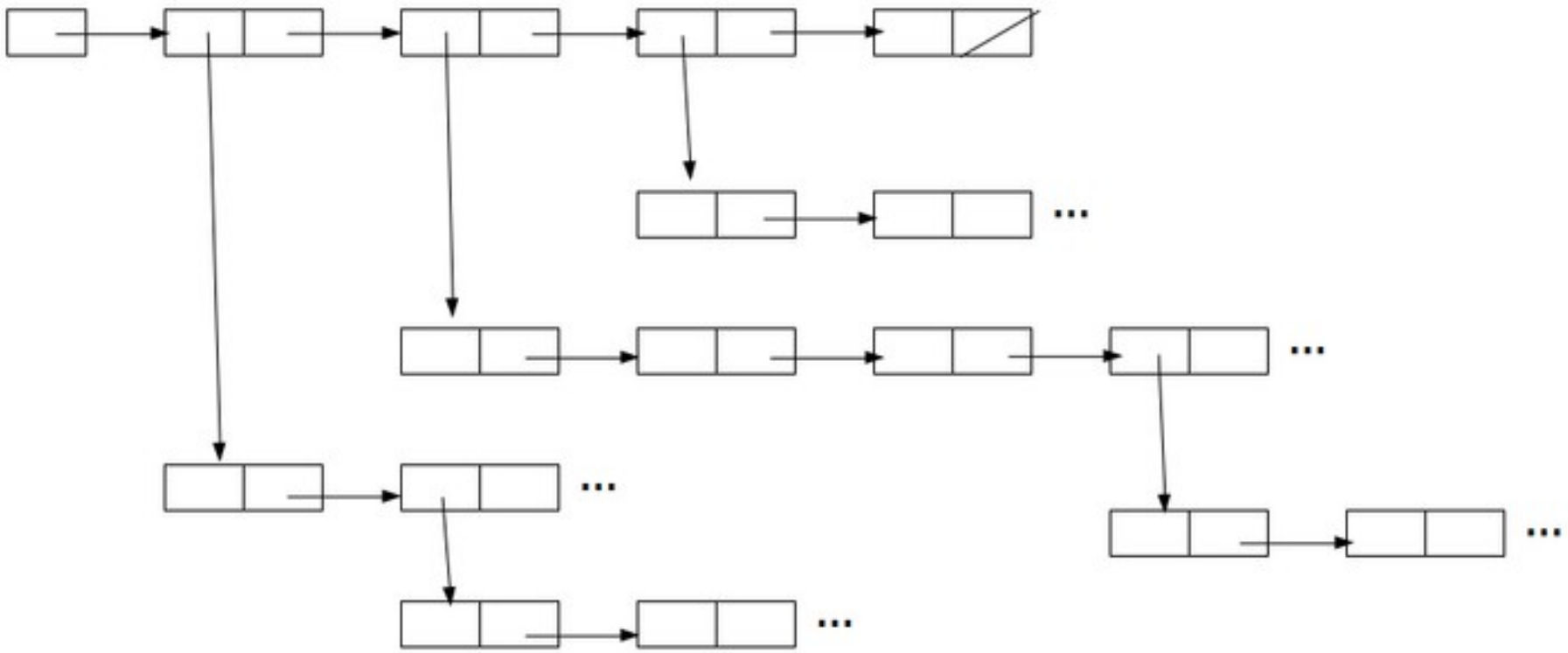
Structures chaînées – Arborescence - Arbre

Introduction

Plusieurs façons de présenter les arbres (et les arborescences), plus ou moins formelles.
On peut voir cela comme un généralisation de listes chaînées qui porteraient comme informations des listes :



De manière plus intuitive on peut dire qu'un arbre est la modélisation d'une organisation hiérarchique de données :
- organigramme d'un service, d'une entreprise, d'un armée :
    - un maréchal M
        - un général G1
            - un colonel C1
                - un lieutenant L1
                    - un sergent S1
                        - un troufion T1
                        - un troufion T2
                        - un troufion T3
                    - un sergent S1
                        - un troufion T1
                        ...
                    - un sergent S1
                        ...
                    ...
                - un lieutenant L2
                    - un sergent S1
                        ...
                    - un sergent S1
                        ...
                …
            - un colonel C2
              ...
        - un général G2
          ...
        - un général G3
          ...
        - un général G4
          ...

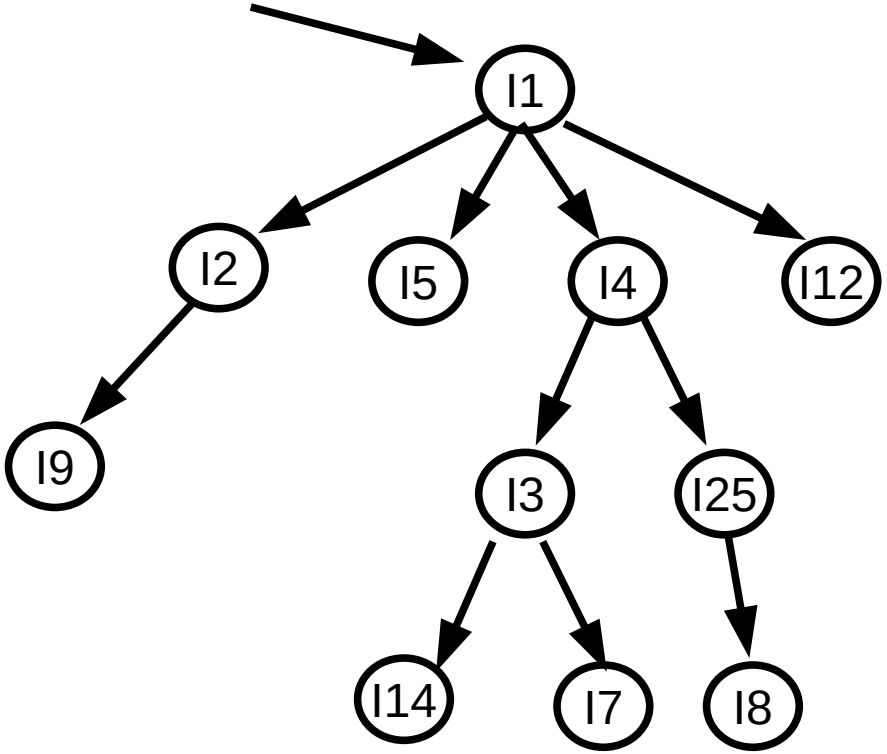
- un livre est découpé en volumes, eux-mêmes découpés en tomes, découpés en parties, découpées en chapitres, découpés en sections, …

- etc.

De manière plus formelle on va définir un arbre comme étant une structure de données à laquelle on accède par une et une seule donnée, qui donne accès à son information et qui fournit l'accès à plusieurs informations suivantes, qui elles-mêmes donnent accès à plusieurs informations, etc.

A est un Arbre si A est un ensemble tel que :
- il existe un élément particulier appelé racine,
- les autres éléments sont partitionnés en ensembles, chaque ensemble étant lui-même un arbre.

Au final on représentera un arbre ainsi :



Les flèches représentent le sens d'accès à l'information :

- on n'accède directement qu'à l'information I1,
- celle-ci donne accès aux informations I2, I4, I5 et I12,
- I2 donne accès à I9,
- etc.

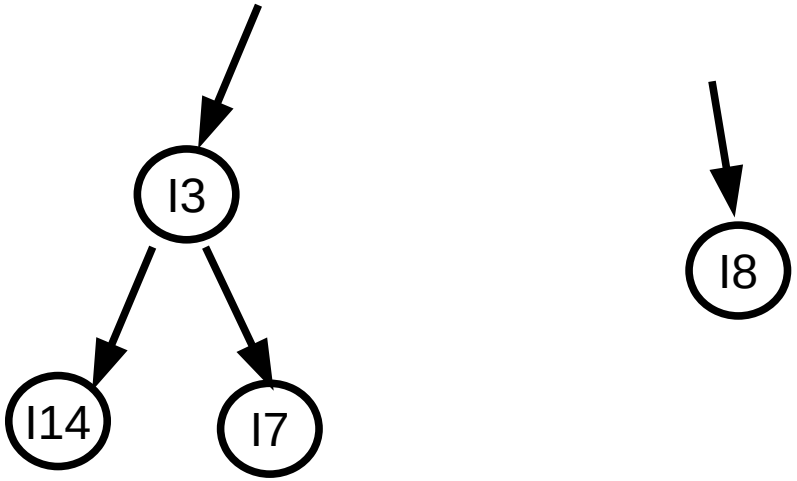
Arbre ou arborescence ?

On confond souvent les arbres et les arborescences, la différence tient dans l'orientation ou non des branches, les branches des arborescences étant orientées et pas celle des arbres. Ainsi le dessin de la page précédente est une arborescence, mais par abus de langage on parle quasi tout le temps d'arbre.

Un peu de vocabulaire :

- Nœud : on appelle nœuds les cellules qui portent les informations de l'arbre.
- Racine : 1er nœud de l'arbre, celui qui donne accès à tous les autres. I1 sur le dessin.
- Sous-arbre :

  un sous-arbre est un sous ensemble d'un arbre qui constitue lui-même un arbre :
  I3, I14 et I17 constitue un sous arbre,
  I8 constitue un sous arbre.



- Fils : on qualifie de fils d'un nœud, les nœuds accessible en suivant une flèche à partir de ce nœud.
- Père : nœud qui donne accès à ses fils.
- Frère : nœud ayant le même père.
- Feuille : nœud sans fils.

Quelques caractéristiques :

- Degré d'un noeud : nombre de fils d'un nœud.
- Profondeur d'un nœud : distance d'un nœud à la racine (en nombre de flèche).
- Hauteur d'un arbre : maximum des profondeurs des nœuds de l'arbre. Un arbre vide est de hauteur 0.

Arbres ordonnés :

Un arbre est ordonné si il existe une relation d'ordre définie sur tous les fils de tous ses nœuds. Les arbres sont très souvent ordonnés.

Arbres binaire

Les arbres binaires sont des arbres dont tous les nœuds ont au plus 2 fils. Ce sont les arbres les plus utilisés.

On peut leur donner une définition récursive :
Un arbre binaire est :

- soit un arbre vide,
- soit un triplet constitué d'une information et de 2 arbres binaires <I, G, D>

I représente l'information et G et D représente arbres binaires nommés généralement fils gauche et fils droit.

Implémentation des arbres binaires

Pour représenter des arbres binaires dans un programme en C++, nous allons utiliser les structures suivantes :

- une classe qui représentera les nœuds de l'arbre, portant une information quelconque (ici un int) :

```
class noeud
{
  int info ;
  noeud * fg, * fd ;

  public :
  ...

} ;
```

- une classe arbre qui donne accès au 1er nœud de l'arbre :

```
class arbre
{
  noeud * racine ;

  public :
  ...

} ;
```

On retrouve bien ici la récursivité de la définition précédente, où la classe **noeud** est définie par elle-même.

Les données membres de la classe **noeud** étant privées, il peut être utile de déclarer amie la classe **arbre** en ajoutant dans la classe **noeud** :

```
friend class arbre ;
```

Implémentation de quelques fonctions des classes nœud et arbre

Un arbre vide est un arbre qui ne contient pas de nœud, c'est à dire dont la racine « pointe sur rien », on aura donc un constructeur d'arbre vide de la forme :

```
arbre::arbre()
{
  racine = NULL ;
}
```

On peut ajouter un constructeur d'arbre recevant un pointeur sur un nœud :

```
arbre::arbre(noeud * N)
{
  racine = N ;
}
```

Bien sûr ces 2 constructeurs peuvent être remplacés par un seul en utilisant une valeur par défaut :

```
arbre::arbre(noeud * N = NULL)
{
  racine = N ;
}
```

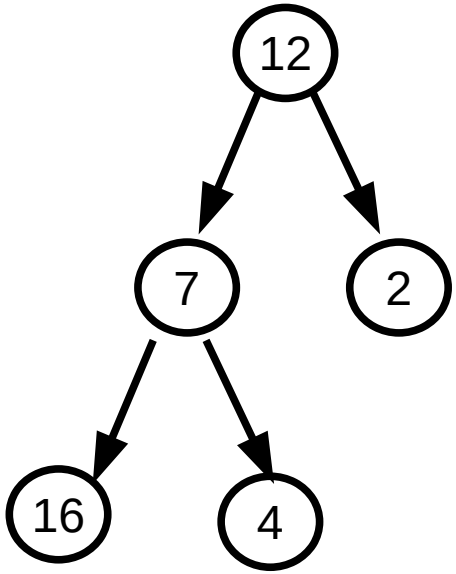
Un constructeur de nœud :

```
noeud::noeud(int I, noeud * G = NULL, noeud * D = NULL)
{
  info = I ;
  fg = G ;
  fd = D ;
}
```

Ces constructeurs permettent d'initialiser des arbres lors de leur déclaration, l'arbre créé ici :

```
Arbre A( new noeud ( 12,
                        new nœud ( 7,
                                   new noeud ( 16 ),
                                   new noeud ( 4 )
                                  ) ,
                         new noeud ( 2 )
                       )
       ) ;
```

correspond à :



Parcours des arbres

On appelle parcours d'un arbre le fait d'énumérer en vue d'un traitement quelconque toutes les informations contenues dans un arbre. Il peut par exemple s'agir de les afficher.

La structure récursive des arbres donne une façon simple de réaliser ce parcours, il suffit de s'aligner sur la structure de l'arbre, portée par la classe nœud :

Définition récursive d'un arbre : un arbre c'est :	Contenu du nœud :	Action à faire dans le parcours :
• Soit un arbre vide	• NULL	• Rien
• Soit une information et 2 arbres	• <I, G, D>	• Traiter le nœud (et seulement le nœud) : ◦ Traiter l'information ◦ Parcourir le fils gauche ◦ Parcourir le fils droit

La programmation découle de cette analyse et repose sur une fonction récursive dont le paramètre est un pointeur sur un nœud.

Exemple avec l'affichage de l'information portée par le nœud comme traitement :

```
    void Parcours(noeud * N)
    {
      if (N == NULL)
        return ;
      else
      {
        cout << N -> info <<
endl ;
        Parcours(N -> fg) ;
        Parcours(N -> fd) ;
      }
    }
```

arbre vide	rien faire
une information et 2 arbres	traiter l'information parcourir le fils gauche parcourir le fils droit

Cette fonction pose un problème lié au fait que les données de la classe **noeud** sont privées et donc il est interdit à cette fonction de les manipuler. Ce point peut être réglé en la déclarant amie, c''est à dire en ajoutant dans la classe la déclaration :

```
    friend void Parcours(noeud *) ;
```

Mais cela ne règle pas tout, en effet pour appeler cette fonction il va falloir lui donner un pointeur sur un arbre, par exemple la racine de l'arbre précédent :

```
    Parcours(A.racine) ;
```

Et là aussi on a un problème car **racine** est une donnée privée de la classe **arbre** ! Et on ne peut pas régler ce problème à priori, car dans ce cas il faut que la fonction qui réalise cet appel ait le droit de le faire, c'est à dire soit elle-même une fonction membre de **arbre** ou soit déclarée amie de **arbre** !

D'où une proposition plus « objet » : mettre la fonction de parcours directement dans la classe arbre, donc en faire une fonction membre de la classe :

```
void arbre::Parcours(noeud * N)
{
  if (N == NULL)
    return ;
  else
  {
    cout << N -> info << endl ;
    Parcours(N -> fg) ;
    Parcours(N -> fd) ;
  }
}
```

Dans ce cas l'appel se fera en utilisant un arbre en argument implicite :

```
A.Parcours(A.racine) ;
```

La nécessité de fournir un pointeur de **noeud** en argument pose toujours des problèmes, d'où l'ajout d'une 2e fonction membre permettant d'appeler la 1re :

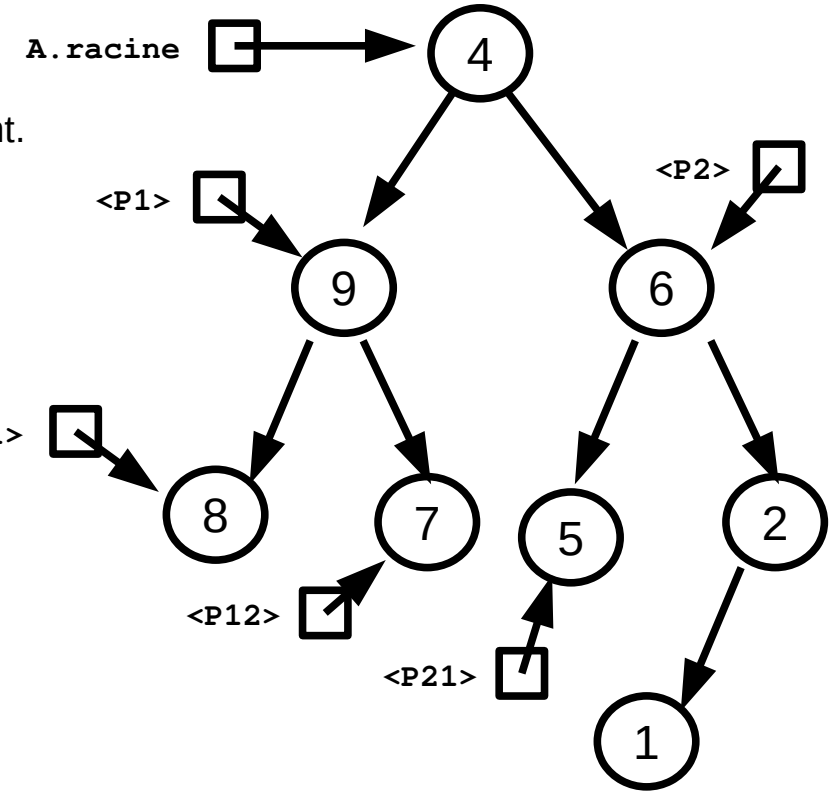
```
void arbre::Parcours()
{
  Parcours(racine) ;
}
```

Au final l'appel se fait ainsi :

```
A.Parcours() ;
```

Fonctionnement de cette fonction

Exemple sur cet arbre :



Les lignes de la fonction sont numérotées pour bien suivre le déroulement.
On en profite pour enlever le **`else`** inutile

```
      void arbre::Parcours(noeud * N)
      {
(1)     if (N == NULL)
          return ;
(2)     cout << N -> info << endl ;
(3)     Parcours(N -> fg) ;
(4)     Parcours(N -> fd) ;
      }
```

L'appel de **`A.Parcours()`** provoque l'appel de **`Parcours(racine)`**

(1) **`N`** (qui vaut la racine de A au 1er appel) n'est pas **`NULL`** donc
- (2) affichage de 4
- (3) appel de **`Parcours(N -> fg <P1>)`**
  - (1) **`<P1>`** n'est pas **`NULL`** donc
    - (2) affichage de 9
    - (3) appel de **`Parcours(<P11>)`**
      - (1) **`<P11>`** n'est pas **`NULL`** donc
        - (2) affichage de 8
        - (3) appel de **`Parcours(NULL)`** **`return`**
        - (4) appel de **`Parcours(NULL)`** **`return`**
    - (4) appel de **`Parcours(<P12>)`**
      - (1) **`<P12>`** n'est pas **`NULL`** donc

(2) affichage de 7
(3) appel de **`Parcours(NULL)`**
(4) appel de **`Parcours(NULL)`**

(4) appel de **`Parcours(N -> fd <P2>)`**
(1) **`<P2>`** n'est pas **`NULL`** donc
(2) affichage de 6
(3) appel de **`Parcours(<P21>)`**
(2) affichage de 5
(3) appel de **`Parcours(NULL)`**
`return`
(4) appel de **`Parcours(NULL)`**
`return`
. . .
(2) affichage de 2
. . .
(2) affichage de 1
. . .

affichage dans l'ordre :
4
9
8
7
6
5
2
1

Dans cet exemple, la racine de chaque sous-arbre est traitée avant ses fils, il y a d'autres possibilités, il y a 3 ordres possibles pour les parcours (si on exclut les inversions des fils) :

- traitement de la racine puis des fils : parcours **préfixe**
- traitement des fils puis de la racine : parcours **postfixe**
- traitement du fils gauche, de la racine, du fils droit : parcours **infixe**

qui correpondent aux 3 fonctions qui ne diffèrent que par l'ordre des lignes :

Préfixe

```
void Parcours(noeud * N)
   {
     if (N == NULL)
       return ;

     /* Traitement */

     Parcours(N -> fg) ;
     Parcours(N -> fd) ;
   }
```

Infixe

```
void Parcours(noeud * N)
   {
     if (N == NULL)
       return ;
     Parcours(N -> fg) ;

     /* Traitement */

     Parcours(N -> fd) ;
   }
```

Postfixe

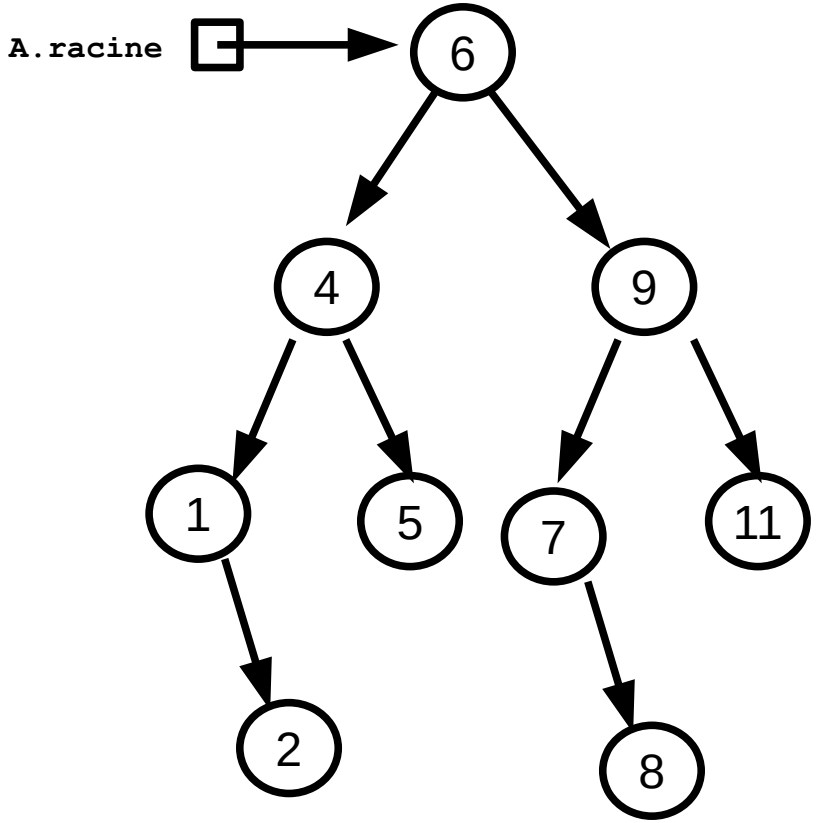
```
void Parcours(noeud * N)
   {
     if (N == NULL)
       return ;
     Parcours(N -> fg) ;
     Parcours(N -> fd) ;

     /* Traitement */

   }
```

Cas des arbres (binaires) ordonnés

Exemple de relation d'ordre ordonnant un arbre :
pour tous les nœuds de l'arbre,

- tous les nœuds du fils gauche sont inférieures à la racine,
- la racine est inférieure à tous les nœuds du fils droit.

Par exemple :



Un parcours infixe de cet arbre va parcourir les nœuds dans cet ordre :

1 2 4 5 6 7 8 9 11