VI. Structures de données arborescentes

1. Arbres binaires
   - 1.1 Définition
   - 1.2 Terminologie
   - 1.3 Mesures sur les arbres binaires
   - 1.4 Arbres binaires particuliers
   - 1.5 Représentation des arbres binaires
     - 1.5.a Utilisation des pointeurs
     - 1.5.b Utilisation des tableaux
   - 1.6 Parcours des arbres binaires
     - 1.6.a Ordre préfixe ou pré-ordre (RGD)
     - 1.6.b Ordre infixe ou symétrique (GRD)
     - 1.6.c Ordre suffixe ou post-ordre (GDR)
   - 1.7 Exemples d'application
2. Arbres planaires généraux
   - 2.1 Définition
   - 2.2 Représentation des arbres planaires
     - 2.2.a Représentations simples
     - 2.2.b Représentation sous forme d'arbres binaires
   - 2.3 Exercices

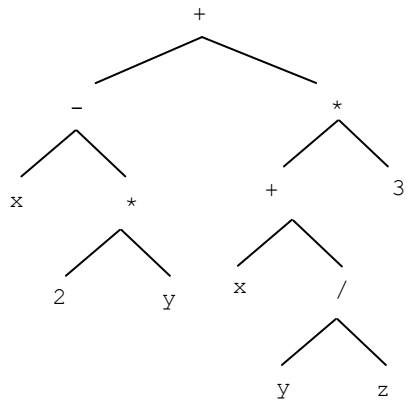
Un arbre est un ensemble de nœuds, organisés de façon hiérarchique, à partir d'un nœud distingué, appelé *racine*. La structure d'arbre est l'une des plus importantes en informatique:

- Organisation des fichiers dans les systèmes d'exploitation,
- Programmes traités par un compilateur,
- ...

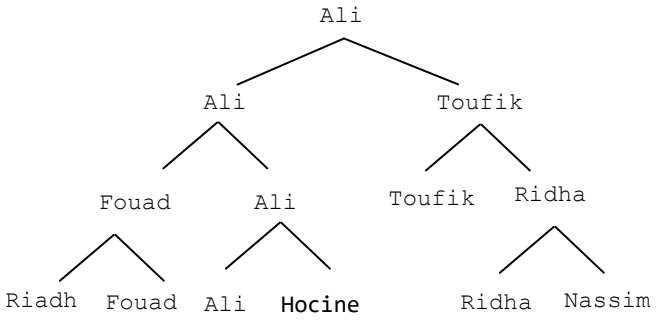
1. Arbres binaires

Exemple 1 : Une expression arithmétique dans laquelle tous les opérateurs sont binaires.

```
(x - (2 * y)) + ((x + (y / z)) * 3)
```



Exemple 2 : Les résultats d'un tournoi de tennis.



1.1. Définition

Un arbre binaire est soit vide (noté ∅), soit de la forme B = <o, B1, B2>, où B1 et B2 sont des arbres binaires disjoints et (o) est un nœud appelé racine.

Remarque : Il est important de noter la non-symétrie gauche-droite des arbres binaires.

1.2. Terminologie

Un arbre dont les nœuds contiennent des éléments est dit *arbre étiqueté*.

Étant donné un arbre B = <o, B1, B2> :

- o est la racine de B.
- B1 est le sous-arbre gauche de la racine de B.
- B2 est le sous-arbre droit.
- On dit que C est un sous-arbre de B si et seulement si : C = B ou C = B1 ou C = B2 ou C est un sous-arbre de B1 ou de B2.
- On appelle *fils gauche* (respectivement *fils droit*) d'un nœud la racine de son sous-arbre gauche (respectivement sous-arbre droit).
- Si un nœud (ni) a pour fils gauche (respectivement droit) un nœud (nj), on dit que ni est le père de nj.
- Chaque nœud n'a qu'un seul père.
- Deux nœuds qui ont le même père sont dits frères.
- Le nœud ni est un ascendant (ancêtre) du nœud nj si et seulement si ni est le père de nj ou un ascendant du père de nj.
- ni est un descendant de nj si et seulement si ni est le fils de nj ou ni est un descendant d'un fils de nj.
- Tous les nœuds d'un arbre binaire ont au plus deux fils.
- Un nœud qui a deux fils est appelé *nœud interne* ou *point double*.
- Un nœud qui a seulement un fils gauche (respectivement droit) est dit *point simple à gauche* (respectivement *point simple à droite*).
- Un nœud sans fils est appelé *nœud externe* ou *feuille*.
- On appelle *branche* de l'arbre B tout chemin[6] de la racine à une feuille de B.
- Un arbre a autant de branches que de feuilles.
- On appelle *bord gauche* (respectivement *bord droit*) de l'arbre B le chemin obtenu à partir de la racine en ne suivant que des liens gauches (respectivement droits).

1.3. Mesures sur les arbres binaires

1. La taille d'un arbre est le nombre de ses nœuds :

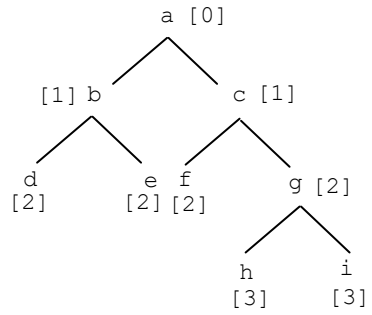
$$\begin{cases} \text{Taille(arbre\_vide)} = 0 \\ \text{Taille}(< o, B1, B2 >) = 1 + \text{Taille}(B1) + \text{Taille}(B2) \end{cases}$$

2. La hauteur d'un nœud (profondeur ou niveau) est le nombre de liens sur l'unique chemin de la racine à ce nœud.

$$\begin{cases} h(x) = 0, & \text{si x est la racine de B} \\ h(x) = 1 + h(y), & \text{si y est le père de x} \end{cases}$$

3. La hauteur d'un arbre :

$$h(B) = \max \{h(x), \quad x \text{ noeud de B}\}$$



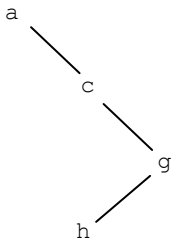
La hauteur de cet arbre = 3.

La hauteur de chaque nœud est indiquée entre `[]`.

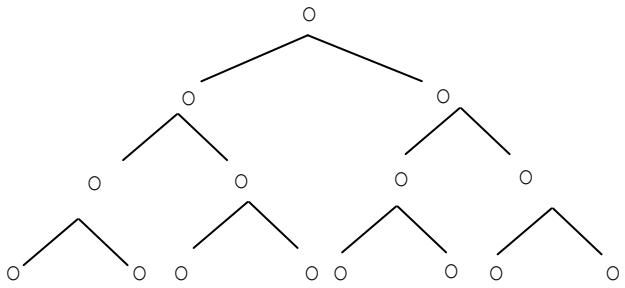
[6] Un chemin est une suite de nœuds consécutifs

1.4. Arbres binaires particuliers

- Un arbre binaire *dégénéré* ou *filiforme* est un arbre formé uniquement de points simples.



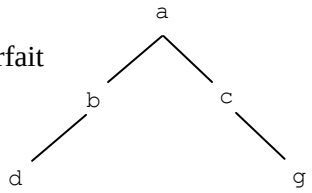
- Un arbre binaire est dit *complet* s'il contient 1 nœud au niveau 0, 2 nœuds au niveau 1, 4 nœuds au niveau 2, 2^h nœuds au niveau h.



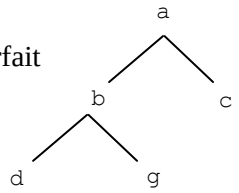
Le nombre total de nœuds d'un arbre complet de hauteur h : $1 + 2 + 2^2 + \ldots + 2^h = 2^{h+1} - 1$

- Un arbre binaire *parfait* est un arbre dont tous les niveaux sont complètement remplis, sauf éventuellement le dernier niveau, et dans ce cas les feuilles (les nœuds du dernier niveau) sont groupées le plus à gauche possible.

Arbre non-parfait

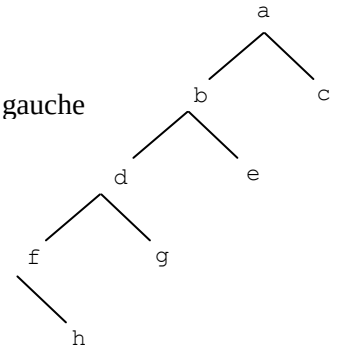


Arbre parfait



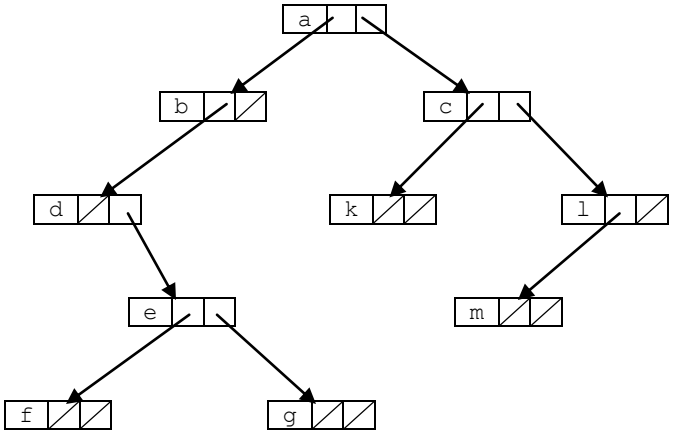
- Un *peigne gauche* (respectivement droit) est un arbre binaire dans lequel tout fils droit (respectivement gauche) est une feuille.

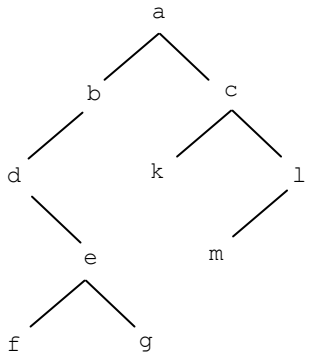
Peigne gauche



1.5. Représentation des arbres binaires

Utilisation des pointeurs





À chaque nœud on associe deux pointeurs, l'un vers le sous-arbre gauche et l'autre vers le sous-arbre droit. L'arbre est déterminé par l'adresse de sa racine. Voici la structures de données :

```
typedef struct node* binaryTree;
struct node{
        element value;
        binaryTree left;
        binaryTree right;
};
```

Utilisation des tableaux

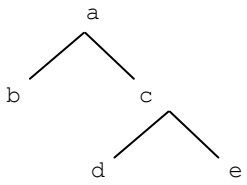
La représentation précédente (i.e., par pointeurs) peut être simulée à l'aide d'un tableau. À chaque nœud de l'arbre on associe un indice dans un tableau à trois champs (valeur, gauche, droit).

```
const int N = -;
struct node{
        element value;
        int left;
        int right;
};

struct tree{
        int root;
        node t[N];
};
```

tree

t	value	left	right
0	b	-1	-1
1	a	0	4
2	d	-1	-1
3	?	?	?
4	c	2	5
5	e	-1	-1
6	?	?	?
	.	.	.
	.	.	.
	.	.	.

root: 1

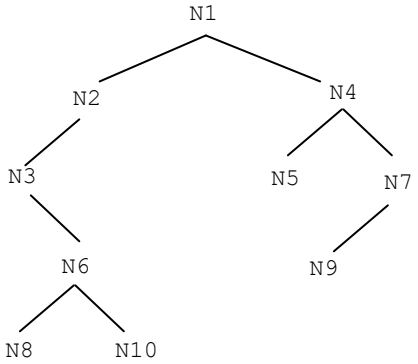


1.6. Parcours des arbres binaires

Le parcours d'un arbre binaire consiste à examiner systématiquement, dans un certain ordre, chacun des nœuds de l'arbre pour y effectuer un certain traitement.

```
void traverse(tree t){
       if (t == NULL){ terminate; }
       else {
              processing_1;
              traverse(t->left);
              processing_2;
              traverse(t->right);
              processing_3;
       }
}
```

Considérons l'exemple (arbre) suivant :



Ordre préfixe ou pré-ordre (RGD) : On examine d'abord la racine suivie de gauche à droite de chaque sous-arbre. Par rapport au sous-programme précédent, processing_2 et processing_3 n'existent pas.

- Résultat du parcours `preorder()` : `n1, n2, n3, n6, n8, n10, n4, n5, n7, n9.`

Ordre infixe ou symétrique (GRD) : On examine le sous-arbre gauche suivi par sa racine et le sous-arbre droit restant. processing_1 et processing_3 n'existent pas.

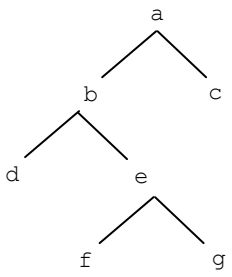
- Résultat du parcours `inorder()` : `n3, n8, n6, n10, n2, n1, n5, n4, n9, n7.`

Ordre suffixe ou post-ordre (GDR) : On examine de gauche à droite chaque sous-arbre et enfin la racine. processing_1 et processing_2 n'existent pas.

- Résultat du parcours `postorder()` : `n8, n10, n6, n3, n2, n5, n9, n7, n4, n1.`

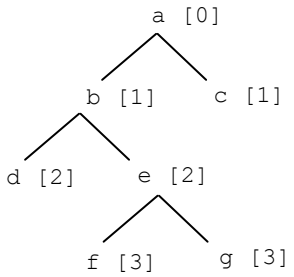
1.7. Exemples d'application

Exercice 1 : écrire un sous-programme *itératif* qui parcours un arbre binaire selon le pré-ordre (RGD).



Preorder : `a, b, d, e, f, g, c`
Taille de l'arbre : `7`

Exercice 2 : écrire un sous-programme qui calcule la taille d'un arbre binaire.
Exercice 3 : écrire un sous-programme qui calcule la hauteur d'un nœud dans un arbre binaire.
Exercice 4 : écrire un sous-programme qui calcule la hauteur d'un arbre binaire.



Hauteur du nœud `c` : `1`

Hauteur de l'arbre : `3`

Solution 1 : parcours itératif d'un arbre binaire.

```
struct nodeTree{
	element value;
	nodeTree* left;
	nodeTree* right;
};

typedef nodeTree* binaryTree;

// recursive version  of preorder traversal
void preorder(binaryTree bt){
	if (bt != NULL){
		treat(bt->value);
		preorder(bt->left);
		preorder(bt->right);
	}
}
```

```
// iterative version of preorder traversal
void preorder(binaryTree bt){
	Stack s;
	initializeStack(s);

	while(bt!= NULL or !isEmpty(s)){
		while(bt != NULL){
			treat(bt->value);
			push(s, bt);
			bt = bt->left;
		}

		if (!isEmpty(s)){
			bt = top(s);
			bt = bt->right;
			pop(s);
		}
	}
}
```

À faire : écrire des sous-programmes itératifs qui réalisent le parcours d'un arbre binaire selon les deux autres modes.

Solution 2 : taille d'un arbre binaire.

```
int size(binaryTree bt){
       if (bt == NULL){ return 0; }
       else { return 1 + size(bt->left) + size(bt->right); }
}
```

Solution 3 : hauteur d'un nœud dans un arbre binaire.

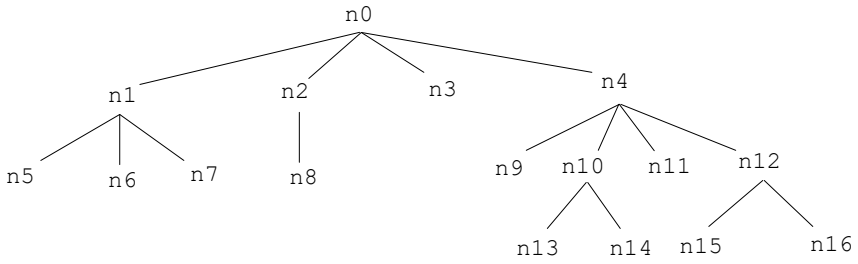
```
int nodeHeight(binaryTree bt, element n){
       return nodeHeightHelper(bt, n, 0);
}

int nodeHeightHelper(binaryTree bt, element n, int level){
       if (bt == NULL){ return -1; }

       if (bt->value == n){ return level; }

       int height = nodeHeightHelper(bt->left, n, level + 1);
       if (height == -1) { height = nodeHeightHelper(bt->right, n, level + 1); }

       return height;
}
```

Solution 4 : hauteur d'un arbre binaire.

```
int height(binaryTree bt){
       if (b == NULL){ return -1; }
       else {
              int h1 = height(bt->left);
              int h2 = height(bt->right);

              if (h1 > h2){ return h1 + 1; }
              else { return h2 + 1; }
       }
}
```

2. Arbres planaires généraux

On présente dans cette section une structure arborescente plus large, appelée arbre planaire général ou brièvement arbre général ou arbre, où le nombre de fils de chaque nœud n'est pas limité à deux.



2.1 Définition

Un arbre A = <o, A1, A2, ..., Ap> est la donnée d'une racine et d'une liste finie, éventuellement vide (si p = 0), d'arbres disjoints.

Une liste finie éventuellement vide d'arbres disjoints est appelée une forêt. Un arbre est donc obtenu en ajoutant une racine à une forêt.

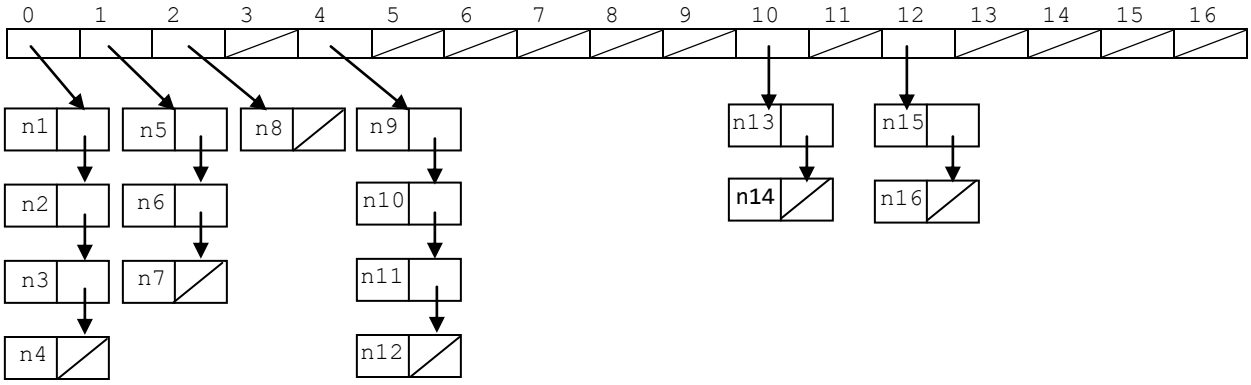
Remarques

- Il n'y a pas de notion gauche-droite dans les arbres.
- Par analogie avec les arbres binaires, la taille d'une forêt est le nombre total des nœuds de tous les arbres qui la composent.

2.2 Représentation des arbres planaires

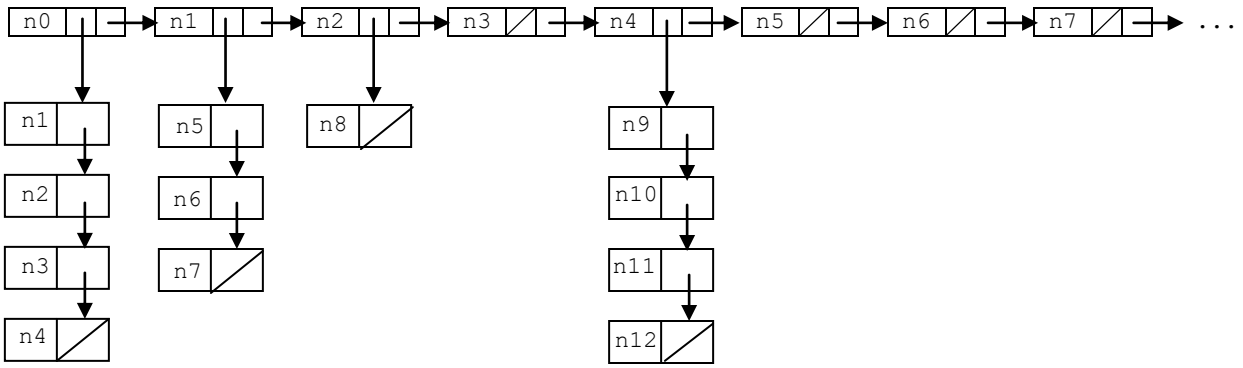
Représentations simples

Différentes façons d'implémenter un arbre sont envisageables. Par exemple, on peut donner pour chaque nœud la liste de ses fils.

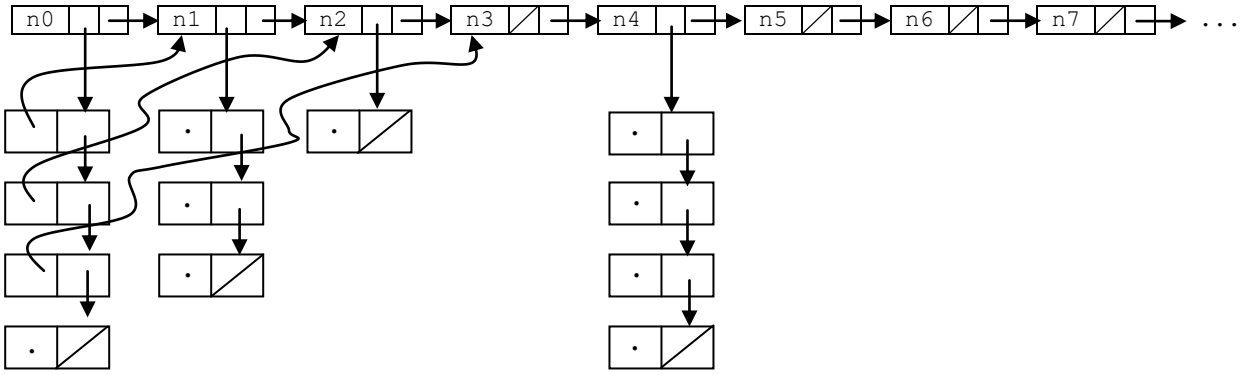



```
const int N = -;

typedef struct node* ptr;
struct node {
   element value;
   ptr next;
};

typedef ptr tree[N];
```

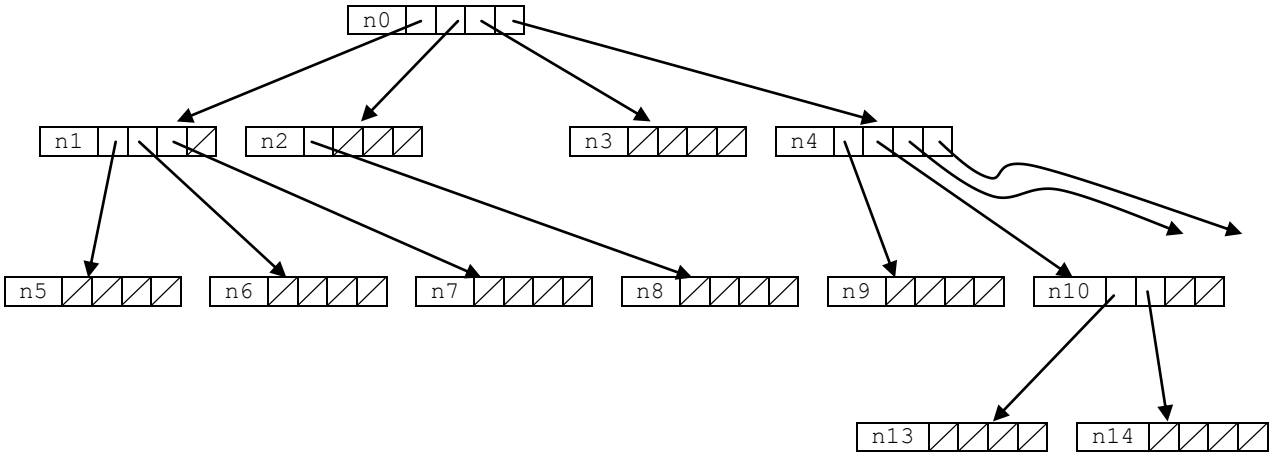
Cette représentation se prête à une gestion dynamique.



Ou



On peut aussi décrire une représentation analogue à celle des arbres binaires où chaque nœud contient un pointeur vers chacun des sous-arbres, et éventuellement un champ pour stocker l'élément contenu dans le nœud.



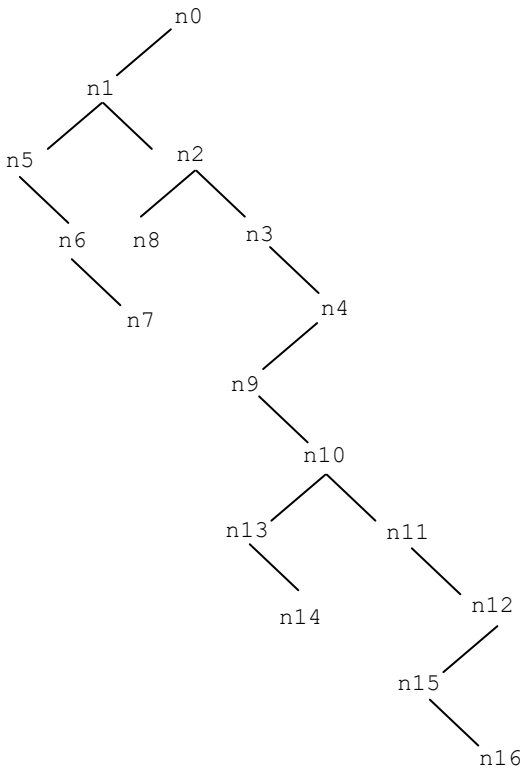
Cette représentation est très gourmande en espace mémoire. C'est le nombre maximal de fils d'un nœud qui détermine le nombre de pointeurs dans tous les nœuds. Dans l'exemple précédent, le nombre de fils ne dépassera jamais 4.

```
typedef struct node* tree;
struct node {
   element value;
   tree child1;
   tree child2;
   tree child3;
   tree child4;
};
```

Représentation sous forme d'arbres binaires

Cette section est consacrée à une représentation très utilisée des arbres généraux sous forme d'arbres binaires. Pour établir une correspondance entre les arbres planaires généraux et les arbres binaires, on utilise la méthode suivante : bijection fils aîné, frère droit. Plus exactement, pour obtenir l'arbre binaire associé à une forêt, on construit pour chaque nœud un lien gauche vers son premier fils (fils aîné), et un lien droit vers son frère situé immédiatement à sa droite dans la forêt. Notez qu'on considère que les racines des différents arbres de la forêt sont des frères.

Le schéma suivant montre l'arbre binaire associé à la forêt de l'exemple précédent.



2.3 Exercices

1. En prenant une représentation pour un arbre planaire, écrire :
   - Un sous-programme qui transforme un arbre planaire en arbre binaire.
   - Un sous-programme qui fait l'inverse.
2. Écrire un sous-programme qui calcule la hauteur d'un nœud dans un arbre planaire.
3. Écrire un sous-programme qui calcule la hauteur d'un arbre planaire