▶ **MP & MP* – Option Informatique**
Quatrième TP Caml
Mardi 23 et 30 octobre 2007

Tris par tas

Pour compléter le tri par insertion, le tri fusion, le tri rapide et le tri à bulle, nous allons voir maintenant le dernier tri classique : le tri par tas.

1 Tableaux dynamiques

Il s'agit d'une structure de données qui sont ressemblent à des tableaux, mais qui permettent d'ajouter des éléments à la fin avec un coût moyen par opération (on parle également de coût amorti) constant.
L'idée de base est d'utiliser des tableaux classiques, mais qu'on a choisi trop grands : ainsi, on peut rajouter des éléments facilement (en temps constant) s'il reste de la place. S'il ne reste plus de place, on crée un nouveau tableau plus grand (on choisira une taille *deux* fois plus importante) et on y recopie le tableau d'origine.

▶ **Question 1** *Définissez un nouveau type Caml* dynarray *pour représenter un tableau dynamique. On a besoin de connaître au moins la taille vue par l'utilisateur et les différents éléments.* Maintenant que l'on a ce type, il faut définir des fonctions de base pour le rendre utilisable :

▶ **Question 2** *Écrivez les fonctions* add *et* take *qui respectivement ajoute un élément à la fin du tableau et enlève le dernier élément du tableau.*

```
add : 'a −> 'a dynarray −> unit = <fun>
take : 'a dynarray −> 'a = <fun>
```

▶ **Question 3** *Comme pour les tableaux Caml d'origine, implémentez les fonctions* get *(qui récupère dans un tableau dynamique un élément d'indice donné - c'est la fonction* vect_item *en Caml),* set *(qui change la valeur d'un élément d'indice donné dans un tableau dynamique - c'est la fonction* vect_assign *en Caml),* make *(qui crée un tableau de taille donnée),* iter *(qui applique une fonction sur tous les éléments d'un tableau dynamique) ainsi qu'une fonction de conversion* dynarray_of_array *qui transforme un tableau en tableau dynamique.*

```
get : 'a dynarray −> int −> 'a = <fun>
set : 'a dynarray −> int −> 'a −> unit = <fun>
make : int −> 'a −> 'a dynarray = <fun>
iter : ('a −> 'b) −> 'a dynarray −> unit = <fun>
dynarray_of_array : 'a vect −> 'a dynarray = <fun>
```

▶ **Question 4** *On multiplie de temps en temps la taille des tableaux utilisés par* 2*. Pour libérer la mémoire utilisée, n'oubliez pas que la fonction* take *pourrait utiliser un tableau de taille plus petite. Cependant, pour garder un coût moyen constant, il ne faut pas libérer la mémoire trop souvent. On décide donc de libérer la mémoire quand la taille utilisée par le tableau dynamique est inférieure à la moitié de la taille disponible.*

```
take : 'a dynarray −> 'a = <fun>
```

2 Arbres binaires

Une méthode simple pour représenter un arbre binaire en Caml consiste à utiliser un type somme, mais cette représentation n'est pas très efficace. C'est pour cette raison que nous allons représenter un arbre binaire sous la forme d'un tableau : la racine sera l'élément d'indice 0, et les deux fils gauche et droit d'un élément d'indice i auront respectivement les indices $2i + 1$ et $2i + 2$.

▶ **Question 5** *Ecrivez les fonctions* pere *et* fils *qui respectivement donnent l'indice du père de l'élément d'indice i et les indices des fils de l'élément d'indice i.*

```
pere : int −> int = <fun>
fils : int −> int ∗ int = <fun>
```

▶ **Question 6** *Quelles sont les formes d'arbre que l'on peut représenter de cette façon ? Comme nous allons utiliser des arbres de tailles variables, nous allons naturellement utiliser les tableaux dynamiques. Si on ajoute un élément à un tableau avec la fonction* add*, à quel noeud cela correspond-il dans l'arbre représenté ?*

3 Tas

On rappelle qu'un *tas* (*heap*, en anglais) est une structure de données similaire à un arbre étiquetté (c'est-à-dire que chaque noeud de l'arbre a une étiquette, ici un entier) tel que l'étiquette d'un père est toujours plus petite (au sens large) que les étiquettes de chacun de ses fils.

Naturellement, il est facile de calculer en temps constant le plus petit élément d'un tas : il suffit d'en prendre la racine ! Cependant, les choses deviennent plus compliquées si on veut retirer des éléments du tas, par exemple si on retire la racine. On pourrait se contenter de retirer la racine et de la remplacer par le plus petit de ses deux fils et de propager le trou ainsi créé jusqu'aux feuilles. Cependant, cela ne garantit pas que l'arbre reste équilibré. La solution utilisée est de remplacer la racine par le dernier élément du tableau (celui renvoyé par la fonction take définie dans la première partie). L'arbre qu'on obtient alors est représentable comme dans la section 3. Malheureusement, l'arbre obtenu est certes représentable, mais ce n'est plus un tas. On va donc rétablir cette charmante propriété en échangeant la racine avec le plus petit de ses deux fils, puis on va progressivement la faire descendre jusqu'à ce qu'elle soit plus petite que ses deux fils (ou jusq'à une feuille). Cette opération s'apelle *percoler*.

▶ **Question 7** *Se persuader que l'on obtient bien un tas à la fin de cette opération.*

▶ **Question 8** *Implémenter la fonction* percole *qui prend en argument un arbre qui est un tas à l'exception d'un unique noeud (d'indice donné). La fonction* percole *va donc faire descendre l'étiquette donnée par échanges successifs jusqu'à obtenir de nouveau une structure de tas.*

```
percole : 'a dynarray −> int −> unit = <fun>
```

▶ **Question 9** *Ecrivez une fonction* take_min *qui renvoie le minimum d'un tas tout en le retirant de ce dernier.*

```
take_min : 'a dynarray −> 'a = <fun>
```

▶ **Question 10** *Quelle est la complexité en temps de cette fonction, en fonction du nombre total n d'éléments dans le tas ?*

Maintenant, nous allons chercher à transformer un tableau quelconque en tas. On va tout d'abord considérer que le tableau est un arbre.

Nous avons vu que si les deux sous-arbres de la racine sont des tas, alors il suffit d'appeller la fonction percole pour transformer l'ensemble en tas. Il faut donc commencer par transformer les deux sous-arbres de la racine en tas... et de façon récursive, on arrivera ainsi aux différentes feuilles.

C'est pour cela que nous allons commencer directement par les feuilles, qui sont bien évidemment des tas. Ensuite, on passe au niveau précédent (ou suivant, bref au niveau d'au-dessus) et on percole pour ce que ces arbres à une, deux ou trois feuilles soient des tas. Peu à peu, nous arriverons à la racine et le tableau entier seront des tas.

▶ **Question 11** *Ecrivez donc cette fonction* heap_from_array *!*

```
heap_from_array : 'a vect −> 'a dynarray = <fun>
```

▶ **Question 12** *En utilisant les questions précédentes, écrivez une fonction de tri qui trie un tableau quelconque.*

```
tri : 'a vect −> 'a vect = <fun>
```

▶ **Question 13** *Quelle est la complexité de ce tri ?*

▶ **Question 14** *Ecrivez une fonction* insere *qui ajoute un élément à un tas. L'idée est de l'ajouter à la fin, puis de percoler à l'envers pour le faire remonter (tant qu'il est plus petit que son père).*

```
insere : 'a dynarray −> 'a −> unit = <fun>
```

▶ **MP & MP* – Option Informatique**
Quatrième TP Caml
Mardi 23 et 30 octobre 2007

Tris par tas

Un corrigé

▶ Question 1

```
type 'a dynarray = { mutable size : int; mutable elements : 'a vect};;
```

▶ Question 2

```
let add = fun elt array ->
let size = array.size in
if size < vect_length array.elements then (
array.elements.(size) <- elt;
array.size <- array.size + 1;
)
else (
let temp = make_vect (max 1 (2*size)) elt in
blit_vect array.elements 0 temp 0 size;
array.elements <- temp;
array.size <- array.size + 1;
)
;;
let take = fun array ->
let size = array.size in
if size <= 0 then failwith "Tableau_dynamique_vide"
else (
array.size <- size - 1;
array.elements.(size - 1);
)
;;
```

▶ Question 3

```
let get = fun array i -> array.elements.(i);;
let set = fun array i elt -> array.elements.(i) <- elt;;
let make = fun length elt -> {size = length; elements = make_vect length elt};;
let iter = fun f array ->
for i = 0 to (array.size - 1) do
f array.elements.(i)
done
;;
let dynarray_of_array = fun array ->
{size = vect_length array; elements = copy_vect array};;
```

▶ Question 4

```
let take = fun array ->
let size = array.size in
if size <= 0 then failwith "Tableau_dynamique_vide";
array.size <- size - 1;
let res = array.elements.(size - 1) in
if(size - 1) <= (vect_length array.elements / 2) then
array.elements <- sub_vect array.elements 0 (size - 1);
res
;;
```

▶ Question 5

```
let pere = fun i -> (i-1) / 2;;
let fils = fun i -> 2*i+1, 2*i+2;;
```

▶ **Question 6** Il faut que les arbres soient équilibrés, c'est-à-dire que chaque père a exactement deux fils (sauf éventuellement au dernier niveau, dans lequel un père peut n'avoir aucun fils).
Ajouter progressivement les noeuds avec la fonction add revient à compléter l'arbre niveau par niveau, remplissant chaque niveau de gauche à droite.

▶ **Question 7** Le seul élément qui fait que notre arbre n'est pas un tas est la racine. On l'échange avec le plus petit de ses deux fils (mettons le gauche). Le sous-arbre droit est un tas, la racine et son premier fils ne gênent pas non plus. Comme on réitère le procédé, et comme seul le sous-arbre descendant de la racine initiale ne forme pas un tas, on finit par tomber sur un tas, quitte à ce que la racine initiale soit une feuille.

▶ Question 8

```
let rec percole = fun arbre pere ->
let valeur = get arbre pere and (fils_g, fils_d) = fils pere in
if fils_g < arbre.size then (* si noeud a un fils gauche *)
let valeur_g = get arbre fils_g in
if fils_d < arbre.size then (* si noeud a aussi un fils droit *)
let valeur_d = get arbre fils_d in
let (min_fils, min_valeur) = mini ((fils_g, valeur_g),(fils_d,valeur_d)) in
if valeur > min_valeur then (
(* on fait l'échange *)
set arbre pere min_valeur;
set arbre min_fils valeur;
percole arbre min_fils
)
else (* si le papa n'a qu'un enfant *)
if valeur > valeur_g then (
set arbre pere valeur_g;
set arbre fils_g valeur;
percole arbre fils_g
)
(* on rappelle que si le papa n'a pas de fils gauche, il n'en a pas non plus à droite ;) *
;;
```

▶ Question 9

```
let take_min = fun arbre ->
if arbre.size = 0 then failwith "tas_vide";
let r = get arbre 0 in
let racine = take arbre in
set arbre 0 racine;
percole arbre 0;
r;;
```

▶ **Question 10** Globalement, l'extraction se fait en temps constant (en moyenne), et la percolation fait seulement une descente dans l'abre, dont la hauteur est globalement en $O(\log(n))$.

▶ **Question 11**

```
let heap_from_array = fun array ->
let arbre = dynarray_of_array array in
if arbre.size > 1 then
for i = pere (arbre.size - 1) downto 0 do
percole arbre i
done;
arbre
;;
```

▶ **Question 12**

```
let tri = fun array ->
let heap = heap_from_array array in
for i = 0 to vect_length array do
array.(i) <- take_min heap
done;
array
;;
```

▶ **Question 13** On fait un tri sur un tableau *quelconque*, il ne peut donc pas être en moins que $O(\log(n))$! Attention, cette remarque n'est valable que pour les tris généraux, vu qu'il est facile de faire une fonction de tri en temps constant si on ne l'applique que sur des tableaux déjà triés. Accessoirement, on fait n opérations qui ont chacune une taille $\log(n)$, donc on atteint bien cette complexité.

▶ **Question 14**

```
let rec inv_percole = fun arbre fils ->
if fils > 0 then
let pere = pere fils in
if get arbre fils < get arbre pere then (
let temp = get arbre fils in
set arbre fils (get arbre pere);
set arbre pere temp;
inv_percole arbre pere;
)
;;
let insere = fun arbre elt ->
add elt arbre;
inv_percole arbre (arbre.size - 1)
;;
```