

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'abord, 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'arbre, 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 chaîne 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'arbre comme nous avons 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 faciliter 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 comparant 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 déséquilibre nous avons affaire au vu de cette valeur et nous vérifions si nous sommes dans le cas d'un déséquilibre intérieur en vérifiant quels sous-arbre est le plus grand du côté du déséquilibre. 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 chaînée, 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 chaînées. Celle-ci contient uniquement la chaîne de caractères étant l'élément inséré et un pointeur vers la cellule suivante de la liste chaînée, 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 chaînées. Nous sommes parti du principe de vouloir promouvoir l'efficacité à la mémoire donc pour notre stratégie de hachage, nous procédons à un rehachage 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 remplaçons 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 chaîne 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 position 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)

Le principe de notre fonction d'intersection correspond à la première approche proposée dans l'énoncé au point 2.2. Nous cherchons l'intersection donc directement depuis la fonction "GetIntersection" dans laquelle nous créons un set pour le plus petit des 2 StringArray. Ensuite, nous parcourons tous les éléments du second StringArray en utilisant la fonction "contains" pour chaque élément dans le set créé du premier StringArray.

Nous considérerons pour ce pseudo-code $N_A < N_B$, et A et B des tableaux de chaîne de caractère. La fonction "createSet" correspond à l'insertion de tous les éléments d'un tableau en un set. La fonction contains retourne vrai ou faux en fonction de si l'élément donné en second argument existe dans le set donné en premier argument. Et enfin, la fonction add ajoute la chaîne de caractère en second argument dans le tableau de chaîne de caractère en premier argument.

```

1  GetIntersection(A, B)
2      setA = createSet(A)
3      for i=1 to B.length
4          if(contains(setA, B.elem))
5              add(intersection, B.elem)

```

1.3 Complexité "SetIntersection"

(1.1.c et 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 plutôt la fonction GetIntersection directement dans StringArray.c pour établir les intersections.

1.4 Temps d'exécutions

(1.2)

Nous allons maintenant étudier des valeurs de temps d'exécutions. Les valeurs répertoriées dans le tableau ci-dessous sont basées sur 10 tests pour lesquels nous avons mélangé les tableaux French.txt et English.txt entre chaque test. De cette base, nous avons enregistré les temps d'exécution pour la création du set à partir de French.txt sur ses 1000 ou 10000 premiers éléments. Nous avons fait de même en répertoriant les temps pour rechercher les intersections entre English.txt et notre set de French.txt ayant 1000 ou 10000 éléments.

Type d'implémentations	Arbre Binaire		Table de Hash	
Taille	10^3	10^4	10^3	10^4
Creation Set	0.000535	0.007985	0.000624	0.00779
Intersection	0.201887	0.221764	0.206211	0.2121

FIGURE 1 – Résultats empiriques de temps d'exécution

1.5 Comparaison avec les résultats théoriques attendus

(1.3)

2 Intersection de deux fichiers

2.1 Complexité minimale

(2.1)

La complexité minimale consisterait à trier 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

Quelle que soit le cas, notre algorithme crée toujours, dans le cas où nous ne faisons qu'un set, le set sur le tableau A étant le plus petit. En effet, la création d'un set est coûteux et il est préférable de le faire sur le plus petit dans le cas où nous ne convertissons pas les 2 tableaux en set. (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 a 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 Cette deuxième approche n'apporterait rien d'intéressant en plus étant donné qu'il faudrait créer un second set et que nous n'avons pas d'algorithme permettant de prendre avantage au fait que les deux bases de données soient enregistrées d'une manière particulière. Nous aurons donc 2 fois $n * \log(n)$ opérations pour créer les 2 sets et $n * \log(n)$ pour la recherche d'intersection.
- Table de Hash Même réflexion que pour l'arbre binaire, nous devrions créer 2 tables ce qui n'est pas gratuit et ne nous avantagerait pas plus pour la recherche d'intersection. En pire cas, nous sommes une nouvelle fois dans une complexité quadratique. Dans le cas où tous les éléments seraient mal répartis et tous dans une seule et unique liste chaînée.

2.3 Cas moyen

(2.2.c)

- Arbre Binaire Étant donné que l'on considère que A et B n'ont pas de doublons, dès lors nous parcourerons en moyenne $\log(n)$ pour atteindre l'élément s'il existe dans le set. Donc toujours $\Theta(n * \log(n))$ comme complexité en cas moyen.
- Table de Hash En moyenne, on considérera une répartition correcte dans la table et donc un accès $O(1)$ pour la fonction contains. Donc nous aurons en cas moyen une complexité linéaire pour le calcul de l'intersection.

2.4 Pertinence d'approche

(2.2.d)

De par les points précédents, il paraît 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 plutôt efficace 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 meilleur cas nous ne parcourerions que le 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 $\Theta(n)$.