Algorithmique et programmation fonctionnelle

Feuille 7 - Tas binaires

Dans cette feuille nous allons manipuler des éléments ayant des priorités (les priorités seront données par des entiers) et nous allons considérer une nouvelle structure de donnée permettant d'implémenter les *files de priorité*.
Les *tas binaires* sont conçus afin de pouvoir réaliser les opérations suivantes rapidement :

- Accès à l'élément de *meilleure priorité*.
- Suppression de *cet* élément.
- Ajout d'un *nouvel élément* (quelconque).
- Fusion de deux tas binaires.

Nous allons prendre la convention que les éléments les plus prioritaires sont ceux ayant une petite priorité.

Comparaison avec les arbres binaires de recherche AVL.

L'avantage des tas binaires sur les arbres AVL est que l'*accès au plus petit élément* dans un tas s'effectue en *temps constant* (*i.e.*, $O(1)$). En revanche, rechercher si un élément quelconque se trouve dans un tas binaire s'effectue en complexité $O(n)$ (où n est la taille du tas binaire) alors que dans un arbre AVL, la complexité était $O(log_2(n))$. Cette situation est classique : telle ou telle structure a des avantages et des inconvénients suivant l'utilisation qu'on veut en faire.

Multiplicités.

Une autre différence avec les arbres binaires de recherche est qu'un tas binaire peut contenir « plusieurs copies » d'un même élément.

Nous impémenterons cette nouvelle structure de données à l'aide d'arbres quasi-parfait. Nous avons vu qu'un arbre quasi-parfait de hauteur h peut avoir entre 2^h et $2^{h+1} - 1$ nœuds et donc que sa hauteur est, asymptotiquement, $O(\log_2 n)$.

1 Tas binaires - définitions et algorithmes

Définition

Un ***tas binaire*** est une structure de données représentée par un arbre binaire étiqueté par des entiers, vérifiant les deux propriétés suivantes :

- propriété sur sa forme : l'arbre doit être ***quasi-parfait***,
- ***propriété de tas*** : l'étiquette d'un nœud doit toujours être inférieure ou égale à chacune de celles de ses fils. Autrement dit, quand on lit les étiquettes sur le long d'une branche en partant de la racine, on obtient une suite croissante (au sens large, un élément peut apparaitre plusieurs fois).

Exercice 1 : Retour de l'élément minimal d'un tas

Où se trouve l'élément minimal dans un tas binaire ? Quelle est la complexité pour le rechercher ?

Exercice 2 : Propriété de tas

1. Proposez un algorithme récursif pour tester si un arbre binaire a la propriété de tas.
2. On veut compter le nombre d'appels récursifs effectués par cette fonction sur un arbre de taille n. On note ce nombre $a(n)$. En vous basant sur l'algorithme, écrivez une relation de récurrence satisfaite par la suite $a(n)$ et en déduire la valeur de $a(n)$.

On veut maintenant insérer un élément dans un tas binaire de façon efficace, ainsi que pouvoir supprimer l'élément de plus petite valeur (c'est-à-dire le plus prioritaire).

Exercice 3 : Insertion d'un élément, suppression de l'élément le plus prioritaire

Rappelez les deux algorithmes suivants (vus en cours), et expliquez précisément la raison pour laquelle leur complexité est $O(\log_2 n)$, où n est le nombre de nœuds du tas.

1. Ajout d'une nouvelle priorité dans le tas,
2. Suppression de l'élément minimal du tas (si l'élément apparaît plusieurs fois dans le tas, on ne supprime qu'une occurrence).

Pour les deux algorithmes de l'exercice 4 nous utiliserons les exercices de la feuille précédente sur les chemins dans un arbre quasi-parfait.

2 Implémentation à l'aide de chemins dans l'arbre

On veut implémenter l'insertion et la suppression dans un tas binaire. Pour supprimer, on doit connaître la liste de directions menant à la « dernière feuille » de l'arbre qui représente le tas. Pour insérer, on doit connaître la liste de directions qui mènerait au premier « emplacement libre » pour une feuille.

On a vu dans la feuille précédente que ces listes de directions peuvent se calculer en fonction du nombre de nœuds dans l'arbre. On va donc représenter un tas par un arbre binaire **et** sa taille. On utilise pour les arbres le type habituel :

```
type 'a btree = Empty | Node of 'a * 'a btree * 'a btree
```

Un tas sera représenté par le type suivant :

```
type binheap = Heap of int btree * int
```

L'arbre représente le tas lui-même, et l'entier est sa taille, qu'il faudra maintenir. Cette taille permettra de calculer la liste de directions menant à la dernière feuille.

Exercice 4 :

Quel est l'intérêt d'ajouter la taille dans la structure de données, dans la mesure où on peut la calculer à partir de l'arbre ?

On rappelle le type direction en OCaml :

```
type direction = L | R.
```

Dans tous nos algorithmes, nous nous servirons de la fonction `get_dirlist : int -> direction list` (voir feuille 6) qui prend en paramètre la taille de l'arbre représentant le tas et renvoie la liste de directions menant à sa dernière feuille.

La recherche de l'élément minimal de l'arbre est trivial. Nous allons donc nous intéresser d'abord à l'insertion d'un nouvel élément dans le tas.

Exercice 5 : Ajout d'un élément dans un tas binaire

L'algorithme vu en cours comporte en deux étapes : insérer le nouvel élément, puis à réparer la propriété de tas.

1. Écrire une fonction `bubble_up : 'a btree -> direction list -> 'a btree` qui implémente la procédure de réparation de l'arbre : `bubble_up` prend en paramètres,

   - un arbre `t` quasi-parfait, qui a la propriété de tas sauf éventuellement pour sa dernière feuille, qui peut porter une valeur plus petite que celle de son père,
   - la liste `dirlist` de directions menant à cette dernière feuille.

   La fonction renvoie l'arbre obtenu en faisant remonter cette valeur à une position qui redonne la propriété de tas, par échanges successifs d'étiquettes entre un nœud et son père.

2. Écrire une fonction `insert : 'a -> 'a btree -> direction list -> 'a btree` qui insère une nouvelle valeur dans un arbre représentant un tas, en cassant éventuellement la propriété de tas (qui pourra être réparée par la fonction précédente `bubble_up`). Cette fonction prend en paramètres :

   - une valeur `x` à ajouter dans l'arbre passé en second argument,
   - un arbre `t` quasi-parfait, qui a la propriété de tas,
   - la liste de directions menant au **premier emplacement libre**.

   La fonction renvoie l'arbre obtenu de `t` en ajoutant une feuille `x` à ce premier emplacement libre.

3. En déduire une fonction `heap_insert : int -> binheap -> binheap` qui retourne le tas obtenu en ajoutant une valeur à un tas existant.

Exercice 6 : Suppression de l'élément minimal dans un tas binaire

L'élément le plus prioritaire est à la racine.
L'algorithme vu en cours comporte deux étapes : remplacer cet élément par un autre élément du tas qui est peu prioritaire, réparer l'arbre pour maintenir la propriété de tas.

1. Écrire une fonction `bubble_down : 'a btree -> 'a btree` qui implémente la procédure de réparation de l'arbre : `bubble_down t` prend en paramètre un arbre `t` quasi-parfait, qui a la propriété de tas sauf éventuellement à la racine, qui peut porter une valeur plus grande que celle d'un ou de chacun de ses fils.

   La fonction renvoie l'arbre obtenu en faisant redescendre cette valeur à une position qui redonne la propriété de tas, par échanges successifs d'étiquettes entre un nœud et son père.

2. Écrire une fonction `delete : 'a btree -> direction list -> ('a btree * 'a)` prenant en paramètre un arbre et la liste de direction menant à une feuille, et qui renvoie une paire contenant l'arbre privé de cette feuille et l'étiquette de celle-ci.

3. Écrire une fonction `heap_delete : binheap -> binheap` prenant en entrée un tas binaire `h` et renvoyant le tas binaire obtenu en supprimant de `h` une occurrence de sa plus petite valeur.

Exercice 7 : Conversion entre listes et tas. Tri par tas

En utilisant les fonctions `heap_insert` et `heap_delete`, écrire :

1. Une fonction `list_to_heap` qui crée un tas binaire contenant exactement les éléments de la liste d'entrée (avec les mêmes multiplicités)
2. Une fonction `heap_to_list : binheap -> int list` qui crée la liste triée de tous les éléments du tas qu'elle prend en entrée.
3. Une fonction `heap_sort : int list -> int list` qui prend en entrée une liste d'entiers et renvoie la liste triée correspondante. Expliquez pourquoi la complexité de cette fonction est $O(n \log n)$.