Tri par tas

Jean-Baptiste Rouquier

1 Remarques sur le TP précédent

Le plus grand élément d'un tas n'est pas nécessairement à la profondeur maximale. Regarder par exemple un arbre dégénéré où au moins l'un des deux fils de chaque nœud est une feuille.

Je n'ai rien contre le fait d'écrire directement la fonction `is_heap` sans passer par `test-racine`. Mais c'est plus difficile et il ne faut pas oublier de comparaison. Il faut lire tout le sujet pour comprendre son articulation et où l'auteur veut vous mener.

Une convention légèrement désuète veut que les lignes ne dépassent pas 80 caractères (c'était une erreur de syntaxe en Fortran !). Des lignes de 100 caractères sont encore lisibles et tolérables, mais 205 (sic) non. Revenez régulièrement à la ligne dans le code pour qu'il soit agréable à lire.

Au lieu de `if y<x || z<x then true else false`, on écrit simplement `y<x || z<x`. De même, au lieu de `let foo = bar+qux in foo` on écrit `bar+qux`.

Toutes les fonctions devront être testées sur un exemple raisonnable. On s'attachera à la clarté du code[1] et l'on ajoutera des commentaires chaque fois que nécessaire.

Un test qui montre que la fonction ne marche pas fait perdre des points.

2 Tableaux Dynamiques

Ce sont des tableaux qui permettent d'ajouter des éléments à la fin, en gardant un coût amorti (ie un coût moyen par opération) constant. L'idée est d'avoir un tableau trop grand, ainsi on a des cases vides à la fin dans lesquelles on peut mettre les nouveaux éléments. Il faut donc mémoriser le nombre d'éléments qui sont réellement dans le tableau, les suivants étant des cases vides :

```
type 'a dynarray = {mutable taille : int; mutable elements : 'a array}
```

Attention, `taille` n'est pas la taille du tableau `elements`.

Si on veut ajouter un élément alors que le tableau `elements` est plein, on en alloue un plus grand, on y recopie `elements` au début (utiliser `Array.blit`), et on utilise ce nouveau tableau à la place.

Voilà pour le principe, la première question est «combien plus grand ?». On implémentera «deux fois plus grand».

Question 2.1. Implémenter `add : 'a -> 'a dynarray -> unit` qui ajoute un élément à la fin du tableau et `take : 'a dynarray -> 'a` qui le retire.

Question 2.2. Sur le modèle du module Array de la librairie standard, implémenter `get, set, make, iter` ainsi que `dynarray_of_array : 'a array -> 'a dynarray`.

Question 2.3 (bonus). On multiplie donc de temps en temps la taille de `elements` par deux. Pour libérer de la mémoire quand on ne l'utilise plus, il faut aussi la diviser par deux lorsque l'on retire un élément et que `taille` devient inférieur à la moitié de la taille de `elements`. Modifier les fonctions précédentes pour inclure cela.

On a obtenu une première structure de données intéressante : des tableaux dont la taille s'adapte dynamiquement aux nombre d'éléments à stocker.

[1] en particulier, éviter les noms de variable à une ou deux lettres

3 Arbres binaires

Pour chercher à être plus efficace, on va représenter les arbres binaires non pas comme un type somme mais sous la forme d'un tableau. La racine est l'élément d'indice 0, les fils de l'élément d'indice i sont les éléments d'indice $2i+1$ et $2i+2$.

Question 3.1. Écrire les fonctions

```
pere : int -> int
fils : int -> int * int
```

Question 3.2. Visualiser et expliquer rapidement quelles sont les formes d'arbres que l'on peut ainsi représenter.

Pour ne pas avoir à recopier le tableau chaque fois que l'on ajoute ou retire un élément, on utilise les tableaux dynamiques de la section 2 pour représenter les arbres.

Question 3.3. Visualiser et expliquer rapidement où est ajouté le noeud lorsque l'on appelle les fonctions `add` et `take` de la section 2.

4 Tas

Definition (rappel). Un ***tas*** (en anglais *heap*) est un arbre tel que pour chaque nœud :
$(*)$ l'étiquette est plus petite, au sens large, que celle de chacun de ses fils.

Cette structure de donnée permet donc de savoir quel est le plus petit élément en temps constant. Mais on voudrait aussi pouvoir le retirer. Un moyen simple est de le supprimer, ce qui crée un trou à la racine, de remplir ce dernier par le plus petit de ses deux fils, et de faire ainsi descendre le trou jusqu'à une feuille (que l'on peut alors supprimer). Mais ceci ne garantit pas que l'arbre reste équilibré et on ne peut plus alors le représenter comme dans la section 3.

La solution est de boucher le trou de la racine par le dernier élément du tableau (celui renvoyé par `take`). L'arbre ainsi obtenu est bien représenté comme dans la section 3.

Mais ce n'est plus un tas (la racine est trop grande). On échange donc la racine avec le plus petit de ses deux fils, et on continue à la faire descendre ainsi jusqu'à ce qu'elle soit plus petite que ses deux fils (ou bien jusqu'à une feuille). Cette opération s'appelle *percoler*.

Question 4.1 (oralement). Se persuader que l'on obtient bien un tas à la fin.

Question 4.2 (difficile). Implémenter la fonction récursive `percole : 'a dynarray -> int -> unit` qui prend un arbre dont tous les nœuds sauf celui d'indice donné en argument vérifient la condition $(*)$ (c'est donc presque un tas). Cette fonction fait descendre l'étiquette de ce nœud par échanges successifs jusqu'à obtenir un tas.

On pourra écrire une fonction `echange : 'a dynarray -> int -> int -> unit` à l'aide de `get` et `set`.

Question 4.3. Implémenter `take_min : 'a dynarray -> 'a` qui renvoie le minimum d'un tas et le retire de ce dernier.

Question 4.4. Quelle est la complexité de cette fonction ?

Reste à créer un tas à partir d'un tableau. Pour cela on considère brutalement ce tableau comme un arbre. Si les fils droit et gauche de la racine sont des tas, il suffit d'appeler `percole` pour que l'arbre entier soit un tas. On commence donc par transformer ces deux sous-arbres en tas. En utilisant bien sûr la même idée, de façon récursive.

C'est donc que l'on commence par les feuilles, qui sont déjà des tas. Puis l'on passe au niveau d'au dessus : on percole pour que les sous-arbres constitués d'une, deux ou trois feuilles soient des tas. Et l'on remonte ainsi jusqu'à la racine. Il suffit donc de percoler les nœuds du dernier au premier.

Question 4.5. Implémenter `taïfie : 'a array -> 'a dynarray`.

On peut montrer que cette fonction a un temps d'exécution linéaire.

Question 4.6. En regroupant les questions précédentes, implémenter `tri : 'a array -> 'a array`. (Si vous n'avez pas le temps de l'écrire, donnez le schéma général de cette fonction).

Question 4.7. Quelle est la complexité de ce tri ?

Question 4.8 (bonus). Implémenter la fonction `insere : 'a -> 'a dynarray -> unit` qui ajoute un élément à un tas. Au lieu de percoler de haut en bas comme dans `take_min`, l'idée est d'ajouter l'élément à la fin du tableau et de le faire monter tant qu'il est plus petit que son père.

On a ainsi une seconde structure de données intéressante, la file avec priorité : on peut construire la structure à partir d'un tableau, insérer un élément et retirer le plus urgent. C'est la bonne structure de données à utiliser pour le TP sur l'algorithme de Dijkstra.