Chapitre 4
Les arbres

- Partie I: Les arbres
- Partie II: Les arbres binaires
- Partie III: Les arbres binaires de recherche

Références

1. «Programmer en Langage C cours et exercices»
Auteur: Claude Delannoy
Code bibliothèque: I-22/DEL
Edition: Eyrolles 5ème Edition (2012)

2. «Exercices et Problème d'Algorithmes»
Auteurs: Bruno Baynat et collaborateurs
Code bibliothèque: I-15/BAY
Edition: Dunod (2003)

3. «Types de Données et Algorithmes»
Auteur: Christine Froidevaux et collaborateurs
Code bibliothèque: I-1/FRO
Edition: McGraw-Hill (1990)

Partie I: Les arbres

Introduction

- Par nature certaines données sont **arborescentes,** voici quelques exemples:
  - Le découpage d'un livre en chapitres, sections, etc;
  - Les organisations hiérarchiques;
  - Les répertoires sous la plupart des systèmes d'exploitation actuels (Windows, Unix , Linux, ...).

- **Donald Knuth**, auteur de l'ouvrage de référence sur l'algorithmique, en plusieurs volumes, intitulé «**The Art of Computer Programming**», considère les **arbres** comme «**les structures de données les plus importantes en informatique**».

- Les **arbres** sont des structures **non linéaires** qui permettent d'obtenir des algorithmes les plus performants.

Intérêt et applications

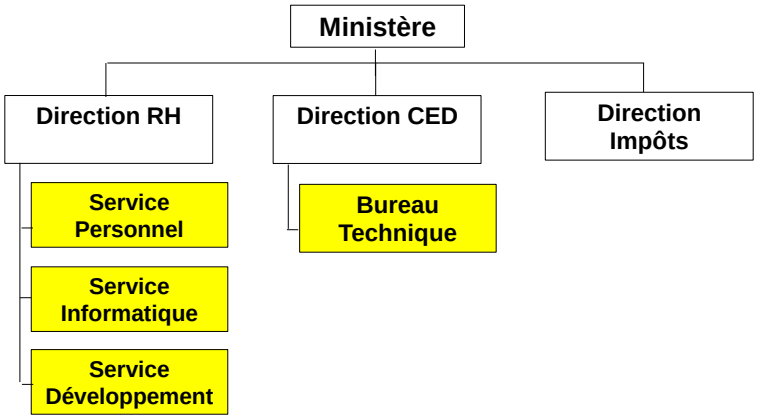
- **Intérêt des arbres:**
  - **Type Abstrait de Données** ou **TAD** (en anglais **Abstract Data Type** ou **ADT**), hiérarchiques;
  - Sont à la base de plusieurs **TAD** avancés: Arbres de recherche, Arbres de décision, Dictionnaires, etc ...

- **Applications des arbres:**
  - Structures hiérarchisées: arbres familiales, organisations hiérarchiques, expressions algébriques,
  - Compilation (arbres syntaxiques),
  - Intelligence artificielle: arbres de décision,
  - Algorithmique: tris, recherche,
  - etc….

Intérêt et applications

■ Exemple 1: Administration

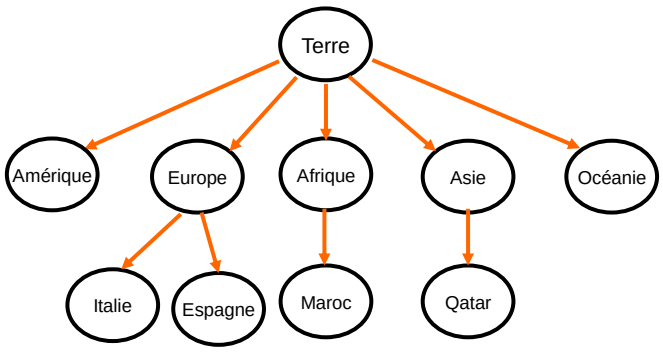
- Voici une représentation d'une administration ministérielle sous forme d'arbre:



Intérêt et applications

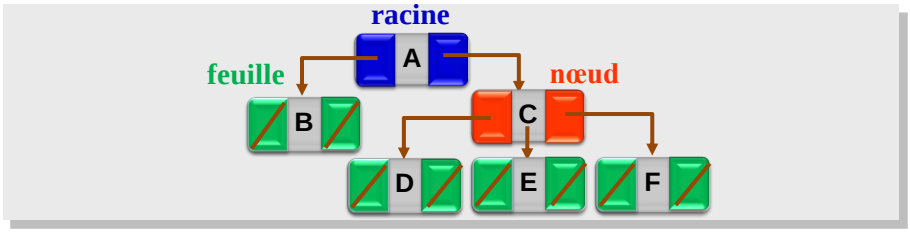
■ Exemple 2: Les cinq continents

- Voici une représentation des cinq continents de la terre sous forme d'arbre :



Définition d'un arbre

- Voyez d'abord cette figure:



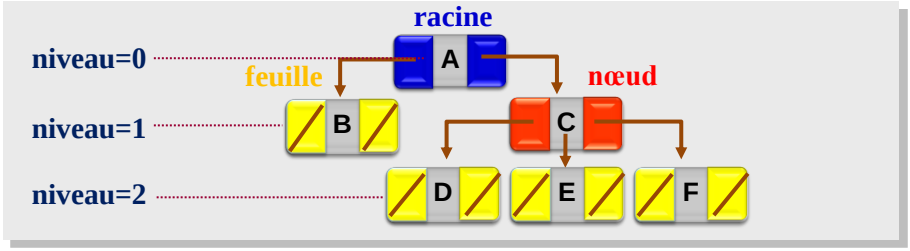
- Un **arbre** est une structure de données composée d'un ensemble **fini** de **nœuds** reliés par des **arêtes**.
- On distingue trois sortes de nœuds :
  - La **racine** : qui ne possède pas de père,
  - Les **nœuds internes**: qui ne sont pas des racines et qui ont des fils,
  - Les **feuilles**: qui n'ont pas de fils.

Remarque

- Un arbre est **connexe** car tout nœud est accessible depuis la racine.
- Un arbre est **sans cycle** car sur tout chemin, on traverse un nœud une et une seule fois.

Vocabulaire: niveau et degré

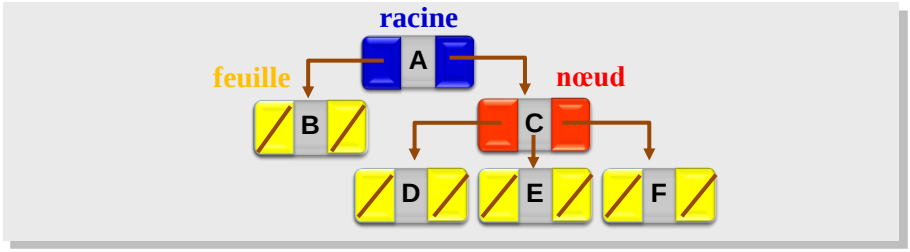
- Voyez encore la figure suivante:



- Le **niveau** d'un nœud est la taille du chemin le séparant de la racine. Dans la figure ci-dessus, **niveau(E)=2**
- Le **degré** d'un nœud est le nombre de ses successeurs directs (fils). Dans la figure ci-dessus, **degré(C) = 3**
- Le **degré** d'un arbre est le **degré maximum** de ses nœuds. Dans la figure ci-dessus, **degré(arbre) = 3**

Vocabulaire: chemin

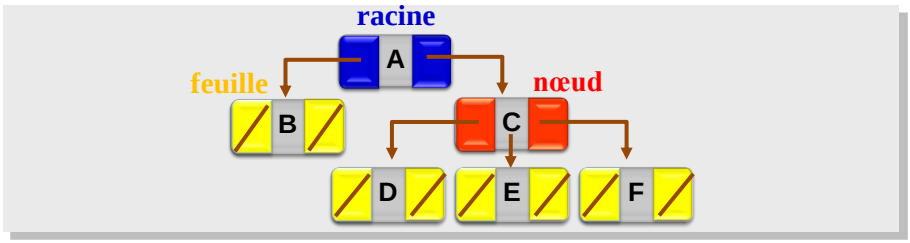
- Voyez encore la figure suivante:



- Une **séquence de nœud** partant du nœud "**A**" en suivant l'orientation des arcs jusqu'au nœud "**E**" s'appelle un **chemin** de "**A**" à "**E**".
- La **langueur d'un chemin** est le nombre d'arcs sur le chemin.
- Il existe exactement un seul **chemin** de la racine à chaque nœud de l'arbre.

Vocabulaire: taille et profondeur

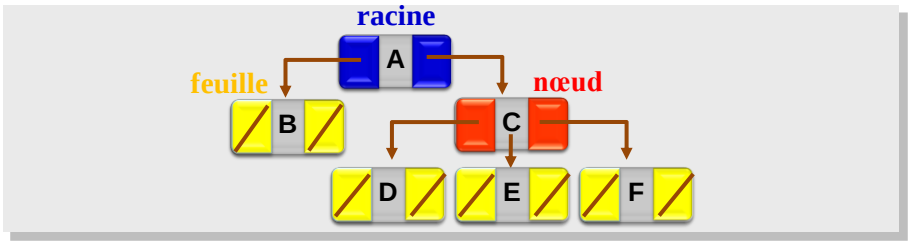
- Voyez encore cette figure:



- La **taille** d'un arbre est le nombre total de nœuds qui le compose. Dans la figure ci-dessus, **taille(arbre) = 6**
- On appelle également la **profondeur** d'un nœud la distance en terme de nœud par rapport à l'origine (la racine).
- Par convention, la racine est de profondeur 0. Dans la figure ci-dessus, **profondeur(E) = 2** et **profondeur(B) = 1**

Vocabulaire: hauteur

- Voyez encore cette figure:

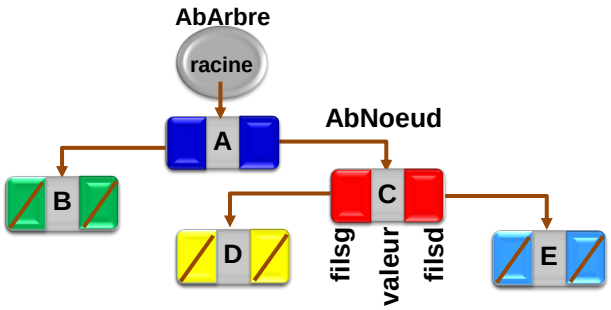


- La **hauteur** d'un nœud est la longueur du plus long chemin partant de ce nœud et aboutissant à une feuille. Dans la figure ci-dessus, **hauteur(C) = 1**, **hauteur(A) = 2** et **hauteur(B) = 0**.
- La **hauteur** d'un arbre est la hauteur de sa racine. Dans la figure ci-dessus, **hauteur(arbre) = 2**.

Partie II: Les arbres binaires

Définition

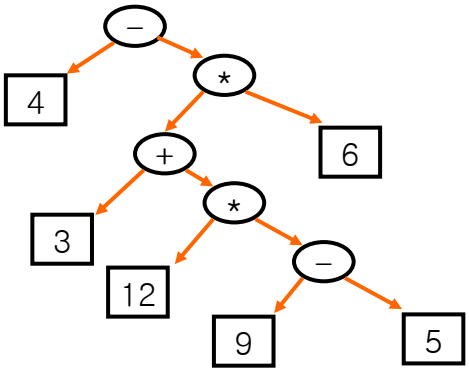
- Voyez la figure suivante:



- Un arbre binaire est un arbre dont chaque nœud admet **au plus deux pointeurs** vers d'autres nœuds.
- Chaque nœud connaît sa valeur stockée **valeur** et ses nœuds dans l'arbre **fils gauche** (**filsg**) et **fils droit** (**filsd**).

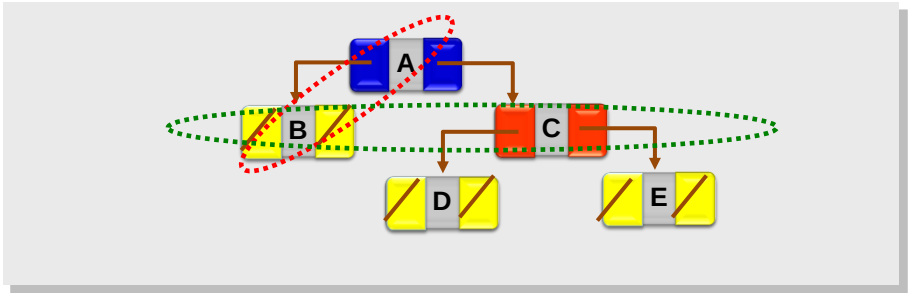
Applications

- Exemple : Expressions algébriques
  - Voici une représentation de l'expression algébrique **4-(3+12*(9-5))*6** sous forme d'arbre binaire en respectant certaines règles de précédence:



Parcours d'un arbre binaire

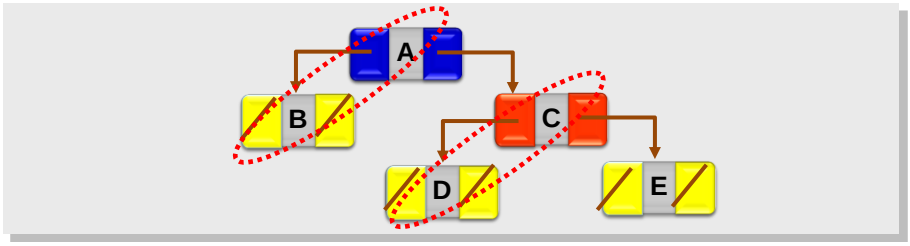
- Voyez le schéma suivant:



- Il existe **deux** catégories de parcours d'un arbre :
  - Les parcours en **profondeurs** effectuent une exploration branche par branche,
  - Le parcours en **largeurs** effectuent une exploration niveau par niveau.

Parcours en profondeurs

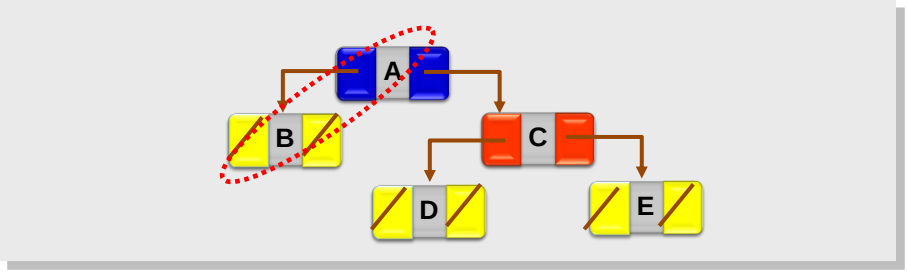
- Voyez le schéma suivant:



- Il existe 6 types de parcours **en profondeurs**: 3 parcours de types **gauche-droite** et 3 parcours de types **droite-gauche.**
- Dans le cas des parcours **gauche-droite**, les trois types de parcours sont :
  - **Préfixe** ou **RGD**: **Racine–filsG–filsD**
  - **Infixe** ou **GRD**: **filsG–Racine–filsD**
  - **Postfixe** ou **GDR**: **filsG–filsD–Racine**

Parcours en profondeurs

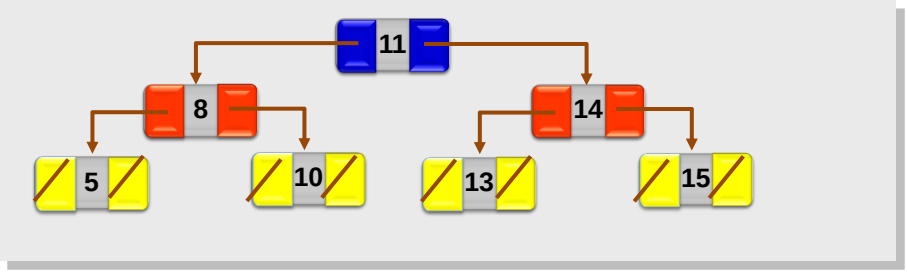
- Exemple 1: Afficher l'information de cet arbre ?



- **Préfixe** ou **RGD:** (Racine–filsG–filsD): ABCDE
- **Infixe** ou **GRD:** (filsG–Racine–filsD): BADCE
- **Postfixe** ou **GDR:** (filsG–filsD–Racine): BDECA

Parcours en profondeurs

- Exemple 2: Afficher l'information de cet arbre ?



Préfixe ou **RGD:** 11,8,5,10,14,13,15

Infixe ou **GRD:** 5,8,10,11,13,14,15

Postfixe ou **GDR:** 5,10,8,13,15,14,11

Structure d'un nœud et structure de contrôle de l'arbre

- Voyez la définition de la structure d'un nœud :

Structure d'un nœud de l'arbre

```
typedef int Ttype; //on définit généralement un type ici un int
typedef struct AbNoeudRepere{
   Ttype valeur;
   struct AbNoeudRepere *filsg; //pointeur sur le fils gauche
   struct AbNoeudRepere *filsd; //pointeur sur le fils droit
} AbNoeud;
```

- Voyez aussi la définition de la structure de contrôle :

Structure de contrôle de l'arbre

```
typedef struct AbArbreRepere{
    AbNoeud *racine;
}AbArbre;
```


Créer une feuille de l'arbre

- Nous aurons besoin de la fonction **CréerFeuille()**.

Fonction CreerFeuille()

```
AbNoeud *CreerFeuille(Ttype val){
  AbNoeud *Fe;
  Fe = (AbNoeud*)malloc(sizeof(AbNoeud));
  Fe->valeur=val;
  Fe ->filsg=NULL;
  Fe ->filsd=NULL;
  return(Fe);
}
```

Créer un nœud de l'arbre

- Nous aurons besoin de la fonction **CréerNœud()**

Fonction CreerNoeud()

```
AbNoeud *CreerNoeud(Ttype val, AbNoeud *filsg, AbNoeud *filsd){
  AbNoeud *No;
  No =(AbNoeud*)malloc(sizeof(AbNoeud));
  No->valeur=val;
  No->filsg=filsg;
  No->filsd=filsd;
  return(No);
}
```


Créer une structure de contrôle de l'arbre

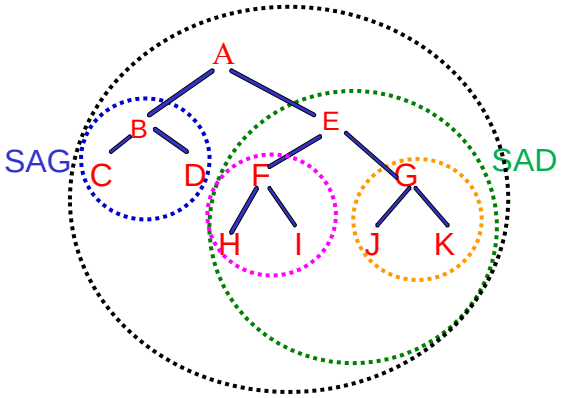
- Nous aurons besoin aussi de la fonction **CreerAbArbre()**

Fonction CreerAbArbre()

```
AbArbre *CreerAbArbre(){
  AbArbre *Ar;
  Ar=(AbArbre*)malloc(sizeof(AbArbre));
  Ar -> racine=NULL;
  return(Ar);
}
```

Arbres et récursivité

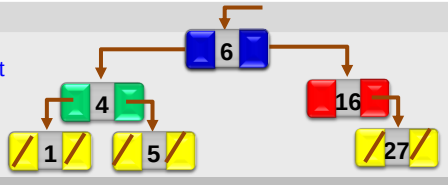
- Les arbres binaires ont une définition **récursive** naturelle.



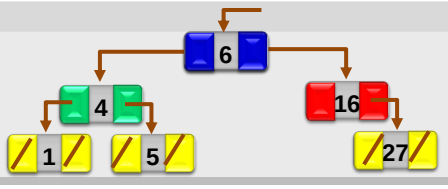
- Nous nous basons sur **la récursivité** pour :
  - Créer un arbre
  - Insérer des nœuds
  - Parcourir un arbre, …

Parcours en profondeurs d'arbre binaire

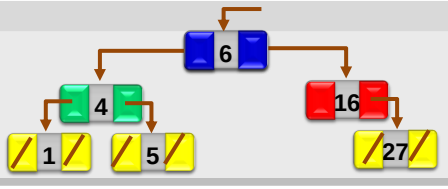
```
void Prefixe(AbNoeud *racine){
 if(racine != NULL){  //arbre non vide
   printf("%d ", racine->valeur); //traitement
   Prefixe(racine -> filsg);  //appel récursif
   Prefixe(racine -> filsd);  //appel récursif
 }
}
```




```
void Infixe(AbNoeud *racine){
 if(racine != NULL){
   Infixe(racine -> filsg);
   printf("%d ", racine -> valeur);
   Infixe(racine -> filsd);
 }
}
```




```
void Postfixe(AbNoeud *racine){
 if(racine != NULL){
   Postfixe(racine -> filsg);
   Postfixe(racine -> filsd);
   printf("%d ", racine -> valeur);
 }
}
```

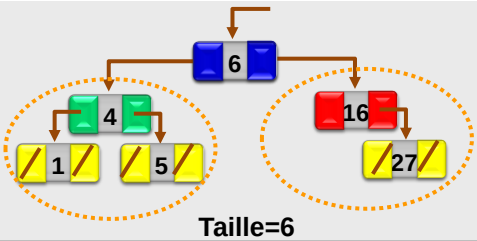


La taille d'un arbre binaire

■ Voici le code de la fonction **Taille()**:

Fonction Taille():

```
int Taille(AbNoeud *racine){
  if(racine == NULL)
    return 0;
  else
    return 1+Taille(racine->filsg)+Taille(racine->filsd);
}
```



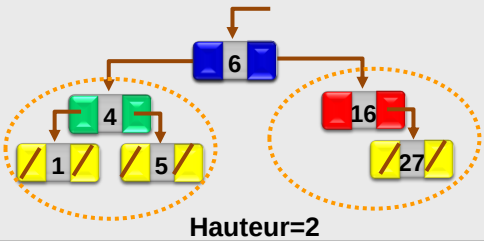
Taille=6

La hauteur d'un arbre binaire

■ Voici le code de la fonction **Hauteur()**:

Fonction Hauteur():

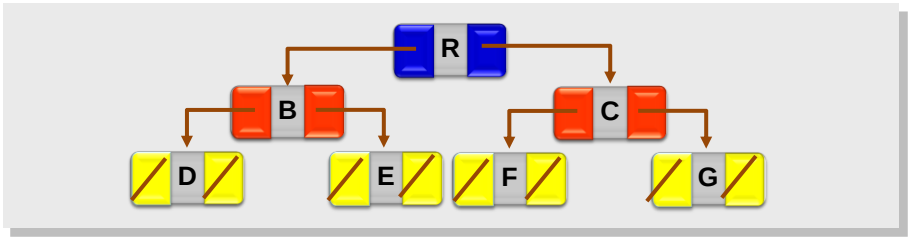
```
int Hauteur(AbNoeud *racine){
 if(racine == NULL)
    return 0;
 else
    return 1+max(Hauteur(racine->filsg),Hauteur(racine->filsd));
}
```



Hauteur=2

Arbre binaire complet

- Voyez encore cette figure:

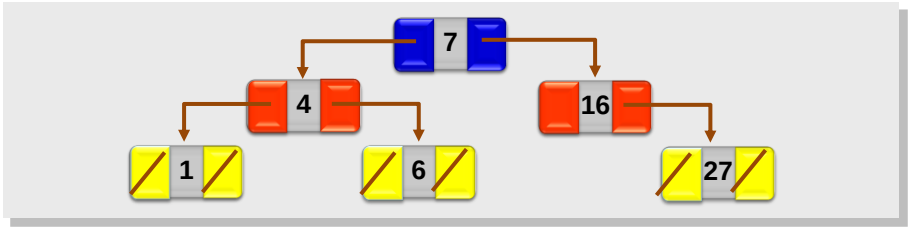


- Un arbre binaire est dit **complet** si :
  - Chaque nœud (non feuille) à exactement deux successeurs,
  - Toutes les feuilles sont au même niveau.
- Propriété :
  - Dans un arbre complet de **N feuilles** il y a **2N-1 noeuds**

Partie III
Les arbres binaires de recherche

Introduction

■ Voyez d'abord cette figure:



■ Un arbre binaire est dit **ordonné** (aussi arbre **de recherche**) si:

1) Les valeurs des nœuds de ses **sous arbres gauches** sont **inférieures ou égales** à la valeur de la **racine**,
2) Les valeurs des nœuds de ses **sous arbres droits** sont **supérieures** à la valeur de la **racine**.

Structure d'un nœud et structure de contrôle de l'arbre

■ Voyez la définition de la structure d'un nœud :

Structure d'un nœud de l'arbre

```
typedef int Ttype; //on définit généralement un type (int, float, struct, ...)
typedef struct AbrNoeudRepere{
   Ttype valeur;
   struct AbrNoeudRepere *filsg; //pointeur sur le fils gauche
   struct AbrNoeudRepere *filsd; //pointeur sur le fils droit
} AbrNoeud;
```

■ Voyez aussi la définition de la structure de contrôle :

Structure de contrôle de l'arbre

```
typedef struct AbrArbreRepere{
    AbrNoeud *racine;
}AbrArbre;
```

Ajouter un nœud dans l'arbre

- Pour ajouter un nœud dans l'arbre, voici une méthode basée sur les propriétés de l'arbre:
- Si l'arbre dans lequel on veut insérer le nœud est vide, alors il s'agit d'une création de l'arbre.
- Sinon, **on compare** la valeur du **nœud à insérer** avec la valeur de la **racine** et l'ajout se fait (récursivement):
  - Si la valeur du nœud est **inférieure** à celle de la racine alors, on l'insère dans le **sous arbre gauche** de la racine,
  - Sinon, on l'insère dans le **sous arbre droit** de la racine.

Remarque1

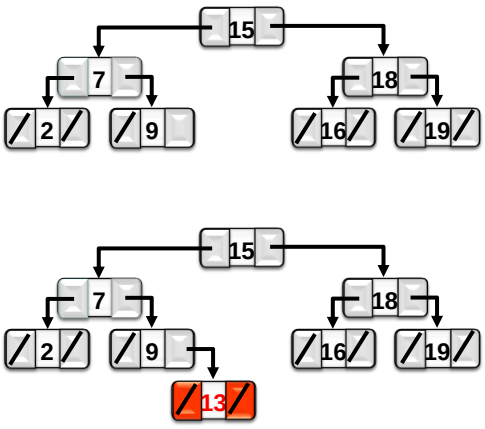
Si l'élément à ajouter est déjà dans l'arbre, l'hypothèse d'unicité des éléments pour certaines applications fait qu'on ne réalise pas l'ajout.

Remarque1

Vous constaterez que nous effectuons des comparaisons sur des valeurs numériques (entier, flottant et caractère). S'il s'agit d'autres types, vous devrez utiliser une fonction de comparaison. (comme strcmp() pour les chaînes de caractère).

Ajouter un nœud dans l'arbre

- Exemple**:** représenter l'arbre obtenu après l'insertion d'un élément de clé **13**.



Ajouter un nœud dans l'arbre

- Voici le code de la fonction AjouterNœud():

Fonction AjouterNœud()

```
Void AjouterNoeud(AbrNoeud *racine, Ttype val){
  if(racine == NULL){  //Arbre vide
     racine = CreerNoeud(val, NULL, NULL);
  }
  else{
      if(val <racine->valeur){
         AjouterNoeud(racine->filsg, val); //recherche dans sous arbre gauche
      }
      else{
         AjouterNoeud(racine->filsd, val); //recherche dans sous arbre droit
      }
  }
}
```


Rechercher un nœud dans l'arbre

- Parmi les avantages des arbres binaires de recherche est qu'ils **optimisent** les recherches dans un arbre. Pour savoir si un élément existe, il faut parcourir seulement une branche de l'arbre. Ce qui fait que le temps de recherche est directement proportionnel à la hauteur de l'arbre.

- L'algorithme se base directement sur les propriétés de l'arbre. En effet, si l'élément que l'on cherche est **plus petit** que la valeur du **nœud** alors on cherche dans le **sous arbre gauche**, sinon, on cherche dans le **sous arbre droit**.

Rechercher un nœud dans l'arbre

■ Pour rechercher un nœud dans un arbre, voici une méthode basée sur les propriétés de l'arbre:

• Si l'arbre est vide, la fonction retourne **0**,

• Si la racine contient la valeur recherchée, la fonction retourne **1**,

• Sinon, on compare la valeur recherchée avec la valeur de la racine et la recherche se fait (récursivement):

  • Si la valeur recherchée est **inférieure** à celle de la racine alors, on fait la recherche dans le **sous arbre gauche**,

  • Sinon, on fait la recherche dans le **sous arbre droit**.

Remarque

Nos recherches ici se contenteront seulement de déterminer si la valeur recherchée existe dans l'arbre. Avec une petite adaptation, on peut récupérer l'arbre dont le contenu de la racine est identique à la valeur recherchée.

Rechercher un nœud dans l'arbre

■ Voici le code de la fonction Existe():

Fonction Existe()

```
int Existe(AbrNoeud *racine, Ttype val){
 if(racine == NULL){ //Arbre vide
    return 0;
 }
 else if(racine->valeur == val){ //La racine contient la valeur recherchée
    return 1;
 }
 else if(racine->valeur > val)
    return Existe(racine->filsg,val); //recherche dans sous arbre gauche
 else
    return Existe(racine->filsd,val); //recherche dans sous arbre droit
}
```

Supprimer un nœud dans l'arbre

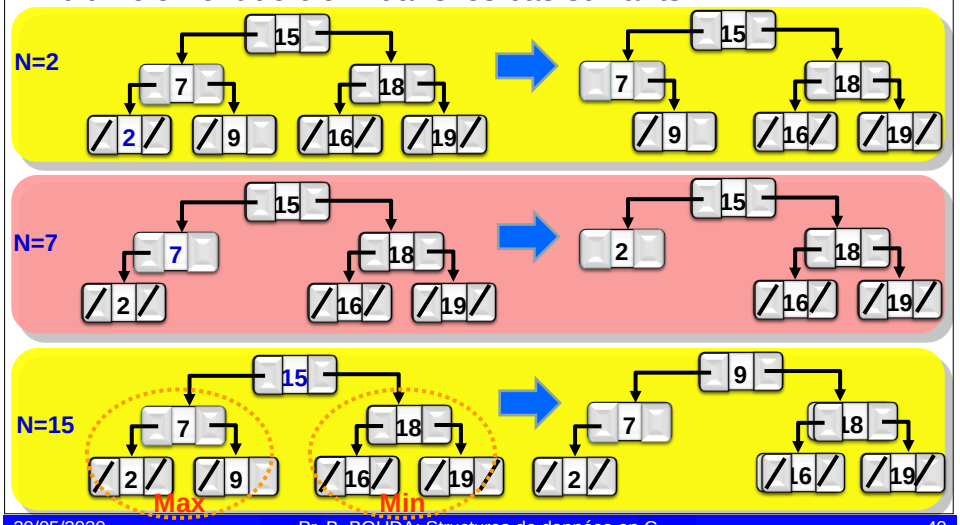
- La suppression est délicate car il faut réorganiser l'arbre pour qu'il vérifie la propriété d'un arbre binaire de recherche.
- Pour supprimer le nœud **N** dans l'arbre, voici une méthode basée sur les propriétés de l'arbre:
  - Si le nœud **N** est une feuille, il suffit de la supprimer,
  - Si **N** à un seul fils, on le remplace par son fils unique,
  - Si **N** à deux fils (cas général), on remplace la valeur de N :
    - Soit par la valeur **la plus grande** de **sous arbre gauche**,
    - Soit par la valeur **la plus petite** de sous arbre droit.

Remarque

La suppression d'un nœud dans un arbre binaire de recherche entraîne une réorganisation de l'arbre !

Supprimer un nœud dans l'arbre

- Exemples**:** représenter l'arbre obtenu après la suppression d'un élément de clé **N** dans les cas suivants:



En résumé

- Nous avons découvert que les arbres constituent un bon moyen de stocker les données en mémoire. Elles sont plus flexibles (mais complexes) que les listes chaînées vu leur aspect **non linéaire.**

- Comme pour les listes chaînées, il n'existe pas en langage C de système de gestion des arbres! C'est un autre moyen de progresser dans la programmation.