Les arbres binaires

Plan

- **Les arbres**
  - Définitions
  - Applications
  - Terminologies et propriétés
- **Les arbres binaires**
  - Définition
  - Implémentation
  - Parcours en Profondeur / Largeur
  - Les arbres binaires de recherche
  - Les arbres binaires parfaits
  - Le Tas

Introduction

Les structures de données linéaires :

- Les tableaux
- Les chaînes de caractères
- Les listes
- Les piles
- Les files
- …

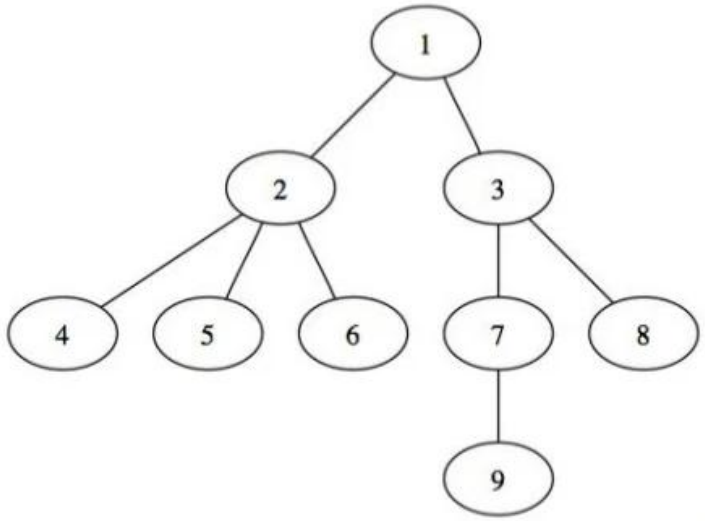
Les structures de données non linéaires :

- Les arbres
- Les graphes
- …

Introduction

Définition:

- Un arbre (tree en anglais) est une structure dynamique et non linéaire
- Un arbre est un ensemble de nœuds tel que :
  - ∃ un nœud unique appelé racine qui n'a pas de supérieur
  - Tous les autres nœuds sont atteints à partir de r par une branche unique
- Représentation graphique



Introduction

Applications:

Les arbres ont plusieurs applications fondamentales en Informatique, et ils sont bien adaptés à la représentation d'informations homogènes et organisées en hiérarchie.

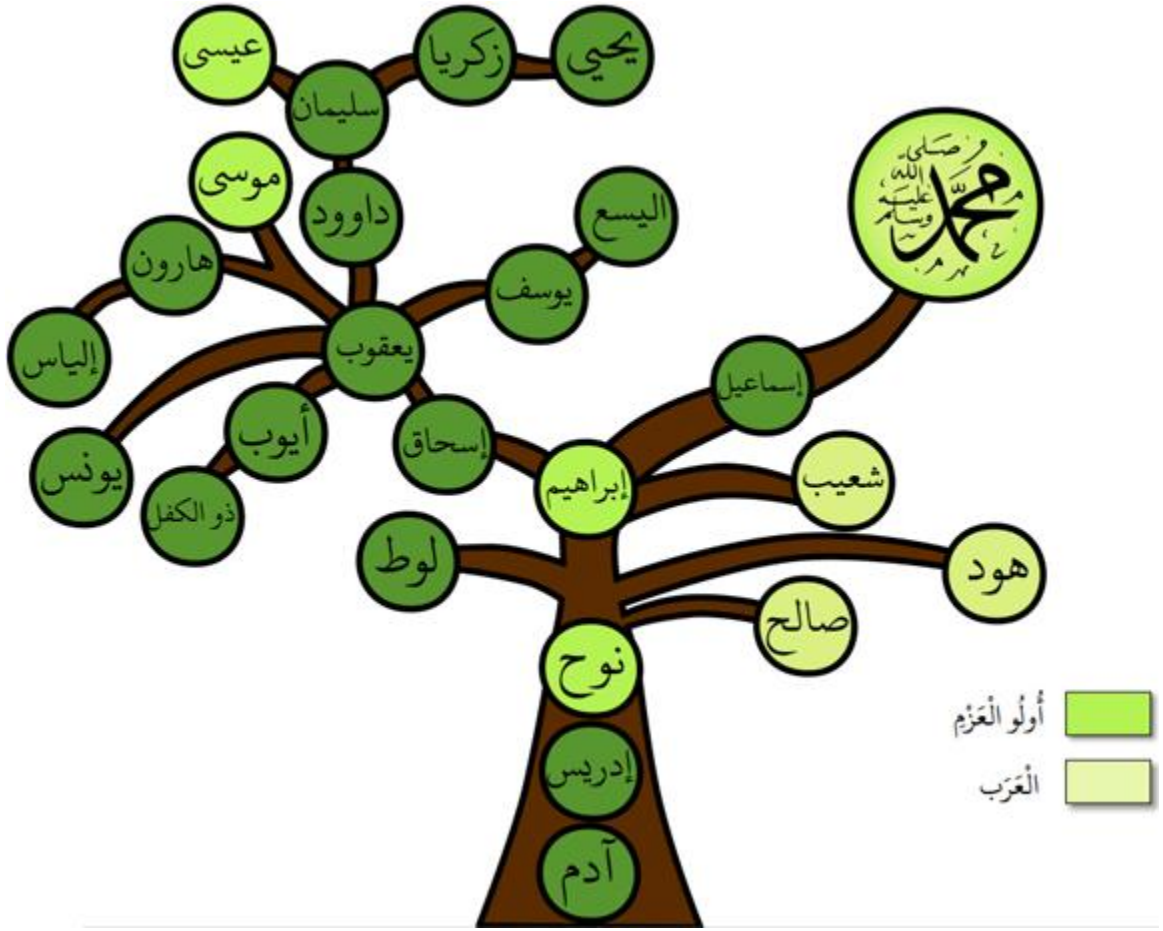
Exemples :

- Modèles pour les structures hiérarchisées
  - Arbre familiale
  - Arbre généalogique
  - Système de fichiers (Windows et Linux)
  - Organigramme d'une entreprise
- Les expressions arithmétiques
- Algorithme de recherche rapide et efficace
- Intelligence artificielle (Arbre de décision)
- …

Introduction

Applications:

Exemple 1 : Arbres généalogiques :



Introduction

Applications:

Exemple 2 : Expressions arithmétiques

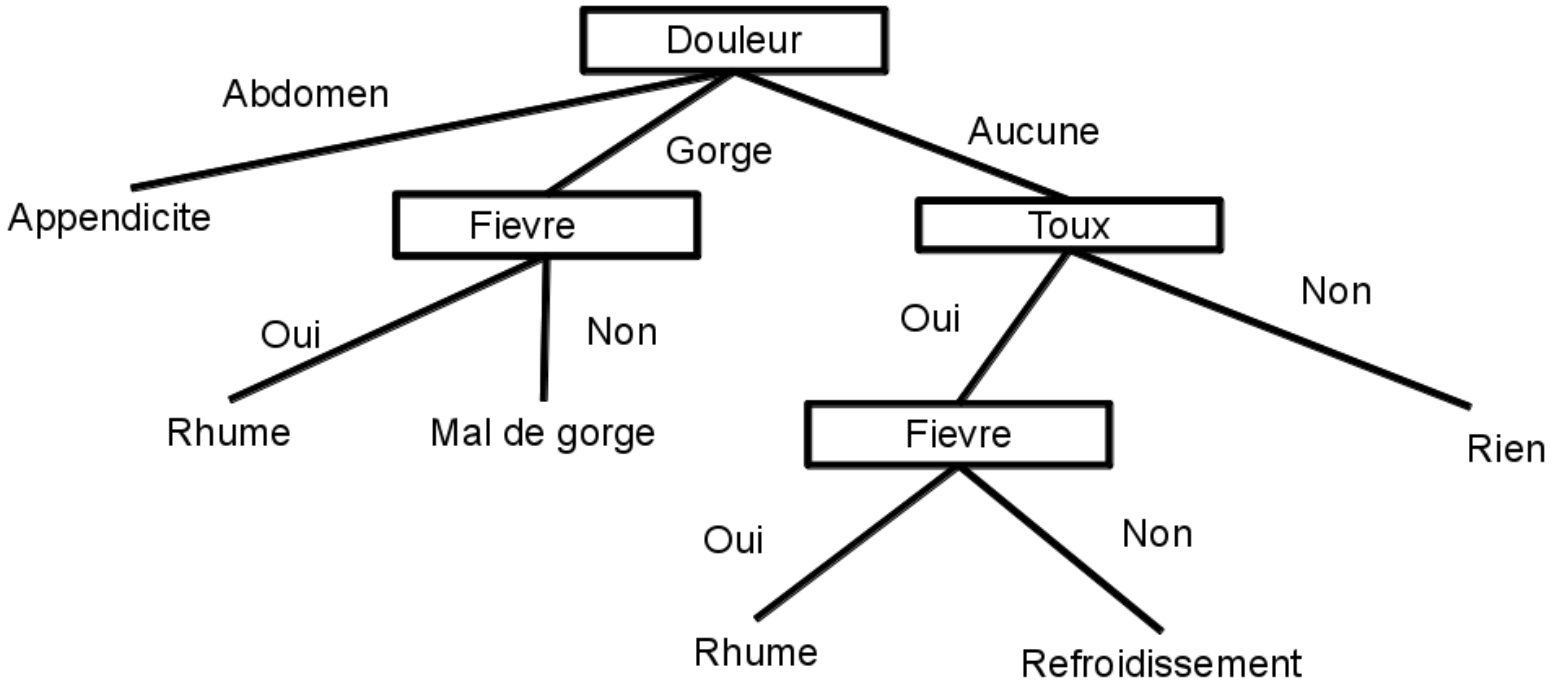
21

4*3 + 2*7-5

Introduction

Applications:

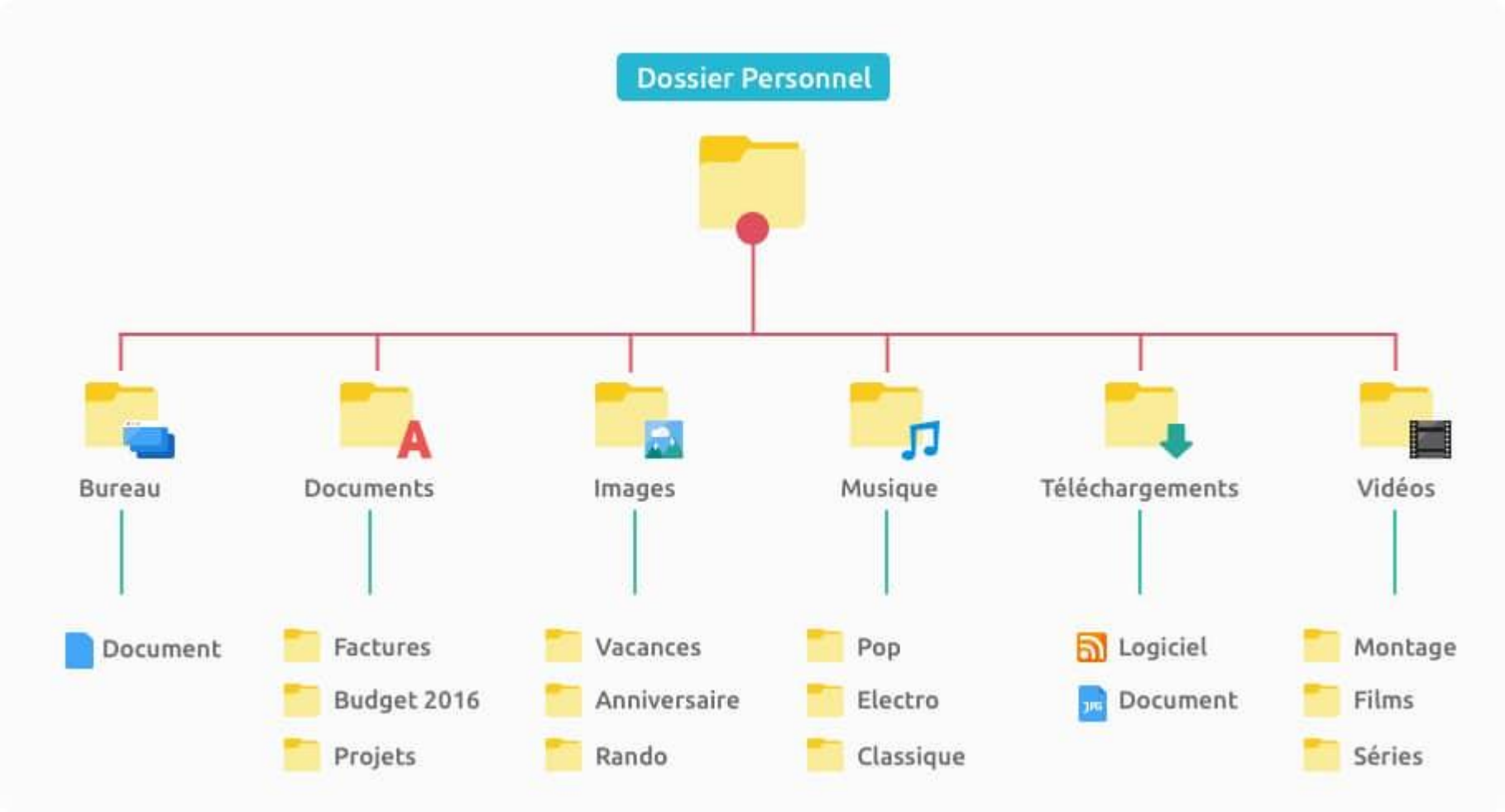
Exemple 3 : Système expert



Introduction

Applications:

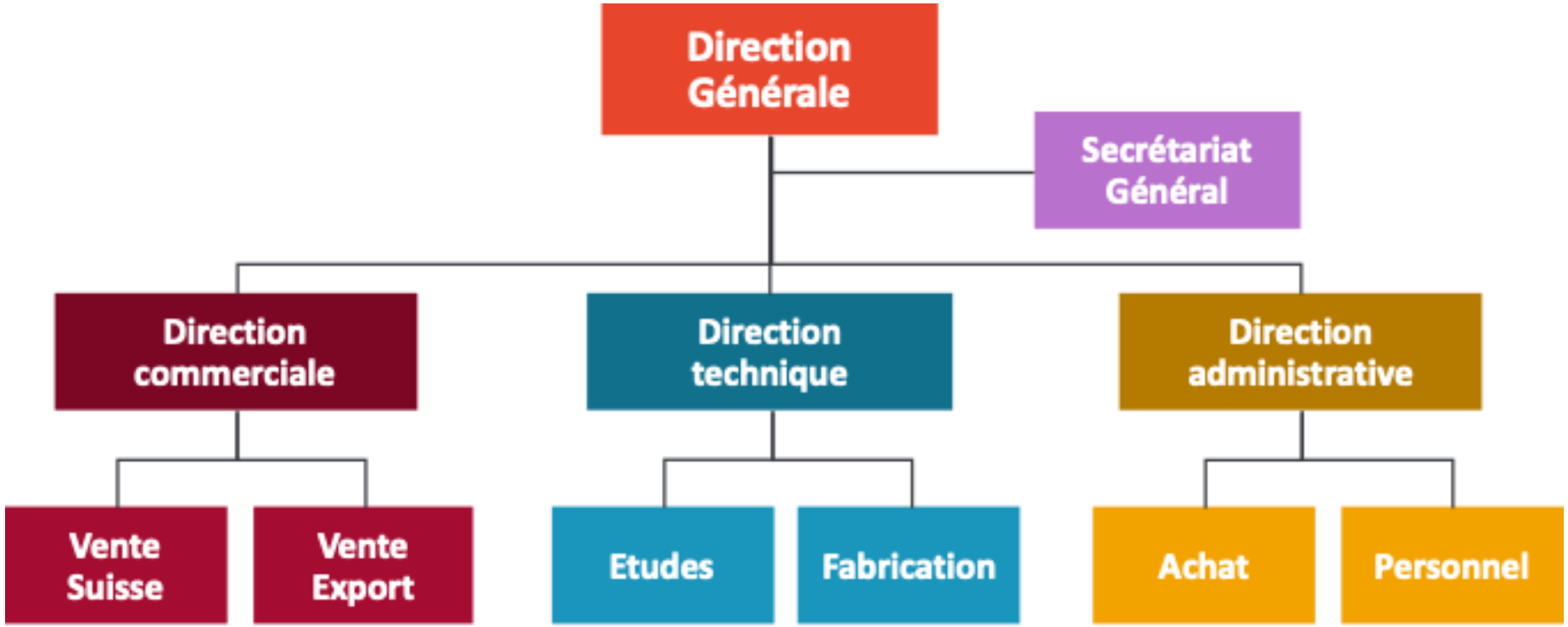
Exemple 4 : Système de fichiers



Introduction

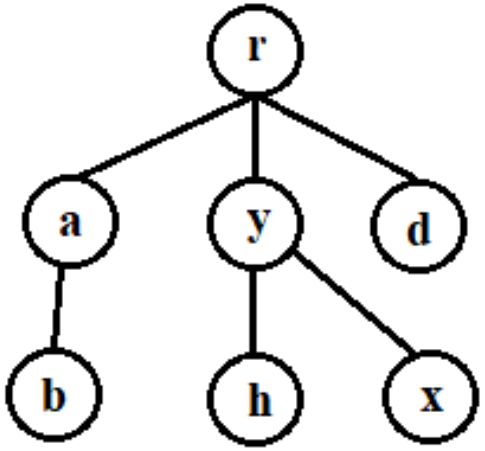
Applications:

Exemple 5 : Organisation d'une entreprise



Terminologie

- *Racine : Le seul nœud sans père (r est une racine)*
- *Feuille : Un nœud qui n'a pas de successeur (b, h, d et x sont des feuilles)*
- *Frère : Un nœud de même parent (a et y sont des frères)*
- *Sous-arbre : Une portion de l'arbre (Le nœud y avec ses fils)*
- *Branche : Une suite de nœuds connectés de la racine à une feuille (r-a-b; r-y-h; r-y-x ; r-d)*
- *Taille : Le nombre de nœuds qu'il possède (La taille est 7)*
- *Degré d'un nœud : Le nombre de ses fils (degre(r)=3 ; degre(a)=1; degre(h)=0)*
- *Degré d'un arbre : Le max des degré des nœuds (Le degré de l'arbre ci-dessous est 3)*

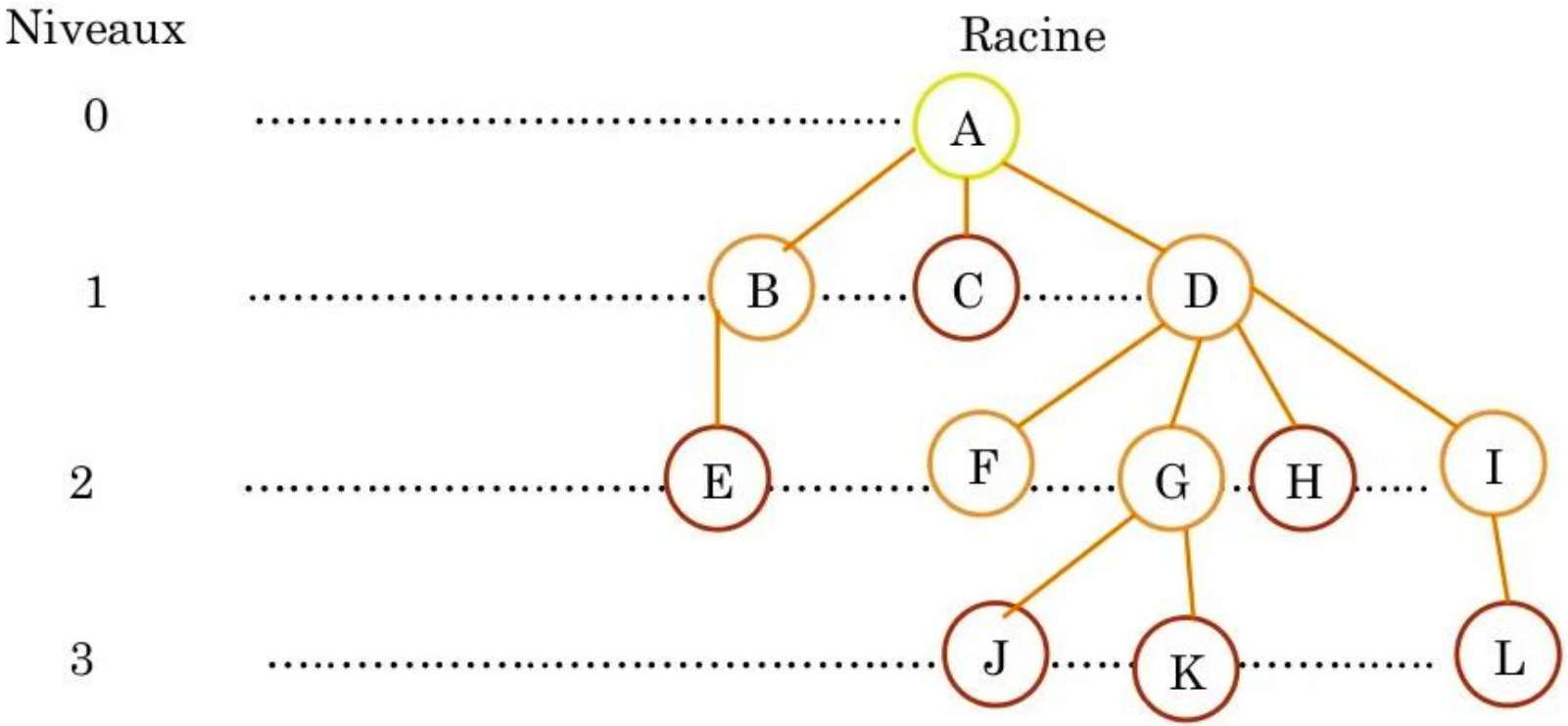


Terminologie

Le niveau d'un nœud (profondeur): La distance qui le sépare de la racine

Le niveau de la racine est : 0

Le niveau de chaque nœud est le niveau du père plus 1

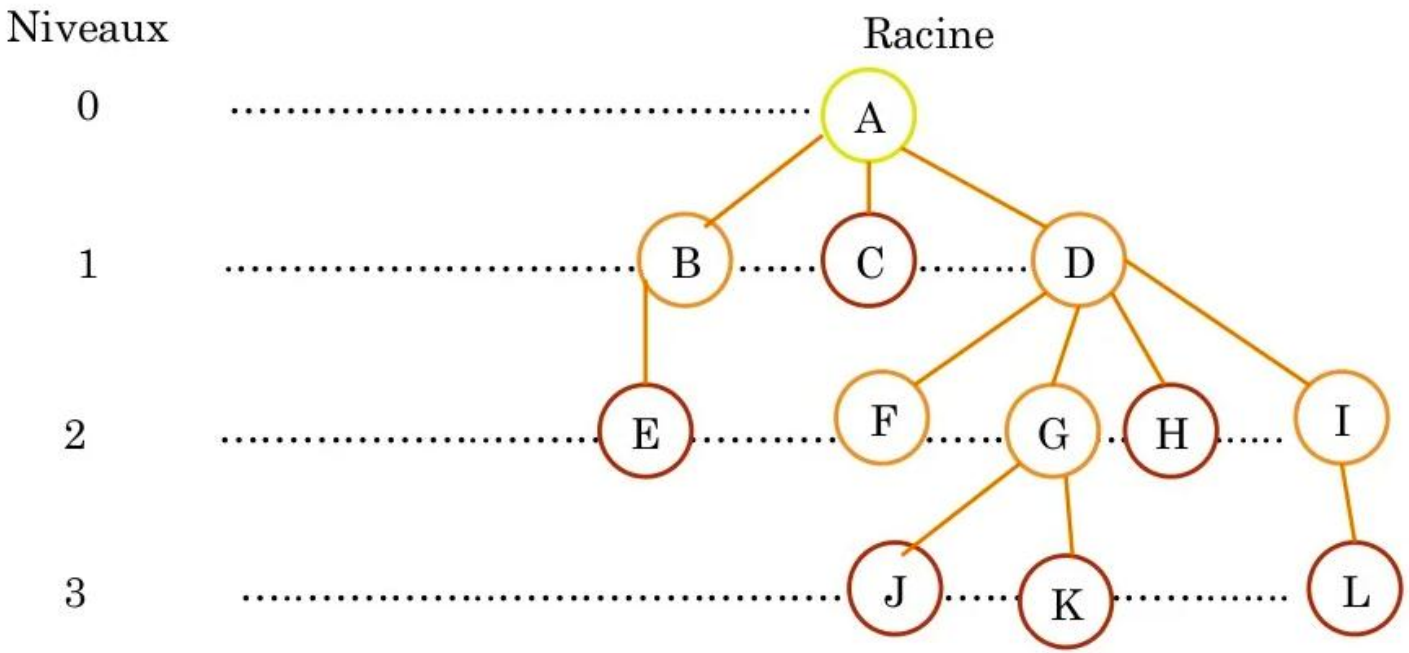


Terminologie

Hauteur d'un arbre (profondeur d'un arbre) : est le plus grand niveau des nœuds, c-à-d la distance entre la racine est la feuilles la plus lointaine

Par convention la hauteur d'un arbre vide est : -1

Hauteur = 3



Terminologie

Classification:

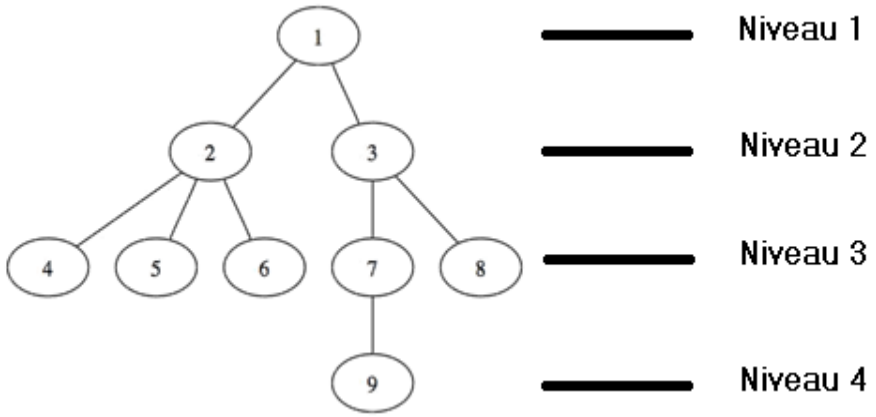
On peut classer les arbres selon deux critères de classification :

- Ordre de hiérarchie
- Arité

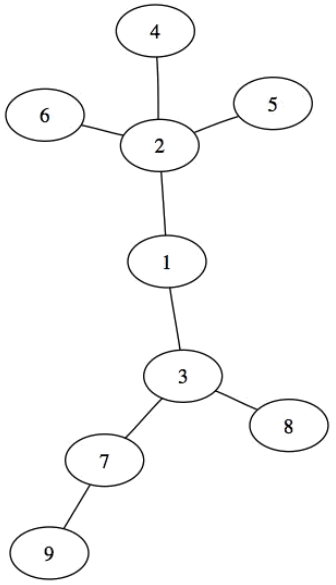
Terminologie

Classification: Classification selon l'ordre de hiérarchie:

- Enraciné :



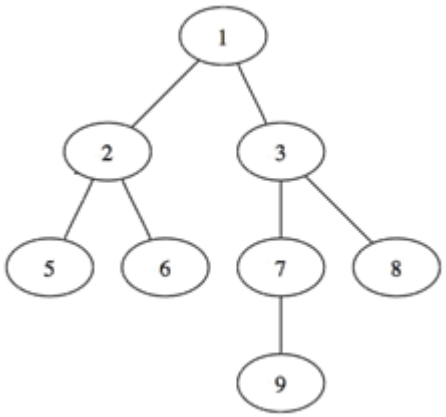
- Non enraciné :



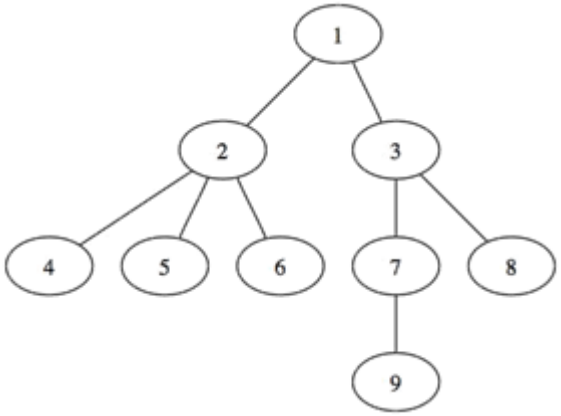
Terminologie

Classification: Classification selon l'arité des nœuds:

- Binaires :



- N-Aires:



Terminologie

Les arbres binaires :

- Les arbres binaires complets
- Les arbres binaires de recherche
- Les arbres binaires équivalents
- Les arbres binaires parfaits
- Les arbres AVL
- Les B-Arbres
- …

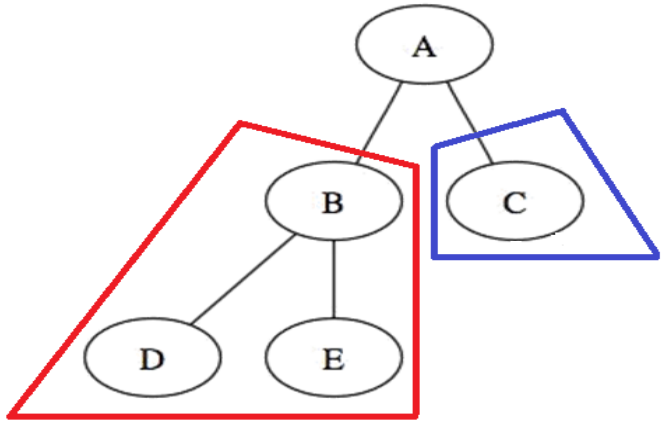
Implémentation

Les arbres binaires forment une structure de données qui peut être définie récursivement de la manière suivante :

Un arbre binaire est :

– soit vide,

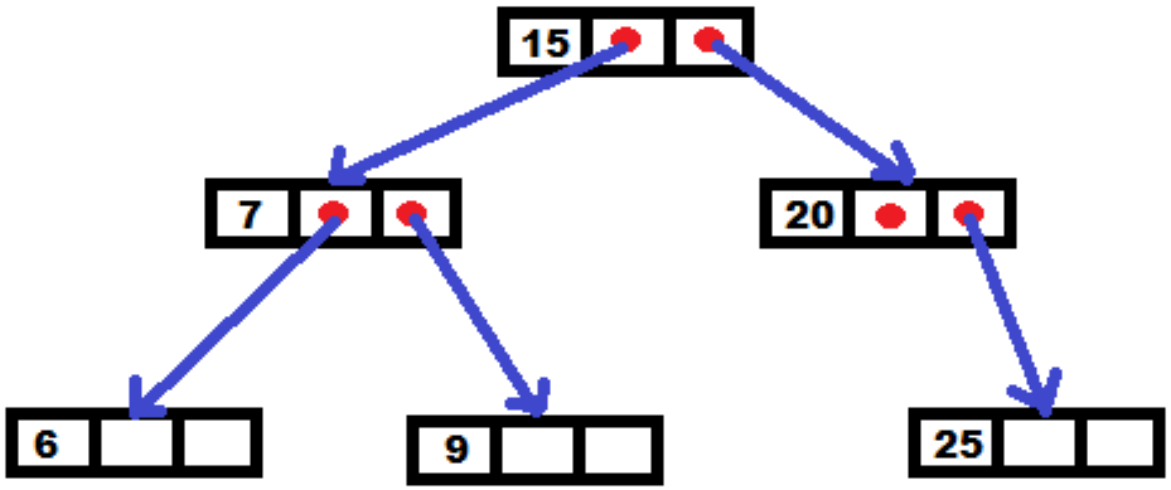
– soit composé d'une racine portant une valeur et d'une paire d'arbres binaires, appelés fils gauche et droit.



Implémentation

Un arbre est un ensemble de nœud relié entre eux, implémenter un arbre c'est tout d'abord modéliser un nœuds, en Langage C un nœud est une structure de trois éléments :

```
typedef struct Node {

    int valeur;
    struct Node *gauche;
    struct Node *droite;

} Node;
```



Pour déclarer un arbre vide :

```
Node *root = NULL;
```

Implémentation

Nous allons implémenter quelques primitives pour manipuler un arbre binaire :

1. Créer un nœud
2. Tester si l'arbre est vide
3. Obtenir l'arbre de gauche
4. Obtenir l'arbre de droite
5. Parcourir les éléments d'un arbre
6. La taille d'un arbre
7. La hauteur d'un arbre
8. Le nombre des feuilles
9. La valeur minimale
10. La valeur maximale
11. Tester si un arbre est une feuille
12. Insérer un nœud
13. Rechercher un nœud
14. Supprimer un nœud

Implémentation

1 - Créer un nœud

```
Node* creerNoeud(int val) {
    Node* newNode = (Node*) malloc(sizeof(Node));
    newNode->valeur = val;
    newNode->gauche = NULL;
    newNode->droite = NULL;
    return newNode;
}
```

Implémentation

2 - Tester si l'arbre est vide

```
int estVide(Node* root) {
    return root==NULL;
}
```

Implémentation

3 – Obtenir l'arbre de gauche

```
Node* gauche(Node* root) {
    return root->gauche;
}
```

Implémentation

4 – Obtenir l'arbre de droite

```
Node* droite(Node* root) {
    return root->droite;
}
```

5 -Algorithmes de parcours :

Nous allons découvrir des algorithmes de parcours d'un arbre. Cela permet de visiter tous les nœuds de l'arbre et éventuellement appliquer une fonction sur ces nœuds.

Nous distinguerons deux types de parcours :

- Le parcours en profondeur permet d'explorer l'arbre en explorant jusqu'au bout une branche pour passer à la suivante.

- Le parcours en largeur permet d'explorer l'arbre niveau par niveau. C'est à dire que l'on va parcourir tous les nœuds du niveau un puis ceux du niveau deux et ainsi de suite jusqu'à l'exploration de tous les nœuds.

5 -Algorithmes de parcours : Parcours en profondeur

Variantes du parcours en profondeur :

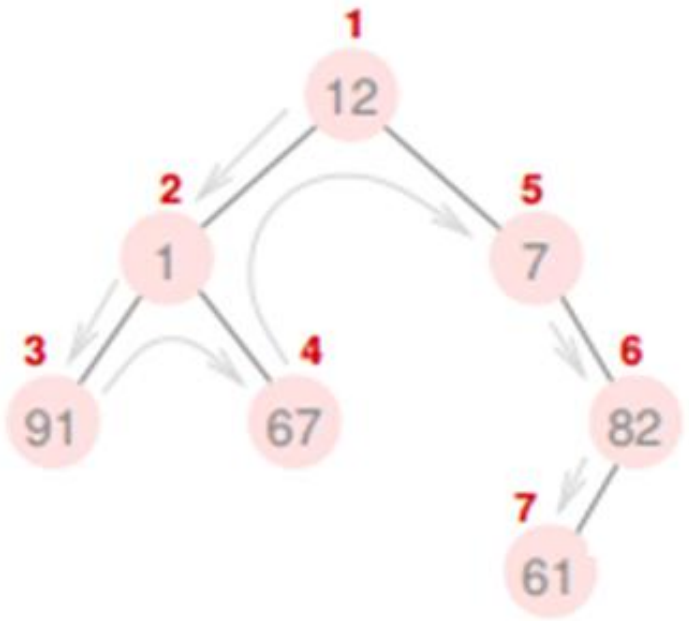
- Parcours préfixe
- Parcours infixe
- Parcours suffixe

5 -Algorithmes de parcours : Parcours en profondeur (Préfixe)

Parcours en profondeur préfixe :

Tout Nœud est suivi des nœuds de son sous-arbre Gauche puis des nœuds de son sous-arbre Droit

Le résultat d'affichage est :
12 ; 1 ; 91 ; 67 ; 7 ; 82 ; 61

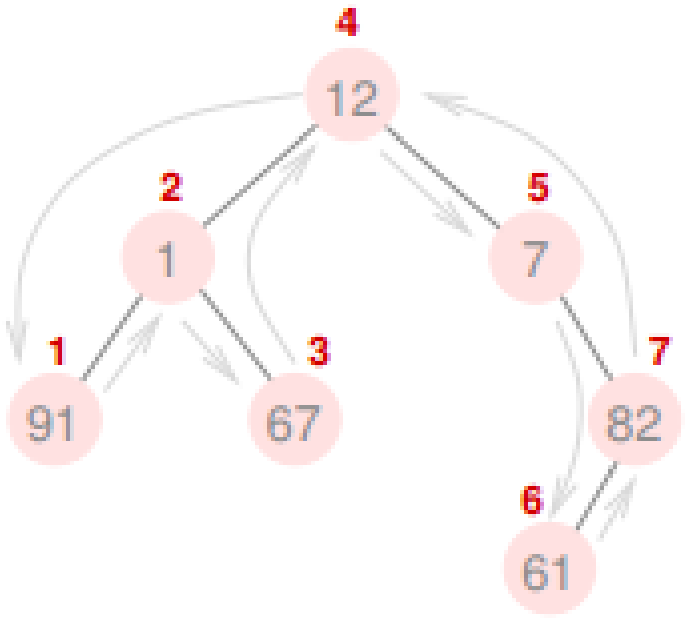


5 -Algorithmes de parcours : Parcours en profondeur (Infixe)

Parcours en profondeur infixe:

Tout Nœud est précédé des nœuds de son sous-arbre Gauche et suivi des noeuds de son sous-arbre Droit

Le résultat d'affichage est :
91 ; 1 ; 67 ; 12 ; 7 ; 61 ; 82

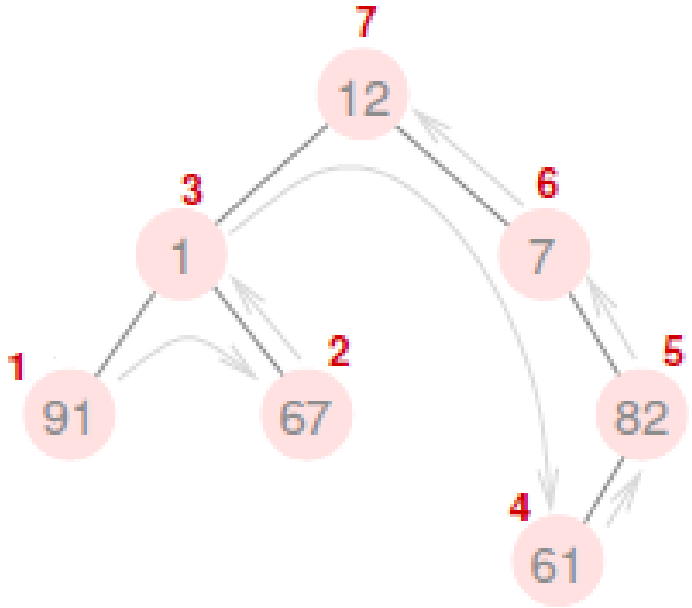


5 -Algorithmes de parcours : Parcours en profondeur (Postfixe)

Parcours en profondeur suffixe (postfixe):

Tout Noeud est précédé des nœuds de son sous-arbre Gauche et des nœuds de son sous-arbre Droit

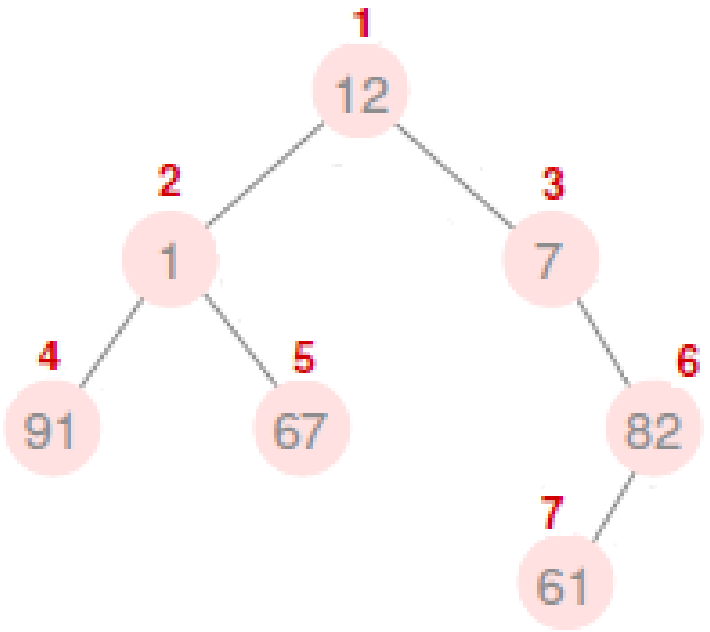
Le résultat d'affichage est :
91 ; 67 ; 1 ; 61 ; 82 ; 7 ; 12



5 -Algorithmes de parcours : Parcours en largeur

Dans ce type de parcours on affiche les valeurs par niveau (les nœud de même profondeur).

Le résultat d'affichage est :
12 ; 1 ; 7 ; 91 ; 67 ; 82 ; 61



Implémentation

Exercices :

Ecrire la fonction **parcoursProfondeurPrefixe(Node* racine)** qui affiche les valeurs des nœuds par l'algorithme de parcours préfix.

```
void parcoursPrefixe(Node* racine) {
    if (racine != NULL) {
        printf("%d ", racine->valeur);
        parcoursPrefixe(racine->gauche);
        parcoursPrefixe(racine->droite);
    }
}
```

Implémentation

Exercices :

Ecrire la fonction **parcoursProfondeurInfixe(Node* racine)** qui affiche les valeurs des nœuds par l'algorithme de parcours infixe.

```
void parcoursInfixe(Node* racine) {
    if (racine != NULL) {
        parcoursInfixe(racine->gauche);
        printf("%d ", racine->valeur);
        parcoursInfixe(racine->droite);
    }
}
```

Implémentation

Exercices :

Ecrire la fonction **parcoursProfondeurPostfixe(Node* racine)** qui affiche les valeurs des nœuds par l'algorithme de parcours postfixe.

```
void parcoursPostfixe(Node* racine) {
    if (racine != NULL) {
        parcoursPostfixe(racine->gauche);
        parcoursPostfixe(racine->droite);
        printf("%d ", racine->valeur);
    }
}
```

Implémentation

Exercices :

Algorithme du parcours en largeur :

Lorsque nous sommes sur un nœud nous traitons ce nœud (par exemple nous l'affichons) puis nous mettons les fils gauche et droit non vides de ce nœud dans la file d'attente, puis nous traitons le prochain nœud de la file d'attente.

Au début, la file d'attente ne contient rien, nous y plaçons donc la racine de l'arbre que nous voulons traiter.

L'algorithme s'arrête lorsque la file d'attente est vide. En effet, lorsque la file d'attente est vide, cela veut dire qu'aucun des nœuds parcourus précédemment n'avait de sous arbre gauche ni de sous arbre droit.

Par conséquent, on a donc bien parcouru tous les nœuds de l'arbre.

Implémentation

```c
#define MAX 100
typedef struct {
    Node* elements[MAX];
    int tete;
    int queue;
    int taille;
} File;

File* creerFile();
estVide(File *f);
enfiler(File *f, Node* r);
defiler(File *, Node* r);
```

```c
void parcoursLargeur(Node* racine) {
    if (racine == NULL) {
        return;
    }

    File* file = creerFile();
    enfiler(file, racine);

    while (!estVide(file)) {
        Node* courant = defiler(file);
        printf("%d ", courant->valeur);

        if (courant->gauche != NULL) {
            enfiler(file, courant->gauche);
        }
        if (courant->droite != NULL) {
            enfiler(file, courant->droite);
        }
    }
}
```

Implémentation

Exercices :

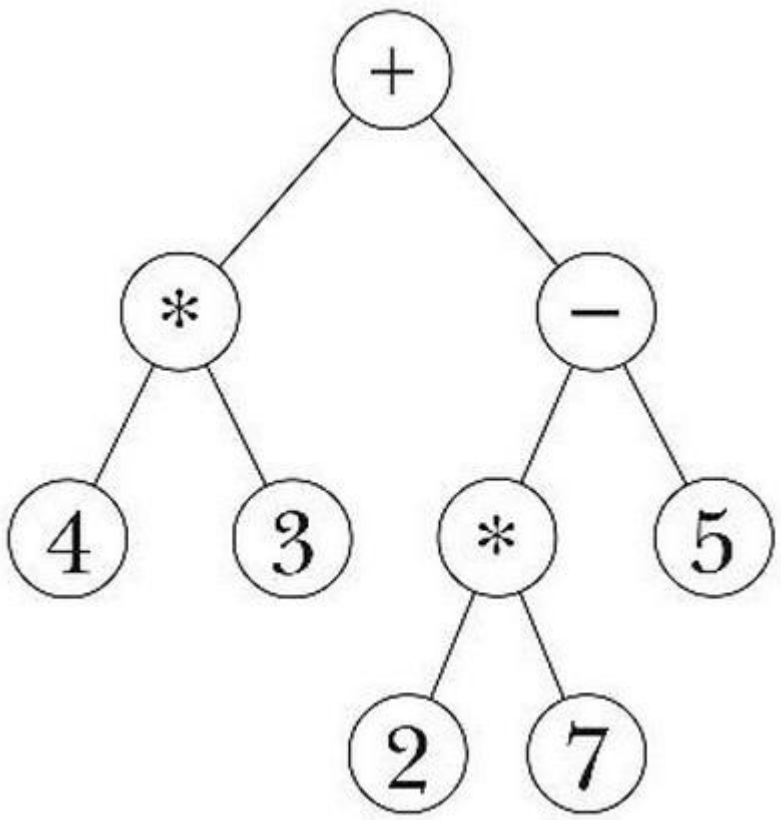
Quel type de parcours doit être utilisé pour reconstruire une expression arithmétique ?

Expression arithmétique:

Ordre préfixe : +x43-x275

Ordre infixe : 4x3+2x7-5

Ordre postfixe : 43x27x5-+



Implémentation

6 – Nombre de nœuds :

```
int tailleArbre(Node* racine) {
    if (racine == NULL) {
        return 0;
    }
    return 1 + tailleArbre(racine->gauche)
             + tailleArbre(racine->droite);
}
```

Implémentation

7 – Hauteur d'un arbre :

```
int hauteurArbre(Node* racine) {
    if (racine == NULL) {
        return -1;
    }
    int hauteurGauche = hauteurArbre(racine->gauche);
    int hauteurDroite = hauteurArbre(racine->droite);
    return 1 + max(hauteurGauche, hauteurDroit);
}
```

Implémentation

8 – Nombre de feuilles:

```
int nombreFeuilles(Node* racine) {
    if (racine == NULL) {
        return 0;
    }
    if (racine->gauche == NULL && racine->droite == NULL) {
        return 1;
    }
    return nombreFeuilles(racine->gauche)
           + nombreFeuilles(racine->droite);
}
```

Implémentation

9 – Max d'un arbre:

```
int maxArbre(Node* racine) {
    if (racine->gauche == NULL && racine->droite == NULL) {
        return racine->valeur;
    }

    if(racine->gauche == NULL) {
        return max(racine->valeur, maxArbre(racine->droite));
    }

    if(racine->droite == NULL) {
        return max(racine->valeur, maxArbre(racine->gauche));
    }

    return max(racine->valeur, maxArbre(racine->gauche), maxArbre(racine->droite));
}
```

Arbre Binaire de Recherche
(ABR)

Arbre binaire de recherche

Les arbres binaires :

- Les arbres binaires complets
- Les arbres binaires de recherche
- Les arbres binaires équivalents
- Les arbres binaires parfaits
- Les arbres AVL
- Les B-Arbres
- …

Arbre binaire de recherche

Définitions :

Un arbre binaire de recherche (ABR) est un arbre binaire qui possède la propriété fondamentale suivante :

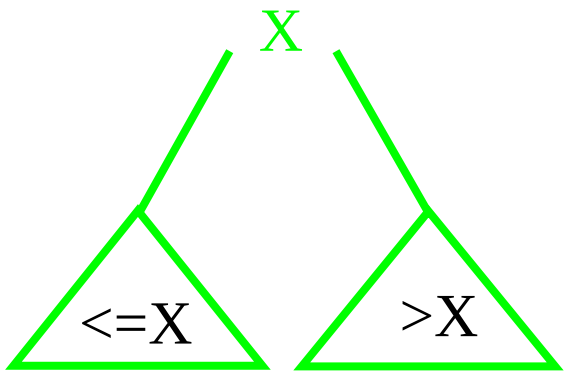
- Tous les nœuds du sous-arbre gauche d'un nœud de l'arbre ont une valeur inférieure ou égale à ce nœud.
- Tous les nœuds du sous-arbre droit d'un nœud de l'arbre ont une valeur supérieure à ce nœud.

Arbre binaire de recherche

Récursivement un *arbre binaire de recherche* est un arbre binaire tel que :

1 - Pour tout nœud x , les nœuds de son sous arbre-gauche s'ils en existent ont des valeurs inférieures ou égales à celle de x, et les nœuds de son sous arbre-droit des valeurs strictement supérieures.

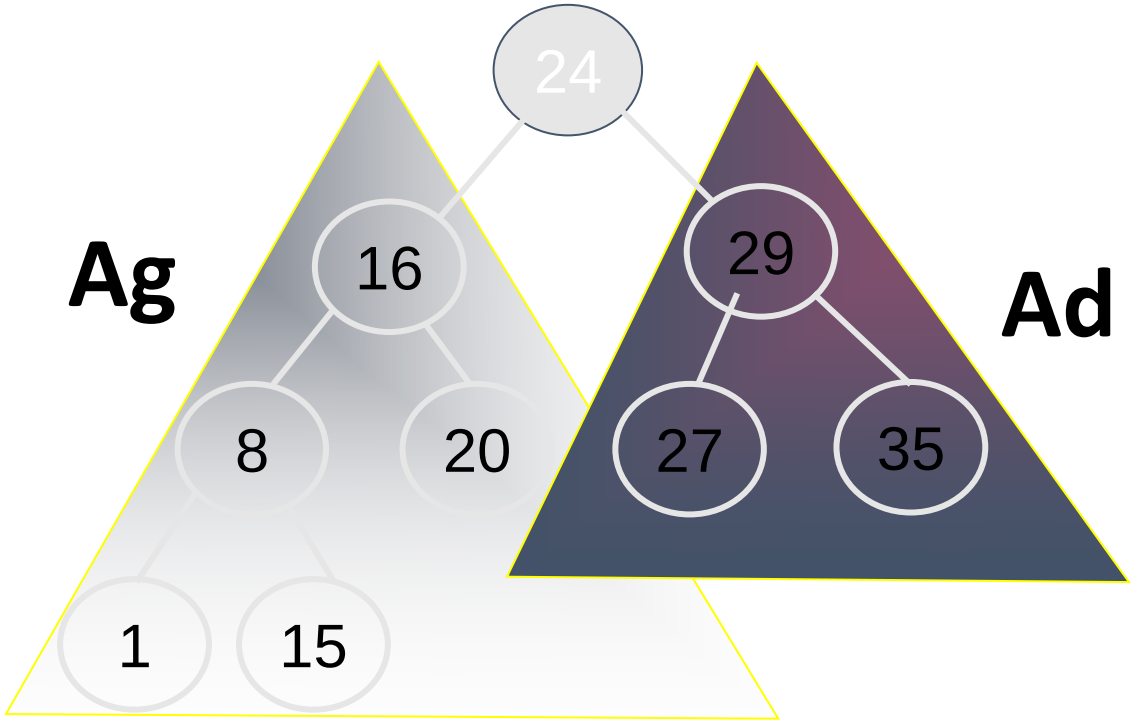
Ce que l'on traduit par $g(A) \leq racine(A) < d(A)$.



Arbre binaire de recherche

Récursivement un *arbre binaire de recherche* est un arbre binaire tel que :

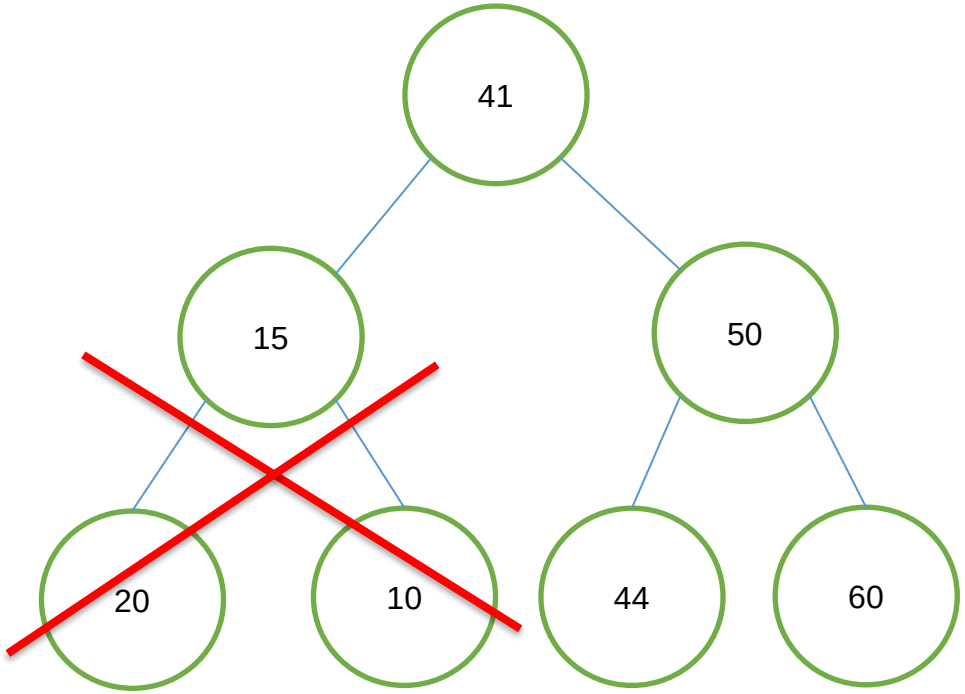
2 - Tout sous-arbre d'un ABR est un ABR : .



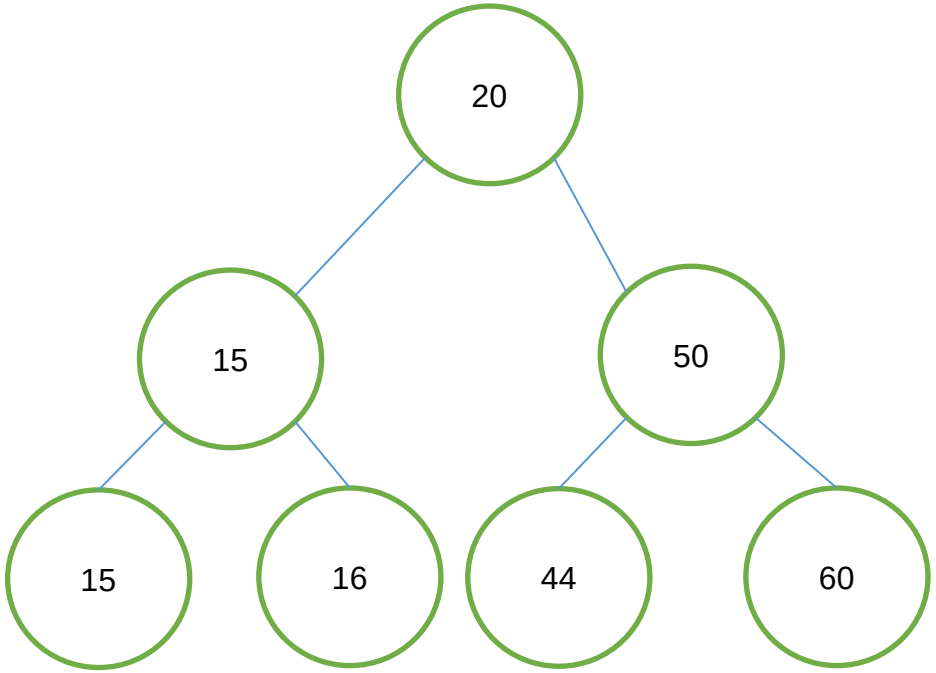
Exemple d'arbre binaire de recherche

Arbre binaire de recherche

Exemples :



Ce n’est pas un arbre binaire de recherche



C’est un arbre binaire de recherche

Arbre binaire de recherche

Applications :

Un arbre binaire de recherche ou ABR est une structure de données utilisée pour représenter un ensemble de valeurs ou objets qui subit à une relation d'ordre.

Exemples :

- Un ensemble d'employé trié par leur hiérarchie dans l'organisation
- Un ensemble de personnes trié par leurs âges

Arbre binaire de recherche

Question : Pourquoi ne pas utiliser un tableau ou liste triée au lieu d'un ABR ?

Comparaison :

Opération	Liste	ABR
Recherche	O(log(n))	O(log(h))
Insertion	O(n)	O(h)
Suppression	O(n)	O(h)
Min	O(n)	O(h)
Max	O(n)	O(h)

NB : Ces complexités suppose que les arbres sont équilibrés

Arbre binaire de recherche

Applications :

Ecrire une fonction qui vérifie si un arbre binaire est un arbre de binaire de recherche:

Vous pouvez utiliser les fonctions maxArbre(A) et minArbre(A) qui retourne respectivement le max et le min d'un arbre binaire.

```
int estAbr(Node* racine) {
    if (racine == NULL) {
        return 1;
    }

    if (racine->gauche != NULL && racine->gauche->valeur > racine->valeur) {
        return 0;
    }

    if (racine->droite != NULL && racine->droite->valeur < racine->valeur) {
        return 0;
    }

    return estAbr(racine->gauche) && estAbr(racine->droite);
}
```

Arbre binaire de recherche

Ecrire une fonction qui retourne la valeur maximale d'un arbre binaire de recherche:

```
int maxAbrR(Node* racine) {
    if(racine->droite == NULL) {
        return racine->valeur;
    }
    return maxAbrR(racine->droite);
}
```

Version récursive

```
int maxAbr(Node* racine) {
    Node* courant = racine;
    while (courant->droite != NULL) {
        courant = courant->droite;
    }
    return courant->valeur;
}
```

Version itérative

Arbre binaire de recherche

Ecrire une fonction qui retourne la valeur minimale d'un arbre binaire de recherche:

```
int minAbrR(Node* racine) {
    if(racine->gauche == NULL) {
        return racine->valeur;
    }
    return minAbrR(racine-> gauche);
}
```

Version récursive

```
int minAbr(Node* racine) {
    Node* courant = racine;
    while (courant->gauche != NULL) {
        courant = courant->gauche;
    }
    return courant->valeur;
}
```

Version itérative

Recherche par dichotomie:

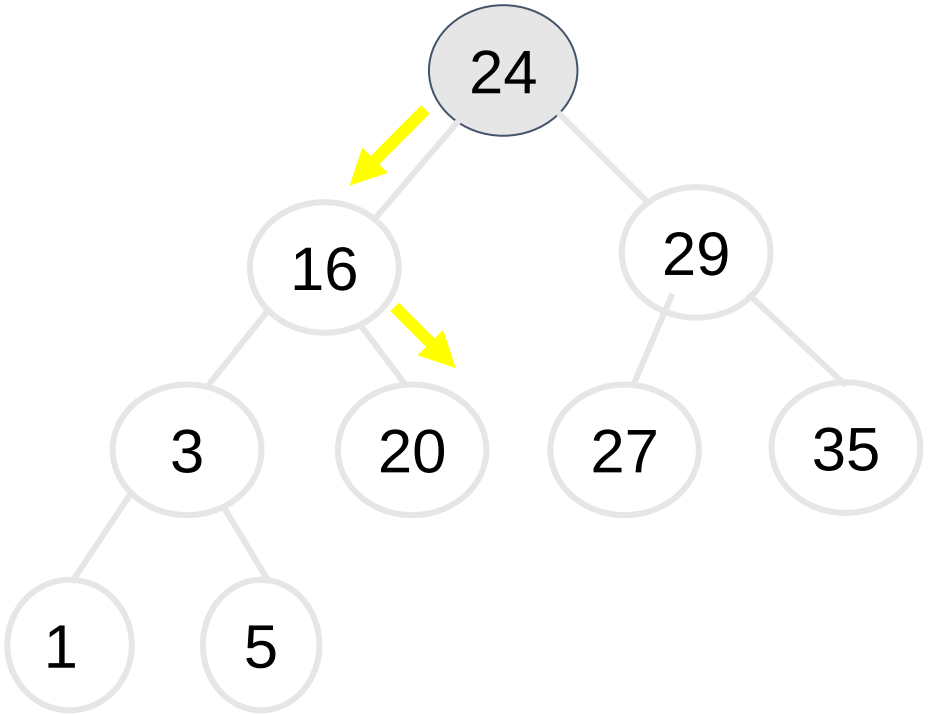
L'algorithme :

- si le sous-arbre sélectionné est vide, l'élément est absent la fonction retourne False
- On compare l'élément à la valeur de la racine , si égalité la fonction retourne True
- si la valeur est plus petite, on recommence **récursivement** dans le sous-arbre gauche ; et réciproquement si la valeur est plus grande dans le sous-arbre droit.

Arbre binaire de recherche

Recherche par dichotomie:

Exemple : Soit à rechercher 20 dans l'arbre suivant



20 est plus petit que 24

20 est plus grand que 16

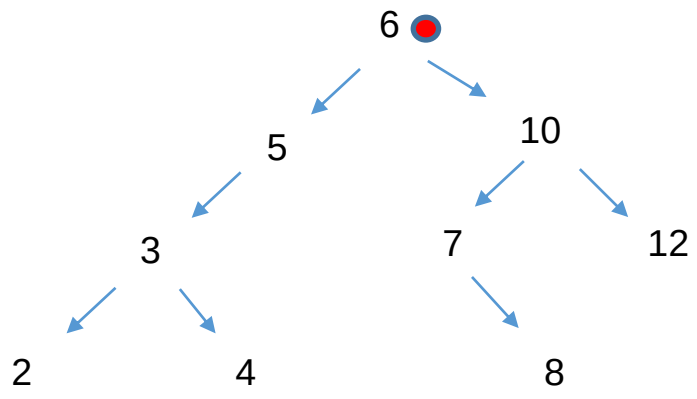
20 est trouvé

Recherche par dichotomie:

```
int rechercheDichotomique(Node* racine, int valeur) {
    if (racine == NULL) {
        return 0;
    }
    if (valeur == racine->valeur) {
        return 1;
    }
    if (valeur < racine->valeur) {
        return rechercheDichotomique(racine->gauche, valeur);
    }
    return rechercheDichotomique(racine->droite, valeur);
}
```

Insertion dans un ABR :

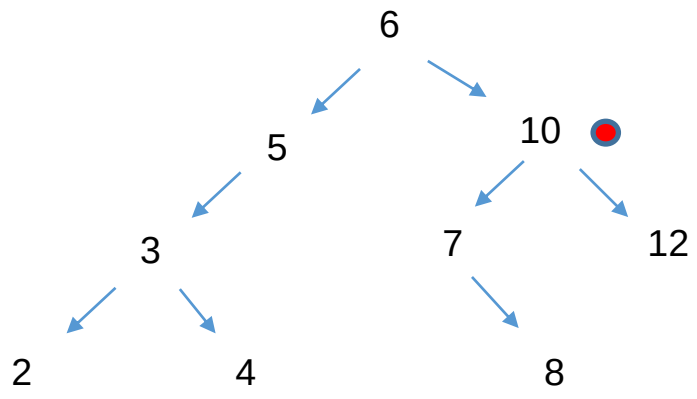
On cherche la clé du nœud à insérer ;
Lorsqu'on arrive à une feuille, on ajoute le nœud comme fils de la feuille en comparant sa clé à celle de la feuille :
- si elle est inférieure, le nouveau nœud sera à gauche ;
- sinon il sera à droite.



Insérer 9

Insertion dans un ABR :

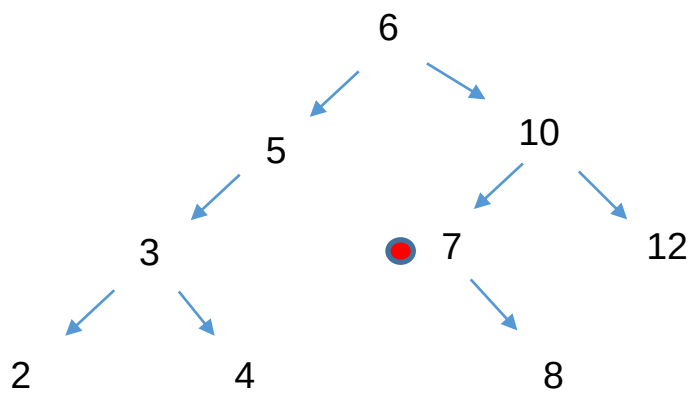
On cherche la clé du nœud à insérer ;
Lorsqu'on arrive à une feuille, on ajoute le nœud comme fils de la feuille en comparant sa clé à celle de la feuille :
- si elle est inférieure, le nouveau nœud sera à gauche ;
- sinon il sera à droite.



Insérer 9

Insertion dans un ABR :

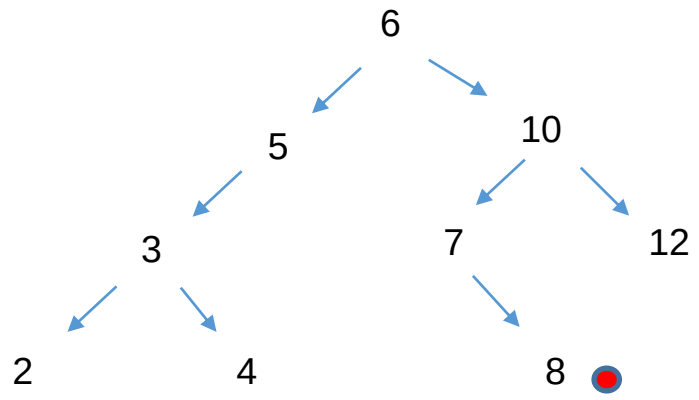
On cherche la clé du nœud à insérer ;
Lorsqu'on arrive à une feuille, on ajoute le nœud comme fils de la feuille en comparant sa clé à celle de la feuille :
- si elle est inférieure, le nouveau nœud sera à gauche ;
- sinon il sera à droite.



Insérer 9

Insertion dans un ABR :

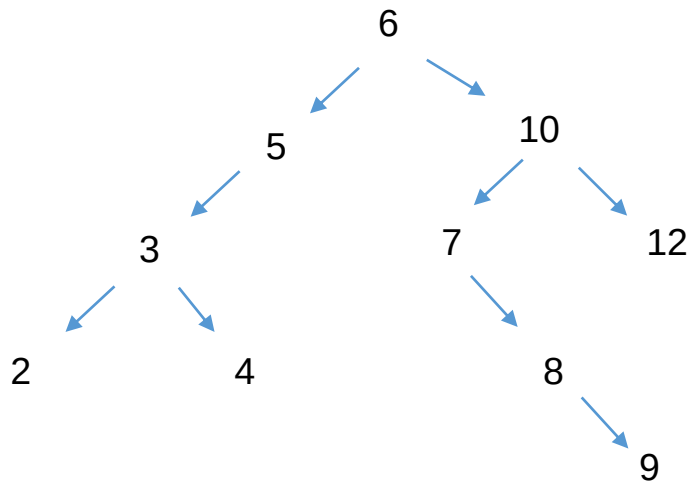
On cherche la clé du nœud à insérer ;
Lorsqu'on arrive à une feuille, on ajoute le nœud comme fils de la feuille en comparant sa clé à celle de la feuille :
- si elle est inférieure, le nouveau nœud sera à gauche ;
- sinon il sera à droite.



Insérer 9

Insertion dans un ABR :

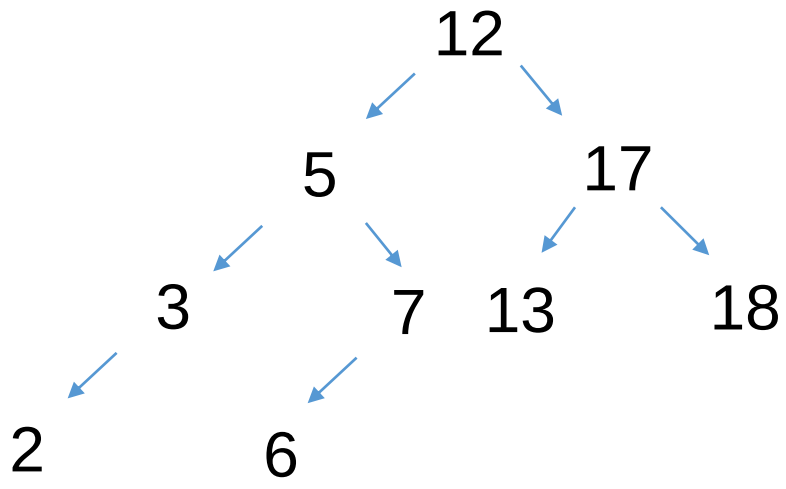
On cherche la clé du nœud à insérer ;
Lorsqu'on arrive à une feuille, on ajoute le nœud comme fils de la feuille en comparant sa clé à celle de la feuille :
- si elle est inférieure, le nouveau nœud sera à gauche ;
- sinon il sera à droite.



Insérer 9

Insertion dans un ABR :

Exercice :
Quel arbre obtient-on en insérant dans l'ordre , 12, 5, 3, 17, 7, 13, 18, 2 et 6 dans un arbre vide ?



Insertion dans un ABR :

Exercice :
Ecrire une fonction récursive **inserer(abr , v)** qui permet d'insérer correctement la valeur v dans l'arbre binaire de recherche abr. La fonction retourne un arbre avec la nouvelle valeur à insérer.

```
Node* insererAbr(Node* racine, int valeur) {
    if (racine == NULL) {
        return creerNoeud(valeur);
    }

    if (valeur < racine->valeur) {
        racine->gauche = insererAbr(racine->gauche, valeur);
    } else if (valeur > racine->valeur) {
        racine->droite = insererAbr(racine->droite, valeur);
    }

    return racine;
}
```

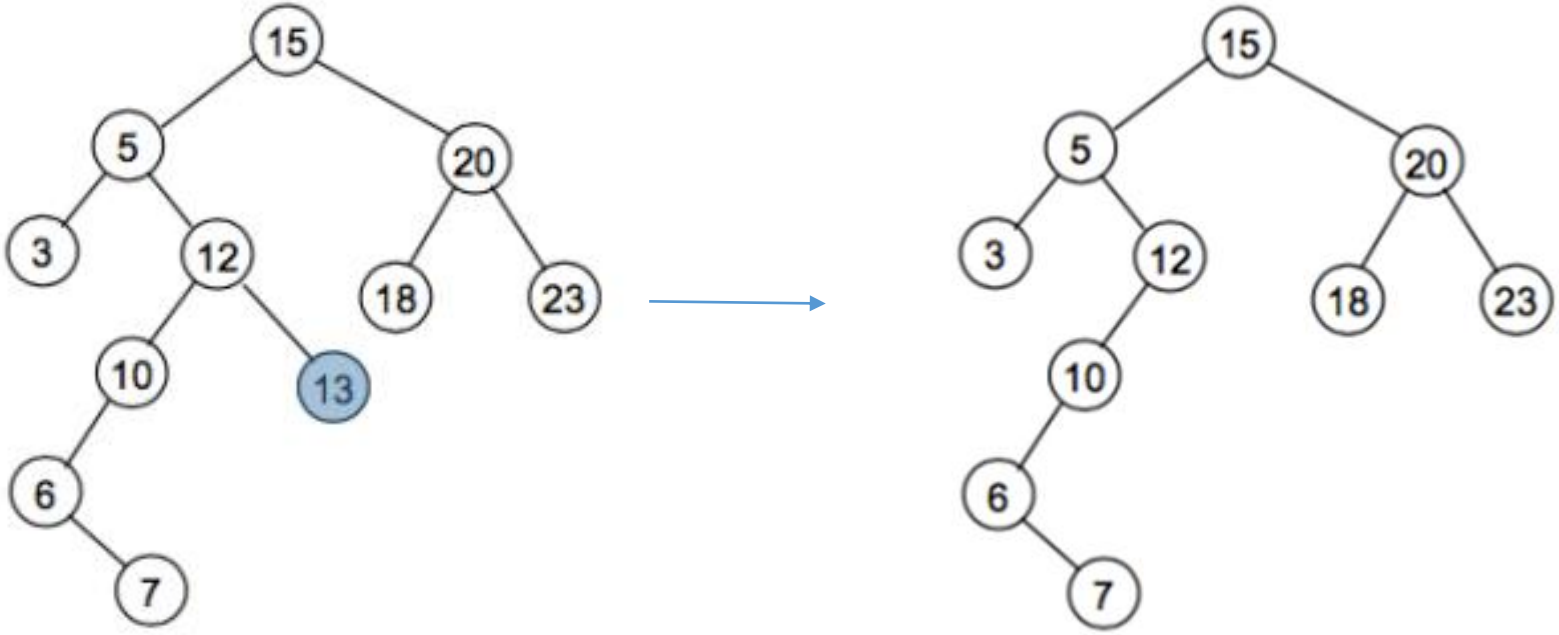
Supprimer un élément d'un ABR :

Pour supprimer un nœud d'un arbre binaire de recherche, on distingue trois cas à traiter :

- L'élément à supprimer n'a pas de fils : il est terminal et il suffit de le supprimer
- L'élément a un fils unique : on supprime le nœud et on relie son fils à son père
- L'élément à supprimer a deux fils : on le remplace par son successeur qui est toujours le minimum de ses descendants droits OU pas son prédécesseur qui est toujours le maximum de ses descendant gauche.

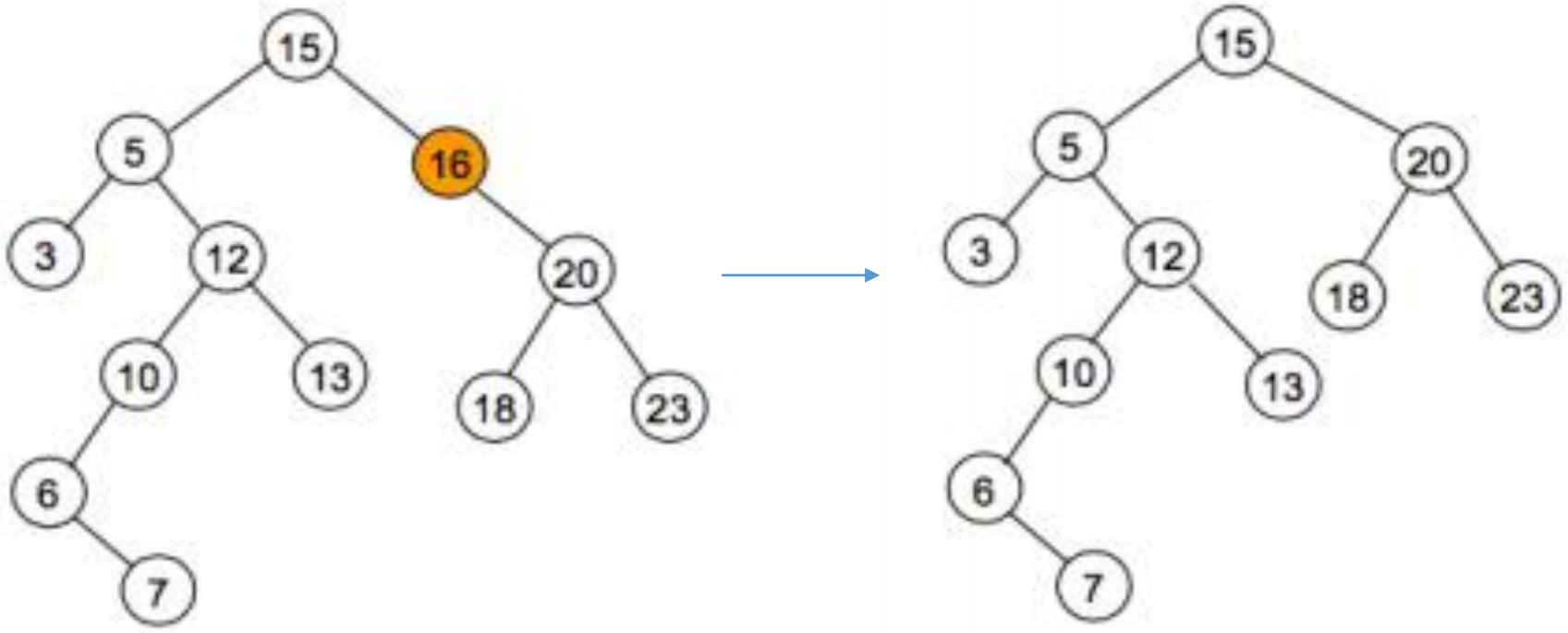
Supprimer un élément d'un ABR :

- Cas N° 1 : Supprimer une feuille



Supprimer un élément d'un ABR :

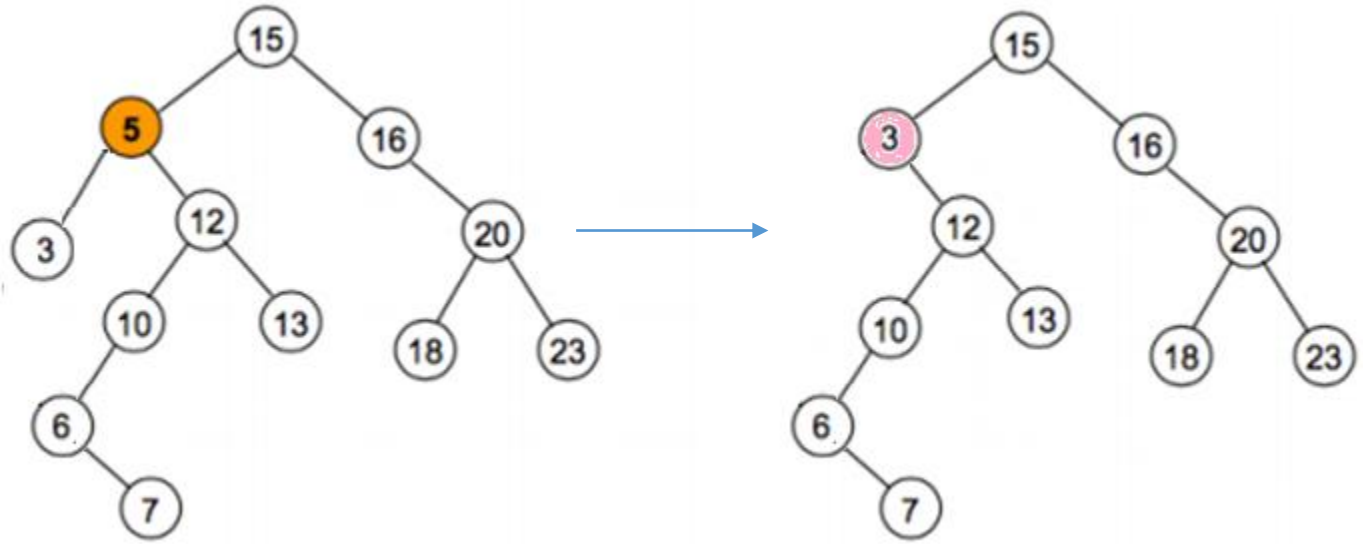
- Cas N° 2 : Supprimer un nœud qui a un seul fils



Supprimer un élément d'un ABR :

- Cas N° 3 : Supprimer un nœud qui a un deux fils

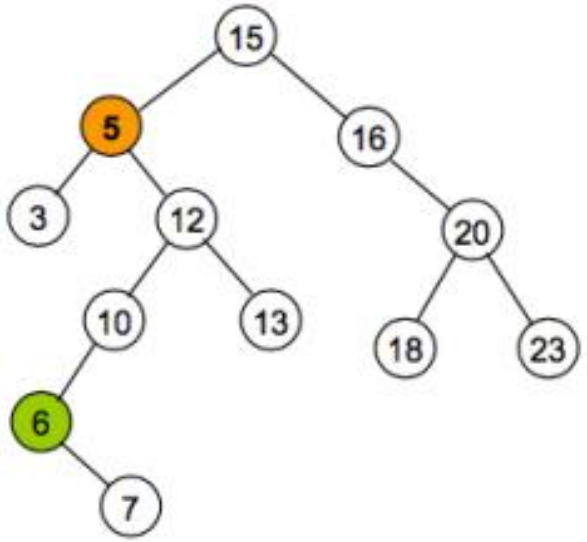
Soit : remplacer le nœud à supprimer par la feuille du sous-arbre gauche contenant la valeur maximal

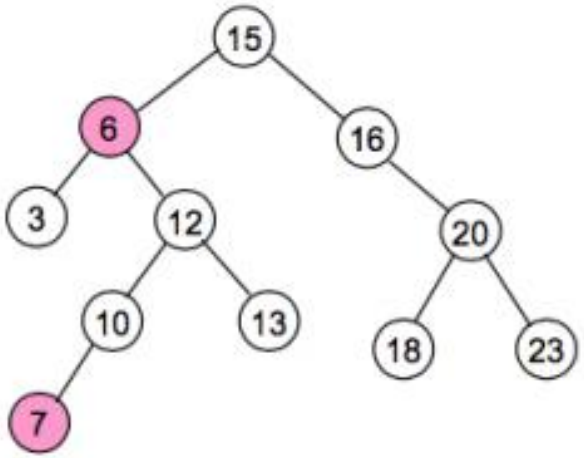


Supprimer un élément d'un ABR :

- Cas N° 3 : Supprimer un nœud qui a un deux fils

Ou : ou par la feuille du sous-arbre droit contenant la valeur minimale





Equilibre :

Un arbre binaire de hauteur h peut contenir jusqu'à $2^{h+1} - 1$ nœuds :

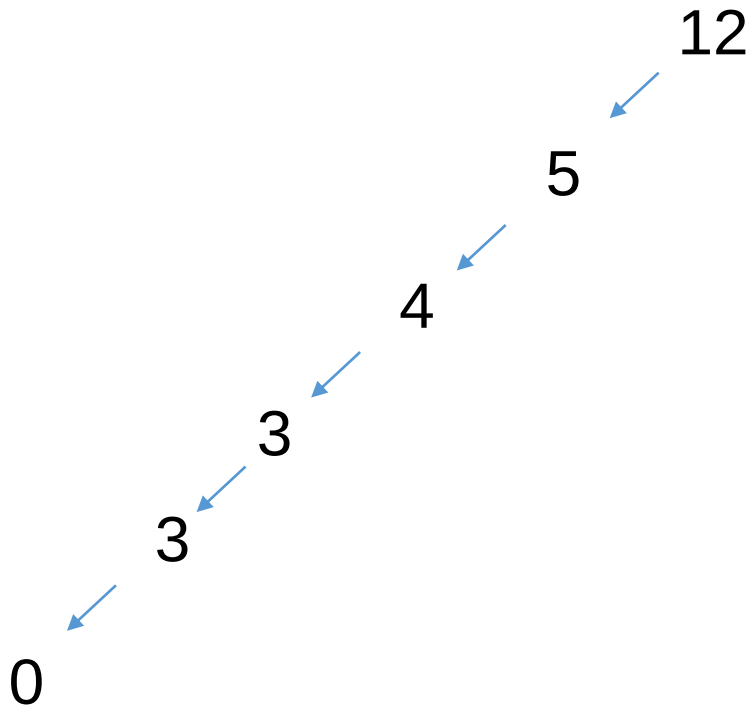
$$1 + 2 + 4 + \ldots + 2^h = 2^{h+1} - 1$$

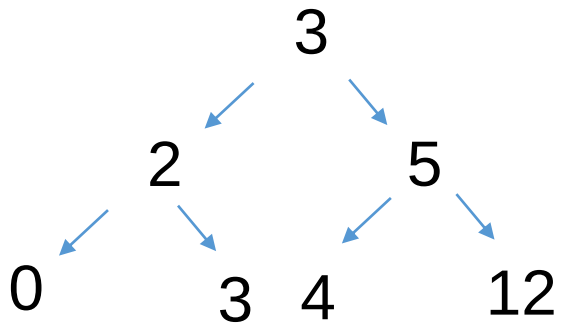
Définition :
Un arbre binaire est équilibré si sa hauteur h est d'un ordre de grandeur logarithmique par rapport au nombre de sommet.

Arbre binaire de recherche

Equilibre :

Exemples : pour la liste de valeurs suivante : 12, 5, 4 , 3, 3 , 0





Arbre Binaire Parfait

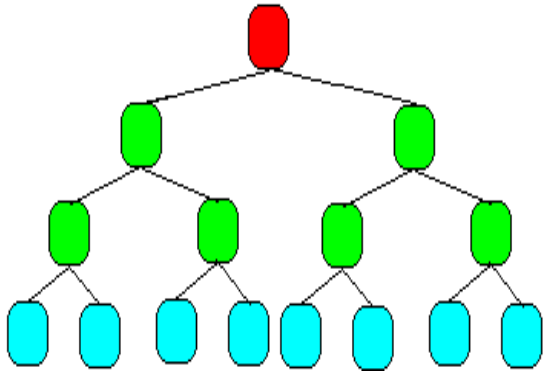
Arbre binaire parfait

Définition :

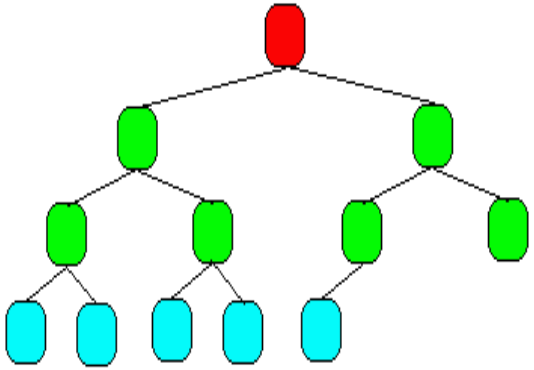
Un arbre binaire est parfait si :

1. Tous les noeuds de chaque niveau sont présents sauf éventuellement au dernier niveau où il peut manquer des noeuds (noeuds terminaux = feuilles)

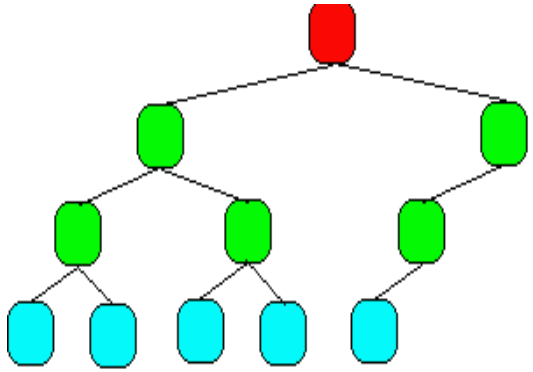
2. Les feuilles du dernier niveau doivent être regroupées à partir de la gauche de l'arbre.



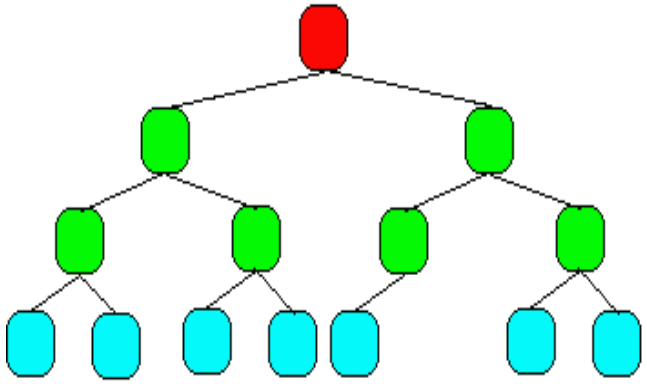
ARBRE BINAIRE PARFAIT



ARBRE BINAIRE PARFAIT



ARBRE BINAIRE NON PARFAIT



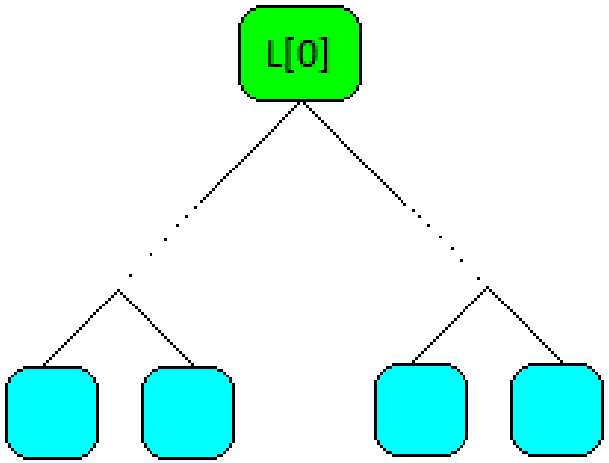
ARBRE BINAIRE NON PARFAIT

Arbre binaire parfait

Implémentation :

Un arbre binaire parfait peut être implémenter avec un simple tableau T , avec les règles suivantes :

Règle 1 : T[0] est la racine de l'arbre



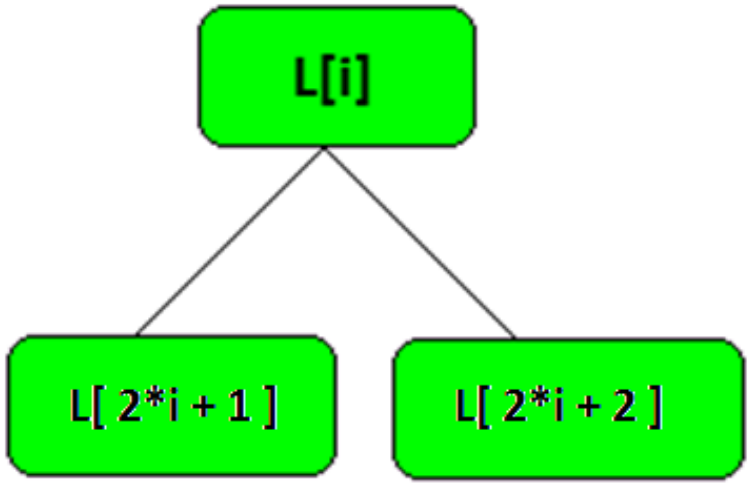
Arbre binaire parfait

Implémentation :

Un arbre binaire parfait peut être implémenter avec une simple liste L , avec les règles suivantes :

Règle 1 : L[0] est la racine de l'arbre

Règle 2: T[2*i+1] et T[2*i+2] sont les feuilles de T[i] :



Arbre binaire parfait

Implémentation :

Un arbre binaire parfait peut être implémenter avec une simple liste L , avec les règles suivantes :

Règle 1 : T[0] est la racine de l'arbre

Règle 2: T[2*i+1] et T[2*i+2] sont les feuilles de T[i] :

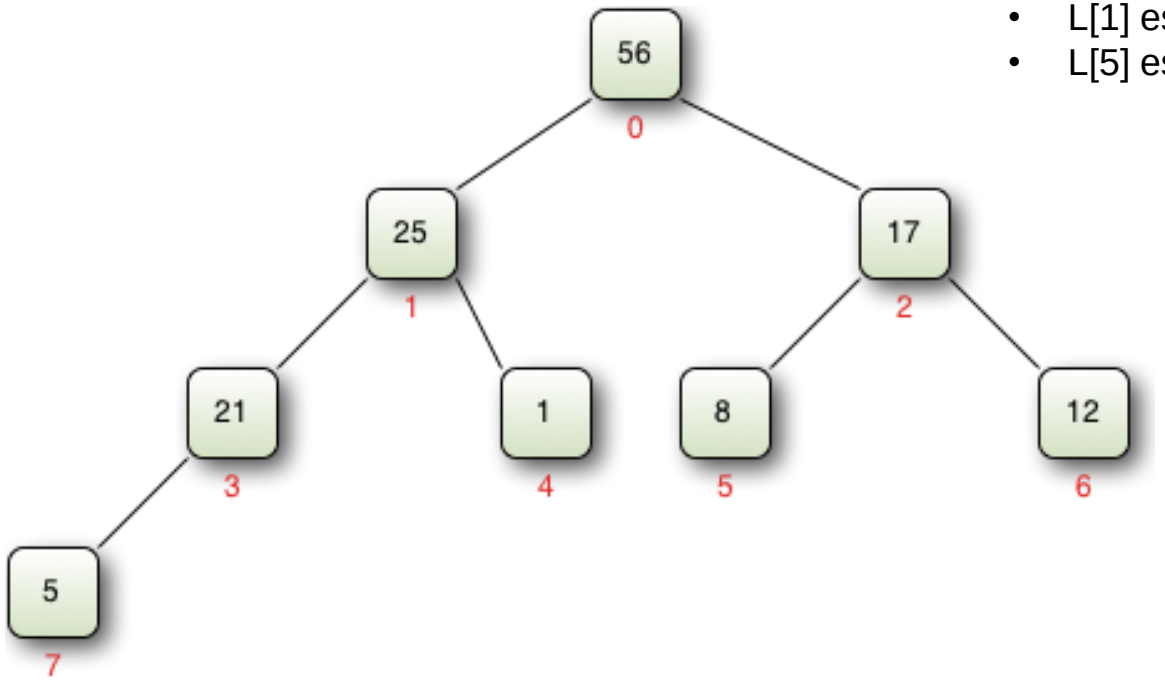
Règle 3 : Si i est supérieur ou égale à len(T)//2, T[i] est une feuille.

Règle 4: Le père de T[i] est T[(i-1)//2]

Arbre binaire parfait

Exemple :

56	25	17	21	1	8	12	5
0	1	2	3	4	5	6	7



- L[1] est le père de L[3] et L[4]
- L[5] est une feuille

Arbre binaire parfait

Applications:

L'idée qui sous-tend d'un arbre parfait consiste à voir le tableau comme un arbre binaire parfait. Il existe plusieurs applications des arbres binaires parfaits :

1. Tas binaire : File de priorité (Tas Min, Tas Max)
2. Tri Maximier

Tas binaire (Max / Min)

Le tas binaire

Problème:

On veut une structure de données qui permet de stocker un ensemble d'éléments qui se distingue par une propriété de comparaisons entre eux, et de pouvoir trouver efficacement à chaque fois l'élément le plus prioritaire.

On veut optimiser :

- Extraire l'élément le plus prioritaire
- Tester si il existe des éléments dans la file de priorité
- Insérer un élément dans la file de priorité
- Modifier la priorité d'un élément

Le tas binaire

Exemple :

Gérer un ensemble d'objet représentant des événements caractérisés par des dates {e1, e2, e3, e4, …}.

e1: conférence le 30/12/2022

e2: anniversaire le 30/09/2022

e3: mariage le 30/11/2022

e4: soutenance le 30/10/2022

Une solution possible : une liste triée par ordre décroissant des dates

e2	**e4**	**e3**	**e1**

Le tas binaire

Complexité des primitives d'une file de priorité implémenté avec une liste :

— supprimer l'élément de priorité maximale en O(n);

— accéder à l'élément de priorité maximale en O(1);

— insérer un nouvel élément associé à une priorité en O(n);

— modifier la priorité d'un élément (si vous pouvez le retrouver dans le tableau) en O(n).

Le tas binaire

Un tas binaire est une structure de données permettant de modéliser une file à priorités. Elle permet de réaliser les opérations suivantes :

— supprimer l'élément de priorité maximale en O(h) (ou O(log(n));

— accéder à l'élément de priorité maximale en O(1);

— insérer un nouvel élément associé à une priorité en O(h) (ou O(log(n));

— modifier la priorité d'un élément (si vous pouvez le retrouver dans le tableau) en O(h) (ou O(log(n)).

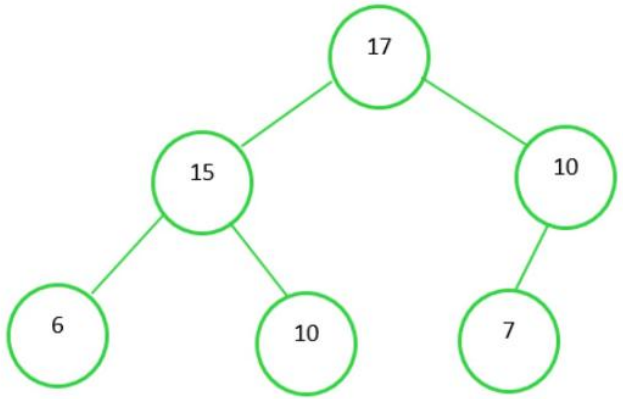
Le tas binaire

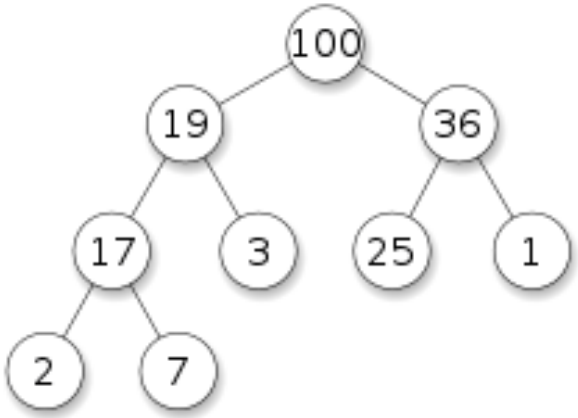
Un arbre binaire est appelé Tas binaire Max (Tas binaire Min) , si :

1. C’est un arbre binaire Parfait
2. La clé présente au nœud racine doit être supérieure ou égale (Inférieur ou égale) aux clés présentes à tous ses enfants. La même propriété doit être récursivement vraie pour tous les sous-arbres de cet arbre binaire

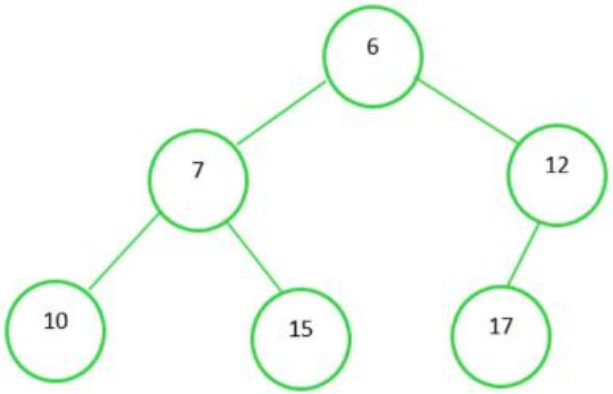
Le tas binaire

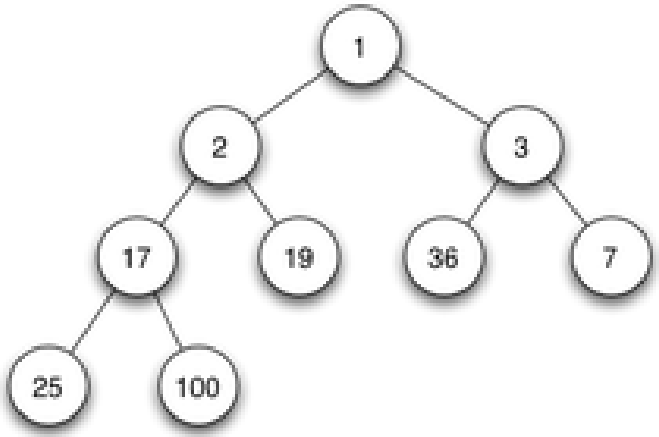
Exemples Tas Max :





Exemples Tas Min :

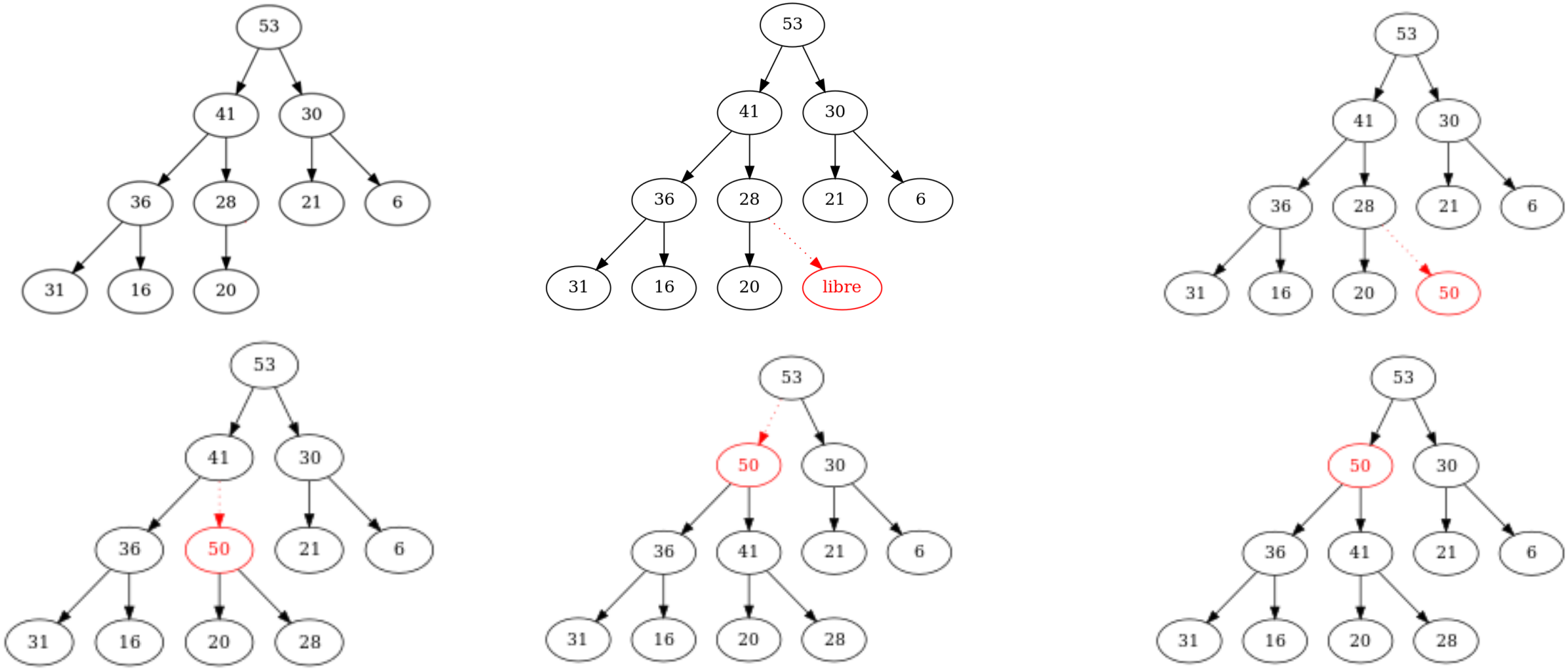




Le tas binaire

Insérer un élément :

Exemple : Insérer la valeur 50 (Percolation vers le haut)



Le tas binaire

Insérer un élément :

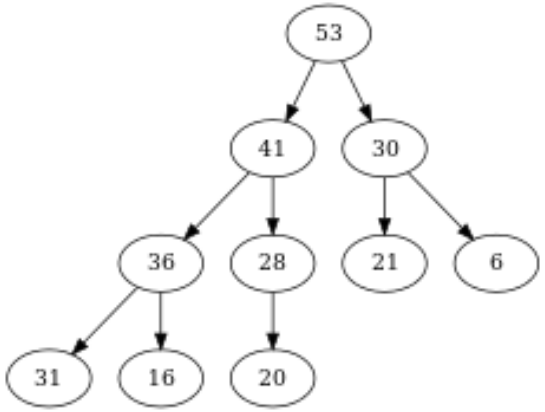
```c
#define MAX 100

typedef struct {
    int taille;
    int T[MAX];
} Tas;
```

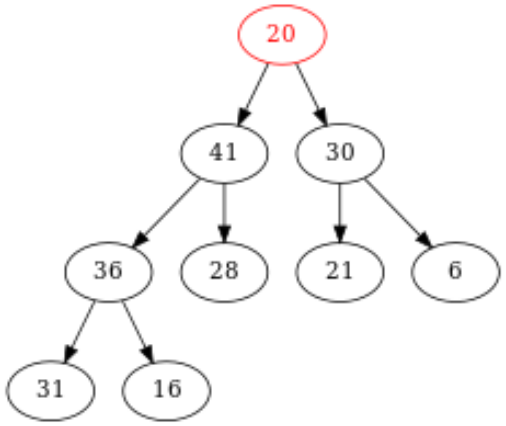
```c
void inserer(Tas *H, int valeur) {
    if (H->taille == MAX) {
        printf("Erreur : tas plein !\n");
        return;
    }
    // 1. insérer la valeur à la fin
    int i = H->taille;
    H->T[i] = valeur;
    H->taille++;

    // 2. faire remonter l'élément (up-heap bubbling)
    while (i > 0) {
        int parent = (i - 1) / 2;

        if (H->T[parent] >= H->T[i])
            break;  // tas valide

        // échanger avec le parent
        int tmp = H->T[parent];
        H->T[parent] = H->T[i];
        H->T[i] = tmp;
        i = parent;  // continuer la remontée
    }
}
```

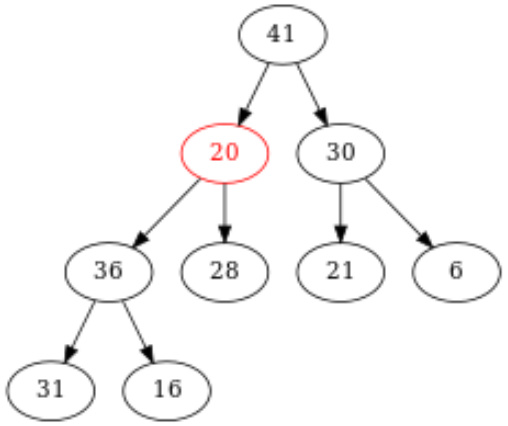
Le tas binaire

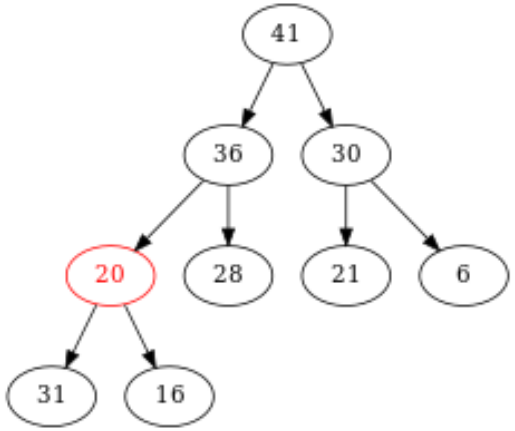
Supprimer l'élément prioritaire :

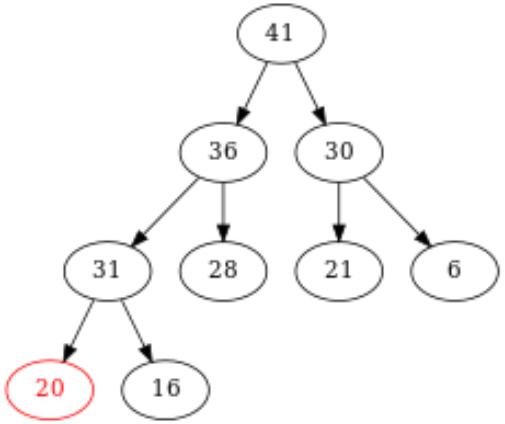
Exemple : Retirer l'élément prioritaire (la racine 53) (Percolation vers le bas) :











Le tas binaire

Supprimer un élément :

```
void entasser(Tas *T, int i) {
    // Indices des enfants gauche et droite
    int gauche = 2 * i + 1;
    int droite = 2 * i + 2;
    // On suppose au début que le plus grand élément est à la position i
    int plusGrand = i;
    // Vérifier si l'enfant gauche existe et est plus grand que l'élément actuel
    if (gauche < T->taille && T->T[gauche] > T->T[plusGrand]) {
        plusGrand = gauche;
    }
    // Vérifier si l'enfant droit existe et est plus grand que le plus grand
    if (droite < T->taille && T->T[droite] > T->T[plusGrand]) {
        plusGrand = droite;
    }
    // Si un enfant est plus grand que le parent, on échange et on continue récursivement
    if (plusGrand != i) {
        // échanger les valeurs
        echanger(&T->T[i], &T->T[plusGrand]);
        entasser(T, plusGrand);
    }
}
```

Le tas binaire

Supprimer un élément :

```
int supprimer(Tas *H) {
    if (H->taille == 0) {
        printf("Erreur : tas vide !\n");
        return -1; // Indiquer une erreur
    }

    int valeurSupprimee = H->T[0];

    // Remplacer la racine par le dernier élément
    H->T[0] = H->T[H->taille - 1];
    H->taille--;

    // Faire descendre l'élément à la racine (down-heap bubbling)
    entasser(H, 0);

    return valeurSupprimee;
}
```

Le tas binaire

Construire un Tas :

On souhaite construire un tas binaire à partir d'un ensemble d'éléments. De manière naïve on part du tas vide et on ajoute les éléments un par un avec l'algorithme ajouter, ce qui donne une complexité en O(n log n). Ce n'est pas optimal, il existe une manière de construire notre tas binaire en O(n):

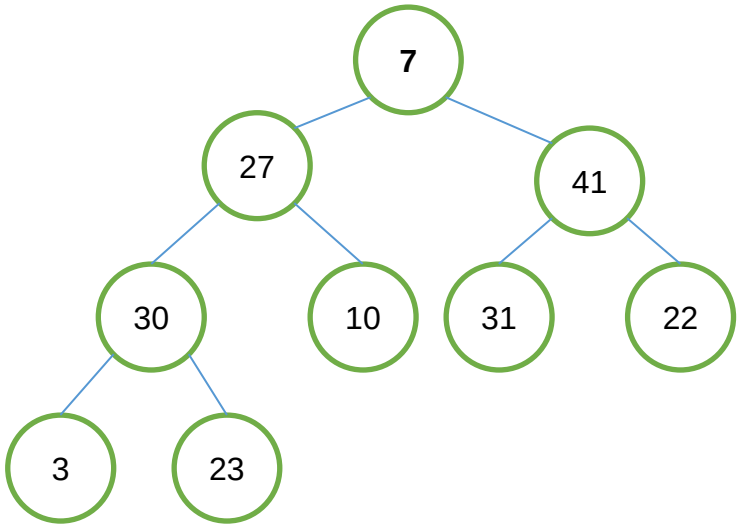
1. On construit un arbre binaire parfait avec tous les éléments sans se soucier de la contrainte d'ordre. Cela prend donc un temps linéaire en O(n).

2. On parcourt les sommets niveau par niveau en partant de l'avant dernier niveau (profondeur h-1) et dans chaque niveau on parcourt les sommets de la droite vers la gauche. Lors de notre parcours on effectue un percolate-down à partir de chaque sommet.

Le tas binaire

Construire un Tas Max

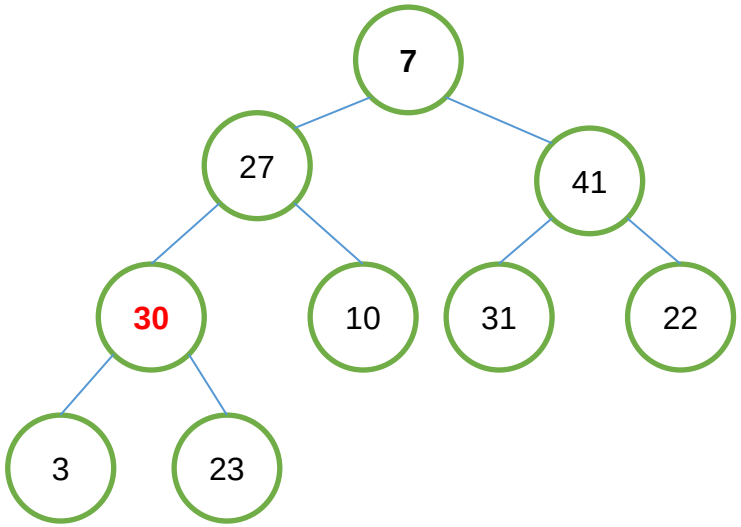
7	27	41	30	10	31	22	3	23



Le tas binaire

Construire un Tas Max

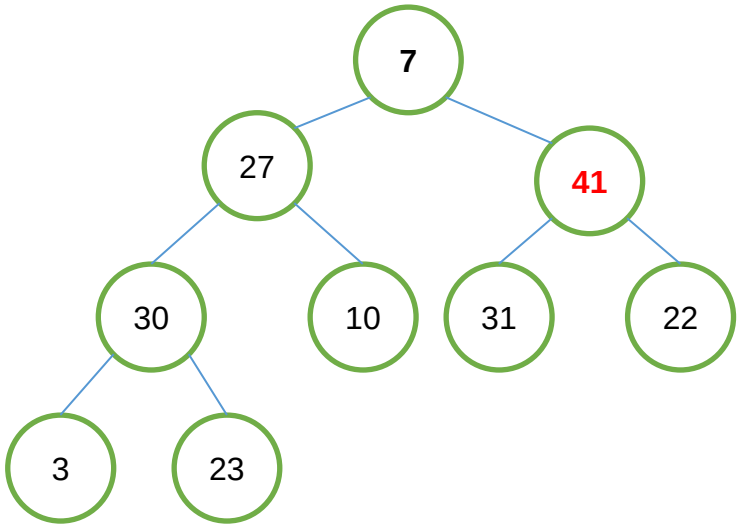
7	27	41	30	10	31	22	3	23



Le tas binaire

Construire un Tas Max

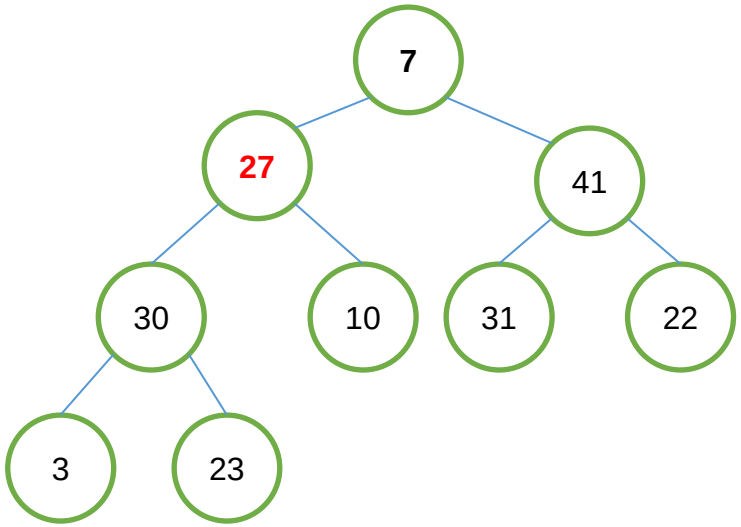
7	27	41	30	10	31	22	3	23



Le tas binaire

Construire un Tas Max

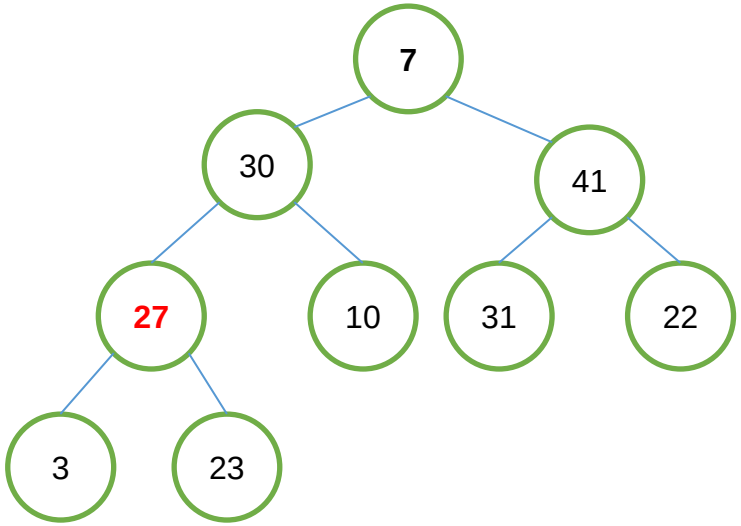
7	27	41	30	10	31	22	3	23



Le tas binaire

Construire un Tas Max

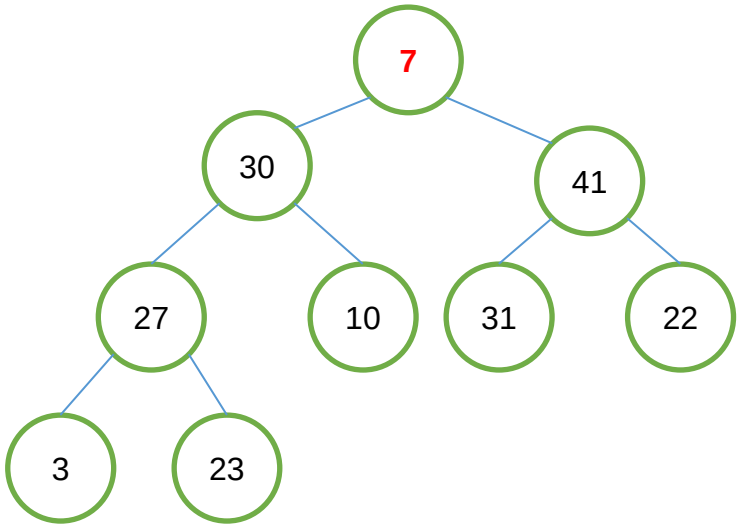
7	27	41	30	10	31	22	3	23



Le tas binaire

Construire un Tas Max

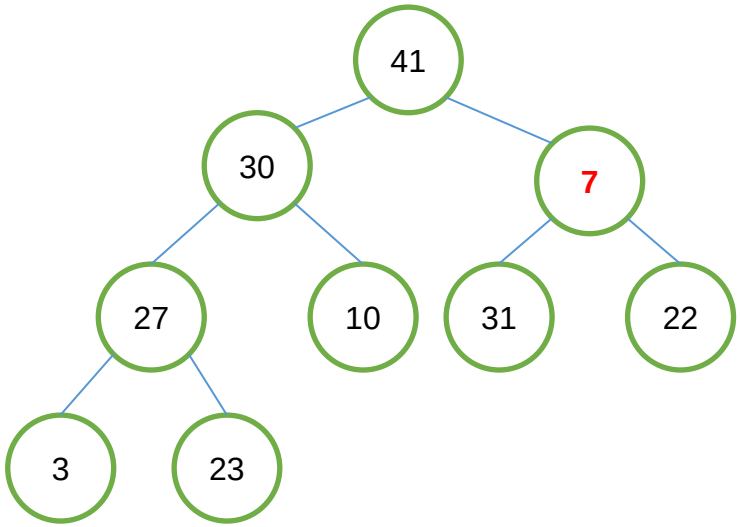
7	27	41	30	10	31	22	3	23



Le tas binaire

Construire un Tas Max

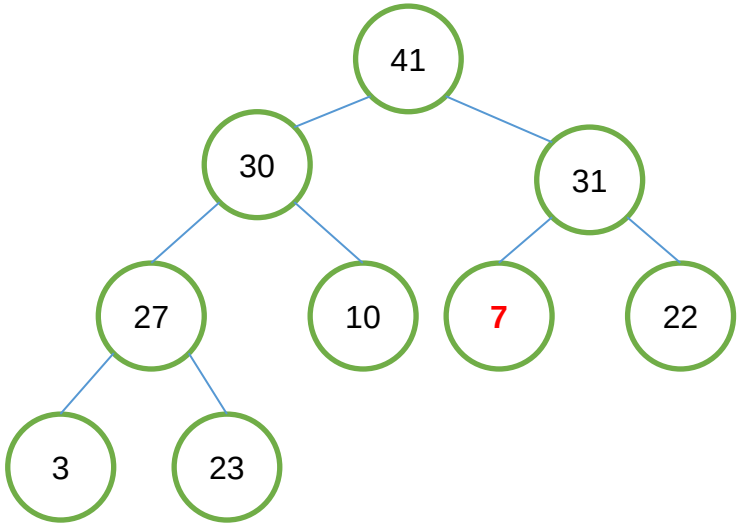
7	27	41	30	10	31	22	3	23



Le tas binaire

Construire un Tas Max

7	27	41	30	10	31	22	3	23



Le tas binaire

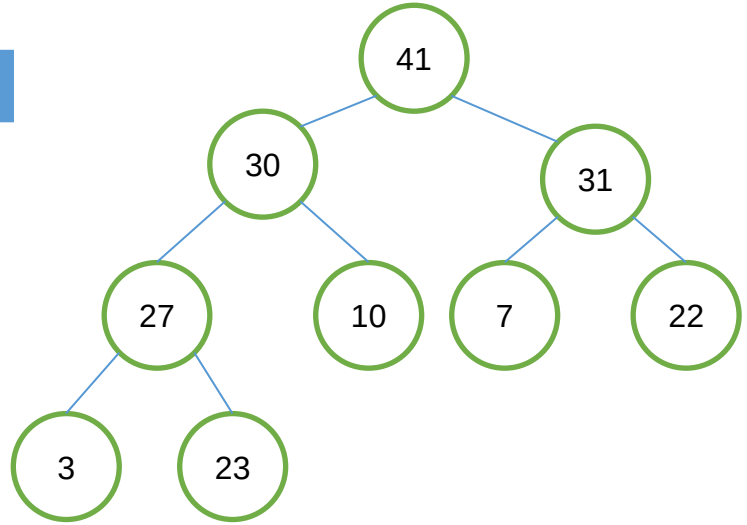
Construire un Tas Max

Liste initiale :

7	27	41	30	10	31	22	3	23

Nouvelle liste (Le tas) :

41	30	31	27	10	7	22	3	23



Le tas binaire

Construire un Tas Max

Exercice :

Donner manuellement le tas du tableau suivant L = { 3, 0, 12, 4, 3, 1, 9}

Le tas est : L = { 12 , 4, 9 , 0 , 3 , 1 , 3 }