

Chapitre II

Les Files et Les Piles

Définitions

Implémentations

Exemples d'application

- Définitions

- File = collection (ensemble) d'éléments gérée en FIFO (First In First Out) - *queue*
- Pile = collection (ensemble) d'éléments gérée en LIFO (Last In First Out) - *stack*

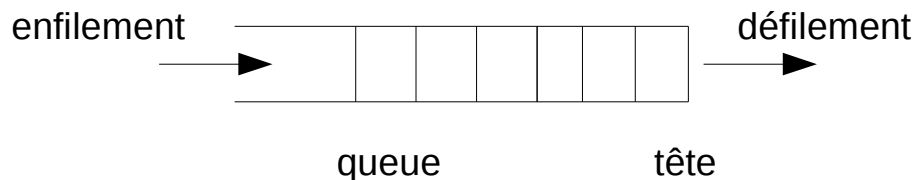
- Utilisations

Files

systèmes d'exploitation

simulations

autres ...

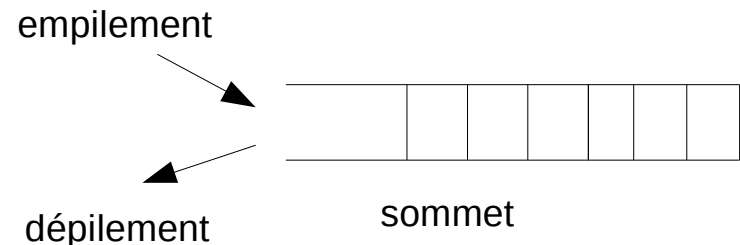


Piles

compilation

gestion des imbrications

autres ...



Exemple de File

Une file contenant 3 elts

queue		tete
CCC	BBB	AAA

après l'enfilement de DDD

queue			tete
DDD	CCC	BBB	AAA

après l'enfilement de EEE

queue				tete
EEE	DDD	CCC	BBB	AAA

après un défilement

queue			tete
EEE	DDD	CCC	BBB

après un 2e défilement

queue		tete
EEE	DDD	CCC

Exemple de Pile

Une pile contenant 3 elts

sommet

CCC	BBB	AAA
-----	-----	-----

après l'empilement de
DDD

sommet

DDD	CCC	BBB	AAA
-----	-----	-----	-----

après l'empilement de
EEE

sommet

EEE	DDD	CCC	BBB	AAA
-----	-----	-----	-----	-----

après un dépilement

sommet

DDD	CCC	BBB	AAA
-----	-----	-----	-----

après un 2e dépilement

sommet

CCC	BBB	AAA
-----	-----	-----

Modèle des Files

CreerFile(var F:File) ==> File

initialise la file F à vide

Enfiler(x:Tqlq; var F:File) File x Elt ==> File

insère x en queue de la file F

Defiler(var x:Tqlq; var F:File) File ==> Elt x File

retire dans x, l'élément en tête de la file F

FileVide(F:File) : Booleen File ==> Bool

FilePleine(F:File) : Booleen File ==> Bool

testent l'état de F (vide ou pleine)

Conditions: Défiler est défini si not FileVide
 Enfiler est défini si not FilePleine

Modèle des Piles

CreerPile(var P:Pile) ==> Pile

initialise la pile P à vide

Empiler(x:Tqlq; var P:Pile) Pile x Elt ==> Pile

insère x au sommet de la pile P

Depiler(var x:Tqlq; var P:Pile) Pile ==> Elt x Pile

retire dans x, l'élément du sommet de la pile P

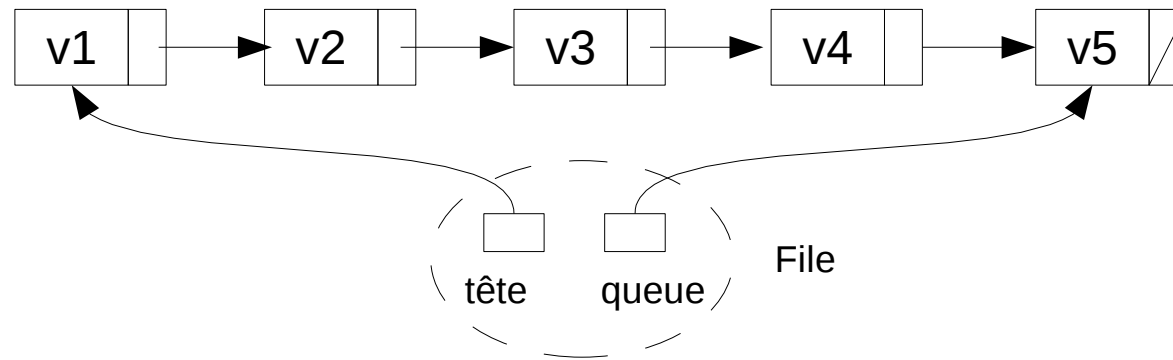
PileVide(P:Pile) : Booleen Pile ==> Bool

PilePleine(P:Pile) : Booleen Pile ==> Bool

testent l'état de P (vide ou pleine)

Conditions: Dépiler est défini si not PileVide
 Empiler est défini si not PilePleine

Implémentation d'une File en dynamique : 1



Une file d'entiers en C

```
struct maillon {  
    int val;  
    struct maillon *adr;  
};  
  
typedef struct {  
    struct maillon *tete, *queue;  
} File;
```

void CreerFile(File *pf)

```
{  
    pf->tete = NIL;  
    pf->queue = NIL;  
}
```

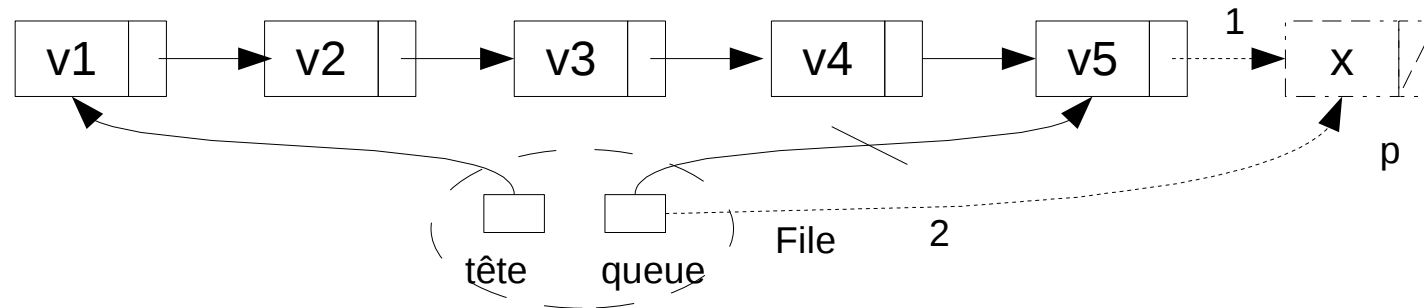
int FileVide(File f)

```
{ return (f.tete == 0); }
```

FilePleine(File f)

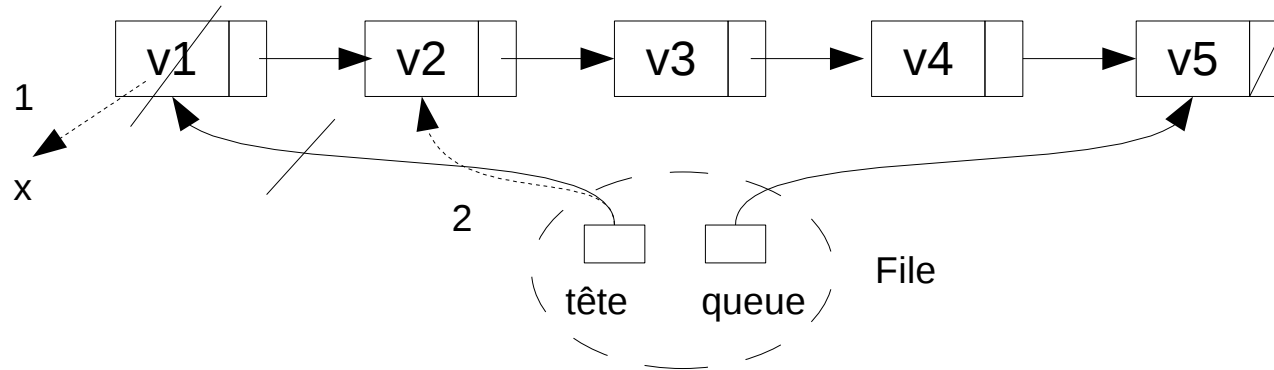
```
{ return 0; } /* jamais pleine */
```

Implémentation d'une File en dynamique : 2



```
int Enfiler( int x, File *pf ) {  
    struct maillon *p;  
    if ( FilePleine(*pf) ) return 0; /* enfilement impossible car FilePleine */  
  
    p = malloc( sizeof(*p) );  
    if ( p == 0 ) return 0;          /* enfilement impossible: malloc a échoué */  
  
    p->val = x; p->adr = 0;  
    if ( pf->queue ) pf->queue->adr = p;    /* 1 */  
    else pf->tete = p;  
    pf->queue = p;                          /* 2 */  
    return 1;  
}
```


Implémentation d'une File en dynamique : 3



```
int Defiler( int *x; File *pf ) {  
    struct maillon *p;  
    if ( FileVide(*pf) ) return 0;  
  
    *x = pf->tete->val;          /* 1 */  
    p = pf->tete;  
    pf->tete = pf->tete->adr; /* 2 */  
    free( p ); p = 0;  
    if ( pf->tete == 0 ) pf->queue = 0;  
    return 1;  
}
```

Implémentations d'une File en statique

- Représentation par tableaux
 - le nombre maximum d'éléments est fixé une fois pour toute
- 3 approches :
 - par décalage
 - pas efficace
 - par flots
 - pas pratique, juste pour introduire la prochaine
 - par tableaux circulaire
 - représentation standard (pratique et efficace)

File en statique : « Par décalages »

```
#define N 100
typedef struct {
    int[N] tab;
    int queue;
} File;
```

tête = l'indice 0 (fixe)

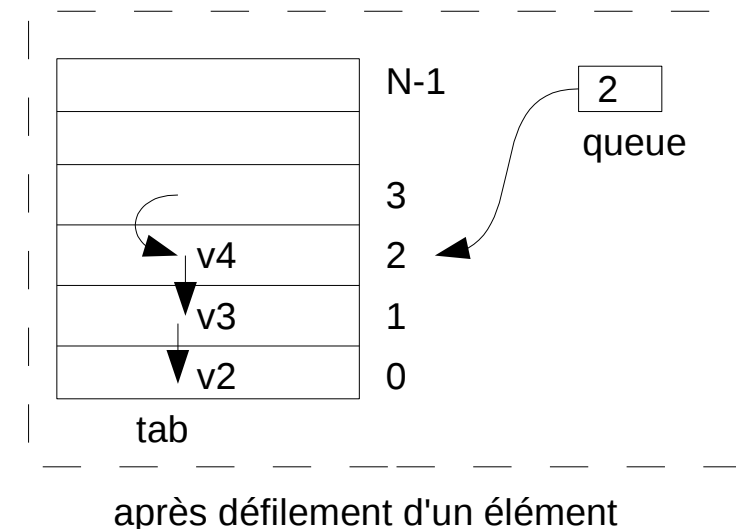
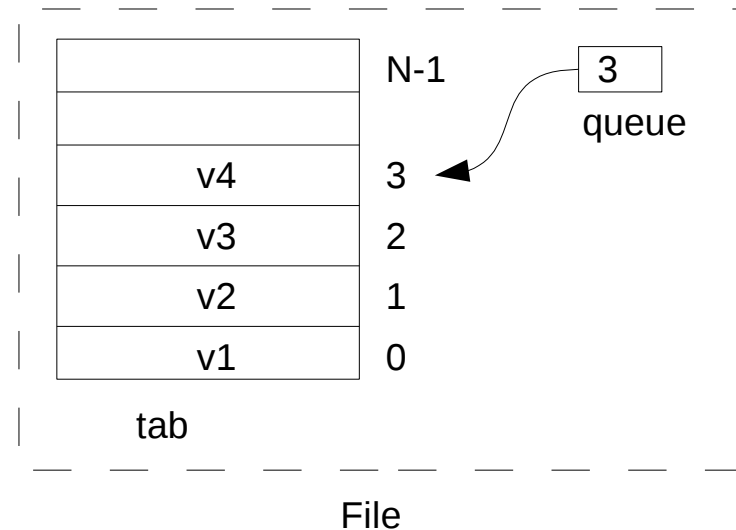
queue = l'indice du dernier elts

défiler = décalages vers le bas
(queue--)

enfiler : rapide (queue++)

FileVide : (queue == -1)

FilePleine : (queue == N-1)



File en statique : « Par flots »

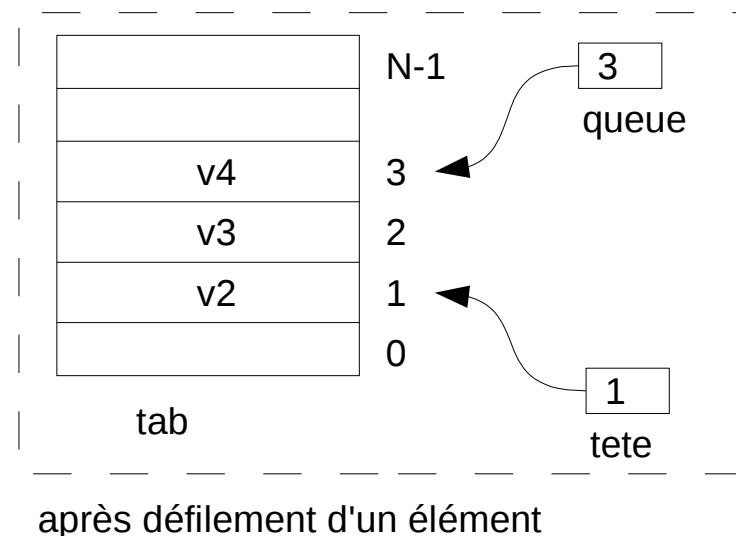
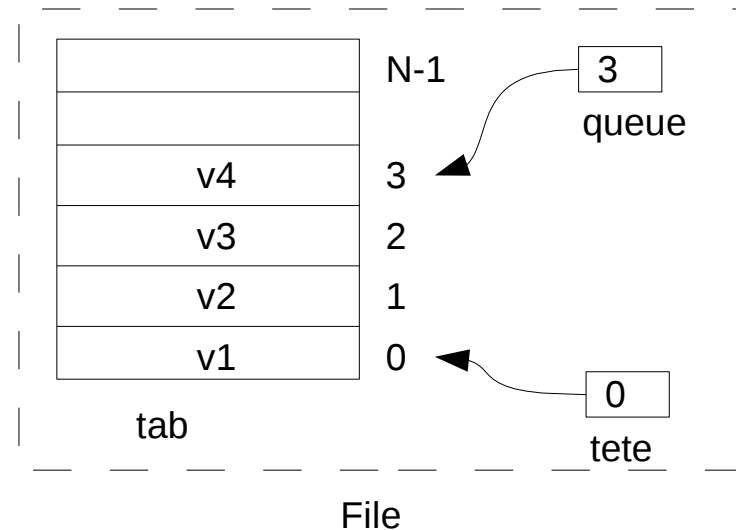
```
#define N 100
typedef struct {
    int[N] tab;
    int tete, queue;
} File;
```

tête = s'incrémente à chaque
défilement

queue = s'incrémente à chaque
enfilement

+ pas de décalages

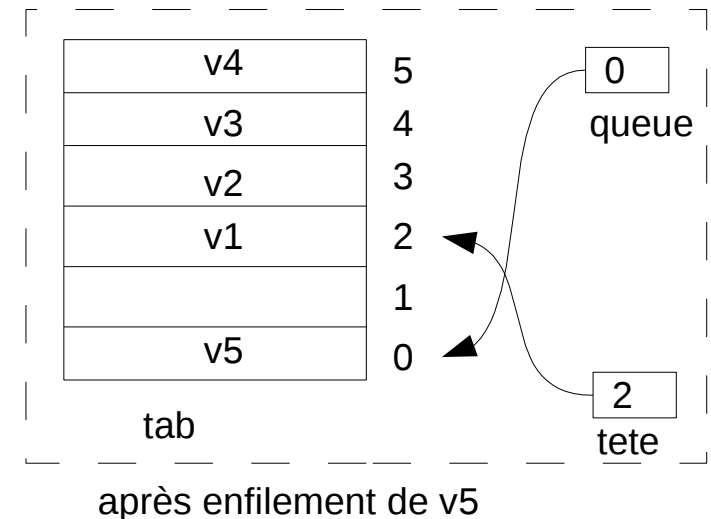
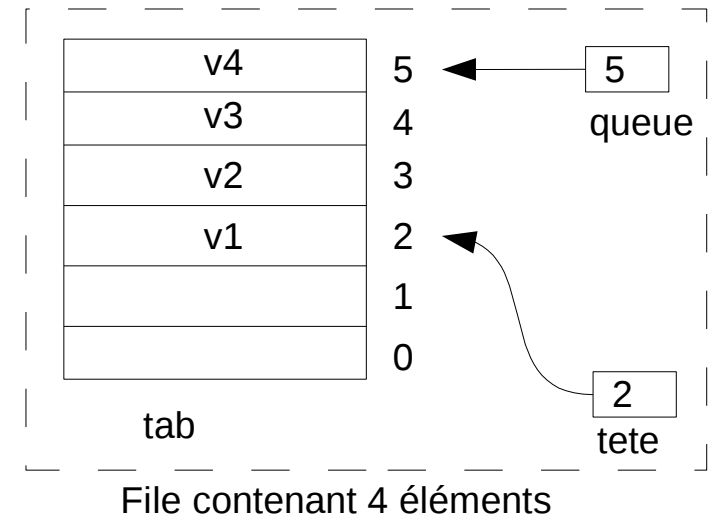
- l'espace libéré est perdu



File en statique : « Par tableau circulaire »

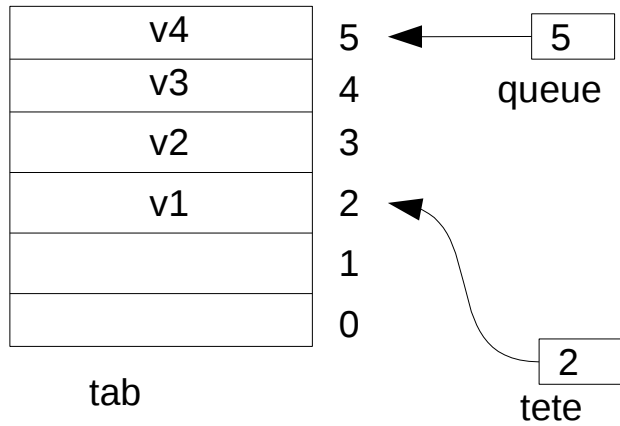
- même principe que l'approche par flots (même déclaration)
- les incrémentations se font modulo N
=> réutilisation des cases libérées.
- efficace : pas de boucle
- réutilise l'espace perdu par les défilements

```
# define N 100
typedef struct {
    int[N] tab;
    int tete, queue;
} File;
```

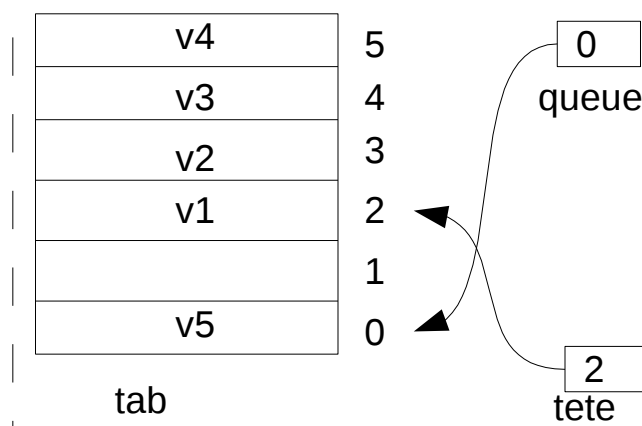


Problème : FilePleine vs FileVide

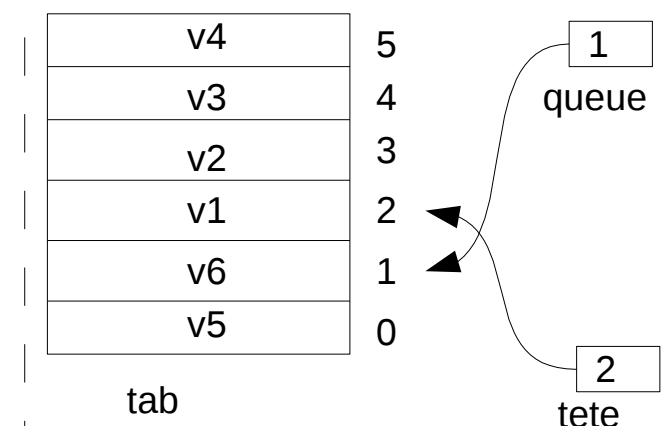
($tete = (queue + 1 \bmod N)$ dans les deux cas)



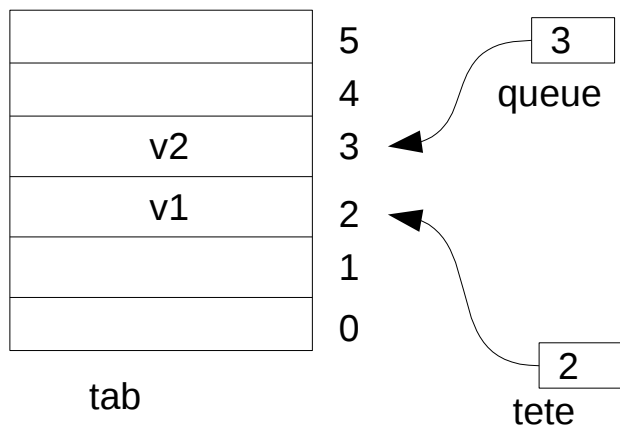
File contenant 4 éléments



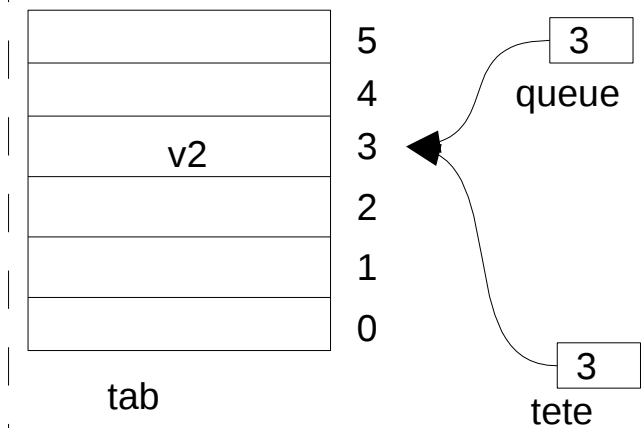
après enfilement de v5



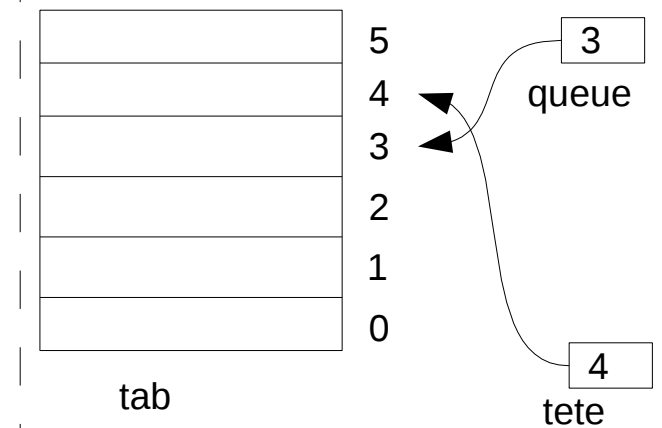
après enfilement de v6 : **FilePleine**



File contenant 2 éléments



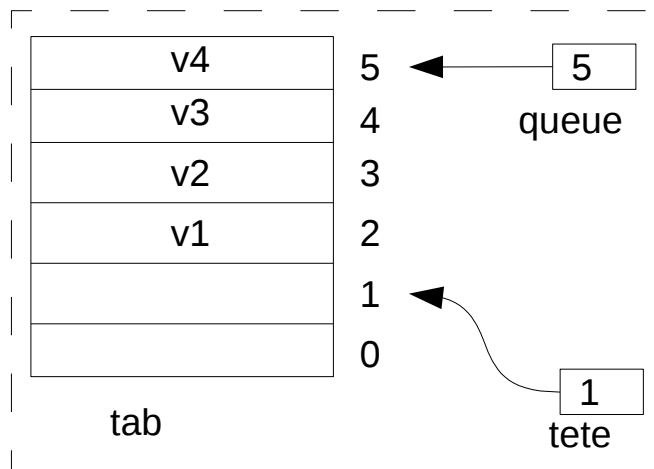
après défilement de v1



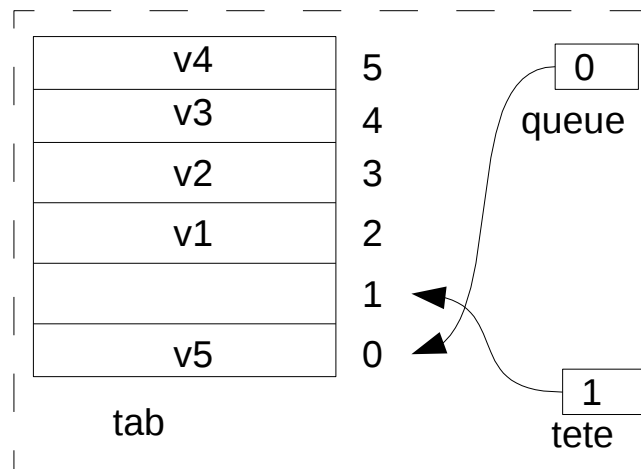
après défilement de v2 : **FileVide**

Solution : sacrifier un élément

par convention, l'élément d'indice *tete* sera sacrifié
le 1er elt se trouve alors à l'indice $(tete+1 \bmod N)$

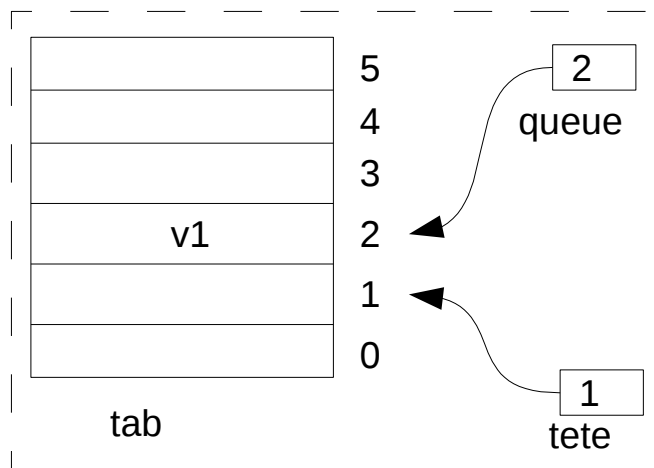


File contenant 4 éléments

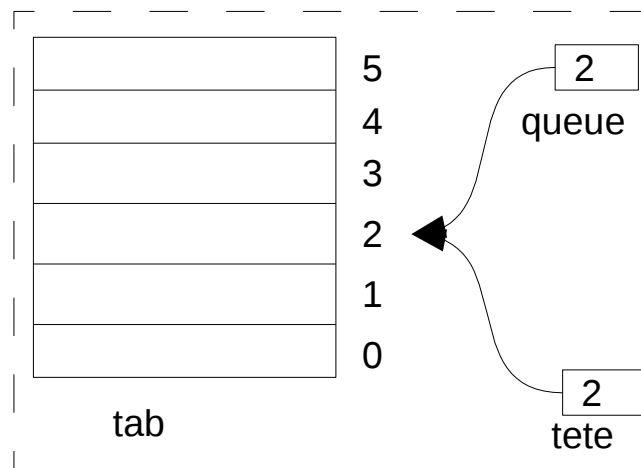


après enfilement de v5 : **FilePleine**

FilePleine ssi :
 $tete = (queue+1 \bmod N)$



File contenant 1 élément



après défilement de v1 : **FileVide**

FileVide ssi :
 $tete = queue$

File en statique : Programmes C

```
void CreerFile ( File *pf ) {  
    pf->tete = N-1;  
    pf->queue = N-1;  
}
```

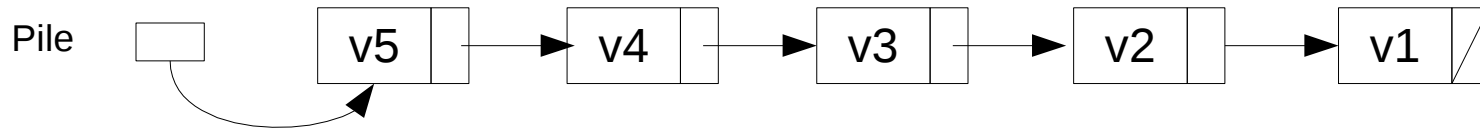
```
int FileVide( File F )  
{  
    return (F.tete == F.queue);  
}
```

```
int FilePleine( File F )  
{  
    return ( F.tete == (F.queue+1 % N) );  
}
```

```
int Enfiler( int x, File *pf ) {  
    if ( FilePleine(*pf) ) return 0;  
  
    pf->queue = (pf->queue + 1) % N;  
    pf->tab[pf->queue] = x;  
    return 1;  
}
```

```
int Defiler( int *x, File *pf ) {  
    if ( FileVide( *pf ) ) return 0;  
  
    pf->tete = (pf->tete+1) % N;  
    *x = pf->.tab[pf->tete];  
    return 1;  
}
```

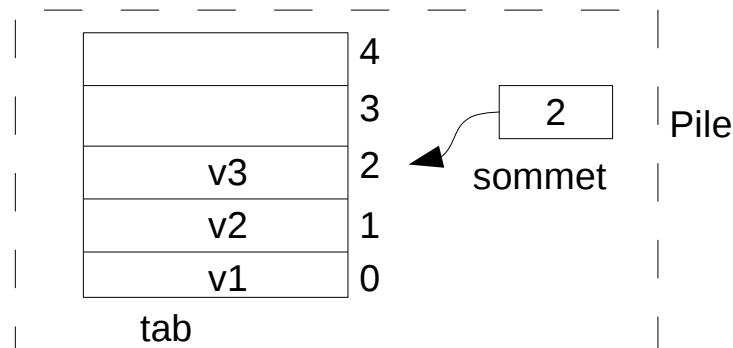

Implémentation d'une Pile en dynamique



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

```
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;  
}
```

Implémentation d'une Pile en statique



```
typedef struct {  
    int[N] tab;  
    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;  
}
```

Exemple d'application

manipulation d'expression arithmétique

Notation infixée: $\langle \text{exp} \rangle \ \langle \text{opérateur} \rangle \ \langle \text{exp} \rangle$

ex : $a + b$, $a + b * c$, $(a + b) * c$, $\sim a + b$

Notation préfixée: $\langle \text{opérateur} \rangle \ \langle \text{exp} \rangle \ \langle \text{exp} \rangle$

ex : $+ a b$, $+ a * b c$, $* + a b c$, $+ \sim a b$

Notation postfixée: $\langle \text{exp} \rangle \ \langle \text{exp} \rangle \ \langle \text{opérateur} \rangle$

ex : $a b +$, $a b c^* +$, $a b + c^*$, $a \sim b +$

Evaluation d'une exp postfixée

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

Principe:

SI opérande
l'empiler

SI Opérateur
depiler le nb de
param requis
et empiler le
résultat.

A la fin :

Le résultat final
se trouve au
sommet de pile

Evaluation d'une expression infixée (plus complexe)

$$5 * (((9+8)*(4*6))+7)$$

			8
		9	9
	5	5	5
1 - empiler(5)			
2 - empiler(9)			6
3 - empiler(8)		4	4
4 - empiler(dépiler() + dépiler())	17	17	17
5 - empiler(4)	5	5	5
6 - empiler(6)			
7 - empiler(dépiler() * dépiler())	24		7
8 - empiler(dépiler() * dépiler())	17	408	408
9 - empiler(7)	5	5	5
10 - empiler(dépiler() + dépiler())			
11 - empiler(dépiler() * dépiler())	415		
12 - écrire(dépiler())	5	2075	

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
- ')' Depiler 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.

Transformation infixée --> postfixée

- exemple : construction elt / elt -

↓
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

*
+

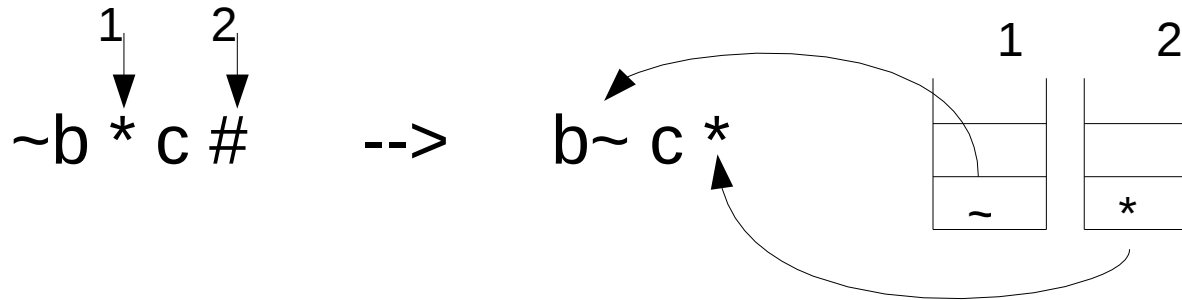
avancer ...

↓
a + b * c # --> a b c * +

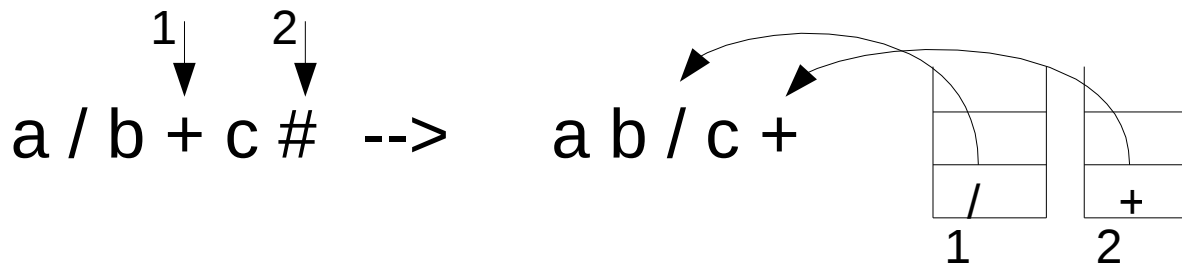
dépiler tout

Transformation infixée --> postfixée

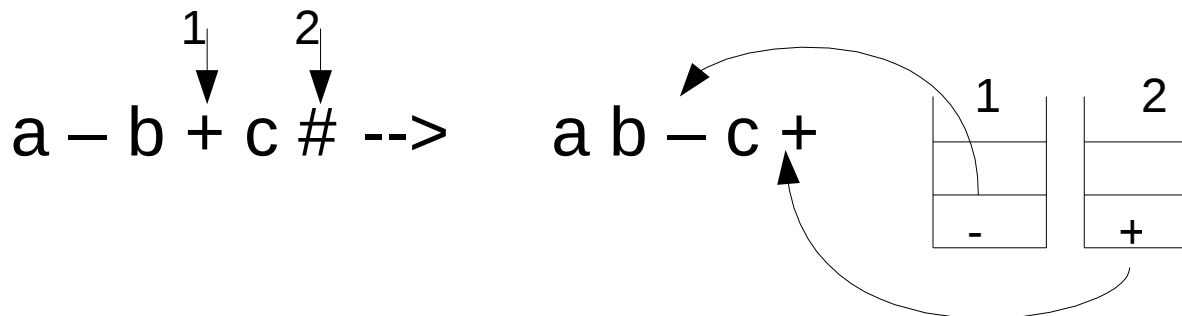
- exemples -



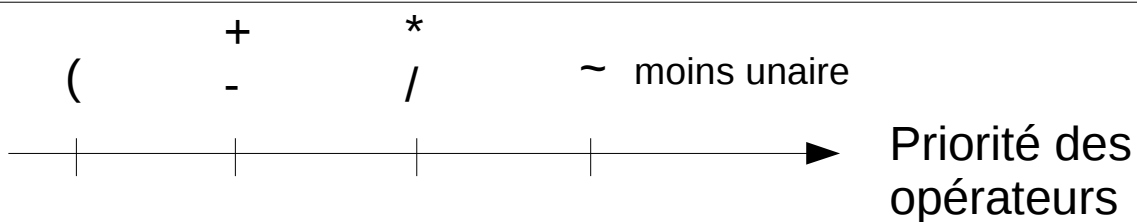
* ne s'empile pas sur ~
prio(*) < prio(~)



+ ne s'empile pas sur /
prio(+) < prio(/)



+ ne s'empile pas sur -
prio(+) = prio(-)




```

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 ] );
    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

```