Chapitre 2 : Les listes

Université Mohamed 1er
Faculté des sciences
Oujda

Sommaire

Terminologie

Terminologie

- **Définition** :
  - Une **liste** est une **séquence finie** formée de 0 ou plusieurs **éléments** de **même type**.
  - Chaque **élément** dans une liste est repéré par son **rang** (sa **place**) dans cette liste.
  - La **longueur** d'une liste est son **nombre d'éléments**.
  - Une liste de **longueur** N dispose alors de N **rangs** (de 1 à N, par exemple).

Terminologie

- **Notation (théorique)** :
  - On note souvent une liste de longueur $N \geq 1$ par:

    $$\lambda = (e_1, e_2, ..., e_N)$$
  - Lorsque N = 0, on dit que la liste est **vide**.
  - Un élément quelconque e_i de la liste λ est caractérisé par :
    - ✓ son **rang** (ou sa **place**) noté ***rang(e)***
    - ✓ son **contenu** noté ***contenu(e)***

Terminologie

- **Exemples** :
  - La listes des entiers impairs ≤ 20 :

    (1, 3, 5, 7, 9, 11, 13, 15, 17, 19)
  - La liste des mois ayant moins de 31 jours :

    (février, avril, juin, septembre, novembre)
  - La liste des gaz parfaits dans l'ordre de leur masse atomique :

    (hélium, néon, argon, krypton, xénon, radon)

Terminologie

- **Parties d'une liste (non vide) $\lambda = (e_1, e_2, \ldots, e_N)$:**
  - L'élément e_1 est appelé la **tête** de la liste : c'est son **premier** élément.
  - $(e_2, \ldots, e_N)$ est appelée la **queue** de la liste (ou son **reste**). Il s'agit bien d'une **liste**.
  - L'élément e_{i+1} est le **successeur** de l'élément e_i (l'élément e_N n'a pas de successeur).
  - L'élément e_{i-1} est le **prédécesseur** de l'élément e_i (l'élément e_1 n'a pas de prédécesseur).

Terminologie

- **Parties d'une liste (non vide) $\lambda = (e_1, e_2, \ldots, e_N)$:**
  - Une **sous-liste** de λ est une liste **extraite** de λ :
    $(e_i, e_{i+1}, \ldots, e_j)$, avec $1 \le i \le j \le N$.
  - Une **sous-séquence** de λ est une liste obtenue à partir de λ en supprimant 0 ou plusieurs de ses éléments.
  - Un **préfixe** de λ est une **sous-liste qui commence au début** de la liste λ :
    $(e_1, e_2, \ldots, e_j)$, avec $1 \le j \le N$.

Terminologie

- **Parties d'une liste (non vide) $\lambda = (e_1, e_2, \ldots, e_N)$:**
  - Une **suffixe** de λ est une sous-liste qui se termine à la fin de la liste λ :
    $(e_i, e_{i+1}, \ldots, e_N)$
- **Propriétés :**
  - Pour toute liste λ, la liste **vide** notée $\varnothing$ et la liste λ elle-même sont des sous-listes, des sous-séquences, des préfixes et des suffixes de λ.
  - Une liste λ de **longueur N**, possède exactement **(N + 1) préfixes** et **(N + 1) suffixes**.

Opérations sur les listes

Opérations sur les listes

- **Opérations fondamentales (de type « dictionnaire ») :**
  - L'**insertion** d'un élément.
  - La **suppression** d'un élément.
  - La **recherche d'**un élément.
- **Quelques opérations de type « requête » :**
  - Savoir si la liste est **vide**.
  - Calculer la **longueur** d'une liste.
  - Calculer le **nombre d'occurrences** d'un élément dans une liste, …etc.

Opérations sur les listes

- **D'autres opérations :**
  - La **fusion** (ou la **concaténation**) de plusieurs listes.
  - La **recopie** d'une liste.
  - Le **renversement** d'une liste.
  - Le **tri** (***sorting***) d'une liste.
  - Le **partitionnement** d'une liste.
  - Etc.

Implémentation contiguë

Implémentation des listes

- **Deux méthodes pour implémenter une liste** :
  - Implémentation **contiguë** à l'aide d'un **tableau**.
  - Implémentation **non contiguë** à l'aide d'une structure de données spéciale appelée **liste chaînée** basée sur les **pointeurs**.

Implémentation contiguë

- **Implémentation contiguë** :
  - Une solution **contiguë** par **tableaux** consiste à créer une **structure** composée de deux **champs**:
    - ✓ Un **tableau** qui **stocke** les **différents éléments** de la liste.
    - ✓ Une **variable** qui **mémorise** le **nombre des éléments** couramment présents dans la liste.
  - Un tableau est un ensemble d'éléments en **nombre fixé**, mémorisés dans une **zone contiguë** et accessibles par un **indice**.

Implémentation contiguë

- **Avantages** :
  - **Simplicité**.
  - **L'opération d'accès** à un élément est **très rapide**.
  - Très utiles pour représenter des **structures linéaires statiques** :
    - ✓ **Ensembles** au sens mathématique
    - ✓ **Vecteurs**
    - ✓ **Matrices**

Implémentation contiguë

- **Inconvénients de cette implémentation** :
  - La **taille** d'un **tableau statique** est **fixe**. D'où la nécessité de spécifier une **borne supérieure** de cette taille au moment de la compilation, ce qui entraîne d'allouer une quantité de mémoire qui peut être **non utilisable** totalement.
  - **L'insertion** en une position intermédiaire d'un nouvel élément est **coûteuse** car elle nécessite le **décalage** des éléments existants.
  - La **suppression effective** d'un élément est **impossible**.

Implémentation contiguë

- **Exemple : une liste d'entiers implémentée par un tableau**
  - **Déclaration de la structure** pour **l'implémentation contiguë** :

```
#define MAX 100
typedef struct LIST
{
  int A[MAX]; /* conteneur des éléments */
  int length;     /* nombre actuel des éléments */
} LIST;
```

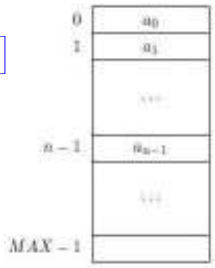
Implémentation contiguë

- **Exemple :**
  - Représentation d'une liste $\lambda = (a_0, a_1, \ldots, a_{N-1})$ en utilisant un tableau A[0…MAX-1].

$\forall i \in \{0, \ldots, N-1\}$:

$A[i] = a_i$

length = N



Implémentation contiguë

- **Opérations de création d'une liste** :
  - **Créer une liste** :
    - ✓ Il suffit **d'initialiser** length à 0.
    - ✓ Complexité : O(1).
  - **Créer une liste depuis un tableau** :
    - ✓ Il suffit de **recopier** les éléments du tableau passé en argument dans le tableau A, puis de **modifier** la valeur de length à n (n la taille du tableau).
    - ✓ Complexité : O(n).

Implémentation contiguë

- **Code C de ces deux opérations** :

```
void createEmptyList(LIST *pL)  {
  pL->length = 0;
}
void createListFromArray(LIST *pL, int t[], int n){
  int i;
  for(i = 0; i < n; i++)
     pL->A[i] = t[i];
  pL->length = n;
}
```


Implémentation contiguë

- **Quelques opérations de type « requête »** :
  - **Tester si une liste est vide** :
    - ✓ Il suffit de tester si length = 0.
    - ✓ Complexité : O(1).
  - **Tester si une liste est saturée (pleine)** :
    - ✓ Il suffit de regarder si length = MAX.
    - ✓ Complexité : O(1).
  - **Accéder au k^{ième} élément d'une liste** :
    - ✓ Il suffit de retourner A[k - 1].
    - ✓ Complexité : O(1).

Implémentation contiguë

- **Code C de ces opérations** :

```
#include "LIST.h"
typedef enum {False, True} Bool;

Bool isEmpty(LIST L)  {
  return (L.length == 0);  }

Bool isFull(LIST L) {
  return (L.length == MAX); }

int getKthElement(LIST L, int k) {
  return L.A[k – 1];  }
```


Implémentation contiguë

- **Opérations d'insertion** :
  - **Insérer un élément à la fin** :
    - ✓ **Algorithme** :
      - **Tester** si la liste n'est pas pleine.
      - Si c'est le cas, **stocker** l'élément « e » dans A[length], puis **incrémenter** length.
    - ✓ **Complexité** : O(1).

Implémentation contiguë

- **Opérations d'insertion** :
  - **Insérer un élément au rang K** :
    - ✓ **Algorithme** :
      - **Tester** si la liste **n'est pas pleine**.
      - Si oui :
        - **Décaler à droite** les éléments A[length], A[length - 1], …, A[k];
        - **Stocker** « e » dans A[k];
        - **Incrémenter** length.
    - ✓ **Complexité** : O(n).

Implémentation contiguë

- **Code C** :

```
void insertAtPositionK(LIST *pL, int e, int k)
{
  if(isFull(*pL) == False)
  {
    int i;
    for(i = pL->length; i > k; i--)
    {  pL->A[i] = pL->A[i - 1]; }
    pL->A[k] = e;
    pL->length++;
  }
}
```


Implémentation contiguë

- **Opérations d'insertion** :
  - **Insérer un élément au début** :
    - ✓ Cette opération est un cas particulier de l'insertion au rang k (k = 0).
    - ✓ **Complexité** : O(n).

Implémentation contiguë

- **Opérations de suppression** :
  - **Supprimer un élément à la fin** :
    - ✓ **Algorithme** :
      - **Tester** si la liste **n'est pas vide**.
      - Si c'est le cas, **décrémenter** length.
    - ✓ **Complexité** : O(1).

Implémentation contiguë

- **Opérations de suppression** :
  - **Supprimer un élément au rang K** :
    - ✓ **Algorithme** :
      - **Tester** si la liste **n'est pas vide**.
      - Si oui :
        - **Décaler à gauche** les éléments A[k+1], A[k+2], …, A[length];
        - **Décrémenter** length.
    - ✓ **Complexité** : O(n).

Implémentation contiguë

- **Opérations de suppression** :
  - **Supprimer un élément au début** :
    - ✓ Cette opération est un cas particulier de la suppression au rang k (k = 0).
    - ✓ **Complexité** : O(n).

Implémentation contiguë

- **Opérations de recherche** :
  - **Recherche séquentielle**
    - ✓ **Algorithme** :
      - **Parcourir** (**traverser**) la liste de gauche à droite (ou de droite à gauche) jusqu'à trouver l'élément « e ».
    - ✓ **Complexité** : O(n).

Implémentation contiguë

- **Opérations de recherche** :
  - **Recherche binaire (dichotomique)**
    - ✓ S'applique uniquement si le tableau est **trié**.
    - ✓ **Algorithme (récursif de type « diviser pour régner »)** :
      - Calculer le milieu ***m*** du tableau A[***d***…***f***].
      - Comparer l'élément recherché avec A[***m***]. Il y a 3 cas:

Implémentation contiguë

- **Opérations de recherche** :
  - **Recherche binaire (dichotomique)**
    - ✓ S'applique uniquement si le tableau est **trié**.
    - ✓ **Algorithme (suite)** :
      - Si ***e*** = A[***m***], on a retourne ***True***.
      - Si ***e*** < A[***m***], on lance la recherche dans le sous-tableau A[***d***...***m*** − 1].
      - Si ***e*** > A[***m***], on lance la recherche dans le sous-tableau A[***m*** + 1...***f***].

Implémentation contiguë

- **Code C de la recherche binaire** :

```
Bool binarySearch(LIST L, int e, int d, int f)  {
  if(d > f)  return False;
  else  {
          int m = (d + f) / 2;
          if(e == L.A[m])  return True;
          else  {
            if(e < L.A[m])
              return binarySearch(L, e, d, m - 1);
            else
              return binarySearch(L, e, m + 1, f);  }
  } }
```


Implémentation contiguë

- **Opérations de recherche** :
  - **Complexité de l'algorithme de la recherche binaire**
    - ✓ À chaque appel, l'algorithme effectue un temps O(1), plus le temps d'un appel récursif avec un sous-tableau dont la taille est la moitié du tableau de l'appel précédent.
    - ✓ T(n) = T(n / 2) + O(1)
    - ✓ **Complexité** : $O(\log_2(n))$.

Implémentation contiguë

- **D'autres opérations nécessitant le parcours d'un** :
  - **Affichage** des éléments
  - **Tester** si le tableau est **trié**
  - **Tri** d'un tableau (**tri par sélection**, **tri par insertion**)
  - **Comptage** du **nombre d'occurrence** d'un élément
  - **Recopie** d'un tableau
  - **Opérations ensemblistes** (**réunion**, **intersection**, et **différence**)
  - …

Représentation chaînée

Représentation chaînée

- **Implémentation chaînée** :
  - La représentation **chaînée** permet de réaliser une **implémentation non contiguë**.
  - Dans la **représentation chaînée**, un élément est appelé un « **nœud** » ou « **cellule** ».
  - Chaque **nœud** est une **structure** composée de deux membres :
    - ✓ Le **premier membre** est utilisé pour **stocker le contenu** d'un **nœud**.
    - ✓ Le **deuxième membre** est utilisé pour **stocker l'adresse** du **nœud suivant**.

Représentation chaînée

- **Implémentation chaînée** :
  - L'existence dans chaque **nœud** d'un **lien** vers le **nœud suivant**, facilite **l'accès séquentiel** aux différents nœuds d'une liste.
  - Il suffit de suivre ces **liens** pour aller d'un **nœud** au **nœud suivant**.

Représentation chaînée

- **Comment réaliser la représentation chaînée** :
  - On peut utiliser des **pointeurs** pour **lier** entre eux les **éléments successifs**.
  - La **liste** est alors déterminée par **l'adresse** de son **premier élément**.
  - Cette représentation nécessite de **l'espace mémoire supplémentaire** pour les **pointeurs**, mais elle permet de traiter facilement la plupart des opérations sur les listes : insertion, suppression, parcours séquentiel, concaténation, se font par une simple manipulation de pointeurs.

Représentation chaînée

- **Types de listes chaînées** :
  - **Liste simplement chaînée** :
    - ✓ Chaque **nœud** possède un **lien** vers le **nœud suivant**.
  - **Liste doublement chaînée** :
    - ✓ Chaque **nœud** possède un **lien** vers le **nœud suivant** et un **lien** vers le **nœud précédent**.
  - **Liste simplement chaînée circulaire** :
    - ✓ **Liste simplement chaînée** dans laquelle le **dernier nœud** possède un **lien** vers le **premier nœud** de la liste.

Représentation chaînée

- **Types de listes chaînées** :
  - **Liste doublement chaînée circulaire** :
    - ✓ **Liste doublement chaînée** dans laquelle :
      - Le **dernier nœud** possède un **lien** vers le **premier nœud** de la liste
      - Le **premier nœud** possède un **lien** vers le **dernier nœud** de la liste.

Représentation chaînée

- **Utilisations des listes chaînées** :
  - Représentation de certains **objets mathématiques**, comme les **polynômes**.
  - Représentation des **grand nombres** (avec des milliers de chiffres).
  - Implémentation des **piles**, des **files d'attente**, des **arbres** et des **graphes**.
  - Représentation de la **table des symboles** d'un **compilateur**.
  - Etc.

Listes simplement chaînées

Listes simplement chaînées

- **Définition** :
  - Une **liste simplement chaînée**, ou brièvement **liste chaînée**, **alloue séparément de l'espace mémoire** pour chaque **nouvel élément** dans un **bloc mémoire propre** appelé un "**nœud**" (ou **cellule**).
  - Chaque **nœud** (***node*** en anglais) contient deux champs :
    - ✓ Un champ "**data**" qui **stocke le contenu** du **nœud**.
    - ✓ Un champ "**next**" qui stocke **l'adresse du nœud suivant** (c'est donc un **pointeur**).

Listes simplement chaînées

- **Caractéristiques** :
  - Chaque **nœud** est **alloué dans le tas** (la **mémoire libre dynamique**) en utilisant la fonction **malloc**().
  - Par conséquent, la **mémoire allouée** au **nœud continue d'exister** jusqu'à sa **libération** explicite via la fonction **free**().
  - La **tête** (**head**) d'une **liste chaînée** est un **pointeur vers le premier nœud** de cette liste.
  - **La liste est complètement connue si l'on dispose de l'adresse de son premier nœud**.

Listes simplement chaînées

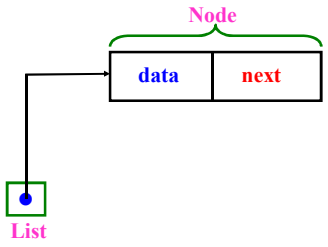
- **Représentation d'une liste chaînée d'entiers en C** :
  - **Définition d'un type « nœud d'une liste chaînée »** :

```
typedef struct node
{
  int data;   /* Ici, les infos sont de type int */
  struct node *next;
} node;
```

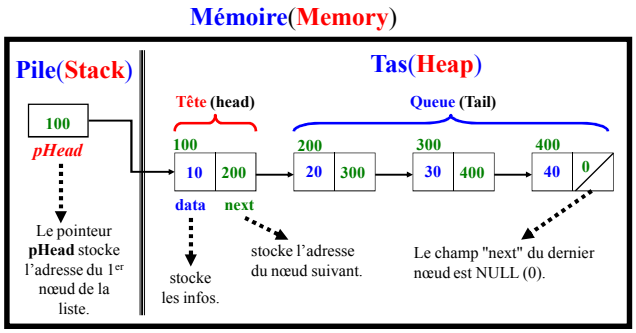
  - **Définition d'un type « liste chaînée »** :

```
typedef node *linkedList;
```


Listes simplement chaînées



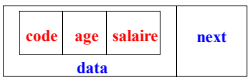
Listes simplement chaînées



Listes simplement chaînées

- **Exercice corrigé** :
  - Définir un type pour représenter des listes d'employés, avec comme information pour chaque employé, son code, son âge et son salaire.

```
typedef struct employe {
  int code;
  int age;
  float salaire;
} employe;
typedef struct node {
  employe data;
  struct node *next;
} node;
typedef node *employeList;
```



Listes simplement chaînées

- **Exercice corrigé** :
  - Définir un type pour représenter des polynômes comme des listes chaînée.

```
typedef struct monome
{
  int deg;       /* le degré d'un monôme */
  float coeff;   /* le coefficient d'un monôme */
  struct monome *next;
} monome;
typedef monome *polynome;
```


Listes simplement chaînées

- **Opérations de bases sur une liste chaînée** :
  - **La création**
  - **L'insertion**
  - **La suppression**
  - **Le parcours**

Listes simplement chaînées

- **Création d'un nœud d'une liste chaînée** :
  - La **création** d'une **liste chaînée** commence par la **création d'un nouveau nœud**.
  - Il faut **allouer suffisamment de mémoire** pour créer un **nœud** en utilisant la fonction **malloc**().
  - La fonction "**getNode**()" sera utilisée pour créer un **nouveau nœud**.
  - En **cas d'échec** de **malloc**, la fonction retournera la valeur **NULL**.
  - Sinon, elle retournera un **pointeur vers le nœud crée**.

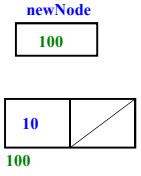
Listes simplement chaînées

- **Création d'un nœud d'une liste chaînée** :
  - **Algorithme** :
    - ✓ **Allouer** de la mémoire pour le **nouveau nœud** (la structure de type "**node**");
    - ✓ **Lire** le **contenu** du **nouveau nœud** dans le champ "**data**";
    - ✓ **Mettre** la valeur **NULL** au champ "**next**" du **noeud crée**.
    - ✓ **Retourner** un **pointeur vers le nœud crée**.
  - **Complexité** : O(1)

Listes simplement chaînées

- **Code C de la création d'un nœud :**

```
node *getNode( )
{
  node *newNode;
  newNode = (node*) malloc(sizeof(node));
  if(newNode != NULL)      {
      printf("Enter a new data : ");
      scanf("%d", &newNode->data);
      newNode->next = NULL;  }
  else
     puts("Espace Memoire insuffisant\n");
  return newNode;
}
```



Listes simplement chaînées

- **Création d'une liste chaînée :**
  - La fonction "**createLinkedList(int** n)" va **créer** une **liste chaînée** contenant « n » **nœuds** (la valeur de n est supposée non nulle).
  - Chaque **nœud**, sera crée en appelant la fonction "**getNode**()" précédemment définie.

Listes simplement chaînées

- **Algorithme de création d'une liste chaînée :**
  - **Créer** le **premier nœud** en faisant appel à "**getNode**()" et **affecter** son **adresse** à **la liste**.
  - **Assigner l'adresse** de la **tête de la liste** au **pointeur** désignant le **nœud courant**.
  - **Répéter** (n - 1) fois, les étapes suivantes :
    - ✓ **Créer** un **nouveau nœud** via "**getNode**()" et **assigner son adresse** au **nœud suivant** du **pointeur** désignant le **nœud courant**.
    - ✓ **Avancer** le pointeur désignant le **nœud courant** (y mettre l'adresse du **nœud suivant**).
  - **Retourner la liste** ainsi créée.

Listes simplement chaînées

- **Code C de la création d'une liste chaînée :**

```
linkedList createLinkedList(int n)
{
  int i;   /* un compteur */
  node *currentNode;  /* le nœud courant */
  linkedList pHead;    /* la liste chaînée à créer */
  /* création du premier nœud de la liste */
  pHead = getNode();
  /* la liste est actuellement formée de son premier
     nœud  */
  currentNode = pHead;
```


Listes simplement chaînées

- **Code C de la création d'une liste chaînée :**

```
  for(i = 1; i < n; i++)
  {
    /* création des (n - 1) nœuds restants */
    currentNode->next = getNode();
    /* création plus chaînage */
    currentNode = currentNode->next; /* avancement */
  }
  return pHead;
  /* pHead pointe toujours vers le premier nœud de la liste */
}
```


Listes simplement chaînées

- **Exercice corrigé :**
  - Écrire un programme qui crée une liste d'entiers de longueur 4.

```
/* Déclaration des types de données */
/* Déclarations & définitions des fonctions */
/* Programme principal */
main( )
{
  linkedList L;
  L = createLinkedList(4);
  system("pause");
}
```

Listes simplement chaînées

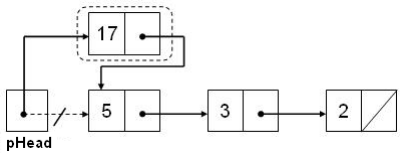
- **Insertion d'un nœud** :
  - L'espace mémoire doit être **alloué** pour le nouveau nœud **avant la lecture des données**.
  - Les données sont ensuite **lues** et **stockées** dans le champ "**data**" du nouveau nœud.
  - Le champ du lien "**next**" du nœud crée est initialisé à **NULL**.
  - Le nœud ainsi crée peut être inséré :
    - ✓ Au **début de la liste;**
    - ✓ À une **place intermédiaire**;
    - ✓ À la **fin de la liste**.

Listes simplement chaînées

- **Insertion d'un nœud au début de la liste** :
  - **Algorithme** :
    - ✓ **Obtenir** le **nouveau nœud** en utilisant la fonction "**getNode**()".
    - ✓ **Assigner l'adresse** de la **tête de la liste** au champ "**next**" du **nouveau nœud**.
    - ✓ **Assigner l'adresse** du **nouveau nœud** au **pointeur pointant** vers la **tête de la liste** (***pHead***).
  - **Complexité** : O(1)

Listes simplement chaînées

- **Insertion d'un nœud au début de la liste** :



Listes simplement chaînées

- **Code C de l'insertion d'un nœud au début de la liste** :

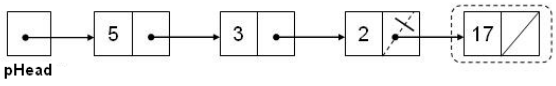
```
void insertAtBeginning(linkedList *pL)  {
  node *newNode = getNode();
  if(newNode != NULL)   {
       newNode->next = *pL;
       *pL = newNode;      }
  else
    puts("Espace Memoire Insuffisant\n");
}
```


Listes simplement chaînées

- **Insertion d'un nœud à la fin de la liste** :
  - **Algorithme** (**itératif**) :
    - ✓ **Obtenir** le **nouveau nœud** en utilisant la fonction "**getNode**()".
    - ✓ Si la **liste** est **vide**, **Assigner l'adresse** du **nouveau nœud** à la **tête de la liste** (***pHead***).
    - ✓ Sinon, **parcourir la liste** jusqu'à son **dernier nœud**, puis **assigner l'adresse** du **nouveau nœud** au champ "**next**" du **dernier nœud de la liste**.
  - **Complexité** : O(n)

Listes simplement chaînées

- **Insertion d'un nœud à la fin de la liste** :



Listes simplement chaînées

- **Code C de l'insertion d'un nœud à la fin de la liste :**

```
void insertAtEnd(linkedList *pL)
{
  node *newNode = getNode();
  if(newNode != NULL)
  {
    if(*pL == NULL)
      *pL = newNode;
    else    {
```


Listes simplement chaînées

- **Code C de l'insertion d'un nœud à la fin de la liste :**

```
        node *current = *pL;
        while(current->next != NULL)
              current = current->next;
        current->next = newNode;    }
    }
    else
      puts("Espace Memoire Insuffisant\n");
}
```


Listes simplement chaînées

- **Insertion d'un nœud à la fin de la liste :**
  - **Algorithme (récursif) :**
    - ✓ Si la **liste** est **vide**, créer un **nouveau noeud**, puis **assigner son adresse** à la **tête de la liste**.
    - ✓ Sinon, appliquer **récursivement** la procédure sur le pointeur "**next**" de la tête de la liste.
  - **Complexité :**
    - ✓ $T(n) = T(n-1) + O(1) = O(n)$

Listes simplement chaînées

- **Code C de l'insertion d'un nœud à la fin de la liste :**

```
void insertAtEndRec(linkedList *pL)
{
  if(*pL == NULL)
    *pL = getNode();
  else
    insertAtEndRec(&(*pL)->next);
}
```


Listes simplement chaînées

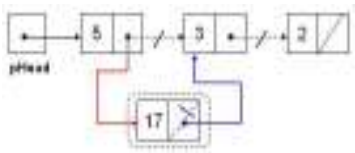
- **Insertion d'un nœud à une position intermédiaire de la liste :**
  - **Algorithme (cas général : 1 < k < N) :**
    - ✓ **Obtenir** le **nouveau nœud** en appelant la fonction "**getNode**()".
    - ✓ **Parcourir** la **liste** jusqu'à **atteindre la position k**, en utilisant deux pointeurs :
      - "**previous**" qui mémorisera le **noeud précédent**.
      - "**current**" qui mémorisera le **nœud courant**.
    - ✓ **Assigner** au pointeur "**next**" du **nœud pointé** par "**previous**", **l'adresse** du **nouveau nœud**.
    - ✓ **Assigner** au pointeur "**next**" du **nouveau nœud**, **l'adresse** du **nœud pointé** par "**current**".

Listes simplement chaînées

- **Insertion d'un nœud à une position intermédiaire de la liste :**
  - **Cas spéciaux :**
    - ✓ Si la liste a une **longueur** inférieure à 3, alors la procédure échoue pour toute valeur du paramètre k.
    - ✓ Sinon, la procédure échoue pour les valeurs de k égales à 1 ou N.
  - **Complexité** : O(n)

Listes simplement chaînées

- **Insertion d'un nœud à une position intermédiaire de la liste** :



Listes simplement chaînées

- **Code C de l'insertion d'un nœud à une position intermédiaire de la liste** :

```
void inseretAtIntermedPos(linkedList *pL, int k)
{
  if(*pL == NULL || (*pL)->next == NULL ||
    (*pL)->next->next == NULL)
    puts("Impossible Operation for all k\n");
  else
  {
    if(k == 1)
```


Listes simplement chaînées

- **Code C de l'insertion d'un nœud à une position intermédiaire de la liste** :

```
    puts("Invalide Position\n");
  else  {
      node *curr = *pL, *prev;
      int pos = 1;
      while(pos < k && curr->next != NULL)
      {
            prev = curr;
            curr = curr->next;
            pos++;
      }
```


Listes simplement chaînées

- **Code C de l'insertion d'un nœud à une position intermédiaire de la liste** :

```
      if(pos == k && curr->next != NULL)
      {
        node *newNode = getNode();
        prev->next = newNode;
         newNode->next = curr;
      }
      else
       puts("Invalide Position\n");   }
}
}
```


Listes simplement chaînées

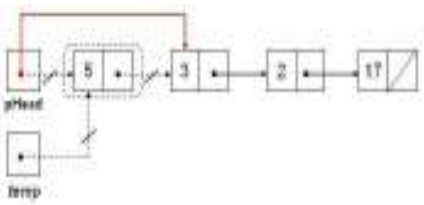
- **Suppression d'un nœud** :
  - La **mémoire occupée par le nœud** à supprimer doit être **restituer** (**libérer**) au système via la fonction "**free**()".
  - Généralement, un nœud peut être supprimé :
    - ✓ En **fin** de liste;
    - ✓ Au **début** de la liste;
    - ✓ En une **position intermédiaire** de la liste.

Listes simplement chaînées

- **Suppression d'un nœud au début de la liste** :
  - **Algorithme** :
    - ✓ Si la liste est **vide**, il n'y a **rien à faire**.
    - ✓ Sinon :
      - **On sauve le premier nœud** de la liste dans une variable temporaire "temp".
      - **Assigner l'adresse du deuxième nœud** au **pointeur vers la tête** de la liste.
      - **Libérer** la mémoire occupée par "temp".
  - **Complexité** : O(1).

Listes simplement chaînées

- **Suppression d'un nœud au début de la liste** :



Listes simplement chaînées

- **Code C de la suppression d'un nœud au début de la liste** :

```
void deleteAtBeginning(linkedList *pL)
{
  if(*pL != NULL)
  {
    node *temp = *pL;
    *pL = (*pL)->next;
    free(temp);
  }
}
```


Listes simplement chaînées

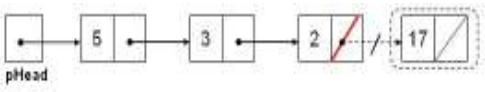
- **Suppression d'un nœud à la fin de la liste** :
  - **Algorithme** :
    - ✓ Si la liste est **vide**, ne rien faire.
    - ✓ Sinon :
      - Si la liste **contient un seul élément**, Affecter **NULL** à **l'adresse de la tête** de la liste.
      - Sinon :
        - **Parcourir** la liste en utilisant deux pointeurs pour **accéder** au **dernier nœud** et à **l'avant dernier nœud** de la liste.

Listes simplement chaînées

- **Suppression d'un nœud à la fin de la liste** :
  - **Assigner** la valeur **NULL** au **champ "next"** de **l'avant dernier nœud**.
  - **Libérer** la mémoire allouée par le **dernier nœud**.
- **Complexité** : O(n)

Listes simplement chaînées

- **Suppression d'un nœud à la fin de la liste** :



Listes simplement chaînées

- **Code C de la suppression d'un nœud à la fin de la liste** :

```
void deleteAtEnd(linkedList *pL)  {
  if(*pL != NULL)     {
    if((*pL)->next == NULL)  *pL = NULL;
    else {
      node *prev, *curr = *pL;
      while(curr->next != NULL)  {
            prev = curr; curr = curr->next;  }
      prev->next = NULL;   free(curr); }
}}
```

Listes simplement chaînées

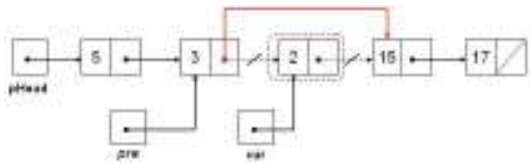
- **Suppression d'un nœud à une position intermédiaire de la liste** :
  - **Algorithme (cas général : 1 < k < N)** :
    - ✓ **Parcourir** la **liste** jusqu'à **atteindre la position k**, en utilisant deux pointeurs :
      - "**previous**" qui mémorisera le **noeud précédent**.
      - "**current**" qui mémorisera le **nœud courant**.
    - ✓ **previous->next** = **current->next**;
    - ✓ **Libérer la mémoire** allouée au nœud pointé par "**current**".

Listes simplement chaînées

- **Suppression d'un nœud à une position intermédiaire de la liste** :
  - **Algorithme (cas spéciaux)** :
    - ✓ Si la liste a une **longueur** inférieure à 3, alors la procédure échoue pour toute valeur du paramètre k.
    - ✓ Sinon, la procédure échoue pour les valeurs de k égales à 1 ou N.
  - **Complexité** : O(n)

Listes simplement chaînées

- **Suppression d'un nœud à une position intermédiaire de la liste** :



Listes simplement chaînées

- **Code C de la suppression d'un nœud à une position intermédiaire de la liste** :

```
void deleteAtIntermedPos(linkedList *pL, int k)
{
  if(*pL == NULL || (*pL)->next == NULL ||
    (*pL)->next->next == NULL)
    puts("Impossible Operation for all k\n");
  else
  {
    if(k == 1)
```


Listes simplement chaînées

- **Code C de la suppression d'un nœud à une position intermédiaire de la liste** :

```
    puts("Invalide Position\n");
  else  {
      node *curr = *pL, *prev;
      int pos = 1;
      while(pos < k && curr->next != NULL)
      {
            prev = curr;
            curr = curr->next;
            pos++;
      }
```


Listes simplement chaînées

- **Code C de la suppression d'un nœud à une position intermédiaire de la liste** :

```
      if(pos == k && curr->next != NULL)
      {
        prev->next = curr->next;
        free(curr);
      }
      else
        puts("Invalide Position\n");  }
}
}
```

Listes simplement chaînées

- **Parcours séquentiel d'une liste chaînée** :
  - **Utilisations** :
    - ✓ **Rechercher** un élément d'une liste;
    - ✓ Effectuer un **même traitement** à tous ou à certains éléments d'une liste : affichage, somme, produit, …etc.
    - ✓ **Trier** une liste;
    - ✓ **Extraire** une partie d'une liste;
    - ✓ Calculer le **nombre d'occurrence** d'un élément dans une liste;
    - ✓ …etc

Listes simplement chaînées

- **Type de parcours d'une liste chaînée** :
  - **Parcours "gauche-droite"** :
    - ✓ On commence par le premier élément de la liste, puis le deuxième élément, jusqu'au dernier élément.
  - **Parcours "droite-gauche" (sens inverse)** :
    - ✓ On commence par le dernier élément de la liste, puis l'avant dernier élément, jusqu'au premier élément.

Listes simplement chaînées

- **Exercices corrigés** :
  - Écrire de manière itérative et récursive une fonction qui **recherche** un élément dans une liste.
  - Écrire de manière itérative et récursive une fonction qui calcule le **nombre d'occurrence** d'un élément dans une liste.
  - Écrire de manière itérative et récursive une fonction qui **met au carré** tous les éléments d'une liste.

Listes simplement chaînées

- **Recherche d'un élément dans une liste chaînée** :

```
Bool search(linkedList L, int e)    {
  if(L == NULL)
    return False;
  else {
        node *temp = L;
        while(temp != NULL) {
           if(temp->data == e)
             return True;
           else
             temp = temp->next;  } /* end while */
  return False; } }
```


Listes simplement chaînées

- **Recherche d'un élément dans une liste chaînée** :

```
Bool searchRec(linkedList L, int e)
{
  if(L == NULL)
    return False;
  if(L->data == e)
    return True;
  else
    return searchRec(L->next, e);
}
```


Listes simplement chaînées

- **Nombre d'occurrence d'un élément dans une liste chaînée** :

```
int nbOccRec(linkedList L, int e)
{
  if(L == NULL)
    return 0;
  else
    return
        nbOccRec(L->next, e) + (L->data == e);
}
```

Listes simplement chaînées

- **Nombre d'occurrence d'un élément dans une liste chaînée :**

```
int nbOcc(linkedList L, int e) {
  if(L == NULL)
    return 0;
  else {
        node *temp = L;
        int n = 0;
        while(temp != NULL) {
           if(temp->data == e)   n++;
           temp = temp->next;  } /* end while */
  return n; } }
```


Listes simplement chaînées

- **Mettre au carré tous les éléments d'une liste chaînée :**

```
void square(linkedList *pL) {
  if(*pL != NULL)
  {
    node *temp = *pL;
    while(temp != NULL)  {
      temp->data *= temp->data;
      temp = temp->next;
  }
}
```


Listes simplement chaînées

- **Mettre au carré tous les éléments d'une liste chaînée :**

```
void squareRec(linkedList *pL)
{
  if(*pL != NULL)
  {
    (*pL)->data *= (*pL)->data;
    squareRec(&(*pL)->next);
  }
}
```


Listes simplement chaînées

- **Affichage à l'envers des éléments d'une liste chaînée :**

```
void displayRev(linkedList L)
{
  if(L != NULL)
  {
    displayRev(L->next);
    printf("%d -> ", L->data);
  }
}
```


Listes doublement chaînées

Listes doublement chaînées

- **Définition :**
  - Une **liste doublement chaînée** (ou encore une **liste bidirectionnelle**) est une **liste chaînée** dans les **deux sens**, c'est-à-dire que chaque **nœud** contient deux **liens** : un **lien vers le nœud précédent** et un lien **vers le nœud suivant**.
  - Mais :
    - ✓ Le **premier nœud** n'a pas de **prédécesseur**;
    - ✓ Le **dernier nœud** n'a pas de **successeur**.

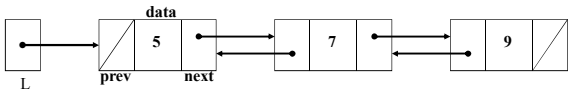
Listes doublement chaînées

- **Codage en C d'une liste doublement chaînée d'entiers** :

```
typedef struct node
{
  struct node *prev; /* lien vers le nœud précédent */
  int data;  /* champ information */
  struct node *next; /* lien vers le nœud suivant */
} node;
typedef node *dblList;
```


Listes doublement chaînées

- **Liste doublement chaînée d'entiers** :



Listes doublement chaînées

- **Opérations** :
  - Les **opérations de type dictionnaire** (**insertion**, **suppression** et **recherche**) sur une **liste doublement chaînée** sont principalement les mêmes que celles sur une **liste simplement chaînée**.

Listes doublement chaînées

- **Création d'un nœud d'une liste bidirectionnelle**:

```
node *getNode(int val)  {
  node *newNode;

  newNode = (node*) malloc(sizeof(node));
  newNode->data = val;
  newNode->prev = NULL;
  newNode->next = NULL;
  return newNode;
}
```


Listes doublement chaînées

- **Insertion au début d'une liste bidirectionnelle** :

```
Liste_Double head;
Node * newNode;
Element x;
  /* Insertion au début */
newNode = getNode(x);
If (head != NULL)
{
    head->prev = newNode;
    newNode->next = head;
}
head = newNode;
```


Listes doublement chaînées

- **Insertion à la fin d'une liste bidirectionnelle** :

```
Liste_Double head;
Node * newNode, *temp;
Element x;
   /* Insertion à la fin */
newNode = getNode(x);
If (head == NULL)
  head = newNode;
Else
    For (temp = head; temp->next != NULL; temp = temp-
    >next)  ;  /* boucle pour atteindre le dernier nœud */
    temp->next = newNode;
    newNode->prev = temp;
```

Listes doublement chaînées

- **Insertion à une position intermédiaire d'une liste bidirectionnelle** :
  - **Obtenir** le **nouveau nœud** en utilisant "**getNode**()" :
    - ✓ *newNode = getNode(x);*
  - **Vérifier** que la **position spécifiée** est **entre** le **premier nœud** et le **dernier nœud**.
  - Sinon, **signaler** une **erreur de position**.
  - Si oui, **parcourir** la **liste** jusqu'à **atteindre la position désirée**.

Listes doublement chaînées

- **Insertion à une position intermédiaire d'une liste bidirectionnelle** :
  - **Exécuter** ensuite les **instructions** suivantes :
    - ✓ *newNode->prev = temp->prev;*
    - ✓ *newNode->next = temp;*
    - ✓ *temp->prev->next = newNode;*
    - ✓ *temp->prev = newNode;*

Listes doublement chaînées

- **Suppression au début d'une liste bidirectionnelle** :

```
Liste_Double head;
Node * newNode, *temp;
  /* Suppression au début */
If (head != NULL)   {
    temp = head;
    head = head->next;
    If (head != NULL)
        head->prev = NULL;
    free(temp);
}
```


Listes doublement chaînées

- **Suppression à la fin d'une liste bidirectionnelle** :

```
Liste_Double head;
Node * newNode, *temp;
  /* Suppression à la fin*/
If (head != NULL)   {
If (head->next == NULL)   { /* la liste a un seul élément */
   free(head);
  head = NULL;              }
Else {/* la liste a plusieurs éléments */
        temp = head;
        While (temp->next != NULL)
                temp = temp->next;
        temp->prev->next = NULL;
        free(temp); } }
```


Listes doublement chaînées

- **Suppression à une position intermédiaire d'une liste bidirectionnelle** :

```
int l = getLength(head);   /* calculer la longueur de la liste */
if(pos <= 1 || pos >= l)
  printf("Invalid position\n");
else   {
            node *temp = head;
            int k = 1;
            while(k < pos)
            { temp = temp->next;   k++; }
            temp->next->prev = temp->prev;
            temp->prev->next = temp->next;
            free(temp);
      }
```


Listes doublement chaînées

- **Suppression d'un nœud sachant son adresse d'une liste bidirectionnelle** :

```
void deleteNode(dblList *pdL, node *p)   {
  if(p->next != NULL)
        p->next->prev = p->prev;
  if(p->prev == NULL)
        *pdL = p->next;
  else
        p->prev->next = p->next;
  free(p);  }
```

Complexité : O(1)

Listes doublement chaînées

- **Explication** :
  - On vérifie que ***p*** ne pointe pas vers le dernier nœud par le test :
    - ✓ ***if(p->next != NULL)***
  - Si tel n'est pas le cas, alors on fait pointer le pointeur précédent du nœud suivant vers le nœud précédent ***p*** :
    - ✓ ***p->next->prev = p->prev;***
  - On teste si ***p*** est le premier nœud :
    - ✓ ***if(p->prev == NULL)***

Listes doublement chaînées

- **Explication** :
  - Si tel est le cas, alors on fait pointer ***pdL** vers le nœud suivant ***p*** :
    - ✓ ****pdL = p->next;***
  - Sinon, on fait pointer le pointeur suivant du nœud précédent vers le nœud suivant ***p*** :
    - ✓ ***p->prev->next = p->next;***
  - À la fin, on libère la mémoire occupée par ***p*** :
    - ✓ ***free(p);***

Listes doublement chaînées

- **Remarque** :
  - L'**opération** de **suppression** d'un nœud d'une **liste simplement chaînée** nécessite le **parcours de la liste** depuis le **début** pour **trouver le nœud précédent** : c'est donc une opération en **O(n).**

Listes doublement chaînées

- **Parcours « gauche » d'une liste bidirectionnelle**:

```
Liste_Double head;
Node * newNode, *temp;
  /* Parcours gauche */
temp = head;
while (temp != NULL)
{
  /* Traiter « temp » */
    temp = temp->next;  /* Avancer */
}
```


Listes doublement chaînées

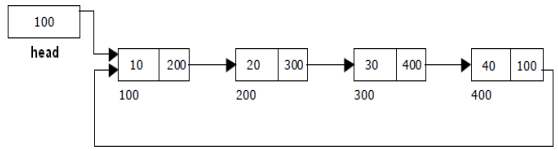
- **Parcours « droit » d'une liste bidirectionnelle**:

```
Liste_Double head;
Node * newNode, *temp;
  /* Parcours droit */
temp = head;
while (temp->next != NULL)
      temp = temp->next;  /* Aller jusqu'au bout */
while(temp != NULL)  {
  /* Traiter « temp » */
  temp = temp->prev;   /* Reculer */
}
```


Listes chaînées circulaires

Listes circulaires

- **Définition** :
  - Une **liste chaînée circulaire** est une **liste simplement chaînée** telle que le **dernier nœud** possède un **lien** vers le **premier nœud**.

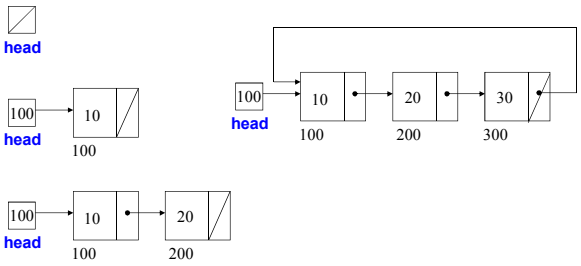


Listes circulaires

- **Création d'une liste circulaire contenant n nœuds** :
  - Obtenir le nouveau nœud en utilisant "**getNode**()" :
    - ✓ *newNode = getNode(x);*
  - Si la liste est vide, affecter le nouveau nœud à *head* :
    - ✓ *head = newNode;*
  - Si la liste n'est pas vide, exécuter les instructions suivantes :
    - ✓ *temp = head;*
    - ✓ *While (temp->next != NULL)*
      - ■ *temp = temp->next;*
    - ✓ *temp->next = newNode;*
  - Répéter les instructions précédentes « n » fois;
  - *newNode->next = head; // bouclage*

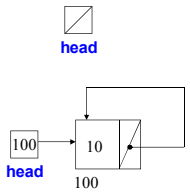
Listes circulaires

- **Création d'une liste circulaire contenant 3 nœuds** :



Listes circulaires

- **Création d'une liste circulaire contenant un seul noeud** :



Listes circulaires

- **Insertion au début d'une liste circulaire** :
  - Obtenir le nouveau nœud en utilisant "**getNode**()" :
    - ✓ *newNode = getNode(x);*
  - Si la liste est vide, affecter le nouveau nœud à *head* :
    - ✓ *head = newNode;*
    - ✓ *newNode->next = head; // bouclage*
  - Si la liste n'est pas vide, exécuter les instructions suivantes :
    - ✓ *last = head;*
    - ✓ *While (last->next != head) /* et non != NULL */*
      - ■ *last = last->next;*
    - ✓ *newNode->next = head;*
    - ✓ *head = newNode;*
    - ✓ *last->next = head; // bouclage*

Listes circulaires

- **Insertion à la fin d'une liste circulaire** :
  - Obtenir le nouveau nœud en utilisant "**getNode**()" :
    - ✓ *newNode = getNode(x);*
  - Si la liste est vide, affecter le nouveau nœud à *head* :
    - ✓ *head = newNode;*
    - ✓ *newNode->next = head; // bouclage*
  - Si la liste n'est pas vide, exécuter les instructions suivantes :
    - ✓ *last = head;*
    - ✓ *While (last>next != head)*
      - ■ *last = last->next;*
    - ✓ *last->next = newNode;*
    - ✓ *newNode->next = head;*

Listes circulaires

- **Suppression au début d'une liste circulaire** :
  - Si la liste est vide, ne rien faire.
  - Si la liste contient un seul élément :
    - ✓ *If (head->next == head)*
      - *free(head);*
      - *head = NULL;*
  - Sinon (la liste contient au moins deux éléments) :
    - ✓ *last = first = head;*
    - ✓ *While (last->next != head)*
      - *last = last->next;*
    - ✓ *head = head->next;*
    - ✓ *last->next = head;*
    - ✓ *free(first);*

Listes circulaires

- **Suppression à la fin d'une liste circulaire** :
  - Si la liste n'est pas vide, exécuter les instructions suivantes :
    - ✓ *last = beforeLast = head;*
    - ✓ *While (last->next != head)*
      { beforeLast = last; last = last->next; }
    - ✓ *beforeLast->next = head;*
    - ✓ *free(last);*
  - Après la suppression du nœud, si la liste est vide, exécuter l'instruction suivante :
    - ✓ *head = NULL;*

Listes circulaires

- **Parcours d'une liste circulaire (affichage comme exemple)** :

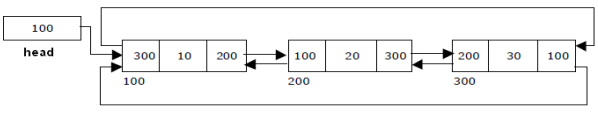
```
temp = head;
do
{
  printf("%d", temp->data);
  temp = temp->next;
} while(temp != head);
```


Listes doublement chaînées circulaires

Listes bidirectionnelles circulaires

- **Définition** :
  - Une **liste doublement chaînée circulaire** (ou liste **bidirectionnelle circulaire**) est une lise chaînée qui vérifie les conditions suivantes:
    - ✓ **Tout nœud** possède un **lien** vers le **nœud précédent** et un **lien** vers le **nœud suivant**.
    - ✓ Le **premier nœud** a pour **prédécesseur** le **dernier nœud** de la liste.
    - ✓ Le **successeur** du **dernier nœud** est le premier **nœud de la liste**.

Listes bidirectionnelles circulaires



Listes bidirectionnelles circulaires

- **Création d'une liste bidirectionnelle circulaire ayant « n » nœuds** :
  - Répéter « n » fois :
    - ✓ Obtenir le nouveau nœud via "**getNode**()" :
      - ■ *newNode = getNode(x);*
    - ✓ Si la liste est vide :
      - ■ *head = newNode;*
      - ■ *newNode->prev = newNode->next = head;*
    - ✓ Sinon :
      - ■ *newNode->prev = head->prev;*
      - ■ *newNode->next = head;*
      - ■ *head->prev->next = newNode;*
      - ■ *head->prev = newNode;*

Listes bidirectionnelles circulaires

- **Insertion au début d'une liste bidirectionnelle circulaire** :
  - Obtenir le nouveau nœud via "**getNode**()" :
    - ✓ *newNode = getNode(x);*
  - Si la liste est vide :
    - ✓ *head = newNode;*
    - ✓ *newNode->prev = newNode->next = head;*
  - Sinon :
    - ✓ *newNode->prev = head->prev;*
    - ✓ *newNode->next = head;*
    - ✓ *head->prev->next = newNode;*
    - ✓ *head->prev = newNode;*
    - ✓ *head = newNode;*

Listes bidirectionnelles circulaires

- **Insertion à la fin d'une liste bidirectionnelle circulaire** :
  - Obtenir le nouveau nœud via "**getNode**()" :
    - ✓ *newNode = getNode(x);*
  - Si la liste est vide :
    - ✓ *head = newNode;*
    - ✓ *newNode->prev = newNode->next = head;*
  - Sinon :
    - ✓ *newNode->prev = head->prev;*
    - ✓ *newNode->next = head;*
    - ✓ *head->prev->next = newNode;*
    - ✓ *head->prev = newNode;*

Listes bidirectionnelles circulaires

- **Insertion à une position intermédiaire d'une liste bidirectionnelle circulaire** :
  - Obtenir le nouveau nœud via "**getNode**()" :
    - ✓ *newNode = getNode(x);*
  - Vérifier que la position spécifiée est entre le premier nœud et le dernier nœud de la liste.
  - Sinon, signaler une erreur de position.
  - Parcourir la liste jusqu'à atteindre la position désirée.
  - Exécuter les instruction suivantes :
    - ✓ *newNode->prev = temp;*
    - ✓ *newNode->next = temp->next;*
    - ✓ *temp->next->prev = newNode;*
    - ✓ *temp->next = newNode;*

Listes bidirectionnelles circulaires

- **Suppression au début d'une liste bidirectionnelle circulaire** :
  - Si la liste n'est pas vide :
    - ✓ *temp = head;*
    - ✓ *head = head->next;*
    - ✓ *temp->prev->next = head;*
    - ✓ *head->prev = temp->prev;*
    - ✓ *free (temp);*

Listes bidirectionnelles circulaires

- **Suppression à la fin d'une liste bidirectionnelle circulaire** :
  - Si la liste n'est pas vide :
    - ✓ *temp = head;*
    - ✓ *While (temp->right != head)*
      - ■ *temp = temp->right;*
    - ✓ *temp->prev->next = temp->next;*
    - ✓ *temp->next->prev = temp->prev;*
    - ✓ *free (temp);*

Listes bidirectionnelles circulaires

- **Suppression à une positon intermédiaire d'une liste bidirectionnelle circulaire** :
  - **Vérifier** que la **position spécifiée** est **entre** le **premier nœud** et le **dernier nœud**.
  - Sinon, **signaler** une **erreur de position**.
  - Si oui, **parcourir** la **liste** jusqu'à **atteindre la position désirée**.
  - **Exécuter** ensuite les **instructions** suivantes :
    - ✓ ***temp->prev->prev = temp->prev;***
    - ✓ ***temp->prev->next = temp->next;***
    - ✓ ***free(temp);***

Listes bidirectionnelles circulaires

- **Parcours d'une liste bidirectionnelle circulaire** :

```
temp = head;
printf ("%d\n", temp->data);
temp = temp->next;
while(temp != head)
{
    printf ("%d\n", temp->data);
    temp = temp->next;
}
```