INFO0250 - Programmation avancée

Projet 2 : Dictionnaires

Groupe s180498-s170220: Martin RANDAXHE, Cyril RUSSE

Table des matières

1 Ensemble

1.1 Implémentation

(1.1.a)

1.1.1 Arbre Binaire

Pour les arbres binaires, nous avons choisi d'implémenter un AVL. Nous allons passer en revue certaines fonctions afin d'expliquer les choix que nous avons faits.

Tout d'abords, au niveau des structures, elles se comptent au nombre de 2.

La structure "Set" principale étant définie dans le .h. Celle-ci contient la racine de l'arbre et la taille. La taille n'est pas très utile mais étant donné qu'il nous a été demandé d'implémenter la fonction permettant de savoir le nombre d'élément dans l'abre, dès lors, nous l'incrémentons de 1 à chaque insertion.

Nous avons créé une seconde structure propre à cette implémentation : "noeud", qui correspond, comme son nom l'indique, à un noeud de l'arbre. Celle-ci contient des pointeurs vers ses fils et son parent(pour la racine il sera nul) et une variable permettant de savoir la taille max de ses sous-arbres, cette variable sera d'une importance cruciale lors du rééquilibrage. Elle contient évidemment la chaine de caractères étant l'élément inséré.

Pour la libération de mémoire, nous sommes tout simplement parti sur une fonction récursive que nous pourrions comparer à une fonction permettant de savoir la hauteur de l'abre comme nous avions vu dans le cours.

Pour ce qui est de l'insertion, nous cherchons l'emplacement où ajouter la nouvelle valeur en partant de la racine. Si celle-ci est nulle, nous le rajoutons à la racine, sinon nous l'ajoutons en tant que fils droit ou gauche du noeud parent que nous venons de trouver. Après cette insertion nous faisons appel aux fonctions d'équilibrage de l'arbre afin de rechercher un éventuel déséquilibre et de le corriger. Nous veillons évidemment aussi à mettre à jour la racine du set en cas de modification de celle-ci après avoir fait appel à l'équilibrage.

Nous en venons donc à la partie la plus importante de cet algorithme qui est l'équilibrage de l'arbre. Pour nous facilitez la tâche, nous avons, comme annoncé précédemment, pour chaque noeud une variable pour la taille max de ses sous-arbres. A chaque insertion, nous utilisons une fonction qui met à jour ces valeurs. Pour ce faire, nous faisons une remontée récursive depuis le nouveau noeud jusqu'à la racine et nous mettons à jour la variable de chaque noeud en comparons les valeurs de hauteurs de chaque fils, la nouvelle valeur de hauteur est dorénavant la hauteur du plus grand sous-arbre + 1.

Pour ce qui est de l'équilibrage en lui-même, à l'aide de ces hauteurs de sous-arbres nous faisons, comme pour l'ajustement des hauteurs, une remontée récursive comparant les hauteurs de sous arbres et dans le cas où leur différence est supérieur à 1 en valeur absolue nous définissons de quel type de désiquilibre nous avons affaire au vue de cette valeur et nous vérifions si nous sommes dans le cas d'un désiquilibre intérieur en vérifiant quels sous-arbre est le plus grand du côté du désiquilibre. Ensuite, nous appliquons des rotations adéquates à la situation. Les rotations s'occupent évidemment de mettre à jour les fils/parents de tous les noeuds nécessitant des changements.

1.1.2 Table de Hash

Comme pour notre arbre binaire, nous avons 2 structures, l'une étant l'implémentation de "Set" défini dans le fichier h, comprenant un tableau de liste chainée, la taille du tableau et le nombre d'éléments dans la table de hash. L'autre structure "Liste" est l'élément qui correspond aux cellules que nous allons utiliser pour faire nos listes chainées. Celle-ci contient uniquement la chaine de caractères étant l'élément inséré et un pointeur vers la cellule suivante de la liste chainée, étant nul si nous sommes au dernier élément de la liste.

Pour ce qui est de la création et de la destruction du set, il n'y a pas grand chose de particulier. Nous avons juste choisi, comme valeur faible pour la taille initiale de notre tableau, 64.

Notre choix de table de hash est basé sur une implémentation en adressage fermé. Ce qui veut dire que nous gérons les superpositions par le biais de listes chainées. Nous sommes parti du principe de vouloir promouvoir l'efficacité à la mémoire donc pour notre stratégie de hachage, nous procédons à un réhachage dans le cas où le nombre d'élément est égal à la taille du tableau ce qui correspondrait dans le meilleur des cas, à ce stade là, à un tableau de n cases contenant chacune une liste de une cellule. Si nous dépassons cette limite, nous doublons la taille du tableau et nous replacons tous les éléments déjà insérés à leur place dans le nouveau tableau. Nous en venons donc à notre fonction de hachage. Celle-ci correspond à la formule suivante :

$$\sum_{i=0}^{n}(elem[i] * 727^i)\%m$$

avec

- elem, la chaine caractère
- n, le nombre de caractère dans elem
- m, la taille du tableau de liste du set

D'où le fait que nous devons repasser tous les éléments déjà insérés dans la fonction de hachage avec la nouvelle taille de tableau, afin de revoir sa postion dans celui-ci.

Enfin, notre fonction "contains", permet de vérifier l'existence d'un élément en utilisant, l'utilité principale d'une table de hachage, l'accès direct. Nous passons donc l'élément dans notre fonction de hachage et vérifions les éléments(s'il y en a) à la position du tableau que nous a renvoyé notre fonction de hachage.

1.2 Fonction d'Intersection

(1.1.b)

1.3 Complexité "SetIntersection"

(1.1.c 1.1.d) Etant donné les choix d'implémentation que nous avons fait et que nous détaillerons dans la seconde section nous n'utilisons pas la fonction "SetIntersection". En effet, au vue de notre choix nous n'avons pas besoin de créer 2 set ce qui est un gain et qui explique que nous ne produisons pas de fonction propre à chaque implémentation et qui ne nous permet pas d'utiliser ces fonctions étant donné que leur définition nous est donné avec 2 set en arguments. Nous utiliserons plutot la fonction GetIntersection directement dans StringArray.c pour établir les intersections.

2 Intersection de deux fichiers

2.1 Complexité minimale

(2.1)
La complexité minimale consisterait à trié un des tableaux donc $\Theta(n * log(n))$ et puis faire une recherche dichotomique pour chaque élément de l'autre tableau, également $\Theta(n * log(n))$. Donc au global, $\Theta(n * log(n))$ également.

2.2 Analyse d'implémentations

(2.2.a)
Première approche :

- Arbre Binaire Cela revient au même si ce n'est que si l'on prend en compte le temps pour créer les arbres, cette approche à l'avantage de n'en créer qu'un.
- Table de Hash Cette approche a aussi l'avantage de ne créer qu'une seule table et la complexité restera quadratique en pire cas.

(2.2.b)
Deuxième approche :

- Arbre Binaire Avoir 2 arbres binaires ne nous apporterait pas grand chose. Tel est la technique que nous avons utilisé, nous parcourons le premier arbre et cherchons dans le deuxième. Nous obtenons, comme prouvé dans la précédente section, une complexité $\Theta(n * log(n))$. Un avantage aurait pu être de faire une implémentation se rapprochant du mergesort comme expliqué au point 2.3 mais nous avons remarqué que pour parcourir l'abre de manière croissante, nous devrions passé par une fonction permettant d'avoir le successeur du noeud actuel et ces montées et descente dans l'abre revienne presqu'au même que les descente que l'on fait lors de notre recherche par la fonction "contains".
- Table de Hash Pour la table de Hash, même chose, nous parcourerons l'entierté d'une d'entre elles et le pire cas consiste en tous les éléments de la seconde dans une seule liste qui implique une complexité quadratique.

2.3 Cas moyen

(2.2.c)
Les conclusions que nous avons faite pour le pire cas restent inchangées, et nos complexités correspondent aux complexités en cas moyen exprimée dans nos analyses au point 1.1.d.

2.4 Pertinence d'approche

(2.2.d)
De par les points précédents, il parait plus judicieux de ne créer qu'un seul ensemble. Car le fait d'en créer 2 n'apporte pas grand chose et implique de prendre le temps de créer 2 ensembles qui n'est pas gratuit en mémoire ni en temps. Pour ce qui est du type d'ensemble, l'arbre binaire est dans tous les cas plutot éfficaces avec une complexité $\Theta(n * log(n))$. Pour ce qui est de la table de Hash, en moyenne, c'est la plus efficace avec une complexité linéaire, mais elle a en pire cas, bien que peu probable, une complexité $\Theta(n^2)$.

2.5 Complexité avec 2 ensembles triés

(2.3)
Nous pourrions procéder d'une manière similaire au mergesort, en effet nous pourrions parcourir

les 2 tableaux parallèlement et si l'on trouve un élément similaire on l'ajoute à notre tableau intersection. En terme de complexité, dans le meilleure cas nous ne parcourerions que la plus petit des 2 tableaux donc $N_A\Theta(n)$. En pire cas, ça serait le même cas qu'un merge de A et B, donc $N_A + N_B$ ce qui revient encore une fois à du $\Theta(n)$.