Piles (Stack)

Pr A. EL ALLAOUI

Notion de Type Abstrait de Données (TAD)

- **La conception d'un algorithme est indépendante de toute implantation**
- **La représentation des données n'est pas fixée ; *celles-ci sont considérées de manière abstraite***
- **On s'intéresse à l'ensemble des opérations sur les données, et aux propriétés des opérations, *sans dire comment ces opérations sont réalisées***
- **On parle de *Type Abstrait de Données* (*TAD*)**

Un TAD (*Data Abstract Type*) est un *ensemble de valeurs muni d'opérations sur ces valeurs, sans faire référence à une implémentation particulière.*

Exemples :

- ***Dans un algorithme qui manipule des `entiers`, on s'intéresse, non pas à la représentation des entiers, mais aux opérations définies sur les entiers : `+, -, *, /`***
- ***Type `booléen`, ensemble de deux valeurs (`faux`, `vrai`) muni des opérations : non, `et`, `ou`***

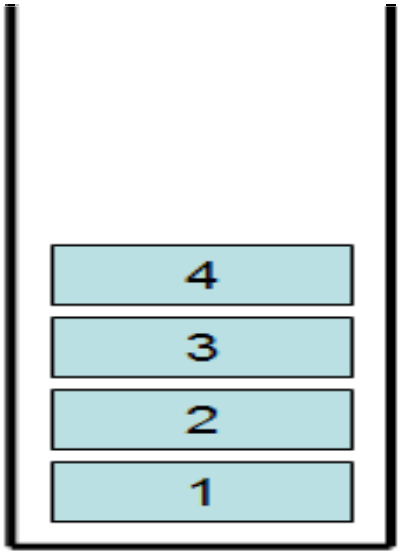
Objectif:

- **Notion de Pile**
- **Implémentation sous forme de tableau**
- **Implémentation sous forme de liste chaînée**
- **Comparaison entre tableaux et listes chaînées**

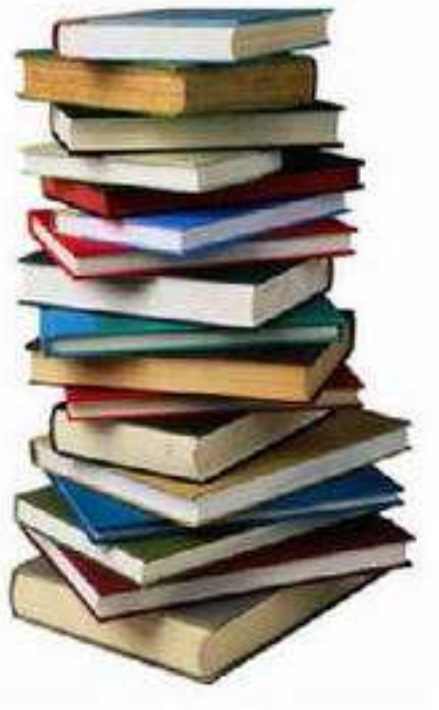
Notion de Pile (Stack)

Le nom de pile vient d'une analogie avec une pile d'assiettes (par exemple) où l'on poserait toujours les assiettes sur le dessus de la pile, et ou l'on prendrait toujours les assiettes sur le dessus de la pile. Ainsi, la dernière assiette posée sera utilisée avant toutes les autres.

- La pile est un espace où on peut accumuler des éléments.
- On ne peut retirer que le plus jeune des éléments ajouté.
- Les tours d'Hanoï.
- Usage très répandu dans la programmation système : gestion des processus
- Les piles sont utilisées pour gérer les appels **récursifs !!!!**



Notion de Pile (Stack)



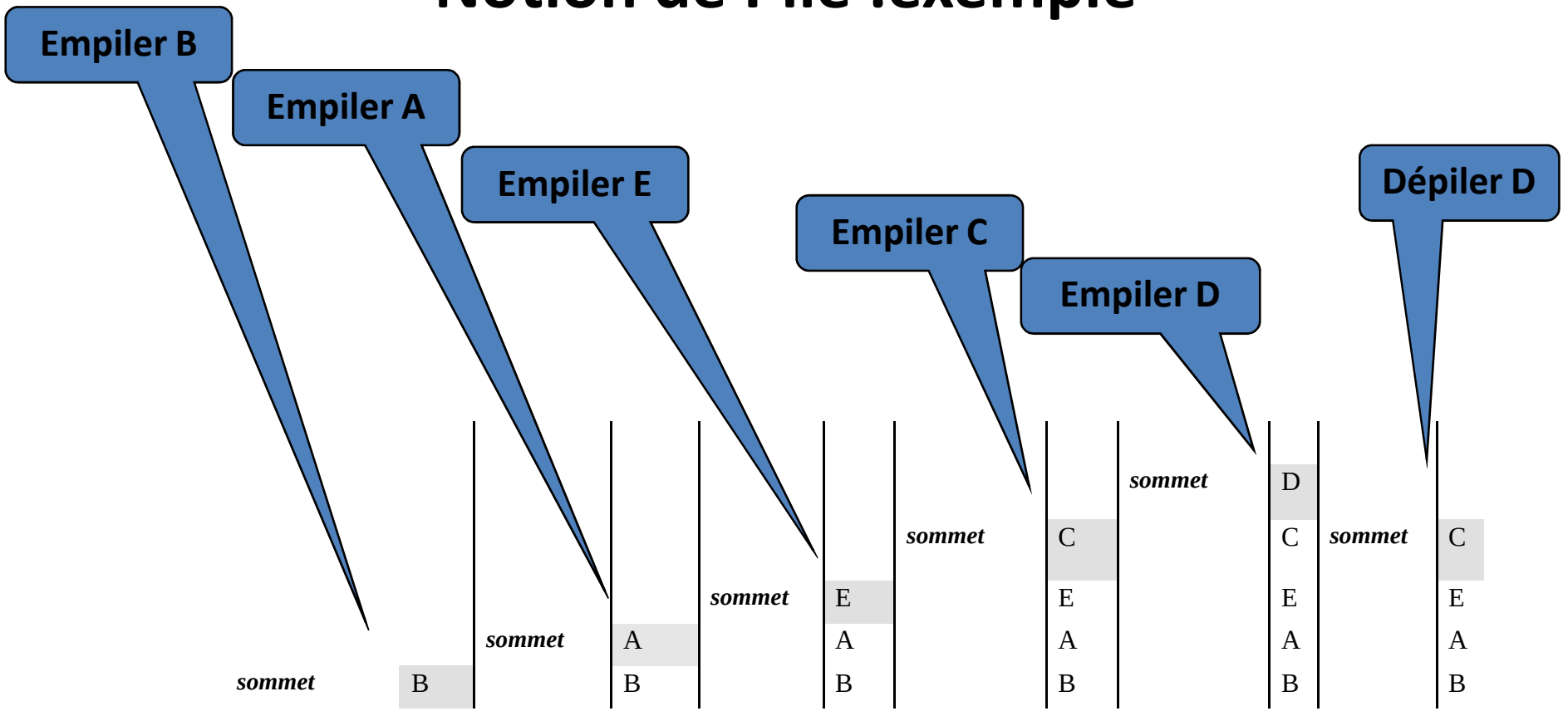
- **Les piles sont très utilisées en informatique**

- **Notion intuitive :**
  - pile d'assiettes, pile de dossiers à traiter, …

- **Une pile est une structure linéaire permettant de sto restaurer des données selon un ordre LIFO (*Last In, F* « dernier entré, premier sorti »)**
- **Dans une pile :**
  - Les insertions (***empilements***) et les suppressions (***dépilements***) sont restreintes à une extrémité appelée ***sommet*** de la pile.

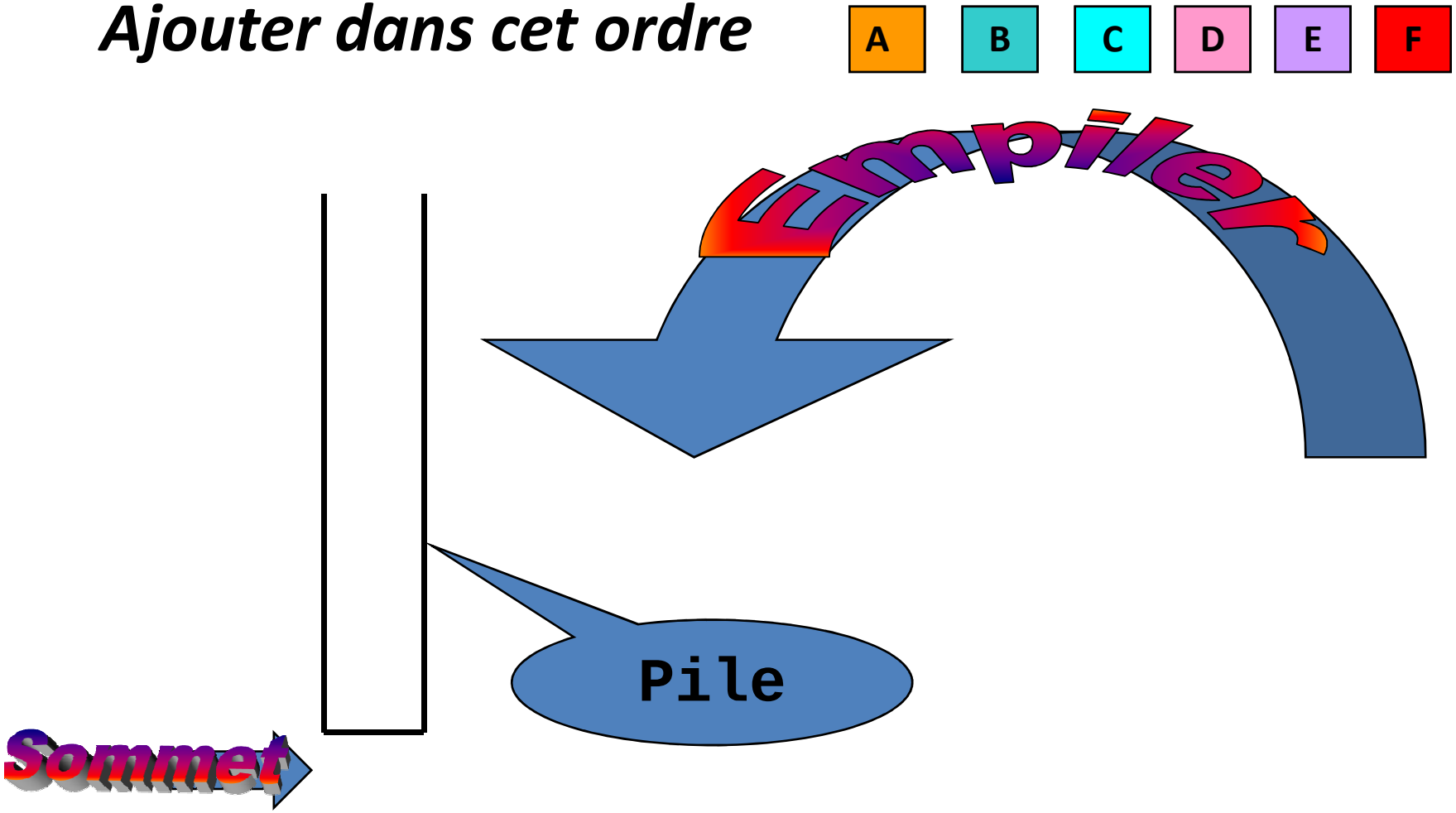
    - Le **sommet** de la pile est le seul élément manipulable
    - Pour manipuler un élément se trouvant au milieu de la pile il faudra dépiler **tout ces prédécesseurs**
    - Si on ajoute un nouvel élément il serra **empiler au dessus du sommet**

Notion de Pile :exemple

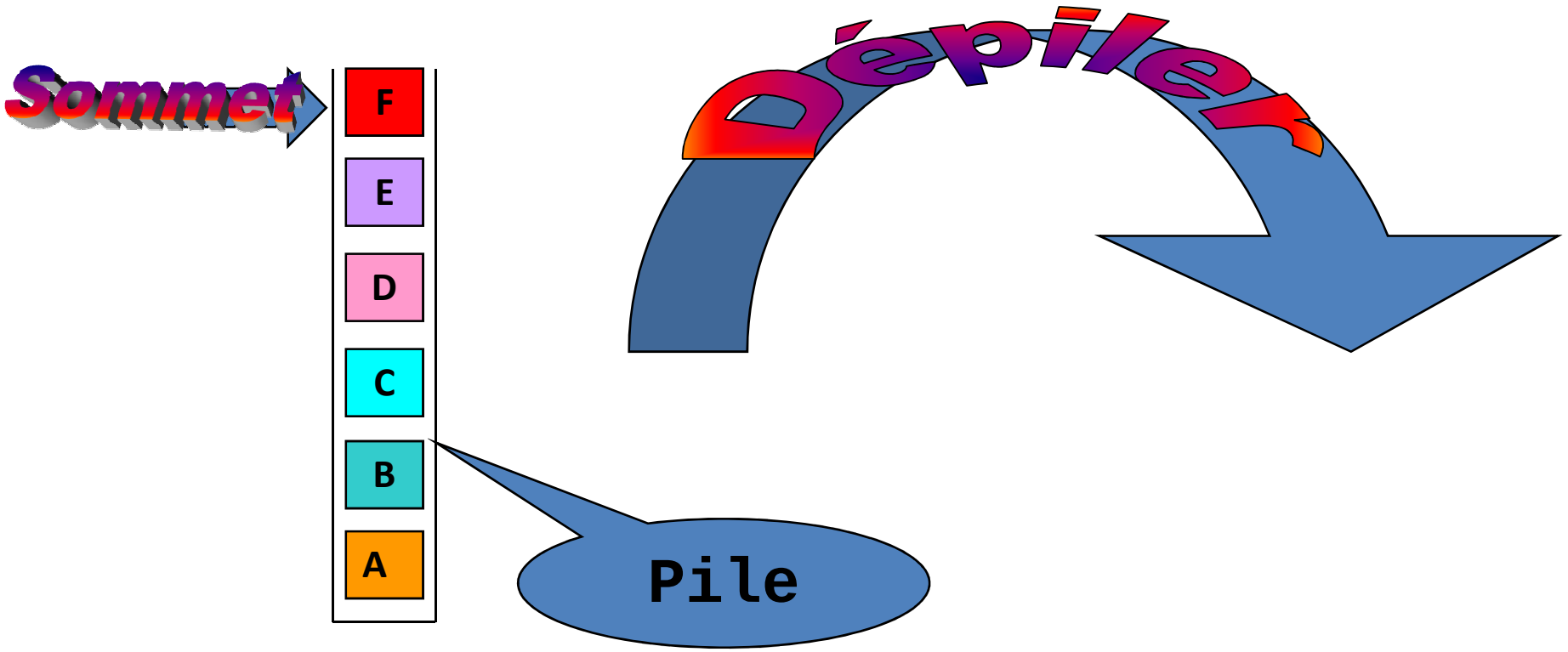


Notion de Pile :exemple (1)

Ajouter dans cet ordre



Notion de Pile :exemple (2)



Opérations sur une pile

- **`pile_vide :`**
  - opération d'initialisation ; *la pile créée est vide*
- **`est_vide :`**
  - teste si pile vide ou non
- **`sommet :`**
  - permet de consulter l'élément situé au sommet ; *n'a pas de sens si pile vide*
- **`empiler :`**
  - ajoute un élément dans la pile
- **`dépiler :`**
  - enlève l'élément situé au sommet de la pile ; *n'a pas de sens si pile vide*

Opérations sur une pile

Primitives :

Une pile peut être implémentée par un **tableau** ou par **une liste chaînée**. Dans les deux cas, il est commode de réaliser sur les piles des opérations de base, appelées ***primitives*** *de gestion des piles.* Les primitives de gestion des piles sont les suivantes :

- **Initialiser** : cette fonction crée une pile vide.
- **EstVide** : renvoie 1 si la pile est vide, 0 sinon.
- **EstPleine** : renvoie 1 si la pile est pleine, 0 sinon.
- **AccederSommet** : permet l'accès à l'information contenue dans le sommet de la pile.
- **Empiler** : permet d'ajouter un élément au sommet de la pile. La fonction renvoie un code d'erreur si besoin en cas de manque de mémoire.

Notion de Pile

Primitives :

- **Depiler** : supprime le sommet de la pile. L'élément supprimé est retourné par la fonction Depiler pour pouvoir être utilisé.
- **Vider** : cette fonction vide la pile.
- **Detruire** : cette fonction permet de détruire la pile.

Le principe de gestion des piles est que, lorsqu'on utilise une pile, on ne se préoccupe pas de la manière dont elle a été implémentée, mais on utilise uniquement les primitives qui sont toujours les mêmes.

Nous allons toutefois étudier l'implémentation des primitives de gestion des piles.

Représentation d'une Pile

- **Représentation contiguë (*par tableau*) :**
  - ***Les éléments de la pile sont rangés dans un tableau***
  - ***Un entier représente la position du sommet de la pile***

- **Représentation chaînée (par pointeurs) :**
  - ***Les éléments de la pile sont chaînés entre eux***
  - ***Un pointeur sur le premier élément désigne la pile et représente le sommet de cette pile***
  - ***Une pile vide est représentée par le pointeur `NULL`***

Implémentation sous forme de tableau

Pour implémenter une pile sous forme de tableau, on crée la structure de données suivante. L'implémentation est donnée pour des données de type float.

Le principe est le même pour n'importe quel type de données.

```
typedef float TypeDonnee;
typedef struct
    {
        int nb_elem; /* nombre d'elements dans la pile */
        int nb_elem_max; /* capacite de la pile */
        TypeDonnee *tab; /* tableau contenant les elements */
    }Pile;
```

Créer une pile vide

La fonction permettant de créer une pile vide est la suivante :

```
Pile Initialiser(int nb_max) {
      Pile pilevide;
      pilevide.nb_elem = 0;          /* la pile est vide */
      pilevide.nb_elem_max = nb_max;    /* capacite nb_max */
                                 /* allocation des elements : */
     pilevide.tab =(TypeDonnee*)malloc(nb_max*sizeof(TypeDonnee));
return  pilevide;
}
```

```
typedef float TypeDonnee;
typedef struct
        {
                int nb_elem; /* nombre d'elements dans la pile */
                int nb_elem_max; /* capacite de la pile */
                TypeDonnee *tab; /* tableau contenant les elements */
        }Pile;
```

Pile vide, pile pleine

La fonction permettant de savoir si la pile est vide est la suivante. La fonction renvoie 1 si le nombre d'éléments est égal à 0. La fonction renvoie 0 dans le cas contraire.

```
int EstVide(Pile P)
{
                    /* retourne 1 si le nombre d'elements vaut 0 */
   return (P.nb_elem == 0) ? 1 : 0;
}
```

La fonction permettant de savoir si la pile est pleine est la suivante :

```
int EstPleine(Pile P)
{
              /* retourne 1 si le nombre d'elements est superieur */
                 /* au nombre d'elements maximum et 0 sinon */
          return (P.nb_elem >= P.nb_elem_max) ? 1 : 0;
}
```

Accéder au sommet de la pile

- Le sommet de la pile est le dernier élément entré, qui est le dernier élément du tableau.
- La fonction effectue un passage par adresse pour ressortir le résultat.
- La fonction permettant d'accéder au sommet de la pile, qui renvoie le code d'erreur 1 en cas de liste vide et 0 sinon, est donc la suivante :

```
int AccederSommet(Pile P, TypeDonnee *pelem)
{
    if (EstVide(P))
                    return 1;          /* on retourne un code d'erreur */
    *pelem = P.tab[P.nb_elem-1];       /* on renvoie l'element */
    return 0;
}
```

Ajouter un élément au sommet

Pour modifier le nombre d'éléments de la pile, il faut passer la pile par adresse. La fonction Empiler, qui renvoie 1 en cas d'erreur et 0 dans le cas contraire, est la suivante :

```
int Empiler(Pile* pP, TypeDonnee elem)
{
    if (EstPleine(*pP))
        return 1;          /* on ne peut pas rajouter d'élément */
        pP->tab[pP->nb_elem] = elem;     /* ajout d'un élément */
        pP->nb_elem++;     /* incrémentation du nombre d'éléments */
    return 0;
}
```

Supprimer un élément

La fonction Depiler supprime le sommet de la pile en cas de pile non vide. La fonction renvoie 1 en cas d'erreur (pile vide), et 0 en cas de succès.

```
char Depiler(Pile *pP, TypeDonnee *pelem)
{
if (EstVide(*pP))
    return 1;                /* on ne peut pas supprimer d'element */
    *pelem = pP->tab[pP->nb_elem-1];  /* on renvoie le sommet */
    pP->nb_elem--;           /* decrementation du nombre d'elements */
    return 0;
}
```

Vider et détruire

```
void Vider(Pile *pP)
{
    pP->nb_elem = 0;            /* reinitialisation du nombre d'elements */
}
void Detruire(Pile *pP)
{
    if (pP->nb_elem_max != 0)
        free(pP->tab);              /* liberation de memoire */
    pP->nb_elem = 0;
    pP->nb_elem_max = 0;            /* pile de taille 0 */
}
```

Implémentation sous forme de liste chaînée

Pour implémenter une pile sous forme de liste chaînée, on crée la structure de données suivante. L'implémentation est donnée pour des données de type float. Le principe est le même pour n'importe quel type de données.

```
typedef float TypeDonnee;

typedef struct Cell
{
    TypeDonnee donnee;
    struct Cell *suivant;              /* pointeur sur la cellule suivante */
}TypeCellule;
typedef TypeCellule* Pile; /* la pile est un pointeur sur la tête de liste */
```

Implémentation sous forme de liste

Créer une pile vide

La fonction permettant de créer une pile vide est la suivante :

```
Pile Initialiser(){
    return NULL;        /* on retourne une liste vide */
}
```

Pile vide, pile pleine

La fonction permettant de savoir si la pile est vide est la suivante. La fonction renvoie 1 si la pile est vide, et renvoie 0 dans le cas contraire.

```
int EstVide(Pile P) {
    return (P == NULL) ? 1 : 0;          /* renvoie 1 si la liste est vide */
}
```

La fonction permettant de savoir si la pile est pleine est la suivante :

```
int EstPleine(Pile P) {
    return 0;                    /* une liste chainee n'est jamais pleine */
}
```

Implémentation sous forme de liste

Accéder au sommet de la pile

Le sommet de la pile est le dernier élément entré qui est la tête de liste. La fonction renvoie 1 en cas de liste vide, 0 sinon.

```
int AccederSommet(Pile P, TypeDonnee *pelem)
{
if (EstVide(P))
    return 1;                    /* on retourne un code d'erreur */
    *pelem = P->donnee;                  /* on renvoie l'element */
    return 0;
}
```

Implémentation sous forme de liste

Ajouter un élément au sommet

La fonction d'ajout d'un élément est une fonction d'insertion en tête de liste

```
void Empiler(Pile* pP, TypeDonnee elem) {
    Pile q;
    q = (TypeCellule*)malloc(sizeof(TypeCellule)); /* allocation */
    q->donnee = elem;       /* ajout de l'element a empiler */
    q->suivant = *pP;       /* insertion en tete de liste */
    *pP =q;                 /* mise a jour de la tete de liste */
}
```

Implémentation sous forme de liste

Supprimer un élément

La fonction Depiler supprime la tête de liste en cas de pile non vide. La fonction renvoie 1 en cas d'erreur, et 0 en cas de succès. La pile est passée par adresse, on a donc un double pointeur.

```
int Depiler(Pile *pP, TypeDonnee *pelem) {
Pile q;
if (EstVide(*pP))
      return 1;                    /* on ne peut pas supprimer d'élément */
  *pelem = (*pP)->donnee;          /* on renvoie d'élément de tête */
   q = *pP;                /* mémorisation d'adresse de la première cellule */
   *pP = (*pP)->suivant;           /* passage au suivant */
    free(q);                    /* destruction de la cellule mémorisée */
return 0;
}
```

Vider et détruire

La destruction de la liste doit libérer toute la mémoire de la liste chaînée (destruction individuelle des cellules).

```
void Detruire(Pile *pP) {   Pile q;
while (*pP != NULL) {                    /* parcours de la liste */
    q = *pP;                          /* memorisation de l'adresse */
    *pP = (*pP)->suivant;      /* passage au suivant avant destruction : */
    free(q);                          /* destruction de la cellule memorisee */
    }
    *pP = NULL;                        /* liste vide */
}
void Vider(Pile *pP) {
    Detruire(pP);                              /* destruction de la liste */
    *pP = NULL;                       /* liste vide */
}
```

Comparaison entre tableaux et listes chaînées

- Dans les deux types de gestion des piles, chaque primitive ne prend que quelques opérations (complexité en temps constant).

- Par contre, la gestion par listes chaînées présente l'énorme avantage que la pile a une capacité virtuellement illimitée (limitée seulement par la capacité de la mémoire centrale), la mémoire étant allouée à mesure des besoins.

- Au contraire, dans la gestion par tableaux, la mémoire est allouée au départ avec une capacité fixée.

Liste chainée

Avantages

- Utilisation optimale de la Mémoire
- L'espace mémoire ne doit pas être contiguë
- La taille de la liste peut ne Pas être définie au préalable
- Toutes les opérations Sur la tête de liste sont en O(1)

Inconvénients

- L'accès aux données Est plus couteux (temps)
- Ajout à la fin de liste en O(n)
- Gestion des pointeurs

Application des piles

- Dans un navigateur web, une pile sert à mémoriser les **pages Web visitées**. L'adresse de chaque nouvelle page visitée est empilée et l'utilisateur dépile l'adresse de la page précédente en cliquant le bouton « Afficher la page précédente ».
- L'évaluation des expressions mathématiques en **notation post-fixée** (ou polonaise inverse) utilise une pile.
- La fonction « **Annuler la frappe** » (en anglais Undo) d'un traitement de texte mémorise les modifications apportées au texte dans une pile.
- Un algorithme de recherche en profondeur utilise une pile pour mémoriser les nœuds visités.
- Les algorithmes récursifs admis par certains langages (LISP, Algol, Pascal, C, etc.) utilisent implicitement une pile d'appel. Dans un langage non récursif (FORTRAN par exemple), on peut donc toujours simuler la récursivité en créant les primitives de gestion d'une pile.