Les Piles et les Files

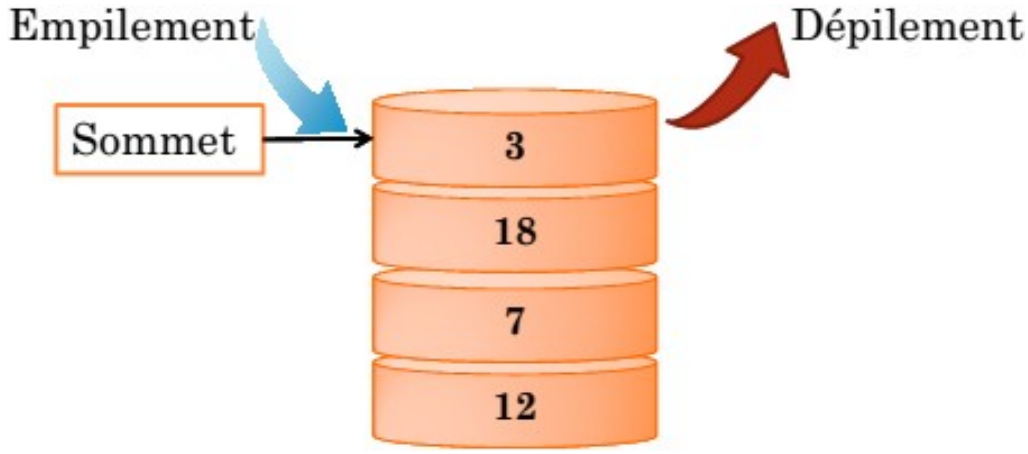
Les Piles

Introduction :

Une pile (en anglais stack) est une structure de données fondée sur le principe «dernier arrivé, premier sorti» (ou LIFO : Last In, First Out).

Une pile peut être définie comme une collection d'éléments dans laquelle tout nouvel élément est inséré à la fin (empilement) et tout élément ne peut être supprimée que de la fin (dépilement).

Les insertions et les suppressions d'éléments se font à une seule et même extrémité de la liste appelée le sommet de la pile.

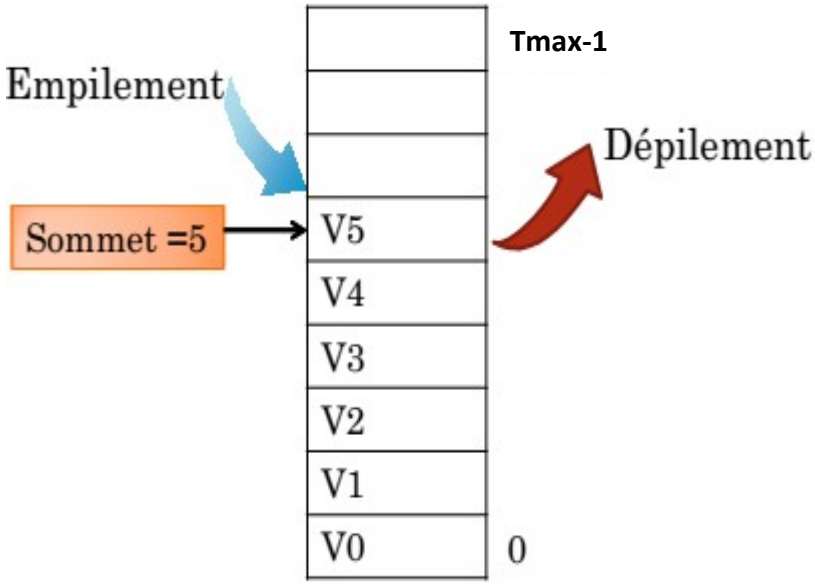


Les piles peuvent être présentées en deux manières :

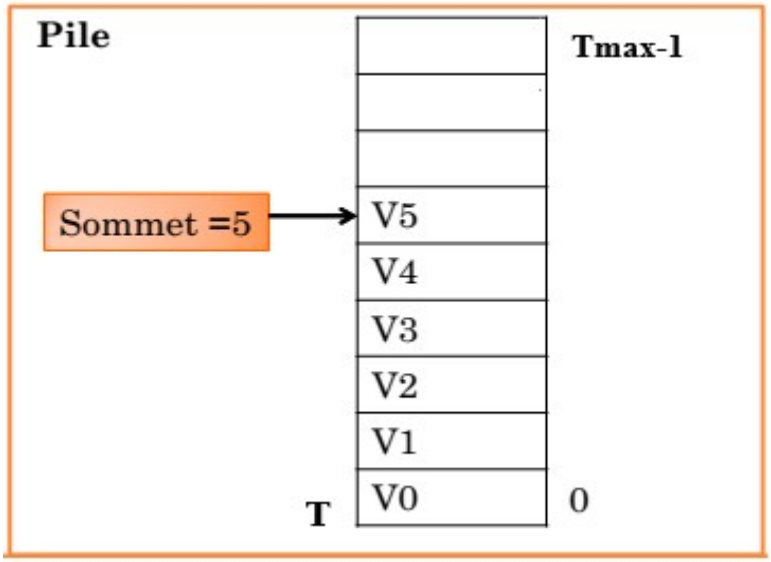
- Statique en utilisant les tableaux,
- Dynamique en utilisant les listes linéaires chaînées.

I. Implémentation statique (par décalage) :

L'implémentation statique des piles utilise les tableaux. Dans ce cas, la capacité de la pile est limitée par la taille du tableau (Tmax). L'ajout à la pile se fait dans le sens croissant des indices, tandis que le retrait se fait dans le sens inverse.



a) Structures de données :

.	*Implémentation dynamique en C*
Schéma	
Définition de la taille maximale	*# define Tmax 1000*
Définition du type pile	*typedef struct { typeqlq T[Tmax] ; int Sommet;} Pile ;*
Déclaration d'une pile	*Pile P ;*

b) Implémentation du modèle en C :

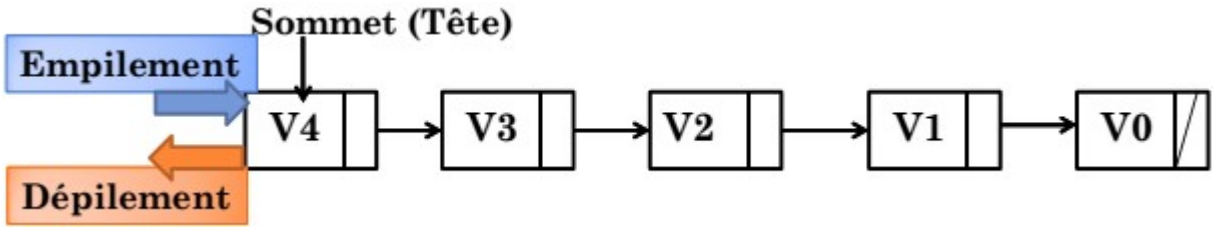
Module	Implémentation en C	Rôle
InitPile(&p) (proc)	void InitPile (pile *p) {(*p).Sommet = -1 ;}	Initialise la pile à vide
Pilepleine(&p); (fct)	int Pilepleine (Pile *p) {if((*p).Sommet ==Tmax-1) return -1; else return 0 ; }	Retourne -1 si la pile est pleine et 0 sinon
Pilevide(&p) ; (fct)	int Pilevide (Pile *p) {if((*p).Sommet == -1) return -1; else return 0 ; }	Retourne -1 si la pile est vide et 0 sinon

Empiler(&p, x); (proc)	void Empiler(Pile *p, typeqlq x){
	if (!Pilepleine(p)) {
	(*p).Sommet++ ;
	(*p).T[(*p).Sommet]=x; } }
		Place l'élément x au sommet de la pile (la pile ne doit pas être pleine).

Desempiler(&p,&x); (proc)	void Desempiler(Pile *p, typeqlq *x) {
	if (! Pilevide(p)) {
	(*x)= (*p).T[(*p).Sommet] ;
	(*p).Sommet-- ; } }
		Retire l'élément au sommet de la pile et le place dans x (la pile ne doit pas être vide)

II. Implémentation dynamique :

L'implémentation dynamique utilise les listes linéaires chinées. Dans ce cas, la pile peut être vide, mais ne peut être jamais pleine, sauf bien sûr en cas d'insuffisance de l'espace mémoire. L'empilement et le dépilement dans les piles dynamique se font à la tête de la liste.



a) Structures de données :
Puisque les piles sont implémentées sous forme de listes linéaires chainées, donc on peut utiliser le fichier de définition des structures de données des listes chainées :
 # include "liste.h"
Puis on a qu'à attribuer le nom **Liste** au type **Pile** :
typedef Liste Pile ;
La pile est identifier par son sommet (l'adresse du premier élément), pour déclarer un sommet d'une pile on écrit :
Pile P ;

b) Implémentation du modèle en C :
Tout d'abord on fait inclure les primitives de manipulations des listes linéaires chainées :
 # include "liste.h"

Module	Implémentation en C	Rôle
InitPile(&p) (proc)	void InitPile (pile *p) {Init(p) ; } /* Init est la procédure qui initialise une LLC */	Initialise la pile à vide
Pilepleine(&p); (fct)	int Pilepleine (Pile *p) {if(freenem() >= sizeof(maillon)) return 0; else return -1 ; }	Retourne -1 si la pile est pleine et 0 sinon
Pilevide(&p) ; (fct)	int Pilevide (Pile *p) {if((*p) == NULL) return -1; else return 0 ; }	Retourne -1 si la pile est vide et 0 sinon
Empiler(&p, x); (proc)	Void Empiler(Pile *p, typeqlq x){ InsererTete(p, x); } /*déjà écrite avec les listes */	Place l'élément x au sommet de la pile (la pile ne doit pas être pleine).
Desempiler(&p,&x); (proc)	void Desempiler(Pile *p, typeqlq *x) { liste S = *p ; (*x)= Val(*p); (*p) = Adr(*p); Libérer(S); }	Retire l'élément au sommet de la pile et le place dans x (la pile ne doit pas être vide)

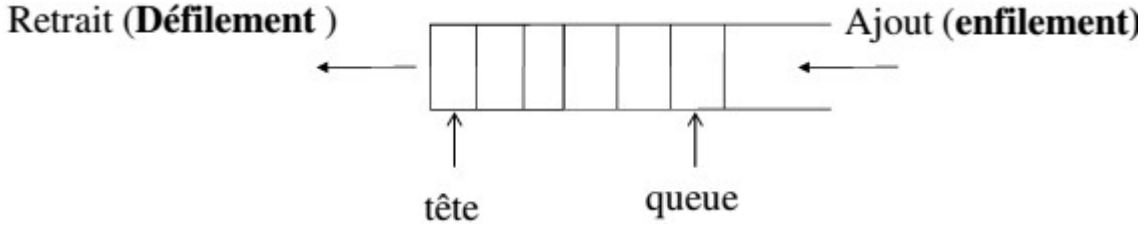
******Les files******

Introduction :

Une file (en anglais queue) est une structure de données basée sur le principe «premier arrivé, premier sorti» (en anglais FIFO : First In First Out),

Le fonctionnement ressemble à une file d'attente : les premières personnes à arriver sont les premières personnes à sortir de la file.

Une file d'attente peut être définie comme une collection d'éléments dans laquelle tout nouvel élément est inséré (ajouté) à la fin (queue) et tout élément ne peut être supprimé (retiré) que du début (Tête).



Les files d'attente peuvent être présentées en deux manières :

Statique en utilisant les **tableaux**,

Dynamique en utilisant les **listes linéaires chaînées**.

L'implémentation statique peut être réalisée par

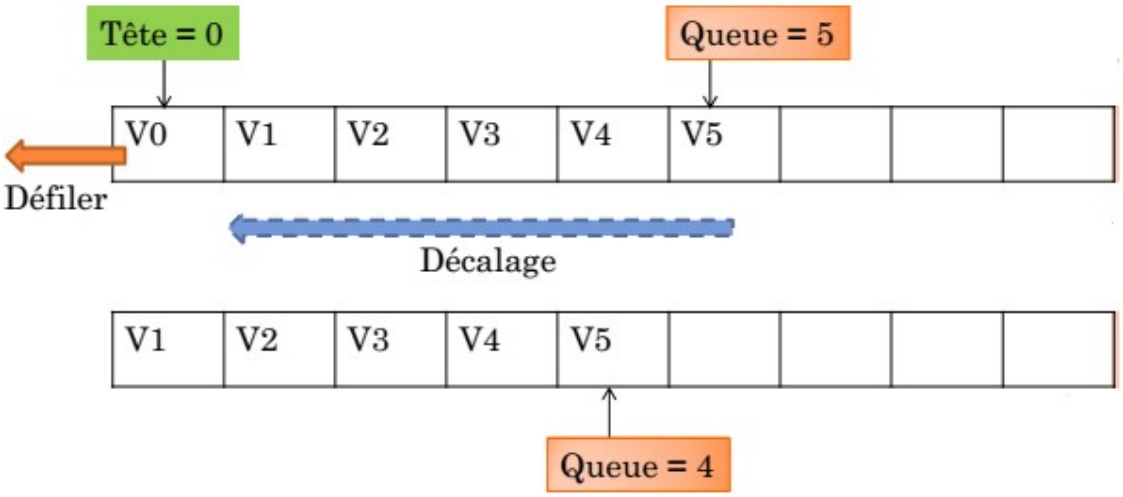
Décalage en utilisant un tableau avec une tête fixe, toujours à 0, et une queue variable.

Tableau circulaire où la tête et la queue sont toutes les deux variables.

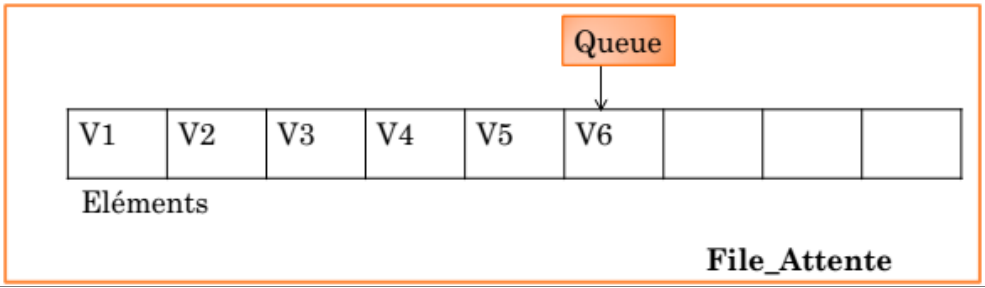
I. Implémentation statique par décalage :

A chaque défilement, on fait un décalage.

La tête n'est plus une caractéristique puisqu'elle est toujours égale à 0.



a) Structures de données :

.	*Implémentation statique en C*
Schéma	
Définition de la taille maximale	*# define Tmax 1000*
Définition du type File	*typedef struct { typeqlq T[Tmax] ; int Queue;} File ;*
Déclaration d'une file	*File F ;*

b) Implémentation du modèle en C :

Module	Implémentation en C	Rôle
InitFile(&F) (proc)	void InitFile (File *F) {F--> Queue = -1 ;}	Initialise la file à vide
Filepleine(&F); (fct)	int Filepleine (File *F) {if (F--> Queue ==Tmax-1) return -1; else return 0 ; }	Retourne -1 si la file est pleine et 0 sinon
Filevide(&F) ; (fct)	int Filevide (File *F) {if (F--> Queue == -1) return -1; else return 0 ; }	Retourne -1 si la file est vide et 0 sinon

Enfiler(&F, x); (proc)	void Enfiler(File *F, typeqlq x){
	if (!Filepleine(F)) {
	F--> Queue++ ;
	F-->T[F--> Queue]=x; } }
		Place l'élément x à la queue de la file (la File ne doit pas être pleine).

Defiler(&F,&x); (proc)	void Defiler(File *F, typeqlq *x) {
	if (! Filevide(F)) {
	(*x)= (F-->T)[0] ;
	for (i=0; i< F-->Queue; i++)
		Retire l'élément à la tête de la file et le place dans x (la file ne doit pas être vide)

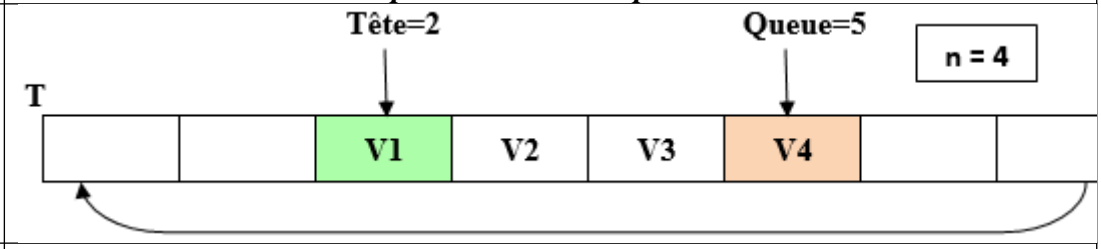
	(F-->T)[i]= (F--> T)[i+1]; F--> Queue-- ; } }	

II. Implémentation statique par tableau circulaire :

Les incrémentations se font modulo TMax afin de réutiliser des cases libérées :

Tete ← (Tete+1) mod TMax & Queue ← (Queue+1) mod TMax

a) Structures de données :

.	***Implémentation statique en C***
Schéma	
Définition de la taille maximale	***# define Tmax 1000***
Définition du type File	***typedef struct { typeqlq T[Tmax] ; int Tete, Queue, n;} File ;*** */*n : nombre des éléments*/*
Déclaration d'une file	***File F ;***

b) Implémentation du modèle en C :

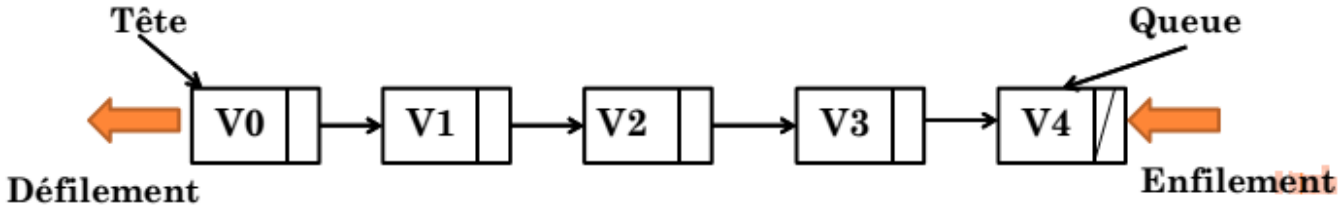
Module	Implémentation en C	Rôle
InitFile(&F) (proc)	void InitFile (File *F) {F--> Queue = -1 ; F--> Tete = 0 ; F--> n= 0;}	Initialise la file à vide
Filepleine(&F); (fct)	int Filepleine (File *F) {if (F--> n ==Tmax) return -1; else return 0 ; }	Retourne -1 si la file est pleine et 0 sinon
Filevide(&F) ; (fct)	int Filevide (File *F) {if (F--> n == 0) return -1; else return 0 ; }	Retourne -1 si la file est vide et 0 sinon

Enfiler(&F, x); (proc)	void Enfiler(File *F, typeqlq x){
	if (!Filepleine(F)) {
	F-->Queue= (F--> Queue+1)%Tmax;
	(F-->T)[F--> Queue]=x;
	(F-->n) ++ ; } }
		Place l'élément x à la queue de la file (la File ne doit pas être pleine).

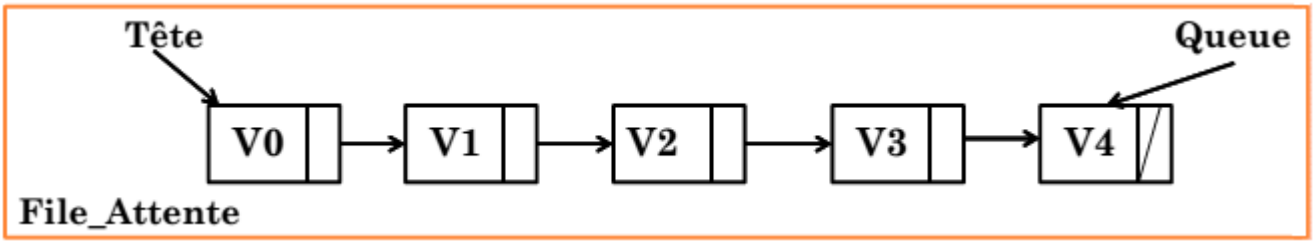
Defiler(&F,&x); (proc)	void Defiler(File *F, typeqlq *x) {
	if (! Filevide(F)) {
	(*x)= (F-->T)[F-->Tete] ;
	F-->Tete= (F-->Tete + 1) % Tmax ;
	(F-->n) -- ; } }
		Retire l'élément à la tête de la file et le place dans x (la file ne doit pas être vide)

III. Implémentation dynamique :

La représentation dynamique utilise une liste linéaire chaînée (LLC). L'enfilement se fait à la queue de la liste et de défilement se fait de la tête. La file d'attente, dans ce cas, peut devenir vide, mais ne sera jamais pleine.



a) Définition des structures de données :



Puisque les files sont implémentées sous forme de listes linéaires chainées, donc on peut utiliser le fichier de définition des structures de données des listes chainées :
 # include "liste.h"
Puisque l'accès à la file se fait à partir de la tête (pour défilement) et à partir de la queue (pour empilement) alors on écrit :
typedef struct { liste Tete, Queue} File ;
Pour déclarer une variable de type File on écrit :
File F ;

b) Implémentation du modèle en C :
Tout d'abord on fait inclure les primitives de manipulations des listes linéaires chainées :
 # include "liste.h"

Module	Implémentation en C	Rôle
InitFile(&p) (proc)	void InitFile (File *F) { F-->Tete = F->Queue = NULL ; }	Initialise la file à vide
Filevide(&p) ; (fct)	int Filevide (File *F) {if(F-->Tete == NULL) return -1; else return 0 ; }	Retourne -1 si la file est vide et 0 sinon

Enfiler(&p, x); (proc)	void Enfiler(File *F, typeqlq x){
	liste p ;
	Allouer(&p) ;
	if(p !=NULL) {
	Aff_Val(p, x) ; Aff_Adr(p, NULL) ;
	if(F-->Queue == NULL) //cas File vide
	F-->Tete = F-->Queue=p;
	else { //cas File non vide
	Aff_Adr(F-->Queue, p);
	F-->Queue=p; }}}
		Place l'élément x à la queue de la file (la file ne doit pas être pleine).

Desemfiler(&p,&x); (proc)	void Desemfiler(File *F, typeqlq *x) {
	liste p;
	if(! FileVide(F)){
	(*x)= Val(F-->Tete);
	p= F-->Tete ;
	F-->Tete= Adr(F-->Tete);
	Libérer(p);
	if(F-->Tete==NULL) //cas File vide
	F-->Queue=NULL; }
		Retire l'élément à la tête de la file et le place dans x (la file ne doit pas être vide)

Utilisation des piles dans l'évaluation des expressions arithmétiques

Une expression arithmétique est écrite au départ en forme infixé. L'évaluation de cette expression n'est pas évidente puisque dès la rencontre d'un opérateur on ne sait pas si on doit évaluer l'opération ou attendre plus loin pour avoir s'il existe des opérateurs plus prioritaires, ainsi le compilateur après les phases de l'analyse lexicale et syntaxique réalise la transformation de l'expression d'une forme infixé vers une forme intermédiaire comme la forme postfixé par exemple :

x+y*z ==> xyz*+

car : xyz*+ = x(y*z)+ = x+y*z.

Exercice : Il est demandé de réaliser une calculette qui prend en entrée une LLC représentant une expression arithmétique en notation infixée, et qui affiche le résultat de l'évaluation de cette expression, en utilisant une pile statique.

Les structures de données :

```
Type Terme = structure
        Op : Chaines de caractères ;
        Prio : {0,1, 2, 3, 4};
Fin ;
Type Maillon = structure
        Val : Terme;
        Suiv : *Maillon ;
Fin ;
Type Liste = *Maillon ;
```

Prio	Signification
0	Opérande
1	Opérateur binaire (+, -)
2	Opérateur binaire (*, /, %)
3	Opérateur unaire (- unaire)
4	Parenthèse ouvrante ou fermante

```
//Pile utilisée pour sauvegarder les termes (chaine + priorité)
Type PileTerme= structure
        T : tableau[TMax] de Terme ;
        Sommet : entier ;
Fin ;
```

{Pile utiliser pour l'évaluation des opérations (une opération comporte deux opérandes et un opérateur sous forme de chaines}

```
Type PileOp= structure
        T : tableau[TMax] de chaines de caractères
        Sommet : entier
Fin ;
```

Algorithme de transformation d'une forme infixée en une forme post fixée :

A partir d'une liste **Linfixe** qui représente une expression sous forme infixé, et à l'aide d'une pile statique **P**, on veut construire une liste **Lpostfixe** qui va contenir la représentation de l'expression précédente sous forme postfixée.

Pour chaque terme de l'expression infixée (élément de la liste Infixée) :

Si c'est un opérande alors on insère dans la liste $L_{postfixée}$

Si c'est une parenthèse ouvrante alors on doit l'empiler tout le temps.

Si c'est une parenthèse fermante alors on doit dépiler jusqu'à trouver la parenthèse ouvrante associée. Les éléments dépilés sont insérés dans la liste $L_{postfixée}$

Si c'est un opérateur alors on dépile à partir de la pile tous les termes ayant une priorité inferieur ou égale à celle de l'opérateur et on les inserts à la fin de la liste $L_{postfixée}$ puis on empile l'opérateur.

```
Procédure InfixeAPostfixe( Linfixe: Liste, var Lpostfixe : Liste) ;
Variable      P: PileTerme;          Q: Liste;      X, Y: Terme ;
Debut
Lpostfixe ←Nil;      Q←Linfixe;          InitPile(P)
Tant que (Q≠Nil) faire
      X ←valeur (Q);
      Si X.prio = 0 alors  //cas opérande
            InsererListeFin (Lpostfixe, X) ;
      Finsi ;
      Si X.op ='(' alors Empiler(P, X) ;      finsi ;
      Si X.op = ')' alors      // depiler et insérer dans Lpostfixe jusqu'à la parenthèse ouvrante
            Tant que P.T[P.sommet]≠ '(' faire
                  Depiler(P, Y);          InsererListeFin (Lpostfixe, Y) ;
            Fait ;
            Depiler(P, Y);    //Dépiler la parenthèse ouvrante
      Finsi ;
      Si X.prio = 1 ou  X.prio =2 ou X.prio =3 alors          //c'est un operateur
          Tant que (non Pilevide(P)) et (P.T[P.Sommet].prio ≥X.prio) et (P.T[P.Sommet].op≠'(') faire
             depiler(P,Y) ;
             InsererLLCFin (Lpostfixe, Y) ;
          Fait ;
          Empiler(P, X) ;
      Finsi ;
      Q←suivant (Q);
Fait ;
// dépiler les opérateur de grande priorité et les insérer dans la liste
Tant que non Pilevide(P) faire
        depiler(P,Y)
        InsererLLCFin (Lpostfixe, Y)
Fait ;
Fin
```

Algorithme d'évaluation d'une forme post fixée :

A partir d'une liste **Lpostfixe** qui représente une expression sous forme postfixée, et à l'aide d'une pile statique **P**, on veut calculer la valeur de l'expression se trouvant dans **Lpostfixe**.

Pour chaque terme de la liste Lpostfixe :

> **Si c'est un opérande** alors on l'empile dans P
> **Si c'est un opérateur** alors
>
> > **Si c'est un opérateur unaire** (moins unaire) alors on dépile une fois, on multiple par (-1) ensuite on empile le résultat
> > **Si c'est opérateur binaire** alors on dépile deux fois, on fait le calcul correspondant, ensuite on empile le résultat.

A la fin : Le résultat final se trouve au sommet de pile

Afin de pouvoir effectuer le calcul d'une opération on utilise la procédure suivante :

```
Procédure Calcul (op : caractère, op1 ; chaine de caractères, op2 : chaine de caractères, var res : chaines de caractères) ;
Variable      X, Y, Z : entier ;
Debut
// convertir les opérandes en entier
```

```
X ←ConvertirChaineEntier(op1) ;
Y ←ConvertirChaineEntier(op2) ;
//réaliser l'opération correspondante
Selon op
        Cas '+': Z←X+Y ;
        Cas '-': Z←X-Y ;
        Cas '*': Z←X*Y ;
        Cas '/': Z←X/Y ;
        Cas '%': Z←X%Y ;
Finselon ;
//convertir le résultat en chaîne de caractère
Res ←ConvertirEntierChaine(Z) ;
Fin
```

Et voici la fonction qui permet d'evaluer l'expression post fixée qui se présente dans la liste Lpostfixe.

```
Fonction Eval_Postfixe (Lpostfixe : Liste) : entier ;
Variable    P : PileOp ;   Q : Liste ;   X : terme ;   Y : chaine de caractères ;
début
Initpile(P);     Q←Lpostfixe ;
Tant que (Q≠Nil) faire
      X←valeur (Q) ;
      Si (X.prio=0) alors      Empiler( P, X.op) ;              //opérande
      Sinon    /* donc c'est un opérateur */
               Si (X.prio=3)          // moins unaire
                        Depiler(P,Y);          Calcul('*','-1', Y, Y);          Empiler(P, Y)
               Sinon
                        Si (X.prio=1) ou (X.prio=2) alors    /* donc opérateur binaire */
                                 Depiler(P,b);            Depiler(P,a);
                                 Calcul(X.op, a, b, a);          Empiler( P, a);
                        Finsi;
               Finsi;
        Finsi;
        Q←suivant (Q) ;
Fait ;
Depiler( P, res ) ;      // le résultat se trouve au sommet
Eval_Postfixe← ConvertirChaineEntier(res);
Fin
```

File d'attente avec priorité

Une file d'attente avec priorité est une collection d'éléments dans laquelle l'insertion ne se fait pas toujours à la queue.

Tout nouvel élément est inséré dans la file, selon sa priorité.)

Le retrait se fait toujours du début.

Dans une file avec priorité, un élément prioritaire prendra la tête de la file même s'il arrive le dernier.

Un élément est toujours accompagné d'une information indiquant sa priorité dans la file.

Par rapport au modèle de la file déjà vue, on ne changera que la primitive 'Enfiler'.

Exercice :

Le processeur possède un buffer permettant de gérer ses programmes à exécuter. Ce buffer est une file d'attente avec priorité où les éléments (programmes) sont caractérisés par une priorité d'exécution : un programme de priorité supérieure est servi même s'il n'est pas arrivé le premier.

Sachant qu'un programme est défini par sa priorité et son mot d'état programme (Program Status Word ou PSW) qui regroupe l'adresse de la prochaine instruction (CO) et les bits du registre de flag.

- Définir les structures de données nécessaires ainsi que le modèle de file, pour chaque type de présentation (statique par décalage, statique par tableau circulaire et dynamique).

1) Implémentation statique par décalage :

```
TYPE prog = STRUCTURE
        prio : ENTIER ;
        PSW : typeqcq ;
FIN ;
TYPE File_Attente = STRUCTURE
        T : TABLEAU[Max] de prog ;
        Queue : ENTIER ;
FIN ;

Procédure Enfiler (var F : File, X : typeqlq) ;
Variable    i : entier ;
Début
SI (NON Filepleine(F)) alors
        i ← F.Queue ;
        //décaler les programmes moins prioritaires jusqu'à trouver la bonne position où enfiler X
        Tant que (I ≥0) & (F.T[i].priorité < X.priorité) faire
                F.T[i+1] ← F.T [i];
                i←i-1;
        Fait;
        //insérer le programme X
        F.T[i] ← X ;
        //mettre à jour la queue
        F.Queue ← F.Queue + 1 ;
Finsi ;
Fin
```

2) Implémentation statique par tableau circulaire :

```
TYPE prog = STRUCTURE
```

```
        prio : ENTIER ;
        PSW : typeqcq ;
FIN ;
TYPE File_Attente = STRUCTURE
        T : TABLEAU[Max] de prog ;
        Queue, Tete, n : ENTIER ;
FIN ;
Procédure Enfiler (var F : File, X : typeqlq) ;
Variable    I, J : entier ;
Début
SI (NON Filepleine(F))
        //mettre à jour la queue
        F.Queue ← (F.Queue + 1) mod Max ;
        //insérer le programme X ;
        F.T[F.Queue] ← X ;              F.n←F.n + 1 ;
        //trier le tableau d'éléments
        I ← F.Queue ;
        Si I=0 alors     J←TMax-1;    Sinon J←I-1 ;          finsi ;
        Tant que (I ≠F.Tete) et (F.T[I].priorité > F.T[J].priorité ) faire
                Tmp←F.T[I] ;          F.T[I] ← F.T[J] ;       F.T[J] ← Tmp ;
                Si I=0 alors     I←TMax-1;    Sinon I←I-1 ;          finsi ;
                Si J=0 alors     J←TMax-1;    Sinon J←J-1 ;          finsi ;
        Fait ;
Finsi ;
Fin
```

3) Implémentation dynamqiue :

```
TYPE prog = STRUCTURE
        prio : ENTIER ;
        PSW : typeqcq ;
FIN
TYPE Maillon = STRUCTURE
        val : prog ;
        suiv: * Maillon ;
FIN
TYPE File_Attente = STRUCTURE
        TêTe, Queue : *Maillon
FIN

Prodéure Enfiler (var F : File, X : typeqlq) ;
Variable    P, S, Q : Liste ;
Début
Allouer (P) ;    Aff_Val(P, X) ;         aff_suiv(P, NIL)
Si (Filevide(F)) alors          F.Tete ← P;    F.Queue ← P ;
Sinon
        Si (X.prio >Valeur(F.Tete).prio) alors         //insertion au début
                aff_suiv(P, F.Tete);              F.Tete←P;
        Sinon
                S←F.Tete ;
                Tant que (S≠Nil) et (valeur(S).prio>=X.prio) faire
                        Q←S ;          S← suivant (S) ;
                Fait ;
                Si (S = Nil) alors //insertion à la fin
                        aff_suiv (Q, P);         F.Queue ← P ;
```

```
                Sinon  //insertion interne
                        aff_suiv(Q, P);          aff_suiv(P, S) ;
                finsi;
        finsi;
finsi;
Fin
```