LIÈGE université
Sciences Appliquées

INFO2050: Advanced Programming
Assignment 2 - Report
November 24^{th}, 2020

Alain Eros Prestige	Musoni	s202208
Ayawo Désiré	Dandji	s197206

Introduction

Dans ce projet nous sommes appelés à implementer les structures de données Hash table et le Binary Search Tree(BST) et ensuite on fera une analyse pour explique leur fonctionement et l'efficacite en utilisant la notation Big-O.

1 Ensemble

1.1 Implémentation

1.2 Analyse

1.2.1

(a) Explication du choix d'implémentation:

- **Hash Table:** Pour implémenter la table hachage, un enregistrement nommé TableDeHachage a été utilisé possédant les champs comme; valeur qui est un tableau de chaine de caractères contenant toutes les valeurs de la table de hachage, clé qui est un tableau d'entiers contenant les clés de chaque valeur permettant de retrouver avec une fonction de hachage très rapidement chaque élément, capacité qui lui est un entier déterminant le nombre d'éléments maximum que la table de hachage peut contenir.

  Finalement, on implémente l'ensemble grâce à un enregistrement possédant les champs tableDeHachage qui est une variable de type TableDeHachage, taille qui est un entier qui permet de connaitre le nombre d'élément de l'ensemble.

  En gros, pour stocker une chaine de caractère provenant d'un fichier, on se sert d'une fonction de hachage assez puissante pour obtenir une clé a priori unique. On stocke cette

clé séquentiellement dans le champs clé, puis on stocke la chaine en question à la position clé obtenue par hachage dans le tableau valeurs. En cas de collision, on se déplace séquentiellement à partir de la position clé jusqu' à trouver une case vide dans le tableau valeur pour stocker la chaine de caractères.

- **Binary Search Tree:** Pour implémenter l'arbre binaire de recherche équilibré, une structure nommée Tree est utilisé. Ses champs sont; valeur qui est une chaine de caractères pour stocker un mot dans un noeud de l'arbre, un mot d'un fichier texte, gauche qui un pointeur vers le type Tree pour matérialiser le fils gauche d'un noeud de l'arbre, droite qui est un pointeur vers le type Tree pour matérialiser le fils droit d'un noeud de l'arbre et hauteur qui est un entier pour connaitre la hauteur de l'arbre.
  Pour l'ensemble, on utilise un enregistrement possédant les champs racine qui est un pointeur vers le type Tree pour désigner le noeud racine de l'arbre binaire et taille qui est un entier pour désigner le nombre d'éléments de l'ensemble. D'une manière simple, on stocke chaque valeur dans l'arbre de manière a toujours maintenir l'arbre équilibré en procédant par différentes rotations des sous arbres chaque fois que nécessaire.

(b) Le principe de fonctionnement du setIntersection, d'abord on recoit deux ensembles S1 et S2 et puis on crée un nouveau tableau vide pour en fin le passer à la fonction parcoursInfixe, lui qui va nous servir à creer un nouveau tableau contentant $S1 \cap S2$.
Pour la fonction parcourInfixe si l'arbre est vide, on ne retourne rien sinon on parcourt l'arbre recursivement de gauche à droite en regardant si la valeur du noeud courant existe aussi dans l'ensemble S2 pour qu'on l'insère dans le nouveau tableau crée récemment.

Pseudo code de la fonction d'instertion:

```
setIntersection(S1,S2)
    new_array=createEmptyArray()
    parcoursInfixe(S1.racine, S2, &new_array)
    return new_array


parcoursInfixe(tree, S2, array)
    if(tree == NULL)
        return
    parcoursInfixe(tree.gauche, S2, array)
    val=tree.valeur
    if(contains(S2, val))
        if(insertInArray(array, tree.valeur)==NULL)
            freeArray(array, false)
            return
    parcoursInfixe(tree.droite, S2, array)
```

(c) Analyse de complexite au pire des cas

- Pour la table de Hachage la complexité est O(N_A N_B) puisqu'on va parcourir tous les N_A éléments de l' ensemble A et ensuite vérifier s'il se retrouve dans B. Le cout de la recherche dans B est O(N_B) au pire des cas en raison des collisions puisqu'il s'agit d'une table de hachage.
- Pour l'arbre binaire de recherche La complexité est O(N_A N_B) puisqu'on va parcourir tous les N_A éléments de l' ensemble A et on recherche chaque élément dans B. Le cout de la recherche dans B est O(N_B) au pire des cas puisque l'arbre s'il est très déséquilibré ,peut être réduit à une liste chainée.

(d) Analyse de complexite au moyen des cas

- Pour la table de Hachage la complexité est de $O(N_A)$ puisque pour chaque éléments de l' ensemble A jusqu'à $N_A - 1$, on recherche cet élément dans B. Le cout de la recherche dans B est $O(1)$ au cas moyens sans collisions.
- Pour l'arbre binaire de recherche la complexité est de $O(N_A log_2(N_B))$ puisqu'on va parcourir tout élément de l' ensemble A jusqu'à $N_A - 1$ en recherchant cet élément dans B. Le cout de la recherche dans B est $O(log_2(N_B))$ dans le cas ou l'arbre est équilibré.

1.2.2

- Calcule de temps moyens nécessaires à l'insertion dans un ensemble contenant N eléments pour des valeurs de N croissantes:

  - Pour table de Hachage:

N éléments	10^1	10^2	10^3	10^4	10^5
Temps(ms)	0,034	0,193	1,388	4,143	21,480

  - Pour l'arbre binaire:

N éléments	10^1	10^2	10^3	10^4	10^5
Temps(ms)	0,014	0,086	0,651	14,212	138,712

  - Pour la liste chainée:

N éléments	10^1	10^2	10^3	10^4	10^5
Temps(ms)	0,015	0,048	2,710	283,595	34417,644

- Calcule de temps (moyens) nécessaires à une recherche dans un ensemble contenant N eléments pour des valeurs de N croissantes. On va rechercher les N_A éléments du fichier French.txt dans N_B elements de English.txt

  - Pour table de Hachage:

N éléments	10^1	10^2	10^3	10^4	10^5
Temps(ms)	6,916	8,650	6,369	5,582	6,464

  - Pour l'arbre binaire:

N éléments	10^1	10^2	10^3	10^4	10^5
Temps(ms)	3,090	4,665	4,203	8,075	10,474

  - Pour la liste chainée:

N éléments	10^1	10^2	10^3	10^4	10^5
Temps(ms)	1,727	13,504	144,602	1338,528	12249,852

1.2.3 Résultats sur deux graphe

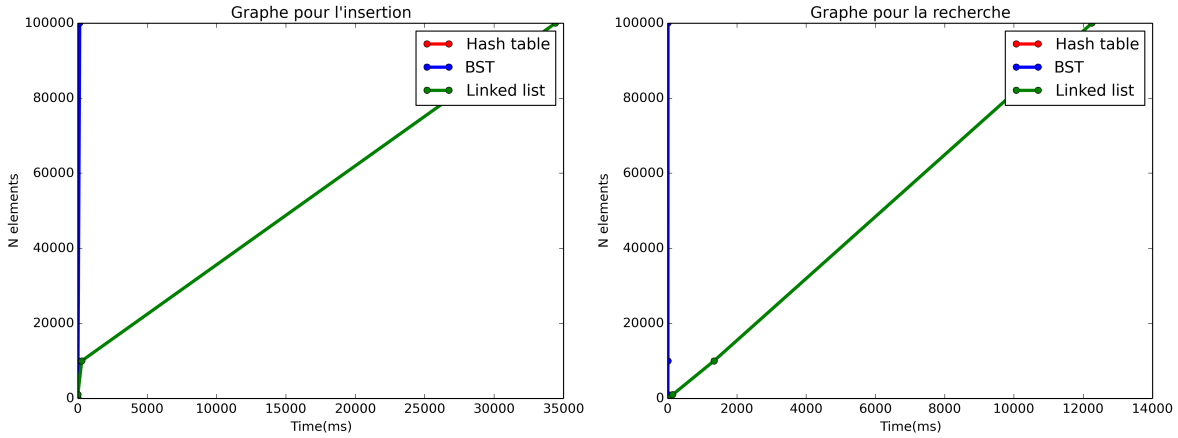

figure1

On remarque que le Hash table et Binary Search Tree(BST) ont des meilleurs temps d'execution dans les deux cas comparés au linked list. Ceci explique d'ailleurs la petitesse de la representation du BST et l'absence du hash table sur le graphe car ayant des temps d'exécution trop pétits que nous avons reporté dans les tableaux ci-dessus.

1.2.4 Discussion

Oui, les résultats expérimentaux sont proches des résultats théoriques, pour la fonction setIntersection. Pour la table de hachage, pour le temps moyen, on avait trouvé $O(N_A)$. Étant donné que pour tous les tests, la taille de A n'a pas changé, les temps obtenus sont sensiblement les même en millisecondes peut importe comment on fait varier le NB. Les valeurs situées entre 5.582ms et 8.65ms.
Pour l'arbre binaire, on avait trouvé comme complexité $O(N_A log_2 N_B)$, ce qui est correcte puisque, si on prend par exemple les deux derniers cas, Soit $N1 = 10^4$, $t1 = 8.075$ et $N2 = 10^5$ et $t2 = 10.474$. Comme N_A ne varie pas alors, on constate que ces rapports sont sensiblement égaux, voir:

$$\frac{(log_2 N2)}{(log_2 N_1)} = 1.25 \text{ et } \frac{t1}{t2} = 1.29$$

2 Intersection de deux fichier

2.1 Implementation

2.2 Analyse

2.2.1

La complexité est $O(N_A N_B)$.
Justification: Étant donné qu' il s'agit de deux tableaux, il faut parcourir tous les éléments de A et à chaque fois parcourir tous les éléments de B pour vérifier l'intersection

2.2.2

(a) Approche 1:

- Table de hachage: La complexité est $O(N_A N_B)$ au pire des cas car, il faut parcourir tous les éléments de B et rechercher dans la table de hachage qui contient trop de collisions

- Arbre binaire: La complexité est $O(N_A N_B)$ au pire des cas car, il faut parcourir tous les éléments de B et rechercher dans l'arbre binaire qui, s'il est trop déséquilibre peut être réduit à une liste chainée dont la recherche a pour complexité $O(N_A)$.

(b) Approche 2:

- Table de hachage: La complexité est $O(N_A N_B)$ au pire des cas ,$S_A$et S_B contiennent beaucoup de collisions donc le parcours de S_B ,la table de hachage a pour complexité $O(N_B)$ et la recherche dans S_A ,la table de hachage a pour complexité $O(N_A)$.
- Arbre binaire: La complexité est $O(N_A N_B)$ au pire des cas ,les deux arbres binaires S_A et S_B sont très déséquilibrés et sont assimilables à des listes chainées donc le parcours de B a pour complexité $O(N_B)$ et la recherche dans S_A a pour complexité $O(N_A)$

(c) Dans le cas moyen
Approche 1:

- Table de hachage: La complexité est $O(N_B)$ au cas moyen car, il faut parcourir tous les éléments dans B ,la complexité étant $O(N_B)$ et rechercher dans la table de hachage avec peu de collisions chaque element de B la complexite de recherche dans la table de hachage est $O(1)$.
- Arbre binaire: La complexité est $O(N_A log_2 N_B)$ dans le cas moyen, il faut parcourir tous les éléments de B et la complexité dans le cas moyen est $O(N_B)$ et rechercher dans l'arbre binaire qui, s'il est assez équilibré , permettra le parcours de complexité $O(log_2(N_A))$.

Approche 2:

- Table de hachage: La complexité est $O(N_B)$ au cas moyen car, il faut parcourir tous les éléments dans une table de hachage S_B , la complexité étant $O(N_B)$ et rechercher dans la table de hachage avec peu de collisions chaque element de S_B la complexité de recherche dans la table de hachage est $O(1)$.
- Arbre binaire: La complexité est $O(log_2(N_A) log_2(N_B))$ dans le cas moyen, il faut parcourir tous les éléments de S_B et la complexité du parcours dans cet arbre binaire dans le cas moyen est $O(log_2(N_B))$ et rechercher dans l'arbre binaire qui, s'il est assez équilibre, permet le parcours de complexité $O(log_2(N_A))$.

(d) En conclusion, Au pire des cas les deux approches se valent mais au cas moyens ,la deuxième approche est meilleure compte tenu de la complexité meilleure pour la table de hachage et l'arbre binaire.

2.2.3

Si A et B sont triés, voici comment procéder Parcourir les éléments de A et rechercher dichotomiquement chacun des element de A dans B comme N_A est inferieure ou égale à N_B cela réduit la complexité. Au pire des cas, la complexité est de $O(N_A N_B)$ Au moyen des cas, la complexité est de $O(N_A log_2 N_B)$, à cause de la recherche dichotomique.