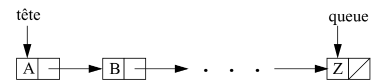
Application de listes dans l'implémentation des types abstraits de-données.

- piles
- files
- arbres binaires
- arbres
- graphes

Files.

Une file est une séquence ordonnée d'éléments dans laquelle tous les ajouts se font à une extrémité (queue de file) et toutes les suppressions se font à l'autre extrémité (tête de file).

Le premier arrivé est le premier servi (ang. FIFO - first in first out).



TAD file.

INITIALISER_FILE (file F);
 // initialise F à une file vide
ENFILER (file F, typeDItem c);
 // insère l'élément avec la valeur c à la fin de la file F
typeDItem DÉFILER (file F);
 // supprime le premier élément de la file F
typeDItem TÊTE (file F);
 // retourne l'élément en tête de la file F
bool FILE_VIDE (file F);
 // retourne vrai si F est vide et faux sinon

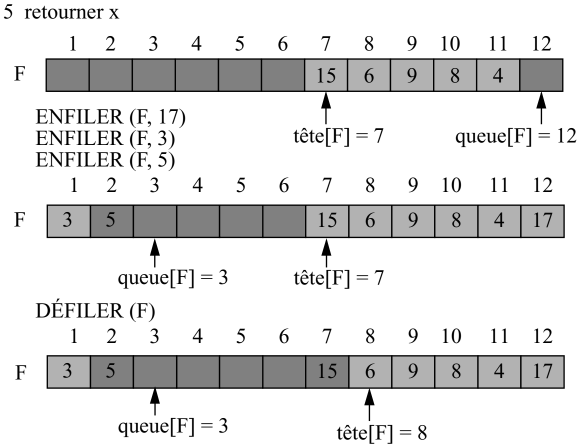
typedef char typeDItem;


```
class file {
public:
   file F;
   INITIALISER_FILE (file F);
        // initialise F à une file vide
   void ENFILER (file F, typeDItem X);
        // insère X à la fin de la file F
   typeDItem DÉFILER (file F);
        // supprime le premier élément de la file F
   typeDItem TÊTE (file F);
        // retourne l élément en tête de la file F
   bool FILE_VIDE (file F);
        // retourne vrai si F est vide et faux sinon
private:
   ...
};
```


Implémentation de file par tableau circulaire.

ENFILER (F, x)
F[queue[F]] ← x
si queue[F] = logueur[F]
 alors queue[F] ← 1
 sinon queue[F] ← queue[F] + 1

DÉFILER (F)
x ← F[tête[F]]
si tête[F] = logueur[F]
 alors tête[F] ← 1
 sinon tête[F] ← tête[F] + 1
retourner x



```
#define MAXFILE 100

typedef char typeDItem;
typeDItem file [ MAXFILE+1 ], tête, queue;

ENFILER ( typeDItem c)
{
   if (queue+1 == tête || tête==0 && fin==MAXFILE) {
      error (“File pleine”);
      return;
   }
   file [queue++] = c;
   if queue > MAXFILE
      queue = 0;
}


typeDItem DÉFILER ( )
{
   if FILE_VIDE ( ) {
      error (“File vide”);
      return
   }
   typeDItem d = file [ tête++ ];
   if (tête > MAXFILE)
      tête = 0;
   return d;
}
```

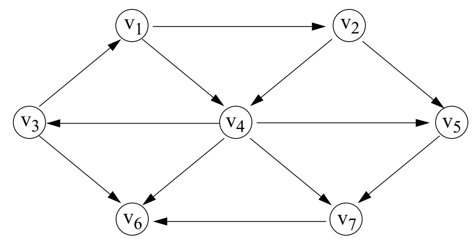


```
INITIALISER_FILE ()
{
   tête = 0;
   queue = 0;
}

bool FILE_VIDE ( )
{
   return tête == queue;
}
```

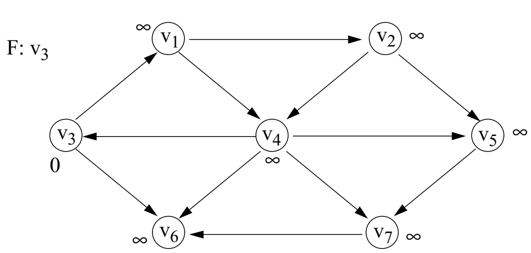
Applications de files.

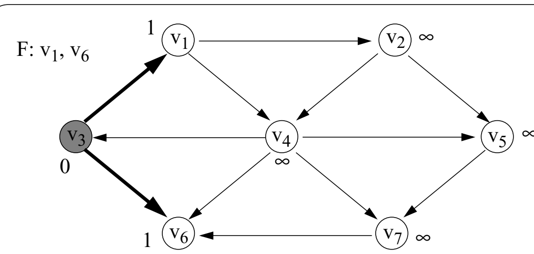
Exemple: Algorithme cherchant le plus court chemin dans un graphe (le cas des arcs sans poids), Weiss p. 288.

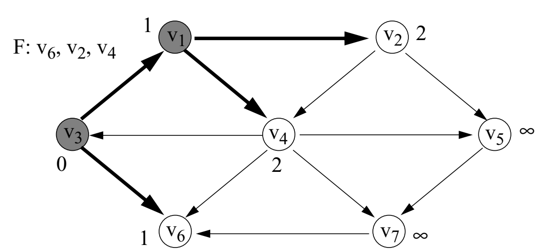


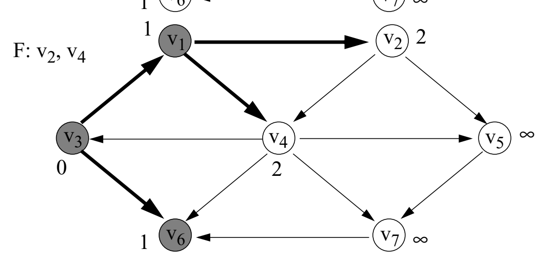

```
PLUS-COURT-SANS-POIDS (G, s)
   INITIALISER (G, s)
   file F ← vide
   ENFILER (F, s)
   tant que F ≠ vide
         u ← DEFILER (F)
         pour chaque sommet v adjacent à u faire
               si distance (v) = ∞ alors
                     distance (v) ← distance (u) + 1
                     PARENT (v) ← u
                     ENFILER (F, v)

INITIALISER (G, s)
   pour chaque sommet s de G
         faire   distance (s) ← ∞
                 PARENT (u) ← NIL
   distance (s) ← 0
```

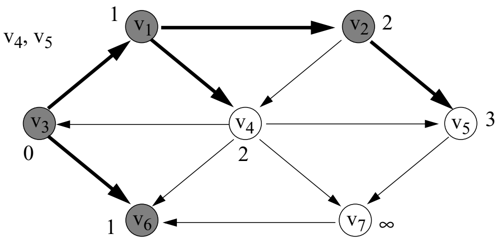




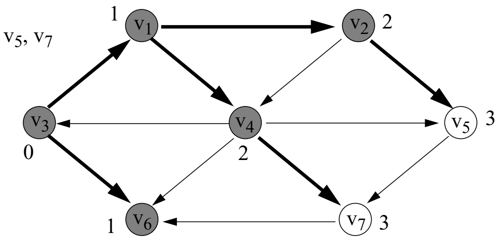




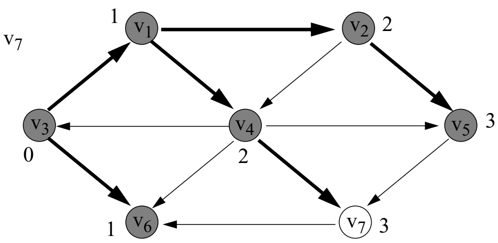
F: v_4, v_5



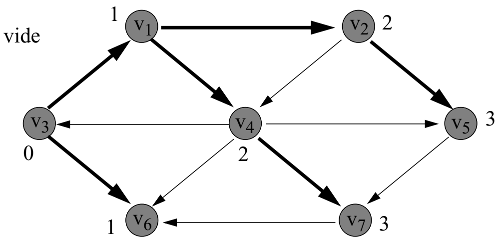
F: v_5, v_7



F: v_7



F: vide



Autres applications de files.

files d'attente (par ex. file de processus dans un système multitâches, queue d'attente pour imprimante, appels téléphoniques en attente)

autres algorithmes sur les graphes (par ex. tri topologique)

simulation (par ex. aérogare)

systèmes de temps réel

tampons d'entrée-sortie

Autres types de files.

la file double (deque, double-ended queue)

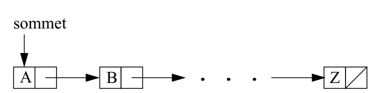
la file d'attente avec les priorités des tâches

la file avec les tâches qui "se découragent"

Piles.

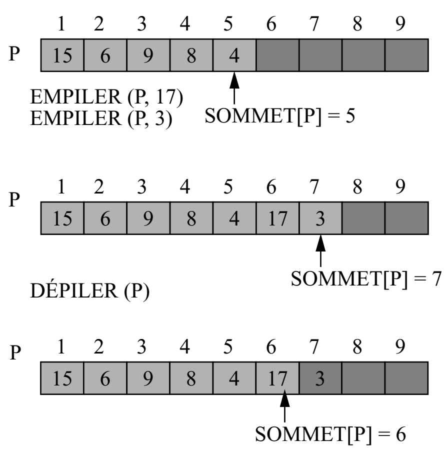
Une pile (ang. stack) est une séquence ordonnée d'éléments dans laquelle tous les ajouts et toutes les suppressions se font à une extrémité (sommet de pile).

Le dernier arrivé est le premier servi (ang. LIFO - last in first out).



TAD pile.

INITIALISER_PILE (pile P);
    // initialise P à une pile vide
EMPILER (pile P, typeDItem c);
    // insère l élément avec la valeur c au sommet de la pile P
typeDItem DÉPILER (pile P);
    // supprime le sommet de la pile P
typeDItem SOMMET (pile P);
    // retourne l'élément au sommet de la pile P
bool PILE_VIDE (pile P);
    // retourne vrai si F est vide et faux sinon




```
PILE_VIDE (P)
si SOMMET[P] = 0
    alors retourner VRAI
    sinon retourner FAUX

SOMMET (pile P);
retourner P[SOMMET[P]]

EMPILER (P, x)              /* ang. “push” */
SOMMET[P] ← SOMMET[P] + 1
P[SOMMET[P]] ← x

DÉPILER (F)                 /* ang. “pop” */
si PILE_VIDE (P)
    alors erreur «débordement négatif»
    sinon
        SOMMET[P] ← SOMMET[P] - 1
        retourner P[SOMMET[P] + 1]
```


Implémentation des piles à l’aide des listes.

```
typedef struct noeud *ptr_noeud;

struct noeud
{
   typeDItem item;
   ptr_noeud suivant;
};

typedef ptr_noeud pile;

bool PILE_VIDE (pile P)
{ return ( P->suivant == NULL ); }

PILE CREER_PILE (void)
{
   pile P;

   P = (pile) malloc (sizeof (struct noeud) );
   if (P == NULL)
      fatal_error (“Il n'y a plus de memoire !!!”);
   return P;
}

void INITIALISER_PILE (pile P)
{
   if (P != NULL)
      P->suivant = NULL;
   else
      error ( “Appeler d’abord creer_pile”);
}
```



```
void EMPILER (typeDItem x, pile P )
{
   ptr_noeud case_temp;
   case_temp = (ptr_noeud) malloc (sizeof (struct noeud) );
   if (case_temp == NULL )
      fatal_error (“Il n'y a plus de memoire !!!”);
   else {
      case_temp->item = x;
      case_temp->suivant = P->suivant;
      P->suivant = case_temp;
   }
}

typeDItem SOMMET (pile P)
{
   if ( pile_vide (P)
      error ("Pile vide");
   else
      return P->suivant->item;
}

void DÉPILER (pile P)
{
   ptr_noeud premiere_case;
   if (pile_vide (P)
      error ("Pile vide");
   else {
      premiere_case = P->suivant;
      P->suivant = P->suivant->suivant;
      free (premiere_case);
   }
}
```

Implémentation de pile à l'aide de tableau (version pleine).

```
struct structure_pile
{
   unsigned int taille_pile;
   int sommet_de_pile;
   typeDItem *tableau_pile;
};

typedef struct structure_pile *PILE;
#define SDP_VIDE (-1) /* signifie la pile vide */

PILE CRÉER_PILE (unsigned int max_elements)
{
   PILE P;
   if (max_elements < MIN_TAILLE_PILE )
      error (“La taille de pile est trop petite”);
   P = (PILE) malloc (sizeof (struct structure_pile) );
   if (P == NULL)
      fatal_error (“Il n'y a plus de memoire !!!”);
   P->tableau_pile = (typeDItem *)
         malloc ( sizeof (typeDItem) * max_elements );
   if ( P->tableau_pile == NULL )
      fatal_error (“Il n'y a plus de memoire !!!”);
   P->sommet_de_pile = SDP_VIDE;
   P->taille_pile = max_elements;
   return ( P ) ;
}

void CRÉER_PILE_VIDE (PILE P )
{ P->sommet_de_pile = SDP_VIDE; }
```



```
void LIBERER_PILE (PILE P)
{
   if ( P != NULL ) {
      free ( P->tableau_pile );
      free ( P );
   }
}

int PILE_VIDE (PILE P)
{ return ( P->sommet_de_pile = SDP_VIDE );}

void EMPILER (typeDItem x, PILE P )
{
   if ( pile_pleine ( P )
      error (“Pile pleine”);
   else
      P->tableau_pile [ ++P->sommet_de_pile ] = x;
}

typeDItem SOMMET (PILE P)
{
   if ( pile_vide (P)
      error ("Pile vide");
   else
      return P->tableau_pile [ P->sommet_de_pile ] ;
}

void DÉPILER (PILE P)
{
   if (pile_vide (P)
      error ("Pile vide");
   else
      P->sommet_de_pile--;
}
```


Application des piles.

L'inversion d'une séquence de caractères.

```
void InverserCars ( )
{
   /* La fonction lit une sequence de caracteres d’entree
    * en les placant sur une pile. Ensuite la fonction depile
    * les caracteres en les affichant a l’ecran. Les caracteres
    * apparaissent en ordre inverse par rapport a la sequence
    * d’entree.
    */

   typeDItem item;
   PILE P;

   while (( item = getchar ()) != '\n')
      EMPILER ( item, P );
   while ( ! pile_vide ( P ) {
      putchar ( sommet ( P ) );
      DÉPILER ( P );
   }
}
```


Vérification de correction d'une structure de parenthèses.

Exemple:
a) a + [b * (c - d)] + e
b) a + [b * (c - d)] + { e * [f - (g + d)] }

a) [( )]
b) [( )] { [( )] }

1. CRÉER_PILE (*P*);
2. **pour** tous les caractères d'entrée **faire**
   2.1. **si** le caractère courant *C* est une parenthèse ouvrante **alors**
         EMPILER (*C*, *P*);
   2.2. **si** le caractère courant C est une parenthèse fermante **alors**
      2.2.1 **si** PILE_VIDE (P)
            alors erreur
            sinon *X* = SOMMET (*P*);
      2.2.2 **si** *X* correspond à *C*
            alors DÉPILER (*P*);
            sinon retourner une erreur;
3. **si** ! PILE_VIDE (*P*)
      alors retourner une erreur;

Structures de parenthèses incorrectes:

a) a + [b * (c - d) }+ e
b) a + [b * (c - d])+ e
c) a + [b * (c - d)] + e } * [f - (g + d)] }
d) a + [b * (c - d)] + { e * [f - (g + d)]

a) [( ) }   b) [( ])
c) [( )] } [( )] }   d) [( )] { [( )]

Vérification syntaxique d'une expression aritmétique.

1. Vérifier la structure de parenthèses.
2. Verifier les couples des items consécutifs:

suiv / prec	(	)	argument	opérateur
(	OK	Erreur	OK	Erreur
)	Erreur	OK	Erreur	OK
argument	Erreur	OK	Erreur	OK
opérateur	OK	Erreur	OK	Erreur

La notation postfixée (polonaise inversée).

Exemples:

Expression infixée	Expression postfixée
1 * 2 + 3	1 2 * 3 +
1 + 2 * 3	1 2 3 * +
(1 + 2) * 3	1 2 + 3 *
1 + 2 * (3 + 4) + 5 * 6	1 2 3 4 + * + 5 6 * +
1 - (2 * (3 + 4)) - (5 - 6) * 7	1 2 3 4 + * - 5 6 - 7 * -
a = (1 - 2) * 3 == 4 / 5 * 6	a 1 2 - 3 * 4 5 / 6 * == =

Évaluation d'une expression en forme postfixée.

ÉVALUATION_POSTFIXÉE (EXPR_POSTFIXÉE EP);
1. **pour** chaque item IT de l'expression postfixée EP
2. **si** IT est un opérande
3. **alors** EMPILER (IT, P)
4. **sinon** /* IT est un opérateur */
5. Arg2 ← SOMMET (P)
6. DÉPILER(P)
7. Arg1 ← SOMMET (P)
8. DÉPILER(P)
9. **cas** IT
10. '+': EMPILER (Arg1 + Arg2, P)
11. '*': EMPILER (Arg1 * Arg2, P)

Opérateurs unaires.

Autres opérateurs (par ex. ternaires).

Conversion de l'expression en notation infixée à la forme postfixée.

CONVERSION_EN_POSTFIXÉE (EXPR_INFIXÉE EI);
1. **pour** chaque item IT de l'expression infixée EI **faire**

cas d'item d'entrée IT	action
'('	EMPILER (IT, P);
opérande	ÉCRIRE (IT)
opérateur	**tantque non** PILE_VIDE (P) **et** SOMMET (P) ≠ '(' **et** (priorité (IT) <= priorité (SOMMET (P)) **faire** ÉCRIRE (SOMMET (P)); DÉPILER (P); EMPILER (IT, P);
')'	**tantque** SOMMET (P) ≠ '(' **faire** ÉCRIRE (SOMMET (P)) DÉPILER (P) DÉPILER (P) /* '(' */
aucun	**tantque** (**non** PILE_VIDE (P)) **faire** ÉCRIRE (SOMMET (P)); DÉPILER (P);

Autre opérateurs (le cas du langage C)

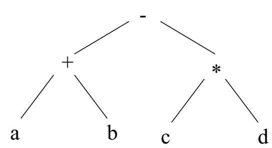
<table>
<tr><th>Priorité</th><th>Opérateurs</th><th>Associativité</th></tr>
<tr><td>15</td><td>( ) [ ] . -></td><td>gauche à droite</td></tr>
<tr><td>14</td><td>! ~ - ++ -- & * (type) sizeof</td><td>droite à gauche</td></tr>
<tr><td>13</td><td>* / %</td><td>gauche à droite</td></tr>
<tr><td>12</td><td>+ -</td><td>gauche à droite</td></tr>
<tr><td>11</td><td><< >></td><td>gauche à droite</td></tr>
<tr><td>10</td><td>< <= > >=</td><td>gauche à droite</td></tr>
<tr><td>9</td><td>== !=</td><td>gauche à droite</td></tr>
<tr><td>8</td><td>&</td><td>gauche à droite</td></tr>
<tr><td>7</td><td>^</td><td>gauche à droite</td></tr>
<tr><td>6</td><td>|</td><td>gauche à droite</td></tr>
<tr><td>5</td><td>&&</td><td>gauche à droite</td></tr>
<tr><td>4</td><td>||</td><td>gauche à droite</td></tr>
<tr><td>3</td><td>?:</td><td>droite à gauche</td></tr>
<tr><td>2</td><td>= *= /= %= += -= <<= >>= &= |= ^=</td><td>droite à gauche</td></tr>
<tr><td>1</td><td>,</td><td>gauche à droite</td></tr>
</table>

Adaptation de l'algorithme de la conversion à la notation postfixée pour les autres types d'opérateurs

Arbres des expressions

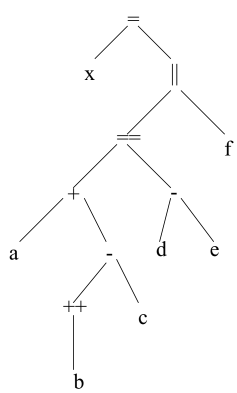
Exemple 1

a + b - c * d



Exemple 2

x = a + (++ b - c) == d - e || f



Exemple 3

* a [b] [c] . d -> e == f ? g : h