Tas et Tas binomiaux

Tas

Cette section est dédiée à la gestion des files de priorité. Un certain nombre de tâches arrivent dans un tampon avec une certaine priorité (plus la tâche est urgente, plus cette entier est petit), l'objectif est d'ordonner efficacement ces tâches en fonction de leur priorité. Cela sera le cas grace à la notion de tas. Mais avant de décrire les tas et leur manipulation, quelques définitions préparatoires.

Un *arbre binaire* est construit récursivement de la manière suivante :

- Un noeud y est un arbre binaire.
- si T_1 et T_2 sont deux arbres binaires et y un noeud, alors $y(T_1, T_2)$ est un arbre binaire.

Le noeud y est appelé la racine de l'arbre. la racine de T_1 est appelé le fils gauche de y et la racine de T_2 le fils droit de y. Le noeud y est le père de la racine de T_1 et de T_2. La valeur y peut être choisi dans n'importe quel ensemble arbitaire (entier, réel, mots,...). C'est le contenu du noeud.

Un arbre binaire est une structure combinatoire qui joue un rôle fondamentale en structure de données. Ce n'est sans doute pas la première fois que vous en voyez et sans doute pas la dernière !

Bien sûr, un arbre peut être représenté graphiquement et ressemble un peu à un arbre !

Exercice : représenter l'arbre $x(z, y(a, b))$.

On définie recursivement la notion de niveau de la manière suivante : Dans un arbre, la racine est au niveau 0. Les noeuds de niveau h sont les fils des noeuds de niveaux $n - 1$.

Remarque : Par induction, on montre que l'on n'a pas plus de 2^h noeuds au niveau h.

Un arbre binaire *presque complet* est un arbre binaire dont tous les niveaux sont saturés, sauf éventuellement son niveau le plus haut qui est rempli en partant de la gauche et jusqu'à un certain point.

Un *tas (binaire)* est un arbre binaire presque complet dont les noeuds sont des entiers et tel que si x a un père (c'est à dire si x n'est pas la racine), le père de x est plus petit que x.

Dans certain ouvrage, les tas sont appelés tax-min.

Nous allons utiliser un tableau pour manipuler des arbres binaires presque complets. Chaque noeud de l'arbre sera stocké dans une case du tableau T. La racine de l'arbre est stockée dans $T[1]$, et le fils gauche (resp. droit) du noeud stocké dans la case j est stocké dans la case $2j$ (resp. la case $2j + 1$. Noter que si l'on classe les noeud de l'arbre, en les énumérant de haut vers le bas et de gauche vers la droite, le j noeud dans cet ordre est exactement le noeud stocké dans la case j du tableau.

On va définir les 3 primitives suivantes afin de se déplacer dans le tableau comme dans un arbre :

Algorithme 1 : Père

Entrées : i un indice du tableau.
Sorties : l'indice du noeud père.

```
1 Père(i)
2 retourner ⌊i/2⌋
```

Algorithme 2 : Gauche

Entrées : i un indice du tableau.
Sorties : l'indice du fils gauche.

```
1 Gauche(i)
2 retourner 2i
```

Algorithme 3 : Droit

Entrées : i un indice du tableau.
Sorties : l'indice du fils droit.

```
1 Droit(i)
2 retourner 2i + 1
```

La question maintenant est d'ordonner un arbre binaire presque complet afin d'en faire un tas. Ou en d'autres termes de passer d'un tableau quelconque en un tas.
Entasser est l'algorithme clé pour construire des tas, qui prend en entrée un tableau T et un indice i. Le tableau T est tel que les arbres binaires enracinés en $Gauche(i)$ et $Droit(i)$ sont des tas (mais $T[i]$ peut être plus grand que ses descendants, violant ainsi la propriété de tas).
Le rôle de *Entasser* est de faire en sorte que le sous-arbre enraciné en i devienne un tas. Pour cela, il doit faire «descendre» la valeur de $T[i]$ dans l'arbre. On note $nom(T)$ le nombre d'élément dans le tas (qui peut être strictement plus petit que la taille du tableau). Plus explicitement :

Algorithme 4 : Entasser

Entrées : T un tableau et i un indice.
Sorties : Rien, mais le tableau est modifié

```
1  Entasser(T, i)
2  g := Gauche(i) ;
3  d := Droit(i) ;
4  si g ≤ nom[T] et T[g] > T[i] alors
5  |  min := g;
6  sinon
7  |  min := i;
8  si d ≤ nom[T] et T[d] > T[i] alors
9  |  min := d;
10 si min ≠ i alors
11 |  échanger T[i] et T[min];
12 |  Entasser(T, min);
```

Le temps d'exécution de la procédure Entasser suivi la récurrence :

$$T(n) \leq T(2n/3) + \Theta(1).$$

Donc, $T(n) = O(\ln(n))$.
Nous sommes prêts pour définir les opérations que l'on veut sur notre file de priorité. Nous allons considérer les opérations suivantes :
$Insérer(T, k)$ qui permettra d'insèrer la clé x dans le tableau T.
$Minimum(T)$ qui retournera l'élément de T minimale.
$ExtraireMin(T)$ qui supprimera et retournera l'élément de T minimale.
$ADiminuerClé(T, i, k)$ qui décroissera la valeur de l'élément d'indice i pour lui donner la nouvelle valeur k.

Notez que l'on ne va pas décrire les algorithmes dans cette ordre, car $Insérer(T, k)$ utilise $DiminuerClé(T, i, k)$ dans sa procédure.

Algorithme 5 : Minimum

Entrées : T un tableau.
Sorties : Le minimum.

```
1 Minimum(T)
2 return T[1]
```

Algorithme 6 : ExtraireMin

Entrées : T un tableau.
Sorties : le minimum et supprime le min de la structure de données T

```
1 ExtraireMin(T)
2 si nom[T] < 1 alors
3 |  erreur file vide
4 min := T[1];
5 T[1] := T[nom[T]];
6 nom[T] := nom[T] − 1;
7 Entasser(T, 1);
8 retourner min;
```

Le temps d'exécution de ExtraireMin est en $O(\ln(n))$.

Algorithme 7 : DiminuerClé

Entrées : T un tableau, i un indice dans T, k la nouvelle valeur de $T[i]$.
Sorties : La structure de données T est mise à jour

```
1 DiminuerClé(T, i, k)
2 si k > T[i] alors
3 |  erreur nouvelle valeur plus grande que valeur actuelle
4 T[i] := k;
5 i > 1 et T[Père(i)] > T[i] permuter T[i] avec T[Père(i)];
6 i := Père(i);
```

Le temps d'exécution de DiminuerClé sur un tas a n elements est en $O(\ln(n))$.

Algorithme 8 : $Insérer$

Entrées : T un tableau, x la nouvelle valeur à insérer.
Sorties : La structure de données T est mise à jour

```
1 Insérer(T, k)
2 nom[T] := nom[T] + 1;
3 T[nom[T]] := ∞;
4 DiminuerClé(T, nom[T], k);
```

Le temps d'exécution de Insérer sur un tas à n éléments est en $O(\ln(n))$.
En résumé, un tas permet de faire toutes les opérations ci-dessus de file de priorité sur un ensemble de taille n en un temps $O(\ln(n))$.

1. Tas binomiaux

Les tas binomiaux sont des structures de données connues sous le nom de tas fusionnables, qui supportent les cinq opérations suivantes :

CréerTas() crée et retourne un nouveau tas sans élément.

Insérer(T, x) insère dans le tas T un noeud x, dont le champ clé a déjà été rempli.

Minimum(T) retourne un pointeur sur le noeud dont la clé est minimale dans le tas T.

ExtraireMin(T) supprime du tas T le noeud dont la clé est minimale et retourne un pointeur sur ce noeud.

Union(T_1, T_2) crée et retourne un nouveau tas qui contient tous les noeuds des tas T_1 et T_2. Les tas T_1 et T_2 sont « détruits » par cette opération.

De plus, les tas binomiaux supportent aussi les deux opérations suivantes :

DiminuerClé(T,x,k) affecte au noud x du tas T la nouvelle valeur de cle k, qui est censee etre inferieure ou egale a la valeur courante de la cle.

Supprimer(T, x) supprime le noeud x du tas T.
Dans le prochain cours, nous aborderons les tas de Fibonacci, voilà un tableau décrivant les temps d'executions des opérations élémentaires dans ces différentes structures de données :

Procédure	Tas (binaire)	Tas binomial	Tas de Fibonacci
CréerTas	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Insérer	$\Theta(\ln(n))$	$O(\ln(n))$	$\Theta(1)$
Minimum	$\Theta(1)$	$O(\ln(n))$	$\Theta(1)$
ExtraireMin	$\Theta(\ln(n))$	$\Theta(\ln(n))$	$O(\ln(n))$
Union	$\Theta(n)$	$O(\ln(n))$	$\Theta(1)$
DiminuerClé	$\Theta(\ln(n))$	$\Theta(\ln(n))$	$\Theta(1)$
Supprimer	$\Theta(\ln(n))$	$\Theta(\ln(n))$	$O(\ln(n))$

Un *arbre binomial* B_k est un arbre défini récursivement comme suit : L'arbre binomial B_0 est constitué d'un noeud unique. L'arbre binomial B_k est constitué de deux arbres binomiaux B_{k-1} qui sont reliés entre eux : la racine de l'un est le fils le plus à gauche de la racine de l'autre.

Lemme 1. *(Propriétés des arbres binomiaux) Pour un arbre binomial B_k,*

(1) Il a 2^k noeuds.

(2) La hauteur de l'arbre est k.

(3) Il existe exactement $\binom{i}{k}$ noeuds à la profondeur i pour $i = 0, 1, \ldots, k$.

(4) La racine est de degré k, qui est superieur a celui de tout autre noeud ; par ailleurs, si les enfants de la racine sont numerotes de la gauche vers la droite par $k-1, k-2, \ldots, 0$, l'enfant i est la racine d'un sous-arbre B_i.

(5) Le degre maximal d'un noeud quelconque dans un arbre binomial a n noeuds est $\log(n)$.

Un tas binomial *T est un ensemble d'arbres binomiaux qui satisfait aux propriétés des tas binomiaux :*
1) Chaque arbre binomial de T obéit à la propriété de tas min : la clé d'un noeud est supérieure ou égale à la clé de son père. On dit d'un tel arbre qu'il est trié en tas min.
2) Quel que soit l'entier k positif, il existe dans T un arbre binomial au plus dont la racine a le degré k.
De plus, un tas binomial contient au plus $\lfloor \log(n) \rfloor + 1$ arbres binomiaux car la taille des arbres forment l'écriture en bases 2 de n.
On va maintenant décrire les algorithmes de manipulation des tas binomiaux.

CréerTasBinomial se contente d'allouer puis de retourner un objet T, où tête$[T] = NIL$
La procedure MinimunTasBinomial retourne un pointeur sur le noeud de clé minimale dans un tas binomial T.

Algorithme 9 : MinimunTasBinomial

Entrées : *T un tas.*
Sorties : *y un pointeur sur le noeud de clé minimale dans T*

```
1  MinimunTasBinomial(T)
2  y := NIL;
3  x := tête[T];
4  min := ∞;
5  tant que x ≠ NIL faire
6      si clé[x] < min alors
7          min := clé[x];
8          y := x;
9      x := frère[x];
10 retourner y;
```

Union de deux tas binomiaux

L'opération qui consiste à unir deux tas binomiaux sert de sous-programme pour la plupart des opérations restant à étudier. La procédure UNION-TAS-BINOMIAUX relie de façon répétée des arbres binomiaux dont les racines ont le même degré.

La procédure suivante relie l'arbre B_{k-1} de racine y à l'arbre B_{k-1} de racine z.

Algorithme 10 : LienBinomial

Entrées : *Deux pointeurs sur les racines de deux arbres.*
Sorties : *Rien*

```
1  LienBinomial(y, z)
2  père[y] := z;
3  frère[y] := fils[z];
4  fils[z] := y;
5  degré[z] := degré[z] + 1;
```

Algorithme 11 : UnionTasBinomiaux

Entrées : *Deux pointeurs sur les racines de deux arbres.*
Sorties : *Rien*

```
1  UnionTasBinomiaux(T_1, T_2)
2  T := CréerTasBinomial();
3  tête[T] := Fusionner les Tas Binomiaux T_1 et T_2 en faisant une liste triée par
     degré de la racine des arbres dans les deux tas.;
4  libère objets T_1 et T_2, mais pas les listes vers lesquelles ils pointent;
5  si tête[T] = NIL alors
6      retourner T;
7  avant_x := NIL;
8  x := tête[T];
9  après_x := frère[x];
10 tant que après_x ≠ NIL faire
11     si (degré[x] ≠ degré[après_x]) ou (frère[après_x] ≠ NIL et
         degré[frère[après_x]] = degré[x]) alors
12         avant_x := x;
13         x := après_x;
14     sinon
15         si clé[x] ≤ clé[après_x] alors
16             frère[x] := frère[après_x];
17             LIEN-BINOMIAL(après_x, x)
18         sinon
19             si avant_x = NIL alors
20                 tête[T] := après_x;
21             sinon
22                 frère[avant_x] := après_x;
23             LIEN-BINOMIAL(x, après_x);
24         x := après_x;
25     après_x := frère[x];
26 retourner T
```

Le temps d'exécution de la fusion des tas binomiaux T_1 et T_2 est en $O(\ln(n))$ car chaque tas à un nombre logarithmique de racines par rapport à sa taille. Chaque itération de la boucle tant que requiert un temps en $O(1)$ et il existe au plus $\lfloor \ln(n_1) \rfloor + \lfloor \ln(n_2) \rfloor + 2$ itérations car chacune d'elle, soit fait descendre les pointeurs d'une position dans la liste des racines de T, soit supprime une racine de la liste des racines. Le temps total est donc $O(\ln(n))$.

Algorithme 12 : *InsérerTasBinomial*

Entrées : *T un tas, x la nouvelle valeur à insérer.*
Sorties : *La structure de données T est mise à jour*

```
1  InsérerTasBinomial(T, x)
2  T' := CréerTasBinomial();
3  père[x] := NIL;
4  frère[x] := NIL;
5  degré[x] := 0;
6  tête[T'] := x;
7  T := UnionTasBinomiaux(T, T');
```

La complexité est clairement en $O(\ln(n))$ comme celle de UnionTasBinomiaux.

Algorithme 13 : ExtraireMinTasBinomiaux

Entrées : *T un tas*
Sorties : *La structure de données T est mise à jour*

```
1  ExtraireMinTasBinomiaux(T)
2  Trouver la racine x de cle minimale dans la liste des racines de T, et supprimer x
     de la liste;
3  T' := CréerTasBinomial();
4  Inverser l'ordre de la liste chainée des enfant de x, et faire pointer tête[T'] sur la
     tête de la liste résultante;
5  T := UnionTasBinomiaux(T, T');
6  retourner x;
```

Quand T contient n noeuds, rechercher le min dans les racines consomme un temps $O(\ln(n))$. Donc, ExtraireMinTasBinomiaux s'execute en $O(\ln(n))$.
La procédure DiminuerCléTasBinomial est en fait similaire à celle des tas binaires :

Algorithme 14 : DiminuerCléTasBinomial

Entrées : *T un tas*
Sorties : *La structure de données T est mise à jour*

```
1  DiminuerCléTasBinomial(T, x, k)
2  si k > clé[x] alors
3  |  erreur « La nouvelle clé est plus grande que la clé courante »
4  ;
5  clé[x] := k;
6  y := x;
7  z := père[y];
8  tant que z ≠ NIL et clé[y] < clé[z] faire
9  |  permuter clé[y] avec clé[z]
10 y := z;
11 z := père[y];
```

La procédure DiminuerCléTasBinomial s'exécute en $O(\ln(n))$ car la profondeur maximale de x est en $O(\ln(n))$.
Enfin, pour supprimer x de T, il suffit de diminuer la clé jusqu'à $-\infty$ et d'extraire le minimum du tas.