CHAPITRE 2 : LES PILES

Une pile est une structure de données simple utilisée pour stocker des données (similaire aux listes chaînées). Dans une pile, l'ordre dans lequel les données arrivent est important. Une pile d'assiettes dans une cafétéria est un bon exemple. Les plats sont ajoutés à la pile au fur et à mesure qu'ils sont lavés et sont placés sur le dessus. Lorsqu'un plat est nécessaire, il est pris en haut de la pile. Le premier plat placé dans la pile est le dernier à être utilisé.

Figure 1 : Une pile de plats

DEFINITION : Une pile est une liste ordonnée dans laquelle l'insertion et la suppression sont effectuées à une extrémité, appelée **haut (top)**. Le dernier élément inséré est le premier à être supprimé. Il s'agit donc du principe Last In First Out (**LIFO**) ou First in Last out (**FILO**).

Des noms spéciaux sont donnés aux deux modifications qui peuvent être apportées à une pile. Lorsqu'un élément est inséré dans une pile, le concept est appelé **push**, et lorsqu'un élément est retiré de la pile, le concept est appelé **pop**. Essayer de faire sortir une pile vide est appelé **underflow** et essayer d'ajouter un élément dans une pile pleine est appelé **overflow**. En général, nous les traitons comme des exceptions.

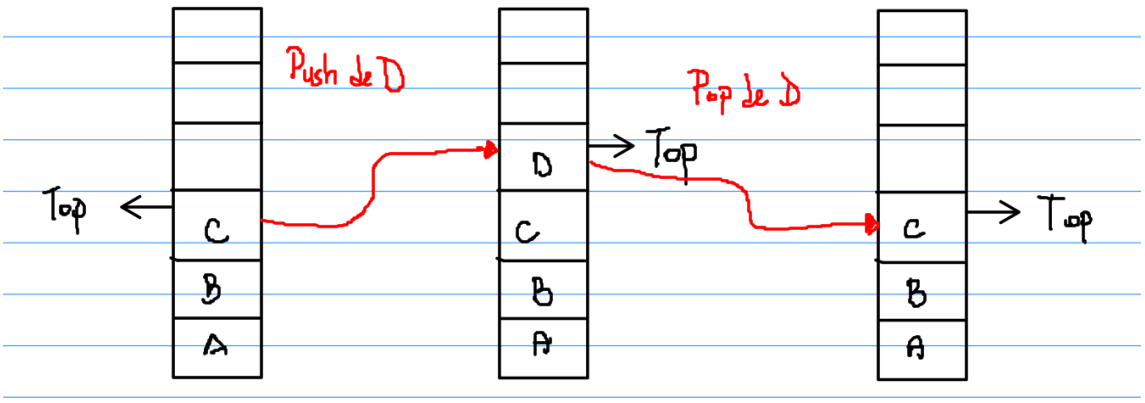

Figure 2 : Exemple de Push et de Pop dans une pile 1/2

LE TYPE ABSTRAIT DE DONNEES PILE :

Pour plus de simplicité, supposons que les données manipulées sont de type entier.

OPERATIONS PRINCIPALES SUR LA PILE :

- **Push(int data)** : Insère *data* **sur** la pile.
- **int Pop()** : Supprime et renvoie le dernier élément inséré dans la pile.

OPERATIONS AUXILIAIRES SUR LA PILE :

- **int Top()** : Retourne le dernier élément inséré sans le supprimer (c'est donc le premier élément sur la pile).
- **int Size()** : Retourne le nombre d'éléments stockés dans la pile (c'est la taille de la pile).
- **int IsEmptyStack()** : Indique si des éléments sont stockés dans la pile ou non (autrement dit, si la pile est vide, on renvoie 0 -False-, sinon, on renvoie 1 -True-).
- **int IsFullStack()** : Indique si la pile est pleine ou non (autrement dit, si la pile est pleine, on renvoie 0 -False-, sinon, on renvoie 1 -True-).

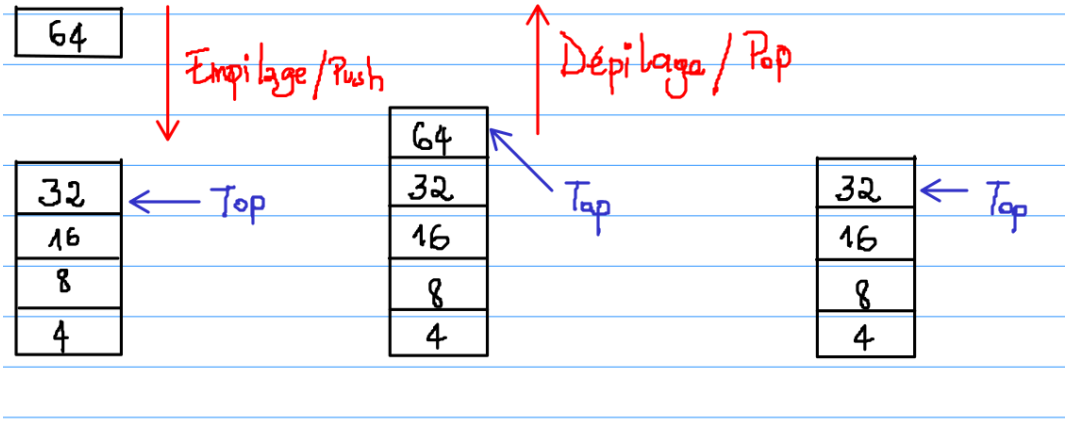

Figure 3: Exemple de Push et de Pop dans une pile 2/2

EXCEPTIONS :

La tentative d'exécution d'une opération peut parfois provoquer une erreur, appelée **exception**. On dit que les exceptions sont "levées" par une opération qui ne peut être exécutée. Dans le TAD Pile, les opérations pop, respectivement push ne peuvent pas être effectuées si la pile est vide respectivement pleine. Essayer d'exécuter pop (push) sur une pile vide (pleine) lève une exception.

APPLICATIONS :

Voici quelques-unes des applications dans lesquelles les piles jouent un rôle important.

APPLICATIONS DIRECTES :

- Equilibrage des symboles : vérifier que toutes les parenthèses/accolades des fonctions sont bien fermées dans un environnement de développement intégré EDI ou IDE en anglais (tel que Visual Studio, PhpStorm, Pycharm, Code::Blocks…)
- Conversions Infix-to-postfix[1] : convertir les expressions mathématiques compréhensibles par les humains en des expressions compréhensibles par les machines. Les expressions infixes sont lisibles et solvables par les humains. Nous pouvons facilement distinguer l'ordre des opérateurs, et nous pouvons également utiliser des parenthèses. L'ordinateur ne peut pas différencier facilement les opérateurs et les parenthèses, c'est pourquoi la conversion postfix est nécessaire.
- Evaluation de l'expression postfix ;
- Mise en œuvre des appels de fonction (y compris la récursion) : les piles sont utilisées pour conserver l'ordre d'exécution des fonctions ;
- Historique des pages visitées dans un navigateur Web : boutons pour reculer ou avancer d'une page ;
- Annulation d'une séquence dans un éditeur de texte : Ctrl + Z ;
- Balises correspondantes en HTML et XML.

APPLICATIONS INDIRECTES :

- Structure de données auxiliaires pour d'autres algorithmes (Exemple : Algorithmes de traversée d'arbre) ;
- Composant d'autres structures de données (Exemple : Simulation de files d'attente[2])

Exemple : Pour vous donner un exemple concret, votre système d'exploitation utilise ce type de structure pour retenir l'ordre dans lequel les fonctions ont été appelées. Imaginez ceci :

- votre programme commence par la fonction **main** (comme d'usage) ;
- vous y appelez la fonction suivante : **ajouter_dans_la_liste** ;
- dans cette fonction, vous faites appel à la fonction **ajouter_en_tete** par exemple ;
- cette fonction appelle à son tour à la fonction **est_vide** ;
- une fois que l'exécution de la fonction **est_vide** est terminée, on retourne à celle de la fonction **ajouter_en_tete** ;
- une fois que l'exécution de la fonction **ajouter_en_tete** est terminée, on retourne à celle de **main** ;
- enfin, une fois la fonction **main** exécutée, il n'y a plus de fonction à appeler, le programme s'achève.

Pour « retenir » l'ordre dans lequel les fonctions ont été appelées, votre ordinateur crée une pile de ces fonctions au fur et à mesure (fig. suivante).

[1] Cf l'annexe qui explicite ce sujet
[2] Plus d'explications au prochain chapitre sur les files

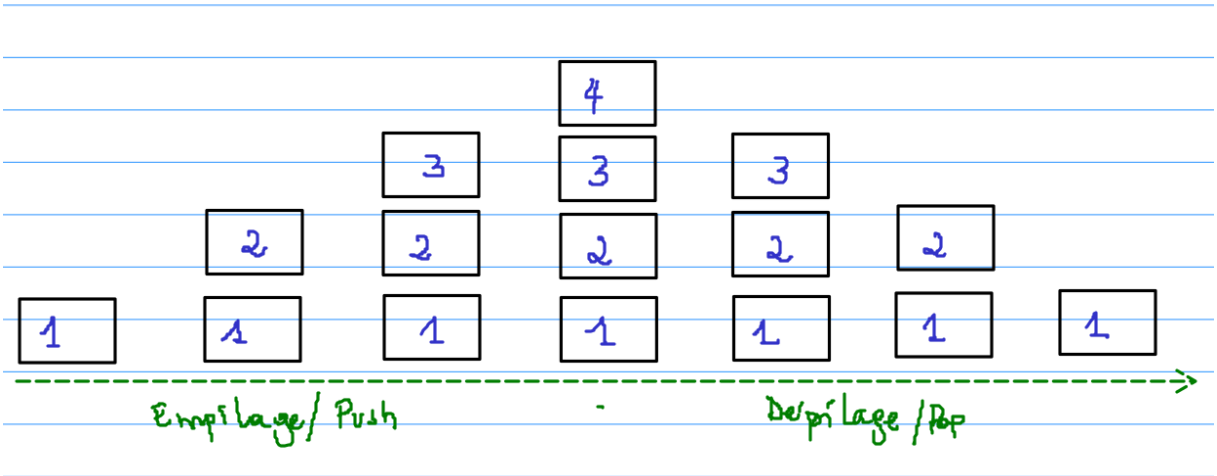

Figure 4 : Enchaînement de l'exécution des fonctions par un ordinateur

1 => main ; 2 => ajouter_dans_la_liste ; 3 => ajouter_en_tete ; 4 => est_vide.

IMPLEMENTATION :

Voici les méthodes les plus couramment utilisées pour implémenter une pile :

- Mise en œuvre simple basée sur un tableau ;
- Mise en œuvre des listes chaînées.

IMPLEMENTATION BASEE SUR UN TABLEAU :

Cette implémentation de la pile utilise un tableau. Dans ce tableau, nous ajoutons des éléments de la gauche vers la droite et utilisons une variable pour suivre l'indice de l'élément supérieur.

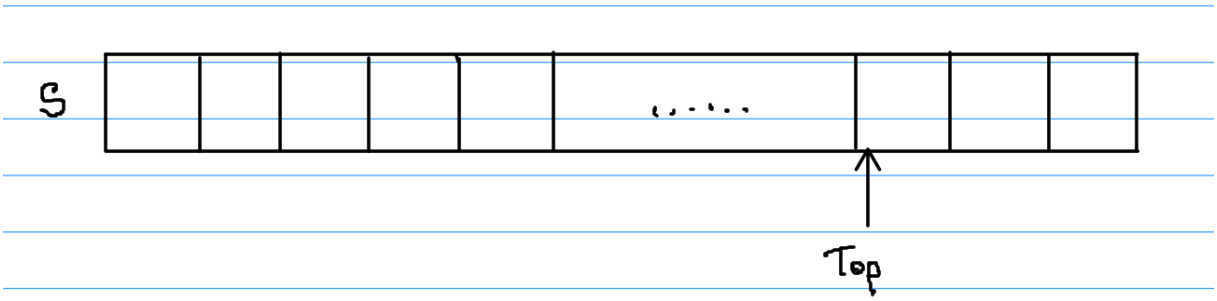

Figure 5 : Implémentation d'une pile à l'aide d'un tableau

IMPLEMENTATION BASEE SUR DES LISTES CHAINNES

Comme pour les listes chaînées, il n'existe pas de système de pile intégré au langage C. Il faut donc le créer nous-mêmes. L'insertion se faisant toujours au début de la pile, le premier élément de la liste sera le dernier élément saisi, donc sa position est en haut de la pile. Il n'y a pas d'utilisation du pointeur fin et le dernier élément de la pile pointe toujours sur NULL.

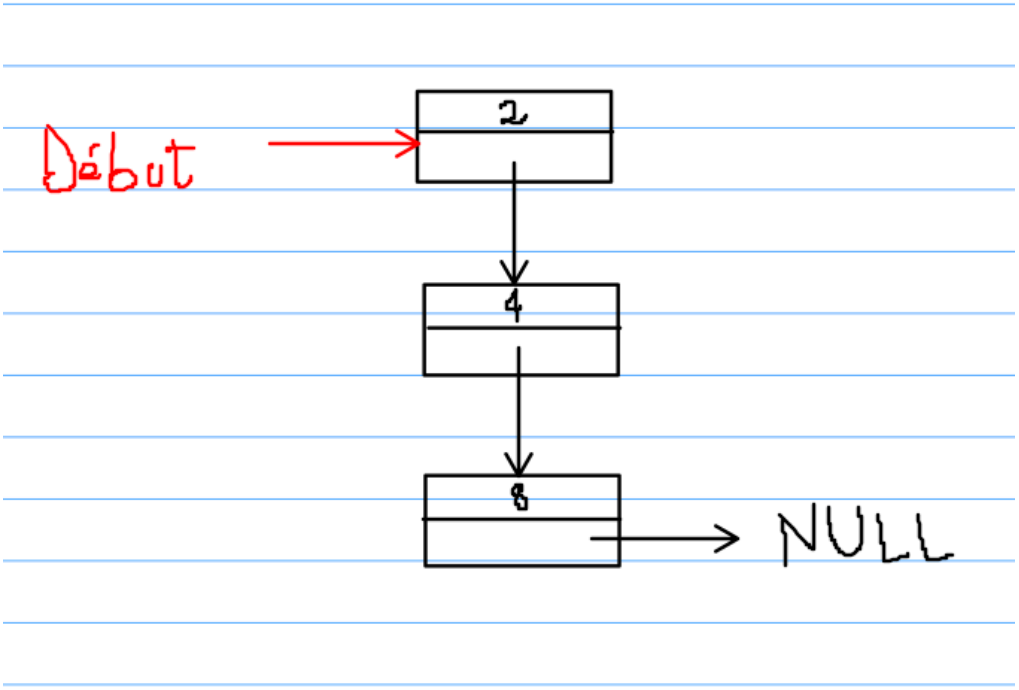

Figure 6 : Implémentation d'une pile à l'aide d'une liste chaînée

Chaque élément de la pile aura une structure identique à celle d'une liste chaînée :

```
2  typedef struct Element Element;
3  struct Element
4  {
5      int nombre;
6      Element *suivant;
7  };
```

Figure 7 : Définition d'un type de données

La structure de contrôle contiendra l'adresse du premier élément de la pile, celui qui se trouve tout en haut.

```
 9  typedef struct Pile Pile;
10  struct Pile
11  {
12      Element *premier;
13  };
```

Figure 8 ; Définition d'un autre type de données

Initialisation de la pile :

```
15 void initialisation (Pile * tas){
16    tas->premier = NULL;
17 }
```

Figure 9 : Initialisation d'une pile

Insertion d'un élément dans la pile :

Voici les étapes de l'algorithme d'insertion (empilage ou push) :

- Déclaration d'élément(s) à insérer ;
- Allocation de la mémoire pour le nouvel élément ;
- Remplissage du contenu du champ de données ;
- Mise à jour du pointeur *premier* vers le nouveau 1er élément (en haut de la pile / top).

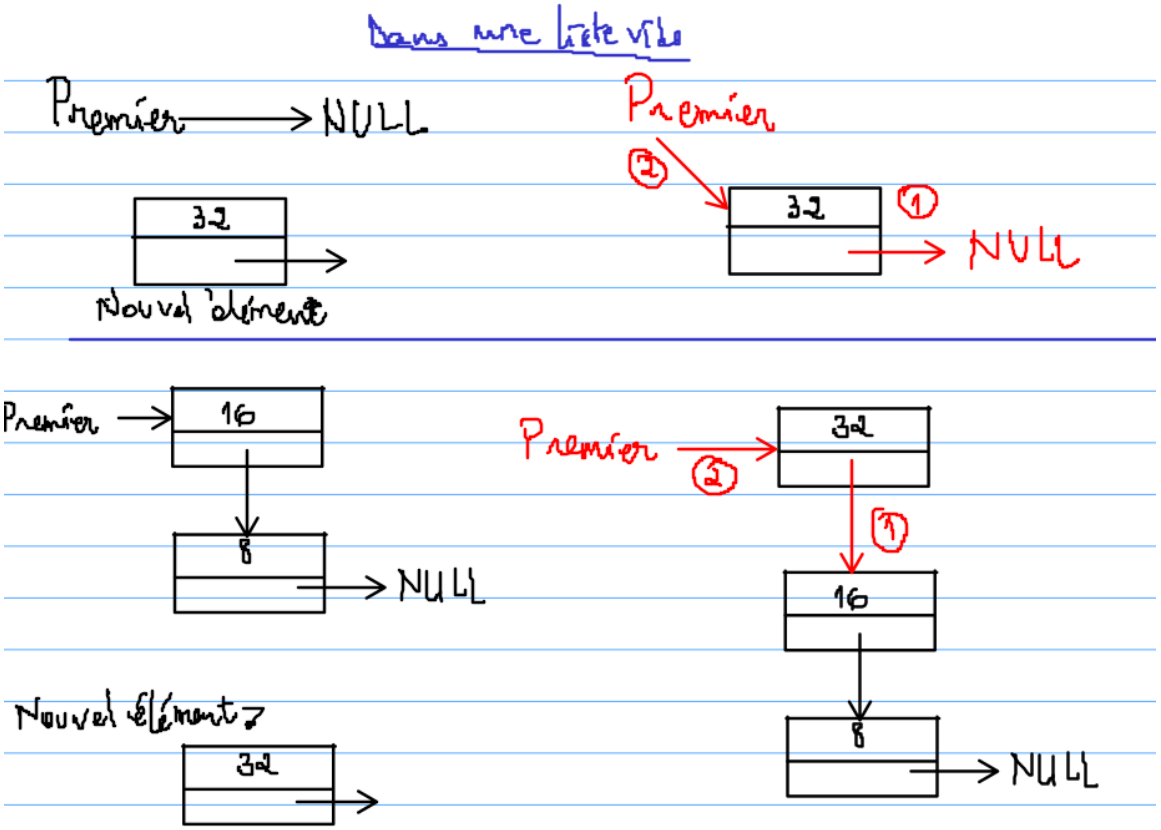

Figure 10 : Illustration de l'insertion d'un élément dans une pile vide

```c
/* empiler (ajouter) un élément dans la pile */
int empiler (Pile * tas, char *donnee){
  Element *nouveau_element = (Element *) malloc (sizeof (Element))) == NULL;
  if (nouveau_element == NULL)
    return -1;
  //copie de la donnée dans le nouvel élément
  strcpy (nouveau_element->donnee, donnee);
  nouveau_element->suivant = tas->premier;
  tas->premier = nouveau_element;
}
```

Figure 11 : Codes de l'insertion d'un élément dans une pile vide

Ôter un élément de la pile :

Pour supprimer (ôter ou dépiler ou pop) l'élément de la pile, il faut tout simplement supprimer l'élément vers lequel pointe le pointeur *premier*.

Cette opération ne permet pas de récupérer la donnée en haut de la pile, mais seulement de la supprimer.

Les étapes :

- Le pointeur *supp_elem* contiendra l'adresse du 1er élément ;
- Le pointeur *premier* pointera vers le 2ème élément (après la suppression du 1er élément, le 2ème sera en haut de la pile).

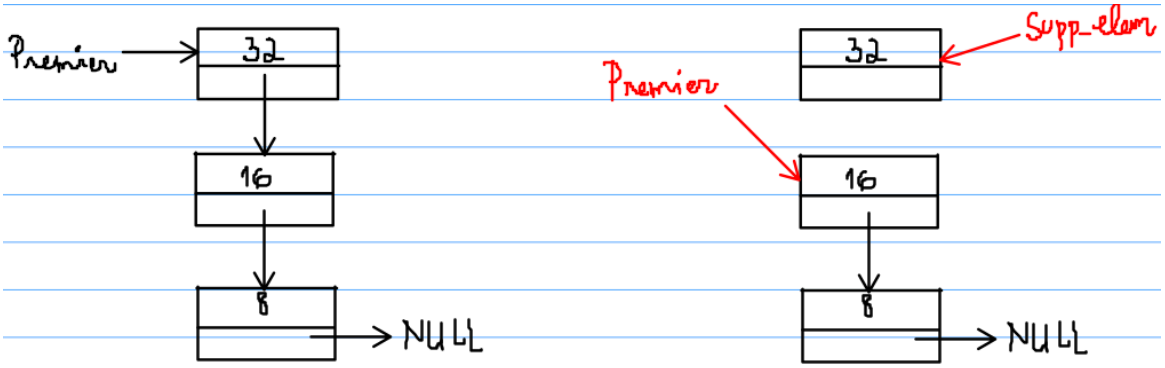

Figure 12 : Illustration de la suppression d'un élément dans une pile vide

```c
int depiler (Pile * tas){
  Element *supp_element;
  if (isEmptyStack(*tas) == 1)
    return -1;
  supp_element = tas->premier;
  tas->premier = tas->premier->suivant;
  free (supp_element->donnee);
  free (supp_element);
  return 0;
}
```

Figure 13 : Codes de la suppression d'un élément dans une pile vide

Affichage de la pile :

Pour afficher la pile entière, il faut se positionner au début de la pile (le pointeur *premier* le permettra). Ensuite, en utilisant le pointeur *suivant* de chaque élément, la pile est parcourue du 1er vers le dernier élément. La condition d'arrêt peut être donnée par la taille de la pile.

```
41  /* affichage de la pile */
42  void affiche (Pile * tas){
43    Element *courant;
44    int i;
45    int taille = size(*tas);
46    courant = tas->premier;
47
48    for(i=0; i<tas->taille; ++i){
49      printf("\t\t%s\n", courant->donnee);
50      courant = courant->suivant;
51    }
52  }
```

Figure 14 : Code de l'affichage d'une pile