque noeud stocke un entier k compris entre m et M (m k M). Un ABRR est un AB tel que pour tout noeud u :
  - si v est dans dans le sous-arbre gauche de u, alors val(v) > val(u)
  - si v est dans dans le sous-arbre droit de u, alors val(v) <= val(u)

Attributs :

```
private int cle ;
private ABRR fils_gauche ;
private ABRR fils_droit ;
private ABRR racine ;
private ABRR parent;
```

Utilisation d'une variable parent de type ABRR utile lors de l'affichage.
Les variables suivantes concernent l'affichage graphique (en console) ;

```
static int niveauCourant   = -1;
static int espaceRestant = -1;
static final int H_SPREAD = 3;
private int profondeur=0;
private int niveau=0;
private int Position=0;
```

4. La classe ABR

Classe représentant un arbre binaire de recherche

```
private int cle ;
private ABR fils_gauche ;
private ABR fils_droit ;
private ABR parent ;
private ABR racine;
```

Utilisation d'une variable parent de type ABRR utile lors de l'affichage.
Les variables suivantes concernent l'affichage graphique (en console) ;

```
static int niveauCourant  = -1;
static int espaceRestant = -1;
static final int H_SPREAD = 3;

private int profondeur=0;
private int niveau=0;
private int Position=0;
```

III - Fonctionnalités implémentées

Dans cette partie, nous allons parler des différentes fonctionnalitées que nous avions du implémentées et une explication des différentes fonctions utilisées pour répondre à ces fonctionnalitées ainsi que leur complexité. Vous trouverez en conclusion de cette partie un tableau résumant les complexités.

1. Fichier vers AABRR

```
private static AABRR file_to_AABRR(File f, AABRR a)
```

Cette fonction se trouve dans la classe Main, et renvoie un AABRR. L'idée est de parcourir le fichier passé en paramètre ligne par ligne, à chaque parcours on va split la ligne pour avoir dans la première partie deux variables m et M, et dans l'autre les noeuds à insérer.
A chaque fin de parcours on va donc insérer dans l'AABRR donnée en paramètre, on va donc s'intéresser maintenant à la fonction insert situé dans la classe AABRR.

```
public void insert(int min, int max, String[] noeuds)
{
      racine = insert_in_AABRR(racine,min,max,noeuds);
}
```

On va donc s'intéresser ici à cette fonction :

```
public AABRR insert_in_AABRR(AABRR a, int min, int max, String[] noeuds )
```

Cette fonction prend en paramètre un AABRR, deux entiers min et max, et un tableau de noeuds à insérer.

L'insertion "classique" d'un arbre binaire de recherche se fait en **O(n)** où n est la taille de l'arbre à arbres binaire de recherche à rebours. Dans le pire cas, on aura à traverser de la racine au feuille. De plus dans notre cas, une fois qu'on à trouver où insérer l'AABRR il nous faut insérer les différents éléments de l'ABRR correspondant. On va donc parcourir le nombre d'éléments qu'on notera p et insérer ces éléments. L'insertion dans un ABRR se fait également en O(p) où p est la taille de l'arbre binaire de recherche à rebours.

En terme de complexité, la fonction `insert_in_AABRR` sera en **O(n*p²)**.

Néanmoins il ne faut pas oublier que l'on va insérer un AABRR à chaque ligne du fichier passé en paramètre. Notons nb_lignes le nombre de ligne que contient le fichier.

En somme la fonction `file_to_AABRR` sera en **O(nb_lignes*n*p²)**.

2. AABRR vers fichier

```
private static void AABRR_to_file(AABRR racine)
```

Cette fonction prend un AABRR en paramètre, ce sera celui qu'on écrira dans un nouveau fichier "written.txt". La technique est la suivante, effectuer un parcours préfixe de l'arbre et stocker les éléments dont nous avons besoin dans une chaîne de caractères.
On aura donc dans notre classe AABRR la fonction :

```
public String parcoursWriteRecur(String s, AABRR a)
```

Cette fonction s'éffectue en **O(n)**, à chaque parcours on effectue également un parcours prefixe sur l'ABRR, qui prend également un temps en **O(p)**.

En somme la fonction `AABRR_to_file` sera en **O(n*p)**.

3. Affichage à l'écran

Pour cette partie, nous n'avons pas réussi à créer une interface graphique représentant l'arbre, néanmoins nous avons réussi à afficher l'arbre de manière plutôt "sympa" en console.
Nous avons pris cette fonction sur internet (source : https://prismoskills.appspot.com/lessons/Binary_Trees/Tree_printing.jsp).
Nous avons du adapter cette fonction à notre propre arbre.

```
static void drawTree(ABR root)
```

Voici comment cette fonction marche :
Elle utilise une fonction classique `public static int profondeur (ABR n)` pour calculer la profondeur de l'arbre.

La fonction drawTree s'effectue en **O(p)**, c'est un parcours de l'arbre en soit on va parcourir chaque élément.

```
static void drawElement(ABR ele, int profondeurChildCount[], int profondeur, LinkedList<ABR> q)
```

Cette fonction est appelé à chaque noeud de l'ABR, elle permet de "dessiner" l'arbre. On effectue une boucle afin de parcourir l'arbre. C'est dans ce cas que l'on utilise la variable *parent* qui permet de savoir lorsqu'on visite un noeud si il a un "frère", c'est à dire s'il faut écrire les backslash ou non.

Cette fonction drawElement s'effectue **O(profondeur*2)** car le pire cas est celui où l'on commence dès la racine et qu'il possède un fils droit et gauche.

En somme pour afficher un AABRR, nous effectuons un parcours préfixe de l'arbre AABRR et nous appelons cette fonction drawTree, on aura donc en terme de complexité : **O(n*p*2*profondeur)**.

Pour un affichage plus trivial, nous utilisons également le parcours préfixe en **O(n*p)**.

4. AABRR aléatoire

```
private static AABRR aleatoire_AABRR(int p, int q)
```

Cette fonction prend en paramètre un entier p représentant le nombre de noeuds souhaités, et un entier q représentant le maximum de M. Elle retourne l'AABRR aléatoire.

Tout d'abord cette fonction teste si il est possible de construire cet arbre ou non. Du fait que l'on ne veut pas d'arbre vide et uniquement des intervalles disjoints il nous faut tester que 2*p soit bien strictement supérieur à q ainsi que p == q, en effet m doit être différent de M.
Une fois ces conditions vérifiés, voici comment nous procédons :
Nous créons un tableau d'entier de taille 2*p, du fait qu'il y est p noeuds ils nous faut 2 valeures distinctes qui représentent l'intervalle de chaque noeud. On initialise dans un premier temps la première ainsi que la deuxième valeur du tableau à 1 et q respectivement. On remplit ensuite le reste du tableau avec des valeurs aléatoires entre 1 et q et différentes des valeurs déjà insérées dans le tableau, le but est de ne pas avoir de doublons. Une fois le tableau créé, on va trier ce tableau et appeler une nouvelle fonction qui va nous créer notre AABRR.

```
private static AABRR create_AABRR_alea(int[] tab, int p)
```

Cette fonction prend en paramètre un tableau d'entier (notre tableau d'intervalle, trié), et le nombre de noeuds à créer.

Pour construire notre AABRR, on va donc créer un arbre vide, on utilisera un nombre aléatoire entre 0 et et p-1 pour savoir quelle sera la racine de notre arbre.
Une fois notre intervalle choisi, il nous faut déterminer le nombre de noeuds qu'on pourra insérer dans cet intervalle, on va donc encore une fois faire un random, pour savoir combien de noeuds vont être insérés.
On créé ensuite notre tableau de noeuds et on le rempli avec des valeurs aléatoires comprises entre l'intervalle choisi.
Une fois inséré, on initialise les valeurs du tableau à 0 de façon à ne pas réécrire cet intervalle et donc le même AABRR.
Il nous faut maintenant remplir le reste de l'AABRR, tant que tous les noeuds ne sont pas rempli, on fait un random entre 0 et p-1, on teste si cet intervalle n'a pas déjà été inséré et de la même façon que pour la racine, on va insérer chaque AABRR.

Pour être honnête, c'est une des fonctions qui nous a demandé le plus d'effort, si l'on veut faire quelques choses de parfaitement aléatoires.

En terme de complexité maintenant :

C'est un peu spécial pour cette fonction car elle comporte beaucoup d'aléatoire et je n'ai pas trouvé la complexité au pire de Math.random(), je vais supposer ici qu'elle trouve le bon élément du premier coup.

La fonction `aleatoire_AABRR` on va parcourir le tableau qui est de taille 2*p, on va ensuite trié le tableau, la fonction java Arrays.sort est en **O(nlogn)** d'après mes sources (https://stackoverflow.com/questions/22571586/will-arrays-sort-increase-time-complexity-and-space-time-complexity)

pour cette fonction on a donc du **O(2*p) + O(nlogn)**, avec p le nombre de noeuds et n le nombre d'élément du tableau soit 2*p.

Passons maintenant à la fonction `create_AABRR_alea` : C'est ici que ça devient un peu compliqué dû à l'aléatoire. Néanmoins puisque on parle ici de complexité au pire, on va parcourir p fois afin d'insérer à chaque fois dans l'AABRR, nous l'avons vu plus haut, la fonction insérer **O(n*p²)** avec p la taille de l'arbre binaire de recherche à rebours, appelons cette taille s pour la différencier avec le nombre de noeud p.
Pour en revenir à notre fonction on aura donc une fonction `create_AABRR_alea` en **O(p*n*s²)** avec p le nombre de noeud à insérer dans l'AABRR, n la taille de l'AABRR et s la taille de l'ABRR.

5. Vérification

```
private static void verif_AABRR(File p)
```

Cette fonction prend en paramètre un fichier et teste si un AABRR est valide, c'est à dire si :

- A est un ABR (sur les valeurs de m)
- tous les intervalles [m;M] des noeuds de A sont disjoints
- tous les arbres A' des noeuds de A sont des ABRR contenant des éléments compris entre m et M

Dans un premier temps on crée l'AABRR, on a donc une complexité de **O(nb_lignes*n*p²)** comme vu plus haut.
ensuite on appelle la fonction `public boolean verif_AABRR()` qui vérifie si un AABRR et valide, pour ce faire on va donc créer une arraylist d'entiers et appeler la fonction `public boolean create_tab(ArrayList<Integer> tab_int, AABRR a)`, cette fonction a comme paramètre une arraylist et un AABRR, on effectue un parcours infixe qui ajoute dans l'arraylist passé en paramètre les valeurs de min et max de l'AABRR, à la fin de ce parcours on test si le tableau est en ordre croissant et si toutes les valeurs (et donc intervalle) sont bien disjoints les unes par rapport aux autres ce qui permet de vérifier que : A est un ABR (sur les valeurs de m) et tous les intervalles [m;M] des noeuds de A sont disjoints.
Afin de tester la dernière propriétés, on créé également une arraylist et on va pour chaque noeuds appeler
`public boolean parcoursInfixeInverse(ArrayList<Integer> arl, int m, int M)` dans la classe ABRR qui effectue un parcours infixe inverse et de la même façon on teste si c'est bien un ABR (inversé dans ce cas) ainsi que toutes les valeurs sont comprises entre m et M. De cette façon nous avons bien vérifiés toutes les propriétés d'un AABRR valide.

Passons maintenant à la complexité :
la fonction `verif_AABRR()` va donc uniquement créé l'arbre à partir du fichier donc : **O(nb_lignes*n*nb_elem*p)**, la fonction `create_tab` effectue un parcours infixe, en **O(n) * un appel de la fonction parcoursInfixeInverse sur chacun de ces noeuds, + un parcours de l'arrayList soit en O(n)**.
Regardons la fonction `parcoursInfixeInverse` , de la même façon on effectue un parcours infixe, où l'on stocke dans une arrayList les éléments, soit en **O(p)**, on parcourt également ce tableau à la fin pour tester les différents cas, soit la fonction dans sa totalité est en **O(p)** pour le parcours infixe inverse + O(p) pour le premier parcours d'arrayList + O(p) pour l'autre parcours ⇒ **O(p) + O(p) +O(p) ⇒ O(p)**.

En somme on aura donc du **O(n*p)** pour cette fonction de vérification (si l'on ne tient pas compte de la création de l'AABRR).

6. Recherche d'un entier

```
public void rechercheRecur(int value, AABRR a)
```

Cette fonction prend en paramètre l'entier à rechercher ainsi qu'un AABRR.
Dans un premier temps on va chercher dans quel AABRR la valeur se trouve, une fois l'intervalle de la valeur trouvé on va appeler la fonction recherche de l'ABRR en question, si cette fonction retourne faux on affiche l'intervalle trouvé.

```
public boolean rechercheRecur(int value, ABRR a)
```

On va se déplacer dans l'arbre de façon à trouver la valeur (ou non).
En terme de complexité cette fonction est en **O(p)**.

En somme `rechercheRecur` dans la classe AABRR s'éffectue en **O(n)** pour trouver l'intervalle, on teste ensuite sur cet intervalle si la valeur est contenue dans l'ABRR, cette fonction est en **O(p)**.
En somme de recherche on aura donc **O(n)** pour chercher l'intervalle + **O(p)** pour chercher la valeur ⇒ **O(n) + O(p)** ⇒ **O(n)** si la taille de l'AABRR est plus grande que celle de l'ABRR recherché, dans l'autre cas **O(p)**.

7. Suppression d'un entier

```
public void suppression(int value)
```

Cette fonction prend en paramètre la valeur à supprimer.
Dans un premier temps on va chercher l'AABRR correspondant à la valeur. Pour cela on utilise
`public AABRR trouverNoeud(int value, AABRR a)`

Cette fonction retourne l'AABRR correspondant et prend en paramètre la valeur à supprimer et un AABRR.
Cette fonction marche de la même façon que la fonction rechercheRecur excepté qu'elle retourne l'AABRR correspondant. Une fois l'AABRR trouvé. On récupère l'ABRR de cette AABRR, et l'on appelle la fonction supprimer de la classe ABRR

```
private ABRR supprimerRecur(ABRR a, int value)
```

Cette fonction était également une des plus dur à réaliser, du fait qu'il fasse "remonter" la valeur.

On parcourt l'arbre, une fois la valeur trouvé (on sait qu'elle existe) car on a utilisé la fonction recherche dans trouverNoeud. On a donc trois cas :

1) *Le noeud à supprimer est une feuille*
   On supprime le noeud
2) *Le noeud à supprimer à un fils*
   On fait "remonter" le fils
3) *Le noeud à supprimer à ces deux fils*
   On utilise la fonction maxValue, on va chercher la valeur la plus grande de son fils droit, et on fait remonter

Cette fonction est en **O(p)**

La fonction `trouverNoeud` comme la fonction de recherche s'éffectue en **O(n) + O(p)**.

On a donc du **O(n) + O(p) + O(p) ⇒ O(n)** si la taille de l'AABRR est plus grande que celle de l'ABRR recherché, dans l'autre cas **O(p)**.

Il reste une dernière "subtilité" à gérer : le cas où l'ABRR contient une seule valeur, on ne peut pas avoir d'arbre vide, on va donc "relier" le parent au fils de l'ABRR qui a été supprimé.

8. Insertion d'un entier

```
public void insertion(int value)
```

Cette fonction permet l'insertion d'une valeur passée en paramètre.
Dans un premier temps on appelle la fonction

```
public AABRR trouverNoeudExistant(int value, AABRR a)
```

Cette fonction permet de trouver un noeud existant, il retourne null si l'intervalle n'existe pas.
Cette fonction marche exactement de la même manière que trouverNoeud, donc complexité ⇒ **O(n)**.
Une fois cette fonction terminée, deux cas ;
La valeur à insérer n'appartient à aucun des intervalles, on affiche que l'on ne peut insérer, ou alors on a l'AABRR où il faut insérer, on va donc appeler la fonction insert sur l'ABRR correspondant. La complexité de cette insertion est de **O(p)**.

En somme, on aura du **O(n) + O(p) ⇒ O(n)** si la taille de l'AABRR est plus grande que celle de l'ABRR recherché, dans l'autre cas **O(p)**.

9. ABR vers AABRR

```
private static AABRR ABR_to_AABRR(ABR a, int k)
```

Cette fonction prend en paramètre un ABR ainsi qu'un entier k qui représente le nombre d'intervalle de l'AABRR.

Tout d'abord il nous faut tester un cas afin d'éviter d'avoir des ABRR vide, le cas où l'on veut créer plus d'intervalles qu'il n'y a d'élément dans l'ABR. Pour cela nous avons une fonction nombre_value qui permet d'avoir le nombre d'élément d'un ABR, cette fonction est en **O(k)** et même **theta(k)** où k est la taille de l'ABR. Une fois ce cas traité, on initialise deux valeurs m et M comme le minimum et le maximum respectivement de l'ABR. On crée également une variable qui représentera la taille de chaque intervalle. On créé un tableau qui sera notre tableau d'intervalle.
Jusque là, la complexité est de **theta(k)**.
On va donc ensuite appeler la fonction

```
private void insert_intervallesRecur(AABRR result, int[] tab_int,
ABR a)
```

Cette fonction fait partie de la classe ABR, elle prend en paramètre un AABRR qui sera l'AABRR créé, un tableau d'entier représentant le tableau d'intervalle, et un ABR.

On créé une variable indice qui représente l'indice de la racine de l'ABR afin de créer la racine de l'AABRR, cette fonction a pour but de créer tous les AABRR sans aucuns ABRR.

Une fois l'AABRR créé, il nous faut maintenant insérer chacune des valeurs de l'ABR. On appelle pour cela la fonction

```
private void insert_for_AABRR_Recur(AABRR result, ABR a)
```

Cette fonction fait également partie de la classe ABR, elle utilise un parcours préfixe sur l'ABR a et insert dans l'AABRR result chaque valeur.

Cette fonction a été une des plus difficiles pour nous à réaliser.

En terme de complexité pour la fonction `insert_intervallesRecur` : La fonction getIndice se fait en **theta(k)**. Le parcours du tableau est en **theta(k)** auquel on multiple la fonction insert qui est en **O(n)**, car contrairement à la première fonction insert vu dans le (1.) on n'insère pas les ABRR. On a donc pour cette fonction ⇒ **O(k*n)**.

Pour la fonction `insert_for_AABRR_Recur` on va donc faire un parcours prefixe en **O(n)** auquel on va appeler la fonction insertion en **O(n)** si la taille de l'AABRR est plus grande que celle de l'ABRR recherché, dans l'autre cas **O(p)**. On aura donc pour cette fonction **O(n*n)** ou **O(n*p)**.

En somme pour la fonction `ABR_to_AABRR` on aura donc ⇒ **theta(k) + O(k*n) + O(n*p)**. Dans le pire cas on a k = n, ⇒ **O(n²)**.

10. AABRR vers ABR

```
private static ABR AABRR_to_ABR(AABRR a)
```

Cette fonction prend en paramètre un AABRR et retourne un ABR correspondant.
On appelle `public void createABRRecur(AABRR a, ABR b)` , fonction de la classe AABRR, qui va réaliser un parcours préfixe et appeler une fonction createABR sur chaque noeud (ABRR).

```
private void createABRRecur(ABRR a,ABR b)
```

Cette fonction également effectuer un parcours préfixe et insérer chaque élément de l'ABRR dans l'ABR.
En terme de compléxité cette fonction nécéssite un parcours préfixe en **O(p)** ainsi que l'insertion dans un ABR en **O(p)** également. On a donc ⇒ **O(p²)**.

De plus, on réalise un premier parcours préfixe sur l'AABRR en **O(n)**, ensuite un parcours préfixe sur l'ABRR en **O(p)**, et ensuite l'insertion dans l'ABR en **O(p)**.

En somme on aura donc du **O(n*p²)**.

Voici un tableau qui résume les complexités en fonction des fonctionnalités :

Fichier vers AABRR	O(nb_lignes*n*p²)
AABRR vers fichier	O(n*p)
Affichage à l'écran	O(n*p*2*profondeur) ou O(n*p)
AABRR aléatoire	O(p*n*s²) ****
Vérification	O(n*p)

Recherche d’un entier	O(n)
Suppression d’un entier	O(n)
Insertion d’un entier	O(n)
ABR vers AABRR	$O(n^2)$
AABRR vers ABR	$O(n*p^2)$

nb_lignes = nombre de ligne du fichier texte.
n = taille de l’AABRR
p = taille de l’ABRR
profondeur = profondeur de l’ABRR

**** Dans ce cas la, p = nombre de noeud à créer dans l’AABRR, s = taille de l’ABRR.

IV - Jeu d’essai

Action (Fonction)	résultats attendu	résulats action
Rechercher(80)	Le résultat attendu: L’élément est trouvé car il est présent dans l’arbre et dans l’interval 78-80	Passe avec succès
Rechercher(95)	Le résultat attendu: L’élément n’est pas présent dans l’arbre mais appartient à l’interval 92-110	Trouve l’interval correspondant de l’élément recherché qui n’est pas présent dans l’arbre. Passe avec succès
Rechercher(500)	Le résultat attendu: L'élément n’est pas présent dans l’arbre ainsi que l’interval	Passe avec succès
aleatoire_AABRR(3,10) 3 représente le nombre de noeud et 10 le valeur Maximal	le résultat attendu: Parmis les trois noeuds retourner il doit avoir un noeud qui contient la valeur minimal qui sera m=1 et la valeur maximal M=10 (marque plutot min(m) = 1 et max(M)	Passe avec succès

	= 10	
verif_AABRR	Vérifie l'arbre rentrer dans le fichier .txt et retourne un résultat positif si ce dernier est correct.	Passe avec succès
Suppression(80)	Vérifie si l'élément est présent dans l'arbre avant de le supprimé et retourne son interval	Passe avec succès
suppression(1)	Supprime l'élément et reconstitue l'arbre A'	Passe avec succès
Insertion (95)	Vérifie l'intervalle de l'élément à insérer et l'insert.	Passe avec succès
Fichier vers AABRR	Nous choisissons le fichier .txt	Passe avec succès

V - Manuel d'utilisation

Nous avons réalisé une vidéo où l'on réalise une démonstration de la compilation du projet, si ça vous intéresse, ça sera peut être plus facile à comprendre qu'un fichier texte. Excusez nous pour la qualité du son.

https://youtu.be/saNOFRqHUSY

Vous trouverez également le fichier texte pour compiler le projet.

VI - Conclusion

En conclusion, on a trouvé très intéressant le projet et les différentes problématiques que l'on devait résoudre, nous avons bien travaillé ensemble pour arriver à ce que l'on voulait faire. Pour avoir un esprit plus critique et un peu de recul sur notre travail, nous sommes un peu déçu du fait que l'on ai pas réussi à réaliser un affichage graphique (en utilisant javafx ou swing par exemple), un autre point que l'on aurait pu améliorer est le menu, en effet on trouve qu'il n'est pas optimal et aurait pu être un peu mieux réalisé.