Programmation avancée

Répétition 5: Dictionnaires

Jean-Michel Begon

novembre 2014

Exercice 1

Soit un arbre binaire de recherche `T` initialement vide.

(a) Déterminez graphiquement l'arbre qui résulte de l'insertion des clés 30, 40, 24, 58, 48, 26, 11, 13, 35 et 36 (dans cet ordre).

(b) Déterminez ensuite graphiquement l'arbre qui résulte de la suppression des clés 13, 58, et 30 (dans cet ordre).

Exercice 2

Soit un ensemble de N valeurs entières distinctes. Ecrivez une fonction calculant le nombre d'arbres binaires de recherche distincts qu'il est possible de construire à partir de ces N valeurs.

Exercice 3

Si on insère un élément z dans un arbre `T` (via `TreeInsert(T, z)`) et puis qu'on supprime cet élément (via `TreeDelete(T, z)`), l'arbre `T` qui en résulte est-il nécessairement identique à l'arbre de départ ?

Exercice 4

La suppression d'une valeur dans un arbre binaire de recherche peut-elle augmenter la hauteur de cet arbre ?

Exercice 5

Soit une table de hachage `D` initialement vide, d'une capacité de 11 éléments et dont les collisions sont gérées au moyen de listes simplement liées. Illustrer graphiquement l'insertion des clés

$$\{23, 56, 44, 13, 88, 94, 16, 27, 9\}.$$

La fonction de hachage utilisée est

$$h(k) = (2k + 5) \mod 11.$$

Exercice 6

Soit une table de hachage `D` initialement vide, d'une capacité de 11 éléments et dont les collisions sont gérées par sondage linéaire avec la fonction de hashage $h(k,i) = (h'(k) - i) \mod 11$ (la fonction de hashage h' est la même que dans l'exercice précédent). Illustrer graphiquement l'insertion des clés

$$\{23, 56, 44, 13, 88, 94, 16, 27, 9\}.$$

Exercice 7

Soit une table de hashage à adressage ouvert `D` initialement vide, d'une capacité initiale de 4 éléments avec un facteur de charge $\alpha = 0.75$ (lorsque la charge dépasse 75%, on copie le contenu de la table dans une nouvelle de capacité double). La fonction de hachage utilisée est

$$h(k,i) = (h'(k) - i) \mod m$$

où m représente la capacité courante de la table. Illustrer graphiquement l'insertion des clés

$$\{23, 56, 44, 13, 88, 94, 16, 27, 9\}.$$

Quel est le temps moyen d'accès à un élément (la moyenne des temps d'une recherche positive) ?

Exercice 8

Dans le JDK1.1, la fonction de hachage d'une chaîne de caractères était implémentée de la façon suivante. Quel est le défaut de cette fonction ?

```
public int hashCode() {
    int hash = 0;
    int skip = length() / 8;
    for (int i = 0; i < length(); i = i + skip) {
        hash = (hash * 37) + charAt(i);
    }
    return hash;
}
```

Exercice 9

Désormais, la fonction de hachage d'une chaîne de caractères est implémentée en Java de la façon suivante :

```
public int hashCode() {
    int hash = 0;
    for (int i = 0; i < length(); i++) {
        hash = (hash * 31) + charAt(i);
    }
    return hash;
}
```

Trouvez des chaînes de caractères de longueur N produisant le même hash.

Exercice 10

Quelle est la complexité pour trouver l'élément minimum dans :
— un arbre de recherche ;
— un arbre de recherche équilibré ;
— une table à adressage direct ;
— une table de hachage.

Ecrire une méthode `printLowerThan(T, x)` qui imprime les valeurs contenues dans un arbre binaire de recherche `T` inférieures ou égales à x. Quelle est la complexité de cette opération ?

Bonus

Bonus 1

Trouvez deux constantes c_1, c_2 telles que le sondage quadratique avec $m = 11$ fonctionne. C'est-à-dire, telles que :

$$h(i) = c_1 * i + c_2 * i^2$$

soit bien une permutation des 11 premiers naturels.

Bonus 2

On souhaiterait représenter des matrices creuses (*i.e.* contenant "beaucoup" de zéros) à l'aide de table(s) de hashage.

(a) A quoi correspondent les clés ?
(b) Proposer une implémentation (structure de données, insertion, recherche en ligne ou en colonne, accès) efficace.
(c) Quelle est la complexité des opérations d'insertion et d'accès aux éléments ?
(d) Proposer un algorithme de multiplication matricielle pour les matrices creuses.
(e) Peut-on faire mieux qu'avec une table de hashage ?

Bonus 3

On désire implémenter un dictionnaire sur un composant disposant de (relativement) peu de mémoire. Le composant alterne entre trois phases :

(a) Phase 1 : le composant reçoit des données externes qu'il stocke dans le dictionnaire.
(b) Phase 2 (transition) : sur base d'un signal externe, le composant défausse une grande partie des informations contenues dans le dictionnaire (on ne sait pas prédire lesquelles à l'avance).
(c) Phase 3 : le composant va générer beaucoup de données sur base de ce qui se trouve dans le dictionnaire. Il envoie ensuite les données générées sur une sortie et recommence la phase 1.

Afin de garantir la terminaison de la phase 3 à temps, le composant doit disposer d'un dictionnaire très efficace. On décide donc d'utiliser une table de hashage par chainage. Le tableau sous-jacent doit être de petite taille afin d'avoir assez d'espace mémoire pour la phase 3. On sait donc qu'il y aura de nombreuses collisions lors de la phase 1.
On se propose d'implémenter la politique de chainage suivante : on maintient les listes de collision triées de manière à gagner un maximum de temps lors de la phase 3 (pour des raisons pratiques il est impossible de réorganiser la table de hashage lors de la phase 2).
Quel est l'impact sur le temps des opérations de recherche (fructueuses ou non) ainsi que sur les insertions et suppressions ?