IFT339 - Exercices sur les arbres

Exercice 1.
Considérez le B-tree vu en classe. Donnez le code de la fonction contient(val), qui retourne vrai si val est présent dans l'arbre, et faux sinon. Vous pouvez supposer qu'un noeud est implémenté comme suit.

```
struct Noeud{
    vector<TYPE> vals;
    vector<Noeud*> enfants;
    Noeud* parent;
};
```

Solution

```
bool contient(const TYPE& val){
    return contient(racine, val);
}

bool contient(Noeud* v, const TYPE& val){
    for (size_t i = 0; i < vals.size(); ++i){
        if (val == v->vals[i])
            return true;
        else if (val < v->vals[i]){
            if (v->enfants.empty()) //feuille
                return false;
            else
                return contient(v->enfants[i], val);
        }
    }
    if (v->enfants.empty()) //feuille
        return false;
    return contient(v->enfants.back(), val);
}
```

Fin de la solution

Exercice 2.
Soit S un ensemble, et soit val une valeur (possiblement dans S ou non). Le *prédécesseur* de val dans S est l'élément maximum x dans S tel que x $<$ val. Si un tel prédécesseur n'existe pas, alors le prédécesseur de val est défini come val. Par exemple, si S stocke les entiers $[2, 5, 7, 9, 11, 14]$, le prédécesseur de 4 est 2, le prédecesseur de 13 est 11, et le prédécesseur de 9 est 7. Le prédécesseur de 2 n'existe pas, alors on retournerait 2.

On a une implémentation d'un ensemble avec un B-Tree. Donnez le code ou pseudo-code d'une procédure qui, étant donné une valeur val, retourne son prédécesseur dans le B-Tree. Votre implémentation devrait prendre un temps $O(\log n)$.

Solution

L'idée est de faire un parcours récursif. On trouve le premier i tel que v->vals[i] est plus grand ou égal à val. Dans ce cas, le prédecesseur est soit v->vals[i - 1], ou bien un descendant dans l'enfant i. On considère les deux cas, avec un cas spécial quand $i = 0$. Pour simplifier, je suppose que si v est une feuille, alors v a un veteur d'enfants à nullptr.

```
const TYPE& get_pred(const TYPE& val){
    return get_pred(racine, val);
}

const TYPE& get_pred(Noeud* v, const TYPE& val){
    if (!v) return val;
    if (val <= v->vals[0])
        return get_pred(v->enfants[0]);


    for (size_t i = 1; i < v->vals.size(); i++){
        if (v->vals[i] > val){
            //le pred est soit v->vals[i-1], ou bien
            //une valeur sous l'enfant i
            const TYPE pre1 = v->vals[i-1];
            const TYPE pre2 = get_pred(v->enfants[i], val);
            if (pre2 != val)   //sachez pourquoi ceci est ok
                return pre2;
            else
                return pre1;
        }
    }
    //si on se rend ici, val est >= à tout
    //todo: unifier les cas
    const TYPE pre1 = v->vals.back();
    const TYPE pre2 = get_pred(v->enfants.back(), val);
    if (pre2 != val)
        return pre2;
    else
        return pre1;
}
```

Fin de la solution

Exercice 3.
Soit la séquence d'insertions 10, 20, 15, 25, 30, 16, 18, 19 dans un arbre AVL. Dessinez l'arbre AVL obtenu après chacune des insertions (en incluant les arbres intermédiaires si des insertions nécessitent plusieurs rotations).

Avec l'arbre AVL obtenu après les insertions ci-haut, dessinez étape par étape les arbres obtenus lorsque l'on supprime 30.

Note: je ne vais pas dessiner la solution. Vous pouvez vous vérifier à l'aide d'outil en ligne, voir
`https://www.cs.usfca.edu/~galles/visualization/AVLtree.html`.

Exercice 4.
Le monceau ne supporte pas la recherche d'une clé en temps $O(\log n)$. Concevez une structure de données qui implémente une file de priorité (insertion, get_min, supprimer_min), mais à laquelle on ajoute la fonction recherche(cle).

Si le monceau contient un noeud dont la clé est cle, alors recherche retourne un Noeud* qui possède cette clé. Sinon, recherche retourne nullptr. Ceci est utile pour modifier la valeur associée à une clé.

La fonction devrait prendre un temps $O(\log n)$, tout en maintenant les complexité des autres opérations. Vous pouvez décrire votre solution en phrases pour énoncer les idées importantes de votre approche.

Suggestion. N'essayez pas de faire la recherche directement sur le monceau, ça ne marche pas. Ma solution utilise un map indexé par les noeuds du monceau.

Solution

L'idée haut-niveau est d'ajouter à la classe Monceau un dictionnaire dict, qui associe à chaque clé l'ensemble des noeuds avec cette clé.

Si le dictionnaire est bien maintenu, la recherche(cle) est facile: on regarde si dict[cle] est non-vide et on retourne n'importe lequel de ses noeuds.

À l'insertion, il faut s'assurer de bien maintenir dict. Quand on créé un noeud w, on insère l'association w et w->cle dans dict. Par contre, quand on percole w il faut que cette info reste valide. On ne peut faire faire un simple swap entre la clé de w et son parent, car il faudra mettre à jour le dictionnaire et l'insertion finira par coûter $O(n(\log n)^2)$ (pensez-y!). Plutôt,

pour percoler w on déplace le noeud w et son parent, tout en préservant les mêmes clés dans les noeuds. Le code ci-bas donne le détail.

Je n'ai pas mis la suppression, mais elle peut s'implémenter de façon similaire.

```
template <typename T>
class Monceau{
    struct Noeud{
        T cle;
        Noeud* gauche, *droit, *parent;
    }
    Noeud* racine;

    map<T, set<Noeud*>> dict;

    void inserer(const T& cle){
        //v = prochain noeud sous lequel insérer
        //w = nouveau noeud créé, initialement enfant de v
        w->cle = cle;

        dict[cle].insert(w);

        while (w->parent && w->cle < w->parent->cle){
            //échanger la position de w et son parent
            //en conservant les clés à la bonne place
            Noeud* gw = w->gauche, *dw = w->droit;
            Noeud* z = w->parent;
            w->parent = z->parent;
            z->parent = w;
            if (w == z->gauche){
                w->droit = z->droit;
                w->gauche = z;
            }
            else {
                w->gauche = z->gauche;
                w->droit = z;
            }
            z->gauche = gw;
```

```
            z->droit = dw;
        }
    }


    Noeud* rechercher(const T &cle){
        if (dict.find(cle) != dict.end() && !dict[cle].empty())
            return *dict[cle].begin();
        return nullptr;
    }
};
```

Fin de la solution

Exercice 5.
Soit T un monceau. On veut permettre la modification d'une clé dans le monceau. Dites comment implémenter la fonction

`modifier_cle(TYPECLE& c_avant, TYPECLE& c_apres)`

qui trouve un noeud avec la clé `c_avant`, et modifie la clé de ce noeud pour `c_apres`. S'il y a plusieurs noeuds avec la clé `c_avant`, prenez n'importe lequel. Le monceau doit conserver ses conditions après la modification.

Vous pouvez supposer qu'un noeud avec la clé recherchée peut être obtenu en temps $O(\log n)$, en utilisant l'exercice précédent.

Solution

Il faut juste modifier la valeur du noeud en question et le faire percoler adéquatement. Il se peut qu'on doive percoler vers le bas, ou vers le haut. Notez que même si la solution ci-bas contient deux boucles, une seule d'entre elles sera exécutée.

```
void modifier_cle(TYPECLE& c_avant, TYPECLE& c_apres){
    Noeud* v = rechercher(c_avant); //exo précédent
```

```
    v->cle = c_apres;
    //cas 1: c_apres < c_avant => faire monter v
    while (v->parent && v->cle < v->parent->cle){
        std::swap(v->cle, v->parent->cle);
        v = v->parent;
    }

    //cas 2: c_apres > c_avant => faire descendre v
    while (v->gauche && v->val > v->gauche->val ||
            v->droit && v->val > v->droit->val){
        Noeud* w = v->gauche;
        if (!w || v->droit && v->droit->val < w->val)
            w = v->droit;
        std::swap(v->cle, w->cle);
        v = w;
    }

}
```

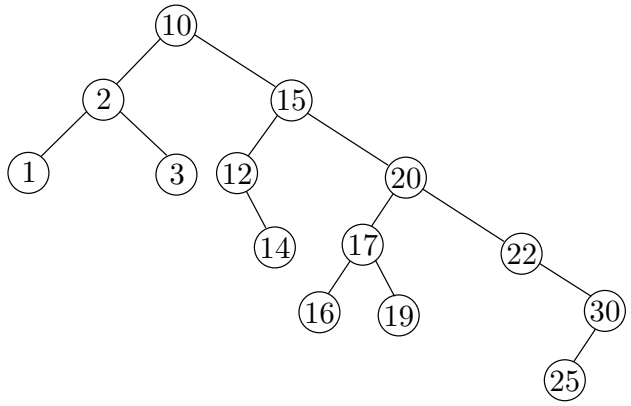
Fin de la solution

Exercice 6.
Soit T un ABR. La *profondeur droite* de T est le nombre de noeuds entre la racine et le noeud contenant la valeur maximum de T (qui est le noeud le plus à droite). Par exemple, dans la figure ci-bas, l'arbre a une profondeur droite de 5.

On nous donne un ABR arbitraire, et on doit lui appliquer des rotations pour le transformer en un ABR de profondeur droite n, donc en une chaîne avec que des fils droits[1].

Notre stratégie est la suivante: on applique des rotations à T de façon à ce que chaque rotation augmente sa profondeur droite.

1. Dans l'ABR ci-dessous, donnez une arête sur laquelle on peut appliquer une rotation pour augmenter la profondeur droite de l'arbre.



2. Argumentez que sur tout ABR de profondeur droite plus petite que n, il existe toujours au moins une arête sur laquelle une rotation augmente la profondeur droite.

3. Donnez le code ou pseudo-code d'une fonction qui transforme un ABR en une chaîne de profondeur droite n en utilisant seulement des rotations. Vous pouvez appeler une fonction `rotation(Noeud* v, Noeud * parent)` en boîte noire, qui fait une rotation sur l'arête entre `v` et son parent.

[1] L'intérêt d'une telle opération est que pendant la période de révision, nous avons vu que si T a une profondeur droite de n, on peut balancer T avec seulement des rotations. Donc pour balancer un ABR, on le transforme en chaîne, puis on balance la chaîne. Le tout est faisable en temps $O(n)$.

Solution

Pour 1), rotation sur l'arête entre 25 et 30 fait l'affaire.

Pour 2), si la profondeur droite est moins que n, il doit y avoir un noeud sur le plus long chemin droit qui a un fils gauche (sinon, ce chemin aurait tous les n noeuds). Une rotation sur cette arête fils gauche va nécessairement augmenter la profondeur droite.

Ce n'est pas n'importe quel fils gauche qui augmente la profondeur droite. Il faut que le fils gauche appartienne au chemin de noeuds à droite à partir de la racine.

Pour 3), je suppose que si je fait une rotation entre v et son enfant gauche, alors v est descendu d'un niveau et c'est l'enfant gauche qui se retrouve à sa place.

```
void lineariser(Noeud* racine){
    Noeud* v = racine;
    while (v){
        if (v->gauche){
            Noeud* old_gauche = v->gauche;
            rotation(v->gauche, v);
            v = old_gauche;
        }
        else
            v = v->droit;
    }
}
```

Fin de la solution