TP8 MPSI : Files de priorité et Graphes

fabien.viger@ens.fr
18 Mai 2005

But du TP

Dans un simulateur d'évènements en temps réel, on a en général une liste finie de tâches, chacune devant être traitée à une certaine date. On veut les traiter dans l'ordre chronologique, mais le problème est que le traitement de la première tâche peut très bien créer d'autres tâches, que l'on doit correctement placer dans la liste de tâches en fonction de leur échéance (pas forcément en dernier, donc). C'est le cas des systèmes d'exploitation par exemple. Dans ce TP, on va s'intéresser aux structures de *tas*, et leur application directe aux files de priorité. On introduira ensuite les graphes, et on verra une application des tas sur un algorithme de graphes bien connu.

Normalement, on doit commencer à coder à partir de la question 2. Bien sûr, **à n'importe quel moment, toute question est la bienvenue.**

Question 1 : File de priorité naïve

Sans nécéssairement la réaliser, réfléchir à une implémentation naïve d'une file de priorité, à base d'une liste où les tâches seraient triées par date d'exécution croissante. Il faudrait pouvoir :

- Avoir accès très vite à la première tâche
- La supprimer
- Insérer des nouvelles tâches à leur bonne place, en fonction de leur date d'éxécution.

Question 2 : Structure de tas

Un tas est un simple arbre binaire, qui, contrairement aux arbres binaires de recherche, respecte les simples *condition de tas* :

1. Tout élément du tas est plus prioritaire que ses deux fils
2. L'arbre est quasi-parfait : toutes les feuilles sont à une même profondeur p ou $p-1$, et celles qui sont à une profondeur p sont toutes situées à gauche de l'arbre.

Montrer que l'élément minimum du tas (donc, celui de priorité maximale) est situé à la racine.

Montrer que l'on peut coder très efficacement cette structure à l'aide d'un simple vecteur : par exemple, le tas donné en exemple serait codé par `[|1; 4; 2; 5; 9; 7; 3; 6; 6|]`.

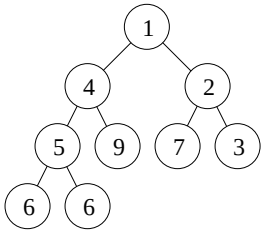

FIG. 1 – *Un exemple de tas*

Soit i la position (dans le vecteur) d'un élément x. Coder les fonctions `fg`, `fd` et `pere` telles que $fg(i)$ soit la position du fils gauche de x, $fd(i)$ celle de son fils droit et $p(i)$ celle de son père.

Question intermédiaire

Recoder la bonne vieille fonction `swap_vect` de type

```
int vect -> int -> int -> unit
```

telle que `swap_vect v i j` modifie le vecteur v en place en échangeant les éléments situés aux positions i et j.

Question 3 : Implémentation

On considère un système d'au plus N tâches, chacune identifiée par leur date d'exécution. On va stocker dans un vecteur d ces dates avec une structure de tas. Le nombre n de tâches stockées pourra être inférieur à N, il faut donc garder ce n stocké quelquepart.

Pour bien fixer les choses, définir le type `tas` : un tas sera un couple (n, d) où n est le nombre de tâches stockées dans le tas et d le vecteur contenant les dates des tâches.

```
type tas == int ref * (int vect);;
```

On utilise le type `int ref` plutôt que le type `int` pour stocker n : ainsi, le tas sera modifiable en place.

Écrire une fonction `cree_tas` de type `int -> tas` qui crée un tas initialement vide, de taille maximale passée en argument.

Écrire une fonction `verif` de type `tas -> bool` qui vérifie si le tas donné est valide.

Verifier que la 1$^{\text{ere}}$ ligne renvoie `true` et la 2$^{\text{eme}}$ `false` :

```
verif (ref 5,[|1;2;4;3;4;99;99;99|]);;
verif (ref 4,[|1;3;2;2;-1|]);;
```

Question 4 : Insertion dans un tas

Supposons que le "dernier" (dernier dans le vecteur) élément du tas vient d'être inséré, et qu'il entre en conflit avec la propriété de tas car il est plus petit que son père (cf dessin).

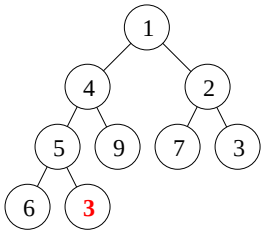

FIG. 2 – *Insertion d'une nouvelle tâche*

Comment, sachant la position i de notre nouvel élément, "réparer" notre tas pour qu'il soit valide, en seulement $O(\log n)$ opérations ?

Écrire une fonction `inser` de type `tas -> int -> unit` telle que

```
inser t date
```

insère une tâche de date *date* dans le tas t. S'aider eventuellement d'une fonction récursive `corrige_up` de type `tas -> int -> unit` qui s'occupera de corriger le défaut situé à la position i, en modifiant le tas en place. On pourra aussi préférer une procédure de correction itérative, integrée dans la fonction `inser`.

Écrire une fonction `rand_tas` de type `int -> tas` créant un tas de taille donnée en argument, et rempli de tâches aux dates aléatoires. Il faut bien sûr utiliser `cree_tas` et `inser`.

Vérifier que la ligne suivante renvoie `true` :

```
verif (rand_tas 20);;
```

Question 5 : Suppression

Quand on traite la première tâche, qui est donc la racine du tas ou encore l'élément en position 0 du vecteur d, il faut la retirer du tas. La solution la plus simple consiste à remplacer la racine `d.(0)` par le dernier élément `d.(n-1)`, et à décrémenter n en $n-1$ mais il faut encore une fois corriger le tas car la racine n'est plus forcément inférieure à ses fils. Comment faire ça en $O(\log n)$ opérations, et proprement ?

Écrire une fonction `pop` de type `tas -> int` qui renvoie la date de la première tâche, et qui l'enlève du tas. S'aider éventuellement d'une fonction récursive `corrige_down`, de principe assez similaire à `corrige_up`, ou utiliser une correction itérative.

Vérifier que la commande suivante s'execute en moins d'une seconde et renvoie `true` :

```
let t = rand_tas 10000 in
for x=1 to 9000 do let yo = pop t in () done;
verif t;;
```

Les tas, ca va vite `:-)`

Question 6 : Tri par tas

Sachant ce qu'on peut faire avec un tas, imaginer un algorithme permettant de trier un vecteur (par ordre croissant ou décroissant, à vous de voir).

Encore mieuxi implémenter cet algorithme pour qu'il fonctionne *en place* (en travaillant uniquement sur le vecteur donné en argument, sans en faire de copie et sans créer de vecteur ou de liste).

Question 7 : Piles FIFO

Un cas plus trivial des files de priorités est la pile FIFO (first in, first out). C'est une file de priorité où toute nouvelle tache qui arrive sera traitée après toutes les tâches déjà stockées. Il est donc inutile de parler de date d'exécution : seul l'ordre d'arrivée compte. La file contient donc plutôt les identifiants (ou numéros) des tâches.

Implémenter un type `fifo` et coder les fonctions suivantes, sachant que la complexité de chacune (sauf la première) devra être $O(1)$:

- `cree_fifo n` crée une fifo vide mais capable de stocker n tâches.
- `vide f` renvoie un booléen qui vaut `true` ssi la fifo f est vide.
- `add_fifo f i` ne renvoit rien (`unit`) mais ajoute la tâche i à la fifo f, et échoue avec un message d'erreur si la fifo est pleine.
- `pop_fifo f` renvoit la première tâche (celle qui a été ajoutée avant toutes les autres) et la supprime de la fifo. Elle doit aussi échouer avec un message d'erreur si la fifo est vide.

Créer une fifo `f` de taille max égale à 3:

```
let f = cree_fifo 3;;
```

puis vérifier que les commandes suivantes :

```
add_fifo f 4; add_fifo f 1; add_fifo f 46;;
pop_fifo f;;
pop_fifo f;;
add_fifo f 5; add_fifo f 7;;
add_fifo f 1;;
```

renvoient quelquechose comme :

```
#- : unit = ()
#- : int = 4
#- : int = 1
#- : unit = ()
#Exception non rattrape: Failure "FIFO pleine !!!"
```

Sinon, c'est qu'il y a un bug...

Question 8 : Graphes

Formellement, un graphe non orienté est un couple (V, E) où V est un ensemble de sommets V et E un ensemble d'arêtes inclus dans $V \times V$. Ici, on va les implémenter simplement, en supposant qu'on a N sommets identifiés par les entiers $[0, \cdots, N-1]$. Un graphe sera simplement un vecteur de listes d'entiers g, tel que `g.(i)`, appelé la *liste d'adjacence* de i, soit la liste des voisins (connectés par une arête) du sommet i.

Encore une fois pour fixer les choses, le type `graphe` correspond à :

```
type graphe == int list vect;;
```

Coder une fonction de type `'a list vect -> int` permettant de compter le nombre d'arêtes d'un graphe (attention, chaque arête (i, j) est présente à la fois dans la liste d'adjacence de i et dans celle de j).

Coder une fonction `rand_graphe` telle que `rand_graphe n m` renvoie un graphe aléatoire de n sommets et de m arêtes (si des arêtes sont dupliquées ce n'est pas grave). Attention, lors de la création d'une arête (i, j) il faut l'ajouter à la fois à la liste d'adjacence de i et à celle de j.

NB : pour la question qui suit, je suis tout disposé à vous faire des dessins au tableau pour aider la compréhension.

Question 9 : Plus court chemin

Un graphe peut représenter un réseau réel. Par exemple, le réseau des routes et des carrefours en france permet à mappy de calculer les meilleurs itinéraires entre deux points. On va s'intéresser de plus près au problème (plus simple) du plus court chemin dans un graphe : Étant donnés deux sommets a et b, comment évaluer le nombre d'arêtes minimal qu'il faut emprunter pour aller de a à b? Ce nombre est appelé la *distance* de a à b.

La solution est le *parcours en largeur* : partant d'un point de départ a, on va d'abord parcourir tous les voisins de a, donc ceux situés à une distance 1, puis tous ceux à une distance 2 (les voisins des voisins de a qui ne sont ni a ni des voisins de a), puis ceux à distance 3, etc... Dès qu'on arrive sur b, on saura à quelle distance de a il se trouve.

Pour faire cela, on utilisera un vecteur `dist` tel que `dist.(i)` soit la distance du sommet i au point de départ, et une pile FIFO contenant tous les sommets que l'on va devoir visiter, dans l'ordre.

Coder la fonction `dist` de type `graphe -> int -> int` donnant, pour le sommet donné en argument, la distance maximale existante avec un autre sommet du graphe.

Quelle est sa complexité ?

Essayer ca :

```
dist (rand_graphe 1000 2000) 0;;
dist (rand_graphe 10000 20000) 0;;
```

et remarquer que la distance maximale, dans un graphe aléatoire, n'augmente pas très vite avec la taille du graphe.. C'est pour le même genre de raison que n'importe qui sur terre peut connaître n'importe qui en passant par un nombre finalement assez faible d'intermédiaires.

Question 10 : Graphes valués : algorithme de Dijkstra

S'inspirer vaguement de l'algorithme développé à la question précédente pour faire la même chose sur un graphe valué (chaque arête a une certaine longueur). Le type d'un tel graphe serait plutôt `(int*int) list vect` puisque la liste d'adjacence d'un sommet i doit contenir les voisins de i mais aussi la longueur des arêtes entre ces voisins et i. Le plus court chemin n'est cette fois pas forcément celui qui emprunte le moins d'arêtes, mais celui qui totalise la plus petite longueur. On suppose également que toutes les longueurs sont positives ou nulles.

Implémenter un algorithme capable de calculer cette plus courte longueur de chemin entre deux sommets (utiliser les tas). Quelle est sa complexité ?