HESTIM
ENGINEERING & BUSINESS SCHOOL

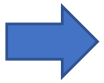
Les listes chaînées

Filière : Ingénierie Informatique & Intelligence Artificielle

Par : Youssef Ouassit

Introduction

- Les tableaux:
  - Stocker un ensemble de données de même type d'une façon linéaire
  - Allocation d'un seul bloc en mémoire avec des cases contiguës
  - Taille connue et fixe des éléments



Accès au élément en **O(1)**

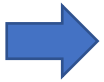
Rechercher un élément en **O(n)**

Ajouter un élément en **O(n)**

Supprimer un élément en **O(n)**

Introduction

- Les listes
  - Taille inconnue au départ, elle varie au cours du temps
  - Allocation indépendante de chaque élément, sous la forme d'un maillon (cellule).
  - Les éléments sont habituellement éparpillés en mémoire.
  - Peuvent contenir des types de données différents



Accès au élément en **O(n)**

Rechercher un élément en **O(n)**

Ajouter un élément en **O(1)**

Supprimer un élément en **O(1)**

Introduction

Plusieurs type de listes :

- **Liste chaînée simple** constituée d'éléments reliés entre eux par des pointeurs.
- **Liste doublement chaînée** où chaque élément dispose non plus d'un mais de deux pointeurs pointant respectivement sur l'élément précédent et l'élément suivant.
- **Liste circulaire** où le dernier élément pointe sur le premier élément de la liste. S'il s'agit d'une liste doublement chaînée alors de premier élément pointe également sur le dernier.

Introduction

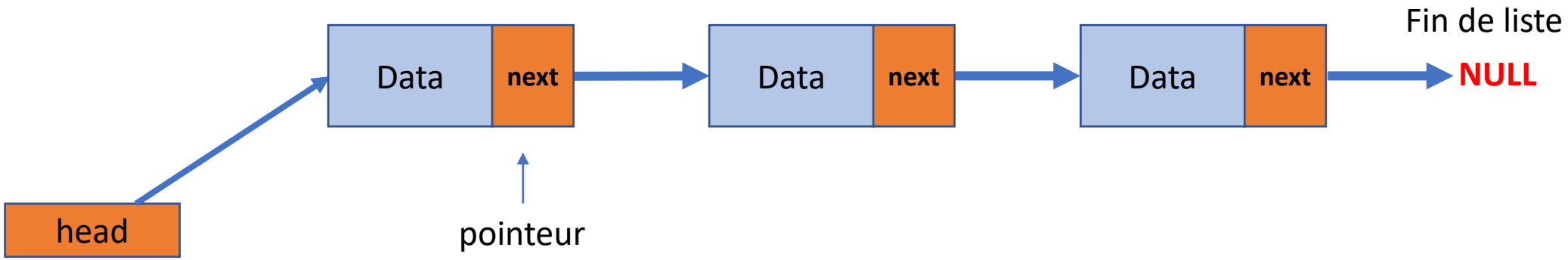
- La façon dont on met en œuvre ces structures dépend des langages, dans notre cours nous allons adopter la représentation en langage C.

- Une méthode simple pour représenter une liste, consiste à dire qu'une liste L soit vide, soit elle est constituée:
  - ✓ d'une tête T (qui est donc la valeur du premier élément de la liste) et
  - ✓ d'une queue Q (qui est le reste de la liste).

LES LISTES SIMPLEMENT CHAÎNEES

Principe :

- Contrairement aux tableaux, les éléments d'une liste chaînée ne sont pas placés côte à côte dans la mémoire.
- Chaque élément peut contenir ce que l'on veut: un ou plusieurs int, double, structures,...
- Chaque élément possède un pointeur vers l'élément suivant.
- Un pointeur "head" qui pointe sur le premier élément
- Le dernier élément point sur NULL



Primitives :

- ❑ Créer une liste
- ❑ Ajouter un élément (Au début, au milieu , à la fin)
- ❑ Parcourir une liste
- ❑ Rechercher un élément
- ❑ Modifier un élément
- ❑ Supprimer un élément

1 - Implémenter un Noeud :

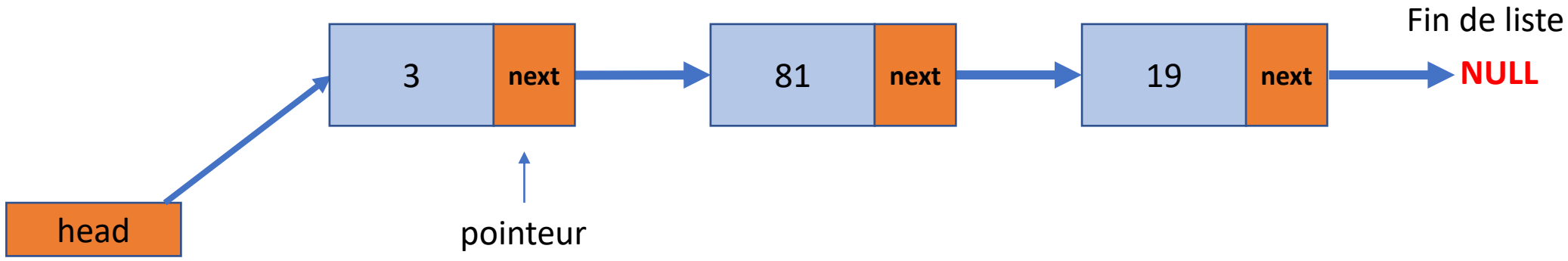
L'implémentation d'une liste chaînées commence par implémenter une cellule (un nœud) :

Node

```
typedef struct node {
        // données
        struct node *next;
} Node;
```

1 - Implémenter un Noeud :

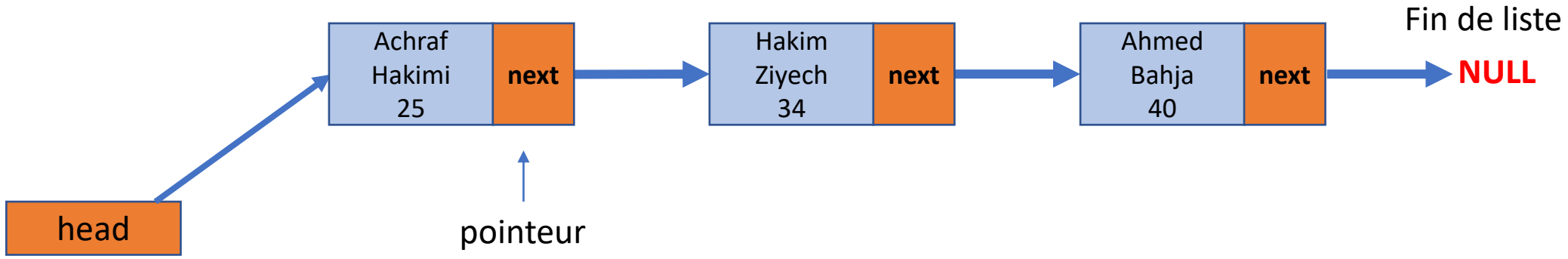
L'implémentation d'une liste chaînées commence par implémenter une cellule (un nœud) :




```c
typedef struct node {
    int val;
    struct node *suiv;
} Node;
```

1 - Implémenter un Noeud :

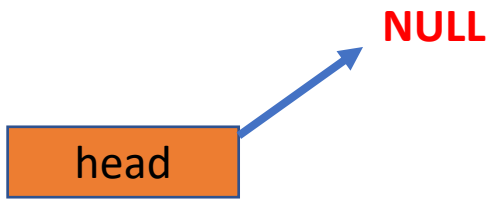
L'implémentation d'une liste chaînées commence par implémenter une cellule (un nœud) :




```c
typedef struct node {
    char nom[20];
    char prenom[20];
    int age;
    struct node *next;
} Node;
```

2 – Créer une liste vide :

Une liste vide est une liste qui ne contient aucun élément, donc le pointeur **tete** pointe sur NULL.



Pour déclarer maintenant une liste vide L :

```
Node *head = NULL;
```

3 - Implémenter les primitives : Allouer un Node

Une fonction qui alloue un nouveau nœud :

```
Node* creerNoeud(int valeur) {
    Node* nouveau = (Node*) malloc(sizeof(Node));
    if (nouveau == NULL) {
        printf("Erreur d'allocation mémoire\n");
        exit(1);
    }
    nouveau->val = valeur;
    nouveau->next = NULL;
    return nouveau;
}
```

Exemple :

```
Node *n = creerNoeud(40)
```

3 - Implémenter les primitives : Insérer au début

Insérer un nœud au début consiste à :

1. Allouer un nouveau nœud.
2. Le suivant de ce nouveau nœud est l'ancien nouveau nœud.
3. La tête de la liste est maintenant ce nouveau nœud.

```
Node* insererDebut(Node* head, int valeur) {
    Node* nouveau = creerNoeud(valeur);
    nouveau->next = head;
    return nouveau; // nouvelle tête
}
```

Exemple :

```
Node *head = NULL;
// autres instructions
head = insererDebut(head, 40);
head = insererDebut(head, 30);
head = insererDebut(head, 120);
```

3 - Implémenter les primitives : Insérer à la fin

Insérer un nœud à la fin consiste à :

1. Allouer un nouveau nœud.
2. Si la liste est vide la nouvelle liste est notre nouveau nœud.
3. Si non, parcourir la liste jusqu'au dernier Node.
4. Le suivant du dernier Node est le nouveau élément.

```
Node* insererFin(Node* head, int valeur) {
    Node* nouveau = creerNoeud(valeur);
    if (head == NULL) return nouveau;

    Node* temp = head;
    while (temp->next != NULL)
        temp = temp->next;
    temp->next = nouveau;
    return head;
}
```

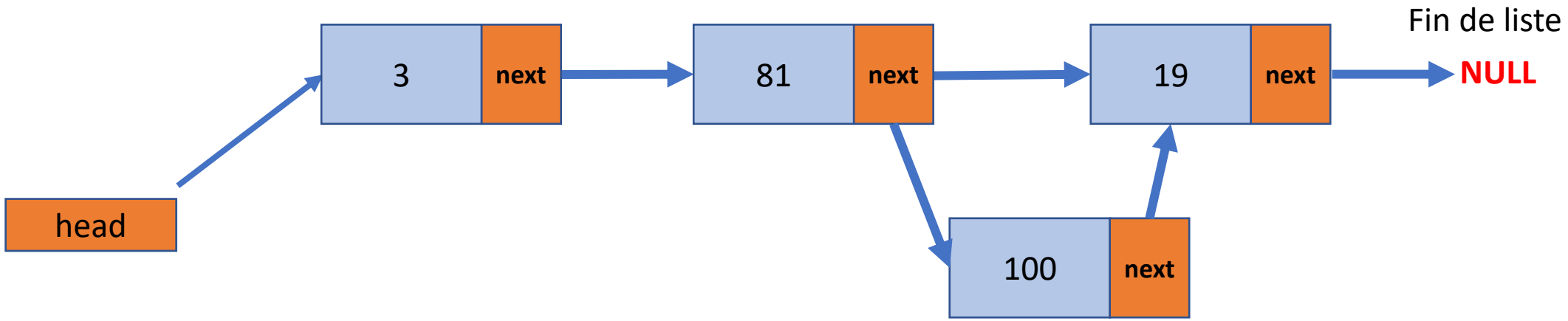
Exemple :

```
Node *head = NULL;
// autres instructions
head = insererFin(head, 40);
head = insererFin(head, 30);
head = insererFin(head, 120);
```

3 - Implémenter les primitives : Insérer après une valeur

Insérer un nœud à la fin consiste à :

1. Allouer un nouveau nœud.
2. Si la liste est vide la nouvelle liste est notre nouveau nœud.
3. Si non, parcourir la liste jusqu'au dernier Node.
4. Le suivant du dernier Node est le nouveau élément.



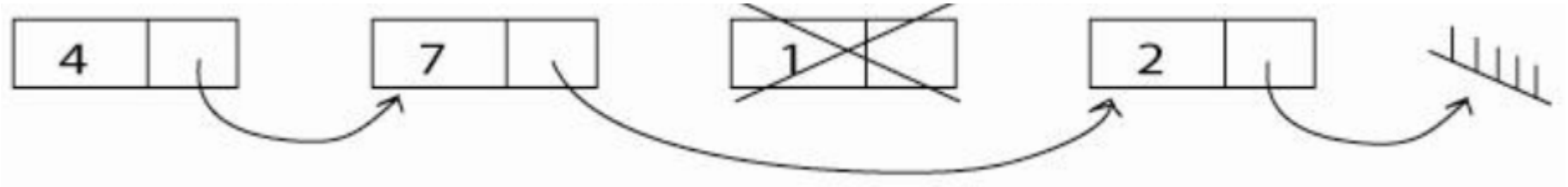
3 - Implémenter les primitives : Insérer après une valeur

```
Node* insererApres(Node* head, int valeur, int cle) {
    Node* temp = head;
    while (temp != NULL && temp->val != cle)
        temp = temp->next;

    if (temp == NULL) {
        printf("Valeur %d non trouvée.\n", cle);
        return head;
    }

    Node* nouveau = creerNoeud(valeur);
    nouveau->next = temp->next;
    temp->next = nouveau;
    return head;
}
```

3 - Implémenter les primitives : Supprimer un élément

Pour supprimer un élément :

1. Si la valeur à supprimer est la première valeur, faite pointer head sur l'élément suivant de l'élément à supprimer.
2. Si non : Chercher l'élément précèdent de l'élément à supprimer.
3. Faire pointer le suivant de cette élément au suivant de l'élément à supprimer.
4. Libérer la mémoire de l'élément supprimé.



3 - Implémenter les primitives : Supprimer un élément

```
Node* supprimerValeur(Node* head, int cle) {
    if (head == NULL) return NULL;

    if (head->val == cle) {
        Node* temp = head;
        head = head->next;
        free(temp);
        return head;
    }

    Node* temp = head;
    while (temp->next != NULL && temp->next->val != cle)
        temp = temp->next;

    if (temp->next == NULL) {
        printf("Valeur %d non trouvée.\n", cle);
        return head;
    }

    Node* aSupprimer = temp->next;
    temp->next = aSupprimer->next;
    free(aSupprimer);
    return head;
}
```

Si l'élément à supprimer est le premier élément de la liste

3 - Implémenter les primitives : Afficher une liste

Pour afficher une liste :

1. Déclarer un pointeur **tmp** qui pointe sur le premier élément.
2. Tant que **tmp** n'est pas NULL afficher sa valeur et passer à l'élément suivant.

```
void afficherListe(Node* head) {
    Node* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->val);
        temp = temp->next;
    }
    printf("NULL\n");
}
```

3 - Implémenter les primitives : Longueur d'une liste

Pour afficher une liste :

1. Déclarer un pointeur **tmp** qui pointe sur le premier élément.
2. Tant que **tmp** n'est pas NULL incrémenter le compteur et passer à l'élément suivant.

```
int longueurListe(Node* tete) {
    int count = 0;
    Node* temp = tete;
    while (temp != NULL) {
        count++;
        temp = temp->next;
    }
    return count;
}
```

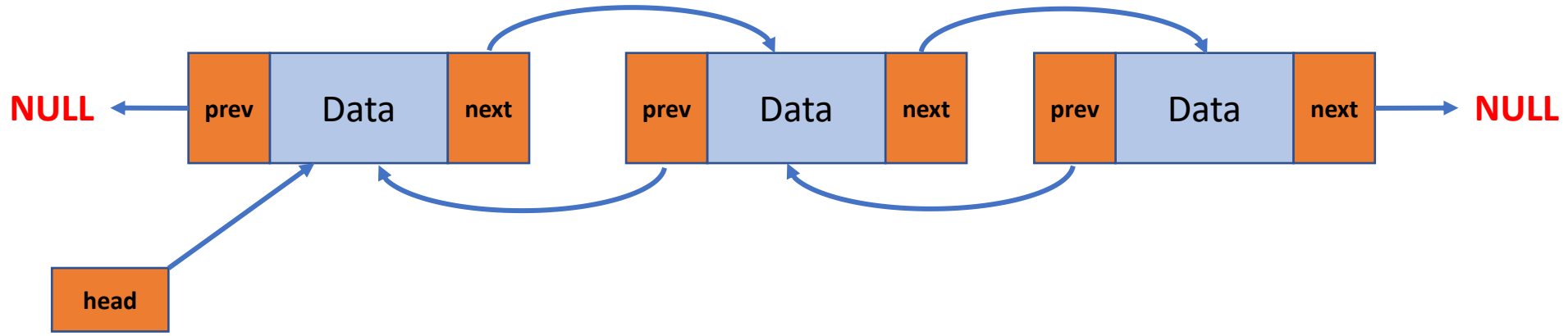
Un exemple complet :

```c
int main() {
    Node* liste = NULL;

    liste = insererDebut(liste, 3);
    liste = insererDebut(liste, 2);
    liste = insererDebut(liste, 1);
    liste = insererFin(liste, 4);

    afficherListe(liste);

    liste = supprimerValeur(liste, 2);
    afficherListe(liste);

    printf("Longueur : %d\n", longueurListe(liste));

    return 0;
}
```

LES LISTES DOUBLEMENT CHAÎNEES

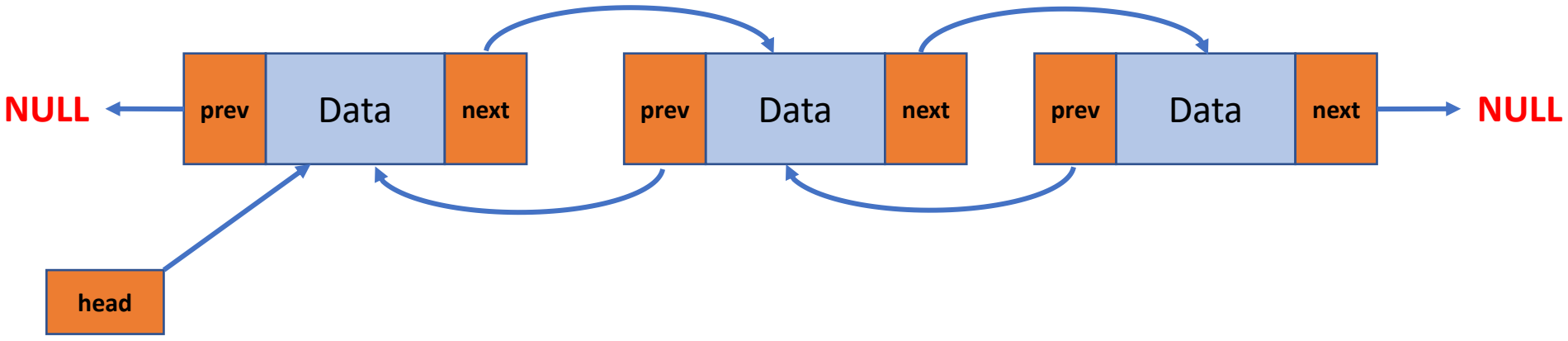
Listes doublement chainées

- Encore appelée liste symétrique, une liste doublement chaînée est une liste dans laquelle chaque élément de la liste a deux pointeurs, l'un pointant sur son suivant et l'autre sur son précédent. La tête d'une liste doublement chaînée à son précédent qui pointe à Null, tandis que la queue à son suivant qui pointe à Null.



Listes doublement chainées

- Elles ont l'avantage d'être réellement dynamiques, c'est dire que l'on peut à loisir les rallonger ou les raccourcir, avec pour seule limite la mémoire disponible.
- De plus, l'insertion d'un composant au milieu d'une liste ne nécessite que la modification des liens avec l'élément précédent et le suivant. Le temps nécessaire pour l'opération sera donc indépendant de la longueur de la liste.



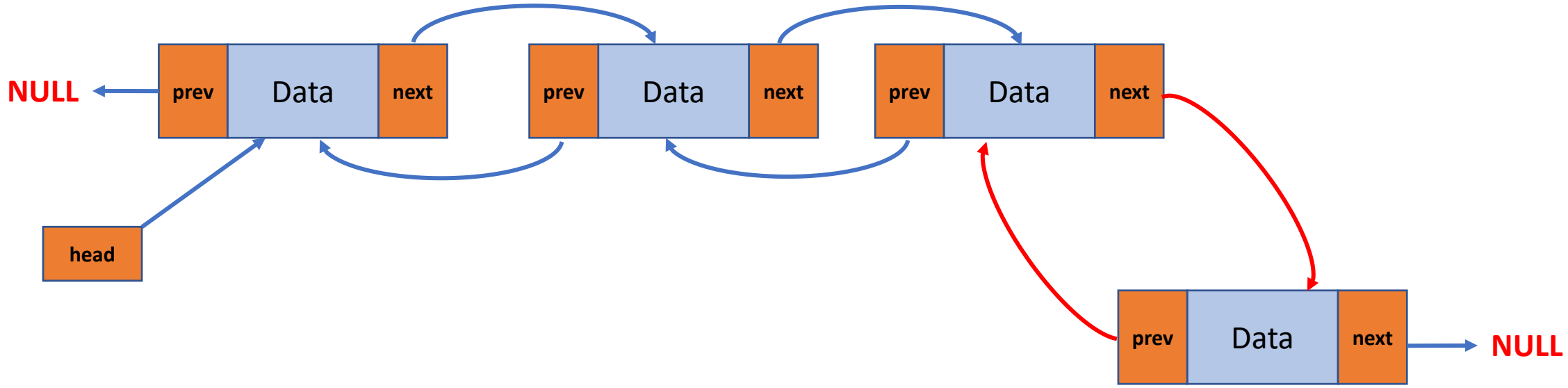
1 - Implémenter un Noeud :

L'implémentation d'une liste chaînées commence par implémenter une cellule (un nœud) :

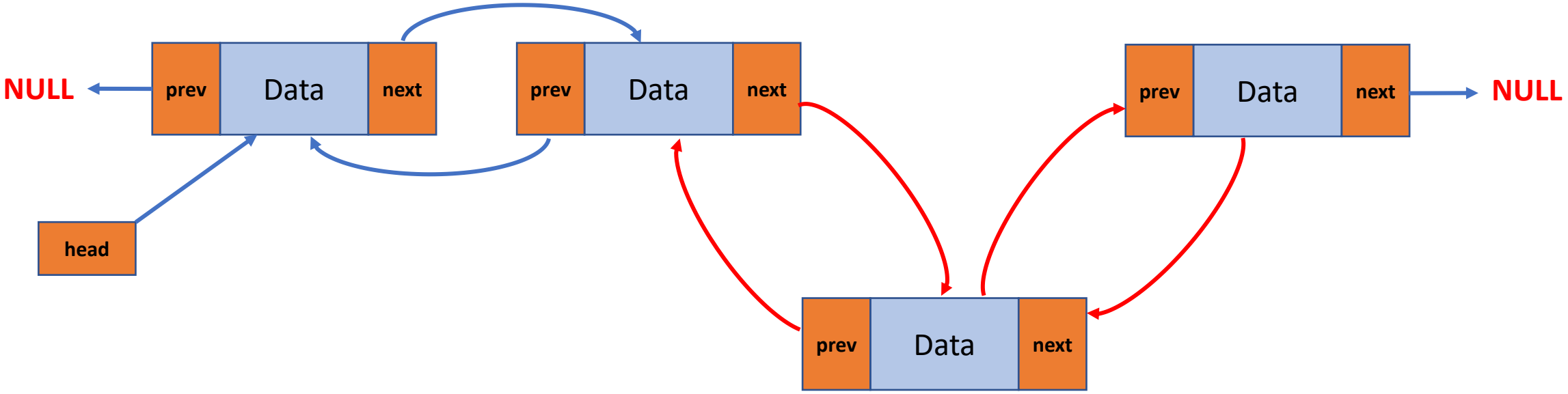
Node

```
typedef struct node {
        struct node *prev;
        // données
        struct node *next;
} Node;
```

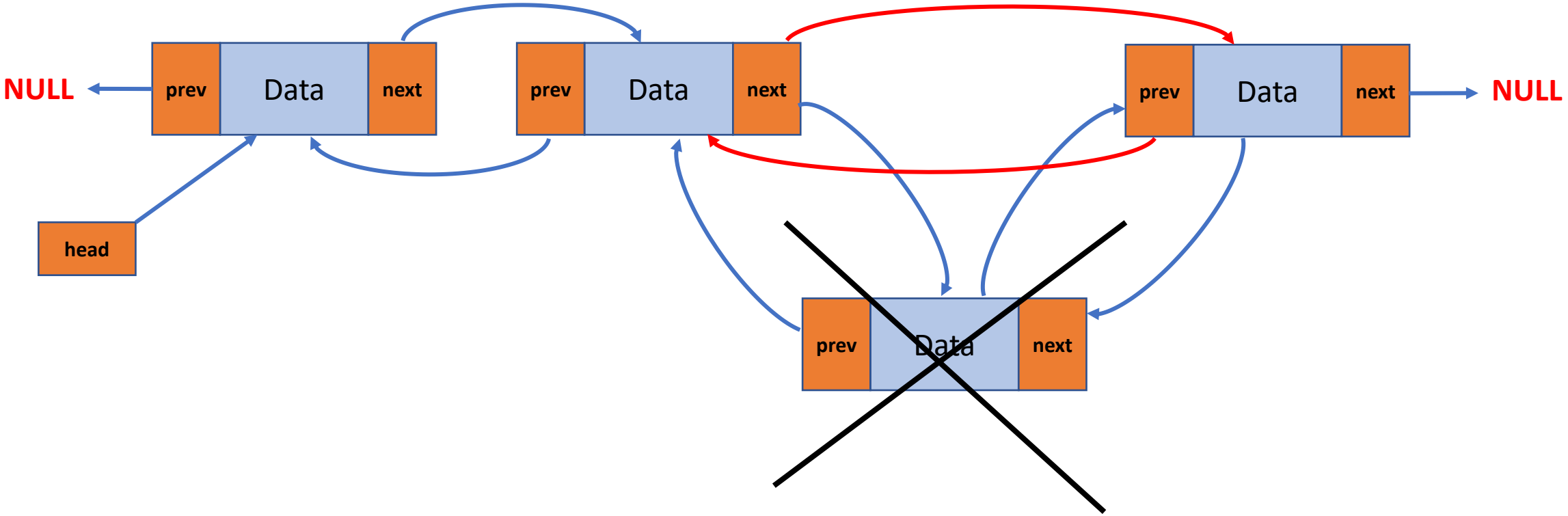
2 – Primitives : Ajouter un élément à la fin



2 – Primitives : Insérer un élément



2 – Primitives : Supprimer un élément

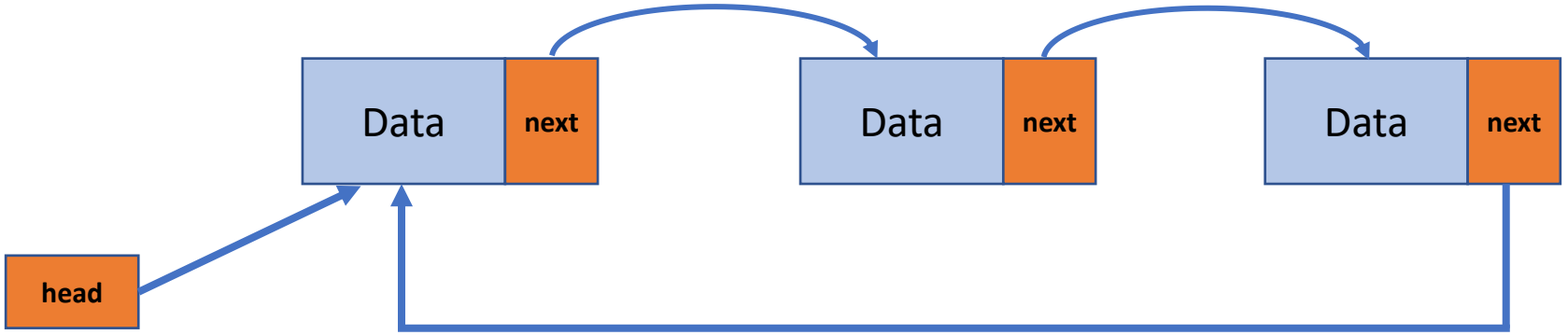


LES LISTES CHAÎNEES CIRCULAIRES SIMPLES

Listes chainées circulaires simple

C’est une liste simplement chaînée où le dernier nœud ne pointe pas sur NULL, mais sur le premier nœud, formant ainsi un cercle. Caractéristiques :

- Pas de début ni de fin “absolue”.
- Permet un parcours en boucle.
- Utilisée pour gérer des cycles (ex. tour de jeu, file circulaire...).



LES LISTES DOUBLEMENT CHAÎNEES CIRCULAIRES

Listes doublement chainées circulaires

C’est une liste doublement chaînée où :

- le pointeur next du dernier nœud pointe vers le premier,
- et le pointeur prev du premier nœud pointe vers le dernier.