Notes de cours Option Info MP/MP* : Tas et files de priorité

Tas binaires

Un arbre binaire est dit *parfait* si tous ses niveaux sont remplis sont éventuellement le dernier, qui est rempli partiellement de la gauche vers la droite. Un arbre parfait est en particulier équilibré (on a $2^{h(A)} \leqslant |A| < 2^{h(A)+1}$ donc $h(A) = \lfloor \log_2(|A|) \rfloor$).

Un *tas-max* (resp. *tax-min*) est un arbre binaire parfait étiqueté par un ensemble ordonné tel que l'étiquette de tout nœud autre que la racine soit inférieure ou égale (resp. supérieure ou égale) à l'étiquette de son père. Dans un tax-max (resp. tax-min), la racine est donc l'élément ..... (resp. .....).

Implémentation d'un tas

On peut représenter un tas par le type `'a arbre`, dans ce cas on a une structure persistante (immuable) mais également par un tableau `'a array`. On numérote les nœuds de la manière suivante: la racine porte le numéro 1, et si un noeud porte le numéro k, son fils gauche porte le numéro $2k$ et son fils droit $2k+1$. Par conséquent, le père du nœud numéroté i est numéroté ..... Dans ce cas la structure est impérative mais statique (on peut modifier les éléments mais pas la taille, ce qui fait qu'on aura souvent un tableau plus grand que le nombre d'éléments du tas). Notons que les tableaux en Caml sont numérotés de 0 à $n-1$, nous allons donc adopter la convention de ne pas mettre d'élément dans la case 0 (on aurait pu aussi tout décaler d'un cran).

Exercice: Déterminer le chemin de la racine au nœud d'indice i.

Construction d'un tas

Dans cette partie, nous considérerons des tas-max représentés par un tableau (où l'on ne met pas d'élément dans la case numérotée 0).

Préservation de la structure de tas

Nous aurons souvent besoin de reconstituer la structure de tas après qu'un élément a été modifié. Pour cela nous avons deux méthodes: faire monter un élément situé trop bas, ou faire descendre un élément situé trop haut.

Pour la montée, cela n'est pas difficile, on commence par écrire une fonction qui échange deux éléments d'indice i et j dans un tableau, et à chaque étape on compare l'élément à son père en les échangeant si besoin. (*fonctions à écrire en TP)*

La descente est plus compliquée, il faut effectivement tenir compte du fait qu'un nœud n'a pas toujours 2 fils, il peut n'en avoir qu'un ou même aucun. Pour cela on intègre en argument le nombre `n` d'éléments totaux présents dans le tas (ce qui correspond aussi à l'indice où est stocké le dernier élément, car on n'écrit pas dans la case 0). Si le nœud a deux fils, on le compare au plus grand des deux. (*à faire en TP*)

Le temps d'exécution de ces deux fonctions est en .....

Construction du tas

Il y a deux manières de construire un tas en partant d'un tableau `t` (on rappelle qu'on n'écrit pas dans la case 0): la première consiste à insérer aux feuilles les éléments de gauche à droite et à les faire remonter. Ce procédé vérifie l'invariant suivant: au moment d'insérer l'élément d'indice k, les $k-1$ premiers éléments forment un tas. (*à écrire en TP*)

Analysons le temps d'exécution de cette fonction: dans le pire des cas, à l'insertion de l'élément d'indice k, on doit remonter jusqu'à la racine (c'est le cas si t est initialement trié dans l'ordre croissant), donc le nombre de permutation nécessaires est égal à la profondeur du nœud. Si on note $p = \lfloor \log(n) \rfloor$ la hauteur du tas, le nombre total de permutations est:

$$\sum_{k=1}^{p-1} k2^k + p(n - 2^p + 1) = p(n+1) - 2^{p+1} + 2 = O(n\log(n))$$

On peut cependant faire mieux en insérant les éléments de droite à gauche et en utilisant la descente plutôt que la montée, en effet on remarque que la moitié droite du tableau correspond aux feuilles, donc si on commence par les insérer il n'y aura aucune descente à faire. (*à écrire en TP)*

Évaluons la complexité: au moment de l'insertion du nœud d'indice k, il y a au plus h_k permutations à faire où h_k est la hauteur du sous-arbre dont la racine est le nœud k. Or, si l'on se donne un entier h entre 0 et $h(T)$, il y a au plus $\lceil \frac{n}{2^{h+1}} \rceil$ nœuds tels que $h_k = h$. Dès lors, le nombre de permutations est majoré par:

$$\sum_{h=1}^{p} \lceil \frac{n}{2^{h+1}} \rceil h \leqslant \frac{p(p+1)}{2} + n\sum_{h=1}^{p} \frac{h}{2^{h+1}} = O(n),$$

car la série de terme général $\frac{h}{2^{h+1}}$ converge. La complexité est linéaire dans ce cas.

Tri par tas

Étant donné un tableau `t`, le principe du tri par tas est le suivant: on transforme le tableau en tas, l'élément maximal est alors la racine. On la place en dernière place du tableau, et on fait descendre la nouvelle racine pour reconstituer un tas. On recommence alors jusqu'à obtenir un tas de longueur 1.

Fonction à écrire en TP

Pour prouver la validité de cet algorithme, on utilise l'invariant suivant : A la fin de chaque itération de la boucle for, le tableau `t[1...k-1]` est un tas constitué des $k-1$ plus petits éléments et `t[k ...n]` contient les $n-k+1$ plus grands éléments dans l'ordre.

Complexité temporelle: .....

Signalons aussi que le tri est en place (on ne recrée pas de nouveau tableau) contrairement au tri fusion.

Comparer le tri par tas aux autres algorithmes de tris vus l'année dernière.

Files de priorité

Une file de priorité est une structure de données qui permet de gérer des éléments auxquels on a attaché une priorité (qui est un entier ou un élément d'un ensemble ordonné). Les opérations que l'on souhaite effectuer sont les suivantes:

- Tester si la file est vide.

- Créer une file vide.
- Ajouter un élément.
- Retirer un élément de priorité maximale.

Évidemment on veut que le coût de ces fonctions soit le plus petit possible.

Les files de priorité peuvent par exemple servir à gérer les tâches d'un ordinateur: chaque tâche est affectée d'une priorité et on veut que l'ordinateur effectue d'abord les tâches de priorité maximale.

Elles peuvent également servir à trier des éléments d'un ensemble ordonné: la priorité d'un élément est l'élément lui-même, on retire au fur et à mesure les éléments de priorité maximale.

Implémentation naïve

On peut modéliser une file de priorité par une liste triée par ordre décroissant de priorité. Dans ce cas:

- L'ajout d'un élément est en .....
- L'accès à l'élément de priorité maximal est en .....
- La liste n'est pas limitée en taille.

Si on utilise un tableau à la place d'une liste, la taille de celui-ci est déterminée dès le début.

Remarque: Si on utilise cette implémentation pour trier, on retrouve le tri ....., de complexité .....

Implémentations avec un tas

On peut également représenter une file de priorité par un tas-max. Dans ce cas l'élément de priorité maximale est la racine. Comme nous l'avons vu, il y a deux implémentations possibles d'un tas:

-Persistante avec un arbre binaire.

-Impérative avec un tableau.

Implémentation persistante

Regardons le coût des fonctions d'ajout d'un élément et de suppression de l'élément de priorité maximale.

- Pour ajouter un élément dans un tas représenté par un arbre binaire, on commence par calculer le chemin de la racine vers le nouveau nœud. A chaque étape de ce chemin, on fait descendre l'élément de priorité minimale. La complexité est en .....
- L'élément de priorité maximale étant la racine, une fois supprimée on la remplace par le "dernier" élément (celui le plus à droite dans la dernière ligne) qu'on fait ensuite redescendre. La complexité est en .....

Implémentation impérative

- L'ajout d'un élément peut se faire à la racine ou aux feuilles comme nous l'avons vu, il faut dans ce cas soit faire remonter soit faire descendre le nouvel élément. Dans les deux cas le coût est en .....
- La suppression se fait comme dans le cas précédent, son coût est en .....

Si on utilise une file implémentée par un tas pour trier, on retrouve exactement le principe du tri par tas.