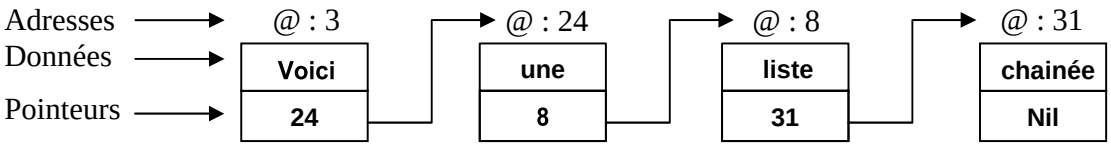
1. Les listes chaînées

Une **liste chaînée** est une structure linéaire qui n'a pas de dimension fixée à sa création. Ses éléments de même type sont éparpillés dans la mémoire et reliés entre eux par des pointeurs. Sa dimension peut être modifiée selon la place disponible en mémoire. La liste est accessible uniquement par sa tête de liste c'est-à-dire son premier élément.
Pour les listes chaînées la séquence est mise en œuvre par le pointeur porté par chaque élément qui indique l'emplacement de l'élément suivant. Le dernier élément de la liste ne pointe sur rien (*Nil*).
On accède à un élément de la liste en parcourant les éléments grâce à leurs pointeurs.
Chaque élément de la liste contient :
- des informations sur l'élément
- un pointeur sur un autre élément de la liste, ou un pointeur NULL s'il n'y a pas d'élément suivant.

4.1. Représentation en mémoire d'une liste

Soit la liste chaînée suivante (@ indique que le nombre qui le suit représente une adresse) :



En allocation dynamique, les emplacements mémoires sont dispersés en mémoire centrale. L'espace est réservé au fur et à mesure des créations des éléments de la liste. La seule limite étant la taille de la mémoire centrale.
Pour accéder au troisième élément de la liste il faut toujours débuter la lecture de la liste par son premier élément dans le pointeur duquel est indiqué la position du deuxième élément. Dans le pointeur du deuxième élément de la liste on trouve la position du troisième élément…

Pour ajouter, supprimer ou déplacer un élément il suffit d'allouer une place en mémoire et de mettre à jour les pointeurs des éléments.
Il existe différents type de listes chainées :
Il existe différents types de listes chaînées :

- **listes simplement chainées** (comme ci-dessus) constituée d'éléments reliés entre eux par des pointeurs.
- **Liste chaînée ordonnée** où l'élément suivant est plus grand que le précédent. L'insertion et la suppression d'élément se font de façon à ce que la liste reste triée.
- **Liste doublement chaînée** où chaque élément dispose non plus d'un pointeur mais de deux pointeurs pointant respectivement sur l'élément précédent et l'élément suivant. Ceci permet de lire la liste dans les deux sens, du premier vers le dernier élément ou inversement.
- **Liste circulaire** où le dernier élément pointe sur le premier élément de la liste. S'il s'agit d'une liste doublement chaînée alors de premier élément pointe également sur le dernier.
- **Les piles**

- **Les files**

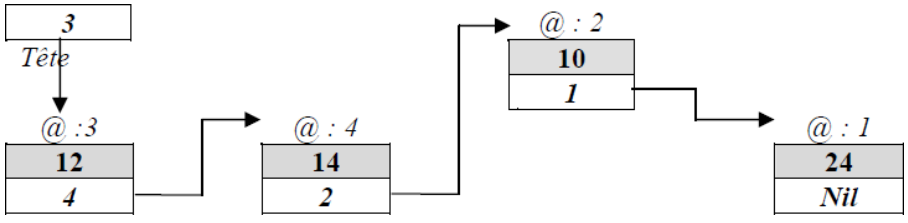
2. Opérations sur les listes chaînées

Une liste chaînée simple est composée :

- d'un ensemble d'éléments tel que chacun :
  - est rangé en mémoire à une certaine adresse,
  - contient une donnée (*Info*),
  - contient un pointeur, souvent nommé *Suivant*, qui contient l'adresse de l'élément suivant dans la liste,
- d'une variable, appelée *Tête*, contenant l'adresse du premier élément de la liste chaînée.

Le pointeur du dernier élément contient la valeur *Nil*. Dans le cas d'une liste vide le pointeur de latête contient la valeur *Nil*. Une liste est définie par l'adresse de son premier élément.

Exemple :



Le 1er élément de la liste vaut 12 à l'adresse 3 (début de la liste chaînée)
Le 2eme élément de la liste vaut 14 à l'adresse 4 (car le pointeur de la cellule d'adresse 3 est égal à 4)

Le 3emeélément de la liste vaut 10 à l'adresse 2 (car le pointeur de la cellule d'adresse 4 est égal à 2)
Le 4$_e$ élément de la liste vaut 24 à l'adresse 1 (car le pointeur de la cellule d'adresse 2 est égal à 1)
Si P a pour valeur 3 Si P a pour valeur 2
P^.Info a pour valeur 12P^.Info a pour valeur 10
P^.Suivant a pour valeur 4 P^.Suivant a pour valeur 1

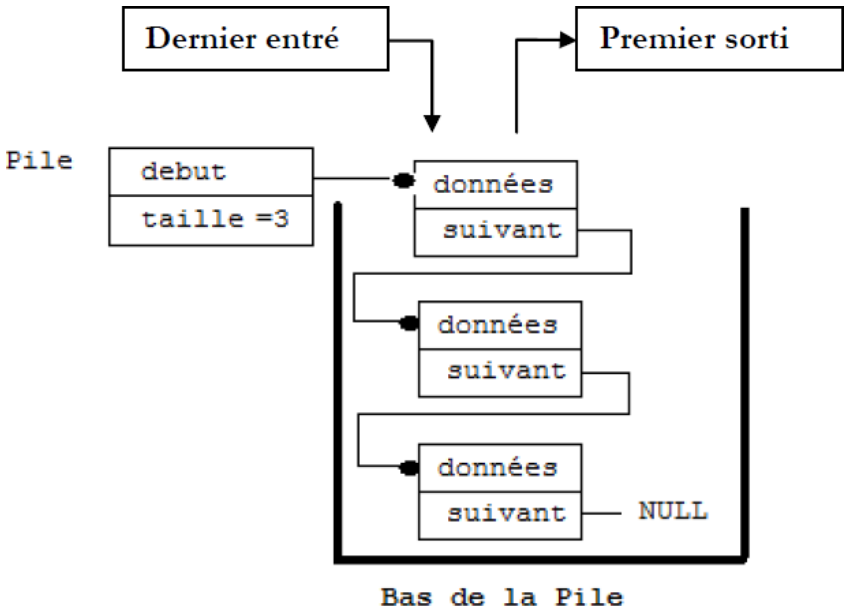
Traitements de base d'utilisation d'une liste chaînée simple
Il faut commencer par définir un type de variable pour chaque élément de la chaîne. En langagealgorithmique ceci se fait comme suit :

Type Element = Structure
valeur : entier
Suivant : Element
Fin structure
Type Liste = ^Element
Variables Tete, P : Liste
	Typedef struct element {
	Intvaleur;
	element *Suivant;
	};
	typedef element *Liste;
	int main()
	{
	ListeTete,P;
	}

Les traitements des listes sont les suivants :
- Créer une liste.
- Ajouter un élément.
- Supprimer un élément.
- Modifier un élément.
- Parcourir une liste.
- Rechercher une valeur dans une liste.

3. LES PILES

La pile est une structure de données, qui permet de stocker les données dans l'ordre LIFO (Last In First Out = Dernier Entré Premier Sorti). L'insertion des données se fait donc toujours au début de la liste ***(sommet de la pile)*** (i.e. par le haut de la pile), donc le premier élément de la liste est le dernier élément inséré, sa position est donc en haut de la pile.



Pour permettre les opérations sur la pile, nous allons sauvegarder certains éléments Le premier élément de la pile, qui se trouve en haut de la pile, va nous permettre de réaliser l'opération de récupération des données situées en haut de la pile.

Pour réaliser cela, une autre structure sera utilisée (ce n'est pas obligatoire, des variables peuvent être utilisées). Voici sa composition :

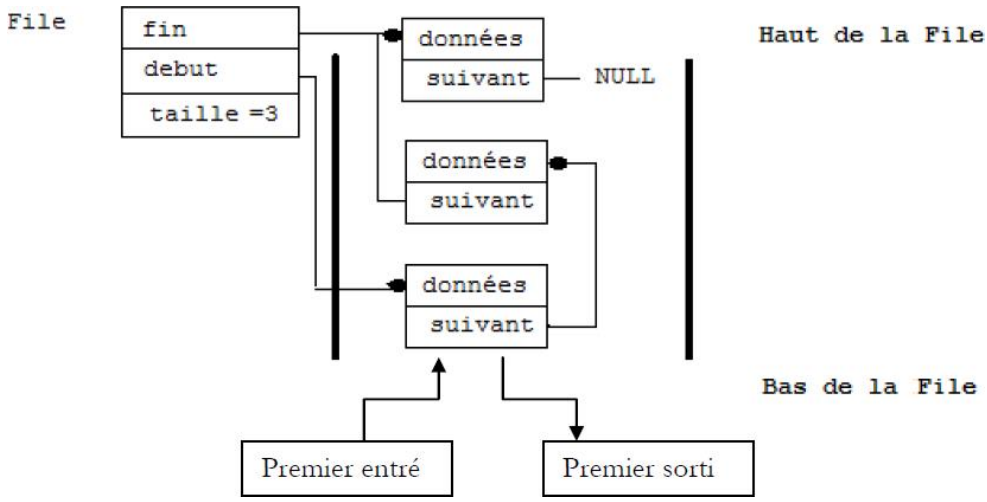
```
typedef struct pile {
Element *debut;
int taille;
};
typedef struct pile Pile ;
```

Le pointeur début contiendra l'adresse du premier élément de la liste. La variable taille contient le nombre d'éléments.

Quatre opérations abstraites sont définies sur le type Pile :
Empiler : Pile X E →Pile
Dépiler : Pile →Pile
Sommet : Pile →E
est- vide? : Pile →booleen
Le rôle de l'opération empiler est d'ajouter un élément en sommet de pile, celui de dépiler de supprimer
Le sommet de pile et celui de sommet de renvoyer l'élément en sommet de pile.
L'opération est - vide? Indique si une pile est vide ou pas.

4. LES FILES

La File diffère de la Pile dans sa façon de gérer les données. En effet, la file permet de stocker les données dans l'ordre FIFO (First In First Out = Premier Entré Premier Sorti). L'insertion des données se fait aussi par en haut de la File, mais la sortie ne se faut plus par le haut comme pour la Pile mais par le bas.



Pour manipuler une File, nous sauvegardons son premier élément, son dernier élément ainsi que sa taille (nombre d'éléments qu'elle contient).

Pour réaliser cela, une autre structure sera utilisée (ce n'est pas obligatoire, des variables peuvent être utilisées). Voici sa composition :

```
typedef struct file{
Element *debut;
Element *fin;
int taille;
};
typefedstruct file File ;
```

Le pointeur début contiendra l'adresse du premier élément de la liste.
Le pointeur fin contiendra l'adresse du dernier élément de la liste. La variable taille contient le nombre d'éléments.

Quatre opérations sont définies sur le type File :
enfiler : File X E →File
defiler : File →File
dernier : File →E
est- vide? : File →booleen
L'Opération ***enfiler*** a pour rôle d'ajouter un élément en queue de file, et l'opération défiler supprime l'élément en tête de file. ***dernier*** retourne le dernier élément de la file et est-vide?

Indique si une file est vide ou pas. Notez que les signatures de ces opérations sont, au mot «File» près, identiques à celles des opérations du type abstrait Pile. Ce sont bien les axiomes qui vont différencier ces deux types abstrait

5. Manipulation des listes, Piles et files

Initialisation

Nous pouvons modéliser un **nœud** de la liste liée en utilisant une structure comme Suit:

```
typedef struct node_t {
   int val;
   struct node * next;
};
```

La structure de nœud comporte deux membres:

- Les données qui stockent les informations
- Un pointeur suivant qui contient l'adresse du prochain nœud.

Notez que nous définissons la structure de manière récursive, ce qui est possible dans C. Désignons notre type de nœud node_t.
Maintenant, nous pouvons utiliser les nœuds. Créons une variable locale qui pointe vers le premier élément de la liste (appelé tête ou Head (sommet)).

```
node_t * head = NULL;
head = (node_t*) malloc (sizeof (node_t));
if (head == NULL) {
return 1;
}

head->val = 0;
head->next = NULL;
```

Nous pouvons mettre dans le champ val du nœud head la taille de la liste (nombre d'éléments)

Ajouter un nœud à la fin de la liste chainée

Pour itérer sur tous les membres de la liste chainée, nous utilisons un pointeur appelé ***current***. Nous l'établissons pour commencer à partir de la tête, puis à chaque étape, nous avançons le pointeur vers l'élément suivant dans la liste, jusqu'à l'arrivée au dernier élément.

```
void push_fin (node_t *head, int val)
 {
   node_t *current = head;
   while (current->next != NULL) {
   current = current->next;
  }
    /* Nous ajoutons un nouveau élément */
    current->next = (node_t*)malloc(sizeof(node_t));
    current->next->val = val;
```

```
        current->next->next = NULL;
    head->val=head->val+1; /* On ajoute 1 à la taille de la liste*/
}
```

Ajouter un nœud au début de la liste

```
    void push_debut (node_t *head, int val) {
        node_t *new_node;
        new_node = (node_t*)malloc(sizeof(node_t));
        new_node->val = val;
        new_node->next = head->next;
        head->next = new_node;
        head->val=head->val+1; /* On ajoute 1 à la taille de la liste*/
    }
```

Suppression du dernier nœud de la liste

```
int remove_last (node_t * head)
{

    /* le cas d'un seul élément dans la liste */
    if (head->next == NULL)
        return 0;


    /* aller vers le dernier élément */
    node_t * current = head;
    while (current->next->next != NULL) {
        current = current->next;
    }

    /* on pointe sur le dernier élément*/
    free(current->next);
    current->next = NULL;
    head->val=head->val-1; /*On change la taille de la liste par -1*/
    return 0;
}
```

Suppression du premier nœud de la liste

```
int Pop(node_t* head)
{
if(head->next == NULL)
return 0;
node_t *front = head->next;
head->next = head->next->next;
front->next = NULL;
/* is this the last node in the list */
if(front == head)
```

```
 head = NULL;
 free(front);
  head->val=head->val-1; /*On change la taille de la liste par -1*/
 return 0;
 }
```

Supprimer un nœud spécifique (par son valeur par exemple)

```
int remove_by_val(node_t *head,int valeur){
node_t *current =head;
node_t *temp_node=NULL;

while (current->next!=NULL)
{
if (current->next->val==valeur)
{
temp_node= current;
current->next =temp_node->next->next;
}
current= current->next;
}

return 0;
}
```

Imprimer la liste

```
 void print_list(node_t *head)
      {
     node_t * current = head->next;
 if (head->next==NULL)
  printf("il n'ya aucun noeud dans la liste \n");
  else{
      printf("la liste contient %d elements \n",head->val);
         while (current != NULL) {
         printf("%d\n", current->val);
         current = current->next;}
     }
 }
```

Programme de test global:

```
#include<stdio.h>
#include<stdlib.h>
typedef struct node {
    int val;
    struct node * next;
} node_t;

/*Ajout à la fin*/

void push_fin (node_t *head, int val)
 {
   node_t *current =  head;
   while (current->next != NULL) {
   current = current->next;
 }
    /* Nous ajoutons un nouveau élément */
    current->next = (node_t*)malloc(sizeof(node_t));
    current->next->val = val;
   current->next->next = NULL;
   head->val=head->val+1;
}
/*Ajout au début*/
void push_debut (node_t *head, int val) {
    node_t *new_node;
    new_node = (node_t*)malloc(sizeof(node_t));
    new_node->val = val;
    new_node->next = head->next;
    head->next = new_node;
    head->val=head->val+1;
}
/*Suppression du dernier élément*/
int remove_last (node_t * head)
{

    /* le cas d'un seul élément dans la liste */
    if (head->next == NULL)
        return 0;


    /* aller vers le dernier élément */
    node_t * current = head;
    while (current->next->next != NULL) {
        current = current->next;
    }
```

```
    /* on pointe sur le dernier élément*/
    free(current->next);
    current->next  =  NULL;
    head->val=head->val-1;
    return 0;

}
/*Suppression du premier élément*/
int Pop(node_t* head)
{
if(head->next == NULL)
return 0;
node_t *front = head->next;
head->next = head->next->next;
front->next = NULL;
/* is this the last node in the list */
if(front == head)
head = NULL;
free(front);
 head->val=head->val-1;
return 0;
}
int remove_by_val(node_t *head,int valeur){
node_t *current =head;
node_t *temp_node=NULL;

while (current->next!=NULL)
{
if (current->next->val==valeur)
{
temp_node= current;
current->next =temp_node->next->next;
}
current= current->next;
}

return 0;
}

void print_list(node_t *head)
     {
    node_t * current = head->next;
if (head->next==NULL)
 printf("il n'ya aucun noeud dans la liste \n");
 else{
```

```
     printf("la liste contient %d elements \n",head->val);
        while (current != NULL) {
       printf("%d\n", current->val);
       current = current->next;}
   }
}
int main()
{
node_t * head = NULL;
head = (node_t*) malloc (sizeof (node_t));
head->val = 0;
head->next = NULL;
push_fin(head,3);
print_list(head);
printf("****************** \n");
push_fin(head,9);
print_list(head);
printf("****************** \n");
push_fin(head,10);
print_list(head);
printf("****************** \n");
push_debut(head,8);
print_list(head);
printf("****************** \n");
push_debut(head,15);
print_list(head);
printf("****************** \n");
remove_by_val(head,9);
print_list(head);
printf("****************** \n");
Pop(head);
print_list(head);
printf("****************** \n");
Pop(head);
print_list(head);
printf("****************** \n");
Pop(head);
print_list(head);
printf("****************** \n");
Pop(head);
print_list(head);
printf("****************** \n");
Pop(head);
print_list(head);
}
```

Résultats d'exécution du programme

```
la liste contient 1 elements
3
******************
la liste contient 2 elements
3
9
******************
la liste contient 3 elements
3
9
10
******************
la liste contient 4 elements
8
3
9
10
******************
la liste contient 5 elements
15
8
3
9
10
******************
la liste contient 5 elements
15
8
3
10
******************
la liste contient 4 elements
8
3
10
******************
la liste contient 3 elements
3
10
******************
la liste contient 2 elements
10
******************
il n'ya aucun noeud dans la liste
******************
il n'ya aucun noeud dans la liste
```