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;
};
```

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 $O(\log n)$.

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.

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.

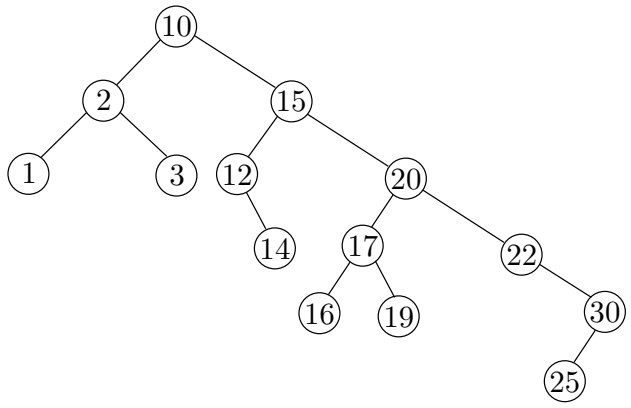
Vous pouvez supposer qu'un noeud avec la clé recherchée peut être obtenu en temps $O(\log n)$, en utilisant l'exercice précédent.

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 $O(n)$.