

# Structures de données, IMA S6

Listes chaînées

*d'après un cours d'A. Miné, ÉNS Ulm.*

Laure Gonnord

<http://laure.gonnord.org/pro/teaching/>

[Laure.Gonnord@polytech-lille.fr](mailto:Laure.Gonnord@polytech-lille.fr)

Université Lille 1 - Polytech Lille

Février 2011



- 1 Introduction
- 2 Listes simplement chaînées
  - Structure
  - Opérations
- 3 Divers sur les listes chaînées

- 1 Introduction
- 2 Listes simplement chaînées
- 3 Divers sur les listes chaînées

# Pourquoi ?

Rappel : la liste contiguë.

```
typedef struct Distribution {  
    int dernpers;  
    char listpers[MAXNUMPERS] [TAILLENOM];  
} Distribution;
```

Complexité en nombre de cases vues :

- Impression
- Recherche
- Insertion dans le cas d'une liste non pleine

## Pourquoi ? - 2

Hypothèse supplémentaire : **liste contiguë triée**

- Impression
- Recherche
- Insertion dans le cas d'une liste non pleine

## Pourquoi ? - 3

Et si la liste contiguë est pleine ? on **réalloue**.

- ▶ La liste chaînée va nous donner un moyen de gérer l'allocation case par case (de manière non contiguë).

## Notations algorithmiques

Pointeurs :

$p$  : **pointeur de** Entier (pointeur)

$p \uparrow$  (valeur pointée)

Allocation/Libération de mémoire :

- Fonction `allouer()` renvoie un pointeur vers une nouvelle cellule allouée
- Action `liberer(P)` récupère la cellule mémoire pointée par  $P$ .

Structures :

**Structure** *pointComplexe*

| x : Entier

| y : Entier

**FStruct**

$pc$  : `pointComplexe` (déclaration)

$pc.x$ ,  $pc.y$  (pour les accès)

- 1 Introduction
- 2 Listes simplement chaînées
  - Structure
  - Opérations
- 3 Divers sur les listes chaînées



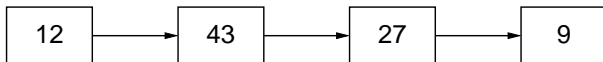
# Principe

**Liste = séquence ordonnée** d'éléments de même type.

Exemple : liste d'entiers (12, 43, 27, 9).

- l'ordre des éléments compte :  $(12, 43, 27, 9) \neq (12, 27, 43, 9)$
- la multiplicité des éléments compte  
 $(12, 43, 27, 9) \neq (12, 43, 27, 9, 9)$

**Liste simplement chaînée =**  
représentation où chaque élément **pointe** sur le suivant.



# Représentation des listes - 1

Exemple : cellule de liste d'**entiers** :

**Structure** *Cellule*

| valeur : Entier

| suivant : pointeur de *Cellule*

**FStruct**

- valeur est le **contenu de la cellule**, (ici, un entier)
- suivant **pointe vers la cellule suivante**,  
ou vaut **NULL** (fin de liste).

Le type de `cell` est récursif (autoréférentiel).

## Représentation des listes - 2

### Implantation en C d'une liste d'entiers :

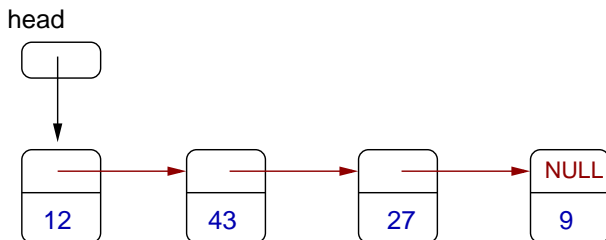
structure de **cellule** pour représenter un élément.

```
typedef struct {
    Cell* next;
    int    data;
} Cell ;
```

- data est le **contenu de la cellule**, (ici, un entier)
- next **pointe vers la cellule suivante**,  
ou vaut **NULL** (fin de liste).

## Représentation des listes - 3

Une liste est représentée par un **pointeur de tête** `Cell*`  
= pointeur sur la première cellule.



Tous les éléments sont **accessibles** depuis la tête de liste.

Par convention, `head` vaut `NULL` si **la liste est vide**.

# Opérations sur les listes

**Structure de données** = type + algorithmes de manipulation.

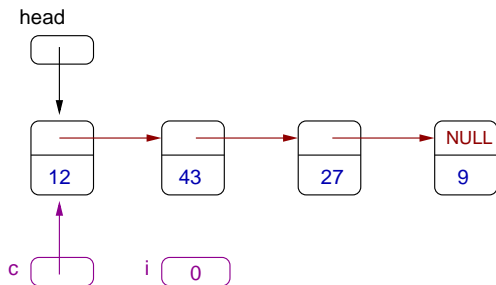
On va développer des fonctions pour les opérations suivantes :

- calcul de la longueur d'une liste,
- recherche d'un élément,
- insertion d'un élément,
- suppression d'un élément,
- concaténation de deux listes,
- destruction d'une liste.

Toutes nos fonctions prennent une tête de liste en argument.

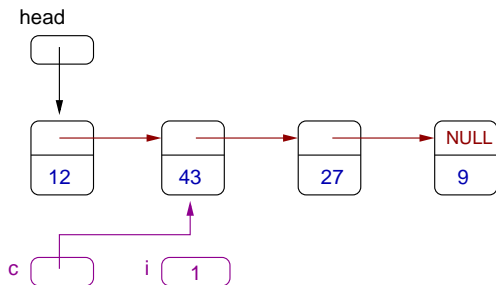
# Calcul de la longueur d'une liste - 1

Principe : on suit les pointeurs `next` jusqu'à rencontrer `NULL` et on compte le nombre de cellules rencontrées.



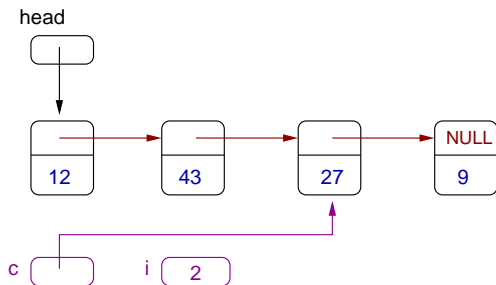
# Calcul de la longueur d'une liste - 1

Principe : on suit les pointeurs `next` jusqu'à rencontrer `NULL` et on compte le nombre de cellules rencontrées.



# Calcul de la longueur d'une liste - 1

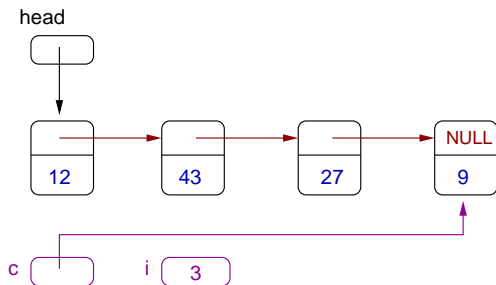
Principe : on suit les pointeurs `next` jusqu'à rencontrer `NULL` et on compte le nombre de cellules rencontrées.





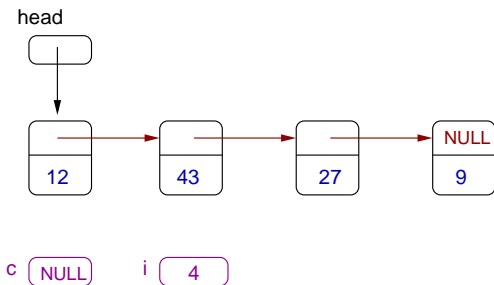
# Calcul de la longueur d'une liste - 1

Principe : on suit les pointeurs `next` jusqu'à rencontrer `NULL` et on compte le nombre de cellules rencontrées.



# Calcul de la longueur d'une liste - 1

Principe : on suit les pointeurs `next` jusqu'à rencontrer `NULL` et on compte le nombre de cellules rencontrées.



## Calcul de la longueur d'une liste - 2

- ▶ Écrire la fonction **en C** (TP)

## Recherche d'un élément

Spécification : renvoie `true` si la liste contient un élément égal à `elem`, `false` sinon.

Principe on suit les pointeurs `next` jusqu'à rencontrer `NULL` ou l'élément.

## Rappels sur la mémoire dynamique

Pour plus de flexibilité, les cellules sont allouées dynamiquement.

```
#include <stdlib.h>
void* malloc (size_t size);
void free (void* ptr);
```

### Effet :

- `malloc` alloue sur le tas un bloc de `size` octets, renvoie un pointeur **non typé** vers la zone allouée. Faire un **cast** !
  - `free` libère le bloc,
  - le bloc est accessible uniquement par pointeur.
- Allouer dynamiquement un tableau de 50 entiers ?

## Création d'une liste (exemple à ne pas suivre)

Pour construire/allouer la liste : (12, 43, 27, 9) :

```
Cell* head;
head = (Cell*) malloc(sizeof(Cell));
head->data = 12;
head->next = malloc(sizeof(Cell));
head->next->data = 43;
head->next->next = malloc(sizeof(Cell));
head->next->next->data = 27;
head->next->next->next = malloc(sizeof(Cell));
head->next->next->next->data = 9;
head->next->next->next->next = NULL;
```

► Peu pratique. On préfère construire une liste par insertions successives.

# Insertion en tête de liste - Principe

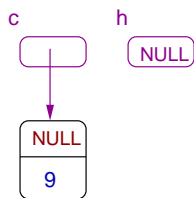
```
Cell* head = NULL;  
head = insere(head, 9);  
head = insere(head, 27);  
head = insere(head, 43);  
head = insere(head, 12);
```

head

NULL

# Insertion en tête de liste - Principe

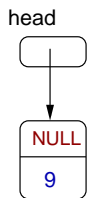
```
Cell* head = NULL;  
head = insere(head, 9);  
head = insere(head, 27);  
head = insere(head, 43);  
head = insere(head, 12);
```





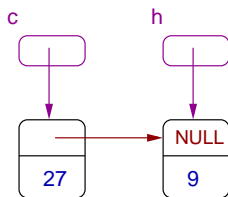
# Insertion en tête de liste - Principe

```
Cell* head = NULL;  
head = insere(head, 9);  
head = insere(head, 27);  
head = insere(head, 43);  
head = insere(head, 12);
```



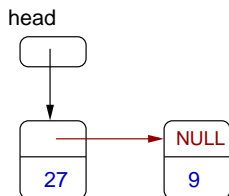
# Insertion en tête de liste - Principe

```
Cell* head = NULL;  
head = insere(head, 9);  
head = insere(head, 27);  
head = insere(head, 43);  
head = insere(head, 12);
```



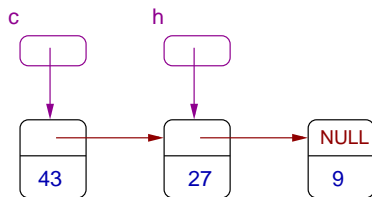
# Insertion en tête de liste - Principe

```
Cell* head = NULL;  
head = insere(head, 9);  
head = insere(head, 27);  
head = insere(head, 43);  
head = insere(head, 12);
```



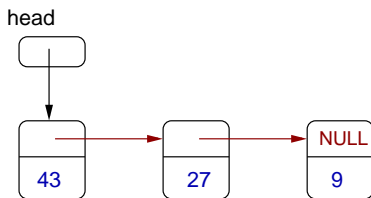
# Insertion en tête de liste - Principe

```
Cell* head = NULL;  
head = insere(head, 9);  
head = insere(head, 27);  
head = insere(head, 43);  
head = insere(head, 12);
```



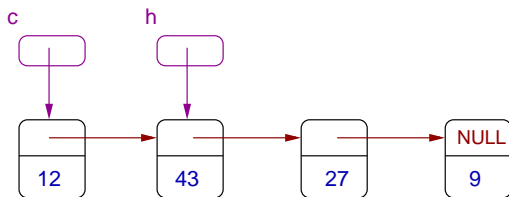
# Insertion en tête de liste - Principe

```
Cell* head = NULL;  
head = insere(head, 9);  
head = insere(head, 27);  
head = insere(head, 43);  
head = insere(head, 12);
```



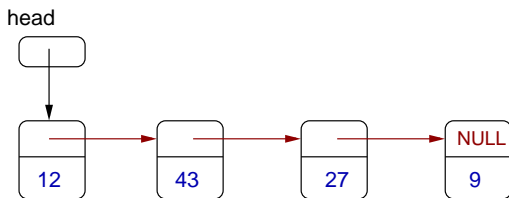
# Insertion en tête de liste - Principe

```
Cell* head = NULL;  
head = insere(head, 9);  
head = insere(head, 27);  
head = insere(head, 43);  
head = insere(head, 12);
```



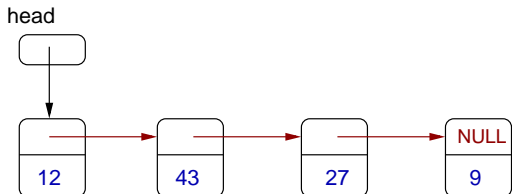
# Insertion en tête de liste - Principe

```
Cell* head = NULL;  
head = insere(head, 9);  
head = insere(head, 27);  
head = insere(head, 43);  
head = insere(head, 12);
```



## Insertion en tête de liste - Principe

```
Cell* head = NULL;  
head = insere(head, 9);  
head = insere(head, 27);  
head = insere(head, 43);  
head = insere(head, 12);
```



### Remarques :

- la liste est dans l'**ordre inverse** de celui des insertions,
- `insere` fonctionne sur une liste vide ou non-vide,
- la tête de liste est modifiée à chaque insertion,
- le coût d'une insertion est **constant**.



# Insertion en tête de liste - Implantation

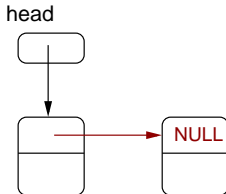
# Insertion en queue de liste

Deux cas à considérer :

- liste vide : on fait pointer **la tête de liste** sur la nouvelle cellule,
- liste non vide : on fait pointer **le champ next de la dernière cellule** sur la nouvelle cellule.



liste vide



liste non vide

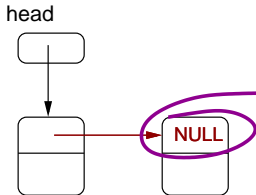
# Insertion en queue de liste

Deux cas à considérer :

- liste vide : on fait pointer **la tête de liste** sur la nouvelle cellule,
- liste non vide : on fait pointer **le champ next de la dernière cellule** sur la nouvelle cellule.



liste vide



liste non vide

## Exemple d'insertion en queue de liste

```
Cell* head = NULL;  
head = insere(head, 12);  
head = insere(head, 43);  
head = insere(head, 27);  
head = insere(head, 9);
```

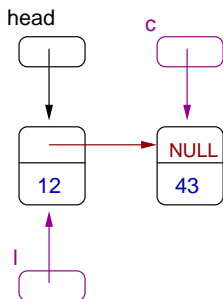
head

NULL



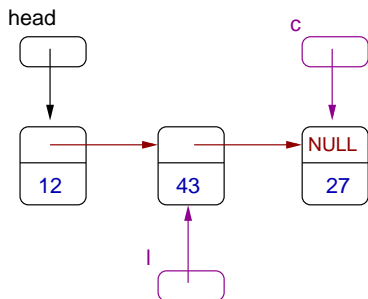
## Exemple d'insertion en queue de liste

```
Cell* head = NULL;  
head = insere(head, 12);  
head = insere(head, 43);  
head = insere(head, 27);  
head = insere(head, 9);
```



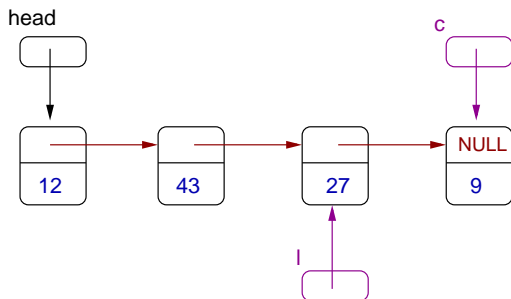
## Exemple d'insertion en queue de liste

```
Cell* head = NULL;  
head = insere(head, 12);  
head = insere(head, 43);  
head = insere(head, 27);  
head = insere(head, 9);
```



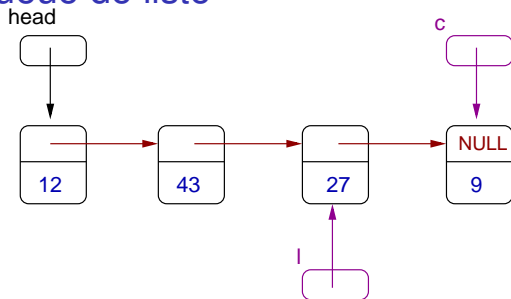
## Exemple d'insertion en queue de liste

```
Cell* head = NULL;  
head = insere(head, 12);  
head = insere(head, 43);  
head = insere(head, 27);  
head = insere(head, 9);
```



## Exemple d'insertion en queue de liste

```
Cell* head = NULL;
head = insere(head, 12);
head = insere(head, 43);
head = insere(head, 27);
head = insere(head, 9);
```

**Remarques :**

- la liste est dans le **même ordre** que celui des insertions,
- la tête de liste n'est modifiée que lors de la première insertion,
- le coût d'une insertion est **linéaire**,  $O(n)$



# Insertion en queue de liste

# Coût de construction d'une liste

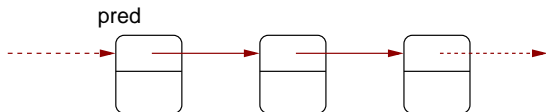
Pour construire une liste à  $n$  éléments :

- Par ajout en tête de liste :  $\mathcal{O}(n)$  **mais ordre inverse !**
- par ajout en queue de liste :  $\mathcal{O}(n^2)$

# Insertion après un élément - 1

## Spécification :

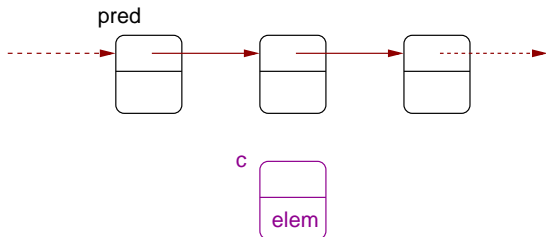
Insère un élément dans une liste , à l'aide d'un pointeur vers la cellule précédente.



# Insertion après un élément - 1

## Spécification :

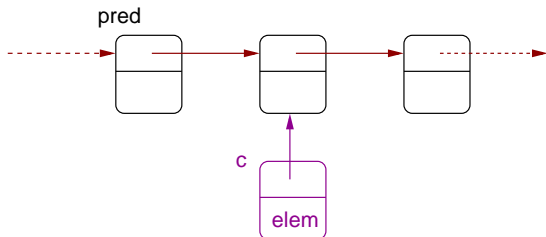
Insère un élément dans une liste , à l'aide d'un pointeur vers la cellule précédente.



# Insertion après un élément - 1

## Spécification :

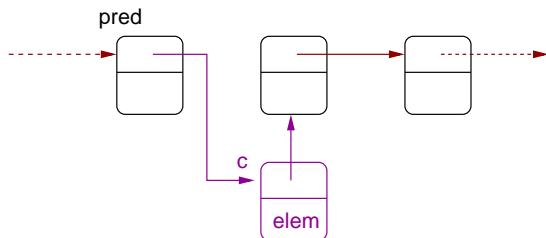
Insère un élément dans une liste , à l'aide d'un pointeur vers la cellule précédente.



# Insertion après un élément - 1

## Spécification :

Insère un élément dans une liste , à l'aide d'un pointeur vers la cellule précédente.



# Insertion après un élément - 1

## Spécification :

Insère un élément dans une liste , à l'aide d'un pointeur vers la cellule précédente.

## Notes :

- coût constant, hors calcul de `pred`,
- `pred` peut être obtenu par une variante de recherche (par exemple si on cherche à insérer dans une liste triée) cf TP,
- on suppose qu'on n'insère pas en première position.  
( $\Rightarrow$  **on suppose que la liste n'est pas vide**)

## Insertion après un élément - 2

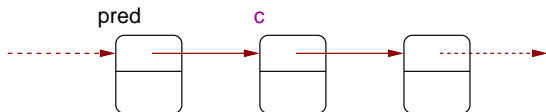
- ▶ L'insertion *avant* une cellule donnée est plus complexe...



# Suppression d'un élément - 1

## Spécification :

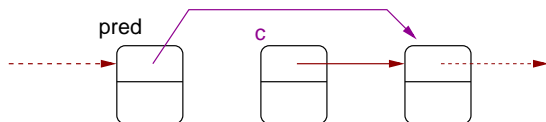
Supprime une cellule de la liste , à l'aide d'un pointeur vers la cellule précédente.



# Suppression d'un élément - 1

## Spécification :

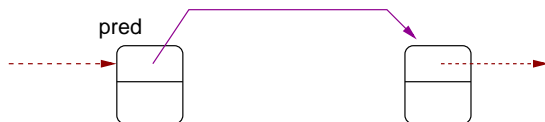
Supprime une cellule de la liste , à l'aide d'un pointeur vers la cellule précédente.



# Suppression d'un élément - 1

## Spécification :

Supprime une cellule de la liste , à l'aide d'un pointeur vers la cellule précédente.



# Suppression d'un élément - 1

## Spécification :

Supprime une cellule de la liste , à l'aide d'un pointeur vers la cellule précédente.

## Notes :

- coût constant, hors calcul de `pred`,
- on suppose qu'on ne détruit pas en première position, ( $\Rightarrow$  on suppose que la liste n'est pas vide)
- on suppose que `pred` a un suivant, (`pred->next  $\neq$  NULL`)  
(on peut par contre avoir `c->next=NULL`).

## Suppression d'un élément - 2

- ▶ Il reste à faire la version suppression complète .

## Suppression d'un élément - 3

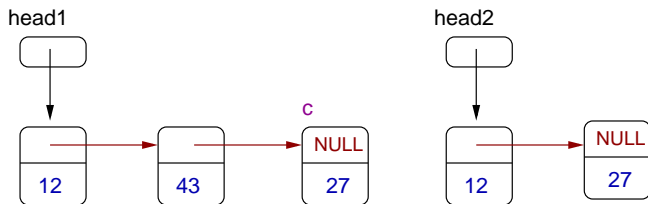
### Spécification :

supprime le premier élément égal à `elem` dans la liste **si il existe**.

- calcule automatiquement `pred`,
- gère les **cas limites** :
  - liste vide, liste à un seul élément,
  - `elem` en tête ou en fin de liste,
  - `elem` non présent dans la liste,
- la tête de liste peut changer,
- coût linéaire au pire, à cause de la recherche de `pred`.

# Suppression d'un élément - 4

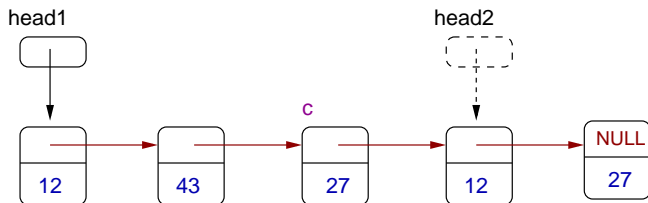
# Concaténation de listes



► Cf TD.



# Concaténation de listes



► Cf TD.

## Destruction totale d'une liste

Directement en  $C$  :

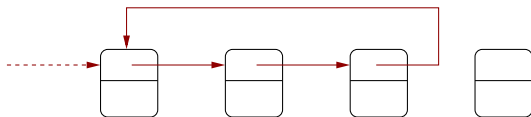
```
void detruit(Cell* head)
{
    Cell *c;
    while (head) {
        c = head->next;
        free(head);
        head = c;
    }
}
```

Coût ?

- 1 Introduction
- 2 Listes simplement chaînées
- 3 Divers sur les listes chaînées

## Erreurs courantes sur les listes

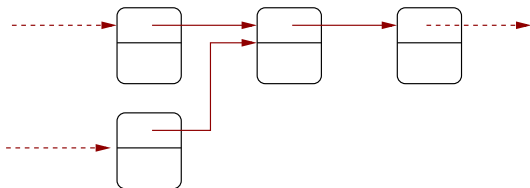
- Erreurs courantes sur les pointeurs et la mémoire dynamique :
  - déréférencer un pointeur NULL,
  - utiliser un bloc après l'avoir libéré,
  - libérer deux fois le même bloc,
  - oublier de libérer un bloc (fuites de mémoire).
  
- Introduction de **cycles** :



(génère des boucles infinies lors des parcours,  
cause des fuites de mémoire, ...)

## Erreurs courantes sur les listes

- **Partage** de cellules entre plusieurs listes :



(effets de bord lors de la modification d'une liste, cause des libérations multiples de blocs, . . . )

- Oubli des **cas limites** :
  - liste vide, listes à un élément,
  - insertion/suppression en première/dernière position,
  - etc.

## Comparaison entres listes et tableaux

Coût comparé des listes simplement chaînées et des tableaux.

	liste chaînée	liste contiguë
recherche d'une valeur	$\mathcal{O}(n)$	$\mathcal{O}(n)$
accès par indice	$\mathcal{O}(n)$	$\mathcal{O}(1)$
insertion en tête	$\mathcal{O}(1)$	$\mathcal{O}(n)$
insertion en queue	$\mathcal{O}(n)$	$\mathcal{O}(n)$
insertion au milieu	$\mathcal{O}(1) / \mathcal{O}(n)$	$\mathcal{O}(n)$
recherche si trié	$\mathcal{O}(n)$ (pas dicho)	$\mathcal{O}(\ln n)$
insertion (indice $p$ ) si trié	$\mathcal{O}(n)$	$\mathcal{O}(n)$ (décalages)

Les listes sont particulièrement efficaces pour :

- l'insertion et la suppression en tête de liste,
- l'insertion et la suppression en milieu de liste, si la cellule précédente est connue.