Algorithmique TD5

Magistère Informatique 1ère Année

16 Octobre 2008

Exercice 1 *Arbres AVL* (Extrait Partiel 2006)

Un AVL est un arbre binaire de recherche (ABR) tel que pour chaque nœud de l'arbre, la différence de hauteur entre le sous-arbre gauche et le sous-arbre droit est d'au plus 1.

a) Soit n le nombre de nœuds et h la hauteur d'un AVL (la hauteur d'un arbre réduit à sa racine est 1). Montrer que $F_{h+2}-1 \leq n \leq 2^h-1$ où F_k est le $k$ième nombre de Fibonacci avec $F_0 = 0, F_1 = 1$ et $F_{k+2} = F_{k+1} + F_k$ pour $k \geq 0$. Montrer que ces bornes sont atteintes.

Montrer que pour $k \geq 0$ on a $F_{k+2} \geq \phi^k$ où $\phi = \frac{1+\sqrt{5}}{2}$. En déduire que $log_2(n+1) \leq h \leq log_\phi(n+1)$ et que la recherche dans un AVL se fait au pire en temps $O(log(n))$.

Pour les algorithmes, un nœud x d'un AVL sera représenté par une structure comportant
- une clé, notée $x \cdot cle$ d'un type totalement ordonné (par exemple de type entier) ;
- un sous-arbre gauche, noté $x \cdot g$ et un sous-arbre droit, noté $x \cdot d$;
- la hauteur de l'arbre, notée $x \cdot h$.

L'arbre vide est noté $NULL$. On notera $h(x)$ la hauteur d'un arbre x avec $h(NULL) = 0$. On notera $n(x)$ le nombre de nœuds d'un arbre x.

b) Le but de cette question est d'écrire une procédure $EQUILIBRER(x)$ pour transformer en AVL en temps constant un ABR de racine x en supposant que ses deux sous-arbres sont des AVL et que la différence de hauteur entre les deux sous-arbres est d'au plus 2.

Proposer un rééquilibrage dans le cas où $h(a \cdot g) - h(a \cdot d) = 2$. On pourra distinguer les cas $h(a \cdot g \cdot g) \geq h(a \cdot g \cdot d)$ et $h(a \cdot g \cdot g) < h(a \cdot g \cdot d)$. Illustrer les transformations sur des dessins. Ecrire la procédure $EQUILIBRER(x)$.

c) Ecrire un algorithme pour insérer une clé c dans un AVL x en temps au pire $O(1 + h(x))$. Si x' est l'arbre obtenu, comparer $h(x')$ et $h(x)$. Justifier la correction de l'algorithme. Montrer que cet algorithme engendre au plus un rééquilibrage.

d) Ecrire un algorithme pour supprimer une clé c dans un AVL x en temps au pire $O(1 + h(x))$. Justifier la correction de l'algorithme (on ne demande pas une preuve de programme). Combien de rééquilibrages peuvent être nécessaires ?

e) Ecrire un algorithme qui réalise la fusion de deux AVL x et y et d'une clé c en supposant que toutes les clés de x sont strictement inférieures à c et que toutes les clés de y sont strictement supérieures à c. Cet algorithme devra fonctionner en temps $O(1 + |h(x) - h(y)|)$. Justifier.

f) Ecrire un algorithme qui réalise la scission d'un AVL x en deux AVL y et z contenant respectivement les clés de x inférieures ou égales à c pour y et strictement supérieures à c pour z. Cet algorithme devra fonctionner en temps $O(1 + |h(x)|)$. Justifier.

Exercice 2 *Tables de Hachage*

Sous l'hypothèse de hachage uniforme montrer que si on tire n clés, la probabilité qu'il y ait au moins une collision est :

$$1 - \frac{m!}{(m-n)!} \cdot \frac{1}{m^n} \approx 1 - e^{-\frac{n^2}{2m}}$$

Exercice 3 *Adressage Ouvert*

On prend une table de hachage où les collisions sont gérées par adressage ouvert Le but est de calculer le nombre de sondages moyen effectué lors d'une recherche *fructueuse* ou *infructueuse*. On appelle m la taille de la table et n le nombre de clés qu'elle contient. Soit c une clé.

1. Exprimer le nombre de configurations pour lesquelles une recherche *infructueuse* de c effectue k sondages.
2. Montrer que :

$$\sum_{0 \leq i \leq l-1} C_i^{l-j-1} = C_l^j$$

   En déduire le coût moyen d'une recherche *infructueuse* :

$$\frac{m+1}{m+1-n}$$

3. Exprimer le coût moyen d'une recherche *fructueuse* de c en fonction du coût des recherches infructueuses qui ont permis la construction de la table. En déduire que ce coût moyen est :

$$\frac{m+1}{n}(H_{m+1} - H_{m-n+1})$$

   Où $H_j = 1 + \frac{1}{2} + \cdots + \frac{1}{j}$, est le $j$ième nombre harmonique.

Exercice 4 *Tris par tas*

1. Transformer un tableau en tas binaire.
2. Transformer un tas binaire en tableau trié.
3. Etudier la complexité de ces opérations.
4. Monter que la construction du tas peut se faire en $O(n)$.

Exercice 6 *Arbres binomiaux*

On définit une suite d'arbres $(B_n)_{n>0}$ par réccurrence sur n comme suit : B_0 est l'arbre à un seul sommet, B_{n+1} est formé de la réunion de deux copies disjointes de B_n, B_n^g et B_n^d. La racine de B_{n+1} est celle de B_n^d, la racine de B_n^g étant le fils le plus à gauche de celle-ci. L'arbre B_n est l'arbre binomial d'ordre n.

1) Démontrer que dans B_n, il y a exactement C_n^k sommets de profondeur k.

2) Démontrer qu'un seul sommet de B_n a n fils, et que pour $0 \leq k < n$, 2^{n-k-1} sommets ont k fils.

3) On numérote les sommets de B_n de 0 à $2n-1$ de la gauche vers la droite en ordre postfixe (fils avant le père). Montrer qu'un sommet est de profondeur $n-k$ si et seulement si son numéro a k "1" en écriture binaire.

4) Montrer que le nombre de fils d'un sommet est égal au nombre de "1" qui suivent le dernier "0" dans l'écriture binaire de son numéro.

Etant donné un entier n, soit :

$$n = \sum_{i \geq 0} b_i 2^i; \qquad b_i \in \{0, 1\}$$

sa décomposition en base 2, soit $I_n = \{i \mid b_i = 1\}$, et soit $\nu(n) = Card(I_n)$. La forêt binomiale d'ordre n est l'ensemble $F_n = \{B_i \mid i \in I_n\}$. La composante d'indice i de F_n est B_i , si $i \in I_n$, et $\emptyset$ sinon.

5) Montrer que F_n a $n - \nu(n)$ arrêtes.

Une file binomiale est une forêt binomiale dont chaque sommet x est muni d'une clé $c(x)$, vérifiant : si y est fils de x, alors $c(y) > c(x)$.

6) Montrer que la recherche du sommet de clé minimale dans une file binomiale F_n peut se faire en $\nu(n) - 1$ comparaisons.

Soient F_n et $F_{n'}$ deux files binomiales ayant des ensembles de clés disjoints. On définit l'opération $UNION(F_n, F_{n'})$ qui retourne une file binomiale ayant pour clés l'union des ensembles de clés comme suit :

- Si $n = n' = 2^p$, alors $F_n = \{B_p\}$, $F_{n'} = \{B'_p\}$, et $UNION(B_p, B'_p)$ est l'arbre B_{p+1} pour lequel $B^g_p = B_p$ et $B^d_p = B'_p$ si la clé de la racine de B_p est plus grande que la clé de la racine de B'_p et l'inverse dans le cas contraire.
- Dans le cas général, on procède en commençant avec les composantes d'indice minimal des deux files, et en construisant une suite d'arbres report. L'arbre report R_0 est vide, et à l'étape $k > 0$, l'arbre report R_k est soit vide, soit un arbre binomial d'ordre k. Etant donnés l'arbre report R_k, la composante C_k de F_n et la composante C'_k de $F_{n'}$, on définit la composante C''_k d'ordre k de $UNION(F_n, F_{n'})$ et l'arbre report R_{k+1} comme suit :
  1. Si $R_k = C_k = C'_k = \emptyset$, alors $C''_k = R_{k+1} = \emptyset$;
  2. Si seule l'une des trois opérandes est non vide, elle devient C''_k, et $R_{k+1} = \emptyset$;
  3. Si deux opérandes sont non vides, alors $C''_k = \emptyset$ et R_{k+1} est l'union des deux opérandes ;
  4. Si les trois opérandes sont non vides, l'une devient C''_k et R_{k+1} est l'union des deux autres opérandes ;

7) En supposant que l'union de deux arbres binomiaux de même ordre prend un temps constant, donner une majoration logarithmique en n et n' du temps de $UNION(F_n, F_{n'})$.

8) Proposer une structure de données qui permet de faire l'union de deux arbres binomiaux en temps constant.

9) Montrer que le nombre de comparaisons de clés pour construire $UNION(F_n, F_{n'})$ est $\nu(n) + \nu(n') - \nu(n + n')$.

10) Montrer comment l'insertion, dans une file binomiale, d'une clé qui n'y figure pas déjà peut se ramener à une union.

11) En déduire un algorithme de construction d'une file binomiale pour un ensemble de n clés, et montrer que cela demande au total $n - \nu(n)$ comparaisons.

12) Montrer que la suppression de la plus petite clé dans une file binomiale F_n peut se faire en $O(log(n))$ opérations, y compris les opération nécessaires pour reconstruire une file binomiale F_{n-1} pour les clés restantes.

13) En déduire un algorithme de tri d'une suite de n clés en temps $O(nlogn)$.