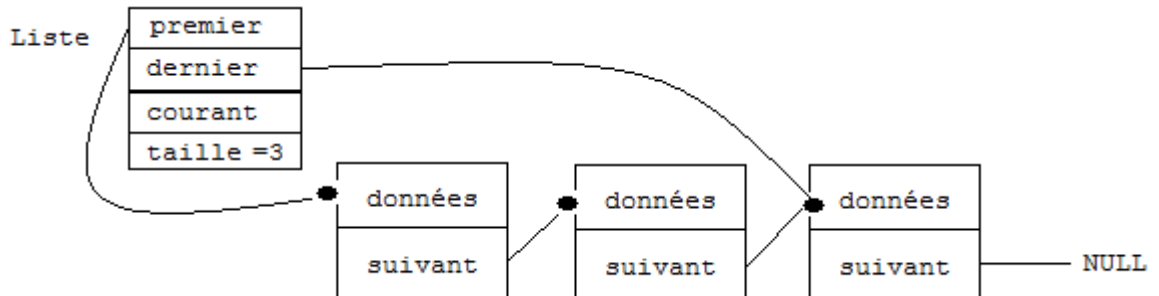


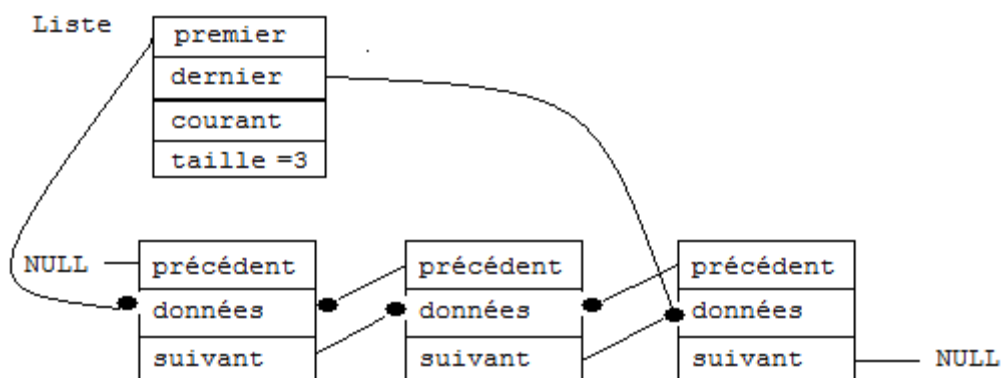


### 3. Aperçu des différentes formes de chaînes

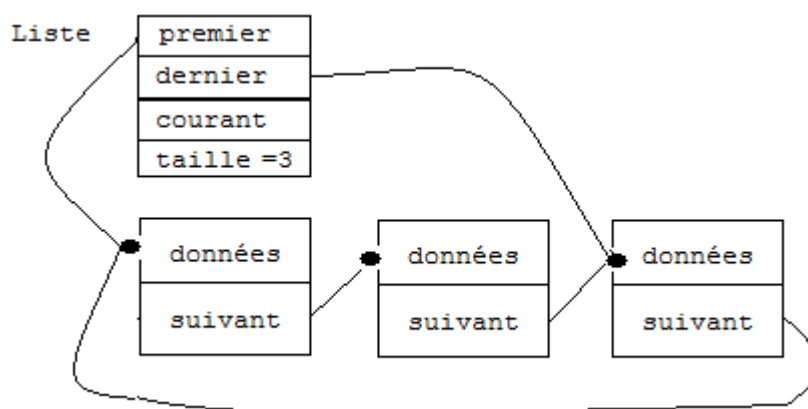
#### 3.1. Une liste simplement chaînée



#### 3.2. Une liste doublement chaînée

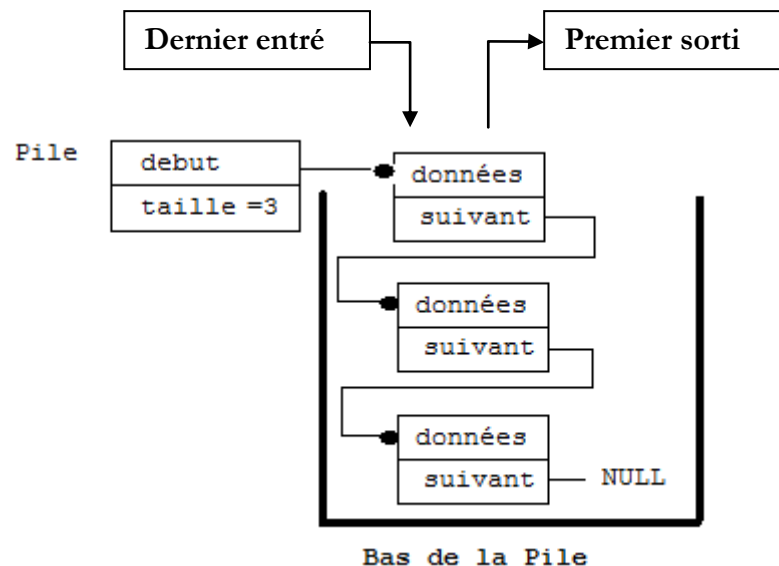


#### 3.3. Une liste circulaire



### 3.4. Une pile

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 (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.

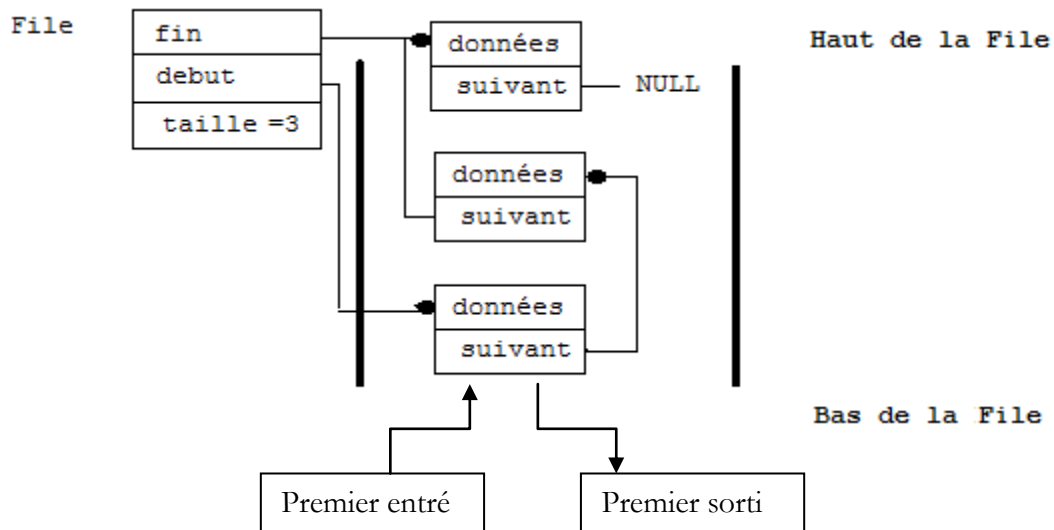
Pour réaliser cela, une autre structure sera utilisée (ce n'est pas obligatoire, des variables peuvent être utilisées). Voici sa composition :

```
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.

### 3.5. Une file

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 fait 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;
};
typedef struct 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.

**Dans la suite du chapitre nous ne traiterons que le cas d'une liste simplement chaînée.**

#### 4. Déclaration d'une liste simplement chaînée

Dans l'exemple de la liste des patients, nous auront les déclarations suivantes :

```
typedef char ch15 [16] ;

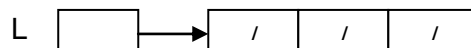
typedef element * PElement;
typedef struct element {
    ch15 Nom;
    int age;
    PElement * Suivant;
} Element;
```

```
typedef struct Liste {
    PElement Premier;
    PElement Dernier;
    PElement Courant; /* facultatif */
    int taille ;
};
```

Le pointeur Courant ici permet de se déplacer dans la liste, mais il n'est pas nécessaire, une variable du même type pourra jouer ce rôle.

## 5. Initialisation d'une liste

```
/* Liste vide */
Liste L;
L->Premier = NULL;
L->Dernier = NULL;
L->taille = 0 ;
```



Pour savoir si une liste est vide, il suffit de vérifier si son premier élément vaut NULL.

## 6. Insérer un élément dans une liste vide

Étapes :

- allocation de la mémoire pour le nouvel élément
- remplir le champ de données du nouvel élément
- le pointeur suivant du nouvel élément pointera vers NULL (vu que l'insertion est faite dans une liste vide on utilise l'adresse du pointeur début qui vaut NULL)
- les pointeurs début et fin pointeront vers le nouvel élément
- la taille est mise à jour

Nous insérons par exemple l'élément suivant :

```
Element patient = { « Dupond », « 45 », NULL } ;
```

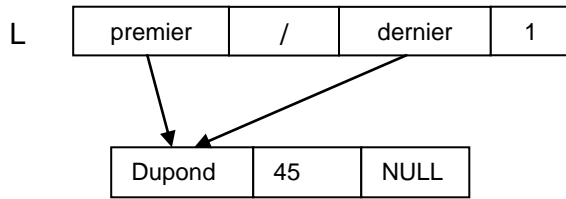
Ci-dessous le script qui réalise ces étapes d'insertion :

```
/* insertion dans une liste vide */
Element * nouveauElement ;

if ( (nouveauElement = (Element *) malloc (sizeof (Element))) == NULL )
    return -1;
if ((nouveauElement->nom = (char *) malloc (16 * sizeof (char))) == NULL)
    return -1;
if ((nouveauElement->age = (int *) malloc (sizeof (int))) == NULL)
    return -1;

nouveauElement->nom = patient.nom ;
nouveauElement->age = patient.age ;
nouveauElement->suivant = patient.suivant ; /* NULL */

L->Premier = nouveauElement;
L->Dernier = nouveauElement;
L->taille++ ;
```

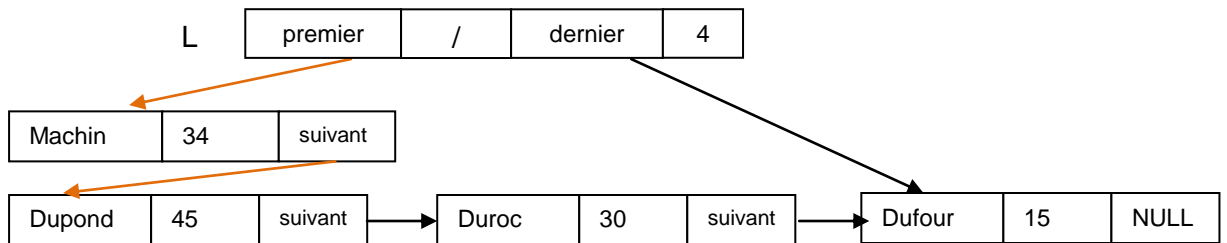


Le pointeur courant `L->Courant` va nous permettre de parcourir la liste ainsi définie.

### 7. Insertions d'éléments dans une liste non vide

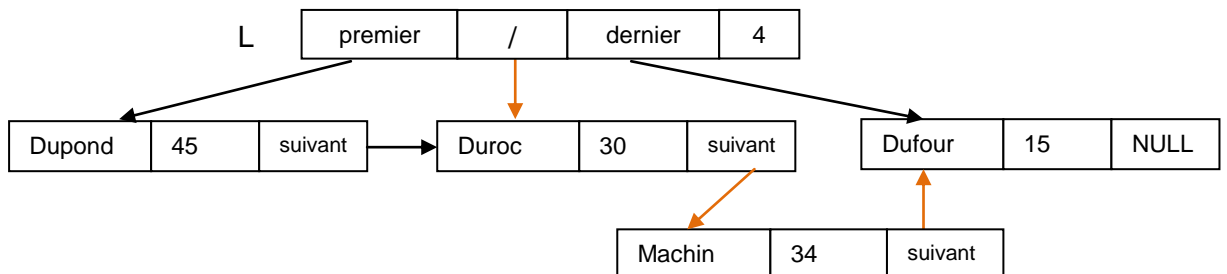
A chaque nouvelle insertion d'un élément, il faut allouer de l'espace mémoire pour ce dernier avant de l'insérer dans la liste.

- En tête de liste :



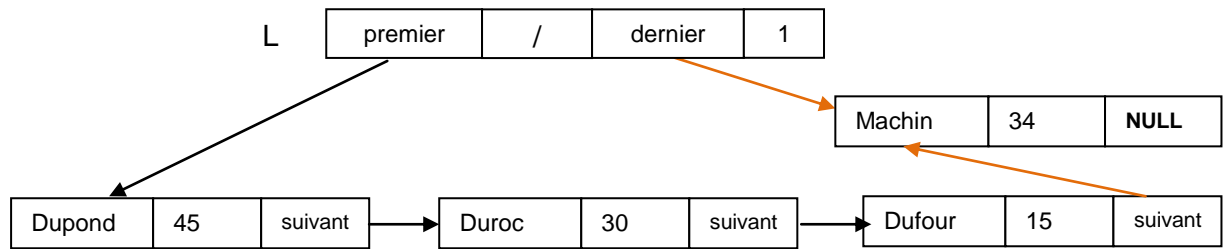
Les flèches rouges indiquent les pointeurs à modifier pour insérer un élément en tête de liste.

- Dans la liste :



Les flèches rouges indiquent ici l'ensemble des pointeurs à modifier pour insérer un élément dans la liste.

- En fin de liste :



Les flèches rouges indiquent ici aussi les pointeurs à modifier pour pouvoir insérer un élément à la fin de la liste.