MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE
Université Hassiba Benbouali de Chlef

Faculté des sciences exactes et informatique

Département de TC en sciences exactes et informatique
1ière année licence S2

Partie I : Structures séquentielles « Listes »

https://goo.gl/J4R515

Les structures de données spécifient la manière de **représenter les données** d'un problème qui peut être résolu par ordinateur à l'aide d'un **algorithme**. Le choix d'une structure de données doit prendre en considération la taille mémoire nécessaire à son implémentation ainsi que sa facilité d'accès. Une structure de données peut être choisie indépendamment du langage de programmation qui est utilisé pour l'écriture du programme manipulant ses données. Ce langage est supposé offrir, d'une façon ou d'une autre, les mécanismes nécessaires pour définir et manipuler ces structures de données. Les langages de programmation modernes permettent d'attribuer et manipuler la mémoire disponible de la machine sous forme de variables isolées les unes des autres, sous forme de tableaux, de pointeurs indiquant la localisation dans la mémoire de l'objet qui nous intéresse ou à l'aide de structures prédéfinies plus complexes.
Dans le cas des variables isolées ou des tableaux statiques, la place en mémoire est attribuée par le compilateur et ne peut faire l'objet de modification pendant l'exécution du programme. Cependant dans le cas de tableaux dynamiques ou listes chaînées, l'allocation de la mémoire nécessaire est effectuée pendant l'exécution et par conséquent, elle peut augmenter ou diminuer selon le besoin.

Les structures de données classiques peuvent être classées en trois catégories distinctes :

- les **structures linéaires** ou **séquentielles** : Ce sont les structures qui peuvent être représentées par des listes linéaires telles que les tableaux ou des listes chaînées ayant une seule dimension. Dans cette catégorie, il y a lieu de citer les **piles**, pour lesquelles les données peuvent être **ajoutées ou supprimées** à partir d'une même extrémité ; et les **files** où les données peuvent être **ajoutées à partir d'une extrémité** tandis qu'elles sont **supprimées de l'autre extrémité**.
- les **structures arborescentes** et en particulier les arbres binaires.
- les **structures relationnelles** qui prennent en compte des relations existant ou non entre les entités qu'elles décrivent.

Les structures de données séquentielles ou linéaires sont appelées ainsi parce que les données sont organisées sous forme d'une liste les unes derrières les autres. Elles peuvent être représentées à l'aide d'un tableau ou sous forme d'une liste chaînée.

I. LISTES

Une liste est une suite ordonnée d'éléments d'un type donné et elle peut contenir zéro, un ou plusieurs éléments.
Exemples :

- (3, 5, 7, 9, 12) est une liste d'entiers
- (Printemps, Eté, Automne, Hiver) est la liste des saisons de l'année.

Le nombre d'éléments d'une liste est appelé **longueur de la liste**. Par conséquent, une liste vide est de longueur nulle. Le premier élément de la liste est appelé la **tête de la liste**. **La queue d'une liste** est définie comme étant son dernier élément pour certains ou bien la liste des éléments obtenue par la suppression de la tête de la liste, par d'autres. Les listes dont les éléments peuvent être ordonnés (par une relation d'ordre totale) sont les listes les plus utilisées en pratique telles que les entiers, chaînes de caractères, etc.

I.1. Opérations sur les listes

Il y a lieu de citer les quatre opérations classiques suivantes qu'on peut effectuer sur les listes :

- **Insertion** : Il s'agit d'insérer un élément au début, à la fin ou bien entre deux éléments consécutifs de la liste.
- **Suppression** : Il s'agit de supprimer un élément quelconque de la liste,
- **Recherche** : Il s'agit de renvoyer la valeur vrai ou faux selon qu'un élément donné, appelé clé de recherche, existe ou non dans la liste. Cette opération peut aussi renvoyer l'adresse de l'élément recherché sous forme d'un indice d'un tableau ou d'un pointeur.
- **Concaténation** : Il s'agit de construire une liste en juxtaposant les éléments de deux listes distinctes.

I.2. Implémentation d'une liste à l'aide d'un tableau

Il s'agit de représenter une liste par un tableau dont l'indice de début est fixé à 0 ou 1. Les éléments sont ainsi stockés dans des cellules (du tableau) contiguës. Dans cette forme de représentation, une liste peut être traversée et de nouveaux éléments peuvent être supprimés de, ou ajoutés à, la fin de la liste (à condition qu'on sache le nombre d'éléments qu'elle contient et pourvu qu'il n'y ait pas de dépassement de la taille maximale déclarée).

L'insertion d'un élément au milieu de la liste peut être problématique, car on doit prendre soin de décaler tous les éléments suivant la position d'insertion (vers la queue) pour pouvoir placer le nouvel élément.

La suppression d'un élément du milieu de la liste nécessite aussi le déplacement des éléments (vers la tête) à partir de la position de suppression.

Dans cette forme d'implémentation, on définit le type **Liste** comme étant une structure (struct C/C++) ou enregistrement (record en Pascal) ayant deux champs. Le premier est un tableau d'éléments du même type que celui des éléments de la liste dont la taille est fixée au maximum d'éléments que la liste peut en avoir. Le second champ est un entier qui représente la position du dernier élément de la liste. Naturellement, il y a une correspondance entre l'ième élément de la liste et l'ième élément du tableau. Les détails d'implémentation d'une liste sous forme d'un tableau en langage C/C++ sont comme suit :

Déclaration

```
const int longMax=100;
struct Liste {
  int Elements[longMax];
  int der;
};
Liste L;   // L'initialisation se fait avec L.der=-1;
```

Insertion d'un élément *x* à la position *p* dans la liste L

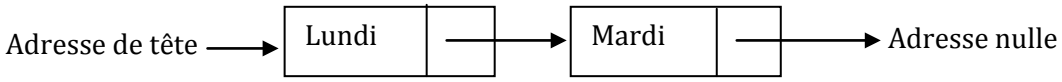
```
void insertion(int x, int p, Liste &L)
{ if (L.der>=longMax-1) cout<<"Erreur ! La liste est pleine\n";
  else if (p>L.der+1 || p<0) cout<<"Erreur ! Position incorrecte\n";
       else { for (int i=L.der;i>=p;i--) L.Elements[i+1]=L.Elements[i];
              L.Elements[p] = x;
              L.der++;
            }
}
```

Suppression de l'élément de position *p* de la liste L

```
void suppression(int p, Liste &L)
{ if (p>L.der || p<0) cout<<"Erreur ! Position incorrecte\n";
  else { for (int i=p;i<=L.der-1;i++) L.Elements[i] = L.Elements[i+1];
         L.der--;
       }
}
```

I.3. Implémentation d'une liste à l'aide d'une liste chaînée

Une liste chaînée est formée d'un ensemble de blocs ou cellules de même type. Chaque bloc contient à la fois les données de la liste ainsi que l'adresse du bloc qui le succède dans la liste. La figure suivante illustre une liste contenant les jours de cours :



La cellule qui se trouve à l'adresse début contient la tête de la liste et le dernier bloc a une adresse ayant la valeur ''nulle'' qui signifie qu'il n'y a pas de cellules la succédant. Par conséquent une liste vide a une adresse début qui est ''nulle''.

L'avantage d'utiliser une liste chaînée au lieu d'un tableau pour représenter une liste est clair. Elle nous permet d'éviter:

- l'utilisation de structure à mémoire contiguë qui nécessite un décalage lors de la procédure d'insertion ou de suppression,
- la réservation d'une longueur maximale de la liste.

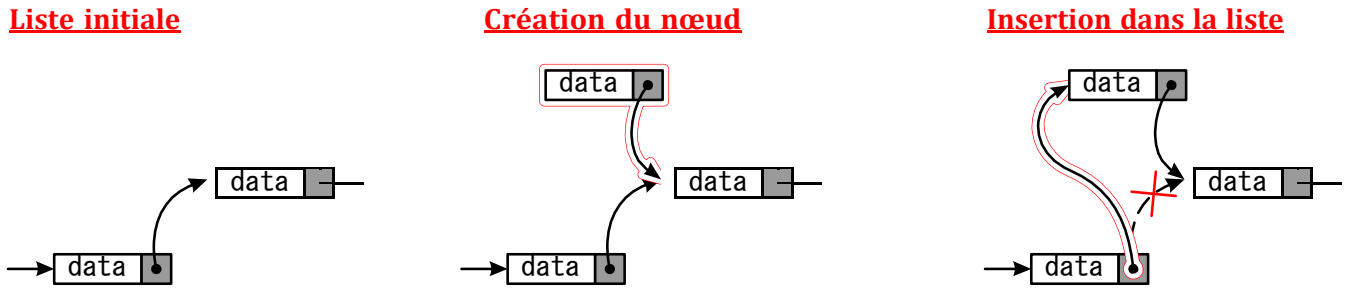
Pour accéder aux différents éléments d'une liste chaînée, on commence par l'adresse début (tête) qui permet d'accéder au premier bloc ou cellule. En accédant au premier bloc de la liste, on peut donc lire son contenu qui comprend, entre autre, l'adresse du deuxième bloc qui nous permet à son tour d'accéder au deuxième bloc et ainsi de suite. Le dernier bloc de la liste doit contenir une adresse spéciale nous permettant de savoir qu'il s'agit du dernier élément de la liste. Cette adresse est appelée nil, **NULL**, ou null dans les différentes implémentations de langages de programmation.

Les détails d'implémentation d'une liste chaînée contenant des **entiers** peuvent être définis en C/C++ comme suit :

Déclaration

```
struct Bloc {
  int element;
  Bloc * suiv;
};
Bloc *tete;   // L'initialisation se fait avec tete=NULL;
```

- **Insertion.** L'insertion d'un élément à la suite d'un élément donné se fait en deux étapes illustrées sur le dessin suivant :



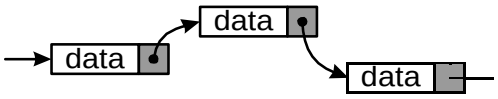
On crée le nœud en lui donnant le bon nœud comme nœud suivant. Il suffit ensuite de faire pointer l'élément après lequel on veut l'insérer vers ce nouvel élément. Le dessin représente une insertion en milieu de liste, mais en général, l'ajout d'un élément à une liste se fait toujours par le début: dans ce cas l'opération est la même, mais le premier élément de liste est un simple pointeur (sans champ data).

Insertion d'un élément *x* après le bloc d'adress *p* dans la liste d'adresse tete

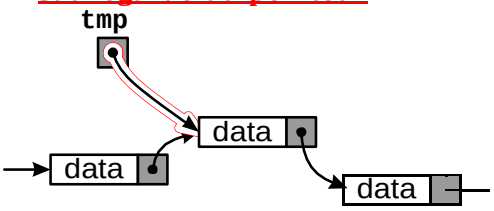
```
void insertion(int x, Bloc *p, Bloc *&tete)
{ Bloc *nouv = new Bloc;
  nouv->element = x;
  if (p==NULL) {  // Insertion en tête
                nouv->suiv = tete;
                tete = nouv;
               }
  else {
        nouv->suiv = p->suiv;
        p->suiv = nouv;
       }
}
```

- **Suppression.** Pour supprimer un nœud c'est exactement l'opération inverse, il suffit de faire attention à bien sauvegarder un pointeur vers l'élément que l'on va supprimer pour pouvoir libérer la mémoire qu'il utilise.

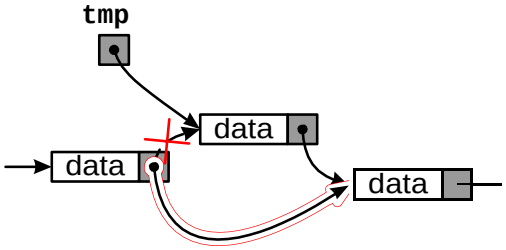
Liste initiale



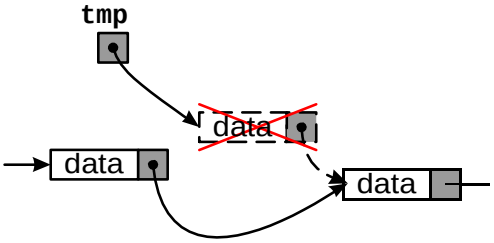
Sauvegarde du pointeur



Suppression dans la liste



Libération de la mémoire



Ici encore, le dessin représente une suppression en milieux de liste, mais le cas le plus courant sera la suppression du premier élément d'une liste.

Suppression de l'élément d'adresse *p* de la liste d'adresse tete

```
void suppression(Bloc *p, Bloc *prec, Bloc *&tete)
{ if (p==tete) tete = p->suiv;
  else prec->suiv = p->suiv;
  delete p;
}
```

On peut determiner l'adresse de l'élément précédent (cas de suppression par exemple) en sauvegardant dans la boucle, l'adresse p avant de la modifier comme suit :

```
    prec = p;
    p = p->suiv;
```

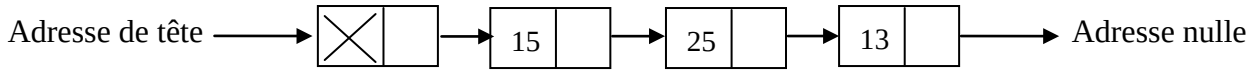
- **Comparaison entre la représentation contigüe et la représentation chainée des listes**
  (Un [O] représente un avantage et un [X] un inconvénient)

Représentation contigüe (Tableaux)		Représentation chaînée (Listes chaînées)	
Statique	[X]	Dynamique	[O]
Pointeur non nécessaire	[O]	Espace mémoire supplémentaire pour le pointeur	[X]
Facilité d'accès T[i]	[O]	L'accès nécessite le parcours de la liste à partir de l'adresse tete	[X]
Les opérations de mise à jour (insertion/suppression) nécessitent des décalages	[X]	Pas de décalages ; juste une mise à jour des chaînages	[O]

- **VARIANTES DE LA REPRESENTATION CHAINEE DES LISTES**

a) Liste chaînée avec élément fictif

Pour éviter un test spécial pour l'insertion et la suppression en début de liste, la liste est représentée sous forme homogène.



L'initialisation de la liste se fait par conséquent avec :

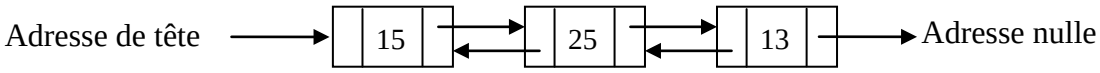
```
tete = new Bloc;
tete->suiv = NULL ;
```

et le parcours d'une telle liste doit prendre en considération l'existence de l'élément fictif comme suit :

```
Bloc *p = tete->suiv;
while (p!=NULL) {
     ...
   p=p->suiv;
}
```

b) Liste doublement chaînée

Dans certaines applications, on désire traverser une liste chaînée dans les deux directions. Etant donné un élément de la liste, on désire se déplacer vers son successeur ainsi que vers son prédécesseur d'une manière efficace et rapide.



Ceci peur être réalisé en définissant deux pointeurs dans la structure ***Bloc*** de la manière suivante :

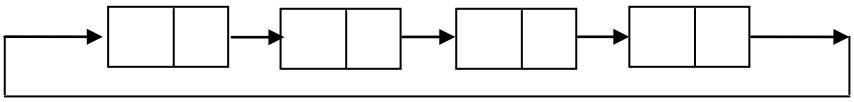
```
struct Bloc {
  int element;
  Bloc *suiv , *prec;
};
```

Les procédures d'insertion et de suppression doivent être modifiées pour tenir compte du pointeur ***prec*** de la manière suivante :

```
void insertion(int x, Bloc *p, Bloc *&tete)
{ Bloc *nouv = new Bloc;
  nouv->element = x;
  nouv->prec = p;
  if (p==NULL) { // Insertion en tête
                 nouv->suiv = tete;
                 if (tete!=NULL) tete->prec = nouv;  // Pb d'une liste vide
                 tete = nouv;
                }
  else {
        nouv->suiv = p->suiv;
        if (p->suiv!=NULL) p->suiv->prec = nouv;  //Pb Insertion en fin de liste
        p->suiv = nouv;
       }
}

void suppression(Bloc *p, Bloc *&tete)
{ if (p->prec!=NULL) p->prec->suiv = p->suiv; else tete = p->suiv;
  if (p->suiv!=NULL) p->suiv->prec = p->prec;
  delete p;
}
```

Remarque : Le parcours des listes peut être encore facilité avec les listes circulaires (simple ou double).



Cependant, il faut donner plus d'attention à l'implémentation des routines d'insertion et de suppression. Par ailleurs, le parcours d'une liste circulaire doit être légèrement modifié comme par exemple :

```
  Bloc *p = tete;
  do {
    ...
    p = p->suiv;
  }
  while (p!=tete);
```

CONCLUSION SUR LES LISTES

Par rapport à un tableau la liste présente deux principaux avantages:

- il n'y a pas de limitation de longueur d'une liste (à part la taille de la mémoire)
- il est très facile d'insérer ou de supprimer un élément au milieu de la liste sans pour autant devoir tout décaler ou laisser un trou.

En revanche, pour un même nombre d'éléments, une liste occupera toujours un peu plus d'espace mémoire qu'un tableau car il faut stocker les pointeurs (de plus le système d'exploitation conserve aussi des traces de tous les segments de mémoire alloués pour pouvoir les désallouer quand le processus s'arrête).