INF 2331- Structures de données

LGI – Semestre 3

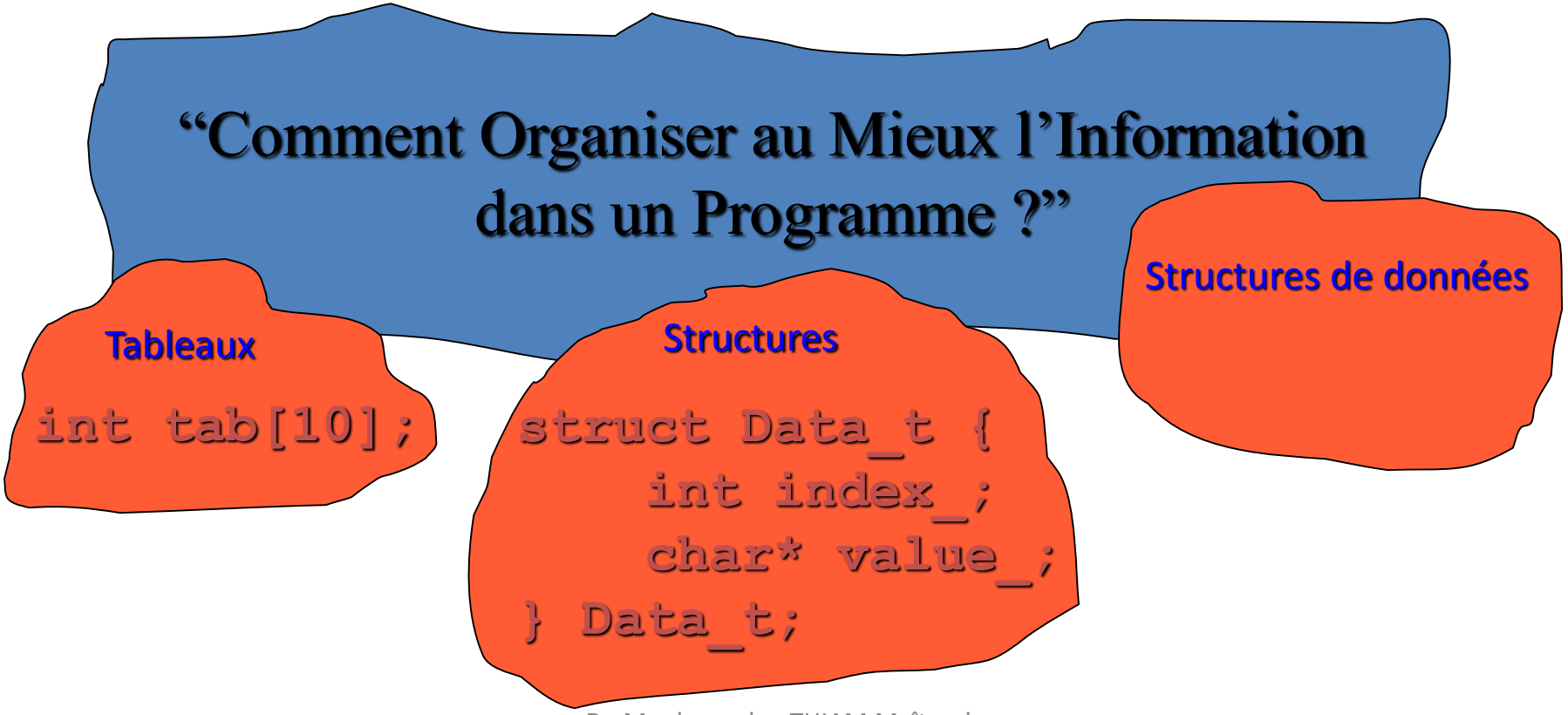
Département informatique

UFR des Sciences et technologies

Université de Thiès

Introduction

Problème métaphysique:



Structure de données

Définition Wikipédia (14/03/2015)

- une structure logique destinée à contenir des données afin de leur donner une organisation permettant de simplifier leur traitement.

- **Exemple :** On peut présenter des numéros de téléphone *

  - par département,
  - par nom
  - par profession (pages jaunes),
  - par numéro téléphonique (annuaires destinés au télémarketing),
  - par rue et/ou
  - une combinaison quelconque de ces classements.

  À chaque usage correspondra une structure d'annuaire appropriée.

Objectifs du cours

- Connaître les principales structures de données manipulées dans un programme
- Connaître les notions de structures linéaires
- Connaître les notions de structures arborescentes
- Connaître des arbres particuliers
- Connaître les principes de recherche rapides
- Savoir appliquer les concepts de base ci-dessus à la programmation (tp)
- Connaître les notions de graphes (facultatif)

Références

- Web
- Aho et al. *Structures de données et algorithmes*, Addisson-Wesley / InterEditions. 1989.
- Aho et Ullman. *Concepts fondamentaux de l'informatique*, Dunod. 1993.
- Sedgewick. *Algorithmes en C*. Addisson-Wesley / InterEditions. 1991.
- Alfred Aho John Hopcroft, Jeffrey Ullman : Structures de données et algorithmique

Overview

1. CHAPITRE 0 : PRÉSENTATION DU COURS
2. CHAPITRE 1 : LES STRUCTURES LINÉAIRES
3. CHAPITRE 2 : ARBRES ET ARBORESCENCES
4. CHAPITRE 3 : QUELQUES ARBRES PARTICULIERS
5. CHAPITRE 4 : RECHERCHES RAPIDES

Overview

1. CHAPITRE 0 : PRÉSENTATION DU COURS
2. CHAPITRE 1 : LES STRUCTURES LINÉAIRES
3. CHAPITRE 2 : ARBRES ET ARBORESCENCES
4. CHAPITRE 3 : QUELQUES ARBRES PARTICULIERS
5. CHAPITRE 4 : RECHERCHES RAPIDES

PRÉSENTATION DU COURS

Présentation générale

- **Unité d'Enseignement**
  - **Titre :** INFORMATIQUE
  - **Sigle** : **INF 233**
- **Élément constitutif**
  - **Titre :** Structures de données
  - **Sigle** : **INF 2331**
- **Autres éléments constitutifs de l'UE (1)**
  - Analyse et Conception des Systèmes d'Information (**INF 2332**)
  - Bases de données (**INF 2333)**

Volume horaire & Notation

- **CM :** 30H
- **TD/TP :** 20H
- **TPE :** 50H
- **Coefficient de l'UE :** 4
- **Crédits de l'UE :** 12
- **Evaluation**
  - **Contrôle des connaissances :** 40%
  - **Examen écrit :** 60%

Responsables

- **Magistral**

  Dr. Mouhamadou THIAM
  Maître de conférences en Informatique
  Intelligence Artificielle : Sémantique Web
  Email : mthiam@univ-thies.sn

- **Travaux dirigés et pratiques**

  M. Papa DIOP
  Ingénieur Systèmes Informatiques et Bases de Données
  Diplômé de l'Université Gaston Berger de Saint-Louis
  Email : papaddiop@gmail.com

Overview

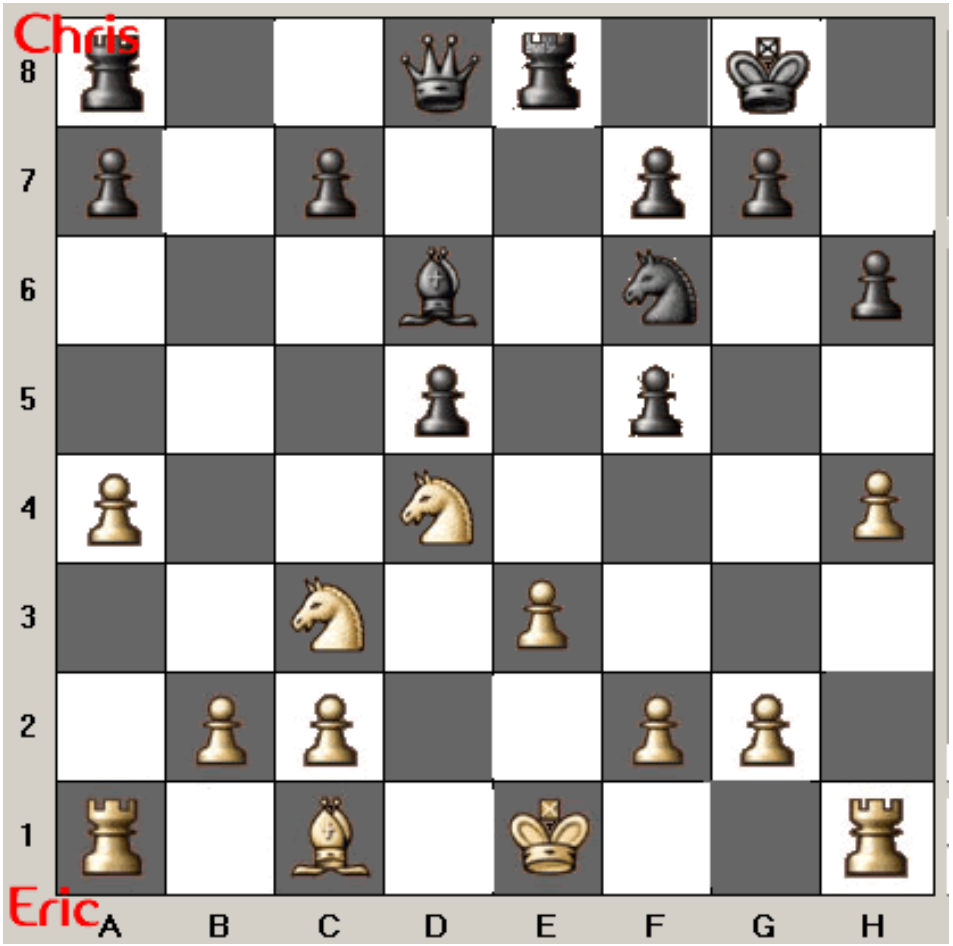
1. CHAPITRE 0 : PRÉSENTATION DU COURS
2. CHAPITRE 1 : LES STRUCTURES LINÉAIRES
3. CHAPITRE 2 : ARBRES ET ARBORESCENCES
4. CHAPITRE 3 : QUELQUES ARBRES PARTICULIERS
5. CHAPITRE 4 : RECHERCHES RAPIDES

CHAPITRE 1 : LES STRUCTURES LINÉAIRES

1) Les tableaux
2) Les pointeurs
3) Les listes chainées
4) Les piles
5) Les queues ou files
6) Applications aux matrices creuses (listes orthogonales)

TABLEAUX

Motivation



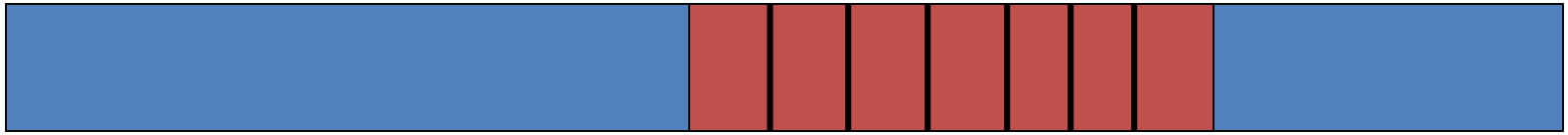
Structure de donnée:

- tableau a 2 dimension

Algorithmes:

- surtout I.A.

Les tableaux



Accès indexé (de 0 à n-1 pour un tableau de n éléments)

Stockage compact

Taille fixe, en général

Réajustement de taille coûteux en temps

Insertion d'élément onéreuse en temps.

QUESTION

- Qui n’est pas
  - Excellent
  - Très bien
  - Bien
  - Passable
  - **Tu peux reprendre tes siestes matinales du lundi**

POINTEURS - RAPPELS

Intuition

- ***Kernighan et Ritchie dans "programming in C"***

« ... Les pointeurs étaient mis dans le même sac que l'instruction ***goto*** *comme une excellente technique de formuler des programmes incompréhensibles. Ceci est certainement vrai si les pointeurs sont employés négligemment, et on peut facilement créer des pointeurs qui pointent 'n'importe où'. Avec une certaine discipline, les pointeurs peuvent aussi être utilisés pour programmer de façon claire et simple. C'est précisément cet aspect que nous voulons faire ressortir dans la suite. ... »*

Définition

- **Pointeur** : *une variable spéciale qui peut contenir l'**adresse** d'une autre variable.*

- La plupart des langages de programmation offrent la possibilité d'accéder aux données dans la mémoire de l'ordinateur à l'aide de ***pointeurs.***

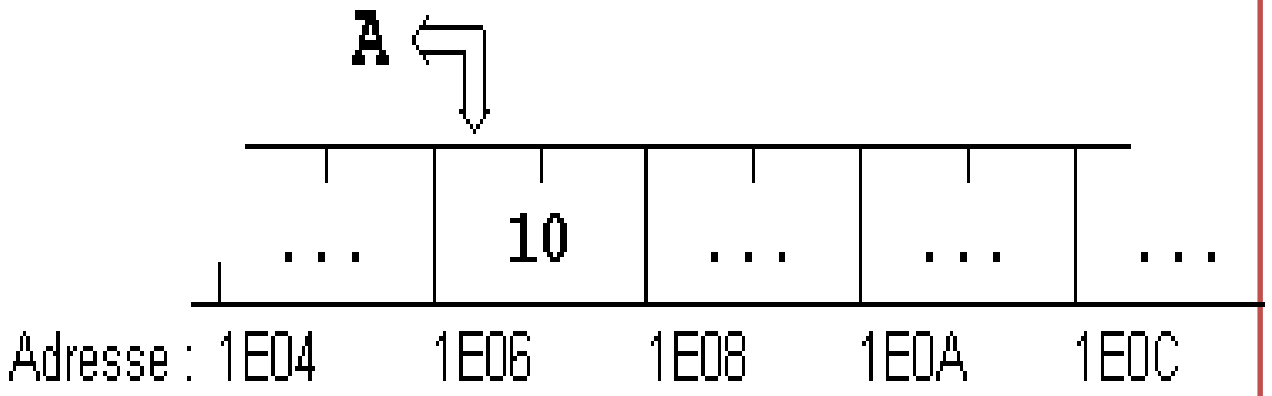
Cas du langage C

- Ils jouent un rôle primordial dans la définition de fonctions
  - passage des paramètres fait toujours par valeur
  - pointeurs sont le seul moyen de passe des variables par adresse.
  - traitement de tableaux et de chaînes de caractères dans des fonctions serait impossible sans l'utilisation de pointeurs

Adressage de variables (1)

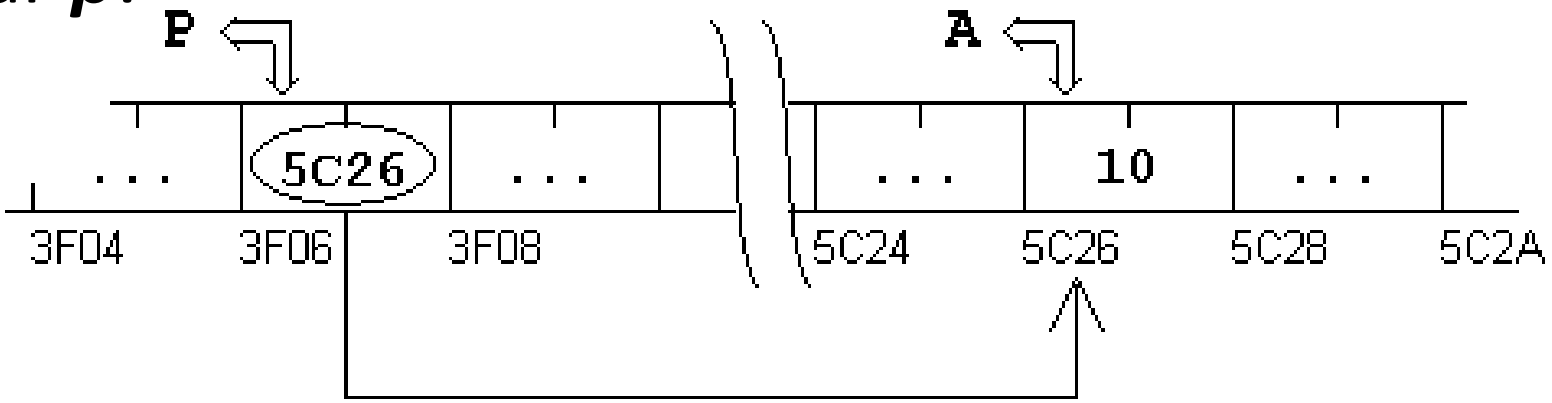
- **Adressage direct :** Accès au contenu d'une variable par le nom de la variable.
- ***Exemple :***

```
short A;
A = 10;
```



Adressage de variables (2)

- **Adressage indirect** : copier l'adresse d'une variable ***a*** (dont nous ne pouvons ou ne voulons pas utiliser) dans un pointeur ***p,*** on peut accéder à l'information de ***a*** en passant par ***p***.



Utilisation de Pointeurs

- Limité à un type de données
- Si ***p*** contient l'adresse de ***a*** alors ''***p*** pointe sur ***a***''
- Un pointeur peut pointer sur différentes adresses
- Le nom d'une variable reste lié à la même adresse

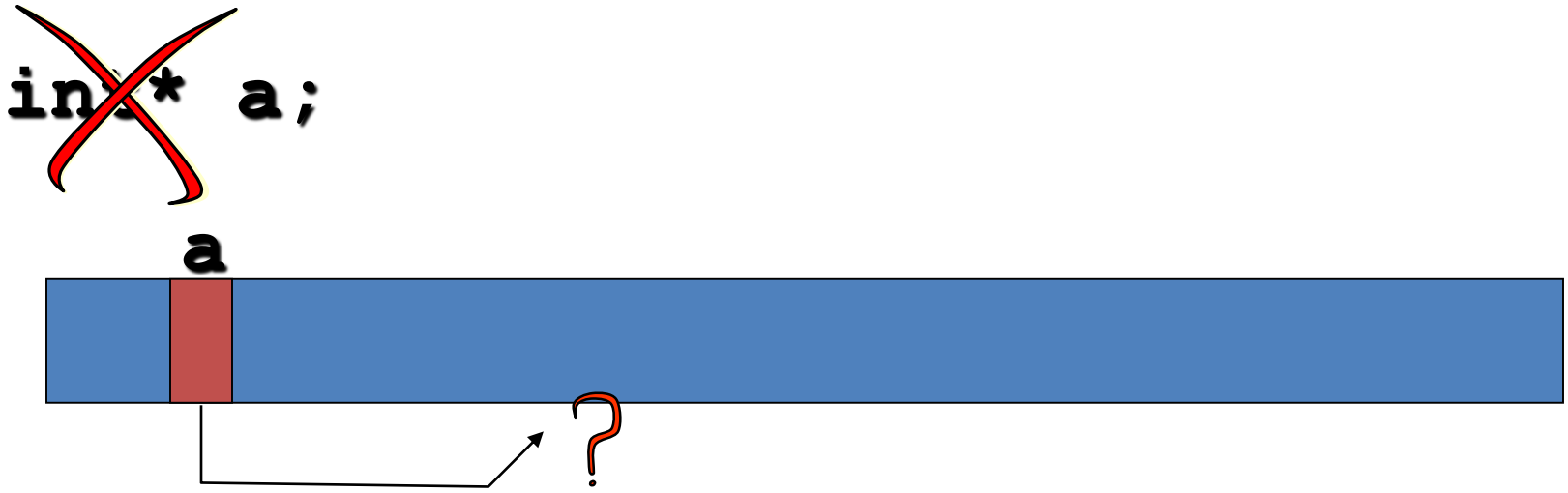
Opérateurs sur les Pointeurs

- Opérateurs de base
  - Adresse de : **&**
  - Contenu de : ***

Déclaration de pointeur (1)

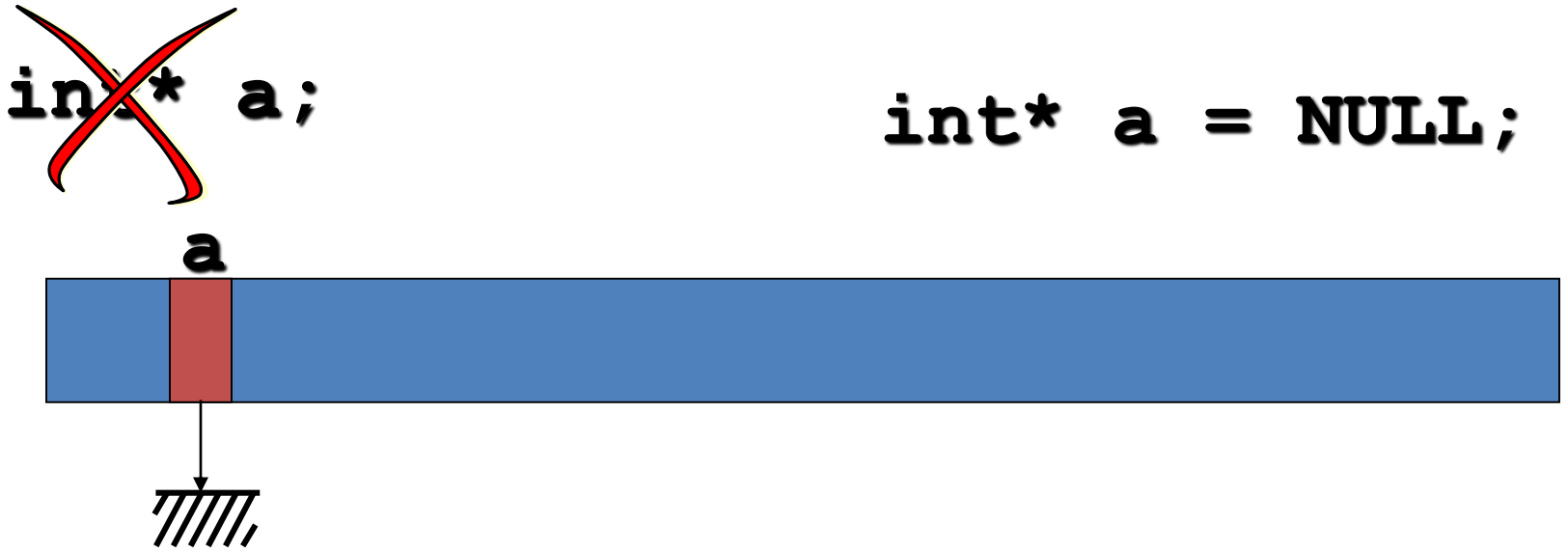
- Déclaration:
  - L'instruction <***Type***> * <***nom_pointeur***>
    - Déclare un pointeur ***nom_pointeur***
    - Reçoit des adresses de variables de type <***Type***>
    - **Exemple** : *int * P*;

Déclaration de pointeur (2)



- Déclaration d'un pointeur vers un entier

Déclaration de pointeur (fin)



- ***Déclaration*** d'un pointeur vers un entier ***et initialisation*** à "NULL"

Utilisation de pointeurs (1)

- *<***nom_pointeur***> désigne le contenu de l'adresse référencée par le pointeur.
- Exemple
  - Int * P;
  - Int A=10, B;
  - P=&A;
  - **B=*P;** // B=10;
  - *P=90; // A = 90

Utilisation de pointeurs (2)

- &<***nom_var***> → l'adresse de la variable
  - S'applique à des variables et des tableaux.
  - Non à des constantes ou expressions

  - **Exemple :**
    - *int N;*
    - *printf("Entrez un nombre entier : ");*
    - *scanf("%d", &N);*

Opérations sur les pointeurs

- Opérations élémentaires sur pointeurs
  - Priorité de **&** et * : même priorité que les opérateurs unaires (!, ++, --).
  - Dans une expression ils sont tous évalués de droite à gauche.
  - Si ***P*** pointe sur ***X*** alors ***P** et **X** sont interchangeables

Incrémentation de pointeur

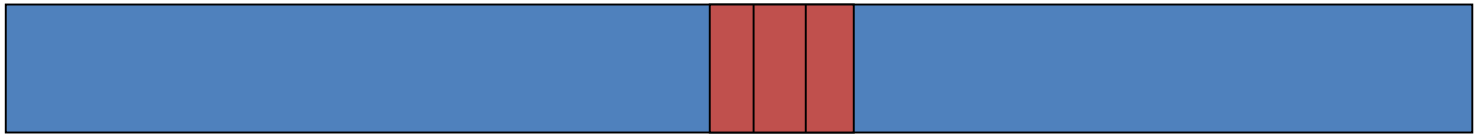
- Exemple : après l'instruction **P = &X**;
  - **Y = *P+1 Y=X+1;**
  - ***P = *P+10 X=X+10;**
  - **++*P ++X**
  - **(*P)++ X++ (parenthèses obligatoires)**

Pointeur *NULL*

- Zéro (0) est utilisé pour dire qu'un pointeur ne pointe ''nulle part''‘
- Exemple: *int * p; p = 0;*
- Les pointeurs sont des variables
  - *Int * p1, *p2; → p1=p2;*

Allocation d'espace (1)

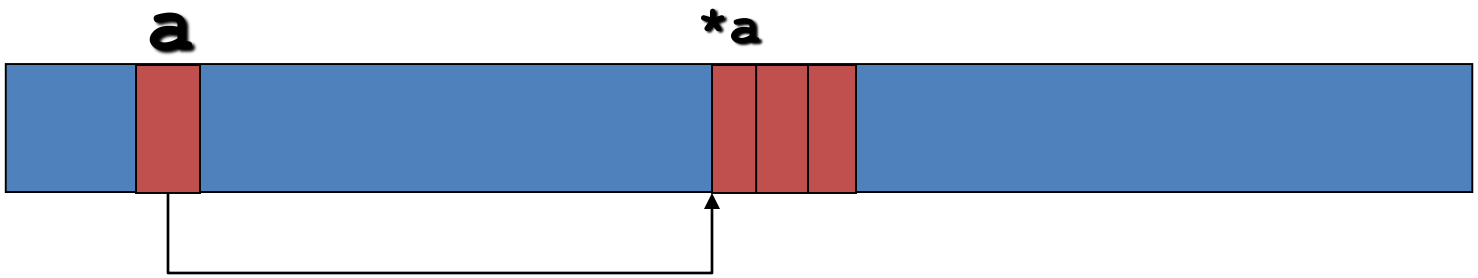
- `malloc(3*sizeof(int));`



- ***Allocation dynamique*** de place mémoire (pour 3 entiers)

Allocation d'espace (2)

- ~~int* a = malloc(3*sizeof(int));~~
- int* a = (int*)malloc(3*sizeof(int));

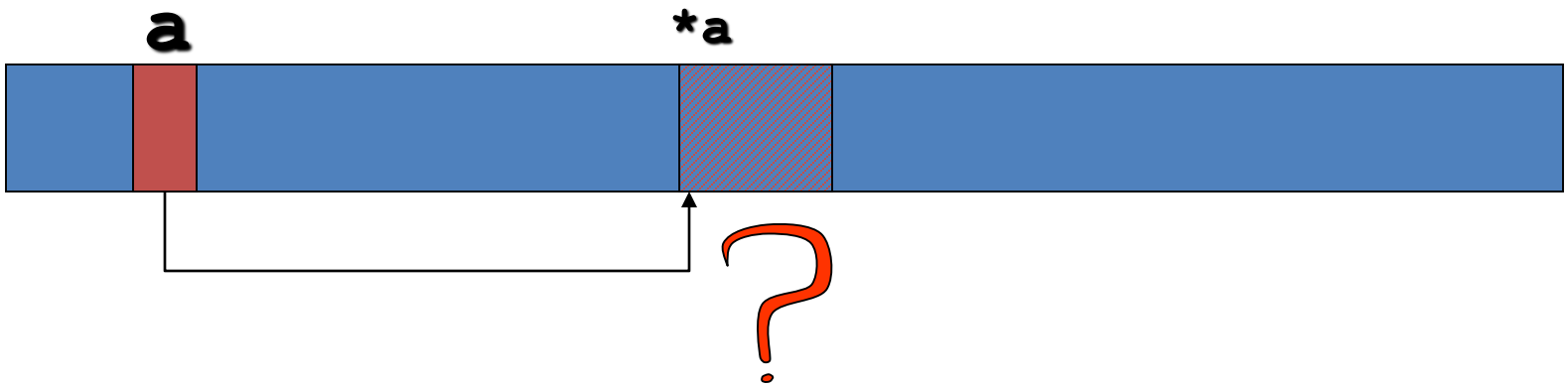


- ***Allocation dynamique*** de place mémoire (pour 3 entiers) et **assignation**

Libération de la mémoire

```
free(a);
```

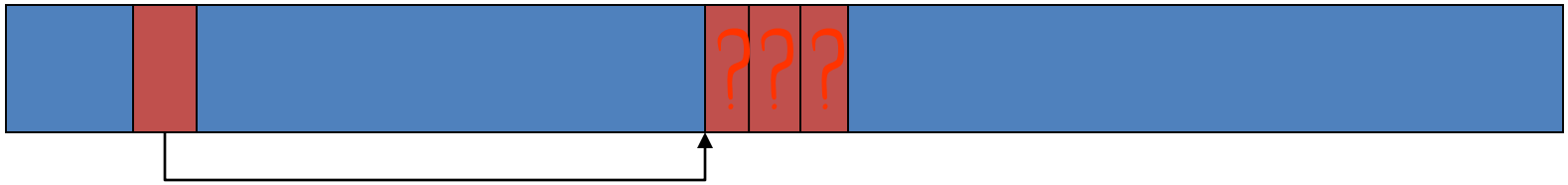
```
a = NULL;
```



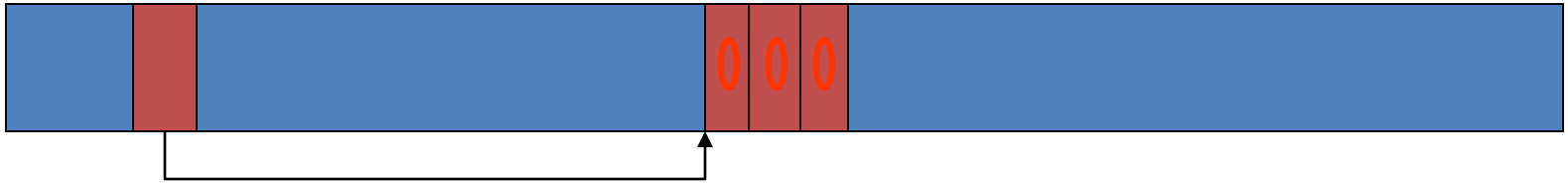
Désallocation dynamique

Réallocation de mémoire

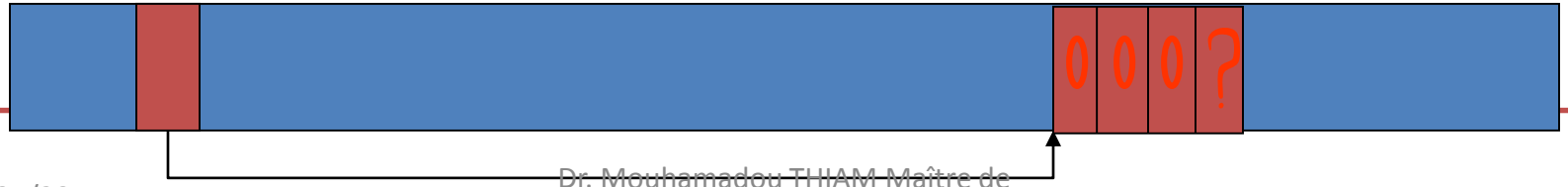
- `int* a = (int*)malloc(3*sizeof(int));`



- `int* a = (int*)calloc(3, sizeof(int));`



- `a = (int*)realloc(4*sizeof(int));`



Remarques

- Pb de malloc pour les tableaux: pas d'initialisation. → calloc

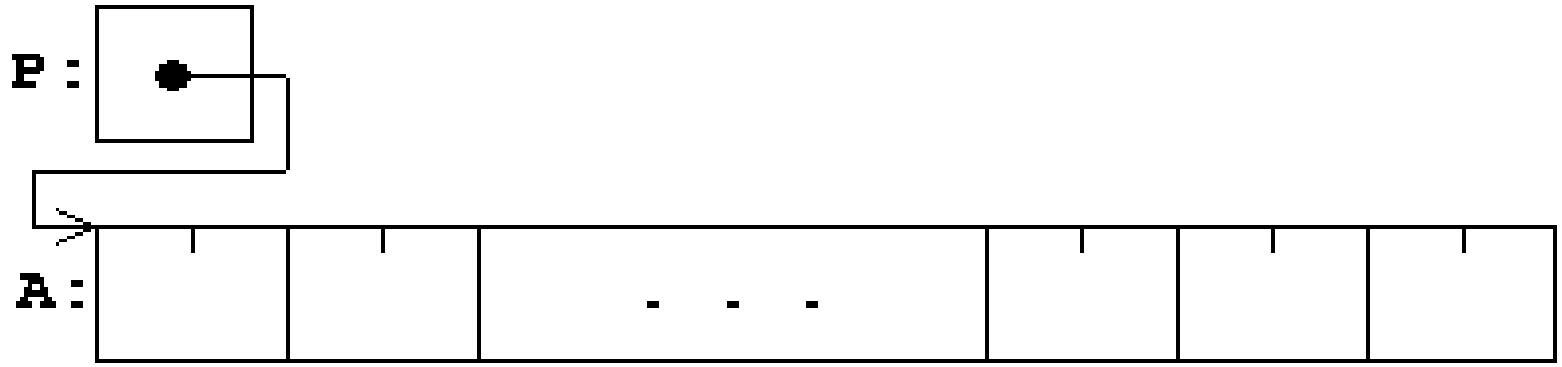
- Coût mémoire de realloc:
  - Alloue un nouveau bloc de mémoire
  - Copie les valeurs de l'ancien bloc dans le nouveau
  - Libère l'ancien bloc
  - Retourne l'adresse du nouveau bloc

Pointeurs et tableaux (1)

- Étroite relation entre pointeur et tableau
- Toute opération avec indices peut être faite à l'aide de pointeurs
- Adressage des composantes d'un tableau
  - nom Tableau = adresse de son premier élément
  - ***&tableau[0]*** et ***tableau*** sont une seule et même adresse
  - Nom tableau = pointeur constant sur son premier élément

Pointeurs et tableaux (2)

- ***Exemple*** :
  - **A** tableau de type **int** : **int A[10]**
  - **P** pointeur sur **int : int *P;**
  - **P = A;** équivalente à **P = &A[0];**



Pointeurs et tableaux (3)

- Si p pointe sur une composante du tableau
  - **P+1** pointe sur la composante suivante
  - **P+i** pointe sur la i-ième composante derrière **P**
  - **P- i** pointe sur la i-ième composante devant **P**.

- Ainsi, après l'instruction **P = A;** (A un tableau)
  - le pointeur P pointe sur A[0], et
  - ***(P+1)** désigne le contenu de A[1]
    ***(P+2)** désigne le contenu de A[2]
  - **…**
  - ***(P+i)** désigne le contenu de A[i]

Pointeurs et tableaux (fin)

- Différences entre un pointeur et un tableau
  - Un pointeur est une variable
    - P=A et P++ sont des opérations permises
  - Un nom de tableau est une constante
    - A=P ou A++ ne sont pas permises

Intérêts des pointeurs

- Gestion de l'espace mémoire en cours d'exécution
- Modifications de variables passées en paramètres de fonction
- Représentation de tableaux: accès direct et indexé
- Références croisées
- Fonctions virtuelles en programmation objet

LISTES CHAINÉES

T.D.A : Définition

Un *type de données abstrait* est composé d'un ensemble d'objets, similaires dans la forme et dans le comportement, et d'un ensemble d'opérations sur ces objets.

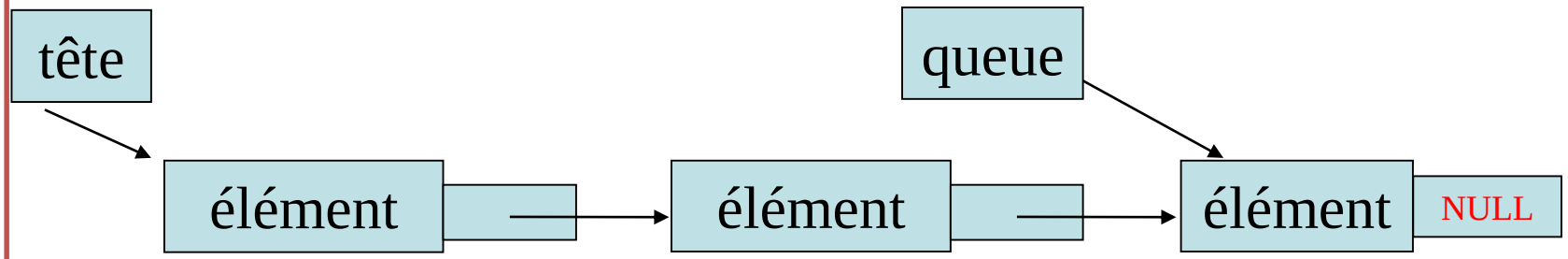
L'implémentation d'un T.D.A. ne suit pas de schéma préétabli. Il dépend des objets manipulés et des opérations disponibles pour leur manipulation.

T.D.A : Avantages

- prise en compte de types complexes.
- séparation des services et du codage.
  - L'utilisateur d'un TDA n'a pas besoin de connaître les détails du codage.
- écriture de programmes modulaires.

Les listes chaînées

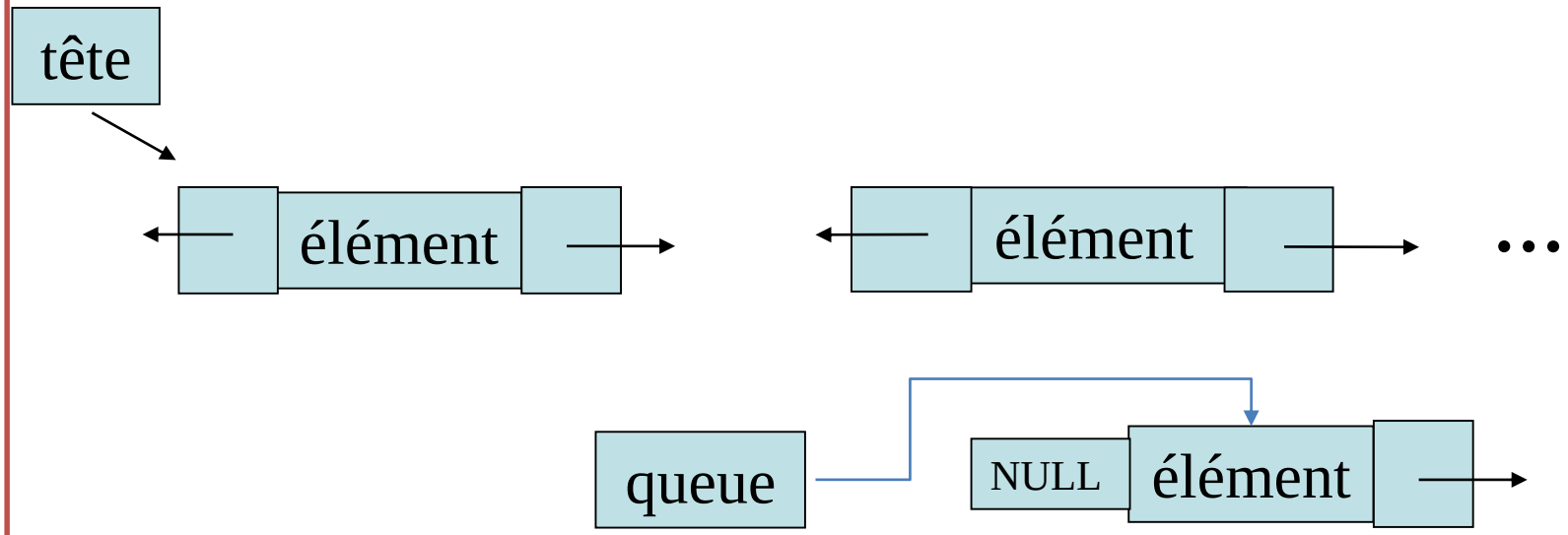
Définition : Une liste chaînée est composée d'une suite finie de cellules (ou couples) formées d'un élément et de l'adresse (ou référence) vers l'élément suivant.



- Le 1er élément est sa tête
- Le dernier est sa queue
- Le pointeur dernier élément a une valeur sentinelle (NULL)

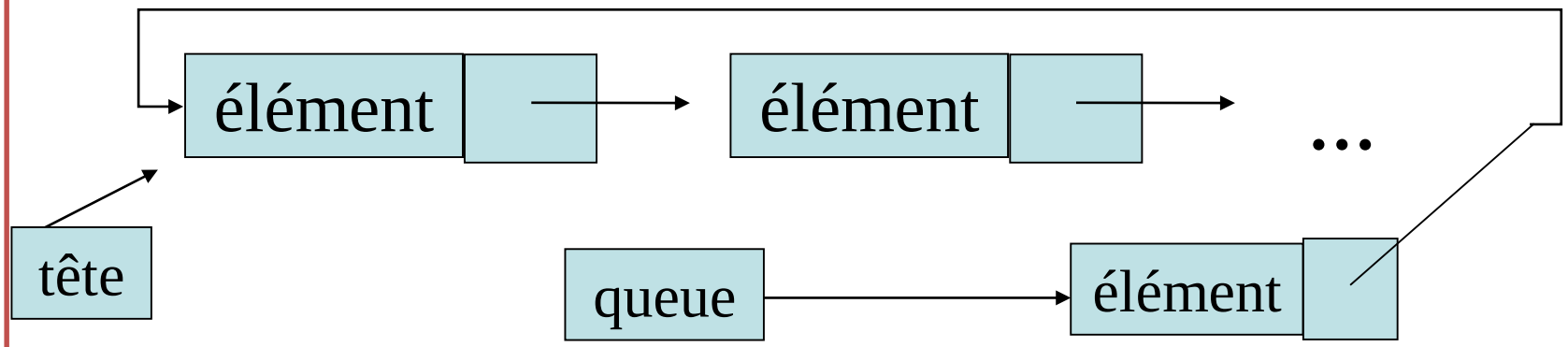
Les listes doublement chaînées

Les cellules d'une liste doublement chaînée admettent aussi un pointeur vers le précédent

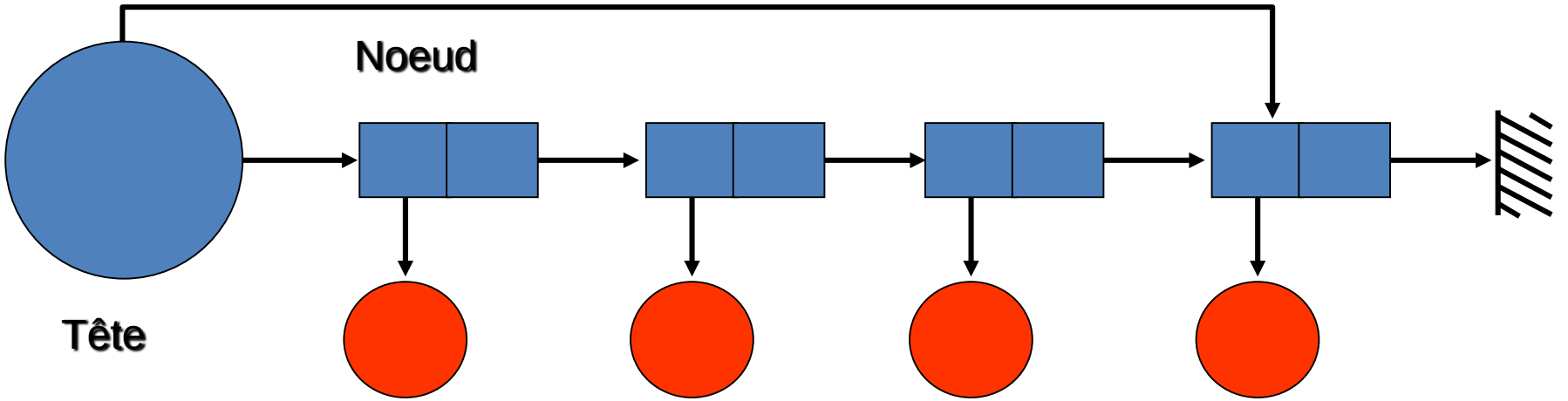


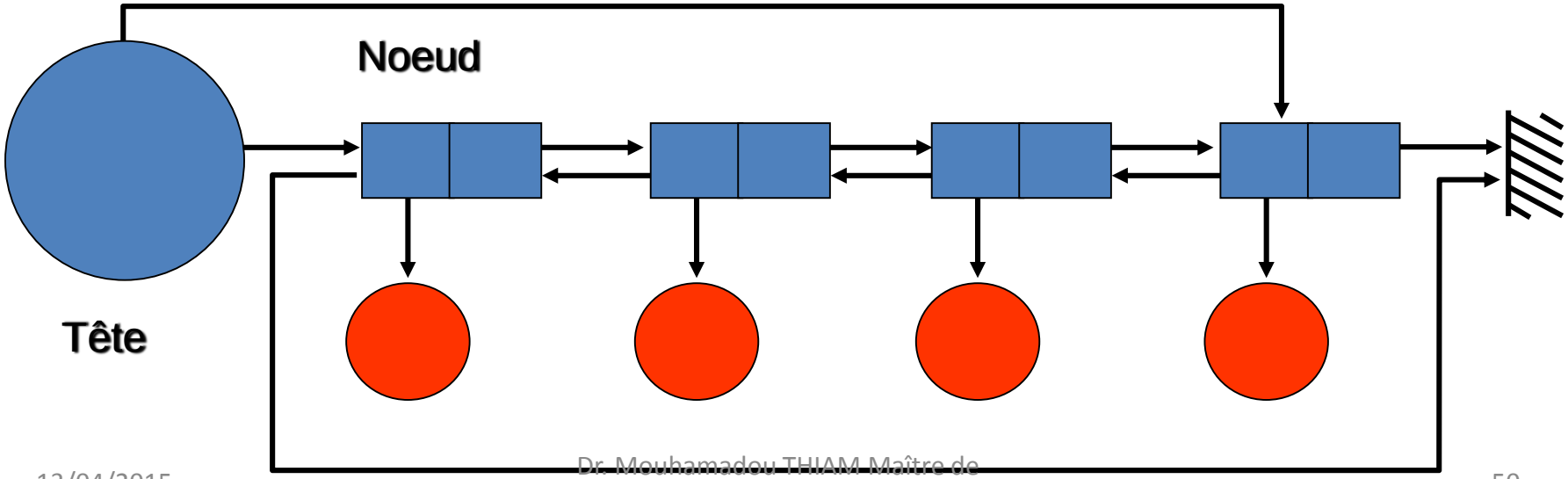
Les listes chaînées circulaires

Définition : Une liste chaînée circulaire admet la même structure qu'une liste classique mais le champs « suivant » de la dernière cellule contient l'adresse de la première cellule.

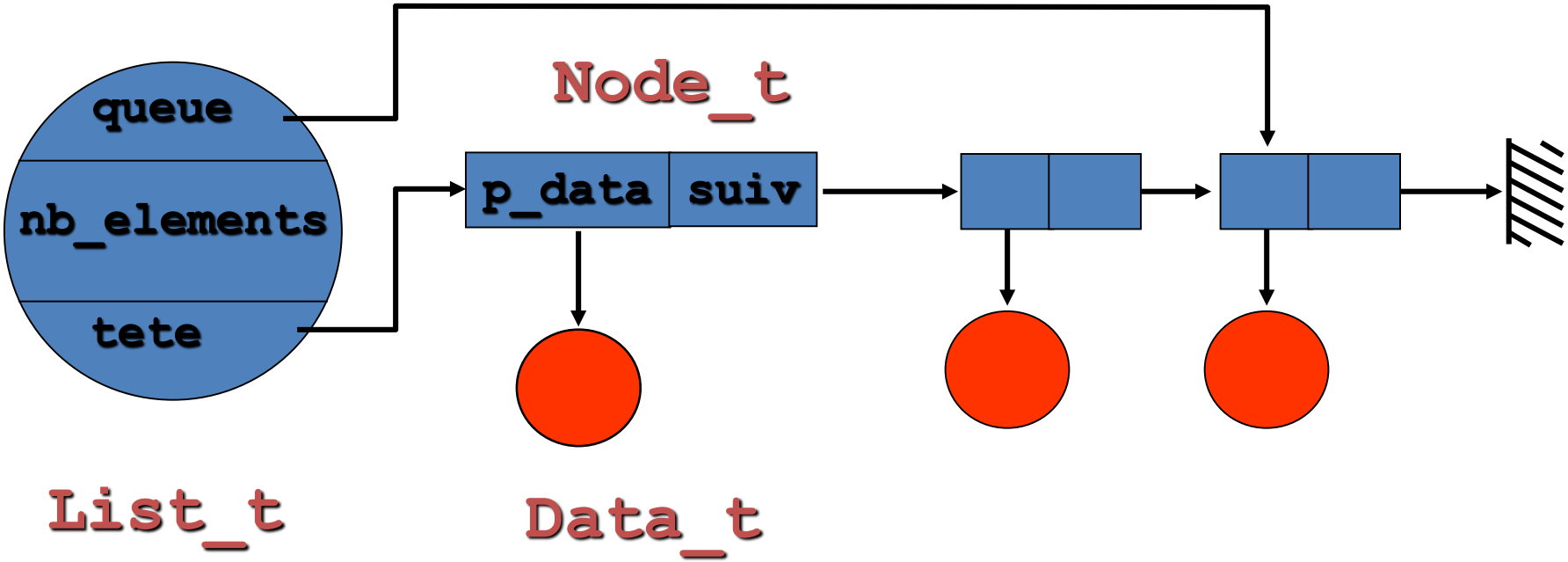


Structures





Structures



Spécifications

- Créer une liste vide
- Créer une liste avec le 1er élément
- Ajouter un élément (début / fin / milieu)
- Concaténer 2 listes
- Retirer un élément (début / fin / milieu)
- Supprimer un élément particulier
- Détruire une liste
- Trier les éléments d'une liste
- Permuter 2 éléments

Liste chaînée : Header

```
typedef struct List_t {
    struct Node_t* tete;
    struct Node_t* queue;
    int nb_elements_;
} List_t;
```

```
typedef struct Node_t {
    struct Data_t* p_data_;
    struct Node_t* suiv;
} Node_t;
```

```
typedef struct Data_t {
    ...
} Data_t;
```

Liste chaînée : Header

```
• List_t* list_create( void );

• int list_insert_item(
    List_t* list, Data_t* item
  );

• int list_append_item(
     List_t* list, Data_t* item
  );

• int list_insert_item_before(
    List_t* list,
    Data_t* to_insert,
     Data* list_item
  );
```

Liste chaînée : Header

- `int list_destroy( List_t* list );`
- `int list_empty( List_t* list );`
- `Data_t* list_remove_head( List_t* list );`
- `Data_t* list_remove_tail( List_t* list );`
-
```
int list_remove_item(
  List_t* list
  Data_t* item
);
```
- `int list_sort( List_t* list );`

Liste chaînée : Utilisation

Avant d'aller plus loin, vérifions si nos spécifications sont suffisantes...

Pour cela, nous allons écrire un programme qui utilise les fonctions du fichier **list.h**, sans nous préoccuper de la façon dont elles sont implantées.

But du programme: construire et trier une liste d'entiers par ordre croissant.

Liste simplement chaînée

- Définir le T.D.A d'un nœud

```
struct nombre{
    int val;
    struct nombre * suiv;
}* Node_t, NBR;
```

Création de la liste

- Définir le T.D.A de la liste

```
typedef struct List_t {
        Node_t tete;
        Node_t queue;
        int nb_elements;
} List_t, *L;
```

Création d'un nœud

```
NBR createNode (int n){
    Node_t s = (Node_t)malloc(sizeof(NBR));
    s→val = n;
    s→suiv = NULL;
    return s;
}
```

Création de la liste vide

```
void createListe (L * suite){
    *suite→tete = NULL;
    *suite→queue = NULL;
    *suite→nb_elements = 0;
}
```

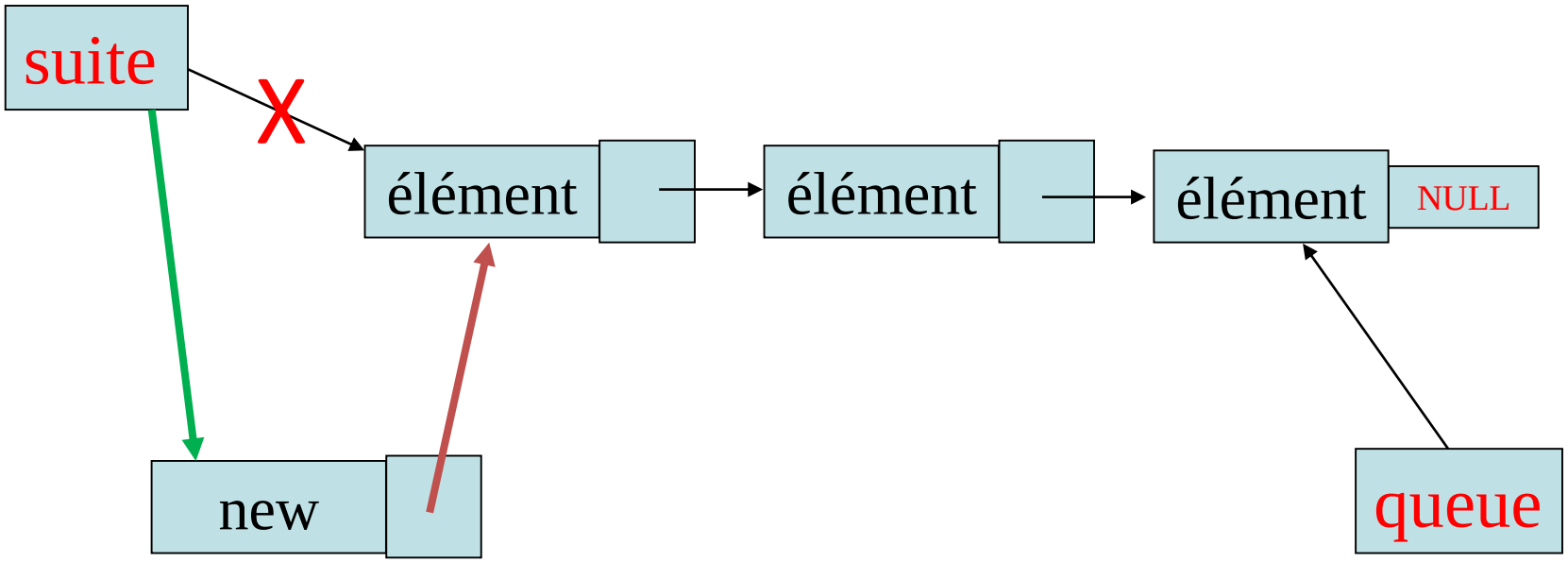
Création de la liste

```
void create (int n, L * suite){
    *suite→tete = createNode(n);
    *suite→queue = *suite→tete ;
    *suite→nb_elements = 1;
}
```

Affichage de la liste

```
void show (L suite){
      Node_t p = suite→tete;
     while (p!=NULL){
         printf("%d ", p→val);
         p = p→suiv;
    }
}
```

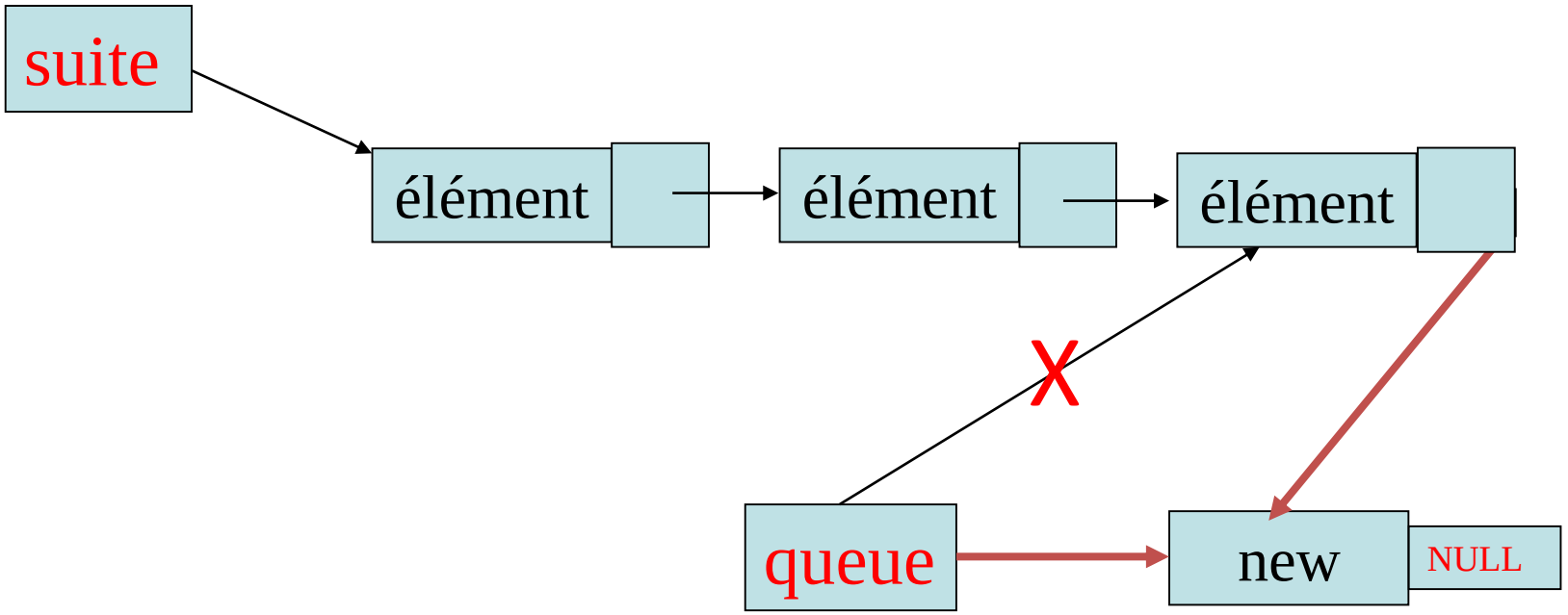
Ajouter en tête de liste



Ajouter en tête de liste

```
void ajoutDeb (L * suite, int n){
    Node_t s = createNode (n);
    s→suiv = *suite→tete;
    *suite→tete = s;
    If (*suite→queue ==NULL)
                *suite→queue = *suite→tete;
    *suite→nb_elements += 1;
}
```

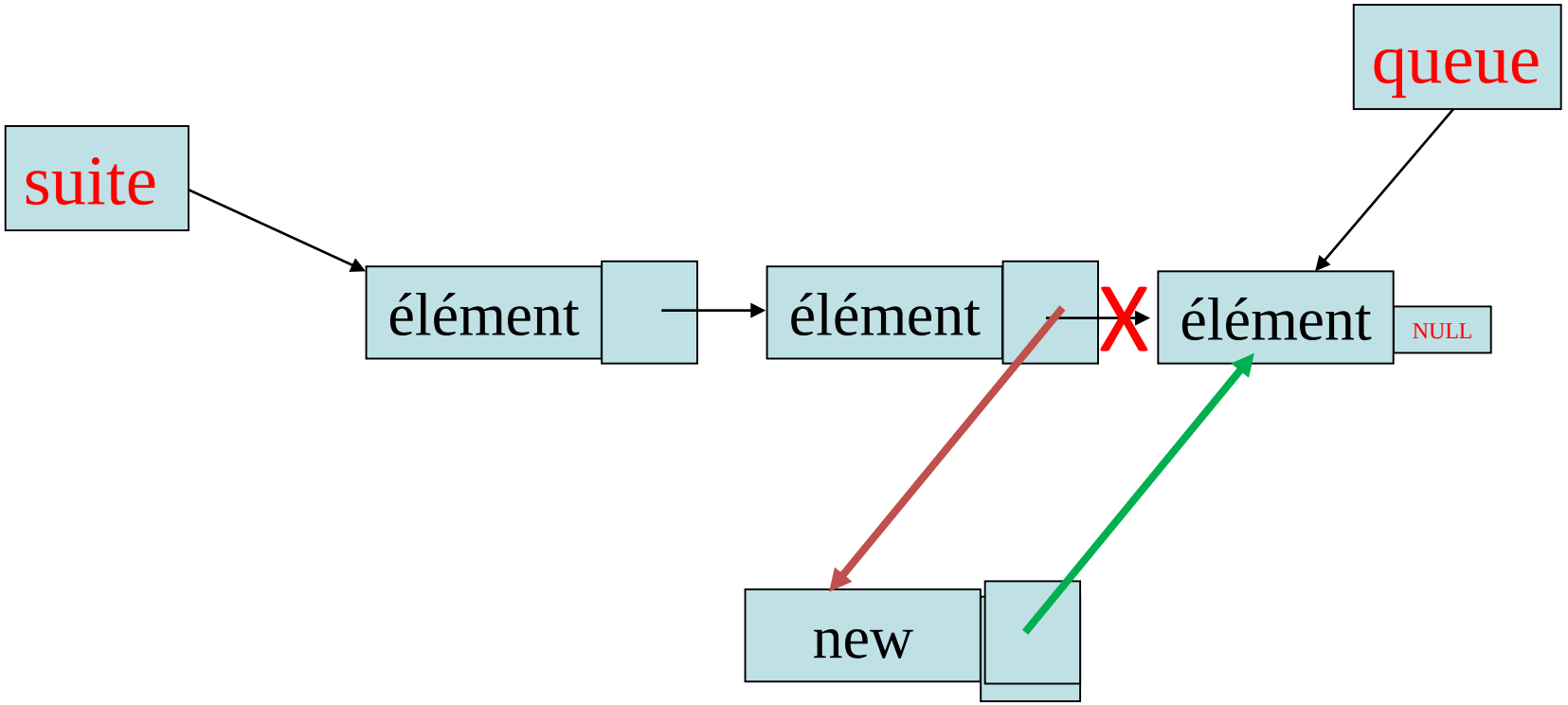
Ajouter en queue de liste



Ajouter en queue de liste

```
void ajoutFin (L* suite, int n){
        Node_t   s = createNode (n),          //nouvel élément
                 p=*suite→tete;               //pointeur de parcours
        if (p == NULL){
                 *suite→tete=*suite→queue = s;
                 nb_elements = 1;
        }
        else{
                 while (p→suiv != NULL)
                          p = p→suiv;
                 p→suiv = s;
                 *suite→queue = s;
                 nb_elements +=1;
        }
}
```

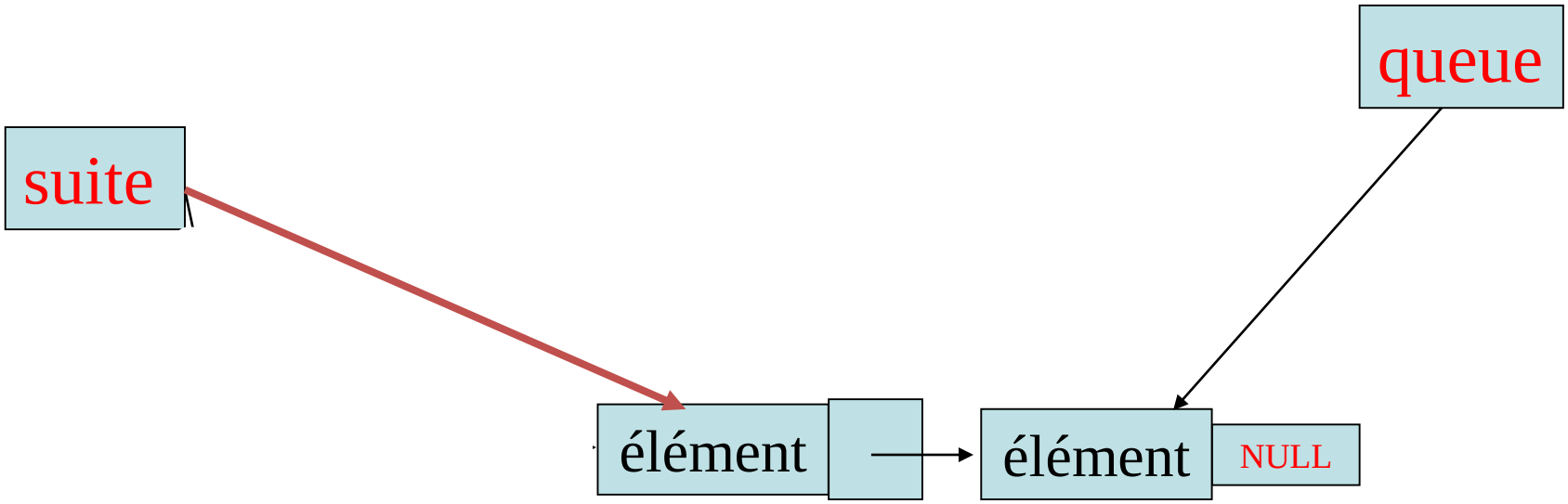
Ajouter dans la liste



Ajouter dans la liste

```
void ajoutDans (Node_t * suite, int n, int m){
 Node_t   s = createNode (n), //nouvel élément
                p=*suite→tete;       //pointeur de parcours
   while (p != NULL && p→val != m)
                p = p→suiv;
   s → suiv = p→suiv
   p→suiv = s;
   if (s→suiv == NULL) *suite→queue = s;
}
```

Retirer en tête de liste



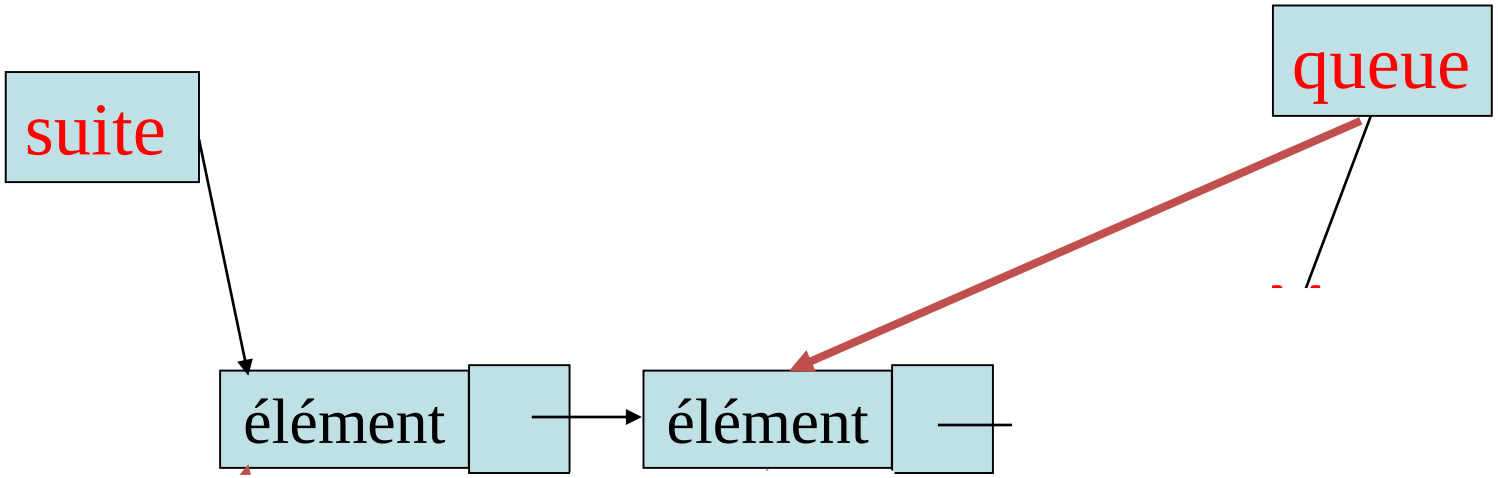
Retirer en tête de liste

- Créer un pointeur de parcours
- Faire pointer ce dernier sur 1er élément
- Rediriger la tête sur son suivant
- Rediriger le suivant de p vers NULL
- Libérer le nouvel élément

Retirer en tête de liste

```
void supDeb (L* suite){
    Node_t p = *suite→tete;
    *suite→tete = *suite→tete→suiv;
    p → suiv = NULL;
    free (p);
}
```

Retirer en fin de liste



Retirer en fin de liste

- Créer 2 pointeurs de parcours
- Faire pointer l'un sur 1er élément
- Parcourir la liste jusqu'au dernier élément
- Garder un pointeur sur son précédent
- Libérer le dernier élément
- Rediriger la queue sur le précédent (q)
- Rediriger q vers NULL

Retirer en fin de liste

```
void supFin (L* suite){
   Node_t p = *suite→tete,   //pointeur de parcours
      q =NULL;               //garde le dernier élément parcouru
   while (p!=NULL && p→suiv != NULL){
      q = p;
      p = p → suiv;

      q→suiv = NULL;
      *suite→queue = q;
      q = NULL;
      free (p);
    }
}
```

Retirer un élément donné

- Créer 1 pointeurs de parcours
- Faire pointer sur 1er élément
- Parcourir la liste jusqu'au précédent de l'élément
- Garder l'élément dans un pointeur
- Rediriger le précédent sur le suivant de l'élément
- Libérer l'élément

Retirer un élément donné

```
void supElem (L* suite, int val){
        Node_t p = *suite→tete,   //pointeur de parcours
                        q=NULL;
        while (p && p→val !=val){
                        q=p;
                        p = p → suiv;
        }
        if (p ) {//p!=NULL
                        q→suiv = p→suiv;
                        if (p→suiv == NULL) *suite→queue = q;
                        else p→suiv = NULL;
                        q = NULL;  free (p);
        }
}
```

Fonctions utiles …

- Vider/Détruire la liste
- Échanger 2 éléments
- Trier la liste
- Concaténer des listes
- Insérer avant/après un élément
- Déterminer la longueur d'une liste

Vider/Détruire la liste

```
void destroy (NBR * suite) {
      while (*suite != NULL)
            supFin (suite);
}
void destroy (NBR * suite) {
      while (*suite != NULL)
            supDeb (suite);
}
```

Échanger 2 éléments

- Plusieurs méthodes
  - Simple → échanger valeurs
  - Complexe → échanger pointeurs
- Dépendant de la question posée

Trier la liste

- Respecter les règles de
  - Insertion (ajout)
  - Retrait
  - Modification
- Utiliser algorithme de tri standard

Concaténer 2 listes

- Faire pointer le dernier élément de la 1ère liste sur le premier élément de la 2nde liste

Longueur d'une liste

```
int size (NBR  suite) {
      if (suite == NULL)
            return 0;
      return 1 + size (suite->suiv);
}
```

Erreurs à gérer

- pour résumer les principales règles à suivre:
  - Toujours tester la validité d'un pointeur avant de l'utiliser.
  - S'assurer de ne jamais perdre l'adresse d'une zone allouée dynamiquement.
  - Dans un programme, toute allocation par **`malloc`** ou **`calloc`** doit être suivie d'une désallocation par **`free`**

Exercices d'application

- Ecrire les versions récursives de toutes les fonctions vues jusque là
  - Longueur de la liste
  - Afficher une liste
  - Afficher une liste dans l'ordre inverse
  - Ajouter un élément
  - Supprimer un élément
  - Détruire/vider une liste

PILES

Motivation

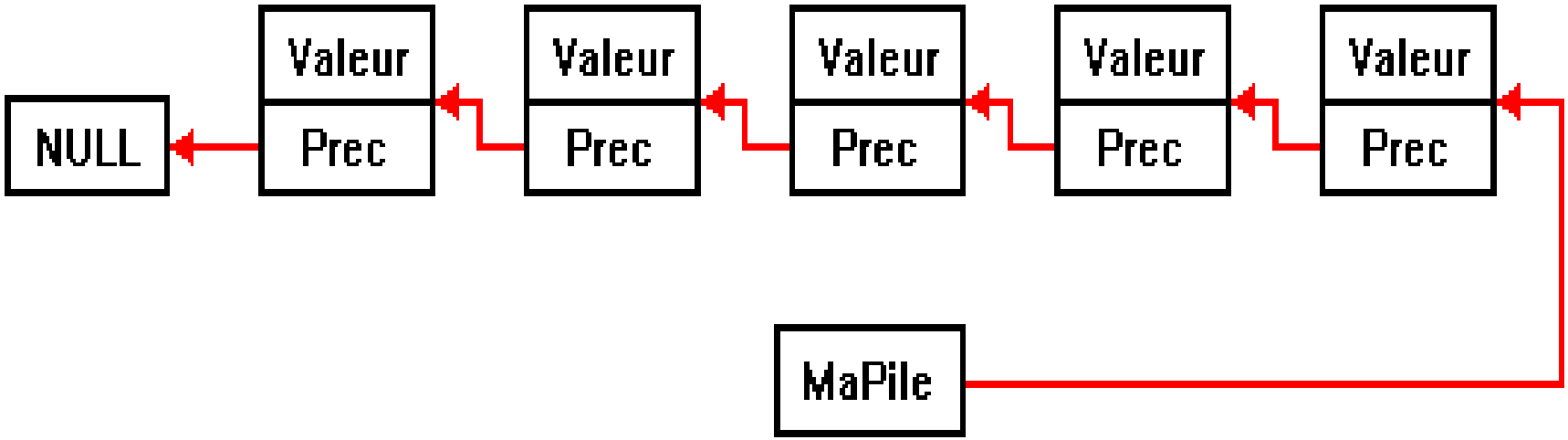
Structure de donnée :

- Pile

Algorithmes

- LIFO (Last In First Out)

Liste chaînée spécialisée



Liste chaînée spécialisée

Pile, ou Tas (Stack): structure LIFO

```
void push(Data_t*)
```

```
Data_t* pop(void)
```

Pile, un TDA

Définition : Une pile est un ensemble dynamique tel que la suppression concerne toujours le dernier élément inséré. Une telle structure est aussi appelé LIFO (last-in, first out).

Opérations :
p.empiler(x) insère un élément à l'entrée de la pile;
p.dépiler() retourne et supprime l'élément en entrée de pile;

Applications

La pile d'exécution : les appels des méthodes dans l'exécution d'un programme sont gérés par une pile.

Éditeur de texte : une pile est fournie par les éditeurs de texte évolués qui possèdent le couple d'actions « annuler-répéter ».

Fonctions utiles …

- Définir le TDA
- Ajouter élément (push)
- Retirer élément (pop)

Définition de la pile

- TDA

```
typedef struct pile {
    int valeur;
    struct pile *prec;
} pile ;
```

- Création de la pile

```
pile *MaPile = NULL;
```

Insertion : push

```
void push(pile **p, int Val) {
        pile *element = malloc(sizeof(pile));
        if(!element) exit(EXIT_FAILURE); /* Si l'allocation a échouée. */
        element→valeur = Val;
        element→prec = *p;
        *p = element; /* Le pointeur pointe sur le dernier élément. */
}
```

Retrait : pop

```
int Pop(pile **p) {
        int Val;
        pile *tmp;
        if(!*p) return -1; /* Retourne -1 si la pile est vide. */
        tmp = (*p)→prec;
        Val = (*p)→valeur;
        free(*p); *p = tmp; /* Le pointeur pointe sur le dernier élément.*/
        return Val; /* Retourne la valeur soutirée de la pile. */
}
```

Taille de la pile : length

```
int length(pile *p) {
        int n=0;
        while(p) {
                n++;
                p = p->prec;
        }
        return n;
}
```

Vider la pile : clear

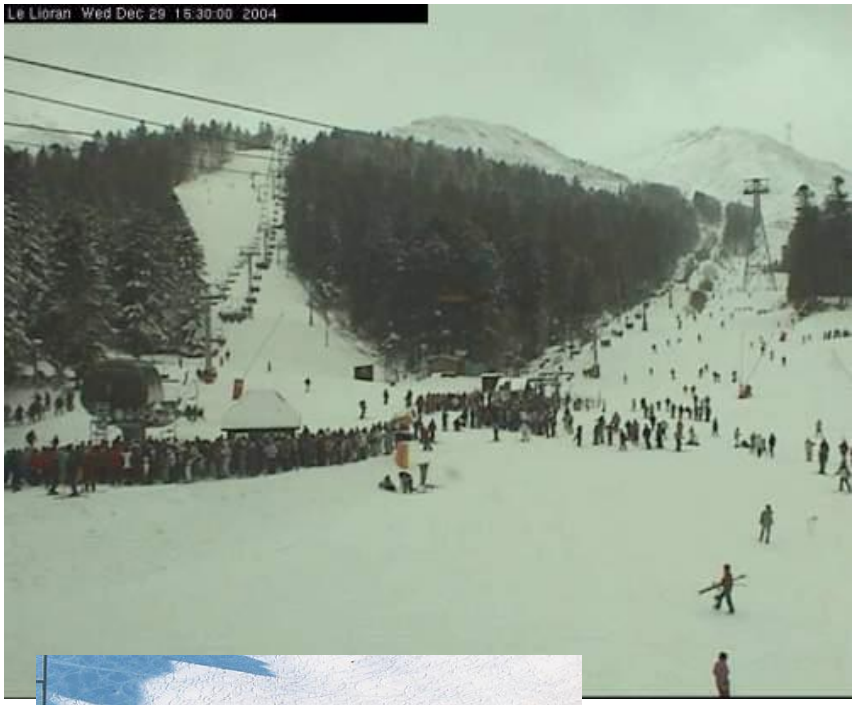
```
void clear(pile **p) {
        pile *tmp;
        while(*p) {
                tmp = (*p)->prec;
                free(*p);
                *p = tmp;
        }
}
```

Afficher la pile : view

```
void view(pile *p) {

        while(p) {

                printf("%d\n",p->valeur);

                p = p→prec;
        }
}
```

QUEUES OU FILES

Motivation



Structure de donnée :

- File

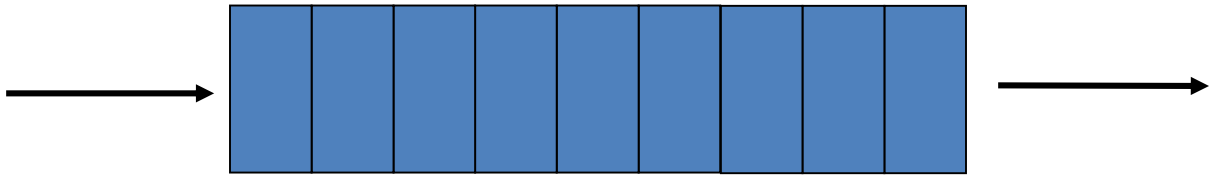
Algorithmes

- FIFO
  (First In First Out)

Aussi: File à priorité

Liste chaînée spécialisée

File, ou queue : structure FIFO



```
void push(Data_t*)                    Data_t* pop(void)
```

File un T.D.A.

Définition : Une file est un ensemble dynamique tel que les insertions se font d'un coté (l'entrée de file) et les suppressions de l'autre coté (la sortie de file). Une telle structure est aussi appelé FIFO (first-in, first out).

Opérations :
f.enfiler(x) ajoute un élément en entrée de file;
f.défiler() supprime l'élément situé en sortie de file.

Applications

Les files d'attentes pour les systèmes de réservations, d'inscriptions, d'accès à des ressources…

Fonctions utiles …

- Destruction
  - Prototype : *int remove(char *nom)*
  - Retourne 0 si réussi
- Renommage:
  - Prototype : *int rename (char* anciennom, char *nouveaunom)*
  - Retourne 0 si réussi

Erreurs à gérer

- **fopen**
  - retourne le pointeur NULL si erreur
  - Exemple : impossibilité d'ouvrir le fichier
- **fgets**
  - retourne le pointeur NULL en cas d'erreur ou si la fin du fichier est atteinte.
- **feof** (FILE *fichier)
  - retourne 0 tant que la fin du fichier n'est pas atteinte.

APPLICATIONS AUX MATRICES CREUSES (LISTES ORTHOGONALES)

Fonctions utiles …

- Destruction
  - Prototype : *int remove(char *nom)*
  - Retourne 0 si réussi
- Renommage:
  - Prototype : *int rename (char* anciennom, char *nouveaunom)*
  - Retourne 0 si réussi

Erreurs à gérer

- **fopen**
  - retourne le pointeur NULL si erreur
  - Exemple : impossibilité d'ouvrir le fichier
- **fgets**
  - retourne le pointeur NULL en cas d'erreur ou si la fin du fichier est atteinte.
- **feof** (FILE *fichier)
  - retourne 0 tant que la fin du fichier n'est pas atteinte.

Structures de données linéaires

Tableaux	**Taille fixe** **Accès direct**
Listes chaînées	**Taille variable** **Accès séquentiel**

Overview

1. CHAPITRE 0 : PRÉSENTATION DU COURS
2. CHAPITRE 1 : LES STRUCTURES LINÉAIRES
3. CHAPITRE 2 : ARBRES ET ARBORESCENCES
4. CHAPITRE 3 : QUELQUES ARBRES PARTICULIERS
5. CHAPITRE 4 : RECHERCHES RAPIDES

CHAPITRE 2 : ARBRES ET ARBORESCENCES

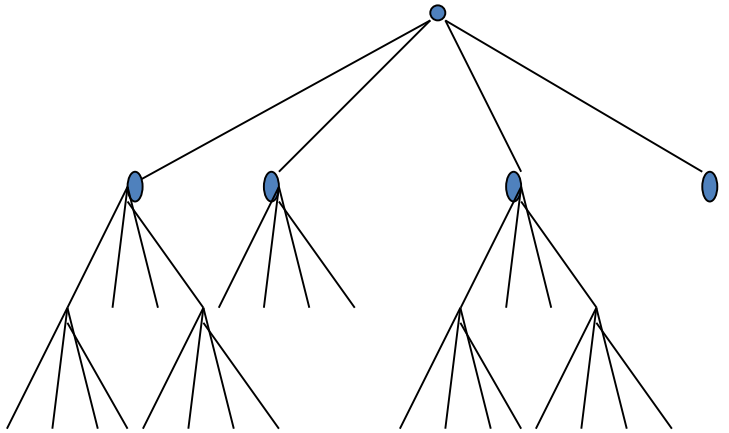
1) Les arbres
2) Arborescences
3) Les arbres binaires
4) Parcours des arbres
5) Applications
    a) Arbres lexicographiques
    b) Sauvegarde et restauration d'arbres

LES ARBRES

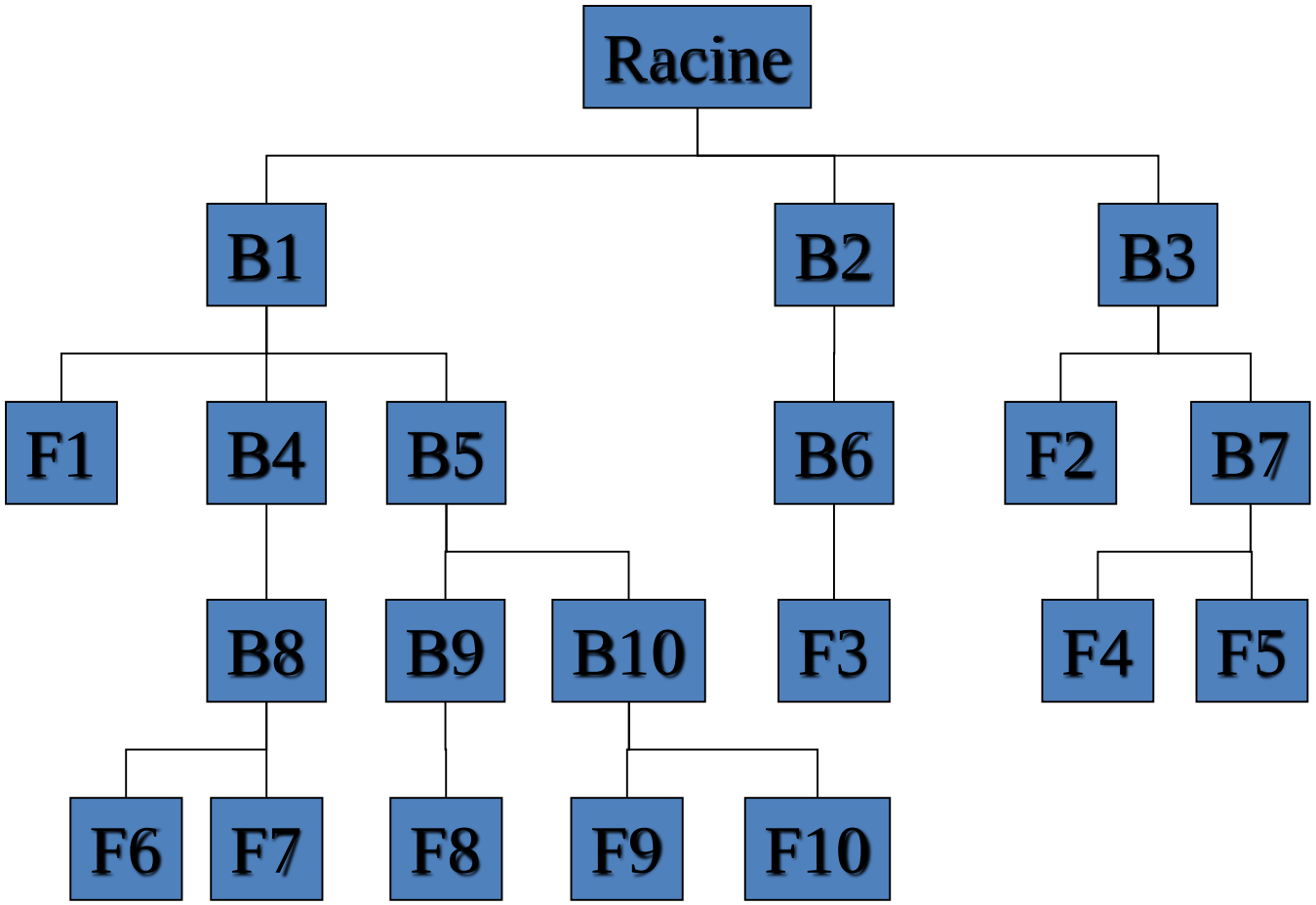
Motivation

- Structure de donnée :
- Arbre

(pour l'élimination des parties cachées)



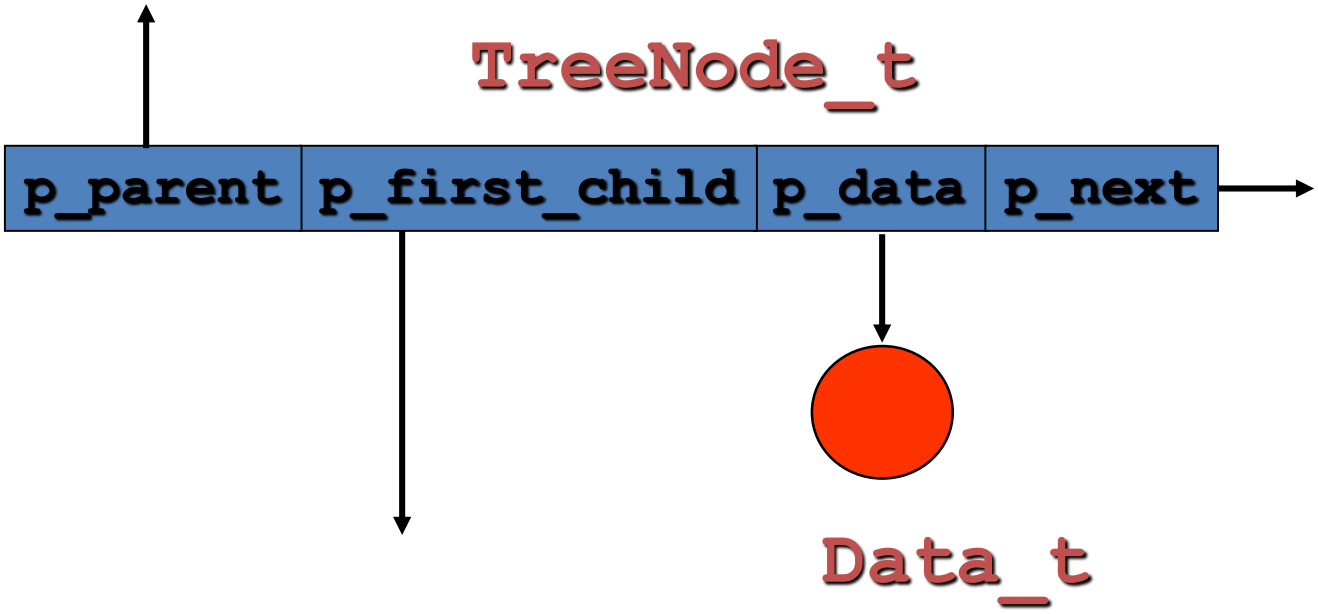
Structures de données hiérarchiques



Arbres: Spécifications

- Créer un arbre
- Parcours pre-order
- Parcours post-order
- Parcours in-order
- Ajout / retrait d'un noeud
- Détruire un arbre

Arbres: Structure de données



Tree.h

```
typedef struct TreeNode_t {
    struct TreeNode_t* p_parent_;
    struct TreeNode_t* p_first_child_;
    Data_t* p_data_;
    struct TreeNode_t* p_next_;
} TreeNode_t;



TreeNode_t* tree_add_node(
    TreeNode_t* p_parent,
    Data_t* p_data
);
```

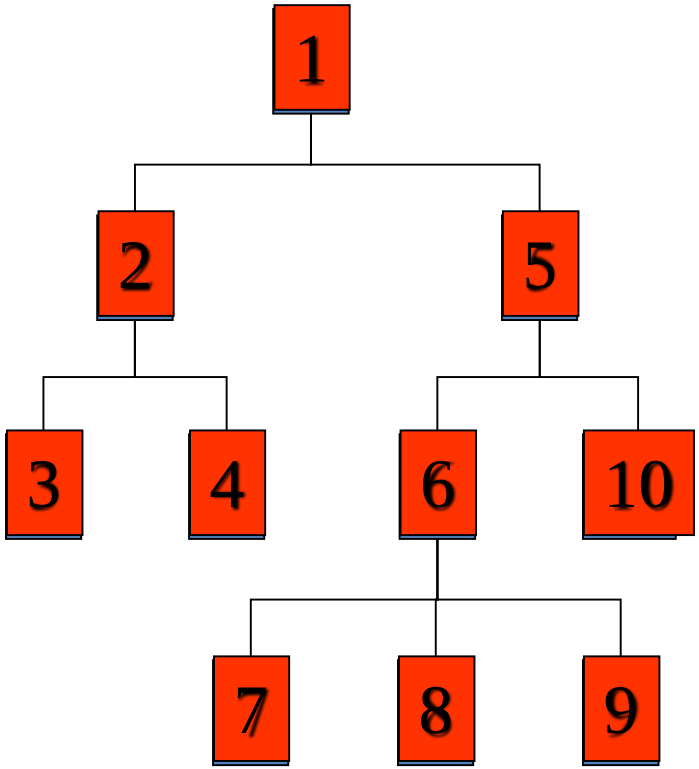
Tree.h

```
TreeNode_t* tree_find_root(
    TreeNode_t* p_parent,
    Data_t* p_data
);


void tree_preorder(
    TreeNode_t* p_root,
    void(* do_it)( Data_t* )
);



void tree_postorder(
    TreeNode_t* p_root,
    void(* do_it)( Data_t* )
);
```

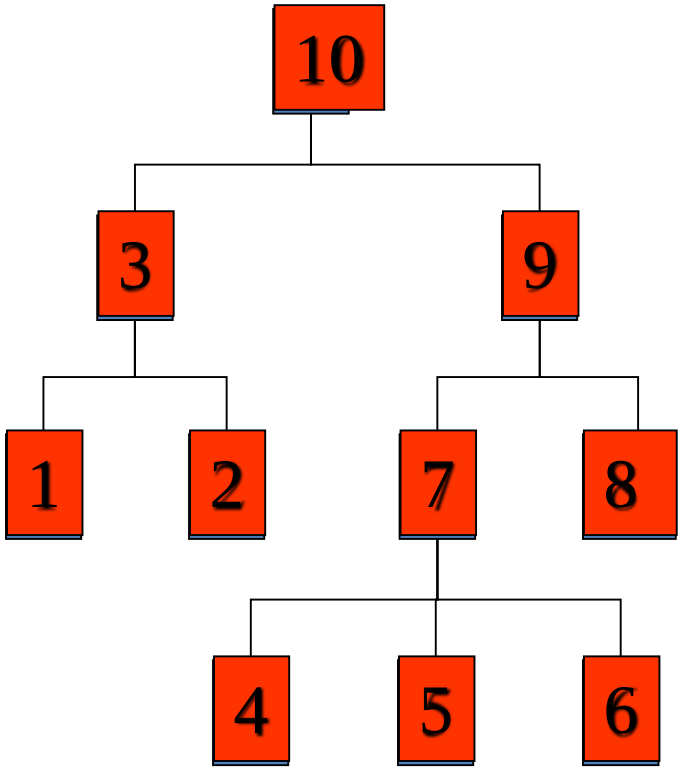
Tree.h

```
void tree_inorder(
    TreeNode_t* p_root,
    void(* do_it)( Data_t* )
);



TreeNode_t* tree_delete_branch(
    TreeNode_t* branch
);
```

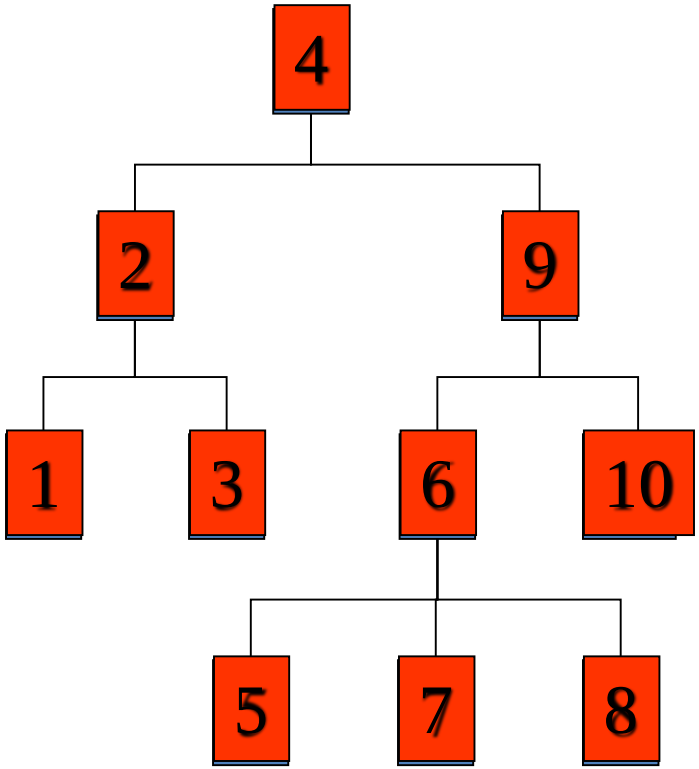
Parcours pre-order



Parcours post-order



Parcours in-order



Fonctions utiles …

- Destruction
  - Prototype : *int remove(char *nom)*
  - Retourne 0 si réussi
- Renommage:
  - Prototype : *int rename (char* anciennom, char *nouveaunom)*
  - Retourne 0 si réussi

Erreurs à gérer

- **fopen**
  - retourne le pointeur NULL si erreur
  - Exemple : impossibilité d'ouvrir le fichier
- **fgets**
  - retourne le pointeur NULL en cas d'erreur ou si la fin du fichier est atteinte.
- **feof** (FILE *fichier)
  - retourne 0 tant que la fin du fichier n'est pas atteinte.

ARBORESCENCES

Fonctions utiles …

- Destruction
  - Prototype : *int remove(char *nom)*
  - Retourne 0 si réussi
- Renommage:
  - Prototype : *int rename (char* anciennom, char *nouveaunom)*
  - Retourne 0 si réussi

Erreurs à gérer

- **fopen**
  - retourne le pointeur NULL si erreur
  - Exemple : impossibilité d'ouvrir le fichier
- **fgets**
  - retourne le pointeur NULL en cas d'erreur ou si la fin du fichier est atteinte.
- **feof** (FILE *fichier)
  - retourne 0 tant que la fin du fichier n'est pas atteinte.

LES ARBRES BINAIRES

Fonctions utiles …

- Destruction
  - Prototype : *int remove(char *nom)*
  - Retourne 0 si réussi
- Renommage:
  - Prototype : *int rename (char* anciennom, char *nouveaunom)*
  - Retourne 0 si réussi

Erreurs à gérer

- **fopen**
  - retourne le pointeur NULL si erreur
  - Exemple : impossibilité d'ouvrir le fichier
- **fgets**
  - retourne le pointeur NULL en cas d'erreur ou si la fin du fichier est atteinte.
- **feof** (FILE *fichier)
  - retourne 0 tant que la fin du fichier n'est pas atteinte.

PARCOURS DES ARBRES

Fonctions utiles …

- Destruction
  - Prototype : *int remove(char *nom)*
  - Retourne 0 si réussi
- Renommage:
  - Prototype : *int rename (char* anciennom, char *nouveaunom)*
  - Retourne 0 si réussi

Erreurs à gérer

- **fopen**
  - retourne le pointeur NULL si erreur
  - Exemple : impossibilité d'ouvrir le fichier
- **fgets**
  - retourne le pointeur NULL en cas d'erreur ou si la fin du fichier est atteinte.
- **feof** (FILE *fichier)
  - retourne 0 tant que la fin du fichier n'est pas atteinte.

APPLICATIONS

Arbres lexicographiques

- Destruction
  - Prototype : *int remove(char *nom)*
  - Retourne 0 si réussi
- Renommage:
  - Prototype : *int rename (char* anciennom, char *nouveaunom)*
  - Retourne 0 si réussi

Sauvegarde et restauration d'arbres

- **fopen**
  - retourne le pointeur NULL si erreur
  - Exemple : impossibilité d'ouvrir le fichier
- **fgets**
  - retourne le pointeur NULL en cas d'erreur ou si la fin du fichier est atteinte.
- **feof** (FILE *fichier)
  - retourne 0 tant que la fin du fichier n'est pas atteinte.

STRUCTURES COMPLEXES : GRAPHES

Structures de données complexes: Les Graphes