Tas binaire maximal - Maximal Binary Heap.

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 être réalisées par des opérateurs de complexité dans le pire cas en $O(log(n))$ pour un arbre contenant n clés.
Lorsque l'on souhaite, pour des besoins applicatifs de gestion de file de priorité par exemple, accéder (faire une recherche ou une suppression) toujours à l'élément maximal de la collection, il est alors nécessaire d'étendre le type abstrait de données afin de rendre l'opération d'accès à l'élément maximal extrêmement efficace, en $O(1)$ au lieu de $O(log(n))$ tout en conservant des complexité en $O(log(n))$ pour les autres opérations..
Un tas binaire est une structure de données, introduite par J.W.J Williams en 1964 permettant d'atteindre ces objectifs et dont les propriétés sont enoncées ci-dessous.

— Un tas binaire est un arbre binaire *complet*.
— Dans un tas binaire, la clé stockée sur chaque nœud est plus grande ($\geq$ - tas maximal) ou plus petite ($\leq$ - tas minimal) que les clés stockées dans ses fils droit et gauche. Il n'y a pas d'ordonnancement particulier entre les fils droits et gauche.

Les tas binaires peuvent être implantés comme une structure de donnée implicite (i.e. dans un tableau, les relations père-fils n'étant pas stockées mais calculées comme une fonction des indices) et sont alors à la base de l'algorithme de tri en place *heap sort*.
Nous nous intéressons, dans ce contrôle, à l'implantation des tas maximaux par une structure de données explicite (les relations père-fils sont explicitement définies par le stockage d'une information) sous forme d'une extension d'une structure d'arbre binaire permettant de respecter les propriétés des tas énoncées ci-dessus.
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 définir la représentation d'un tas maximal ainsi que les opérateurs principaux pour sa construction et son parcours.

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 `maxheaptree.h` définissant l'interface publique d'un tas maximal, interface très similaire à celle des arbres binaires de recherche vus en TP,
  - un fichier `maxheaptree.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 `maxheaptree.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 nœud dans un arbre binaire complet

Un tas maximal étant un arbre complet, il donc est nécessaire d'établir l'invariant des arbres complets à la construction de l'arbre.

Invariant des arbres complets

Dans un arbre binaire *complet*, tous les niveaux de l'arbre doivent êtres remplis (l'arbre de la figure 1 est un arbre complet). En considérant que la racine est au niveau 0 de l'arbre, un niveau i possède alors 2^i nœuds. Seul le dernier niveau peut être remplis partiellement et ses nœuds sont alors tassés à gauche (remplissage de gauche à droite du dernier niveau).

Cet invariant doit ensuite être maintenu par toute opération de modification de l'arbre : Add et Remove. Si l'on analyse la manière dont se rempli un arbre complet, niveau par niveau et de gauche à droite, l'ajout d'un nouvel élément se fera donc toujours à la suite du dernier élément ajouté. L'algorithme d'ajout d'un nœud dans un arbre binaire complet doit donc rechercher le premier nœud incomplet rencontré lors d'un parcours en profondeur d'abord commençant sur le dernier nœud ajouté.
En considérant que l'on dispose du dernier élément ajouté, nommé *last* ci après, et que l'on recherche le nœud y, ce parcours correspond à un des 3 cas ci-dessous :

1. *last* est la racine de l'arbre, elle est donc incomplète et *y=last*.
2. *last* est le fils gauche de son père qui est donc incomplet. On a donc *y=parent(last)*.
3. *last* est le fils droit de son père, on remonte alors dans l'arbre en suivant la relation parent tant que le parent est le fils droit de son père. Soit z le nœud sur lequel on s'arrête. Si z a un fils droit, z = *right(z)*. Le premier nœud incomplet y se trouve maintenant au bout de la branche gauche de z.

1.2.2 Ajout d'un élément dans un tas maximal.

La fonction `void mhtree_add(ptrMaxHeapTree t, int v)`, à pour objectif d'insérer la valeur v dans le tas binaire t en respectant l'invariant d'ordonnancement des tas binaires.

Invariant d'ordonnancement des tas binaires

Dans un tas binaire, la clé stockée sur chaque nœud est plus grande ($\geq$ - tas maximal) ou plus petite ($\leq$ - tas minimal) que les clés stockées dans ses fils droit et gauche. Il n'y a pas d'ordonnancement particulier entre les fils droits et gauche.

L'insertion d'une valeur dans un tas binaire commence donc d'abord par l'insertion d'une valeur dans un arbre binaire complet. Après cette insertion, l'invariant d'ordonnancement peut alors ne plus être vérifiée pour le père du nœud ajouté. Pour rétablir cet invariant, tant que la valeur ajoutée est plus grande que la valeur du père, on remonte dans l'arbre en permutant les valeurs. Cette opération est réalisée par la fonction `void percolate_up(ptrNode x)`.

2 Travail à réaliser

2.1 Définition de la représentation interne d'un tas binaire maximal

1. En prenant soin de ne stocker que les informations nécessaires et en utilisant l'implantation fournie pour les arbres binaires (le TAD `Node`), écrire la représentation interne du TAD `MaxHeapTree` par la définition complète de la structure `struct _maxheaptree`.
2. Programmer les fonctions `MaxHeapTree *mhtree_create()` construisant un tas binaire vide et `bool mhtree_empty(const MaxHeapTree *t)` retournant vrai si un tas binaire est vide.

2.2 Opérateur d'ajout d'un élément dans un arbre binaire complet

Après avoir programmé la fonction `ptrNode search_free_node(ptrNode last)` de recherche du premier nœud incomplet situé après le nœud `last` selon l'algorithme décrit précédemment, programmer la fonction `void mhtree_add(ptrMaxHeapTree t, int v)` ajoutant la valeur `v` dans te tas binaire `t` sans établir l'invariant d'ordonnancement.

2.3 Opérateur de parcours préfixes de l'arbre

Programmer, de façon récursive, l'opérateur `void node_depth_prefix(const Node *n, OperateFunctor f, void *userData)` effectuant un parcours préfixe en profondeur d'abord à partir du nœud `n`.
En utilisant la fonction précédente, programmer l'opérateur `void mhtree_depth_prefix(const MaxHeapTree *t, OperateFunctor f, void *userData)` de parcours du tas binaire maximal `t`.
Pour vérifier ces 3 premières questions, vous pouvez compiler (make) et exécuter votre programme en lançant la commande `./maxheaptree_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.

2.4 Programmer l'établissement de l'invariant d'ordonancement dans un tas binaire

Programmer la fonction `void percolate_up(ptrNode x)` qui rétabli l'invariant d'ordonnancement dans un tas binaire maximal.
Après avoir programmé cette fonction, modifiez votre fonction `void mhtree_add(ptrMaxHeapTree t, int v)` pour établir l'invariant à partir du nœud nouvellement créé.
Pour vérifier la bonne construction du tas binaire et de vos invariants de structure, vous pouvez compiler (make) et exécuter votre programme en lançant la commande `./maxheaptree_test ../Test/testfilesimple.txt`.
Le fichier pdf produit par `$make pdf` correspond à l'arbre de la figure 2.

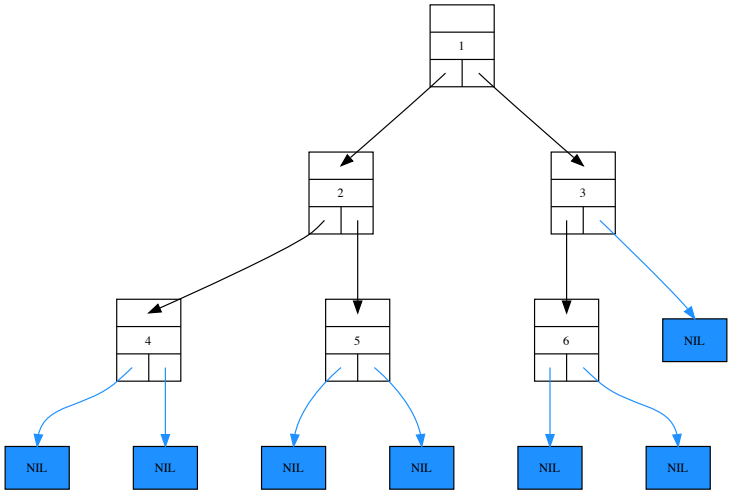

FIGURE 1 – Arbre construit avec le fichier testfilesimple.txt.

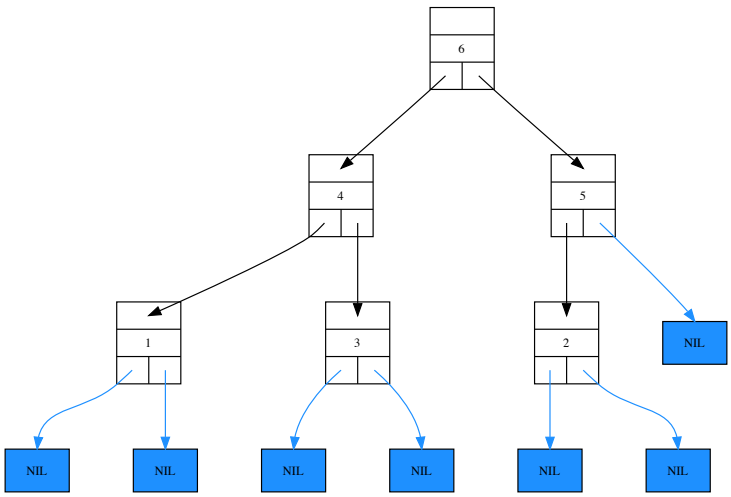

FIGURE 2 – Tas binaire maximal correspondant au ficher de données testsimple.txt.