```
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
```

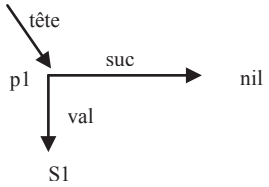
4.2 Les files

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 ».

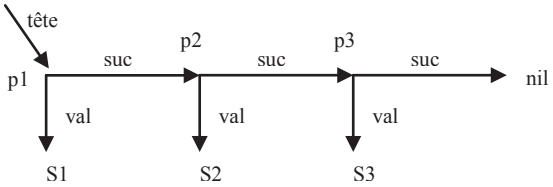
Exemple : file d'attente, par exemple queue (disciplinée) au téléski

Le matin :

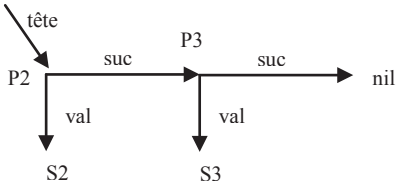
Arrivée du skieur S1



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

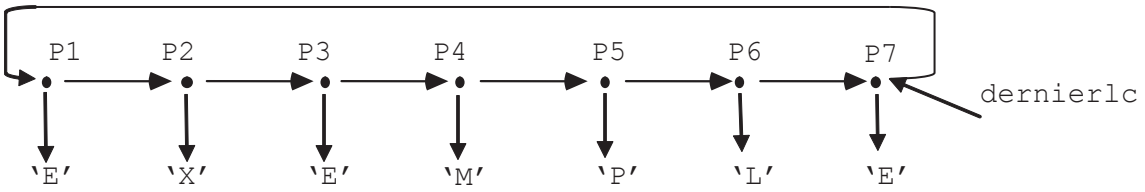
Corrigé

```
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.

Exercice :

Pour jouer à AM-STRAM-GRAM, des enfants forment une ronde, choisissent l'un d'eux comme le premier, commencent à compter à partir de celui-ci et décident que le kième enfant doit quitter la ronde. Ils recommencent ensuite en comptant à partir de l'enfant qui suivait celui qui est sorti, et ainsi de suite. En représentant la ronde des enfants par une liste circulaire, écrire l'algorithme logique de la fonction qui permet d'imprimer la suite des prénoms des enfants dans l'ordre où ils sont sortis de la ronde. On suppose que la ronde est non vide au départ et que les enfants sont représentés par leurs prénoms.

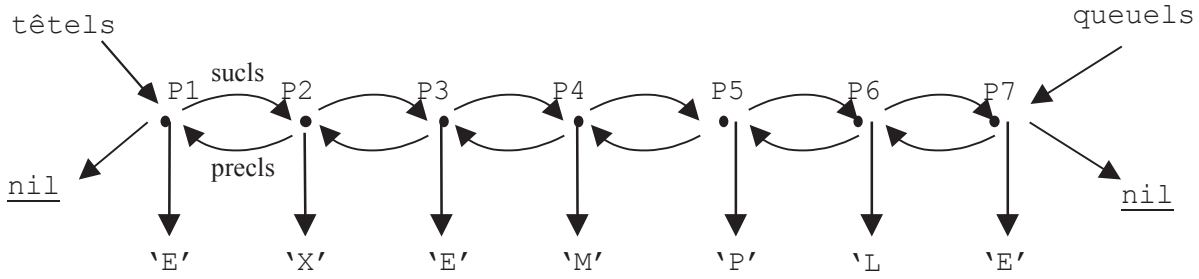
```
Corrigé
fonction AM-STRAM-GRAM (ronde InOut : Ronde, nomPremier : chaîne, k : entier)
    début (*1*)
    p ← rondeChercherDépart (ronde, nomPremier)
    tantque(*2*) non estvidelc (ronde) faire
            // on compte jusqu'au k^ième
            pour i de 1 à k-1 faire (*3a*)
                    p ← suclc (ronde, p)
            fpour
            // l'enfant à la place pSup quitte la ronde
            pSup ← p
            écrire (vallc (ronde, pSup))
            p ← suclc (ronde, p)
            suplc (ronde, pSup)
    fintantque(*2*)
    fin (*1*)
Lexique
  Ronde = ListeCirc (chaîne)
  ronde : Ronde
  nomPremier : chaîne, nom de l'enfant à partir duquel on commence à compter
  k : entier
  p : Place, place courante dans la liste
  i : entier, compteur
  psup : Place, place à supprimer

fonction rondeChercherDépart (ronde : Ronde, nom : chaîne) : Place
    début (*1*)
    p ← dernierlc (ronde)
    arrêt ← faux
    tantque(*2*) non arrêt faire
        si vallc (ronde, p) = nom alors (*3a*)
                arrêt ← vrai
        sinon (*3s*)
                p ← suclc (ronde, p)
                si  p = dernierlc (ronde)  alors (*4a*)
                        arrêt ← vrai
                fsi(*4*)
        fsi(*3*)
    fintantque(*2*)
    retourne p
    (* si le nom n'a pas été trouvé dans la ronde, on commence par le dernier élément de la liste circulaire*)
    fin (*1*)
Lexique
  ronde : Ronde, liste circulaire non vide
  nom : chaîne, nom de l'enfant à partir duquel on commencera à compter
  p : Place, place courante dans la liste puis place de l'enfant cherché s'il existe dans la liste, sinon place du dernier
  arrêt : booléen, à vrai si on a trouvé l'enfant à partir duquel on commencera à compter ou si on a parcouru la liste
      en entier
```

Attention: On mémorise une place dans une variable pour y accéder après une suppression dans la liste. Une représentation contiguë ne sera donc pas possible pour cette liste car les places seraient modifiées lors de la suppression et la place mémorisée ne correspondrait plus à celle souhaitée.

5.2 Les listes symétriques

Une liste symétrique est une liste telle que chaque élément désigne l'élément suivant et l'élément précédent.



L'intérêt des listes symétriques réside dans le fait qu'il est facile d'extraire un élément à partir de sa place. Il n'est pas nécessaire de parcourir la liste pour retrouver le précédent.

Définition abstraite :

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

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

Description fonctionnelle des opérations :

-	têtels :	ListeSym (Valeur)	→	Place
-	queuels :	ListeSym (Valeur)	→	Place
-	valls :	ListeSym (Valeur) x Place-{nil}	→	Valeur
-	sucls :	ListeSym (Valeur) x Place-{nil}	→	Place
-	precls :	ListeSym (Valeur) x Place-{nil}	→	Place
-	finls :	ListeSym (Valeur) x Place	→	booléen
-	lsvide :		→	ListeSym (Valeur)
-	adjtls :	ListeSym (Valeur) x Valeur	→	
-	suptls :	ListeSym (Valeur)-{lsvide( )}	→	
-	adjqls :	ListeSym (Valeur) x Valeur	→	
-	supqls :	ListeSym (Valeur)-{lsvide( )}	→	
-	adjls :	(ListeSym (Valeur)-{lsvide( )})x(Place-{nil})xValeur	→	
-	supls :	(ListeSym (Valeur)-{lsvide( )})x(Place-{nil})	→	
-	chgls :	(ListeSym (Valeur)-{lsvide( )})x(Place-{nil})xValeur	→	

Les opérations *adjtls*, *suptls*, *adjqls*, *supqls* , *adjls*, *supls* et *chgls* modifient la liste symétrique donnée en paramètre.

Exercice :

On appelle palindrome un mot qui se lit de la même façon de gauche à droite et de droite à gauche. Par exemple, les mots « elle » et « radar » sont des palindromes. Un mot étant représenté par une liste symétrique de caractères, écrire l'algorithme logique de la fonction qui indique si un mot est un palindrome. On suppose que le mot à analyser comprend au moins 2 lettres.

Corrigé

```
Fonction motChercherPalindrome (mot : Mot) : booléen
        début (*1*)
        pDeb ← têtels (mot)
        pFin ← queuels (mot)
        palindrome ← vrai
        arrêt ← faux
        tant que non arrêt faire (*2*)
            vDeb ← valls ( mot, pDeb)
```

```
                vFin ← valls ( mot, pFin)
                si vDeb ≠ vFin alors (*3a*)
                        arrêt ← vrai
                        palindrome ← faux
                sinon (*3s*)
                        pDeb ← sucls (mot, pDeb)
                        si pDeb = pFin alors (*4a*)
                                arrêt ← vrai
                        sinon (*4s*)
                                pFin ← precls (mot, pFin)
                                si pDeb = pFin alors (*5a*)
                                        arrêt ← vrai
                                fsi (*5*)
                        fsi (*4*)
                fsi (*3*)
        ftant (*2*)
        retourne palindrome
        fin (*1*)
Lexique
        Mot = ListeSym (caractère)
        mot : Mot
        pDeb :  Place, place courante à partir du début
        pFin : Place, place courante à partir de la fin
        vDeb : caractère, caractère courant à partir du début
        vFin : caractère, caractère courant à partir de la fin
        arrêt : booléen , à vrai lorsque le mot a été parcouru en entier ou dès qu'il ne peut plus être un palindrome
        palindrome : booléen, à vrai si le mot est un palindrome
```

6 Exercices récapitulatifs

Exercice 1 : liste d'admissions

Reprenons l'exemple de la liste d'admissions des étudiants dans une école. Rappelons qu'elle est triée par note décroissante et qu'elle contient pour chaque étudiant son nom et une note.
1) Ecrire l'algorithme logique de la fonction de recherche de la place d'un étudiant à l'aide de son nom dans la liste des admissions.
2) Même question pour l'adjonction d'un couple (nom, note) dans la liste, supposée non vide au départ.
3) Même question pour la suppression dans la liste d'un étudiant donné par son nom. On suppose la liste non vide au départ.

Correction question 1 :

```
Fonction lEtudChercherElément (listeEtudiant : Liste(Etudiant), nomEtudiant : chaîne) : Place
        début (*1*)
        placeElémEtudiant ← tete (listeEtudiant)
        trouve ← faux
        tant que non finliste (listeEtudiant, placeElémEtudiant) et non trouve faire (*2*)
            élémEtudiant ← val (listeEtudiant, placeElémEtudiant)
            si élémEtudiant.nom = nomEtudiant
                    alors (*3a*) trouve ← vrai
                    sinon (*3s*) placeElémEtudiant ← suc (listeEtudiant, placeElémEtudiant)
            fsi (*3*)
        ftant (*2*)
        retourne placeElémEtudiant
        fin (*1*)
Lexique
        listeEtudiant : Liste(Etudiant)
        nomEtudiant : chaîne, nom de l'étudiant dont on cherche la place dans listeEtudiant
        placeElémEtudiant : Place, place de l'étudiant cherché ou nil s'il n'y est pas
        élémEtudiant : Etudiant
        trouve : booléen, à vrai si l'étudiant cherché est dans la liste
```