P31 Développement efficace - épisode 5

1 - Les tas

Un tas (`heap`) est un arbre binaire *presque* complet (tous les niveaux sont remplis, sauf éventuellement le dernier, qui est rempli de gauche à droite) qui vérifie la propriété suivante : la valeur de chaque noeud est supérieure ou égale à la valeur de ses deux fils.

Ainsi, le noeud de plus grande valeur est toujours directement accessible car il se situe à la racine de l'arbre.

Les opérations possibles sur un tas sont :

- `insert` : ajouter un élément dans le tas
- `max` : accéder à l'élément de plus grande valeur du tas
- `extract` : supprimer l'élément de plus grande valeur du tas

L'opération `insert` peut être réalisée en ajoutant un élément à la fin de l'arbre (en remplissant le dernier niveau ou en créant un nouveau niveau si le précédent est complet), puis en le faisant remonter (par échanges successifs) jusqu'à ce que la propriété de tas soit vérifiée.

L'opération `extract` peut être réalisée en écrasant l'élément de plus grande valeur (la racine) avec le dernier élément de l'arbre (la feuille la plus à droite du dernier niveau), puis en le faisant descendre (par échanges successifs) jusqu'à ce que la propriété de tas soit vérifiée.

Pour nous aider dans cette tâche, on va définir deux opération supplémentaires :

- `heap_send_up` : remonter un élément dans l'arbre jusqu'à ce que la propriété de tas soit vérifiée
- `heap_send_down` : descendre un élément dans l'arbre jusqu'à ce que la propriété de tas soit vérifiée

2 - Mise en oeuvre

On peut mettre en oeuvre une structure d'arbre binaire de manière "classique" avec un ensemble de noeuds qui contiennent chacun, en plus des données, des pointeurs vers son père, son fils gauche et son fils droit.

Cependant, dans le cas d'un arbre binaire *presque* complet, on peut éviter la dispersion de la mémoire et l'usage de pointeurs en stockant simplement les données dans un `vector`. Chaque noeud est identifié par son index dans le `vector` et on matérialise les relations hiérarchiques entre les noeuds de la façon suivante :

- le noeud d'indice `i` a pour fils gauche le noeud d'indice `2i+1` et pour fils droit le noeud d'indice `2i+2`
- le noeud d'indice `i` a pour père le noeud d'indice `(i-1)/2` (division entière)

> ***Créer les fichiers `containers/heap.h` et `containers/heap.c`.***
> ***Définir la structure `heap` et l'ensemble des fonctions suivantes :***

- initialisation et libération :

```
// en plus de la taille des éléments, on fournit également un pointeur vers une
// fonction de comparaison d'éléments, à stocker dans la structure
// (cette fonction est une donnée essentielle du tas car elle permet de définir
// la propriété de tas)
void heap_init(heap *h, size_t value_size,
               int (*compare)(void const *a, void const *b));
void heap_free(heap *h);
```

- accès aux propriétés courantes :

```
size_t heap_size(heap const *h);
size_t heap_value_size(heap const *h);
```

- navigation dans l'arbre :

```c
// les trois fonctions suivantes sont indépendantes d'un tas en particulier
size_t heap_left_child(size_t i);
size_t heap_right_child(size_t i);
size_t heap_parent(size_t i);
```

- mise à jour de la propriété de tas :

```c
void heap_send_up(heap *h, size_t i);
void heap_send_down(heap *h, size_t i);
```

- manipulation des données :

```c
// cette fonction peut être écrite en suivant la procédure décrite plus haut
void heap_insert(heap *h, void const *value);
// copie la valeur contenue dans la racine du tas à l'adresse pointée par value
void heap_max(heap const *h, void *value);
// cette fonction peut être écrite en suivant la procédure décrite plus haut
void heap_extract(heap *h, void *value);
```

3 - Construction d'un tas

On peut construire un tas en partant d'un tas vide et en insérant les éléments un par un. Il est aussi possible de partir d'un `vector` quelconque (c'est-à-dire d'un arbre binaire *presque* complet quelconque) et de le transformer en tas.

Pour cela, on peut partir des constatations suivantes :

- si les deux sous-arbres d'un noeud sont des tas, alors il suffit d'appliquer `heap_send_down` sur ce noeud pour obtenir un tas.
- toutes les feuilles de l'arbre sont des tas.

> À quels indices se trouvent les feuilles de l'arbre ?

> ***Écrire la fonction suivante qui transforme le `vector` interne du `heap` `h`, supposé quelconque, en un tas.***
>
> ```c
> void heap_build(heap *h);
> ```

4 - Tri par tas

Le principe du tri par tas est le suivant :

- on construit un tas à partir du `vector` à trier
- en parcourant les index `i` de la dernière à la première case du `vector` :
  - on extrait l'élément de plus grande valeur du tas
  - on le place dans la case `i` du `vector`

Dans un premier temps, on pourra réaliser la première étape en insérant les éléments un par un dans un tas, que l'on libérera à l'issue de l'algorithme.

> ***Créer les fichiers `algos/heap_sort.h` et `algos/heap_sort.c`.***
> ***Définir la fonction suivante :***
>
> ```c
> void heap_sort_vec(vector *v,
>                    int (*compare)(void const *a, void const *b));
> ```

L'inconvénient de cette approche est que l'on a besoin d'une zone mémoire supplémentaire, de même taille que le `vector` initial, pour stocker le tas intermédiaire.

Il est possible de mettre en oeuvre cet algorithme *"in place"*, en utilisant directement les données du `vector` initial au sein du `vector` interne d'un tas. On peut alors transformer ce `vector` en tas à l'aide de la fonction `heap_build`. Il ne reste alors plus qu'à extraire les éléments de plus grande valeur du tas un par un pour les placer à la fin du `vector`, tout en faisant *croire* au tas que le `vector` qu'il contient est de plus en plus petit.

> ***Réfléchir à la mise en oeuvre de cette version de l'algorithme, puis, si la réflexion est concluante, l'écrire.***