STRUCTURES DE DONNÉES
MIP — S4

Pr. K. Abbad & Pr. Ad. Ben Abbou & Pr. A. Zahi
Département Informatique
FSTF
2019-2020

Séance 4

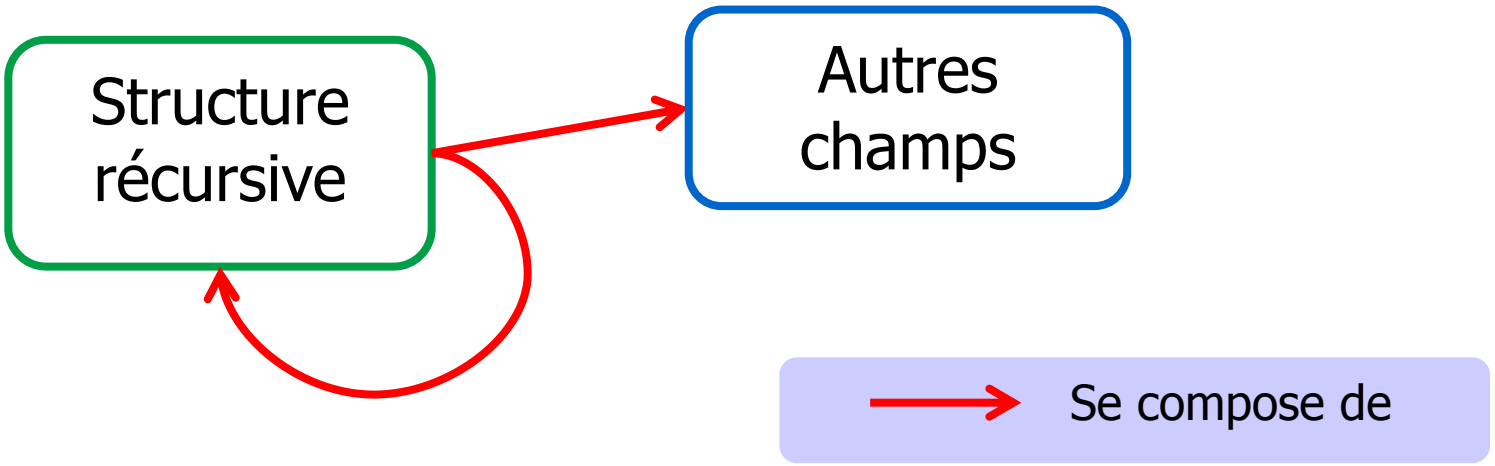
- Les structures récursives
- Les Piles

Séance 4.1 - Les structures récursives

- Définition
- Représentation

Structures récursives — *Définition*

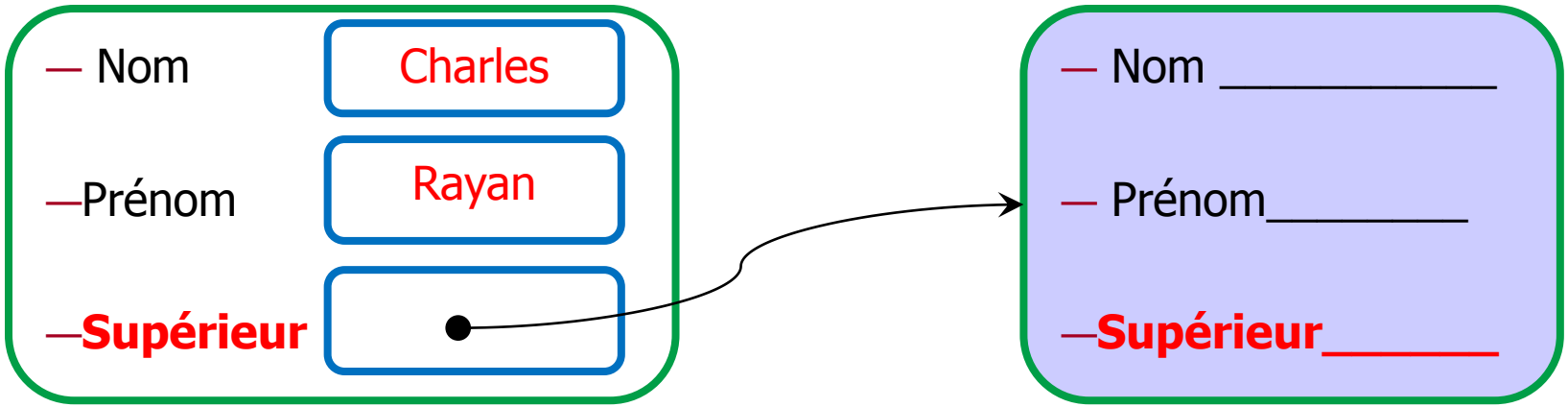
- Une structure **R** est dite récursive si elle contient un champ de type R.



- Une structure récursive permet de représenter des entités naturellement récursives.

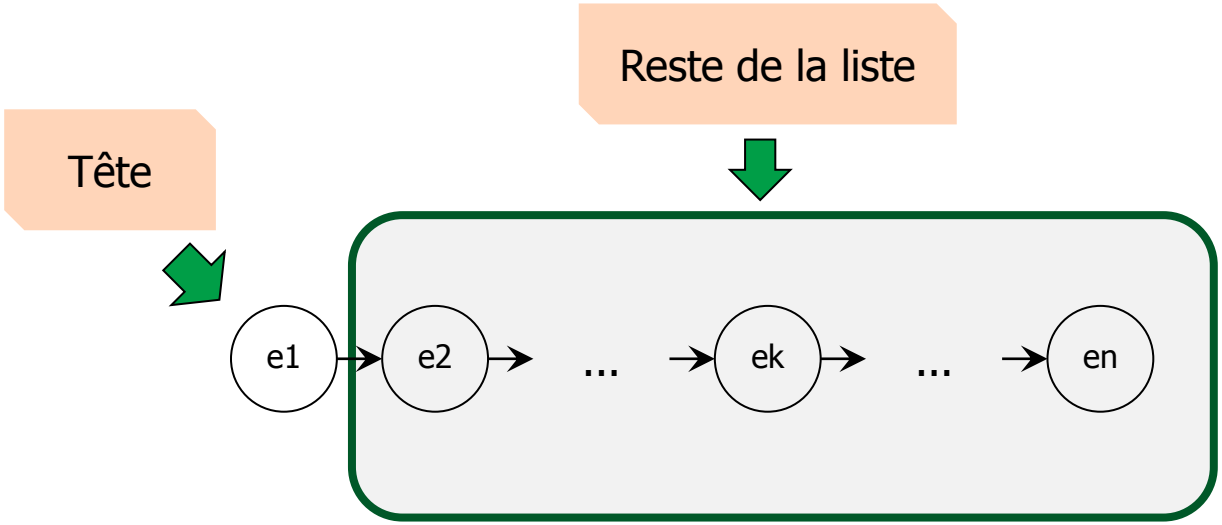
Structures récursives — *Exemples*

- Un **militaire** est caractérisé par :
  - Un *nom* et un *prénom*
  - Un supérieur hiérarchique unique qui est un **militaire.**



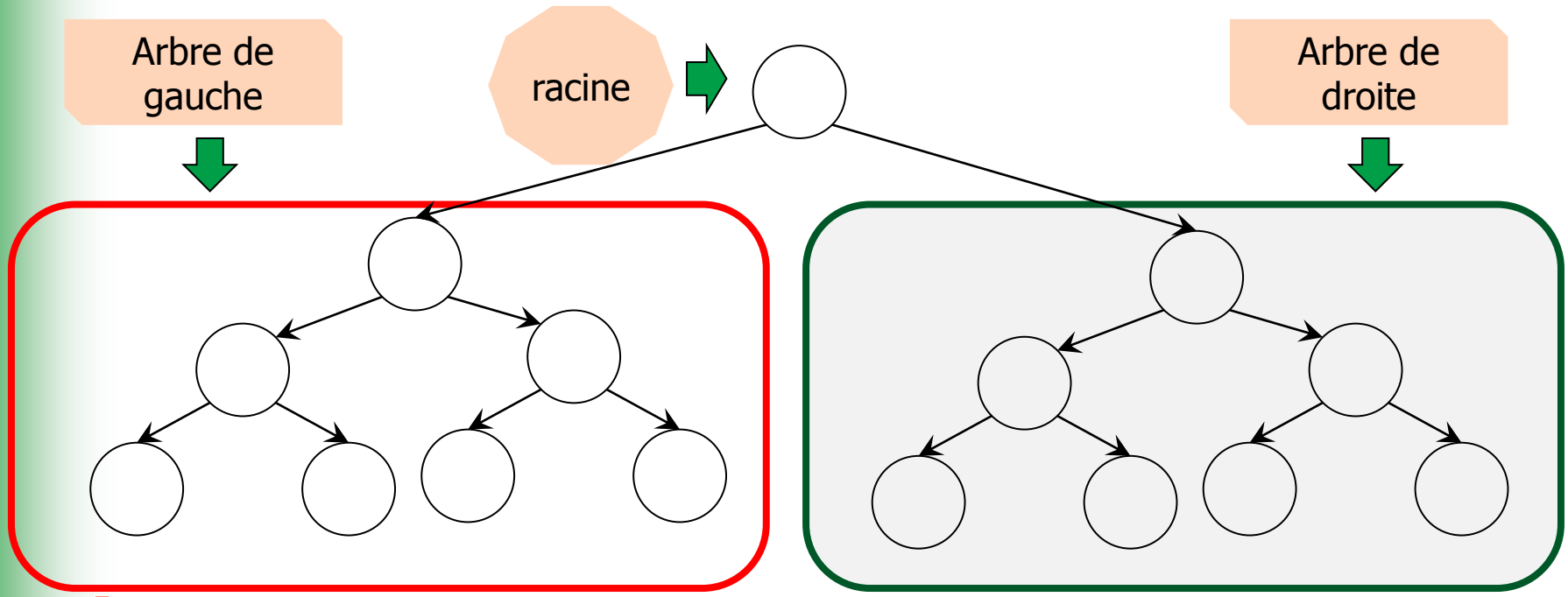
Structures récursives — *Exemples*

- Une **liste** d'éléments est composée par:
  - Une *tête* qui est un élément est d'une liste.
  - Une *queue* qui est une **liste.**



Structures récursives — *Exemples*

- Un **arbre binaire** est défini par:
  - Une racine qui est un élément.
  - Une branche gauche qui est un **arbre.**
  - Une branche droite qui est un **arbre.**



Structures récursives — *Implémentation*

- Le champ qui exprime la récursivité doit être un **pointeur** sur la structure en question.
- **Syntaxe (forme 1)**

```
typedef struct nom_structure* ptr_nomstructure;

typedef struct {
        ...
         ptr_nomstructure champ_recursif;

} nom_structure;
```

- **Syntaxe (forme 2)**

```
typedef struct Nom{
        ...
         struct Nom * champ_recursif;

} nom_structure;
```

Structures récursives — *Implémentation*

- Exemple du militaire(1ere forme)

```
typedef struct militaire* ptr_mil;

typedef struct {
        char nom[20];
        char prenom[20];
        ptr_mil supérieur;

} militaire
```

Structures récursives — *Implémentation*

- Exemple du militaire(2 eme forme)

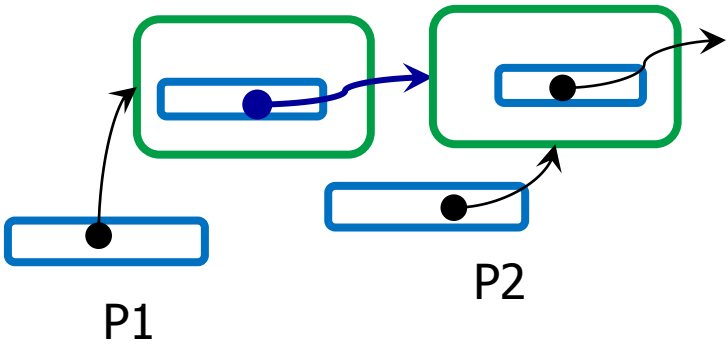
```
typedef struct militaire{
        char nom[20];
        char prenom[20];
        struct militaire *supérieur;

} Militaire;
typedef struct militaire* ptr_mil;
```

Il faut affecter une adresse valide au champ récursif lors de la création.

Structures récursives — *Implémentation*

- Exemple

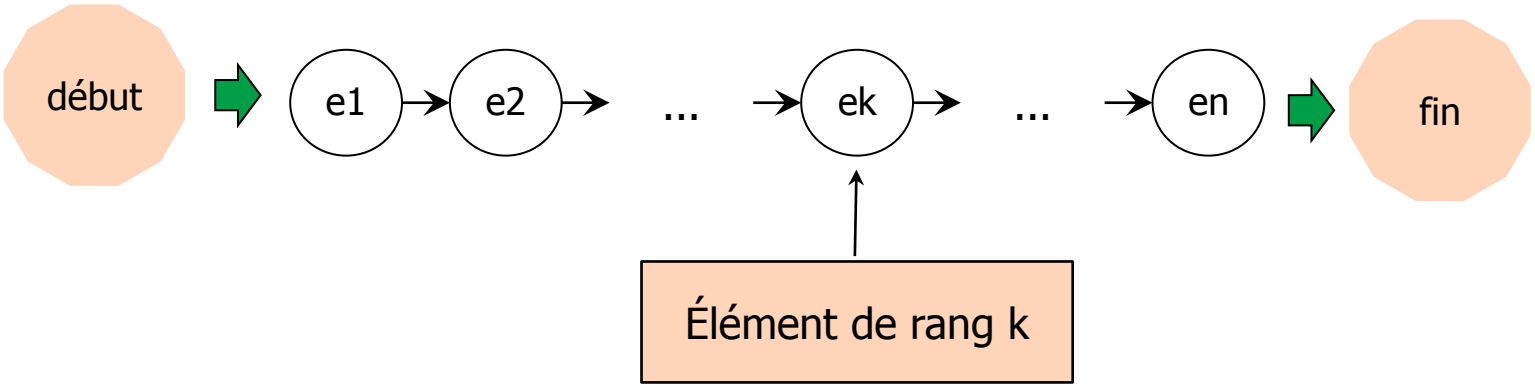
```
typedef struct militaire* ptr_mil;
struct {
        char nom[20];
        char prenom[20];
         ptr_mil supérieur;
} militaire;
Main(){
ptr_mil P1,P2;
P1=(ptr_mil )malloc(sizeof(militaire));
P2=(ptr_mil )malloc(sizeof(militaire));
P1->superieur =P2;
….
}
```



Représentation chainée

Listes —*Définition*

- Une *liste* est une suite finie d'éléments *ordonnés* selon leur place:
- Les éléments sont de même type et sont repérés par leur *rang* dans la liste.
- Une liste peut être vide.



Listes —*Caractérisation*

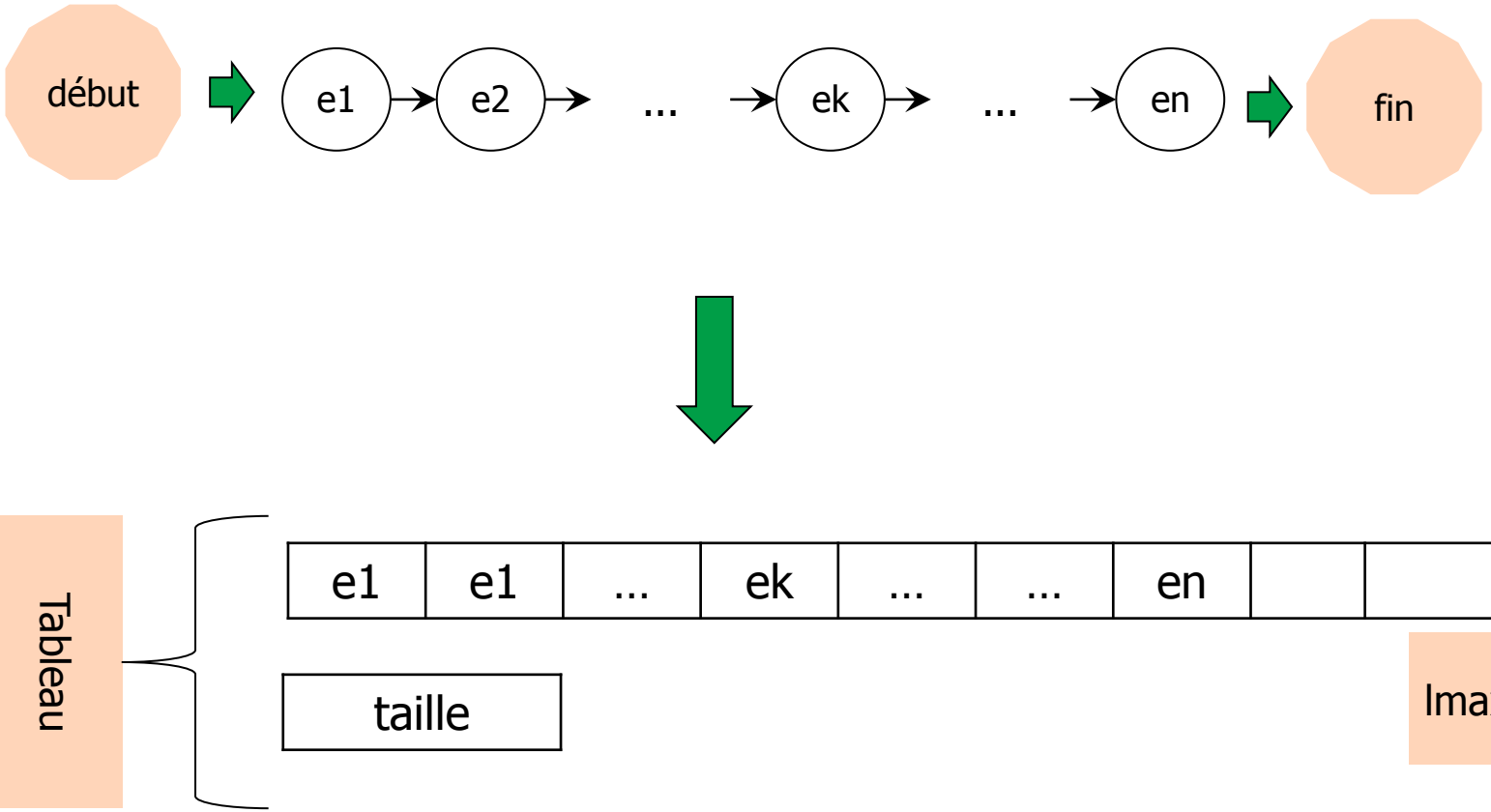
- Une liste est un type caractérisé par les opérations suivantes:
  - Création /initialisation
  - Ajout d'un élément à n'importe quelle position.
  - Suppression d'un élément situé à n'importe quelle position.
  - Test de la liste vide.

- Cas particuliers
  - Pile — Organisation **LIFO** (Last In First Out)
    - L'ajout et le retrait se font à partir du sommet (tête)
  - File — Organisation **FIFO** (First In First Out)
    - L'ajout se fait à la queue.
    - Le retrait à partir de la tête.

Listes —*Représentation*

- Deux représentations sont possibles:
  - Représentation *contigüe* à l'aide de tableaux d'éléments.
  - Représentation *chaînée* à l'aide de *pointeurs* qui établissent le chainage entre les éléments:

Listes —*Représentation contigüe*



Listes —*Représentation contigüe*

- Une structure qui contient :
  - Un tableau, éventuellement dynamique, d'éléments
  - La taille maximale;
  - La taille réelle;

```
typedef struct {

        Elem* t;
        int lmax;
        int taille;
} LISTE
```

- Une liste est identifiée par une variable de type LISTE.

Listes —*Représentation contigüe*

- La relation de *succession* est mise en œuvre par le mécanisme d'indexation.
  - chaque élément est identifié par un indice dans le tableau.

Accès direct et rapide aux éléments.

- La longueur maximale de la liste doit être déterminée au préalable.

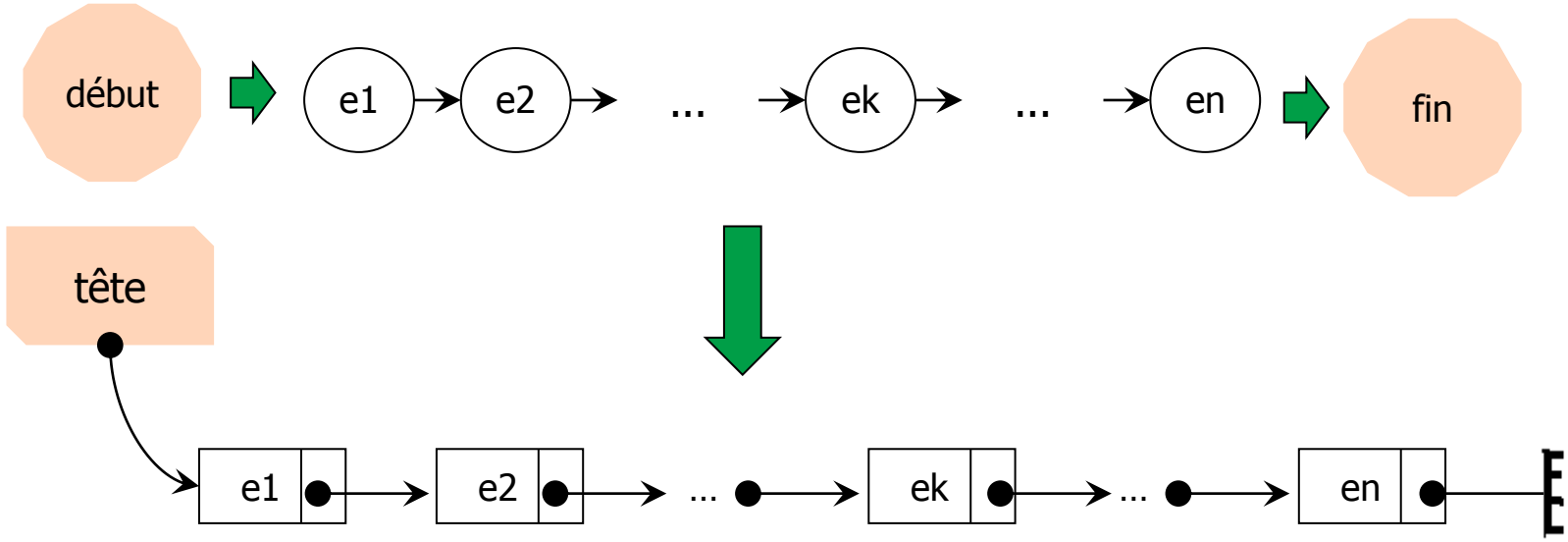
Gaspillage de la mémoire.

Listes —*Représentation chainée*

- Représentation *chaînée* à l'aide de *pointeurs* qui établissent le chainage entre les éléments:
  - **simple** —*dans un sens*
  - **double** —*dans les deux sens*
  - **circulaire** —*lien entre de dernier élément et le premier*

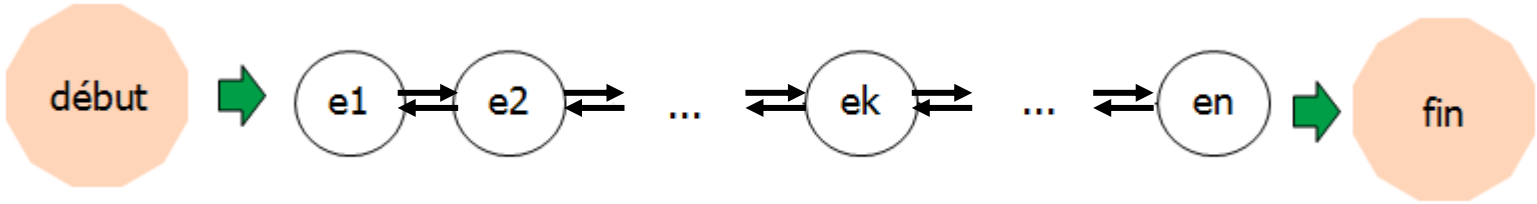
Listes — *Représentation chainée*

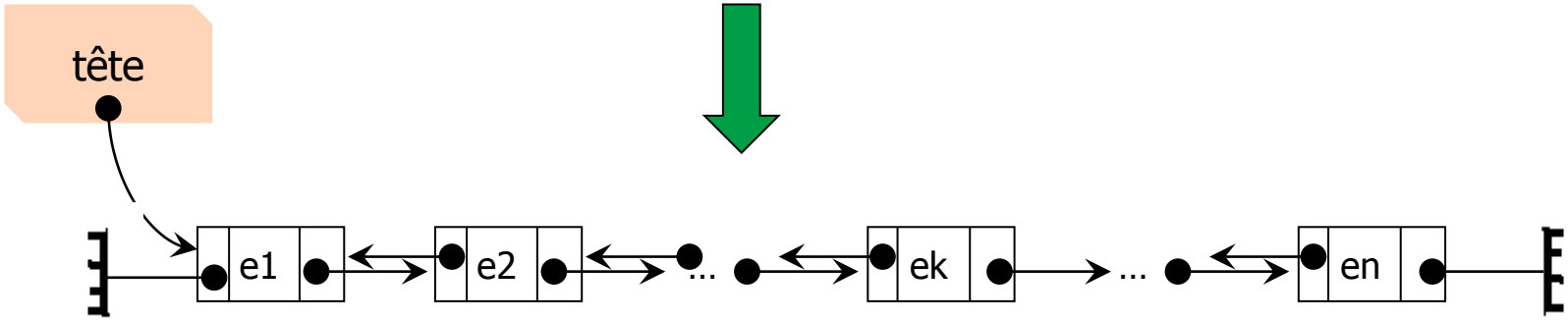
- Représentation avec *chainage simple*



Listes —*Représentation chainée*

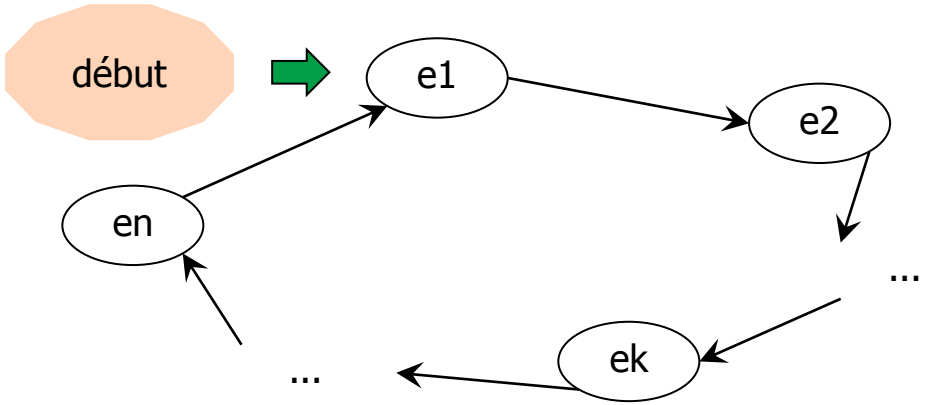
- Représentation avec *chainage double*

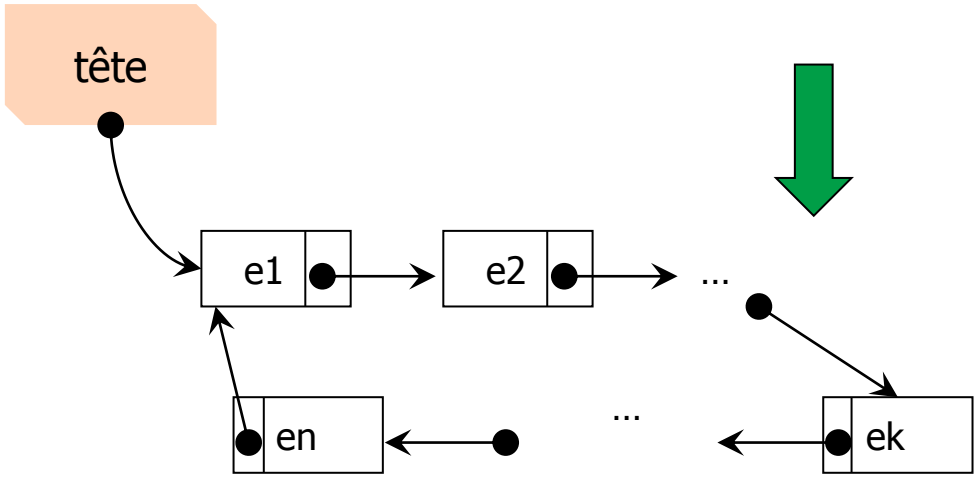




Listes —*Représentation chainée*

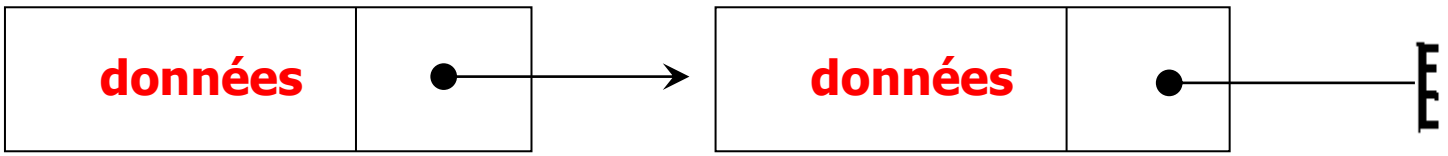
- Représentation avec *chainage circulaire*





Listes —*chainage simple*

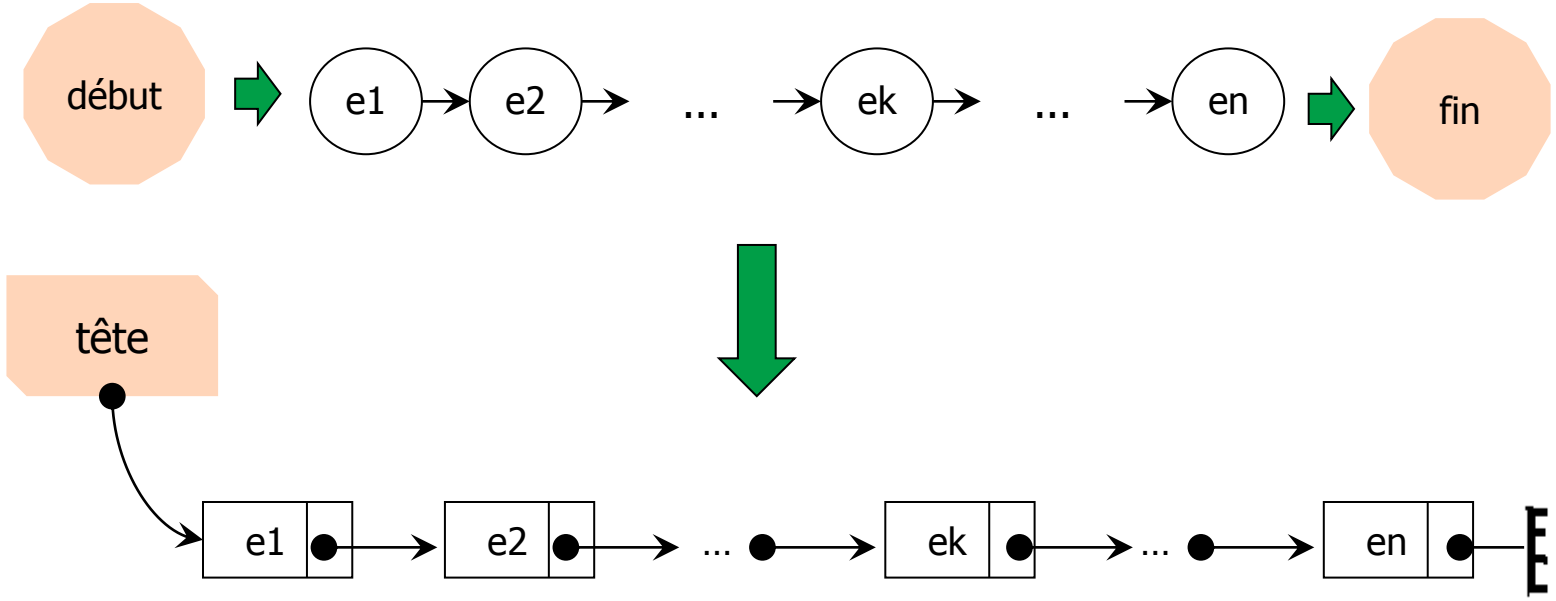
- Représentation par une structure récursive:
  - se base sur la notion de *maillon* qui est constitué :
    - d'un champ de *données* (float, int, structures, etc.)
    - d'un *pointeur* qui contient l'adresse du maillon suivant.
  - le pointeur vaut NULL si le maillon n'a pas de suivant



Maillon qui n'a pas de suivant

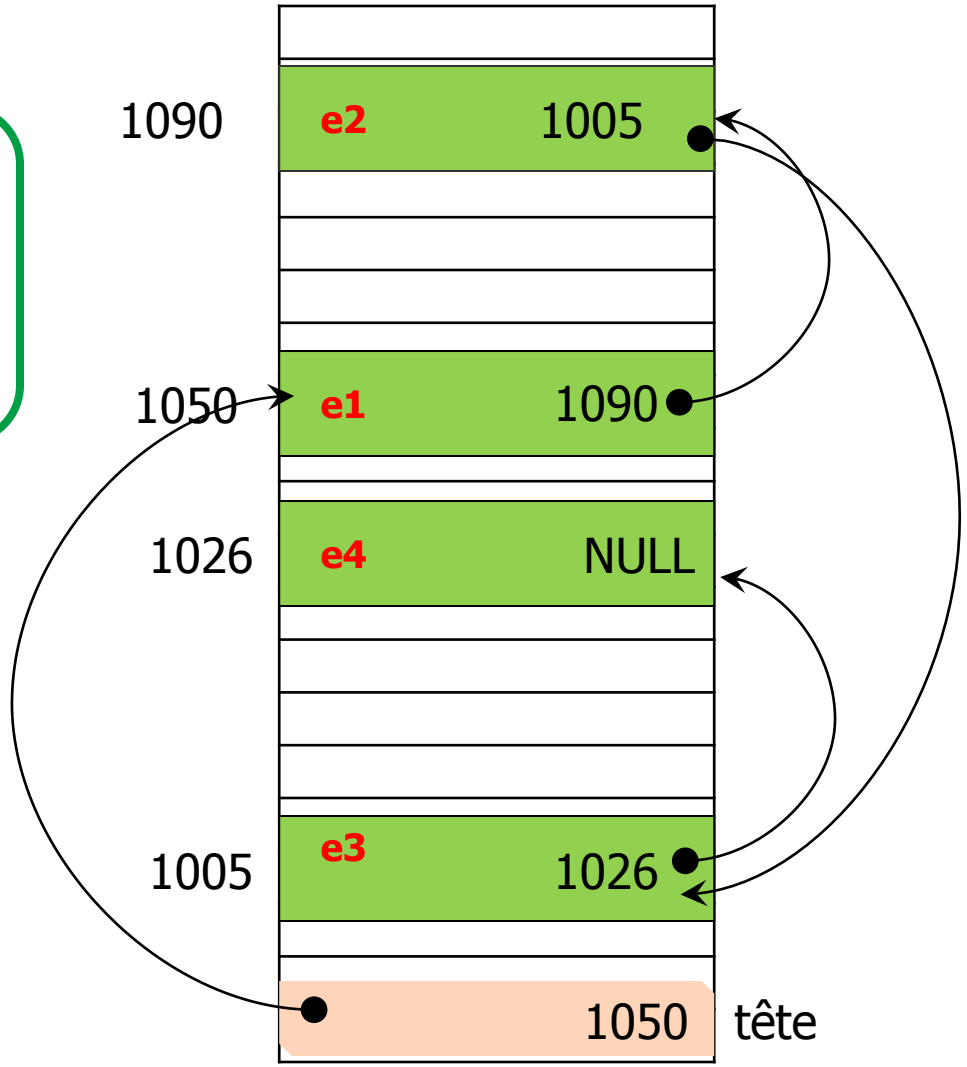
Listes —*chainage simple*

- Une liste est définie par le pointeur **tête** qui contient l'adresse du premier élément.
- Une liste vide ne contient pas de maillons (**tête =NULL)**



Listes —*chainage simple*

Les maillons d'une liste ne sont pas toujours placées dans l'ordre en mémoire et encore moins de façon contigüe.



Listes —*chainage simple*

- La relation de *succession* est mise en œuvre par des pointeurs:
  - chaque élément possède l'adresse de son successeur
  - une liste chaînée est identifiée par le pointeur sur le premier élément.
  - Accès séquentiel: pour arriver un élément il faut passer par ses prédécesseurs

l'Accès aux éléments prend plus de temps que les tableaux.

- La taille de la liste est inconnue au départ, la liste peut avoir autant d'éléments que votre mémoire le permet.

Optimisation de la mémoire.

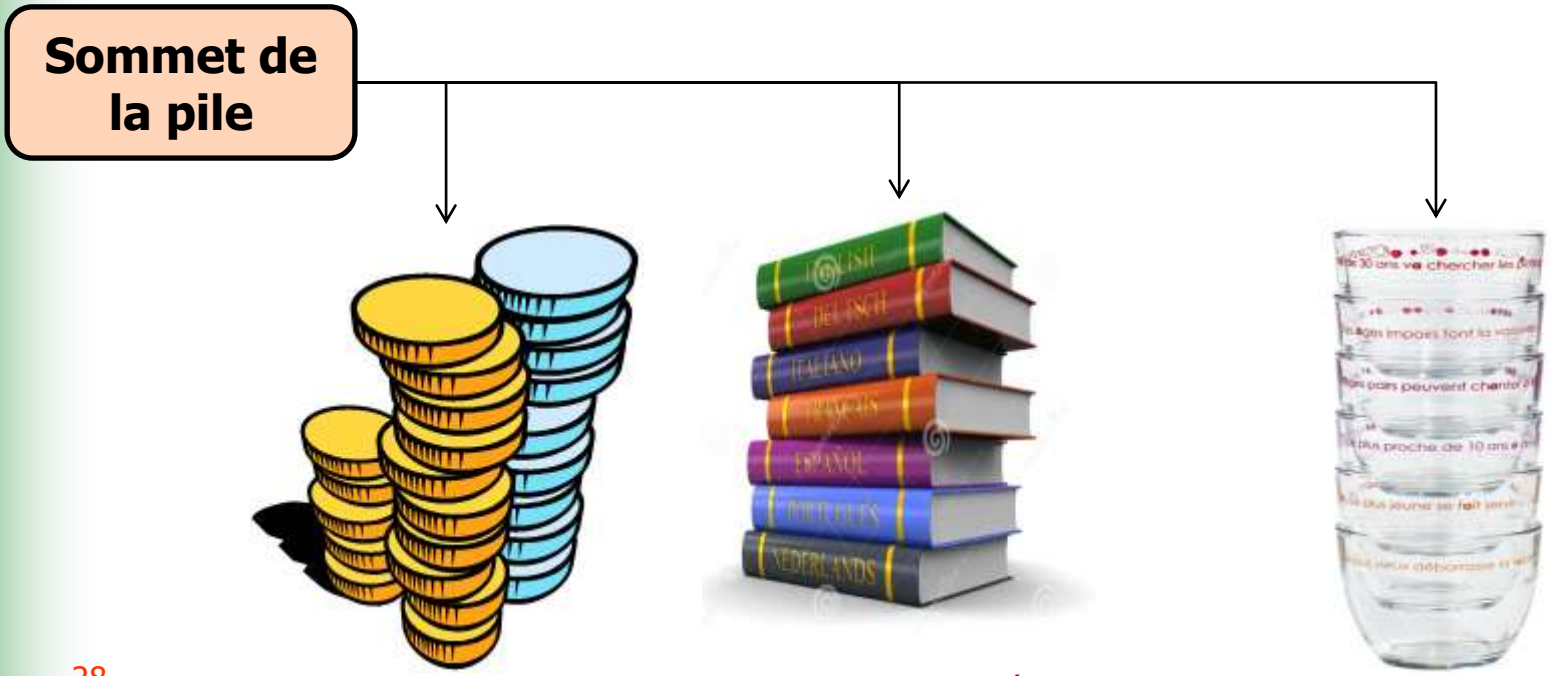
Séance 4.2 - Les piles

- Définition
- Exemple
- Opérations de base
- Représentation contigüe
- Représentation chainée(non contigüe)

Les Piles — *Définition*

- Notion intuitive — *Piles d'objets*

➡ l'ajout d'un objet se fait au sommet.
le retrait d'un objet se fait depuis le sommet.

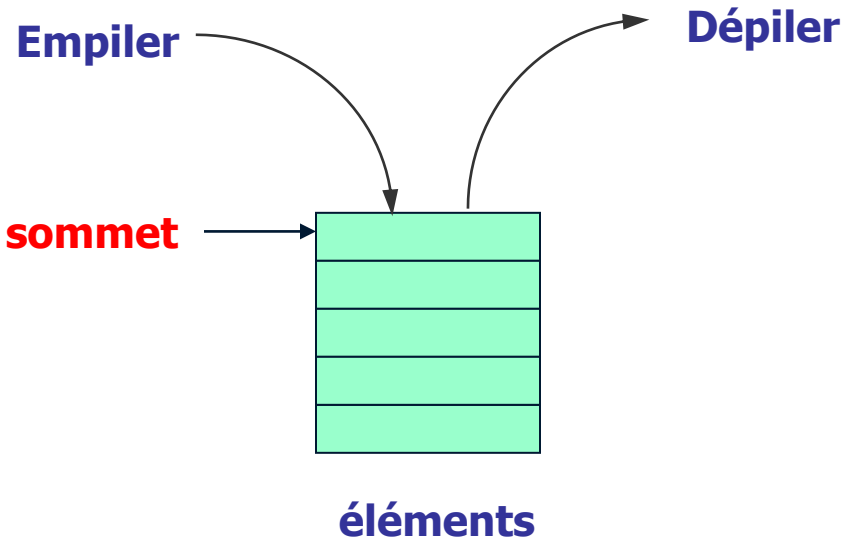


Les Piles — *Définition*

- Une pile est une **structure** destinée au stockage des données en cours de traitement par un programme.

- Une pile est une **liste particulière** fondée sur le principe premier entrée dernier sorti (**L**ast **I**n **F**irst **O**ut) .

➡ L'ajout et le retrait se font à la même place (le **sommet**).



Les Piles — *Utilisation*

- Fonction *undo (défaire)* dans les logiciels de bureautique.

- Gestion par le compilateur des appels de fonctions:

  ⇨ les paramètres, l'adresse de retour et les variables locales sont stockées dans la pile de l'application.

- Mémorisation des appels de procédures imbriquées au cours de l'exécution des fonctions récursives.

Les Piles — *Opérations de base*

- **Empiler** — *Ajoute un élément au sommet de la pile.*
- **Dépiler** — *Supprime l'élément au sommet de la pile si la pile est non vide.*
- **Sommet** — *récupère l'élément qui au sommet de la pile si la pile est non vide.*
- **InitPile** — *Crée une pile vide.*
- **PileVide** — *Détermine si une pile est vide ou non.*

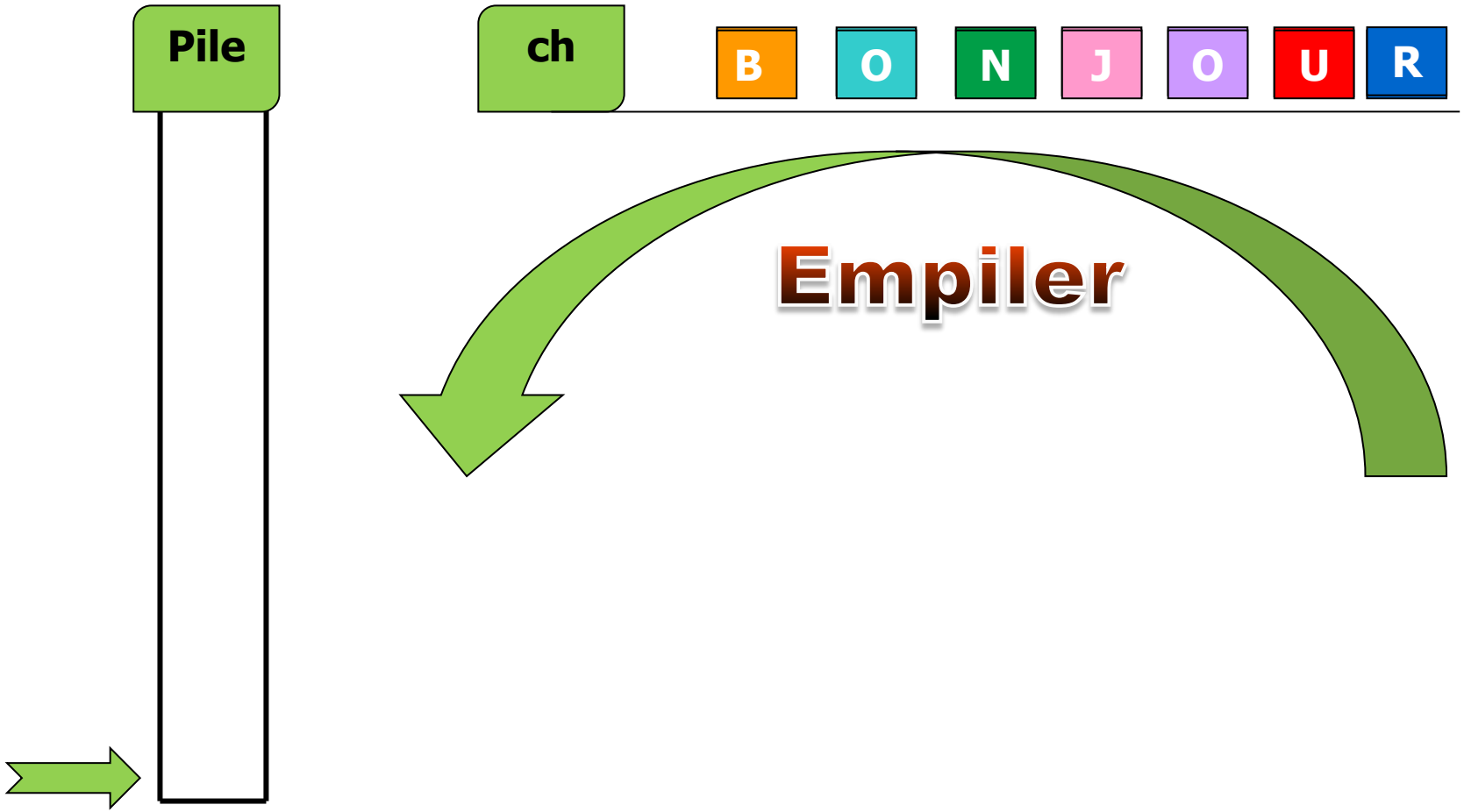
Les Piles — *Opérations de base*

- Exemple —*inverser une chaine de caractères*

```
void main () {
    char ch[15]; int i; int n;
 /* On suppose que PILE est déjà définie*/
    PILE P;
    P= InitPile ();
 /*saisir ch*/
    gets(ch);  n=strlen(ch);

 /*Recopier ch dans la pile, élément par élément*/

    for(i=0; i<n; i++)
            empiler(&P,ch[i]);

/*retirer, élément par élément, les éléments de la pile et
les stocker dans ch */

    for(i=0; i<n; i++)
    {
       Sommet(P,&ch[i]) ;
       depiler(&P);
    }
}
```

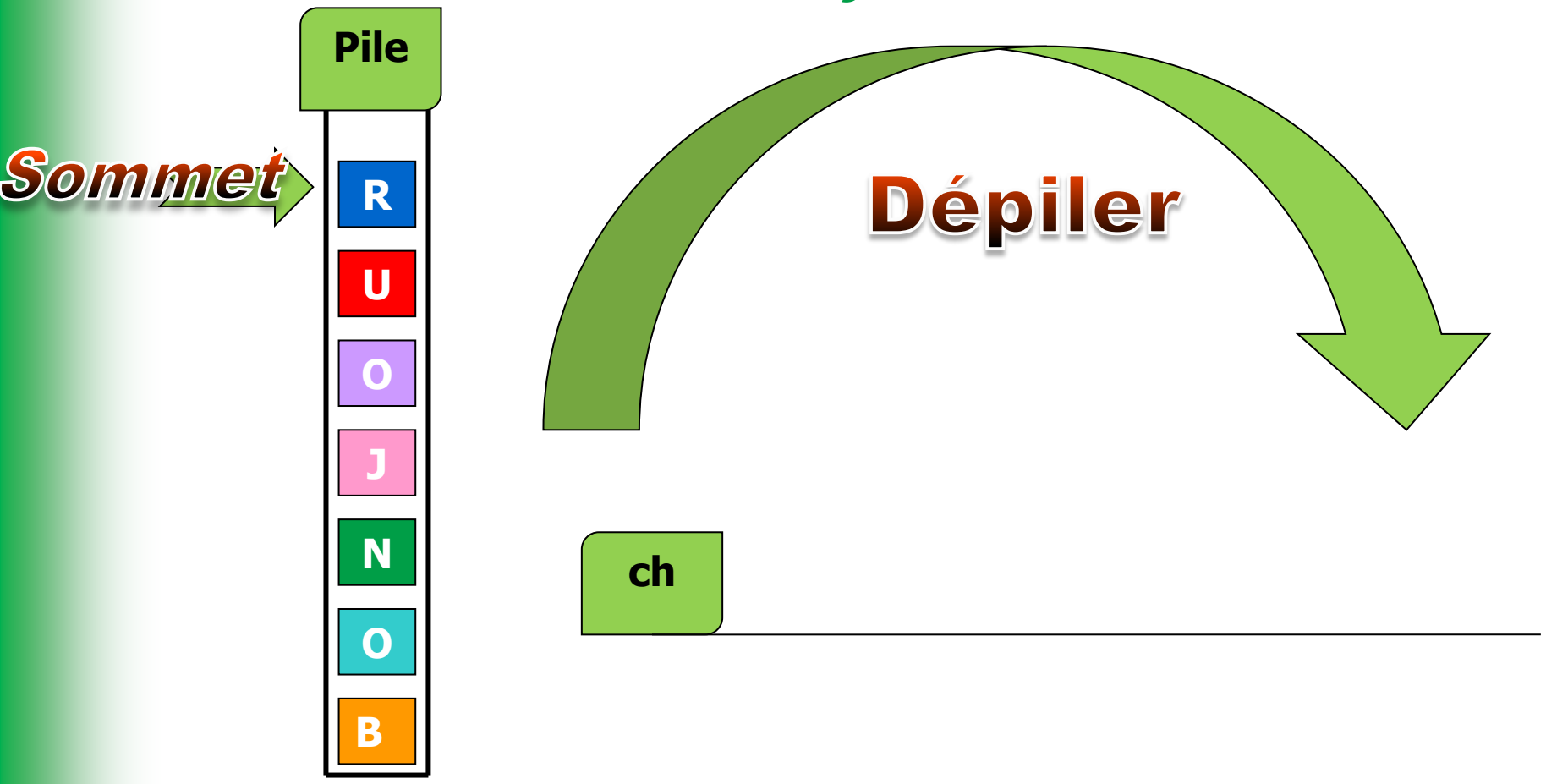
Les Piles — *Opérations de base*

- Inverser la chaine *Bonjour*



Les Piles — *Opérations de base*

- Inverser la chaine *Bonjour*



Les Piles — *Représentation*

- Représentation contiguë
  - Implémentation par un tableau.
- Représentation chainée
  - Ensemble d'éléments liés séquentiellement (par des pointeurs)

Les Piles — *Représentation contigüe*

- Implémentation par un tableau.

- La pile est représentée par une structure définie par:
  - Sommet — un entier qui représente l'indice de l'élément qui se trouve au sommet de la pile.

  - Tab — un tableau d'éléments de la pile.

- Définition d'une PILE d'entiers.

```
#define CAP 10
typedef struct {
    int Sommet;
    int Tab[CAP];
 } PILE;
```

Les Piles — *Représentation contigüe*

- Le tableau peut être :
  - statique — *la taille maximale est fixée.*
  - dynamique — *réservation dynamique de la mémoire et la taille maximale peut être modifiée.*

- Il faut faire un contrôle de taille lors de l'empilement pour gérer le dépassement de capacité.

  ➡ Utilisation de la fonction *realloc* pour redimensionner le tableau (dans la cas d'un tableau dynamique)

Les Piles — *Représentation contigüe*

- Initialisation de la Pile —*le sommet est mis à l'indice -1*

```
PILE InitPile ()
  {
       PILE P;
       P.Sommet =-1;
       return P;
  }
```

- Pile vide ? —*le sommet est à l'indice -1*

```
int PileVide(PILE P)
  {
       if(P.Sommet == -1)
              return 1;
       return 0;
  }
```

Les Piles — *Représentation contigüe*

- Récupérer la valeur du sommet
  - Il faut vérifier si la pile n'est pas vide avant de prendre la valeur du sommet.
  - La fonction retourne 1 si la valeur est prise et 0 sinon

```
int Sommet(PILE P, int *x)
  {
       if (!PileVide(P)) {
             *x = P.Tab[P.Sommet];
             return 1;
       }
       return 0;
  }
```

Les Piles — *Représentation contigüe*

- Empiler un élément
  - Il faut vérifier si la pile n'est pas pleine avant d'insérer l'élément.
  - La fonction retourne 1 si la valeur est insérée et 0 sinon

```
int Empiler(PILE *P, int x)
  {
       if (P->Sommet != CAP-1) {
              (P->Sommet)++;
              P->Tab[P->Sommet]=x;
              return 1;
       }
       return 0;
  }
```

Les Piles — *Représentation contigüe*

- Dépiler un élément
  - Il faut vérifier si la pile n'est pas vide avant de retirer l'élément du sommet.
  - La fonction retourne 1 si la valeur est retirée et 0 sinon.

```
int Depiler(PILE *P)
  {
       if (!PileVide(*P)) {
              P->Sommet--;
             return 1;
       }
       return 0;
  }
```

Les Piles — *Représentation contigüe*

- Exemple

```
#define CAP 10
typedef struct {

        int Sommet;
        int Tab[CAP];
} PILE;

PILE InitPile ()
  {...}
int PileVide(PILE P)
  {...}
int Sommet(PILE P,int *x)
  {...}
int Empiler(PILE *P,int x)
  {...}
int Depiler(PILE *P)
  {...}
```

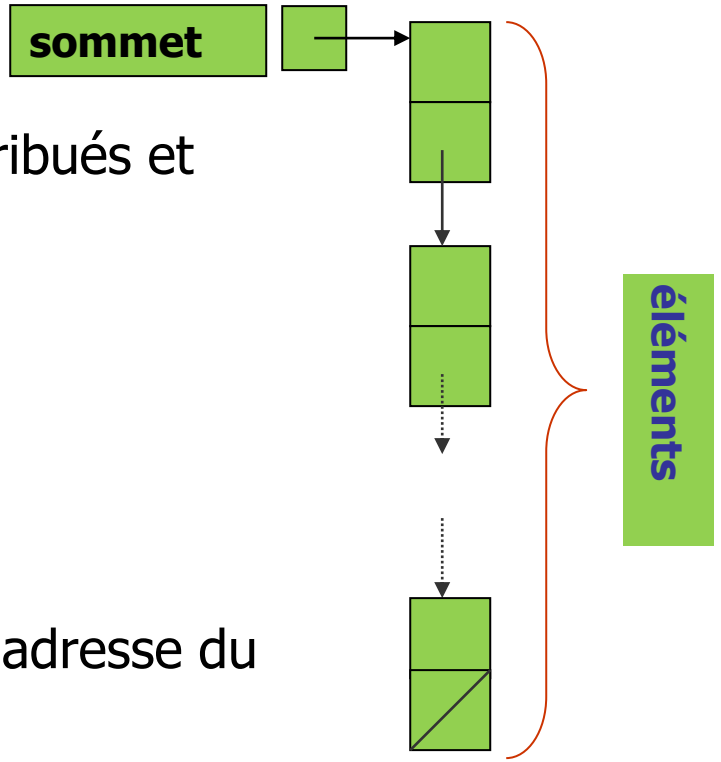
```
main(){
        PILE P;
        int a,b,c,x
        P= InitPile();
        i=0;
        while(i<10){
             scanf("%d",&x)
             a= Empiler(&P,x);
             i++;
             if(a==0) return 0;
        }
       while((!PileVide(P)) {
             b= Sommet(P,&x);
             c= depiler(&P);
             if(b!=0)printf("%d\n",x);
        }
        . . . . .
}
```

Les Piles — *Représentation contigüe*

- La représentation contigüe par un tableau présente les inconvénients suivants :
  - la taille maximale doit être fixée.
  - les éléments du tableau sont stockés dans des cases mémoires adjacentes.
  - difficulté de redimensionner le tableau.
    - La fonction *realloc* est très coûteuse en temps .

Représentation chaînée

Les Piles — *Représentation Chaînée*

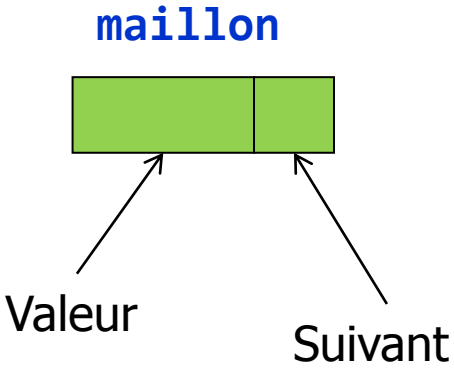


- Une pile est une suite de maillons distribués et organisés séquentiellement.
- Un maillon possède :
  - les données de l'élément.
  - un lien (un pointeur) vers son successeur.
- La pile est identifiée par son sommet (adresse du premier maillon) .
- Les opérations sur la pile sont basées sur la manipulation des liens (pointeurs) .

Les Piles — *Représentation Chaînée*

- Définition d'une pile d'entiers

```
/* type maillon */
   typedef struct Maillon{

         int valeur;
         struct Maillon * suivant;
    } maillon;

/* type PILE */
   typedef maillon* PILE;
```



Les Piles — *Représentation Chaînée*

- Initialisation de la Pile —*le sommet est mis à NULL*

```
PILE InitPile ()
  {
       return NULL;
  }
```

- Pile vide ? —*le sommet est à NULL*

```
int PileVide(PILE P)
  {
       if(P == NULL)
              return 1;
       return 0;
  }
```

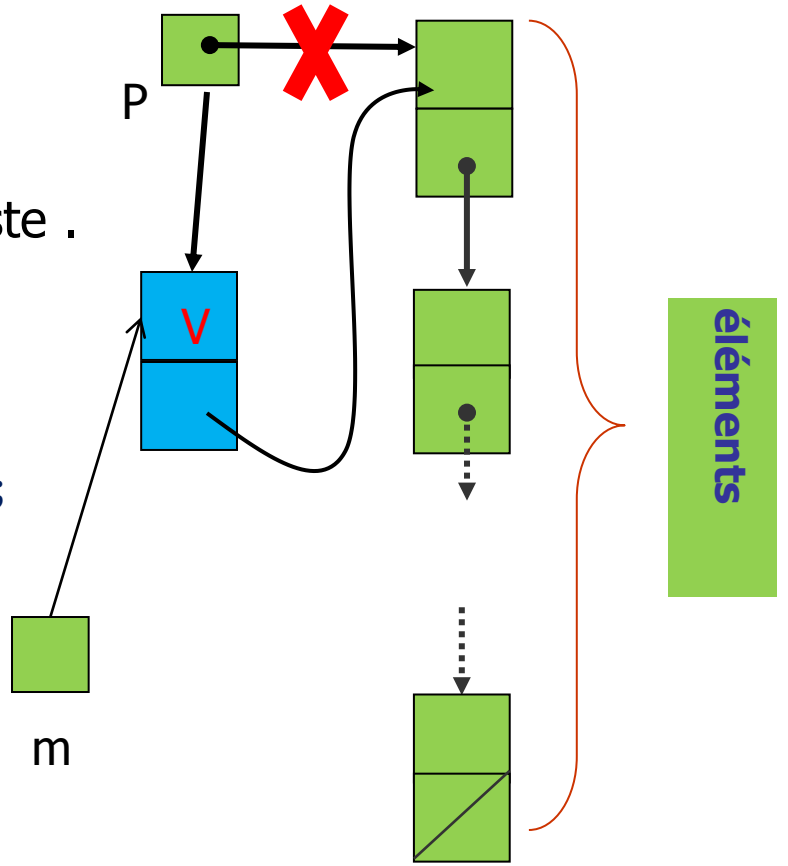
Les Piles — *Représentation chaînée*

- Récupérer la valeur du sommet
  - Il faut vérifier si la pile n'est pas vide avant de prendre la valeur du sommet.
  - La fonction retourne 1 si la valeur peut être prise et 0 sinon

```
int Sommet(PILE P, int *x)
  {
       if (!PileVide(P)) {
              *x = P->valeur;
              return 1;
       }
       return 0;
  }
```

Les Piles — *Représentation chaînée*

- Empiler un élément V dans la PILE P
  - Créer un maillon avec la valeur de l'élément,
  - Ajouter l'élément à la tête de la liste .

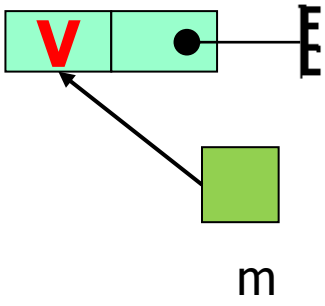



```
maillon * m;
m=(maillon *)malloc(sizeof(maillon);
m->valeur = V;
m->suivant=P;
P=m;
```

Les Piles — *Représentation chaînée*

- Créer maillon d'une pile d'entiers
  - Réserver l'espace mémoire pour l'élément
  - Remplir les champs du maillon : valeur par V et le pointeur par NULL
  - La fonction retourne l'adresse de l'espace réservé et NULL si l'espace n'est pas réservé.

```
maillon* creerMaillon(int V)
{
  maillon *m;
  m=(maillon *)malloc(sizeof(maillon);
     if(m!=NULL){
               m->valeur = V;
               m->suivant = NULL;
     }
        return m;
  }
```



Les Piles — *Représentation chaînée*

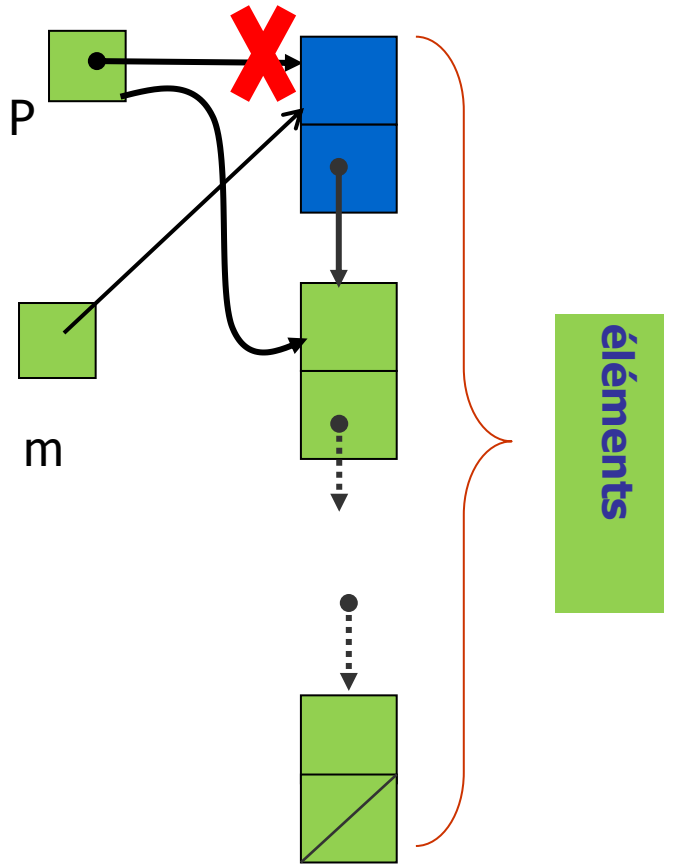
- Empiler un élément
  - Créer un maillon avec la valeur de l'élément,
  - Ajouter l'élément à la tête de la liste .
  - La fonction retourne 1 si la valeur est insérée et 0 sinon

```
int Empiler(PILE *P, int V)
  {
        maillon* m;
        m=creerMaillon(V);
        if(m!=NULL){
                m->suivant = *P;
                *P=m;
                return 1;
        }
        return 0;
  }
```

Les Piles — *Représentation chaînée*

- Dépiler un élément de la PILE P

```
maillon* m;
if (!PileVide(P)){
        m=P;
        P=P->suivant;
        free(m);
}
```



Remarque:

Si la Pile est vide on fait rien

Les Piles — *Représentation chaînée*

- Dépiler un élément
  - créer un pointeur m, tel que m et p pointent sur la même zone,
  - Pointer la tête sur la zone pointée par le suivant de tête ,
  - Libérer la zone pointée par m ,
  - La fonction retourne 1 si l'élément de la tête est supprimé et 0 sinon

```
int depiler(PILE *P){
        maillon* m;
        if (!PileVide(*P)){
                m= *P;
                 *P= (*P)->suivant;
                free(m);
                return 1;
        }
         return 0;
   }
```

Les Piles — *Représentation chaînée*

- Exemple

```
typedef struct {
        int valeur;
        struct maillon *suivant;
} maillon;
typedef maillon* PILE;

PILE InitPile ()
  {...}
int PileVide(PILE P)
  {...}
int Sommet(PILE P,int *x)
  {...}
int Empiler(PILE *P,int x)
  {...}
int Depiler(PILE *P)
  {...}
```

```
main(){
        PILE P;
        int a,b,c,x
        P= InitPile ();
        i=0;
        while(i<10){
             scanf("%d",&x)
             a= Empiler(&P,x);
             i++;
             if(a==0) return 0;
        }
       while((!PileVide(P)) {
             b= Sommet(P,&x);
             c= depiler(&P);
             if(b!=0)printf("%d\n",x);
        }
        . . . . .
}
```