Les piles (LIFO)

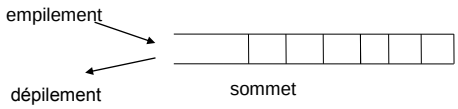
1. *Définition, principe, domaine d'application*
2. *Modèle*
3. *Implémentation*
4. *Exemple d'application*

Les piles (LIFO)

1. Définition, principe, domaine d'application

La pile constitue l'un des concepts les plus utilisés dans la science des ordinateurs.
Une pile peut être définie comme une collection d'éléments dans laquelle tout nouveau élément est inséré à la fin et tout élément ne peut être supprimé que de la fin.

C'est le principe "**LIFO**", abréviation de "Last In First Out" qui veut dire "dernier entré premier servi".



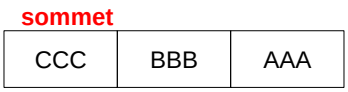
La pile est très utilisée dans le domaine de la compilation : résolution de la portée des objets, récursivité, évaluation d'expressions, etc..
Nous verrons aussi que la pile est utilisée pour le parcours des arbres et pour résoudre un grand nombre de problèmes.

3

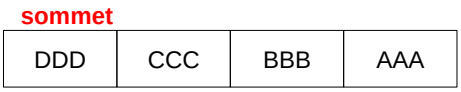
Les piles (LIFO)

Exemple

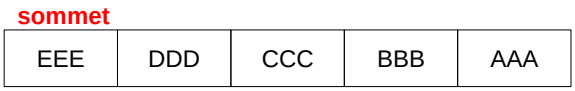
Une pile contenant 3 éléments



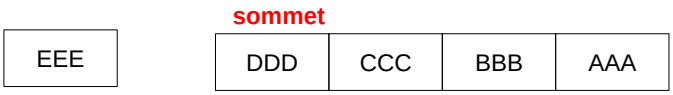
après l'empilement de DDD



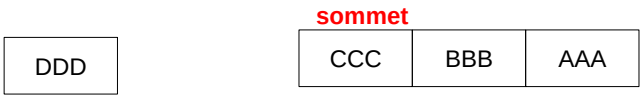
après l'empilement de EEE



après un dépilement



après un 2e dépilement



4

Les piles (LIFO)

2. Modèle

On définit une machine abstraite sur les piles avec l'ensemble des opérations suivantes :

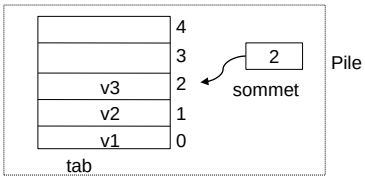
Créerpile(P)	créer une pile vide
Empiler(P,Val)	ajouter Val en sommet de pile.
Dépiler(P,Val)	retirer dans Val l'élément en sommet de pile.
Pilevide((P)	tester si la pile est vide.
Pilepleine(P)	tester si la pile est pleine

5

Les piles (LIFO)

3. Implémentation

3.1. Au moyen de tableaux



Description algorithmique :

```
TYPE   pile = STRUCTURE
            Sommet : ENTIER
            Tab : TABLEAU (1..Max) DE Typeqq
FIN
VAR   P : pile
Créerpile(P)
            P.Sommet := 0
Pilevide(P)
            Pilevide := ( P.Sommet = 0)
```

6

Les piles (LIFO)

```
Pilepleine(P)
        Pilepleine := (P.Sommet = Max)

Empiler(P, X)
        SI NON Pilepleine(P)
                P.Sommet := P.Sommet + 1
                P.Elements(P.Sommet) := X
        SINON
                "Overflow"
        FSI
Depiler(P, X)
        SI NON Pilevide(P)
                X := P.Elements(P.Sommet)
                P.Sommet := P.Sommet – 1
        SINON
                " Underflow"
        FSI
```

7

Les piles (LIFO)

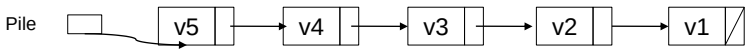
En C :

```
typedef struct {
        int   tab [N];
        int sommet;
} Pile;
void CreerPile( Pile *p )
        { p->sommet = -1; }
int PileVide( Pile p )
        { return (p.sommet == -1); }
int PilePleine( Pile p)
        { return (p.sommet == N-1); }
int Empiler( int x, Pile *p ) {
        if ( PilePleine(*p) ) return 0;
        p->tab[++(p->sommet) ] = x;
        return 1;}
int Depiler( int *x, Pile *p ) {
        if ( PileVide(*p) ) return 0;
        *x = p->tab[p->sommet--];
        return 1;}
```

8

Les piles (LIFO)

3.2. Au moyen des listes linéaires chaînées



Un empilement consiste à faire une insertion au début et un dépilement une suppression au début.

Description Algorithmique

```
TYPE   S = STRUCTURE
           Info : Typeqq
           Suiv : POINTEUR(S)
FIN
VAR   P : POINTEUR(S)
Créerpile(P)
           P := NIL
Pilevide(P)
           Pilevide := (P = NIL)
```

9

Les piles (LIFO)

```
Empiler(P, X)
           Allouer Q(S)
           Aff_Adr(Q, P)
           Aff_Val(Q, X)
           P := Q
Dépiler(P, X)
           SI   NON Pilevide(P)
                      X := Valeur(P)
                      Sauv := P
                      P := Suivant(P)
                      Liberer(Sauv)
           SINON
                      "Underflow"
           FSI
```

10

Les piles (LIFO)

En C :

```
struct maillon {
        int val;
        struct maillon *adr;
        };
typedef struct maillon *Pile;
void  CreerPile( Pile *p )
        { *p = 0; }
int  PileVide( Pile  p)
        { return (P==0); }
int  PilePleine( Pile p)
        { return 0; }
```

11

Les piles (LIFO)

```
int   Empiler( int x, Pile *p) {
        struct maillon *q;
        if ( PilePleine(*p) ) return 0;
        q = malloc( sizeof(*q) );
        q->val = x;
        q->adr = *p;
        *p = q;
        return 1;
        }

int   Depiler( int *x, Pile *p ) {
        struct maillon *q;
        if ( PileVide(*p) ) return 0;
        *x = (*p)->val;
        q = *p;
        *p = (*p)->adr;
        free(q);
        return 1;
        }
```

12

4. Exemple d'application

4.1. Manipulation d'expression arithmétique

- Notation infixée: <exp> <opérateur> <exp>
  ex : a + b, a + b * c, (a+b) * c, ~a + b
- Notation préfixée: <opérateur> <exp> <exp>
  ex : + a b, + a *bc, * +ab c, + ~a b
- Notation postfixée: <exp> <exp> <opérateur>
  ex : a b +, a bc* +, ab+ c *, a~ b +

13

4.2. Evaluation d'une expression postfixée

Principe:

SI opérande

l'empiler

SI opérateur

dépiler le nb de paramètres requit et empiler le résultat.

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

14

Les piles (LIFO)

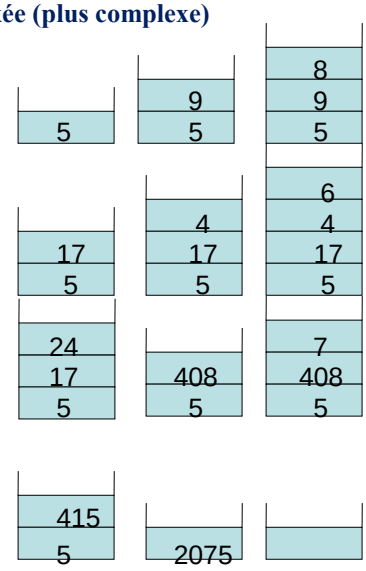
```
EVAL_Postfixe( T:chaine ) : Reel
        i := 1;
        CreerPile(p);
        TQ T[i] <> '#'
                SI  Operande( T[i] )
                        Empiler( p,T[i] )
                SINON                           /*  donc c'est un opérateur */
                        SI Binaire( T[i] )
                                Depiler(p,x);
                                Depiler(p,y);
                                Empiler( p, Calcul( x, T[i], y ) );
                        SINON                   /* donc uniaire */
                                Depiler(p,y);
                                Empiler( p, Calcul( 0, T[i], y ) );
                        FSI
                FSI;
                i := i+1
        FTQ;
        SI Non PileVide(p) :
                Depiler( p, x )
        Sinon
                x:=0
        FSI;
        EVAL_Postfixe := x
```


Les piles (LIFO)

4.3. Evaluation d'une expression infixée (plus complexe)

Soit à évaluer : **5 * (((9+8)*(4*6))+7)**

1 - empiler(5)
2 - empiler(9)
3 - empiler(8)
4 - empiler(dépiler() + dépiler())
5 - empiler(4)
6 - empiler(6)
7 - empiler(dépiler() * dépiler())
8 - empiler(dépiler() * dépiler())
9 - empiler(7)
10 - empiler(dépiler() + dépiler())
11 - empiler(dépiler() * dépiler())
12 - écrire(dépiler())



4.4. Transformation infixée → postfixée

Règles de transformation

- Les opérandes gardent le même ordre dans les deux notations
- Les opérateurs sont empilés, en prenant la précaution de dépiler d'abord tous les opérateurs plus prioritaires
- Les dépilements se font dans la chaîne postfixée
- '(' est empilée sans faire de test
- ')' Dépiler tous les opérateurs dans la chaîne postfixée, jusqu'à trouver une '(', qui sera écartée
- A la fin, tous les opérateurs encore dans la pile seront dépilés dans la chaîne postfixée.

17

a + b * c #	-->	a		avancer ...
a + b * c #	-->	a		empiler +
a + b * c #	-->	a b		avancer...
a + b * c #	-->	a b		empiler *
a + b * c #	-->	a b c		avancer ...
a + b * c #	-->	a b c * +		dépiler tout
				

18

Les piles (LIFO)

```
Trans( Inf : chaine;  var post : chaine )
        i := 1;
        j := 0;
        CreerPile(p);
        TQ Inf [ i ]  <> '#'
            SI  Inf [ i] = '('  :
                    Empiler(p, '(' )   /* 1. cas d'une '(' */
            SINON
                    SI Operateur( Inf [ i ] )   /* 2. cas d'un opérateur */
                            stop := FAUX;
                            TQ  Non stop et Non PileVide(p)
                                    Depiler(p,x);
                                    SI Prio(x) < Prio( Inf[ i] )
                                            Empiler(p,x);
                                            stop := VRAI;
                                    SINON
                                            j++;
                                    post[ j ] := x;
                            FTQ
                            Empiler( p, Inf[ i ] );
```


Les piles (LIFO)

```
                    SINON
                            SI Inf[ i ] = ')'                /* 3. cas d'une ')' */
                                    stop := FAUX;
                                    TQ Non stop et Non PileVide(p)
                                            Depiler(p, x);
                                            SI ( x = '(' )
                                                    stop := VRAI
                                            SINON
                                                    post[ ++j ] := x
                                            FSI
                                    FTQ
                            SINON                /* 4. cas d'un opérande */
                                    post[ ++j ] := Inf[ i ]
                            FSI
                    FSI
            FSI
            i++;
    FTQ
```