Course descriptions : semester 2, Data Science and Big Data..

Programmation & Algorithmique II

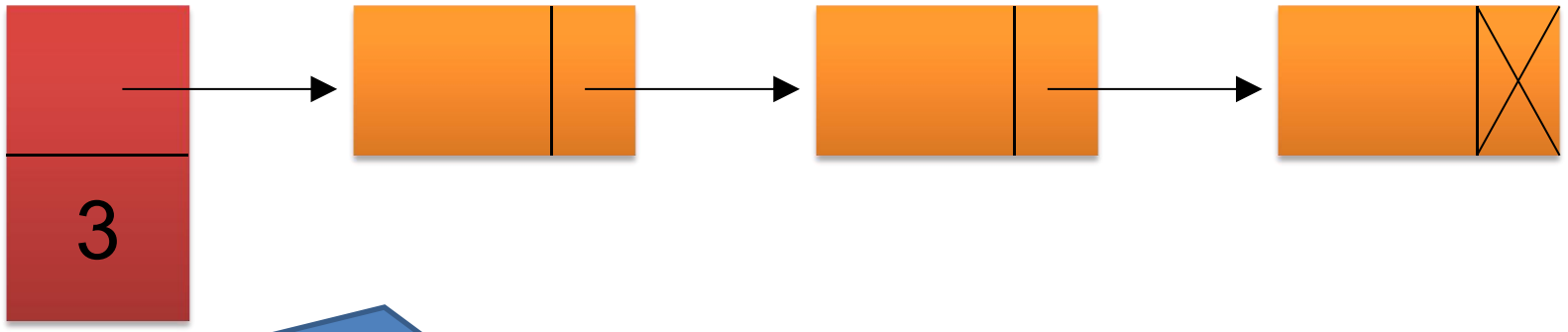
CM 5 : listes chaînées (2)

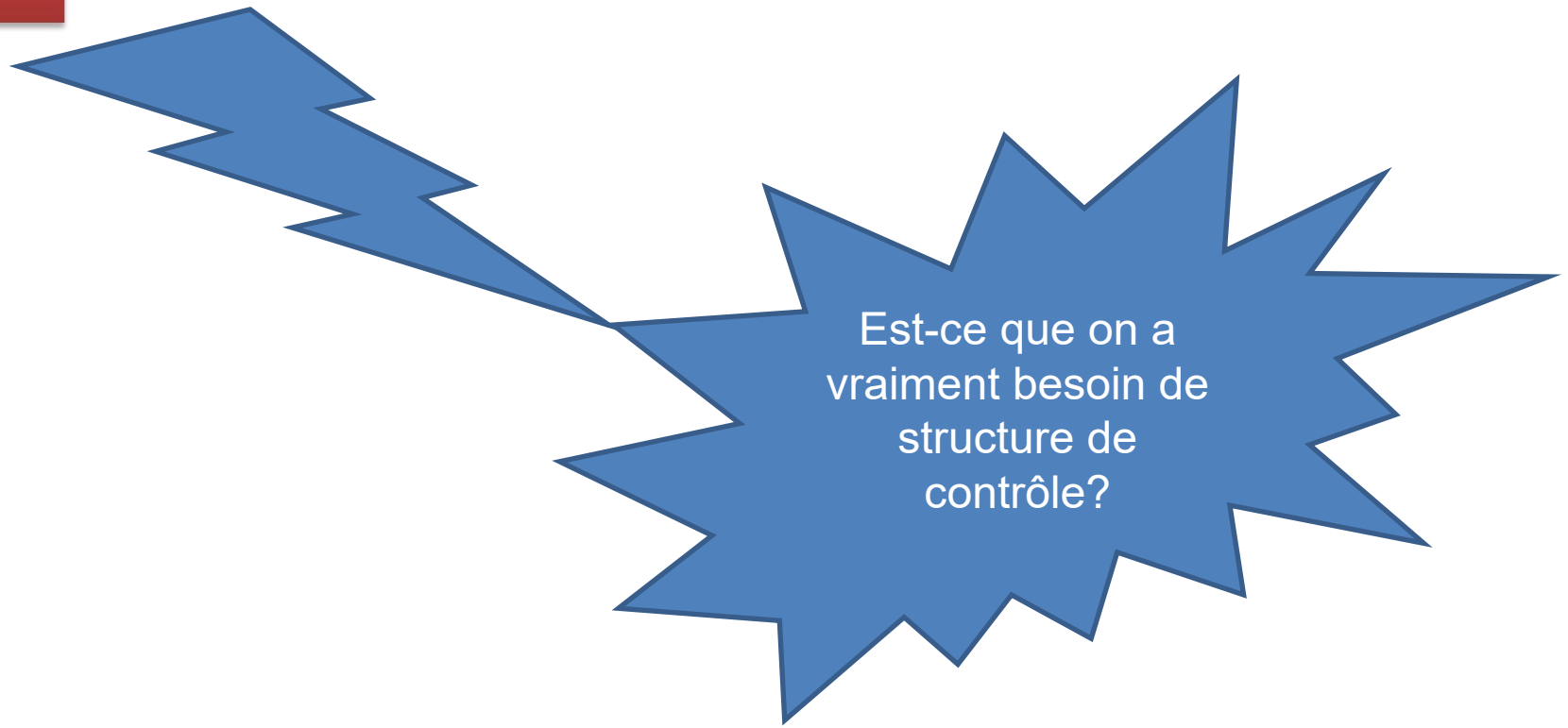
PLAN

- Les Listes simplement chaînées.
  - La représentations minimal avec un seul pointeurs
- Les listes chainées circulaires
  - La représentation graphique
  - Les services des listes chaînées circulaire
    - Création
    - Ajout
    - Suppression
    - Consultation/recherche

LISTES CHAINÉES

- **Représentation simpliste d'une liste simplement chaînée**



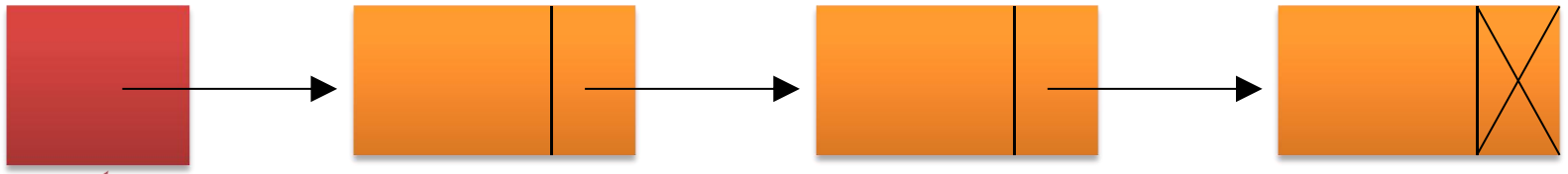


浙江科技學院 ZHEJIANG UNIVERSITY OF SCIENCE & TECHNOLOGY

CERGY PARIS UNIVERSITÉ

LISTES CHAINÉES

- **Représentation minimal d'une liste simplement chaînée**



Pointeur vers le premier élément

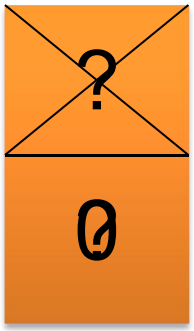
- **Implémentation C d'une liste simplement chaînée en utilisant un pointeur et une structure**

```c
#include <stdio.h>
#include <stdlib.h>
//1) structure Element pour représenter un élément de la liste chaînée
typedef struct Element Element;
struct Element
{
        int nombre; // donnée
        Element *suivant; // pointeur vers l'élément suivant de la liste
};
```

Création de la liste

On fait pointer la tête vers NULL.

On initialise le nombre de nœuds à 0.



- **Implémentation C d'une liste simplement chaînée en utilisant un pointeur et une structure**

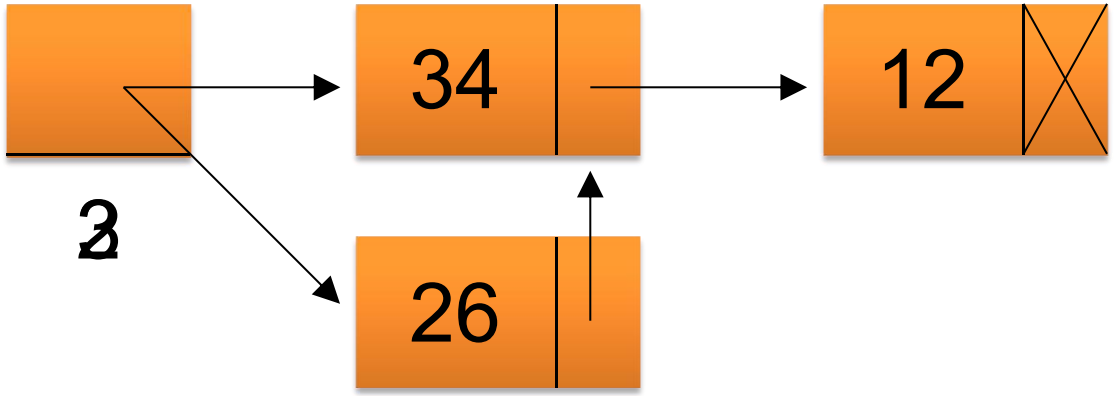
```
Element *initialisation()
{
    Element *liste = NULL;

    return liste;
}
```

- **Implémentation C d'une liste simplement chaînée en utilisant deux structures**

```
Liste *initialisation()
{
    // Liste *liste = (Liste *)malloc(sizeof(Liste));
    Liste *liste = malloc(sizeof(*liste));

    if (liste == NULL)
    {
        exit(EXIT_FAILURE);
    }

    liste->premier = NULL;
    liste->cnt = 0;

    return liste;
}
```

Ajout au début de la liste

On tente de créer un nouveau nœud.
SI la création fut un succès ALORS
    On fait pointer le nouveau nœud vers le premier élément de la liste (NULL si celle-ci est vide).
    On initialise le nœud.
    On fait pointer la tête de liste sur le nouveau nœud.
    On incrémente le nombre de nœuds.
FINSI



- **Implémentation C d'une liste simplement chaînée en utilisant deux structures**

```
Element *inserTete(Element *liste, int nvNombre)
{
    /* Création du nouvel élément */
    Element *nouveau = malloc(sizeof(*nouveau));
    if (nouveau == NULL)
    {
        exit(EXIT_FAILURE);
    }
    /* Insertion de l'élément au début de la liste */
    nouveau->suivant = liste;
    nouveau->nombre = nvNombre;
    liste = nouveau;
    return liste;
}
```

- **Implémentation C d'une liste simplement chaînée en utilisant deux structures**

```
void inserTete(Liste *liste, int nvNombre)
{
    /* Création du nouvel élément */
Element *nouveau = malloc(sizeof(*nouveau));
    if (liste == NULL || nouveau == NULL)
    {
        exit(EXIT_FAILURE);
    }

    /* Insertion de l'élément au début de la liste */
    nouveau->suivant = liste->premier;
    nouveau->nombre = nvNombre;
    liste->premier = nouveau;
    liste->cnt +=1;
}
```

Merci de compléter le reste des implémentations avec un seul pointeur des opérations sur les listes chaînées.

QUIZ

Question : Que ce qu'il fait la fonction en dessus ?
Choix:

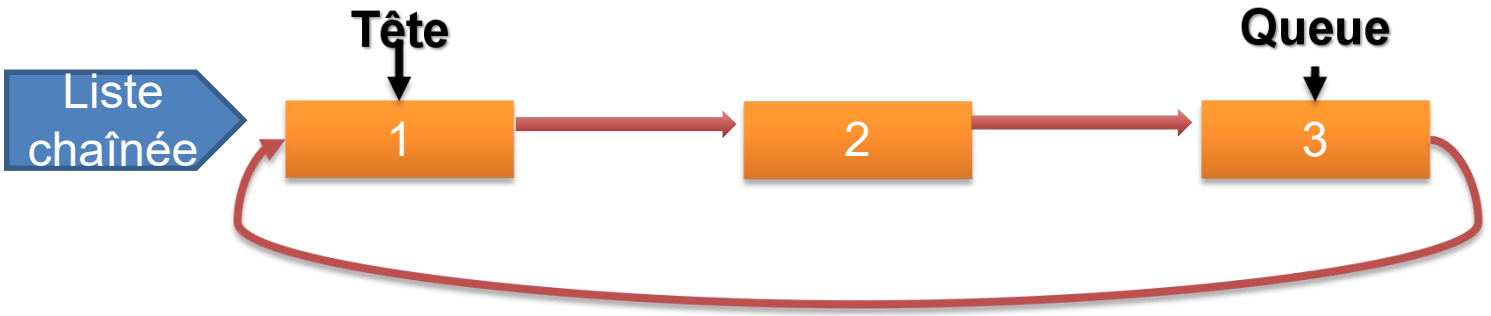
A- Ajoute un élément au début de la liste
B- Retire un élément de début de la liste
C- Retire un élément de la fin de la liste

5 minute

```
void fonction(Liste *liste){
    if (liste->premier == NULL){
        printf("\nLa liste est vide \n");
        exit(0);
    }

    Element *E = liste->premier;
    liste->premier = liste->premier->suivant;
    free(E);
    liste->cnt--;
}
```

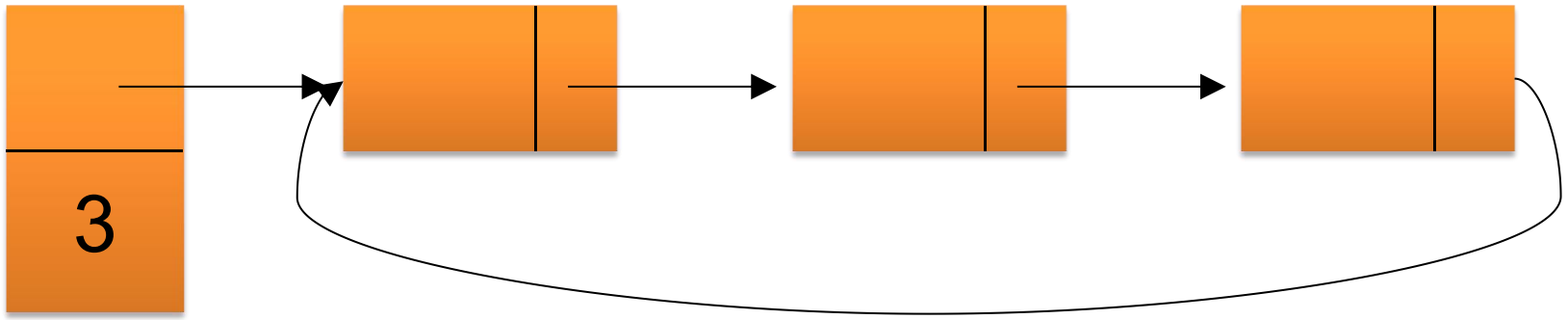
- **Qu'est-ce qu'une liste chainée circulaire ?**
  - Une liste chaînée circulaire est une liste simplement chaînée dont le dernier élément pointe vers le premier élément de la liste.
  - Le seul inconvénient d'une liste chaînée circulaire est la complexité de l'itération
    - **Notez qu'il n'y a pas de valeurs NULL dans la partie suivant de l'un des éléments de la liste**

	donnée	suivant
FF01	5	FF04
FF02		
FF03		
FF04	7	FF07
FF05		
FF06		
FF07	9	FF08
FF08	15	FF10
FF09		
FF10	0	FF01



LISTES CHAINÉES

- **Représentation simpliste d'une liste simplement chaînée**



On reconnaît le dernier élément de liste si sa partie suivant pointe vers le premier élément de la liste

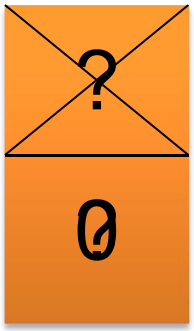
- **Implémentation C d'une liste simplement chaînée en utilisant deux structures**

```c
#include <stdio.h>
#include <stdlib.h>
//1) structure Element pour représenter un élément de la liste chaînée
typedef struct Element Element;
struct Element
{
        int nombre; // donnée
        Element *suivant; // pointeur vers l'élément suivant de la liste
};

//2) La structure de contrôle
typedef struct Liste Liste;
struct Liste
{
        Element *premier; // pointeur vers le premier élément de la liste
        int cnt = 0;   // compteur des éléments de la liste
};
```

Création de la liste

On fait pointer la tête vers NULL.

On initialise le nombre de nœuds à 0.



- **Implémentation C d'une liste simplement chaînée en utilisant deux structures**

```
Liste *initialisation()
{
    // Liste *liste = (Liste *)malloc(sizeof(Liste));
    Liste *liste = malloc(sizeof(*liste));

    if (liste == NULL)
    {
        exit(EXIT_FAILURE);
    }

    liste->premier = NULL;
    liste->cnt = 0;

    return liste;
}
```

Ajout au début de la liste

On tente de créer un nouveau nœud.

SI la création fut un succès ALORS

On fait pointer le nouveau nœud vers le premier élément de la liste (Vers lui même si celle-ci est vide).

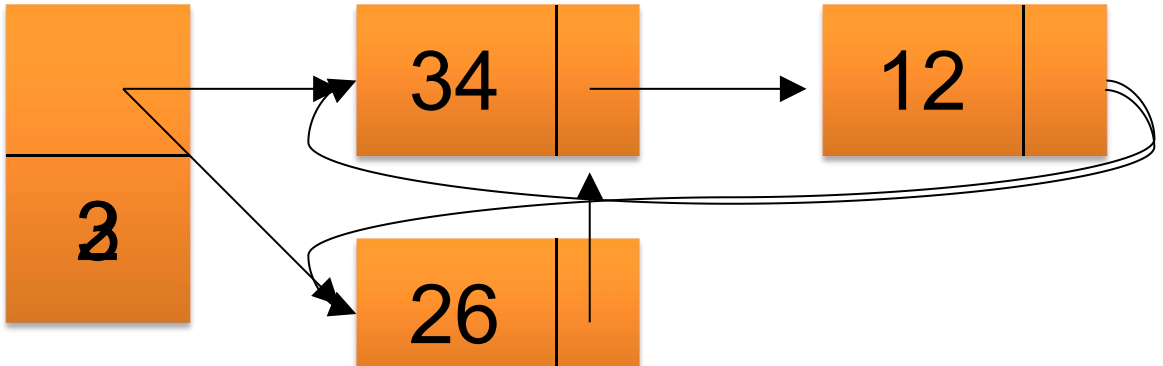
On initialise le nœud.

On fait pointer la tête de liste sur le nouveau nœud.

On incrémente le nombre de nœuds.

Localiser le dernier noeud et le faire pointer vers le nouvel noeud

FINSI



LISTES CHAINÉES

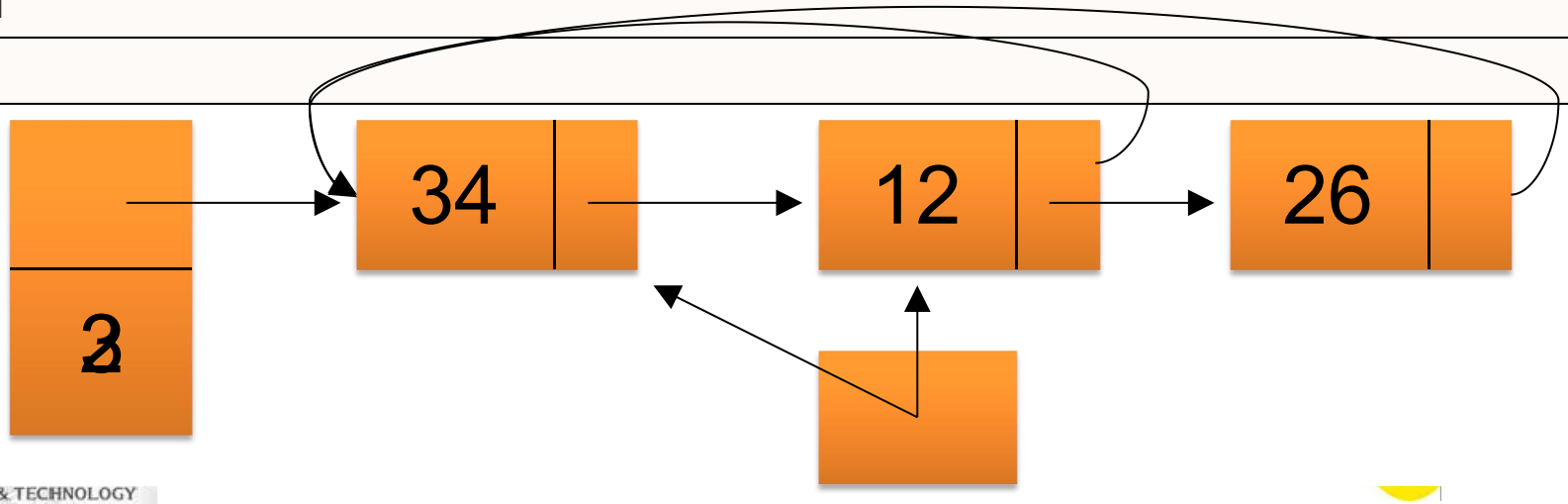
- **Implémentation C d'une liste simplement chaînée en utilisant deux structures**

```
void inserTete(Liste *liste, int nvNombre)
{
   /* Création du nouvel élément */
   Element *nouveau = malloc(sizeof(*nouveau));
   if (nouveau == NULL)
   {
     exit(EXIT_FAILURE);
   }

   if(liste->premier == NULL)
   {
     /* Insertion de l'élément au début de la liste si la liste est vide*/
     nouveau->suivant = nouveau;
     nouveau->nombre = nvNombre;
     liste->premier = nouveau;
     liste->cnt +=1;
   }
   else
   {
     /* Insertion de l'élément au début de la liste si la liste n'est pas vide*/
     nouveau->suivant = liste->premier;
     nouveau->nombre = nvNombre;
     Element *actuel = liste->premier;
     while (actuel->suivant != liste->premier){
                    actuel = actuel->suivant;
     }
     liste->premier = nouveau;
     actuel->suivant = liste->premier;
     liste->cnt +=1;
   }
}
```

Pour faire une insertion au début d'une liste circulaire il faut vérifier
1- si la liste est vide → donc l'adresse suivant de la première élément est le même de lui-même.
2- si la liste n'est pas vide on faire l'insertion come pour une liste simple en changeant l' adresse suivant de dernier élément de la liste à l 'adresse de nouveau élément inséré.

Ajout à la fin de la liste

SI la liste est vide ALORS
    On ajoute au début de la liste
SINON
    On tente de créer un nouveau nœud.
    SI la création fut un succès ALORS
        On positionne un pointeur sur le dernier élément de la liste (voir consulter $i^{ème}$ élément).
        On initialise les champs du nouveau nœud.
        On fait pointer le dernier nœud sur le nouveau nœud.
        On incrémente le nombre de nœuds.
    FINSI
FINSI

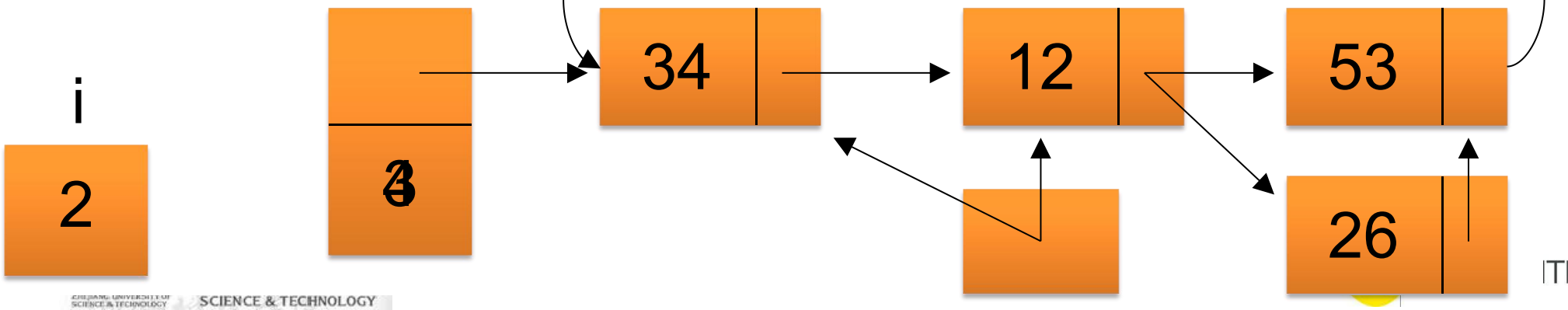


- **Implémentation C d'une liste simplement chaînée en utilisant deux structures**

```
void inserQueue(Liste *liste, int nvNombre){
    /* verification que la liste n'est pas vide*/
    if (liste->premier == NULL){
        printf("\nLa liste est vide\n");
        inserTete(liste, nvNombre);
    }
    /* création du nouvel élément*/
    Element *nouveau = (Element *)malloc(sizeof(Element));
    if (nouveau == NULL){
        printf("\nerreur d'allocation mémoire\n");
        exit(0);
    }
    nouveau->nombre = nvNombre;
    nouveau->suivant = liste->premier;

    /* rattachement de le nouveau élément à la fin de queue de la liste */
    Element *actuel = liste->premier;
    while(actuel->suivant != liste->premier){
        actuel = actuel->suivant;
    }
    actuel->suivant = nouveau;
    liste->cnt++;
}
```

Ajout à la ième position

SI i = 0 OU la liste est vide ALORS
    On ajoute au début de la liste
SINON SI i < nombre de nœuds de la liste ALORS
    On tente de créer un nouveau nœud.
    SI la création fut un succès ALORS
        On positionne un pointeur sur le i-1ème nœud.
        On initialise les données du nouveau nœud.
        On fait pointer le nouveau nœud sur le ième nœud (NULL si on ajoute à la fin).
        On fait pointer le i-1ème nœud sur le nouveau.
        On incrémente le nombre de nœuds.
    FINSI
FINSI



LISTES CHAINÉES

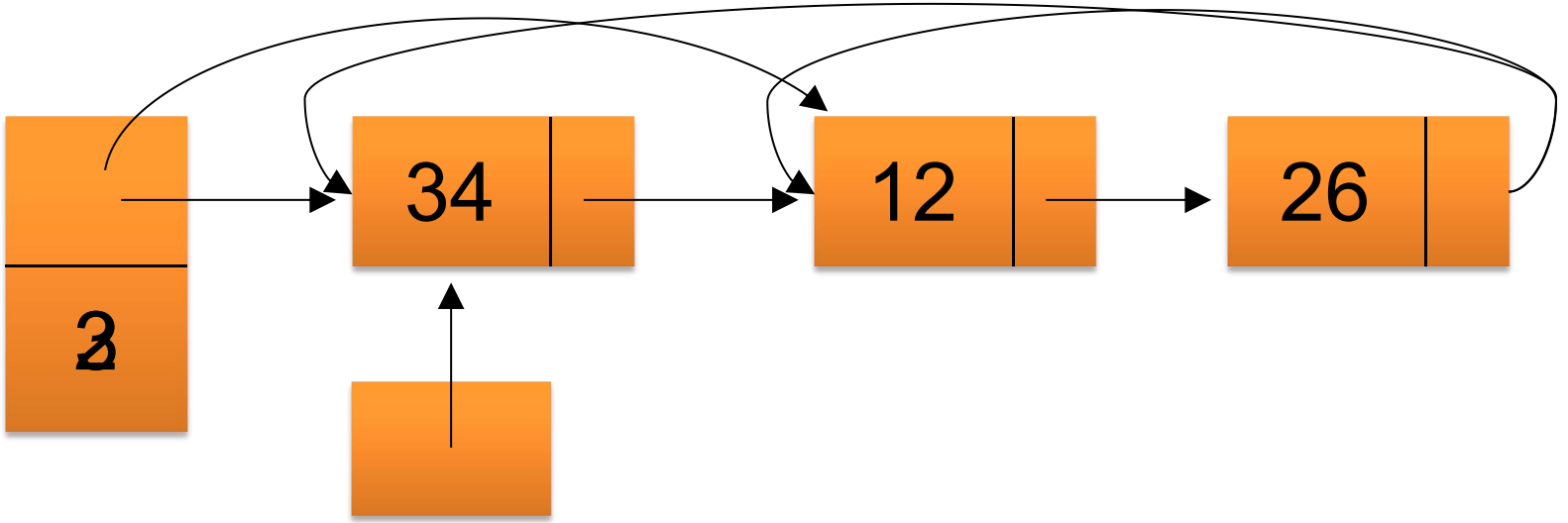
- **Implémentation C d'une liste simplement chaînée en utilisant deux structures**

```
void inserPos(Liste *liste, int v, int i){
    if (liste->premier == NULL || i == 0){
        inserTete(liste, v);
    }
    else{
            if (i < liste->cnt){
                    Element *nouveau = (Element *)malloc(sizeof(Element));
                    if (nouveau == NULL){
                            printf("\nerreur d'allocation mémoire\n");
                            exit(0);
                    }
                    Element *actuel = liste->premier;
                    for (int x = 1; x<i; x++ ){

                            actuel = actuel->suivant;
                    }
                    nouveau->nombre = v;
                    nouveau->suivant = actuel->suivant;
                    actuel->suivant = nouveau;
                    liste->cnt += 1;
            }
            else{
                    printf("la posisition est plus grand que le nombre des éléments --> inserQueue");
                    inserQueue(liste, v);
            }
    }
}
```

On commence l'itération de 1 car le pointeur actuel a déjà l'adresse du premier élément avant la boucle

Retrait au début de la liste

SI la liste n'est pas vide ALORS
On fait pointer un pointeur sur le premier élément de la liste.
On fait pointer la tête de liste sur le deuxième nœud (NULL s'il n'y avait qu'un seul nœud).
On détruit le nœud pointé par le pointeur.
On décrémente le nombre de nœuds.
FINSI

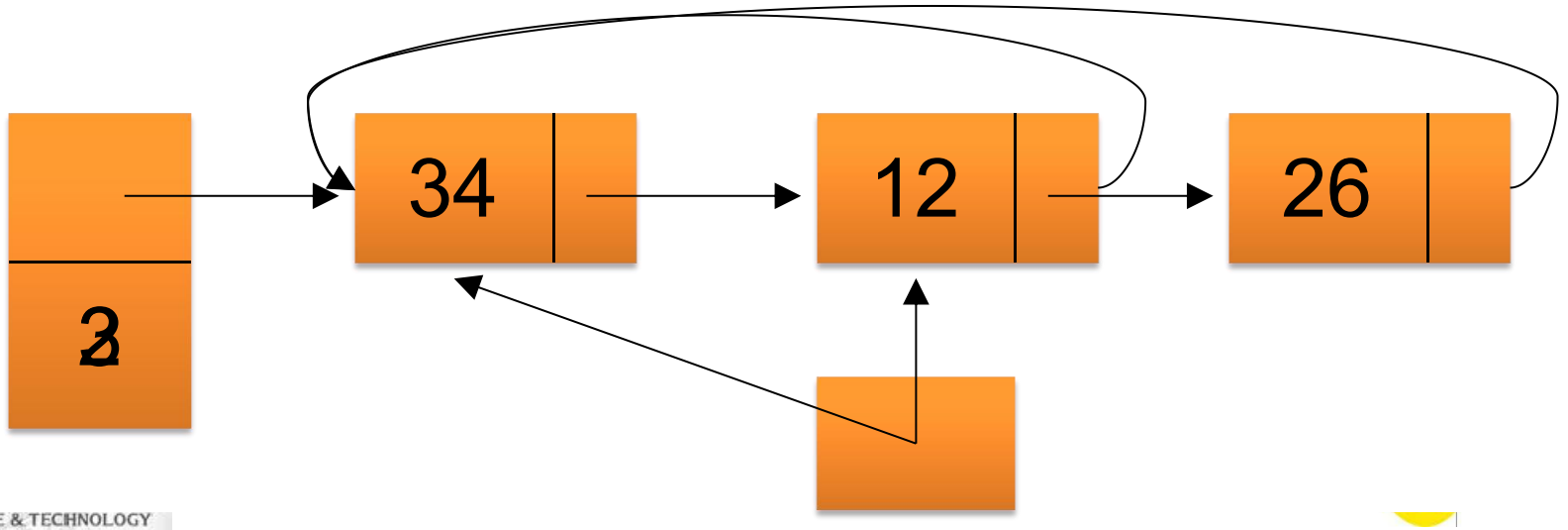


- **Implémentation C d'une liste simplement chaînée en utilisant deux structures**

```
void suppTete(Liste *liste){
    if (liste->premier == NULL){
        printf("\nLa liste est vide \n");
        exit(0);
    }

   Element *suppE = liste->premier;
   Element *actuel = liste->premier;
   while(actuel->suivant != liste->premier)
   {
            actuel = actuel->suivant;
   }
    liste->premier = liste->premier->suivant;
    actuel->suivant = liste->premier;
    free(suppE);
    liste->cnt--;
}
```

Retrait à la fin de la liste

SI la liste n'est pas vide ALORS
    SI la liste ne contient qu'un seul élément ALORS
        Retirer au début de la liste.
    SINON
        On fait pointer un pointeur sur l'élément à l'indice nombre de nœuds – 1.
        On détruit le dernier nœud.
        On fait pointer l'avant dernier vers le premier élément.
        On décrémente le nombre de nœuds.
    FINSI
FINSI

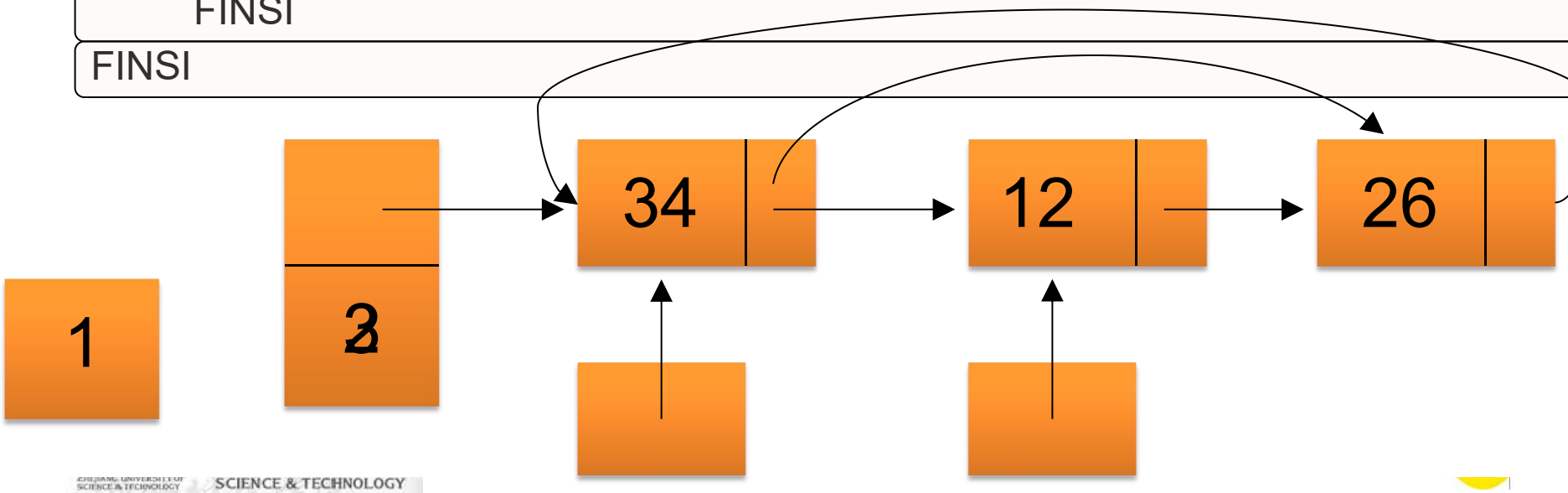


- **Implémentation C d'une liste simplement chaînée en utilisant deux structures**

```
void suppQueue(Liste *liste){
   /* verification que la liste n'est pas vide*/
   if (liste->premier == NULL){
      printf("\nLa liste est vide\n");
      exit(0);
   }
   /* suppression de le dernier élément de la liste */
   if (liste->cnt == 1)
                        suppTete(liste);
   else{

                        Element *actuel = liste->premier;
                        while(actuel->suivant->suivant != liste->premier){
                                actuel = actuel->suivant;
                        }
                        free(actuel->suivant);
                        actuel->suivant = liste->premier;
                        liste->cnt--;
            }
}
```

Retrait à la ième position

SI i ≥ 0 et i < nombre de nœuds de la liste ALORS
    SI la liste ne contient qu'un seul élément ALORS
        Retirer au début de la liste.
    SINON
        On fait pointer un pointeur sur l'élément à l'indice i – 1.
        On fait pointer un pointeur sur l'élément à l'indice i.
        On fait pointer le i-1ème nœud sur le i+1ème nœud (NULL si on détruit le dernier).
        On détruit le ième nœud.
        On décrémente le nombre de nœuds.
    FINSI
FINSI

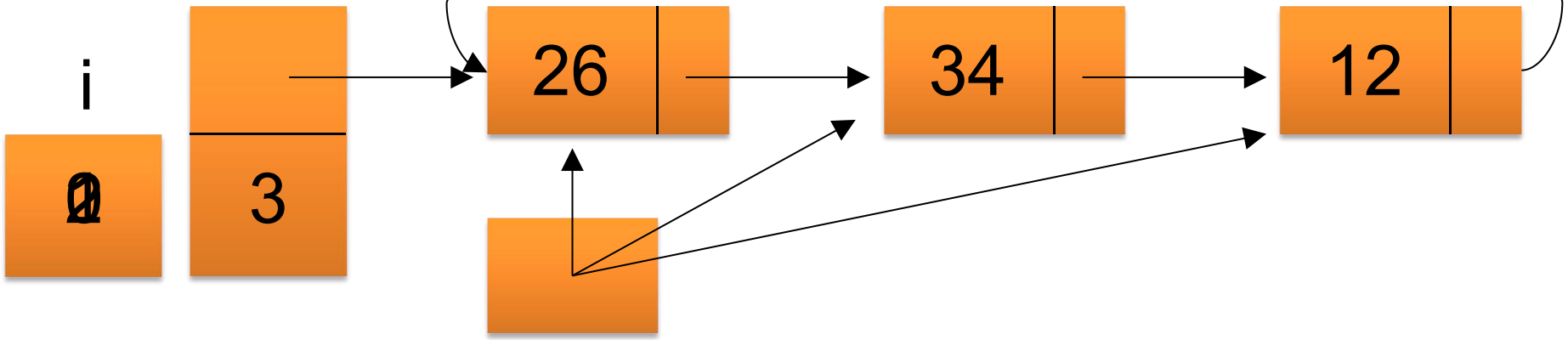


- **Implémentation C d'une liste simplement chaînée en utilisant deux structures**

```
void suppElemPos(Liste *liste, int i){
    if (liste->premier == NULL && i < 1){
        printf("\n La liste est vide\n");
        exit(0);
    }
    if (liste->cnt == 1)
                            suppTete(liste);
    Element *avant = liste->premier;
    for(int x = 1; x < i; x++)
                avant = avant->suivant;
    Element *suppElem = avant->suivant;
    avant->suivant = suppElem->suivant;
    free(suppElem);
    liste->cnt--;
}
```

Consultation du ième élément

```
SI i ≥ 0 et i < nombre de nœuds de la liste ALORS
    On fait pointer un pointeur sur le premier nœud.
    TANT QUE i > 0 BOUCLE
        Faire pointer le pointeur vers le prochain nœud.
        i ← i - 1
    FIN TANT QUE
FINSI
```



- **Implémentation C d'une liste simplement chaînée en utilisant deux structures**

```
int consultElem(Liste *liste, int i){

        if (i >= 0 && i < liste->cnt)
        {
                Element *actuel = liste->premier;
                while(i>0)
                {
                        actuel = actuel->suivant;
                        i--;
                }
        return actuel->nombre;
        }
        else exit(0);
}
```

- **Implémentation C d'une liste simplement chaînée en utilisant deux structures**

```
void afficheListe(Liste *liste)
{
  Element *actuel = liste->premier;
  while(actuel->suivant != liste->premier)
  {
    printf("%d -> ", actuel->nombre);
    actuel = actuel->suivant;
  }
  printf("%d ->>>> %d\n", actuel->nombre,
actuel->suivant->nombre);
}
```

Console Shell

```
> clang-7 -pthread -lm -o main circulaire.c main.c
> ./main
Ajout au début de la liste
5 ->>>> 5
Ajout au début de la liste
4 -> 5 ->>>> 4
Ajout au début de la liste
3 -> 4 -> 5 ->>>> 3
Ajout au début de la liste
2 -> 3 -> 4 -> 5 ->>>> 2
Ajout à la fin de la liste
2 -> 3 -> 4 -> 5 -> 6 ->>>> 2
Ajout à la 2ème position
2 -> 3 -> 0 -> 4 -> 5 -> 6 ->>>> 2
Retrait au début de la liste
3 -> 0 -> 4 -> 5 -> 6 ->>>> 3
Retrait à la fin de la liste
3 -> 0 -> 4 -> 5 ->>>> 3
Retrait à la position 1
3 -> 4 -> 5 ->>>> 3
Consultation de l' élément à la position 1
l'élément à position 1 : 4
>
```