IN 101 - TD 10
25 novembre 2011

Corrigé

ENSTA ParisTech

Solution 1 : Hauteur d'un arbre binaire

1.a] Par définition de la hauteur d'un arbre, un arbre de hauteur h possède au moins $h+1$ éléments. De plus, il existe des arbres de hauteur h possédant $h+1$ 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.

1.b] Un arbre de hauteur h possède un nombre maximal d'éléments lorsque tous les niveaux sont complètement remplis. 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 :

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

1.c] Considérons un arbre à n éléments dont les niveaux sont remplis au maximum. La hauteur d'un tel arbre est alors exactement de $\log_2(n+1)$. Plus précisément, puisque d'après la question précédente $n \leq 2^{h+1} - 1$, on a $h \geq \log_2(n+1) - 1$, d'où :

$$h \geq \lceil \log_2(n+1) - 1 \rceil \iff h \geq \lfloor \log_2(n) \rfloor.$$

1.d] Cette propriété se démontre par récurrence. Notons P_n la propriété « *tout arbre binaire ayant au plus n nœuds et dont les nœuds ont soit deux fils, soit aucun, possède 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_n vraie. Un arbre à $n+1$ éléments se compose d'une racine et de deux sous-arbres (droit et gauche), à respectivement p et q éléments ($1+p+q = n+1$). 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+1} est vraie et donc P_n est vraie pour tout n.

Solution 2 : Arbres binaires de recherche

2.a] La propriété définissant un arbre binaire s'imposant à chacun des nœuds de l'arbre, elle définit par la même occasion chaque sous-arbre comme un arbre binaire de recherche à part entière.

2.b] Un parcours infixe 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 clefs 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.

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

Un code en C, pour un arbre `A`, serait :

```
1  type_info ABR_search(int v, node* A) {
2    if (A == NULL) {
3      printf("Recherche infructueuse.\n");
4      return NULL;
5    }
6    if (v < A->key) {
7      return ABR_search(v, A->left);
8    } else if (v == A->key) {
9      printf("Noeud trouvé.\n");
10     return A->info;
11   } else {
12     return ABR_search(v,A->right);
13   }
14 }
```

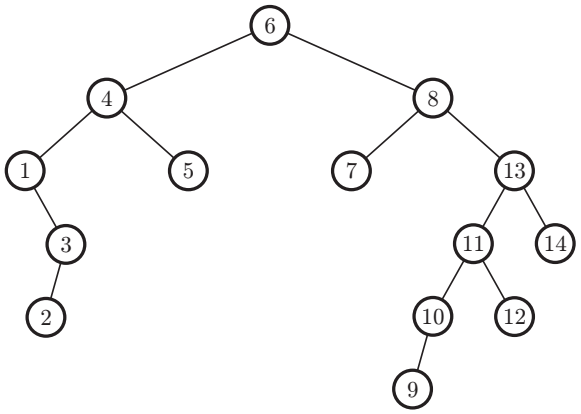
La fonction `ABR_search` consiste à suivre un chemin descendant dans l'arbre jusqu'au succès ou l'échec qui sera obtenu lorsqu'on arrivera à une feuille. Dans le pire des cas, le nombre d'appels récursifs (ou de comparaisons requises) sera de l'ordre de $\Theta(h)$, où h est la hauteur de l'arbre `A`.

2.d] L'opération d'insertion d'une clef 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,

```
1  void ABR_insert(int v, node** A) {
2    if ((*A) == NULL) {
3      node* n = (node*) malloc(sizeof(node));
4      n->key = v;
5      n->left = NULL;
6      n->right = NULL;
7      (*A) = n;
8      return;
9    }
10   if (v < (*A)->key) {
11     ABR_insert(v, &((*A)->left));
12   } else {
13     ABR_insert(v, &((*A)->right));
14   }
15 }
```

Tout comme pour la recherche, la complexité est linéaire en la profondeur de l'arbre car chaque appel récursif descend d'un niveau dans l'arbre.

2.e] On obtient l'arbre suivant qui a une hauteur de 5 (ce qui n'est pas optimal pour un arbre contenant 14 nœuds) :



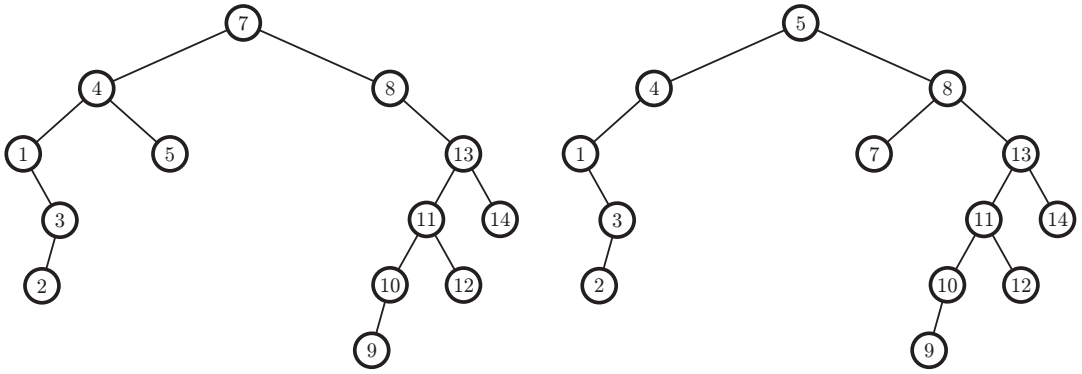
2.f] Les nœuds s'affichent dans l'ordre :
- préfixe : 6, 4, 1, 3, 2, 5, 8, 7, 13, 11, 10, 9, 12, 14
- infixe : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14
- postfixe : 2, 3, 1, 5, 4, 7, 9, 10, 12, 11, 14, 13, 8, 6

2.g] Avec un parcours en largeur les nœuds doivent s'afficher dans l'ordre : 6, 4, 8, 1, 5, 7, 13, 3, 11, 14, 2, 10, 12, 9. Entre le moment où l'on affiche un nœud et celui où l'on affiche ses fils, tout un niveau de nœuds doit être affiché. Il est donc nécessaire de stocker quelque part les fils du nœud que l'on affiche tant que ce n'est pas leur tour d'apparaître. La solution est donc d'utiliser une *file* : on commence par placer la racine dans une file vide, et à chaque étape on sort un nœud de la file, on l'affiche, et on ajoute ses fils (si le nœud en a) à la file. L'algorithme s'arrête quand la file est vide. Naturellement, les nœuds ajoutés en premiers seront affichés en premier aussi (car on utilise une file : premier entré, premier sorti), et donc on affichera d'abord la racine, puis tous les nœuds du niveau 1, puis tous ceux du niveau 2...

2.h] Pour les trois différents cas :
- si le nœud à supprimer a deux fils vides, on le remplace par l'arbre vide (on fait pointer sont père vers `NULL`),
- s'il a un et un seul fils vide, on le remplace par son autre fils,
- sinon, on calcule son *successeur* dans l'arbre, c'est-à-dire le nœud possédant la plus petite clef plus grande que la sienne : on se rend aisément compte qu'il s'agit du nœud de clef minimale de son sous-arbre droit. On remplace alors sa clef par celle de son successeur, que l'on supprime ensuite aisément car il n'a pas de fils gauche (cas précédent).
  Une autre solution est de remplacer la clef du noeud à supprimer par celle de son *prédécesseur*, *i.e.* du nœud possédant la plus grande clef inférieure à la sienne (c'est le nœud de clef maximale de son sous-arbre gauche). On supprime ensuite aisément ce prédécesseur car il n'a pas de fils droit (cas précédent encore une fois).

Cette opération s'exécute aussi en $\Theta(h)$ où h est la hauteur de l'arbre : c'est le coût de la recherche du nœud à supprimer puis de son successeur (si besoin) car le reste des opérations ce fait en temps constant.

2.i] La racine de l'arbre a deux fils, donc on doit remplacer ce nœud par son successeur (ou son prédécesseur). Deux solutions sont donc possibles :



Solution 3 : Dénombrement d'arbres binaires de recherche

3.a] La racine d'un ABR est toujours le premier élément que l'on y insère. Pour qu'il n'ai qu'un seul fils il faut donc que ce nœud soit le plus petit ou le plus grand des n nœuds. La probabilité pour la racine de n'avoir qu'un seul fils est donc $\frac{2}{n}$.

3.b] Comme précédemment, si la racine n'a qu'un seul fils, ce fils sera forcément le deuxième élément inséré dans l'arbre. Pour qu'il n'ait qu'un seul fils il faut que ce soit le plus petit ou le plus grand des $n-1$ nœuds restants. Cela arrive avec une probabilité $\frac{2}{n-1}$. La probabilité d'avoir un arbre de hauteur maximum (c'est-à-dire de hauteur $n-1$) est donc :

$$\prod_{i=2}^{n} \frac{2}{i} = \frac{2^{n-1}}{n!}.$$

3.c] Encore une fois, la racine est toujours le premier élément inséré. Les éléments du sous-arbre gauche sont tous plus petits que la racines, et ceux du sous-arbre droit plus grands. Il y aura donc p éléments dans chacun des sous-arbres de la racine si le premier élément inséré est l'élément médian de la liste d'éléments (p plus petits que lui et p plus grands). Il n'y a donc qu'un choix possible. La probabilité d'avoir deux sous-arbres de p éléments est donc $\frac{1}{n}$.

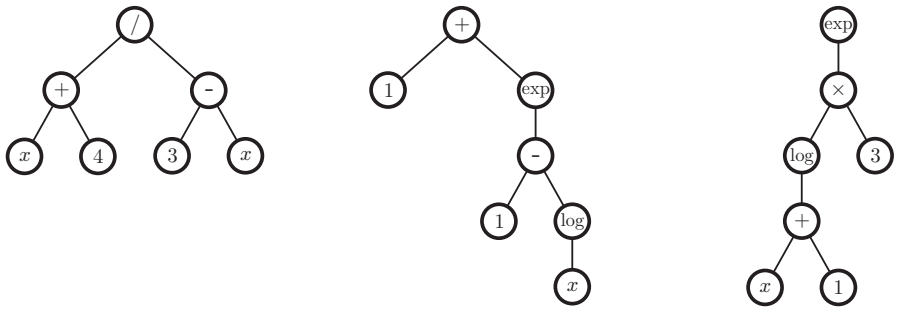
3.d] Si la racine a deux sous-arbres de même taille p, la probabilité pour l'un des fils de la racine d'avoir deux sous-arbres de taille égale (donc $\frac{p-1}{2}$) se calcule de la même façon est vaut $1/p$. La probabilité pour que les quatre sous-arbres des fils de la racine aient tous exactement $\frac{p-1}{2}$ nœuds est donc $\frac{1}{n} \times \frac{1}{p^2}$. De façon générale, la probabilité pour qu'un arbre de $n = 2^{h+1} - 1$ nœuds soit de hauteur h est donc :

$$\prod_{i=1}^{h} \left(\frac{1}{2^{i+1}-1} \right)^{2^{h-i}}$$

Pour $1 = 2$ cette probabilité vaut $\frac{1}{3}$, pour $h = 2$ elle vaut $\frac{1}{63}$, pour $h = 3$ elle vaut $\frac{1}{59535}$ et pour $h = 4$ elle vaut environ $2^{-36,6}$.

Solution 4 : Arbres pour la représentation d'expressions arithmétiques

4.a] On obtient les arbres suivants :



4.b] Pour évaluer un arbre en a, on regarde le symbole du nœud courant (la racine au départ), et en fonction de ce que contient ce nœud on retourne un résultat différent :

- si le nœud contient un `+`, on retourne l'évaluation du fils gauche en a `+` l'évaluation du fils droit en a,
- de même, si le nœud contient un `-`, un `*` ou un `/`, on applique cette même opération aux évaluations en a des fils gauche et droits de notre nœud et on retourne le résultat,
- si le nœud contient `exp` ou `log`, on retourne l'exponentielle (ou le log) de l'évaluation de son fils unique en a,
- si le nœud contient une constante, on retourne cette constante,
- si le nœud contient `x`, on retourne a.

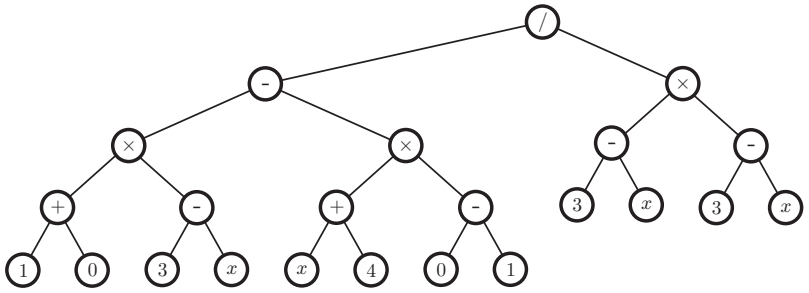
Il suffit donc d'écrire une fonction récursive très simple, principalement composée d'une série de `if` pour tester le contenu du nœud courant.

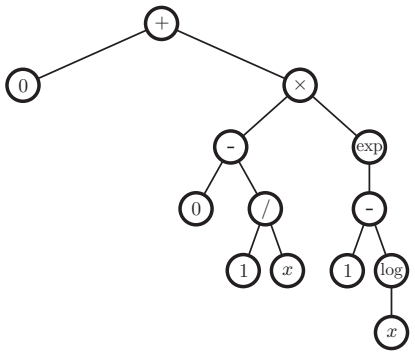
4.c] Pour dériver un arbre il faut procéder de façon similaire à l'évaluation : il faut faire une série de `if` pour tester le contenu du nœud à dériver, mais au lieu de simplement retourner une valeur, il faut maintenant construire un arbre. Voici un exemple de ce à quoi peut ressembler le code d'une fonction de dérivation :

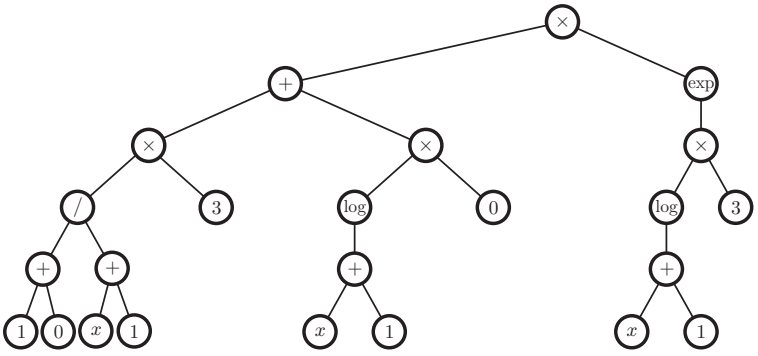
```
1  node* derive(node* A) {
2    .......
3    if (A->symbol == '+') {
4      node* n = (node*) malloc(sizeof(node));
5      n->symbol = '+';
6      n->left = derive(A->left);
7      n->right = derive(A->right);
8      return n;
9    }
10   if (A->symbol == '*') {
11     node* n = (node*) malloc(sizeof(node));
12     node* nl = (node*) malloc(sizeof(node));
13     node* nr = (node*) malloc(sizeof(node));
14     n->symbol = '+';
15     n->left = nl;
16     n->right = nr;
17     nl->symbol = '*'
```

```
18      nl->left = derive(A->left);
19      nl->right = clone(A->right);
20      nr->symbol = '*'
21      nr->left = clone(A->left);
22      nr->right = derive(A->right);
23      return n;
24    }
25    if (A->symbol == 'log') { // ce n'est pas du vrai C ça !
26      node* n = (node*) malloc(sizeof(node));
27      n->symbol = '/';
28      n->left = derive(A->left); // on suppose que le fils unique est dans left
29      n->right = clone(A->left);
30      return n;
31    }
32    .....
33  }
```

Pour nos trois arbres de départ, les arbres ainsi construits ne sont pas très "beaux" et nécessiteraient une étape de simplification. Voila à quoi il pourraient ressembler.







Solution 5 : Pour aller plus loin : mise en œuvre des ABR

```c
#include <stdio.h>

typedef struct st_node {
  int key;
  struct st_node *left;
  struct st_node *right;
} node;

/* Construit un arbre a partir de la valeur v de la clef de la racine
   et des sous-arbres droit (rs) et gauche (ls) */
node* build (int v, node* L, node* R) {
  node* nw = (node*) malloc(sizeof(node));
  nw->key = v;
  nw->left = L;
  nw->right = R;
  return nw;
}
/* Impression infixe d'un arbre binaire */
void print_inorder (node* A) {
  if (A != NULL) {
    print_inorder(A->left);
    printf("%d ",A->key);
    print_inorder(A->right);
  }
}
/* Recherche de v dans l'arbre binaire de recherche A */
int search (int v, node* A) {
  if (A == NULL) {
    return 0;
  }
  if (v == A->key) {
    return 1;
  }
  if (v < A->key) {
    return search(v, A->left);
  } else {
    return search(v, A->right);
  }
}
/* Recherche du minimum dans un arbre binaire de recherche */
int minimum (node* A) {
  if (A == NULL) {
    return -1;
  }
  if (A->left == NULL) {
    return A->key;
  }
  return minimum(A->left);
}
/* Recherche du maximum */
int maximum (node* A) {
  if (A == NULL) {
    return -1;
```

```
54    }
55    if (A->right == NULL) {
56      return A->key;
57    }
58    return maximum(A->right);
59  }
60  /* Comptage du nombre d'éléments contenus dans l'arbre A */
61  int number_of_nodes (node* A) {
62    if (A == NULL) {
63      return 0;
64    }
65    return 1 + number_of_nodes(A->left) + number_of_nodes(A->right);
66  }
67  /* Mesure de la hauteur de l'arbre A */
68  int height (node* A) {
69    if (A == NULL) {
70      return -1;
71    }
72    int hl = height(A->left);
73    int hr = height(A->right);
74    if (hl > hr) {
75      return 1 + hl;
76    } else {
77      return 1 + hr;
78    }
79  }
80  /* Insertion de v dans l'arbre A */
81  void insert (int v, node** A) {
82    if ((*A) == NULL) {
83      (*A) = build(v,NULL,NULL);
84    } else {
85      if (v < (*A)->key) {
86        insert(v, &((*A)->left));
87      } else {
88        insert(v, &((*A)->right));
89      }
90    }
91  }
92  /* Suppression de v dans l'arbre A */
93  int remove (int v, node** A) {
94    if ((*A) == NULL) {
95      return 0;
96    }
97    if (v == (*A)->key) {
98      if ((*A)->left == NULL) {
99        node* tmp = (*A)->right;
100       free(*A);
101       (*A) = tmp;
102       return 1;
103     }
104     if ((*A)->right == NULL) {
105       node* tmp = (*A)->left;
106       free(*A);
107       (*A) = tmp;
108       return 1;
```

```
109     }
110     int successor = minimum((*A)->right);
111     remove(successor, (*A)->right);
112     (*A)->key = successor;
113     return 1;
114   } else {
115     if (v < (*A)->key) {
116       return remove(v, &((*A)->left));
117     } else {
118       return remove(v, &((*A)->right));
119     }
120   }
121 }
122 /* Libération complète de la mémoire occupée par un arbre */
123 void free_tree (node* A) {
124   if (A != NULL) {
125     free_tree(A->left);
126     free_tree(A->right);
127     free(A);
128   }
129 }
```