POLYTECH SORBONNE

Algorithmique générale
MAIN-3, année 2022
Séance TP N°3
Mars 2022
VOYLES Evan

Objectif

☞ Arbres binaires de recherche.

Problème [Arbres binaires de recherce aléatoires]

La documentation générale pour ce travail se trouve ici, la documentation technique se trouve là, et voici le répertoire git.

Partie A : Structure Abr

La structure `bin_tree_t` et toutes les fonctions définies autour cet objet encodent l'interface pour un arbre binaire de recherche.

☞ l'insertion d'une nouvelle clé dans l'arbre.

```
// Instantiation
BinTree *root = newBinTree(10);

// insertion
addKeyBST(root, 1);
addKeyBST(root, 0);
addKeyBST(root, 13);
addKeyBST(root, 5);
addKeyBST(root, 8);
addKeyBST(root, 25);
addKeyBST(root, 3);
```

Cet extrait de code encode l'arbre suivant (obtenu après un appel à la fonction `createDotBST`) :

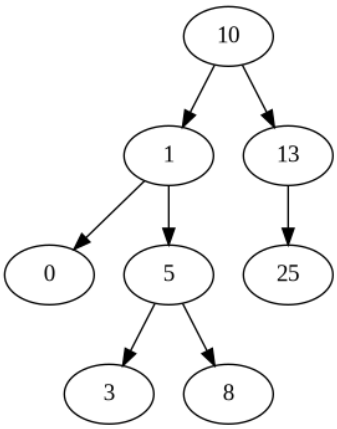


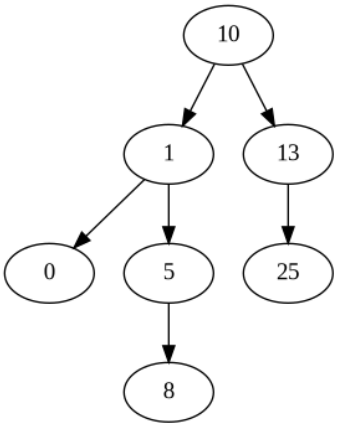
Figure 1 – L'arbre binaire créé au-dessus.

☞ la suppression d'une clé de l'arbre

Pour supprimer des clés de l'arbre, on appelle la fonction `removeKeyBST`. L'implémentation de cette fonction traite trois cas distincts qu'on doit considérer pour enlever une clé de l'arbre. Le premier cas est si le noeud que l'on souhaite enlever n'a pas d'enfans. Pour ce cas-là, il est facile de supprimer l'élément de l'arbre parce que nous pouvons tout simplement mettre à jour le pointeur de son ancêtre pour pointer vers `NULL`.

```
removeKeyBST(root, 3);
```

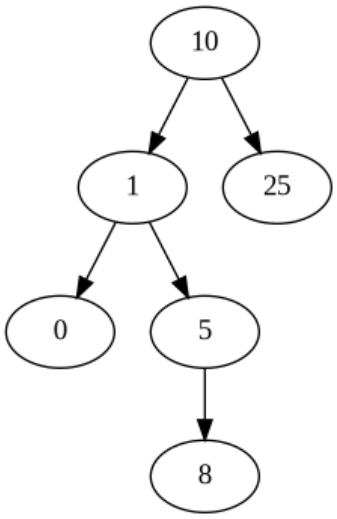
ce qui donne :



Le deuxième cas qu'on considère c'est quand le noeud que l'on voudrait enlever a un seul enfant. Pour le résoudre, on fait "condenser" la chaine qui consiste aux trois noeuds : le parent, celui qu'on va supprimer, et le seul enfant du noeud à supprimer. Pour le visualiser, on va supprimer la clé 13 en condensant $\{10, 13, 25\} \mapsto \{10, 25\}$.

```
removeKeyBST(root, 13);
```

pour avoir

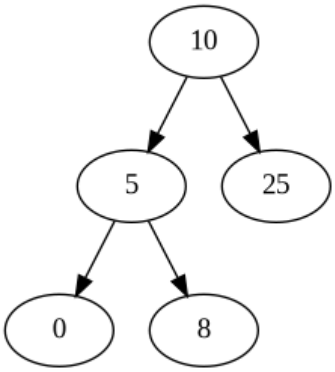


.

Un troisième cas s'apparait quand on veut supprimer un élément qui a deux enfants. Dans ce cas-là on devrait tout d'abord trouver la clé qui va remplacer celle à supprimer. Pour le trouver, on doit parcourir l'arbre à partir de clé ciblée pour trouver le prochain élément en termes de magnitude.

```
removeKeyBST(root, 1);
```

et l'arbre binaire qui en résulte :



.

☞ la suppression de l'arbre.

Pour supprimer un arbre, on utilise la fonction `releaseBST` ce qui recois en argument un `BinTree **` et qui free toute la mémoire qui est alloué par rapport à l'arbre binaire qui est representé par sa racine. On passe un `BinTree **` et pas un `BinTree *` parce qu'à la fin de libérer la mémoire on reinitialiser le pointeur à `NULL`.

```
releaseBST(&root); // root -> NULL
```

☞ *facteur de déséquilibre*

Pour calculer la facteur de déséquilibre, on implémente tout d'abord une fonction qui calcule récursivement la hauteur d'un arbre binaire à partir d'une racine. Ensuite, on calcule la hauteur du sousarbe à gauche et le sousarbre à droite pour renvoyer la facteur de déséquilibre qui est la positive différence entre ces deux hauteurs.

☞ l'export de votre arbre au format .dot

Pour ce faire on appelle la fonction `createDotBST` qui prend en argument la racine d'un arbre binaire à visualiser et deuxièmement le nombre d'un fichier pour sauvegarder le contenu. C'est cette fonction que l'on a utilisé pour créer les figures dans ce rapport.

Partie B : Hauteur moyenne et facteur de déséquilibre moyen

Le code pour cette partie se trouve dans la fonction `partie_b` qui est définie dans le fichier `main.c` du projet.

Grosso modo, on génère 50 tableaux de taille **N** et on calcule le moyen hauteur et facteurs de déséquilibre. On à choisi les valeurs entre 0 et N - 1, mélangé par la procédure Fischer-Yates pour avoir des valeurs unique et aléatoire. Voici un exemple :

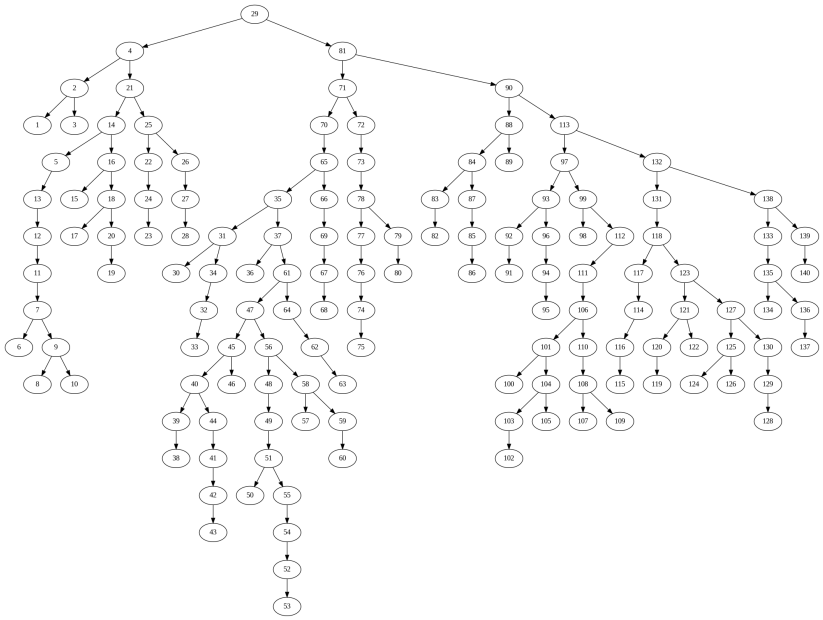


Figure 2 – Un arbre binaire aléatoire de hauteur 17, facteur de déséquilibre 6. La sous-arbre à gauche a une hauteur de 10 et celui à droite a une hauteur de 16.

.

En fixant `tailleMax` à 10E5, l'on obtient les résultats suivants :

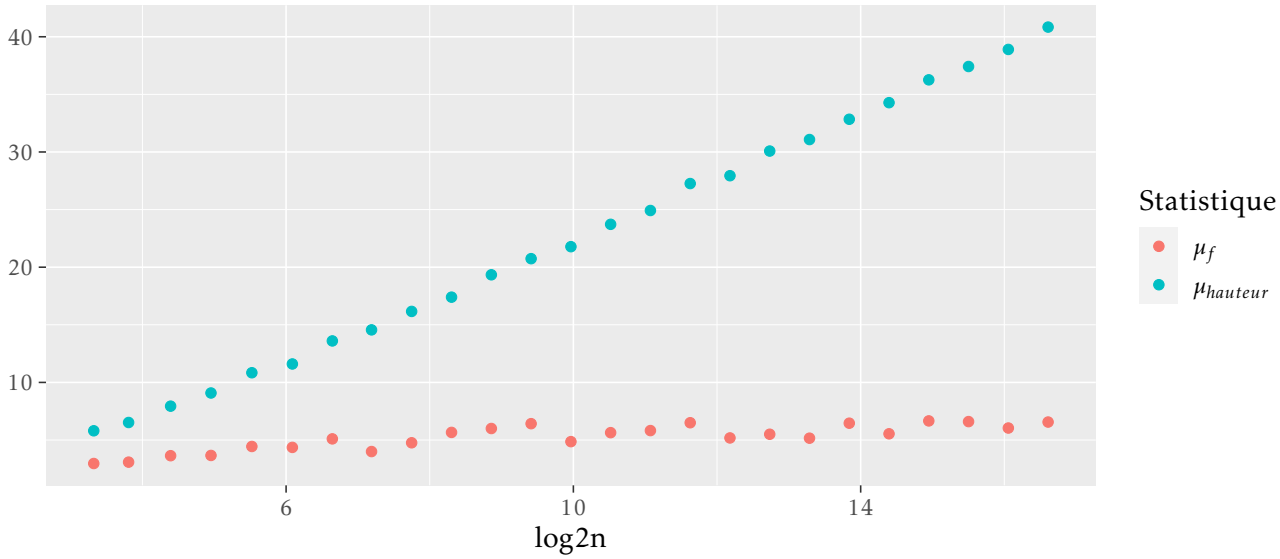


FIGURE 3 – Les variations du moyen facteur de déséquilibre et moyen hauteur par rapport à $\log_2 n$. Il est évident qu'il y a une corresponance linéaire entre nos statistiques observés et on peut constater que μ_f, $\mu_{hauteur} \in \Theta(\log_2 n)$.

Remarques :

Le code source de ce document se trouve sur le github dans le répertoire `tex/`. Le binaire `bst` compilé pour x86-64 est inclu dans le zip rendu sur moodle. Sinon, vous pouvez téléchargé le projet et suivre les instructions du `README.md` pour construire le projet tp3.