Algorithmique — IN102
TD 3

16 décembre 2005

Exercice 1 *Clairement, il existe des arbres de hauteur h à $h+1$ éléments : il suffit pour cela que leurs nœuds internes aient au plus un fils non vide. On a alors un arbre isomorphe à une liste.*

Exercice 2 *Il existe des arbres de hauteur h contenant $2^{h+1}-1$ éléments. Il 'sagit des arbres dont tous les niveaux de profondeur $\leq h$ sont complètement remplis. En effet, on a alors un nœud de profondeur 0, 2 nœuds de profondeur 1, 4 nœuds de profondeur 2, ..., soit 2^p nœuds de profondeur p, ce qui donne un nombre total de nœuds égal à :*

$$2^0 + 2^1 + 2^2 + \ldots + 2^h = \frac{2^{h+1}-1}{2-1} = 2^{h+1} - 1$$

Exercice 3 *Considérons un arbre à n éléments dont les «niveaux» sont remplis au maximum. La hauteur d'un tel arbre est alors de l'ordre de $\lfloor \log_2(n) \rfloor$.*

Plus précisément, puisque d'après la question précédente $n \geq 2^{h+1} - 1$, on a $h \geq \log(n+1) - 1$, d'où $h \geq \lceil \log(n+1) - 1 \rceil$ et donc

$$h \geq \lfloor \log(n+1) \rfloor$$

Exercice 4 *Cette propriété se démontre par récurrence. Notons P_n la propriété «*tout arbre binaire à au plus n éléments dont les nœuds ont soit deux fils, soit aucun, a un nombre de nœuds internes égal au nombre de feuilles moins un*».*

- *P_1 est clairement vraie : l'unique arbre à un élément est composé d'une feuille et n'a aucun nœud interne.*
- *Supposons P_i vraie pour tout $i < n$. Un arbre à n éléments se compose d'une racine et de deux sous-arbres (droit et gauche), à respectivement p et q éléments ($1 + p + q = n$). Les propriétés P_q et P_p sont vraies par hypothèse de récurrence : la somme des nœuds internes des sous-arbres gauche et droit est donc inférieure de 2 au nombre total de feuilles. La racine est elle aussi un nœud interne, ce qui ramène à 1 cette différence. On conclut que P_n est vraie pour tout n.*

Exercice 5 *La propriété définissant un arbre binaire s'imposant à chacun des nœuds de l'arbre, elle définit par la même occason chaque sous-arbre comme un arbre binaire de recherche à part entière.*

Exercice 6 *Un parcours infixé imprime le sous-arbre gauche, puis imprime la racine, puis le sous-arbre droit. D'après la propriété définissant les arbres binaires de recherche on imprime d'abord tout ce qui est plus petit que la racine, puis la racine elle-même,, puis tout ce qui est plus grand. Cette procédure étant appliquée à chaque niveau de l'arbre, les clés seront imprimées dans l'ordre croissant.*

Un algorithme de tri consiste donc à insérer tous les éléments à trier dans un arbre binaire de recherche initialement vide, puis à les extraire par un parcours infixe.

Exercice 7 *La recherche d'une clé dans un arbre binaire de recherche imite la recherche dichotomique : il suffit de comparer la clé recherchée avec celle du nœud courant : si ce n'est pas la clé recherchée, il faut poursuivre la recherche récursivement dans le sous-arbre gauche ou droit, selon le résultat de cette comparaison.*

Un code en pseudo-C, pour un arbre `a`*, serait :*

```
rechercheArbre(arbre a, int c) {
  si a est vide alors renvoyer échec
  si c = clé(a) alors renvoyer info(a)
  si c < clé(a)
    alors rechercheArbre(filsGauche(a), c)
    sinon rechercheArbre(filsDroit(a), c)
}
```

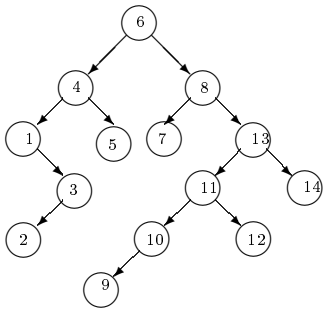
La fonction `rechercheArbre` *consiste à suivre un chemin descendant dans l'arbre jusqu'au succès ou l'échec qui sera obtenu lorsqu'on arrivera à une feuille. Dans le pire des cas, le nombre d'appels récursifs (ou de comparaisons requises) sera de l'ordre de* $\Theta(h)$*, où* h *est la hauteur de l'arbre* `a`*.*

Exercice 8 *L'opération d'insertion d'une clé consiste à suivre le bon chemin dans l'arbre jusqu'à arriver à une feuille qu'on remplace alors par un nouveau nœud interne pourvu de deux fils vides,*

```
void insererArbre(arbre &a, c) {
  si a est vide alors a = noeud(vide, c, vide) sinon
  si c ≤ clé(a) alors inserer(filsGauche(a), c)
  sinon insererArbre(filsDroit(a), c)
}
```

Tout comme pour la recherche, la complexité est linéaire en la profondeur de l'arbre.

Exercice 9



Exercice 10 *Dans l'ordre :*

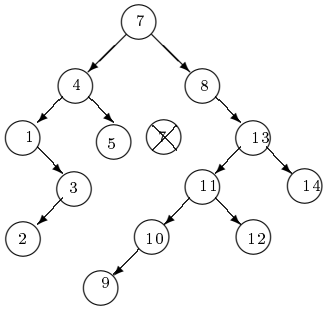
- *préfixé : 6 4 1 3 2 58 7 13 11 10 9 12 14*
- *infixé : 1 2 3 4 5 6 7 8 9 10 11 12 13 14*
- *postfixé : 2 3 1 5 4 7 9 10 12 11 14 13 8 6*

Exercice 11

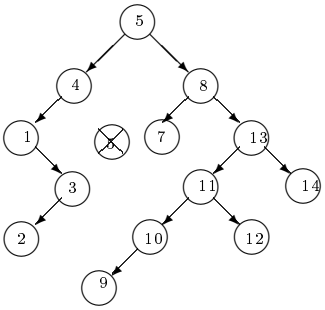
1. *Si le nœud à supprimer a deux fils vides, on le remplace par l'arbre vide ;*
2. *s'il a un et un seul fils vide, on le remplace par son autre fils ;*
3. *sinon, on calcule son* successeur *dans l'arbre, c'est-à-dire le nœud possédant la plus petite clé plus grande que la sienne : on se rend aisément compte qu'il s'agit du nœud de clé minimale de son fils droit. On remplace alors sa clé par celle de son successeur, que l'on supprime ensuite aisément car il n'a pas de fils gauche (cas 2).*

Sans détailler, on montre que cette opération s'exécute aussi en $\Theta(h)$ *où* h *est la hauteur de l'arbre.*

Exercice 12 *Il y a en fait deux possibilités : l'important étant de garantir que le résultat reste un arbre binaire de recherche. On peut obtenir l'un des arbres suivants :*



ou bien



Exercice 13

- *l'unique arbre sans élément est l'arbre vide :* $C_0 = 1$
- *l'unique arbre à 1 élément est formé d'une racine avec fils gauche et droit vides, donc* $C_1 = 1$
- *les deux seuls arbres à 2 éléments sont formés d'une racine et d'un élément qui est soit fils droit, soit fils gauche de la racine, donc* $C_2 = 2$
- *un arbre à 3 éléments est formé d'une racine et d'un sous-arbre gauche à 0, 1 ou 2 éléments ainsi que d'un fils droit à (respectivement) 2, 1 ou 0 éléments. Par conséquent,* $C_3 = C_0C_2+C_1C_1+C_2C_0 = 2+1+2 = 5$.
- *de même,* $C_4 = C_0C_3 + C_1C_2 + C_2C_1 + C_3C_0 = 5 + 2 + 2 + 5 = 14$.

Exercice 14 *Généralisant le raisonnement tenu précédemment pour* C_3 *et* C_4*, on obtient :*

$$C_n = \sum_{p+q=n-1} C_p \times C_q$$

Exercice 15 *En développant* $C(x)^2$*, on obtient :*

$$C(x)^2 = \left(\sum_{n\geq 0} C_n x^n\right) \times \left(\sum_{n\geq 0} C_n x^n\right) = \sum_{n\geq 0}\left(\sum_{p+q=n} C_pC_q\right) x^n$$

et par conséquent :

$$xC(x)^2 + 1 = 1 + x\sum_{n\geq 0} C_{n+1}x^n = \sum_{n\geq 0} C_n x^n = C(x)$$

On déduit de cette relation que :

$$C(x) = \frac{1 \pm \sqrt{1-4x}}{2x}$$

Pour savoir laquelle de ces deux solutions est la bonne, on utilise le fait que $C(0) = C_0 = 1$.
Or $\lim_{x\to 0^+} \frac{1+\sqrt{1-4x}}{2x} = +\infty$ *alors que* $\lim_{x\to 0} \frac{1+\sqrt{1-4x}}{2x} = 1 = C_0$*. Donc :*

$$C(x) = \frac{1 - \sqrt{1-4x}}{2x}$$

Exercice 16 *En utilisant le développement en série formelle de* $\sqrt{1-x}$*, on obtient :*

$$\begin{aligned} C(x) &= \frac{1-\sqrt{1-4x}}{2x} = \frac{1}{2x}\left(1 - \left(1 - \sum_{n=1}^{+\infty} \frac{2}{4^n\,(2n-1)} \begin{pmatrix} 2n-1 \\ n \end{pmatrix} 4^n x^n\right)\right) \\ &= \sum_{n=1}^{+\infty} \frac{1}{2n-1}\begin{pmatrix} 2n-1 \\ n \end{pmatrix} x^{n-1} = \sum_{n=0}^{+\infty} \frac{1}{2n+1}\begin{pmatrix} 2n+1 \\ n+1 \end{pmatrix} x^n \end{aligned}$$

En identifiant le coefficient de x^n *à* C_n*, on obtient :*

$$C_n = \frac{1}{2n+1}\begin{pmatrix} 2n+1 \\ n+1 \end{pmatrix} = \frac{(2n+1)!}{(2n+1)(n+1)!n!} = \frac{(2n)!}{(n+1)n!n!} = \frac{1}{n+1}\begin{pmatrix} 2n \\ n \end{pmatrix}$$

Exercice 17 *La formule de Stirling permet d'obtenir une valeur approchée de* C_n *quand* n *devient grand :*

$$C_n = \frac{(2n)!}{(n+1)n!n!} \approx \frac{(2n)^{2n}(e^n)^2\sqrt{2\pi\times 2n}}{(n+1)e^{2n}(n^n)^2(2\pi\times n)} \approx \frac{4^n}{n\times\sqrt{\pi\times n}} \approx \frac{1}{\sqrt{\pi}} 4^n n^{-\frac{3}{2}}$$

Exercice 18 *Le nombre d'arbres binaires à* n *éléments dont le sous-arbre gauche est vide est égal au nombre total d'arbres binaires à* $n-1$ *éléments. Par conséquent, la probabilité d'avoir un sous-arbre droit ou gauche vide (si* $n \geq 3$*) est égale à :*

$$\frac{2C_{n-1}}{C_n} = \frac{2}{n}\,\frac{(2n-2)!}{(n-1)!(n-1)!}\,(n+1)\,\frac{n!n!}{(2n)!} = \frac{n+1}{2n-1} \approx \frac{1}{2}$$

Un arbre binaire quelconque a donc une chance sur deux d'avoir l'un de ses sous-arbres immédiats qui est vide !

Exercice 19 *Il y a exactement* $(C_n)^2$ *arbres à* $2n+1$ *éléments ayant des fils droit et gauche comportant exactement* n *éléments chacun. Par conséquent, la probabilité que cela se produise est égale à :*

$$\frac{(C_n)^2}{C_{2n+1}} \approx \frac{\frac{1}{\pi}4^{2n}n^{-3}}{\frac{1}{\sqrt{\pi}}4^{2n+1}(2n+1)^{-\frac{3}{2}}} = \mathcal{O}(n^{-\frac{3}{2}})$$

Exercice 20

```
#include <iostream>
using namespace std;
struct noeud {
  int cle;
  noeud *gauche;
  noeud *droit;
};
typedef noeud *arbre;
// Construit un arbre à partir de la valeur v de la clé de la racine
// et des sous-arbres droit (sad) et gauche (sag)
arbre branche(int v, arbre sag, arbre sad) {
  arbre nouv = new noeud;
  nouv->cle = v; nouv->gauche = sag; nouv->droit = sad;
  return nouv;
}

void erreur(char *s) {
  cerr << "Erreur: " << s << endl;
  exit(1);
}

// Impression infixée d'un arbre binaire:
void imprimeInfixe(arbre a) {
  if (a == NULL)
    return;
  else {
    imprimeInfixe(a->gauche);
    cout << a->cle << " ";
    imprimeInfixe(a->droit);
  }
}
// Recherche de v dans l'arbre binaire de recherche a:
bool recherche(int v, arbre a) {
  if (a == NULL) return false;
  if (v == a->cle) return true;
  if (v < a->cle) return recherche(v, a->gauche);
  else return recherche(v, a->droit);
}
// Recherche du minimum dans un arbre:
int minimum(arbre a) {
  if (a == NULL) return -1;
  if (a->gauche == NULL) return a->cle;
  else return minimum(a->gauche);
}
```

```
// Recherche du maximum dans un arbre:
int maximum (arbre a) {
  if (a == NULL) return -1;
  if (a→droit == NULL) return a→cle;
  else return maximum(a→droit);
}
// Comptage du nombre d'éléments contenus dans l'arbre a:
int nbNoeuds(arbre a) {
  if (a == NULL) return 0;
  else return 1 + nbNoeuds(a→gauche) + nbNoeuds(a→droit);
}
// Mesure de la hauteur de l'arbre a:
int hauteur(arbre a) {
  if (a == NULL) return -1;
  int hg = hauteur(a→gauche);
  int hd = hauteur(a→droit);
  if (hg > hd) return 1+hg;
  else return 1+hd;
}
// Insertion de v dans l'arbre a:
void insertion(int v, arbre &a) {
  if (a == NULL)
    a = branche(v,NULL,NULL);
  else
    if (v < a→cle)
      insertion(v, a→gauche);
    else
      insertion(v, a→droit);
}
// Suppression de v dans l'arbre a:
bool supprime(int v, arbre &a) {
  if (a == NULL) return false;
  if (v == a→cle) {
    if (a→gauche == NULL) {
      arbre tmp = a→droit;
      delete a;
      a = tmp;
      return true;
    }
    if (a→droit == NULL) {
      arbre tmp = a→gauche;
      delete a;
      a = tmp;
      return true;
    }
    int val = minimum(a→droit);
    supprime(val, a→droit);
    a→cle = val;
    return true;
  }
  else
```

```
    if (v < a→cle)
      return supprime(v, a→gauche);
    else
      return supprime(v, a→droit);

}
// Libération complète de la mémoire occuppée par un arbre:
void libereArbre(arbre a) {
  if (a ≠ NULL) {
    libereArbre(a→gauche);
    libereArbre(a→droit);
    delete a;
  }
}
```