Algorithmique et structures de données

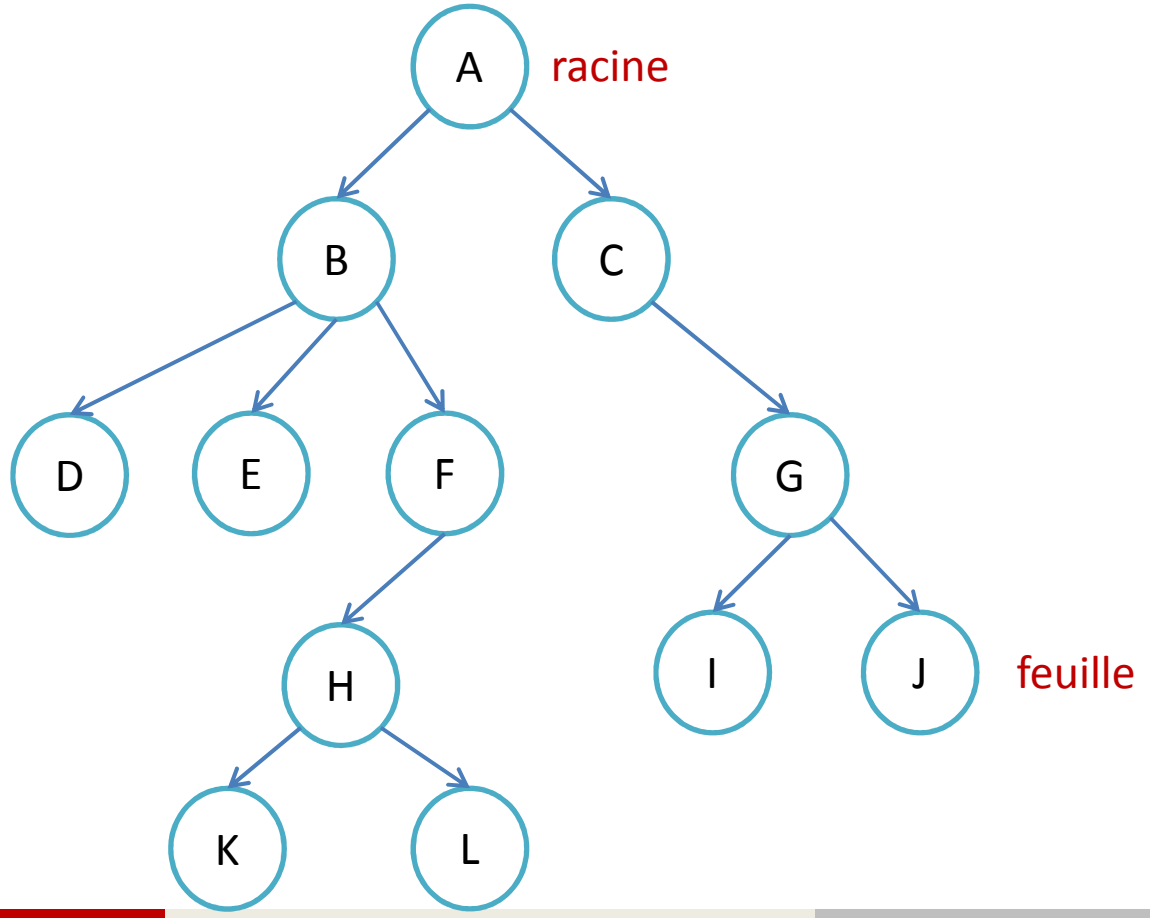
Chapitre 9 – Les arbres

Samar MOUCHAWRAB, PhD Eng

2A Cycle Préparatoire – Semestre 1
2017/2018

Introduction

Un arbre est une structure composée de nœuds et de feuilles (nœuds terminaux) reliés par des branches. On le représente généralement en mettant la racine en haut et les feuilles en bas (contrairement à un arbre réel).



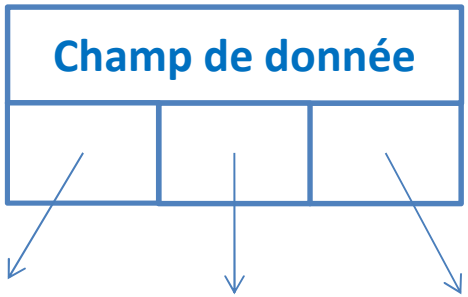
Principe

Un nœud

L'élément de base d'un arbre s'appelle le nœud. Il est constitué :

– d'un champ de donnée ;

– d'un ou plusieurs pointeurs vers des nœuds.



Notion d'arbre

Un arbre est un pointeur sur un nœud. Un arbre vide est un arbre qui ne contient pas de nœud. Il a donc la valeur NULL.

Terminologie

On utilise classiquement la terminologie suivante pour décrire les arbres :

Père : Le père A d'un nœud B est l'unique nœud tel que :
– un des sous arbres contient B
– hauteur(B) − hauteur(A) = 1.
Fils : Les fils d'un nœud sont les nœuds de ses sous arbres de hauteur immédiatement supérieure.
Frères : Les frères d'un nœud A sont les nœuds qui possèdent le même père que A.
Racine : La racine de l'arbre est l'unique nœud qui n'a pas de père.
Sous arbre : Un sous arbre d'un nœud A est un arbre dont la racine est un fils de A.
Feuille : Une feuille de l'arbre est un nœud qui n'a pas de fils.
Branche : Un arc qui relie deux nœuds est une branche.

Terminologie

Profondeur : la profondeur d'un nœud est la distance en terme de nœuds par rapport à la racine. Par convention, la racine est de profondeur 0.

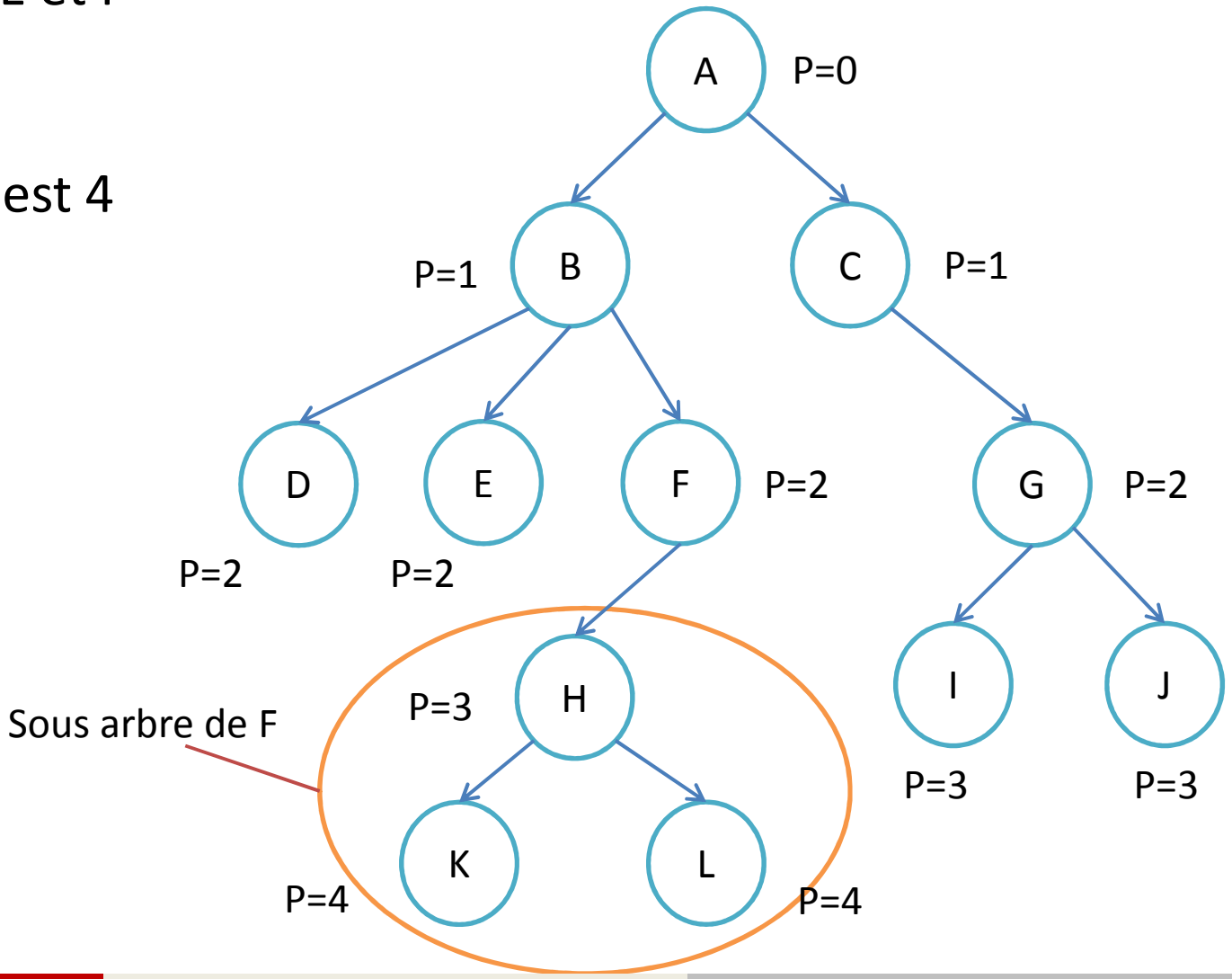
La **hauteur de l'arbre** est alors la profondeur maximale de ses nœuds. C'est à dire la profondeur à laquelle il faut descendre dans l'arbre pour trouver le nœud le plus loin de la racine.

On peut aussi définir la hauteur de manière récursive : la hauteur d'un arbre est le maximum des hauteurs de ses fils.

La hauteur d'un arbre est très importante. En effet, c'est un repère de performance. La plupart des algorithmes que nous verrons dans la suite ont une complexité qui dépend de la hauteur de l'arbre. Ainsi plus l'arbre aura une hauteur élevée, plus l'algorithme mettra de temps à s'exécuter.

Exemple

A est la racine
B est le père de D, E et F
I et J sont fils de G
K et L sont frères
Hauteur de l'arbre est 4



Arbres n-aires

Un arbre dont les nœuds ont au plus n fils est un arbre n-aire.
Lorsque n vaut 2, l'arbre est dit binaire.

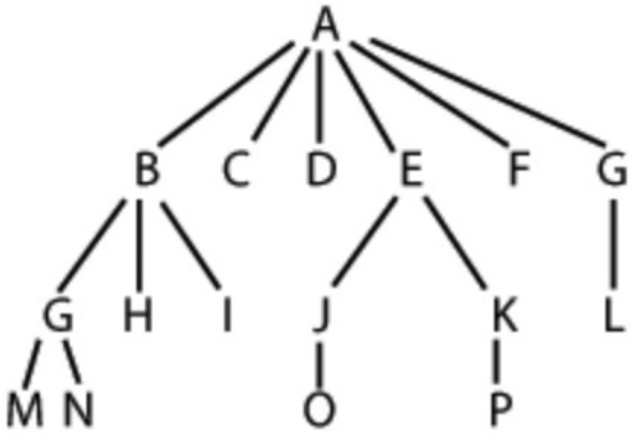
Dans le cas particulier de l'arbre binaire on utilise les termes de fils gauche et fils droit d'un nœud, ainsi que les notions de sous arbre gauche et de sous arbre droit.

La forme d'arbre la plus simple à manipuler est l'arbre binaire. De plus les formes plus générales d'arbres (avec $n > 2$) peuvent être représentées par des arbres binaires.

Conversion d'un arbre quelconque en binaire

Dans l'arbre de gauche, A a 6 fils : {B, C, D, E, F, G}. Il peut être converti en arbre binaire (comme celui de droite).

Cet arbre binaire peut être considéré comme l'arbre original incliné en longueur, avec les côtés noirs de gauche représentant les premiers fils et les côtés bleus de droite représentant ses prochains frères.



Applications des arbres

Les arbres sont des structures de données omniprésentes dans des domaines variés.

- On les trouve comme des arbres de décision, dans les tournois, en généalogie, comme arborescence des répertoires, ou expressions arithmétiques, ...
- En algorithmique du texte, on les utilise pour la compression, la recherche de motifs, la détection de répétitions, ...
- En géométrie algorithmique (quadtrees, octrees, KD-trees, arbres rouge-noir, ...)
- En linguistique et en compilation (arbres syntaxiques)
- Pour la gestion du cache, les bases de données, les systèmes de fichiers, ...

Arbres binaires

Un arbre binaire est une structure permettant de stocker une collection de données de même type et qui répond aux critères suivants.

Définition récursive: un arbre binaire est :

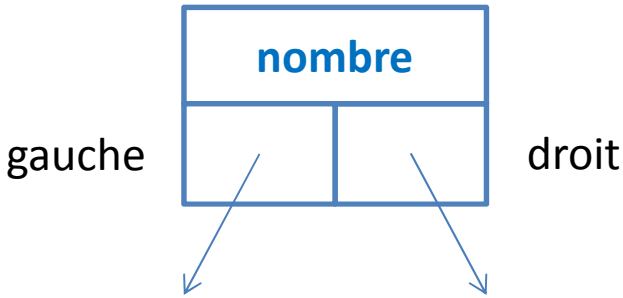
- Soit vide,
- Soit un nœud contenant une donnée et ayant 2 fils (gauche et droit) qui sont eux-mêmes des arbres binaires.

✓ L'espace mémoire utilisé par un arbre n'est pas contigu.
✓ La taille d'un arbre est inconnue à priori.
✓ Un arbre binaire est constitué de nœuds qui sont liés entre eux par des pointeurs.
✓ On ne peut accéder directement qu'au premier élément (la racine) de l'arbre.
✓ Pour accéder à un élément quelconque d'un arbre, il faut descendre dans l'arbre jusqu'à cet élément.

Implémentation

Implémentation d'un nœud en C qui correspond à un nœud de l'arbre et que l'on peut dupliquer autant de fois que nécessaire :

```
typedef struct Noeud Noeud;
struct Noeud {
    int nombre;
    Noeud *gauche;
    Noeud *droit;
};
```



Implémentation de l'arbre binaire en C :

```
typedef struct Arbre_Binaire Arbre_Binaire;
struct Arbre_Binaire {
    Noeud *racine;
};
```

Initialisation

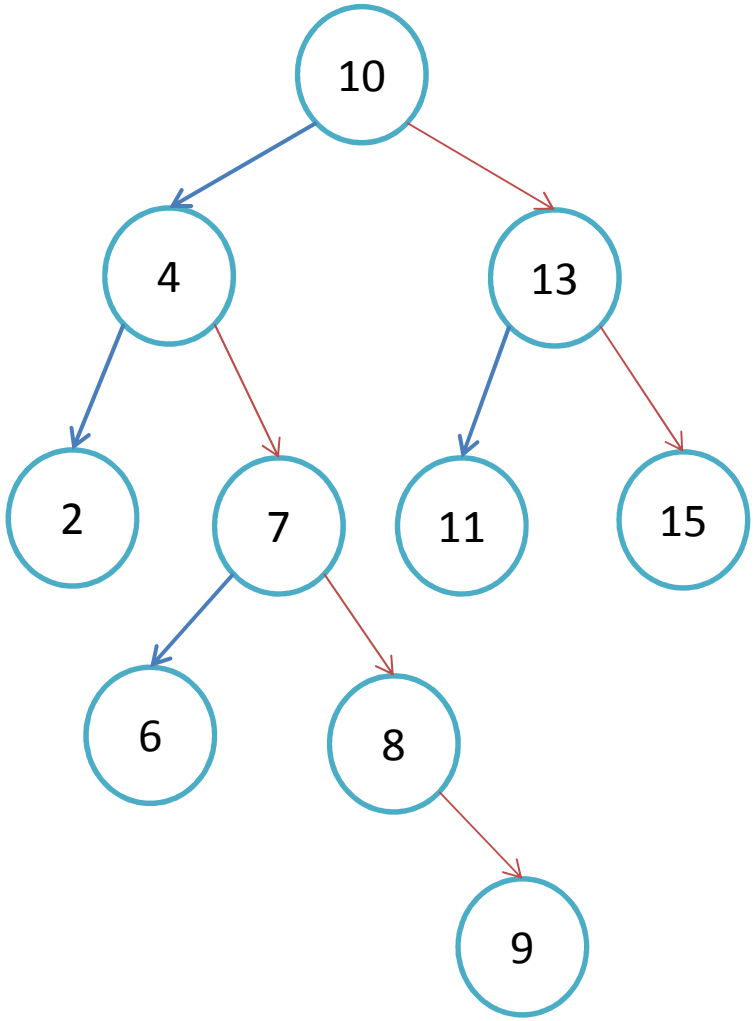
La fonction d'initialisation crée la structure (l'arbre) et la racine. L'arbre contient un seul nœud alors les pointeurs gauche et droit pointent à NULL :

```
Arbre_Binaire * initialisation(int Nb) {
    Arbre_Binaire *arbre = malloc(sizeof(*arbre));
    Noeud *noeud = malloc(sizeof(*noeud));

    if (arbre == NULL || noeud == NULL) {
        exit(EXIT_FAILURE);
    }

    noeud->nombre = Nb;
    noeud->gauche = NULL;
    noeud->droit = NULL;
    arbre->racine = noeud;

    return arbre;
}
```

Arbres binaires de recherche

Un arbre binaire de recherche (ABR) est un arbre binaire dans lequel chaque nœud possède une clé, telle que (1) chaque nœud du sous-arbre gauche ait une clé inférieure ou égale à celle du nœud considéré, et que (2) chaque nœud du sous-arbre droit possède une clé supérieure ou égale à celle-ci.

Selon la mise en œuvre de l'ABR, on pourra interdire ou non des clés de valeur égale. Les nœuds que l'on ajoute deviennent des feuilles de l'arbre.



Recherche dans un ABR

Un arbre binaire de recherche est fait pour faciliter la recherche d'informations.
La recherche dans un arbre binaire d'un nœud ayant une clé particulière est un procédé récursif:

Soit un sous arbre de racine n_i,
– si la valeur recherchée est celle de la racine n_i, alors la recherche est terminée. On a trouvé le nœud recherché.
– sinon, si n_i est une feuille (pas de fils) alors la recherche est infructueuse et l'algorithme se termine.
– si la valeur recherchée est plus grande que celle de la racine alors on explore le sous arbre de droite c'est à dire que l'on remplace n_i par son nœud fils de droite et que l'on relance la procédure de recherche à partir de cette nouvelle racine.
– de la même manière, si la valeur recherchée est plus petite que la valeur de n_i , on remplace n_i par son nœud fils de gauche avant de relancer la procédure.

Ajout d’un nœud dans un ABR

La fonction d’insertion d’un nœud insère l’élément dans l'arbre en tenant compte des critères de tri.

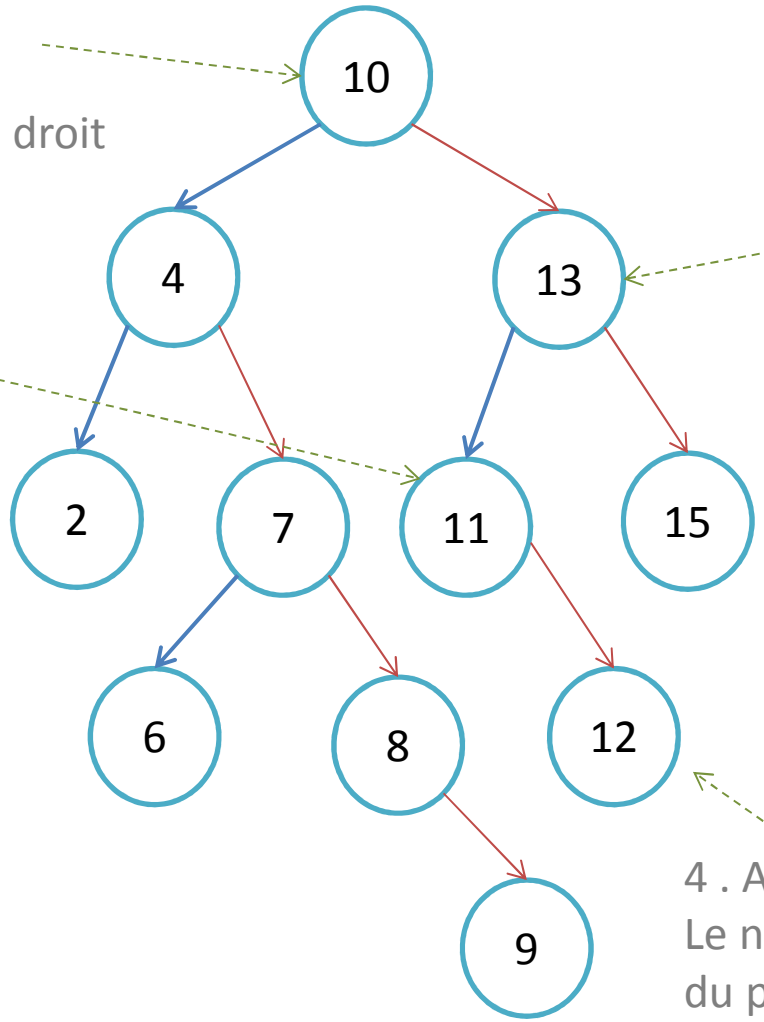
Donc si l'élément n'est pas le premier, on boucle afin d'avancer de nœud en nœud jusqu‘à atteindre un emplacement libre (pointeur à NULL) et à chaque nœud on part à droite si la clé est supérieure à celle du nœud courant ou à gauche si elle est inférieure ou égale à celle du nœud courant.

Le cas où l’arbre au départ est vide impose d'affecter l'adresse du nouveau nœud à la racine de l'arbre passé en paramètre à la fonction.

Exemple

Ajouter 12 à l'arbre.

1 . Actuel ici
12 > 10
On passe au fils droit



2 . Actuel ici
12 < 13
On passe au fils gauche

3 . Actuel ici
12 > 11
On passe au fils droit

4 . Actuel = NULL
Le nouveau nœud est attaché à droite du père 11

Ajout d'un nœud dans un ABR

```
void addNode(Arbre_Binaire *arbre, int Nb) {
    Noeud *nouveau = malloc(sizeof(Noeud));
    Noeud *tmp;

    if (arbre == NULL || nouveau == NULL) {
        exit(EXIT_FAILURE);
    }

    nouveau->nombre = Nb;
    nouveau->gauche = NULL;
    nouveau->droit = NULL;
    Noeud *actuel = arbre->racine;

    if (actuel != NULL) {
        while(actuel != NULL) {
            tmp = actuel;
            if (Nb == actuel->nombre) {
                exit(EXIT_FAILURE);
            }
```

Ajout d'un nœud dans un ABR

```
                else if (Nb > actuel->nombre ) {
                        actuel = actuel->droit;
                        if(actuel == NULL) tmp->droit = nouveau;
                 } else {
                        actuel = actuel->gauche;
                        if(actuel == NULL) tmp->gauche = nouveau;
                 }
            }
     }
     else
        arbre->racine = nouveau;
}
```

Chercher une clé dans un ABR

Le principe est identique à la fonction d'insertion. On suit le cheminement en partant à droite ou à gauche selon la valeur de la clé. A chaque nœud on vérifie si on est en présence de l'élément recherché, si oui on retourne la valeur 1. Quand on arrive au bout de la branche si on ne l'a pas trouvé on retourne 0. On est certain qu'il ne se trouve pas dans une autre branche. Il n'y a donc pas besoin de tester.

```
int searchNode(Arbre_Binaire *arbre, int Nb) {
    if (arbre == NULL ) {
        exit(EXIT_FAILURE);
    }
    Noeud *actuel = arbre->racine;
    while(actuel != NULL) {
        if(Nb == actuel->nombre) return 1;
        if(Nb > actuel->nombre ) actuel = actuel->droit;
        else actuel = actuel->gauche;
    }
    return 0;
}
```

Affichage d'un ABR

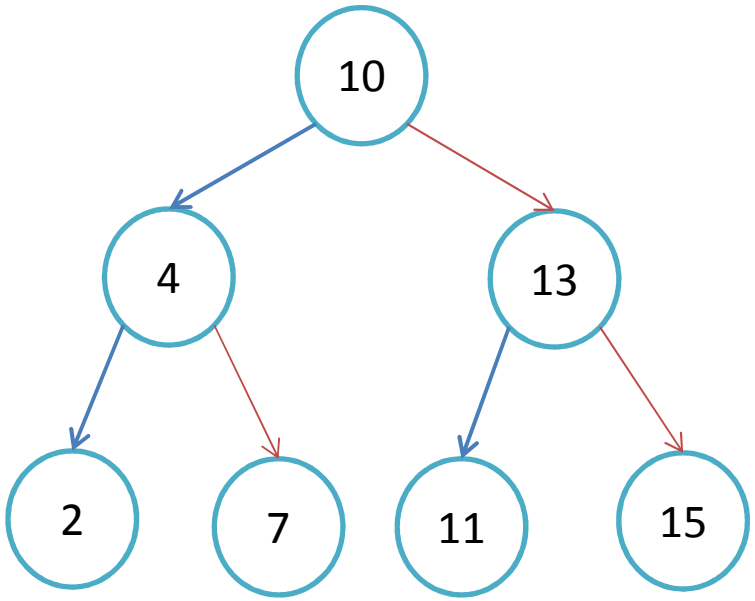
Pour cela il va falloir parcourir l'arbre complet, et là nous avons guère le choix si l'on veut faire cela de façon simple, nous devons utiliser des fonctions récursives.

```
void printTree(Noeud *racine) {
    if (racine == NULL ) {
        return;
    }

    printTree(racine->gauche);

    printf("Cle = %d\n", racine->nombre);

    printTree(racine->droit);
}
```

Exemple

Si on exécute la fonction récursive sur l'arbre suivant :

```
printTree(Noeud : 10)
    printTree(Noeud : 4)
        printTree(Noeud : 2)
            printTree(Noeud : NULL)
            printf(Cle : 2)
            printTree(Noeud : NULL)
        printf(Cle : 4)
        printTree(Noeud : 7)
            printTree(Noeud : NULL)
            printf(Cle : 7)
            printTree(Noeud : NULL)
    printf(Cle : 10)
    printTree(Noeud : 13)
        printTree(Noeud : 11)
            printTree(Noeud : NULL)
            printf(Cle : 11)
            printTree(Noeud : NULL)
        printf(Cle : 13)
        printTree(Noeud : 15)
            printTree(Noeud : NULL)
            printf(Cle : 15)
            printTree(Noeud : NULL)
```



Parcours d’un arbre

Le parcours d’un arbre 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 et le parcours en largeur.

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 1 puis ceux du niveau 2 et ainsi de suite jusqu'à l'exploration de tous les nœuds.

Parcours d'un arbre en profondeur

On distingue **deux types de parcours d'arbre en profondeur** : parcours à gauche, et parcours à droite. Un parcours en profondeur à gauche signifie que l'on parcourt les branches de gauche avant les branches de droite.

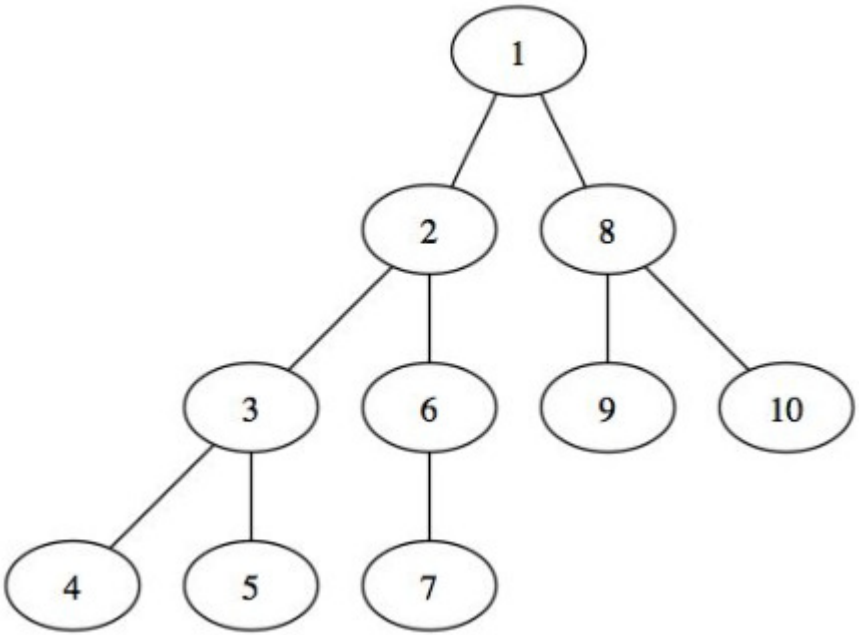
La deuxième caractéristique du parcours est dit préfixe. Cela signifie que l'on affiche la racine de l'arbre, on parcourt tout le sous arbre de gauche, une fois qu'il n'y a plus de sous arbre gauche on parcourt les éléments du sous arbre droit. Ce type de parcours peut être résumé en trois lettres : R G D (pour Racine Gauche Droit).

On a aussi deux autres types de parcours : le **parcours infixe** et le **parcours suffixe** (appelé aussi postfixe). Le parcours infixe affiche la racine après avoir traité le sous arbre gauche, après traitement de la racine, on traite le sous arbre droit (c'est donc un parcours G R D). Le parcours postfixe effectue donc le dernier type de schéma : sous arbre gauche, sous arbre droit puis la racine, c'est donc un parcours G D R.

Parcours d’un arbre en profondeur

Exemple: Parcours en profondeur

Le parcours en profondeur à gauche de l'arbre suivant donne (RGD) :
1, 2, 3, 4, 5, 6, 7, 8, 9, 10



Parcours d’un arbre en largeur

Pour le parcours d’un arbre en largeur, **on utilise une structure de données de type file d'attente**. Le principe est le suivant, 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.

Parcours d'un arbre en largeur

Exemple : parcours de l'arbre ci-dessous

La file est au départ vide, on commence par lire la racine, on la met dans la file d'attente.

File : A

On traite A (retirée de la file), on met son fils gauche et son fils droit dans la file

File : B -> C

On traite B, il n'a pas de fils

File : C

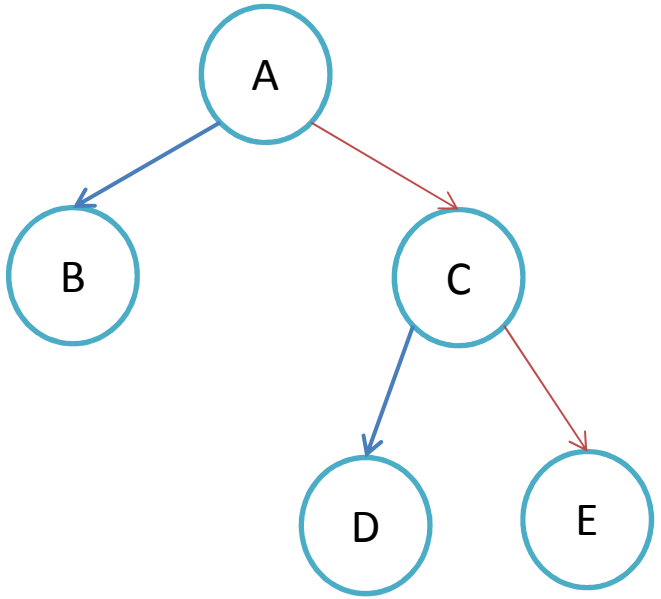
On traite C, retiré de la file et ses fils sont mis dans la file

File : D -> E

On traite D, pas de fils

File : E

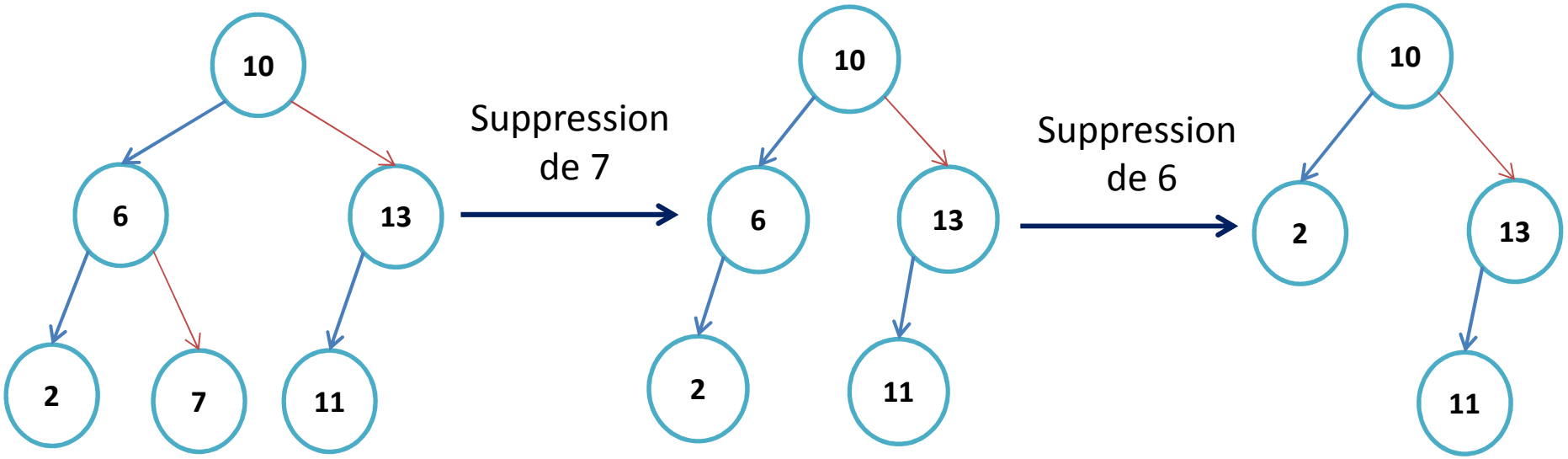
On traite E, file vide, algorithme terminé



Suppression d'un nœud

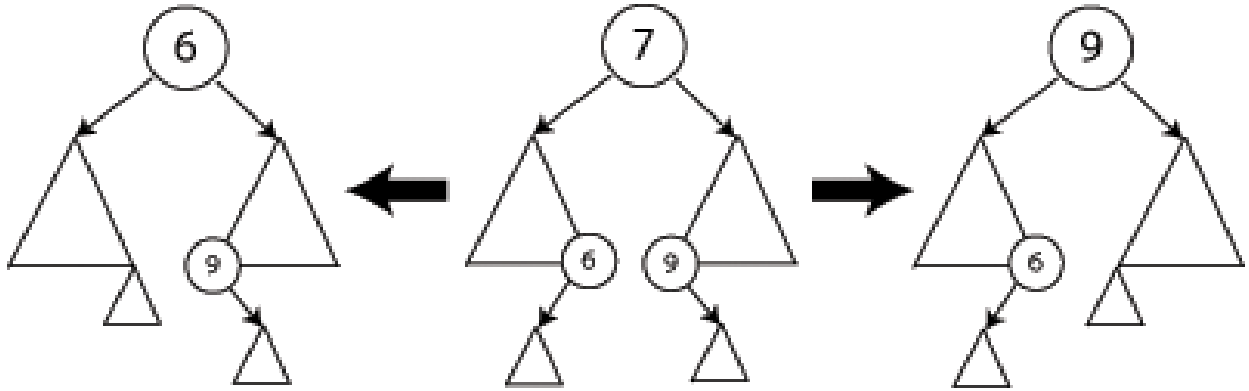
Plusieurs cas sont à considérer, une fois que le nœud à supprimer a été trouvé à partir de sa clé :

1. Suppression d'une feuille : Il suffit de l'enlever de l'arbre vu qu'elle n'a pas de fils.
2. Suppression d'un nœud avec un fils : Il faut l'enlever de l'arbre en le remplaçant par son fils.



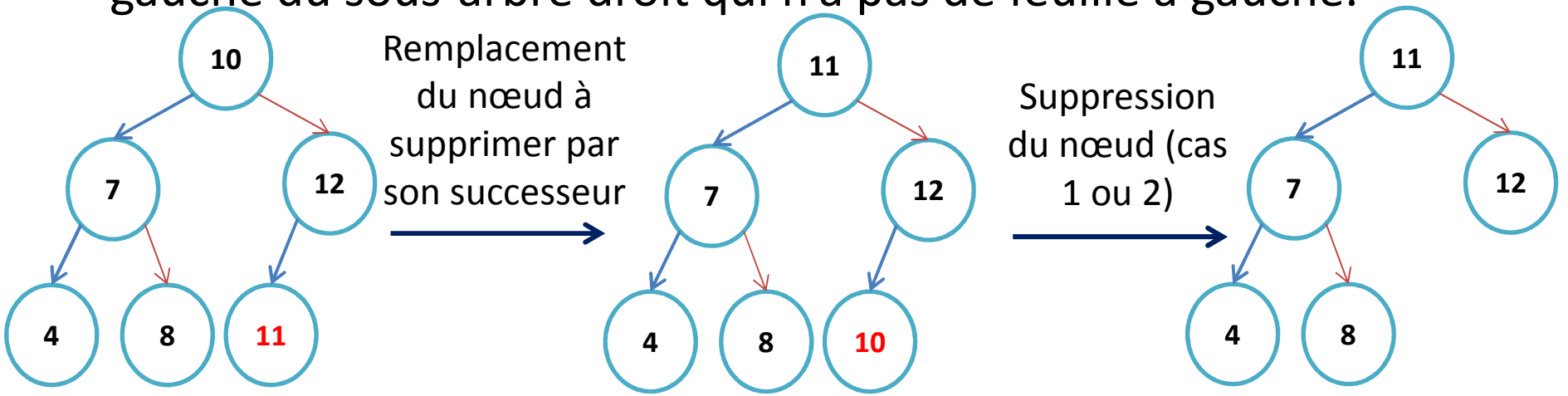
Suppression d’un nœud

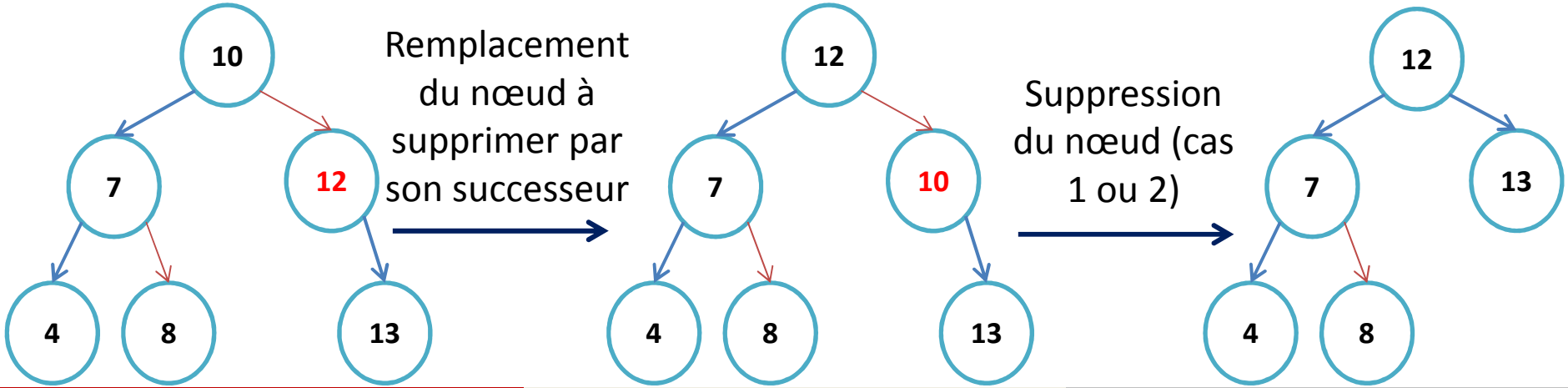
3. Suppression d'un nœud avec deux fils : Supposons que le nœud à supprimer soit appelé N (le nœud de valeur 7 dans le graphique ci-dessous). On échange le nœud N avec son successeur le plus proche (le nœud le plus à gauche du sous-arbre droit ci-dessous, le nœud de valeur 9) ou son plus proche prédécesseur (le nœud le plus à droite du sous-arbre gauche ci-dessous, le nœud de valeur 6). Cela permet de garder une structure d'arbre binaire de recherche. Puis on applique à nouveau la procédure de suppression à N, qui est maintenant une feuille ou un nœud avec un seul fils.



Suppression d'un nœud

3. Exemple : suppression de 10, remplacement par son successeur le plus proche. Le successeur le plus proche est ou bien une feuille qui est la plus à gauche du sous-arbre droit, ou bien le nœud le plus à gauche du sous-arbre droit qui n'a pas de feuille à gauche.





Exercice

Ecrire une fonction qui retourne 1 si arbre est vide et 0 si arbre est non vide.

```
int estVide (Arbre_Binaire *arbre) {
        if (arbre->racine == NULL) return 1;
        else return 0;
}
```

Exercice

Ecrire une fonction qui retourne 1 si le nœud est une feuille et 0 dans le cas contraire.

```
int estFeuille(Noeud *noeud) {
        if (noeud == NULL)
                return 1;
        else if ((noeud->gauche==NULL)&& (noeud->droit==NULL))
                return 1;
        else
                return 0;
}
```

Exercice

Ecrire une fonction qui retourne le sous arbre gauche d'un arbre.

```
Arbre_Binaire * gauche (Arbre_Binaire *arbre) {
        if ( estVide(arbre) )
                return NULL;
        else {
                Arbre_Binaire *g = malloc(sizeof(*g));
                g->racine = arbre->racine->gauche;
                return g;
        }
}
```

Exercice

Ecrire une fonction qui calcule la hauteur d'un arbre.

```
int max(int a, int b) {
        if (a>b) return a;
        else return b;
}

int hauteur (Arbre_Binaire *arbre) {
        if ( arbre == NULL )
                return -1;
        else if (estFeuille(arbre->racine))
                return 0;
        else
                return 1 + max( hauteur(gauche(arbre) ) ,
                        hauteur(droit(arbre) ) );
}
```

Exercice

Ecrire une fonction qui calcule la taille d'un arbre (nombre total de nœuds).

```
int taille_arbre(Arbre_Binaire *arbre) {
    if (arbre == NULL || arbre->racine == NULL) {
        return 0;
    } else if (estFeuille(arbre->racine))
        return 1;
    else
        return (1 + taille_arbre(gauche(arbre)) +
                taille_arbre(droit(arbre)));
}
```

Tas

Après les arbres binaires de recherche, nous allons voir un nouveau type d'arbres : les arbres tassés, appelé aussi tas (heap en anglais). **Ce sont aussi des arbres binaires**.

Sur les arbres binaires de recherche on peut constater un défaut, qui est le fait que certaines branches peuvent être beaucoup plus longues que d'autres dans certaines circonstances. Ce qui a un effet néfaste sur les performances.

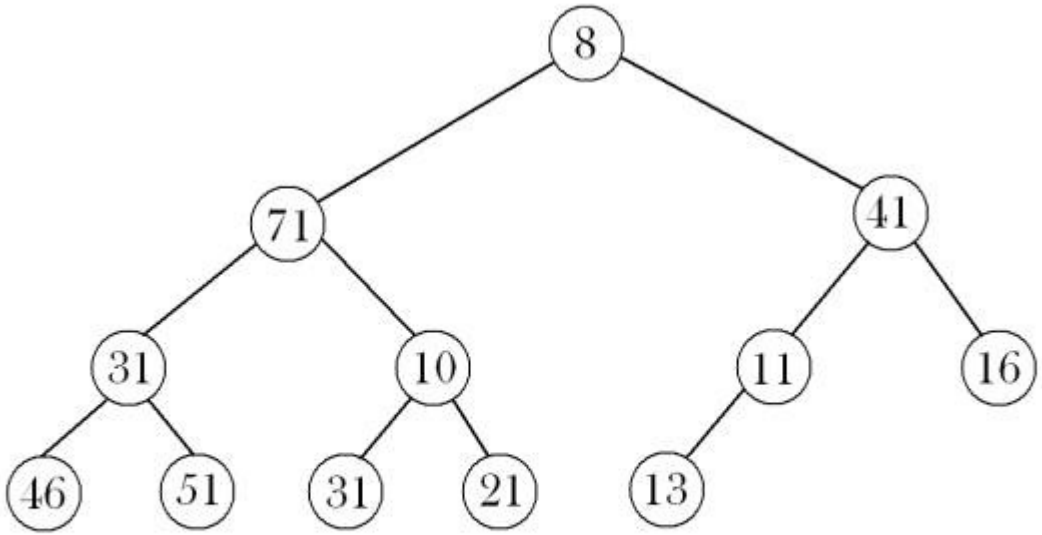
Le tas pallie à ce problème. La hauteur de sa plus grande branche ne pourra être supérieure que d'une unité par rapport à la plus petite. C'est à dire que si sa plus grande branche passe par 15 nœuds sa plus petite passera par 14 nœuds au minimum (15 au cas où l'arbre soit complet).

Bien sûr comme tous les arbres, les données seront insérées selon un certain ordre, afin de les retrouver facilement.

Tas

Un tas est un arbre binaire tel que, h la hauteur de l'arbre :

- pour tout $i \in [0, h-1]$, il y a exactement 2^i nœuds à la profondeur i ;
- une feuille a une profondeur h ou $h - 1$;
- les feuilles de profondeur maximale sont tassées sur la gauche.



Application des Tas

L’application de tas comme structure de données la plus utilisée est la file de priorité.

Une file de priorité est un type abstrait élémentaire sur laquelle on peut effectuer trois opérations :
- insérer un élément ;
- extraire l'élément ayant la plus grande clé (ou la plus petite) ;
- tester si la file de priorité est vide ou pas.

Application des files de priorités

Des applications des files de priorité comme :

- Ordonnancement de tâches d'un système d'exploitation
- Contrôle aérien
- Chercher la sortie dans un labyrinthe
- Admission des patients à la salle d'urgence

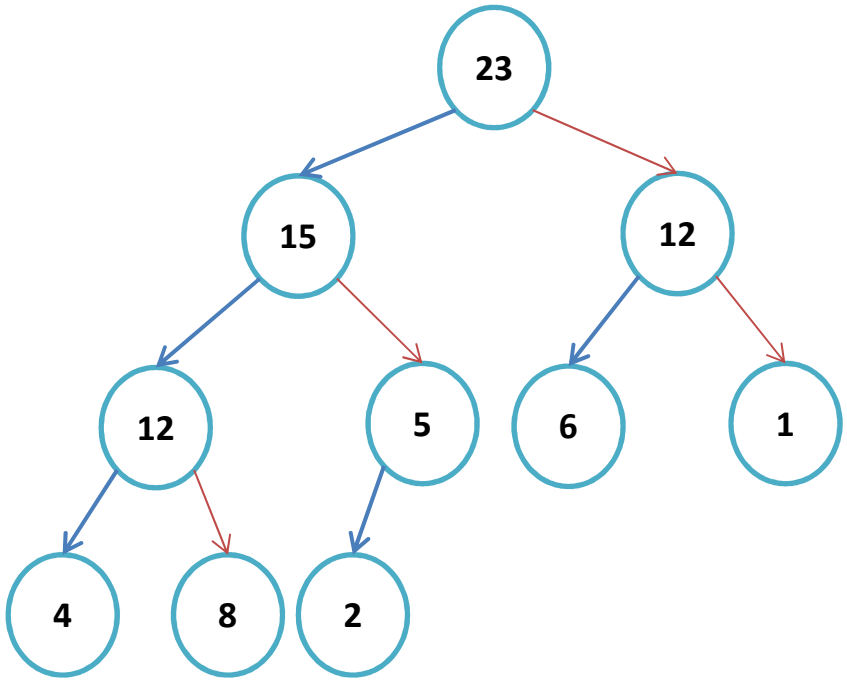
Les problèmes à traiter sont ?

- Les tâches qui nécessitent le moins de temps doivent être traitées en premier
- Les avions qui ont moins de carburant doivent descendre avant
- Les noeuds les plus probables sont à explorer en premier
- Les patients ayant les cas les plus urgents doivent rentrer en premier

Tas comme file de priorité

Un tas représentant une file de priorités est un arbre binaire tassé tel que le contenu de chaque nœud soit supérieur ou égal à celui de ses fils.

Dans le cas où on traite les nœuds ayant la clé la plus petite en premier, alors l'ordre sera plutôt que le contenu de chaque nœud soit inférieur ou égal à celui de ses fils.



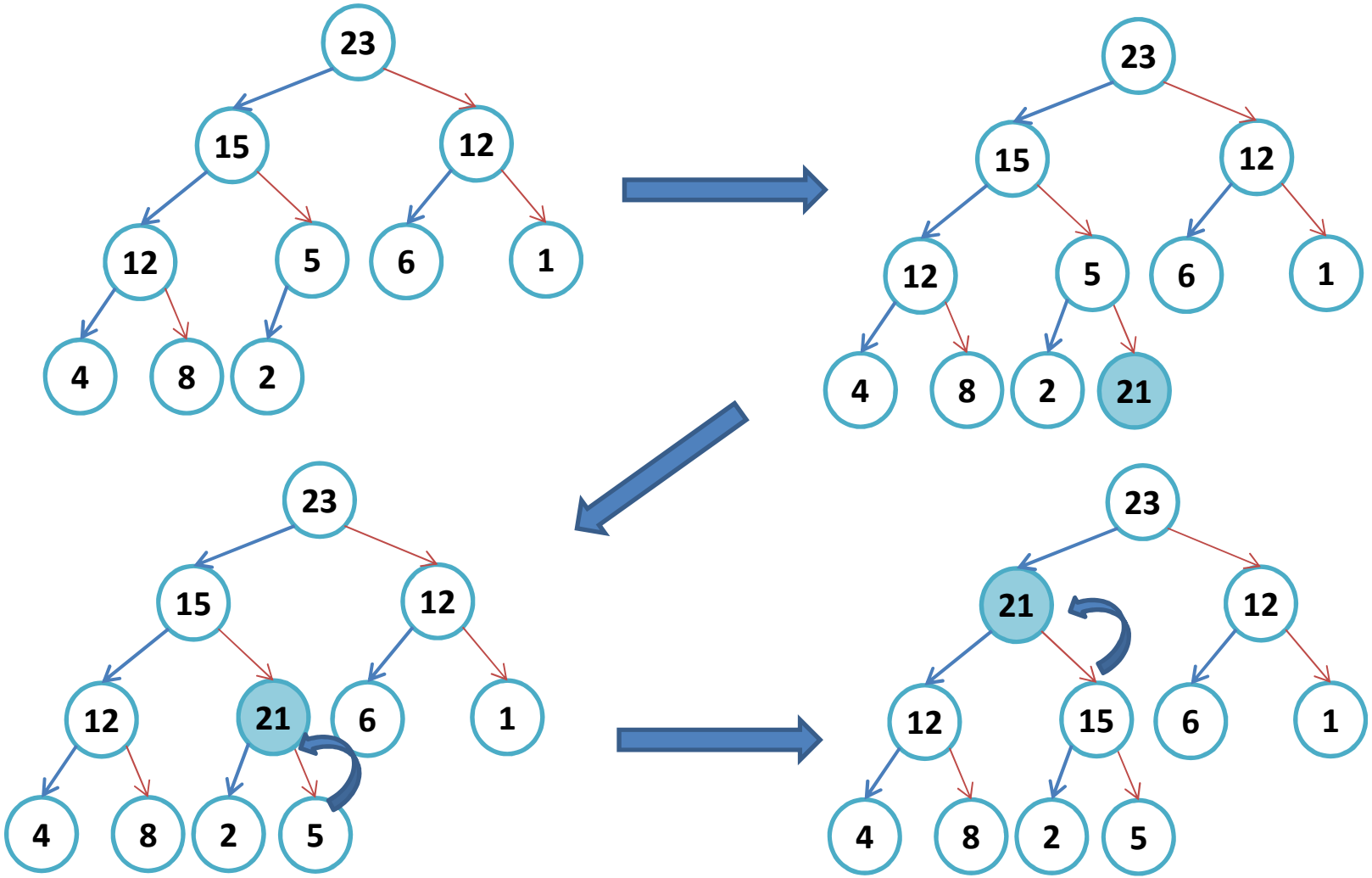
Insertion d'un élément dans tas file de priorité

Pour insérer un élément dans un tas file de priorité, le principe est le suivant :

- Insérer le nouveau nœud dans la première position libre pour respecter la contrainte d'un tas (arbre binaire tassé à gauche)
- Permuter répétitivement avec le parent jusqu'à ce que la condition de tas file de priorité soit respectée (le contenu de chaque nœud soit supérieur ou égal à celui de ses fils)

Insertion d'un élément dans tas file de priorité

Exemple : Insérer 21



Suppression d'un élément d'un tas file de priorité

L'élément à supprimer est l'élément ayant la plus grande priorité. Cet élément est celui contenu dans la racine.

Le principe de l'algorithme de suppression est le suivant:

1. Supprimer la racine
2. Mettre la dernière feuille à la place de la racine
3. Trouver le plus grand fils du noeud
4. Permuter les deux noeuds si nécessaire pour respecter les contraintes d'un tas file de priorité
5. Répéter les étapes 3 et 4 tant que nécessaire

Suppression d'un élément d'un tas file de priorité

Exemple : Supprimer un élément