Algorithmique et Programmation - TD 4

Arbres

2009-2010 : Semestre 1

Table des matières

1 Arbre Binaire de Recherche randomisé - Treap

On appelle *treap* une structure basées sur des noeuds représentés par une clé et une priorité. Les clés répondent à une structure d'arbre binaire de recherche, les priorités répondent à une structure de tas.

1.1 Déterminisation

Montrons par récurrence que cette structure détermine l'arbre de manière unique.
Il n'y a qu'une structure à un élément, ce qui garantit l'initialisation.
Pour créer une telle structure comportant n éléments, on met le noeud avec la plus petite propriété à la racine, pour respecter la structure de tas, et on divise les éléments restants en fonction de leur clé : ceux qui ont une clé plus petit que la racine iront dans son sous-arbre gauche, les autres dans son sous-arbre droit. Ces sous-arbres comportant moins de n éléments, ils sont fixés de manière unique par hypothèse de récurrence.

1.2 Insertion par liste triée

Si on insère dans un ABR les éléments dans l'ordre des priorités croissantes, on le complètera de la racine vers les feuilles. L'insertion par ABR garantira la structure d'ABR, tandis que la propriété de tri de la liste d'éléments initiale garantit que les noeuds ajoutés ont tous une priorité plus grande que celle des noeuds ajoutés avant eux qui ont une profondeur moindre, donc en particulier leur père : la structure de tas est respectée.

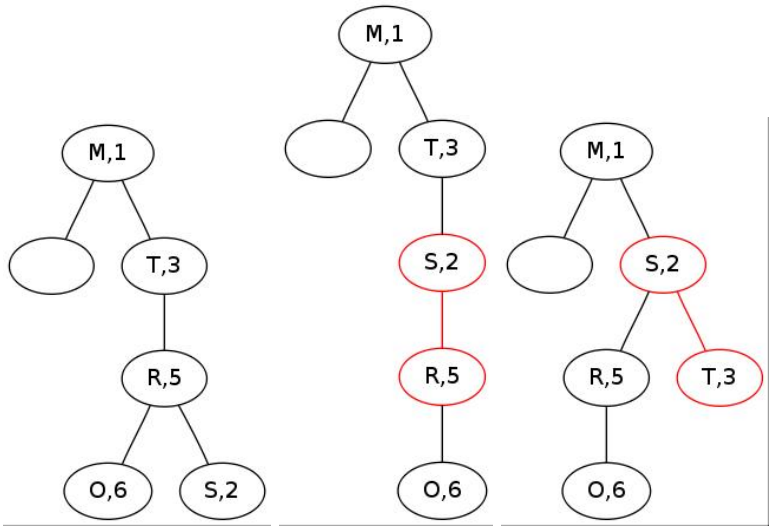
1.3 Opérations élémentaire

1.3.1 Recherche

La recherche s'effectue comme dans un ABR, en fonction de la profondeur de la feuille si elle est présente, ou de celle de son prédecesseur ou successeur si elle ne l'est pas.

1.3.2 Insertion

Elle se fait comme dans un ABR, puis on corrige l'arbre obtenu comme on le fait pour les tas : on remonte l'élément si sa priorité pose problème, en effectuant des rotations qui ne modifient pas la structure d'ABR. Voici un exemple :



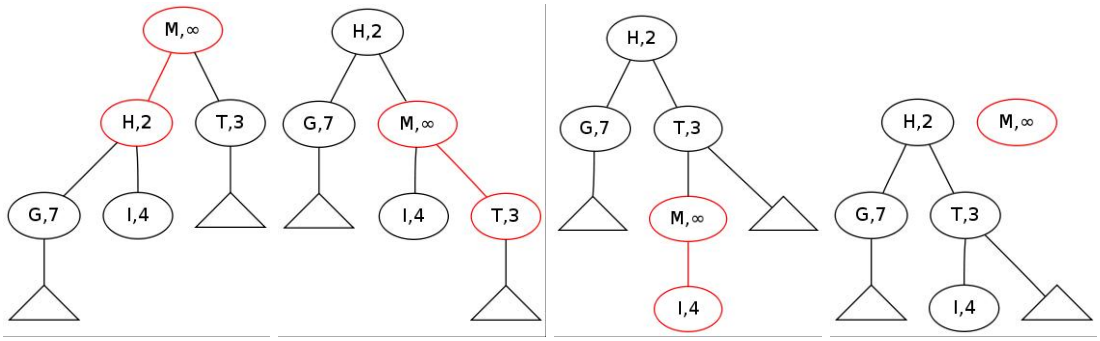
On a un temps d'éxecution proportionnel à la hauteur de l'arbre dans le pire des cas.

1.3.3 Découper par rapport à une clé

Il suffit d'ajouter la nouvelle clé avec la priorité $-\infty$: elle sera remontée à la racine, et séparera l'arbre en deux sous-arbres : la propriété d'ABR garantira que les sous-arbres contiendront tous des clés d'une part plus petite d'autre part plus grande que la clé de comparaison. On effectue donc des remontées par rotations, pour une complexité de l'ordre de la hauteur de l'arbre.

1.3.4 Suppression

Il s'agit encore une fois de donner à la clé à supprimer une priorité fixée, cette fois ci $+\infty$. On veillera ensuite à la descendre par rotations successives pour conserver la propriété d'ABR, avec son fils de plus petite priorité pour respecter la structure de tas. On effectue autant de descentes que la hauteur de l'arbre.



1.3.5 Fusion

On ne considère que le cas de la fusion d'un treap avec un second dont tous les éléments ont une clé supérieure au premier. Il s'agit de les mettre tous les deux en sous-arbres d'un arbre de racine ayant une clé quelconque mais une priorité de $+\infty$, que l'on corrige ensuite en faisant descendre ce noeud avant de le supprimer. La descente a une complexité de l'ordre de la hauteur de l'arbre.

1.4 Espérance de hauteur d'un treap randomisé

On considère que les priorités sont des variables aléatoires uniformément distribuées. On note x_k le noeud qui a la k^{ime} plus petite clé, et la variable indicatrice A_k^i qui vaut 1 si x_i est un ancètre propre de x_k. La hauteur d'un noeud sera alors :

$$h_{x_k} = \sum_{i=1}^{n} A_i^k \Longrightarrow [h_{x_k}] = \sum_{i=1}^{n} [A_i^k]$$

Somme qui se calcule aisément car, notant $p_{i,k}$ la probabilité pour que x_i soit ancètre propre de x_k, A_i vaut 1 avec la probabilité $p_{i,k}$ et 0 sinon. Son espérance est donc $p_{i,k}$ et il vient :

$$[h_{x_k}] = \sum_{i=1}^{n} p_{i,k}$$

Pour préciser cette somme, nous travaillerons sur l'ensemble $X(i,k) = \{x_i, x_{i+1}, \ldots, x_k\}$ si $i < k$, $X(i,k) = \{x_k, x_{k+1,\ldots,x_i}\}$ sinon. Montrons par récurrence que

Si $i \neq k$, alors x_i est un ancetre propre de x_k si et seulement si il a la plus petite priorité au sein de $X(i,k)$

Cette proposition est trivialement vraie si x_k et x_i sont les deux seuls éléments distincts.
Dans le cas général, si x_i est racine, il a la plus petite priorité de tous les noeuds possible et est l'ancètre de tous, donc en particulier de x_k. Inversement, si x_k est racine, ces deux propriétés sont impossibles : x_k n'a pas d'ancetre et sa priorité est moindre par rapport à x_i.

Si la racine de l'arbre est alors x_j, avec $j \neq i$ et $j \neq k$,
- Soit les noeuds étudiés sont dans le meme sous-arbre, à droite ou à gauche de x_j, et l'hypothèse de récurrence appliquée à ce sous-arbre démontre le résultat.
- Soit ils sont chacun dans un sous arbre séparé, x_i n'étant donc pas ancetre de x_k, et alors $x_j \in X(i,k)$ car $i < j < k$ ou $k < j < i$. Etant donné que c'est la racine de l'arbre, elle a la plus petite priorité dans tout l'arbre et en particulier dans $X(i,j)$, ce qui valide le résultat.

On obtient donc que la probabilité
$p_{i,k} = 0$ si $i = k$ ou $\frac{1}{|k-i|+1}$ sinon, $|k-i|+1$ étant le nombre d'éléments de $X_{i,k}$. D'où

$$[h_{x_k}] = \sum_{i=1}^{n} p_{i,k} = \sum_{i=1}^{k-1} \frac{1}{k-i+1} + \sum_{i=k+1}^{n} \frac{1}{i-k+1}$$

En effectuant le changement de variables $j = k-i+1$ et $l = i-k+1$,

$$[h_{x_k}] = \sum_{j=2}^{k-1} \frac{1}{j} + \sum_{l=2}^{n-k+1} \frac{1}{l}$$

$$[h_{x_k}] = H_k - 1 + H_{n-k+1} - 1$$

Où H_k est le k^{ime} terme de la série harmonique. D'après :

$$\sum_{i=1}^{n} \frac{1}{i} < 1 + \int_1^n \frac{dx}{x} = 1 + [ln(x)]_1^n = 1 + ln(n)$$

On a la majoration :

$$[h_{x_k}] = H_k - 1 + H_{n-k+1} - 1 \leq 2 * H_n - 2 \leq 2 * ln(n)$$

Car la série harmonique est croissante.

On a ainsi estimé la hauteur moyenne des noeuds, elle est de la forme $log(n)$, ce qui sera donc le cout de toutes les opérations en nombre de rotations.
On peut montrer que cette espérance est en réalité une valeur réalisée avec une très grande probabilité.

1.5 Tri par treap

On peut trier les éléments d'une liste en les insérant dans un treap avec une priorité aléatoire pour obtenir un Arbre Binaire de Recherche randomisé comme celui étudié ci-dessus. Pour avoir la liste triée, il suffit alors de le parcourir par le parcours usuel des ABR, ce qui permet de retourner la liste triée.
L'analogie avec le tri *Quicksort* est immédiate : celui-ci consistait à diviser récursivement la liste par rapport à un pivot. La structure d'ABR nous garantit que c'est ici le cas, les noeuds sont divisés en deux sous arbres, de clés réparties par rapport à la racine. La propriété d'équilibre permise par la randomisation permet d'éviter le pire des cas de *Quicksort* et d'optimiser sa complexité pour l'approcher de $O(n*log(n))$

2 Splay Trees

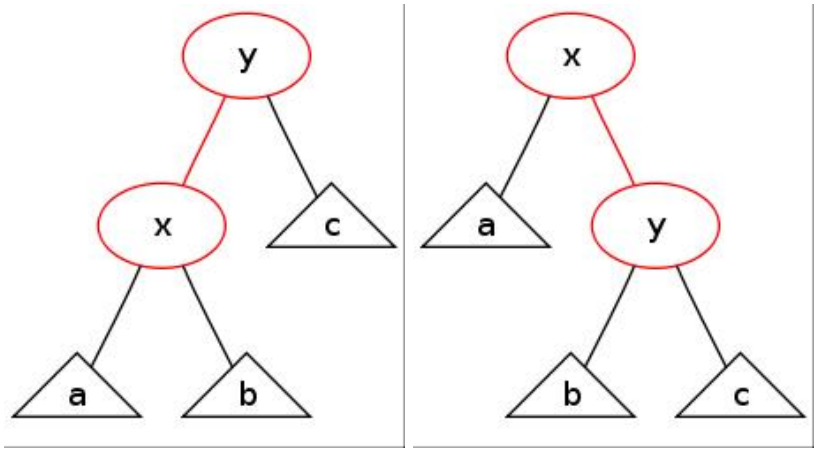
2.1 Analyse amortie

On rappelle la définition d'un potentiel ϕ en analyse amortie :
$\phi_0 = 0$ et $\phi_i > 0$ pour tout état i. Le coût amorti sera $ca_i = c_i + \phi_i - \phi_{i-1}$ où c_i est le cout total de l'état i, et vérifiera :
cout total $\leq$ cout total amorti
On travaillera ici sur des Arbres Binaires de Recherches, qui seront automatiquement équilibrés par des rotations simples ou doubles implémentées lors de l'insertion, de la suppression ou de la recherche d'un élément (on l'amène à la racine, ou son prédécesseur ou son successeur s'il n'est pas dans l'arbre). Pour

les calculs, nous définirons les notations :
$|v|$ la taille du sous arbre de racine v,
$rang(v) = log(|v|)$,
Et le potentiel $\phi = \sum_v rang(v)$

2.2 Cout amorti des rotations

2.2.1 Rotation simple



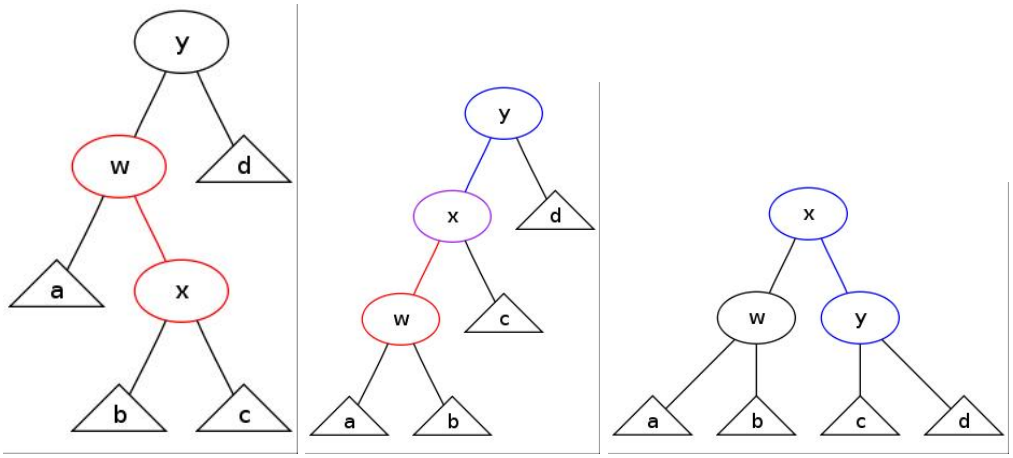
En notant avec un prime l'état final, cette opération coute :

$$Ca = 1 + \phi' - \phi = 1 + r'(x) + r'(y) - r'(x) - r'(y)$$

Avec 1 pour la rotation effective. En effet, les seuls rangs qui changent sont ceux de x et y.
Or $r'(y) < r(y)$ et $r'(x) > r(x)$ d'où

$$Ca = 1 + \phi' - phi \leq 1 + r'(x) - r(x) \leq 1 + 3 * r'(x) - 3 * r(x)$$

2.2.2 Double rotation en "zigzag"



Le cout amorti d'une telle opération est :

$$Ca = 2 + \phi' - \phi = 2 + r'(x) + r'(y) + r'(w) - r(x) - r(y) - r(w)$$

Or $r'(x) = r(y)$ d'où

$$Ca = 2 + r'(w) + r'(y) - r(x) - r(w)$$

De plus $r(x) \leq r(w)$ donc

$$Ca \leq 2 + r'(y) + r'(w) - 2 * r(x) = 2 + log(|y|') + log(|w|') - 2 * r(x)$$

Or on voit clairement que $|y|' + |w|' + 1 = |x|'$, et

$$log(|y|') + log(|w|') = 2 * (\frac{1}{2} * log(|y|') + \frac{1}{2} * log(|w|')) \leq 2 * (log(\frac{|y|' + |w|'}{2}))$$

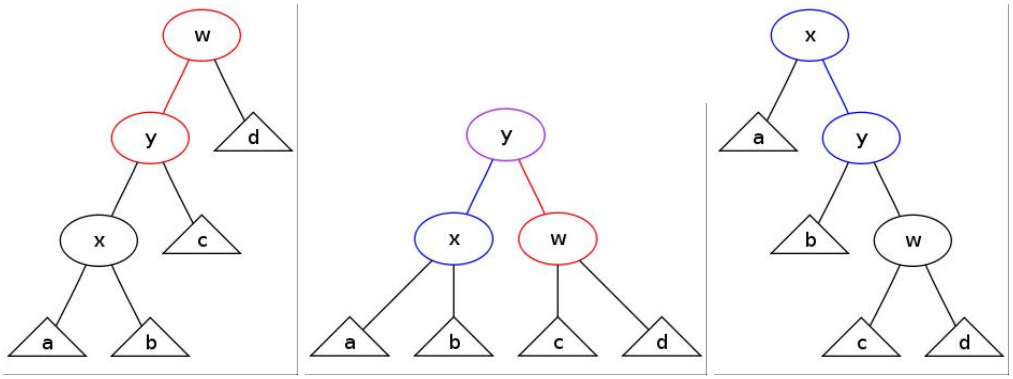
Par la concavité du logarithme en base 2,

$$log(|y|') + log(|w|') \leq 2 * (log(\frac{|x|' - 1}{2})) \leq 2 * log(|x|') - 2 * log(2) = 2 * log(|x|') - 2$$

D'où le résultat :

$$Ca \leq 2 + 2 * r'(x) - 2 - 2 * r(x) \leq 3 * r'(x) - 3 * r(x)$$

2.2.3 Double rotation en "roller-coaster"



Le cout amorti d'une telle opération est :

$$Ca = 2 + \phi' - \phi = 2 + r'(x) + r'(y) + r'(w) - r(x) - r(y) - r(w)$$

Or $r'(x) = r(w)$, donc