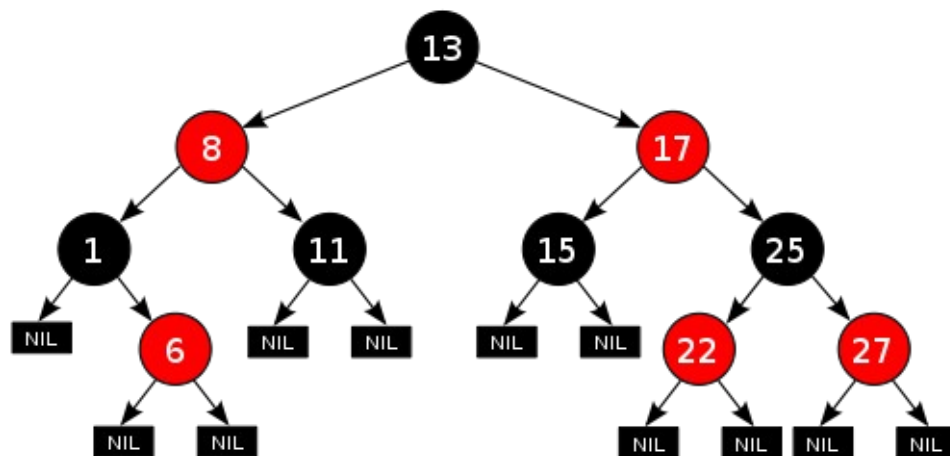


Algorithmique 2 – Projet d'année 2011



Lib RougeNoir & Gestion d'étudiants

Auteur: Cédric Lood

Date: 09 Mai 2011

Table des matières

1 Introduction.....	3
1.1 But du document.....	3
1.1 Description du programme utilisateur.....	3
2 Etude de la complexité de l'algorithme.....	3
2.1 Etude expérimentale de la hauteur de l'arbre.....	3
2.1 Complexité des fonctions d'interface.....	5
3 Découpe du programme.....	6
3.1 Diagramme de classe.....	6

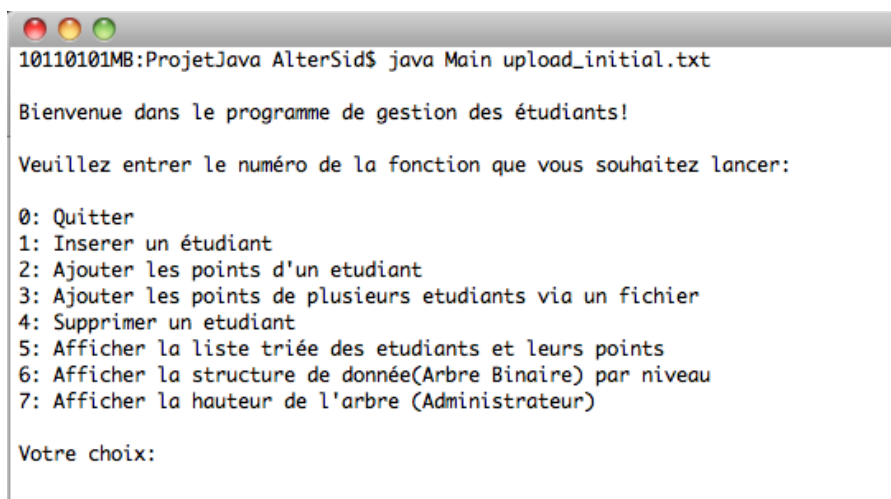
1 Introduction

1.1 But du document

Dans le cadre du cours d'algorithmique 2, il nous est demandé de fournir un programme utilisant une structure de données de type arbre binaire de recherche balancé (Rouge Noir). Le code étant fourni par ailleurs (voir archive digitale), ce document se contentera de présenter certaines analyses relatives à des mesures de complexité de l'algorithme. Une étude expérimentale détaillée des hauteurs d'arbres obtenus avec différents jeux de données sera aussi présentée.

1.1 Description du programme utilisateur

Le programme utilisant l'arbre balancé comme structure de données a pour but de gérer les grades d'un ensemble relativement stable d'étudiants. Après un upload initial, l'utilisateur se verra présenter le menu de gestion suivant :



```
10110101MB:ProjetJava AlterSid$ java Main upload_initial.txt

Bienvenue dans le programme de gestion des étudiants!

Veuillez entrer le numéro de la fonction que vous souhaitez lancer:

0: Quitter
1: Insérer un étudiant
2: Ajouter les points d'un étudiant
3: Ajouter les points de plusieurs étudiants via un fichier
4: Supprimer un étudiant
5: Afficher la liste triée des étudiants et leurs points
6: Afficher la structure de donnée(Arbre Binaire) par niveau
7: Afficher la hauteur de l'arbre (Administrateur)

Votre choix:
```

L'utilisateur pourra donc faire les changements nécessaires à l'ensemble des étudiants et afficher par exemple le résultat des modifications avant de quitter le programme. Il est à noter que la structure de données n'est pas sauvegardée lors de la sortie du programme et les changements effectués seront dès lors perdus (le but n'étant pas ici le réalisme du programme mais bien l'application des principes théoriques vus aux cours).

2 Etude de la complexité de l'algorithme

2.1 Etude expérimentale de la hauteur de l'arbre

Propriété: Un arbre rouge-noir avec n noeuds internes a une hauteur d'au plus $2 \cdot \log(n+1)$.

Afin de vérifier que cette propriété est bien conservée pour l'implantation de la structure de données, plusieurs fichiers d'upload ont été créés avec un nombre croissant de données (ceux-ci sont disponibles dans l'archive digitale fournie). Les données des fichiers ont été triées par ordre croissant de clé afin de maximiser le nombre de réorganisations interne de l'arbre (tests de stress sur la structure de données).

Afin de mesurer la hauteur de l'arbre, deux fonctions supplémentaires ont été implantées:

1. Une fonction récursive permettant de calculer la hauteur d'un noeud donné:

```
private int hauteur(NoeudRN n){
    if(n==sentinelle)
        return 0;
    else {
        int hGauche = hauteur(n.getGauche());
        int hDroite = hauteur(n.getDroit());
        return Math.max(hGauche,hDroite)+1;
    }
}
```

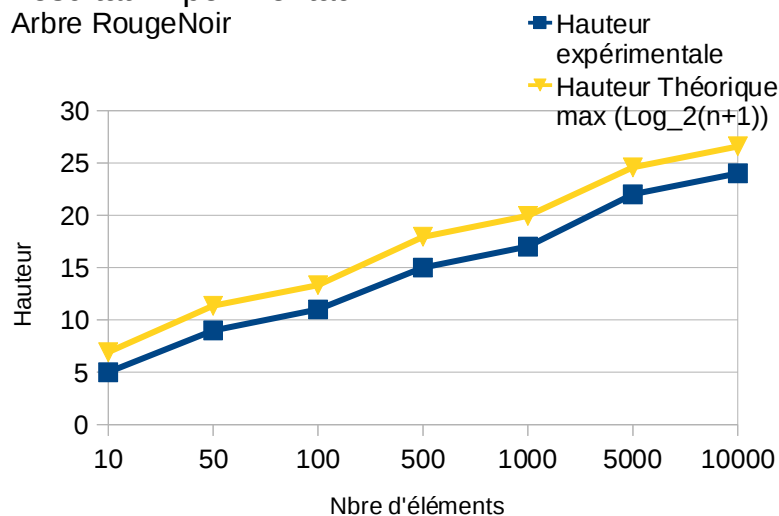
2. Une fonction appelant cette première en passant la racine de l'arbre et retournant donc la hauteur totale de l'arbre.

```
public int hauteurArbre(){
    return hauteur(racine);
}
```

Voici un tableau présentant les résultat expérimentaux ainsi que le graphe associé:

Nombre d'éléments (n)	Hauteur expérimentale	Hauteur Théorique max ($2 * \text{Log}_2(n+1)$)
10	5	6.92
50	9	11.34
100	11	13.32
500	15	17.94
1000	17	19.93
5000	22	24.58
10000	24	26.58

Résultat Expérimentaux
Arbre RougeNoir



On peut observer grâce au graphe la stricte observance de la propriété énoncée ci-dessus. La démonstration de cette propriété étant fournie dans le syllabus (page 16 – édition 2), je ne la recopierai pas ici.

2.1 Complexité des fonctions d'interface

Une fois la hauteur de l'arbre garantie comme étant au maximum de $2 \cdot \log_2(n+1)$ la plupart des fonctions se retrouve avec une complexité en $O(\log n)$ ou $O(1)$. Les fonctions privées sont toutes de l'une de ces 2 catégories (démonstrations dans le syllabus – chapitre 2), nous nous limiterons donc ici à étudier de manière sommaire la complexité des fonctions de l'interface publique :

- **public void** insert(**int** matriculeEtudiant, **int** pointInitial)

Appels aux constructeurs Etudiant, NoeudRN : $O(1)$

Appel à la fonction privée insert : $O(\log n)$

=> Complexité totale : $O(\log n)$

- **public void** remove(**int** matricule)

Appel à la fonction de recherche de noeud : $O(\log n)$

Appel à la fonction delete : $O(\log n)$

=> Complexité totale : $O(\log n)$

- **public void** addPoints(**int** matriculeEtudiant, **int** points)

Appel à la fonction de recherche de noeud : $O(\log n)$

Appel à la fonction ajoutPoints : $O(1)$

=> Complexité totale : $O(\log n)$

- **public int** getSize()

Cette fonction retourne la variable de classe *nbreEntree* : $O(1)$

- **public void** printBreadthFirst()

Etant donné que cette fonction devra parcourir les n noeuds de l'arbre afin d'afficher leur contenu (affichage en $O(1)$), la complexité du parcours est en $O(n)$

- **public void** printInOrder()

idem printBreadthFirst

- **public** RougeNoir(String fichier)

La boucle externe est utilisée afin de parcourir les n lignes du fichier : $O(n)$

La boucle interne quand à elle fait appel à la fonction insert : $O(\log n)$

=> Complexité totale : $O(n \log n)$

- **public void** addFromFile(String fichier)

Même principe que pour le constructeur RougeNoir, à la différence que le nombre d'entrée du fichier vaut k (k pouvant être différent de n). la boucle interne fait par ailleurs appel à la fonction rechercheNoeud pour vérifier si le noeud est déjà présent (et dans ce cas on ajoute seulement les points) ou bien s'il faut l'insérer.

=> Complexité totale : $O(k \log n)$

3 Découpe du programme

3.1 Diagramme de classe

