```
                                placePrécédente ← place
                                place ← lEntier[place].suc
                        fsi(*4*)
                ftantque(*3*)
                si valeur = élément alors(*5a*)
                        (*valeur appartient déjà à la liste*)
                        si place = placePrécédente alors(*6a*)
                                (*c'est le 1er élément*)
                                lEntier[0].suc ← lEntier[place].suc
                        sinon(*6s*)
                                lEntier[placePrécédente].suc ← lEntier[place].suc
                                (* en appliquant la traduction systématique, on écrirait :
                                placePrécédente ← 0
                                tantque(*20*) lEntier[placePrécédente].suc ≠ place faire
                                        placePrécédente ← lEntier[placePrécédente].suc
                                ftantque(*20*)
                                lEntier[placePrécédente].suc ← lEntier[place].suc
                                *)
                        fsi(*6*)
                sinon(*5s*) (*valeur n'appartient pas à la liste*)
                        placeLibre ← lEntier[-1].suc
                        lEntier[-1].suc ← lEntier[-1].suc+1
                        si place = placePrécédente alors(*7a*)
                                (*valeur < à toutes les valeurs*)
                                lEntier[placeLibre].val ← valeur
                                lEntier[placeLibre].suc ← lEntier[0].suc
                                lEntier[0].suc ← placeLibre
                        sinon(*7s*)
                                lEntier[placeLibre].val ← valeur
                                lEntier[placeLibre].suc ← lEntier[placePrécédente].suc
                                (*ou lEntier[placeLibre].suc ← place*)
                                lEntier[placePrécédente].suc ← placeLibre
                        fsi(*7*)
                fsi(*5*)
        fsi(*8*)
    fpour(*2*)
    retourne lEntier
    fin(*1*)
Lexique
    LEntierChaînéeTableau = tableau < val : entier, suc : entier > [-1...bs]
    lEntier : LEntierChaînéeTableau, tableau des valeurs et des successeurs
    nbValeur : entier, nombre de valeurs à traiter
    i : entier, indice d'itération
    bs : entier, nombre maximum d'éléments de la liste lEntier
    valeur : entier, suite des données
    place : entier, place courante dans lEntier
    placePrécédente : entier, place précédant place dans lEntier
    fini : booléen, vrai lorsqu'on a trouvé l'entier ou lorsque la valeur courante est supérieure à la valeur cherchée
    élément : entier, valeur courante
    placeLibre : entier, place libre sélectionnée pour la valeur courante
```

4 Les piles et les files

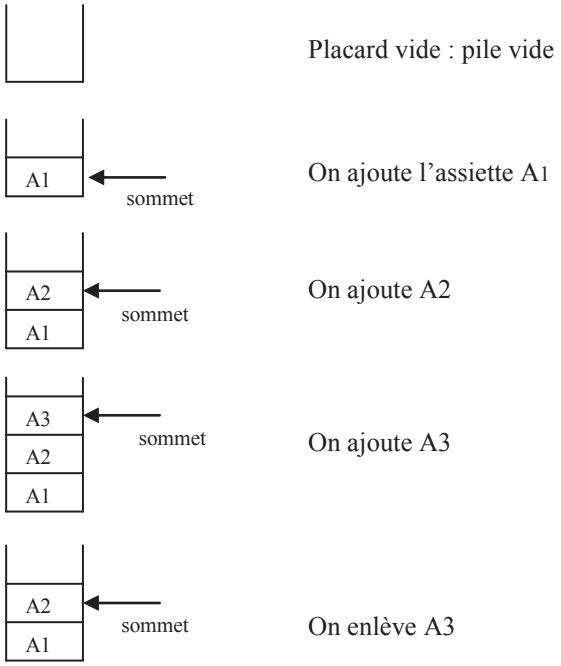
Pour beaucoup d'applications, les seules opérations à effectuer sur les listes sont des insertions et des suppressions aux extrémités.

4.1 Les piles

Une **pile** est une liste dans laquelle toutes les adjonctions et toutes les suppressions se font à une seule extrémité appelée *sommet*. Ainsi, le seul élément qu'on puisse supprimer est le plus récemment entré. Une pile a une structure « LIFO » pour « Last In, First Out » c'est-à-dire « dernier entré premier sorti ». Une bonne image pour se représenter une pile est une pile d'assiettes : c'est en haut de la pile qu'il faut prendre ou mettre une assiette !

Exemple de la pile d'assiettes :

On ne pose d'assiettes qu'au "sommet" de la pile. On n'en enlève également qu'au sommet.



On enlève une assiette : c'est A2.
Etc...

Les opérations sur une pile sont :

- tester si une pile est vide (*estVidePile*) ;
- accéder au sommet (*sommet*);
- empiler un élément (*empiler*);
- retirer l'élément qui se trouve au sommet (*dépiler*),
- créer une pile vide (*pileVide*).

Définition abstraite du type pile :

Soit *Valeur* un ensemble de valeurs (par exemple des entiers). On appelle type **pile de Valeur** et on note **Pile(Valeur)** l'ensemble des piles dont les valeurs sont des éléments de *Valeur*.

Ensembles définis et utilisés : *Pile*, Valeur,booléen

Description fonctionnelle des opérations :

-	pileVide :		→	Pile (Valeur)
-	sommet :	Pile (Valeur)-{pileVide( )}	→	Valeur
-	estVidePile :	Pile (Valeur)	→	booléen
-	empiler :	Pile (Valeur) x Valeur	→	
-	dépiler :	Pile (Valeur)-{pileVide( )}	→	

Les opérations empiler et dépiler modifient la pile donnée en paramètre.

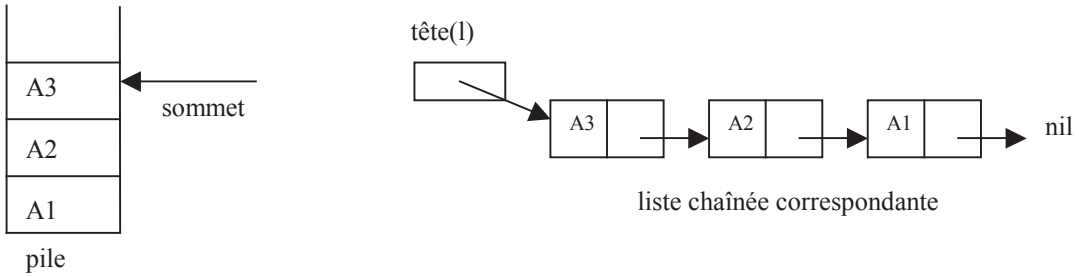
Utilité :

Les piles sont des structures fondamentales, et leur emploi dans les programmes informatiques est très fréquent. Le mécanisme d'appel des sous-programmes suit ce modèle de pile. Les logiciels qui proposent une fonction « undo » se servent également d'une pile pour défaire, en ordre inverse, les dernières actions effectuées par l'utilisateur.

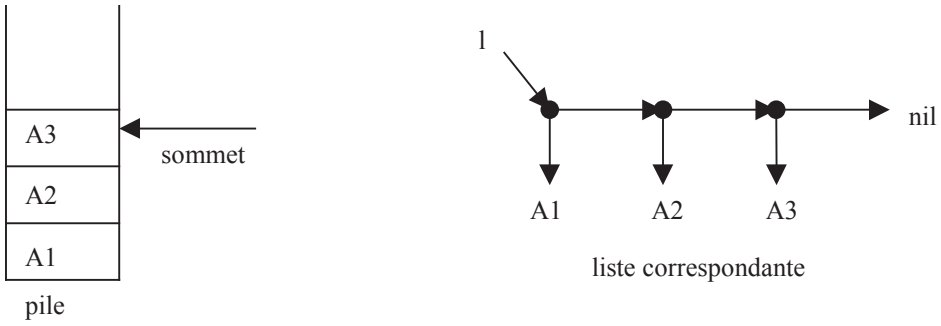
Représentations :

On peut utiliser pour implémenter les piles toutes les représentations étudiées pour les listes. Les opérations empiler et dépiler travailleront soit sur la tête de liste (dans le cas des représentations chaînées), soit sur la queue de liste (dans le cas des représentations contiguës) pour limiter la complexité.

Exemple :
Considérons le cas de la pile d'assiettes contenant 3 éléments. Dans le cas où on choisit une représentation chaînée, cette pile correspond à une liste dans laquelle les adjonctions et suppressions se font uniquement en tête :



Dans le cas où on choisit une représentation contiguë, la pile correspond à une liste dans laquelle les adjonctions et suppressions se font uniquement en queue :



Exercice :

On désire évaluer des expressions postfixées formées d'opérandes entiers positifs et des 2 opérateurs «+» et «-». On rappelle que, dans la notation postfixée, l'opérateur suit ses opérandes. Par exemple, l'expression infixée suivante :

(7 + 12) + (5 – 3) s'écrit 7 12 + 5 3 - +

L'évaluation d'une expression postfixée se fait simplement à l'aide d'une pile. L'expression est lue de gauche à droite. Chaque opérande lue est empilé et chaque opérateur trouve ses 2 opérandes en sommet de pile qu'il remplace par le résultat de son opération. Lorsque l'expression est entièrement lue, sans erreur, la pile ne contient qu'une seule valeur, le résultat de l'évaluation.

Question 1 : Ecrire l'algorithme logique de la fonction d'évaluation d'une expression postfixée supposée syntaxiquement correcte. On suppose disposer d'une fonction *chaineEntierConvertir* qui convertit une chaîne de caractères représentant un entier en cet entier. Les opérateurs et les opérandes sont séparés par des blancs et l'expression est suivie d'un « . ».

Question 2 : On choisit de représenter la pile de manière contiguë par un couple : tableau et indice du sommet, à l'aide du type suivant :

PileEntier = <tab : tableau entier [1..MAXTAB], sommet : entier>

On suppose que la taille du tableau est suffisante et donnée par la constante MAXTAB. Ecrire les algorithmes de programmation des opérations sur les piles.

Corrigé :
Question 1 :

```
fonction ExpPostfEvaluer ( ) : entier
    début (*1*)
    p ← pileVide( )
    chLue ← lire( )
    tant que chLue ≠ « . » faire (*2*)
        si chLue = « + » alors (*3a*)
            x ← opérandeRécupérer (p)
            y ← opérandeRécupérer (p)
            empiler (p, x + y)
        sinon (*3s*)
            si chLue = « - » alors (*4a*)
                x ← opérandeRécupérer (p)
                y ← opérandeRécupérer (p)
                empiler (p, y - x)
            sinon (*4*)
                opérande ← chaineEntierConvertir (chLue)
                empiler (p, opérande)
            fsi (*4*)
        fsi (*3*)
        chLue ← lire( )
    ftant (*2*)
    si non estVidePile (p) alors (*7a*)
        valeur ← sommet (p)
    fsi (*7*)
    retourne valeur
    fin (*1*)
Lexique
    p : Pile (entier)
    valeur : entier, valeur de l'expression
    chLue : chaîne, chaîne lue (un opérande, un opérateur ou « . »)
    x : entier, un opérande de la pile
    y : entier, un opérande de la pile
    opérande : entier, un opérande lue

fonction opérandeRécupérer (p InOut : Pile (entier)) : entier
    début (*1*)
    si non estVidePile (p) alors (*2a*)
        x ← sommet (p)
        dépiler (p)
    fsi (*2*)
    retourne x
    fin (*1*)
Lexique
    p : Pile (entier)
    x : entier, un opérande de la pile
```

Question 2 :

```
fonction sommet (p : PileEntier) : entier
    début
    valeur ← p.tab[p. sommet]
    retourne valeur
    fin

fonction pileVide ( ) : PileEntier
    début
    p.sommet ← 0
    retourne p
    fin

fonction estVidePile (p : PileEntier) : booléen
    début
    test ← (p.sommet = 0)
    retourne test
    fin
```

```
fonction empiler (p InOut : PileEntier, v : entier)
    début
    si p. sommet < MAXTAB alors (* test facultatif : ce cas ne devrait pas arriver*)
          p.sommet ← p.sommet + 1
          p.tab [p.sommet] ← v
    fsi
    fin

fonction dépiler (p InOut : PileEntier)
    début
    p.sommet ← p.sommet - 1
    fin
```

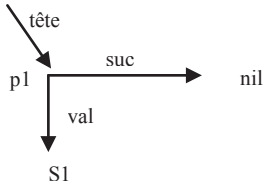
4.2 Les files

Une **file** est une liste dans laquelle toutes les adjonctions se font en queue et toutes les suppressions en tête. Autrement dit, on ne sait ajouter des éléments qu'en queue et le seul élément qu'on puisse supprimer est le plus anciennement entré. Par analogie avec les files d'attente, on dit que l'élément présent depuis le plus longtemps est le premier, on dit aussi qu'il est en tête. Une file a une structure "FIFO" (First In, First Out) c'est-à-dire « premier entré premier sorti ».

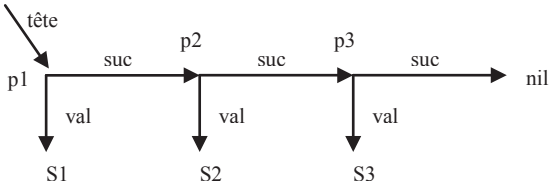
Exemple : file d'attente, par exemple queue (disciplinée) au téléski

Le matin :

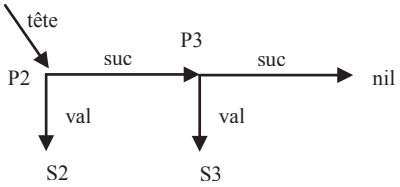
Arrivée du skieur S1



Arrivée des skieurs S2 puis S3



Montée d'un skieur : c'est S1. Il est supprimé de la file.



etc.

Les opérations sur une file sont :

- tester si une file est vide (*estVideFile*) ;
- accéder au premier élément de la file (*premier*);
- ajouter un élément dans la file (*adjfil*);
- retirer le premier élément de la file (*supfil*),
- créer une file vide (*fileVide*).

Définition abstraite du type file :

Soit *Valeur* un ensemble de valeurs (par exemple des entiers). On appelle type **file de Valeur** et on note **File(Valeur)** l'ensemble des files dont les valeurs sont des éléments de *Valeur*.

Ensembles définis et utilisés : *File*, Valeur, booléen

Description fonctionnelle des opérations :

-	fileVide :		→	File (Valeur)
-	premier :	File (Valeur)- {fileVide( )}	→	Valeur
-	estVideFile :	File (Valeur)	→	booléen
-	adjfil :	File (Valeur) x Valeur	→	
-	supfil :	File (Valeur)-{fileVide( )}	→	

Les opérations *adjfil* et *supfil* modifient la file donnée en paramètre. L'opération *adjfil* est parfois appelée *enfiler* et l'opération *supfil défiler*.

Utilité :

Le modèle de file est très utilisé en informatique. On le retrouve dans de nombreuses situations, comme, par exemple, dans la file d'attente d'un gestionnaire d'impression d'un système d'exploitation.

Représentations :

On peut utiliser pour implémenter les files toutes les représentations étudiées pour les listes. Mais pour limiter la complexité, on a intérêt, pour les files, à gérer un indicateur de tête. Une représentation souvent satisfaisante est la représentation contiguë dans un tableau avec tête mobile. Dans ce cas, la file est représentée par un triplet (tableau, tête, nombre d'éléments). Lorsqu'on arrive en fin de tableau et non en fin de file, on continue le parcours en se plaçant au début du tableau. Le nombre d'éléments de la file est bien sûr limité à la taille du tableau.

Exemple :
Soit une file d'attente de skieurs contenant 5 éléments. Elle peut être représentée de la manière suivante :

	1	2	3	4	5	6	7	8	9	10
f.tab :	s4	s5						s1	s2	s3
f.tête :	8									
f.nb :	5									

Exercice :

Ecrire les algorithmes de programmation des fonctions *fileVide*, *premier*, *adjfil*, *supfil* et *estVideFile* dans le cas d'une file d'entiers représentée de manière contiguë à l'aide d'un triplé (tableau, tête, nombre d'éléments) comme proposée dans l'exemple précédent. On suppose que la taille du tableau est suffisante et donnée par la constante MAXTAB.

Corrigé

- fonction **fileVide** ( ) : FileEntier
  début
  f.nb ← 0
  f.tête ← 1 (* pour éviter le cas particulier de l'adjonction dans une file vide *)
  retourne f
  fin
  Lexique
  FileEntier = <tab : tableau entier [1..MAXTAB], tête : entier, nb : entier >
  f : FileEntier

- fonction **premier** (f : FileEntier) : entier
  début
  retourne f.tab [f.tête]
  fin
  Lexique
  f : FileEntier

- fonction **adjfil** (f InOut: FileEntier, v : entier)
  début
  indice ← (f.tête + f.nb – 1) mod MAXTAB + 1
  /*ou : si f.tete + f.nb > MAXTAB alors indice ← f.tete + f.nb - MAXTAB sinon indice ← f.tete + f.nb fsi */
  f.tab [indice] ← v
  f.nb ← f.nb + 1
  fin
  Lexique
  f : FileEntier
  v : entier, valeur à ajouter dans la file
  indice : entier, indice de *v* dans *f.tab*

- fonction **supfil** (f InOut : FileEntier)
  début
  f.tête ← f.tête mod MAXTAB + 1
  /*ou : si f.tete = MAXTAB alors f.tete ← 1 sinon f.tete ← f.tete + 1 fsi */
  f.nb ← f.nb - 1
  fin
  Lexique
  f : FileEntier

- fonction **estVidefile** (f : FileEntier) : booléen
  début
  retourne (f.nb = 0)
  fin
  Lexique
  f : FileEntier

4.3 Les files avec priorité

Les files avec priorité remettent en question le modèle FIFO des files ordinaires. Avec ces files, l'ordre d'arrivée des éléments n'est plus respecté. Les éléments sont munis d'une priorité et ceux qui possèdent les priorités les plus fortes sont traités en premier.

Les opérations sur une file avec priorité sont :

- tester si la file est vide (*estVidefp)* ;
- accéder à l'élément le plus prioritaire de la file (*premierfp*);
- ajouter un élément et sa priorité dans la file (*adjfp*);
- retirer l'élément le plus prioritaire de la file (*supfp*),
- créer une file vide (*fpVide*).

Définition abstraite du type filePriorité :

Soit *Valeur* un ensemble de valeurs (par exemple des entiers), munies d'une priorité prise dans un ensemble notée *Priorité* qui est pourvu d'une relation d'ordre total permettant d'ordonner les éléments du plus prioritaire au moins prioritaire. On appelle type **FilePriorité de Valeur** et on note **FilePriorité(Valeur)** l'ensemble des files avec priorité dont les valeurs sont des éléments de *Valeur*.

Ensembles définis et utilisés : *FilePriorité*, Valeur, Priorité, booléen

Description fonctionnelle des opérations :

-	fpVide :		→	FilePriorité (Valeur)
-	premierfp :	FilePriorité (Valeur) -{fpVide( )}	→	Valeur
-	estVidefp :	FilePriorité (Valeur)	→	booléen
-	adjfp :	FilePriorité (Valeur) x Valeur x Priorité	→	
-	supfp :	FilePriorité (Valeur)-{fpVide( )}	→	

Les opérations *adjfp* et *supfp* modifient la file avec priorité donnée en paramètre

Utilité :

Les systèmes d'exploitation utilisent fréquemment les files avec priorité, par exemple, pour gérer l'accès des travaux d'impression à une imprimante, ou encore l'accès des processus au processeur.

Représentations :

On peut utiliser pour implémenter les files avec priorité toutes les représentations étudiées pour les listes. Si on choisit une liste non ordonnée, l'adjonction peut se faire en tête de liste. Les opérations *premierfp* et *supfp* nécessitent alors une recherche linéaire de l'élément le plus prioritaire. Cette recherche peut demander un parcours complet de la liste. Si on choisit une liste ordonnée, on peut placer les éléments par ordre de priorité décroissante. C'est alors l'opération d'adjonction qui peut nécessiter un parcours complet de la liste.

Exercice :

On s'intéresse à la gestion des travaux en attente d'impression. Ils sont munis d'une priorité dépendant des utilisateurs. Cette priorité est exprimée sous forme d'un entier de 1 pour les plus prioritaires à 5 pour les moins prioritaires. Les travaux sont placés dans une file avec priorité. On choisit une représentation contiguë dans un tableau avec tête mobile sans ordonner les éléments. Un élément de la file est un couple (nom du fichier à imprimer, sa priorité).

Exemple :

	1	2	3	4	5	6	7	8	9	10	
f.tab :	LS_pho	FP_fic	AN_fac					LS_let	CD_ess	AN_pai	nom
	2	5	1					2	4	1	priorité

f.tête : 8

f.nb : 6

Question 1 : Ecrire l'algorithme de programmation de la fonction *premierfp*. On suppose que la taille du tableau est suffisante et donnée par la constante MAXTAB.

```
Corrigé
fonction premierfp (f : FilePrioritéTravaux) : chaîne
début (*1*)
    maxPr ← f.tab [f.tête].priorité
    nomFichPr ←f.tab [f.tête].nom
    i ← f.tête mod  MAXTAB + 1
    (* ou : si f.tete = MAXTAB alors i ← 1 sinon i ← f.tete + 1 fsi *)
    nbParcouru ← 1
    arrêt ← maxPr = 1
    tantque(*2*) non arrêt et nbParcouru ≤ f.nb faire
        si f.tab [i].priorité < maxPr alors (*3a*)
                maxPr ← f.tab [i].priorité
                nomFichPr ← f.tab[i].nom
        fsi(*3*)
        nbParcouru ← nbParcouru + 1
        i ← i mod  MAXTAB + 1
        (* ou bien :   si i = MAXTAB alors i ← 1 sinon i ← i + 1 fsi  *)
        arrêt ← (maxPr = 1)
    fintantque(*2*)
    nomFich ← nomFichPr
    retourne nomFich
fin (*1*)
Lexique
      FilePrioritéTravaux = < tab : tableau Travail [1..MAXTAB], tête : entier, nb : entier >
      Travail = < nom : chaîne, priorité : entier >
      f : FilePrioritéTravaux
      nomFich : chaîne, nom du fichier à imprimer
      maxPr : entier, la plus grande priorité rencontrée dans la file à un instant du parcours
      nomFichPr : chaîne, nom du fichier ayant cette priorité
      i : entier, indice de parcours dans f.tab
      arrêt : booléen, à vrai dès qu'on rencontre un travail de priorité 1
      nbParcouru : entier, nombre d'éléments parcourus au rang i
```

Question 2 : Même question en supposant que les éléments sont ordonnés par priorité dans la file.

Exemple :

	1	2	3	4	5	6	7	8	9	10	
f.tab :	LS_pho	CD_ess	FP_fic					AN_pai	AN_fac	LS_let	nom
	2	4	5					1	1	2	priorité

f.tête :	8
f.nb :	6

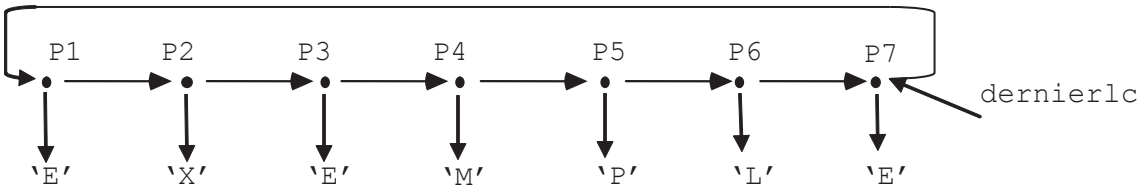
Corrigé

```
fonction premierfp (f : FilePrioritéTravaux) : chaîne
début (*1*)
    nomFich ← f.tab [f.tête].nom
    retourne nomFich
fin (*1*)
Lexique
      f : FilePrioritéTravaux
      nomfich : chaîne, nom du fichier à imprimer
```

5 Les listes circulaires et les listes symétriques

5.1 Les listes circulaires

Une liste circulaire est une liste telle que le dernier élément de la liste a pour successeur le premier. On peut ainsi parcourir toute la liste à partir de n'importe quel élément. Il faut pouvoir identifier la tête de liste. Mais il est plus avantageux de remplacer l'indication sur le premier élément par une indication sur le dernier, ce qui donne facilement accès au dernier et au premier qui est le suivant du dernier.



Définition abstraite :

Soit *Valeur* un ensemble de valeurs (par exemple des entiers). On appelle type **Liste Circulaire de Valeur** et on note **ListeCirc (Valeur)** l'ensemble des listes circulaires dont les valeurs sont des éléments de *Valeur*.

Ensembles définis et utilisés : *ListeCirc*, Valeur, Place, booléen

Description fonctionnelle des opérations :

-	dernierlc :	ListeCirc (Valeur)	→	Place
-	vallc :	ListeCirc (Valeur) x Place-{nil}	→	Valeur
-	suclc :	ListeCirc (Valeur) x Place-{nil}	→	Place
-	estvidelc :	ListeCirc (Valeur)	→	booléen
-	lcvide :		→	ListeCirc (Valeur)
-	adjtlc :	ListeCirc (Valeur) x Valeur	→	
-	suptlc :	ListeCirc (Valeur)-{lcvide( )}	→	
-	adjqlc :	ListeCirc (Valeur) x Valeur	→	
-	supqlc :	ListeCirc (Valeur)-{lcvide( )}	→	
-	adjlc :	(ListeCirc (Valeur)-{lcvide( )})x(Place-{nil}) x Valeur	→	
-	suplc :	(ListeCirc (Valeur)-{lcvide( )})x(Place-{nil})	→	
-	chglc :	(ListeCirc (Valeur)-{lcvide( )})x(Place-{nil}) x Valeur	→	

Les opérations *adjtlc*, *suptlc*, *adjqlc*, *supqlc* , *adjlc*, *suplc* et *chglc* modifient la liste circulaire donnée en paramètre. L'opération *estvidelc* rend vrai si la liste circulaire est vide. Si la liste contient un seul élément, celui-ci est son propre successeur.