Chapitre 2
Les listes chainées

- Partie I: listes simplement chaînées
- Partie II: listes doublement chaînées

Références

1. «Programmer en Langage C cours et exercices»
Auteur: Claude Delannoy
Code bibliothèque: I-22/DEL
Edition: Eyrolles 5ème Edition (2012)

2. «Exercices et Problème d'algorithmes»
Auteurs: Bruno Baynat et collaborateurs
Code bibliothèque: I-15/BAY
Edition: Dunod (2003)

3. «Types de données et Algorithmes»
Auteur: Christine Froidevaux et collaborateurs
Code bibliothèque: I-1/FRO
Edition: McGraw-Hill (1990)

Partie I:
Listes simplement chainées

Introduction

- Nous avons vu précédemment que pour stocker une **collection d'informations** de **types différents** et éviter le problème posé lors de la **réservation de la place mémoire** (effectuée lors de la déclaration d'un tableau par exemple), nous utilisons une **structure de donnée**.

- Une **liste chaînée** est une **structure de données** qui est constituée d'un ensemble d'**éléments** chaînés entre eux contenant les informations à mémoriser.

- Dans une **liste chaînée** l'emplacement mémoire nécessaire à chaque **élément** est alloué **dynamiquement**.

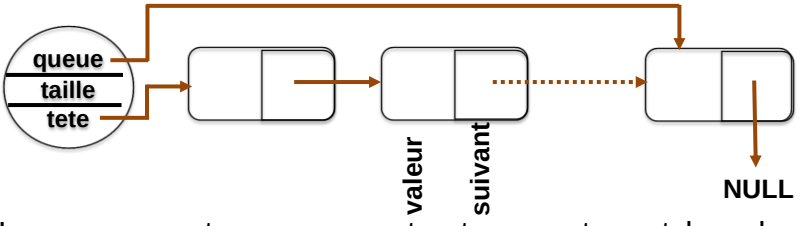
Introduction

- **Intérêt des listes simplement chaînées :**
  - **Type Abstrait de Données** ou **TAD** (**Abstract Data Type** ou **ADT en anglais**), linéaire et asymétrique;
  - Permet de manipuler des ensembles dynamiques;
  - Sont à la base de plusieurs **TAD** avancés: files, piles, arbres, ...

- **Applications des listes simplement chaînées :**
  - Système d'exploitation: file d'attente des processus;
  - Client/serveur: file d'attente des messages;
  - Base de données: pool de connexion avec un SGBD;
  - Programmation: Langage Lisp;
  - etc….

Définition d'une liste simplement chainée

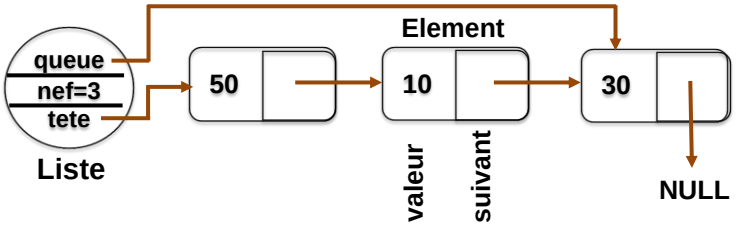
- Les listes simplement chainées sont un regroupement ordonné de données structurées de manière à ce que chaque composante sache où se trouve la suivante.



- Une composante sera une structure contenant la valeur de l'information mémorisée et également un pointeur sur la composante suivante. Une liste simplement chainée est accessible par l'adresse de sa première composante.
- On supposera dans la suite que les valeurs à mémoriser sont d'un type structures, composé de plusieurs champs, nommé valeur.

Définition d'une liste simplement chainée

- Pour commencer, voyez cette figure :



- Une liste simplement chaînée peut être vu comme:
  - Une structure de contrôle **(Liste)**
  - Une structure d'éléments enchaînés **(Element).**
- Une **Liste** possède un pointeur «**tete**», un pointeur «**queue**» et un nombre d'éléments final (**nef**).
- Un **Element** possède une valeur stockée «**valeur**» et un pointeur vers le prochain élément «**suivant**».

Structure d'un élément vide de la liste chaînée

- En implémentation avec les **pointeurs**, voyez cette définition:

Structure d'un élément

```
typedef int Ttype; //on définit généralement un type  (int, float, struct, ...)
typedef struct ElementRepere{
 Ttype valeur;
 struct ElementRepere *suivant;
}Element;
```

- Un élément de la liste chaînée est définie par le mot clé **struct**, suivi du nom de la structure (**ElementRepere**) et de la liste de ses champs (**valeur** et **suivant**), entre accolades.

Structure de contrôle d'une liste chaînée

- Voyez encore cette définition:

Structure de contrôle

```
typedef struct ListeRepere{
  Element *tete;
  Element *queue;
  int nef;
}Liste;
```

- Une structure de contrôle est définie par le mot clé **struct**, suivi du nom de la liste (**ListeRepere**) et de la liste de ses champs (**tete**, **queue** et **nef**), entre accolades.

Utiliser une liste chaînée

- Au cours de sa déclaration on peut utiliser une liste chaînée en travaillant individuellement sur chacun de ses champs.
- La désignation d'un **champ** par une **variable** de type pointeur peut se faire de l'un des deux manières:
  - **Variable . champ**; (voir les structures dans **Ch1 !**)
  - **Variable → champ**;

Exemples

```
Liste *Li;                    Liste *Li;
Li → tete;          ⟺        (*Li).tete;
Li → queue;                   (*Li).queue;
```


Créer un élément vide d'une liste chaînée

- La fonction **CréerElement()** va nous permettre de réserver à chaque fois l'espace mémoire pour un élément **vide** de la liste:

Fonction CreerElement()

```
Element * CreerElement( ){
  Element *El;
  El=(Element*)malloc(sizeof(Element));
  El→valeur=0;
  El→suivant=NULL;
  return(El);
}
```

Remarque 1

- L'initialisation de **suivant** à **NULL** permet d'éviter le bug de violation d'accès (le pointeur point vers une donnée du système) !

Créer une structure de contrôle d'une liste chaînée

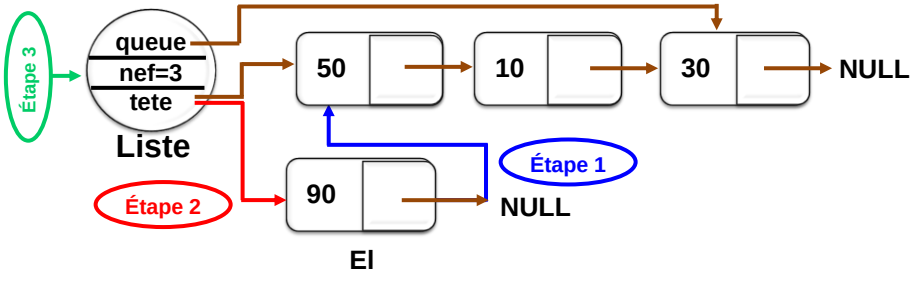
- La fonction **CréerListe()** va nous permettre de réserver l'espace mémoire pour la structure de contrôle.

Fonction CreerListe()

```
Liste * CreerListe( ){
  Liste *Li;
  Li = (Liste*)malloc(sizeof(Liste));
  Li→tete = NULL;
  Li→queue = NULL;
  Li→nef = 0;
  return(Li);
}
```


Insérer un élément à la tête d'une liste chaînée

- Voyez d'abord la représentation schématique suivante:



Pré-chainage

- **Allouer un espace mémoire au nouveau élément (El) ensuite, mettre la valeur à ajouter au début dans ce nouveau élément.**

Les étapes:

1. **«suivant» de «El» pointe sur le premier élément;**
2. **«tete» pointe sur le nouveau élément;**
3. **Mettre à jour la taille de la liste.**

Insérer un élément à la tête d'une liste chaînée

- Voici le code de la fonction **AjoutDebut():**

Fonction AjoutDebut()

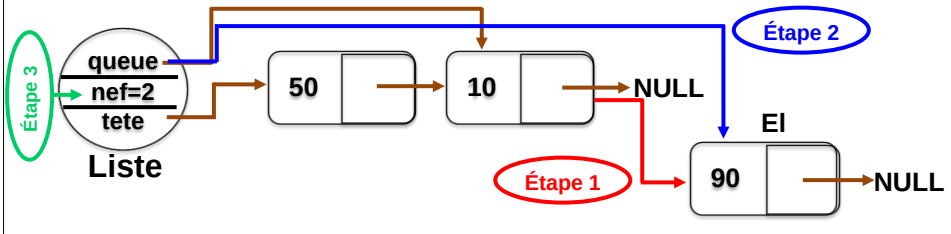
```
void AjoutDebut(Liste* Li, Ttype vad){
 Element* El;
 El = CreerElement();
 if(Li == NULL || El == NULL){
   exit(EXIT_FAILURE); //quitter le pgm
 }
 El→valeur = vad;
 //cas d'une liste vide
 if(Li→tete==NULL){
    Li→tete = El;
    Li→queue = El;
 }
```

Fonction AjoutDebut()

```
 //cas d'une liste non vide
 else{
  El→suivant = Li→tete;
  Li→tete = El;
  }
  Li→nef++;
}
```


Insérer un élément à la fin d'une liste chaînée

- Voyez d'abord la représentation schématique suivante:



Pré-chainage

- **Allouer un espace mémoire au nouveau élément (El) ensuite, mettre la valeur à ajouter à la fin dans ce nouveau élément.**

Les étapes :

1. **«suivant» de «queue» pointe sur le nouveau élément;**
2. **«queue» pointe sur le nouveau élément;**
3. **Mettre à jour la taille de la liste.**

Insérer un élément à la fin d'une liste chaînée

- Voici le code de la fonction **AjoutFin():**

Fonction AjoutFin()

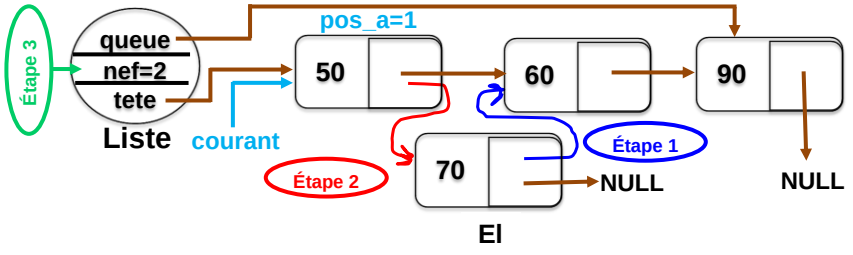
```
void AjoutFin(Liste* Li, Ttype vaf){
  Element* El;
  El = CreerElement();
  if(Li == NULL || El == NULL){
    exit(EXIT_FAILURE);//quitter le pgm
  }
  El→valeur = vaf;

//cas de liste vide
  if(Li→tete==NULL){
      Li→tete = El;
      Li→queue = El;
  }
```

Fonction AjoutFin() suite

```
  //cas de liste non vide
  else{
     Li→queue→suivant=El;
     Li→queue = El;
   }
   Li→nef++;
}
```


Insérer un élément après une position

- Voyez d'abord la représentation schématique suivante:



Pré-chainage

- **Allouer un espace mémoire au nouveau élément (El) ensuite, mettre la valeur à ajouter après une position (vap) dans ce nouveau élément.**
- **Chercher la position d'ajout « pos_a» et en faire pointer «courant».**

Les étapes :

1. **«suivant» de «El» pointe sur «suivant» de «courant»,**
2. **«suivant» de «courant» pointe sur «El».**
3. **Mettre à jour la taille de la liste.**

Insérer un élément après une position

- Voici le code de la fonction **AjoutPosition():**

Fonction AjoutPosition()

```
void AjoutPosition(Liste* Li,
       Ttype vap, int pos_a){
Element *El, *courant;
int i;
El = CreerElement();
if(Li == NULL || El == NULL){
    exit(EXIT_FAILURE);
}
  //Les conditions imposés pour pos_a
if(pos_a<1 || pos_a >= Li→nef){
    printf("\nImpossible d'insérer
    après cette position);
}
```

Fonction AjoutPosition() suite

```
else{
 El →valeur = vap;
 //On cherche la position d'ajout (pos_a)
 courant = Li→tete;
 for(i=1; i<pos_a; i++){
    courant = courant→suivant;
 }

 El→suivant =courant→suivant;
 courant→suivant = El;
 Li→nef++;
}
}
```


Afficher une liste chaînée

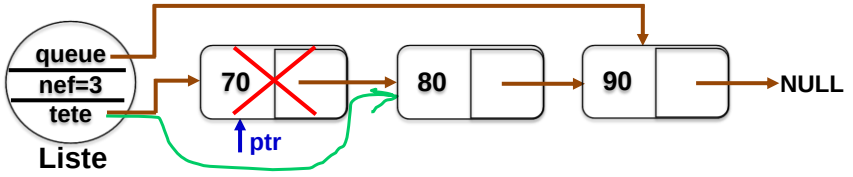
- La fonction **AfficherListe()** va nous afficher une liste chaînée

Fonction AfficherListe ()

```
void AfficherListe(Liste* Li) {
  Element* ptr;
  ptr= Li→tete ;
  printf("Li = ");                  //pour la forme
  while(ptr  != NULL){
       printf(" %d→ ", ptr→valeur );
       ptr=ptr→suivant;
   }
  printf("NULL\n");
}
```

Supprimer un élément à la tête d'une liste chaînée

- Voyez d'abord la représentation schématique suivante:



Pré-chainage

- **Initialiser un pointeur «ptr» à la tête de la liste pour récupérer la valeur à supprimer au début et libérer l'élément.**

Les étapes:

1. **«tete» pointe sur «suivant» de l'élément à supprimer;**
2. **Récupérer la valeur supprimée (vsd) et libérer «ptr»;**
3. **Mettre à jour la taille de la liste.**

Supprimer un élément à la tête d'une liste chaînée

- Voici le code de la fonction **SupprimerDebut() :**

Fonction SupprimerDebut()

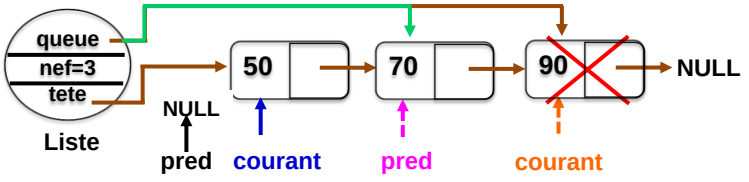
```
Ttype SupprimerDebut(Liste*Li){
  Ttype vsd; //valeur supprimée au début
  Element* ptr;
  if(Li == NULL){
    exit(EXIT_FAILURE);
  }
  //cas d'une liste vide
  if(Li→tete==NULL){
   printf("\nImpossible de supprimer ");
  }
  else{
     ptr = Li→tete;
     Li→tete = Li→tete→suivant;
```

Fonction SupprimerDebut()

```
    vsd=ptr→valeur;
    free(ptr);

    if(Li→tete == NULL){
        Li→queue = NULL;
    }
    Li→nef--;
    return(vsd);
  }
}
```

Supprimer un élément à la fin d'une liste chaînée

- Voyez la représentation schématique suivante:



Les étapes :

1. **Localiser le dernier élément de la liste par «courant» et «pred»;**
2. **Faire pointer «queue» sur «pred»;**
3. **Récupérer la valeur supprimée (vsf) et libérer «courant»;**
4. **Faire pointer le suivant de «queue» à «NULL»;**
5. **Mettre à jour la taille de la liste.**

Supprimer un élément à la fin d'une liste chaînée

- Voici le code de la fonction **SupprimerFin() :**

Fonction SupprimerFin()

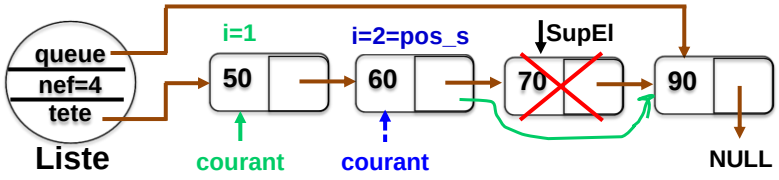
```
Ttype SupprimerFin(Liste* Li){
 Ttype vsf;
 Element *courant ,*pred;
 if(Li == NULL ){
    exit(EXIT_FAILURE);
 }
 if(Li→nef == 0){//cas d'une liste vide
    printf("\nImpossible de supprimer ");
 }
 else{//cas d'une liste  non vide
  courant = Li→tete;
  pred = NULL;
  while(courant→suivant != NULL){
     pred = courant;
     courant = courant→suivant;
  }
```

Fonction SupprimerFin()

```
  Li→queue=pred;
  vsf = courant→valeur;
  free(courant);

  if(queue != NULL){
     pred→suivant = NULL;
  }

  else{
     Li→tete = NULL;
  }
  Li→nef--;
  return(vsf);
 }
}
```

Supprimer un élément après une position

- Voyez la représentation suivante:



Les étapes:

1. **Localiser la position de suppression par «courant» et l'élément à supprimer par «SupEl»;**
2. **«suivant» de courant pointe sur «suivant» de «suivant» de courant;**
3. **Récupérer la valeur supprimée (vsp) et libérer «SupEl».**
4. **Mettre à jour la taille de la liste.**

Supprimer un élément après une position

- Voici le code de la fonction **SuppPos() :**

Fonction SuppPos()

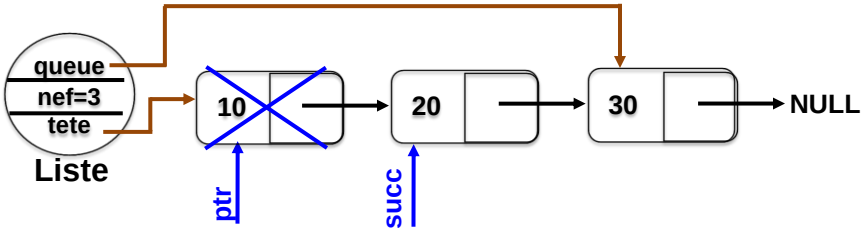
```
Ttype SuppPos(Liste*Li, int pos_s){
Element *courant,*SupEl;
Ttype  vsp;
int i;
if(Li == NULL){
    exit(EXIT_FAILURE);
}
//Les conditions imposés
if(Li→nef<=1||pos_s<1||pos_s>=Li→nef){
    printf("\nImpossible de supprimer ");
}
else{
   courant = Li→tete;
   for(i = 1; i < pos_s; i++){
        courant = courant→suivant;
   }
```

Fonction SuppPos()

```
SupEl=courant→suivant;
courant→suivant=
courant→suivant→suivant;
vsp = SupEl→valeur;
free(SupEl);

if(courant→suivant==NULL){
    Li→queue = courant;
}
Li→nef--;
return(vsp);
 }
}
```

Vider une liste chaînée

- Voyez la représentation schématique suivante:



Pré-chainage

- **Initialiser «ptr» à «tete» et «succ» au successeur de «tete» .**

Les étapes :

1. **Parcourir la liste par «ptr» et «succ» et libérer la mémoire de chaque élément pointé par «ptr»;**
2. **Initialiser «tete» et «queue» à NULL;**
4. **Mettre à jour la taille de la liste.**

Vider une liste chaînée

- Voici le code de la fonction **ViderListe() :**

Fonction ViderListe()

```
void ViderListe(Liste* Li){
  Element* ptr, * succ;
  ptr = Li→tete;
  while(ptr != NULL){
      succ = ptr→suivant;
      free(ptr);
      ptr = succ;
   }
  Li→tete = NULL;
  Li→queue = NULL;
  Li→nef = 0;
}
```

En résumé

- Nous avons découvert que les listes chaînées constituent un nouveau moyen de stocker les données en mémoire. Elles sont plus flexibles que les tableaux puisque, on peut ajouter et supprimer des «cases» sans aucun problème.
- Dans une liste simplement chaînée, chaque élément est une structure qui contient l'adresse de l'élément suivant.
- En C, Il n'existe pas de système de gestion des listes chaînées et donc, il faut l'écrire par nous-mêmes ! C'est un excellent moyen de progresser en programmation.

Partie II: Listes doublement chainées

Introduction

- Pour stocker une collection d'informations et éviter le problème d'**asymétrie** de données posé par les listes simplement chaînées, on utilise une structure de donnée améliorée appelée **liste doublement chaînée**.

- Une **liste doublement chaînée** est une structure de données constituée d'un ensemble d'**éléments** chaînés entre eux dans une **double sens** et contenant les informations à mémoriser.

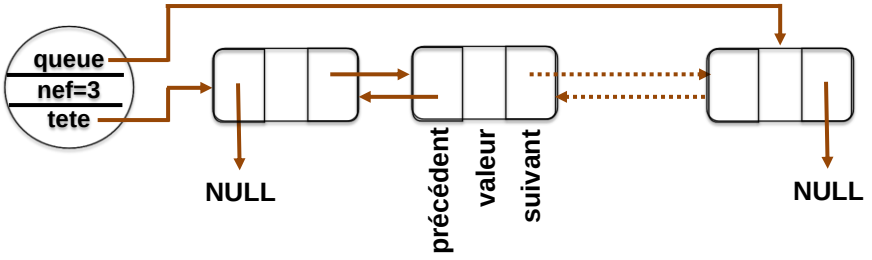
Introduction

- **Intérêt des listes doublement chaînées :**
  - **Type Abstrait de Données** ou **TAD** (**Abstract Data Type** ou **ADT en anglais**), linéaire et symétrique;
  - Permet de manipuler des ensembles dynamiques;
  - Sont à la base de plusieurs **TAD** avancés: chemins élémentaires, circuits élémentaires, etc.

- **Applications des listes doublement chaînées :**
  - Modélisation des réseaux;
  - Graphes;
  - Algorithmes géométriques;
  - etc.

Définition d'une liste doublement chainée

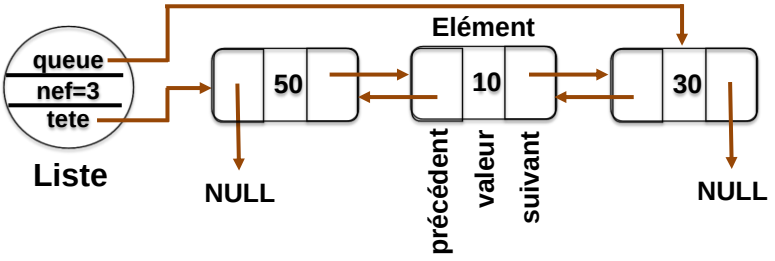
- Les listes doublements chainées sont un regroupement ordonné de données structurées de manière à ce que chaque composante sache où se trouve la suivante et la précédente.



- Une composante sera une structure contenant la valeur de l'information mémorisée et également un pointeur sur la composante suivante et un autre pointeur sur la composante précédente. De cette manière, la liste peut être repérée par l'intermédiaire de n'importe lequel de ses éléments en faisant une recherche avant ou une recherche arrière (symétrique).

Définition d'une liste doublement chainée

- Pour commencer, voyez cette figure :



- Une **liste doublement chaînée** peut être vu comme :
  - Une structure de contrôle **(Liste);**
  - Une structure d'éléments enchaînés **(Element).**
- Une **Liste** possède une **tête**, une **queue** et un **nombre d'éléments finale (nef).**
- Chaque **Element** connaît sa(ses) valeur(s) stockée(s) **valeur** et ses éléments dans la liste **précédent** et **suivant.**

Structure d'un élément vide et structure de contrôle

- Voyez la définition de la structure d'un élément :

Élément de la liste

```
typedef float Ttype; //Le programmeur peut spécifier Ttype
typedef struct ElementRepere{
  Ttype valeur;
  struct ElementRepere *precedent; //pointeur sur l'élément précédent
  struct ElementRepere *suivant; //pointeur sur l'élément suivant
}Element;
```

- Voyez aussi la définition de la structure de contrôle :

Structure de contrôle

```
typedef struct ListeRepere{
    Element *tete;
    Element *queue;
    int nef;
}Liste;
```


Créer un élément vide de la liste

- La fonction **CréerElement()** va nous permettre de réserver à chaque fois l'espace mémoire pour un élément **vide** de la liste chaînée, voici sa composition :

Fonction CreerElement()

```
Element* CreerElement( ){
  Element* El;
  El = (Element*)malloc(sizeof(Element));
  El→valeur=0;
  El→precedent=NULL;
  El→suivant=NULL;
  return(El);
}
```

Créer une structure de contrôle de la liste

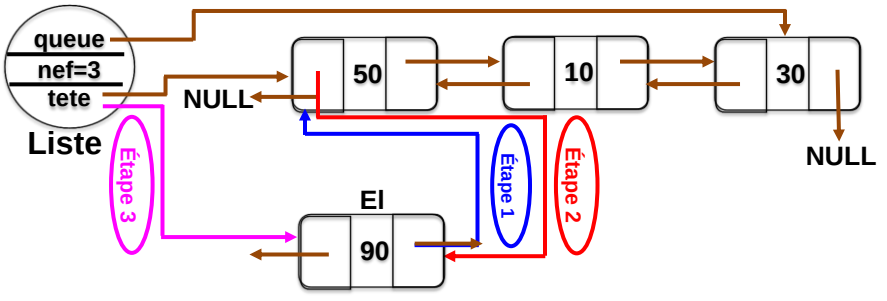
- La fonction **CréerListe()** va nous permettre de réserver l'espace mémoire pour la liste, voici sa composition:

Fonction CreerListe()

```
Liste* CreerListe( ){
  Liste* Li;
  Li = (Liste*)malloc(sizeof(Liste));
  Li→tete=NULL;
  Li→queue=NULL;
  Li→nef=0;
  return(Li);
}
```


Insérer un élément à la tête de la liste

- Voyez tout d'abord la représentation schématique suivante:



Pré-chainage

- **Allouer un espace mémoire au nouveau élément (El) ensuite, mettre la valeur à ajouter au début (vad) dans ce nouveau élément.**

Les étapes à suivre

1. **Effectuer successivement Étape 1, Etape 2 et Etape 3.**
2. **Mettre à jour la taille de la liste.**

Insérer un élément à la tête de la liste

■ Voici le code de la fonction **AjoutDebut():**

Fonction AjoutDebut()

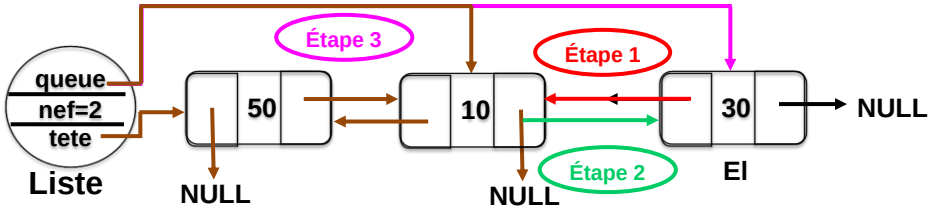
```
void AjoutDebut(Liste* Li, Ttype vad){
  Element* El;
  El = CreerElement();
  if(Li == NULL || El == NULL){
     exit(EXIT_FAILURE);
   }
  El→valeur = vad;

 /*cas de la liste vide */
  if(Li→tete == NULL){
     Li→tete=El;
     Li→queue=El;
  }
```

suite

```
//cas de la liste non vide
 else{
  El→suivant=Li→ tete;
  Li→tete→precedent=El;
  Li→tete = El;
 }
 Li→nef ++;
}
```


Insérer un élément à la fin de la liste

■ Voyez la représentation schématique suivante:



Pré-chainage

- **Allouer un espace mémoire au nouveau élément (El) ensuite, mettre la valeur à ajouter à la fin (vaf) dans ce nouveau élément.**

Les étapes:

1. **Effectuer successivement Étape 1, Etape 2 et Etape 3;**
2. **Mettre à jour la taille de la liste.**

Insérer un élément à la fin de la liste

- Voici le code de la fonction **AjoutFin():**

Fonction AjoutFin()

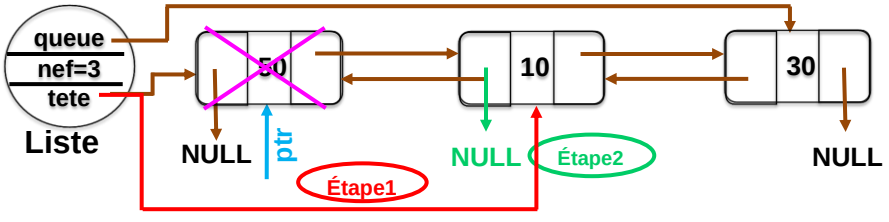
```
void AjoutFin(Liste* Li, Ttype vaf){
 Element* El;
 El = CreerElement();
 if(Li == NULL || El == NULL){
    exit(EXIT_FAILURE);
 }
 El→valeur = vaf;

/*cas de la liste vide */
 if(Li→tete == NULL){
    Li→tete=El;
    Li→queue=El;
 }
```

suite

```
 //cas de la liste non vide
else{
  El→precedent =Li→queue;
  Li→ queue→ suivant= El;
  Li→ queue = El;
}
Li→ nef ++;
}
```


Supprimer un élément à la tête de la liste

- Voyez la représentation suivante:



Pré-chainage

- **Initialiser un pointeur «ptr» au début de la liste.**

Les étapes :

1. **Effectuer successivement Étape 1 et Etape 2;**
2. **Récupérer la valeur supprimée au début (vsd) et libérer «ptr».**
3. **Mettre à jour la taille de la liste.**

Supprimer un élément à la tête de la liste

- Voici le code de la fonction **SupprimerDebut():**

Fonction SupprimerDebut()

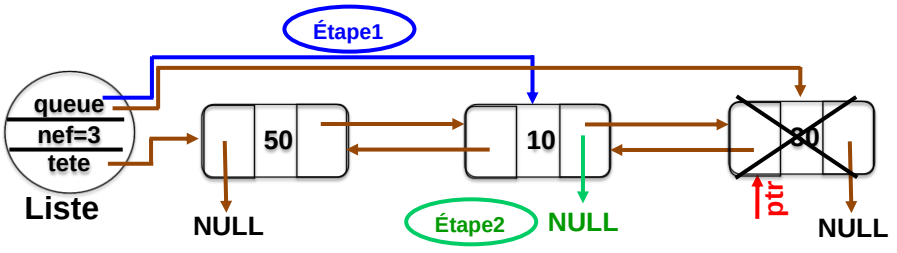
```
Ttype SupprimerDebut(Liste*Li){
 Ttype vsd; //valeur supprimée au début
 Element* ptr;
 if(Li == NULL){
   exit(EXIT_FAILURE);
 }
 //cas d'une liste vide
 if(Li→tete==NULL){
  printf("\nImpossible de supprimer");
 }
 else{
   ptr = Li→tete;
    Li→tete = Li→tete→suivant;
    Li→tete→precedent=NULL;
```

SupprimerDebut()

```
   vsd=ptr→valeur;
   free(ptr);

   if(Li→tete == NULL){
       Li→queue = NULL;
    }
    Li→nef--;
    return(vsd);
  }
}
```


Supprimer un élément à la fin de la liste

- Voyez d'abord la représentation schématique suivante:



Pré-chainage

- **Initialiser un pointeur «ptr» à la fin de la liste.**

Les étapes :

1. **Effectuer successivement Étape 1 et Etape 2;**
2. **Récupérer la valeur supprimée au début (vsd) et libérer «ptr».**
3. **Mettre à jour la taille de la liste.**

Supprimer un élément à la fin de la liste

- Voici le code de la fonction **SupprimerFin():**

Fonction SupprimerFin()

```
Ttype SupprimerFin(Liste* Li){
  Ttype vsf;
  Element *ptr;
  if(Li == NULL ){
     exit(EXIT_FAILURE);
  }

 //Cas de la liste est vide
  if(Li→tete==NULL){
    printf("\nImpossible de supprimer");
  }

  //Cas de la liste non vide
  else{
    ptr= Li→queue;
    Li→queue =ptr→precedent;
    Li→ queue →suivant=NULL;
```

Fonction SupprimerFin()

```
    vsf = ptr→valeur;
    free(ptr);

    if(Li→tete == NULL){
        Li→queue = NULL;
    }
    Li→nef--;
    return(vsf);
  }
}
```


Afficher une liste doublement chaînée

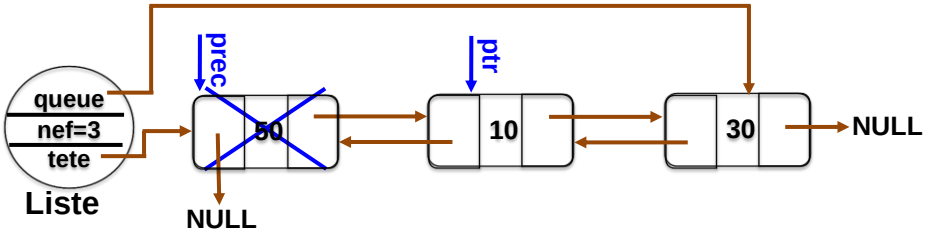
- La fonction **AfficherListe()** pour afficher une liste chaînée.

Fonction AfficherListe ()

```
void AfficherListe(Liste* Li) {
   Element* ptr;
    ptr= Li→tete;
   printf("Li = ");                    //pour la forme
   printf("NULL←");
  while(ptr != NULL){        /*condition d'arrêt: fin de la liste*/
        printf("%f←→", ptr→valeur );
        ptr=ptr→suivant;
   }
   printf("NULL\n");
}
```

Vider une liste doublement chaînée

- Voyez d'abord la représentation schématique suivante:



Pré-chainage

- **Initialiser «ptr» à «tete» et son précèdent «prec» à «NULL» .**

Les étapes :

1. **Parcourir la liste par «ptr» et «prec» et libérer la mémoire de chaque élément pointé par «prec»;**
2. **Initialiser «tete» et «queue» à «NULL»;**
3. **Mettre à jour la taille de la liste.**

Vider une liste doublement chaînée

- Voici le code de la fonction **ViderListe() :**

Fonction ViderListe()

```
void ViderListe(Liste* Li){
  Element *ptr, *prec;
  ptr=Li→tete;
  prec=NULL;
  while (ptr != NULL){
      prec = ptr;
      ptr = ptr→suivant;
      free(prec);
   }
   Li→tete = NULL;
   Li→queue = NULL;
   Li→nef = 0;
}
```

En résumé

- Nous avons découvert que les listes doublement chaînées constituent une version améliorée des listes simplement chaînées.
- Chaque élément est une structure qui contient non seulement l'adresse de son élément suivant, mais aussi l'adresse de son précèdent.