IFT339 - Exercices sur les arbres

Dans les questions qui suivent, on suppose qu'un arbre T est représenté par un objet Noeud*, qui est la racine de l'arbre. Si T est binaire, l'objet noeud a deux enfants gauche, droit. Si T n'est pas nécessairement binaire, les enfants d'un noeud sont dans un vector<Noeud*>. On suppose aussi que chaque noeud connaît son parent.

Exercice 1.
Supposons que la classe Noeud possède une variable membre hauteur, représentant la hauteur du noeud.

a) En supposant qu'un arbre binaire T est donné, donnez le code d'un fonction qui affecte la variable hauteur de chaque noeud correctement. Quelle est la complexité de votre algorithme?

Solution

```
size_t get_haut(Noeud* v){
    if (!v)
        return 0;
    return v->hauteur;
}

void assign_hauteur(Noeud* v){
    if (!v)
        return;

    assign_hauteur(v->gauche);
    assign_hauteur(v->droit);

    v->hauteur = 1 + max(get_haut(v->gauche), get_haut(v->droit
   ));
}

Appel initial: assign_hauteur(racine_de_T);
```

La complexité est proportionnelle au nombre d'appels récursifs, fois le temps passé par appel. Ce code fait un parcours postordre et visite chaque noeud une fois, donc le nombre d'appels est $O(n)$. Le temps passé par appel est clairement $O(1)$, et la complexité est donc $O(n)$.

Fin de la solution

b) Si T n'est pas nécessairement binaire, que devez-vous modifier pour implémenter la même fonction? Quelle est la complexité de cet algorithme modifié?

Solution

```
size_t get_haut(Noeud* v){
    if (!v)
        return 0;
    return v->hauteur;
}

void assign_hauteur(Noeud* v){
    size_t max_h = 0;

    for (Noeud* enf : v->enfants){
        assign_hauteur(enf);
        max_h = max(max_h, get_haut(enf));
    }

    v->hauteur = 1 + max_h;
}
```

Ici, l'analyse naïve "Nb d'appels $\times$ temps par appel" donnerait $O(n^2)$, ce qui est une surestimation. En fait, le temps d'un appel avec noeud `v` est proportionnel à `v→enfants.size()`. Si on fait la somme des temps passés pour chaque appel, on se retrouve avec la sommation

$$\sum_{v \in T} v \to enfants.size()$$

Cette somme est $O(n)$, parce que chaque noeud est additionné une seule fois dans cette expression (un noeud est ajouté seulement quand v est son parent). La complexité reste donc $O(n)$.

Fin de la solution

Exercice 2.

a) Donnez le code d'une fonction qui reçoit un arbre T et qui détermine si T est un ABR ou non.

Solution

Encore un parcours post-ordre! Chaque noeud a besoin du min ou max de ses sous-arbres enfants. Il faudrait aussi un mécanisme pour donner l'info du min/max aux appels parents. Aussi, si un enfant a un problème, il faudrait avertir le parent. J'ai donc une struct pour stocker toutes ces informations et les retourner aux parents.

```
struct ABRInfo{
    TYPE min;
    TYPE max;
    bool est_ABR;

    ABRInfo(TYPE& min, TYPE& max, bool est_ABR){
        this->min = min;
        this->max = max;
        this->est_ABR = est_ABR;
    }
};

ABRInfo est_ABR(Noeud* v){
    if (v->est_feuille())
        return ABRInfo(v->val, v->val, true);
    TYPE curmin = v->val;
    TYPE curmax = v->val;
    if (v->gauche){
        ABRInfo ginfo = est_ABR(v->gauche);
        if (ginfo.max > v->val || !ginfo.est_ABR)
```

```
            return (v->val, v->val, false);
        curmin = ginfo.min;
    }
    if (v->droit){
        ABRInfo dinfo = est_ABR(v->droit);
        if (dinfo.min < v->val || !dinfo.est_ABR)
            return (v->val, v->val, false);
        curmax = dinfo.max;
    }
    //tout est ok
    return ABRInfo(curmin, curmax, true);
}
//Appel initial:
ABRInfo info = est_ABR(racine);
cout<<(info.est_ABR ? "OK" : "Pas OK")<<endl;
```

Fin de la solution

b) Donnez le code d'une fonction qui détermine si T est un arbre AVL ou non.

Solution

Notez qu'il faut vérifier que T est un ABR (fait plus bas), en plus de vérifier les conditions sur la hauteur, ce que le code suivant fait.

```
struct AVLInfo{
    size_t hauteur;
    bool est_AVL;

    AVLInfo(size_t h, bool b){
        hauteur = h;
        est_AVL = b;
    }
};
```

```
AVLInfo est_AVL(Noeud* v){
    if (v->est_feuille())
        return AVLInfo(1, true);
    size_t hg = 0, hd = 0;
    if (v->gauche){
        AVLInfo ginfo = est_AVL(v->gauche);
        if (!ginfo.est_AVL)
            return AVLInfo(0, false);
        hg = ginfo.hauteur;
    }
    if (v->droit){
        AVLInfo dinfo = est_AVL(v->droit);
        if (!dinfo.est_AVL)
            return AVLInfo(0, false);
        hd = dinfo.hauteur;
    }
    if (abs(hg - hd) > 1)
        return AVLInfo(0, false);
    return AVLInfo( 1 + max(hg, hd), true);
}
//Appel initial:
ABRInfo info = est_ABR(racine);
if (info.est_ABR){
    AVLInfo avlinfo = est_AVL(racine);
    if (avlinfo.est_AVL)
        cout<<"Tout est OK!"<<endl;
}
```

Fin de la solution

Exercice 3.
Donnez une version itérative, donc non-récursive, de la fonction contient d'un

ABR.

Solution

```
bool contient(const TYPE& val){
    Noeud* v = racine;

    while (v){
        if (val == v->val)
            return true;
        else if (val < v->val)
            v = v->gauche;
        else
            v = v->droit;
    }

    return false;   //on n'a rien trouvé
}
```

Fin de la solution

Exercice 4.
Montrez qu'avec un arbre AVL, on peut trier une liste de n éléments en temps $O(n \log n)$.

Solution

L'observation principale est qu'un parcours in-ordre donnera les éléments d'un ABR dans un ordre trié. Rappelons que le parcours in-ordre visite l'enfant gauche, puis le noeud, puis l'enfant droit. Ce faisait, pour un noeud v, on va visiter que des valeurs $< \mathsf{v} \to \mathsf{val}$, puis v, puis ensuite que des valeurs $> \mathsf{v} \to \mathsf{val}$. Puisque c'est vrai pour tout noeud, le parcours sera donc trié.

On va donc:
- insérer nos n valeurs dans un arbreAVL (temps $O(n \log n)$)
- énumérer les valeurs de l'arbre en in-ordre (temps $O(n)$)
ce qui donne un temps $O(n \log n + n) = O(n \log n)$.

```
vector<int> triAVL(const vector<int>& elts)
{
    ArbreAVL avl;
    for (int i : elts)
    {
        avl.inserer(i);
    }

    return avl.get_inordre();
}

//version publique
vector<int> ArbreAVL::get_inordre()
{
    vector<int> elts;
    get_inordre(racine, elts);
}

//version récursive: recoit en réf le vecteur à remplir
void ArbreAVL::get_inordre(Noeud* v, vector<int>& elts)
{
    if (v->gauche)
        get_inordre(v->gauche, elts);
    elts.push_back(v->val);
    if (v->droit)
        get_inordre(v->droit, elts);
}
```

Fin de la solution

Exercice 5.
Vous avez une liste d'entiers dans un vecteur vec, et vous voulez les éléments uniques de vec. Vous devez donc produire une liste dans laquelle chaque nombre présent dans vec se trouve une seule fois dans la liste (donc sans doublons). Par exemple, si vec = [1,5,3,2,1,3,5,4] alors on voudrait en sortie [1,5,3,2,4] (l'ordre des éléments de sortie n'est toutefois pas important).

Avec ce qu'on a vu en classe, ceci peut se faire en temps $O(n \log n)$, avec n le nombre d'éléments de vec. Montrez comment.

Solution

Sachant que la classe set implémente un arbre balancé, on va l'utiliser pour détecter les doublons.

```
vector<int> get_unique(const vector<int> &vec){
    set<int> s;
    for (size_t i = 0; i < vec.size(); ++i){
        if (s.find(vec[i]) == s.end()){
            //note: on pourrait enlever le if ci-haut, car
            //insert vérifie déjà les doublons.
            //Je le laisse pour raisons pédagogiques.
            s.insert(vec[i]);
        }
    }

    vector<int> ret;
    //itération sur le set
    for (auto it = s.begin(); it != s.end(); ++it){
        ret.push_back(*it);
    }
    return ret;
}
```

On appelle n fois $find$ et $insert$ dans le pire cas, ce qui demande un temps $O(n \log n)$ pour la première boucle. La deuxième boucle ne fait que n fois push et prend un temps $O(n)$. Le total est donc $O(n \log n + n) = O(n \log n)$.

Fin de la solution

Exercice 6.
Une métrique parfois utilisée pour mesurer le débalancement d'un arbre possiblement non-binaire est le *poids en profondeur*.

Soit T un arbre et v un de ses noeuds. La profondeur de v, dénotée $prof(v)$, est le nombre d'arêtes à parcourir pour passer de v à la racine (donc la racine a une profondeur de 0, ses enfants on une profondeur de 1, ses petits-enfants de 2, etc.). On définit le poids de profondeur de T, dénoté $PP(T)$, comme la somme des profondeurs de tous ses noeuds :

$$PP(T) = \sum_{v \in T} prof(v)$$

Où $v \in T$ signifie « v est un nœud de T ».

a) Donnez le code d'un algorithme qui calcule $PP(T)$.

Solution

```
int getPP(noeud* racine)
{
    return getPP(racine, 0);
}

int getPP(noeud* v, int prof)
{
    if (!v)
        return 0;
    int ret = 0;
    for (noeud* w : v->enfants)
        ret += getPP(w, prof + 1);
    return ret + prof;
}
```

Fin de la solution

b) Si T est un arbre avec n noeuds, quel est le minimum de $PP(T)$? Quel est le maximum? Donnez des exemples d'arbres qui atteignent le minimum et le maximum.

Solution

Le minimum est atteint par un arbre "étoile", c'est à dire un arbre avec une racine et $n-1$ enfants, dont le poids en profondeur est $n-1$, et le maximum par une chaîne, dont le poids est $1+2+\ldots+n-1 = n(n-1)/2$.

Fin de la solution

Exercice 7.
Dans un *parcours en profondeur* d'un arbre T, on commence par visiter la racine, puis on répète la règle suivante jusqu'à ce que tous les nœuds aient été visités : *« parmi tous les nœuds non-visités dont le parent est visité, on visite le nœud le plus profond »*

Montrez comment implémenter un parcours en profondeur.

Solution

Observez que le parcours en profondeur ne spécifie pas quel choix faire s'il y a plusieurs options. Donc, un parcours post-ordre fera l'affaire, car dans un parcours post-ordre, on va toujours à un niveau de profondeur plus bas lorsque c'est possible. Vous pouvez donc regarder les solutions aux exercices précédents pour revoir comment faire un parcours postordre.

Fin de la solution

Exercice 8.
Dans un *parcours en largeur*, on fait le contraire de l'exercice précédent. C'est-à-dire, on commence par visiter la racine, puis on répète le règle suivante : *« parmi tous les nœuds non-visités qui ont un parent visité, on visite le nœud le moins profond ».*

En fait, ce parcours commence par la racine, puis visite tous ses enfants, puis lorsque c'est fait, visite tous ses petits-enfants, puis ses petits-petits-enfants, etc.

Montrez comment implémenter le parcours en largeur d'un arbre.

Solution

Pour le parcours en largeur, il faut travailler un peu plus. Une fonction récursive ne fonctionnera pas, car lorsqu'on entre dans une récursion on a oublié quel était le noeud de profondeur minimum.

La manière classique est de lister les sommets dans une file d'attente en ordre de profondeur. C'est-à-dire, en mettant dans une file les noeuds rencontrés et en ajoutant leurs enfants dans l'ordre, on s'assure de toujours visiter les noeuds dans l'ordre de profondeur. Ceci est parce que l'itération 1 ajoutera en file les noeuds de profondeur 2, puis l'itération 2 ajoutera tous les noeuds de profondeur 3, etc.

```
void parcours_largeur(Noeud* racine)
{
    queue<Noeud*> file;
    file.push(racine);

    while (!file.empty())
    {
        Noeud* v = file.front();
        file.pop();
        visiter(v);
        for (Noeud* e : v->enfants)
            file.push(e);
    }
}
```

Fin de la solution

Exercice 9.
Soit T un ABR. Implémentez une fonction rebalancer, qui créé un nouvel ABR dont la hauteur est $O(\log n)$. J'y arrive en temps $O(n \log n)$.

Suggestion. Triez les éléments de T, puis recréez un nouvel ABR en y insérant les valeurs dans un ordre optimal. Pour voir comment faire ça, pratiquez-vous à trouver un ordre d'insertion de 1-2-3-4-5-6-7 qui minimise la hauteur.

Solution

On commence par obtenir les éléments de l'ABR en ordre trié dans un vecteur vec. Ensuite, on créé un nouvel ABR en insérant dans l'ordre idéal. Il suffit de remarquer que si on met l'élément médian à la racine d'un ABR, la moitié des éléments seront à sa gauche et l'autre moitié à sa droite. On insère donc la médiane en premier, puis on fait les insertions à gauche et à droite dans un ordre optimal, de façon récursive.

```
ABR rebalancer(ABR &mon_abr){
    vector<TYPE> elts = avl.get_inordre(); //voir exo de tri
    ABR nouvel_abr;
    remplir_abr(nouvel_abr, elts, 1, elts.size());
}

void remplir_abr(ABR& abr, vector<TYPE> &vec, int min, int max)
{
    if (min == max){
        abr.inserer(vec[min]);
    }
    else if (min < max){
        int median = (max - min)/2 + min;
        abr.inserer(vec[median]);
        remplir_abr(abr, vec, 1, median - 1);
        remplir_abr(abr, vec, median + 1, vec.size());
    }
}
```

Pour la complexité, il faut un temps $O(n)$ pour obtenir le vecteur trié. Ensuite pour la fonction remplir_abr, on remarque que chaque appel insère exactement un élément dans l'ABR. Le nombre de fois que la fonction sera exécutée est donc $O(n)$. Chaque appel fait une insertion en temps $O(\log n)$, et la complexité totale de remplir_abr est $O(n \log n)$.

Fin de la solution

Exercice 10.
Soit un nœud x d'un arbre T dans lequel chaque nœud a une valeur différente. On dénote par $c(x)$ l'ensemble des valeurs qui apparaissent dans les nœuds du sous-arbre enraciné en x. Supposons que l'on vous donne un ensemble de valeurs C (vous pouvez supposer que C est un vector, un array, une list, à vous de choisir...) et que l'on vous demande *« est-ce qu'il existe un nœud x dans T tel que $c(x) = C$ »*? En d'autres termes, est-ce qu'il y a un sous-arbre

qui contient exactement les valeurs de C?

a) Implémentez une fonction qui retourne un tel nœud x s'il existe, ou faux sinon. Si n est la taille de T et m la taille de C, quelle est la complexité?

Ce n'est pas trop difficile d'y arriver en temps $O(n^2 \log m)$. Pouvez-vous atteindre $O(n \log m)$?

Solution

Pour atteindre $O(n^2 \log n)$, faites un parcours post-ordre où chaque noeud calcule $c(x)$ récursivement, en prenant l'union des éléments des enfants et en ajoutant sa propre valeur. Ensuite, pour chaque élément de x, on regarde si cet élément est dans C. Chacun des n noeuds fait potentiellement n requêtes en temps $O(\log m)$, ce qui explique $O(n^2 \log m)$. On peut être un peu plus intelligent.

L'idée est de faire un parcours post-ordre. Pendant ce parcours, chaque noeud dit à son parent combien de noeuds dans C il a trouvé, et combien de noeuds pas dans C il a trouvé. Le noeud qu'on cherche doit avoir $C.size()$ noeuds dans C, et 0 pas dans C. Puisqu'on nous permet de choisir le type de C, je choisis de représenter C avec un set (si C était un vecteur, alors on pourrait prendre un temps $O(n \log n)$ pour le convertir en set).

```
//version publique de l'appelant
bool contientC(Noeud* racine, set<int>& C)
{
    Noeud* res = nullptr;
    contientC(racine, C, res);
    return (res != nullptr);
}

//version récursive, qui retourne (dansC, pasDansC)
//leNoeud est une valeur de retour
pair<int, int> contientC(Noeud* v, set<int>& C, Noeud* &
leNoeud)
{
    int nbIn = 0;
    int nbOut = 0;
    if (C.find(v->val) != C.end())
        nbIn++;
```

```
        else
            nbIn--;

        for (noeud* e : v->enfants)
        {
            pair<int, int> resv = recurs(v, C, leNoeud);
            nbIn += resv.first;
            nbOut += resv.second;
        }

        if (nbIn == C.size() && nbOut == 0)
            leNoeud = v;
        return make_pair(nbIn, nbOut);
    }
```

Fin de la solution

b) Implémentez la même fonction, mais dans le cas où T est un ABR. Mon implémentation nécessite un temps $O(h \log m + m \log m)$, où h est la hauteur de T.

Solution

C'est un peu long de faire le code détaillé, alors je vous le donne en pseudo-code ultra-haut-niveau.

Sur entrée T et C, il faut:

1. trouver v de T qui contient $C[0]$ dans l'arbre. Si v n'existe pas, return false. [Temps $O(h)$]

2. en remontant de v à la racine, trouver le premier noeud w tel que la valeur de $w.parent$ n'est pas dans C (ou $w = racine$ si ça n'existe pas). On fait au plus h requêtes en temps $O(\log m)$ [Temps $O(h \log m)$]

3. S'il existe un noeud x tel que $c(x) = C$, il faut que ce soit w. Donc, on vérifie que le sous-arbre en w a exactement les valeurs de C. Pour ce faire:

   (a) On compte le nombre de noeuds sous w. Si, pendant le compte, on arrive à $C.size()+1$ noeuds, on retourne false car il est clair qu'un noeud sous w a une valeur pas dans C. Si le nombre de noeuds est plus petit que $C.size()$, on retourne false. [Temps $O(m)$]

   (b) Sinon, il y a $C.size()$ noeuds sous w. Pour chacun, on vérifie si sa valeur est dans C. Chaque vérification se fait en temps $O(\log m)$. Donc le temps passé ici est $O(m \log m)$. [Temps $O(m \log m)$]

Au total, la complexité est $O(h + h \log m + m + m \log m)$, ce qui est égal à $O((h + m) \log m)$.

Fin de la solution