R1.01
DEV1

Les listes chaînées

Etienne Carnovali

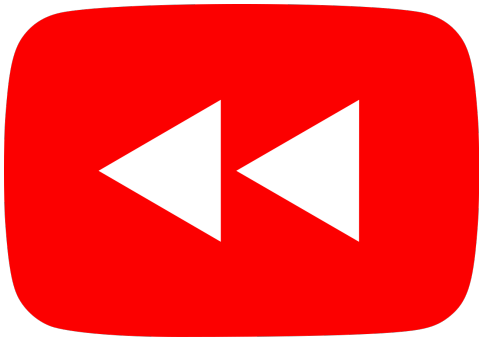
Jean-Michel Bohé

Marwa Hamdi

Ronan Champagnat

Au cours précédent

- Les fonctions
- Les pointeurs
- Les fichiers



Aujourd'hui

- Rappel : les tableaux
- Les listes chaînées :
  - Qu'est-ce que c'est ?
  - Comment on les utilisent ?
- Différences entre les tableaux et les listes
- Les piles et les files

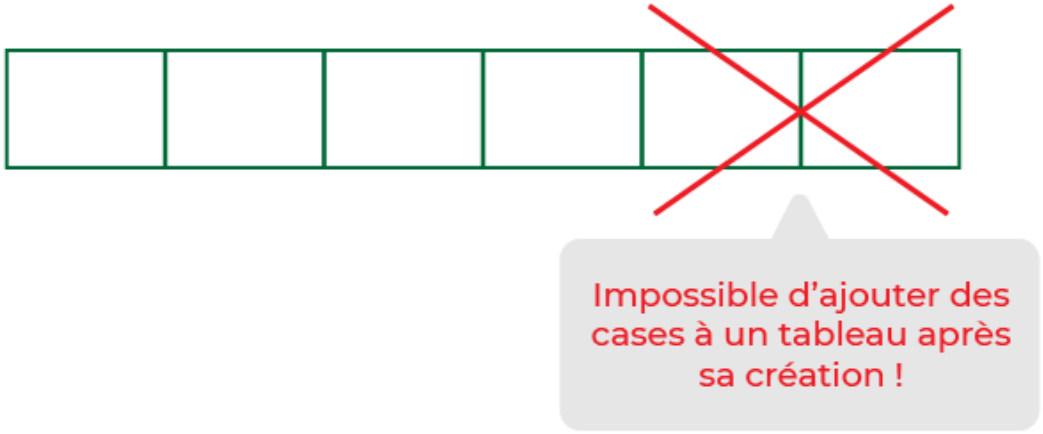
Rappel : Les tableaux

- Un tableau peut être représenté en mémoire comme ceci :

Un tableau de 4 cases en mémoire (représentation horizontale)

Rappel : Les tableaux

- Le problème des tableaux est qu'ils sont figés.
- Impossible de les agrandir (à moins d'en créer de nouveaux, plus grands) ;
- ou d'y insérer une case au milieu (à moins de décaler tous les autres éléments) :



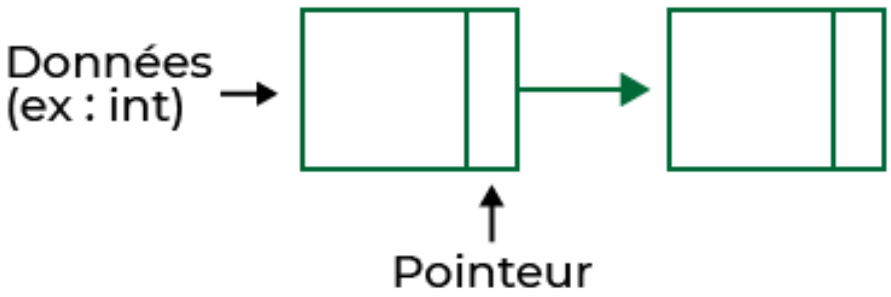
Les listes chaînées

Les listes chaînées

- Une liste chaînée est une **structure de données linéaire,** dans laquelle les éléments sont stockés dans des **emplacements mémoire non contigus.**
- Dans une liste chaînée, chaque objet de la collection est **lié uniquement à l'objet suivant** de la collection.
- L'objet d'une liste chaînée s'appelle un **nœud** et se compose de deux parties : données et un pointeur. La partie **données définit la valeur de l'objet ;** la partie **pointeur indique le nœud suivant.**

Les listes chaînées

- Chaque nœud peut contenir ce que l'on veut : un ou plusieurs `int`, `double`…
- En plus de cela, chaque élément possède un pointeur vers l'élément suivant :



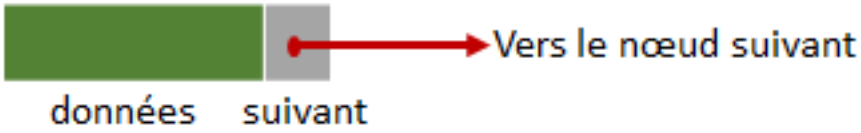
Les listes chaînées

- Les nœuds sont agencés entre eux : ils forment une chaîne de pointeurs, d'où le nom de "**liste chaînée**".
- Contrairement aux tableaux, les nœuds d'une liste chaînée ne sont **pas placés côte à côte dans la mémoire.**
- Chaque case pointe vers une autre case en mémoire, qui n'est pas nécessairement stockée juste à côté.

Construire une liste chaînée

- Pour concevoir un conteneur comme une liste chaînée simple, nous utilisons deux types de structures différentes : la **liste** et le **nœud**.

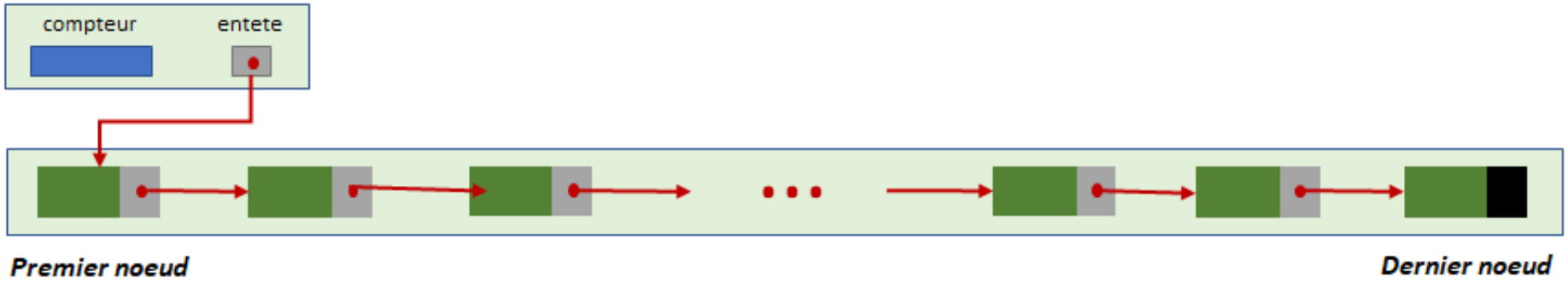
La liste



Le nœud

Construire une liste chaînée

- Cette structure Liste contient un pointeur vers le premier élément de la liste. En effet, il faut conserver l'adresse du premier élément pour savoir où commence la liste.
- Si on connaît le premier élément, on peut retrouver tous les autres en "sautant" d'élément en élément à l'aide des pointeurs suivant.



Créer la structure Liste

- La liste permettant de contrôler l'ensemble de la liste chaînée aura la forme suivante :

```
struct Liste
{
    int compteur;
    Noeud *entete;
};
```

- Un type de données entier, `compteur`, qui contient le nombre de nœuds dans la liste chaînée.
- Un `entete`, qui pointe vers le premier nœud de la liste.

Créer la structure Nœud

- Chaque nœud de la liste aura la structure suivante :

```
struct Noeud
{
    int nombre;
    Noeud *suivant;
};
```

- Une donnée, ici un nombre de type `int` : on pourrait remplacer cela par n'importe quelle autre donnée (un double, un tableau…). Cela correspond à ce que vous voulez stocker, c'est à vous de l'adapter en fonction des besoins de votre programme.
- Un pointeur vers un élément du même type appelé suivant. C'est ce qui permet de lier les éléments les uns aux autres : chaque élément sait où se trouve l'élément suivant en mémoire.

Initialisez une liste

- La fonction d'initialisation est la toute première que l'on doit appeler. Elle crée la structure de contrôle de la liste.

```
Liste* initialisation()
{
    Liste *liste = new Liste;

    if (liste == nullptr) //Est-ce que l'allocation dynamique s'est bien passée
    {
        exit(EXIT_FAILURE);
    }

    // initialisation des attributs de la liste
    liste->compteur = 0;
    liste->entete = nullptr;

    return liste;
}
```

Créer un nœud

- C'est la fonction la plus simple, ici nous créons un nouveau nœud, puis mettons à jour l'attribut `donnees` à la valeur transmise à la fonction et initions le pointeur suivant à `nullptr`.

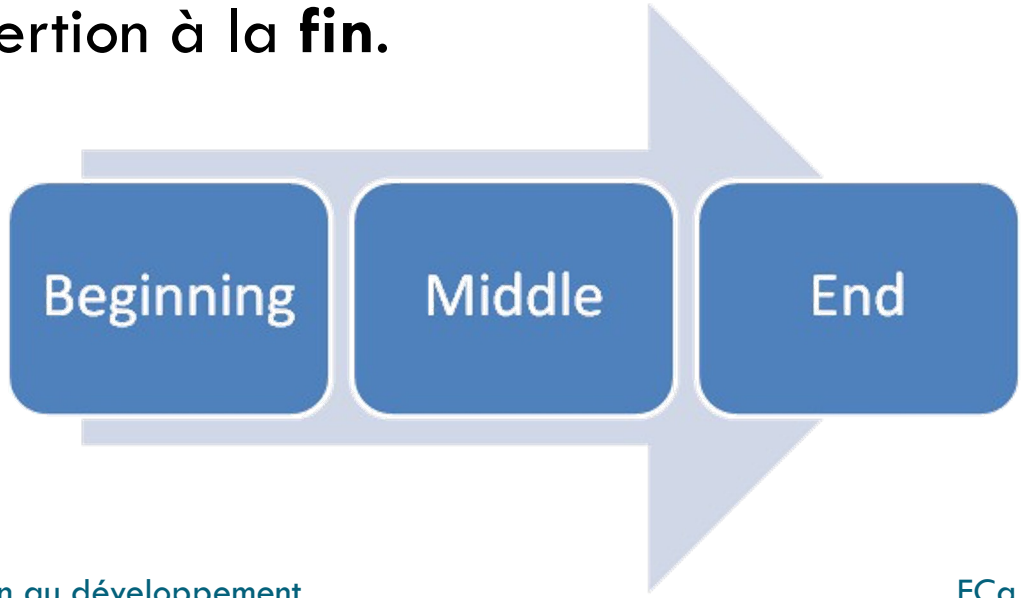
```
Noeud* creerNoeud(int valeur)
{
    Noeud *nouveauNoeud = new Noeud;

    if (nouveauNoeud == nullptr)
    {
        exit(EXIT_FAILURE);
    }

    nouveauNoeud->donnees = valeur;
    nouveauNoeud->suivant = nullptr;
    return nouveauNoeud;
}
```

Insérer d'un nœud

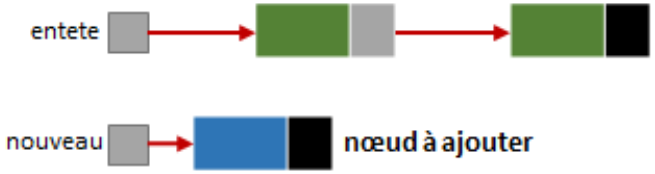
- L'insertion se fait à un endroit spécifié. On peut avoir trois cas :
  - L'insertion au **début,**
  - l'insertion au **milieu,**
  - ou l'insertion à la **fin.**



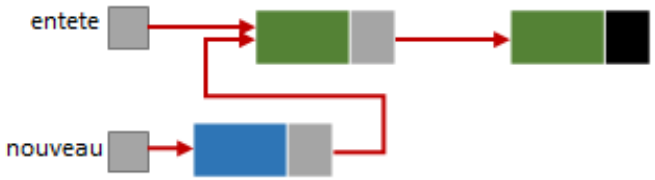
Insérer un nœud – au début

- L'insertion au début peut être effectuée à l'aide de deux opérations (après avoir créé un nœud)

(1) – nouveau = creerNoeud(…)

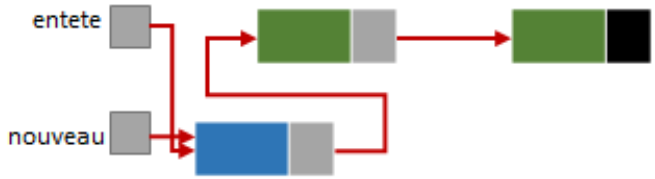


(2) – nouveau->suivant = liste->entete



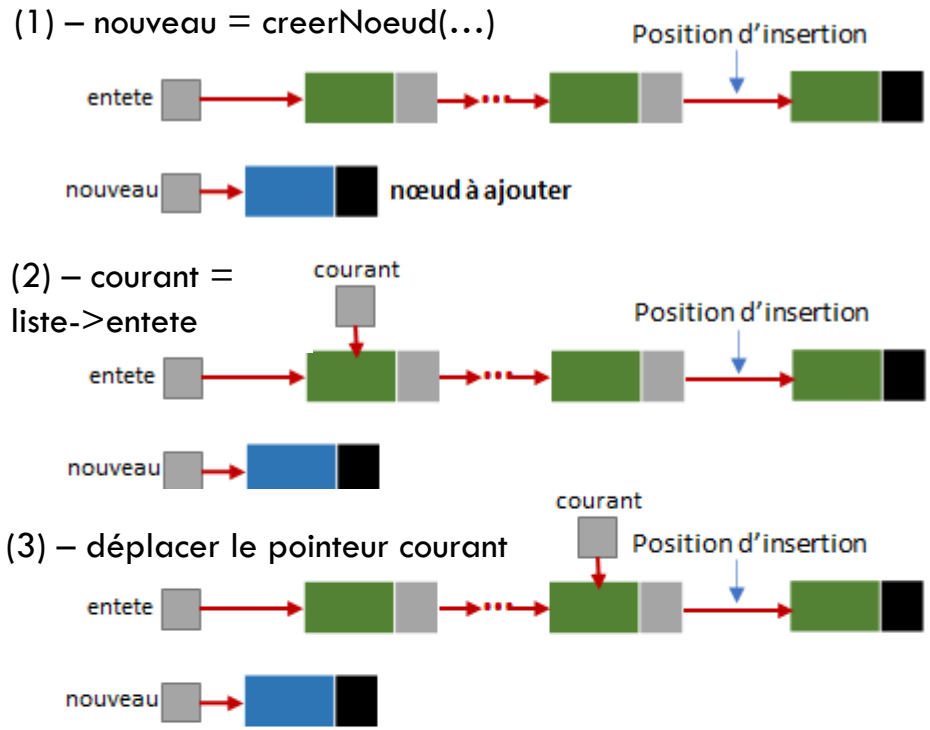
Ne pas faire l'étape 3 avant l'étape 2 sinon vous perdez l'adresse du premier élément de la liste.

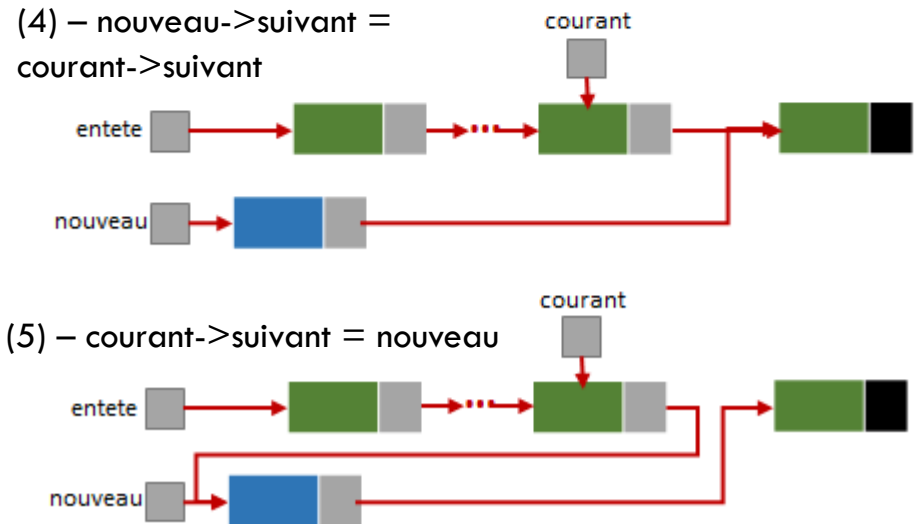
(3) – liste->entete = nouveau



Insérer un nœud – au milieu

- L'insertion au milieu est plus complexe. Il faut avoir un pointeur `courant` et le déplacer jusqu'au nœud situé avant la position d'insertion. Nous pouvons alors insérer le nœud.





Insérer un nœud – à la fin

- L'insertion à la fin est un cas particulier d'insertion au milieu dans lequel le pointeur actuel doit se déplacer vers le dernier nœud.
- Le nouveau nœud est inséré après le dernier nœud.

Insérer un nœud


```
void inserer(Liste *liste, int pos, int valeur)
{
    if (pos < 0 || pos > liste->compteur){
        cout << "Erreur! La position est invalide." << endl;
        return;
    }

    Noeud *nouveau = creerNoeud(valeur);

    //Insertion au début
    if (pos == 0){
        nouveau->suivant = liste->entete;
        liste->entete = nouveau;
    }
    // Insertion au milieu
    else{
        Element *courant; liste->premier;
        for (int i = 1; i < pos; i++){
            courant = courant->suivant;
        }
        nouveau->suivant = courant->suivant;
        courant->suivant = nouveau;
    }
    liste->compteur++;
}
```

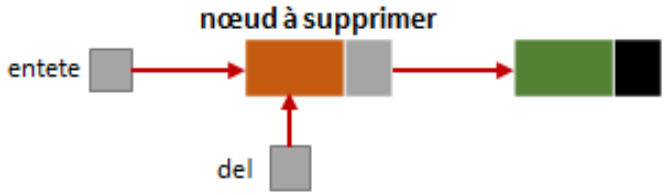
Supprimer un nœud

- La suppression ne pose pas de difficulté supplémentaire. Il faut cependant adapter les pointeurs de la liste dans le bon ordre pour ne perdre aucune information.
- Si la liste n'est pas vide, nous avons trois cas : suppression du premier nœud, suppression d'un nœud au milieu, ou suppression du dernier nœud.

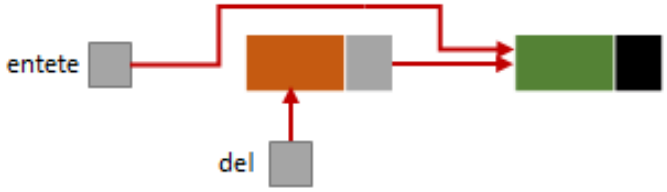
Supprimer un nœud – au début

- La suppression du premier nœud est très simple :

(1) – del = liste->entete



(2) – liste->entete = liste->entete->suivant



(3) – supprimer del

FUITE DE MÉMOIRE
Ne pas oublier de supprimer le nœud qui n'est plus utilisé (étape 3).

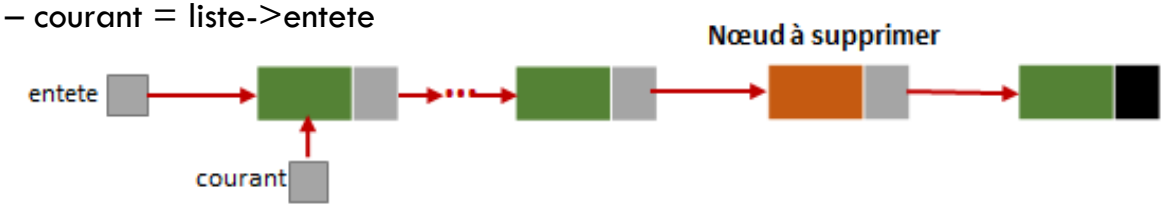
Mais ne pas le faire avant l'étape 2 sinon on perd l'adresse du second nœud.

Supprimer un nœud – au milieu ou à la fin

- La suppression d'un nœud au milieu est plus complexe. Nous devons avoir un pointeur `courant` pour pointer sur le nœud avant celui à supprimer.
- Nous pouvons alors utiliser un autre pointeur, `del`, pour pointer sur le nœud à supprimer.
- On peut alors effacer le nœud.

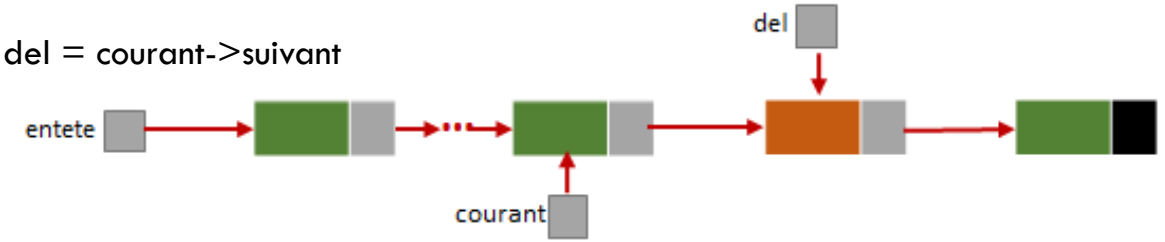
Supprimer un nœud – au milieu ou à la fin

(1) – courant = liste->entete



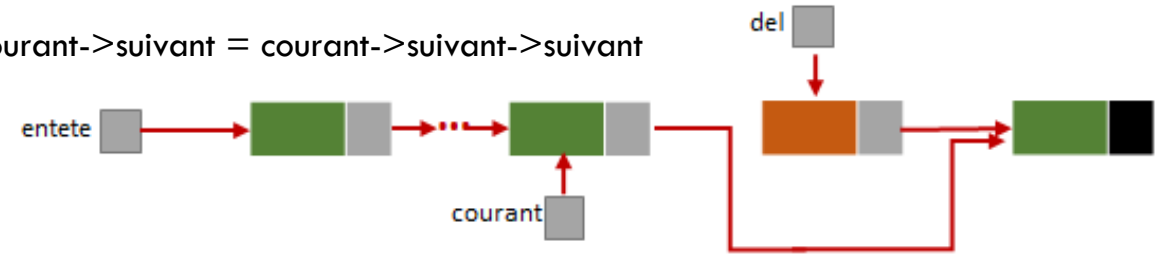
(2) – déplacer courant vers le nœud avant celui à supprimer

(3) – del = courant->suivant

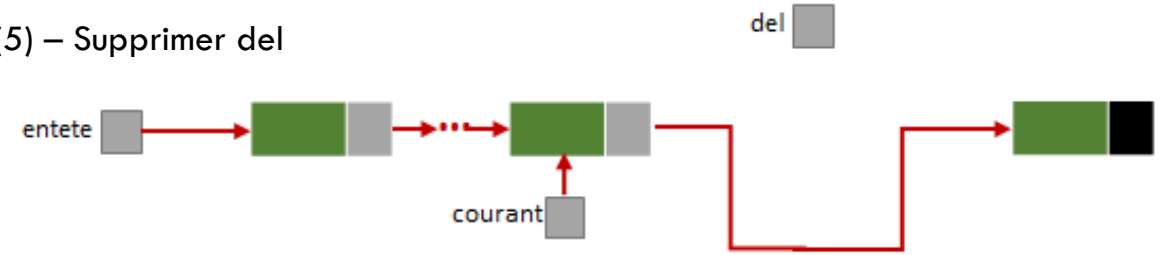


Supprimer un nœud – au milieu ou à la fin

(4) – courant->suivant = courant->suivant->suivant



(5) – Supprimer del



Supprimer un nœud


```
void supprimer(Liste *liste, int pos){
    if (pos < 0 || pos > liste->compteur-1){
        cout << "Erreur! La position est invalide." << endl;
        return;
    }

    if (pos == 0){
        Noeud *del = liste->entete;
        liste->entete = liste->entete->suivant;
        delete del;
    }

    else{
        Noeud *courant = liste->entete;
        for (int i = 0; i < pos-1; i++){
            courant = courant->suivant;
        }
        Noeud del = courant->suivant;
        courant->suivant = courant->suivant->suivant;
        delete del;
    }

    liste->compteur--;
}
```

Parcourir une liste chaînée

- Il suffit de partir du premier élément et d'afficher chaque élément un à un en "sautant" de bloc en bloc :

```
void afficherListe(Liste *liste)
{
    if (liste->compteur == 0)
    {
        cout << "La liste est vide" ;
    }

    Noeud *actuel;
    for(actuel = liste->entete ; actuel != nullptr ; actuel = actuel->suivant)
    {
        cout << actuel->nombre << " ->" ;
    }
}
```

Synthèse

- Les listes chaînées constituent un nouveau moyen de stocker des données en mémoire. Elles sont plus flexibles que les tableaux car on peut **ajouter et supprimer des « cases » à n'importe quel moment**.
- Dans une liste chaînée, chaque élément est une structure qui **contient l'adresse de l'élément suivant**.
- Il est conseillé de créer **une structure de contrôle** (du type Liste dans notre cas) qui retient l'adresse du premier élément.
- Il existe une version améliorée – mais plus complexe – des listes chaînées appelée « **listes doublement chaînées** », dans lesquelles chaque élément possède en plus l'adresse de celui qui le précède.

Différences entre les tableaux et les listes

Base de comparaison	Tableau	Liste chaînée
De base	C'est un ensemble cohérent d'un nombre fixe d'éléments de données.	C'est un ensemble ordonné comprenant un nombre variable d'éléments de données.
Taille	Fixe et spécifié lors de la déclaration.	Pas besoin de spécifier; grandir et rétrécir pendant l'exécution.
Allocation de stockage	L'emplacement de l'élément est alloué pendant la compilation.	La position de l'élément est attribuée pendant le temps d'exécution.
Ordre des éléments	Stockés consécutivement	Stocké au hasard
Accéder à l'élément	Accès direct ou aléatoire, c.-à-d., Spécifiez l'index ou l'indice de tableau.	Accès séquentiel, c'est-à-dire cheminement à partir du premier nœud de la liste par le pointeur.
Insertion et suppression d'élément	Lent relativement car le changement est nécessaire.	Plus facile, rapide et efficace.
Recherche	Recherche binaire et recherche linéaire	recherche linéaire
Mémoire requise	Moins (uniquement les valeurs)	Plus (stockage des adresses en plus des valeurs)

Les piles et les files

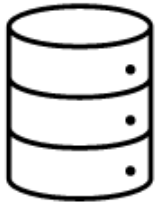
Les piles et les files

- Les piles et les files sont deux variantes des listes chaînées qui permettent de contrôler la manière dont sont ajoutés les nouveaux éléments.
- Cette fois, on ne va plus insérer de nouveaux éléments au milieu de la liste, mais seulement au début ou à la fin.
- Les piles et les files sont très utiles pour des programmes qui doivent traiter des données qui arrivent au fur et à mesure.

Les piles

- Nous utilisons de nombreux types de piles différents dans notre vie quotidienne. On parle souvent d'une pile de disques ou d'une pile de livres.

Pile de livres

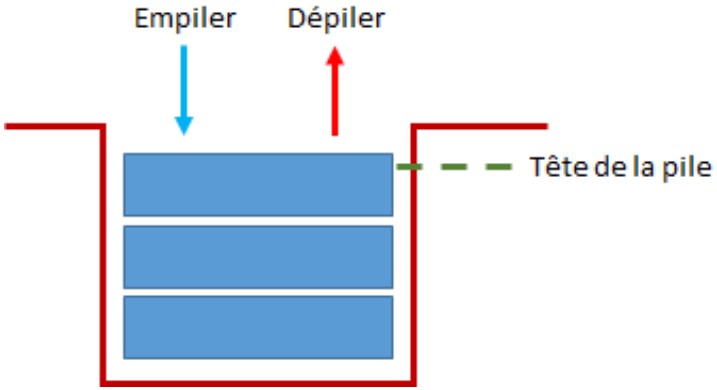


Pile de disques

- Si vous souhaitez supprimer un objet autre que celui de tête de la pile, vous devez d'abord supprimer tous les objets au-dessus.

Les piles

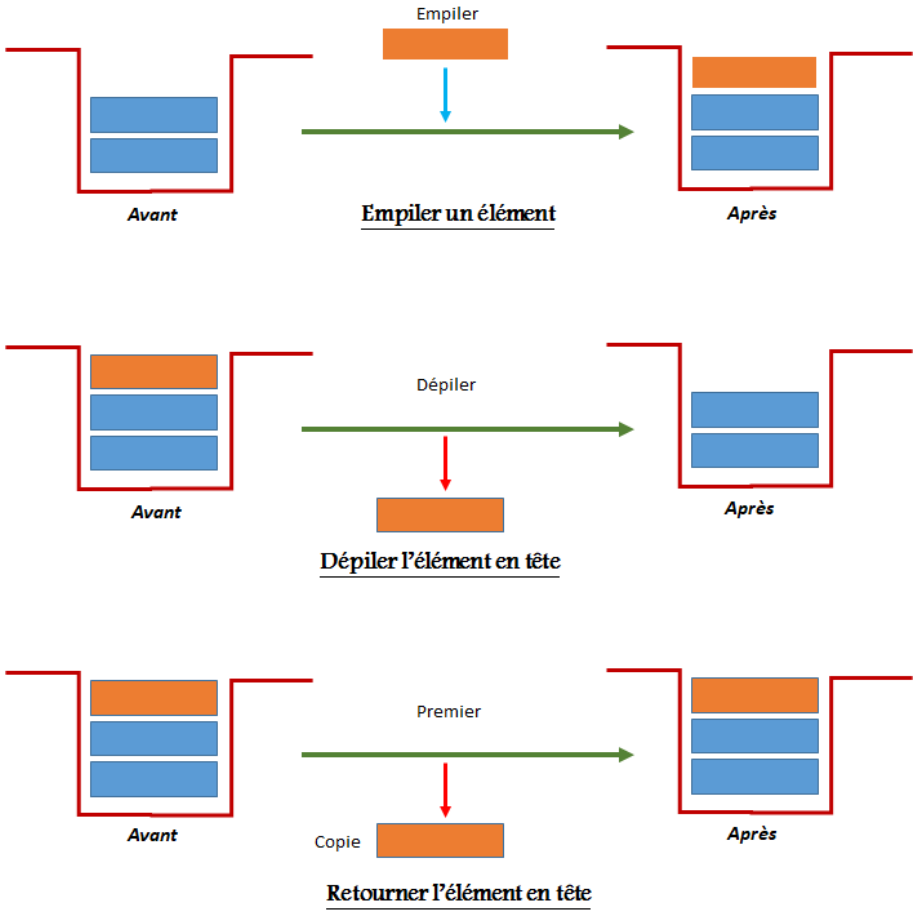
- Une pile est un conteneur implémenté sous la forme d'une liste linéaire dans laquelle tous les ajouts et suppressions sont limités à une extrémité, appelée tête de la pile (entête).



- Les piles sont appelées structure de données dernier entré, premier sorti (LIFO – Last In First Out).

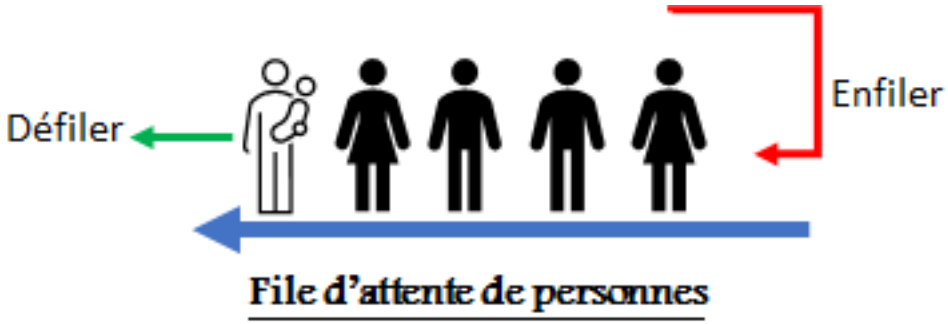
Les piles

- Nous rencontrons normalement trois opérations de base pour une structure de données de pile : empiler, dépiler et premier.



Les files

- Une file de personnes attendant le bus dans une gare routière est une file d'attente ;
- une liste d'appels mis en attente pour être répondus par un opérateur téléphonique est une file d'attente ;
- une liste de travaux en attente d'être traités par un ordinateur est une file d'attente.



Les files

- La file d'attente implémente le mécanisme **FIFO**, c'est-à-dire que l'élément inséré en premier est également supprimé en premier.
- En d'autres termes, l'élément le moins récemment ajouté est supprimé en premier dans une file d'attente.

File d'attente d'éléments de données

Les files

- L'implémentation de la file d'attente est assez similaire à l'implémentation de la liste chaînée, avec quelques petites modifications, Insertion à la fin et suppression de la tête.

- La structure de données de la file d'attente comporte les opérations suivantes :
  - `enfiler()` : Ajoute un nouvel élément à la fin de la file d'attente.
  - `défiler()` : retire un élément du début de la file et retourne sa valeur.
  - `premier()` : retourne la valeur de l'élément au début de la file.
  - `taille()` : retourne le nombre d'éléments dans la file.
  - `estVide()` : retourne 1 si la file est vide, sinon retourne 0.

Synthèse

- Les piles et les files permettent **d'organiser en mémoire des données qui arrivent au fur et à mesure**.
- Elles utilisent **un système de liste chaînée** pour assembler les éléments.
- Dans le cas des piles, les données s'ajoutent **les unes au-dessus des autres**. Lorsqu'on extrait une donnée, on récupère la dernière qui vient d'être ajoutée (la plus récente). On parle d'algorithme **LIFO** pour "Last In First Out".
- Dans le cas des files, les données s'ajoutent **les unes à la suite des autres**. On extrait la première donnée à avoir été ajoutée dans la file (la plus ancienne). On parle d'algorithme **FIFO** pour "First In First Out".


KEEP
CALM
AND
MAY THE FORCE
BE WITH YOU