Algorithmique et Structures de Données

—

Corrigé de l'examen écrit

—

G1: pascal.monasse*(at)*enpc.fr G2: thomas.belos*(at)*enpc.fr
G3: nicolas.audebert*(at)*lecnam.net G4: clement.riu*(at)*enpc.fr

29/03/2021

Les exercices sont indépendants. Il n'est pas interdit d'utiliser votre portable pour tester vos algorithmes, mais évidemment pas le wifi.

1 Encore un algorithme de tri

1. Étant donnés deux tableaux *triés* d'entiers $T1$ et $T2$ de taille de taille respective $N1$ et $N2$, proposer sous forme de code C++ leur fusion dans un tableau trié T (de taille donc $N = N1 + N2$).

```
void fusion(int T1[],int N1, int T2[],int N2, int T[]) {
    for(int i=0,j=0; i<N1 || j<N2;) {
        while(i<N1 && (j==N2||T1[i]<=T2[j])) {
            T[i+j]=T1[i];
            ++i;
        }
        while(j<N2 && (i==N1||T2[j]<=T1[i])) {
            T[i+j]=T2[j];
            ++j;
        }
    }
}
```

2. Quelle est sa complexité en fonction de N ? À chaque entrée dans un while, i ou j augmente de 1, donc $O(N)$.
3. Proposer suivant ce principe un algorithme triant un tableau de 2^n entiers. Diviser pour régner : on sépare en deux moitiés le tableau, on trie chacun (récursivité) et on fusionne.
4. L'écrire sous forme de code C++ (faisant appel à la fonction `fusion` de la première question).

```
void tri(int T[], int N) {
    N /= 2;
    int* T1=new int[N];
    int* T2=new int[N];
    for(int i=0; i<N; i++) { T1[i]=T[i]; T2[i]=T[i+N]; }
    tri(T1,N);
    tri(T2,N);
    fusion(T1,N, T2,N, T);
    delete[] T1;
    delete[] T2;
}
```

5. Quelle est sa complexité ? $C_N = 2C_{N/2} + N$ (comme QuickSort), donc $C_N = O(N \log N)$.
6. Commenter cette complexité. Optimal sans cas "pire" à la différence de QuickSort. Inconvénient : besoin de mémoire supplémentaire pour les tableaux temporaires $T1$ et $T2$.

2 Arbre binaire de recherche

1. Rappeler la propriété vérifiée pour tout noeud d'un arbre représentant une file de priorité. Chaque noeud porte une priorité au moins aussi grande que ses enfants éventuels.
2. Quelle est la complexité pour trouver le plus grand entier dans une file de priorité ? Le plus petit entier ? Le plus grand est à la racine $O(1)$, pour le plus petit il faut tout parcourir $O(N)$ (ou du moins toutes les feuilles, il peut y en avoir $N/2$. On ne peut pas avoir les deux en même temps.
3. On appelle arbre binaire (au maximum deux enfants par noeud) de recherche (ABR) un arbre ayant la propriété que chaque noeud ait une valeur au moins aussi grande que tout noeud du sous-arbre gauche (s'il existe) et pas plus grande qu'aucun noeud du sous-arbre droit (s'il existe). Dessiner deux ABR différents contenant les entiers de 1 à 7, dont l'un complet (les feuilles sont toutes à la même profondeur, les autres noeuds ont tous deux enfants). Complet : racine 4, enfants 2 (d'enfants 1 et 3) et 6 (d'enfants 5 et 7). Profondeur maximale : racine 7 d'enfant gauche 6, d'enfant gauche 5, etc.
4. Montrer que si $N = 2^n - 1$ ($n \geq 1$) il existe un ABR complet contenant N entiers. Mettre le médian à la racine. Construire un ABR complet avec les $2^{n-1} - 1$ plus petits et les $2^{n-1} - 1$ plus grands (solution récursive !).
5. Proposer un algorithme pour trouver le plus petit élément d'un ABR et un autre pour trouver le plus grand. Plus petit : partir de la racine et aller vers l'enfant gauche jusqu'à tomber sur une feuille, elle porte le min. Plus grand : idem vers la droite.
6. Quelle est leur complexité en fonction de N si l'ABR est complet ? On parcourt en profondeur jusqu'à atteindre une feuille, donc $O(n) = O(\log N)$.
7. Proposer un algorithme pour trouver si un entier est présent dans l'ABR. Soit i la valeur à chercher. Si i est égal à la racine, terminé. Si i plus petit (resp. plus grand) : si pas d'enfant gauche (resp. droit), échec, sinon recherche dans le sous-ABR gauche (resp. droit).
8. Quelle est sa complexité pour un ABR complet ? La complexité la pire si l'ABR n'est pas complet ? En cas d'échec, on doit aller jusqu'au fond de l'ABR, donc $O(n)$. Pour un ABR complet c'est $O(\log N)$, sinon on peut avoir au pire $n = N$ et donc $O(N)$.
9. Proposer un algorithme pour trouver le prédecesseur ou le successeur immédiat de la valeur d'un noeud. Trouver le plus grand du sous-ABR gauche ou le plus petit du sous-ABR droit. Si le sous-ABR n'existe pas et qu'on est à droite (resp. gauche) du parent, c'est ce dernier le prédecesseur (resp. successeur).
10. Proposer un algorithme pour la fonction push(i) qui insère la valeur i dans un ABR. Chercher i dans l'ABR. S'il n'est pas présent, c'est qu'on n'arrive à un noeud de valeur j sans pouvoir descendre. Si $i < j$, on ajoute donc cet enfant gauche non présent, si $i > j$ même chose à droite.
11. Complexité de l'opération push ? La même que pour la recherche, $O(\log N)$ pour un ABR complet, $O(N)$ au pire.
12. Pour pop(i), qui supprime la valeur i (supposée présente une seule fois) d'un ABR, on commence par trouver le noeud. Comment faire si ce noeud a 0 ou 1 enfant ? Si 0 enfant pour le noeud a, on supprime cette feuille bêtement. Si un enfant b, on met tout le sous-ABR de racine b comme enfant du parent de a (ou nouvelle racine si a l'était).
13. Si le noeud a deux enfants, comment faire le pop ? (utiliser le prédecesseur ou successeur immédiat) On arrive au noeud a avec deux enfants. Prendre le prédécesseur b de a (dans le sous-arbre gauche), b n'a pas d'enfant droit. Remonter son éventuel enfant gauche comme enfant droit du parent de b (ancienne place de b) et mettre la valeur de b dans a.
14. Finalement, quelle est la complexité du pop ? Pareil, c'est $O(1)$ après la recherche, donc $O(\log N)$ dans le cas favorable, $O(N)$ au pire.