Arbres binaires cousus - Threaded binary trees.

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.

Extension d'un Type Abstrait de Données

L'adaptation d'un type abstrait de données aux besoins d'une application est réalisée grâce à l'*extension* de ce TAD et de son implantation.
L'*extension* du TAD consiste en la définition d'un nouvel invariant de structure venant étendre l'invariant initial du TAD.
L'*extension* de son implantation consiste en la programmation efficace de la représentation concrète du TAD et de ses opérateurs.

Ce contrôle a pour objectif l'extension d'un TAD à des besoins applicatifs spécifiques.

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 contenant n clés.
L'invariant de structure défini sur le TAD arbres binaires de recherche en fait une structure de données ordonnée qu'il est alors possible de parcourir de façon à traiter les éléments dans l'ordre croissant (ou décroissant) de leur clé. Un tel parcours peut s'écrire simplement de façon récursive mais nécessite alors un espace de travail de l'ordre de $O(log(n))$ (gestion de la pile d'appels récursif). En 1968, Donald Knuth a posé la question de l'existence d'un algorithme de parcours de l'arbre dans l'ordre de ses clés, n'utilisant pas de mémoire supplémentaire et ne modifiant pas l'arbre pendant le parcours. L'algorithme de parcours infixe itératif d'un arbre binaire vu en cours et implanté en TP est une des réponses à cette question mais présente une complexité en temps de l'ordre de $O(nlog(n)))$.
En 1979, J.H. Morris à proposé une solution permettant de ramener la complexité en espace en $O(1)$ et la complexité en temps en $O(log(n))$. Cette solution, appelée ***Threaded trees*** *(arbres cousus)* tire partie de la place perdue dans la structure d'arbre lors du stockage des feuilles pour lesquelles les pointeurs vers le sous-arbre gauche et le sous-arbre droit sont inutilisés (ou désignent une sentinelle, NIL).

Un arbre cousu (figure 2) est construit en remplaçant tous les pointeurs vers un sous arbre droit qui devraient être nuls par un pointeur vers le successeur du nœud dans l'arbre.

Bien évidemment, il faut alors pouvoir différencier la sémantique de ce pointeur en rajoutant une information permettant d'identifier la nature du pointeur.
Nous nous intéressons, dans ce contrôle, à l'implantation des arbres doublement cousus. Chaque nœud cousu peut référencer son prédécesseur (dans le fils gauche d'un nœud interne n'ayant qu'un fils droit), son successeur (dans le fils droit d'un nœud interne n'ayant qu'un fils gauche) ou les deux à la fois (dans une feuille).

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 cousu et l'exploitation de cette couture pour effectuer une traversée itérative dans l'ordre de l'arbre.

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 `tbstree.h` définissant l'interface publique d'un arbre cousu, interface identique à celle des arbres binaires de recherche vus en TP et étendue pour la gestion des coutures,
  - un fichier `tbstree.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 `tbstree.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 tbstree_add(ptrThreadedBinaryTree *t, int v)`, un constructeur du TAD, a pour objectif d'insérer la valeur v dans l'arbre t en respectant l'invariant de structure des arbres binaires de recherche et son extension dans le cadre des arbres cousus :

Invariant de couture

Dans un arbre binaire de recherche cousu, un nœud est cousu à gauche (respectivement à droite) s'il ne possède pas de fils gauche (respectivement droit) et qu'il possède un prédécesseur (respectivement successeur) dans l'arbre. Le fils gauche (respectivement droit) de ce nœud désigne alors son prédécesseur (respectivement successeur) dans l'arbre.

L'insertion d'une valeur dans un arbre cousu se fait de façon très similaire à l'insertion d'une valeur dans un arbre binaire de recherche et correspond aux étapes suivantes :

1. Parcours de l'arbre selon l'invariant des arbres binaires de recherche pour recherche de la position d'insertion.
2. Arrêt de la recherche lorsque l'on trouve une couture ou une feuille (NIL)
3. Création du nouveau nœud en remplacement de la couture ou de la feuille trouvée.
4. Mise à jour des informations de couture sur le nouveau nœud inséré.

Les étapes 1, 2 et 3 ci-dessus sont identiques, à la condition d'arrêt près, à l'étape d'insertion dans un arbre binaire de recherche. L'étape 4 permet d'établir l'invariant de couture sur le nouveau nœud.

1.2.2 Parcours de l'arbre en exploitant les coutures.

Si l'on souhaite faire un parcours de l'arbre dans le sens croissant de ses clés, on va pouvoir exploiter les coutures pour passer directement d'un nœud à son successeur sans remonter dans l'arbre. Si le nœud est cousu, on utilise le lien pour accéder à son successeur.
Si le nœud n'est pas cousu, alors, il faut parcourir l'arbre pour atteindre son successeur. Ce parcours est systématique et correspond à rechercher le nœud le plus à gauche du sous arbre droit.

2 Travail à réaliser

2.1 Programmer l'opérateur d'ajout d'un élément dans un arbre cousu

En utilisant l'opérateur fourni `ThreadedBinaryTree *tbstree_cons(int root)`, qui construit une feuille de l'arbre et établi son invariant de structure en l'absence de prédécesseur et de successeur, et en exploitant les informations de couture stockées dans la structure `struct _tbstree`, programmez les étapes 1 à 3 de l'algorithme d'insertion dans la fonction
`void tbstree_add(ptrThreadedBinaryTree *t, int v)`.

2.2 Programmer les opérateurs de parcours infixe et préfixes

Programmer, de façon récursive, les opérateurs `void tbstree_depth_infix(const ThreadedBinaryTree *t, OperateFunctor f, void *userData)` et `void tbstree_depth_prefix(const ThreadedBinaryTree *t, OperateFunctor f, void *userData)` effectuant respectivement un parcours infixe et préfixe en profondeur d'abord.
Pour vérifier ces 2 premières questions, vous pouvez compiler (make) et exécuter votre programme en lançant la commande `./threadedtree_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.

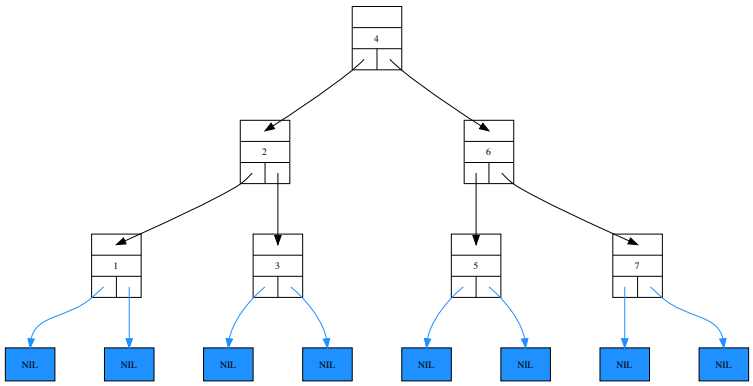

FIGURE 1 – Arbre construit avec le fichier testfilesimple.txt.

2.3 Programmer l'établissement de l'invariant de structure lors de l'insertion d'une valeur

Programmer la fonction `void fix_tree_threads(ptrThreadedBinaryTree current)` qui met à jour les informations de couture concernant le nœud nouvellement inséré `current`. Cette mise à jour concerne d'une part le parent du nœud, qui était cousu et ne l'est plus sur la relation `parent<->current` et le nœud lui même qui doit être cousu sur ses deux liens fils.

Après avoir programmé cette fonction, modifiez votre fonction `void tbstree_add(ptrThreadedBinaryTree *t, int v)` pour établir l'invariant de structure sur le nœud nouvellement créé.
Pour vérifier la bonne construction de l'arbre et de vos invariants de structure, vous pouvez compiler (make) et exécuter votre programme en lançant la commande `./threadedtree_test ../Test/testfilesimple.txt`. le fichier pdf produit par `$make pdf` correspond à l'arbre de la figure 2.

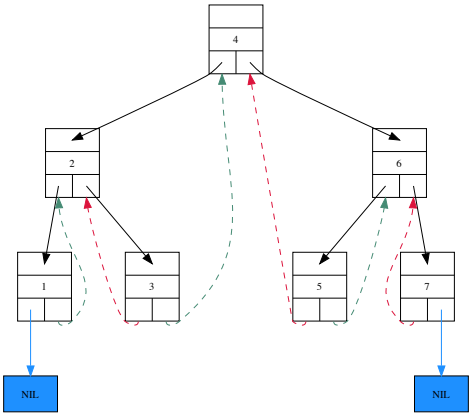

FIGURE 2 – En vert pointillé, coutures droites. En rouge pointillé, coutures gauches.

2.4 Traversée de l'arbre en exploitant les coutures

Programmer la fonction itérative `void tbstree_inorder(const ThreadedBinaryTree *t, OperateFunctor f, void *userData)` effectuant un parcours préfixe de l'arbre, dans l'ordre croissant des clés en exploitant les coutures, et qui applique l'opérateur `f(n, userData)` sur tous les nœuds n traversés.