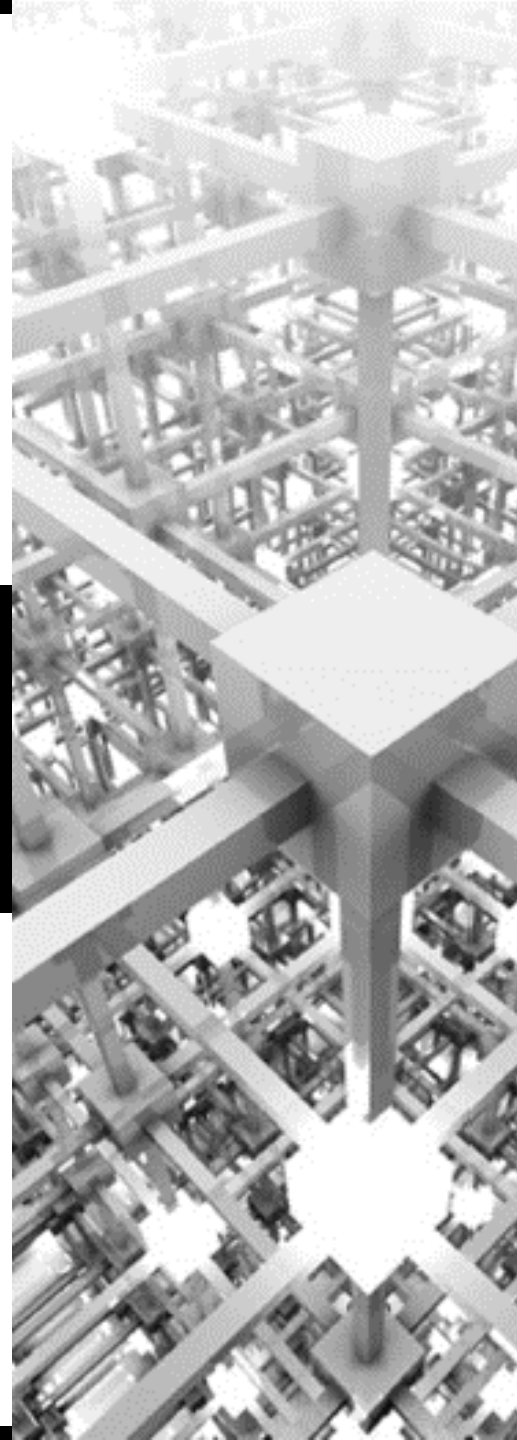
ENSAM

STRUCTURES DE DONNÉES EN C

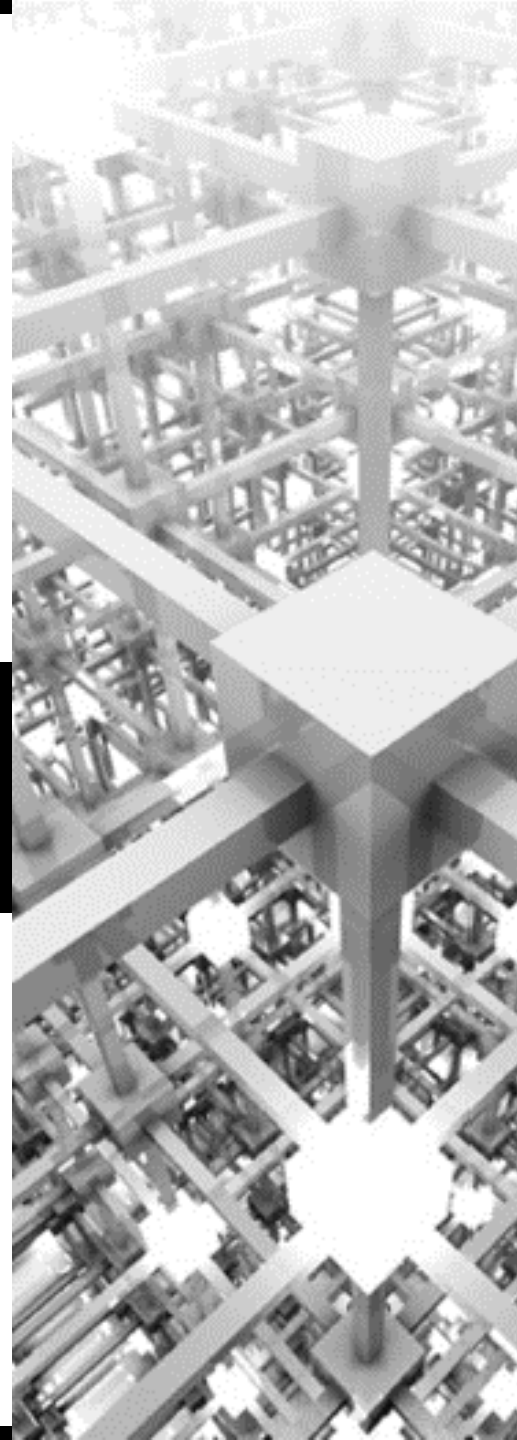
1ère Année «Cycle ingénieur : Intelligence Artificielle et Génie Informatique»

2023/2024

Dep. Informatique
Pf. CHERGUI Adil



UH2C | ENSAM



Séance 6

LES ARBRES BINAIRES DE RECHERCHE

Objectifs de la séance :

- Le rôle et l'utilité des ABR
- Comprendre les règles de gestion des ABR
- L'implémentation des arbres binaires

Pf. Adil CHERGUI

INTRODUCTION

Définition

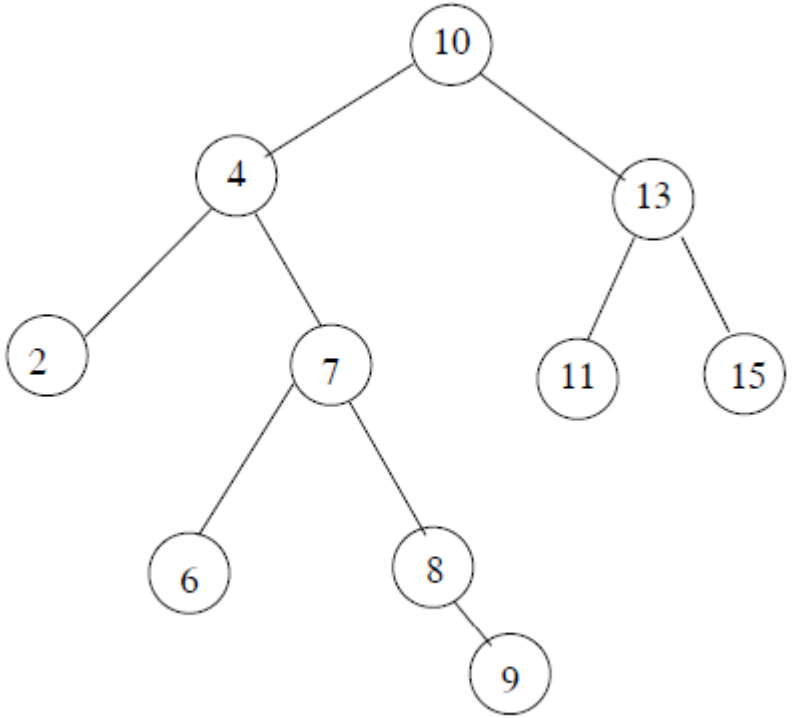
Dans un **arbre binaire de recherche (ABR)** tous les éléments sont dotés d'une clé (valeur) et ils sont positionnés dans l'arbre en fonction de cette clé.

Cette clé rend lisible et utilisable la notion d'**ordre** dans un arbre.

La clé est en général (dans le cours) un nombre qui est ajouté au **nœud**. C'est grâce à ce nombre que le nœud peut être rangé et identifié à une place précise dans l'arbre.

Tous les éléments peuvent ensuite être retrouvés rapidement grâce à leur clé, **sans être obligé** de parcourir **tout** l'arbre systématiquement. Les éléments rangés dans l'arbre sont en quelque sorte triés en fonction de leur clé. Il est facile d'**ajouter** des nouveaux éléments ou de **supprimer** des éléments sans modifier l'ordre de l'arbre.

Exemple d'arbre de recherche:



INTRODUCTION

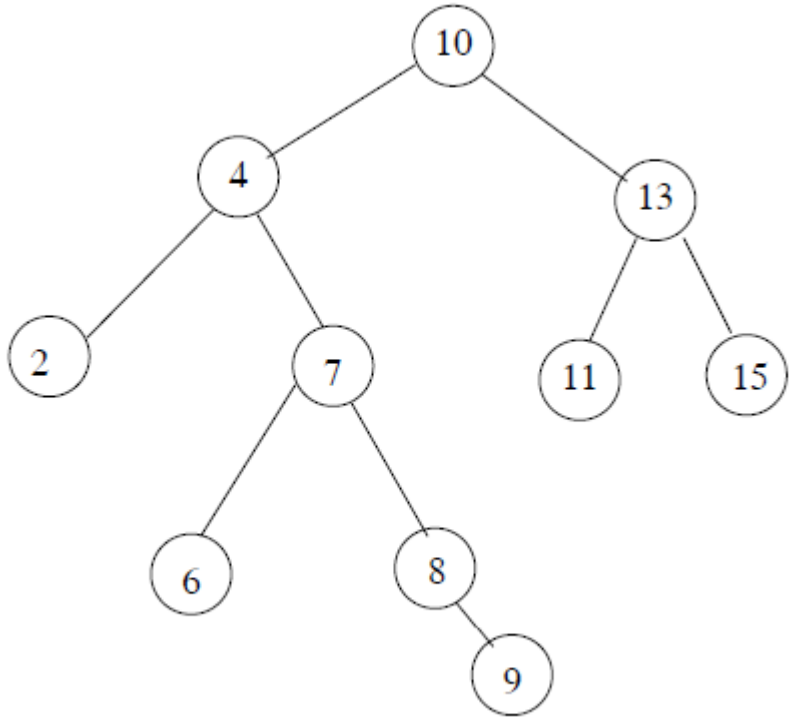
Propriétés

Un **Arbre Binaire de Recherche (abrégé ABR)** est un arbre binaire qui respecte à tout moment trois propriétés fondamentales auxquelles obéissent toutes les clés des nœuds :

- Tous les nœuds du **Sous-Arbre de Gauche (abrégé SAG)** d'un nœud de l'arbre ont une clé (valeur) **inférieure ou égale** à la sienne.
- Tous les nœuds du **Sous-Arbre de Droite (abrégé SAD)** d'un nœud de l'arbre ont une clé (valeur) **supérieure ou égale** à la sienne.
- Nous ajoutons à cela, une troisième règle : Toutes les clés sont différentes, il n'y a pas de clés identiques.

L'exemple ci-contre représente un arbre de recherche.

Exemple d'arbre de recherche:



STRUCTURE DE DONNÉES

Structure représentant les nœuds

Notre objectif est de mettre en place un **arbre binaire de recherche** avec **ses fonctions usuelles** de traitement. Il s'agit d'un **arbre dynamique** (même concept que les listes chainées mais cette fois ce n'est pas linéaire!!).

La clé est un entier dans les codes chapitre (le champ *Valeur*). Mais cela n'empêche pas que on peut intégrée d'autres **objets simples** ou **structurés** dans les cas réels, pourvus que la **composante** sur laquelle la **relation d'ordre** est formulée soit **identifiée**. Nous utilisons dans ce chapitre la structure ci-contre :

Code

```
typedef struct Arbre
{
    int Valeur;
    struct Arbre * FG;
    struct Arbre * FD;
} Arbre;
```

LES FONCTIONS USUELLES

Insertion d'un élément dans l'arbre

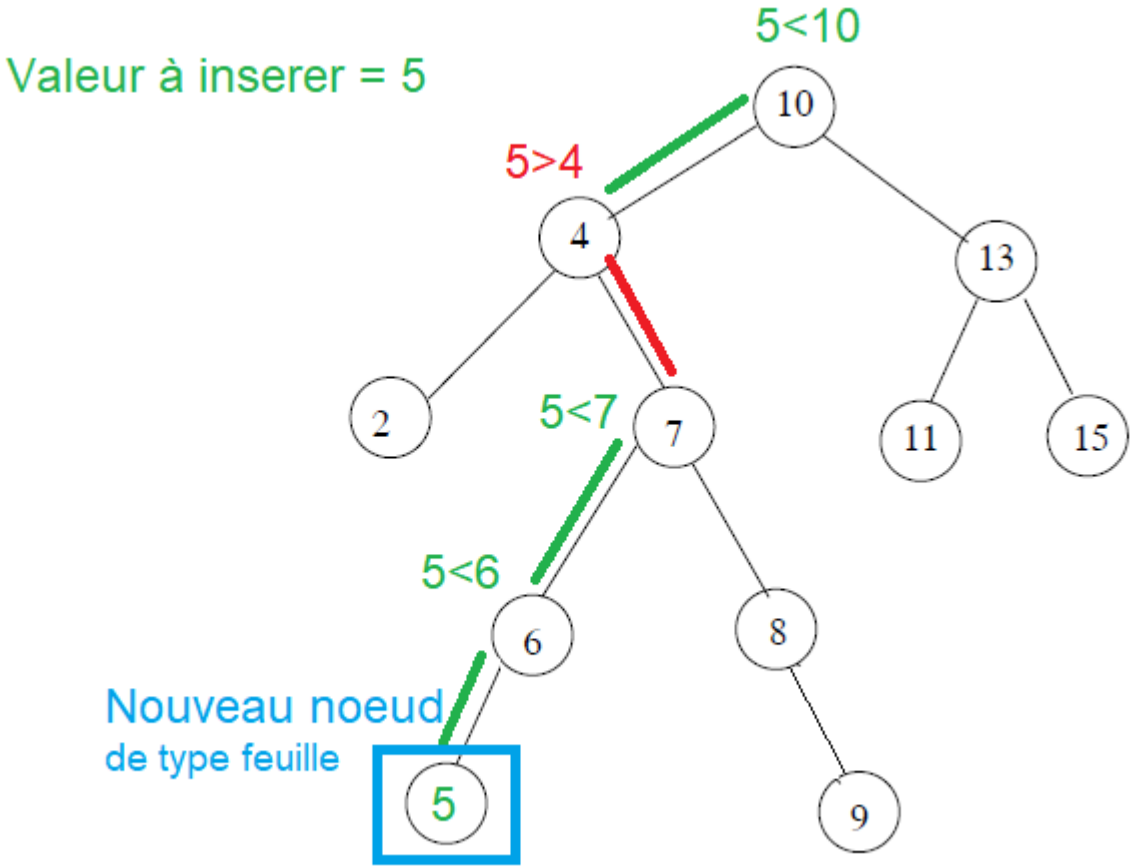
L'insertion d'un élément dans un arbre binaire de recherche se fait **toujours** en **feuille de l'arbre**, **jamais** au niveau des **nœuds internes**.

Pour connaître l'endroit adéquat où insérer il faut descendre dans l'arbre en s'orientant à chaque nœud en fonction de la clé.

Si l'arbre en question est vide, nous créons **un nouveau nœud (feuille)** dans lequel nous allons mettre la valeur.

Si la valeur du nouveau nœud est **plus petite** que celle du **nœud courant** de l'arbre, nous procédons à l'insertion dans son **SAG** (Sous-Arbre Gauche), sinon l'insertion se fera dans son **SAD** (Sous-Arbre Droit), jusqu'à arriver au niveau des feuilles, là où se trouve la bonne place.

Exemple d'insertion



LES FONCTIONS USUELLES

Insertion d'un élément dans l'arbre

D'après la définition du procédé de l'insertion, nous constatons que dans le codage de cette fonction, nous devons utiliser la ***récursivité***.

Pour insérer dans l'arbre un élément il suffit de partir de la **racine** et de s'orienter à chaque nœud en fonction de la valeur comparée à celle du nouveau nœud à insérer. Si la nouvelle clé est inférieure partir à gauche sinon partir à droite et *il ne peut pas y avoir de clé identique (Unique)* par définition dans l'arbre de recherche. La descente continue jusqu'à arriver à une **feuille**, tout en bas de l'arbre.

Une fois arrivé à la bonne place sur un pointeur à **NULL** de la feuille, créer un nœud, lui affecter la valeur à insérer et affecter l'adresse de ce nouveau nœud au pointeur de la feuille.

Code fonction : InserValeur

```
Arbre * InserValeur(Arbre * Ar,int valeur)
{
    if (Ar!=NULL)
    {
        if (Ar->Valeur > valeur)
        {
            Ar->FG=InserValeur(Ar->FG,valeur);
        }
        else
        {
            Ar->FD=InserValeur(Ar->FD,valeur);
        }
    }
    else
    {
        Ar=(Arbre *)malloc(sizeof(Arbre));
        Ar->Valeur=valeur;
        Ar->FD=NULL;
        Ar->FG=NULL;
    }
return Ar;
}
```

LES FONCTIONS USUELLES

Rechercher de valeur dans l'arbre

Un arbre binaire de recherche est fait pour faciliter la recherche d'informations. La recherche d'un nœud particulier de l'arbre peut être définie simplement de manière récursive:

Soit un sous-arbre de racine ***Ar***,

– si la valeur recherchée est celle de la racine ***Ar***, alors la recherche est terminée. On a trouvé le nœud recherché.

– sinon, si ***Ar*** 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 ***Ar*** 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 ***Ar***, on remplace ***Ar*** par son nœud fils de gauche avant de relancer la procédure..

Si l'arbre **est équilibré,** chaque itération divise par 2 le nombre de nœuds candidats. La complexité est donc en ***O(log2 n***) si **n** est le nombre de nœuds de l'arbre.

LES FONCTIONS USUELLES

Rechercher de valeur dans l'arbre

Pour rechercher un élément il suffit de partir de la racine avec valeur à rechercher. La fonction prend en paramètre la racine et valeur.

La racine ne sera pas modifiée, c'est juste un parcours.

La fonction retourne l'adresse du nœud qui correspond à la valeur et NULL si cette clé ne correspond à aucun élément de l'arbre. La fonction est récursive. Elle s'appelle elle-même vers la gauche ou vers la droite tant que le nœud n'est pas trouvé.

Code fonction : RechercheValeur

```
Arbre * RechercheValeur(Arbre * Ar,  int valeur)
{
    if (Ar!=NULL)
    {
        if (Ar->Valeur>valeur)
        {
            Ar=RechercheValeur(Ar->FG,valeur);
        }
        else
        {
            if (Ar->Valeur<valeur)
            {
                Ar=RechercheValeur(Ar->FD,valeur);
            }
        }
        return Ar;
    }
    return Ar;
}
```

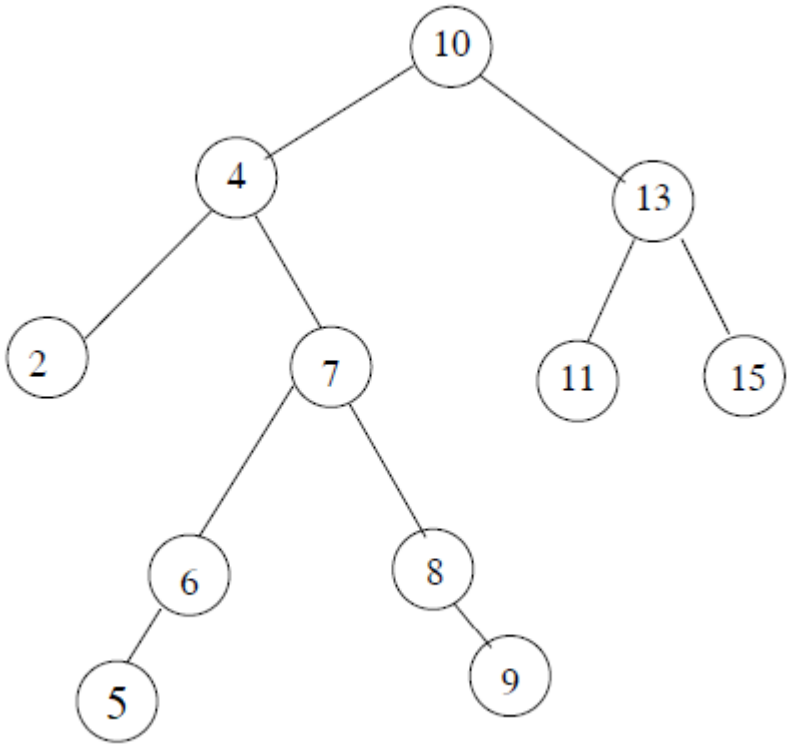
LES FONCTIONS USUELLES

Suppression d’un élément dans l'arbre de recherche

Comme l'ajout, la suppression où qu'elle soit, doit respecter l'ordre de l'arbre de recherche. Il faut distinguer trois cas :

- Suppression d’une feuille
- Suppression d’un nœud à un seul fils
- Suppression d’un nœud qui a deux fils

Exemple :



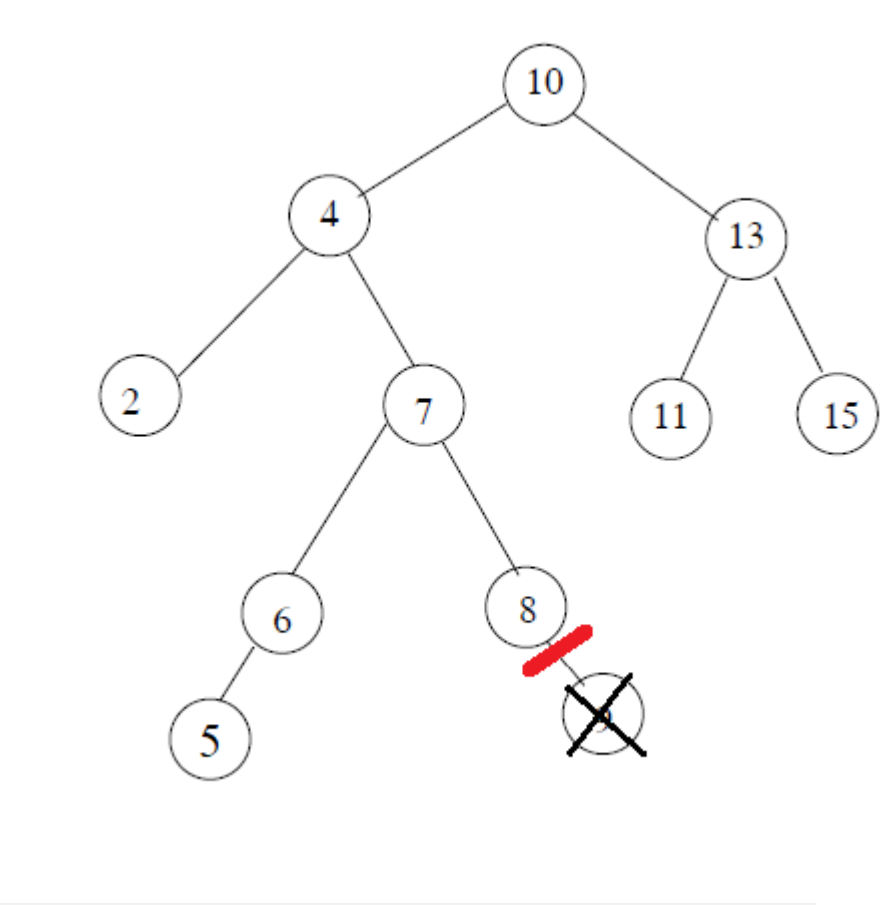
LES FONCTIONS USUELLES

Suppression d’une feuille

Supprimer une feuille d'un arbre est assez simple :

Pour supprimer le nœud 9 il suffit de mettre le fils droit du nœud 8 à NULL. Par exemple :

Exemple :

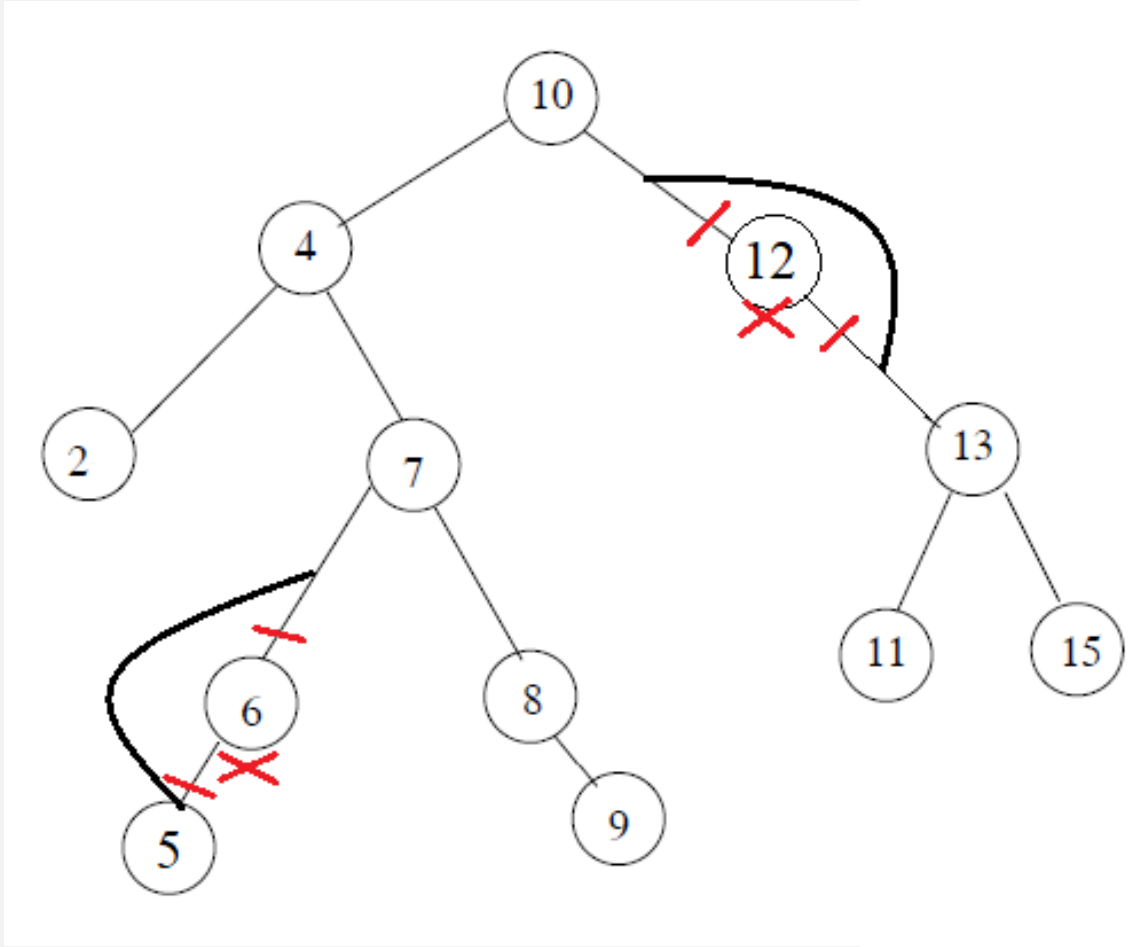


LES FONCTIONS USUELLES

Suppression d'un nœud à un seul fils

C'est simple également pour chaque nœud qui n'a qu'un seul fils. Pour supprimer le nœud 12 il suffit de faire pointer le fils droit du nœud 10 sur le nœud 13. Pour supprimer le nœud 6 il faut faire pointer les fils gauche du nœud 7 sur le nœud 6. C'est comme si nous sommes là dans la configuration de suppression d'une cellule d'une liste chainée.

Exemple :



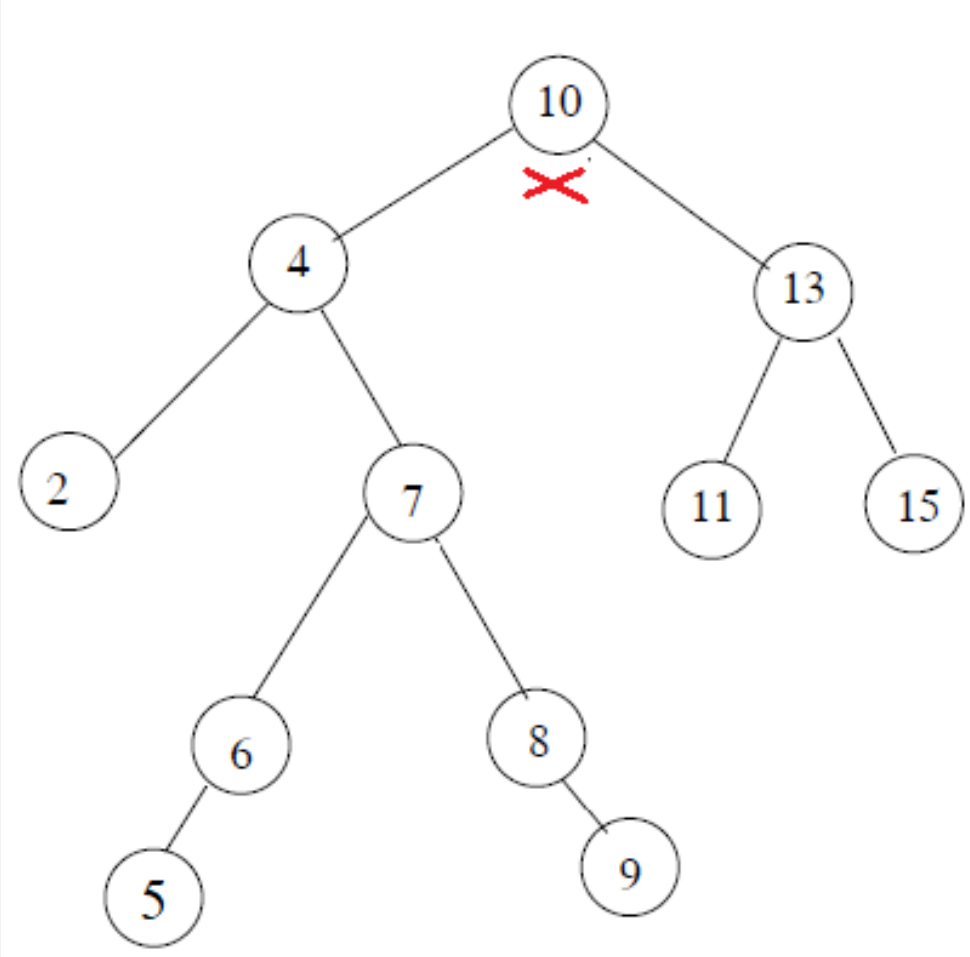
LES FONCTIONS USUELLES

Suppression d'un nœud qui a deux fils

Mais ça se complique si le nœud à supprimer a deux fils.
Par exemple, comment supprimer le nœud 10 ?

Que faire des nœud 4 et 13 ? Comment les accrocher ?

Exemple :



LES FONCTIONS USUELLES

Suppression d'un nœud qui a deux fils

La première solution consiste à chercher
le **plus grand nœud** dans le sous **arbre de gauche**

ou

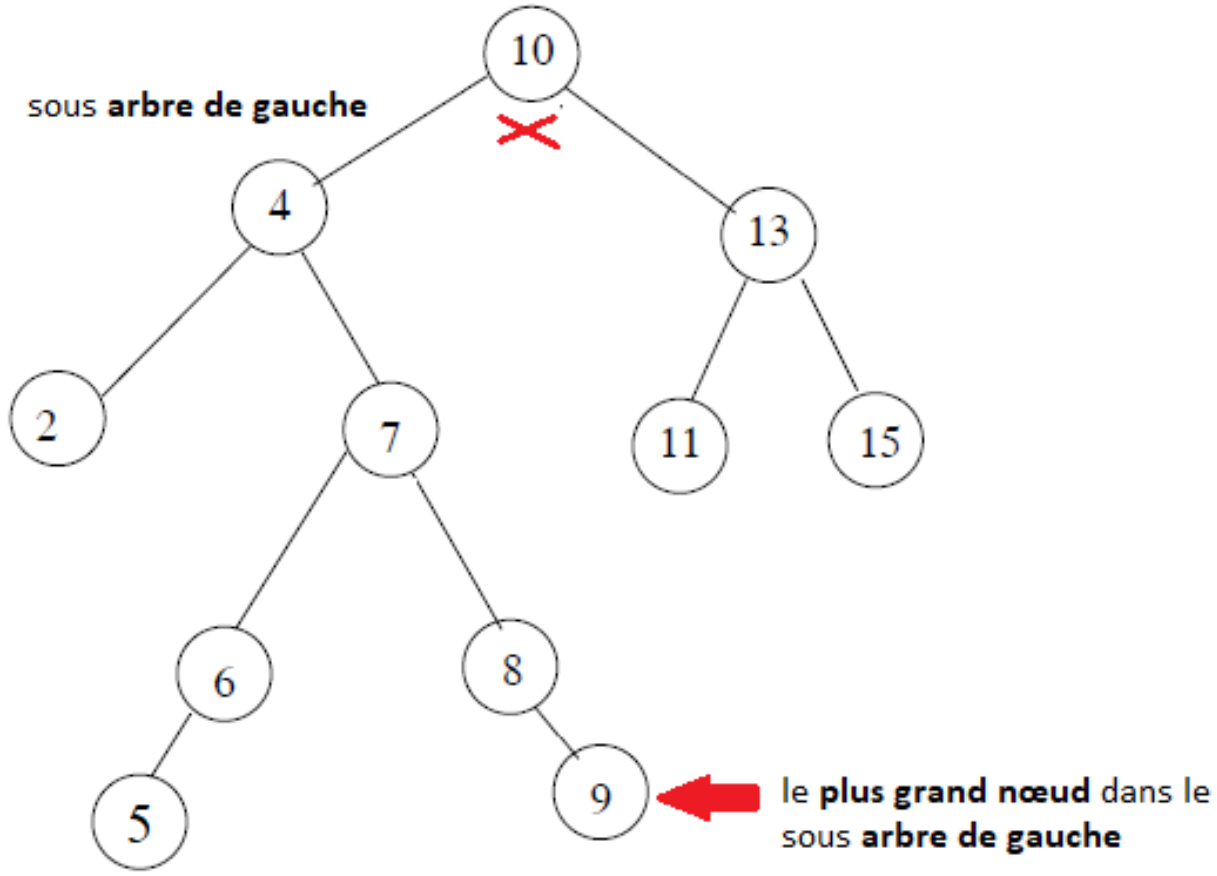
le **plus petit nœud** dans le sous **arbre de droite**
(9 ou 11 pour le nœud 10).
Ils seront nécessairement

- soit **une feuille**,
- soit un **nœud n'ayant qu'un seul fils.**

Ensuite :

- le supprimer en conservant sa valeur et
- copier cette valeur dans le nœud à supprimer

Exemple :



LES FONCTIONS USUELLES

Suppression d'un nœud qui a deux fils

La première solution consiste à chercher
le **plus grand nœud** dans le sous **arbre de gauche**

ou

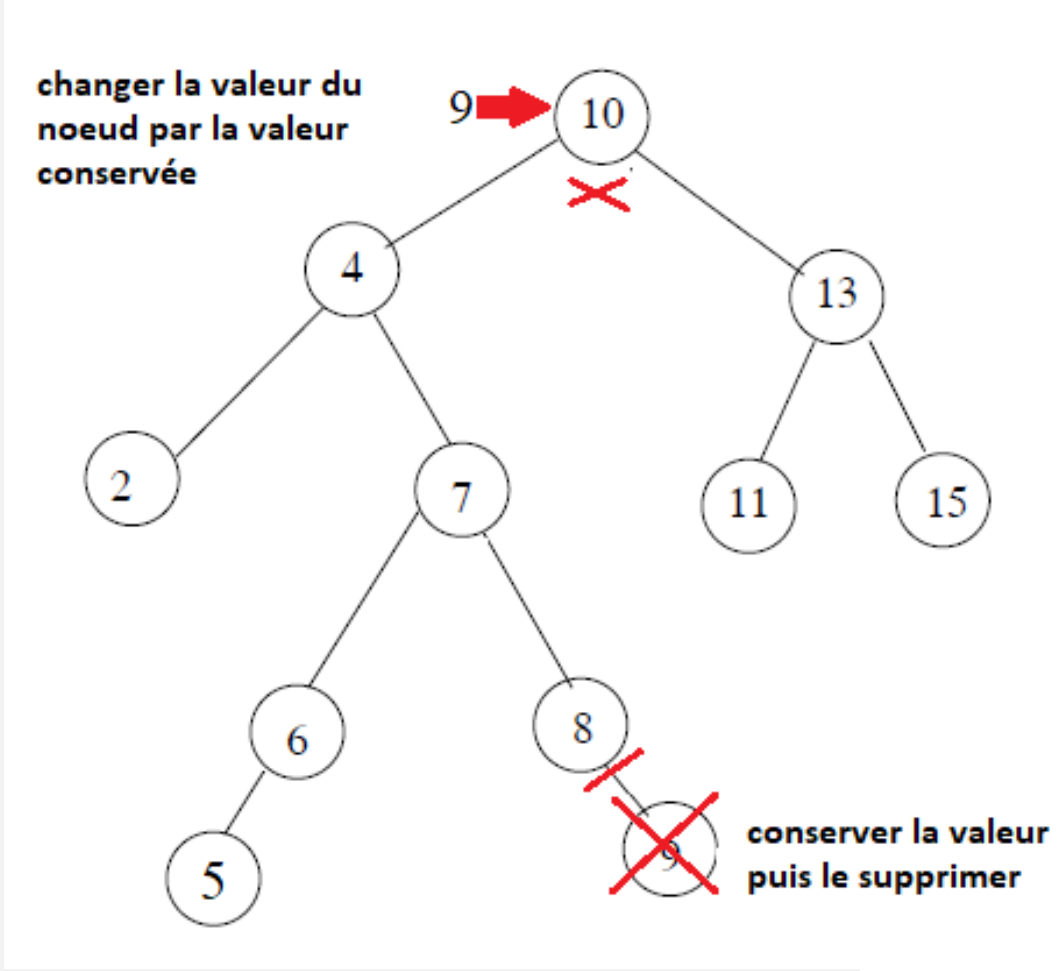
le **plus petit nœud** dans le sous **arbre de droite**
(9 ou 11 pour le nœud 10).
Ils seront nécessairement

- soit **une feuille**,
- soit un **nœud n'ayant qu'un seul fils.**

Ensuite :

- le supprimer en conservant sa valeur et
- copier cette valeur dans le nœud à supprimer

Exemple :



LES FONCTIONS USUELLES

Suppression d'un nœud qui a deux fils

La première solution consiste à chercher
le **plus grand nœud** dans le sous **arbre de gauche**

ou

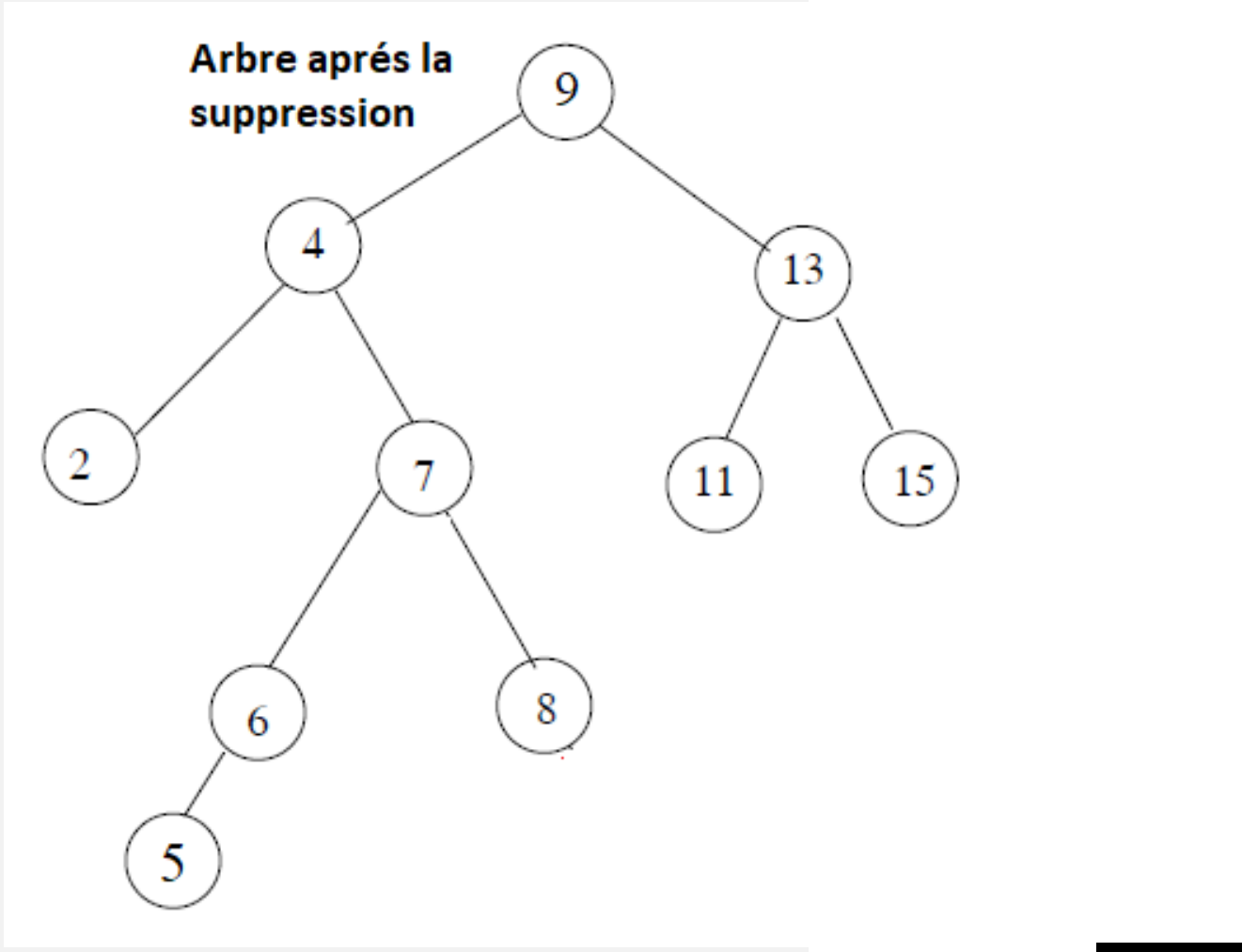
le **plus petit nœud** dans le sous **arbre de droite**
(9 ou 11 pour le nœud 10).
Ils seront nécessairement

- soit **une feuille**,
- soit un **nœud n'ayant qu'un seul fils.**

Ensuite :

- le supprimer en conservant sa valeur et
- copier cette valeur dans le nœud à supprimer

Exemple :



LES FONCTIONS USUELLES

Suppression d'un nœud qui a deux fils

La première solution consiste à chercher
le **plus grand nœud** dans le sous **arbre de gauche**

ou

le **plus petit nœud** dans le sous **arbre de droite**
(9 ou 11 pour le nœud 10).

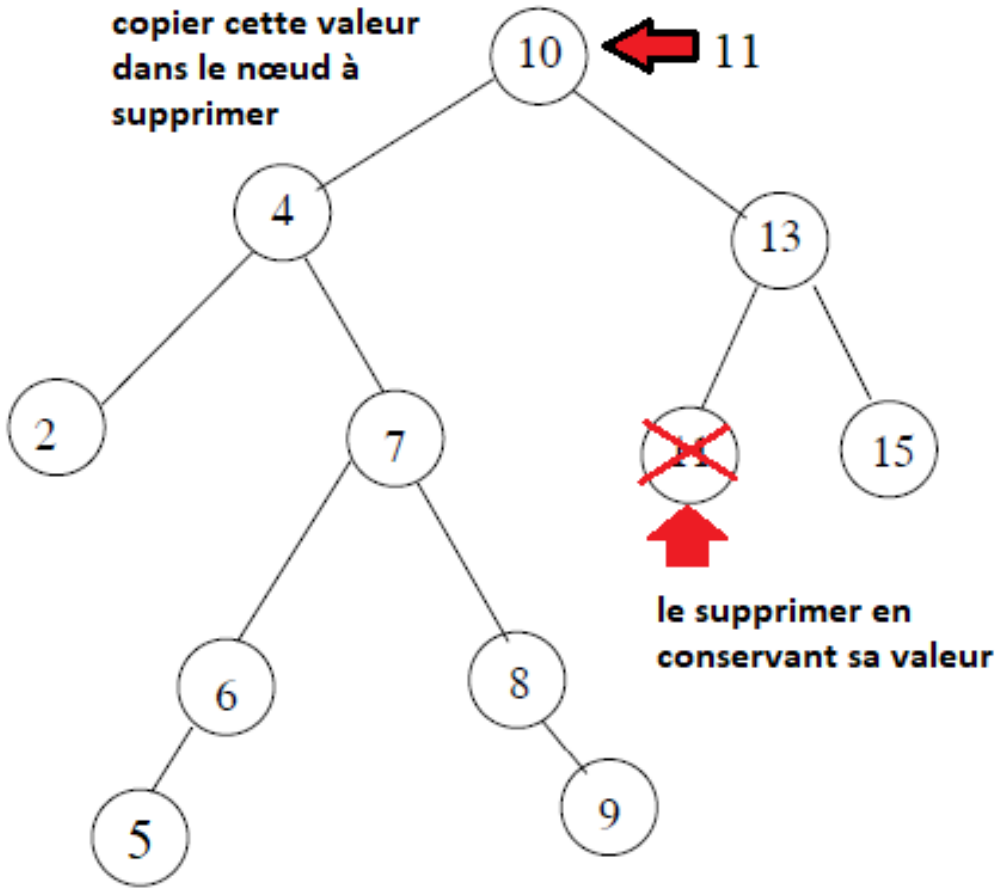
Ils seront nécessairement

- soit **une feuille**,
- soit un **nœud n'ayant qu'un seul fils.**

Ensuite :

- le supprimer en conservant sa valeur et
- copier cette valeur dans le nœud à supprimer

Exemple :



LES FONCTIONS USUELLES

Suppression d'un nœud qui a deux fils

La première solution consiste à chercher
le **plus grand nœud** dans le sous **arbre de gauche**

ou

le **plus petit nœud** dans le sous **arbre de droite**
(9 ou 11 pour le nœud 10).

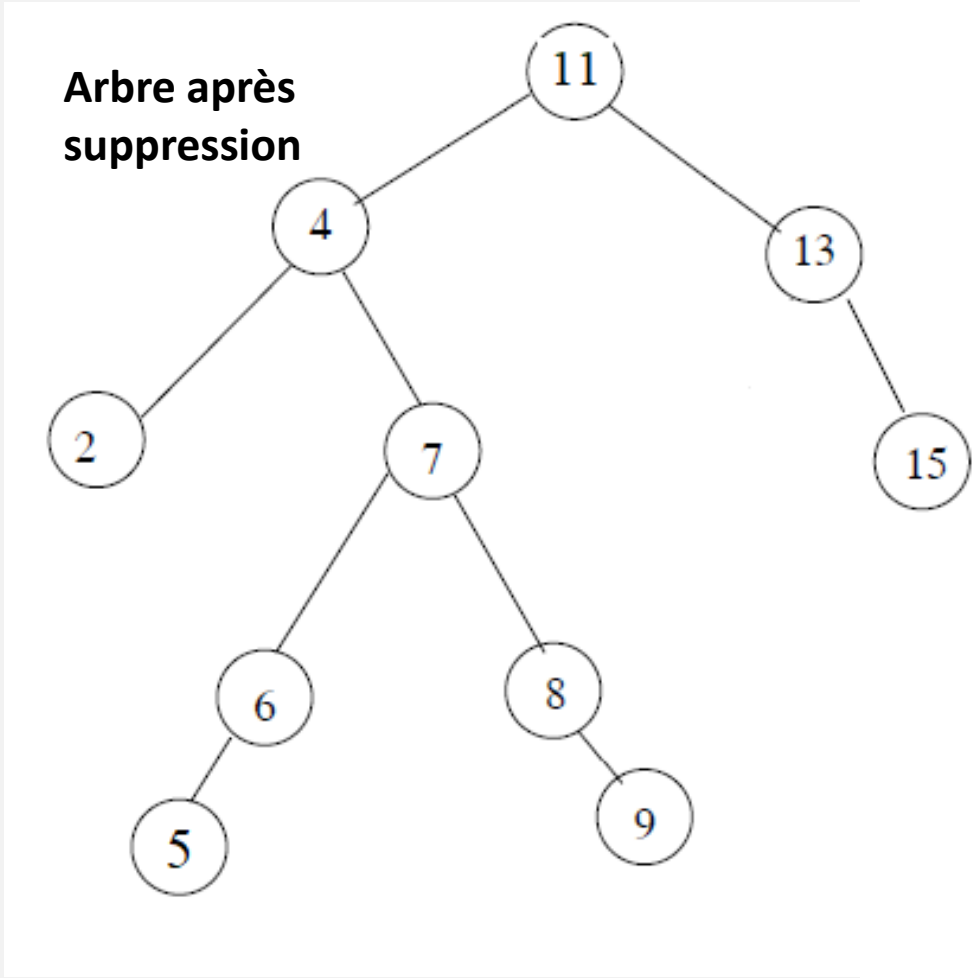
Ils seront nécessairement

- soit **une feuille**,
- soit un **nœud n'ayant qu'un seul fils.**

Ensuite :

- le supprimer en conservant sa valeur et
- copier cette valeur dans le nœud à supprimer

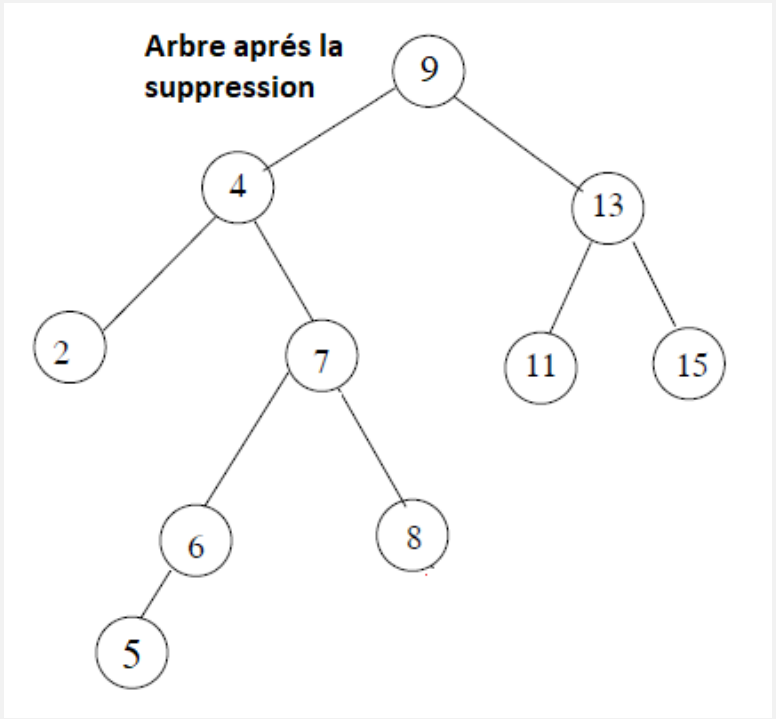
Exemple :

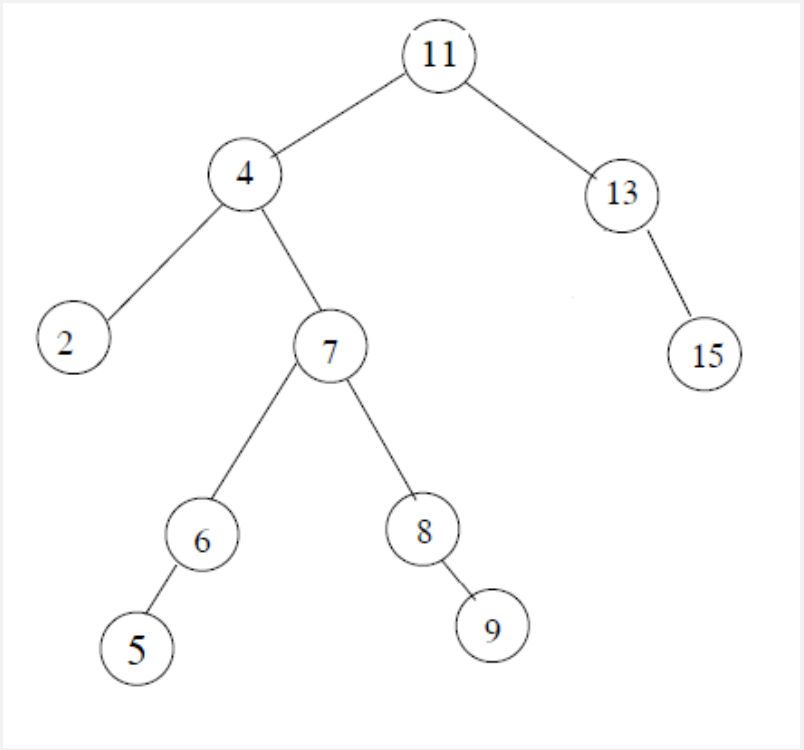


LES FONCTIONS USUELLES

Suppression d'un nœud qui a deux fils

Par les deux manières, après la suppression nous obtenons des arbres qui **respectent** toujours les conditions d'arbre binaires de recherche.





LES FONCTIONS USUELLES

Suppression d’un nœud : remarque

Ce choix d'implémentation peut contribuer à **déséquilibrer** l'arbre.

En effet, puisque ce sont toujours des feuilles du sous-arbre gauche qui sont supprimées (ou celle de droit), une utilisation fréquente de cette fonction amènera à un arbre plus **lourd** à **droite qu'à gauche**.

Ce qui se traduit par un **déséquilibre.**

On peut remédier à cela en **alternant** successivement la suppression du minimum du fils droit avec celle du maximum du fils gauche, plutôt que toujours choisir ce dernier.

Il est par exemple possible d'utiliser un **facteur aléatoire** : le programme aura une chance sur deux de choisir le fils droit et une chance sur deux de choisir le fils gauche.

ÉTUDE SUR LA COMPLEXITÉ DES OPÉRATIONS

Comparaison entre les différents structures de données

Lorsque nous avons un certain nombre **(n)** de valeurs à stocker et à manipuler par une implémentation d'une structure de données particulière. Toute manipulation se base normalement sur trois opérations fondamentales :

1. l'insertion
2. La recherche
3. La suppression

Selon la structure de données utilisées, les complexités des opérations changent d'une structure à une autre. Voici la table de complexité par rapport aux différentes structures de données déjà vu:

Implantation	Rechercher	Insérer	Supprimer
Tableau non ordonné	$O(n)$	$O(1)$	$O(n)$
Liste non ordonnée	$O(n)$	$O(1)$	$O(1)$
Tableau ordonné	$O(\log n)$	$O(n)$	$O(n)$
Liste ordonnée	$O(n)$	$O(n)$	$O(1)$
Arbre de recherche	$O(h)$	$O(h)$	$O(h)$

Avec :
n le nombre des éléments
h la hauteur de l'arbre qui représente les **n** données

ÉTUDE SUR LA COMPLEXITÉ DES OPÉRATIONS

Complexité des opérations sur les ABR

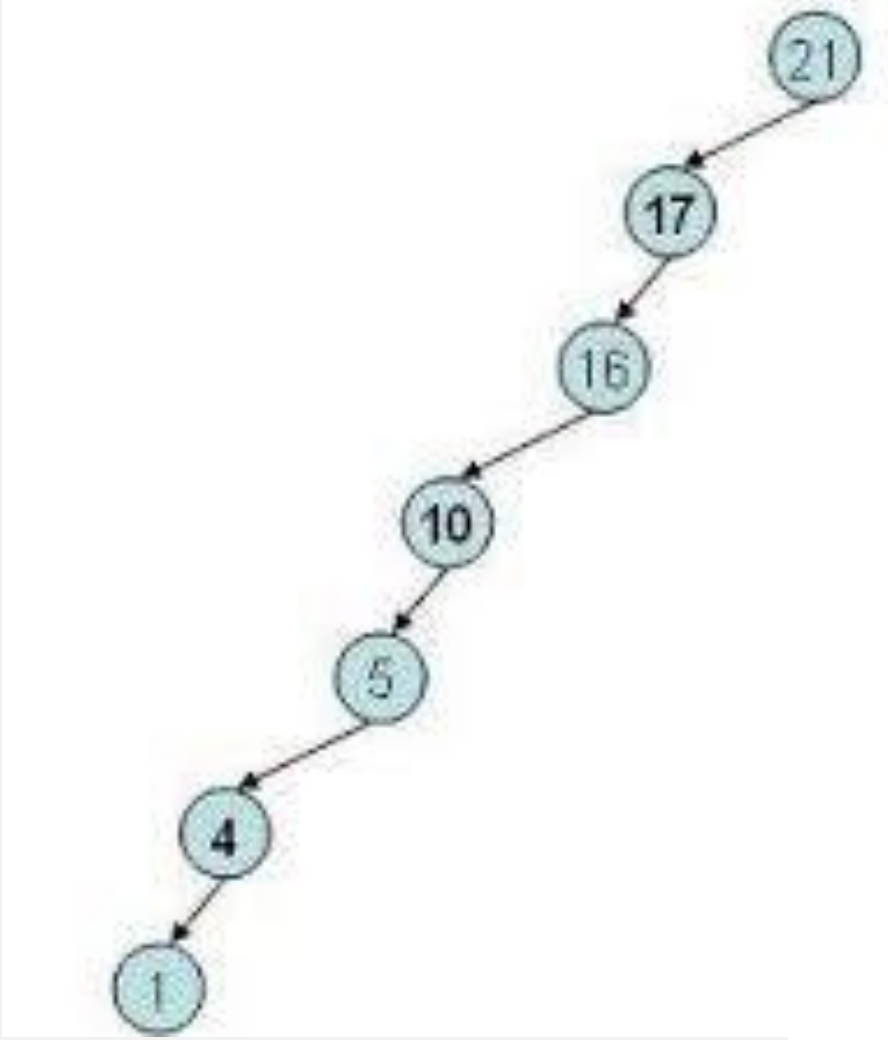
Et donc la complexité de ces opérations lorsque la structures des arbres de recherche est utilisée dépends directement de **h** : la hauteur de l'arbre.

Cependant nous avons vus que la disposition de l'arbre diffère en fonction de la séquence dont laquelle les valeurs ont été introduites. Par conséquences, on peut tomber dans certaines dispositions ou l'arbre devient totalement **inefficace** vis-à-vis des opérations fondamentales.

Nous prenons le cas où nous introduisons une séquence décroissante de nombres [21, 17, 16 ,10, 5, 4, 1]. L'arbre qui sera construit serait un **Arbre totalement déséquilibré (**appelé aussi **Arbre dégénéré).**

Dans ce cas **h = n** et par conséquence : toutes les opérations seront de complexité linéaire $O(n)$

Exemple : Arbre totalement déséquilibré



ÉTUDE SUR LA COMPLEXITÉ DES OPÉRATIONS

Complexité des opérations sur les ABR

Les performances dépendent de la façon d'entrer les informations. Si un arbre est très **déséquilibré**, ses performances sont instables.. La recherche n'est plus réellement dichotomique dans un arbre déséquilibré.

Le problème principal avec **les arbres binaires de recherche** est qu'il n'y a aucune garantie que l'arbre soit équilibré, à savoir que les hauteurs de chacun des sous-arbres soient à peu près égales. Une suite malheureuse d'insertions ou de suppressions peut provoquer un arbre ayant la forme d'une liste plutôt que d'un arbre. Dans ce cas, les opérations ne sont plus de complexité O(log n), mais O(n).

Il existe des techniques pour obtenir des arbres équilibrés, c'est-à-dire pour garantir une hauteur logarithmique en nombre d'éléments.

Le prochain chapitre est consacré à l'une de ces techniques.

Exemple : Arbre déséquilibré