Informatique, TP 2
Les tas

Thomas Blanc

8 novembre 2012

1 Introduction

1.1 Objectifs

On veut mettre au point un algorithme prenant en entrée un tableau, et qui trie ce dernier. Pour cela, on utilisera la méthode du tri par tas qui est complexe, mais très efficace.

1.2 Tri naïf

(a) Écrire une fonction *trouver_max : 'a array -> int* qui prend en entrée un tableau et renvoie l'indice du plus grand élément. Vérifier que la complexité de la fonction en temps est linéaire et que la complexité en espace est constante.

(b) Reprendre le code de la fonction précédente et écrire une autre fonction *trouver_max_limit : 'a array -> int -> int* qui fait la même chose mais en ne regardant que les n premiers éléments du tableau, avec n deuxième argument de la fonction.

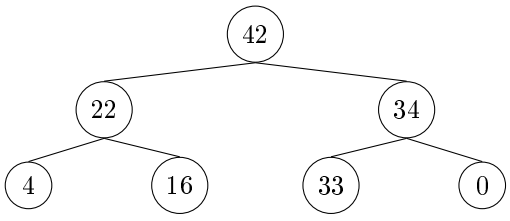
(c) Écrire une fonction *tri_naif : 'a array -> unit* qui trie un tableau en utilisant la fonction *trouver_max_limit*. Quelle est sa complexité ?

2 Le tri par tas

2.1 principe

L'idée de cet algorithme est simple : plutôt que de trier le tableau, on va commencer par l'organiser en tas. Mais qu'est-ce que c'est qu'un tas ?

Un tas est un arbre binaire équilibré dont le plus grand élément est tout en haut et dont la règle est simple : tout élément doit être plus grand que ses enfants.



Bien sûr, on ne va pas construire un arbre, cela prendrait une place inutile en mémoire, on va plutôt organiser le tableau comme s'il était un arbre.

Pour cela on dira que les enfants de l'élément *t.(i)* sont les éléments *t.(2*i)* et *t.(2*i+1)*.

42	22	34	4	16	33	0

En donnant cette forme au tableau, on pourra le trier très rapidement.

Une propriété du tas que l'on utilisera très souvent : un sous-tas (c'est à dire une sous-partie du tas) est lui-même un tas.

2.2 Créer un tas

(a) Écrire des fonctions *fils_gauche*, *fils_droit* et *parent*, toutes de type *int -> int* qui pour un indice i renvoient l'indice des enfants et du père.

(b) Lorsque l'on aura notre tas, on en enlèvera petit à petit des éléments. On veut que la structure de tas soit toujours maintenue. Écrire donc la fonction *recreer_tas : 'a array -> int -> unit* qui prend en entrée un tableau et la coordonée d'un point, qui suppose que les enfants de ce point forment des tas triés mais que ce point n'est pas forcément à sa place, et qui fera descendre le point autant que nécessaire. Calculer la complexité de cette procédure.

(c) On va construire notre tas en utilisant *recreer_tas*. Notons qu'un point seul est un tas. Quelle est la complexité de cet algorithme ?

2.3 Trier le tableau

(a) Lorsque le tas est formé, prenez l'élément le plus grand et échangez le avec le dernier élément du tableau, puis effectuez un *recreer_tas* modifié pour ne pas prendre en compte l'élément qu'on a mis au bout du tableau.

(b) Écrire la fonction *tri_par_tas : 'a array -> unit*

(c) Pourquoi ne pas permettre de trier selon un ordre différend de celui dont on a l'habitude ? Écrire *tri_par_tas_generique : ('a -> 'a -> int) -> 'a array -> unit* qui prend comme premier argument une fonction du même genre que *compare*

3 Files de priorité

3.1 Fonctionnement

On veut écrire une file de priorité, c'est à dire une structure de donnée permettant trois actions :

inserer : *'a t -> 'a -> 'a t* pour insérer une nouvelle valeur.

maximum : *'a t -> 'a* qui donne le haut du tas.

pop : *'a t -> 'a t* qui supprime le haut du tas.

3.2 Mise en place

On va commener par définir notre type :

```
type 'a t = 'a tas * ('a -> 'a -> int)
and 'a tas = Noeud of 'a * 'a tas * 'a tas | Feuille
```

Écrire maintenant la fonction *maximum : 'a t -> 'a* qui renvoie le haut du tas s'il y en a un et envoie l'exception *Not_found* si le tas est vide

3.3 Actions sur le tas

(a) Écrire la fonction *inserer*. Attention de mettre la nouvelle valeur sans détruire la structure d'arbre.

(b) Écrire la fonction *pop* en reprenant le principe de *recreer_tas*

3.4 Améliorations

(a) Écrire la fonction *fusion : 'a t -> 'a t -> 'a t* qui fusionne deux tas en supposant qu'ils ont la même fonction de comparaison.

(b) Définir un nouveau type de tas où le nombre d'enfants d'un tas n'est pas forcément deux mais un nombre défini dans le type. Quels sont les avantages et inconvénients d'une telle structure ?

4 Le tri rapide

4.1 Fonctionnement

On va laisser de coté les tas et mettre en place un nouveau type de tri : le tri rapide. Ce tri fonctionne sur le principe du diviser pour régner.

Pour cela, on effectuera un raisonnement en trois étapes :

1. prendre un des éléments x du tableau, mettre à gauche les éléments plus petit que lui, et à droite les éléments plus grands. Mettre x au milieu et renvoyer son indice.
2. Trier le sous-tableau des éléments plus petits que x.
3. Trier le sous-tableau des éléments plus grands que x.

Notez que cet algorithme produit deux appels récursifs.

4.2 Partition

Écrire une fonction *partition : ('a -> 'a -> int) -> 'a t -> int -> int -> int*. Cette fonction prend en entrée :

1. Une fonction de comparaison.
2. Le tableau à trier.
3. L'indice du premier élément du sous-tableau sur lequel on travaille.
4. L'indice du dernier élément du sous-tableau.

Cette fonction doit choisir un élément du sous-tableau (lequel n'est pas important, prenez le premier par exemple). Elle doit ensuite ranger les éléments plus petit que cet élément sur la gauche et ceux plus grand sur la droite. Elle met après cela l'élément choisi entre les deux sous-tableaux créés et renvoie l'indice trouvé.

4.3 Mise en place

Écrire la fonction *tri_rapide : ('a -> 'a -> int) -> 'a array -> unit*, n'oubliez pas la condition d'arrêt qui évite de trier des sous-tableaux vides.

4.4 Preuve et complexité

(a) Montrer par récursion que le tableau ainsi obtenu est trié.

(b) Quelle est la complexité dans le pire des cas ? Commenter.