Cours
Les Structures de données

SMI4

Année 2019-2020

Pr F.Omary

Plan du cours

- Introduction générale
- Types abstraits de données: TAD
- Première partie : Structures de Données Linéaires
  - Listes
  - Piles
  - Files
- Deuxième partie: Structures de Données non Linéaires
  - Arbres
  - Tables de hachage
  - Graphes

Introduction Générale

Objectif Principal:

Méthodologie de construction de programmes par abstraction

Introduction générale(suite)

- **La programmation modulaire est très conseillée:**

- **Elle consiste en la constitution de parties « sous-programmes », indépendantes les unes des autres**

- **Ces sous programmes (ou modules) pouvant être réutilisés même dans d'autres programmes.**

- **lorsqu'on utilise un module, peu importe pour l'utilisateur la façon dont les opérations sont programmées. Mais il importe de connaître les opérations que l'on peut faire sur les données.**

Introduction générale (suite)

- Exemple: en langage C on connaît bien la fonction « scanf » mais on ne sait pas comment elle est implémentée.
- **Mais** :Un module n'est rien si l'on dispose pas de structures de données appropriée pour stocker ses données.

Plus précisément:

Algorithme + Structures de donnés= Programme

Types Abstraits de Données (TADs)

Spécification & Implémentation

6

Terminologie

- **Spécification**: Définition formelle du comportement d'une structure de données
  - Dit ce que doit faire la structure de données
  - Ne dit pas comment faire (choix de l'implémentation)
  - Précis et rigoureux
  - Doit éviter de poser des contraintes d'implémentation.

Motivations

- Par analogie avec les types primitifs tels que:
  - Le type int : représente un entier.
  - IL est fourni avec des opérations : + - / * %.

  Il n'est pas nécessaire de connaître la représentation interne ou les algorithmes de ces opérations pour les utiliser.
- En faire de même avec des types plus complexes et indépendamment du langage de programmation.
- Mettre en place un type dont la représentation interne est cachée
- Définir les opérations nécessaires pour manipuler les données

Motivations(suite)

Autrement dit:

- La conception d'un algorithme est indépendante de toute implantation

- **La représentation des données n'est pas fixée ; *celles-ci sont considérées de manière abstraite***

- On s'intéresse à l'ensemble des opérations sur les données, et aux propriétés des opérations, ***sans dire comment ces opérations sont réalisées***

Définition d'un TAD

- **Définition: Un TAD (*Data Abstract Type*) est un** ensemble de valeurs muni d'opérations sur ces valeurs, sans faire référence à une implémentation particulière.
- **Exemples :**
  - Dans un algorithme qui manipule des entiers, on s'intéresse, non pas à la représentation des entiers, mais aux opérations définies sur les entiers : +, -, *, /
  - Type booléen, ensemble de deux valeurs (faux, vrai) muni des opérations : non, et, ou
- **Un TAD est caractérisé par :**
  - ***sa signature : définit la syntaxe du type et des opérations ;***
  - ***sa sémantique : définit les propriétés des opérations.***

Signature d'un TAD

- **Comporte :**
  - *Le nom du TAD ;*
  - *Les noms des types des objets utilisés par le TAD ;*
  - *Pour chaque opération, l'énoncé des types des objets qu'elle reçoit et qu'elle renvoie.*
- **Décrite par les paragraphes :**
  - `Type`
  - `Utilise`
  - `Opérations`

Signature d'un TAD
Exemple : TAD Booléen

Type **Booléen**

Opérations

vrai : → Booléen

faux : → Booléen

non : Booléen → Booléen

et : Booléen x Booléen → Booléen

ou : Booléen x Booléen → Booléen

Nom du TAD

Nom de l'opération

Deux arguments de type Booléen

Type valeur de retour

Sémantique d'un TAD

- **Précise :**
  - *Les domaines de définition (ou d'application) des opérations ;*
  - *Les propriétés des opérations.*

- **Décrite par les paragraphes :**
  - `Préconditions`
  - `Axiomes`

Exemple 1 de TAD (TAD Booléen)

14

```
Type Booléen
Opérations
    vrai    : → Booléen
    faux    : → Booléen
    non     : Booléen → Booléen
    et  : Booléen × Booléen → Booléen
    ou  : Booléen × Booléen → Booléen
Préconditions

Axiomes
   Soit, a, b : Booléen
   non(vrai) = faux
   non(non(a)) = a
   vrai et a = a
   faux et a = faux
   a ou b = non(non(a) et non(b))
```

Aucune précondition

Exemple 2 de TAD (TAD Vecteur)

Type Vecteur

Utilise Entier, Elément

Opérations

vect : Entier → Vecteur

changer_ième : Vecteur x Entier x Elément → Vecteur

ième : Vecteur x Entier → Elément

taille : Vecteur → Entier

Préconditions

vect(i) *est_défini_ssi* i ≥ 0

ième(v,i) *est_défini_ssi* 0 ≤ i < taille(v)

changer_ième(v,i,e) *est_défini_ssi* 0 ≤ i< taille(v)

Axiomes

Soit, i, j : Entier, e : Elément, v : Vecteur

si 0 ≤ i < taille(v) *alors* ième(changer_ième(v,i,e),i) = e

si 0 ≤ i < taille(v) *et* 0 ≤ j < taille(v) *et* i ≠ j

alors ième(changer_ième(v,i,e),j) = ième(v,j)

taille(vect(i)) = i

taille(changer_ième(v,i,e)) = taille(v)

Opérations

- **Trois catégories d'opérations (ou de primitives)**
  - ***De Constructions :*** **type spécifié apparaît, *uniquement*, comme résultat ;**
  - ***D'Observations :*** **type spécifié apparaît, *uniquement*, comme argument ;**
  - ***De Transformations :*** **type spécifié apparaît, *à la fois*, comme argument et comme résultat ;**
- **Constante : opérateur *sans argument***

Opérations Partielles

- Une opération peut ne pas être définie partout
- Cela dépend de son domaine de définition
- Ceci est traité dans le paragraphe **`Préconditions`**
- **Exemple :**
  - Opérations `ième` et `changer_ième` du TAD `Vecteur`

Réutilisation des TADs

- Quand on définit un type, on peut réutiliser des types déjà définis
- La signature du type défini est l'union des signatures des types utilisés enrichie des nouvelles opérations
- Le type hérite des propriétés des types qui le constituent
- **Exemples :**
  - Types `Entier` et `Elément` utilisés par le TAD `Vecteur`

Choix des Axiomes

- **Le système d'axiomes doit être :**
  - ***non contradictoire (consistance)***
  - ***complet (complétude suffisante)***

Notion de Structure de Données

- On dit aussi ***structure de données concrète***
- Correspond à **l'implémentation d'un TAD**
- Composée d'un algorithme pour chaque opération, plus éventuellement des données spécifiques à la structure pour sa gestion
- Un même TAD peut donner lieu à plusieurs structures de données, avec des performances différentes

Implémentation d'un TAD

- **Pour implémenter un TAD :**
  - Déclarer la structure de données retenue pour représenter le TAD : ***L'interface***
  - Définir les opérations primitives dans un langage particulier : ***La réalisation***

- **Exigences :**
  - Conforme à la spécification du TAD ;
  - Efficace en terme de complexité d'algorithme.

- **Pour implémenter, on utilise :**
  - Les types élémentaires ou de base (entiers, caractères, ...)
  - Les pointeurs ;
  - Les tableaux et les enregistrements ;
  - Les types prédéfinis.

- Plusieurs implémentations possibles pour un même TAD

Implémentation d'un TAD en C

- **Utiliser la programmation modulaire (*voir cours Programmation*) :**
  - Programme découpé en plusieurs fichiers, même de petites tailles (*réutilisabilité, lisibilité, etc.*)
  - Chaque composante logique (*un module*) regroupe les fonctions et types autour d'un même thème.

- **Pour chaque module *truc*, créer deux fichiers :**
  - **fichier *truc.h* : *l'interface*** (*la partie publique*) ; contient la ***spécification*** de la structure ;
  - **fichier *truc.c* : *la définition*** (*la partie privée*) ; contient ***la réalisation*** des opérations fournies par la structure. Il contient au début l'inclusion du fichier *truc.h*

- **Tout module ou programme principal qui a besoin d'utiliser les fonctions du module *truc*, devra juste inclure le *truc.h***

- **Un module C implémente un TAD :**
  - ***L'encapsulation :*** détails d'implémentation cachés ; l'interface est la partie visible à un utilisateur
  - ***La réutilisation :*** placer les deux fichiers du module dans le répertoire où l'on développe l'application.

Structures de Données Linéaires

listes, Piles & Files

23

Classification

Classification des structures de données

- Une structure de données linéaire est une structure dans laquelle les éléments (ou données) sont reliés séquentiellement.

- Une structure de données non linéaires permettent de relier un élément à plusieurs autres éléments.

Structures de Données Linèaires

- **Étude des structures de données linéaires : listes, piles et files**
- **Une structure linéaire est un arrangement linéaire d'éléments liés par la relation successeur**
  - **Exemple :** ***Un tableau*** (*la relation successeur est implicite*).
- **Pour chaque structure, on présente :**
  - ***une définition abstraite ;***
  - ***les différentes représentations en mémoire ;***
  - ***une implémentation en langage C ;***
  - ***quelques applications.***

Les Listes

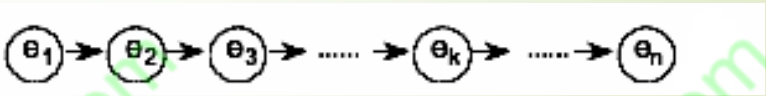
26

Notion de Liste (List) (1)

- **Généralisation des piles et des files**
  - Structure linéaire dans laquelle les éléments peuvent être traités les uns à la suite des autres
  - Ajout ou retrait d'éléments n'importe où dans la liste
  - Accès à n'importe quel élément
- **Une liste est une suite finie, *éventuellement vide*, d'éléments de *même type* repérés par leur *rang* dans la liste**
- **Chaque élément de la liste est rangé à une certaine *place***
- **Exemple :**
  - une liste de 5 entiers L = <4, 1, 7, 3, 1> (*place de rang 1 contient la valeur 4*)
  - une liste vide L2 = <>

Notion de Liste (List) (2)

- Les éléments d'une liste sont donc *ordonnés* en fonction de leur place



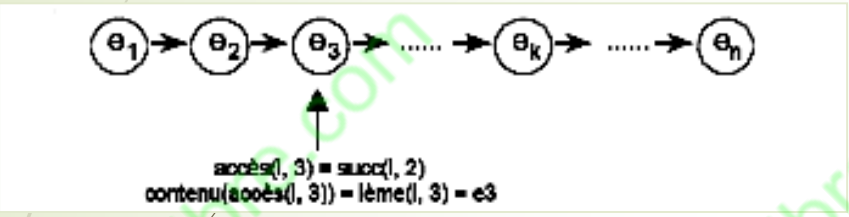
- *On définit une fonction* notée ***succ*** qui, appliquée à toute place sauf la dernière, fournit la place suivante

- *Le nombre total d'éléments*, et par conséquent de places, est appelé ***longueur de la liste***

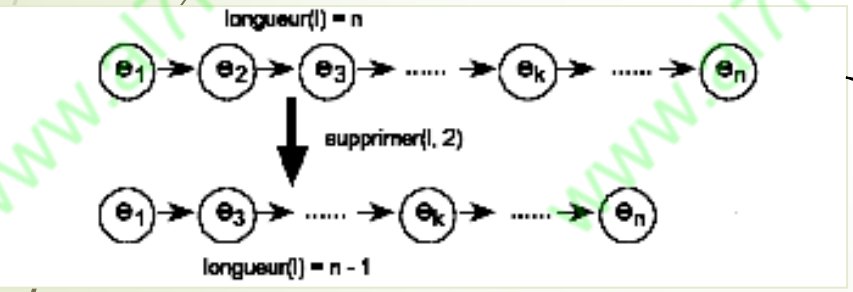
- ***Une liste vide*** est d'une longueur égale 0

Exemples de Liste

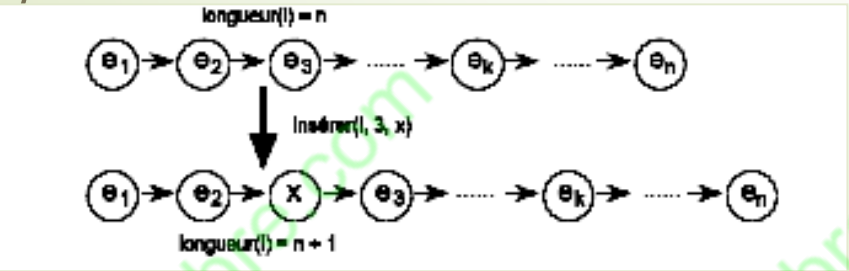
Liste vide



Accès à l'élément de rang 3 dans une liste à n éléments



Suppression de l'élément au rang 2
→ longueur(liste) =n-1



Ajout de l'élément x au rang 3
→ longueur(liste) =n+1

Type Abstrait Liste (1)

Type Liste
Utilise Elément, Booléen, Place
Opérations

- **liste_vide : → Liste**
- **longueur : Liste → Entier**
- **insérer : Liste x Entier x Elément → Liste**
- **supprimer : Liste x Entier → Liste**
- **kème : Liste x Entier → Elément**
- **accès : Liste x Entier → Place**
- **contenu : Liste x Place → Elément**
- **succ : Liste x Place → Place**

Préconditions

- **insérer(l,k,e)** ***est-défini-ssi*** **1 ≤ k ≤ longueur(l)+1**
- **supprimer(l,k)** ***est-défini-ssi*** **1 ≤ k ≤ longueur(l)**
- **kème(l,k)** ***est-défini-ssi*** **1 ≤ k ≤ longueur(l)**
- **accès(l,k)** ***est-défini-ssi*** **1 ≤ k ≤ longueur(l)**
- **succ(l,p)** ***est-défini-ssi*** **p ≠ accès(l,longueur(l))**

`k = longueur(l) + 1` signifie ajout en fin de liste

Type Abstrait Liste (2)

Axiomes

Soit, e : Elément, l, l' : Liste, k, j : Entier

si l = liste_vide alors longueur(l) = 0
sinon si l = insérer(l',k,e) alors longueur(l)=longueur(l')+1
sinon soit l = supprimer(l',k)alors longueur(l)=longueur(l')-1

si 1 ≤ j < k alors kème(insérer(l,k,e),j) = kème(l,j)
sinon si j = k alors kème(insérer(l,k,e),j) = e
sinon kème(insérer(l,k,e),j) = kème(l,j-1)
si 1 ≤ j < k alors kème(supprimer(l,k),j) = kème(l,j)
sinon kème(supprimer(l,k),j) = kème(l,j+1)

succ(l,accès(l,k)) = accès(l,k+1)
contenu(l,accès(l,k)) = kème(l,k)

si 1 ≤ k < j ≤ longueur(l) alors
  contenu(l,accès(supprimer(l,j),k)) = contenu(l,accès(l,k))
si 1 ≤ j ≤ k ≤ longueur(l) alors
  contenu(l,accès(supprimer(l,j),k) = contenu(l,accès(l,k+1))
si 1 ≤ j < k ≤ 1+longueur(l) alors
  contenu(l,accès(insérer(l,k,e),j) = contenu(l,accès(l,j))
si 1 ≤ k = j ≤ 1+longueur(l) alors
  contenu(l,accès(insérer(l,k,e),j) = e
si 1 ≤ k < j ≤ 1+longueur(l) alors
  contenu(l,accès(insérer(l,k,e),j) = contenu(l,accès(l,j-1))

Extension Type Abstrait Liste

Extension Type Liste

Opérations

concaténer : Liste x Liste → Liste

est_présent : Liste x Elément → Booléen

Préconditions

Axiomes

Soit, e : Element, l, l' : Liste, k, j : Entier

longueur(concaténer(l,l')) = longueur(l) + longueur(l')

si k ≤ longueur(l)
  alors kème(concaténer(l,l'),k)= kème(l,k)
sinon kème(concaténer(l,l'),k)= kème(l',k-longueur(l))

si longueur(l) = 0 alors est_présent(l,e) = faux
sinon si e = kème(l,1) alors est_présent(l,e) = vrai
      sinon est_présent(supprimer(l,1),e)= est_présent(l,e)

Opérations sur une Liste (1)

- **liste_vide : → Liste**
  - Opération d'initialisation ; la liste créée est vide
- **longueur : Liste → Entier**
  - Retourne le nombre d'éléments dans la liste
- **insérer : Liste x Entier x Elément : → Liste**
  - *insérer(L,j,e)* : liste obtenue à partir de *L* en remplaçant la place de rang *j* par une place contenant *e*, sans modifier places précédentes et en décalant places suivantes
- **supprimer : Liste x Entier : → Liste**
  - *supprimer(L,j)* : liste obtenue à partir de *L* en supprimant la place de rang *j* et son contenu, sans modifier places précédentes et en décalant places suivantes

Opérations sur une Liste (2)

- **kème : Liste x Entier → Elément**
  - Fournit l'élément de rang donné dans une liste
- **accès : Liste x Entier → Place**
  - Connaître la place de rang donné : *accès(L,k)* est la place de rang *k* dans la liste *L*
- **contenu : Liste x Place → Elément**
  - Connaître l'élément d'une place donnée. *contenu(L,p) = e* : dans la liste *L*, la place *p* contient l'élément *e*
- **succ : Liste x Place → Place**
  - Passer de place en place. *succ(L,p) = p'* : dans la liste *L*, la place qui succède à la place *p* est la place *p'*. *Opération indéfinie si place en entrée est la dernière place de la liste*

Opérations Auxiliaires sur une Liste

- **`concaténer : Liste x Liste → Liste`**
  - Accroche la deuxième liste en entrée à la fin de la première liste
- **`est_présent : Liste x Elément → Booléen`**
  - Teste si un élément figure dans une liste

Représentation Contiguë d'une Liste

- **Les éléments sont rangés les uns à côté des autres dans un tableau**
  - ***La ième case du tableau contient le ième élément de la liste***
  - ***Le rang est donc égal à la place ; ce sont des entiers***

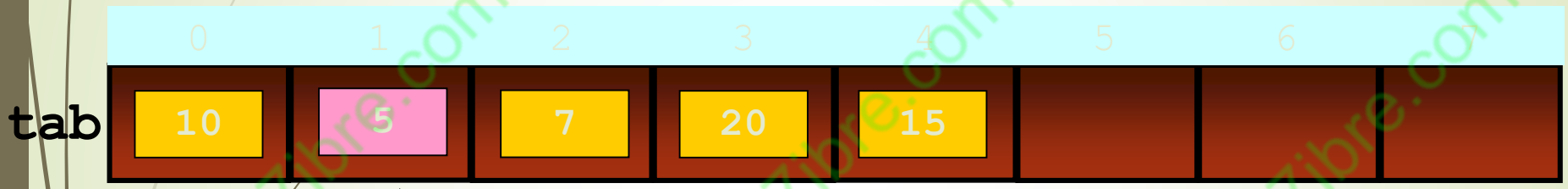
- **La liste est représentée par une structure(ou enregistrement):**
  - ***Un tableau représente les éléments***
  - ***Un entier représente le nombre d'éléments dans la liste***

Note :La longueur maximale, `MAX_LISTE`, de la liste doit être connue

Ajout dans une Liste Contiguë Exemple (1)

Liste avant insertion de la valeur 5

Ajout dans une Liste Contiguë Exemple (2)

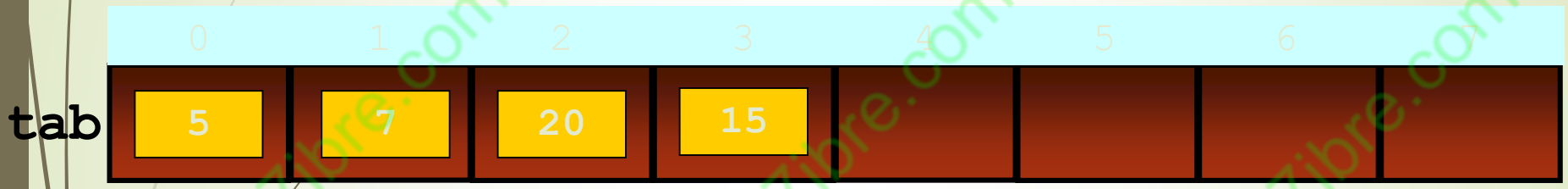


Valeur ajoutée au deuxième rang

Liste après insertion de la valeur 5

Suppression dans une Liste Contiguë Exemple (1)

Suppression dans une Liste Contiguë Exemple (2)



Liste après suppression de la valeur 10

Représentation contigüe d'une Liste

- **Rappel sur la notion** :**Enregistrement**

L'enregistrement est l'outil principal de construction de structures de données complexes.

Il permet de regrouper dans une structure l'ensemble des caractéristiques associées à une entité.

- Exemple: si un client est caractérisé par un *nom, une adresse et un code postale* alors la notion d'enregistrement permettra de regrouper dans une seule structure l'ensemble de ces caractéristiques, pourtant de natures différentes
- **Autrement dit**: une enregistrement permet de regrouper des entités hétérogènes mais liés logiquement les unes des autres.

Représentation contigüe d'une Liste

- Définition1:un enregistrement est un ensemble d'éléments de types différents repérés par un nom. Ses éléments sont appelés des champs
- Définition2: un enregistrement (appelé aussi structure dans certains langages) est un type complexe construit à partir de types plus simples.
- IL existe trois catégories d'enregistrement (ou structures):

  Structure anonyme: elle n'est pas réutilisable puisqu'elle ne possède pas de nom:

Exemple:

```
Struct {
        float  re;
        float   im;
        } C1, C2.
```

Représentation contigüe d'une Liste

- L'accès aux champs s'effectue ainsi:

  C1.re=4.5

  C1.im=6

- Une autre utilisation de ce type d'enregistrement suppose sa redéfinition.
- Donc pour pouvoir la réutiliser, il faut la munir d'un nom.
- Structure semi-nommée

```
Struct Complexe {
        float   re;
        float   im;
      } C1, C2
```

Représentation contigüe d'une Liste

- Structure nommée:
- le type *Complexe* peut être construit et nommé ainsi:

```
typedef   struct {
        float  re;
        float  im;
    } Complexe;
```

S'il est nécessaire de déclarer cela se fera comme suit:

```
Complexe   C1, C2;
```

- Composition d'enregistrement:
- Exemple:
- //définition du type Adresse

Représentation contigüe d'une Liste

- //définition du type Adresse
- Typedef struct {
  int numero;
  char nomRue[50]
  char codePostal[5]
  char ville[20]
  } *Adresse*
- // définition du type client
- Typedef struct {
  char nom[15]
  char prenom[15]
  Adresse adresse;
  } *Client*

Représentation contigüe d'une Liste

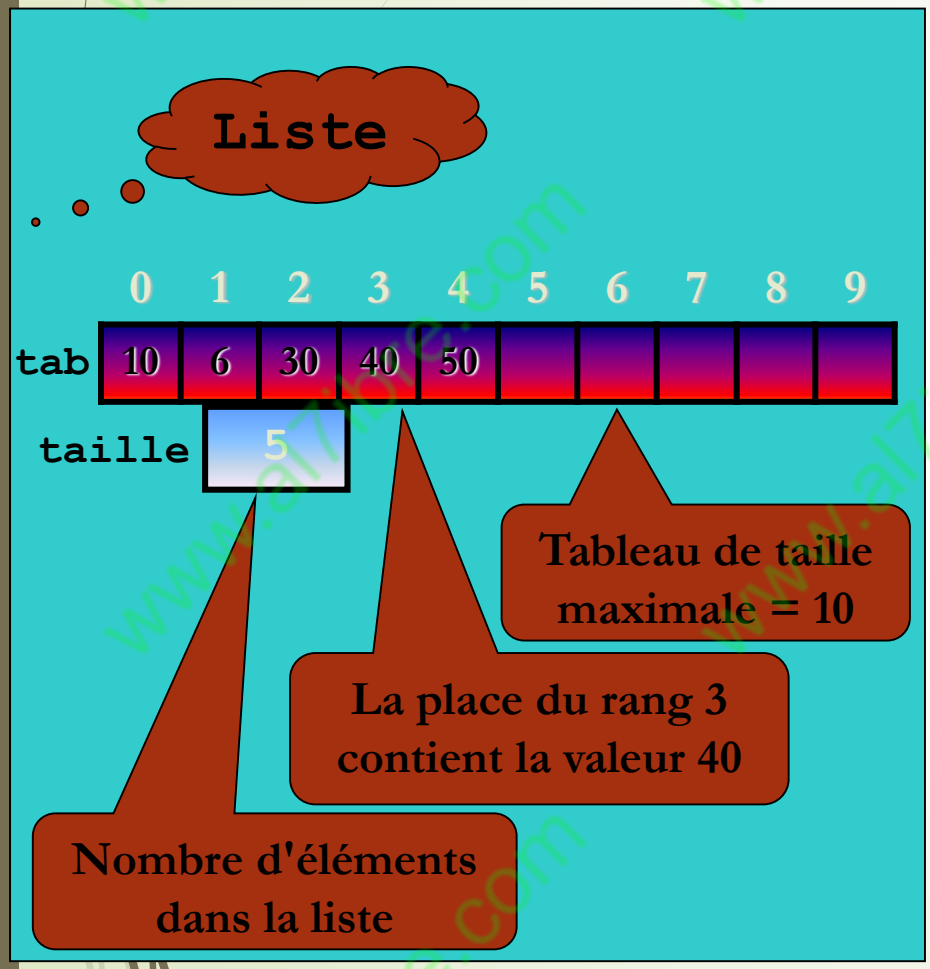
- L'accès aux champs de ces structures imbriquées peut nécessiter plusieurs occurrences de l'opérateur point(.)
- Exemple:

```
//déclaration de trois variables
   Client  A, B, C;
```

L'affectation d'un numéro de rue dans le champ numéro du champ adresse d'un client 'A' se fera comme suit:

```
A.adresse.numero=105 ;
```

Liste Contiguë (Contiguous List)




```
 /* Liste contiguë en C */

// taille maximale liste
#define MAX_LISTE 10

// type des éléments
typedef int Element;

// type Place
typedef int Place;

// type Liste
typedef struct {
 Element tab[MAX_LISTE];
 int taille;
} Liste;
```

Spécification d'une Liste Contiguë


```
                                    /* fichier "TListe.h" */
#ifndef _LISTE_TABLEAU
#define _LISTE_TABLEAU

// Définition du type liste (implémentée par tableau)
#define MAX_LISTE 100 /* taille maximale de la liste */
typedef int element;  /* les éléments sont des int */

typedef int Place;  /* la place = le rang (un entier) */

typedef struct {
 element tab[MAX_LISTE];  /* les éléments de la liste */
 int taille;   /* nombre d'éléments dans la liste */
} Liste;

// Déclaration des fonctions gérant la liste
Liste liste_vide (void);
int longueur (Liste l);
Liste inserer (Liste l, int i, element e);
Liste supprimer (Liste l, int i);
element keme (Liste l, int k);
Place acces (Liste l, int i);
element contenu (Liste l, Place i);
Place succ (Liste l, Place i);
#endif
```

type Liste : une structure à deux champs

Algorithme : option1:Modélisation contigüe statique

- Constante Max_Liste=1000

  Type Liste=Structure

  Tab[Max_Liste]: Element

  Taille: entier

  Fin strucure
- /*insertion d'un élément dans une liste L*/
- Fonction insérer (Var L:Liste, rang: entier, e:Element): entier
- Var i : entier

Algorithme : option1:Modélisation contigüe statique

Début

SI rang <1 ou rang>(L.taille+1) ou L.taille=Max_liste alors

Erreur

Finsi

Pour i depuis L.taille JSQ rang faire Pas-1

L.tab[i] ← L.tab[i-1]

Finpour // Cette boucle décale les élts à droite de la rang

L. tab[rang-1] ← e // insertion de l'élèment e à sa place

L.taille ← L.taille+1

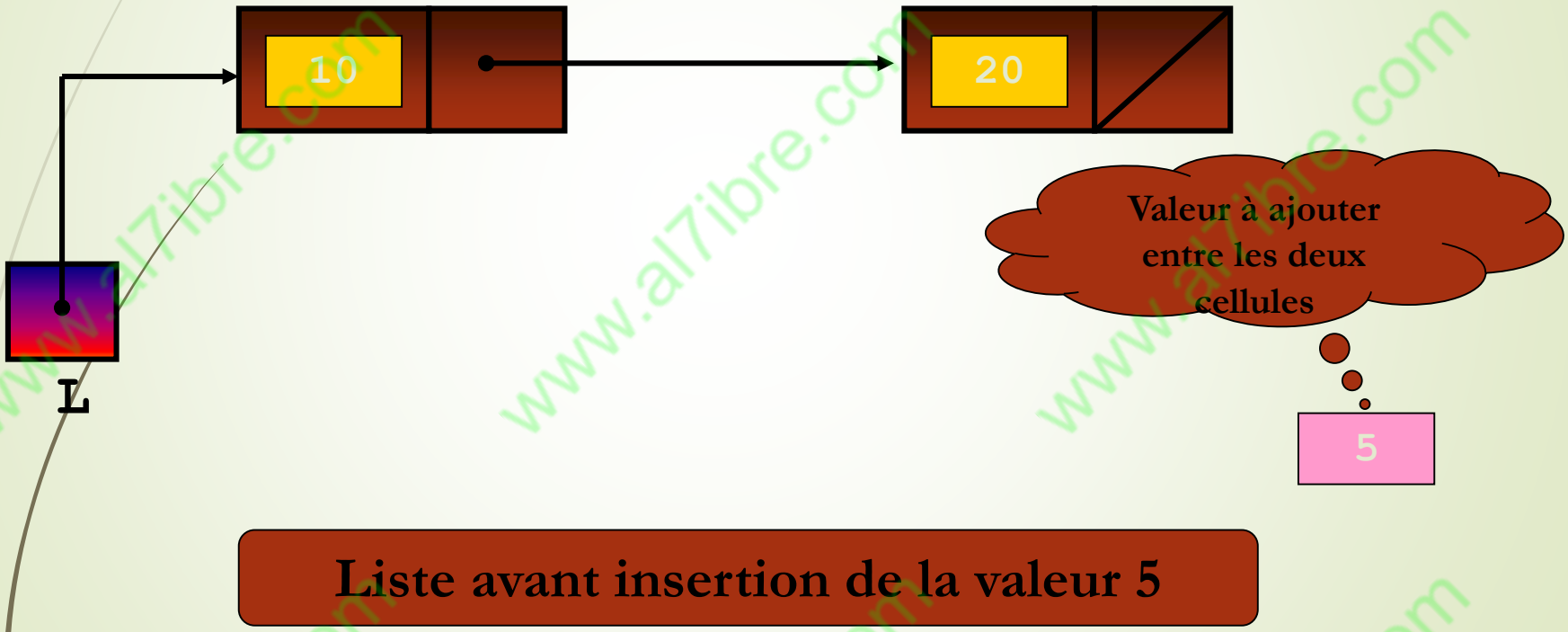
Retourne (L)

FIN

Représentation Chaînée d'une Liste

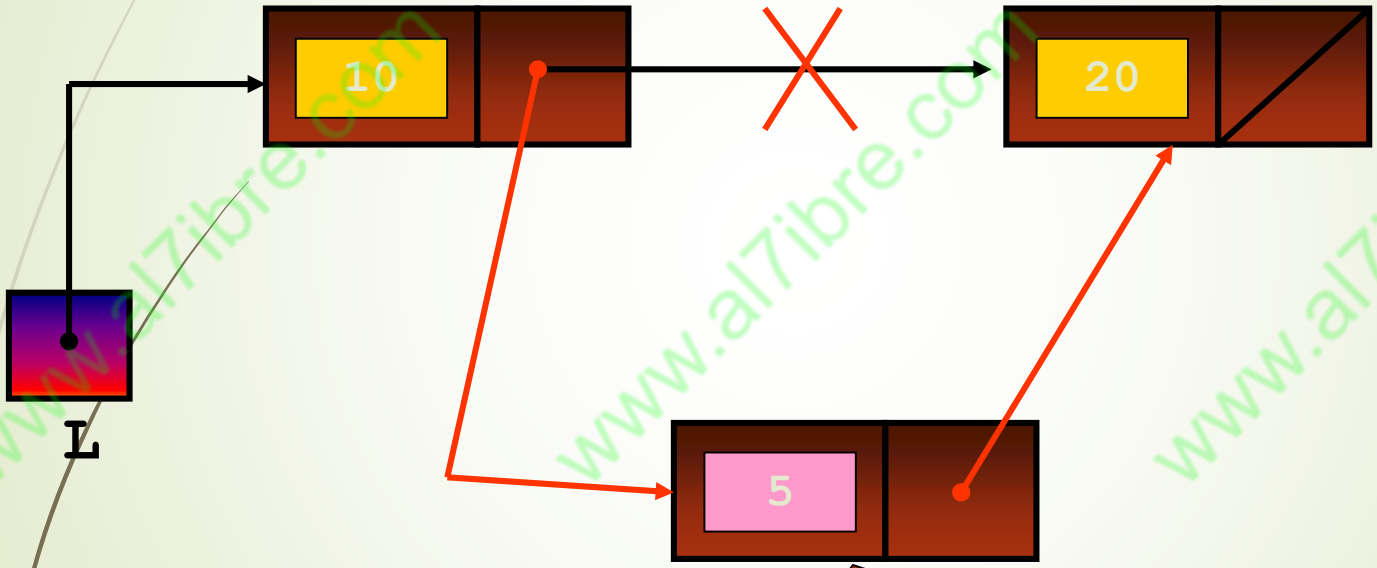
- **Les éléments ne sont pas rangés les uns à côté des autres**
  - ***La place d'un élément est l'adresse d'une structure qui contient l'élément ainsi que la place de l'élément suivant***
  - ***Utilisation de pointeurs pour chaîner entre eux les éléments successifs***
- **La liste est représentée par un pointeur sur une structure en langage C**
  - ***Une structure contient un élément de la liste et un pointeur sur l'élément suivant***
  - ***La liste est déterminée par un pointeur sur son premier élément***
  - ***La liste vide est représentée par la constante prédéfinie NULL***

Ajout dans une Liste Chaînée Exemple 1



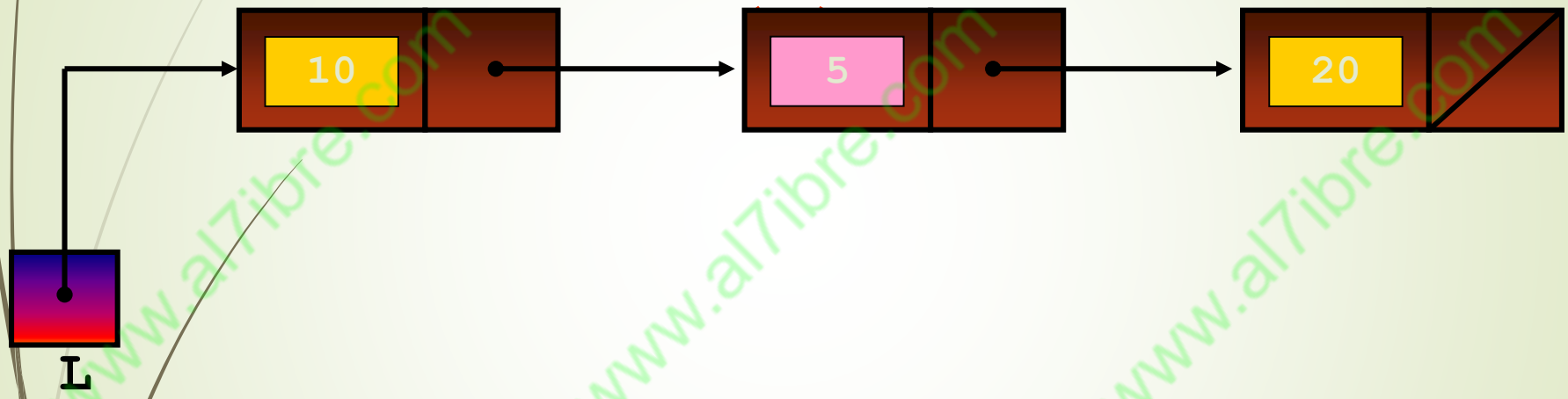
Liste avant insertion de la valeur 5

Ajout dans une Liste Chaînée Exemple (2)



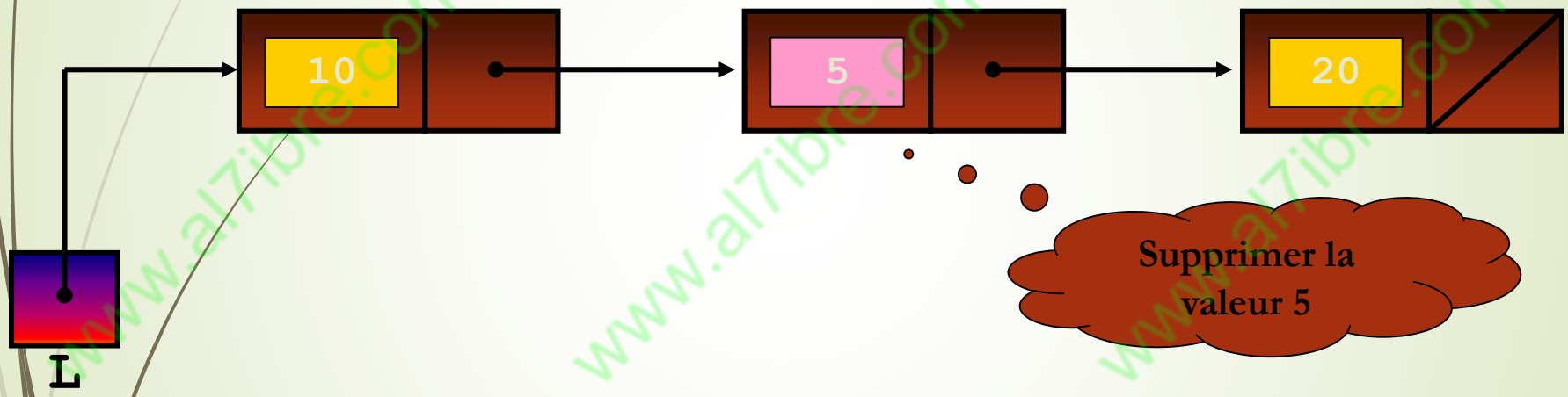
Création d'une nouvelle cellule contenant la valeur 5 et mise à jour des liens

Ajout dans une Liste Chaînée Exemple (3)



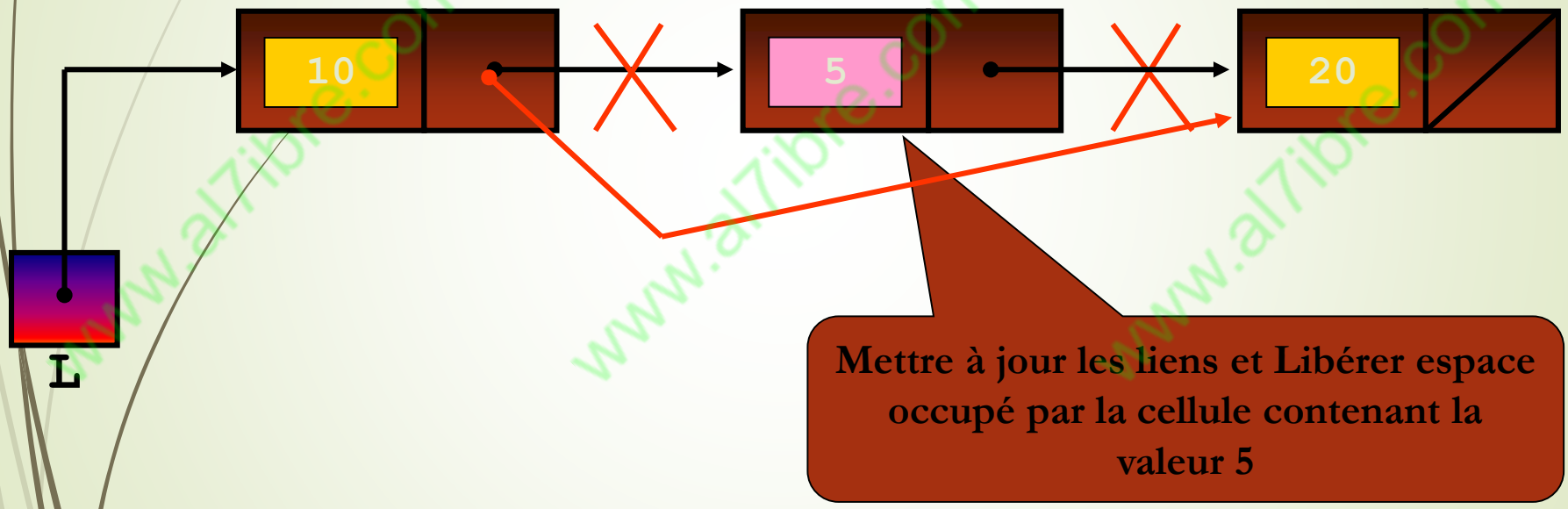
Liste après insertion de la valeur 5

Suppression dans une Liste Chaînée Exemple (1)

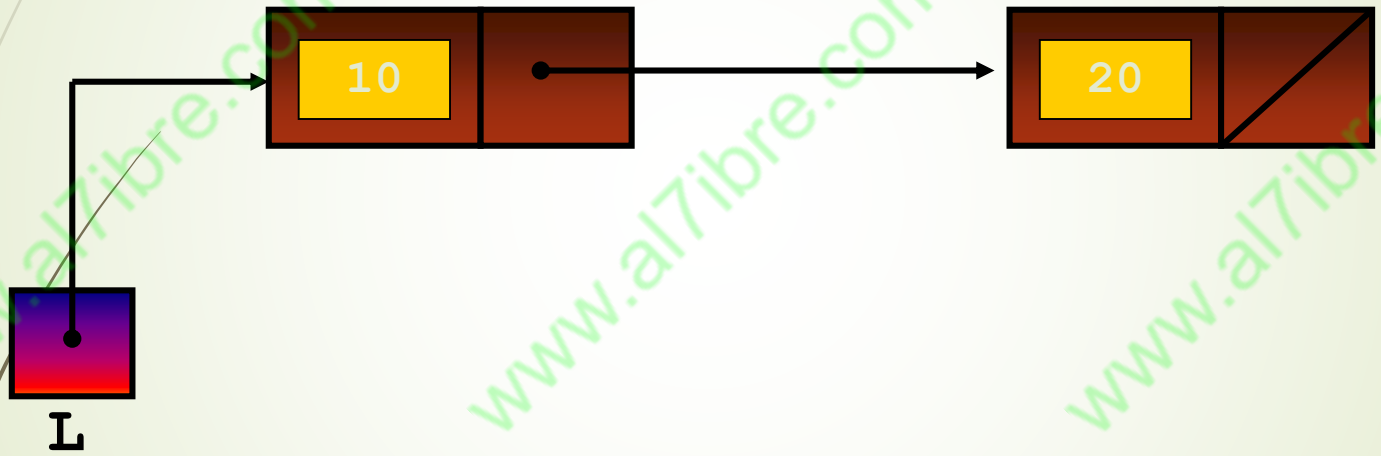


Liste avant suppression de la valeur 5

Suppression dans une Liste Chaînée Exemple 2

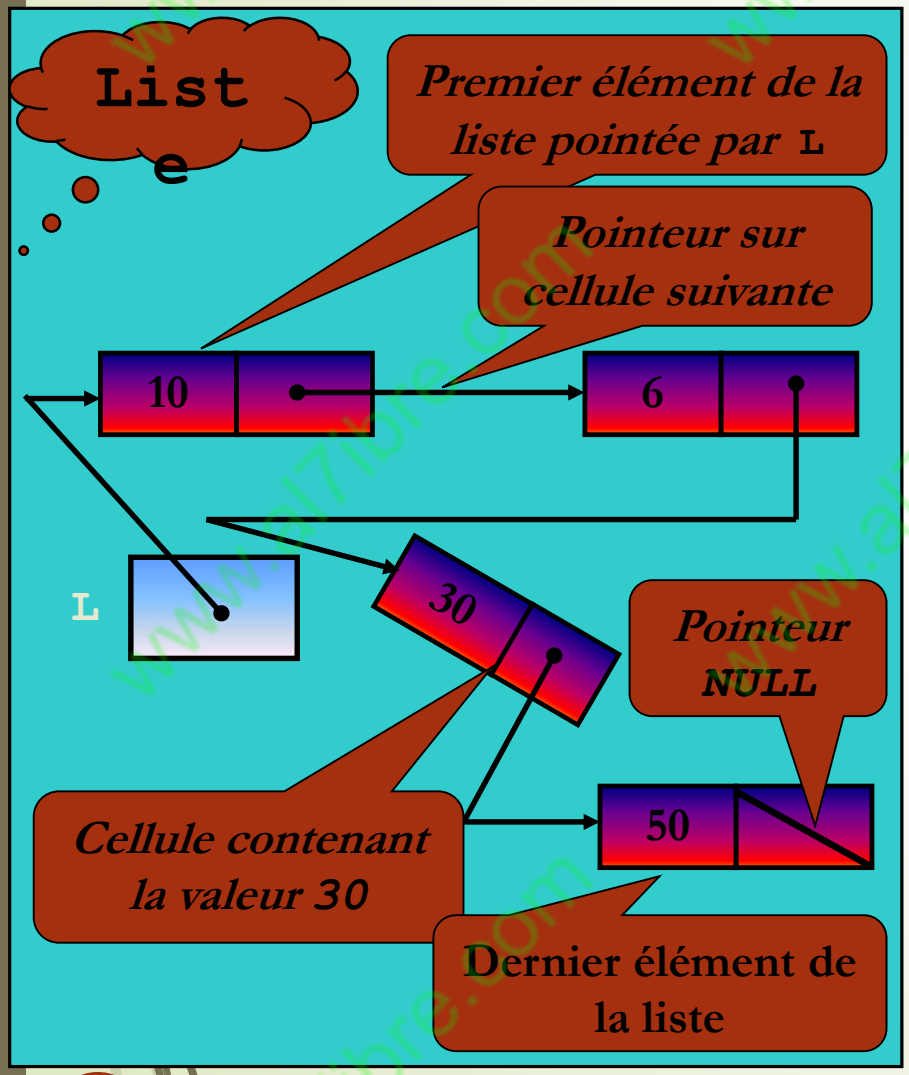


Suppression dans une Liste Chaînée Exemple (3)



Liste après suppression de la valeur 5

Liste Chaînée (Linked List)




```
   /* Liste chaînée en C */

// type des éléments
typedef int element;

// type Place
typedef struct Cellule* Place;

// type Cellule
typedef struct Cellule {
  element valeur;
  struct Cellule *suivant;
 } Cellule;

// type Liste
typedef Cellule *Liste;
```

Spécification d'une Liste Chaînée


```
                                    /* fichier "CListe.h" */
#ifndef _LISTE_CHAINEE
#define _LISTE_CHAINEE

// Définition du type liste (implémentée par pointeurs)
typedef int element;  /* les éléments sont des int */

typedef struct cellule *Place;  /* la place = adresse cellule */

typedef struct cellule {
 element valeur;  /* un éléments de la liste */
 struct cellule *suivant;   /* adresse cellule suivante */
} Cellule;

typedef Cellule *Liste;

// Déclaration des fonctions gérant la liste
Liste liste_vide (void);
int longueur (Liste l);
Liste inserer (Liste l, int i, element e);
Liste supprimer (Liste l, int i);
element keme (Liste l, int k);
Place acces (Liste l, int i);
element contenu (Liste l, Place i);
Place succ (Liste l, Place i);
#endif
```

type Liste : un pointeur de Cellule

Réalisation d'une Liste Chaînée (1)


```
Liste liste_vide(void) {
 return NULL;
}

int longueur(Liste l) {
 int taille=0;
 Liste p=l;
 while (p) {
   taille++;
   p=p->suivant;
 }
 return taille;
}
Liste inserer(Liste l, int i, element e) {
// précondition :1 ≤ i < longueur(l)+1
 if (i<1 || i>longueur(l)+ 1 {
  printf("Erreur : rang non valide !\n");
  exit(-1);
 }
 Liste pc = (Liste)malloc(sizeof(Cellule));
 pc->valeur=e;
 pc->suivant=NULL;
 if (i==1){
   pc->suivant=l;
   l=pc;
 }
 else {
   int j;
   Liste p=l;
   for (j=1; j<i-1; j++)
     p=p->suivant;
   pc->suivant=p->suivant;
   p->suivant=pc;
 }
 return l;
}

Place acces(Liste l, int k) {
// pas de sens que si 1 ≤ k ≤ longueur(l)
 int i;
 Place p;
 if (k<1 || k>longueur(l)) {
   printf("Erreur: rang invalide !\n");
   exit(-1);
 }
 if (k == 1)
   return l;
 else {
   p=l;
   for(i=1; i<k; i++)
     p=p->suivant;
   return p;
 }
}
```

Réalisation d'une Liste Chaînée (2)


```
element contenu(Liste l, Place p) {
// pas de sens si longueur(l)=0 (liste vide)
 if (longueur(l) == 0) {
   printf("Erreur: liste vide !\n");
   exit(-1);
 }
 return p->valeur;
}

Place succ(Liste l, Place p) {
// pas de sens si p dernière place de liste
 if (p->suivant == NULL) {
   printf("Erreur: suivant dernière place!\n");
   exit(-1);
 }
 return p->suivant;
}

element keme(Liste l, int k) {
// pas de sens que si 1 <= k <= longueur(l)
 if (k<1 || k>longueur(l)) {
   printf("Erreur : rang non valide !\n");
   exit(-1);
 }
 return contenu(l, acces(l,k));
}
```

```
Liste supprimer(Liste l, int i) {
// précondition : 1 ≤ i ≤ longueur(l)
 int j;
 Liste p;
 if (i<1 || i>longueur(l)) {
   printf("Erreur: rang non valide!\n");
   exit(-1);
 }
 if (i == 1) {
   p=l;
   l=l->suivant;
 }
 else {
   Place q;
   q=acces(l,i-1);
   p=succ(l,q);
   q->suivant=p->suivant;
 }
 free(p);
 return l;
}
```

Remarques (1)

- **Ajout au milieu d'une liste connaissant la place qui précède celle où s'effectuera l'ajout**
  - **`ajouter : Liste x Place x Elément → Liste`**
  - *`ajouter(L,p,e)` : liste obtenue à partir de `L` en ajoutant une place contenant l'élément `e`, juste après la place `p`*
- **Enlever un élément d'une liste connaissant sa place**
  - **`enlever : Liste x Place → Liste`**
  - *`enlever(L,p)` : liste obtenue à partir de `L` en supprimant la place `p` et son contenu*

Remarques


```
Liste ajouter(Liste l,
              Place p, element e) {
 Liste pc;
 pc=(Liste)malloc(sizeof(Cellule));
 if (pc == NULL) {
   printf("Erreur: Problème de "
          "mémoire\n");
   exit(-1);
 }
 pc->valeur = e;
 pc->suivant = p->suivant;
 p->suivant = pc
 return l;
}
```

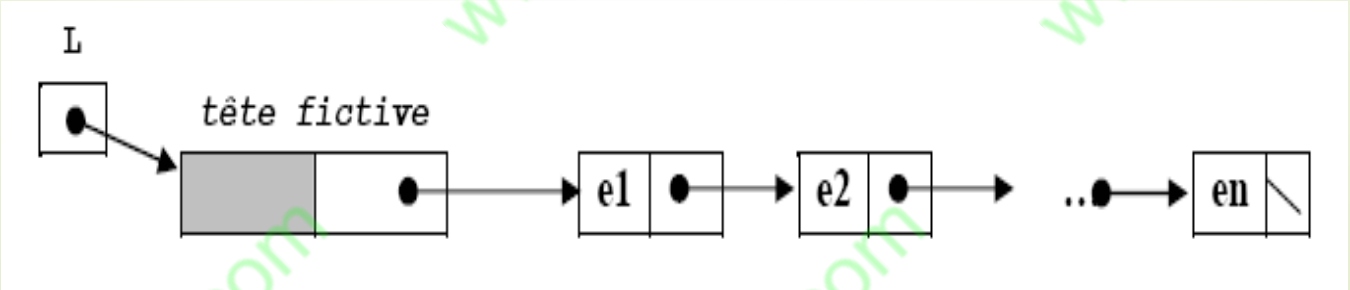
```
Liste enlever(Liste l, Place p) {
// p pointe élément à supprimer
 Place pred; // pred pointe avant p
 if (p == l)
   l = succ(l,p);
   else {
     pred=l;
     while (succ(l,pred) != p)
       pred = succ(l,pred);
     pred->suivant = p->suivant;
   }
 free(p);
 return l;
}
```

Variantes de Listes Chaînées

- ***Liste avec tête fictive***
- ***Liste chaînée circulaire***
- ***Liste doublement chaînée***
- ***Liste doublement chaînée circulaire***
- ***Liste triée***

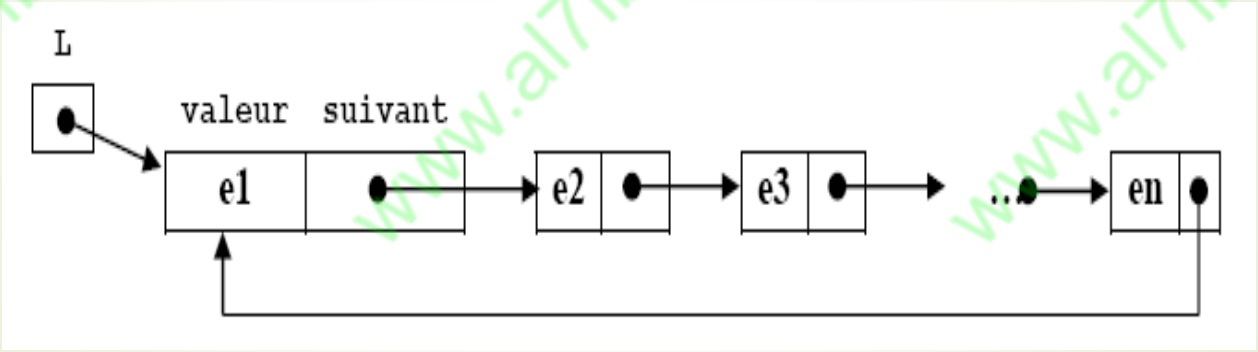
Liste avec Tête Fictive

- **Eviter d'avoir un traitement particulier pour le cas de la tête de liste (*opérations d'insertion et de suppression*)**
  - Mettre en tête de liste une zone qui ne contient pas de valeur et reste toujours en tête



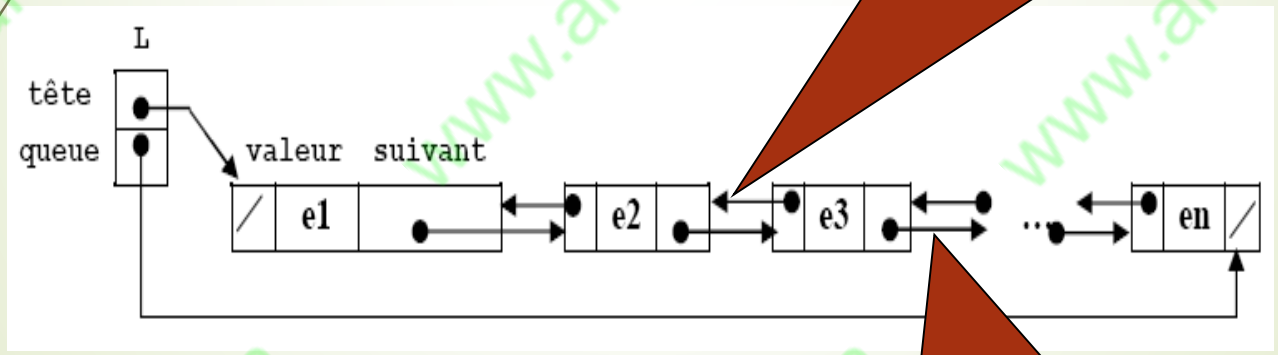
Liste Circulaire

- **Le suivant du dernier élément de la liste est le premier élément**



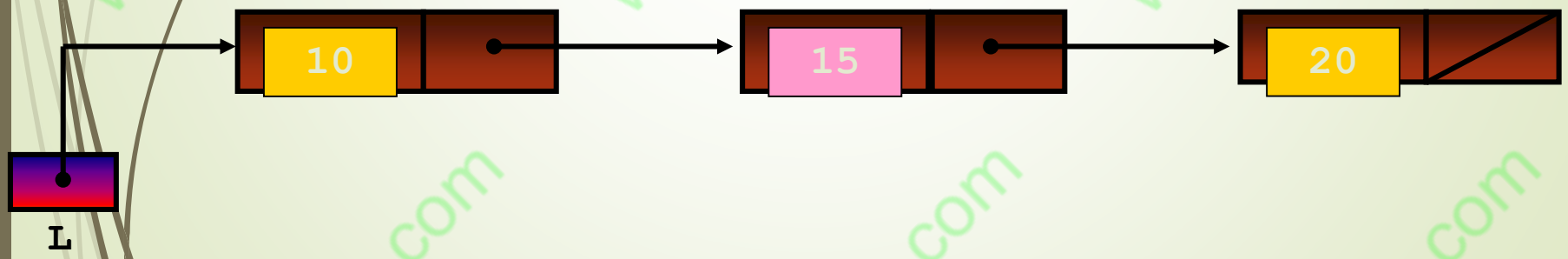
Liste Doublement Chaînée

- **Faciliter le parcours de la liste dans les deux sens**
  - utiliser un double chaînage ; chaque place repérant à la fois la place qui la précède et celle qui la suit



Liste Triée

- Dans cette liste, il existe un ordre total sur les clés
- L'ordre des enregistrements dans la liste respecte l'ordre sur les clés



Complexité

- **n désigne le *nombre d'éléments* d'une liste**

Opération	Représentation contiguë	Représentation chaînée
liste_vide	O(1)	O(1)
ajout/suppression en tête	O(n)	O(1)
ajout/suppression générale	O(n)	O(n)
ajout/suppression en queue	O(1)	O(n)
accès	O(1)	O(n)
concaténation	O(n)	O(1)

Les Piles & Files (Stacks)

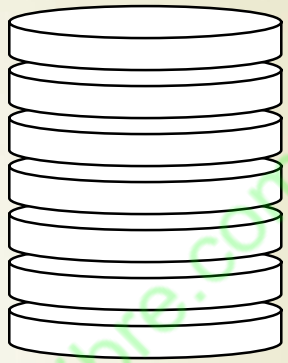
Pr F.Omary

FSR-Université MohammedV

2019-2020

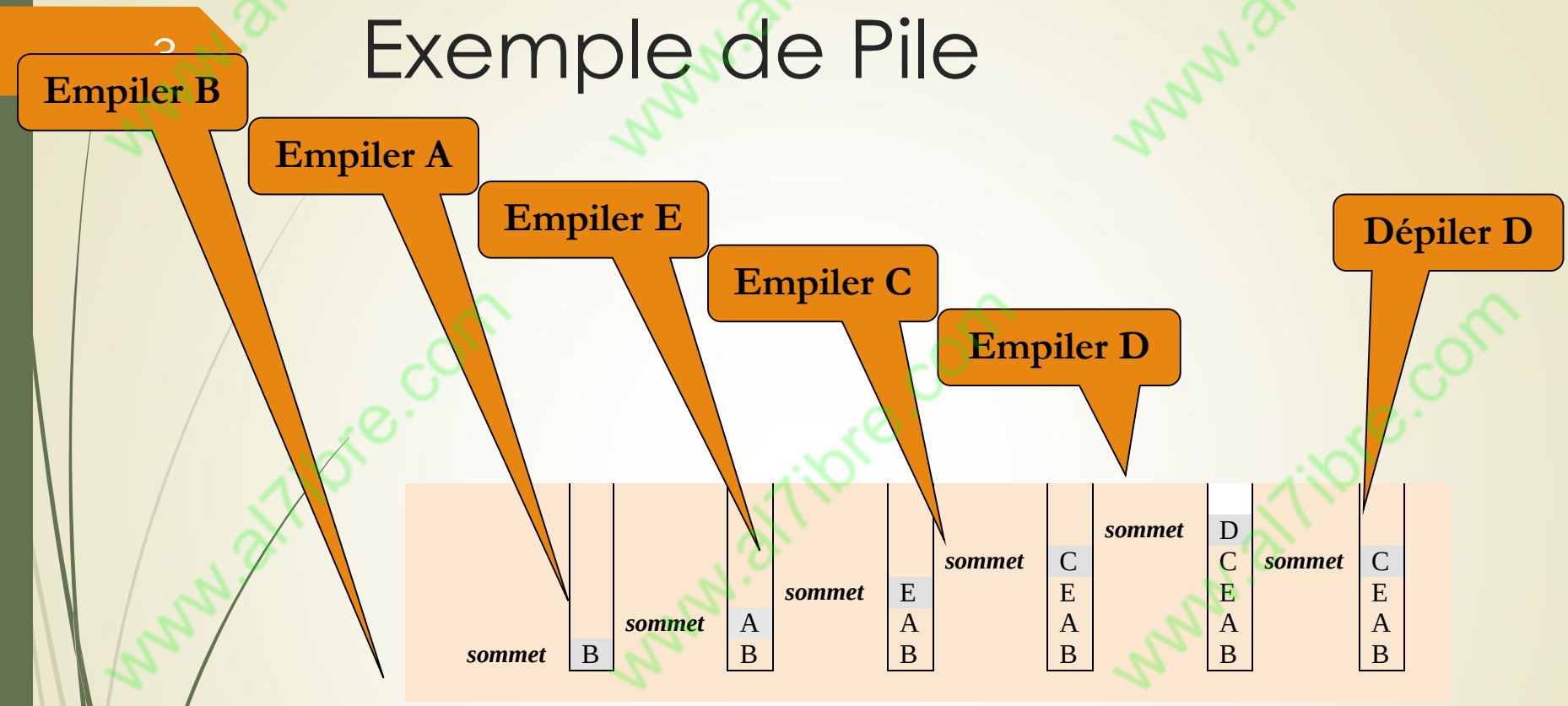
Notion de Pile (Stack)

- Les piles sont très utilisées en informatique



- Notion intuitive :
  - pile d'assiettes, pile de dossiers à traiter, …
- Une pile est une structure linéaire permettant de stocker et de restaurer des données selon un ordre LIFO (Last In, First Out ou « dernier entré, premier sorti »)
- Dans une pile :
  - Les insertions (empilements) et les suppressions (dépilements) sont restreintes à une extrémité appelée sommet de la pile.

Exemple de Pile



Type Abstrait Pile

Type Pile
Utilise Elément, Booléen
Opérations
  pile_vide : → Pile
  est_vide : Pile → Booléen
  empiler : Pile x Elément → Pile
  dépiler : Pile → Pile
  sommet : Pile → Elément
Préconditions
  dépiler(p) *est-défini-ssi* est_vide(p) = faux
  sommet(p) *est-défini-ssi* est_vide(p) = faux
Axiomes
  Soit, e : Element, p : Pile
  est_vide(pile_vide) = vrai
  est_vide(empiler(p,e)) = faux
  dépiler(empiler(p,e)) = p
  sommet(empiler(p,e)) = e

Opérations sur une Pile

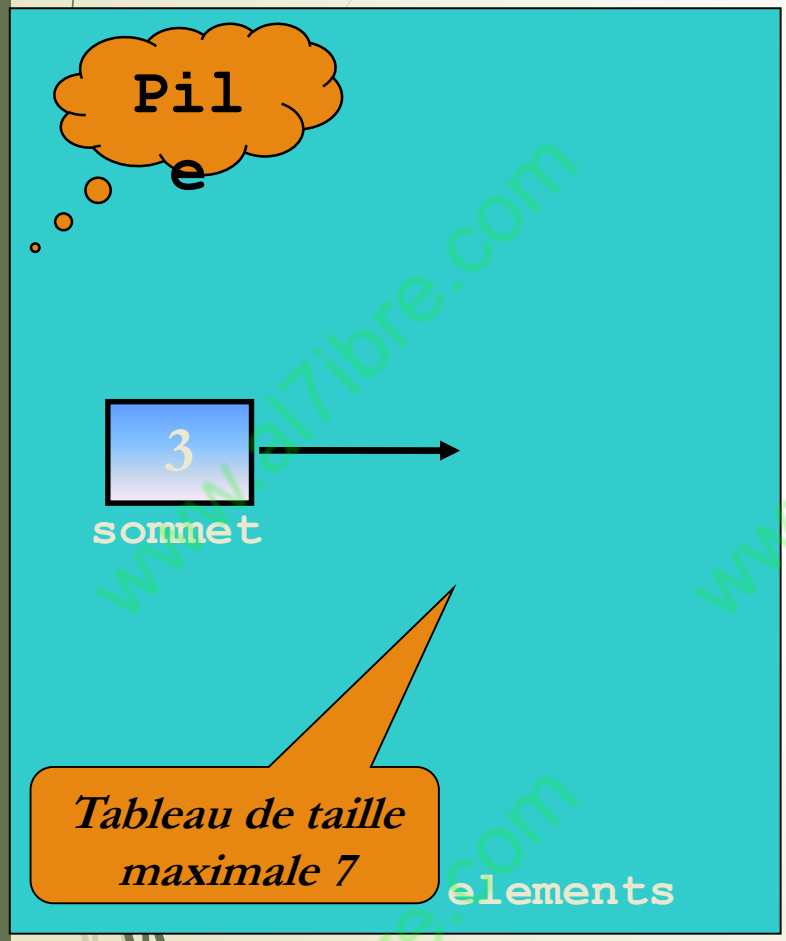
- **pile_vide : → Pile**
  - opération d'initialisation ; *la pile créée est vide*
- **est_vide : Pile → Booléen**
  - teste si pile vide ou non
- **sommet : Pile → Elément**
  - permet de consulter l'élément situé au sommet ; *n'a pas de sens si pile vide*
- **empiler : Pile x Elément → Pile**
  - ajoute un élément dans la pile
- **dépiler : Pile → Pile**
  - enlève l'élément situé au sommet de la pile ; *n'a pas de sens si pile vide*

Représentation d'une Pile

- **Représentation contiguë (*par tableau*) :**
  - *Les éléments de la pile sont rangés dans un tableau*
  - *Un entier représente la position du sommet de la pile*

- **Représentation chaînée (par pointeurs) :**
  - *Les éléments de la pile sont chaînés entre eux (voir listes chaînées)*
  - *Un pointeur sur le premier élément désigne la pile et représente le sommet de cette pile*
  - *Une pile vide est représentée par le pointeur* `NULL`

Pile Contiguë




```
    /* Pile contiguë en C */

// taille maximale pile
#define MAX_PILE 7

// type des éléments
typedef int Element;

// type Pile
typedef struct {
 Element elements[MAX_PILE];
 int sommet;
} Pile;
```

Spécification d'une Pile Contiguë

```
/*fichier "Tpile.h" */
#ifndef _PILE_TABLEAU
#define _PILE_TABLEAU

#include "Booleen.h"
// Définition du type Pile (implémentée par un tableau)
#define MAX_PILE 7  /* taille maximale d'une pile */
typedef int Element;  /* les éléments sont des int */

typedef struct {
 Element elements[MAX_PILE]; /* les éléments de la pile */
 int sommet;  /* position du sommet */
} Pile;
// Déclaration des fonctions gérant la pile
Pile pile_vide ();
Pile empiler ( Pile p, Element e );
Pile depiler ( Pile p );
Element sommet ( Pile p );
Booleen est_vide ( Pile p );

#endif
```

type Pile : une structure à deux champs

Réalisation d'une Pile Contiguë

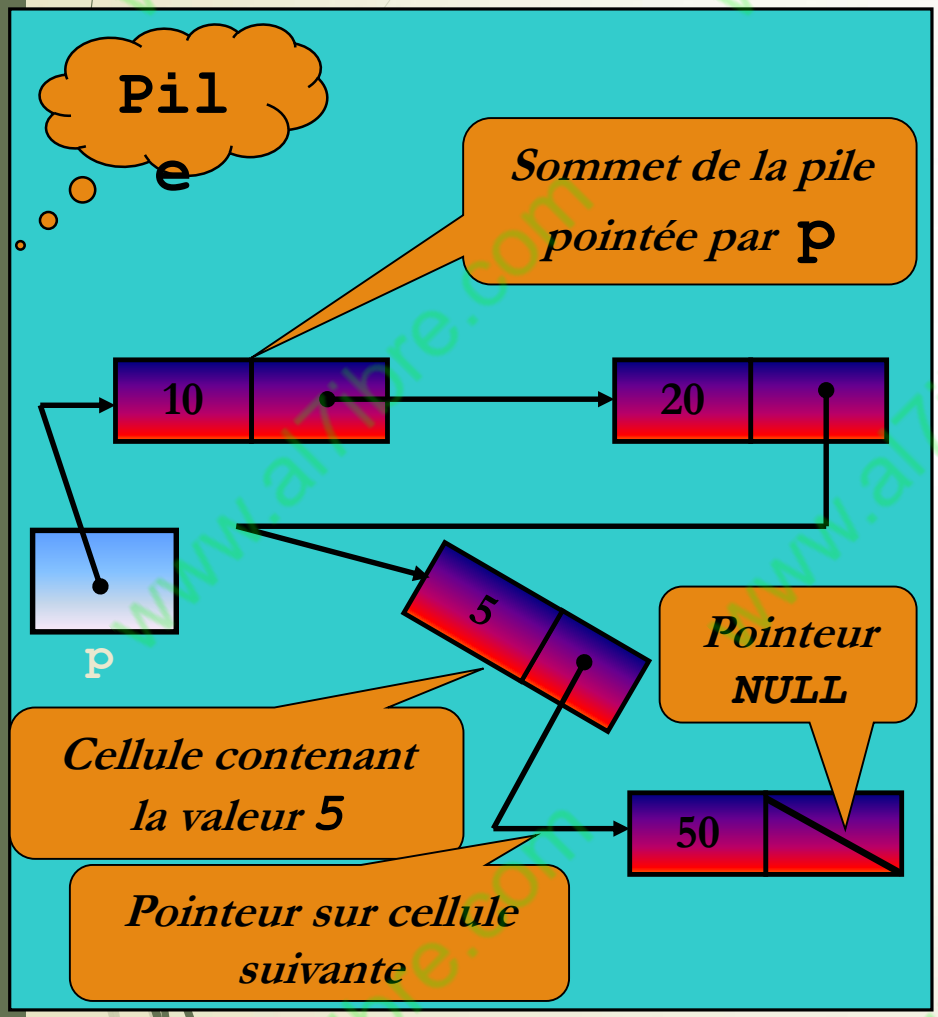

```
/* fichier "Tpile.c" */

#include "Tpile.h"
// Définition des fonctions gérant la pile
// initialiser une nouvelle pile
 Pile pile_vide() {
 Pile p;
 p.sommet = -1;
 return p;
}
// tester si la pile est vide
Booleen est_vide(Pile p) {
 if (p.sommet == -1) return vrai;
 return faux;
}
// Valeur du sommet de pile
Element sommet(Pile p) {
/* pré-condition : pile non vide ! */
 if (est_vide(p)) {
  printf("Erreur: pile vide !\n");
  exit(-1);
 }
 return (p.elements)[p.sommet];
}
```

```
// ajout d'un élément
Pile empiler(Pile p, Element e) {
 if (p.sommet >= MAX_PILE-1) {
  printf("Erreur : pile pleine !\n");
  exit(-1);
 }
 (p.sommet)++;
 (p.elements)[p.sommet] = e;
 return p;
}

// enlever un élément
Pile depiler(Pile p) {
/* pré-condition : pile non vide !*/
 if (est_vide(p)) {
  printf("Erreur: pile vide !\n");
  exit(-1);
 }
 p.sommet--;
 return p;
}
```

Exemple d'Utilisation d'une Pile Contiguë


```
                    /* fichier "UTpile.c" */
#include <stdio.h>
#include "Tpile.h"

int main () {
 Pile p = pile_vide();

 p = empiler(p,50);
 p = empiler(p,5);
 p = empiler(p,20);
 p = empiler(p,10);
 printf("%d au sommet après empilement de 50, 5, 20 et"
        " 10\n", sommet(p));
 p = depiler(p);
 p = depiler(p);
 printf("%d au sommet après dépilement de 10 et 20\n",
        sommet(p));
 return 0;
}
```

Pile chaînée




```
   /* Pile chaînée en C */

// type des éléments
typedef int element;

// type Cellule
typedef struct cellule {
   element valeur;
   struct cellule *suivant;
 } Cellule;

// type Pile
typedef Cellule *Pile;
```

Complexité

- **Les opérations sur les piles sont toutes en O(1)**
- **Ceci est valable pour les deux représentations proposées**

Applications d'une Pile Exemples (1)

- **Vérification du bon équilibrage d'une expression parenthèsée :**
  - Pour vérifier qu'une expression parenthèsée est équilibrée, à chaque rencontre d'une parenthèse ouvrante on l'empile et à chaque rencontre d'une parenthèse fermante on dépile ;
- **Evaluation des expressions arithmétiques postfixées (*expressions en notation polonaise inverse*) :**
  - Pour évaluer une telle expression, on applique chaque opérateur aux deux opérandes qui le précédent. Il suffit d'utiliser une pile dans laquelle les opérandes sont empilés, alors que les opérateurs dépilent deux éléments, effectuent l'opération et empilent le résultat. Par exemple, l'expression postfixée ***2 3 5 * + 1 –*** s'évalue comme suit : ***((2 (3 5 *) +) 1 –) = 16*** ;
- **Conversion d'une expression en notation infixe (*parenthèsée*) en notation postfixée ;**

Applications d'une Pile
Exemples (2)

- **Gestion par le compilateur des appels de fonctions :**
  - les paramètres, l'adresse de retour et les variables locales sont stockés dans la pile de l'application
- **Mémorisation des appels de procédures imbriquées au cours de l'exécution d'un programme, et en particulier les appels des procédures récursives ;**
- **Parcours « en profondeur » de structures d'arbres (*voir arbres*) ;**

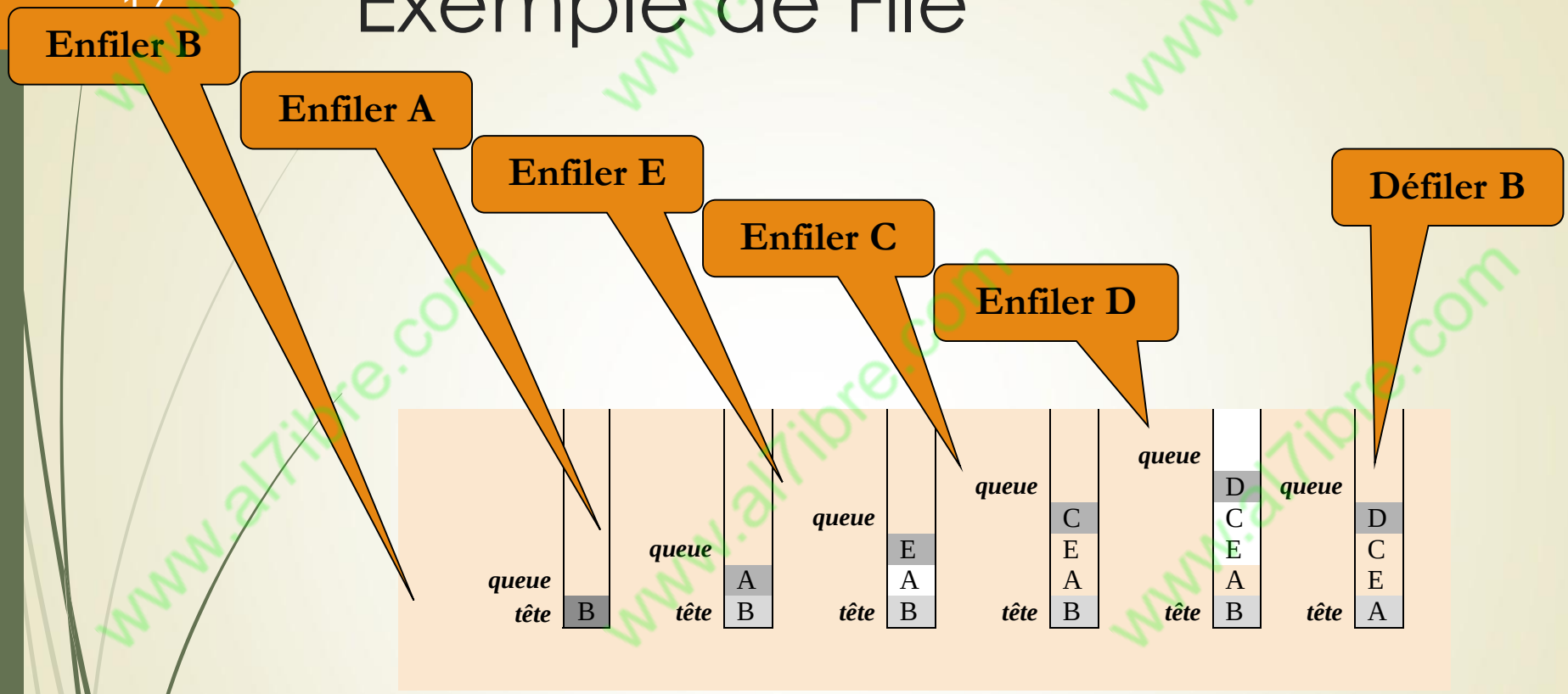
Les Files (Queues)

Notion de File (Queue)

- **Les files sont très utilisées en informatique**
- **Notion intuitive :**
  - File d'attente à un guichet, file de documents à imprimer, …

- **Une file est une structure linéaire permettant de stocker et de restaurer des données selon un ordre FIFO (*First In, First Out* ou « premier entré, premier sorti »)**
- **Dans une file :**
  - Les insertions (***enfilements***) se font à une extrémité appelée ***queue*** de la file et les suppressions (***défilements***) se font à l'autre extrémité appelée ***tête*** de la file

Exemple de File



Type Abstrait File


```
Type File
Utilise Elément, Booléen
Opérations
    file_vide    : → File
    est_vide     : File → Booléen
    enfiler  : File x Elément → File
    défiler  : File → File
    tête     : File → Elément
Préconditions
    défiler(f) est-défini-ssi est_vide(f) = faux
    tête(f) est-défini-ssi est_vide(f) = faux
Axiomes
    Soit, e : Element, f : File
    est_vide(file_vide) = vrai
    est_vide(enfiler(f,e)) = faux
    si est_vide(f) = vrai alors tête(enfiler(f,e)) = e
    si est_vide(f) = faux alors tête(enfiler(f,e)) = tête(f)
    si est_vide(f) = vrai alors défiler(enfiler(f,e)) = file_vide
    si est_vide(f) = faux
      alors défiler(enfiler(f,e)) = enfiler(défiler(f),e)
```

Opérations sur une File

- **file_vide : → File**
  - opération d'initialisation ; *la file créée est vide*
- **est_vide : File → Booléen**
  - teste si file vide ou non
- **tête : File → Elément**
  - permet de consulter l'élément situé en tête de file ; *n'a pas de sens si file vide*
- **enfiler : File x Elément → File**
  - ajoute un élément dans la file
- **défiler : File → File**
  - enlève l'élément situé en tête de file ; *n'a pas de sens si file vide*

Représentation d'une File

- **Représentation contiguë (*par tableau*) :**
  - *Les éléments de la file sont rangés dans un tableau*
  - *Deux entiers représentent respectivement les positions de la tête et de la queue de la file*
- **Représentation chaînée (par pointeurs) :**
  - *Les éléments de la file sont chaînés entre eux (voir listes chaînées)*
  - *Un pointeur sur le premier élément désigne la file et représente la tête de cette file*
  - *Un pointeur sur le dernier élément représente la queue de file*
  - *Une file vide est représentée par le pointeur* `NULL`

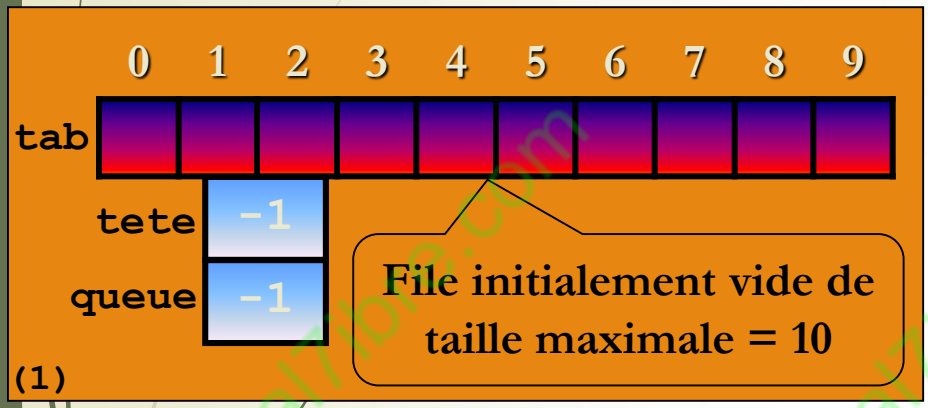
Représentation Contiguë d'une File (par tableau simple)

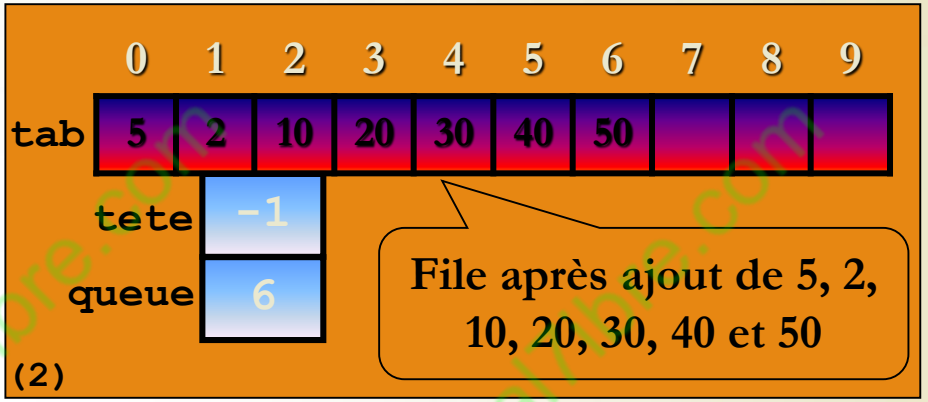
- ***tête de file :*** position précédant premier élément
- ***queue de file :*** position du dernier élément

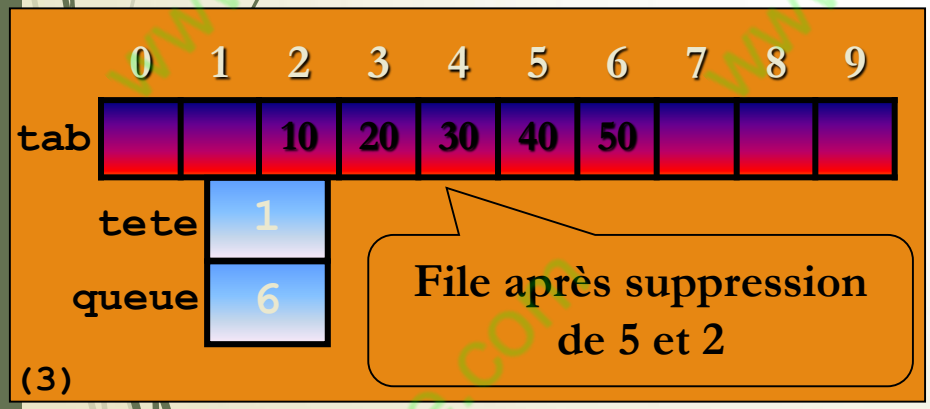
- ***Initialisation :*** `tête ← queue ← -1`

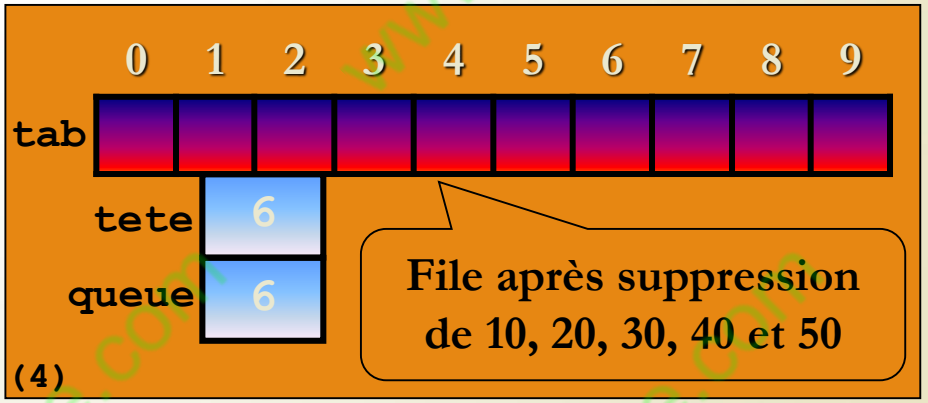
- ***Inconvénient :*** on ne peut plus ajouter des éléments dans la file, alors qu'elle n'est pas pleine !

Représentation Contiguë d'une File (par tableau simple) (1)

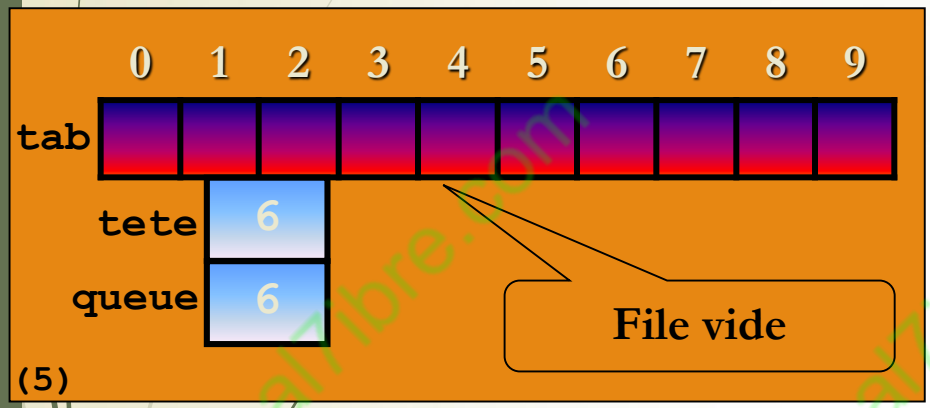


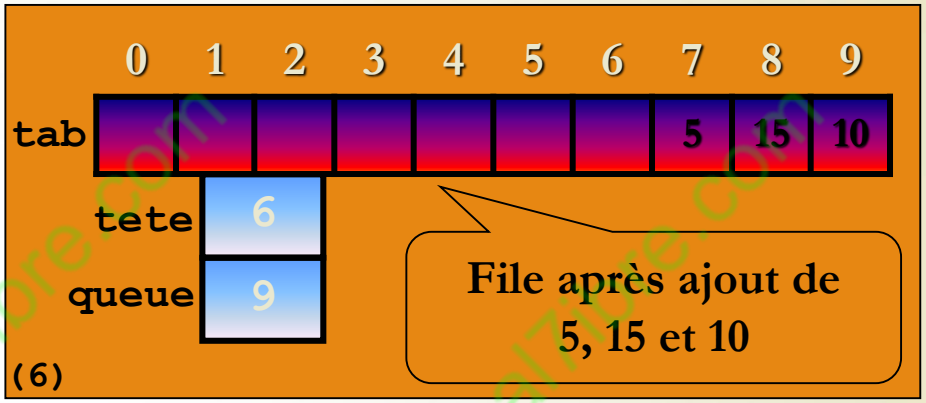


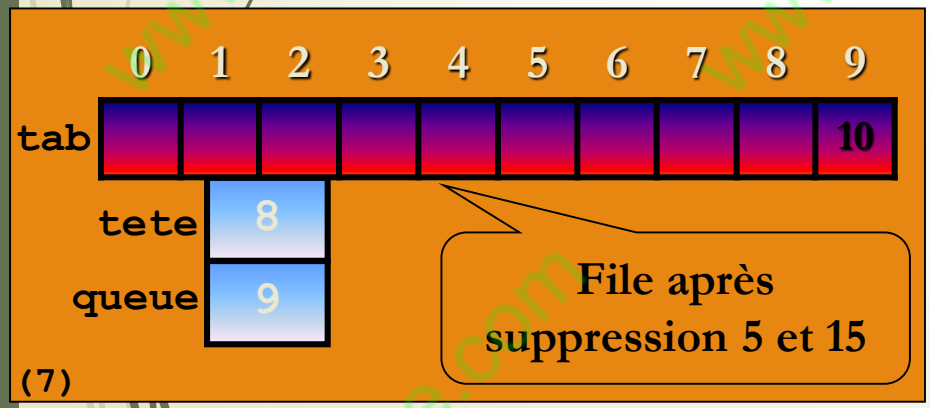


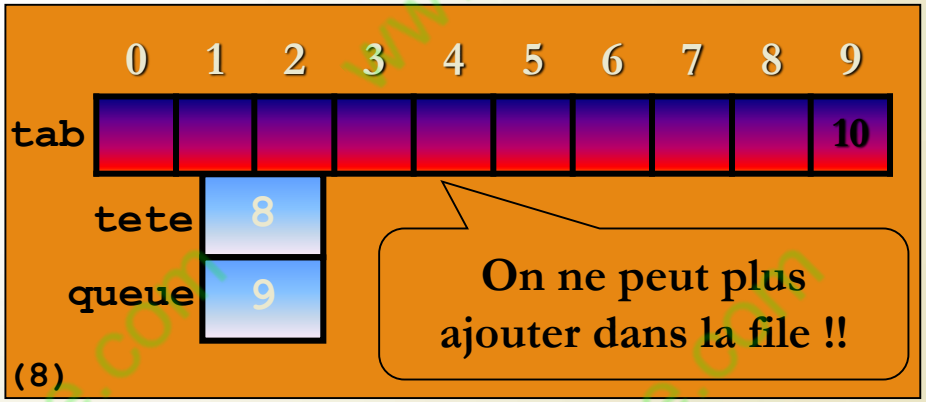


Représentation Contiguë d'une File (par tableau simple) (2)

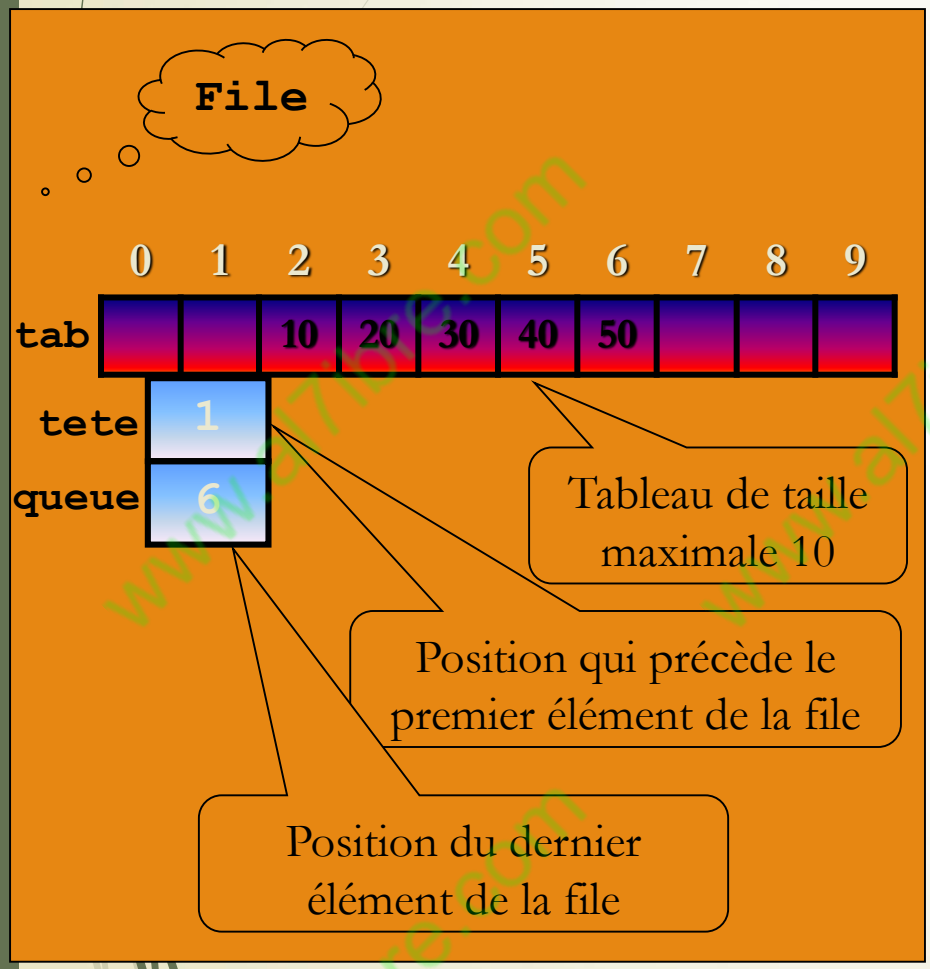








File Contiguë




```
  /* File contiguë en C */

// taille maximale file
#define MAX_FILE 10

// type des éléments
typedef int Element;

// type File
typedef struct {
 Element tab[MAX_FILE];
 int tete;
 int queue;
} File;
```

Spécification d'une File Contiguë


```
                    /* fichier "Tfile.h" */
#ifndef _FILE_TABLEAU
#define _FILE_TABLEAU

#include "Booleen.h"

// Définition du type File (implémentée par un tableau simple)
#define MAX_FILE 10  /* taille maximale d'une file */
typedef int Element;  /* les éléments sont des int */

typedef struct {
 Element tab[MAX_FILE]; /* les éléments de la file */
 int tete;  /* position précédant premier élé
 int queue; /* position dernier element */
} File;
// Déclaration des fonctions gérant la pile
File file_vide ();
File enfiler ( File f, Element e );
File defiler ( File f );
Element tete ( File f );
Booleen est_vide ( File f );

#endif
```

type File : une structure à trois champs

Réalisation d'une File Contiguë

```
/* fichier "Tfile.c"*/

#include "Tfile.h"
// Définition des fonctions gérant la file
// initialiser une nouvelle file
File file_vide() {
 File f;
 f.queue = f.tete = -1;
 return f;
}
// tester si la file est vide
Booleen est_vide(File f) {
 if (f.tete == f.queue) return vrai;
 return faux;
}
// valeur en tête de file
Element tete(File f) {
/* pré-condition : file non vide ! */
 if (est_vide(f)) {
  printf("Erreur: file vide !\n");
  exit(-1);
 }
 return (f.tab)[f.tete+1];
}
```

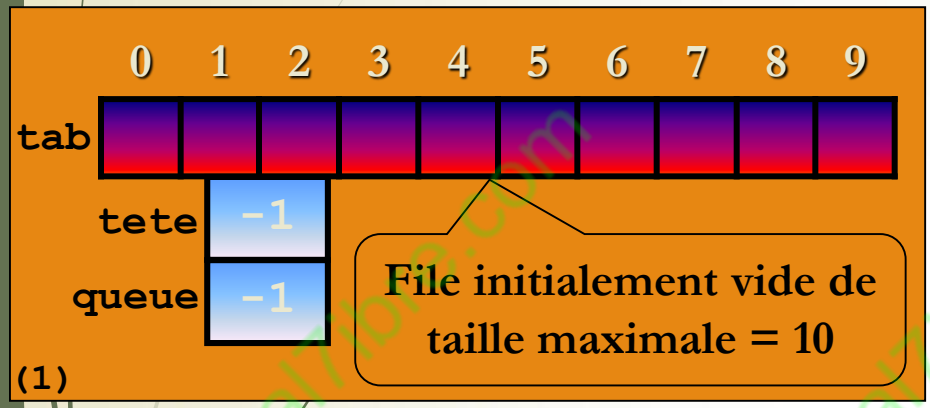
```
// ajout d'un élément
File enfiler(File f, Element e) {
 if (f.queue == MAX_FILE-1) {
  printf("Erreur: on ne peut ajouter !\n");
  exit(-1);
 }
 (f.queue)++;
 (f.tab)[f.queue] = e;
 return f;
}

// enlever un élément
File defiler(File f) {
/* pré-condition : file non vide !*/
 if (est_vide(f)) {
  printf("Erreur: file vide !\n");
  exit(-1);
 }
 f.tete++;
 return f;
}
```

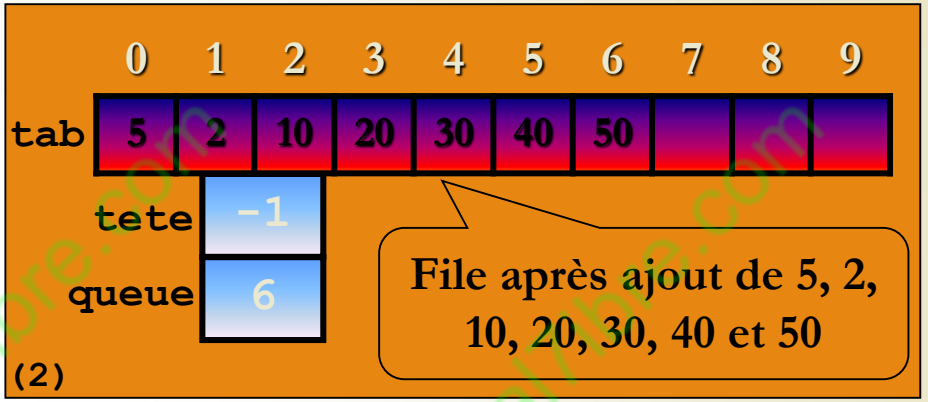
Représentation Contiguë d'une File (par tableau simple avec décalage)

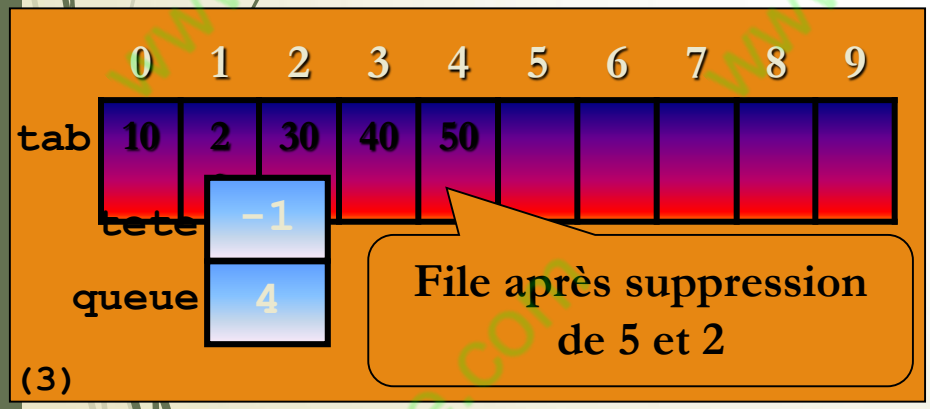
- ***Décaler les éléments de la file après chaque suppression***

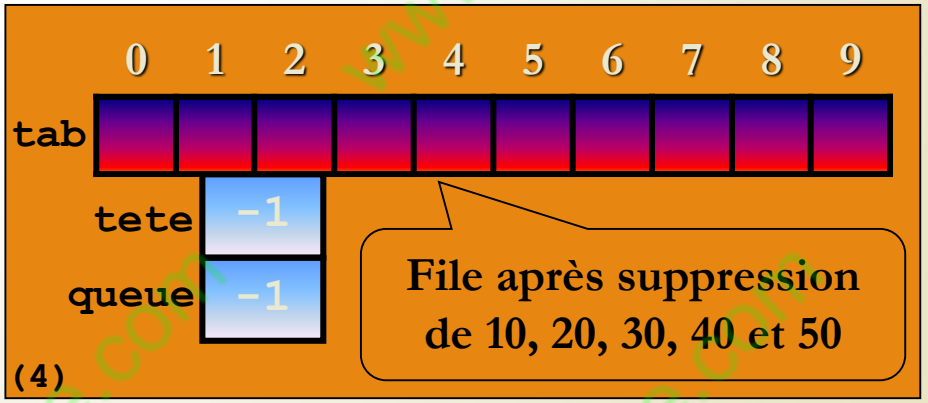
- ***Inconvénient :*** décalage très coûteux si la file contient plusieurs d'éléments

Représentation Contiguë d'une File (par tableau simple avec décalage)

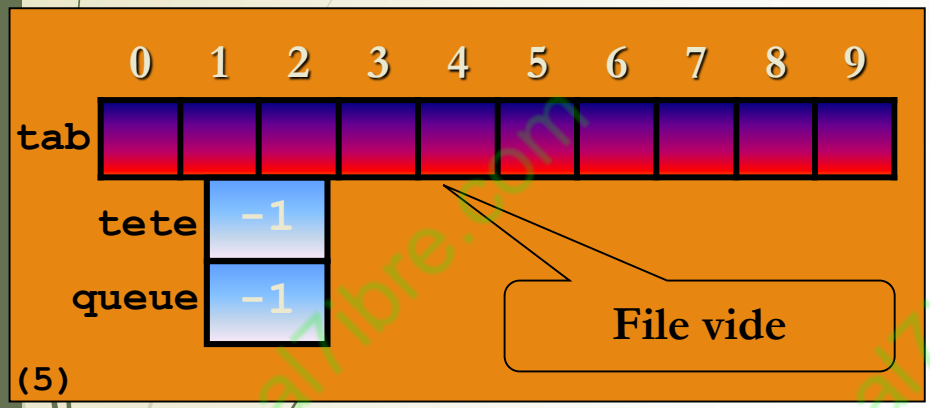


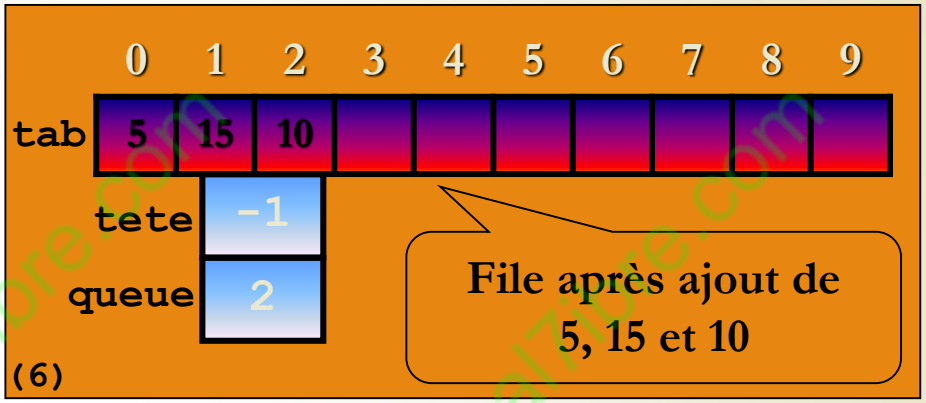


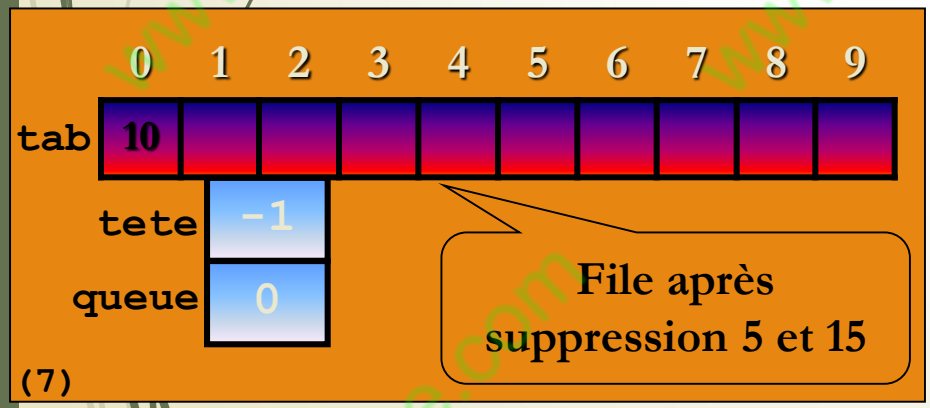


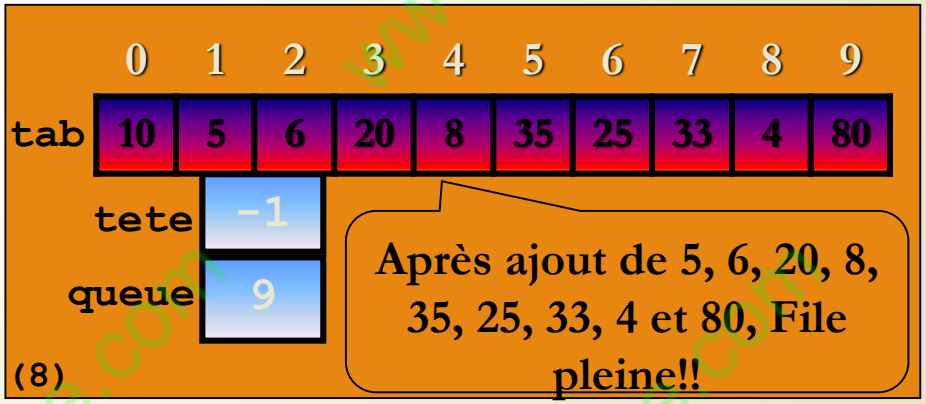


Représentation Contiguë d'une File (par tableau simple avec décalage)









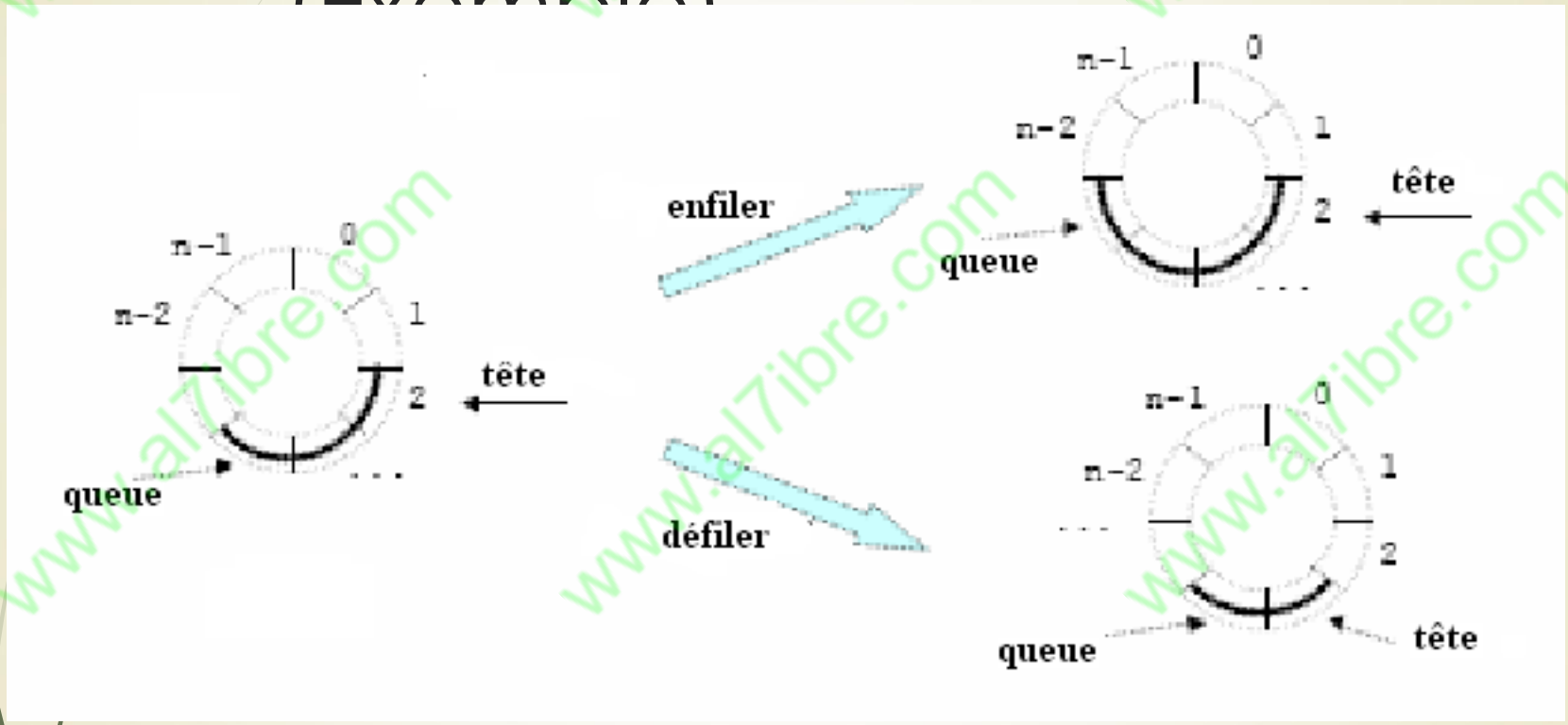
Représentation Contiguë d'une File (Par tableau circulaire)

- ***Gérer le tableau de manière circulaire :*** suivant de l'élément à la position `i` est l'élément à la position **`(i+1) modulo MAX_FILE`**

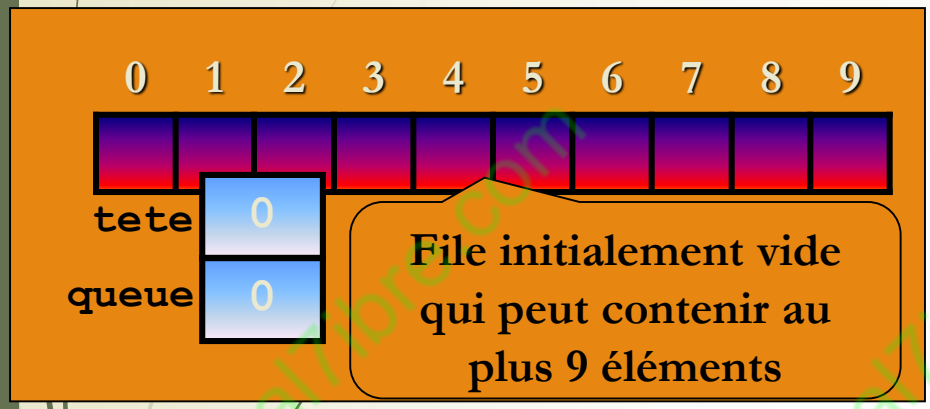
- ***Convention :*** file autorisée à contenir **`MAX_FILE-1`** éléments

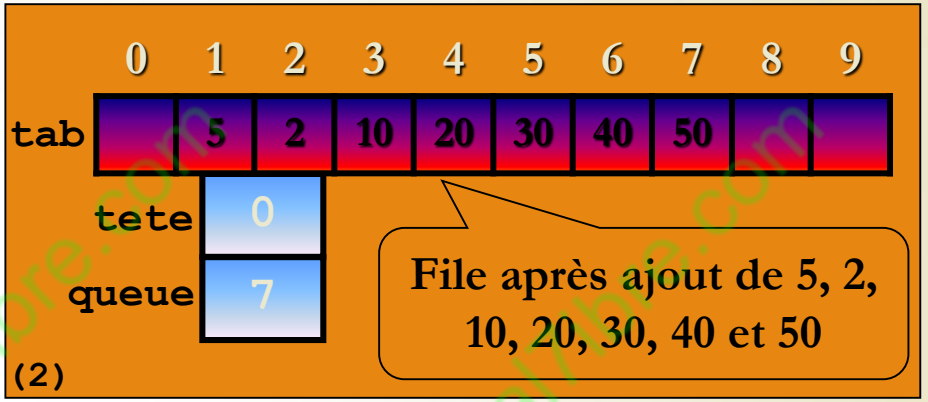
- ***Initialisation :*** **`tête ← queue ← 0`**

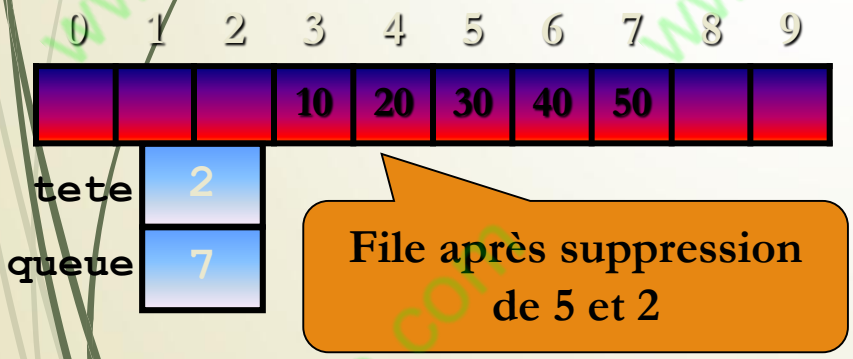
File Contiguë Circulaire (Exemple)

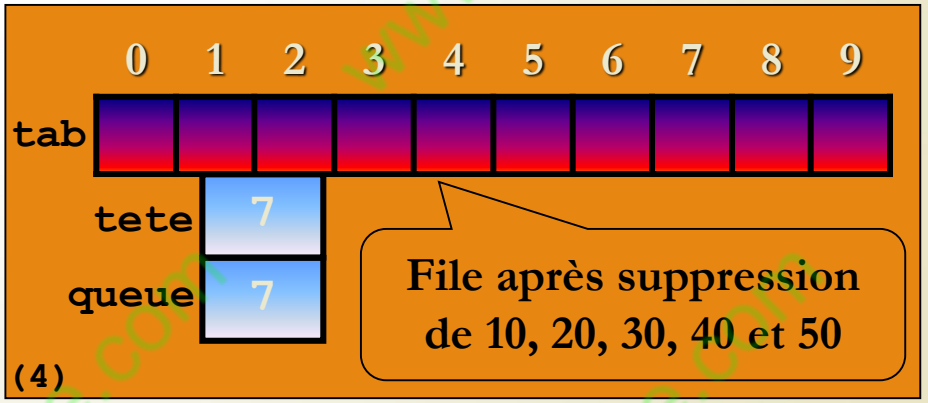


Représentation d'une File Contiguë circulaire(1)

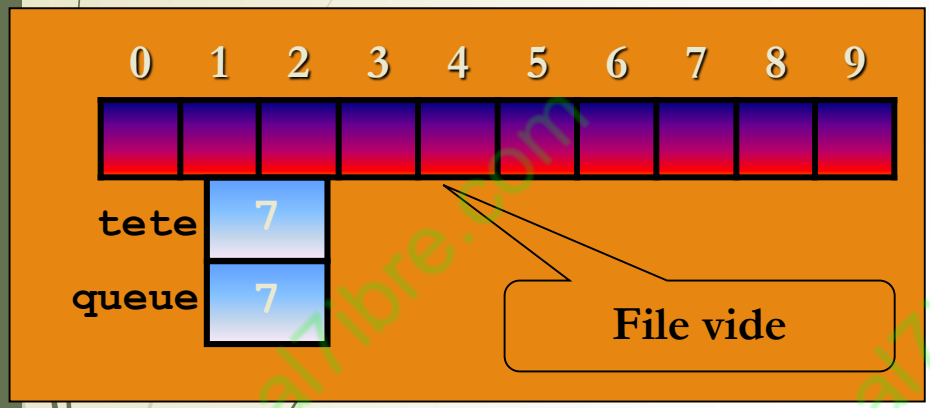


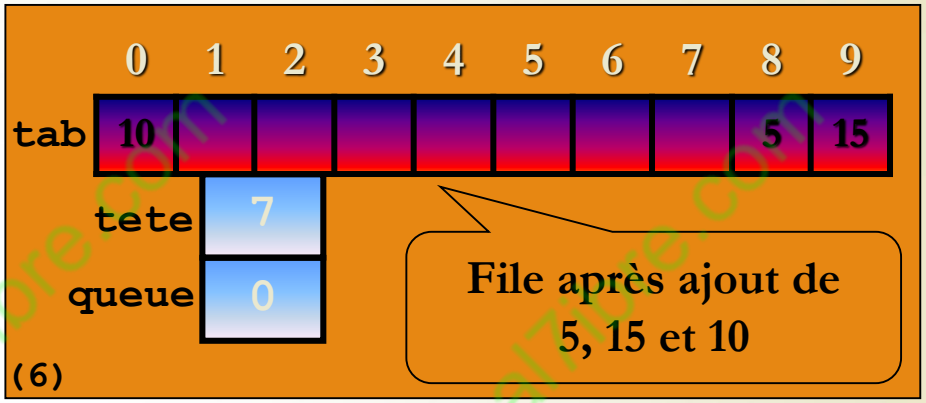


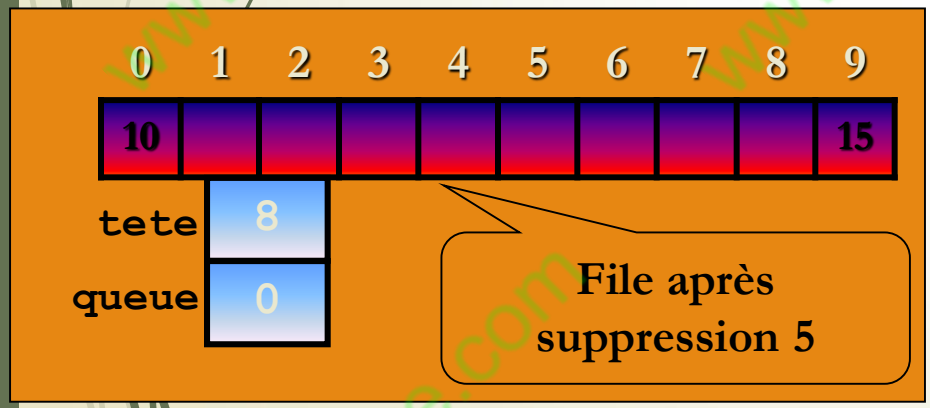


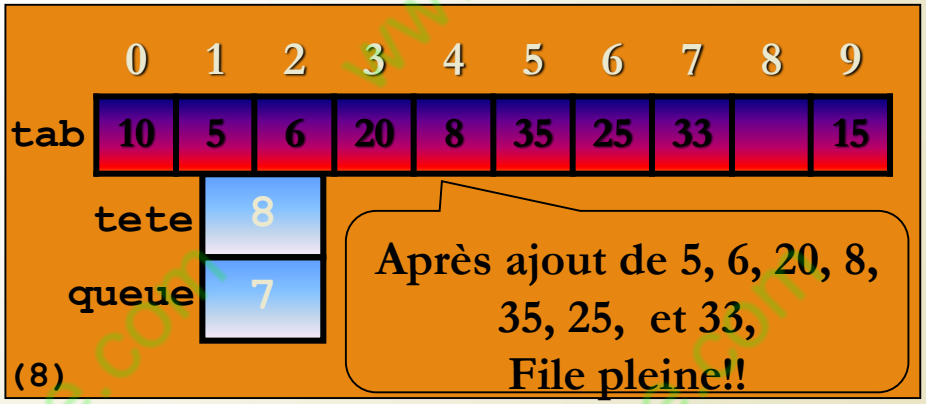


Réalisation d'une File Contiguë Circulaire (2)







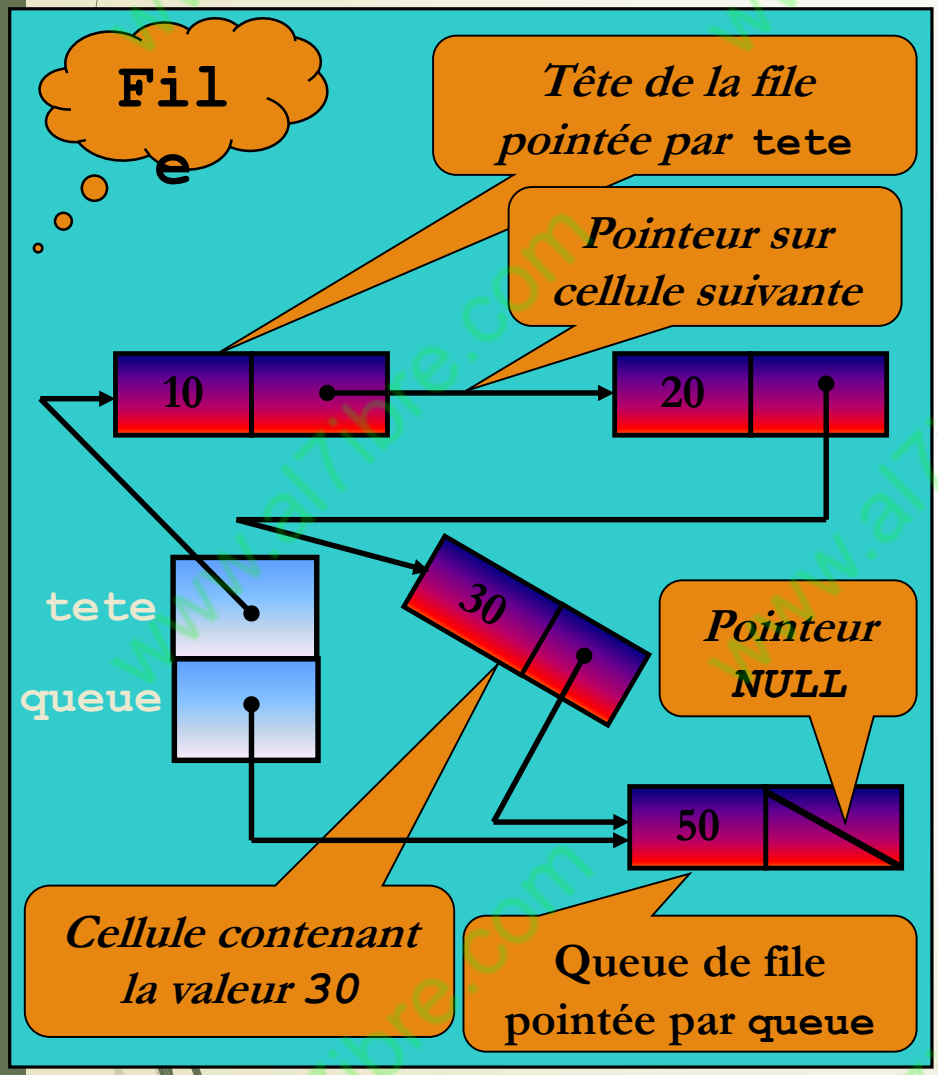


Réalisation d'une File Contiguë Circulaire


```
        /* fichier "TCfile.c" */
#include "Tfile.h"
// Définition des fonctions gérant la file

// initialiser une nouvelle file
File file_vide() {
 File f;
 f.queue = f.tete = 0;
 return f;
 }
// tester si la file est vide
Booleen est_vide(File f) {
 if (f.tete == f.queue) return vrai;
 return faux;
 }
// valeur en tête de file
Element tete(File f) {
/* pré-condition : file non vide ! */
 if (est_vide(f)) {
  printf("Erreur: file vide !\n");
  exit(-1);
  }
  return (f.tab)[(f.tete+1) % MAX_FILE];
  }
```

```
// ajout d'un élément
File enfiler(File f, Element e) {
 if (f.tete == (f.queue+1) % MAX_FILE) {
  printf("Erreur : file pleine !\n");
  exit(-1);
 }
 f.queue=(f.queue+1) % MAX_FILE;
 (f.tab)[f.queue] = e;
 return f;
}

// enlever un élément
File defiler(File f) {
/* pré-condition : file non vide !*/
 if (est_vide(f)) {
  printf("Erreur: file vide !\n");
  exit(-1);
 }
 f.tete=(f.tete+1) % MAX_FILE;
 return f;
}
```

File chaînée




```
   /* File chaînée en C */

// type des éléments
typedef int element;

// type Cellule
typedef struct cellule {
   element valeur;
   struct cellule *suivant;
 } Cellule;

// type File
typedef struct {
 Cellule *tete;
 Cellule *queue;
} File;
```

Complexité

- **Les opérations sur les files sont toutes en O(1)**
- **Ceci est valable pour les deux représentations : file contiguë circulaire et file chaînée**

Applications d'une File Exemples

- **Gestion des travaux d'impression d'une imprimante :**
  - Cas d'une imprimante en réseau, où les tâches d'impressions arrivent aléatoirement de n'importe quel ordinateur connecté. Les tâches sont placées dans une file d'attente, ce qui permet de les traiter selon leur ordre d'arrivée
- **Ordonnanceur (*dans les systèmes d'exploitation*) :**
  - Maintenir une file de processus en attente d'un temps machine ;
- **Parcours « en largeur » d'un arbre (*voir arbres*)**

Cours Structures de données

Arbres (Trees)

Pr F.Omary

2019-2020

Objectifs

- **Etudier des structures non linéaires**
  - **Arbres binaires**
  - **Arbres binaires de recherche**
  - **Arbres maximiers ou Tas**
  - **Arbres équilibrés**

Contenu

- Introduction
- Terminologie
- Arbres binaires
- Arbres binaires de recherche
- Arbres maximiers ou Tas
- Arbres équilibrés

Arbres (Trees)

Introduction

Notion d'Arbre (Tree)

- Les arbres sont les structures de données les plus importantes en informatique
- Ce sont des *structures non linéaires* qui permettent d'obtenir des algorithmes plus performants que lorsqu'on utilise des structures de données linéaires telles que les listes et les tableaux
- Ils permettent une organisation naturelle des données

Notion d'Arbre (Tree) Exemples

- ***Organisation des fichiers dans les systèmes d'exploitation ;***
- ***Organisation des informations dans un système de bases de données ;***
- ***Représentation de la structure syntaxique des programmes sources dans les compilateurs ;***
- ***Représentation d'une table de matières ;***
- ***Représentation d'un arbre généalogique ;***
- ***…***

Arbres (Trees)

Terminologie

Terminologie (1)

- Un arbre est un ensemble d'éléments appelés ***nœuds*** (***ou sommets***), liés par une relation (*dite de "parenté"*) induisant une structure hiérarchique parmi ces nœuds.

- Un nœud, comme tout élément d'une liste, peut être de n'importe quel type.

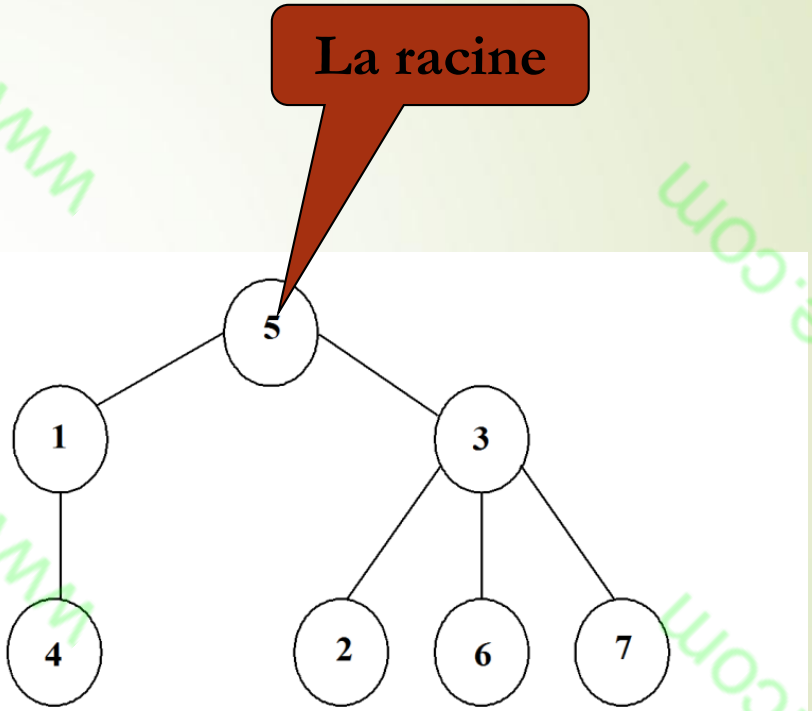
Terminologie (1) (suite)

- D'une manière plus formelle, une structure d'arbre de type de base T est :
  - soit la ***structure vide*** ;
  - soit un noeud de type T, appelé ***racine***, associé à un nombre fini de structures d'arbre disjointes du type de base T appelées ***sous arbres***
- C'est une définition récursive ; la récursivité est une propriété des arbres et des algorithmes qui les manipulent
- **Une liste** est un cas particulier des arbres (***arbre dégénéré***), où tout noeud a au plus un sous arbre

Illustration & Exemple

- Pour illustrer une structure d'arbre, on modélise le plus souvent un nœud par une information inscrite dans un cercle et les liens par des traits.

- ***Par convention***, on dessine les arbres avec la racine en haut et les branches dirigées vers le bas.



Exemple d'arbre formé de 7 nœuds (des entiers)

Terminologie (2)

- **La terminologie utilisée dans les structures d'arbres est empruntée :**
  - aux arbres généalogiques :
    - *Père ;*
    - *Fils ;*
    - *Frère ;*
    - *Descendant ;*
    - ...
  - et à la botanique :
    - *Feuille ;*
    - *Branche ;*
    - ...

Terminologie (3)

- ***Fils (ou enfants) :***
  - Chaque nœud d'un arbre pointe vers un ensemble éventuellement vide d'autres nœuds ; ce sont ses fils (*ses enfants*).
  - Sur l'exemple précédent, le nœud 5 a deux fils : 1 et 3, le nœud 1 a un fils : 4, et le nœud 3 a trois fils : 2, 6 et 7.
- ***Père :***
  - Tous les nœuds d'un arbre, sauf un, ont un père et un seul. Un nœud p est père du nœud n si et seulement si n est fils de p.
  - Par exemple, le père de 2 est 3, celui de 3 et 5.
- ***Frères :***
  - Deux nœuds ayant le même père.
  - Les nœuds 2, 6 et 7 sont des frères.
- ***Racine :***
  - Le seul nœud sans père.
  - 5 est la racine de l'arbre précédent.

Terminologie (4)

- ***Feuilles (ou nœuds terminaux, ou nœuds externes) :***
  - Ce sont des noeuds sans fils.
  - Par exemple, 4, 2, 6 et 7.
- ***Nœud interne :***
  - Un noeud qui n'est pas terminal.
  - Par exemple, 1, 3 et 5.
- ***Degré d'un noeud :***
  - Le nombre de fils de ce noeud.
  - Sur l'exemple, 5 est de degré deux, 1 est de degré un, 3 est de degré trois et les feuilles (4, 2, 6, 7) sont de degré nul.
- ***Degré d'un arbre (ou arité) :***
  - Plus grand degré des nœuds de l'arbre. Un arbre de degré n est dit *n-aire*
  - Sur l'exemple, l'arbre est un arbre 3-aire.

Terminologie (5)

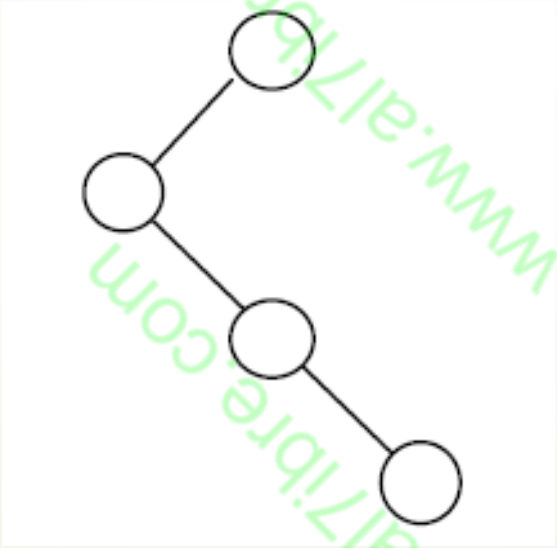
- *Taille d'un arbre :*
  - Le nombre total des nœuds de l'arbre.
  - Sur l'exemple, l'arbre est de taille 7.
- *Chemin :*
  - Une suite de noeuds d'un arbre $(n_1, n_2, ..., n_k)$ tel que n_i = père(n_i+1) pour $1 \leq i \leq k-1$ est appelée chemin entre le nœud n_1 et le nœud n_k.
  - La longueur d'un chemin est égale au nombre de nœuds qu'il contient moins 1.
  - Sur l'exemple, le chemin qui mène du nœud 5 au nœud 6 est de longueur 2.
- *Branche :*
  - Un chemin qui commence à la racine et se termine à une feuille.
  - Par exemple, les chemins (5, 1, 4), (5, 3, 2), (5, 3, 6) et (5, 3, 7).
- *Ancêtre :*
  - Un nœud A est un ancêtre d'un nœud B s'il existe un chemin de A vers B.
  - Par exemple, les ancêtres de 2 sont 2, 3 et 5
- *Descendant :*
  - Un nœud A est un descendant d'un nœud B s'il existe un chemin de B vers A.
  - Sur l'exemple, 5 admet les 7 nœuds de l'arbre comme descendants.

Terminologie (6)

- *Sous arbre :*
  - Un sous arbre d'un arbre A est constitué de tous les descendants d'un nœud quelconque de A.
  - Les ensembles de noeuds {3, 2, 6, 7} et {2} forment deux sous arbres de l'exemple précédent.
- *Hauteur (ou profondeur, ou niveau) d'un noeud :*
  - Longueur du chemin qui relie la racine à ce nœud.
  - La racine est elle même de hauteur 0, ses fils sont de hauteur 1, et les autres noeuds de hauteur supérieure à 1.
- *Hauteur d'un arbre :*
  - Plus grande profondeur des nœuds de l'arbre supposé non vide, c'est-à-dire $h(A) = Max\{h(x) ; x \text{ noeud de } A\}$
  - L'arbre de l'exemple est de profondeur 2.
  - *Par convention*, un arbre vide a une hauteur de -1.

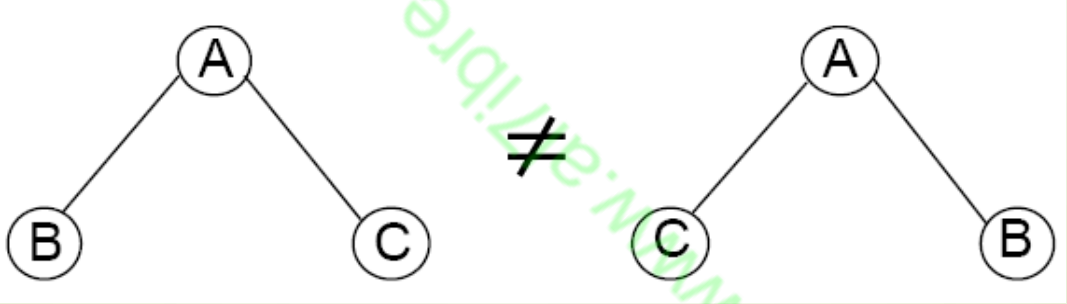
Terminologie (7)

- ***Arbre dégénéré ou filiforme :***
  - **Un arbre dont chaque nœud a au plus au fils**



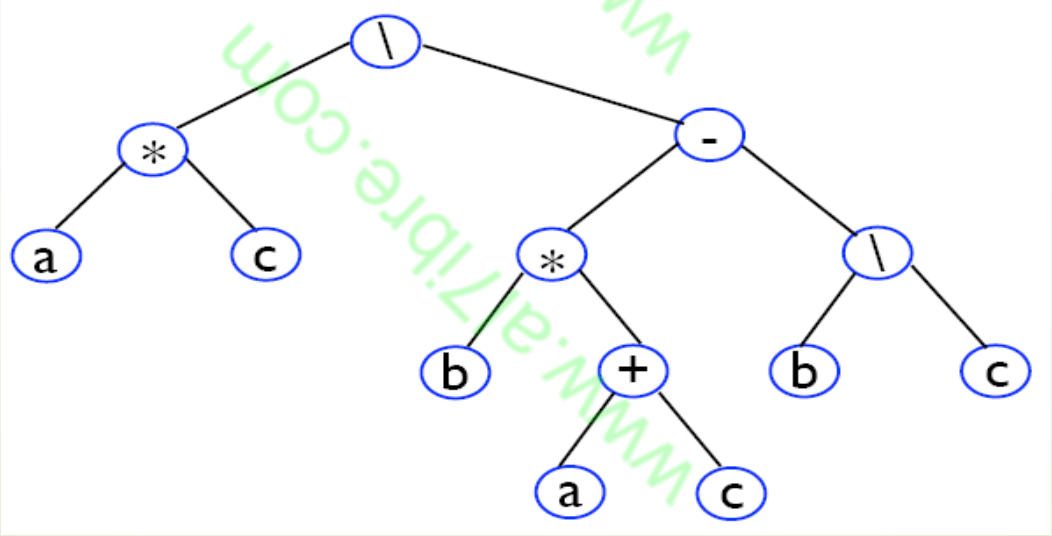
Terminologie (7)

- ***Arbre ordonné :***
  - **Un arbre où la position respective des sous arbres reflète une relation d'ordre. En d'autres termes, si un nœud a k fils, il existe un 1er fis, un 2ème fils, …, et un kème fils.**
  - **Les deux arbres de la figure qui suit sont différents si on les regarde comme des arbres ordonnés, mais identiques si on les regarde comme de simples arbres.**



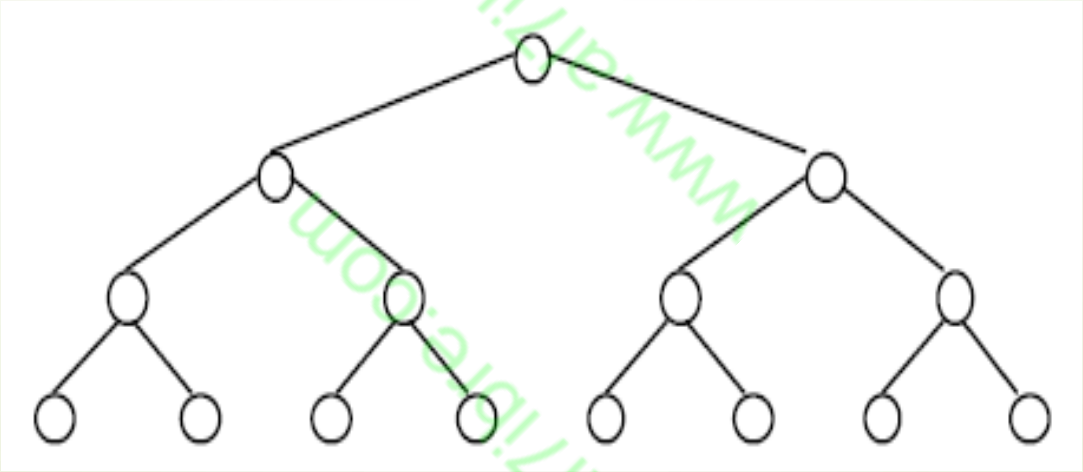
Terminologie (8)

- ***Arbre binaire :***
  - **Un arbre où chaque noeud a au plus deux fils.**
  - **Quand un nœud de cet arbre a un seul fils, on précise s'il s'agit du *fils gauche* ou du *fils droit*.**
  - **La figure qui suit montre un exemple d'arbre binaire dans lequel les nœuds contiennent des caractères.**



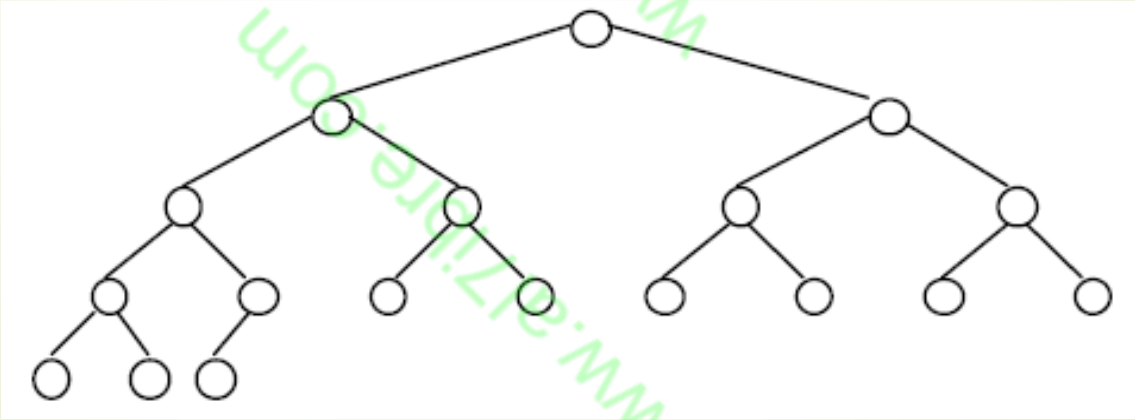
Terminologie (9)

- ***Arbre binaire complet :***
  - **Arbre binaire dont chaque niveau est rempli.**



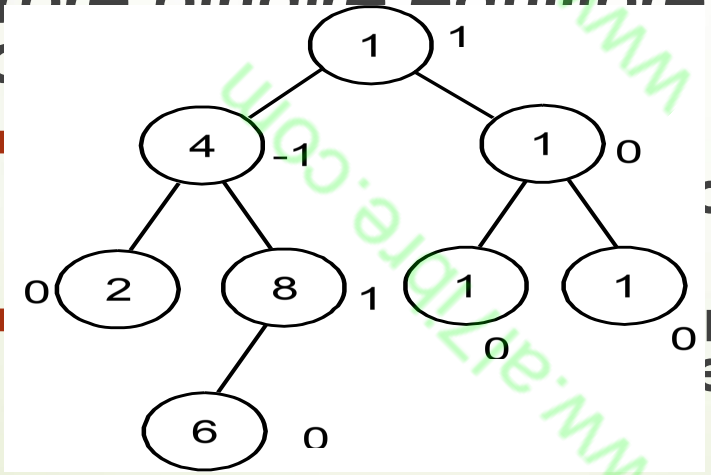
Terminologie (10)

- ***Arbre binaire parfait (ou presque complet) :***
  - **Arbre binaire dont chaque niveau est rempli sauf éventuellement le dernier**
  - **Dans ce cas les nœuds terminaux (*feuilles*) sont groupés le plus à gauche possible.**



Terminologie (11)

- ***Facteur d'équilibre d'un nœud d'un arbre binaire :***
  - **Hauteur du sous arbre partant du fils gauche du nœud moins la hauteur du sous arbre partant de son fils droit.**
- ***Arbre binaire équilibré (au sens des ha***
  - **pour chaque nœud, cteur d'équilibre est**
  - **n place à côté de eur d'équilibre.**

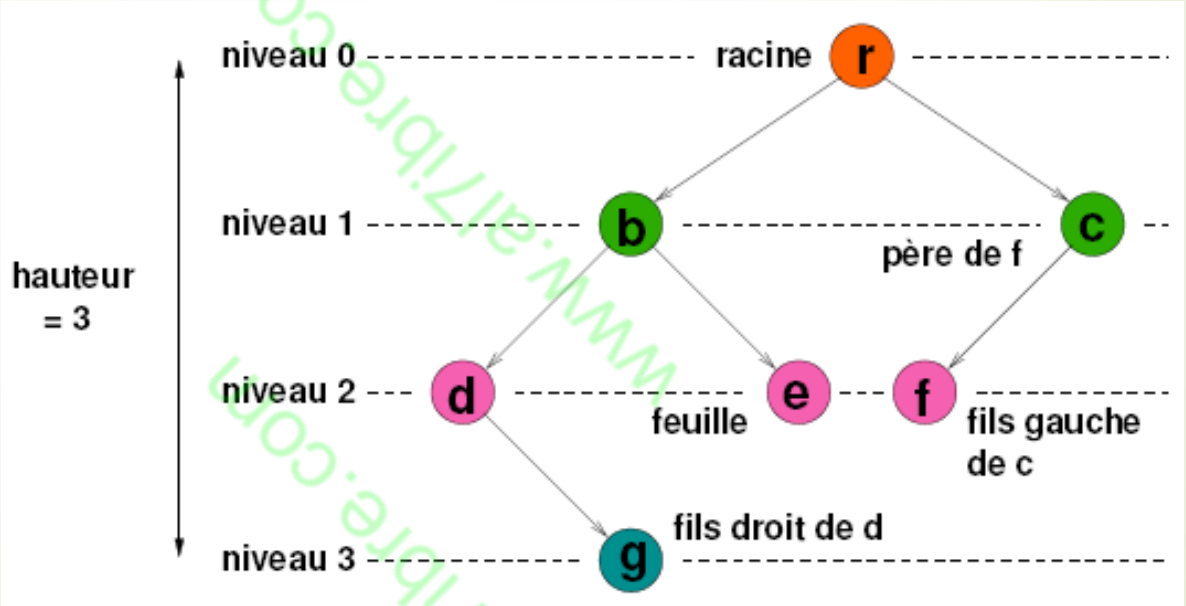


Arbres Binaires (Binary Trees)

Définition

- **Un arbre binaire A** est :
  - soit **vide** (**A = ( )** ou **A = ø**),
  - soit **de la forme A = <r, A1, A2>**, c-à-d composé :
    - d'un nœud **r** appelé ***racine*** contenant un élément
    - et de deux arbres binaires disjoints **A1** et **A2**, appelés respectivement ***sous arbre gauche*** (*ou* ***fils gauche***) et ***sous arbre droit*** (*ou* ***fils droit***).

Exemple d'arbre binaire



Type Abstrait Arbre_Binaire

```
Type Arbre_Binaire
Utilise Noeud, Elément, Booléen
Opérations
  arbre_vide : → Arbre_Binaire
  est_vide   : Arbre_Binaire → Booléen
  cons       : Noeud x Arbre_Binaire x Arbre_Binaire → Arbre_Binaire
  racine     : Arbre_Binaire → Noeud
  gauche     : Arbre_Binaire → Arbre_Binaire
  droite     : Arbre_Binaire → Arbre_Binaire
  contenu    : Noeud → Elément
Préconditions
  racine(A) est-défini-ssi est_vide(A) = faux
  gauche(A) est-défini-ssi est_vide(A) = faux
  droite(A) est-défini-ssi est_vide(A) = faux
Axiomes
  Soit, r : Nœud, A1, A2 : Arbre_Binaire
  racine(<r, A1, A2>) = r
  gauche(<r, A1, A2>) = A1
  droite(<r, A1, A2>) = A2
```

Opérations sur un Arbre Binaire (1)

- **`arbre_vide : → Arbre_Binaire`**
  - opération d'initialisation; crée un arbre binaire vide.
- **`est_vide : Arbre_Binaire → Booléen`**
  - teste si un arbre binaire est vide ou non.
- **`cons : Noeud x Arbre_Binaire x Arbre_Binaire → Arbre_Binaire`**
  - *cons(r,G,D)* construit un arbre binaire dont le sous arbre gauche est *G* et le sous arbre droit est *D*, et *r* est le nœud racine qui contient une donnée de type Elément.
- **`racine : Arbre_Binaire → Noeud`**
  - si `A` est un arbre binaire non vide alors `racine(A)` retourne le nœud racine de `A`, sinon un message d'erreur.

Opérations sur un Arbre Binaire (2)

- **`gauche : Arbre_Binaire → Arbre_Binaire`**
  - si `A` est un arbre binaire non vide alors `gauche(A)` retourne le sous arbre gauche de `A`, sinon un message d'erreur.
- **`droite : Arbre_Binaire → Arbre_Binaire`**
  - si `A` est un arbre binaire non vide alors `droite(A)` retourne le sous arbre droit de `A`, sinon un message d'erreur.
- **`contenu : Noeud → Elément`**
  - permet d'associer à chaque noeud d'un arbre binaire une information de type Elément.

Opérations Auxiliaires

```
Extension Type Arbre_Binaire
Utilise Entier, Booléen
Opérations
  taille   : Arbre_Binaire → Entier
  hauteur  : Arbre_Binaire → Entier
  feuille  : Arbre_Binaire → Booléen
Préconditions

Axiomes
  Soit, r : Noeud, A1, A2 : Arbre_Binaire

  taille(arbre_vide) = 0
  taille(<r, A1, A2>) = 1 + taille(A1) + taille(A2)
  hauteur(arbre_vide) = -1
  si hauteur(A1) > hauteur(A2) alors hauteur(<r, A1, A2>) = 1+hauteur(A1)
  sinon hauteur(<r, A1, A2>) = 1 + hauteur(A2)
  si est_vide(A) = faux et est_vide(gauche(A)) = vrai
   et est_vide(droit(A)) = vrai
  alors feuille(A) = vrai
  sinon feuille(A) = faux
```

Parcours d'arbre binaire

- **Un parcours d'arbre permet d'accéder à chaque nœud de l'arbre :**
  - **Un traitement (*test, affichage, comptage, etc.*), dépendant de l'application considérée, est effectué sur l'information portée par chaque nœud**
  - **Chaque parcours de l'arbre définit un ordre sur les nœuds**
- **On distingue :**
  - ***Les parcours de gauche à droite (le fils gauche d'un nœud précède le fils droit) ;***
  - ***Les parcours de droite à gauche (le fils droit d'un nœud précède le fils gauche).***
- **On ne considèrera que les parcours de gauche à droite**
- **On distingue aussi deux catégories de parcours d'arbres :**
  - ***Les parcours en profondeur ;***
  - ***Les parcours en largeur.***

Parcours en profondeur

- **Soit un arbre binaire A = <r, A1, A2>**

- **On définit trois parcours en profondeur de cet arbre :**
  - ***Le parcours préfixe ;***
  - ***Le parcours infixe ou symétrique ;***
  - ***Le parcours postfixe ou suffixe.***

Parcours en profondeur
Parcours préfixe

- **En abrégé *RGD* (*Racine, Gauche, Droit*)**
- **Consiste à effectuer dans l'ordre :**
  - ***Le traitement de la racine r ;***
  - ***Le parcours préfixe du sous arbre gauche A1 ;***
  - ***Le parcours préfixe du sous arbre droit A2.***
- ***L'ordre correspondant s'appelle l'ordre préfixe***

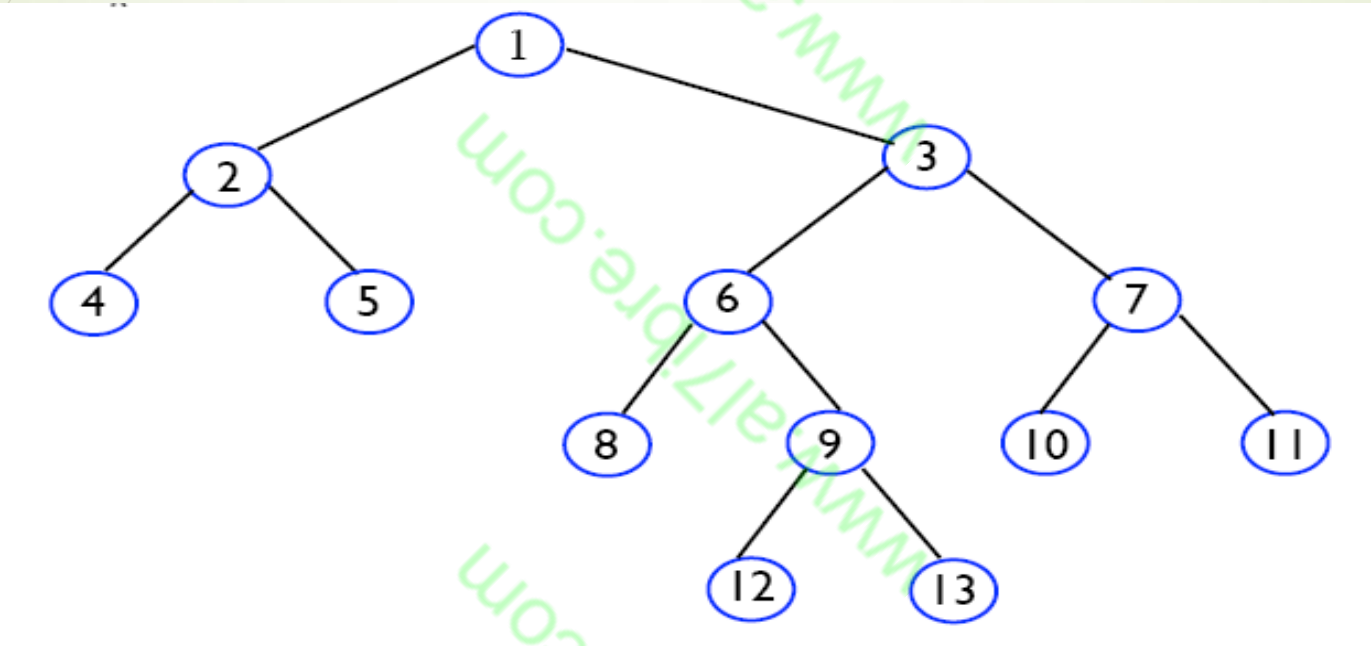
Parcours en profondeur
Parcours infixe ou symétrique

- **En abrégé *GRD* (*Gauche, Racine, Droit*)**
- **Consiste à effectuer dans l'ordre :**
  - ***Le parcours infixe du sous arbre gauche A1 ;***
  - ***Le traitement de la racine r ;***
  - ***Le parcours infixe du sous arbre droit A2.***
- ***L'ordre correspondant s'appelle l'ordre infixe***

Parcours en profondeur parcours postfixe ou suffixe

- **En abrégé *GDR* (*Gauche, Droit, Racine*)**
- **Consiste à effectuer dans l'ordre :**
  - ***Le parcours postfixe du sous arbre gauche A1 ;***
  - ***Le parcours postfixe du sous arbre droit A2 ;***
  - ***Le traitement de la racine r.***
- ***L'ordre correspondant s'appelle l'ordre suffixe***

Exemple de Parcours en profondeur (affichage du contenu des nœuds)



Le parcours préfixe affiche les nœuds dans l'ordre : ***1, 2, 4, 5, 3, 6, 8, 9, 12, 13, 7, 10, 11***
Le parcours infixe affiche les nœuds dans l'ordre : ***4, 2, 5, 1, 8, 6, 12, 9, 13, 3, 10, 7, 11***
Le parcours postfixe affiche les nœuds dans l'ordre : ***4, 5, 2, 8, 12, 13, 9, 6, 10, 11, 7, 3, 1***

Parcours en largeur

- **On explore les noeuds :**
  - ***niveau par niveau,***
  - ***de gauche à droite,***
  - ***en commençant par la racine.***

- **Exemple :**
  - **Le parcours en largeur de l'arbre de la figure précédente affiche la séquence d'entiers suivante : *1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13***

Représentations d'un arbre binaire

- **Représentation par tableau (*par contiguïté*)**
- **Représentation par pointeurs (*par chaînage*)**

Représentation contiguë d'un arbre binaire

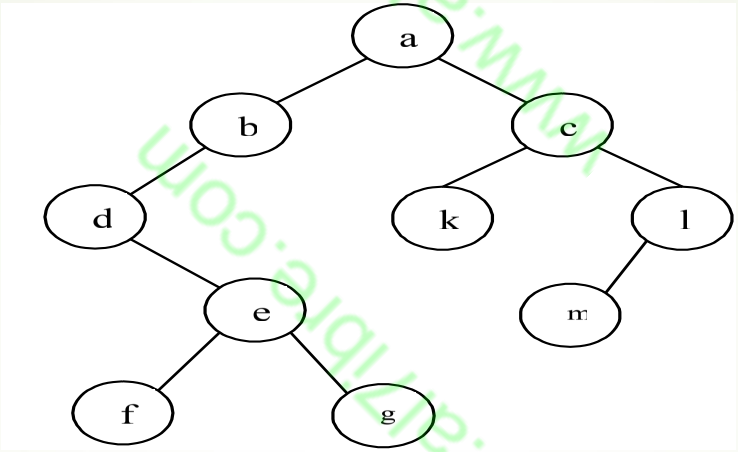
- **On caractérise un arbre binaire par :**
  - sa taille (*nombre de nœuds*) ;
  - sa racine (*indice de son emplacement dans le tableau de nœuds*)
  - un tableau de nœuds.

- **Chaque nœud contient trois données :**
  - une information de type Elément ;
  - deux entiers (*indices dans le tableau désignant respectivement l'emplacement des fils gauche et droit du nœud*).

Représentation contiguë d'un arbre binaire

```
#define NB_MAX_NOEUDS 15
typedef int Element;
typedef struct noeud {
 Element val;
 int fg;
 int fd;
} Noeud;
typedef Noeud TabN[Nb_MAX_NOEUDS];
typedef struct arbre {
 int nb_noeuds;
 int racine;
 TabN les_noeuds;
} Arbre_Binaire
```

Exemple de Représentation contiguë



nb_noeuds	racine
10	2

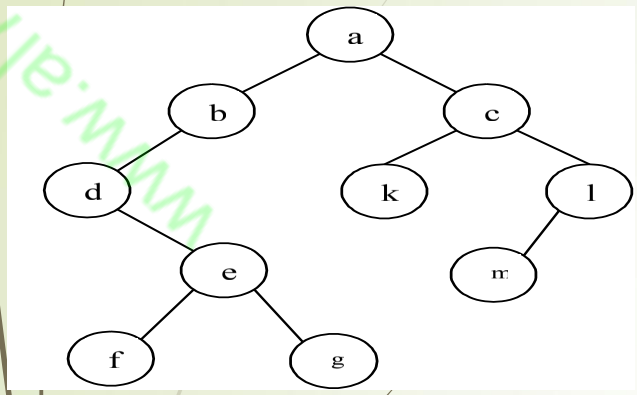
les_noeuds

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
val		d	a	g	b	c		f	m	e	l		k		
fg		-1	4	-1	1	12		-1	-1	7	8		-1		
fd		9	5	-1	-1	10		-1	-1	3	-1		-1		

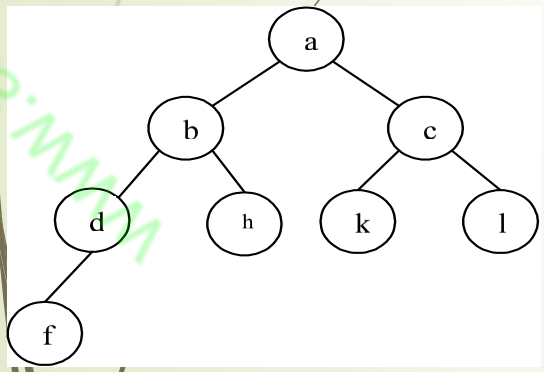
Autre représentation contiguë d'un arbre binaire

- Repose sur l'ordre hiérarchique (*numérotation des nœuds niveau par niveau et de gauche à droite*)
- On rappelle que pour stocker un arbre binaire de hauteur h, il faut un tableau de $2^{h+1}-1$ éléments
- On organise le tableau de la façon suivante :
  - Le noeud racine a pour indice 0 (*en langage C*) ;
  - Soit le noeud d'indice i dans le tableau, son fils gauche a pour indice 2i +1, et son fils droit a pour indice 2(i+1).
- Représentation idéale pour les arbres binaires parfaits. En effet, elle ne gaspille pas d'espace.

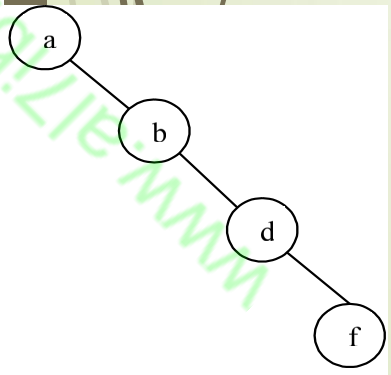
Autre représentation contiguë d'un arbre binaire (Exemples)



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
a	b	c	d		k	l		e					m				f	g



0	1	2	3	4	5	6	7
a	b	c	d	h	k	l	f



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a		b				d								f

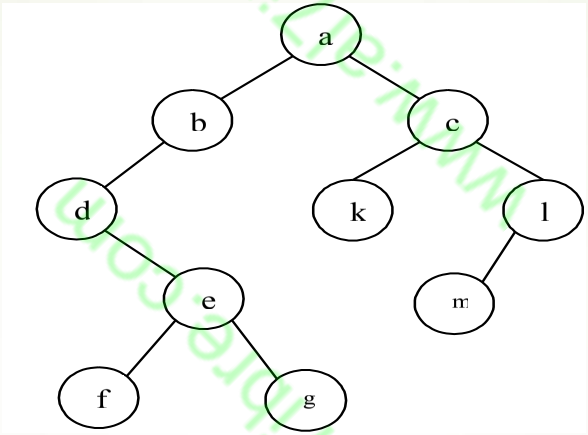
Représentation chaînée d'un arbre binaire

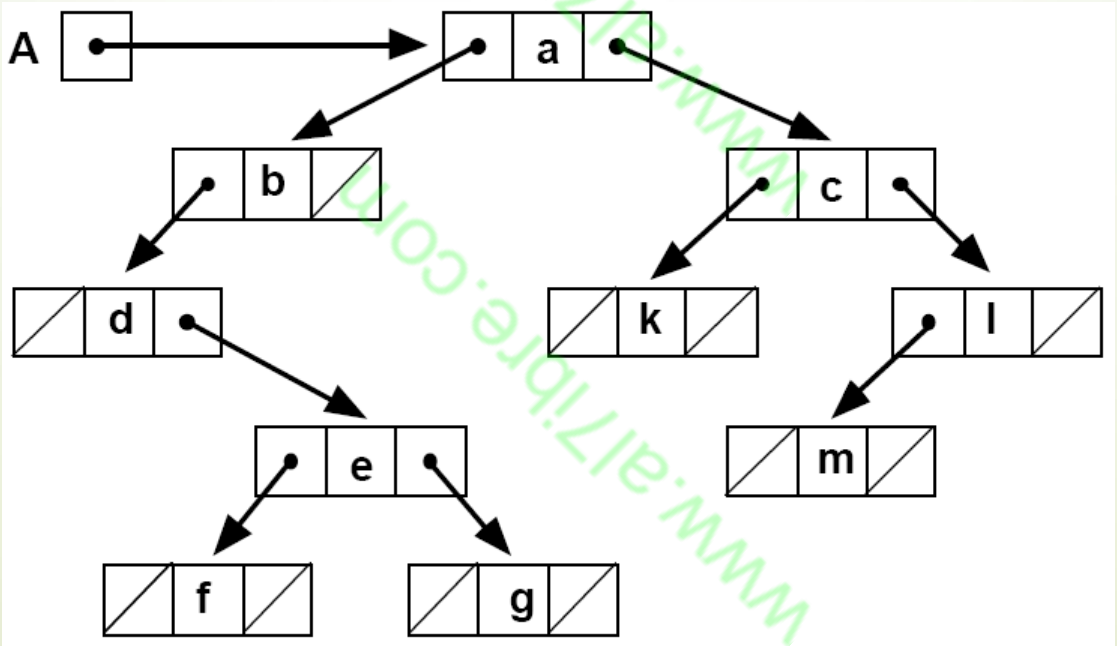
- **Chaque nœud a trois champs :**
  - **val (*l'élément stocké dans le noeud*) ;**
  - ***fg* (*pointeur sur fils gauche*) ;**
  - ***fd* (*pointeur sur fils droit*).**
- **Un arbre est désigné par un pointeur sur sa racine**
- **Un arbre vide est représenté par le pointeur NULL**

Représentation chaînée en C d'un arbre binaire


```
typedef int Element;
typedef struct noeud *Pnoeud;
typedef struct noeud {
 Element val;
 Pnoeud fg;
 Pnoeud fd;
} Noeud;
typedef Noeud *Arbre_Binaire;
```

Exemple de Représentation chaînée d'un arbre binaire





Réalisation chaînée d'un arbre binaire


```
Arbre_Binaire arbre_vide() {
 return NULL;
}

Booleen est_vide(Arbre_Binaire A) {
 return A == NULL ;
}

Pnoeud nouveau_noeud(Element e) {
// faire une allocation mémoire et placer l'élément e
// en cas d'erreur d'allocation, le pointeur renvoyé est
   NULL
 Pnoeud p = (Pnoeud) malloc(sizeof(Noeud));
 if (p != NULL) {
  p->val = e;
  p->fg = NULL;
  p->fd = NULL;
 }
 return (p);
}
```

Réalisation chaînée d'un arbre binaire


```
Arbre_Binaire cons(Noeud *r,
                   Arbre_Binaire G,
                       Arbre_Binaire D) {
 r->fg = G ;
 r->fd = D ;
 return r ;
}


Noeud racine(Arbre_Binaire A) {
// précondition : A est non vide !
 if (estvide(A)) {
  printf("Erreur : Arbre vide !\n");
  exit(-1);
 }
 return (*A) ;
}
```

```
Arbre_Binaire gauche(Arbre_Binaire A) {
// précondition : A est non vide !
 if (estvide(A)) {
  printf("Erreur : Arbre vide !\n");
  exit(-1);
 }
 return A->fg ; /* ou bien (*A).fg; */
}

Arbre_Binaire droite(Arbre_Binaire A) {
// précondition : A est non vide !
 if (estvide(A)) {
  printf("Erreur : Arbre vide !\n");
  exit(-1);
 }
 return A->fd ; /* ou bien (*A).fd; */
}

Element contenu(Noeud n) {
 return n.val;
}
```

Exemples d'Applications d'Arbre Binaire

- ***Recherche dans un ensemble de valeurs*** :
  - Les arbres binaires de recherche ;
- ***Tri d'un ensemble de valeurs :***
  - Le parcours GRD d'un arbre binaire de recherche ;
  - Un algorithme de tri efficace utilisant une structure de *tas ;*
- ***Représentation d'une expression arithmétique*** :
  - Un parcours GDR pour avoir une notation postfixée ;
- ***Méthodes de compression :***
  - Le codage de *Huffman u*tilisant des arbres binaires ;
  - La compression d'images utilisant des *quadtrees (arbres quaternaires, ou chaque nœud non feuille a exactement quatre fils) ;*
- …

Arbres de Recherche Equilibrés Exemples (3)

- ***Les B arbres :***
  - *Arbres de recherche équilibrés qui sont conçus pour être efficaces sur d'énormes masses de données stockées sur mémoires secondaires ;*
  - *Chaque nœud permet de stocker plusieurs clés ;*
  - *Généralement, la taille d'un nœud est optimisée pour coïncider avec la taille d'un bloc (ou page) du périphérique, en vue d'économiser les coûteux accès d'entées sorties.*
- ...

Arbres Binaires de Recherche (Binary Search Trees)

Pr F.Omary

2019-2020

Notion d'Arbre binaire de recherche

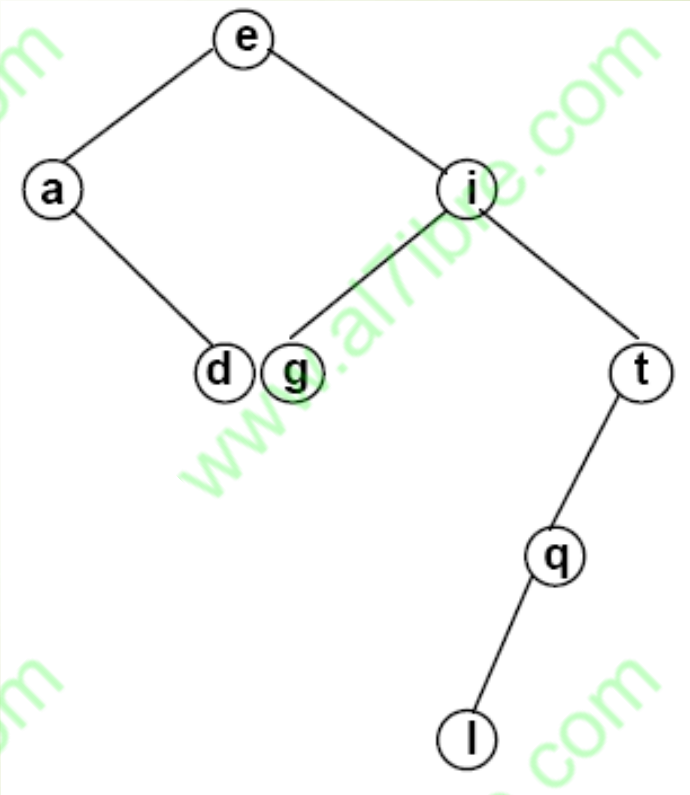
- **C'est un arbre binaire particulier :**
  - ***Permet d'obtenir un algorithme de recherche proche dans l'esprit de la recherche dichotomique ;***
  - ***Pour lequel les opérations d'ajout et de suppression d'un élément sont aussi efficaces.***
- **Cet arbre utilise l'existence d'une relation d'ordre sur les éléments, représentée par une fonction clé, à valeur entière.**

Arbre binaire de recherche Définition

- **Un arbre binaire de recherche (*binary search tree* en anglais), en abrégé *ABR*, est un arbre binaire tel que pour tout nœud :**
  - ***les clés de tous les noeuds du sous-arbre gauche sont inférieures ou égales à la clé du nœud,***
  - ***les clés de tous les noeuds du sous-arbre droit sont supérieures à la clé du nœud.***
- ***Chaque nœud d'un arbre binaire de recherche désigne un élément qui est caractérisé par une clé (prise dans un ensemble totalement ordonné) et des informations associées à cette clé.***
- ***Dans toute illustration d'un arbre binaire de recherche, seules les clés sont représentées. On supposera aussi que toute clé identifie de manière unique un élément.***

Arbre binaire de recherche Exemple

- **L'arbre de la figure suivante est un arbre binaire de recherche**



- **Cet arbre représente l'ensemble :**

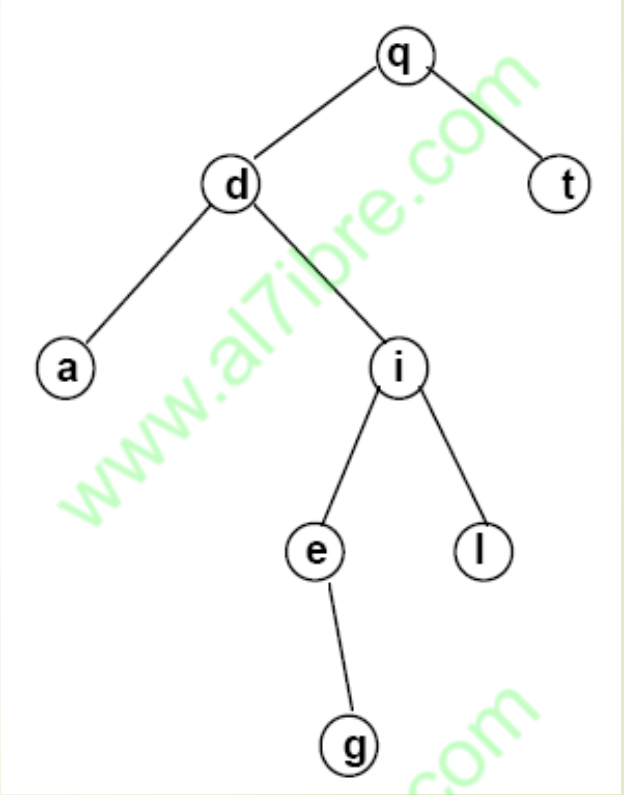
$$E = \{a, d, e, g, i, l, q, t\}$$

muni de l'ordre alphabétique

Arbre binaire de recherche Remarque

- **Plusieurs représentations possibles d'un même ensemble par un arbre binaire de recherche**
- **En effet, la structure précise de l'arbre binaire de recherche est déterminée :**
  - **par l'algorithme d'insertion utilisé,**
  - **et par l'ordre d'arrivée des éléments.**
- **Exemple :**
  - **L'arbre binaire de recherche de la figure qui suit représente aussi**

E = *{a, d, e, g, i, l, q, t}*



Opérations sur les arbres binaires de recherche

- **Le type abstrait arbre binaire de recherche, noté `Arbre_Rech`, est décrit de la même manière que le type `Arbre_Binaire`**

- **On reprend les opérations de base des arbres binaires, excepté le fait que dans des arbres binaires de recherche, on suppose l'existence de l'opération clé sur le type abstrait `Element`**

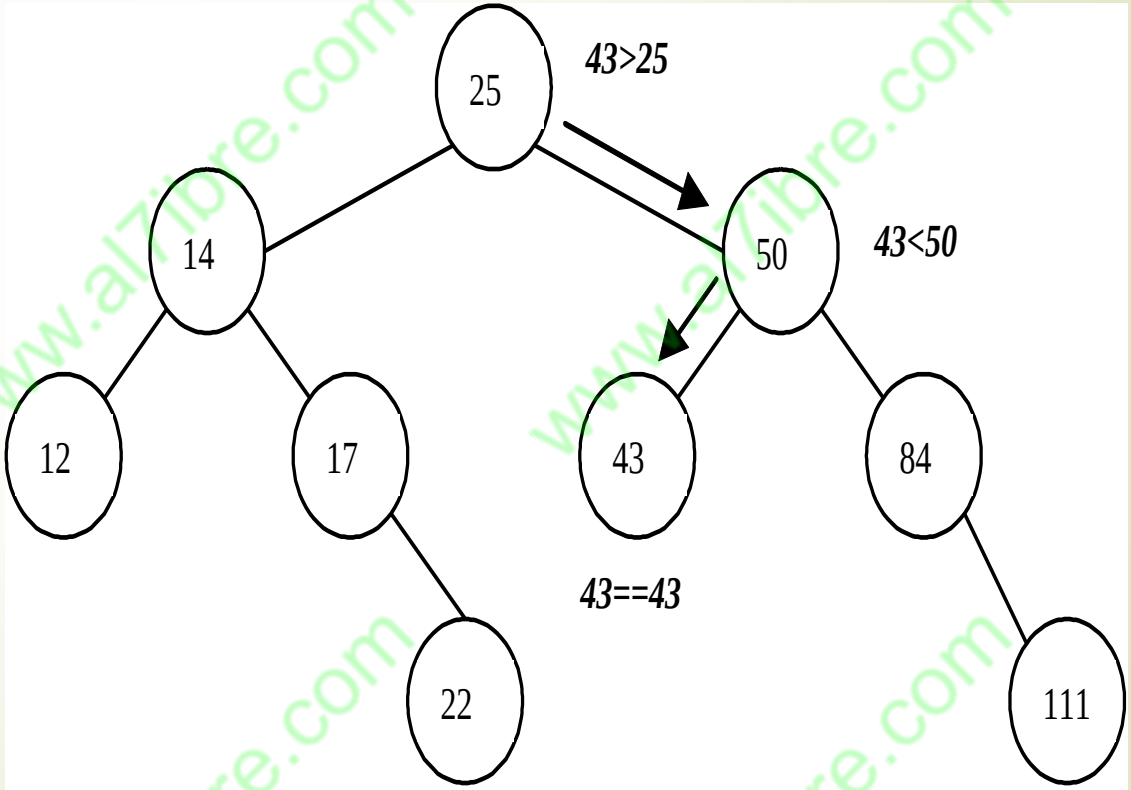
- **On définit, en tenant compte du critère d'ordre, les opérations spécifiques de ce type d'arbre concernant :**
  - ***la recherche d'un élément dans l'arbre ;***
  - ***l'insertion d'un élément dans l'arbre ;***
  - ***la suppression d'un élément de l'arbre.***

Recherche d'un élément

- **Principe de l'algorithme :**
  - On compare la clé de l'élément cherché à la clé de la racine de l'arbre ;
  - Si la clé est supérieure à la clé de la racine, on effectue une recherche dans le fils droit ;
  - Si la clé est inférieure à la clé de la racine, on effectue une recherche dans le fils gauche ;
  - La recherche s'arrête quand on ne peut plus continuer (*échec*) ou quand la clé de l'élément cherché est égale à la clé de la racine d'un sous arbre (*succès*).

Recherche d'un élément Exemple

- la figure suivante illustre la recherche de l'élément de clé 43 dans un arbre binaire de recherche.
- Les flèches indiquent le chemin de la recherche



Recherche d'un élément Spécification

```
Extension Type Arbre_Rech
Utilise Elément, Booléen
Opérations
  Rechercher : Elément x Arbre_Rech → Booléen
Axiomes
  Soit, x : Elément, r : Nœud, G, D : Arbre_Rech

  Rechercher(x, arbre_vide) = faux
  si clé(x) = clé(contenu(r))
   alors Rechercher(x, <r, G, D>) = vrai
  si clé(x) < clé(contenu(r))
   alors Rechercher(x, <r, G, D>) = Rechercher(x,
  G)
  si clé(x) > clé(contenu(r))
   alors Rechercher(x, <r, G, D>) = Rechercher(x,
  D)
```

Recherche d'un élément Réalisation en C

```
Booleen Rechercher (Arbre_Rech A, Element e) {
 if ( est_vide(A) == vrai )
   return faux; // e n'est pas dans l'arbre
 else {
   if ( e == A->val )
      return vrai; // e est dans l'arbre
   else if ( e < A->val )
           // on poursuit la recherche dans le SAG
   du
           // noeud courant
           return Rechercher(A->fg , e);
        else
           // on poursuit la recherche dans le SAD
   du
           // noeud courant
           return Rechercher(A->fd , e);
   }
}
```

Recherche d'un élément
Autre Spécification

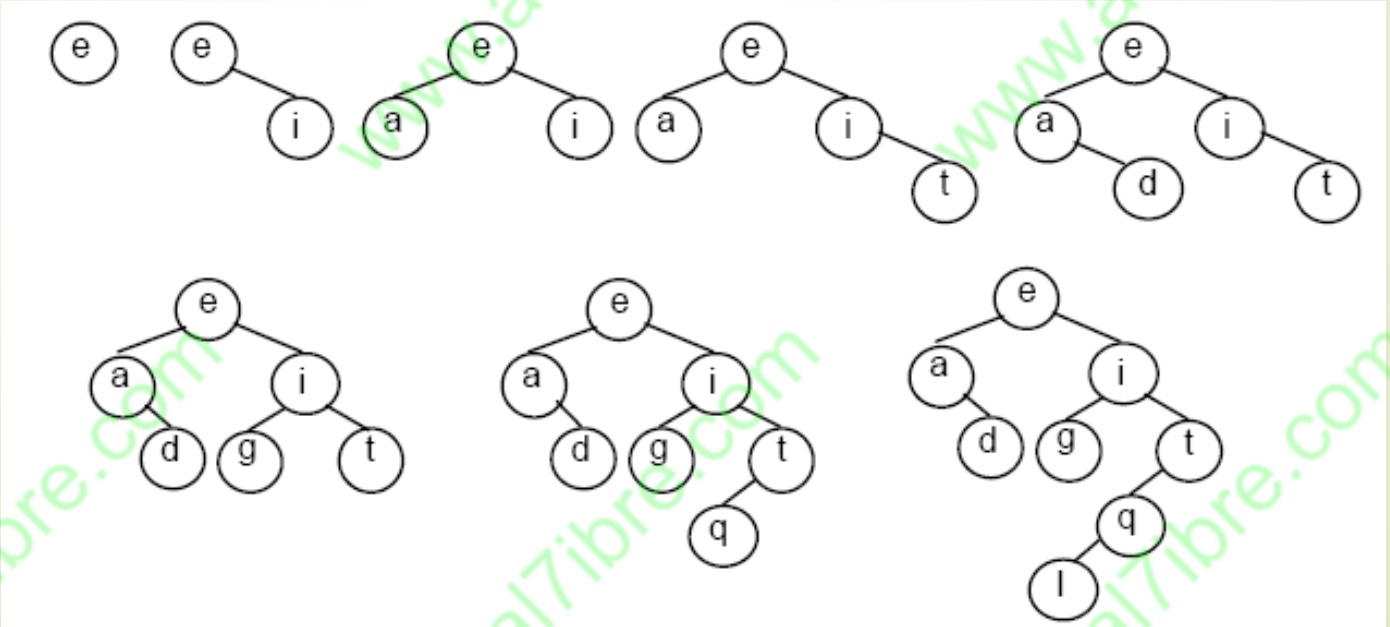
```
Extension Type Arbre_Rech
Utilise Elément
Opérations
  Rechercher : Elément x Arbre_Rech → Arbre_Rech
Axiomes
  Soit, x : Elément, r : Nœud, G, D : Arbre_Rech

  Rechercher(x, arbre_vide) = arbre_vide
  si clé(x) = clé(contenu(r))
   alors Rechercher(x, <r, G, D>) = <r, G, D>
  si clé(x) < clé(contenu(r))
   alors Rechercher(x, <r, G, D>) = Rechercher(x, G)
  si clé(x) > clé(contenu(r))
   alors Rechercher(x, <r, G, D>) = Rechercher(x, D)
```

Ajout d'un élément

- La technique d'ajout spécifiée ici est dite "*ajout en feuille*", car tout nouvel élément se voit placé sur une feuille de l'arbre
- Le principe est simple :
  - si l'arbre initial est vide, le résultat est formé d'un arbre binaire de recherche réduit à sa racine, celle-ci contenant le nouvel élément ;
  - sinon, l'ajout se fait (*récursivement*) dans le fils gauche ou le fils droit, suivant que l'élément à ajouter est de clé inférieure ou supérieure à celle de la racine.
- Remarque :
  - 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

Ajout d'un élément
Exemple

- **Les figures suivantes illustrent l'ajout successif de e, i, a, t, d, g, q et l dans un arbre binaire de recherche, initialement vide**



Ajout "en feuille" d'un élément Spécification

Extension Type Arbre_Rech

Utilise Elément

Opérations

Ajouter_feuille : Elément x Arbre_Rech → Arbre_Rech

Axiomes

Soit, x : Elément, r : Nœud, G, D : Arbre_Rech

Ajouter_feuille(x, arbre_vide) = <x, arbre_vide, arbre_vide>

si clé(x) ≤ clé(contenu(r))

alors

Ajouter_feuille(x, <r, G, D>) = <r, Ajouter_feuille(x, G), D>

sinon

Ajouter_feuille(x, <r, G, D>) = <r, G, Ajouter_feuille(x, D)>

Ajout "en feuille" d'un élément Réalisation

```
fonction Ajouter_feuille(x : Elément, A : Arbre_Rech ) :
   Arbre_Rech
 si est_vide(A) alors
   Pnoeud r = nouveau_noeud(x)
   si est_vide(r) alors <erreur>
   retourner cons(r, arbre_vide(), arbre_vide())
 sinon
  si x > contenu(racine(A)) alors
   retourner cons(A, gauche(A), Ajouter_feuille(x, droite(A)))
  sinon
   Si x< contenu(racine(A) alors
   retourner cons(A, Ajouter_feuille(x, gauche(A)) ,droite(A))
  fsi
 fsi
fsi
ffonction
```

Ajout "en feuille" d'un élément Réalisation en C

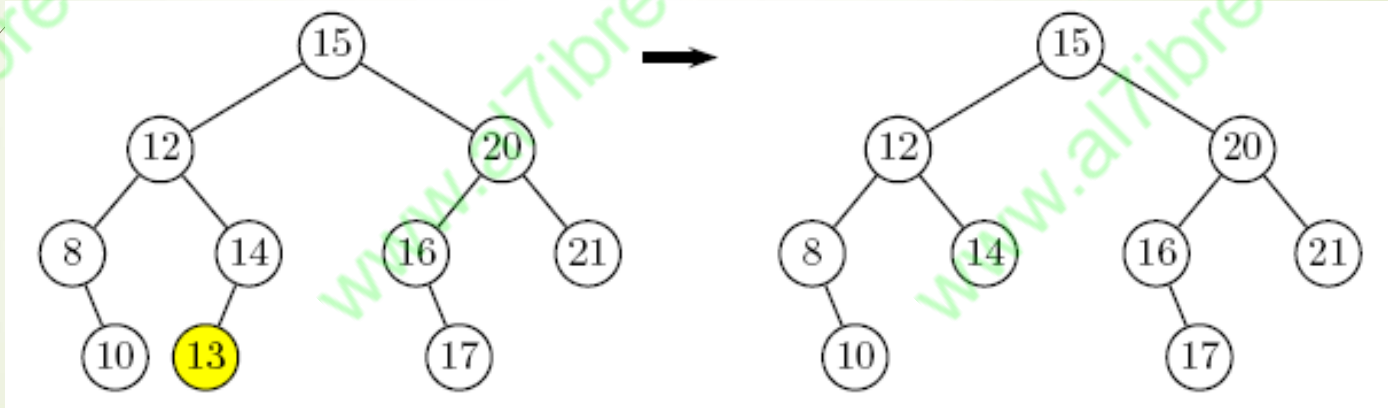
```
Arbre_Rech Ajouter_feuille(Element x, Arbre_Rech A) {
 if (est_vide(A)) {
   Pnoeud r = nouveau_noeud(x);
   if (r == NULL) {
     printf("Erreur : Pas assez de mémoire !\n");
     exit(-1);
   }
  return cons(r, arbre_vide(), arbre_vide());
 }
 else
   if (x > contenu(racine(A)))
     return cons(A, gauche(A), Ajouter_feuille(x, droite(A)));
   else
   if (x < contenu (racine(A))// pas d'ajout lorsque x=contenu(A)
     return cons(A, Ajouter_feuille(x, gauche(A)), droite(A));
}
```

Suppression d'un élément

- **La suppression est délicate :**
  - **Il faut réorganiser l'arbre pour qu'il vérifie la propriété d'un arbre binaire de recherche**
- **La suppression commence par la recherche du nœud qui porte l'élément à supprimer. Ensuite, il y a trois cas à considérer, selon le nombre de fils du noeud à supprimer :**
  - **si le noeud est sans fils (*une feuille*), la suppression est immédiate ;**
  - **si le noeud a un seul fils, on le remplace par ce fils ;**
  - **si le noeud a deux fils (*cas général*), on choisit de remplacer ce nœud, soit par le plus grand élément de son sous arbre gauche (*son prédécesseur*), soit par le plus petit élément de son sous arbre droit (*son successeur*).**

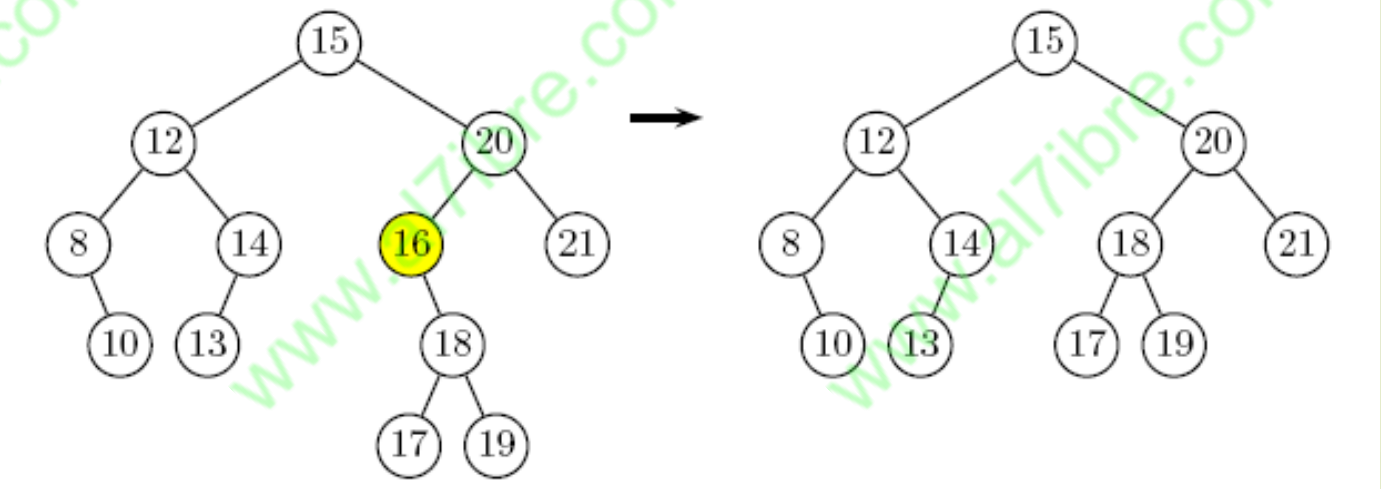
Suppression d'un élément
Exemple 1

- **La figure qui suit illustre la suppression de la feuille qui porte la clé 13**



Suppression d'un élément Exemple 2

- **La figure qui suit illustre la suppression du nœud qui porte la clé 16**

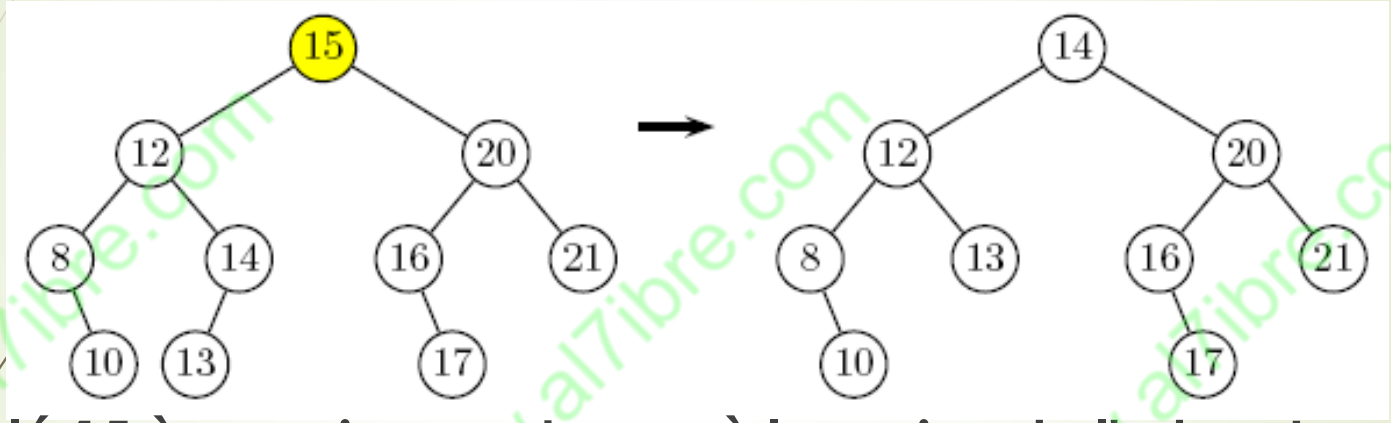


- **Ce nœud n'a qu'un seul fils ; le sous arbre de racine portant la clé 18**

- **Ce sous arbre devient fils gauche du nœud qui porte la clé 20**

Suppression d'un élément Exemple 3

- La figure qui suit illustre le cas d'un nœud à deux fils.



- La clé 15 à supprimer se trouve à la racine de l'arbre. La racine a deux fils ; on choisit de remplacer sa clé par la clé de son prédécesseur.
- Ainsi, la clé 14 est mise à la racine de l'arbre. On est alors ramené à la suppression du nœud du prédécesseur.
- Comme le prédécesseur est le nœud le plus à droite du sous arbre gauche, il n'a pas de fils droit, donc il a zéro ou un fils, et sa suppression est couverte par les deux premiers cas.

Suppression d'un élément
Cas général

- **On choisit ici de remplacer le noeud à supprimer par son prédécesseur (*le nœud le plus à droite de son sous arbre gauche*)**

- **On a besoin de deux opérations supplémentaires :**
  - ***une opération `Max` qui retourne l'élément de clé maximale dans un arbre binaire de recherche ;***
  - ***une opération `SupprimerMax` qui retourne l'arbre privé de son plus grand élément.***

Suppression d'un élément: Spécification

Extension Type Arbre_Rech
Utilise Elément
Opérations

Max : Arbre_Rech → Elément
SupprimerMax : Arbre_Rech → Arbre_Rech
Supprimer : Elément x Arbre_Rech → Arbre_Rech

Pré-conditions

Max(A) *est_défini_ssi* est_vide(A) = faux
SupprimerMax(A) *est_défini_ssi* est_vide(A) = faux

Axiomes

Soit, x : Elément, r : Nœud, G, D : Arbre_Rech
si est_vide(D) = vrai alors Max(<r, G, D>) = r
sinon Max(<r, G, D>) = Max(D)
si est_vide(D) = vrai alors SupprimerMax(<r, G, D>) = G
sinon SupprimerMax(<r, G, D>) = <r, G, SupprimerMax(D)>
Supprimer(x, arbre_vide) = arbre_vide
si clé(x) = clé(contenu(r)) et est_vide(D) = vrai
alors Supprimer(x, <r, G, D>) = G
sinon si clé(x) = clé(contenu(r)) et est_vide(G) = vrai
alors Supprimer(x, <r, arbre_vide, D>) = D
sinon si clé(x) = clé(contenu(r))
alors Supprimer(x, <r, G, D>) = <Max(G),SupprimerMax(G), D>
si clé(x) < clé(contenu(r))
alors Supprimer(x, <r, G, D>) = <r, Supprimer(x, G), D>
si clé(x) > clé(contenu(r))
alors Supprimer(x, <r, G, D>) = <r, G, Supprimer(x, D)>

Suppression d'un élément Réalisation

```
fonction Max(A : Arbre_Rech) : Pnoeud
(* A doit être non vide ! *)
 si est_vide(droite(A))
  alors retourner A


 sinon retourner Max(droite(A))
 fsi
ffonction

fonction SupprimerMax(A : Arbre_Rech) : Arbre_Rech
(* A doit être non vide ! *)
 si est_vide(droite(A))
 alors
  retourner gauche(A)
 sinon
  retourner cons(A, gauche(A), SupprimerMax(droite(A)))
 fsi
ffonction
```

Cette fonction retourne un pointeur sur le nœud contenant la plus grand élément d'un arbre binaire de recherche

Cette fonction supprime le plus grand élément d'un arbre binaire de recherche

Suppression d'un élément
Réalisation (suite)

```
fonction Supprimer(x : Elément, A : Arbre_Rech) : Arbre_Rech
 si est_vide(A) alors retourner A    (* ou <erreur> *)
 sinon
  si x > contenu(racine(A)) alors
   retourner cons(A, gauche(A), Supprimer(x ,droite(A)))
  sinon
   si x < contenu(racine(A)) alors
    retourner cons(A, Supprimer(x, gauche(A)), droite(A))
   sinon // x= contenu (racine(A))
    si est_vide(droite(A)) alors retourner gauche(A)
    sinon
     si est_vide(gauche(A)) alors retourner droite(A)
     sinon // ni droite (A) est vide ni gauche(A)
      retourner cons(Max(gauche(A)), SupprimerMax(gauche(A)), droite(A))
     fsi
    fsi
   fsi
  fsi
 fsi
ffonction
```

Arbre Binaire de Recherche Complexité des Opérations

- **On montre que, les opérations de recherche, insertion et suppression dans un arbre binaire de recherche contenant n éléments sont :**
  - en moyenne en $O(\log_2(n))$;
  - dans le pire des cas en O(h) ;

  où h désigne la hauteur de l'arbre

- **Si l'arbre est dégénéré, sa hauteur étant n-1, ces trois opérations sont en O(n)**

- **Si l'arbre est équilibré, les opérations sont en $O(\log_2(n))$ (*d'où leur intérêt...*)**

Arbres Maximiers ou Tas (Heaps)

Notion d'Arbre Maximier (ou Tas)

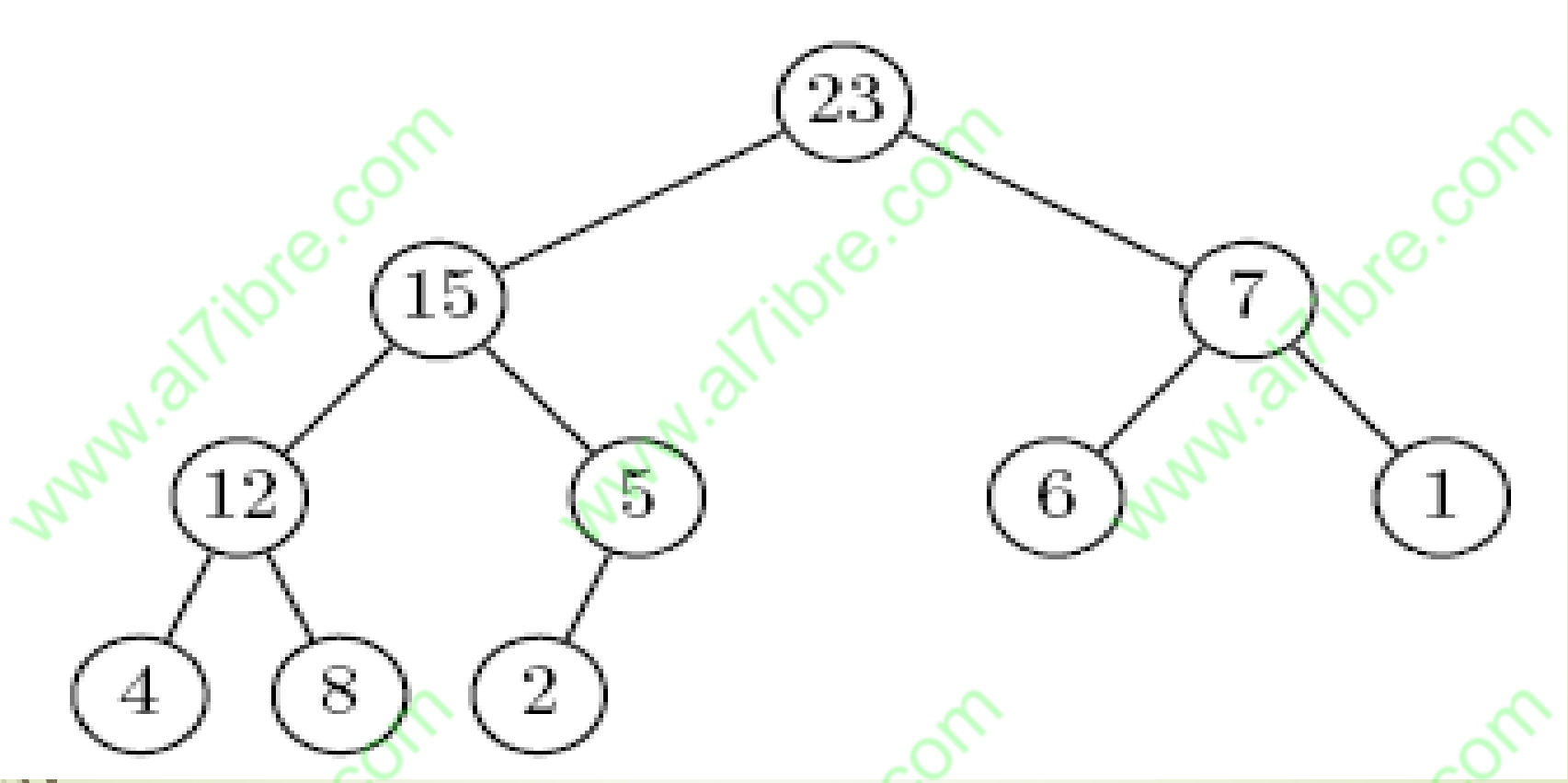
- Appelé aussi monceau (*Heap en anglais)*
- C'est un arbre binaire parfait tel que la clé de chaque noeud est supérieure ou égale aux clés de tous ses fils
- L'élément maximum de l'arbre se trouve donc à la racine

- Rappel :
  - Pour un arbre binaire parfait, tous les niveaux sont entièrement remplis sauf éventuellement le dernier et, dans ce cas, les feuilles du dernier niveau sont regroupées le plus à gauche possible

- Un tas est *un arbre binaire partiellement ordonné* :
  - Les nœuds sur chaque branche sont ordonnés sur celle-ci ;
  - Ceux d'un même niveau ne le sont pas nécessairement.

- Un tas dans lequel chaque nœud enfant a une clé inférieure (*resp., supérieure*) ou égale à la clé de son père est appelé *arbre maximier (max heap) (resp., arbre minimier (max heap))*

Arbre Maximier (ou Tas) Exemple



Type Abstrait Tas

Type Tas

Utilise Booléen, Elément

Opérations

tas_vide : → Tas
est_vide : Tas → Booléen
max : Tas → Elément
ajouter : Tas x Elément → Tas
supprimerMax : Tas → Tas
appartient : Tas x Elément → Booléen

Préconditions

max(T) *est_défini_ssi* est_vide(T) = faux
supprimerMax(T) *est_défini_ssi* est_vide(T) = faux
ajouter(T,e) *est_défini_ssi* appartient(T,e) = faux

Axiomes

Soit, T, T1 : Tas, e : Elément

si est_vide(T) = vrai alors appartient(T,e) = faux

appartient(T,max(T)) = vrai

si appartient(T,e) = vrai alors max(T) ≥ e

Opérations sur un Tas

- **tas_vide : → Tas**
  - Opération d'initialisation; crée un tas vide
- **est_vide : Tas → Booléen**
  - Vérifie si un tas est vide ou non
- **max : Tas → Elément**
  - Retourne le plus grand élément d'un tas
- **ajouter : Tas x Elément → Tas**
  - Ajoute un élément dans un tas
- **supprimerMax : Tas → Tas**
  - Supprime le plus grand élément d'un tas
- **appartient : Tas x Elément → Booléen**
  - Vérifie si un élément appartient ou non à un tas

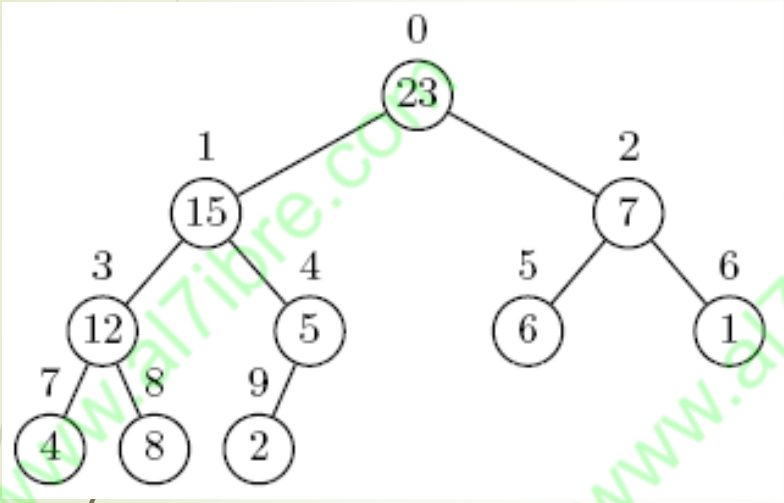
Représentation d'un Tas

- Il existe une représentation compacte pour les arbres binaires parfaits, et donc pour les tas :
  - La représentation par tableau, basée sur la numérotation des nœuds niveau par niveau et de gauche à droite
- Les numéros d'un nœud sont donc les indices dans un tableau. En outre, ce tableau s'organise de la façon suivante :
  - le noeud racine a pour indice 0 ;
  - soit le noeud d'indice i dans le tableau, son fils gauche a pour indice 2i +1, et son fils droit a pour indice 2(i+1) ;
  - si un nœud a un indice i ≠ 0, alors son père a pour indice

$$\lfloor (i-1)/2 \rfloor$$

- On déduit de cette organisation, où n désigne le nombre d'éléments du tas, que :
  - un nœud d'indice i est une feuille si 2i+1 ≥ n
  - un nœud d'indice i a un fis droit si 2(i+1) < n

Représentation d'un Tas
Exemple



0	1	2	3	4	5	6	7	8	9
23	15	7	12	5	6	1	4	8	2

Un tas avec sa numérotation hiérarchique

Représentation du tas par un tableau

Représentation en C d'un Tas

```
#define MAX_ELEMENTS 200  // taille maximum du tas
typedef int Element   // un élément est un int
typedef struct {
 int taille;  // nombre d'éléments dans le tas
 Element tableau[MAX]; // les éléments du tas
} Tas;
```

Opérations sur un Tas

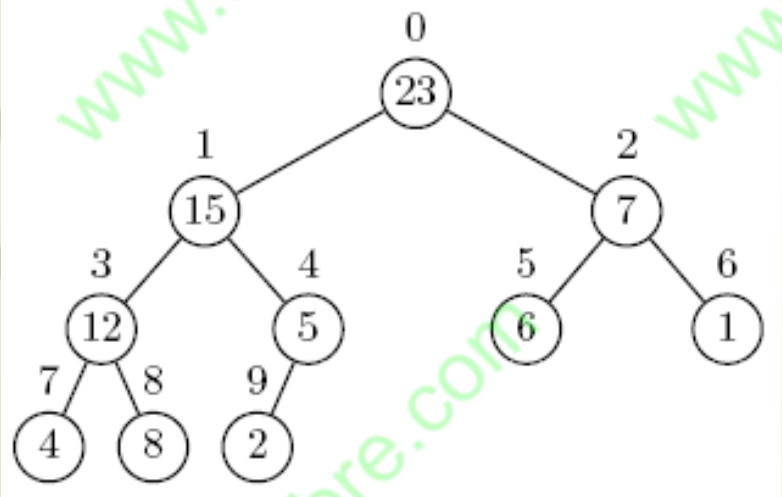
- **Trois opérations fondamentales :**
  - ***Ajout d'un élément ;***
  - ***Suppression du maximum ;***
  - ***Recherche du maximum.***

Opération d'Ajout

- ***Principe :***
  - **Créer un nouveau nœud contenant la clé du nouvel élément ;**
  - **Insérer cette clé le plus à gauche possible sur le dernier niveau du tas (*ou si le dernier niveau est plein, à l'extrême gauche d'un nouveau niveau*). La nouvelle clé est insérée dans la première case non utilisée du tableau ;**
  - **Faire "remonter cette nouvelle clé" à sa place en la permutant avec la clé de son père, tant qu'elle est plus grande que celle de son père.**

Opération d'Ajout Exemple (1)

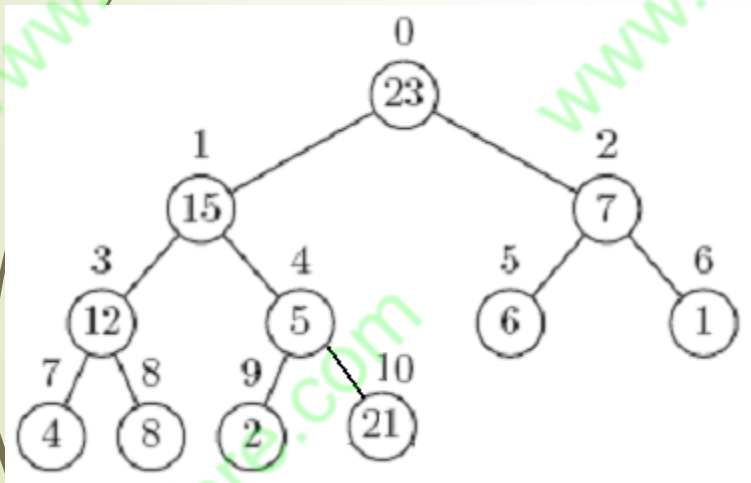
- **Supposons qu'on veuille insérer la valeur 21 dans le tas représenté ci-dessous :**
  - **On place la valeur 21 juste à droite de la dernière feuille,**
  - **c'est-à-dire dans la case d'indice 10 dans le tableau.**



0	1	2	3	4	5	6	7	8	9
23	15	7	12	5	6	1	4	8	2

Opération d'Ajout Exemple (2)

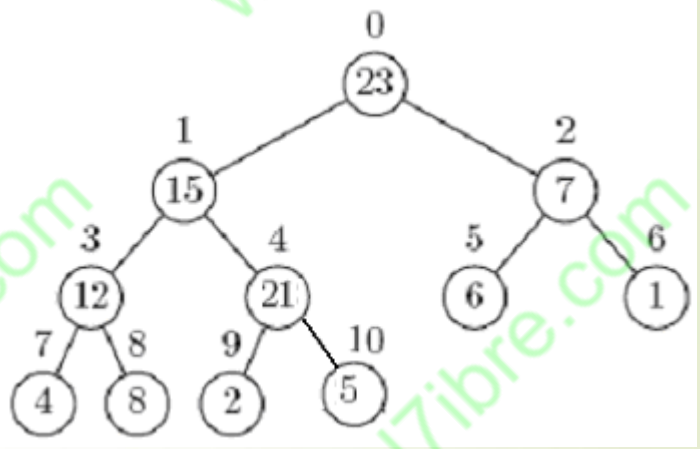
- **On compare 21, la nouvelle donnée insérée, avec la donnée contenue dans le nœud père, autrement dit on compare la donnée de la case d'indice 10 du tableau avec la donnée de la case d'indice = 4.**
  - **Puisque 21 est plus grand que 5, on les échange.**



0	1	2	3	4	5	6	7	8	9	10
23	15	7	12	5	6	1	4	8	2	21

Opération d'Ajout Exemple (3)

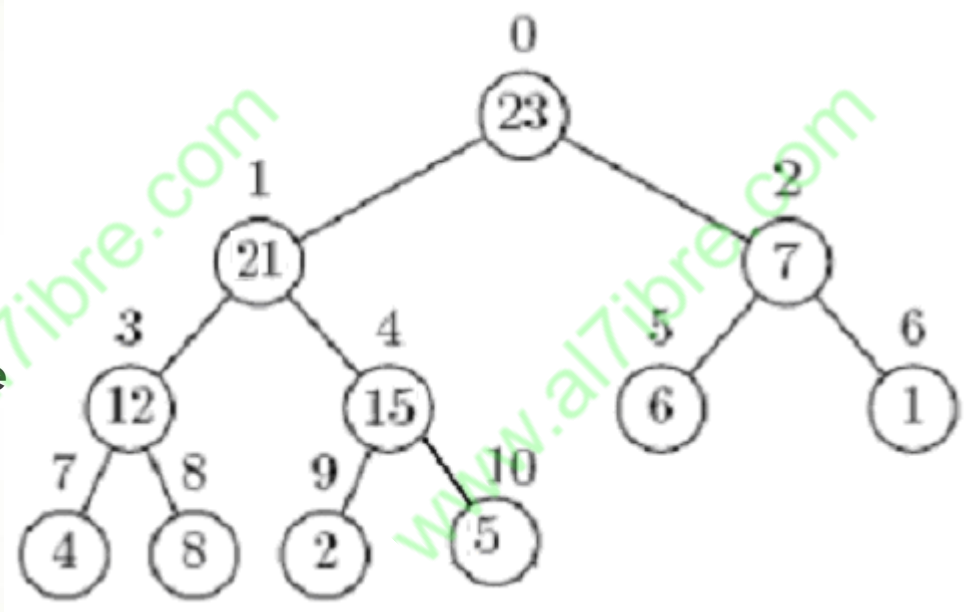
- **Le nouvel arbre binaire obtenu n'est pas un tas :**
  - **La valeur 21 du nœud d'indice 4 est plus grande que la valeur 15 de son nœud père (*d'indice* = 1)**
  - **Echanger les contenus des nœuds d'indices respectifs 1 et 4**



0	1	2	3	4	5	6	7	8	9	10
23	15	7	12	21	6	1	4	8	2	5

Opération d'Ajout Exemple (4)

- **Puisque 21 est plus petit que 23 :**
  - **L'opération d'ajout est terminée ;**
  - **On a bien obtenu un tas.**



0	1	2	3	4	5	6	7	8	9	10
23	21	7	12	15	6	1	4	8	2	5

Opération d'Ajout Pseudo-code

```
fonction ajouter(Tas t, Elément e) : Tas
début
   i ← t.taille
   t.taille ← i+1
   t.tableau[i] ← e
    tant que ((i > 0) et
          (t.tableau[i] > t.tableau[(i-1) div 2]))
    faire {
       échanger(t.tableau[i], t.tableau[(i-1) div
    2]
       i ← (i-1) div 2
    }
    retourner (t)
fin
```

Opération d'Ajout Complexité

- **La complexité de l'opération d'ajout est en O(h), où h est la hauteur du tas :**
  - **On ne fait que remonter un chemin ascendant d'une feuille vers la racine (en s'arrêtant éventuellement avant).**
  - **La hauteur d'un tas de taille n est précisément égale à** $\lfloor \log_2(n) \rfloor$

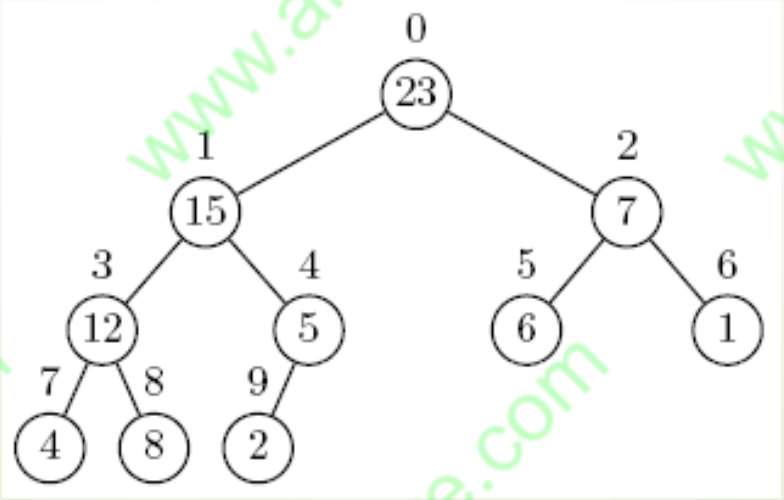
et donc l'ajout demande un temps O(log(n)).

Opération de Suppression du Maximum

- **Principe :**
  - ***Remplacer la clé du nœud racine par la clé du nœud situé le plus à droite du dernier niveau du tas. Ce dernier nœud est alors supprimé ;***
  - ***Réorganiser l'arbre, pour qu'il respecte la définition du tas, en faisant descendre la clé de l'élément de la racine à sa bonne place en permutant avec le plus grand des fils.***

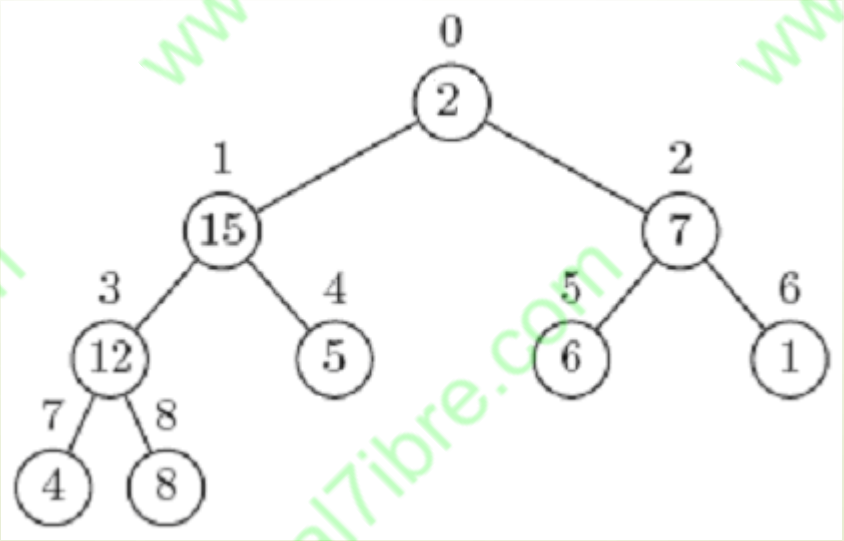
Opération de Suppression du Maximum (Exemple) (1)

- ***Supposons qu'on désire supprimer la valeur 23 contenue dans la racine du tas illustré par la figure suivante :***



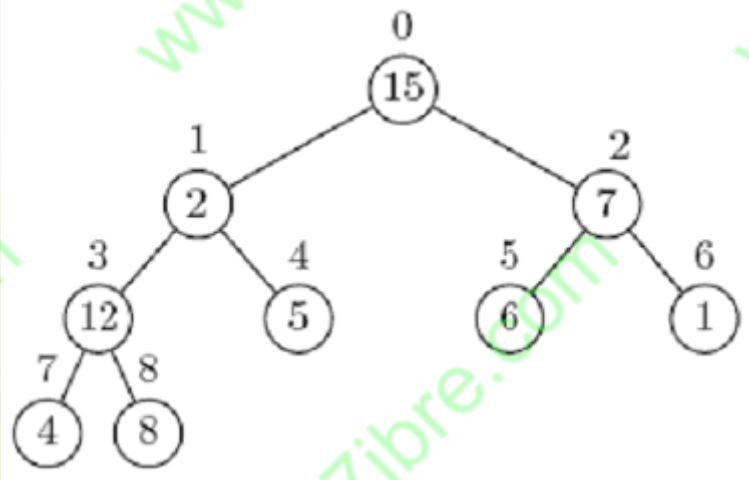
Opération de Suppression du Maximum (Exemple) (2)

- ***On commence alors par remplacer le contenu du nœud racine par celui du dernier nœud du tas :***
  - ***Ce dernier nœud est alors supprimé ;***
  - ***Ceci est illustré par la figure suivante :***



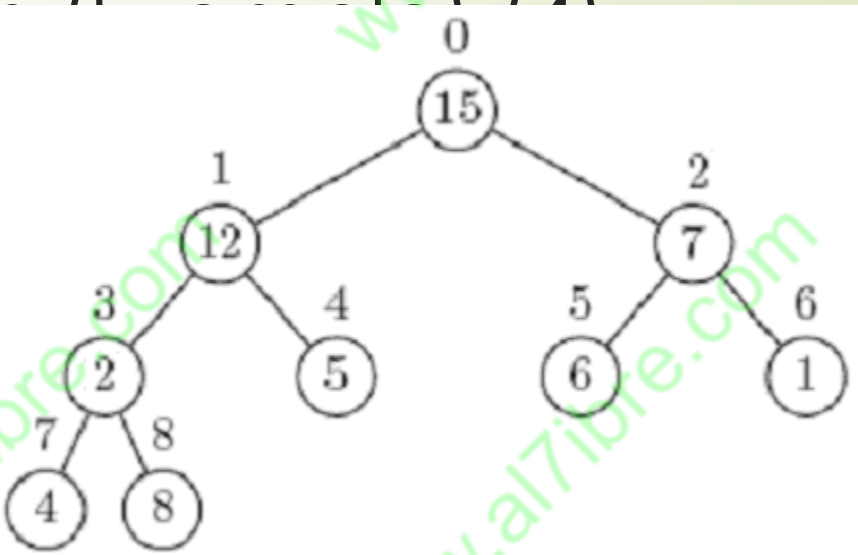
Opération de Suppression du Maximum (Exemple) (3)

- ***L'arbre obtenu est parfait mais n'est pas un tas :***
  - *la clé contenue dans la racine a une valeur plus petite que les valeurs des clés de ses fils ;*
  - *Cette clé de valeur 2 est alors échangée avec la plus grande clé de ses fils, à savoir 15 ;*
  - *L'arbre obtenu est représenté par la figure suivante :*

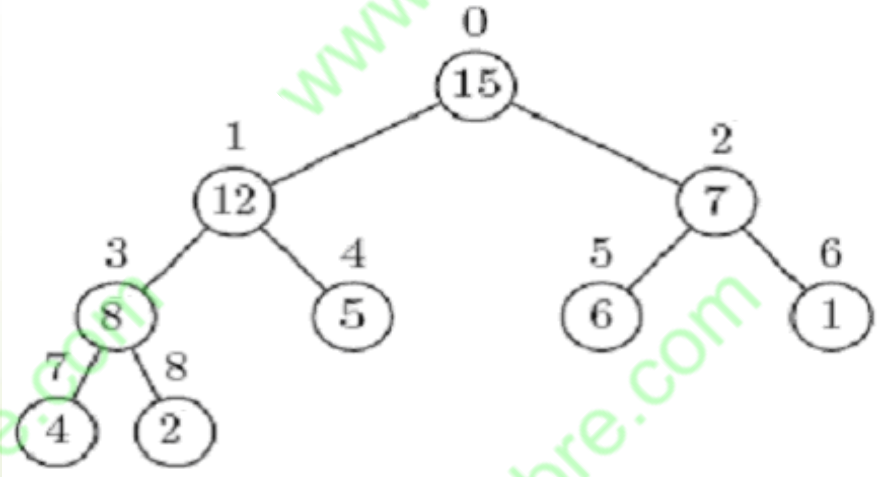


Opération de Suppression du Maximum (Exemple) (4)

- ***Encore une fois, cet arbre n'est pas un tas. On le réorganise pour qu'il respecte la définition du tas***



- ***Le dernier arbre obtenu est bien un tas ; il est illustré par la figure suivante*** :



Opération de Suppression du Maximum (Pseudo-code) (1)

- **Une version qui utilise la procédure *Entasser***
- **La procédure *Entasser* :**
  - **permet de faire descendre la valeur en t[i] de manière que l'arbre de racine en i devienne un tas ;**
  - **suppose que les sous arbres de racines en *2i+1 (fils gauche du nœud en i) et en 2i+2* (*fils droit du nœud en i*) sont des tas.**

Opération de Suppression du Maximum (Pseudo-code) (2)

```
procédure Entasser(Tableau t[0 .. n-1], Entier
  i)
début
   si ((2i+2 == n) ou (t[2i+1] ≥ t[2i+2])) alors
     k ← 2i+1
  sinon
    k ← 2i+2
  fsi
  si t[i] < t[k] alors
     échanger (t[i], t[k])
     si k ≤ ((n div 2)- 1) alors
       Entasser(t, k)
    fsi
  fsi
fin
```

Opération de Suppression du Maximum (Pseudo-code) (3)

```
fonction supprimerMax (t : tas) : tas
(* le tas t est supposé non vide !! *)

début
   t.taille ← t.taille - 1
   t.tableau[0] ← t.tableau[t.taille]
   Entasser(t, 0)
   retourner (t)
fin
```

Opération de Suppression du Maximum (Complexité)

- **La complexité de la suppression est la même que celle de l'insertion, c-à-d O(log(n)) :**
  - ***En effet, on ne fait que suivre un chemin descendant depuis la racine.***

Opération de Recherche du Maximum (Pseudo-code & Complexité)

- ***L'opération de recherche du maximum est immédiate dans les tas***
- ***Elle prend un temps constant O(1)***

```
fonction Max (t : tas) : Elément
(* le tas t est supposé non vide !! *)
début
 retourner (t.tableau[0])
fin
```

Exemples d'Applications des Tas

- **Files de priorités (*Priority queues*) :**
  - ***Les tas sont fréquemment utilisés pour implémenter des files de priorités.***
  - ***A l'opposé des files standard, une file de priorités détruit l'élément de plus haute (ou plus basse) priorité.***
  - ***La signification de la "priorité" d'un élément dépend de l'application***
  - ***A tout instant, on peut insérer un élément de priorité arbitraire dans une file de priorités. Si l'application souhaite la destruction de l'élément de plus haute priorité, on utilise un arbre maximier.***

- **Tri par tas (*Heapsort*):**
  - ***Les opérations sur les tas permettent de résoudre un problème de tri à l'aide d'un algorithme appelé tri par tas (heapsort).***
  - ***Cet algorithme a la même complexité temporelle, O(n log(n)), que le tri rapide (quicksort). Mais, en pratique, une bonne implémentation de ce dernier le bat d'un petit facteur constant.***

Algorithme du Tri par Tas (Principe)

- **Supposons qu'on veut trier, en ordre croissant, un tableau T de n éléments.**

- **Principe :**
  - **L'algorithme du tri par tas commence, en utilisant la fonction *ConstruireTas*, par construire un tas dans le tableau à trier T ;**
  - **Ensuite, il prend l'élément maximal du tas, qui se trouve en T[0], l'échange avec T[n-1], et rétablit la propriété de tas, en utilisant l'appel de fonction *Entasser(T,0)* pour le nouveau tableau à n-1 éléments (*la case T[n-1] n'est pas considérée*) ;**
  - **L'algorithme de tri par tas répète ce processus pour le tas de taille n-1 jusqu'à la taille 2.**

Algorithme du Tri par Tas (Pseudo-code) (1)

```
fonction Tri_par_Tas(Tableau T[0 .. n-1]) :
   Tableau
début
   T ← ConstruireTas(T)
   pour i ← (n-1) à 1 par pas -1 faire
      Echanger(T[0], T[i])
      n ← n-1
      Entasser (T, i)
   retourner (T)
fin
```

ConstruireTas produit un tas à partir d'un tableau T

Entasser sert à garantir le maintien de la propriété de tas pour l'arbre de racine en i

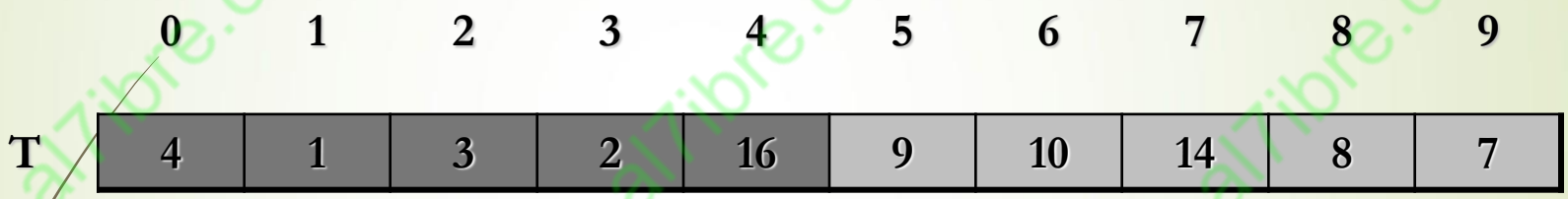
Algorithme du Tri par Tas (Pseudo-code) (2)

```
fonction ConstruireTas(Tableau T[0 .. n-1]) : Tas
début
   pour i ← ((n div 2) - 1) à 0 par pas -1 faire
    Entasser (T, i)
   retourner (T)
fin
```

Les feuilles sont des tas à un élément !

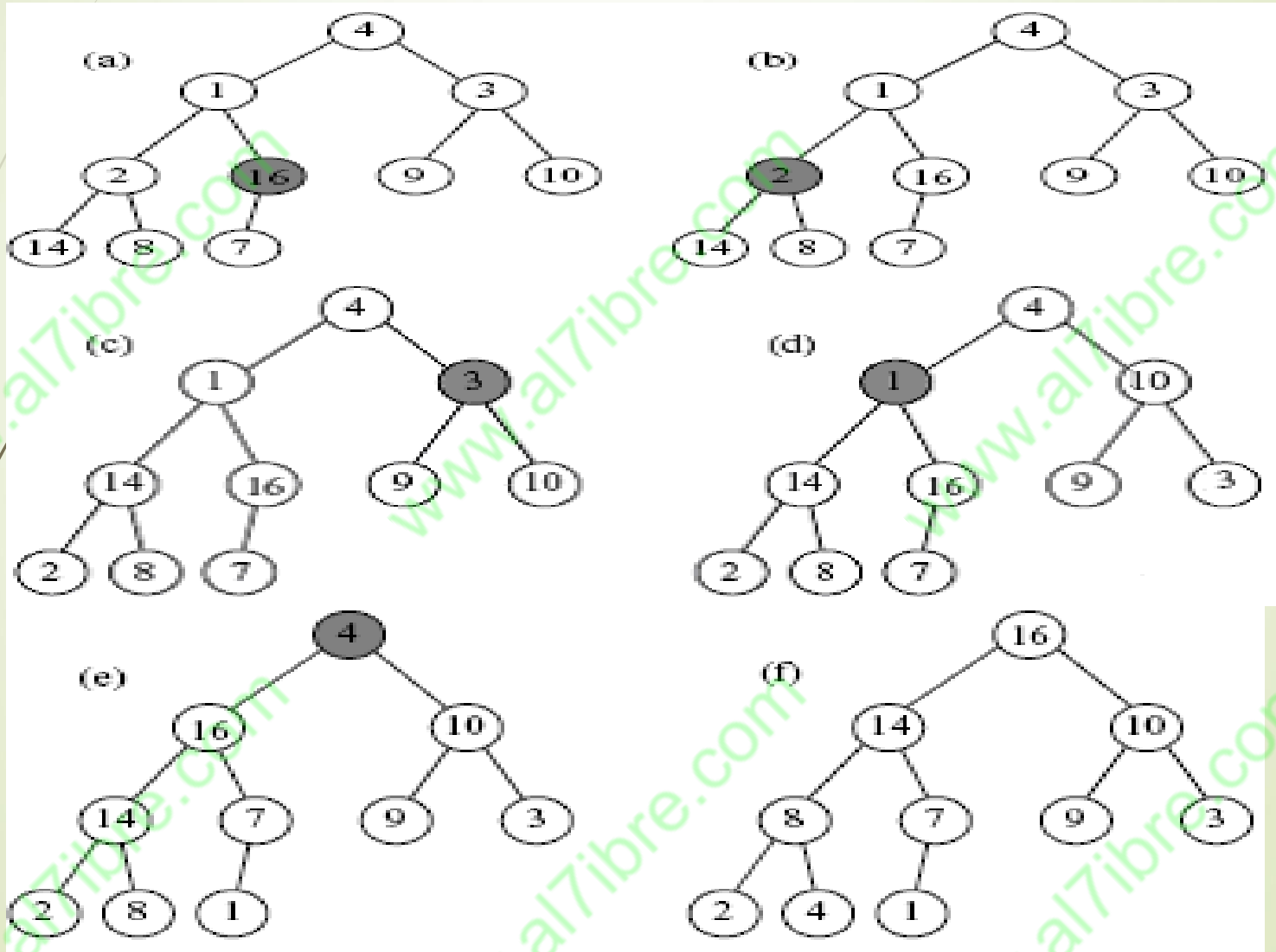
ConstruireTas Exemple (1)

- ***Illustration de l'action ConstruireTas sur un tableau d'entiers contenant 10 éléments***



- ***Remarquer que les nœuds qui portent les valeurs 9, 10, 14, 8 et 7 sont biens des feuilles, et donc des tas à un élément.***

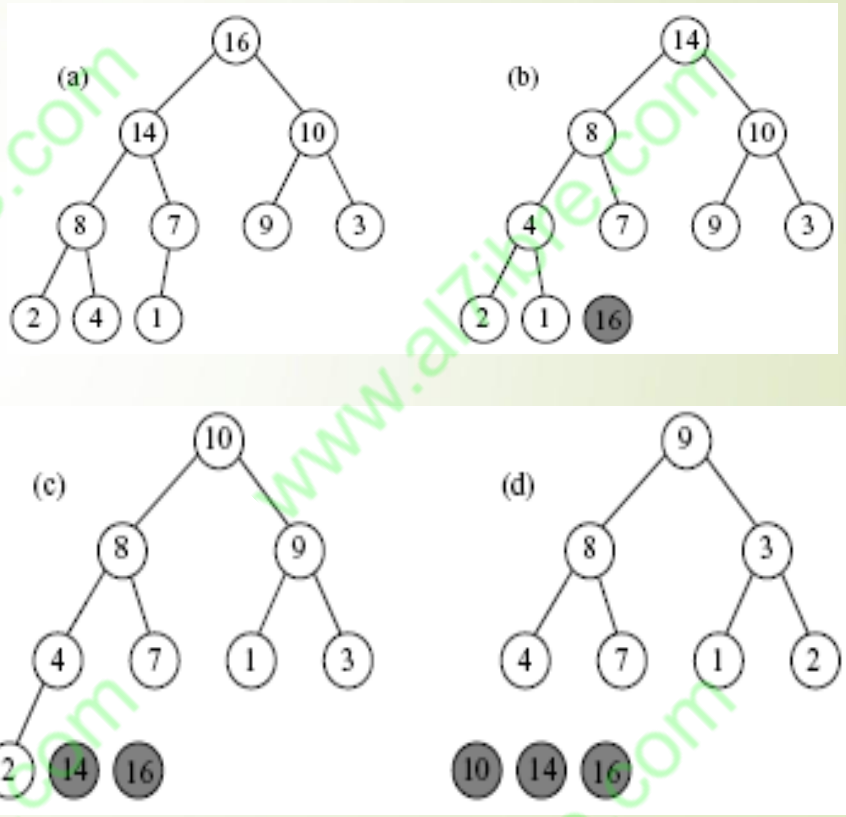
ConstruireTas: Exemple (2)



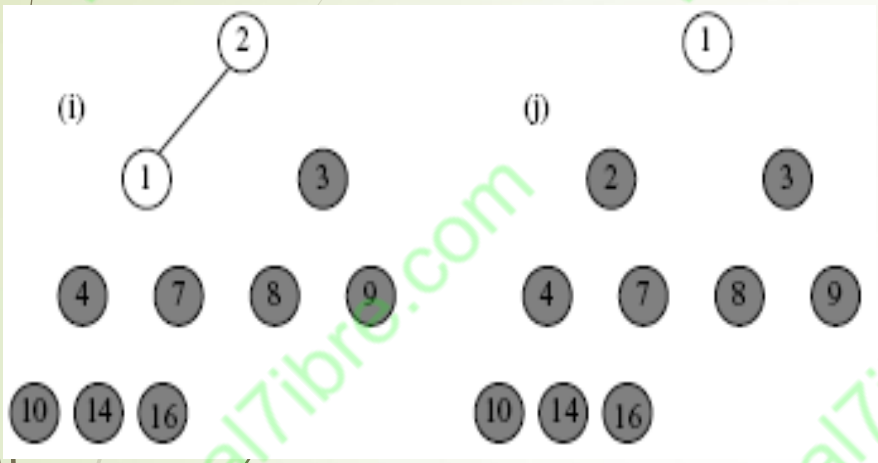
Tri par Tas : Exemple (1)

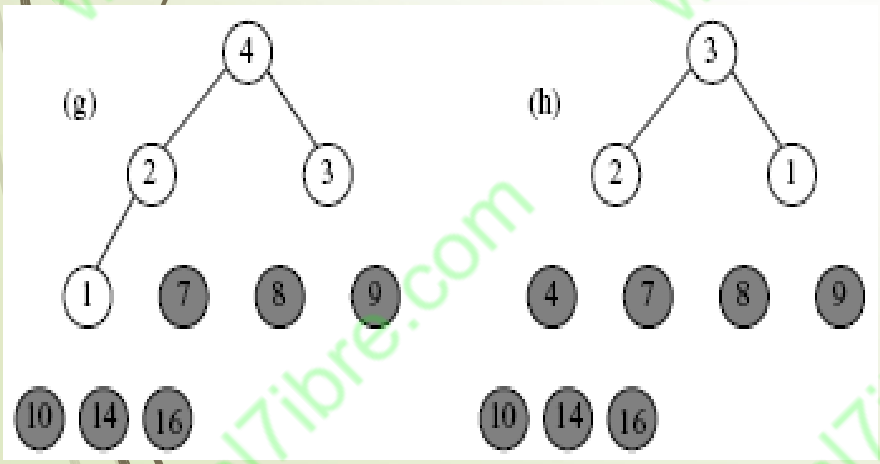
- ***Les figures qui suivent illustrent l'action du tri par tas après construction du tas***

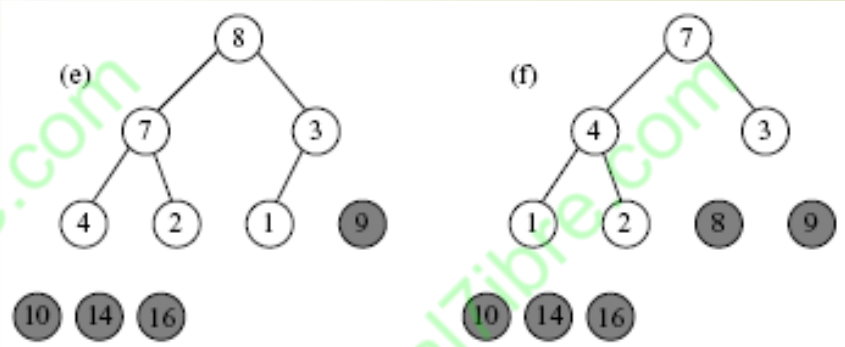
- ***Chaque tas est montré au début d'une itération de la boucle***



Tri par Tas : Exemple (2)







	0	1	2	3	4	5	6	7	8	9
T	1	2	3	4	7	8	9	10	14	16

Tableau final : trié

Algorithme du Tri par Tas Complexité

- **On montre que l'appel à ConstruireTas prend un temps O(n)**
- ***Chacun des (n-1) appels à Entasser prend un temps O(log(n))***
- **Par conséquent, l'algorithme du tri par tas s'exécute en O(n log(n))**

Introduction aux Arbres de Recherche Equilibrés

- (***Balanced Search Trees***)

Notion d'Arbres de Recherche Equilibrés

- *La définition des arbres équilibrés impose que la différence entre les hauteurs des fils gauche et des fils droit de tout noeud ne peut excéder 1*
- *Il faut donc maintenir l'équilibre de tous les noeuds au fur et à mesure des opérations d'insertion ou de suppression d'un nœud*
- *Quand il peut y avoir un déséquilibre trop important entre les deux fils d'un noeud, il faut recréer un équilibre par :*
  - *des rotations d'arbres ou par éclatement de nœuds (cas des arbres B)*
- *Les algorithmes de rééquilibrage sont très compliqués :*
  - *On cite entre autres, quelques exemples d'arbres équilibrés pour les quels les opérations de recherche, d'insertion et de suppression sont en O(log(n))*

Arbres de Recherche Equilibrés Exemples (1)

- ***Les arbres AVL :***
  - *Introduits par Adelson-Velskii Landis Landis (d'où le nom d'AVL) dans les années 60 ;*
  - *Un arbre AVL est un arbre binaire de recherche stockant une information supplémentaire pour chaque noeud : son facteur d'équilibre ;*
  - *Le facteur d'équilibre représente la différence des hauteurs entre son sous arbre gauche et son sous arbre droit ;*
  - *Au fur et à mesure que des nœuds sont insérés ou supprimés, un arbre AVL s'ajuste de lui-même pour que tous ses facteurs d'équilibres restent à 0, -1 ou 1.*

Arbres de Recherche Equilibrés Exemples (2)

- ***Les arbres rouges et noirs :***
  - ***Des arbres binaires de recherche qui se maintiennent eux-mêmes approximativement équilibrés en colorant chaque nœud en rouge ou noir ;***
  - ***En contrôlant cette information de couleur dans chaque noeud, on garantit qu'aucun chemin ne peut être deux fois plus long qu'un autre, de sorte que l'arbre reste équilibré.***

Arbres de Recherche Equilibrés Exemples (3)

- ***Les B arbres :***
  - *Arbres de recherche équilibrés qui sont conçus pour être efficaces sur d'énormes masses de données stockées sur mémoires secondaires ;*
  - *Chaque nœud permet de stocker plusieurs clés ;*
  - *Généralement, la taille d'un nœud est optimisée pour coïncider avec la taille d'un bloc (ou page) du périphérique, en vue d'économiser les coûteux accès d'entées sorties.*
- ...

Cours Structures de données

Arbres (Trees)

Pr F.Omary

2019-2020

Objectifs

- **Etudier des structures non linéaires**
  - **Arbres binaires**
  - **Arbres binaires de recherche**
  - **Arbres maximiers ou Tas**
  - **Arbres équilibrés**

Contenu

- Introduction
- Terminologie
- Arbres binaires
- Arbres binaires de recherche
- Arbres maximiers ou Tas
- Arbres équilibrés

Arbres (Trees)

Introduction

Notion d'Arbre (Tree)

- **Les arbres sont les structures de données les plus importantes en informatique**
- **Ce sont des *structures non linéaires* qui permettent d'obtenir des algorithmes plus performants que lorsqu'on utilise des structures de données linéaires telles que les listes et les tableaux**
- **Ils permettent une organisation naturelle des données**

Notion d'Arbre (Tree) Exemples

- ***Organisation des fichiers dans les systèmes d'exploitation ;***
- ***Organisation des informations dans un système de bases de données ;***
- ***Représentation de la structure syntaxique des programmes sources dans les compilateurs ;***
- ***Représentation d'une table de matières ;***
- ***Représentation d'un arbre généalogique ;***
- ***…***

Arbres (Trees)

Terminologie

Terminologie (1)

- Un arbre est un ensemble d'éléments appelés ***nœuds*** (***ou sommets***), liés par une relation (*dite de "parenté"*) induisant une structure hiérarchique parmi ces nœuds.

- Un nœud, comme tout élément d'une liste, peut être de n'importe quel type.

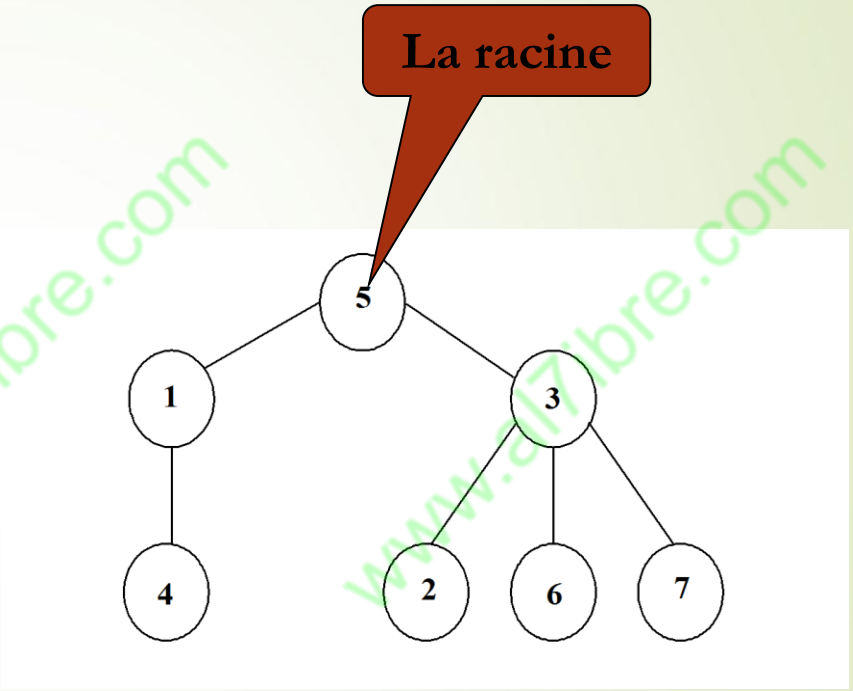
Terminologie (1) (suite)

- D'une manière plus formelle, une structure d'arbre de type de base T est :
  - soit la ***structure vide*** ;
  - soit un noeud de type T, appelé ***racine***, associé à un nombre fini de structures d'arbre disjointes du type de base T appelées ***sous arbres***
- C'est une définition récursive ; la récursivité est une propriété des arbres et des algorithmes qui les manipulent
- **Une liste** est un cas particulier des arbres (***arbre dégénéré***), où tout noeud a au plus un sous arbre

Illustration & Exemple

- Pour illustrer une structure d'arbre, on modélise le plus souvent un nœud par une information inscrite dans un cercle et les liens par des traits.

- ***Par convention***, on dessine les arbres avec la racine en haut et les branches dirigées vers le bas.



Exemple d'arbre formé de 7 nœuds (des entiers)

Terminologie (2)

- **La terminologie utilisée dans les structures d'arbres est empruntée :**
  - **aux arbres généalogiques :**
    - ***Père ;***
    - ***Fils ;***
    - ***Frère ;***
    - ***Descendant ;***
    - **…**
  - **et à la botanique :**
    - ***Feuille ;***
    - ***Branche ;***
    - **…**

Terminologie (3)

- ***Fils (ou enfants) :***
  - Chaque nœud d'un arbre pointe vers un ensemble éventuellement vide d'autres nœuds ; ce sont ses fils (*ses enfants*).
  - Sur l'exemple précédent, le nœud 5 a deux fils : 1 et 3, le nœud 1 a un fils : 4, et le nœud 3 a trois fils : 2, 6 et 7.
- ***Père :***
  - Tous les nœuds d'un arbre, sauf un, ont un père et un seul. Un nœud p est père du nœud n si et seulement si n est fils de p.
  - Par exemple, le père de 2 est 3, celui de 3 et 5.
- ***Frères :***
  - Deux nœuds ayant le même père.
  - Les nœuds 2, 6 et 7 sont des frères.
- ***Racine :***
  - Le seul nœud sans père.
  - 5 est la racine de l'arbre précédent.

Terminologie (4)

- ***Feuilles (ou nœuds terminaux, ou nœuds externes) :***
  - Ce sont des noeuds sans fils.
  - Par exemple, 4, 2, 6 et 7.
- ***Nœud interne :***
  - Un noeud qui n'est pas terminal.
  - Par exemple, 1, 3 et 5.
- ***Degré d'un noeud :***
  - Le nombre de fils de ce noeud.
  - Sur l'exemple, 5 est de degré deux, 1 est de degré un, 3 est de degré trois et les feuilles (4, 2, 6, 7) sont de degré nul.
- ***Degré d'un arbre (ou arité) :***
  - Plus grand degré des nœuds de l'arbre. Un arbre de degré n est dit *n-aire*
  - Sur l'exemple, l'arbre est un arbre 3-aire.

Terminologie (5)

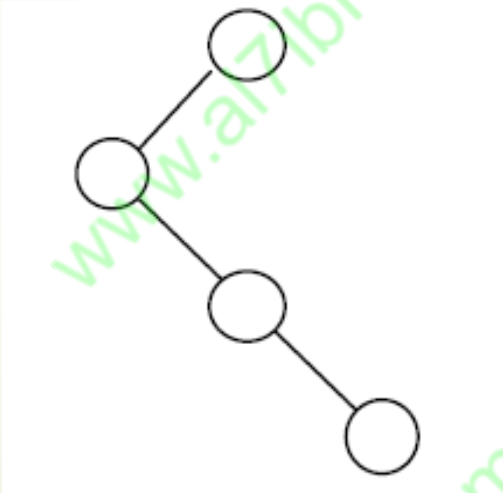
- ***Taille d'un arbre :***
  - Le nombre total des nœuds de l'arbre.
  - Sur l'exemple, l'arbre est de taille 7.
- ***Chemin :***
  - Une suite de noeuds d'un arbre $(n_1, n_2, \ldots, n_k)$ tel que n_i = père(n_i+1) pour $1 \leq i \leq k-1$ est appelée chemin entre le nœud n_1 et le nœud n_k.
  - La longueur d'un chemin est égale au nombre de nœuds qu'il contient moins 1.
  - Sur l'exemple, le chemin qui mène du nœud 5 au nœud 6 est de longueur 2.
- ***Branche :***
  - Un chemin qui commence à la racine et se termine à une feuille.
  - Par exemple, les chemins (5, 1, 4), (5, 3, 2), (5, 3, 6) et (5, 3, 7).
- ***Ancêtre :***
  - Un nœud A est un ancêtre d'un nœud B s'il existe un chemin de A vers B.
  - Par exemple, les ancêtres de 2 sont 2, 3 et 5
- ***Descendant :***
  - Un nœud A est un descendant d'un nœud B s'il existe un chemin de B vers A.
  - Sur l'exemple, 5 admet les 7 nœuds de l'arbre comme descendants.

Terminologie (6)

- ***Sous arbre :***
  - **Un sous arbre d'un arbre A est constitué de tous les descendants d'un nœud quelconque de A.**
  - **Les ensembles de noeuds {3, 2, 6, 7} et {2} forment deux sous arbres de l'exemple précédent.**
- ***Hauteur (ou profondeur, ou niveau) d'un noeud :***
  - **Longueur du chemin qui relie la racine à ce nœud.**
  - **La racine est elle même de hauteur 0, ses fils sont de hauteur 1, et les autres noeuds de hauteur supérieure à 1.**
- ***Hauteur d'un arbre :***
  - **Plus grande profondeur des nœuds de l'arbre supposé non vide, c'est-à-dire *h(A) = Max{h(x) ; x noeud de A}***
  - **L'arbre de l'exemple est de profondeur 2.**
  - ***Par convention*, un arbre vide a une hauteur de -1.**

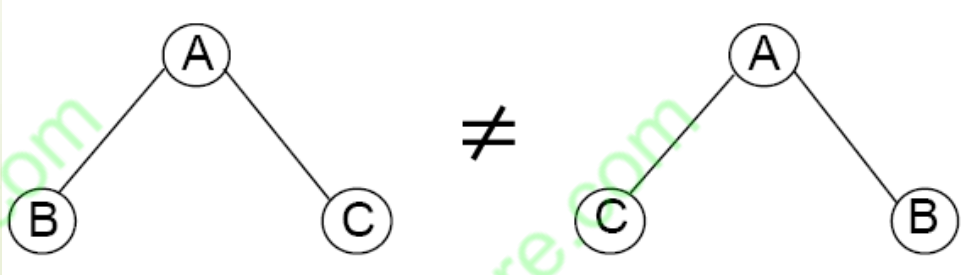
Terminologie (7)

- ***Arbre dégénéré ou filiforme :***
  - **Un arbre dont chaque nœud a au plus au fils**



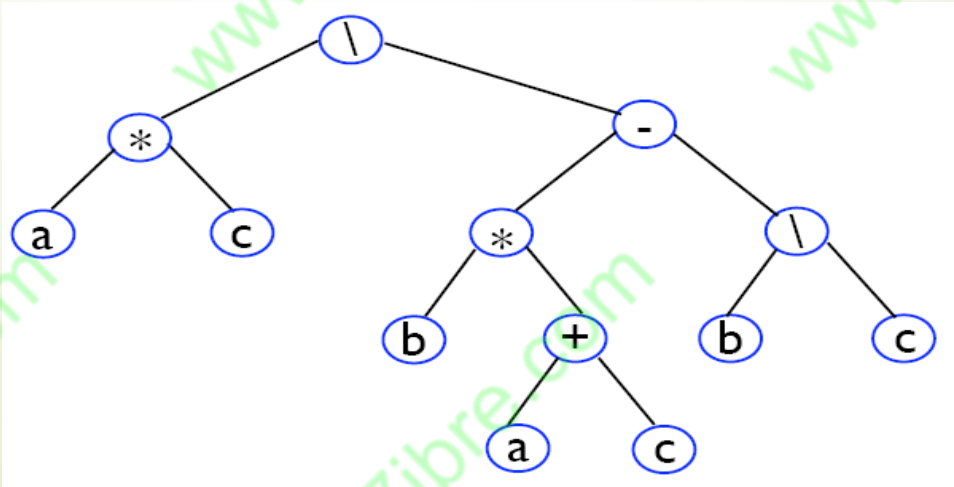
Terminologie (7)

- ***Arbre ordonné :***
  - **Un arbre où la position respective des sous arbres reflète une relation d'ordre. En d'autres termes, si un nœud a k fils, il existe un 1er fis, un 2ème fils, …, et un kème fils.**
  - **Les deux arbres de la figure qui suit sont différents si on les regarde comme des arbres ordonnés, mais identiques si on les regarde comme de simples arbres.**



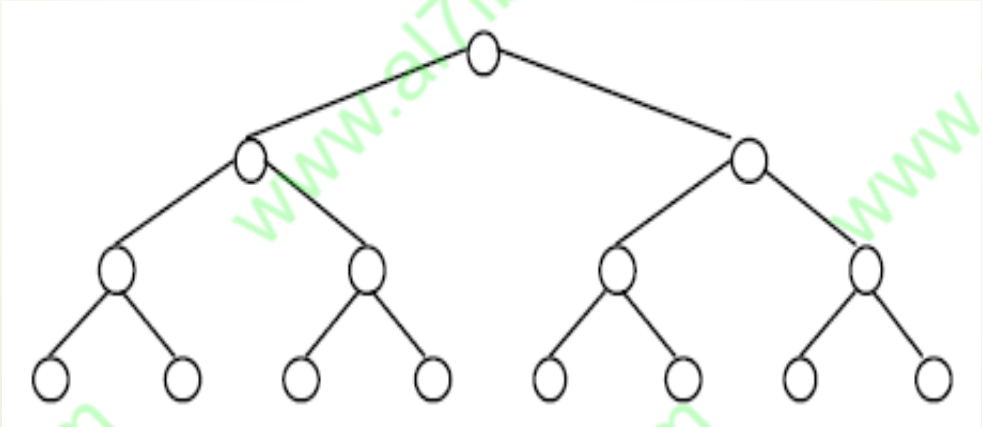
Terminologie (8)

- ***Arbre binaire :***
  - **Un arbre où chaque noeud a au plus deux fils.**
  - **Quand un nœud de cet arbre a un seul fils, on précise s'il s'agit du *fils gauche* ou du *fils droit*.**
  - **La figure qui suit montre un exemple d'arbre binaire dans lequel les nœuds contiennent des caractères.**



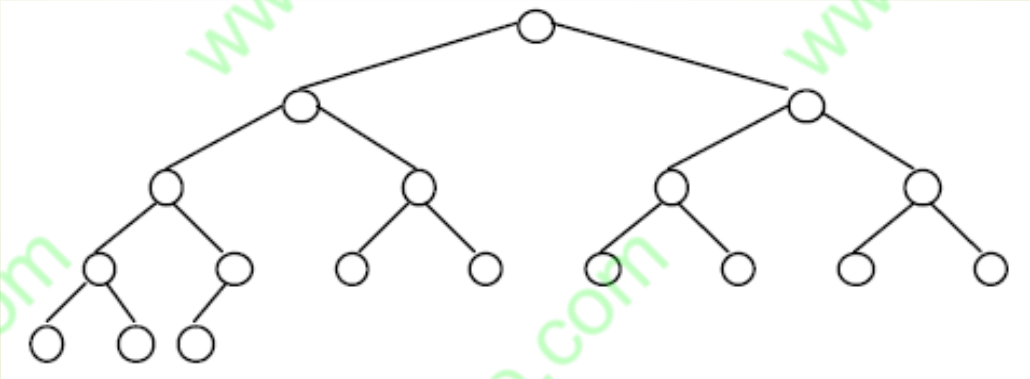
Terminologie (9)

- ***Arbre binaire complet :***
  - **Arbre binaire dont chaque niveau est rempli.**



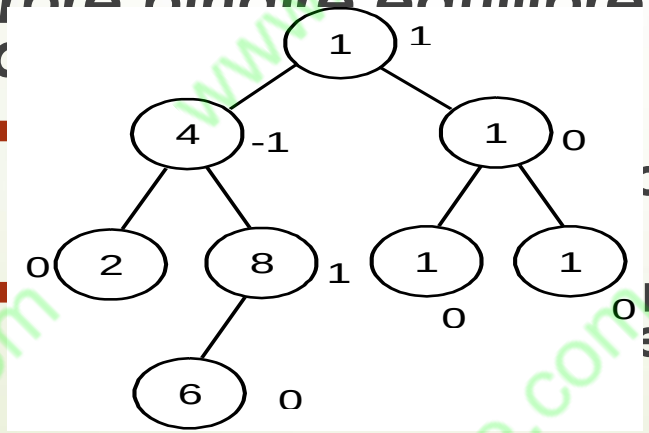
Terminologie (10)

- ***Arbre binaire parfait (ou presque complet) :***
  - **Arbre binaire dont chaque niveau est rempli sauf éventuellement le dernier**
  - **Dans ce cas les nœuds terminaux (*feuilles*) sont groupés le plus à gauche possible.**



Terminologie (11)

- ***Facteur d'équilibre d'un nœud d'un arbre binaire :***
  - **Hauteur du sous arbre partant du fils gauche du nœud moins la hauteur du sous arbre partant de son fils droit.**
- ***Arbre binaire équilibré (au sens des ha***
  - **pour chaque nœud, cteur d'équilibre est**
  - **n place à côté de eur d'équilibre.**

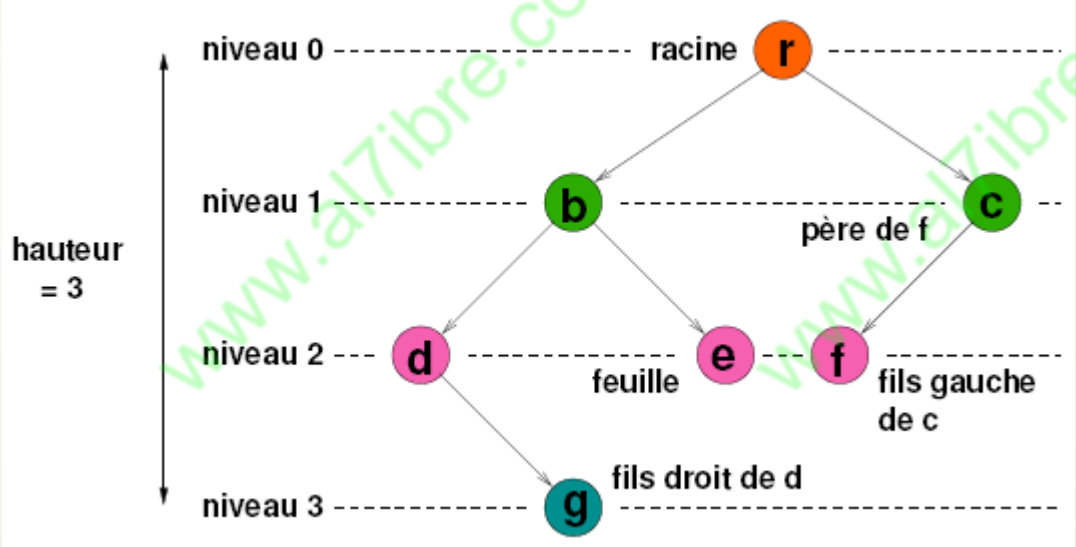


Arbres Binaires (Binary Trees)

Définition

- **Un arbre binaire A** est :
  - soit **vide** (**A = ( )** ou **A = ø**),
  - soit **de la forme A = <r, A1, A2>**, c-à-d composé :
    - d'un nœud **r** appelé ***racine*** contenant un élément
    - et de deux arbres binaires disjoints **A1** et **A2**, appelés respectivement ***sous arbre gauche*** (*ou* ***fils gauche***) et ***sous arbre droit*** (*ou* ***fils droit***).

Exemple d'arbre binaire



Type Abstrait Arbre_Binaire

```
Type Arbre_Binaire
Utilise Noeud, Elément, Booléen
Opérations
  arbre_vide : → Arbre_Binaire
  est_vide   : Arbre_Binaire → Booléen
  cons       : Noeud x Arbre_Binaire x Arbre_Binaire → Arbre_Binaire
  racine     : Arbre_Binaire → Noeud
  gauche     : Arbre_Binaire → Arbre_Binaire
  droite     : Arbre_Binaire → Arbre_Binaire
  contenu    : Noeud → Elément
Préconditions
  racine(A) est-défini-ssi est_vide(A) = faux
  gauche(A) est-défini-ssi est_vide(A) = faux
  droite(A) est-défini-ssi est_vide(A) = faux
Axiomes
  Soit, r : Nœud, A1, A2 : Arbre_Binaire
  racine(<r, A1, A2>) = r
  gauche(<r, A1, A2>) = A1
  droite(<r, A1, A2>) = A2
```

Opérations sur un Arbre Binaire (1)

- **`arbre_vide : → Arbre_Binaire`**
  - opération d'initialisation; crée un arbre binaire vide.
- **`est_vide : Arbre_Binaire → Booléen`**
  - teste si un arbre binaire est vide ou non.
- **`cons : Noeud x Arbre_Binaire x Arbre_Binaire → Arbre_Binaire`**
  - *cons(r,G,D)* construit un arbre binaire dont le sous arbre gauche est *G* et le sous arbre droit est *D*, et *r* est le nœud racine qui contient une donnée de type Elément.
- **`racine : Arbre_Binaire → Noeud`**
  - si `A` est un arbre binaire non vide alors `racine(A)` retourne le nœud racine de `A`, sinon un message d'erreur.

Opérations sur un Arbre Binaire (2)

- **`gauche : Arbre_Binaire → Arbre_Binaire`**
  - si `A` est un arbre binaire non vide alors `gauche(A)` retourne le sous arbre gauche de `A`, sinon un message d'erreur.
- **`droite : Arbre_Binaire → Arbre_Binaire`**
  - si `A` est un arbre binaire non vide alors `droite(A)` retourne le sous arbre droit de `A`, sinon un message d'erreur.
- **`contenu : Noeud → Elément`**
  - permet d'associer à chaque noeud d'un arbre binaire une information de type Elément.

Opérations Auxiliaires

```
Extension Type Arbre_Binaire
Utilise Entier, Booléen
Opérations
  taille   : Arbre_Binaire → Entier
  hauteur  : Arbre_Binaire → Entier
  feuille  : Arbre_Binaire → Booléen
Préconditions

Axiomes
  Soit, r : Noeud, A1, A2 : Arbre_Binaire

  taille(arbre_vide) = 0
  taille(<r, A1, A2>) = 1 + taille(A1) + taille(A2)
  hauteur(arbre_vide) = -1
  si hauteur(A1) > hauteur(A2) alors hauteur(<r, A1, A2>) = 1+hauteur(A1)
  sinon hauteur(<r, A1, A2>) = 1 + hauteur(A2)
  si est_vide(A) = faux et est_vide(gauche(A)) = vrai
   et est_vide(droit(A)) = vrai
  alors feuille(A) = vrai
  sinon feuille(A) = faux
```

Parcours d'arbre binaire

- **Un parcours d'arbre permet d'accéder à chaque nœud de l'arbre :**
  - **Un traitement (*test, affichage, comptage, etc.*), dépendant de l'application considérée, est effectué sur l'information portée par chaque nœud**
  - **Chaque parcours de l'arbre définit un ordre sur les nœuds**
- **On distingue :**
  - ***Les parcours de gauche à droite (le fils gauche d'un nœud précède le fils droit) ;***
  - ***Les parcours de droite à gauche (le fils droit d'un nœud précède le fils gauche).***
- **On ne considèrera que les parcours de gauche à droite**
- **On distingue aussi deux catégories de parcours d'arbres :**
  - ***Les parcours en profondeur ;***
  - ***Les parcours en largeur.***

Parcours en profondeur

- **Soit un arbre binaire A = <r, A1, A2>**

- **On définit trois parcours en profondeur de cet arbre :**
  - ***Le parcours préfixe ;***
  - ***Le parcours infixe ou symétrique ;***
  - ***Le parcours postfixe ou suffixe.***

Parcours en profondeur
Parcours préfixe

- **En abrégé *RGD* (*Racine, Gauche, Droit*)**
- **Consiste à effectuer dans l'ordre :**
  - ***Le traitement de la racine r ;***
  - ***Le parcours préfixe du sous arbre gauche A1 ;***
  - ***Le parcours préfixe du sous arbre droit A2.***
- ***L'ordre correspondant s'appelle l'ordre préfixe***

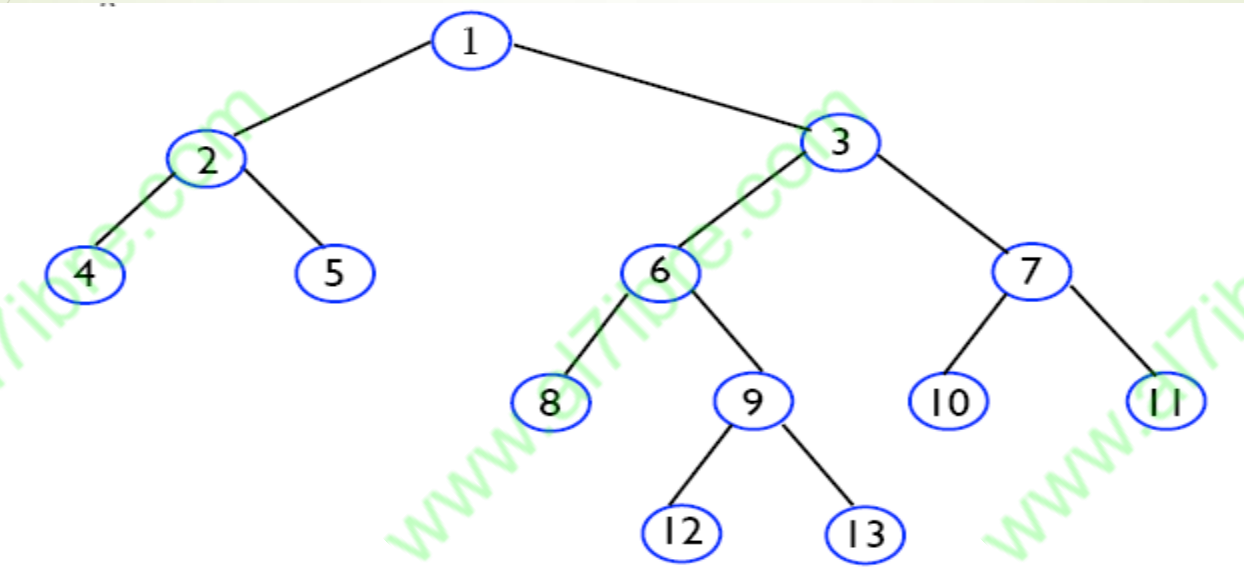
Parcours en profondeur
Parcours infixe ou symétrique

- **En abrégé *GRD* (*Gauche, Racine, Droit*)**
- **Consiste à effectuer dans l'ordre :**
  - ***Le parcours infixe du sous arbre gauche A1 ;***
  - ***Le traitement de la racine r ;***
  - ***Le parcours infixe du sous arbre droit A2.***
- ***L'ordre correspondant s'appelle l'ordre infixe***

Parcours en profondeur parcours postfixe ou suffixe

- **En abrégé *GDR* (*Gauche, Droit, Racine*)**
- **Consiste à effectuer dans l'ordre :**
  - ***Le parcours postfixe du sous arbre gauche A1 ;***
  - ***Le parcours postfixe du sous arbre droit A2 ;***
  - ***Le traitement de la racine r.***
- ***L'ordre correspondant s'appelle l'ordre suffixe***

Exemple de Parcours en profondeur (affichage du contenu des nœuds)



Le parcours préfixe affiche les nœuds dans l'ordre : ***1, 2, 4, 5, 3, 6, 8, 9, 12, 13, 7, 10, 11***
Le parcours infixe affiche les nœuds dans l'ordre : ***4, 2, 5, 1, 8, 6, 12, 9, 13, 3, 10, 7, 11***
Le parcours postfixe affiche les nœuds dans l'ordre : ***4, 5, 2, 8, 12, 13, 9, 6, 10, 11, 7, 3, 1***

Parcours en largeur

- **On explore les noeuds :**
  - ***niveau par niveau,***
  - ***de gauche à droite,***
  - ***en commençant par la racine.***

- **Exemple :**
  - **Le parcours en largeur de l'arbre de la figure précédente affiche la séquence d'entiers suivante : *1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13***

Représentations d'un arbre binaire

- **Représentation par tableau (*par contiguïté*)**
- **Représentation par pointeurs (*par chaînage*)**

Représentation contiguë d'un arbre binaire

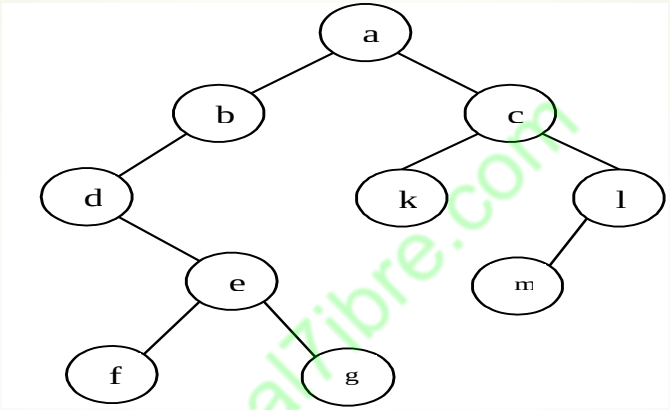
- **On caractérise un arbre binaire par :**
  - **sa taille (*nombre de nœuds*) ;**
  - **sa racine (*indice de son emplacement dans le tableau de nœuds*)**
  - **un tableau de nœuds.**

- **Chaque nœud contient trois données :**
  - **une information de type Elément ;**
  - **deux entiers (*indices dans le tableau désignant respectivement l'emplacement des fils gauche et droit du nœud*).**

Représentation contiguë d'un arbre binaire

```
#define NB_MAX_NOEUDS 15
typedef int Element;
typedef struct noeud {
 Element val;
 int fg;
 int fd;
} Noeud;
typedef Noeud TabN[Nb_MAX_NOEUDS];
typedef struct arbre {
 int nb_noeuds;
 int racine;
 TabN les_noeuds;
} Arbre_Binaire
```

Exemple de Représentation contiguë



nb_noeuds	racine
10	2

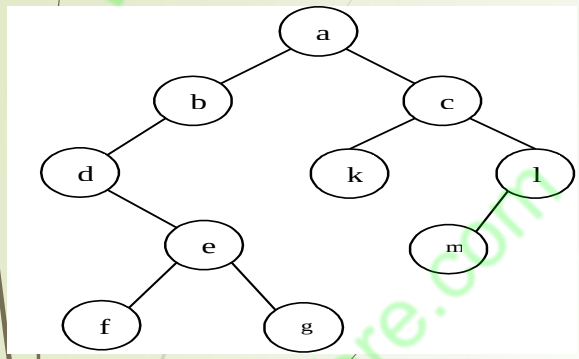
les_noeuds

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
val		d	a	g	b	c		f	m	e	l		k		
fg		-1	4	-1	1	12		-1	-1	7	8		-1		
fd		9	5	-1	-1	10		-1	-1	3	-1		-1		

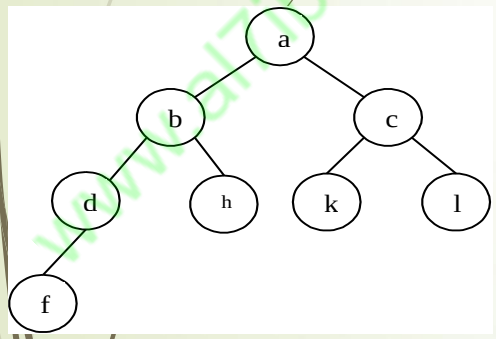
Autre représentation contiguë d'un arbre binaire

- Repose sur l'ordre hiérarchique (*numérotation des nœuds niveau par niveau et de gauche à droite*)
- On rappelle que pour stocker un arbre binaire de hauteur h, il faut un tableau de 2^{h+1}-1 éléments
- On organise le tableau de la façon suivante :
  - Le noeud racine a pour indice 0 (*en langage C*) ;
  - Soit le noeud d'indice i dans le tableau, son fils gauche a pour indice 2i +1, et son fils droit a pour indice 2(i+1).
- Représentation idéale pour les arbres binaires parfaits. En effet, elle ne gaspille pas d'espace.

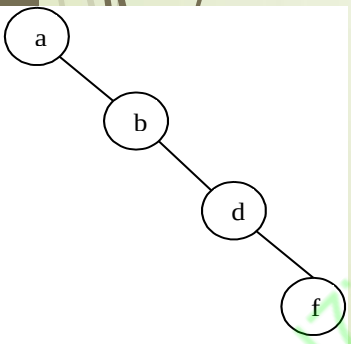
Autre représentation contiguë d'un arbre binaire (Exemples)



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
a	b	c	d		k	l		e					m				f	g



0	1	2	3	4	5	6	7
a	b	c	d	h	k	l	f



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a		b				d								f

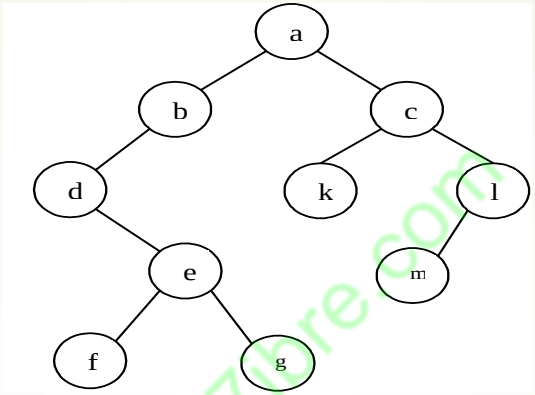
Représentation chaînée d'un arbre binaire

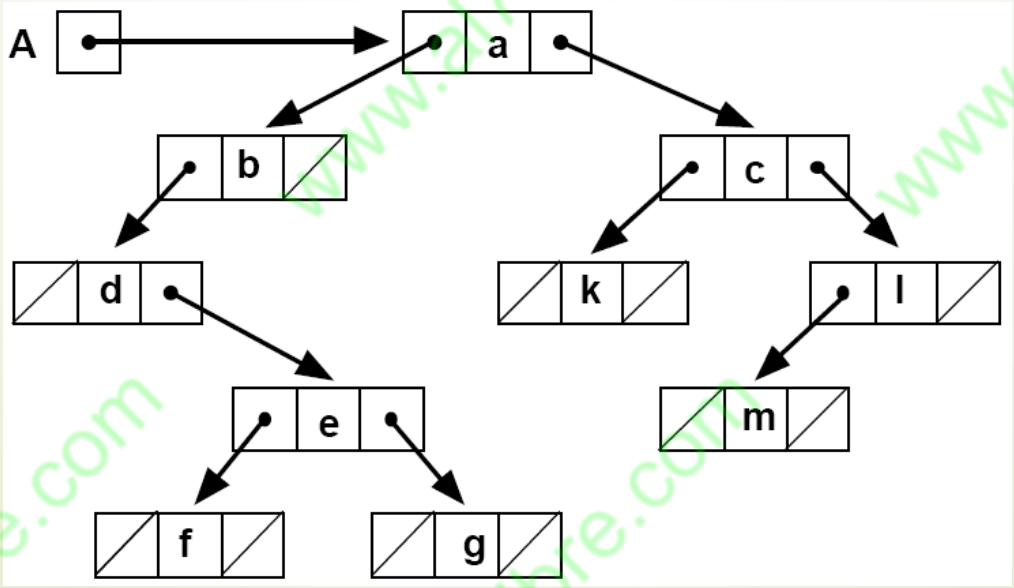
- **Chaque nœud a trois champs :**
  - **val (*l'élément stocké dans le noeud*) ;**
  - ***fg* (*pointeur sur fils gauche*) ;**
  - ***fd* (*pointeur sur fils droit*).**
- **Un arbre est désigné par un pointeur sur sa racine**
- **Un arbre vide est représenté par le pointeur NULL**

Représentation chaînée en C d'un arbre binaire


```
typedef int Element;
typedef struct noeud *Pnoeud;
typedef struct noeud {
 Element val;
 Pnoeud fg;
 Pnoeud fd;
} Noeud;
typedef Noeud *Arbre_Binaire;
```

Exemple de Représentation chaînée d'un arbre binaire





Réalisation chaînée d'un arbre binaire


```
Arbre_Binaire arbre_vide() {
 return NULL;
}

Booleen est_vide(Arbre_Binaire A) {
 return A == NULL ;
}

Pnoeud nouveau_noeud(Element e) {
// faire une allocation mémoire et placer l'élément e
// en cas d'erreur d'allocation, le pointeur renvoyé est
   NULL
 Pnoeud p = (Pnoeud) malloc(sizeof(Noeud));
 if (p != NULL) {
  p->val = e;
  p->fg = NULL;
  p->fd = NULL;
 }
 return (p);
}
```

Réalisation chaînée d'un arbre binaire


```
Arbre_Binaire cons(Noeud *r,
                   Arbre_Binaire G,
                      Arbre_Binaire D) {
 r->fg = G ;
 r->fd = D ;
 return r ;
}


Noeud racine(Arbre_Binaire A) {
// précondition : A est non vide !
 if (estvide(A)) {
  printf("Erreur : Arbre vide !\n");
  exit(-1);
 }
 return (*A) ;
}
```

```
Arbre_Binaire gauche(Arbre_Binaire A) {
// précondition : A est non vide !
 if (estvide(A)) {
  printf("Erreur : Arbre vide !\n");
  exit(-1);
 }
 return A->fg ; /* ou bien (*A).fg; */
}

Arbre_Binaire droite(Arbre_Binaire A) {
// précondition : A est non vide !
 if (estvide(A)) {
  printf("Erreur : Arbre vide !\n");
  exit(-1);
 }
 return A->fd ; /* ou bien (*A).fd; */
}

Element contenu(Noeud n) {
 return n.val;
}
```

Exemples d'Applications d'Arbre Binaire

- ***Recherche dans un ensemble de valeurs*** :
  - Les arbres binaires de recherche ;
- ***Tri d'un ensemble de valeurs :***
  - Le parcours GRD d'un arbre binaire de recherche ;
  - Un algorithme de tri efficace utilisant une structure de *tas ;*
- ***Représentation d'une expression arithmétique*** :
  - Un parcours GDR pour avoir une notation postfixée ;
- ***Méthodes de compression :***
  - Le codage de *Huffman u*tilisant des arbres binaires ;
  - La compression d'images utilisant des *quadtrees (arbres quaternaires, où chaque nœud non feuille a exactement quatre fils) ;*
- …

Arbres de Recherche Equilibrés Exemples (3)

- ***Les B arbres :***
  - *Arbres de recherche équilibrés qui sont conçus pour être efficaces sur d'énormes masses de données stockées sur mémoires secondaires ;*
  - *Chaque nœud permet de stocker plusieurs clés ;*
  - *Généralement, la taille d'un nœud est optimisée pour coïncider avec la taille d'un bloc (ou page) du périphérique, en vue d'économiser les coûteux accès d'entées sorties.*
- ...

Arbres Binaires de Recherche (Binary Search Trees)

Pr F.Omary

2019-2020

Notion d'Arbre binaire de recherche

- **C'est un arbre binaire particulier :**
  - ***Permet d'obtenir un algorithme de recherche proche dans l'esprit de la recherche dichotomique ;***
  - ***Pour lequel les opérations d'ajout et de suppression d'un élément sont aussi efficaces.***
- **Cet arbre utilise l'existence d'une relation d'ordre sur les éléments, représentée par une fonction clé, à valeur entière.**

Arbre binaire de recherche Définition

- **Un arbre binaire de recherche (*binary search tree* en anglais), en abrégé *ABR*, est un arbre binaire tel que pour tout nœud :**
  - ***les clés de tous les noeuds du sous-arbre gauche sont inférieures ou égales à la clé du nœud,***
  - ***les clés de tous les noeuds du sous-arbre droit sont supérieures à la clé du nœud.***
- ***Chaque nœud d'un arbre binaire de recherche désigne un élément qui est caractérisé par une clé (prise dans un ensemble totalement ordonné) et des informations associées à cette clé.***
- ***Dans toute illustration d'un arbre binaire de recherche, seules les clés sont représentées. On supposera aussi que toute clé identifie de manière unique un élément.***

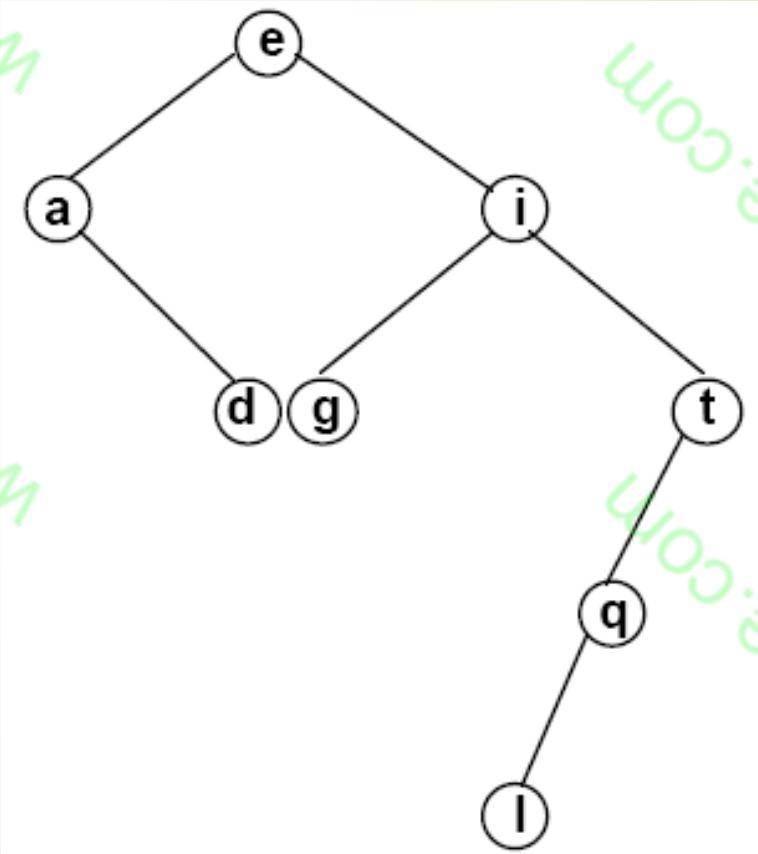
Arbre binaire de recherche Exemple

- **L'arbre de la figure suivante est un arbre binaire de recherche**

- **Cet arbre représente l'ensemble :**

$$E = \{a, d, e, g, i, l, q, t\}$$

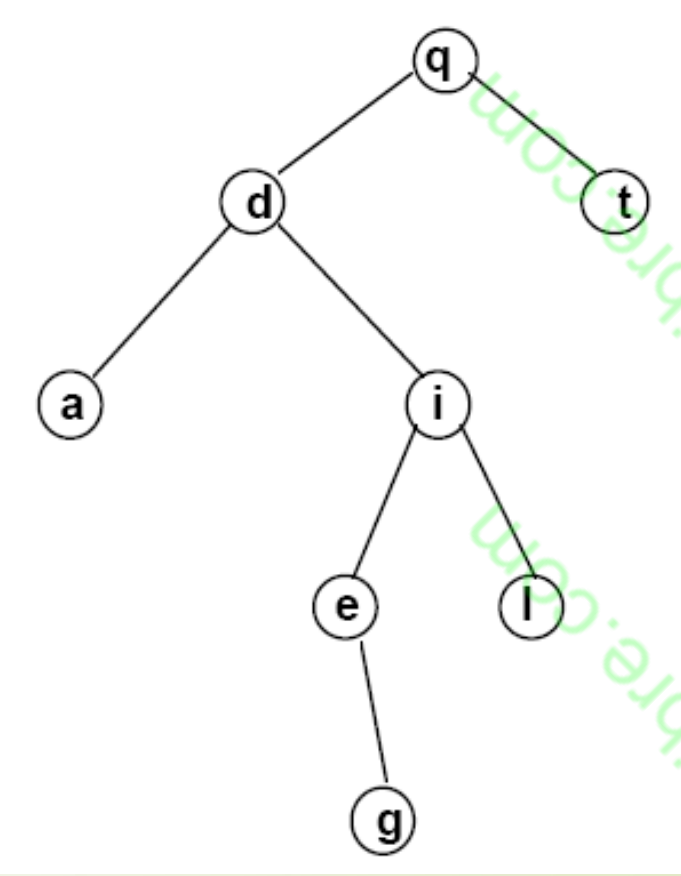
muni de l'ordre alphabétique



Arbre binaire de recherche Remarque

- **Plusieurs représentations possibles d'un même ensemble par un arbre binaire de recherche**
- **En effet, la structure précise de l'arbre binaire de recherche est déterminée :**
  - **par l'algorithme d'insertion utilisé,**
  - **et par l'ordre d'arrivée des éléments.**
- **Exemple :**
  - **L'arbre binaire de recherche de la figure qui suit représente aussi**

$$E = \{a, d, e, g, i, l, q, t\}$$



Opérations sur les arbres binaires de recherche

- Le type abstrait arbre binaire de recherche, noté `Arbre_Rech`, est décrit de la même manière que le type `Arbre_Binaire`

- On reprend les opérations de base des arbres binaires, excepté le fait que dans des arbres binaires de recherche, on suppose l'existence de l'opération clé sur le type abstrait `Element`

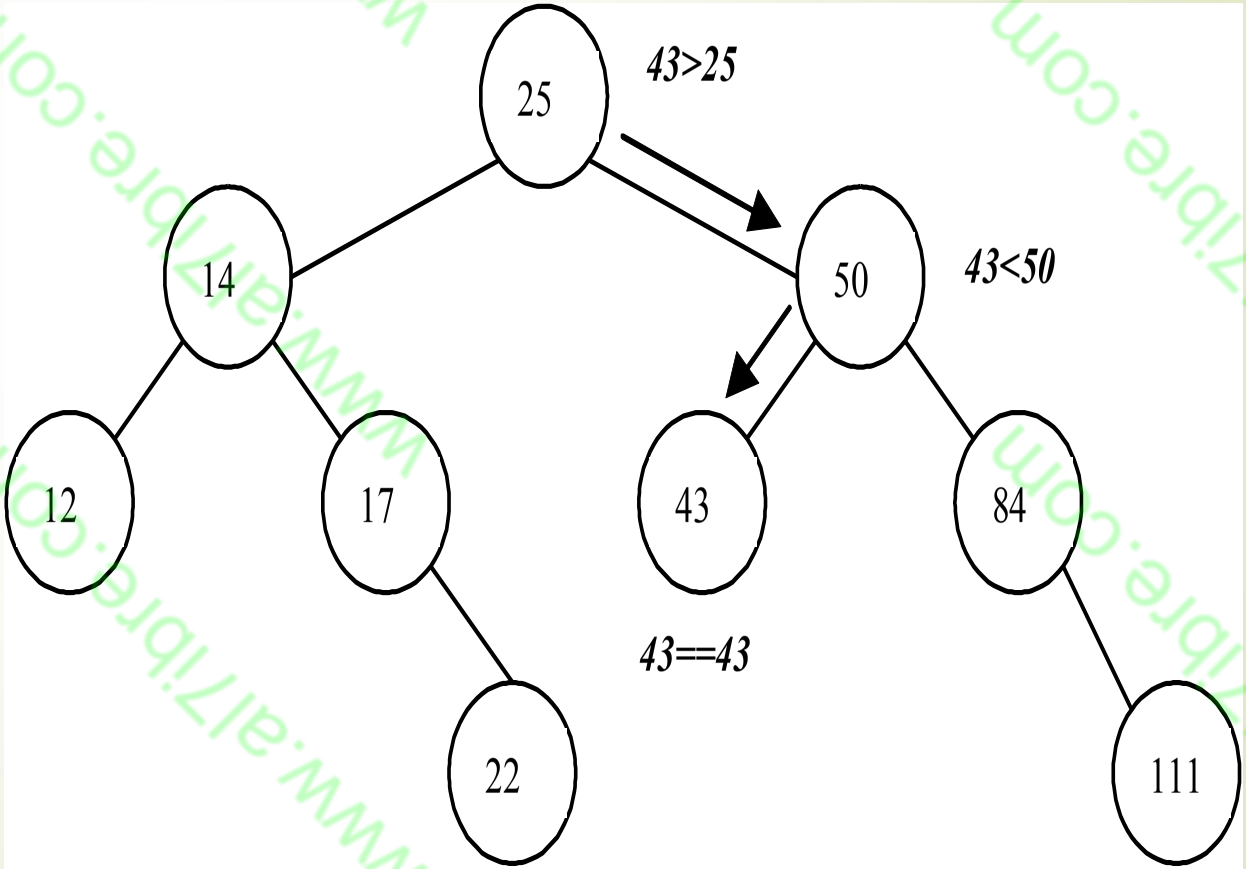
- On définit, en tenant compte du critère d'ordre, les opérations spécifiques de ce type d'arbre concernant :
  - *la recherche d'un élément dans l'arbre ;*
  - *l'insertion d'un élément dans l'arbre ;*
  - *la suppression d'un élément de l'arbre.*

Recherche d'un élément

- **Principe de l'algorithme :**
  - On compare la clé de l'élément cherché à la clé de la racine de l'arbre ;
  - Si la clé est supérieure à la clé de la racine, on effectue une recherche dans le fils droit ;
  - Si la clé est inférieure à la clé de la racine, on effectue une recherche dans le fils gauche ;
  - La recherche s'arrête quand on ne peut plus continuer (*échec*) ou quand la clé de l'élément cherché est égale à la clé de la racine d'un sous arbre (*succès*).

Recherche d'un élément Exemple

- la figure suivante illustre la recherche de l'élément de clé 43 dans un arbre binaire de recherche.
- Les flèches indiquent le chemin de la recherche



Recherche d'un élément Spécification

```
Extension Type Arbre_Rech
Utilise Elément, Booléen
Opérations
  Rechercher : Elément x Arbre_Rech → Booléen
Axiomes
  Soit, x : Elément, r : Nœud, G, D : Arbre_Rech

  Rechercher(x, arbre_vide) = faux
  si clé(x) = clé(contenu(r))
   alors Rechercher(x, <r, G, D>) = vrai
  si clé(x) < clé(contenu(r))
   alors Rechercher(x, <r, G, D>) = Rechercher(x,
  G)
  si clé(x) > clé(contenu(r))
   alors Rechercher(x, <r, G, D>) = Rechercher(x,
  D)
```

Recherche d'un élément
Réalisation en C

```
Booleen Rechercher (Arbre_Rech A, Element e) {
 if ( est_vide(A) == vrai )
   return faux; // e n'est pas dans l'arbre
 else {
   if ( e == A->val )
      return vrai; // e est dans l'arbre
   else if ( e < A->val )
           // on poursuit la recherche dans le SAG
   du
           // noeud courant
           return Rechercher(A->fg , e);
         else
           // on poursuit la recherche dans le SAD
   du
           // noeud courant
           return Rechercher(A->fd , e);
   }
}
```

Recherche d'un élément
Autre Spécification

```
Extension Type Arbre_Rech
Utilise Elément
Opérations
  Rechercher : Elément x Arbre_Rech → Arbre_Rech
Axiomes
  Soit, x : Elément, r : Nœud, G, D : Arbre_Rech

  Rechercher(x, arbre_vide) = arbre_vide
  si clé(x) = clé(contenu(r))
   alors Rechercher(x, <r, G, D>) = <r, G, D>)
  si clé(x) < clé(contenu(r))
   alors Rechercher(x, <r, G, D>) = Rechercher(x, G)
  si clé(x) > clé(contenu(r))
   alors Rechercher(x, <r, G, D>) = Rechercher(x, D)
```

Ajout d'un élément

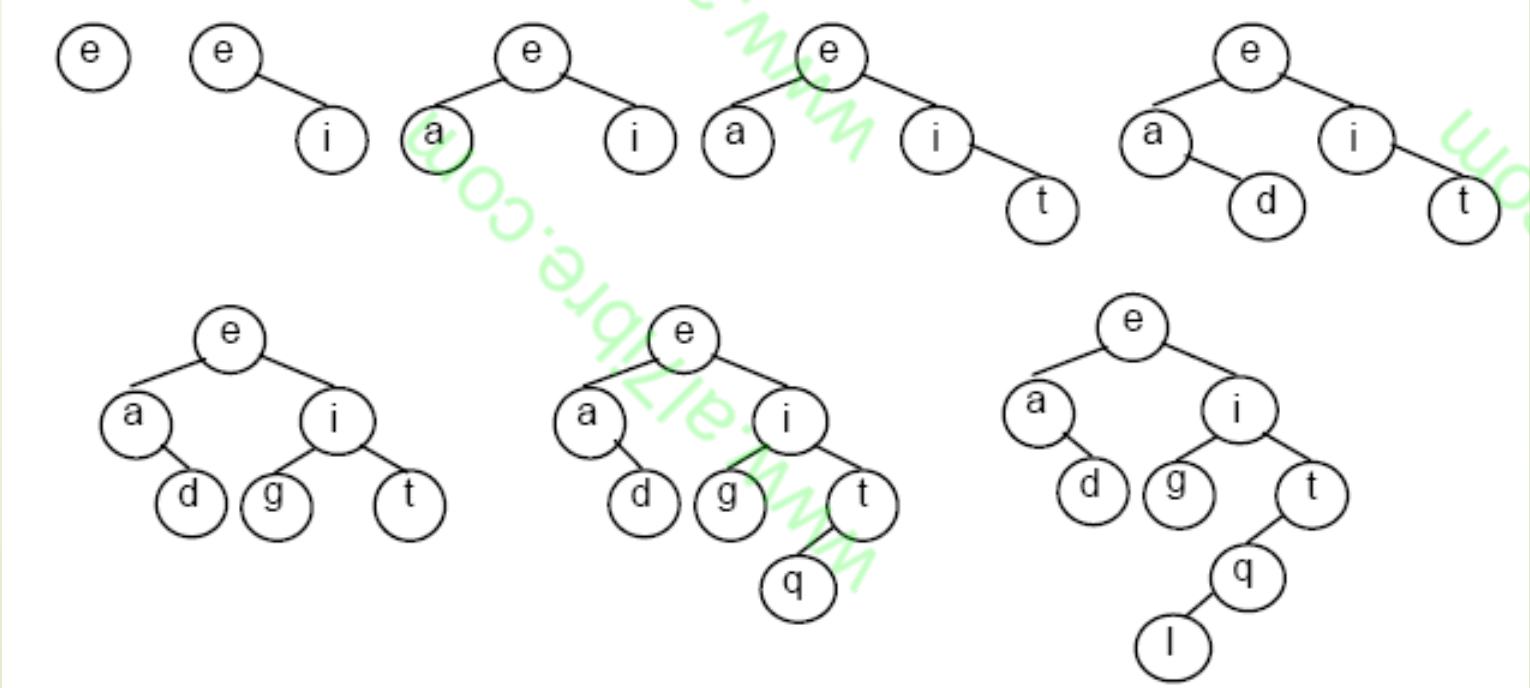
- La technique d'ajout spécifiée ici est dite "*ajout en feuille*", car tout nouvel élément se voit placé sur une feuille de l'arbre

- Le principe est simple :
  - si l'arbre initial est vide, le résultat est formé d'un arbre binaire de recherche réduit à sa racine, celle-ci contenant le nouvel élément ;
  - sinon, l'ajout se fait (*récursivement*) dans le fils gauche ou le fils droit, suivant que l'élément à ajouter est de clé inférieure ou supérieure à celle de la racine.

- Remarque :
  - 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

Ajout d'un élément
Exemple

- **Les figures suivantes illustrent l'ajout successif de e, i, a, t, d, g, q et l dans un arbre binaire de recherche, initialement vide**



Ajout "en feuille" d'un élément Spécification

Extension Type Arbre_Rech

Utilise Elément

Opérations

Ajouter_feuille : Elément x Arbre_Rech → Arbre_Rech

Axiomes

Soit, x : Elément, r : Nœud, G, D : Arbre_Rech

Ajouter_feuille(x, arbre_vide) = <x, arbre_vide, arbre_vide>

si clé(x) ≤ clé(contenu(r))

alors

Ajouter_feuille(x, <r, G, D>) = <r, Ajouter_feuille(x, G), D>

sinon

Ajouter_feuille(x, <r, G, D>) = <r, G, Ajouter_feuille(x, D)>

Ajout "en feuille" d'un élément Réalisation

```
fonction Ajouter_feuille(x : Elément, A : Arbre_Rech ) :
   Arbre_Rech
 si est_vide(A) alors
   Pnoeud r = nouveau_noeud(x)
   si est_vide(r) alors <erreur>
   retourner cons(r, arbre_vide(), arbre_vide())
 sinon
  si x > contenu(racine(A)) alors
   retourner cons(A, gauche(A), Ajouter_feuille(x, droite(A)))
  sinon
   Si x< contenu(racine(A) alors
   retourner cons(A, Ajouter_feuille(x, gauche(A)) ,droite(A))
  fsi
 fsi
fsi
ffonction
```

Ajout "en feuille" d'un élément Réalisation en C

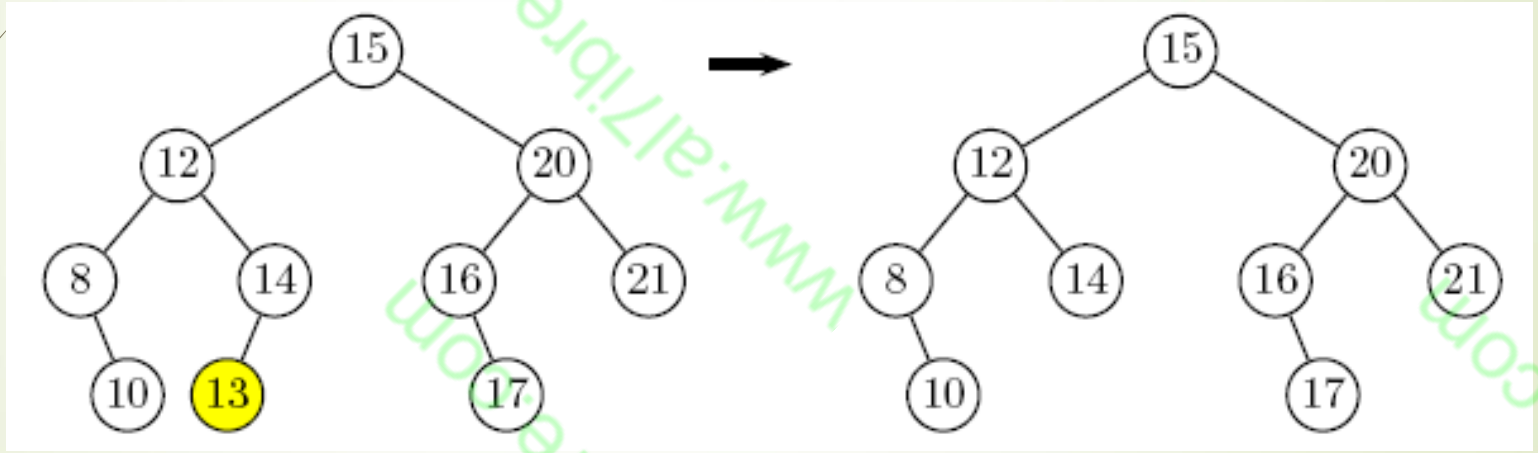
```
Arbre_Rech Ajouter_feuille(Element x, Arbre_Rech A) {
 if (est_vide(A)) {
   Pnoeud r = nouveau_noeud(x);
   if (r == NULL) {
     printf("Erreur : Pas assez de mémoire !\n");
     exit(-1);
   }
  return cons(r, arbre_vide(), arbre_vide());
 }
 else
   if (x > contenu(racine(A)))
     return cons(A, gauche(A), Ajouter_feuille(x, droite(A)));
   else
   if (x < contenu (racine(A))// pas d'ajout lorsque x=contenu(A)
     return cons(A, Ajouter_feuille(x, gauche(A)), droite(A));
}
```

Suppression d'un élément

- La suppression est délicate :
  - Il faut réorganiser l'arbre pour qu'il vérifie la propriété d'un arbre binaire de recherche
- La suppression commence par la recherche du nœud qui porte l'élément à supprimer. Ensuite, il y a trois cas à considérer, selon le nombre de fils du noeud à supprimer :
  - si le noeud est sans fils (*une feuille*), la suppression est immédiate ;
  - si le noeud a un seul fils, on le remplace par ce fils ;
  - si le noeud a deux fils (*cas général*), on choisit de remplacer ce nœud, soit par le plus grand élément de son sous arbre gauche (*son prédécesseur*), soit par le plus petit élément de son sous arbre droit (*son successeur*).

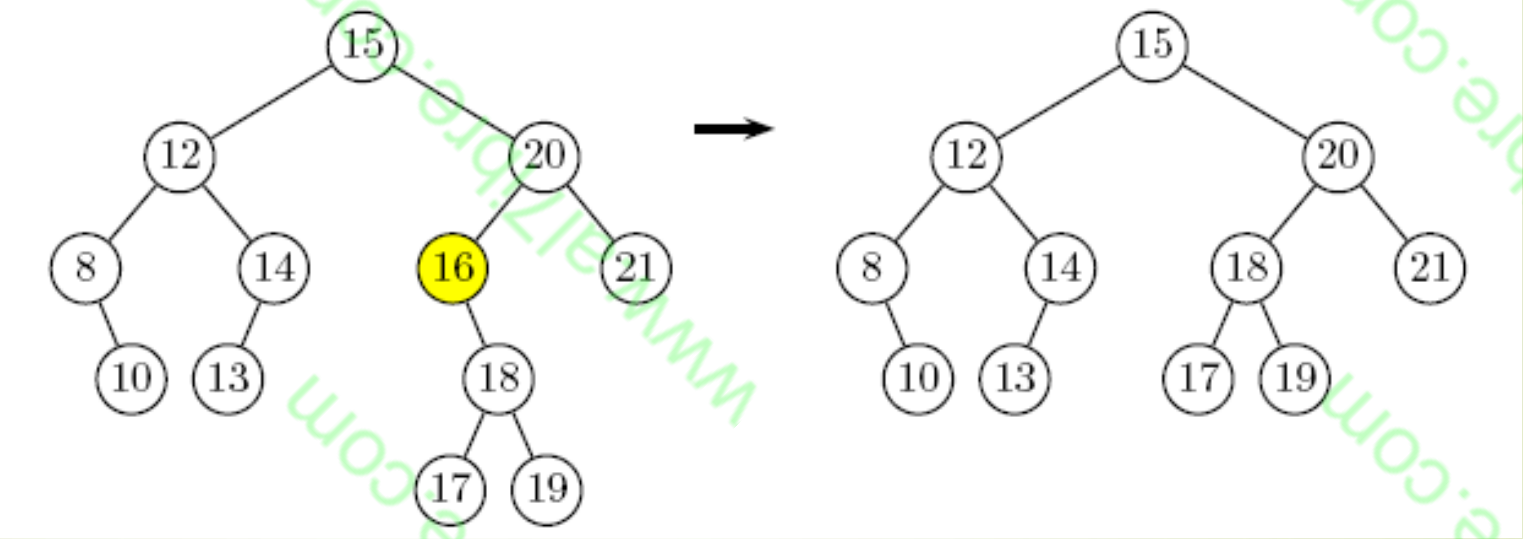
Suppression d'un élément
Exemple 1

- **La figure qui suit illustre la suppression de la feuille qui porte la clé 13**



Suppression d'un élément
Exemple 2

- **La figure qui suit illustre la suppression du nœud qui porte la clé 16**

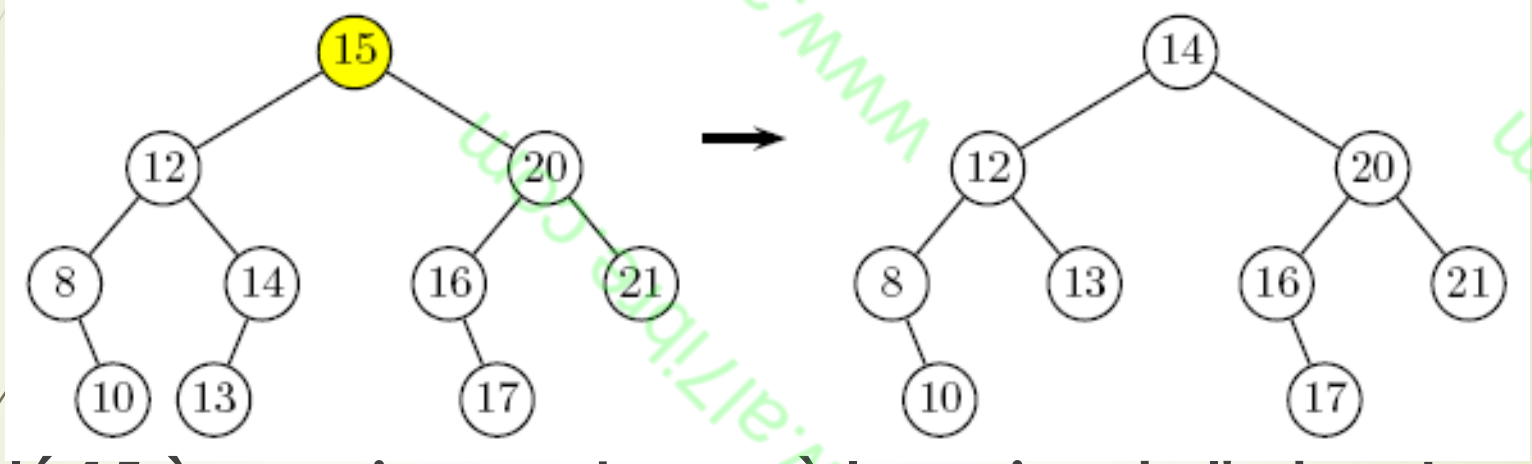


- **Ce nœud n'a qu'un seul fils ; le sous arbre de racine portant la clé 18**

- **Ce sous arbre devient fils gauche du nœud qui porte la clé 20**

Suppression d'un élément
Exemple 3

- La figure qui suit illustre le cas d'un nœud à deux fils.



- La clé 15 à supprimer se trouve à la racine de l'arbre. La racine a deux fils ; on choisit de remplacer sa clé par la clé de son prédécesseur.
- Ainsi, la clé 14 est mise à la racine de l'arbre. On est alors ramené à la suppression du nœud du prédécesseur.
- Comme le prédécesseur est le nœud le plus à droite du sous arbre gauche, il n'a pas de fils droit, donc il a zéro ou un fils, et sa suppression est couverte par les deux premiers cas.

Suppression d'un élément
Cas général

- **On choisit ici de remplacer le noeud à supprimer par son prédécesseur (*le nœud le plus à droite de son sous arbre gauche*)**

- **On a besoin de deux opérations supplémentaires :**
  - ***une opération `Max` qui retourne l'élément de clé maximale dans un arbre binaire de recherche ;***
  - ***une opération `SupprimerMax` qui retourne l'arbre privé de son plus grand élément.***

Suppression d'un élément: Spécificatior

Extension Type Arbre_Rech
Utilise Elément
Opérations

Max : Arbre_Rech → Elément
SupprimerMax : Arbre_Rech → Arbre_Rech
Supprimer : Elément x Arbre_Rech → Arbre_Rech

Pré-conditions

Max(A) *est_défini_ssi* est_vide(A) = faux
SupprimerMax(A) *est_défini_ssi* est_vide(A) = faux

Axiomes

Soit, x : Elément, r : Nœud, G, D : Arbre_Rech
si est_vide(D) = vrai alors Max(<r, G, D>) = r
sinon Max(<r, G, D>) = Max(D)
si est_vide(D) = vrai alors SupprimerMax(<r, G, D>) = G
sinon SupprimerMax(<r, G, D>) = <r, G, SupprimerMax(D)>
Supprimer(x, arbre_vide) = arbre_vide
si clé(x) = clé(contenu(r)) et est_vide(D) = vrai
alors Supprimer(x, <r, G, D>) = G
sinon si clé(x) = clé(contenu(r)) et est_vide(G) = vrai
alors Supprimer(x, <r, arbre_vide, D>) = D
sinon si clé(x) = clé(contenu(r))
alors Supprimer(x, <r, G, D>) = <Max(G),SupprimerMax(G), D>
si clé(x) < clé(contenu(r))
alors Supprimer(x, <r, G, D>) = <r, Supprimer(x, G), D>
si clé(x) > clé(contenu(r))
alors Supprimer(x, <r, G, D>) = <r, G, Supprimer(x, D)>

Suppression d'un élément Réalisation

```
fonction Max(A : Arbre_Rech) : Pnoeud
(* A doit être non vide ! *)
 si est_vide(droite(A))
  alors retourner A


 sinon retourner Max(droite(A))
 fsi
ffonction

fonction SupprimerMax(A : Arbre_Rech) : Arbre_Rech
(* A doit être non vide ! *)
 si est_vide(droite(A))
 alors
  retourner gauche(A)
 sinon
  retourner cons(A, gauche(A), SupprimerMax(droite(A)))
 fsi
ffonction
```

Cette fonction retourne un pointeur sur le nœud contenant la plus grand élément d'un arbre binaire de recherche

Cette fonction supprime le plus grand élément d'un arbre binaire de recherche

Suppression d'un élément Réalisation (suite)

```
fonction Supprimer(x : Elément, A : Arbre_Rech) : Arbre_Rech
 si est_vide(A) alors retourner A   (* ou <erreur> *)
 sinon
  si x > contenu(racine(A)) alors
   retourner cons(A, gauche(A), Supprimer(x ,droite(A)))
  sinon
   si x < contenu(racine(A)) alors
    retourner cons(A, Supprimer(x, gauche(A)), droite(A))
   sinon // x= contenu (racine(A))
    si est_vide(droite(A)) alors retourner gauche(A)
    sinon
     si est_vide(gauche(A)) alors retourner droite(A)
     sinon // ni droite (A) est vide ni gauche(A)
      retourner cons(Max(gauche(A)), SupprimerMax(gauche(A)), droite(A))
     fsi
    fsi
   fsi
  fsi
 fsi
ffonction
```

Arbre Binaire de Recherche Complexité des Opérations

- **On montre que, les opérations de recherche, insertion et suppression dans un arbre binaire de recherche contenant n éléments sont :**
  - en moyenne en $O(\log_2(n))$;
  - dans le pire des cas en $O(h)$;

  où h désigne la hauteur de l'arbre

- **Si l'arbre est dégénéré, sa hauteur étant n-1, ces trois opérations sont en O(n)**

- **Si l'arbre est équilibré, les opérations sont en $O(\log_2(n))$ (*d'où leur intérêt...*)**

Arbres Maximiers ou Tas (Heaps)

Notion d'Arbre Maximier (ou Tas)

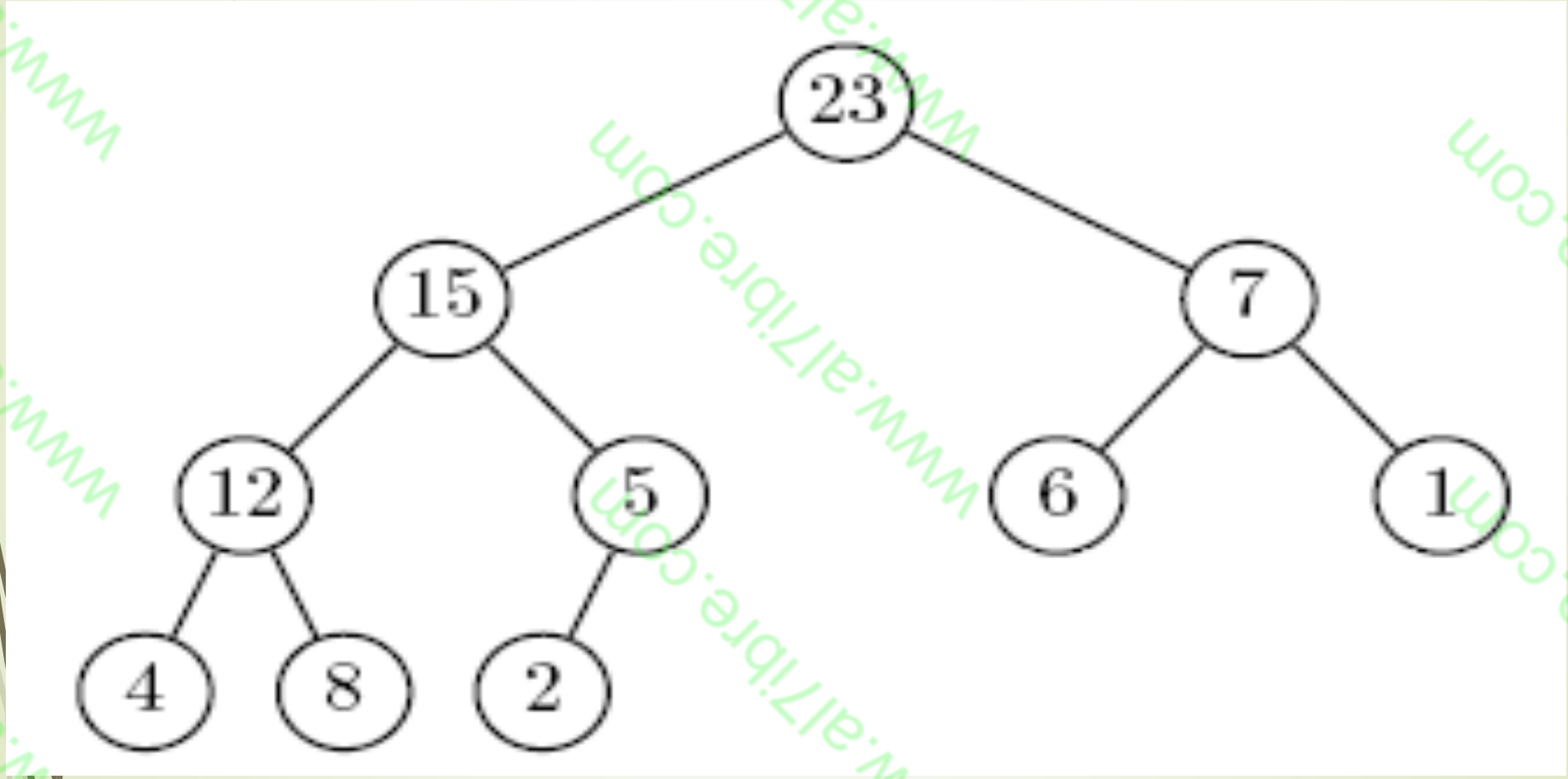
- Appelé aussi monceau (*Heap en anglais)*
- C'est un arbre binaire parfait tel que la clé de chaque noeud est supérieure ou égale aux clés de tous ses fils
- L'élément maximum de l'arbre se trouve donc à la racine

- Rappel :
  - Pour un arbre binaire parfait, tous les niveaux sont entièrement remplis sauf éventuellement le dernier et, dans ce cas, les feuilles du dernier niveau sont regroupées le plus à gauche possible

- Un tas est *un arbre binaire partiellement ordonné* :
  - Les nœuds sur chaque branche sont ordonnés sur celle-ci ;
  - Ceux d'un même niveau ne le sont pas nécessairement.

- Un tas dans lequel chaque nœud enfant a une clé inférieure (*resp., supérieure*) ou égale à la clé de son père est appelé *arbre maximier* (*max heap*) (*resp., arbre minimier (max heap)*)

Arbre Maximier (ou Tas) Exemple



Type Abstrait Tas

Type Tas

Utilise Booléen, Elément

Opérations

tas_vide : → Tas
est_vide : Tas → Booléen
max : Tas → Elément
ajouter : Tas x Elément → Tas
supprimerMax : Tas → Tas
appartient : Tas x Elément → Booléen

Préconditions

max(T) *est_défini_ssi* est_vide(T) = faux
supprimerMax(T) *est_défini_ssi* est_vide(T) = faux
ajouter(T,e) *est_défini_ssi* appartient(T,e) = faux

Axiomes

Soit, T, T1 : Tas, e : Elément

si est_vide(T) = vrai alors appartient(T,e) = faux

appartient(T,max(T)) = vrai

si appartient(T,e) = vrai alors max(T) ≥ e

Opérations sur un Tas

- **tas_vide : → Tas**
  - Opération d'initialisation; crée un tas vide
- **est_vide : Tas → Booléen**
  - Vérifie si un tas est vide ou non
- **max : Tas → Elément**
  - Retourne le plus grand élément d'un tas
- **ajouter : Tas x Elément → Tas**
  - Ajoute un élément dans un tas
- **supprimerMax : Tas → Tas**
  - Supprime le plus grand élément d'un tas
- **appartient : Tas x Elément → Booléen**
  - Vérifie si un élément appartient ou non à un tas

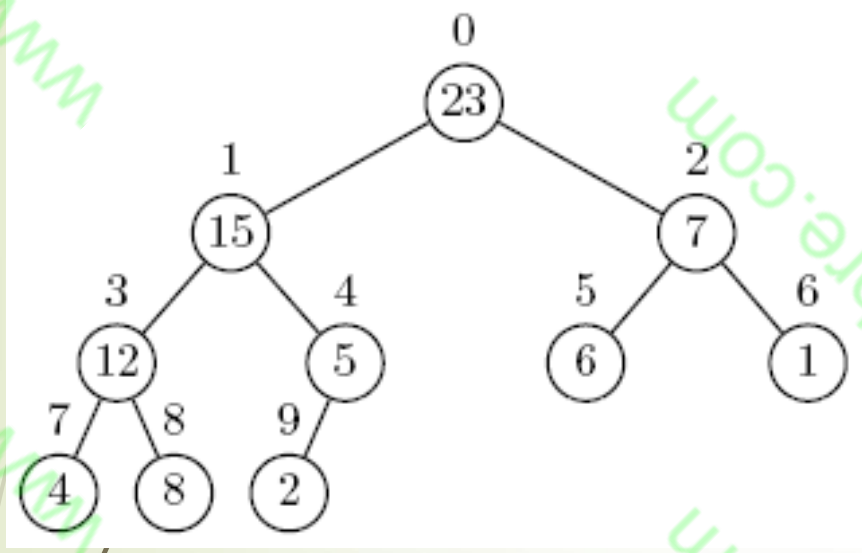
Représentation d'un Tas

- Il existe une représentation compacte pour les arbres binaires parfaits, et donc pour les tas :
  - La représentation par tableau, basée sur la numérotation des nœuds niveau par niveau et de gauche à droite
- Les numéros d'un nœud sont donc les indices dans un tableau. En outre, ce tableau s'organise de la façon suivante :
  - le noeud racine a pour indice 0 ;
  - soit le noeud d'indice i dans le tableau, son fils gauche a pour indice 2i +1, et son fils droit a pour indice 2(i+1) ;
  - si un nœud a un indice i ≠ 0, alors son père a pour indice

$$\lfloor (i-1)/2 \rfloor$$

- On déduit de cette organisation, où n désigne le nombre d'éléments du tas, que :
  - un nœud d'indice i est une feuille si 2i+1 ≥ n
  - un nœud d'indice i a un fis droit si 2(i+1) < n

Représentation d'un Tas Exemple



Un tas avec sa numérotation hiérarchique

0	1	2	3	4	5	6	7	8	9
23	15	7	12	5	6	1	4	8	2

Représentation du tas par un tableau

Représentation en C d'un Tas

```
#define MAX_ELEMENTS 200  // taille maximum du tas
typedef int Element   // un élément est un int
typedef struct {
 int taille;  // nombre d'éléments dans le tas
 Element tableau[MAX]; // les éléments du tas
} Tas;
```

Opérations sur un Tas

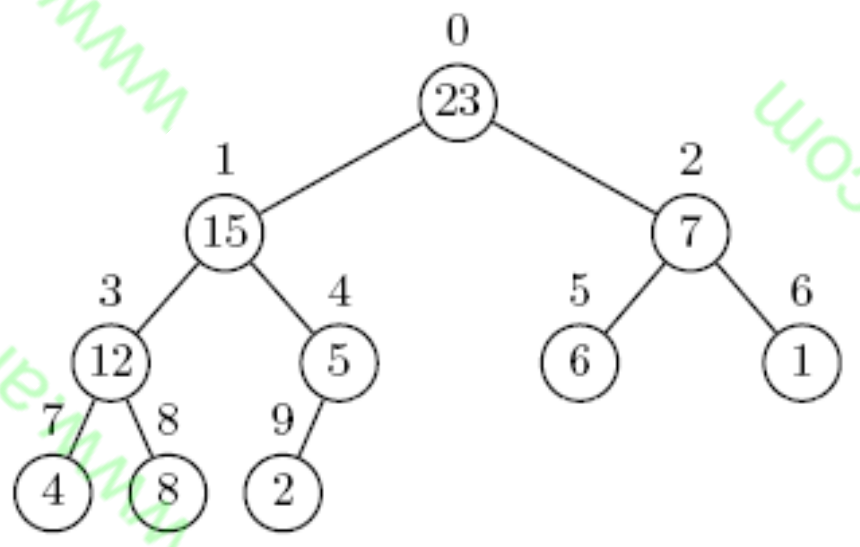
- **Trois opérations fondamentales :**
  - ***Ajout d'un élément ;***
  - ***Suppression du maximum ;***
  - ***Recherche du maximum.***

Opération d'Ajout

- ***Principe :***
  - Créer un nouveau nœud contenant la clé du nouvel élément ;
  - Insérer cette clé le plus à gauche possible sur le dernier niveau du tas (*ou si le dernier niveau est plein, à l'extrême gauche d'un nouveau niveau*). La nouvelle clé est insérée dans la première case non utilisée du tableau ;
  - Faire "remonter cette nouvelle clé" à sa place en la permutant avec la clé de son père, tant qu'elle est plus grande que celle de son père.

Opération d'Ajout Exemple (1)

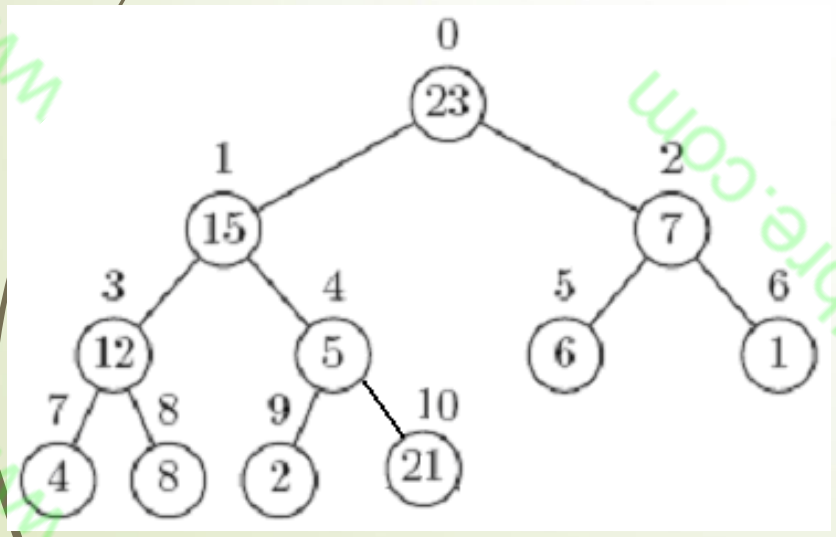
- **Supposons qu'on veuille insérer la valeur 21 dans le tas représenté ci-dessous :**
  - **On place la valeur 21 juste à droite de la dernière feuille,**
  - **c'est-à-dire dans la case d'indice 10 dans le tableau.**



0	1	2	3	4	5	6	7	8	9
23	15	7	12	5	6	1	4	8	2

Opération d'Ajout
Exemple (2)

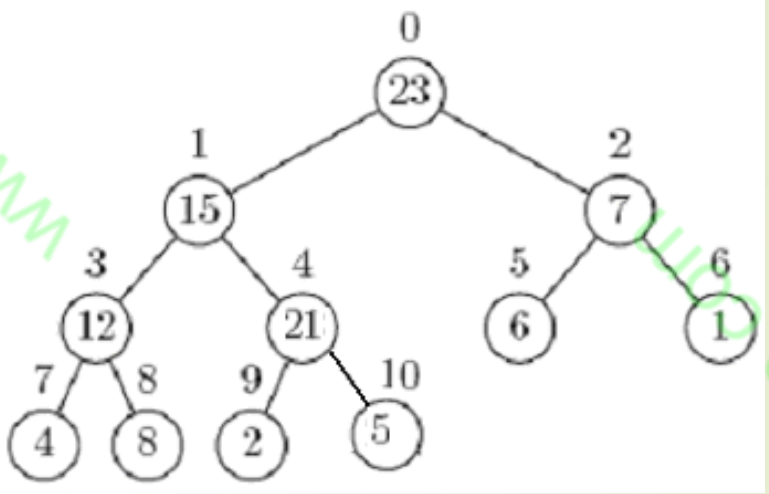
- **On compare 21, la nouvelle donnée insérée, avec la donnée contenue dans le nœud père, autrement dit on compare la donnée de la case d'indice 10 du tableau avec la donnée de la case d'indice = 4.**
  - **Puisque 21 est plus grand que 5, on les échange.**



0	1	2	3	4	5	6	7	8	9	10
23	15	7	12	5	6	1	4	8	2	21

Opération d'Ajout Exemple (3)

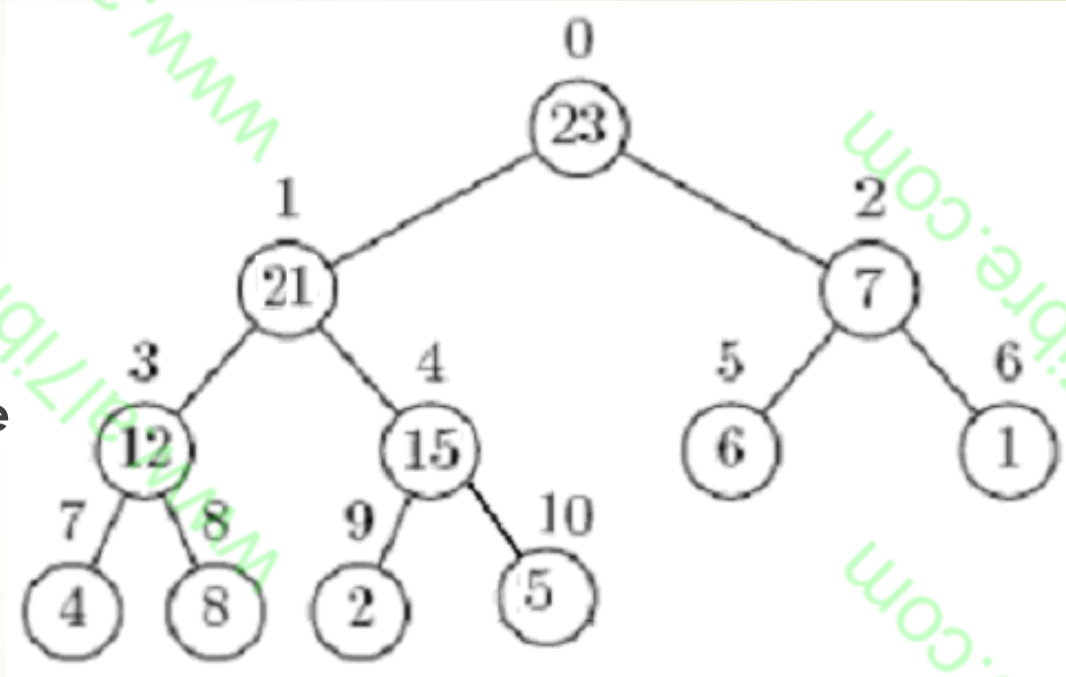
- **Le nouvel arbre binaire obtenu n'est pas un tas :**
  - La valeur 21 du nœud d'indice 4 est plus grande que la valeur 15 de son nœud père (*d'indice* = 1)
  - Echanger les contenus des nœuds d'indices respectifs 1 et 4



0	1	2	3	4	5	6	7	8	9	10
23	15	7	12	21	6	1	4	8	2	5

Opération d'Ajout Exemple (4)

- **Puisque 21 est plus petit que 23 :**
  - L'opération d'ajout est terminée
  - On a bien obtenu un tas.



0	1	2	3	4	5	6	7	8	9	10
23	21	7	12	15	6	1	4	8	2	5

Opération d'Ajout Pseudo-code

```
fonction ajouter(Tas t, Elément e) : Tas
début
   i ← t.taille
   t.taille ← i+1
   t.tableau[i] ← e
   tant que ((i > 0) et
         (t.tableau[i] > t.tableau[(i-1) div 2]))
   faire {
      échanger(t.tableau[i], t.tableau[(i-1) div
   2]
      i ← (i-1) div 2
   }
   retourner (t)
fin
```

Opération d'Ajout Complexité

- **La complexité de l'opération d'ajout est en O(h), où h est la hauteur du tas :**
  - **On ne fait que remonter un chemin ascendant d'une feuille vers la racine (en s'arrêtant éventuellement avant).**
  - **La hauteur d'un tas de taille n est précisément égale à** $\lfloor \log_2(n) \rfloor$

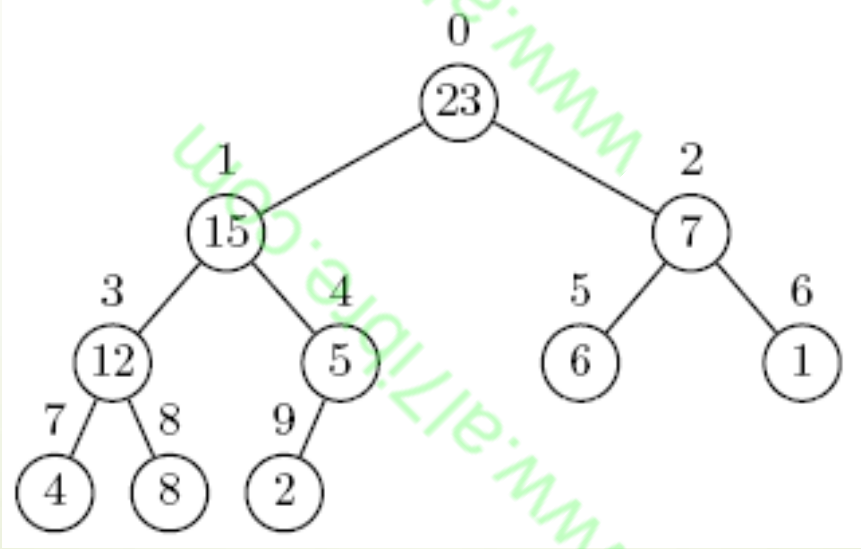
et donc l'ajout demande un temps O(log(n)).

Opération de Suppression du Maximum

- **Principe :**
  - ***Remplacer la clé du nœud racine par la clé du nœud situé le plus à droite du dernier niveau du tas. Ce dernier nœud est alors supprimé ;***
  - ***Réorganiser l'arbre, pour qu'il respecte la définition du tas, en faisant descendre la clé de l'élément de la racine à sa bonne place en permutant avec le plus grand des fils.***

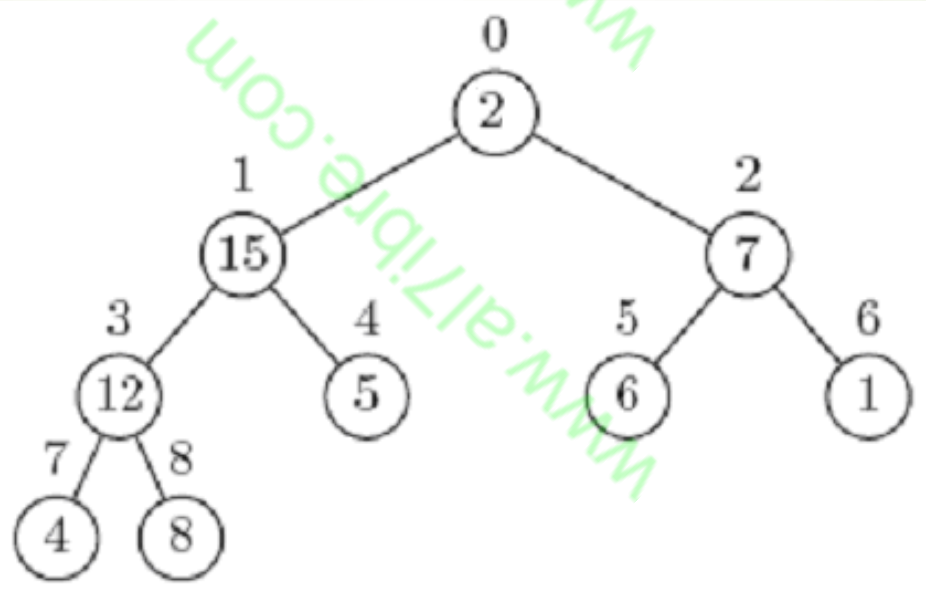
Opération de Suppression du Maximum (Exemple) (1)

- ***Supposons qu'on désire supprimer la valeur 23 contenue dans la racine du tas illustré par la figure suivante :***



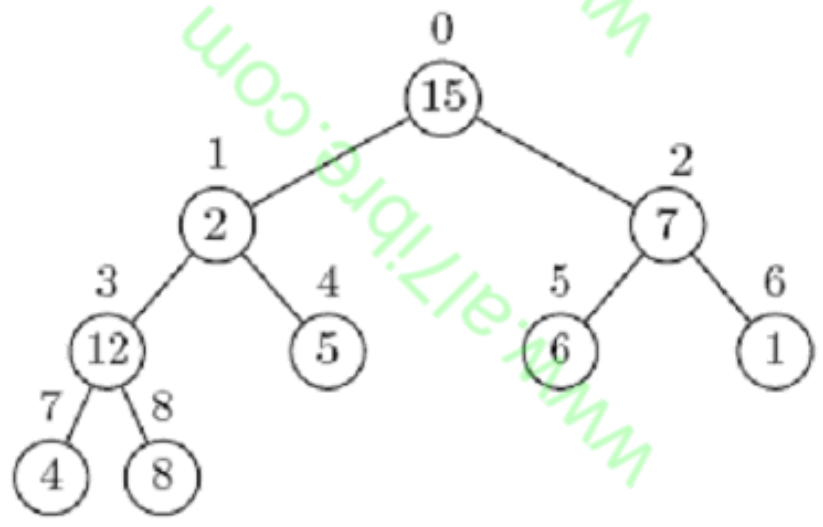
Opération de Suppression du Maximum (Exemple) (2)

- ***On commence alors par remplacer le contenu du nœud racine par celui du dernier nœud du tas :***
  - ***Ce dernier nœud est alors supprimé ;***
  - ***Ceci est illustré par la figure suivante :***



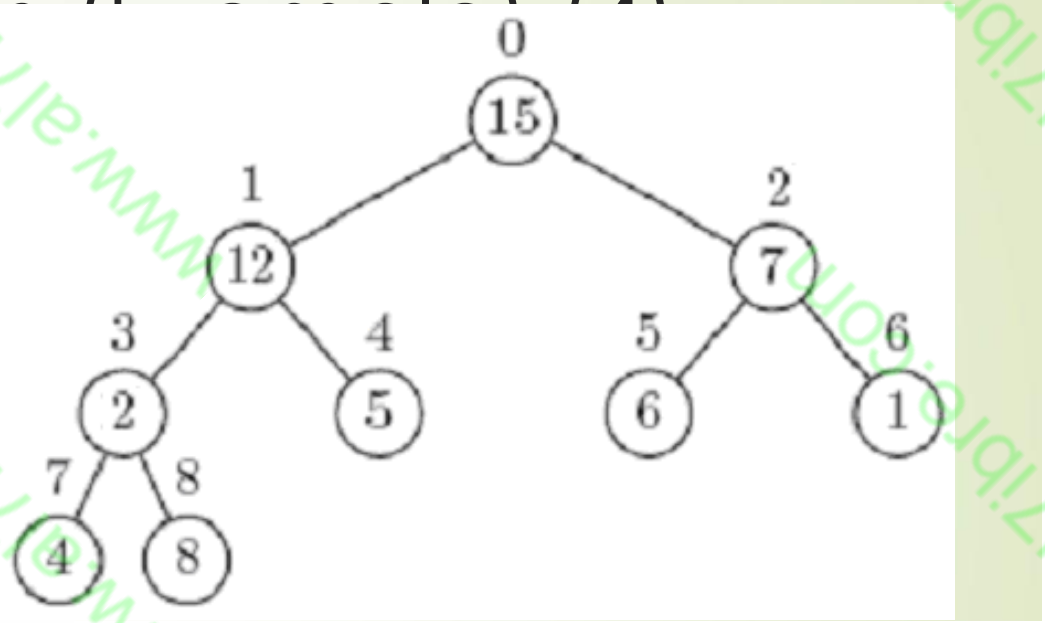
Opération de Suppression du Maximum (Exemple) (3)

- ***L'arbre obtenu est parfait mais n'est pas un tas :***
  - *la clé contenue dans la racine a une valeur plus petite que les valeurs des clés de ses fils ;*
  - *Cette clé de valeur 2 est alors échangée avec la plus grande clé de ses fils, à savoir 15 ;*
  - *L'arbre obtenu est représenté par la figure suivante :*

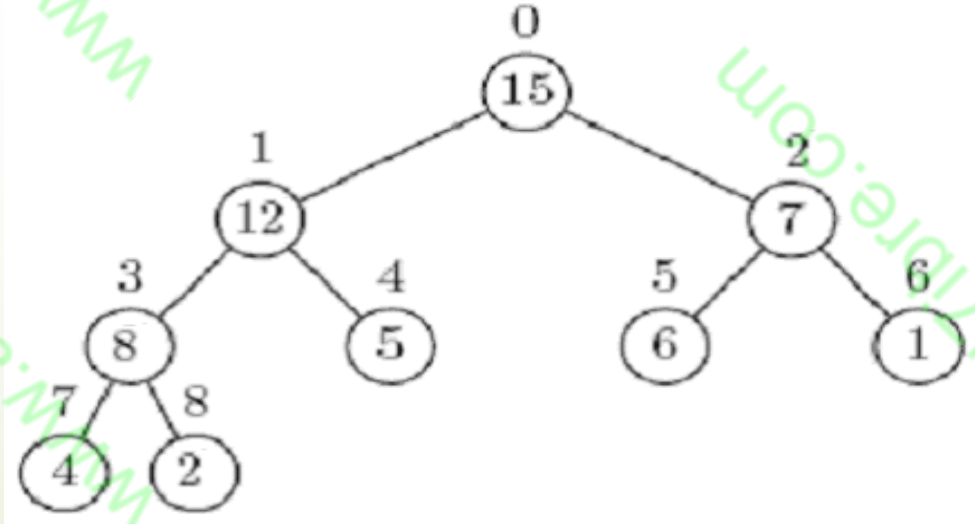


Opération de Suppression du Maximum (Exemple) (4)

- ***Encore une fois, cet arbre n'est pas un tas. On le réorganise pour qu'il respecte la définition du tas***



- ***Le dernier arbre obtenu est bien un tas ; il est illustré par la figure suivante :***



Opération de Suppression du Maximum (Pseudo-code) (1)

- **Une version qui utilise la procédure *Entasser***

- **La procédure *Entasser* :**
  - **permet de faire descendre la valeur en t[i] de manière que l'arbre de racine en i devienne un tas ;**
  - **suppose que les sous arbres de racines en *2i+1 (fils gauche du nœud en i) et en 2i+2* (*fils droit du nœud en i*) sont des tas.**

Opération de Suppression du Maximum (Pseudo-code) (2)

```
procédure Entasser(Tableau t[0 .. n-1], Entier i)
début
   si ((2i+2 == n) ou (t[2i+1] ≥ t[2i+2])) alors
      k ← 2i+1
   sinon
      k ← 2i+2
   fsi
   si t[i] < t[k] alors
      échanger (t[i], t[k])
      si k ≤ ((n div 2)- 1) alors
         Entasser(t, k)
      fsi
   fsi
fin
```

Opération de Suppression du Maximum (Pseudo-code) (3)

```
fonction supprimerMax (t : tas) : tas
(* le tas t est supposé non vide !! *)

début
   t.taille ← t.taille - 1
   t.tableau[0] ← t.tableau[t.taille]
   Entasser(t, 0)
   retourner (t)
fin
```

Opération de Suppression du Maximum (Complexité)

- **La complexité de la suppression est la même que celle de l'insertion, c-à-d O(log(n)) :**
  - ***En effet, on ne fait que suivre un chemin descendant depuis la racine.***

Opération de Recherche du Maximum (Pseudo-code & Complexité)

- ***L'opération de recherche du maximum est immédiate dans les tas***
- ***Elle prend un temps constant O(1)***

```
fonction Max (t : tas) : Elément
(* le tas t est supposé non vide !! *)
début
 retourner (t.tableau[0])
fin
```

Exemples d'Applications des Tas

- **Files de priorités (*Priority queues*) :**
  - *Les tas sont fréquemment utilisés pour implémenter des files de priorités.*
  - *A l'opposé des files standard, une file de priorités détruit l'élément de plus haute (ou plus basse) priorité.*
  - *La signification de la "priorité" d'un élément dépend de l'application*
  - *A tout instant, on peut insérer un élément de priorité arbitraire dans une file de priorités. Si l'application souhaite la destruction de l'élément de plus haute priorité, on utilise un arbre maximier.*

- **Tri par tas (*Heapsort*):**
  - *Les opérations sur les tas permettent de résoudre un problème de tri à l'aide d'un algorithme appelé tri par tas (heapsort).*
  - *Cet algorithme a la même complexité temporelle, O(n log(n)), que le tri rapide (quicksort). Mais, en pratique, une bonne implémentation de ce dernier le bat d'un petit facteur constant.*

Algorithme du Tri par Tas (Principe)

- **Supposons qu'on veut trier, en ordre croissant, un tableau T de n éléments.**

- **Principe :**
  - **L'algorithme du tri par tas commence, en utilisant la fonction *ConstruireTas*, par construire un tas dans le tableau à trier T ;**
  - **Ensuite, il prend l'élément maximal du tas, qui se trouve en T[0], l'échange avec T[n-1], et rétablit la propriété de tas, en utilisant l'appel de fonction *Entasser(T,0)* pour le nouveau tableau à n-1 éléments (*la case T[n-1] n'est pas considérée*) ;**
  - **L'algorithme de tri par tas répète ce processus pour le tas de taille n-1 jusqu'à la taille 2.**

Algorithme du Tri par Tas (Pseudo-code) (1)

```
fonction Tri_par_Tas(Tableau T[0 .. n-1]) :
   Tableau
début
   T ← ConstruireTas(T)
   pour i ← (n-1) à 1 par pas -1 faire
      Echanger(T[0], T[i])
      n ← n-1
      Entasser (T, i)
   retourner (T)
fin
```

ConstruireTas produit un tas à partir d'un tableau T

Entasser sert à garantir le maintien de la propriété de tas pour l'arbre de racine en i

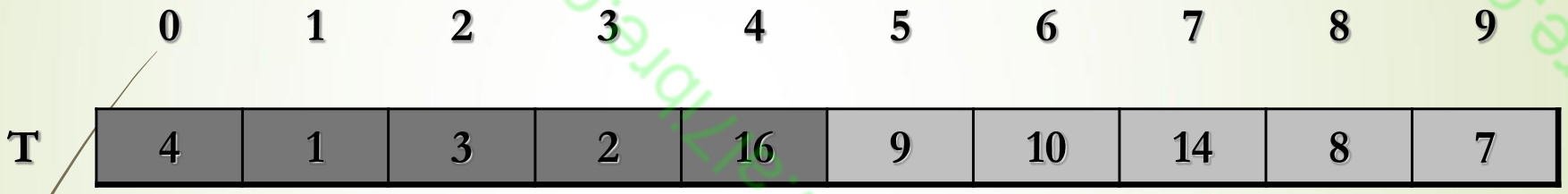
Algorithme du Tri par Tas (Pseudo-code) (2)

```
fonction ConstruireTas(Tableau T[0 .. n-1]) : Tas
début
   pour i ← ((n div 2) - 1) à 0 par pas -1 faire
    Entasser (T, i)
   retourner (T)
fin
```

Les feuilles sont des tas à un élément !

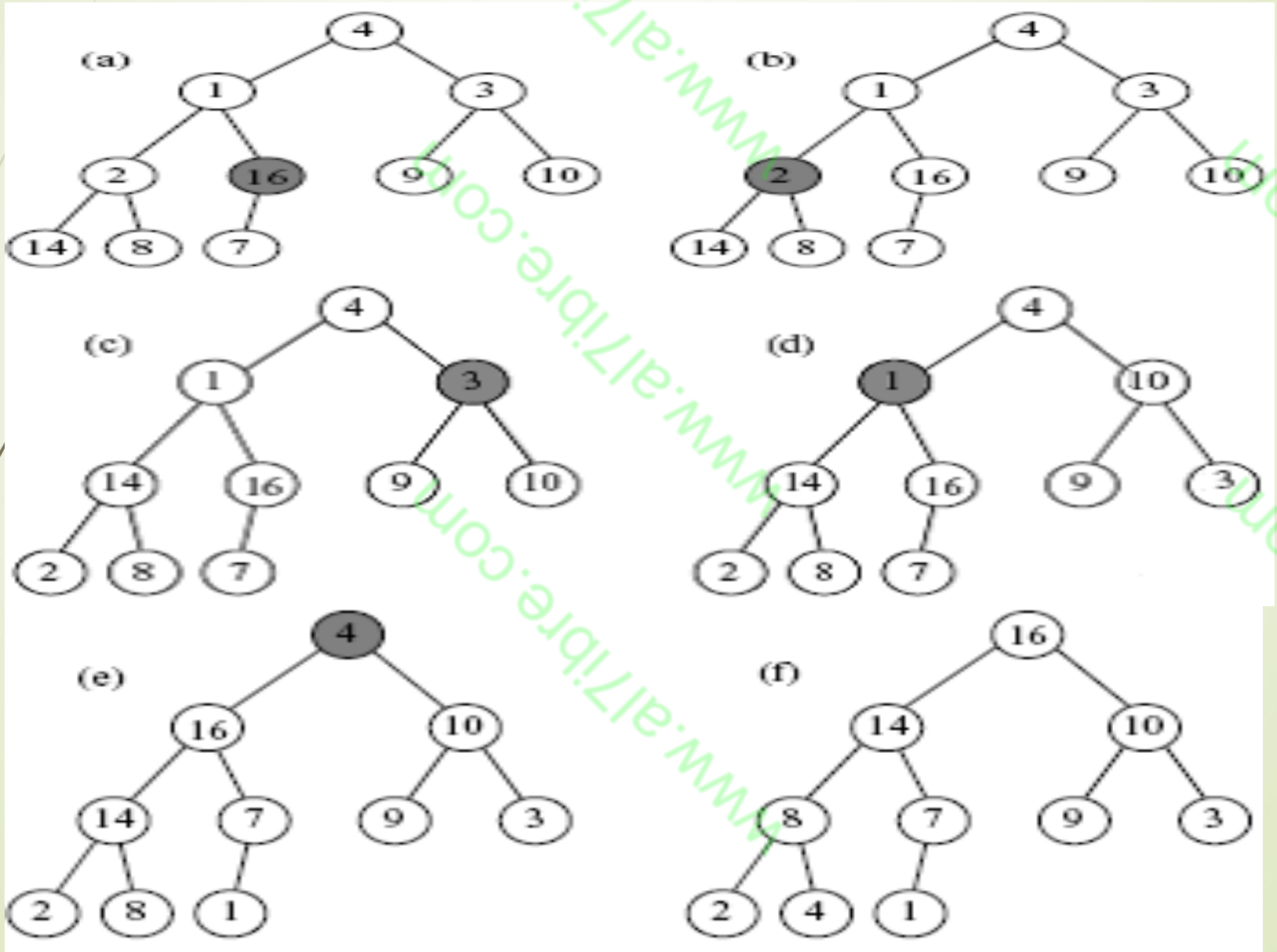
ConstruireTas Exemple (1)

- ***Illustration de l'action ConstruireTas sur un tableau d'entiers contenant 10 éléments***



- ***Remarquer que les nœuds qui portent les valeurs 9, 10, 14, 8 et 7 sont biens des feuilles, et donc des tas à un élément.***

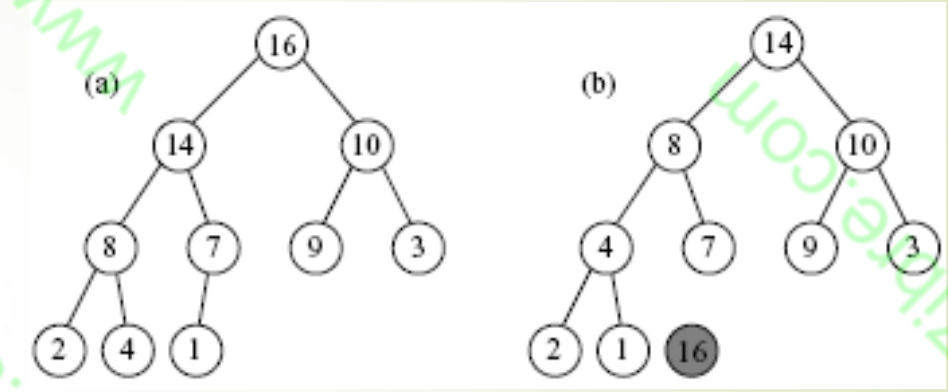
ConstruireTas: Exemple (2)

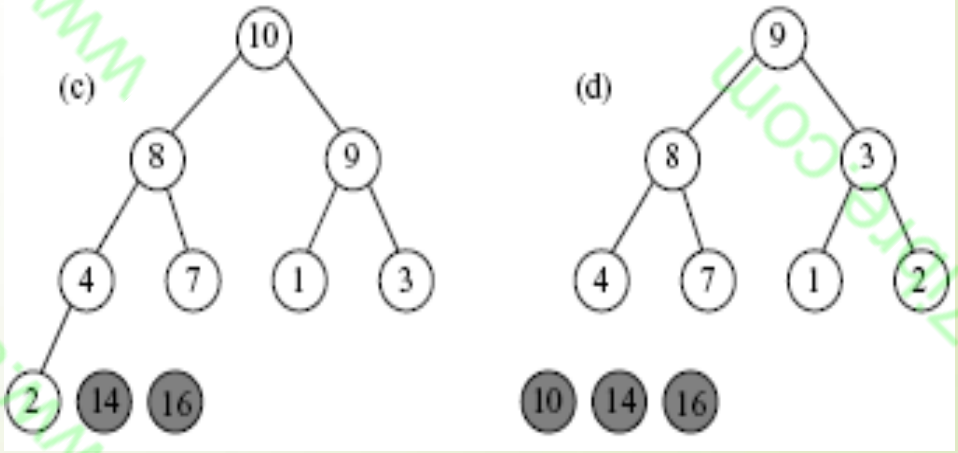


Tri par Tas : Exemple (1)

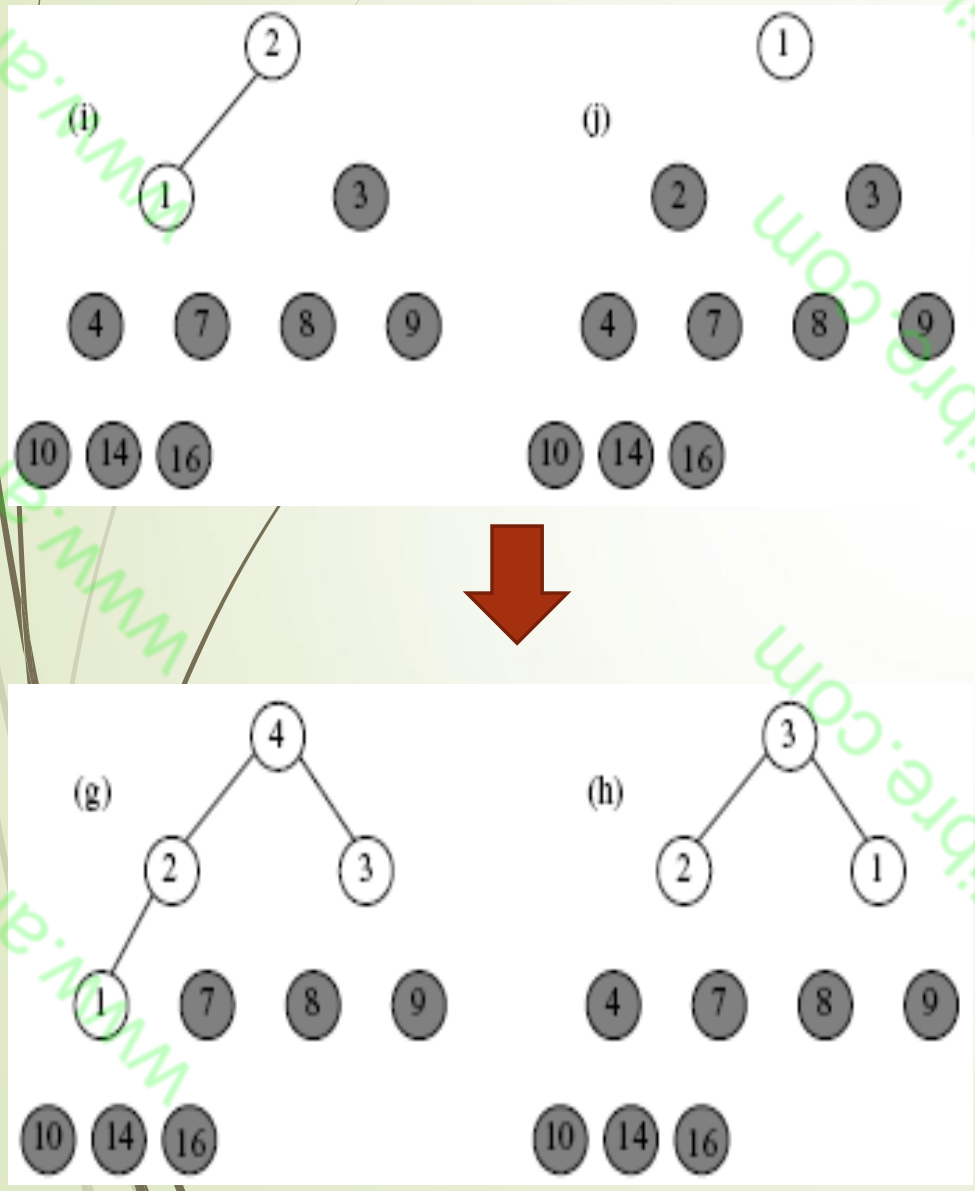
- ***Les figures qui suivent illustrent l'action du tri par tas après construction du tas***

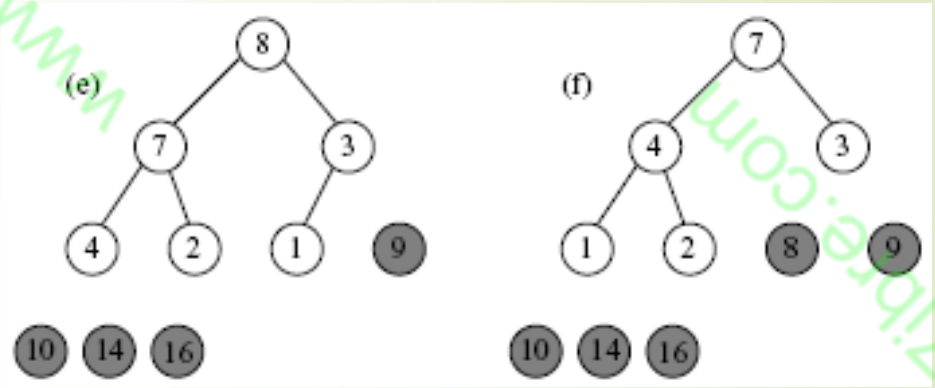
- ***Chaque tas est montré au début d'une itération de la boucle***





Tri par Tas : Exemple (2)





	0	1	2	3	4	5	6	7	8	9
T	1	2	3	4	7	8	9	10	14	16

Tableau final : trié

Algorithme du Tri par Tas Complexité

- **On montre que l'appel à ConstruireTas prend un temps O(n)**
- ***Chacun des (n-1) appels à Entasser prend un temps O(log(n))***
- **Par conséquent, l'algorithme du tri par tas s'exécute en O(n log(n))**

Introduction aux Arbres de Recherche Equilibrés

- ***(Balanced Search Trees)***

Notion d'Arbres de Recherche Equilibrés

- *La définition des arbres équilibrés impose que la différence entre les hauteurs des fils gauche et des fils droit de tout noeud ne peut excéder 1*
- *Il faut donc maintenir l'équilibre de tous les noeuds au fur et à mesure des opérations d'insertion ou de suppression d'un nœud*
- *Quand il peut y avoir un déséquilibre trop important entre les deux fils d'un noeud, il faut recréer un équilibre par :*
  - *des rotations d'arbres ou par éclatement de nœuds (cas des arbres B)*
- *Les algorithmes de rééquilibrage sont très compliqués :*
  - *On cite entre autres, quelques exemples d'arbres équilibrés pour les quels les opérations de recherche, d'insertion et de suppression sont en O(log(n))*

Arbres de Recherche Equilibrés Exemples (1)

- ***Les arbres AVL :***
  - *Introduits par Adelson-Velskii Landis Landis (d'où le nom d'AVL) dans les années 60 ;*
  - *Un arbre AVL est un arbre binaire de recherche stockant une information supplémentaire pour chaque noeud : son facteur d'équilibre ;*
  - *Le facteur d'équilibre représente la différence des hauteurs entre son sous arbre gauche et son sous arbre droit ;*
  - *Au fur et à mesure que des nœuds sont insérés ou supprimés, un arbre AVL s'ajuste de lui-même pour que tous ses facteurs d'équilibres restent à 0, -1 ou 1.*

Arbres de Recherche Equilibrés Exemples (2)

- ***Les arbres rouges et noirs :***
  - ***Des arbres binaires de recherche qui se maintiennent eux-mêmes approximativement équilibrés en colorant chaque nœud en rouge ou noir ;***
  - ***En contrôlant cette information de couleur dans chaque noeud, on garantit qu'aucun chemin ne peut être deux fois plus long qu'un autre, de sorte que l'arbre reste équilibré.***

Arbres de Recherche Equilibrés Exemples (3)

- ***Les B arbres :***
  - *Arbres de recherche équilibrés qui sont conçus pour être efficaces sur d'énormes masses de données stockées sur mémoires secondaires ;*
  - *Chaque nœud permet de stocker plusieurs clés ;*
  - *Généralement, la taille d'un nœud est optimisée pour coïncider avec la taille d'un bloc (ou page) du périphérique, en vue d'économiser les coûteux accès d'entées sorties.*
- ...

Les Structures de Données

Les arbres AVL

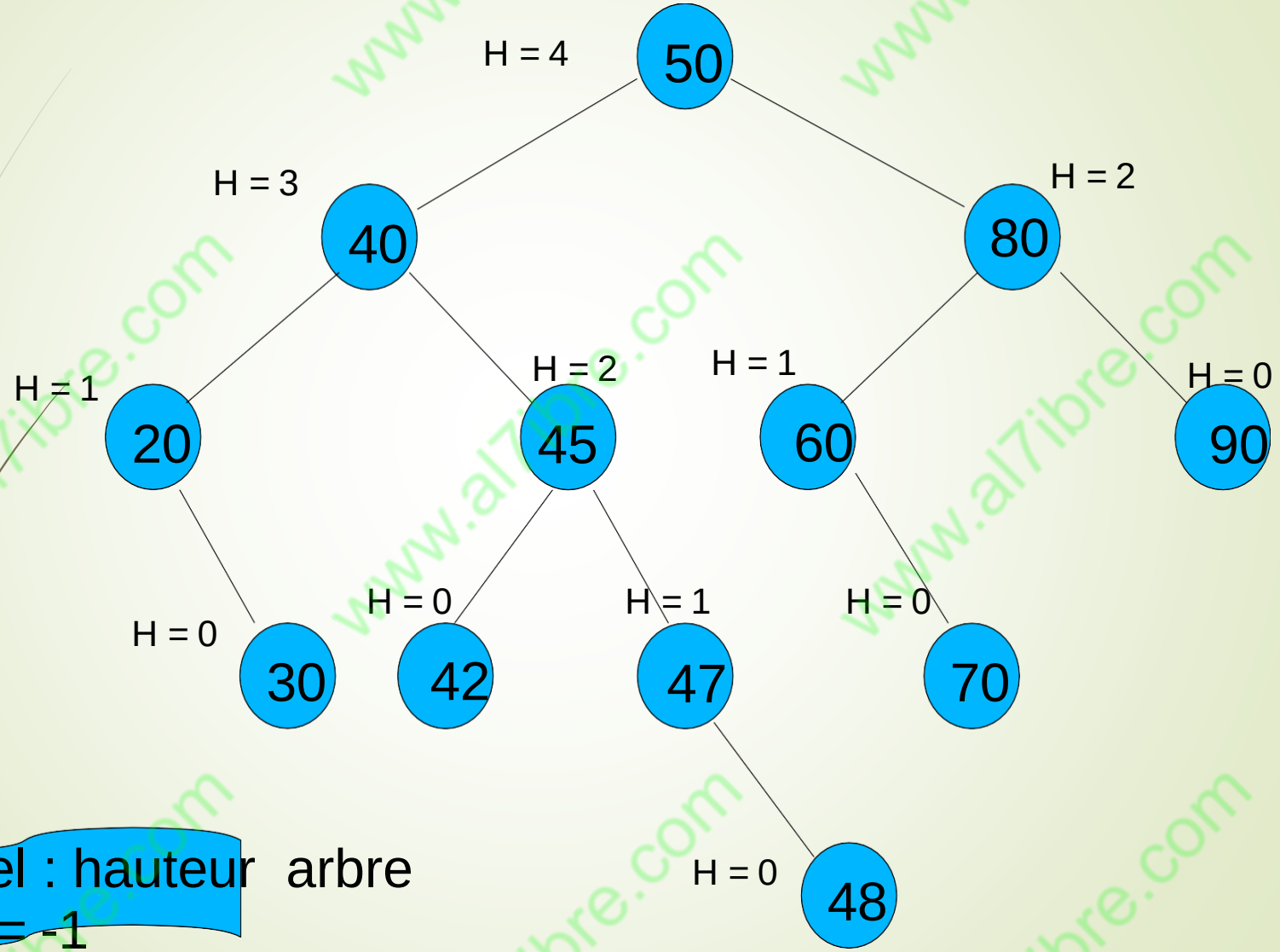
Pr F.Omary

2019-2020

Introduction

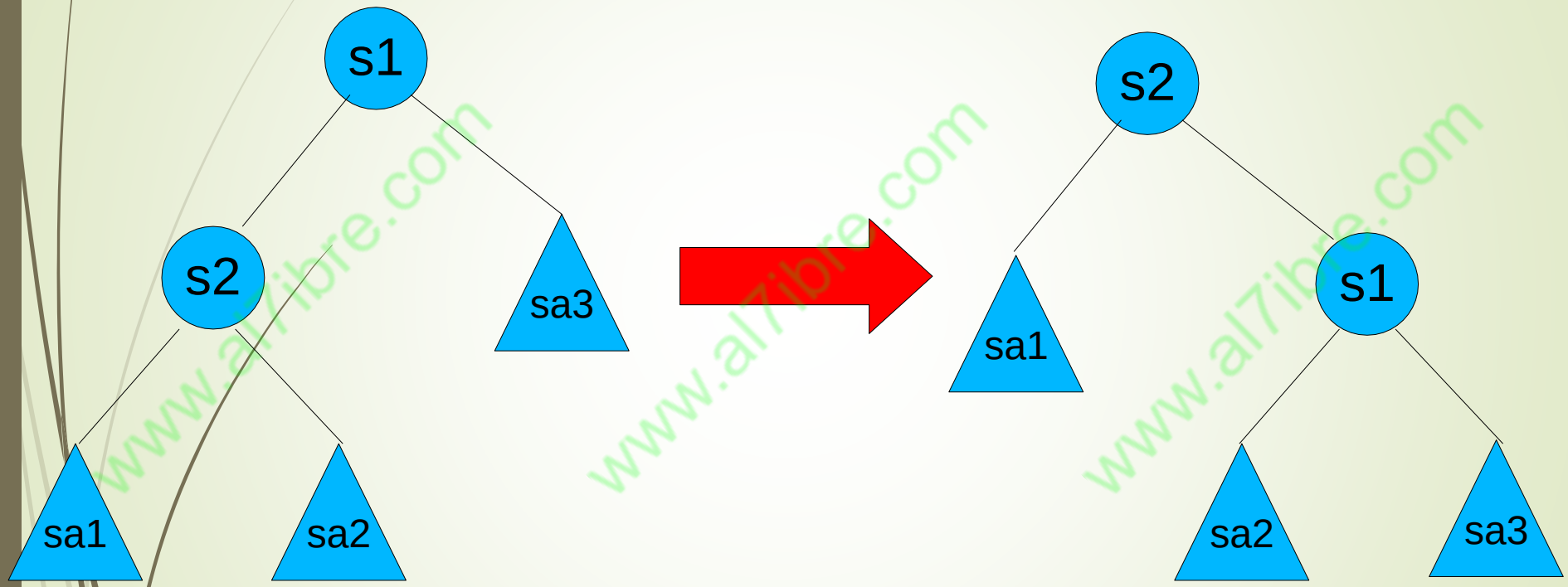
- Pourquoi
  - Les arbres équilibrés rendent les recherches plus efficaces
  - Trouver comment maintenir un arbre relativement équilibré au fur et à mesure des insertions (et suppression)
- Solution :
  - Les arbres AVL (Adelson-Velskii et Landis) : pour tout sommet, les hauteurs des sous- arbres gauche et droit diffèrent d'au plus 1.
- Rmq : un arbre AVL N'EST PAS un arbre équilibré

Exemple d'arbre AVL



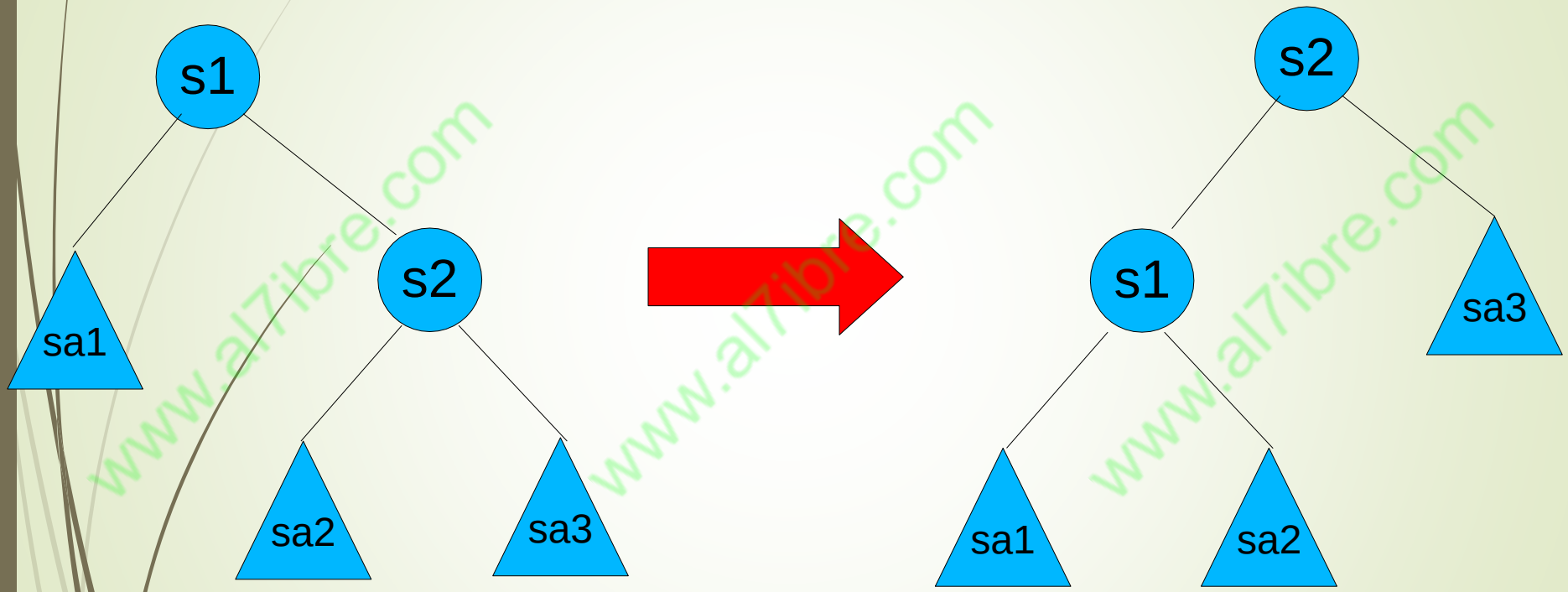
Rappel : hauteur arbre vide = -1

Préambule : Rotation droite



Rotation droite autour de S1, notée rd(S1)

Préambule : Rotation gauche



Insertion dans un arbre AVL

- Le principe de l'insertion dans un arbre AVL est le suivant :

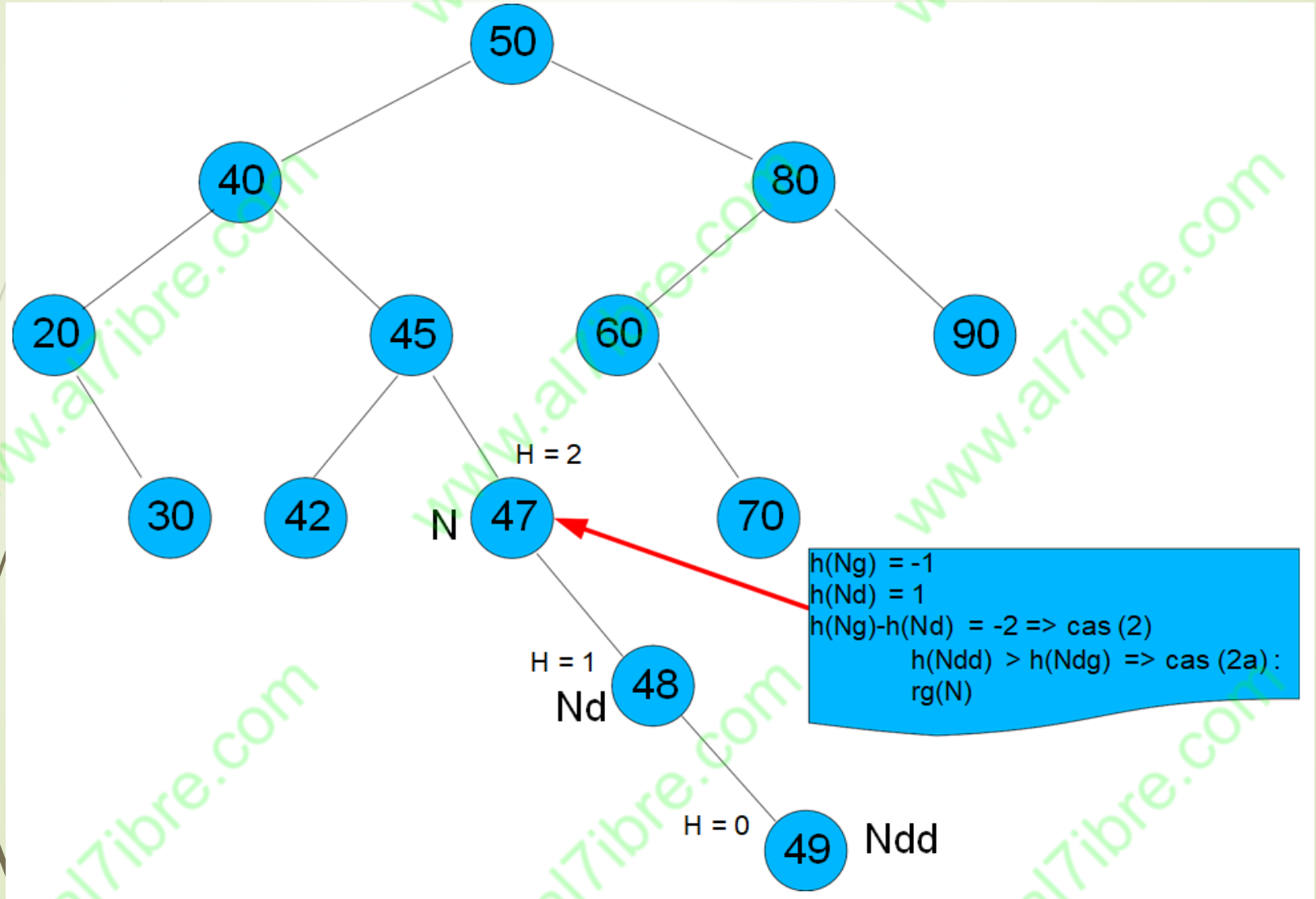
- insérer le nouveau nœud au bon endroit
- au fur et à mesure de la remontée dans l'arbre (du nœud père du nœud inséré à la racine), rééquilibrer l'arbre en effectuant les rotations appropriées

Choix des rotations

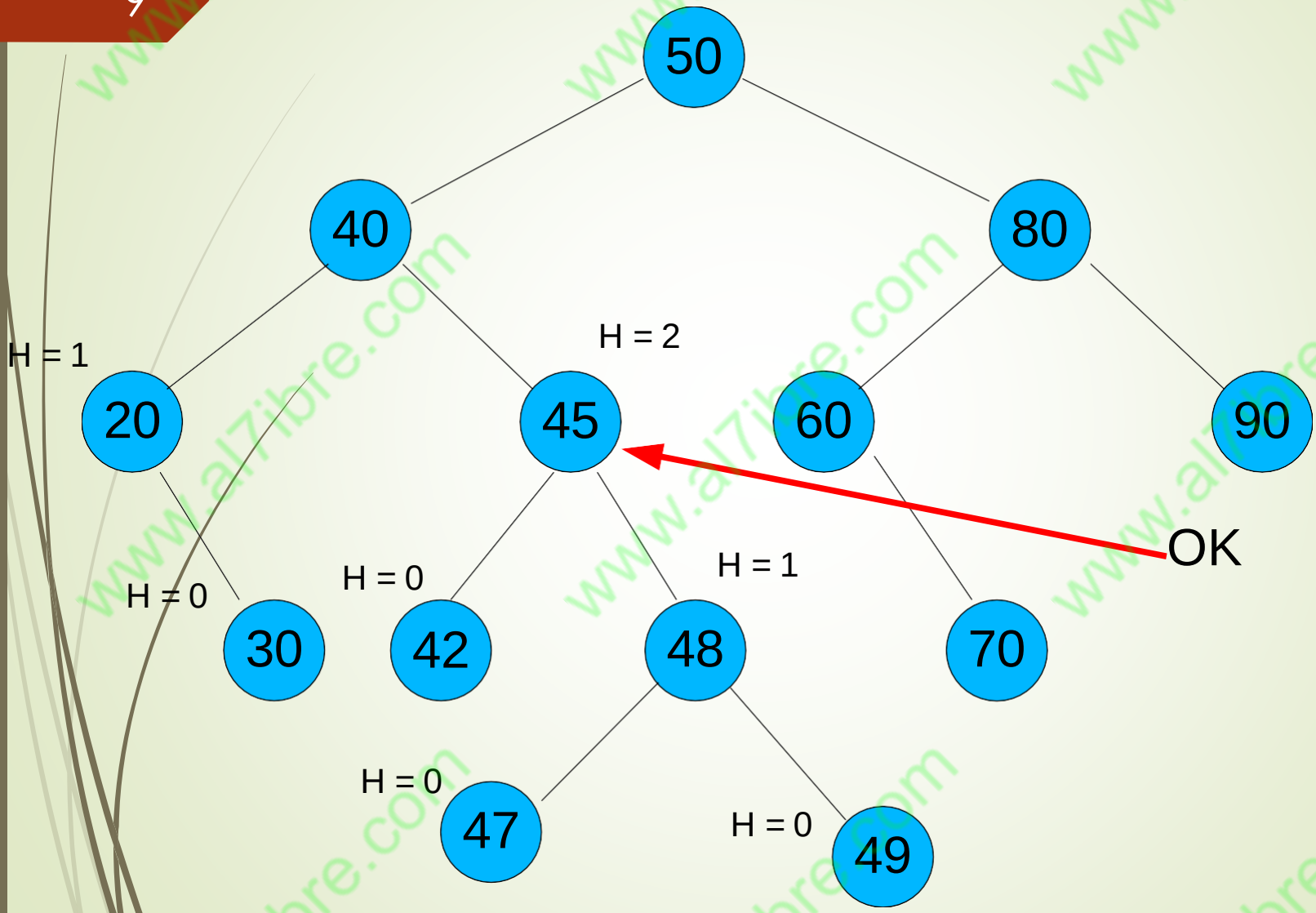
- Notations
  - Soient N le noeud courant, Ng son fils gauche et Nd sont fils droit.
  - Soient Ngg le fils gauche de Ng et Ngd le fils droit de Ng

    Soient Ndg le fils gauche de Nd et Ndd le fils droit de Nd
  - Soit h(x) la hauteur de l'arbre de racine le noeud x.
- Algorithme
  - Si |h(Ng)-h(Nd)| <= 1, ne rien faire
  - Sinon
    - Si h(Ng)-h(Nd) = 2 **cas (1)**
      - Si h(Ngg) > h(Ngd) Alors rd(N) **cas (1a)**
      - Sinon rg(Ng) puis rd(N) **cas (1b)**
    - Sinon (h(Ng)-h(Nd) = -2) **cas (2)**
      - Si h(Ndd) > h(Ndg) Alors rg(N) **cas (2a)**
      - Sinon rd(Nd) puis rg(N) **cas (2b)**
    - Fsi
  - Fsi

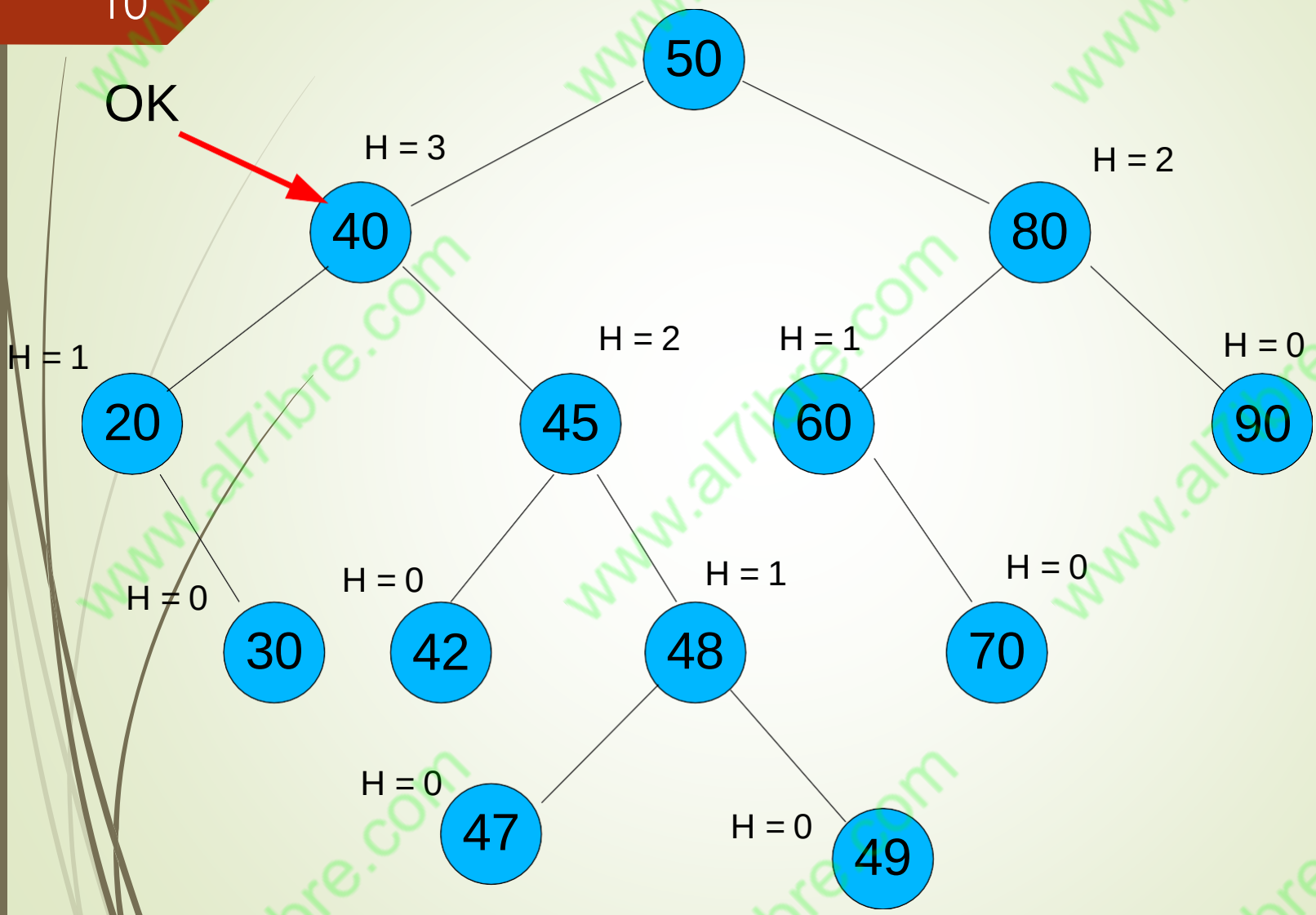
Exemple d'ajout : 49



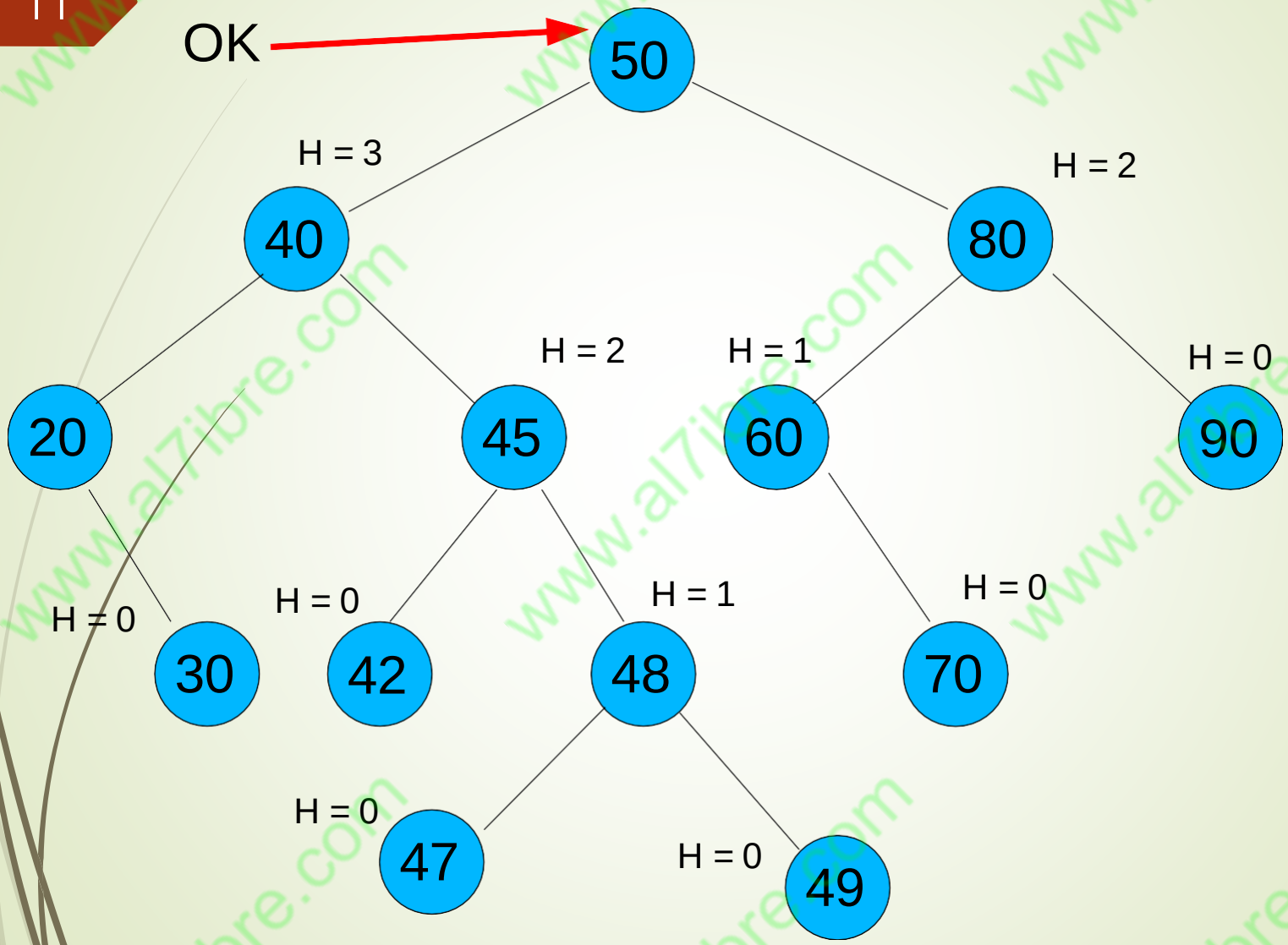
Exemple d'ajout : 49



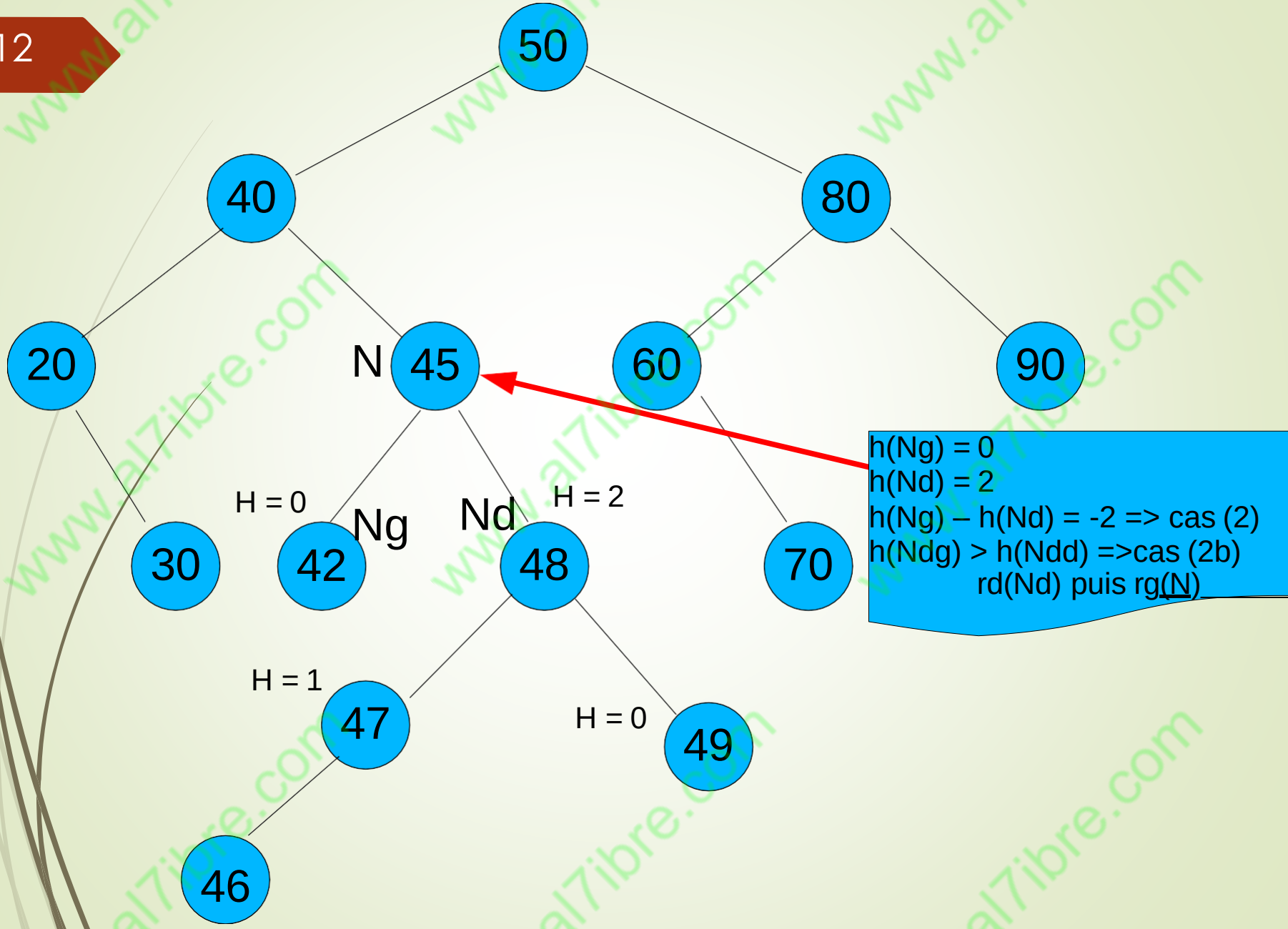
Exemple d'ajout : 49



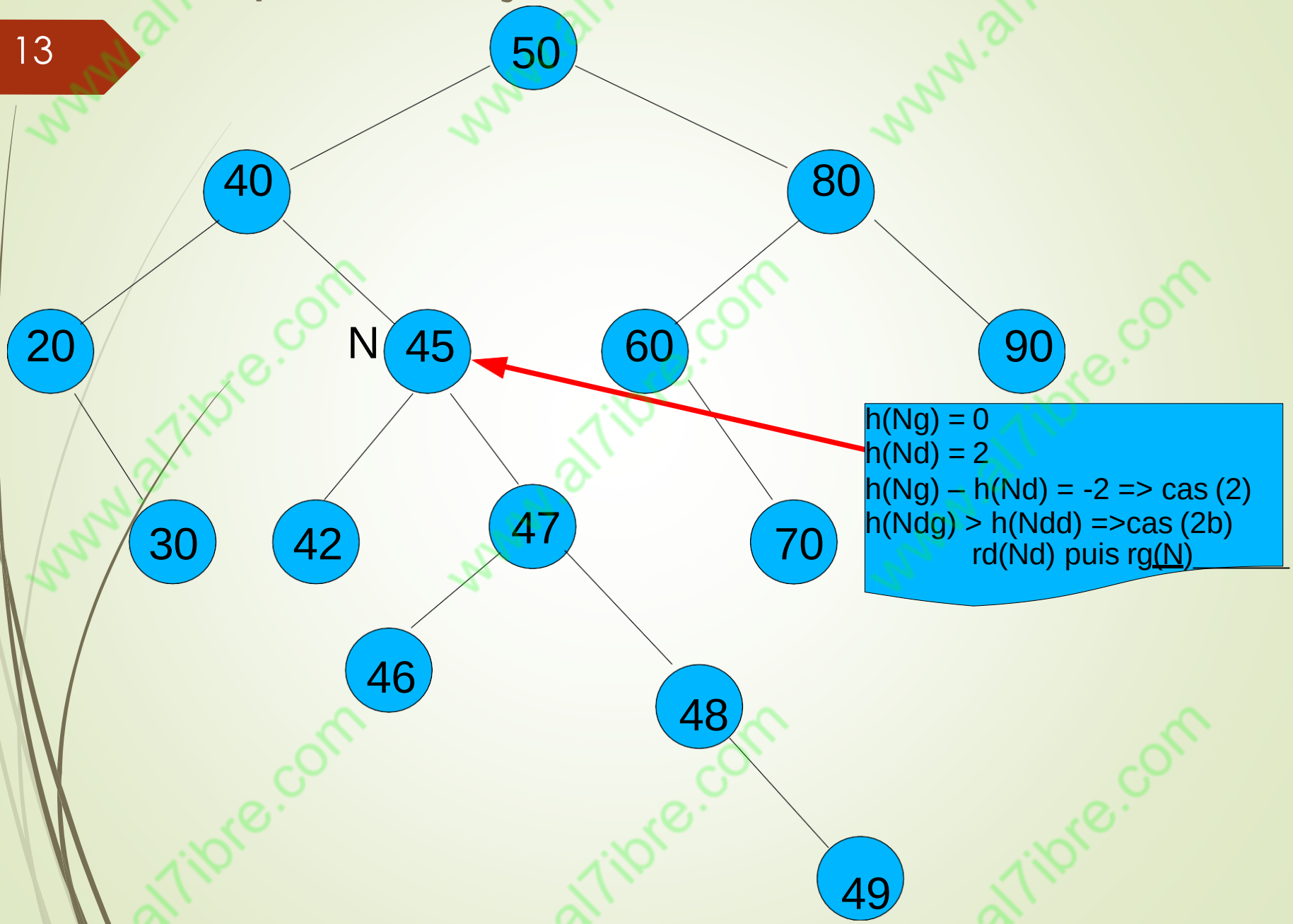
Exemple d'ajout : 49



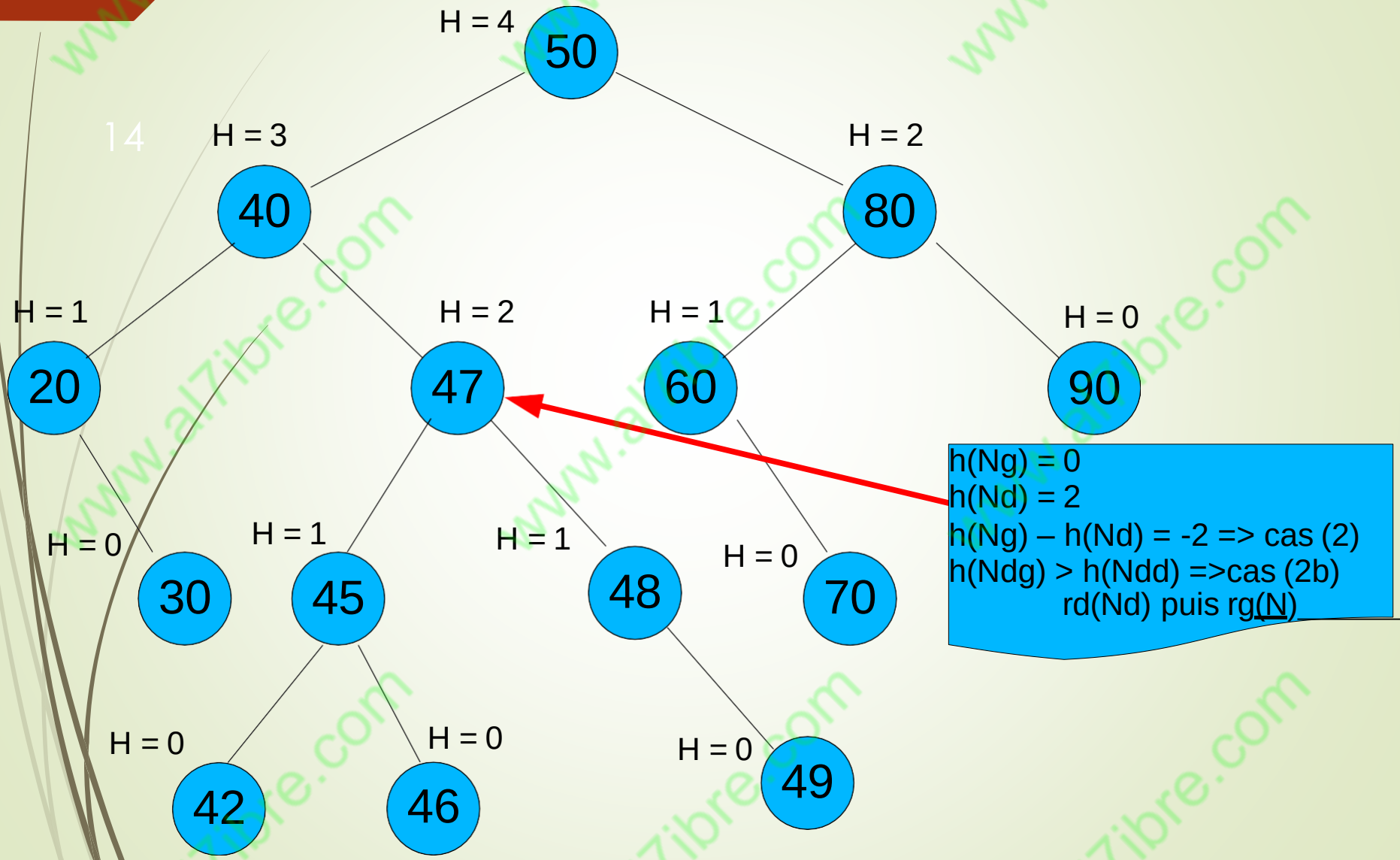
Exemple d'ajout : 46



Exemple d'ajout : 46



Exemple d'ajout : 46



Graphes

1

Objectifs

- **Etudier une nouvelle structure de données non linéaire, plus générale, où chaque élément peut posséder plusieurs prédécesseurs et plusieurs successeurs :**
  - **Terminologie**
  - **Type Abstrait de Données Graphe**
  - **Représentation et implémentation**
  - **Parcours d'un graphe**

2

Notion de Graphes

- **Les graphes sont l'une des structures de données les plus utilisées en informatique :**
  - Les algorithmes permettant de les manipuler constituent les fondements de l'informatique
  - Il existe des centaines de problèmes informatiques qui sont définis en termes de graphes
- **Les graphes servent généralement à modéliser des problèmes en termes de relations ou de connexions entre des objets**
- **Les objets sont représentés par des sommets**
- **Les relations ou connexions sont représentées par des arcs reliant les sommets**
- **Les graphes peuvent être orientés (les arcs vont d'un sommet à l'autre dans un sens précis) ou non orientés (les arcs n'ont pas de sens)**

3

Exemples

- **Dans une carte de liaisons aériennes, les villes sont des sommets du graphe et l'existence d'une liaison aérienne entre deux villes est la relation du graphe**
- **Dans le graphe du flot de contrôle d'un programme, les boîtes (instructions ou tests) sont les sommets, et les flèches indiquent les enchaînements possibles entre celles-ci**
- **Dans une entreprise où certaines tâches doivent être exécutées avant d'autres, on peut schématiser l'ordonnancement des tâches par un graphe où les sommets sont les tâches et où il existe un arc entre deux tâches t_i et t_j seulement si t_i doit être terminée juste avant d'exécuter t_j**

4

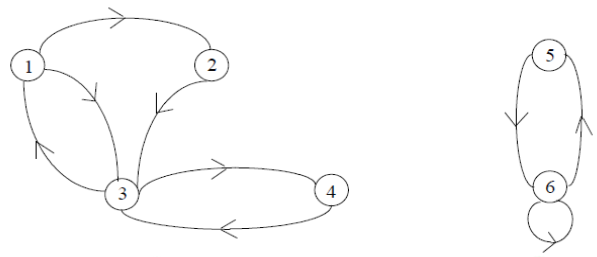
Graphe Orienté (Définitions)

- **Un graphe orienté *G* est un couple (*S*,*A*), où :**
  - ***S* est un ensemble fini d'éléments appelés sommets** (***vertex*** en anglais, au pluriel ***vertices***)
  - ***A* est un ensemble fini de paires (ordonnées) de sommets, appelées arcs (*arc* en anglais)**
- **On écrit *G* = (*S*,*A*) pour représenter le graphe**
- **Un graphe orienté est dit complet si quels que soient deux sommets distincts, il existe un arc les reliant dans un sens ou dans l'autre**

5

Graphe Orienté (Exemple 1)

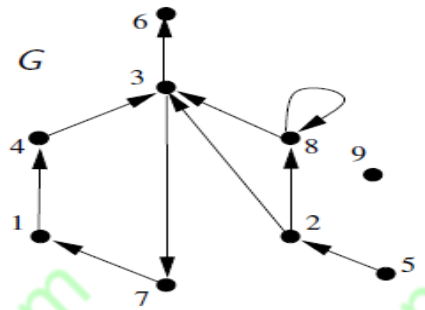
- Soient *S* ={1,2,3,4,5,6} et *A*={(1;2),(1;3),(2;3),(3;1),(3;4),(4;3),(5;6),(6;5),(6;6)}
- (*S*, *A*) est un graphe orienté qui peut être représenté par :



6

Graphe Orienté (Exemple 2)

- Soit le graphe orienté G=(S,A) où
  - S = {1,2,3,4,5,6,7,8,9} et
  - A={(1,4);(2,3);(2,8);(3,6);(3,7);(4,3);(5,2);(7,1);(8,3);(8,8)}



7

Graphe Orienté (Terminologie) (1)

- Soit $G = (S, A)$ un graphe orienté. Si $X = (a,b) \in A$, on dit que :
  - a est **adjacent** à b
  - a est **un prédécesseur** de b.
  - b est **un successeur** de a.
  - a est **l'origine** de l'arc X.
  - b est **l'extrémité** de l'arc X.
  - X est **incident** au sommet a et au sommet b.
  - De plus, si $a = b$, on dit que X est **une boucle**.

8

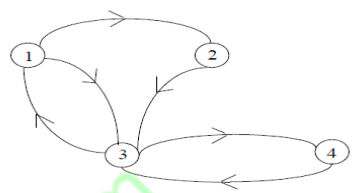
Graphe Orienté (Terminologie) (2)

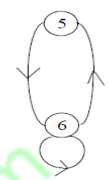
- On appelle **chemin** d'un graphe orienté *G* une suite (finie) d'arcs de *G* telle que l'extrémité d'un arc est toujours confondue avec l'origine du suivant.
- L'origine du premier arc de la suite est appelé **origine du chemin**.
- L'extrémité du dernier arc de la suite est appelé **extrémité du chemin**
- La **longueur d'un chemin** est le nombre d'arcs qui le composent
- Un chemin est dit **simple** si tous les arcs qui le composent sont différents.
- Un chemin est dit **élémentaire** si tous les sommets qui le composent sont différents.
- On appelle **circuit** tout chemin dont l'origine et l'extrémité sont confondues.

9

Exemple

- En reprenant **l'exemple 1**, on a :
  - {(1,3);(3,1);(1,2)} est un chemin simple non élémentaire d'origine 1 et d'extrémité 2.
  - {(1,2);(2,3);(3,4)} est un chemin simple et élémentaire. Ce chemin est de longueur 3
  - {(2,3);(3,4);(4,3);(3,1);(1,2)} est un circuit simple et non élémentaire
  - {(6,6)}





10

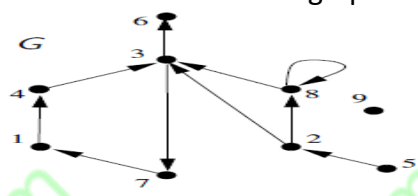
Graphe Orienté (Terminologie) (3)

- Soient u et v deux sommets d'un graphe orienté. On dit que :
  - v est **un descendant** de u s'il existe un chemin allant u à v
  - v est **un ascendant** de u s'il existe un chemin allant v à u.
  - Un sommet v tel qu'il n'existe aucun chemin de u à v dans G est dit **inaccessible** (ou **non atteignable**) à partir de u.
  - Un sommet est dit **isolé** s'il n'est accessible par aucun autre sommet du graphe
- Il est à noter que les sommets d'un circuit sont tous ascendants et descendants les uns des autres

11

Exemple

- En reprenant **l'exemple 2** :
  - On considère le chemin {(5,2);(2,8);(8,3);(3,6)}.
  - Le sommet 6 est descendant du sommet 5 mais l'inverse n'est pas vrai.
  - Le sommet 5 est ascendant du sommet 6
  - Le sommet 2 est inaccessible depuis le sommet 3.
  - Le sommet 9 est **isolé** du reste du graphe



12

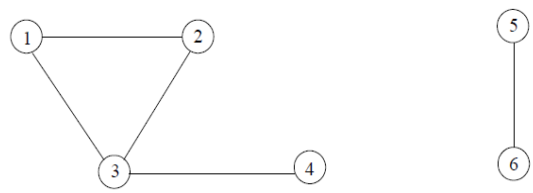
Graphe Non Orienté (Définitions)

- **Un graphe (simple) non orienté *G* est un couple (*S*,*A*), où :**
  - *S* est un ensemble fini de **sommets**.
  - *A* est un ensemble fini de paires (non ordonnées) de sommets de *S*, appelées **arêtes** (**edge** en anglais)
- **On écrit *G* = (*S*,*A*) pour représenter le graphe**
- **Un graphe non orienté est dit complet si quels que soient deux sommets distincts, il existe une arête les reliant**

13

Graphe Non Orienté (Exemple 1)

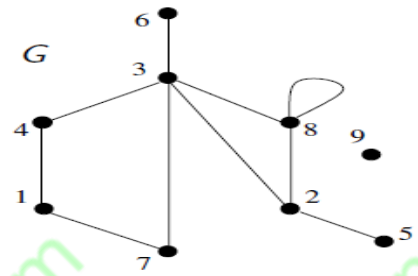
- Soient *S* ={1,2,3,4,5,6} et *A*={(1;2),(1;3),(2;3),(3;4),(5;6)}
- (*S*, *A*) est un graphe non orienté qui peut être représenté par :



14

Graphe Non Orienté (Exemple 2)

- Soit le graphe non orienté G=(S,A) où
  - S = {1,2,3,4,5,6,7,8,9} et
  - A={(1,4);(1,7);(2,3);(2,5);(2,8);(3,4);(3,6); (3,7);(3,8);(8,8)}



15

Graphe Non Orienté (Définitions)

- Soit $G = (S, A)$ un graphe non orienté.
  - Si $X = \{a,b\} \in A$, on dit que a et b sont **voisins**.
  - On appelle **chaîne** de G une suite (finie) d'arêtes de G telle que 2 arêtes consécutives dans la suite ont un sommet commun.
  - Un **cycle** est une chaîne dont l'origine et l'extrémité sont confondues.
  - Une chaîne est dite **élémentaire** si elle ne contient pas plusieurs fois le même sommet
  - La **longueur** d'une chaîne est le nombre d'arêtes qui la composent.

16

Graphe Connexe/Fortement Connexe

- Un graphe non orienté $G=(S, A)$ est dit **connexe** si et seulement si, pour toute paire de sommets distincts $\{x,y\}$ de S, il existe une chaîne entre les sommets x et y.
- Un graphe orienté $G=(S, A)$ est dit **fortement connexe** si et seulement si, pour toute paire de sommets distincts $\{x,y\}$ de S, il existe un chemin de x à y et un chemin de y à x.

17

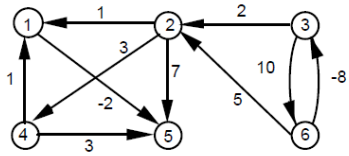
Notion de Graphe Valué

- Dans de nombreuses applications, il est naturel d'associer **une valeur** (on dit aussi **un coût** ou **un poids**) aux arcs ou aux arêtes du graphe.
- **Un graphe valué** (ou **pondéré**), orienté (resp. non orienté) est un triplet (S,A,C) où S est un ensemble fini de sommets, A un ensemble fini d'arcs (resp. d'arêtes) et C une fonction de A à valeurs réelles appelée **fonction coût**
- Ainsi, on pourra traiter des problèmes tels que la recherche du plus court chemin entre deux sommets d'un graphe

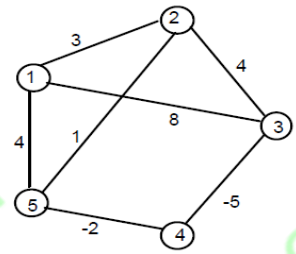
18

Exemples de Graphes Valués

- **Exemple de graphe orienté valué :**



- **Exemple de graphe non orienté valué :**



19

Distance et Diamètre

- **La distance entre deux sommets** d'un graphe est la plus petite longueur des chaînes, ou des chemins, reliant ces deux sommets.
- **Le diamètre d'un graphe** est la plus longue des distances entre deux sommets.

20

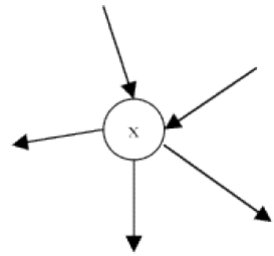
Notion de Degré

- Dans un graphe *orienté*, si X=(u, v) est un arc, on dit que **X est incident à u vers l'extérieur**. Le nombre d'arcs ayant leur extrémité initiale en u, se note $\mathbf{d^{o+}(u)}$ et s'appelle le ***demi-degré extérieur*** **de u**. c'est le nombre de successeurs de u.
- On définit de même les notions d'**arc incident vers l'intérieur** et de **demi-degré intérieur** qui est noté $\mathbf{d^{o-}(u)}$. le nombre de prédécesseurs de u.
- Dans un graphe orienté (resp. non orienté), on appelle **degré d'un sommet**, et on note $\mathbf{d^{o}(u)}$, le nombre d'arcs (resp. d'arêtes) dont u est une extrémité.
- Dans le cas d'un graphe orienté, on a $\mathbf{d^{o}(u) = d^{o+}(u) + d^{o-}(u)}$, pour tout sommet u. C'est le nombre de sommets adjacents à u.
- Un sommet de degré **1** (resp. **0**) est dit **sommet pendant** (resp. **isolé**)
- Un graphe est dit **régulier** si les degrés de tous ses sommets sont égaux

21

Exemple

- Dans le graphe suivant :
  - $\mathbf{d^{\circ+}(x) = 3, d^{\circ-}(x) = 2}$ **et d(x) = 5**



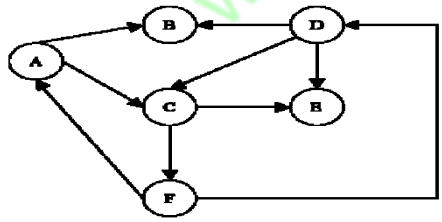
22

Sous-Graphe et Graphe partiel

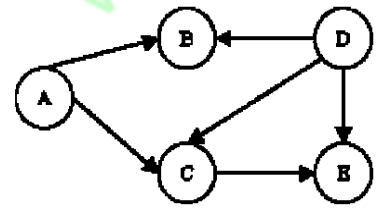
- Soit *G=(S,A)* un graphe. **Le sous-graphe de *G* engendré par *S'*** (un sous-ensemble de *S*) est le graphe *G'* dont les sommets sont les éléments de *S'* et dont les arcs (resp. les arêtes) sont les arcs (resp. les arêtes) de *G* ayant leurs deux extrémités dans *S'*. Autrement dit, on ignore les sommets de *S\S'* ainsi que les arcs ayant au moins une extrémité dans *S\S'*.
- Soit *G=(S,A)* un graphe. **Le graphe partiel de *G* engendré par *A'*** (un sous-ensemble de *A*) est le graphe *G' =(S,A')* dont les sommets sont les éléments de *S* et dont les arcs (resp. les arêtes) sont ceux de *A'*. Autrement dit, on élimine de G les arcs (resp. arêtes) de *A\A'*.

23

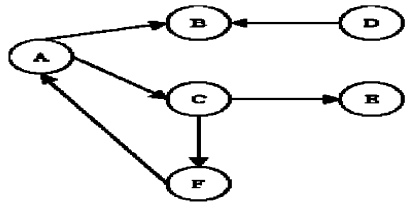
Exemples



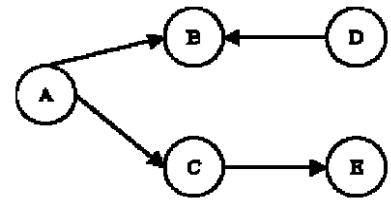
Un graphe G



Un sous-graphe de G



Un graphe partiel de G



Un sous-graphe partiel de G

24

Composantes Connexes d'un Graphe Non Orienté

- On définit la relation :

  v est accessible à partir de u si et seulement si il existe un chemin de longueur k ≥ 0 d'origine u et d'extrémité v.

- C'est une relation d'équivalence :
  - elle est *réflexive* car k = 0 est admis; elle est *symétrique* car le graphe est non orienté; elle est *transitive*, car on "concatène" les chemins.

- Par définition, **les *composantes connexes*** d'un graphe *non orienté* G sont les classes d'équivalence pour la relation: **« être accessible à partir de »**.
  - **D'une autre manière, on appelle composante connexe un sous-graphe connexe maximal.**

25

Composantes Fortement Connexes d'un Graphe Orienté

- Pour un graphe orienté, la relation "*être accessible à partir de*" est toujours réflexive et transitive, mais elle n'est plus symétrique. On considère alors sa symétrisée :

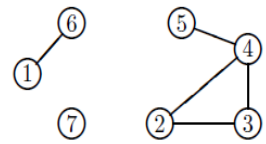
  v et u sont mutuellement accessibles si et seulement si il existe un chemin (de longueur k ≥ 0) d'origine u et d'extrémité v et un chemin (de longueur l ≥ 0) d'origine v et d'extrémité u.

- Par définition, **les *composantes fortement connexes*** d'un graphe *orienté* sont les classes d'équivalence de G pour la relation : **« être mutuellement accessibles »**.
  - **D'une autre manière, on appelle composante fortement connexe un sous-graphe fortement connexe maximal.**

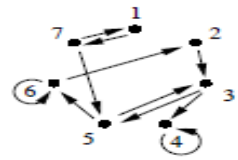
26

Exemples

- Le graphe suivant a trois composantes connexes : {1,6}, {7} et {2,3,4,5}



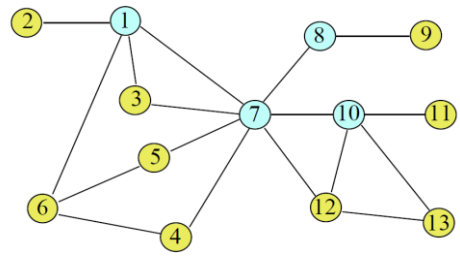
- Le graphe suivant a trois composantes fortement connexes : {1,7}, {2,3,5,6} et {4}



27

Point d'Articulation d'un Graphe

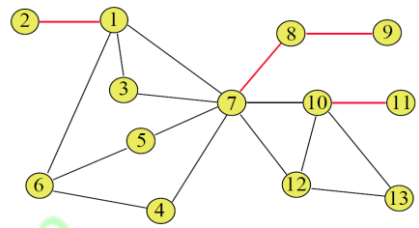
- C'est un sommet d'un graphe, qui, si on le supprime, déconnecte le graphe
- Dans le graphe suivant, les sommets 1, 7, 8 et 10 sont des points d'articulation



28

Pont d'un Graphe

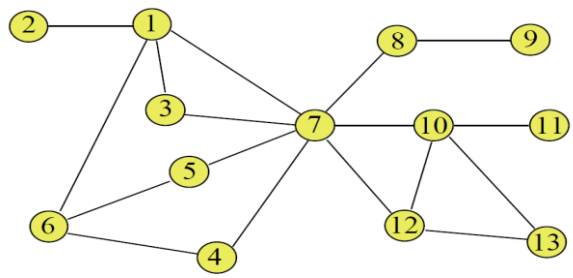
- C'est une arête d'un graphe, qui, si on la supprime, déconnecte le graphe
- Dans le graphe suivant, les arcs (1,2), (7,8), (8,9) et (10,11) sont des ponts



29

Graphe Bi-Connexe

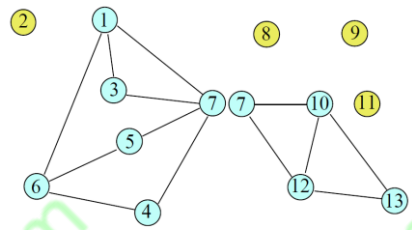
- Un graphe connexe sans point d'articulation est dit **bi-connexe**
- Le graphe suivant n'est pas bi-connexe



30

Composantes Bi-Connexes

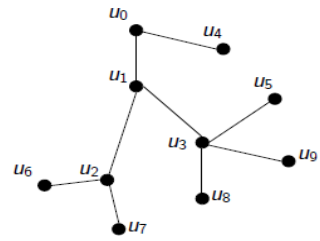
- Un graphe peut ne pas être bi-connexe mais contenir des **composantes bi-connexes**
- Dans une composante bi-connexe, il existe un circuit entre deux sommets quelconques



31

Notion d'Arbre

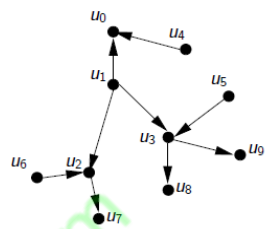
- Un graphe non orienté où tous les sommets sont accessibles les uns des autres est dit connexe.
- On appelle **arbre** un graphe non orienté connexe et sans cycle.



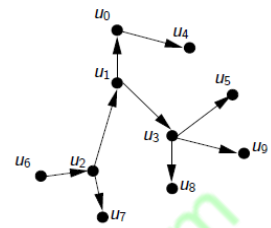
32

Notion d'Arborescence

- Lorsqu'on oriente les arêtes d'un arbre, le graphe obtenu est appelé **une arborescence**.
- Dans une arborescence, on appelle racine un sommet pour lequel tous les autres sommets sont accessibles (il n'existe pas toujours de racine).



Arborescence sans racine

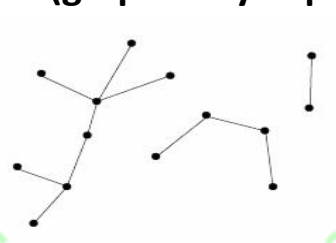


u_6 est racine.

33

Notion de Forêt

- **Une *forêt*** est un graphe non orienté (resp. orienté) dont chaque composante connexe (resp. fortement connexe) est un **arbre** (resp. **une arborescence**).
- **Exemple de forêt (graphe acyclique) :**



34

Type Abstrait Graphe

- Parfois, le graphe est donné une fois pour toutes. Les opérations intéressantes sont :
  - **test d'existence d'un arc** (**d'une arête** entre deux sommets)
  - **test d'existence d'un sommet** parmi les successeurs d'un autre sommet
  - **énumération des successeurs d'un sommet**. Pour ce faire, il faut connaître **le demi-degré extérieur** de tout sommet et **le ième successeur** d'un sommet
  - **…**
- Le plus souvent, le graphe est évolutif ; on veut donc lui appliquer les opérations :
  - **ajout et suppression d'un sommet**
  - **ajout et suppression d'un arc**
  - …
- **Deux types abstraits :**
  - un pour les graphes orientés
  - un autre pour les graphes non orientés
- Ces deux types abstraits utilisent **le type Sommet** :
  - pour distinguer les sommets d'un graphe, **on les étiquette**, soit par des chaînes de caractères, soit par des numéros (ce qui va être utilisé dans la suite)

35

Type Abstrait Sommet

```
Type Sommet {on étiquette un sommet par un numéro}

Utilise Entier

Opérations
   créer       : Entier → Sommet
   modifier    : Sommet x Entier → Sommet
   numéro      : Sommet → Entier

Axiomes
   numéro(som(i)) = i, pour tout entier i
```

36

Spécification des graphes orientés (Conventions)

- Quand on ajoute un sommet, celui-ci est isolé (il n'a aucun arc incident) ;
- Quand on ajoute un arc, si les sommets adjacents à cet arc n'appartiennent pas au graphe, on les ajoute ;
- Quand on retire un arc, les sommets adjacents ne sont pas retirés ;
- Quand on retire un sommet, tous les arcs incidents sont supprimés.

37

Type Abstrait Graphe (Orienté) (1)

```
Type Graphe {cas orienté}
Utilise Sommet, Entier, Booléen
Opérations
  graphe_vide      : → Graphe
  ajouter_sommet   : Sommet x Graphe → Graphe
  ajouter_arc      : Sommet x Sommet x Graphe → Graphe
  est_sommet       : Sommet x Graphe → Booléen
  est_arc          : Sommet x Sommet x Graphe → Booléen
  d°+              : Sommet x Graphe → Entier
  ième_succ        : Entier x Sommet x Graphe → Sommet
  d°-              : Sommet x Graphe → Entier
  ième_pred        : Entier x Sommet x Graphe → Sommet
  supprimer_sommet: Sommet x Graphe → Graphe
  supprimer_arc    : Sommet x Sommet → Graphe
```

38

Type Abstrait Graphe (Orienté) (2)

Préconditions

ajouter_sommet(s,g) *est-défini-ssi* est_sommet(s,g) = faux

ajouter_arc(s,s',g) *est-défini-ssi* s ≠ s' ET est_arc(s,s',g) = faux

$d^{\circ+}$(s,g) *est-défini-ssi* est_sommet(s,g) = vrai

$d^{\circ-}$(s,g) *est-défini-ssi* est_sommet(s,g) = vrai

ième_suc(i,s,g) *est-défini-ssi* est_sommet(s,g) = vrai
ET (i ≤ $d^{\circ+}$(s,g)) = vrai

supprimer_sommet(s,g) *est-défini-ssi* est_sommet(s,g) = vrai

supprimer_arc(s,s'g) *est-défini-ssi* est_arc(s,s',g) = vrai

39

Type Abstrait Graphe (Orienté) (3)

Axiomes {pour est_sommet}

est_sommet(s,graphe_vide()) = faux

si s = s' alors est_sommet(s,ajouter_sommet(s',g)) = faux

si s ≠ s' alors est_sommet(s,ajouter_sommet(s',g)) = vrai

si s = s' OU s = s'' alors est_sommet(s,ajouter_arc(s',s'',g)) = faux

si s ≠ s' ET s ≠ s'' alors
est_sommet(s,ajouter_sommet(s'',g)) = est_sommet(s,g)

40

Type Abstrait Graphe (Orienté) (4) (Opérations Auxiliaires)

```
premsucc          : Sommet x Graphe → Sommet
succsuivant       : Sommet x Sommet x Graphe → Sommet
coût              : Sommet x Sommet x Graphe → Réel
ajouter_arc_valué : Sommet x Sommet x Réel x Graphe → Graphe
nb_sommets        : Graphe → Entier
nb_arcs           : Graphe → Entier
```

41

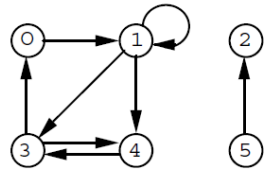
Représentations des Graphes

- **Deux implémentations classiques :**
  - **Par matrice d'adjacence**
  - **Par liste d'adjacence**
- **D'autres implémentations efficaces pour certains algorithmes :**
  - **Matrice d'incidence**
  - **Liste des arcs**
  - **…**

42

Représentation par Matrice d'Adjacence (1)

- **Correspond au cas où l'ensemble de sommets du graphe n'évolue pas**
- **On représente l'ensemble des arcs par un tableau de booléens**
- **Le graphe est représenté par une matrice carrée de booléens, dite matrice d'adjacence, de dimension n si le graphe a n sommets**

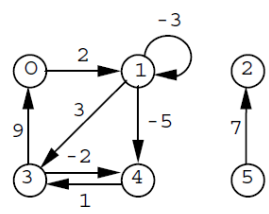


	0	1	2	3	4	5
0	0	1	0	0	0	0
1	0	1	0	1	1	0
2	0	0	0	0	0	0
3	1	0	0	0	1	0
4	0	0	0	1	0	0
5	0	0	1	0	0	0

43

Représentation par Matrice d'Adjacence (2)

- **Dans le cas où le graphe est non orienté, la matrice est *symétrique***
- **Dans le cas où le graphe est valué, on utilise une matrice où :**
  - **l'élément d'indices i et j a pour valeur le poids de l'arc/arête du sommet i au sommet j, si cet arc/arête existe,**
  - **et sinon une valeur dont on sait qu'elle ne peut être un poids: par exemple, le plus grand entier utilisable si les poids sont des entiers bornés supérieurement.**



	0	1	2	3	4	5
0	0	2	0	0	0	0
1	0	-3	0	3	-5	0
2	0	0	0	0	0	0
3	9	0	0	0	-2	0
4	0	0	0	1	0	0
5	0	0	7	0	0	0

44

Représentation par Matrice d'Adjacence (3)

- **Avantages :**
  - tester l'existence d'un arc (ou d'une arête) entre deux sommets: on accède directement à l'élément de la matrice (en un temps constant).
  - il est facile d'ajouter ou de retirer un arc ou une arête
  - il facile de parcourir tous les successeurs ou prédécesseurs d'un sommet.
- **Inconvénients :**
  - n tests quel que soit le nombre de successeurs de i. Il en est de même du calcul de d^{o+} ou de d^{o-}.
  - une consultation complète de la matrice requiert un temps d'ordre n^2
  - exige un espace mémoire de **$O(n^2)$** si le graphe a n sommets, quel que soit le nombre d'arcs ou d'arêtes du graphe.
- Pour remédier à cet inconvénient, on préfère souvent utiliser une représentation appelée "**par listes d'adjacence**".
- Cette représentation convient pour les petits graphes et lorsque l'accès aux successeurs, et surtout aux prédécesseurs, est important

45

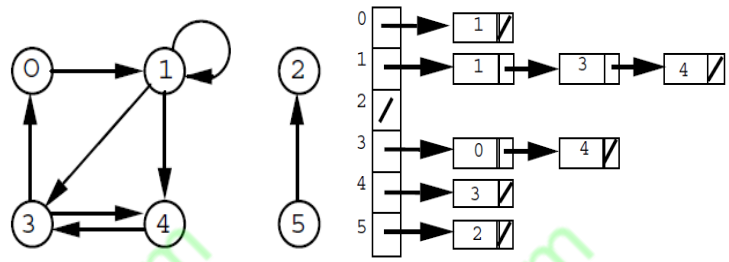
Implémentation en C d'un Graphe par Matrice d'Adjacence

```
#define N_MAX 20
typedef struct {
  int g[N_MAX][N_MAX];
  int n;
} GrapheM;
```

46

Représentation par Liste d'Adjacence (1)

- **C'est un tableau de listes chaînées :**
  - **La dimension du tableau est de *n (nombre de sommets)***
  - **Chaque sommet du tableau contient une liste chaînée de sommets qui lui sont adjacents (liste de ses successeurs)**



47

Représentation par Liste d'Adjacence (2)

- **Avantages :**
  - l'espace mémoire utilisé est, pour un graphe orienté avec n sommets et m arcs, en **O(n+m)**.
  - dans le cas d'un graphe non orienté avec m arêtes, l'espace mémoire est en **O(n+2m)**.
  - pour faire un traitement sur les successeurs d'un sommet s, le nombre de sommets parcourus est exactement le nombre de successeurs de s, soit $d^{\circ +}(s)$.
- **Inconvénients :**
  - exige, dans le pire de cas, un temps d'ordre n pour tester s'il existe un arc (resp. une arête) entre un sommet donné x et un sommet y (cas où la liste d'adjacence est de longueur n-1 et où y est en fin de liste) ou pour l'ajout d'un arc ou d'une arête (avec test de non répétition).
  - ne permet pas de calculer facilement les opérations relatives aux prédécesseurs ($d^{\circ -}$ et ième_pred).
- Représentation convenable pour les grands graphes :
  - utilisation de moins d'espace mémoire et parcours rapide des successeurs d'un sommet

48

Graphes

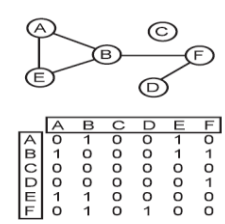
(suite)

Implémentation en C d'un Graphe par Liste d'Adjacence

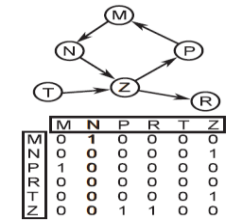
```
#define N_MAX 20
typedef struct cellule {
  int sommet;
  struct cellule* suiv;
} Cellule;
typedef Cellule* Liste;

typedef struct {
  Liste a[N_MAX];
  int n;
} GrapheL;
```

49

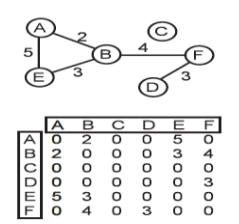
Matrices vs Listes d'Adjacences de Graphes Orientés et Non Orientés



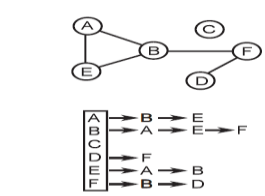
(a) Adjacency matrix of an undirected graph.



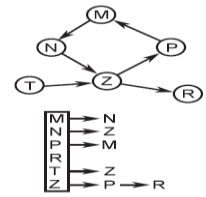
(b) Adjacency matrix of a directed graph.



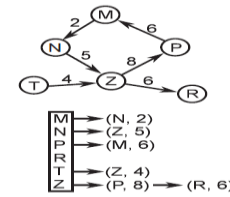
(c) Adjacency matrix of an undirected weighted graph.



(a) Adjacency list of an undirected graph.



(b) Adjacency list of a directed graph.



(c) Adjacency list of a directed weighted graph.

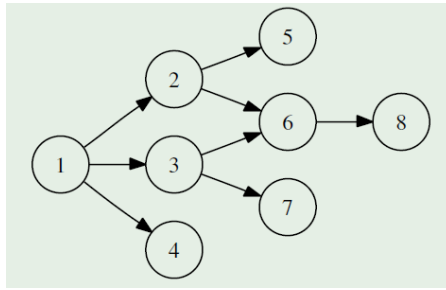
50

Parcours d'un Graphe

- **Parcours de tous les sommets :**
  - **visiter chaque sommet du graphe une seule fois**
  - **appliquer un même traitement en chaque sommet**
- **Parcours à partir d'un sommet s :**
  - **Parcours en profondeur d'abord (Depth First Search)**
    - **le principe consiste à descendre le plus "profondément" dans le graphe à partir de s, en prenant toujours à "gauche", avant de revenir pour prendre une autre direction**
  - **Parcours en largeur d'abord (Breadth First Search)**
    - **le principe consiste à visiter les sommets situés à une distance 1 de s, puis ceux situés à une distance 2 de s, etc...**
    - **d'une autre manière, lorsqu'un sommet x est atteint, tous ses successeurs y sont visités avant de visiter les autres descendants de x.**

51

Parcours d'un Graphe (Exemple)

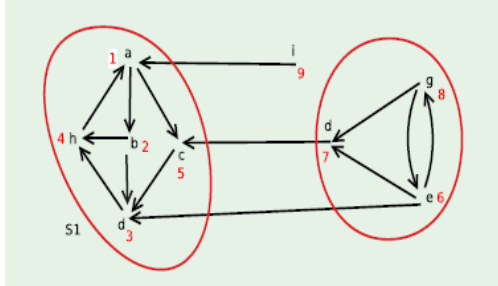


- **Parcours à partir du sommet 1 :**
  - **Parcours en profondeur : 1, 2, 5, 6, 8, 3, 7, 4**
  - **Parcours en largeur : 1, 2, 3, 4, 5, 6, 7, 8**

52

Parcours en Profondeur (Exemple)

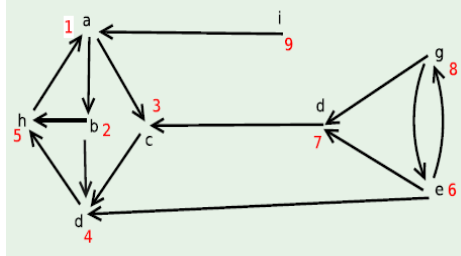
- les numéros correspondants aux sommets donnant l'ordre dans lequel les sommets sont visités.



53

Parcours en Largeur (Exemple)

- La numérotation indiquée correspond à un ordre de visite lors d'un parcours en largeur



54

Comment implémenter les deux parcours dans un graphe ?

- **Le type de parcours est fonction du TAD utilisé pour stocker les sommets à traiter :**
  - **Pile → Parcours en profondeur**
  - **File → Parcours en largeur**
- **Parcours en profondeur**
  - **Algorithme récursif : l'utilisation de la pile est implicite (appels récursifs)**
  - **Algorithme itératif : l'utilisation de la pile est explicite**
- **Parcours en largeur**
  - **Algorithme itératif : l'utilisation de la file est explicite**
- **Dans tous les cas il faut un mécanisme pour éviter de boucler indéfiniment :**
  - **Marquer les noeuds**
  - **Lister les noeud traités**

55

Algorithme de Parcours en Profondeur (1)

```
Algorithme parcoursProfondeur(g : Graphe)
 Entrée : un graphe

 Variables locales
   atteint : tableau[Sommet] de booléens
   (* atteint[x] <=> le sommet x a été atteint *)
   x : Sommet
Début
  pour tout sommet x de g faire
    atteint[x] ← faux
  fpour
  pour tout sommet x de g faire
    si non atteint[x] alors RechercheProf(x)
  fpour
Fin
```

56

Algorithme de Parcours en Profondeur (2)

```
Algorithme RechercheProf(x : Sommet)
 Entrée : un sommet d'un graphe
 (* Étant donnée un sommet x non atteint, cet
    algorithme marque x, et tous les sommets y
    descendants de x tels qu'il existe un chemin
    [x,y] dont aucun sommet n'est marqué *)

 Variables locales :
   y : Sommet

Début
 atteint[x] ← vrai
 pour tout successeur y de x faire
   si non atteint[y] alors RechercheProf(y)
 fpour
Fin
```

57

Algorithme de Parcours en Profondeur (Complexité)

- La phase d'initialisation du tableau atteint est en O(n)
- L'itération de l'algorithme principal est réalisée exactement en n étapes, pour chacune il y'a au moins un test réalisé :
  - O(n+m), soit O(max(n,m)) dans le cas des listes d'adjacences
  - $O(n^2)$ dans le cas des matrices d'adjacences

58

Parcours en profondeur
(Ordre de Traitement-Numérotation)

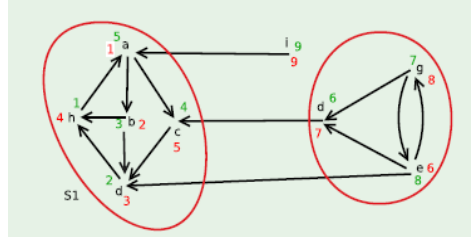
- **L'objet des parcours de graphes concerne des traitements que l'on souhaite opérer sur les graphes :**
  - Les traitements s'opèrent parfois sur les sommets visités. Il est alors possible d'opérer un traitement avant ou après la visite du sommet.
  - Il y'a donc soit **un traitement en préOrdre ou ordre préfixé**, soit **en postOrdre ou ordre postfixé**. Ceci se traduit par une modification de la procédure de recherche en profondeur.

```
Algorithme traitement_en_préOrdre(x : Sommet)
 Variables locales :
   y : Sommet
Début
 atteint[x] ← vrai
 <traiter x>
 pour tout successeur y de x faire
  si non atteint[y] alors rechercheProf(y)
 fpour
Fin
```

59

Parcours en profondeur
(Traitement en postOrdre)

```
Algorithme traitement_en_postOrdre(x : Sommet)
  Variables locales :
      y : Sommet
Début
  atteint[x] ← vrai
  pour tout successeur y de x faire
      si non atteint[y] alors rechercheProf(y)
 <traiter x>
Fin
```



Exemple : Numérotation en préOrdre(rouge) et en postOrdre(vert) pour un parcours en profondeur

Mise en œuvre en C du parcours en profondeur (1)

```
#define N_MAX 20

typedef struct cellule {
 int sommet;
 struct cellule *suiv;
} Cellule
typedef cellule* Liste;

typedef struct {
 Liste a[N_MAX];
 int n;
} GrapheL;

typedef int atteint[N_MAX];

/* variables globales déclarées:
GrapheL g;
atteint m; */
```

61

Mise en œuvre en C du parcours en profondeur (2)

```
void parcoursProf(int x) {
 Liste p;
 m[x]=1;
 p=g.a[x];
 while(p!=NULL) {
  if(!m[p->sommet]) parcoursProf(p->sommet);
  p=p->suiv;
 }
}

void main(){
 int x;
 for(x=1;x<g.n;x++) m[x]=0;
 for(x=1;x<g.n;x++)
  if(!m[x]) parcoursProf(x);
}
```

62

Mise en œuvre en C du parcours en Largeur (1)

- **Principe de l'algorithme :**
  - Il repose sur la notion de file.
  - Lors de la visite d'un sommet s, tous ses successeurs non encore atteints vont être rangés dans la file de manière à conserver la priorité liée aux distances depuis le sommet origine.

```
typedef struct {
  Liste tete,queue;
} File;

void enfiler(int x,File* f);
int defiler(File* f);
int fileVide(File* f);
```

63

Mise en œuvre en C du parcours en largeur (2)

```
void parcoursLarg(int x) {
 Liste p;
 initFileVide(&f);
 enfiler(x,&f);
 m[x]=vrai;
 while(!fileVide(f)) {
  x=defiler(f);
  p=g.a[x];
  while(p!=NULL) {
   if (!m[p->sommet]) {
     m[p->sommet]=vrai;
     enfiler(p->sommet,f);
   }
   p=p->suiv;
  }
 }
}
```

```
void main(){
 int x;
 for(x=1;x<g.n;x++) m[x]=0;
 for(x=1;x<g.n;x++){
  if(!m[x])
     parcoursLarg(x);
 }
}
```

64

Algorithme de Parcours en Largeur (Complexité)

- Identique à celle du parcours en profondeur :
  - O(n+m) pour les listes d'adjacence
  - $O(n^2)$ pour les matrices d'adjacence

65

Quelques Applications des Parcours

- **Accessibilité :**
  - Pour connaître les sommets accessibles depuis un sommet donné d'un graphe (orienté ou non), il suffit de faire un parcours en profondeur à partir de ce sommet, en marquant les sommets visités
- **Composantes connexes :**
  - Pour déterminer les composantes connexes d'un graphe, il suffit d'appliquer d'une manière répétitive le parcours DFS ou BFS sur tous les sommets non encore visités. Il est clair qu'une composante connexe est constituée du sous graphe dont les sommets sont visités par un seul appel à DFS ou BFS
- **Graphe orienté sans circuit :**
  - Un graphe orienté comporte un circuit si et seulement si, lors du parcours des sommets accessibles depuis un sommet, on retombe sur ce sommet. Pour savoir si un graphe est sans circuit, il suffit donc d'adapter DFS ou BFS, en maintenant une liste des sommets critiques (en cours de visite)
- **…**

66

Algorithme du Parcours en Profondeur Récursif

```
Algorithme parcoursEnProfondeurRecursif
                          (g : Graphe, s : Sommet )

Entrées : un graphe et un sommet

Début
 si non estMarque(s) alors
   marquer(s)
   traiter(g,s)
   pour chaque successeur s' de s faire
     parcoursEnProfondeurRecursif(g,s')
   fpour
  fsi
Fin
```

67

Algorithme du Parcours en Largeur

```
Algorithme parcoursEnLargeurIteratif (g : Graphe, s : Sommet )
 Entrées : un graphe et un sommet du graphe
 Variables locales :
  f : File<Sommet>
  sCourant : Sommet

Début
  f ← file()
  f ← enfiler(f,s)
  tantque non estVide(f) faire
    sCourant ← obtenirElement(f)
    f ← defiler(f)
    marquer(sCourant)
    traiter(g,sCourant)
    pour chaque successeur s' de sCourant faire
      si non estMarque(s') alors
        f ← enfiler(f,s')
      fsi
    fpour
  ftantque
Fin
```

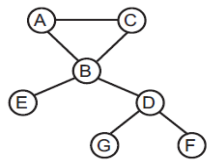
68

Graphes (Applications)

- **Algorithmes sur les graphes :**
  - Algorithmes résolvant les problèmes modélisés par les graphes. Par exemple, les problèmes liés à l'optimisation des connexions et du routage. On peut citer les algorithmes de calcul des arbres de recouvrement minimaux, la recherche des plus courts chemins, …
- **Tri topologique :**
  - Il s'agit d'un tri linéaire des sommets dans un graphe orienté acyclique de telle sorte que tous les arcs vont de gauche à droite. L'une de ses utilisations les plus courantes est de déterminer un ordre acceptable dans l'accomplissement d'un certain nombre de tâches dépendant les unes des autres
- **Coloration de graphes :**
  - On tente de donner une couleur aux sommets de façon à ce qu'il n'y ait pas deux sommets de même couleur reliés par un arc. Parfois, on s'intéresse à déterminer le nombre minimum de couleurs réalisant ce but.
- **Problèmes de cycles hamiltoniens :**
  - On travaille sur des cycles hamiltoniens, des chemins passant exactement une fois par tous les sommets d'un graphe avant de revenir au sommet de départ. Le problème du voyageur de commerce en est un cas particulier, dans lequel on recherche le circuit hamiltonien de coût minimum.
- **Problèmes de clique :**
  - On travaille sur des régions du graphe où chaque sommet est connecté d'une façon ou d'une autre à tous les autres sommets. Ces régions s'appelent des cliques. On recherche une clique maximale dans un graphe, ou une clique d'une certaine taille, …
- **…**

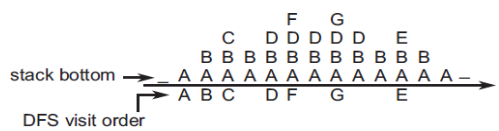
69

Parcours en Profondeur et Pile (Exemple)



	A	B	C	D	E	F	G
A	0	1	1	0	0	0	0
B	1	0	1	1	1	0	0
C	1	1	0	0	0	0	0
D	0	1	0	0	0	1	1
E	0	1	0	0	0	0	0
F	0	0	0	1	0	0	0
G	0	0	0	1	0	0	0

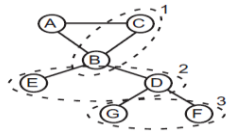
A → B → C
B → A → C → D → E
C → A → B
D → B → F → G
E → B
F → D
G → D



Contents of the queue during a depth first search

70

Parcours en Largeur et File (Exemple)



	A	B	C	D	E	F	G
A	0	1	1	0	0	0	0
B	1	0	1	1	1	0	0
C	1	1	0	0	0	0	0
D	0	1	0	0	0	1	1
E	0	1	0	0	0	0	0
F	0	0	0	1	0	0	0
G	0	0	0	1	0	0	0

A	B → C
B	A → C → D → E
C	A → B
D	B → F → G
E	B
F	D
G	D

BFS visit order	queue (front)
	A
A	B C
B	C D E
C	D E
D	E F G
E	F G
F	G
G	_

Contents of the queue during a breadth first search. The vertices grouped by the dotted ovals represent their distance from the start vertex.

71