




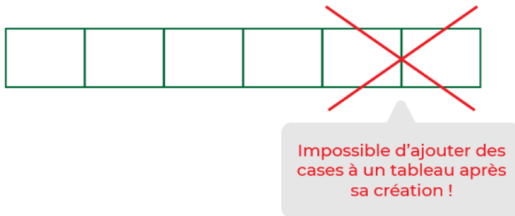
Programmation impérative

Cours n°8 : Listes chaînées et structures de données associées

Licence 1 Informatique & Vidéoludisme, Université Paris 8

Problème des tableaux

- ▶  Le problème des tableaux est qu'ils sont **figés**. Il est impossible de les agrandir (à moins d'en créer de nouveaux, plus grands).
- ▶  il est impossible d'y insérer une case au milieu sans décaler tous les autres éléments.
- ▶  Le langage C ne propose pas d'autre système de stockage de données, mais il est possible de créer une nouvelle structure de données.





Définition

Une **liste chaînée** est un moyen d'organiser une série de données en mémoire. Cela consiste à assembler des structures en les liant entre elles à l'aide de pointeurs.

- Une **liste chaînée** est souvent représentée graphiquement comme ceci :



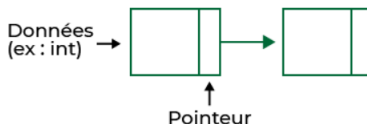
- Une liste chaînée est un assemblage **de structures** liées par **des pointeurs**. Elle peut contenir n'importe quel type de variable: **int**, **double**, **int***, etc.



Contrairement aux tableaux :

Les éléments 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.


- Une représentation plus juste est donc la suivante :



Créer une liste chaînée

- ▶ Nous allons créer une liste chaînée de **int**.
 - ▶ Nous pouvons aussi bien créer une liste chaînée contenant des **nombres décimaux** ou même **des tableaux** et **des structures**.
 - ▶ Le principe des listes chaînées s'adapte à n'importe quel type de données, mais ici, nous allons faire simple pour bien comprendre le principe.
- ▶ **Étape 1** : On commence par créer une structure adaptée à l'une des cases de notre liste chaînée. Elle doit contenir un champ de type **int**, ainsi qu'un pointeur vers l'élément (du même type) suivant :

```
typedef struct Element Element;  
struct Element{  
    int nombre;  
    Element *suivant;  
};
```

- ▶  Grâce au pointeur sur **suivant**, chaque élément sait où se trouve l'élément suivant en mémoire.



Souvenez-vous !

Les cases ne sont pas côte à côte en mémoire. C'est la **grosse différence par rapport aux tableaux**. Cela offre davantage de souplesse car on peut plus facilement ajouter de nouvelles cases par la suite au besoin.

Créer une liste chaînée II

- **Étape 2** : Il faut créer la structure de contrôle, en dupliquant la structure précédente autant de fois qu'il y a d'éléments dans notre liste. Pour ce faire, il suffit de fixer un pointeur sur le premier élément de notre liste :

```
typedef struct Liste{  
    element *premier;  
}Liste;
```

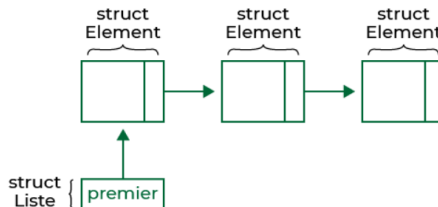
- Cette structure contient **un pointeur vers le premier élément de la liste**.
- En effet, il faut conserver l'adresse du premier élément pour savoir où commence la liste.

Créer une liste chaînée II

- **Étape 2** : Il faut créer la structure de contrôle, en dupliquant la structure précédente autant de fois qu'il y a d'éléments dans notre liste. Pour ce faire, il suffit de fixer un pointeur sur le premier élément de notre liste :

```
typedef struct Liste{  
    element *premier;  
}Liste;
```

- Cette structure 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.
- Il suffit de créer qu'un **seul exemplaire** de la structure liste. Elle permet de contrôler toute la liste.



Fin d'une liste chaînée, fonctions de base

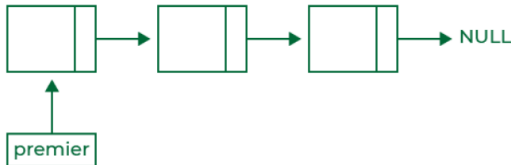
-  Il faut indiquer un endroit où arrêter de parcourir la liste.




Solution !



Il suffit de faire pointer le dernier élément de la liste vers **NULL**, c'est-à-dire de mettre son pointeur **suivant** à **NULL**



Fin d'une liste chaînée, fonctions de base

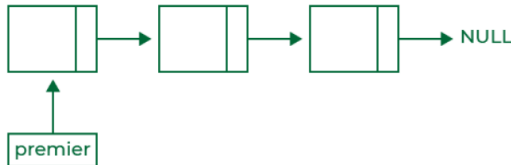
-  Il faut indiquer un endroit où arrêter de parcourir la liste.



Solution !



Il suffit de faire pointer le dernier élément de la liste vers **NULL**, c'est-à-dire de mettre son pointeur **suivant** à **NULL**



- Nous allons maintenant voir comment gérer le contenu d'une liste chaînée à partir de quelques fonctions de base :

- (1) Initialisation de la liste.
- (2) Ajouter un élément dans la liste.
- (3) Supprimer un élément de la liste.
- (4) Afficher le contenu de la liste.
- (5) Libérer la liste de la mémoire.

(1) Initialiser une liste chaînée

- ▶ On va écrire une première fonction qui va permettre de **créer une liste chaînée « élémentaire »** : elle ne contiendra qu'un élément (initialisé à `nbr` passé en paramètre) et un pointeur vers l'élément suivant initialisé à `NULL`.



À retenir !



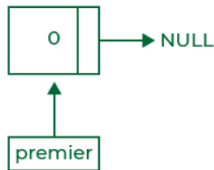
La fonction d'initialisation est **la toute première** que l'on doit appeler. Elle crée la structure de contrôle et le premier élément de la liste.

```
1 Liste *initialisation(int nbr) {
2     Liste *liste = malloc(sizeof(*liste));
3     Element *element = malloc(sizeof(*element));
4     if (liste == NULL || element == NULL){
5         exit(EXIT_FAILURE);
6     }
7     element->nombre = nbr;
8     element->suivant = NULL;
9     liste->premier = element;
10    return liste;
11 }
```

(1) Initialiser une liste chaînée II

Quelques explications sur cette fonction :

- ▶ On crée la structure nommée **liste** de type **Liste**.
- ▶ On alloue dynamiquement la mémoire pour la structure avec un **malloc**. La taille à allouer est calculée automatiquement avec `sizeof(*liste)`.
- ▶ On alloue de même le premier élément, et on vérifie si les allocations dynamiques ont fonctionné.
- ▶ En cas d'erreur, arrêter immédiatement le programme via **exit()**.
- ▶ Si tout s'est bien passé, on définit les valeurs de notre premier élément : **la donnée nombre** est mise à **0** par défaut et **le pointeur suivant** pointe vers **NULL**.



(2) Ajout d'un élément dans une liste chaînée

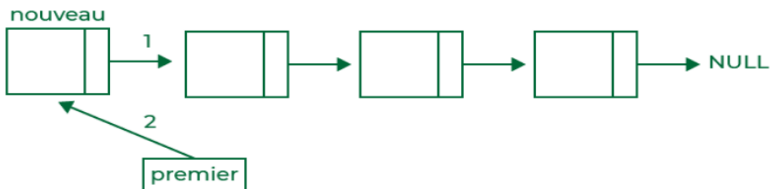
- On va écrire une fonction qui va permettre d'**ajouter un élément au début** d'une liste chaînée. Notez que cette fonction pourrait être adaptée pour ajouter un élément à n'importe quelle position de la liste.

```
1 void insertion(Liste *liste, int nvNombre){
2     /* Creation du nouvel element */
3     Element *nouveau = malloc(sizeof(*nouveau));
4     if (liste == NULL || nouveau == NULL)
5         exit(EXIT_FAILURE);
6     nouveau->nombre = nvNombre;
7     /* Insertion de l'element au debut */
8     nouveau->suivant = liste->premier;
9     liste->premier = nouveau;
10 }
```

(2) Ajout d'un élément dans une liste chaînée II

Quelques explications sur cette fonction :

- ▶ Créer un nouvel élément qui va contenir le nombre que l'on veut stocker.
- ▶ Allouer l'espace nécessaire au stockage du nouvel élément et on y place le nouveau nombre `nvNombre`.
- ▶ Faire pointer notre nouvel élément vers son futur successeur, qui est l'actuel premier élément de la liste.
- ▶ Faire pointer le pointeur du premier élément de la liste vers notre nouvel élément.



(2) Ajout d'un élément dans une liste chaînée III

? Exercice 1

Écrire une fonction qui permet :

- ▶ d'insérer un nouvel élément à la fin d'une liste chaînée.
- ▶ d'insérer un nouvel élément à une position i passée en paramètre. Si i est plus grand que le nombre d'éléments, on insérera à la fin.

(2) Ajout d'un élément dans une liste chaînée III

? Exercice 1

Écrire une fonction qui permet :

- ▶ d'insérer un nouvel élément à la fin d'une liste chaînée.
- ▶ d'insérer un nouvel élément à une position *i* passée en paramètre. Si *i* est plus grand que le nombre d'éléments, on insérera à la fin.

▶ Solution :

```
1 void insertion_fin(Liste *liste, int
   nvNombre){
2     Element *nouveau = malloc(sizeof(*
   nouveau));
3     if (liste == NULL || nouveau ==
   NULL)
4         exit(EXIT_FAILURE);
5     nouveau->nombre = nvNombre;
6     Element *prem = liste->premier;
7     while (prem->suivant != NULL){
8         prem = prem->suivant;
9     }
10    nouveau->suivant=prem->suivant;
11    prem->suivant = nouveau;
12 }
```

```
1 void insertion_position(Liste *liste,
   int nvNombre, int pos){
2     Element *nouveau = malloc(sizeof(*
   nouveau));
3     if (liste == NULL || nouveau ==
   NULL)
4         exit(EXIT_FAILURE);
5     nouveau->nombre = nvNombre;
6     Element *prem = liste->premier;
7     int i=1;
8     while ((i<pos-1) && (prem->suivant
   != NULL)){
9         prem = prem->suivant;
10        i++;
11    }
12    nouveau->suivant=prem->suivant;
13    prem->suivant = nouveau;
14 }
```

(3) Suppression d'un élément dans une liste chaînée

- On va écrire une fonction qui va permettre de **supprimer un élément au début** d'une liste chaînée. Notez que cette fonction pourrait être adaptée pour supprimer un élément à n'importe quelle position de la liste.



Remember !

La suppression ne pose pas de difficulté supplémentaire. Il faut juste **adapter les pointeurs** de la liste dans le bon ordre pour **ne perdre aucune information**.

```
1 void suppression(Liste *liste){
2     if (liste == NULL)
3         exit(EXIT_FAILURE);
4     if (liste->premier != NULL){
5         Element *aSupprimer = liste->premier;
6         liste->premier = liste->premier->suivant;
7         free(aSupprimer);
8     }
9 }
```

(3) Suppression d'un élément dans une liste chaînée II

Quelques explications sur cette fonction :

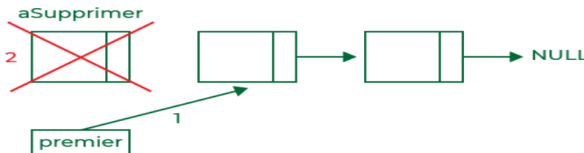
- ▶ Vérifier que le pointeur qu'on nous envoie n'est pas `NULL`.
- ▶ Vérifier ensuite qu'il y a **au moins 1 élément** dans la liste.



Attention !

Il faut bien respecter l'ordre sinon on perd des données.

- ▶ Adapter ensuite le pointeur `premier` vers le nouveau premier élément. **C'est l'actuel deuxième élément, donc `premier->suivant`.**
- ▶ Supprimer l'élément correspondant à notre pointeur `aSupprimer` avec un **free**.



(3) Suppression d'un élément dans une liste chaînée III

? Exercice 2

Écrire une fonction qui permet :

- ▶ de supprimer l'élément de fin d'une liste chaînée.
- ▶ de supprimer un élément à une position i passée en paramètre. Si i est plus grand que le nombre d'éléments, on supprimera le dernier.

(3) Suppression d'un élément dans une liste chaînée III

? Exercice 2

Écrire une fonction qui permet :

- ▶ de supprimer l'élément de fin d'une liste chaînée.
- ▶ de supprimer un élément à une position *i* passée en paramètre. Si *i* est plus grand que le nombre d'éléments, on supprimera le dernier.

▶ Solution :

```
1 void suppression_fin(Liste *liste){
2     if (liste == NULL)
3         exit(EXIT_FAILURE);
4     Element *prem = liste->premier;
5     while (prem->suivant->suivant !=
6            NULL){
7         prem=prem->suivant;
8     }
9     prem->suivant=NULL;
10    free(prem->suivant);
11 }
```

```
1 void suppression_position(Liste *liste
2     , int pos){
3     if (liste == NULL)
4         exit(EXIT_FAILURE);
5     Element *prem = liste->premier;
6     int i=1;
7     while ((i<pos-1) && (prem->suivant
8         ->suivant != NULL)){
9         prem=prem->suivant;
10        i++;
11    }
12    Element *temp=prem->suivant;
13    prem->suivant=temp->suivant;
14    free(temp);
15 }
```

(4) Affichage d'une liste chaînée

- On va écrire une fonction qui va permettre d'afficher une liste chaînée sous la forme `premier->suivant1->suivant2->...`



À retenir !



Il suffit de partir du premier élément et d'afficher chaque élément un à un en "sautant" d'un élément vers le suivant.

```
1 void afficherListe(Liste *liste){
2     if (liste == NULL)
3         exit(EXIT_FAILURE);
4     Element *actuel = liste->premier;
5     while (actuel != NULL){
6         printf("%d -> ", actuel->nombre);
7         actuel = actuel->suivant;
8     }
9     printf("NULL\n");
10 }
```

(5) Libérer une liste chaînée

- Pour libérer de la mémoire une liste chaînée, on libère l'élément actuel et on passe au suivant tant que l'élément n'est pas NULL :

```
1 void libererListe(Liste *liste){
2     if (liste == NULL){
3         return;
4     }
5     Element *actuel = liste->premier;
6     Element *next;
7     while (actuel != NULL){
8         next = actuel->suivant;
9         free(actuel);
10        actuel = next;
11    }
12 }
```

Un exemple complet d'utilisation

```
1 int main(){
2     Liste *maListe = initialisation(0);
3     insertion(maListe, 4);
4     insertion(maListe, 8);
5     insertion(maListe, 15);
6     suppression(maListe);
7     afficherListe(maListe);
8     libererListe(maListe);
9
10    return 0;
11 }
```

► Résultat :

8 -> 4 -> 0 -> NULL



Remarques

1. Vous trouverez dans différentes ressources plusieurs manières de gérer une liste chaînée. Parfois, on se contente juste de la structure `Element` et on initialise un pointeur vers l'élément de tête :

```
Element* prem = NULL;
```

2. Globalement, l'idée est toujours la même : un élément d'une liste chaînée contient une donnée, et un pointeur vers l'élément suivant de la liste.

Exercices : récursivité et itérativité

- ▶ De même qu'en Racket, une liste est une structure naturellement récursive dans le sens où une liste est séparée en :
 - ▶ son premier élément,
 - ▶ le reste de la liste, qui est lui-même une liste chaînée.

Exercices : récursivité et itérativité

- ▶ De même qu'en Racket, une liste est une structure naturellement récursive dans le sens où une liste est séparée en :
 - ▶ son premier élément,
 - ▶ le reste de la liste, qui est lui-même une liste chaînée.

? Exercice 2

- ▶ Écrire une fonction itérative qui renvoie la somme de tous les éléments d'une liste chaînée.
- ▶ Écrire une fonction récursive qui produit le même résultat.

▶ Solution :

```
1 int somme_elem(Liste *liste){
2     int res=0;
3     Element *actuel = liste->premier;
4     while (actuel != NULL){
5         res+= actuel->nombre;
6         actuel = actuel->suivant;
7     }
8     return res;
9 }
```

```
1 int somme_elem_rec(Liste *liste){
2     if (liste == NULL || liste->
3         premier == NULL){
4         return 0;
5     }
6     else{
7         Element *prem = liste->premier
8         ;
9         Liste *tail = malloc(sizeof(
10            Liste*));
11         tail->premier = prem->suivant;
12         return prem->nombre +
13            somme_elem_rec(tail);
14     }
15 }
```


Listes doublement chaînées.



Warning !

Dans une liste (simplement) chaînée, on ne sait pas à partir d'un élément donné quel est l'élément précédent. Il est donc impossible de revenir en arrière à partir d'un élément avec ce type de liste.



Définition :

Les listes "**doublement chaînées**" ont des pointeurs dans les deux sens et n'ont pas ce défaut. Elles sont néanmoins plus complexes.

► Avantages :


- Faible sur le coût.
- Ajout et suppression en temps constant à la fin de la liste.
- Parcours possible à l'envers.

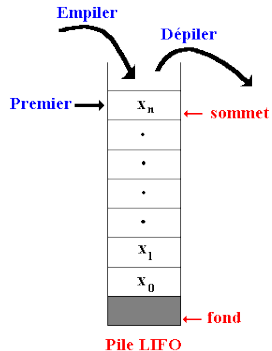
► Structure d'une liste doublement chaînée :

```
1 typedef struct DbElement DbElement;  
2 struct DbElement{  
3     int nombre;  
4     DbElement *suivant;  
5     DbElement *precedent;  
6 };  
7 typedef struct DbListe{  
8     DbElement *premier;  
9     DbElement *dernier;  
10 }DbListe;
```

Piles et files.


Piles : Last In, First Out

- ▶  Les piles et les files sont des structures de données particulières souvent utilisées en algorithmique. En langage C, elles seront implémentées à l'aide de listes chaînées.
- ▶ Une **pile** (en anglais LIFO : **Last In, First Out**) est une structure de données modélisant une pile d'objets de la vie quotidienne : on entasse les éléments au dessus des autres, puis c'est le dernier qu'on a ajouté à la pile qui sera retiré en premier.



Fonctions de piles (LIFO)

- ▶ `int est_vide(pile p)` : Tester si la pile est vide.
- ▶ `void empiler(Element e, pile* p)` : Ajouter `e` en tête de `p`.
- ▶ `Element depiler(pile* p)` : Retirer et renvoie le premier élément de `p`

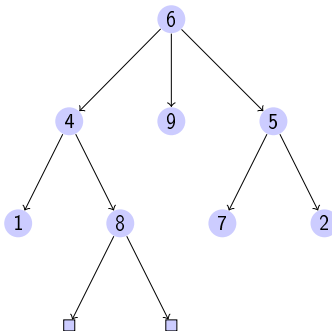
- ▶  Les piles et les files sont des structures de données particulières souvent utilisées en algorithmique. En langage C, elles seront implémentées à l'aide de listes chaînées.
- ▶ Une **file** (en anglais FIFO : **F**irst **I**n, **F**irst **O**ut) est une structure de données modélisant une file (d'attente) de la vie quotidienne : on ajoute des personnes à la fin de la file, et les personnes qui sortent en premier sont celles en début de file.



Fonctions de files (FIFO)

- ▶ `int empty(file f) :` Tester si la file est vide.
- ▶ `void enfiler(Element e, file* f) :` Ajouter `e` à la fin de `f`.
- ▶ `Element defiler(file* f) :` Supprime l'élément en tête de file, et le renvoie.

- Un arbre est une structure de données correspondant deux types d'éléments : des sommets, et des arcs reliant deux sommets.

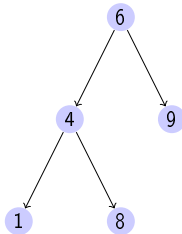


- Les rectangles noirs représentent des éléments vides, pour indiquer qu'un sommet n'a pas de successeur. Ce sont des éléments qui seront encodés par des arbres vides. On ne les représentera pas en pratique.
- Les sommets de cet arbre sont aussi appelés des **nœuds**. Chaque nœud possède des successeurs. On dit que 4, 9 et 5 sont les successeurs, ou fils, de 6.
- Réciproquement, le nœud 4 a un **parent**, qui est le nœud 6. Seul le nœud 6 (celui situé tout en haut) n'admet pas de parent.

Arbres binaires

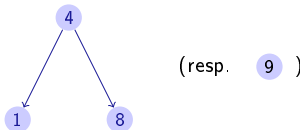
- Un arbre binaire est un arbre dans lequel chaque nœud admet au plus 2 fils. Il peut en admettre qu'un seul, dans ce cas on supposera toujours que c'est le plus à gauche.

- **Exemple :**



- 6 est le nœud de l'arbre appelé **racine** de *A*,

- la partie



est appelée le **fils gauche** (resp. le **fils-droit**) de l'arbre *A*.

- Les nœuds 1, 8 et 9 sont appelés des **feuilles**, ce sont les nœuds qui n'admettent pas de fils, au bout des branches. Un nœud qui n'est pas une feuille est appelé nœud **non-terminal**.

Structure d'arbre binaire

- ▶ Un arbre binaire est représenté par 3 données : un nœud racine et deux arbres binaires : le fils gauche et le fils droite.
- ▶ On va donc encoder un arbre binaire dans une structure contenant :
 - ▶ la donnée de la racine, dans notre cas un `int` mais ce pourrait être autre chose;
 - ▶ un pointeur vers le fils gauche qui est du même type;
 - ▶ un pointeur vers le fils droit qui est du même type.
- ▶ **Structure :**

```
1 typedef struct BTree BTree;  
2 struct BTree{  
3     int root;  
4     BTree *left;  
5     BTree *right;  
6 };
```

- ▶ Pour les fonctions de base sur les arbres binaires, voir le TP n°8 !