Rapport de projet (SDA 2)

Paul HENG Cyril LAJARGE

Mai 2018

Ensemble ordonné

Implémentation

Notre ensemble de données est constitué de 2 parties. A la manière d'une liste chaînée, nous utilisons un système de maillons contenant un élément (ici un entier) et un pointeur vers l'élément suivant. Après réflexion, nous avons également choisi d'utiliser un couple de pointeurs : l'un vers le début de l'ensemble, et l'autre vers sa fin. Ce choix a été motivé par la nécessité d'optimiser la complexité de certaines opérations. En effet, avoir un accès direct au dernier élément permet d'éviter de faire un parcours entier de l'ensemble quand on souhaite effectuer une insertion à la fin de l'ensemble, et cela permet également de faire des tests préliminaires en cas d'intersection nulle entre 2 ensembles.

La fonction *intersection*

L'implémentation de cette fonction commence par des tests préliminaires, retournant immédiatement un ensemble vide si une des conditions suivantes est remplie :

- les 2 ensembles entrés en arguments sont vides
- l'un des deux ensemble est strictement plus grand que l'autre ($\Rightarrow$ le dernier élément d'un ensemble est strictement plus petit que le premier élément de l'autre)

Si aucune des conditions n'est vérifiée, on effectue le parcours simultané des 2 ensembles en commençant au premier élément de chacun. A chaque fois, un test est effectué pour comparer les 2 éléments courants, soient e_1 et e_2 des éléments respectivement du premier et du second ensemble :

- si $e_1 < e_2$, on passe à l'élément suivant pour le premier ensemble
- si $e_1 > e_2$, on passe à l'élément suivant pour le second ensemble

- si $e_1 = e_2$, on ajoute cet élément à l'ensemble de retour de la fonction, et on passe à l'élément suivant pour les deux ensembles

L'itération s'arrête quand on arrive à la fin d'un des deux ensembles.

Pire cas et complexité

Insertion en fonction de n

Le pire cas serait l'insertion d'un élément à la l'avant-dernière place, forçant à effectuer un parcours quasi complet de l'ensemble. L'insertion en fin d'ensemble ne pose aucun problème étant donné que l'accès est immédiat grâce au pointeur en fin d'ensemble.

La complexité est en $\theta(n)$

Appartenance d'un élément en fonction de n

Le pire cas possible est que l'élément soit compris entre l'avant dernier élément (inclus) et le dernier élément (exclus) de l'ensemble ordonné, soit pour un ensemble $e_1, e_2, .., e_n$ et un élément x

$$e_{n-1} \leq x < e_n$$

La complexité est en $\theta(n)$

Intersection de deux ensembles

Le pire cas possible est que les deux ensembles soient de même taille et qu'une des conditions soit remplie :

- leur dernier élément soit identique
- le dernier élément de l'un est le seul strictement supérieur au dernier élément de l'autre

La complexité est en $\theta(m)$

Arbre binaire de recherche

Implémentation

Notre structure d'arbre est en deux parties. La première partie est une structure contenant une chaine de caractère et un pointeur sur un ensemble ordonné. La

seconde partie, qui est la partie principale, est une structure composée d'un élément de la première structure ainsi que de deux pointeurs sur la seconde structure, l'un pour l'arbre droit, l'autre pour l'arbre gauche.

La fonction *getAverageDepth*

La fonction *getAverageDepth* utilise une fonction auxiliaire appelée *getTotalDepth* . Elle utilise le résultat de cette fonction auquel elle divise le résultat de la fonction *getNumberString* qui renvoie le nombre de mots différents et donc le nombre de noeuds dans un arbre afin de renvoyer la hauteur moyenne d'un noeud de l'arbre.

La fonction *getTotalDepth*

La fonction *getTotalDepth* calcule la somme des profondeurs de tous les noeuds d'un arbre. Elle prend donc pour argument un pointeur sur un arbre.

Cette fonction est récursive.

- Si le pointeur sur l'arbre est nulle, on retourne 0. C'est la condition d'arrêt.
- Sinon, on calcule la profondeur de l'arbre grâce à la fonction *getHeight* à laquelle on ajoute l'appel de la fonction *getTotalDepth* avec l'arbre droit puis avec l'arbre gauche.

La fonction *isBalanced*

La fonction *isBalanced* détermine si un arbre donné est équilibré 1ou non. Elle prend pour argument un pointeur sur un arbre.

- Si le pointeur sur l'arbre est nul, on retourne 0, c'est à dire vrai.
- Sinon:
  - Si la différence de profondeur(en valeur absolue) de l'arbre droit et l'arbre gauche est plus petite que 1 et que l'arbre droit ainsi que l'arbre gauche sont équilibrés, on retourne 0.
  - Sinon, on retourne 1, c'est à dire faux.

Complexité de la fonction *FindCooccurences*

Comparaison de la hauteur moyenne d'un noeud entre un arbre non équilibré et un arbre équilibré

La hauteur moyenne d'un noeud d'un arbre non équilibré est supérieure à celle d'un arbre équilibré contenant le même nombre de noeud. En effet, avec l'exemple

du sujet, en ayant l'arbre équilibré, on obtient une hauteur moyenne de 0,846154. En revanche, si il n'est pas équilibré, la hauteur moyenne est 1,230769.

Complexité dans le pire des cas pour la fonction *FindCooccurences* pour un arbre non équilibré

En supposant qu'on recherche deux mots dans un arbre binaire de recherche non équilibré contenant n mots et qu'un mot apparaît au plus dans k phrases différentes, il faudra boucler sur les deux mots qu'on recherche. De plus, dans le pire des cas, ces mots vont apparaître en même temps dans k différentes phrases. Il faudra donc appeler la fonction *intersect* à k reprises. En supposant que ces k phrases contiennent l'ensemble des mots de l'arbre, la complexité de *intersect* est dans le pire des cas en $\theta(n * n)$. Ces calculs sont effectués par la fonction *FindCooccurencesAux*. Cette fonction est récursive et elle est appelée n fois. La fonction *FindCooccurences* effectue une boucle sur les mots recherchés pour savoir s'ils existent. La fonction *exist* parcours dans le pire des cas tout l'arbre donc sa complexité est en $\theta(n)$. *FindCooccurences* appelle ensuite *FindCooccurencesAux*. On a donc une complexité qui est en: $\theta(k * n * n * n + n)$ soit $\theta(k * n^3 + n)$

Complexité de la fonction *FindCooccurences* pour un arbre équilibré

Dans le pire des cas, l'équilibrage de l'arbre ne change pas la complexité. En effet, dans ce cas, il est toujours nécessaire de parcourir l'ensemble des noeuds de l'arbre. En revanche, en général(pire des cas exclus), la fonction sera plus rapide car l'apparition des mots se fera potentiellement plus rapidement vu que la hauteur moyenne des noeuds est plus faible que celle de l'arbre non équilibré.