Arbres binaires de recherche - implantation statique.

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 êtres réalisées par des opérateurs de complexité dans le pire cas en $O(log(n))$ pour un arbre équilibré contenant n clés. Nous nous intéressons, dans ce contrôle, à l'implantation des arbres binaires de recherche par un tableau de structures. Le problème de l'équilibrage de l'arbre ne sera pas abordé.

Dans une telle représentation, l'arbre est représenté par un tableau de nœuds, les liens entre les nœuds étant par conséquent représentés par des indices dans le tableau de nœuds. La représentation mémoire d'un nœud est une structure qui contient :
- les données du nœud, de façon générale un couple clé-valeur, représenté dans ce contrôle, et sans perte de généralité, par un entier *key*.
- un tableau de liens, *links*, vers les deux nœuds fils (les indices du fils gauche et droit dans le tableau de nœuds) et vers le nœud père (l'indice du nœud père dans le tableau de nœuds).

Si un lien n'existe pas (e.g. la racine de l'arbre n'a pas de nœud père), la valeur du lien est fixée à -1.

À partir du code fourni dans l'archive `controleTP.tar.gz` associée à ce sujet, construite sur le même modèle que les archives de TP et proposant une représentation interne (ainsi que les opérateurs fondamentaux de gestion de cette représentation) reposant sur le concept de tableau de structures décrit ci-dessus, l'objectif de ce contrôle est de programmer l'opérateur d'ajout d'une clé dans un arbre binaire de recherche, la recherche de l'existence d'une clé ainsi que la programmation de différents parcours de l'arbre pour fournir des fonctionnalités spécifiques.

1.1 Description du code fourni

Le code fourni 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 similaire à celle des arbres binaires de recherche vus en TP mais adaptée à la représentation privée de l'arbre,
  - un fichier `tree.c` définissant l'implantation et l'interface privée du TAD et à compléter pour ce contrôle.
- Un 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 et définitions de propriétés.

1.2.1 Ajout d'un élément de l'arbre.

La fonction `int tree_add(StaticSearchTree *t, int k)`, un constructeur du TAD, a pour objectif d'insérer la valeur k dans l'arbre t en respectant l'invariant de structure des arbres binaires de recherche implanté sous forme de tableau et appelés *arbre statique* dans la suite du sujet.

Les nœuds de l'arbre étant stockés dans un tableau, ils sont identifiés par des indices valides pour un tableau et la valeur `-1`, qui est un indice non valide, est utilisée pour indiquer qu'un nœud n'a pas de père ou qu'il n'a pas de fils sur le lien concerné.

L'insertion d'une valeur dans un *arbre statique* correspond, comme pour les arbres de recherche implantés en TP, à une étape de recherche de la feuille sur laquelle insérer le nouveau nœud suivie d'une insertion du nouveau nœud dans le tableau de nœuds. Ces deux étapes consistent en les opérations suivantes :

1. Parcours de l'arbre selon l'invariant des arbres binaires de recherche pour trouver le lien vers une feuille à mettre à jour.
2. Création du nouveau nœud avec mise à jour du lien identifié précédemment.

Dans cet algorithme, un lien est représenté par l'adresse mémoire de l'entier correspondant à l'indice du nœud dans le tableau. Ainsi, si le nouveau nœud doit être le fils gauche de son père, le lien à mettre à jour correspondra à l'adresse mémoire du champ `links[LEFT]` de ce nœud père. Si le nouveau nœud doit être le fils droit de son père, le lien à mettre à jour correspondra à l'adresse mémoire du champ `links[RIGHT]` de ce nœud père.

Une fois le lien identifié, l'ajout du nœud à l'arbre avec initialisation de ses propriétés en tant que feuille rattachée à un nœud père se fera par l'appel à la fonction fournie `int tree_add_node(StaticSearchTree *tree, int key, int parent, int* link)` avec `key` la valeur du nœud à créer, `parent` l'indice du nœud parent et `link` le lien à mettre à jour. Cette fonction ajoute le nouveau nœud au tableau de nœuds (en redimensionnant éventuellement ce tableau) et met à jour le lien indiqué avec l'indice du nouveau nœud dans le tableau. Cette fonction renvoie cet indice.

1.2.2 Parcours de l'arbre sans tenir compte de sa topologie.

Ce parcours, appelé parcours direct dans la suite du sujet, est un simple parcours linéaire du tableau de nœud, le nombre de nœuds de l'arbre étant stocké dans le champ `size` de la structure `StaticSearchTree`.

1.2.3 Plus proche ancêtre commun de deux nœuds.

Le plus proche ancêtre commun (ou *nca* pour *Nearest Common Ancestor*) de deux nœuds n_1 et n_2 est le nœud $n = nca(n_1, n_2)$ de l'arbre tel que, si $key(n_1) < key(n_2)$ alors $key(n_1) <= key(n) <= key(n_2)$. Ainsi,

— n_1 est l'ancêtre commun de n_1 et n_2 si n_2 est dans le sous-arbre droit de n_1,
— n_2 est l'ancêtre commun de n_1 et n_2 si n_1 est dans le sous-arbre gauche de n_2,
— si $n \neq n_1 \wedge n \neq n_2$ alors n_1 et n_2 sont respectivement dans le sous-arbre gauche et le sous-arbre droit de l'arbre enraciné en n

Par exemple, sur l'arbre de la figure 1, $nca(5, 7) = 6$, $nca(3, 6) = 4$ et $nca(1, 2) = 2$.

1.2.4 Distance entre deux nœuds.

La distance entre deux nœuds n_1 et n_2 dans un arbre est le nombre d'arêtes de l'arbre constituant le chemin le plus court entre n_1 et n_2. Ce chemin passe nécessairement par le plus proche ancêtre commun de n_1 et n_2.

2 Travail à réaliser

2.1 Opérateur d'ajout d'un élément dans un arbre statique

1. En utilisant l'opérateur `int tree_add_node(StaticSearchTree *tree, int key, int parent, int* link)` décrit ci-dessus, programmer, **de façon itérative**, la fonction `int tree_add(StaticSearchTree *t, int k)` qui ajoute un nœud de clé `k` dans l'arbre binaire de recherche `t`.
2. Programmer, **de façon itérative**, l'opérateur `void tree_map_on_nodes(const StaticSearchTree *t, VisitFunctor f, void *userData)` effectuant un parcours direct de l'arbre et appelant le foncteur `f` avec comme paramètres l'arbre, l'indice du nœud visité et les données utilisateur `userData` sur chaque nœud de l'arbre.

Pour vérifier cette première question, vous pouvez compiler (make) et exécuter votre programme en lançant la commande `./tree_test ../Test/testfilesimple.txt`.

Le résultat attendu dans le terminal est le suivant :

```
$ ./tree_test ../Test/testfilesimple.txt
Adding keys to the tree : 0 (4) - 1 (6) - 2 (7) - 3 (5) - 4 (2) - 5 (3) - 6 (1)
Exporting the tree to ../Test/testfilesimple.dot : 4 6 7 5 2 3 1
...
```

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.

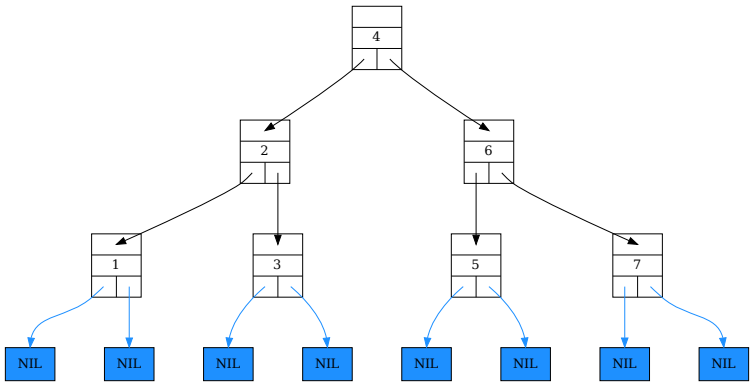

FIGURE 1 – Arbre construit avec le fichier testfilesimple.txt.

2.2 Opérateur de recherche d'un élément dans un arbre statique.

Programmer **de façon récursive et en temps optimal** , l'opérateur `int tree_search(const StaticSearchTree *t, int k)` qui cherche si la clé k est présente dans l'arbre t. Cet opérateur retourne l'indice dans le tableau de nœud de la clé trouvée ou -1 si la clé n'est pas trouvée.

Pour vérifier cette question, vous pouvez compiler (make) et exécuter votre programme en lançant la commande `./tree_test ../Test/testfilesimple.txt`. Le résultat attendu dans le terminal est le suivant :

```
$ ./tree_test ../Test/testfilesimple.txt
...
Searching keys in the tree.
        Value 7 found at index 2.
        Value 8 not found.
        Value 1 found at index 6.
        Value 0 not found.
...
```

2.3 Opérateurs de parcours de l'arbre dans l'ordre croissant ou décroisant des clés

— Programmer, **de façon itérative** et avec un stockage minimum, l'opérateur `void tree_inorder(const StaticSearchTree *t, VisitFunctor f, void *userData)` effectuant un parcours de l'arbre dans l'ordre croissant de ses clés.

— En identifiant les symétries permettant de passer d'un parcours **croissant** à un parcours **décroissant**,

1. introduisez de façon privée au module `tree` le type `ChildAccessFunctors` d'une structure permettant de stocker les fonctions d'accès aux fils d'un nœud,
2. programmez la fonction, privée au module `tree`, `void tree_visit_infix(const StaticSearchTree *t, VisitFunctor f, void *userData, ChildAccessFunctors functors)`, qui effectue le parcours en utilisant les opérateurs d'accès aux fils stockés dans le paramètre `functors`.
3. réécrire la fonction `void tree_inorder(const StaticSearchTree *t, VisitFunctor f, void *userData)` pour qu'elle appelle, avec les paramètres adéquats, la fonction `tree_visit_infix`

— Programmer la fonction `void tree_reverseorder(const StaticSearchTree *t, VisitFunctor f, void *userData)` effectuant un parcours de l'arbre dans l'ordre **décroissant** de ses clés.

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

Le résultat attendu est le suivant :

```
$ ./tree_test ../Test/testfilesimple.txt
...
Inorder visit of the tree : 1 2 3 4 5 6 7
Reverse order visit of the tree : 7 6 5 4 3 2 1
...
```

2.4 Recherche du plus proche ancêtre commun et distance entre nœuds.

— Programmer la fonction `int tree_nearest_common_ancestor(const StaticSearchTree *t, int k1, int k2)` qui renvoie l'indice du nœud étant le plus proche ancêtre commun aux nœuds de clés k_1 et k_2. Vous prendrez soin de ne pas introduire de contraintes sur k_1 et k_2 autres que la précondition de cette opération, nécessitant que k_1 et k_2 soient des clés valides de l'arbre.

— Programmer la fonction `int tree_distance(const StaticSearchTree *t, int k1, int k2)` qui calcule la distance entre les nœuds de clés k_1 et k_2 dans l'arbre t.

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

Le résultat attendu est le suivant :

```
$ ./tree_test ../Test/testfilesimple.txt
...
Searching nearest common ancestor of keys.
        nearest_common_ancestor(3, 5) --> 4 (0)
        nearest_common_ancestor(1, 4) --> 4 (0)
        nearest_common_ancestor(7, 5) --> 6 (1)
        nearest_common_ancestor(2, 2) --> 2 (4)
Computing distances between keys.
        distance(3, 5) --> 4
        distance(1, 4) --> 2
        distance(7, 5) --> 2
        distance(2, 2) --> 0
```