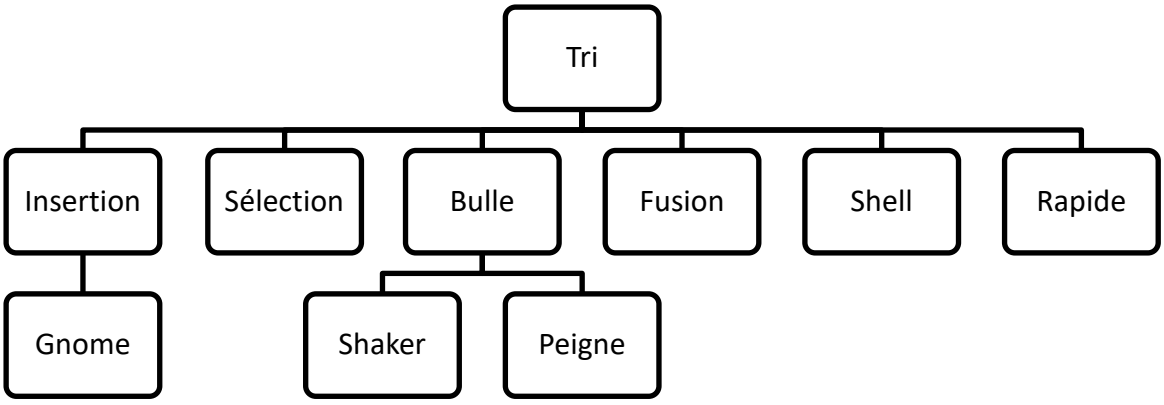
ALGORITHMES DE TRI DES TABLEAUX

I. Introduction

Un algorithme de tri permet d'ordonnée les données dans un tableau, une liste ou un fichier selon un critère bien déterminé dans ordre croissant ou décroissant. L'objectif principal de tri des données est de faciliter et minimiser le temps de recherche. Par exemple, si on a un tableau contenant mille réelles et on veut vérifier l'existence ou non d'un élément, il sera plus difficile de trouver cet élément si le tableau n'est pas trié puisque nous allons effectuer une recherche exhaustive. Par contre, si le tableau est trié le temps de recherche sera moins important puisque on peut faire des sauts pour localiser rapidement l'élément en utilisant par exemple une recherche dichotomique.

Il existe plusieurs algorithmes de tri dont la différence réside principalement dans le temps d'exécution exprimé par ce qu'on appelle la complexité algorithmique du programme. Egalement, on peut les distinguer par leurs stabilités. Un algorithme de tri est stable s'il ne modifie pas l'ordre initial des éléments égaux. Un algorithme de tri est instable s'il modifie l'ordre initial des éléments égaux.

La figure ci-dessous illustre quelques algorithmes de tri très connus.



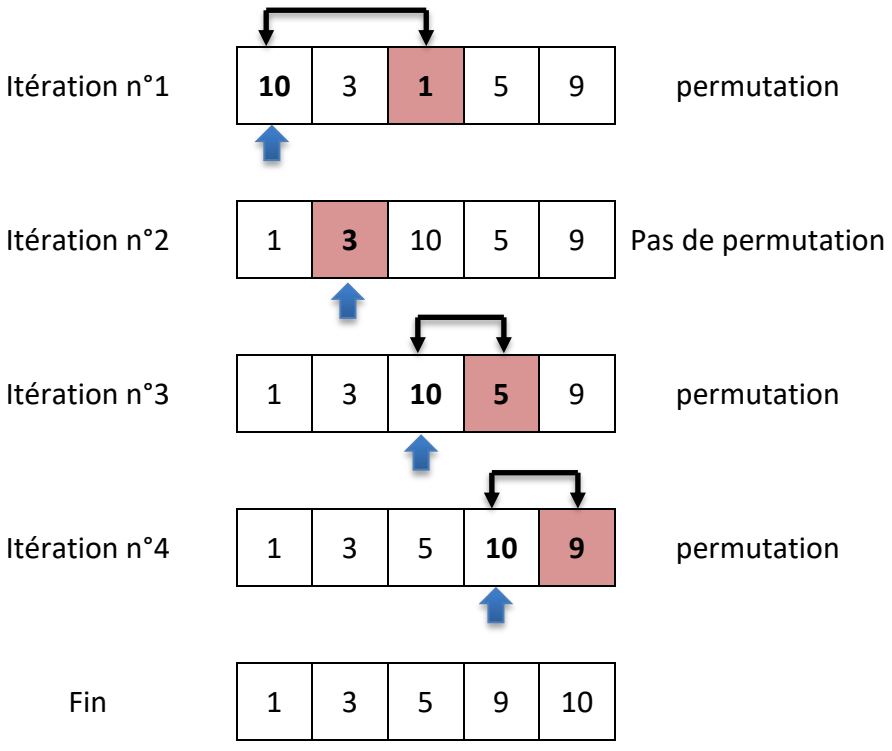
Dans ce cours nous allons s'intéressé aux trois types les plus simples qui sont : le tri par sélection, le tri par insertion et le tri à bulle.

II. Le tri par sélection

L'algorithme de tri par sélection consiste à chercher le plus petit élément du tableau puis le permuter avec le premier élément du tableau, puis faire la même chose mais permuter le petit élément avec le deuxième élément du tableau et ainsi de suite jusqu'à terminer tous le tableau. Il existe une autre méthode de tri par sélection qui utilise un autre tableau comme résultat. Donc au lieu de permuter avec le premier élément du tableau, il suffit de copier le minimum dans l'autre tableau et ainsi de suite mais toute en remplaçant la valeur du minimum par la valeur maximale dans le tableau original.

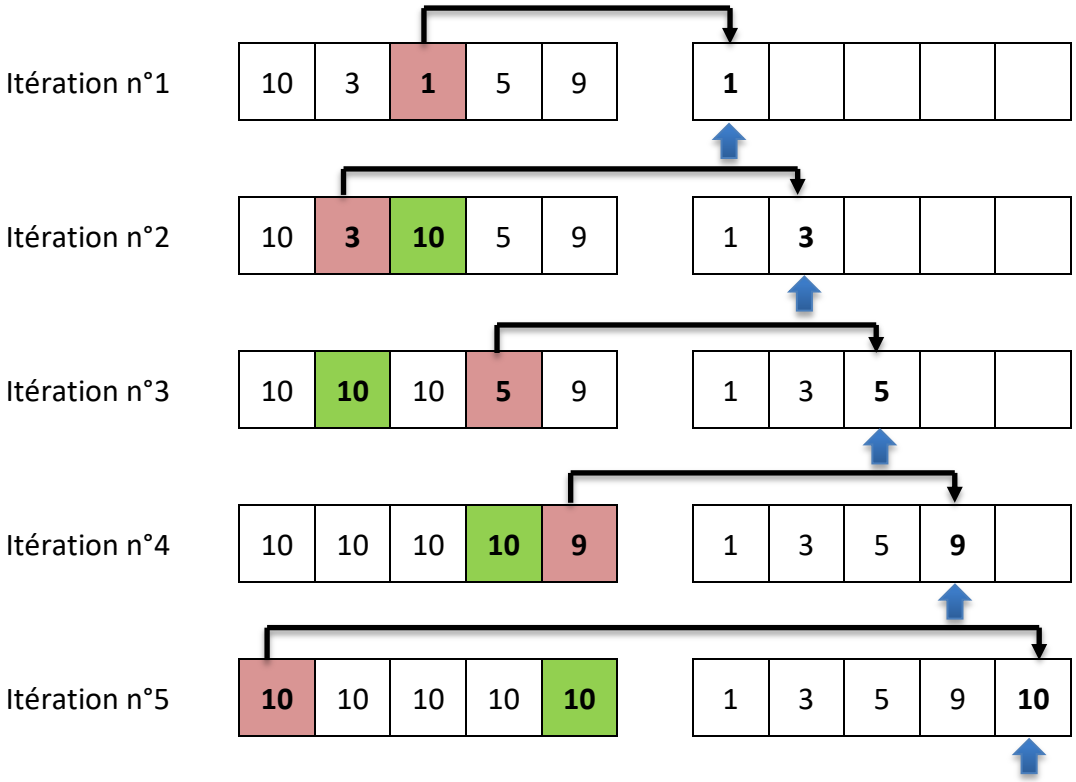
On a le tableau suivant contenant cinq entier et on veut le trié en utilisant le tri par sélection. La case en rouge indique le minimum qu'on va permuter avec l'élément courant. La flèche en bleu indique l'élément courant.

- **Méthode avec la permutation**



- **Méthode avec un autre tableau**

La case en vert indique le changement du minimum par le maximum du tableau pour ne pas être considéré dans la prochaine recherche.



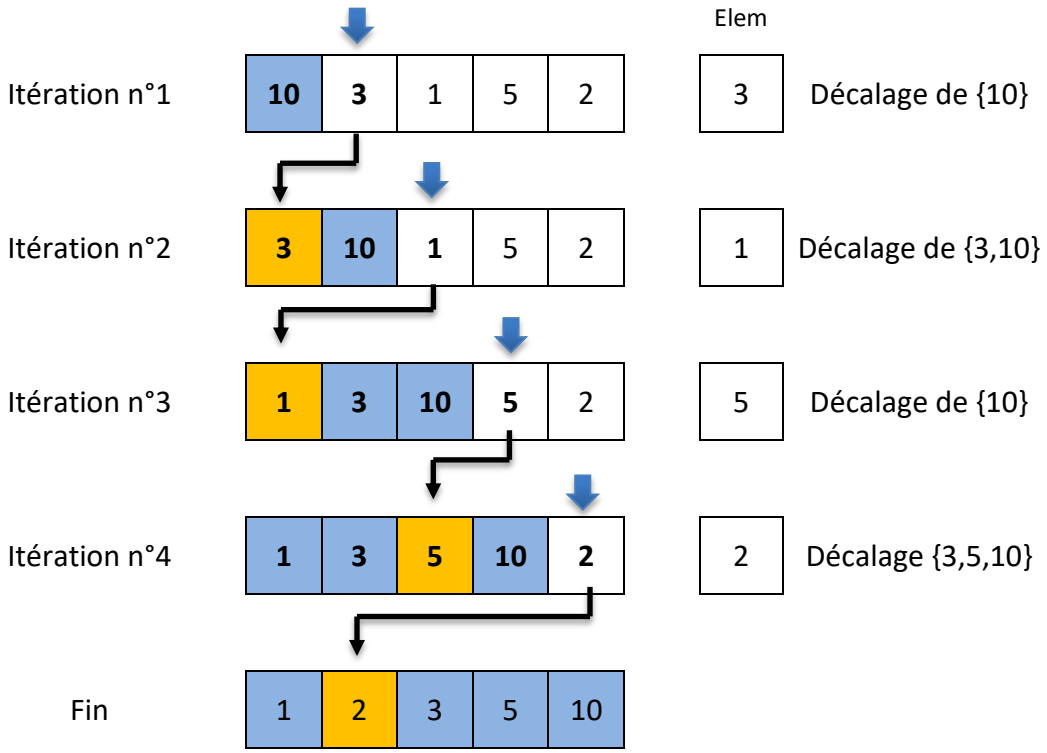
Le pseudo-code de l'algorithme de tri par sélection en utilisant la méthode de permutation est écrit comme suit sachant que *n* est le nombre d'élément du tableau:

```
Procédure Tri_Selection(T : tableau, n : entier)
      i,min : entier
Début
      Pour i de 0 à n - 2
            min ← MinTab(T,i,n)    // donner la position du minimum
            Si min <> i alors
                  permutTab(T,i,min)
            Fin si
      Fin pour
Fin procédure
//-----------------------------------------------------------------
Fonction MinTab(T : tableau, i : entier, n : entier) retourne entier
      min,j : entier
Début
      Min ← i
      Pour j de i+1 à n-1
            si T[j]<T[min] alors
                  min ← j
```

```
            Fin si
      Fin pour
      MiniTab ← min
Fin fonction
//-------------------------------------------------------------------
Procédure PermutTab(T : tableau, i : entier, j : entier)
      bf : type de l'élément
Début
      Bf ← T[i]
      T[i] ← T[j]
      T[j] ← bf
Fin procédure
```

III. Le tri par insertion

Le tri par insertion consiste à prendre l'i^{ème} élément du tableau et chercher sa bonne position puis effectuer un décalage jusqu'à cette position puis l'insérer. Dans ce qui suit une exécution manuelle de l'algorithme. On a un tableau de cinq entiers, l'algorithme commence par l'indice 1 puis cherche la bonne position de cet élément toute en sauvegardant sa valeur en dehors du tableau. Puis effectuer une série de décalage jusqu'à la bonne position dans la laquelle l'élément va être insérer et ainsi de suite.

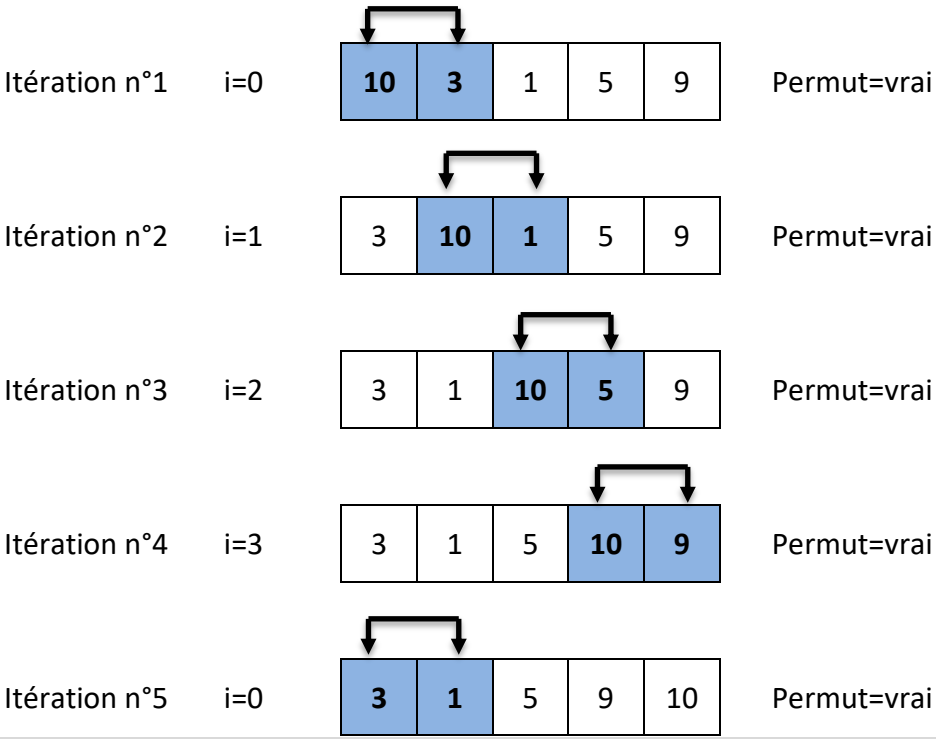


Le pseudo-code de la méthode de tri par insertion est écrit comme suit sachant que n est le nombre des éléments du tableau.

```
Procédure Tri_Insertion(T : tableau, n : entier)
      i : entier
Début
      Pour i de 1 à n - 1
            Décalage(T,i,n)
      Fin pour
Fin procédure
//-------------------------------------------------------------------
Procédure Décalage(T : tableau, i : entier)
      j : entier
      elem : type de l'élément
Début
      elem ← T[i]
      j ← i-1
      Tant que j>=0 et elem<T[j]
            T[j+1] ← T[j]
            j ← j-1
      Fin tant que
      T[j+1] ← elem
Fin fonction
```

IV. Le tri à bulle

Le tri à bulle consiste à comparer les éléments consécutifs deux à deux du tableau puis faire des permutations puis répéter l'opération dès le début. L'algorithme s'arrête lors qu'il n'y aura plus de permutation. Dans l'exemple qui suit on décrit l'exécution de l'algorithme de tri à bulle. Le bleu indique qu'il y a une permutation entre les éléments et le vert indique qu'il n'y a pas de permutation.



Itération n°6	i=1	1	**3**	**5**	9	10	
Itération n°7	i=2	1	3	**5**	**9**	10	
Itération n°8	i=3	1	3	5	**9**	**10**	
Itération n°9	i=0	**1**	**3**	5	9	10	Permut=faux
Itération n°10	i=1	1	**3**	**5**	9	10	
Itération n°11	i=2	1	3	**5**	**9**	10	
Itération n°12	i=3	1	3	5	**9**	**10**	Fin

Le pseudo-code de la méthode de tri à bulle est écrit comme suit sachant que n est le nombre des éléments du tableau.

```
Procédure Tri_Bulle(T : tableau, n : entier)
     permut : booléen
     i : entier
Début
     permut ← vrai
     i=0
     Tant que permut=vrai
         permut ← faux
         Pour i=0 à n-2
             Si T[i]>T[i+1] alors
                PermutTab(T,i,i+1)
                permut ← vrai
             Fin si
         Fin pour
     Fin tant que
Fin procédure
```

```
//-----------------------------------------------------------------
Procédure PermutTab(T : tableau, i : entier, j : entier)
     bf : type de l'élément
Début
     bf ← T[i]
     T[i] ← T[j]
     T[j] ← bf
Fin procédure
```

ALGORITHMES DE RECHERCHE DANS UN TABLEAU

I. Introduction

Un tableau est une structure de donnée complexe qui peut contenir un grand nombre d'éléments de même type. Plusieurs opérations peuvent être réalisées sur les éléments d'un tableau comme la saisie, l'affichage, le tri, la recherche, etc. Le temps d'exécution de ces opérations augmente avec le nombre des éléments du tableau. Certaines opérations ne peuvent être qu'exhaustive, c'est-à-dire il faut parcourir séquentiellement tout le tableau. D'autres opérations, comme la recherche, peuvent être optimisées pour réduire le temps d'exécution. Dans ce qui suit, nous allons voir les deux algorithmes de recherche dans un tableau, à savoir la recherche séquentielle et la recherche dichotomique. De plus, nous allons présenter la méthode de hachage qui est utilisée pour la résolution des problèmes de recherche dans les tableaux.

II. La recherche séquentielle

La recherche séquentielle consiste à parcourir tous les éléments d'un tableau du début jusqu'à la fin ou inversement. La recherche séquentielle est très couteuse en temps d'exécution surtout lorsque le nombre d'élément du tableau est très important. Les opérations suivantes nécessitent un parcours séquentiel complet du tableau :

- Saisie et affichage des éléments.
- Application d'une fonction sur les éléments d'un tableau (somme, moyenne, écart type).
- Recherche du nombre d'occurrence d'un élément.
- Recherche du minimum, maximum et mode.
- Recherche de toutes les positions d'un élément.
- Vérifier si un tableau est trié dans un ordre croissant ou décroissant.

Cependant, certaines opérations nécessitent un parcours séquentiel partiel du tableau :

- Recherche de l'existence ou la position d'un élément (si l'élément n'existe pas alors on a un parcours complet du tableau).
- Recherche séquentielle dans un tableau trié (on arrête la recherche si la valeur à chercher devient inférieur à l'élément courant du tableau quand il est trié dans un ordre croissant).

Le pseudo-code suivant illustre l'opération de recherche séquentielle dans un tableau de réelle trié :

```
Fonction Existe_Tableau_Trie(Tab : tableau, v : réelle, n : entier) retourne entier
      i:entier
Début
      i←0
      Tant que i<n et Tab[i]<v
            i←i+1
      Fin tant que
      Si i=n alors
            Retourner -1
      Sinon
            Si Tab[i]=v alors
                  Retourne i
            Sinon
                  Retourne -1
            Fin si
      Fin si
Fin fonction
```

III. La recherche dichotomique

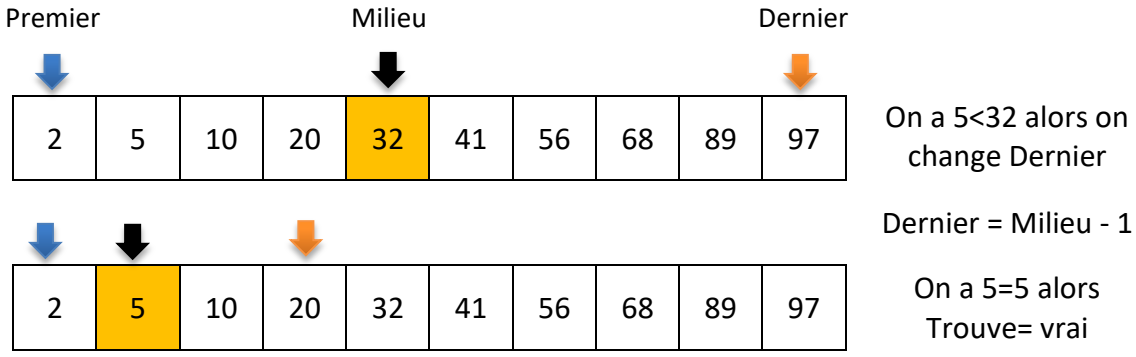
La recherche dichotomique est basée sur l'idée de diviser pour régner. Ainsi, la recherche dichotomique est plus rapide que la recherche séquentielle. Cependant, elle n'est possible que dans le cas où le tableau est trié sinon l'algorithme ne fonctionne pas. La recherche dichotomique consiste à prendre le milieu du tableau et le comparer avec la valeur recherchée (on considère que le tableau est trié dans un ordre croissant). S'ils sont égaux alors le problème est résolu, sinon si la valeur est inférieure au milieu du tableau alors on cherche dans la première moitié du tableau sinon on cherche dans la deuxième moitié du tableau. L'algorithme continu jusqu'à trouver l'élément ou la plage de recherche est terminée.

Le pseudo-code de l'algorithme de la recherche dichotomique est décrit comme suit.

```
Fonction Recherche_Dicho (Tab : Tableau, v : entier, n : entier) retourne
entier
      premier, dernier, milieu : entier
      trouve :booléen
Début
      premier ←0
      dernier ← n-1
      trouve ← faux
      Tant que trouve = faux et premier <= dernier
           milieu ← (premier + dernier) / 2
           Si (v < T[milieu]) Alors
               dernier ← milieu – 1
           Sinon
               Si (v > T[milieu]) Alors
                   premier ← milieu + 1
               Sinon
                   trouve ← Vrai
               Fin Si
           Fin Si
      Fin tant que
      Si (trouve = vrai) Alors
          Retourne milieu
      Sinon
          Retourne -1
      Fin Si
Fin fonction
```

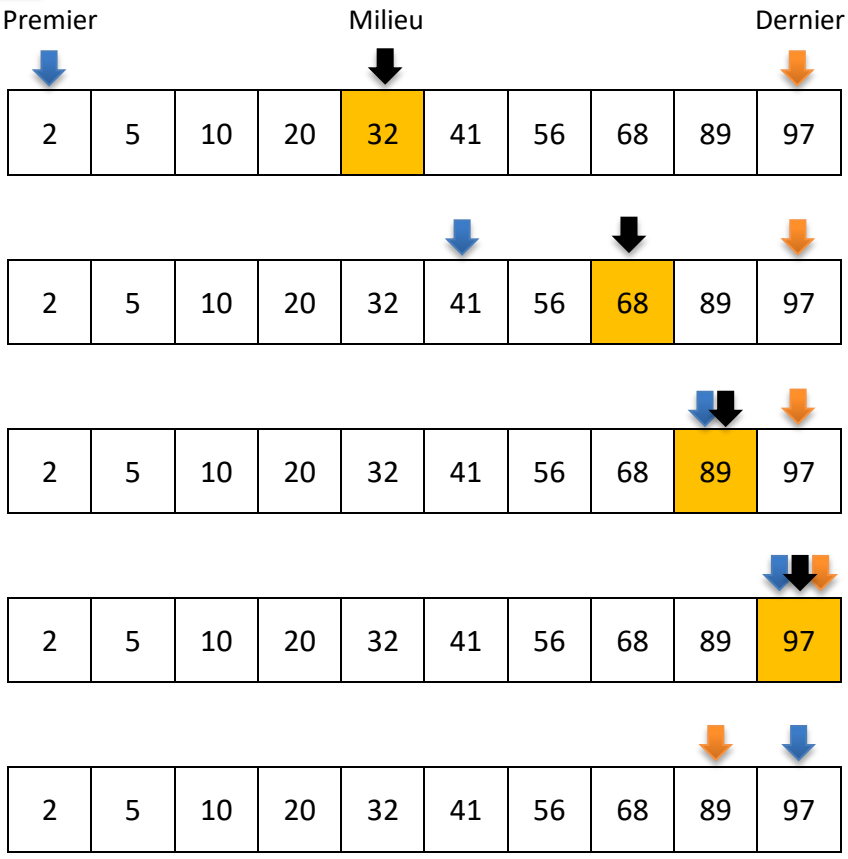
1. Exemple 1: l'élément existe dans le tableau

Soit le tableau suivant contenant 10 entiers triés dans un ordre croissant. On veut chercher l'existence de la valeur 5 dans le tableau avec l'algorithme de recherche dichotomique.



2. Exemple 2 : l'élément n'existe pas dans le tableau

Soit le tableau suivant contenant 10 entiers triés dans un ordre croissant. On veut chercher l'existence de la valeur 95 dans le tableau avec l'algorithme de recherche dichotomique.


Premier
Milieu
Dernier
2 5 10 20 32 41 56 68 89 97
On a 95>32 alors on change Premier
Premier = Milieu + 1
2 5 10 20 32 41 56 68 89 97
On a 95>68 alors on change Premier
Premier = Milieu + 1
2 5 10 20 32 41 56 68 89 97
On a 95>89 alors on change Premier
Premier = Milieu + 1
2 5 10 20 32 41 56 68 89 97
On a 95<97 alors on change Dernier
Dernier = Milieu – 1
2 5 10 20 32 41 56 68 89 97
Dernier > Premier alors on arrête
Fin

LA RECURSIVITE

I. Introduction

La récursivité est une notion très utilisée en programmation qui permet à une procédure ou une fonction de s'appeler elle-même directement ou indirectement. La récursivité est une autre alternative qui permet de résoudre certains problèmes itératifs. Généralement, un algorithme récursif peut être remplacé par son équivalent itératif et inversement mais ceci n'est pas toujours possible. Certains problèmes ou structures de données ne peuvent être traités qu'en utilisant des algorithmes récursifs.

II. Définition de la récursivité

1. Règles

Il existe un ensemble de règles à définir pour qu'un algorithme récursif soit bien fonctionnel, il faut :

- Déterminer les cas qui nécessitent un appel récursif et les cas qui ne nécessitent pas un appel récursif.
- Déterminer la condition d'arrêt ou de terminaison pour que l'algorithme se termine et ne fait pas une boucle infinie.
- Utiliser les appels récursifs avec des données plus proches des données satisfaisant la condition de terminaison.

Ci-dessous un exemple général d'appel récursif de procédure et de fonction :

```
Procédure Pro(paramètres)
Début
      Si condition alors
            Arrêt
      Sinon
            Pro(paramètres_autres_valeurs)
      Fin si
Fin procédure
```

```
Fonction Fonc(paramètres) retourne type
Début
      Si condition alors
            retourne valeur
      Sinon
            retourne Fonc(Paramètres_autres_valeurs)
      Fin si
Fin fonction
```

2. Récursivité indirecte ou croisée

On dit qu'on a une récursivité indirecte ou une récursivité croisée lorsqu'une procédure A, sans appel récursif, appelle une procédure B qui elle-même appelle A.

Exemple :

```
Procédure A(paramètres)
      …
      B(valeurs)
      …
Fin procédure

Procédure B(paramètres)
      …
      A(valeurs)
      …
Fin procédure
```

3. Performances de la récursivité

Les problèmes de mémoire et de rapidité sont les critères qui permettent de faire le choix entre la solution itérative et la solution récursive. Pour le calcul de la fonction factorielle, les différences entre les deux solutions ne se voient pas trop. Pour l'utilisation de la mémoire, il y aura dépassement dans les calculs avant la saturation de la mémoire.

4. Choix entre itération et récursivité

Le choix de la solution itérative peut être imposé par le langage de programmation. En effet, certains langages ne sont pas récursifs. D'autre part, seuls les problèmes dont la solution se base sur une relation de récurrence avec une condition de sortie peuvent être résolus de façon récursive.

Outre ces deux contraintes, le choix doit être fait uniquement en fonction des critères d'efficacité (contraintes de temps et d'espace). Si la solution récursive satisfait ces critères, il n'y a pas lieu de chercher systématiquement une solution itérative.

III. Etude d'exemples

1. La fonction factorielle

a. Solution itérative

```
Fonction Fact(n : Entier) retourne Entier
     f,i : entier
Début
     f ← 1
     Pour i de 2 à n Faire
          f ← f * i
     Fin Pour
     Retourne f
Fin fonction
```

b. Solution récursive

La récursivité est définie par :

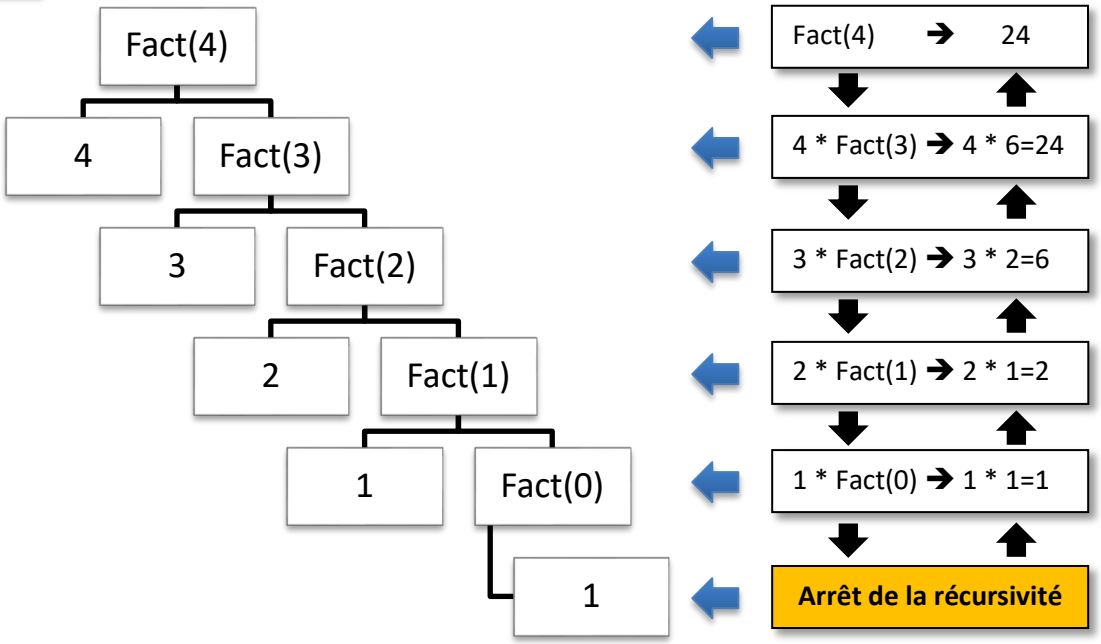
- Case de base ou d'arrêt : 0 ! = 1
- La relation de récurrence : n ! = n * (n-1)!

Cette relation donne la fonction récursive suivante :

```
Fonction Fact (n : Entier) retourne Entier

Début
     Si (n = 0) Alors
         Retourne 1
     Sinon
         Retourne n * Fact(n-1)
     Fin Si
Fin
```

Considérons le calcul de 4! par la fonction récursive définie ci-dessus :



2. La fonction Fibonacci

Le calcul des nombres de « Fibonacci » sont définis comme suit:

- Fibonacci (0) = 0,
- Fibonacci (1) = 1,
- Pour tout n > 1, Fibonacci (n) = Fibonacci (n-1) + Fibonacci (n-2)

a. Solution itérative

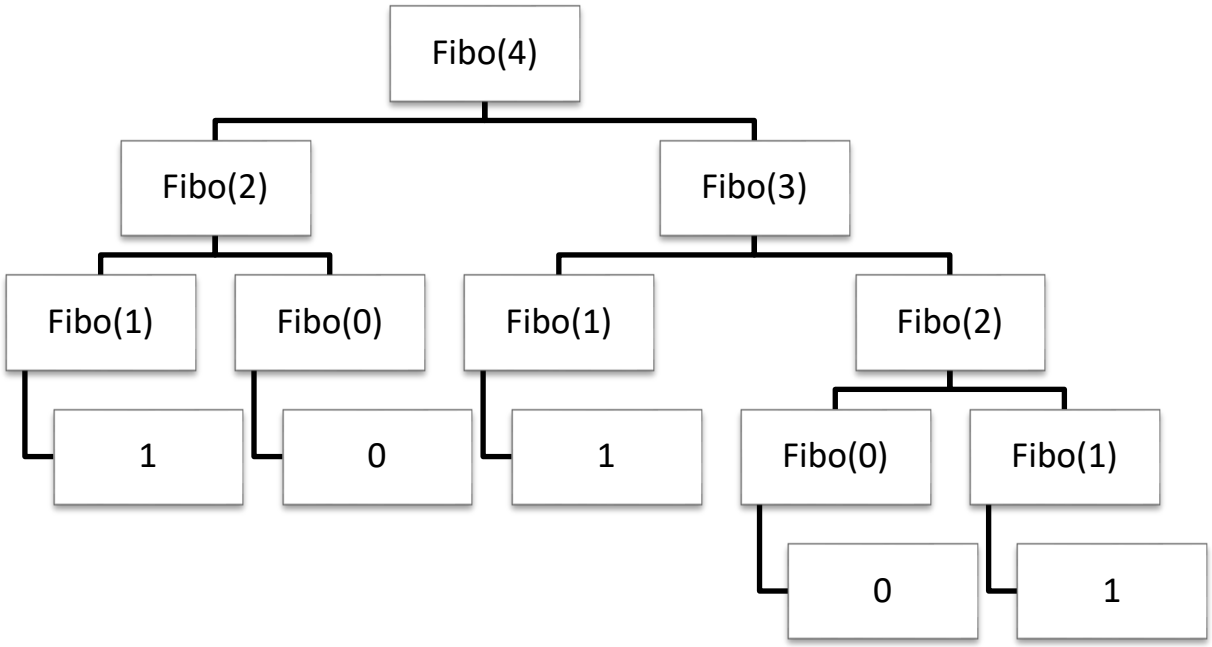
```
Fonction Fibo(n : entier) retourne entier
      f0,f0,i : entier
Début
      f0←0
      f1←1
      Pour i de 2 à n
          f ← f0 + f1
          f0 ← f1
          f1 ← f
      Fin Pour
      Retourne f
Fin fonction
```

b. Solution récursive

```
Fonction Fibo(n : entier) retourne entier

Début
      Si n<=1 alors
            Retourne n
      Sinon
            Retourne Fibo(n-1) + Fibo(n-2)
      Fin si
Fin fonction
```

Considérons le calcul de Fibonacci(4) par la fonction récursive définie ci-dessus :



Donc Fibonacci(4) = 3

LISTES CHAINEES

I. Introduction

Les tableaux sont des structures de données fréquemment utiliser pour résoudre les problèmes de stockage d'un ensemble important d'élément de même type en mémoire. Cependant, l'utilisation des tableaux présente des contraintes concernant la taille et l'espace mémoire occupée. Les tableaux doivent avoir une taille fixe définie au début du programme pour pouvoir allouer cette espace dans la mémoire centrale de manière contigüe. Ceci peut constituer un problème si la taille du tableau est très grande. En outre, il est possible que l'utilisation de cet espace réservé soit réduite, ce qui engendre une perte de la capacité mémoire de la machine. Par exemple, c'est une perte d'espace mémoire si nous créons un tableau de mille éléments puis nous utilisons seulement dix cases. Encore, si le tableau est saturé il n'est pas possible d'ajouter d'autre élément.

Pour résoudre les problèmes de mémoire mentionnés, il faut utiliser les structures abstraites de données, tel que les listes, les files, les piles, les arbres, les graphes. Dans ce chapitre, nous allons étudier les listes.

II. Définition

Les listes sont des structures de données abstraites qui permettent de créer des éléments non contigus et non limités. Une liste chainée est simplement une liste d'objet de même type dans laquelle chaque élément contient :

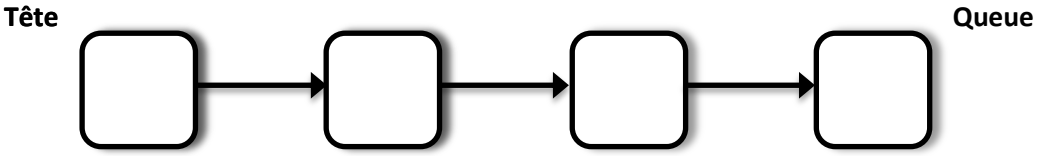
- Des informations relatives au fonctionnement de l'application. Par exemple des noms et prénoms de personnes avec adresses et numéros de téléphone pour un carnet d'adresse.

- L'adresse de l'élément suivant ou une marque de fin s'il n'y a pas de suivant. C'est ce lien via l'adresse de l'élément suivant contenue dans l'élément précédent qui fait la « chaine » et permet de retrouver chaque élément de la liste.

L'adresse de l'objet suivant peut être :

- Une adresse mémoire récupérée avec un pointeur (liste chainée dynamique).
- Un indice de tableau récupéré avec un entier (liste chainée statique).
- Une position dans un fichier : C'est le numéro d'ordre de l'objet dans le fichier.

Que la liste chainée soit bâtie avec des pointeurs ou des entiers, c'est toujours le terme de pointeur qui est utilisé : chaque élément « pointe » sur l'élément suivant, c'est-à-dire possède le moyen d'y accéder. Une liste chainée peut se représenter ainsi :



Les listes chainées dynamique donnent une solution pour remédier aux problèmes de l'optimisation de l'utilisation de la mémoire, mais elles ont aussi des inconvénients. Le tableau suivant résume les différences entre les tableaux et les listes.

	Tableau	Liste chainée
Taille en mémoire	La taille doit être définie lors de la création du tableau. La taille du tableau fait partie de la définition du tableau même pour les tableaux dynamiques.	La liste chainée dynamique n'a pas pour sa définition de nombre d'éléments. Ils sont ajoutés ou soustraits à la demande, pendant le fonctionnement du programme.
Supprimer un élément	impossible dans un tableau d'enlever une case. Il est éventuellement possible de masquer un élément mais pas de retirer son emplacement mémoire.	Aucun problème pour supprimer un élément de la liste et libérer la mémoire correspondante.
Accès à un élément	Le tableau permet d'accéder à chaque élément directement. C'est très rapide.	Pour accéder à un élément il faut parcourir la liste séquentiellement jusqu'à trouver l'élément, ça peut être long.
Tris	Les tris de tableau ne nécessitent pas de reconstruire les liens entre les emplacements mémoire du tableau. Il y a juste à manipuler les valeurs afin de les avoir dans l'ordre voulu.	Trier une chaine revient à construire une autre chaine en insérant dans l'ordre voulu chaque élément. Le mieux est de construire sa chaine en mettant les éléments dans l'ordre dès le départ plutôt que d'avoir à réorganiser l'ordre des éléments dans la chaine.

III. Les types de listes chainées

Les listes chainées sont des objets alloués dynamiquement d'une manière non contigüe dans la mémoire. Ainsi, il est essentiel de lier les éléments de la liste pour pouvoir la parcourir. Chaque liste chainée est caractérisée par :

- Adresse du premier élément : Tête de la liste.
- Adresse du dernier élément : Queue de la liste.
- Adresse de l'élément courant.
- Le nombre courant des éléments.

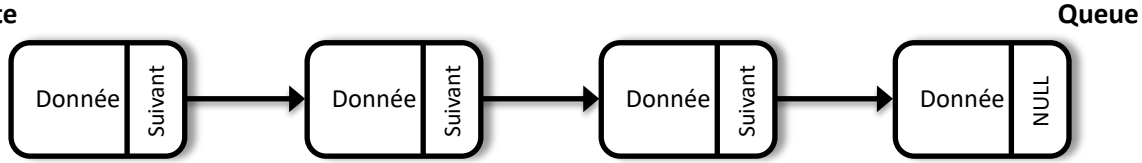
Un élément de la liste est une structure contenant :

- Un champ de données : qui peut être une donnée simple (entier, réel, etc.) ou composée tel qu'une structure.
- Un ou deux champs contenant une adresse : l'élément suivant et précédent.

Il existe deux modes de chainage des éléments d'une liste : le chainage simple et le chainage double.

1. Liste à chainage simple

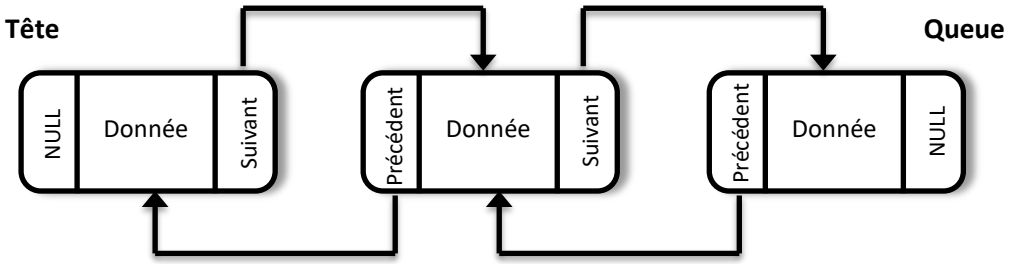
Elle consiste à ajouter à chaque élément un champ qui contient l'adresse mémoire de l'élément suivant dans la liste. Le schéma suivant donne exemple de chainage simple :



Ainsi, il est possible de parcourir la liste d'un élément à un autre dans un seul sens en commençant par la tête de la liste puis prendre l'adresse de l'élément suivant jusqu'à arriver à l'adresse vide (NULL).

2. Liste à chainage double

Elle consiste à ajouter à chaque élément deux champs, le premier contient l'adresse mémoire de l'élément suivant, le deuxième champ contient l'adresse de l'élément précédent. Le schéma suivant donne exemple de chainage simple :



Une liste à chainage double permet de parcourir la liste dans les deux sens de la tête vers la queue et inversement ce qui rend le parcours de la liste plus flexible et plus rapide.

IV. Les actions sur une liste chainée

Les actions sur liste chainée que ce soit à chainage simple ou double sont les mêmes. Il s'agit des fonctions ou procédures qui permettent la gestion de la liste :

Initialisation d'une liste : initialiser les pointeurs tête et queue à NULL et le nombre d'élément à zéro.

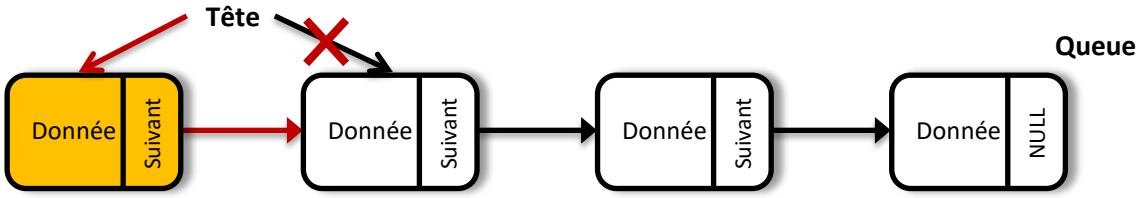
- Vérifier une liste si elle est vide ou non.
- Donner le nombre des éléments d'une liste.
- Ajout d'un élément : au début, au milieu et à la fin.
- Suppression d'un élément : au début, au milieu et à la fin.
- Modification d'un élément.
- La rechercher d'un élément : en utilisant une liste triée et non triée.
- Vider toute la liste.
- Copier une liste.

La différence entre les mises à jour pour les deux chainages c'est que pour le chainage double il faut mettre à jour les deux champs « Suivant » et « Précédent », tandis que pour le chainage simple il faut mettre à jour le champ « Suivant » seulement. Egalement, les champs « Tête », « Queue », et « Nombre » doivent être mis à jour selon le cas lors de l'ajout ou de la suppression.

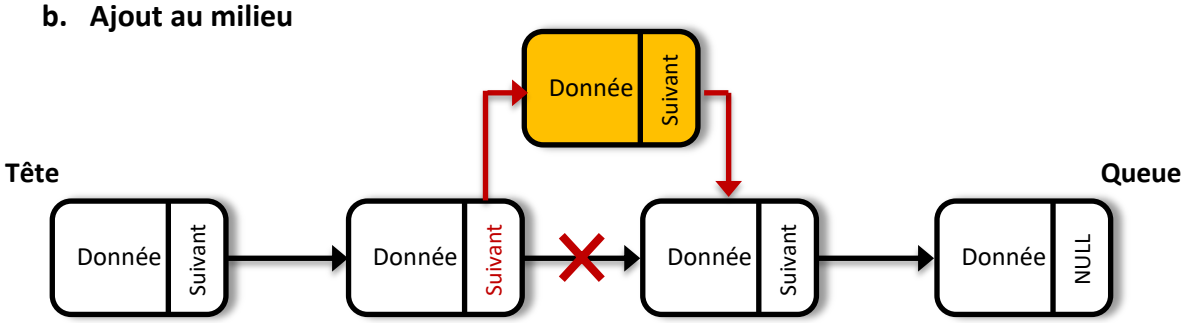
1. Ajout d'un élément

Pour ajouter un élément à la liste il faut le créer, c'est-à-dire lui allouer de l'espace mémoire puis l'ajouter à la position désirée soit au début, soit au milieu, soit à la fin de la liste. L'insertion de l'élément implique la mise à jour des champs adresses suivants et précédents qui sont concernés par l'ajout. Egalement, il faut mettre à jour le champ qui indique le nombre d'élément de la liste et mettre à jour selon le cas le pointeur de la tête ou le pointeur de la queue.

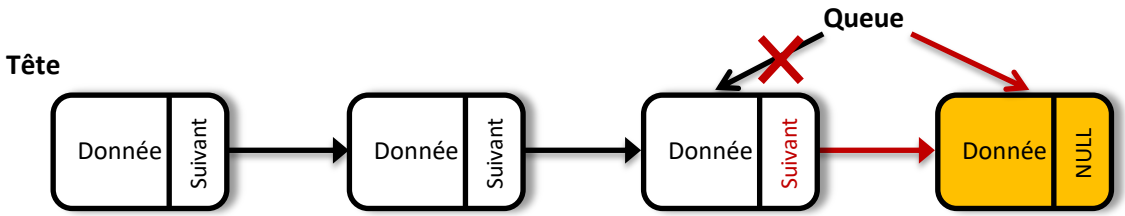
a. Ajout au début



b. Ajout au milieu



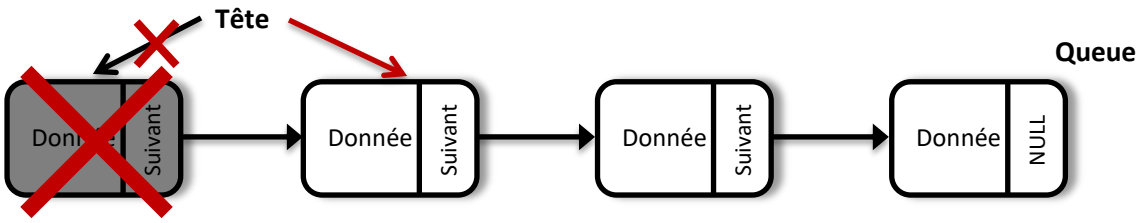
c. Ajout à la fin



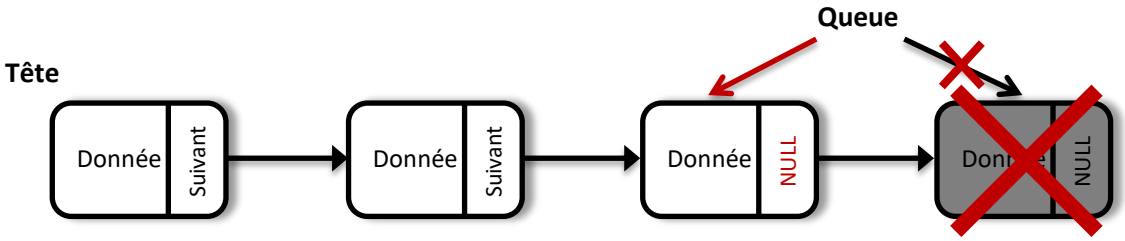
2. Suppression d'un élément

La suppression d'un élément de la liste implique des modifications comme l'ajout d'élément. La suppression du premier élément implique le changement du pointeur de la tête vers l'élément suivant. La suppression du dernier élément implique le changement du pointeur de la queue vers l'élément précédent. Si l'élément à supprimer est au milieu alors il faut changer le chainage avec l'élément précédent et l'élément suivant.

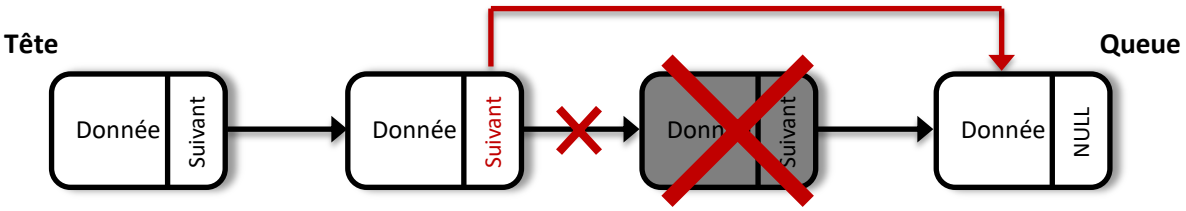
a. Supprimer la tête



b. Supprimer la queue



c. Supprimer au milieu



V. Quelques algorithmes

Dans ce qui suit nous allons présenter quelques algorithmes pour la gestion d'une liste à simple chainage. Il s'agit des opérations : d'ajout à la fin, de recherche, et de destruction de toute la liste. Chaque élément de la liste contient un entier et un pointeur qui pointe sur l'élément suivant. On suppose que les éléments de la liste ne sont pas triés. Les structures qui définissent la liste sont comme suit :

```
Structure Element
      Donnée : entier
      Suivant : pointeur sur Element
Fin structure

Structure Liste
      Tête : pointeur sur Element
      Queue : pointeur sur Element
      Taille : entier
Fin structure
```

1. Ajout d'un élément à la fin

```
Procédure Ajout_Liste_Fin(L : pointeur sur Liste, D : entier)
      Ptr : pointeur sur Element
Début
      Ptr←Allouer(Element)
      Ptr->Donnée←D
      Ptr->Suivant←NUll

      Si L->Queue<>NULL alors    // si la liste n'est pas vide
            L->Queue->Suivant←Ptr //chainage avec le nouvel élément
      Fin si

      L->Queue←Ptr
      L->Taille← L->Taille+1

      Si L->Tete=NULL alors // si la liste est vide
            L->Tete←L->Queue
      Fin si
Fin procédure
```

2. Recherche d'un élément

```
Fonction Recherche_Liste(L : Liste, D : entier) retourne entier
      Ptr : pointeur sur Element
      Pos : entier
Début
      Ptr←L.Tête
      Pos←0
      Tant que Ptr<>NULL et Ptr->Donnée<>D
            Ptr←Ptr->Suivant
            Pos←Pos+1

      Fin Tant que
      Si Ptr<>Null alors
             Retourne Pos
      Sinon
            Retourne -1
      Fin si
Fin fonction
```

3. Détruire toute la liste

```
Procédure Détruire_Liste(L : Pointeur sur Liste)
      Ptr, Suiv : pointeur sur Element
Début
      Ptr←L->Tête
      Tant que Ptr<>Null
            Suiv←Ptr->Suivant
            Libérer(Ptr)
            Ptr←Suiv
      Fin Tant que
      L->Tête←NULL
      L->Queue←NULL
      L->Taille←0
Fin Procédure
```