Arbres binaires de recherche avec rang.

Contrôle terminal de TP – durée 1h30

NOTES : éléments de notation

— la bonne utilisation du pouvoir expressif du langage C sera prise en compte.
— la présentation et la lisibilité du code écrit seront prises en compte.

1 Contexte.

Les arbres binaires de recherche, et leurs variantes, définissent un type abstrait de données très efficace pour la réalisation des opérations de dictionnaire, ADD, REMOVE et SEARCH qui peuvent toutes être réalisées par des opérateurs de complexité dans le pire cas en $O(log(n))$ pour un arbre équilibré contenant n clés. Le problème de l'équilibrage de l'arbre ne sera pas abordé dans ce sujet.

L'invariant de structure défini sur le TAD arbres binaires de recherche en fait une structure de données implicitement ordonnée et il est alors possible de la parcourir de façon à traiter les éléments dans l'ordre croissant (ou décroissant) de leur clé.

Lorsqu'une application requière de pouvoir accéder aux clés de l'arbre en fonction de leur position dans la suite de valeur ordonnée par la relation d'ordre ayant servi à construire l'arbre de recherche, les algorithmes de parcours infixe, prefixe ou postfixe ne sont pas adaptés et l'utilisation d'un itérateur sur l'arbre de recherche ne permet de répondre au problème qu'avec une complexité en temps de $O(nlog(n)))$.

Nous nous intéressons, dans ce contrôle, à l'augmentation du TAD d'arbre de recherche de façon à pouvoir accéder à une clé en fonction de sa position dans la liste de valeur ordonnées en un temps de l'ordre $O(log(n))$. Pour cela, nous allons ajouter à l'invariant d'implantation du TAD la notion de rangs d'un nœud selon la définition suivante :

Invariant de rang dans un arbre binaire de recherche

Dans un arbre binaire de recherche augmenté du rang (voir figure 2), pour tout nœud n, on appelle $lrank(n)$ le nombre de clés présentes dans l'arbre enraciné en n qui sont strictement inférieures à la clé du nœud n. De façon symétrique, on appelle $grank(n)$ le nombre de clés présentes dans l'arbre enraciné en n qui sont strictement supérieures à la clé du nœud n.

À partir du code fourni dans l'archive `controleTP.tar.gz` associée à ce sujet et construite sur le même modèle que les archives de TP, l'objectif de ce contrôle est de programmer l'opérateur d'ajout d'une clé dans un arbre augmenté par le rang et l'exploitation de cette information pour effectuer une recherche d'une clé par son rang.

1.1 Description de l'archive fournie.

L'archive fournie contient :

— Un sous répertoire `Code` contenant :
  — un fichier Makefile permettant de compiler l'application et de produire les résultats attendus,
  — un fichier `main.c` proposant un programme de test de l'implantation,
  — un fichier `tree.h` définissant l'interface publique d'un arbre binaire de recherche, interface identique à celle des arbres binaires de recherche vus en TP et étendue par l'ajout d'opérateurs de recherche par le rang,

— un fichier `tree.c` définissant l'implantation et l'interface privée du TAD et à compléter pour ce contrôle. Dans cette implantation, la repésentation interne des liens vers les sous arbre et le parent sont représentés par un tableau de pointeurs. L'implantation des opérateurs tiens alors compte de cette représentation interne tout en restant très proche de celle réalisée en TP.

— Un sous répertoire `Test` contenant des fichiers de test.

Le code fourni ne devra en aucun cas être modifié. Seul devra être complété le fichier `tree.c` en dessous du cartouche à compléter par votre nom, prénom et numéro d'étudiant.

```
/*****************************************************************************/
/**                                Control start here                       **/
/*****************************************************************************/
/**
 *  Nom :                       Prenom :                    Num Etud  :
 **/
```

1.2 Description des algorithmes à mettre en place.

1.2.1 Ajout d'un élément de l'arbre.

La fonction `void tree_add(BinarySearchTree **t, int k)`, un constructeur du TAD, a pour objectif d'insérer la clé k dans l'arbre t en respectant l'invariant de structure des arbres binaires de recherche et son extension pour inclure le rang du nœud dans l'ordre croissant des clés ainsi que dans l'ordre décroissant. Dans notre implantation de cet invariant, les fonctions $lrank(n)$ et $grank(n)$ correspondent aux données `int lrank;` et `int grank;` stockées dans le nœud.

L'insertion d'une clé dans un arbre augmenté par le rang se fait de façon très similaire à l'insertion d'une clé dans un arbre binaire de recherche. La seule différence est que, lors du parcours de l'arbre pour trouver la feuille sur laquelle insérer le nouveau nœud, les informations de rang (`int lrank;` et `int grank;`) sont mises à jour dans l'arbre.

1.2.2 Parcours de l'arbre selon le rang.

Lorsque l'on cherche la clé se situant à une certaine position dans la liste ordonnée des clés, pour trouver par exemple, la i^{eme} clé dans l'ordre croissant des clés, l'algorithme exploite les informations de rang qui ont été établies à la construction de l'arbre. L'algorithme décrit ci-dessous considère que l'on souhaite trouver la i^{eme} clé en partant de la plus petite (en position i dans l'ordre croissant des clés). Par symétrie, le même algorithme peut être utilisé pour trouver la i^{eme} clé en partant de la plus grande (en position i dans l'ordre décroissant des clés). Par exemple, dans la liste de clés `6 8 10 12 14 16 18`, la 3^{eme} clé dans l'ordre croissant est la clé `10`, la 3^{eme} clé dans l'ordre décroissant est la clé `14`.

Lorsque l'on visite un nœud, son rang dans l'ordre croissant des clés est calculé comme le nombre de nœuds qui lui sont inférieurs plus 1. Si ce rang est égal à la position recherchée, l'algorithme s'arrête et renvoie la clé du nœud visité. Si le rang du nœud visité est supérieur à la position recherchée, la recherche continue sur le sous-arbre gauche (les clés inférieures) avec la même position. Si le rang du nœud visité est inférieur à la position recherchée, la recherche continue sur le sous-arbre droit (les clés supérieures) en soustrayant le rang du nœud à la position recherchée.

2 Travail à réaliser.

2.1 Opérateur d'ajout d'une clé dans un arbre binaire de recherche.

En utilisant l'opérateur fourni `BinarySearchTree *tree_cons(int key)`, qui construit une feuille de l'arbre et établi son invariant de structure en tant que feuille de l'arbre, programmer, **de façon itérative, et sans mettre à jour les rangs**, la fonction `void tree_add(BinarySearchTree **t, int v)` qui insère une clé dans un arbre binaire de recherche.

2.2 Opérateur de parcours préfixe en profondeur d'abord de l'arbre.

Programmer, **de façon itérative**, l'opérateur `void tree_depth_prefix(const BinarySearchTree *t, OperateFunctor f, void *userData)` effectuant un parcours préfixe en profondeur d'abord de l'arbre et appelant la fonction `f(n, userData)` sur chaque nœud visité.

Pour vérifier ces 2 premières questions, vous pouvez compiler (make) et exécuter votre programme en lançant la commande `./tree_test ../Test/testfilesimple.txt`.

Vous pouvez produire le fichier pdf de visualisation de l'arbre que vous avez créé en tapant `make pdf`. Si vous ouvrez le fichier pdf `testfilesimple.dot.pdf` contenu dans le répertoire `../Test`, vous obtiendrez l'arbre de la figure 1 dans lequels les rangs sont initialisés à 0.

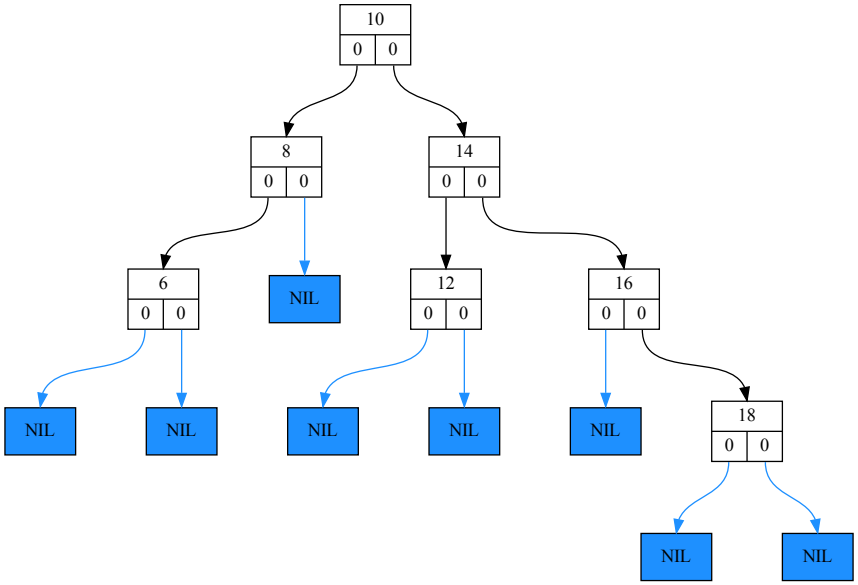

FIGURE 1 – Arbre construit avec le fichier testfilesimple.txt.

2.3 Opérateurs de parcours de l'arbre selon l'ordre croissant et décroissant des clés.

Programmer, **de façon récursive**, les opérateurs

— `void tree_inorder(const BinarySearchTree *t, OperateFunctor f, void *userData)`

— `void tree_reverseorder(const BinarySearchTree *t, OperateFunctor f, void *userData)`

et effectuant, respectivement, un parcours de l'arbre dans l'ordre croissant et dans l'ordre décroisant des clés et exécutant la fonction `f(n, userData)` sur chaque nœuds visités.

Le résultat attendu (affichage sur votre terminal) lors d'une nouvelle éxécution de votre programme est le suivant :

```
$ ./tree_test ../Test/testfilesimple.txt
...
Inorder visit of the tree : 6 8 10 12 14 16 18
Reverse order visit of the tree : 18 16 14 12 10 8 6
...
```

2.4 Établissement de l'invariant de structure lors de l'insertion d'une clé.

Modifier le constructeur `void tree_add(BinarySearchTree **t, int v)` de façon à établir l'invariant de rang sur les nœuds de l'arbre lors de tout ajout d'une nouvelle clé.

En exploitant l'invariant ainsi établi, programmer la fonction `int tree_size(const BinarySearchTree *t)` calculant, en temps constant, la taille de l'arbre.

Pour vérifier la bonne construction de l'arbre et de vos invariants de structure, vous exécuter votre programme et générer les fichiers pdf. Le résultat attendu (affichage sur votre terminal) est le suivant :

```
$ ./tree_test ../Test/testfilesimple.txt
...
Tree has 7 nodes.
...
```

Le fichier pdf produit correspond à l'arbre de la figure 2.

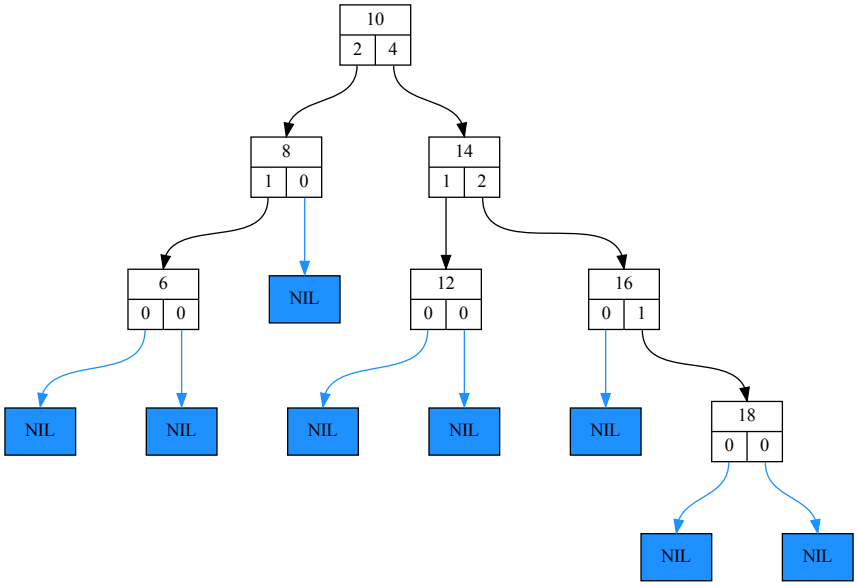

FIGURE 2 – Arbre augmenté par les rangs des nœuds.

2.5 Recherche d'une clé par son position

1. Programmer, **de façon récursive**, la fonction `int tree_kth_lower(const BinarySearchTree *t, int p)` effectuant la recherche de la clé qui se trouve à la position `p` dans l'ordre croissant des clés.
2. En identifiant les symétries permettant de passer d'une recherche dans l'ordre **croissant** à une recherche dans l'ordre **décroissant** des clés (e.g. accès aux sous-arbres, accès au rang, ...),
   (a) introduisez de façon privée au module `tree` le type `RankAccessors` d'une structure permettant de stocker les fonctions d'accès aux fils d'un nœeud,
   (b) programmer la fonction privée au module, `int kth_element(const BinarySearchTree *t, int r, RankAccessors accessors)`, qui effectue la recherche en utilisant les opérateurs d'accès aux fils stockés dans le paramètre `accessors`.
   (c) réécrire la fonction `int tree_kth_lower(const BinarySearchTree *t, int rank)` pour qu'elle appelle, avec les paramètres adéquats, la fonction `kth_element`
3. Programmer la fonction `int tree_kth_greater(const BinarySearchTree *t, int p)` effectuant la recherche de la clé qui se trouve à la position `p` dans l'ordre décroissant des clés pour qu'elle appelle, avec les paramètres adéquats, la fonction `kth_element`.

Exécuter le programme et vérifier que la sortie sur le terminal est cohérente avec les données de l'arbre.

```
$ ./tree_test ../Test/testfilesimple.txt
...
Searching K-th lower key of the tree.
        1st lower : 6
        3rd lower : 10
        7th lower : 18
Searching K-th greater key of the tree.
        1st greater : 18
        4th greater : 12
        7th greater : 6
```