1 Hauteur d'un arbre binaire

Exercice 1

Il existe de toute évidence des arbres à $h+1$ éléments de hauteur égale à h : il suffit que tous 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 taille $2^{h+1}-1$ et de hauteur h : les arbres dont tous les niveaux de profondeur $\leq h$ sont complètement remplis. En effet, on a alors un nœud de profondeur 0, deux nœuds de profondeur 1, quatre nœuds de profondeur 2, etc., soit 2^p nœuds de profondeur p, ce qui donne un nombre total de nœuds égal à :

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

Exercice 3

Considérons un arbre à n éléments dont les "niveaux" sont remplis au maximum. La hauteur d'un tel arbre est alors égale à $\lfloor \log_2(n) \rfloor$.

Exercice 4

Cette propriété se démontre par récurrence. Notons P_n la propriété *"tout arbre binaire dont les nœuds ont soit deux fils, soit aucun, a un nombre de nœuds internes égal au nombre de feuilles moins un"*.

La propriété P_1 est clairement vraie car l'unique arbre a un seul élément a une feuille et aucun nœud interne.

Supposons P_i vraie pour tout $i < n$. Un arbre à n éléments se compose d'une racine, d'un sous-arbre gauche à p éléments et d'un sous-arbre droit à q éléments ($p+q+1=n$). On peut appliquer les propriétés P_p et P_q, vraies d'après l'hypothèse de récurrence : le nombre total de noeuds internes est égal à un (la racine) plus ceux de gauche plus ceux de droite. Le nombre de feuilles est égal au nombre de feuilles à gauche plus à droite. Par conséquent, on en déduit facilement que P_n est vraie.

2 Arbres binaires de recherche

2.1 Définition

Exercice 5

La propriété définissant un arbre binaire de recherche est fondamentalement récursive. Tout sous-arbre doit par conséquent être un arbre binaire de recherche.

Exercice 6

Un parcours infixé imprime récursivement le sous-arbre de gauche, la racine et le sous-arbre de droite. D'après la propriété définissant les arbres binaires de recherche, on imprime donc d'abord tout ce qui est plus petit que la racine, puis la racine, puis tout ce qui est plus grand. Comme cette procédure est récursive, les clés de l'arbre apparaissent dans l'ordre croissant.

Un algorithme de tri consiste donc à insérer tous les éléments d'un tableau un à un dans un arbre binaire de recherche et à en faire un parcours infixé pour en extraire les éléments triés.

2.2 Opérations sur les arbres binaires de recherche

Exercice 7

La recherche d'une clé dans un arbre binaire de recherche imite la recherche dichotomique : il suffit de comparer la clé à découvrir avec celle du nœud courant, et si on ne l'a pas trouvée, de répéter l'opération sur le sous-arbre droit ou gauche selon le cas. Ceci nous donne le pseudo-code suivant, pour un arbre A :

Algorithme 1 RECHERCHE-ARBRE(A, *clé*)
1 **si** $A =$ FEUILLE **alors retourner** *recherche_infructueuse*
2 **si** *clé* $=$ CLÉ(A) **alors retourner** INFORMATION(A)
3 **si** *clé* $<$ CLÉ(A)
4 **alors retourner** RECHERCHE-ARBRE(GAUCHE(A), *clé*)
5 **sinon retourner** RECHERCHE-ARBRE(DROIT(A), *clé*)

La fonction RECHERCHE-ARBRE consiste à suivre un chemin descendant dans l'arbre jusqu'au succès ou à l'échec, qui sera prononcé lorsqu'on atteindra une feuille. Ce qui signifie que dans le pire des cas, le nombre d'appels récursifs (ou de comparaisons requises) sera en $\Theta(h)$, 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 :

Algorithme 2 INSERTION-ARBRE(A, *clé*)
1 **si** $A =$ FEUILLE
2 **alors** $A \leftarrow$ BRANCHE-BIN(*clé*, FEUILLE, FEUILLE)
3 **sinon si** *clé* $\leq$ CLÉ(A)
4 **alors** INSERTION-ARBRE(GAUCHE(A), *clé*)
5 **sinon** INSERTION-ARBRE(DROIT(A), *clé*)

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

Exercice 9

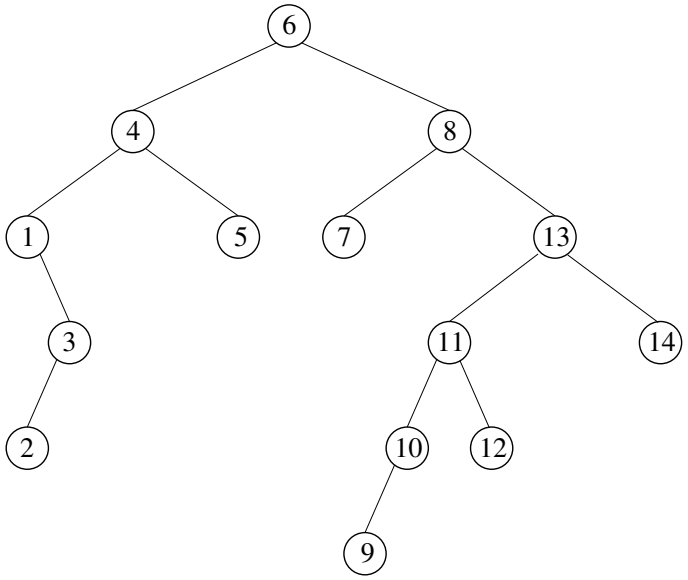

FIGURE 1 – Arbre binaire de recherche obtenu en ajoutant successivement les entiers 6, 8, 13, 7, 4, 1, 5, 11, 3, 14, 10, 2, 9 et 12 dans un arbre initialement vide

Voir figure 1.

Exercice 10

Les résultats des différents parcours sont :
— préfixé : 6, 4, 1, 3, 2, 5, 8, 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 détruit alors purement et simplement, c'est-à-dire qu'on le remplace par l'arbre vide ;
2. s'il a un et un seul fils vide, on le remplace par son autre fils, non vide ;
3. sinon, on commence par calculer son *successeur* dans l'arbre, i.e. le nœud possédant la plus petite clé plus grande que la sienne : il n'est pas très difficile de se rendre compte qu'il s'agit du nœud de clé minimale de son fils droit ; on remplace alors sa clé par celle de son successeur, qu'on supprime finalement de l'arbre, ce qui est aisé car il ne peut avoir de fils gauche (on emploie donc la méthode précédente).

Sans rentrer dans le détail, on prouve que cette opération s'exécute aussi en $\Theta(h)$, où h est la hauteur de l'arbre.

Exercice 12

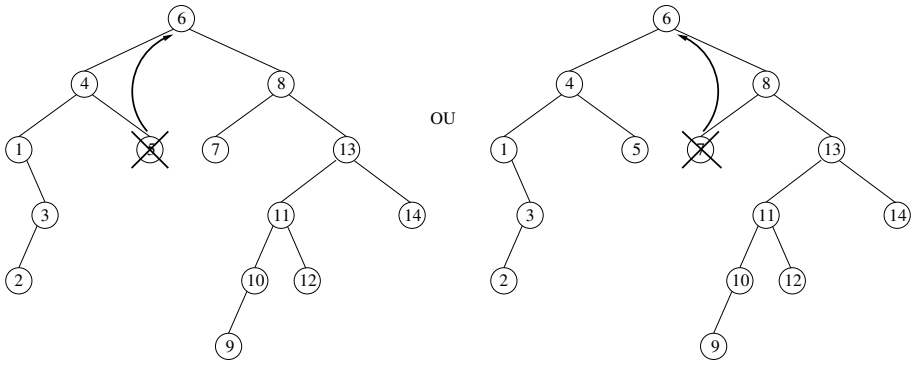

FIGURE 2 – Suppression de la racine de l'arbre construit à l'exercice 9

Voir figure 2.

3 Dénombrement des arbres binaires

Exercice 13

— l'unique arbre sans élément est l'arbre vide : $C_0 = 1$
— l'unique arbre a un seul élément est formé d'une racine avec des fils droit et gauche vides : $C_1 = 1$
— les deux seuls arbres à deux éléments sont formés d'une racine et d'un élément, soit à droite, soit à gauche : $C_2 = 2$
— un arbre a trois éléments est formé d'une racine et d'un sous-arbre gauche avec 0, 1 ou 2 éléments, le sous-arbre droit ayant alors 2, 1 ou 0

éléments, respectivement. 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

On peut généraliser le raisonnement tenu pour le calcul de C_3 et C_4. On obtient ainsi

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

Exercice 15

En développant $C(x)^2$, on obtient

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

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 immédiatement de cette relation que

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

Afin de savoir laquelle des deux solutions est la bonne, on utilise le fait que $C(0) = C_0 = 1$. Or $\lim\limits_{x\to 0^+} \dfrac{1+\sqrt{1-4x}}{2x} = +\infty$ alors que $\lim\limits_{x\to 0} \dfrac{1-\sqrt{1-4x}}{2x} = 1 = C_0$ donc

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

Exercice 16

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

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

$$= \sum_{n=1}^{+\infty} \frac{1}{2n-1} \begin{pmatrix} 2n-1 \\ n \end{pmatrix} x^{n-1} = \sum_{n=0}^{+\infty} \frac{1}{2n+1} \begin{pmatrix} 2n+1 \\ n+1 \end{pmatrix} x^n$$

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

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

Exercice 17

La formule de Stirling permet d'obtenir un 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 une 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 de l'ordre d'une chance sur deux d'avoir un sous-arbre gauche ou droit vide !

Exercice 19

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

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

4 Pour aller plus loin : implémentation des arbres binaires de recherche

Exercice 20

```
#include <stdio.h>
#include <stdlib.h>

struct noeud
{ int cle;
  struct noeud *gauche;
  struct noeud *droit;
};
typedef struct noeud *arbre;
```

```
// construit un arbre artir de la valeur v de la cle
// la racine et des sous-arbres droit (sad) et gauche (sag)
arbre branche(arbre sag, arbre sad, int v)
{ arbre nouv;
  nouv=(arbre) malloc(sizeof(struct noeud));
  nouv→cle=v;
  nouv→gauche=sag;
  nouv→droit=sad;
  return nouv;
}

// Impression infixd'un arbre binaire
void imprime_infixe(arbre a)
{ if (a≠NULL)
    {
      imprime_infixe(a→gauche);
      printf("%d ",a→cle);
      imprime_infixe(a→droit);
    }
}

// Recherche de v dans l'arbre binaire de recherche a
int recherche(arbre a, int v)
{ if (a==NULL) return 0;
  if (a→cle==v) return 1;
  if (v<a→cle) return recherche(a→gauche,v);
  else return recherche(a→droit,v);
}

// Recherche du minimum
int minimum(arbre a)
{ if (a==NULL) return −1;
  if (a→gauche==NULL) return a→cle;
  else return minimum(a→gauche);
}

// Recherche du maximum
int maximum(arbre a)
{ if (a==NULL) return −1;
  if (a→droit==NULL) return a→cle; else return minimum(a→droit);
}

// Comptage du nombre d'ments contenu dans l'arbre a
int nbnoeud(arbre a)
```

```
{ if (a==NULL) return 0;
  return 1+nbnoeud(a→gauche)+nbnoeud(a→droit);
}

// Mesure de la hauteur de l'arbre a
int hauteur(arbre a)
{ int hautg, hautd;
  if (a==NULL) return −1;
  hautg=hauteur(a→gauche);
  hautd=hauteur(a→droit);
  if (hautg>hautd) return 1+hautg; else return 1+hautd;
}

// Insertion de v dans l'arbre a
void insertion(arbre *a, int v)
{ if (*a==NULL)
    *a=branche(NULL,NULL,v);
  else
    if (v≤(*a)→cle) insertion(&((*a)→gauche),v);
    else insertion(&((*a)→droit),v);
}

// Suppression de v dans l'arbre a
int supprime(arbre *a, int v)
{ arbre temp;
  int val;
  if (*a==NULL) return 0;
  if ((*a)→cle==v)
    {
      if ((*a)→gauche==NULL)
    {
      temp=(*a)→droit;
      free(*a);
      *a=temp;
    }
      else
    if ((*a)→droit==NULL)
      {
        temp=(*a)→gauche;
        free(*a);
        *a=temp;
      }
      else
    {
```

```
      val=minimum((*a)→droit);
      supprime(&((*a)→droit),val);
      (*a)→cle=val;
    }
      return 1;
    }
  else
    if (v<(*a)→cle)
      return supprime(&((*a)→gauche),v);
    else
      return supprime(&((*a)→droit),v);
}

// Libtion compl de la mire occuppar un arbre
void free_arbre(arbre a)
{ if (a≠NULL)
    {
      free_arbre(a→gauche);
      free_arbre(a→droit);
      free(a);
    }
}

// Programme de test
int main()
{ arbre abr=NULL;
  insertion(&abr,4); insertion(&abr,5); insertion(&abr,8);
  insertion(&abr,9); insertion(&abr,2); insertion(&abr,3);
  insertion(&abr,1); insertion(&abr,7);

  imprime_infixe(abr);
  printf("\n");
  printf("nbnoeud = %d\n",nbnoeud(abr));
  printf("hauteur = %d\n",hauteur(abr));

  supprime(&abr,4);
  imprime_infixe(abr);
  printf("\n");
  printf("nbnoeud = %d\n",nbnoeud(abr));
  printf("hauteur = %d\n",hauteur(abr));
  exit(0);
}
```