Université Pierre et Marie Curie **Master Informatique**
14 Novembre 2006 **MI015-ALGAV**

Devoir sur table

Durée : 2h.
Seul document autorisé : une double feuille A4 « antisèche »
Le barème donné, sur 24 points, est indicatif.

1 Arbres autoadaptatifs [6 points]

Cet exercice traite des *arbres autoadaptatifs*, qui sont des arbres binaires de recherche munis d'une opération d'*adaptation*, à base de rotations, qui permet de "remonter" un élément à la racine en gardant la propriété d'arbre de recherche.
Pour décrire les algorithmes demandés dans l'exercice, on utilisera les primitives sur les arbres binaires, auxquelles on pourra ajouter la fonction d'adaptation dont la spécification est la suivante
`adaptation` : *Element * ABR* $\rightarrow$ *ABR*
`adaptation(x, T)` *renvoie un ABR* T' *contenant le même ensemble d'éléments que* T *; si* x *est dans* T*, alors* x *est l'élément à la racine de* T' *; et sinon l'élément à la racine de* T' *est soit* x^- *soit* x^+*, où* x^- *est le plus grand élément de* T *inférieur à* x *et* x^+ *est le plus petit élément de* T *supérieur à* x*.*
Cette opération est réalisée en localisant, par une descente classique dans un ABR, x dans T (ou x^+ ou x^- lorsque x n'est pas dans T). Le nœud localisé est ensuite remonté vers la racine par une série de rotations. Les seules rotations autorisées sont des **doubles rotations**, droite-droite ou gauche-gauche (voir figure 1) ou gauche-droite ou droite-gauche (voir figure 2) ; dans le cas où le nœud localisé est à profondeur impaire, on doit effectuer, en plus des rotations doubles, une unique rotation simple.

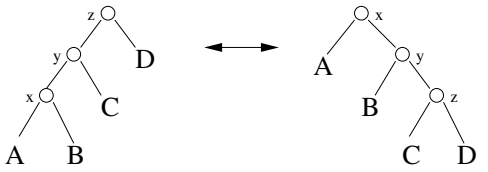


Figure 1 : Rotations droite–droite (–>) et gauche–gauche (<–)

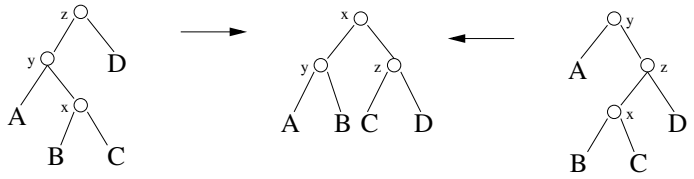
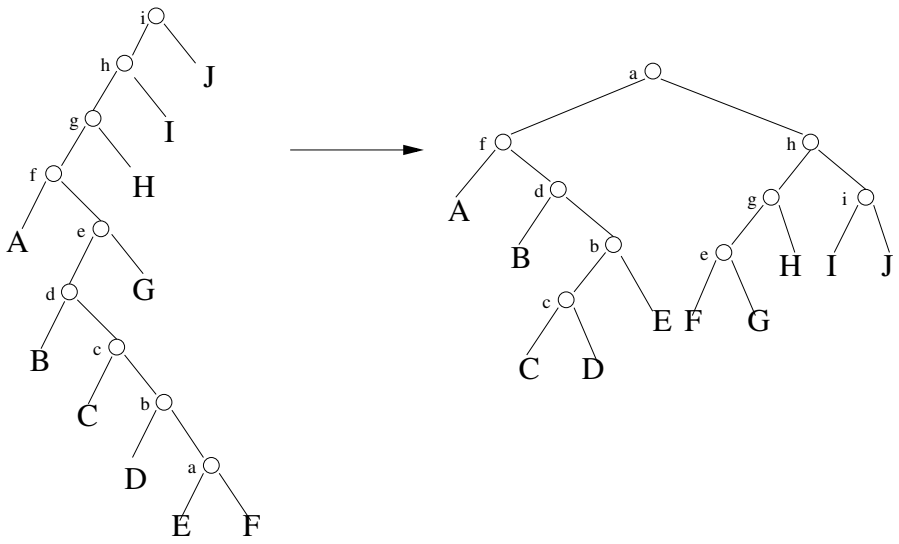


Figure 2 : Rotations gauche–droite (–>) et droite–gauche (<–)

La figure suivante montre l'arbre T' obtenu en adaptant l'arbre de gauche T par `adaptation(a,T)`.



1- Expliciter (par des dessins d'arbres) les différentes étapes (rotations) permettant de passer de l'arbre T à l'arbre T' dans la figure précédente.

2- Décrire l'algorithme `adaptation(x, T)`.

3- Décrire un algorithme d'ajout d'un élément x dans un arbre T, qui soit en temps constant après avoir adapté T.

4- Décrire un algorithme de suppression d'un élément x d'un arbre T, qui soit en temps constant après avoir réalisé les adaptations nécessaires.

2 Hachage chaîné dynamique [6 points]

Dans la méthode de hachage avec chaînage séparé, on dispose d'une table T de taille m et d'une fonction de hachage $h : \mathcal{C} \to [0..m-1]$, où $\mathcal{C}$ est un ensemble de n clés. On suppose h uniforme, c'est-à-dire que les images des clés sont équiréparties. T est une table de listes : la liste $T[i]$ contient tous les éléments dont la valeur de hachage vaut i.

1- Définir les algorithmes de recherche et d'ajout d'un élément x dans une table de taille m par cette méthode (on utilisera des primitives sur les listes, dont on décrira les spécifications).
Faire l'analyse de la complexité dans le pire des cas de ces algorithmes, en précisant quelles sont les opérations comptées dans cette analyse.
Montrer qu'en moyenne, le nombre de comparaisons entre éléments dans une recherche avec succès est de l'ordre de $\frac{n}{2m}$, lorsqu'il y a n éléments dans une table de taille m.

2- Pour diminuer le temps de recherche dans les listes, on décide de redimensionner la table lorsque le nombre d'éléments devient trop grand : à l'ajout d'un nouvel élément, si le nombre n d'éléments est supérieur à la taille m de la table, on réorganise les éléments sur une table de taille $2m + 1$, avant d'effectuer l'ajout.
Écrire la fonction de réorganisation, puis la nouvelle fonction d'ajout. Quelle est la complexité de la fonction d'ajout, en moyenne et dans le pire des cas ?
Étudier le coût amorti d'une suite de n opérations d'ajout et de recherche.

3 Coût amorti [6 points]

Cet exercice est issu du partiel de l'an dernier, dont le corrigé a été fait en TD.
Une file FIFO est une structure de données linéaire qui supporte deux opérations
– *ajouter(x,F)*, qui ajoute un élément à la file,
– *enlever(F)*, qui enlève de la file l'élément le plus anciennement présent.
On peut implanter une file F à l'aide de deux piles P_1 et P_2 (et les opérations classiques *empiler(x,P)* et *dépiler(P)*), de la façon suivante :
– *ajouter(x,F)* : *empiler(x,*P_1*)*
– *enlever(F)* : si P_2 est vide alors dépiler tous les éléments de P_1 et les empiler au fur et à mesure dans P_2. Puis (dans tous les cas) dépiler P_2.

On veut calculer le coût amorti des opérations *ajouter(x,F)* et *enlever(F)* en essayant différentes fonctions de potentiel. Le coût est mesuré en nombre d'opérations faites sur les piles.
essai 1 : la fonction de potentiel vaut deux fois le nombre d'éléments de la pile P_1,
essai 2 : la fonction de potentiel est égale au nombre d'éléments de P_1 moins le nombre d'éléments de P_2,
essai 3 : la fonction de potentiel est égale au nombre d'éléments de P_1 plus le nombre d'éléments de P_2.
Pour chacune des trois fonctions précédentes, donner le coût amorti des opérations *ajouter(x,F)* et *enlever(F)* et dire s'il est possible que le coût réel total soit supérieur au coût amorti total. Peut-on retenir ces trois fonctions ?

4 Questions de TD [6 points]

Exercice 1 : Manipulations de bits
En utilisant *uniquement* les opérateurs logiques de décalage, le `NON`, le `ET` et le `OU`, donner des algorithmes pour :
– positionner un bit donné à 1.
– positionner un bit donné à 0.
– effectuer une permutation circulaire des bits dans un sens et dans l'autre.

Exercice 2 : Transformation d'un arbre 2-3-4 en arbre bicolore
Rappeler le principe de l'algorithme de transformation d'un arbre 2-3-4 en arbre bicolore.
Montrer que cet algorithme produit bien un arbre bicolore et donner un encadrement de la hauteur de l'arbre bicolore en fonction de celle de l'arbre 2-3-4 dont il est issu.