TD - Listes, Piles, Files

Structures & Pointeurs

Ce document a pour objectif de vous familiariser avec certaines structures de données algorithmiques abstraites basiques (les listes, les piles, et les files) en utilisant plusieurs de leurs implémentations concrètes à base de tableaux et de pointeurs.

Définition informelle d'une structure de données[1] : « *En informatique, une structure de données est une manière d'organiser les données pour les traiter plus facilement. Une structure de données est une mise en œuvre concrète d'un type abstrait.* ».

1 Listes

Les *listes* sont une des structures de données fondamentales. Il s'agit de simples conteneurs stockant des éléments. Ces conteneurs peuvent avoir une taille fixée à l'avance ou limité par l'espace disponible dans la machine qui les contiendra. Plusieurs types de listes existent : listes chaînées, listes doublement chaînées, listes circulaires, ... Pour ces exercices, nous nous contenterons sur les listes stockées dans des tableaux, et dans des listes chaînées avec pointeurs.

Les listes sont des conteneurs permettant d'accéder aux éléments par leur index : on ajoute ou supprime un élément en tête, en queue, ou n'importe où au milieu, et on recherche la position des éléments selon leur index pour pouvoir accéder aux données contenues. Ainsi, plusieurs opérations essentielles sont nécessaires pour offrir une gestion correcte des listes. Vous retrouverez les explications précises dans le cours associé.

1.1 Listes avec tableaux

Parmi les implémentations possibles, on peut s'appuyer sur les tableaux. Pour cela, on va définir au tout début un nombre maximal de cases dans lesquelles on pourra insérer des valeurs, et un nombre indiquant combien de valeurs sont actuellement stockées.

```
struct liste_t
  int[32]   tab
  entier    max_len
  entier    nb_elt
fin struct
```

1) Écrivez les différents algorithmes permettant de gérer une liste à base de tableaux à taille fixe. Les algorithmes visés sont décrits dans l'énumération juste après cette question. Dans cette implémentation, les éléments seront tous des entiers strictement positifs (donc non nuls).

— **`CreerListe_TabFixe() : Liste_TabFixe`**
La fonction alloue la mémoire pour la structure contenant le tableau. Elle fixe également la taille maximale utilisable dans le tableau

1. Wikipedia : Structure de données

— **AjouterElementQueue_TabFixe(Liste_TabFixe, Elt) : booléen**
La fonction ajoute un élément en fin de liste s'il y a assez de place, puis renvoie *vrai*. Ou, si la liste est pleine, la fonction renvoie *faux*

— **AjouterElementTete_TabFixe(Liste_TabFixe, Elt) : booléen**
La fonction ajoute un élément en début de liste et décale tous les éléments s'il y a assez de place, puis renvoie *vrai*. Ou, si la liste est pleine, la fonction renvoie *faux*

— **AjouterElementMilieu_TabFixe(Liste_TabFixe, Elt, Pos) : booléen**
La fonction ajoute un élément dans la liste à la position indiquée par *Pos* (ni au début ni à la fin) et décale les éléments nécessaires s'il y a assez de place, puis renvoie *vrai*. Si la liste est pleine, ou que la position n'existe pas, ou que la position n'est ni utilisée ni contigüe à celles utilisées, alors la fonction renvoie *faux*

— **AjouterElement_TabFixe(Liste_TabFixe, Elt, Pos) : booléen**
La fonction ajoute un élément dans la liste à la position *Pos* s'il y a assez de place et renvoie ensuite *vrai*. Elle renvoie *faux* si la liste est pleine, ou si la position n'existe pas, ou si la position n'est ni utilisée ni contigüe à celles utilisées. Cette fonction s'appuie évidemment sur les 3 précédentes fonctions d'ajout

— **SupprimerElementQueue_TabFixe(Liste_TabFixe) : booléen**
La fonction retire un élément en fin de liste s'il y en a au moins un, puis renvoie *vrai*. Ou, si la liste est vide, la fonction renvoie *faux*

— **SupprimerElementTete_TabFixe(Liste_TabFixe) : booléen**
La fonction retire un élément en début de liste s'il y en a au moins un et décale tous les éléments suivants, puis elle renvoie *vrai*. Ou, si la liste est vide, la fonction renvoie *faux*

— **SupprimerElementMilieu_TabFixe(Liste_TabFixe, Pos) : booléen**
La fonction retire un élément dans la liste à la position indiquée par *Pos* (ni au début ni à la fin) et décale les éléments nécessaires s'il y en a au moins un, puis renvoie *vrai*. Si la liste est vide, ou que la position n'existe pas, ou que la position n'est pas utilisée, alors la fonction renvoie *faux*

— **SupprimerElement_TabFixe(Liste_TabFixe, Pos) : booléen**
La fonction retire l'élément à la position *Pos* de la liste si la position est utilisée et renvoie ensuite *vrai*. Elle renvoie *faux* si la liste est vide, ou si la position n'existe pas, ou si la position n'est pas utilisée. Cette fonction s'appuie évidemment sur les 3 précédentes fonctions de suppression

— **ViderListe_TabFixe(Liste_TabFixe) : entier**
La fonction retire tous les éléments de la liste, puis elle renvoie le nombre d'éléments supprimés. Une liste vide peut être vidée, et renverra donc 0

— **SupprimerListe_TabFixe(Liste_TabFixe)**
La procédure libère la mémoire contenant la structure et le tableau. Cette procédure vide éventuellement la liste si nécessaire

— **TailleMaxListe_TabFixe(Liste_TabFixe) : entier**
La fonction renvoie le nombre maximal d'éléments que peut stocker le tableau interne à la structure

— **TailleListe_TabFixe(Liste_TabFixe) : entier**
La fonction renvoie la taille de la liste, c'est-à-dire le nombre d'éléments présents dedans

— **RechercherElement_TabFixe(Liste_TabFixe, Elt) : entier**
La fonction cherche un élément *Elt* dans la liste, puis renvoie l'index de la position du premier élément égal depuis la tête. Si aucun élément n'est trouvé, la fonction renvoie -1. On considère que le premier élément en tête est à la position 0. (Par exemple, dans la liste [5 4 4 6], si on cherche l'élément 4, la fonction renverra 1 car c'est la position du premier élément 4 trouvé)

— **RecupererElement_TabFixe(Liste_TabFixe, Pos) : entier**
La fonction récupère l'élément présent à l'index *Pos* de la liste et le renvoie. S'il n'est pas trouvé, la fonction renvoie -1

Afin de se passer du nombre maximal d'éléments insérable dans le tableau, on peut imaginer un conteneur qui agrandit le tableau à chaque fois que l'espace vient à manquer, et qui libère l'espace en trop lorsque trop d'éléments ont été supprimés. On peut fixer des limites selon les cas pratiques rencontrés, ou proposer diverses options à l'utilisateur (par exemple lors de la création de la liste, l'utilisateur peut indiquer le nombre initial de cases voulues, de combien les ajouts se feront, et au bout de quel pourcentage d'occupation on libèrera les cases en trop).

2) (*optionnel*) Écrivez les nouvelles versions de la structure de la liste, de la fonction de création, de la fonction d'ajout d'un élément, et de la fonction de suppression d'un élément pour pouvoir agrandir et réduire le tableau lorsque cela est nécessaire.

1.2 Listes chaînées avec pointeurs

En utilisant les pointeurs, on peut implémenter de façon plus générale encore les listes : on alloue de la mémoire pour chaque élément inséré, et on libère cette mémoire lors de la suppression de l'élément associé. Le nombre maximum d'éléments que l'on pourra stocker dépendra exclusivement de la quantité de mémoire disponible et non plus de la taille de la structure algorithmique allouée. Une autre différence majeure se situera dans la vitesse d'accès aux éléments : pour atteindre l'élément en position N, il est nécessaire de faire N déréférencements avant d'atteindre l'élément visé (et non plus renvoyer la case directement).

En s'appuyant sur les pointeurs, chaque élément est en réalité une structure complète renvoyant éventuellement vers l'élément suivant OU signalant qu'il s'agit du dernier élément. La création de la liste chaînée génèrera donc une structure vide qu'il faudra remplir par des ajouts successifs. Cependant, nous allons implémenter une version extrêmement épurée de cette structure afin de mieux comprendre les pointeurs. Dans ce cas précis, il n'y aura pas de fonction de création/suppression de la structure générale, mais uniquement des fonctions d'ajout et suppression d'éléments. Ces fonctions d'ajout/suppression renverront à chaque fois le nouveau pointeur de tête de liste qu'il faut absolument conserver (si on perd ce pointeur, on perd l'adresse du premier élément, donc on ne peut plus le retrouver en mémoire et celui-ci sera définitivement alloué/impossible à réutiliser par le système et d'autres programmes).

```
struct liste_p
  Elt      elt
  liste_p  *next
fin struct
```

Ainsi, on peut trouver 3 cas possibles :

- *la liste est vide* : le pointeur de liste est à **NULL**
- *la liste contient un seul élément (le premier qui est aussi le dernier)* : le pointeur de liste pointe vers une structure en mémoire, dont le champs **next** est à la valeur **NULL**
- *la liste contenant N éléments* : le pointeur de liste pointe vers une structure qui est le premier élément, et son champs **next** contient une valeur qui n'est pas **NULL**

3) Écrivez les différents algorithmes permettant de gérer une liste à base de pointeurs. Les algorithmes visés sont décrits dans l'énumération juste après cette question. Dans cette implémentation, les éléments seront tous des entiers strictement positifs (donc non nuls). Une liste vide correspond au pointeur **NULL**.

— **AjouterElementQueue_Poi(Liste_Poi, Elt) : Liste_Poi**
La fonction ajoute un élément en fin de liste puis renvoie un pointeur sur la tête de liste (éventuellement la nouvelle tête de liste)

— **AjouterElementTete_Poi(Liste_Poi, Elt) : Liste_Poi**
La fonction ajoute un élément en début de liste et décale tous les éléments, puis renvoie un pointeur sur la nouvelle tête de liste

— **AjouterElementMilieu_Poi(Liste_Poi, Elt, Pos) : Liste_Poi**
La fonction ajoute un élément dans la liste à la position indiquée par *Pos* (ni au début ni à la fin) et décale les éléments nécessaires, puis renvoie renvoie un pointeur sur la tête de liste (éventuellement la nouvelle tête de liste). Si la position n'existe pas alors la fonction renvoie *NULL*

— **AjouterElement_Poi(Liste_Poi, Elt, Pos) : Liste_Poi**
La fonction ajoute un élément dans la liste à la position *Pos* et renvoie ensuite un pointeur sur la tête de liste (éventuellement la nouvelle tête de liste). Si la position n'existe pas, elle renvoie *NULL*. Cette fonction s'appuie évidemment sur les 3 précédentes fonctions d'ajout

— **SupprimerElementQueue_Poi(Liste_Poi) : Liste_Poi**
La fonction retire un élément en fin de liste s'il y en a au moins un et le libère de la mémoire, puis renvoie un pointeur sur la tête de liste (éventuellement la nouvelle tête de liste). Si la liste est vide, la fonction renvoie *NULL*

— **SupprimerElementTete_Poi(Liste_Poi) : Liste_Poi**
La fonction retire un élément en début de liste s'il y en a au moins un, le libère de la mémoire, et décale tous les éléments suivants, puis elle renvoie un pointeur sur la nouvelle tête de liste. Si la liste est ou devient vide, la fonction renvoie *NULL*

— **SupprimerElementMilieu_Poi(Liste_Poi, Pos) : Liste_Poi**
La fonction retire un élément dans la liste à la position indiquée par *Pos* (ni au début ni à la fin) et décale les éléments nécessaires s'il y en a au moins un, puis renvoie un pointeur sur la tête de liste (éventuellement la nouvelle tête de liste). Si la liste est vide, ou que la position n'existe pas, alors la fonction renvoie *NULL*

— **SupprimerElement_Poi(Liste_Poi, Pos) : Liste_Poi**
La fonction retire l'élément à la position *Pos* de la liste et le libère de la mémoire et renvoie ensuite un pointeur sur la tête de liste (éventuellement la nouvelle tête de liste). Si la liste est vide, ou que la position n'existe pas, alors la fonction renvoie *NULL*. Cette fonction s'appuie évidemment sur les 3 précédentes fonctions de suppression

— **ViderListe_Poi(Liste_Poi) : entiers**
La fonction retire tous les éléments de la liste et libère donc tous les maillons de la mémoire, puis elle renvoie le nombre d'éléments supprimés. Une liste vide peut être vidée, et renverra donc 0

— **SupprimerListe_Poi(Liste_Poi)**
La procédure libère tous les maillons de la liste. Cette procédure vide donc la liste. Une liste vide ne peut pas être supprimée, mais la procédure ne fera rien de spécial dans ce cas

— **TailleListe_Poi(Liste_Poi) : entier**
La fonction renvoie la taille de la liste, c'est-à-dire le nombre d'éléments présents dedans

— **RechercherElement_Poi(Liste_Poi, Elt) : entier**
La fonction cherche un élément *Elt* dans la liste, puis renvoie l'index de la position du premier élément égal depuis la tête. Si aucun élément n'est trouvé, la fonction renvoie -1. On considère que le premier élément en tête est à la position 0. (Par exemple, dans la liste [5 4 4 6], si on cherche l'élément 4, la fonction renverra 1 car c'est la position du premier élément 4 trouvé)

— **RecupererElement_Poi(Liste_Poi, Pos) : Liste_Poi**
La fonction récupère l'élément présent à l'index *Pos* de la liste et renvoie un pointeur vers le maillon qui le contient. S'il n'est pas trouvé, la fonction renvoie un pointeur *NULL*. On considère que le premier élément en tête est à la position 0

2 Piles

Les **piles**, ou **stacks** en anglais, sont des structures visant à stocker les données dans l'ordre d'arrivée, mais ne permettant leur récupération uniquement dans l'ordre inverse. Dans une pile, on ne peut accéder qu'à la dernière donnée stockée, celle se situant au *sommet* de la pile. Ces structures sont aussi appelées **LIFO** (*Last In First Out*).

Deux opérations permettent d'utiliser une pile :

— **`PUSH`** : permettant d'*empiler* une donnée supplémentaire dans la pile

— **`POP`** : permettant de *dépiler* une donnée depuis la pile

On ajoute donc une donnée en l'empilant avec un **`PUSH`**, et il est possible de directement y accéder, car elle est au sommet de la pile. À l'inverse, pour accéder à une donnée tout au fond de la pile, il est nécessaire de dépiler autant d'éléments que nécessaire avec un **`POP`**.

Afin d'implémenter une pile, il est donc nécessaire d'avoir un espace de stockage ordonné (un tableau numéroté ou une liste chaînée), et un indicateur de l'élément en haut de la pile. Nous allons maintenant voir comment implémenter une pile avec des listes chaînées et un tableau de taille fixe.

2.1 Piles avec tableaux

Une implémentation avec un tableau de taille fixe impose cette fois une limitation : la pile aura une taille maximale, et on peut refuser l'ajout d'un élément si la pile est déjà pleine. La structure diffère également du fait que le tableau est alloué une seule fois lors de sa création (voire même lors de la compilation dans le cas statique).

```
struct pile_t
  int[32]   tab
  entier    max_len
  entier    nb_elt
  entier    head
fin struct
```

On notera cette fois que plusieurs informations distinctes doivent être conservées : le tableau, le numéro de case correspondant au sommet de la pile (*head* dans notre cas), la taille du tableau (le nombre maximum d'objets pouvant être stockés), le nombre d'éléments dans le tableau.

Dans le cas d'un tableau de taille fixe, le pointeur de sommet ne peut pas utiliser la valeur **`NULL`** comme indicateur de tableau vide, car cette valeur est égale à 0 (ce qui laisserait à penser que le sommet est effectivement à la case 0). Plusieurs solutions sont possibles pour indiquer le sommet de la pile et le cas vide :

— On enregistre dans la structure de la pile une variable servant à compter le nombre d'éléments présents (le sommet peut donc prendre n'importe quelle valeur tant que la pile est vide).

— On utilise un entier relatif pour indiquer le sommet, et -1 indique que la pile est vide (l'ajout d'un élément décalera le sommet à 0, c'est-à-dire la case où sera l'élément).

— On place le sommet de la pile sur la première case non utilisée, et l'accès au premier élément se fait donc en retirant 1 au pointeur de sommet (ainsi, un sommet à la case 0 indique que la pile est vide). Attention : dans ce cas précis, un tableau plein aura un sommet hors des cases du tableau (il ne faudra donc *jamais* le déréférencer s'il atteinte une telle valeur).

4) Écrivez les différents algorithmes permettant de gérer une pile à base de tableaux à taille fixe. Les algorithmes visés sont décrits dans l'énumération juste après cette question. Dans cette implémentation, les éléments seront tous des entiers strictement positifs (donc non nuls).

Les principales opérations se résument ainsi :

— Création : on alloue en mémoire le tableau (sauf s'il est statique), et on fixe le sommet de la pile à la valeur prévue pour démarrer (-1, 0, ou toute autre valeur choisie) [éventuellement, on met à jour le nombre d'objets dans le tableau en le fixant à 0].

— Empiler : si le tableau est plein, on retourne une erreur, sinon, on ajoute un élément, et on décale le sommet de la pile [éventuellement, on met à jour le nombre d'objets dans le tableau].

— Dépiler : si la pile est vide, on retourne une erreur, sinon, on réduit la valeur du sommet de la pile [éventuellement, on met à jour le nombre d'objets dans le tableau].

— Vider : on fixe le sommet de la pile à la valeur prévue pour démarrer (-1, 0, ou toute autre valeur choisie) [éventuellement, on met à jour le nombre d'objets dans le tableau en le fixant à 0].

— Taille : on renvoie le nombre d'éléments présents dans la pile.

— TailleMax : on renvoie le nombre d'éléments maximum stockables dans la pile.

— Sommet : on renvoie le dernier élément ajouté (cela dépend de comment le sommet a été implémenté !).

2.2 Piles avec pointeurs

Une implémentation à l'aide d'une liste chaînée permet d'exploiter la mémoire et d'être donc beaucoup plus flexible en terme de nombre maximum d'éléments.

```
struct pile_p
  struct maillon_p  *head
fin struct

struct maillon_p
  Elt               elt
  struct maillon_p  *next
fin struct
```

On y retrouve plusieurs fois la structure typique des listes chaînées (un élément et un pointeur vers l'élément suivant) en tant que maillons, ainsi qu'une structure contenant un pointeur indiquant le sommet de la pile (*head* dans notre cas).
L'unique cas particulier concerne une pile vide : le pointeur de sommet vaut dans ce cas **NULL**, et il n'y a donc aucun maillon en mémoire. Il s'agit également de l'état dans lequel se trouve une pile vidée ou nouvellement créée.

5) Écrivez les différents algorithmes permettant de gérer une pile à base de pointeurs. Les algorithmes visés sont décrits dans l'énumération juste après cette question. Dans cette implémentation, les éléments seront tous des entiers strictement positifs (donc non nuls). Une liste vide correspond au pointeur **NULL**.

Les principales opérations se résument ainsi :

— Création : on alloue en mémoire la structure générale de la pile, et on fixe le sommet de la pile à **`NULL`**.

— Empiler : on alloue en mémoire un nouvel élément dont le pointeur *next* pointe vers l'actuel élément au sommet de la pile, puis, on met à jour le pointeur de sommet de la pile vers l'adresse de ce nouvel élément.

— Dépiler : si la pile est vide, on retourne une erreur, sinon, on récupère tout d'abord l'adresse de l'élément suivant celui au sommet, puis, on libère l'élément au sommet, puis, on met à jour le pointeur de sommet de la pile vers l'adresse de l'élément suivant.

— Vider : on dépile successivement tous les éléments jusqu'à obtenir un sommet à **`NULL`**.

— Taille : on renvoie le nombre d'éléments présents dans la pile.

— Sommet : on renvoie le contenu de l'élément au sommet de la pile.

3 Files

Les **files**, ou **queues** en anglais, sont des structures visant à stocker et rendre les données dans l'ordre d'arrivée. Une file dispose donc d'une *tête* contenant l'élément le plus ancien (inséré avant tous les autres), et une *queue* contenant l'élément inséré le plus récemment. Ces structures sont aussi appelées **FIFO** (*First In First Out*).

Deux opérations permettent d'utiliser une file :

— **ENQUEUE** : permettant d'*enfiler* une donnée supplémentaire dans la file

— **DEQUEUE** : permettant de *défiler* une donnée depuis la file

On ajoute donc une donnée en l'enfilant avec un **ENQUEUE**, celle-ci se retrouve en *queue* de file, c'est-à-dire au fond de la file. On récupère une donnée en défilant avec un **DEQUEUE**, celle-ci se trouvait en *tête* de file, c'est-à-dire qu'elle attendait son tour depuis son insertion. On accède donc aux éléments dans l'ordre d'arrivée.

Afin d'implémenter une file, il est donc nécessaire d'avoir un espace de stockage ordonné (un tableau numéroté ou une liste chaînée), et deux indicateurs pour l'élément tête de file et l'élément en queue de file. Nous allons maintenant voir comment implémenter une file avec des listes chaînées et un tableau de taille fixe.

3.1 Files avec tableaux

Une implémentation avec un tableau de taille fixe impose cette fois une limitation : la file aura une taille maximale, et on peut refuser l'ajout d'un élément si la file est déjà pleine. La structure diffère également du fait que le tableau est alloué une seule fois lors de sa création (voire même lors de la compilation dans le cas statique).

```
struct file_t
  int[32]   tab
  entier    max_len
  entier    head
  entier    tail
fin struct
```

On notera cette fois que plusieurs informations distinctes doivent être conservées : l'adresse du tableau, le numéro de case correspondant à la tête de la file (*head*), le numéro de case correspondant à la queue de la file (*tail*), la taille du tableau (le nombre maximum d'objets pouvant être stockés), et le nombre d'éléments dans la file.

Une façon d'implémenter le tableau permet de supposer que la case 0 contiendra toujours l'élément de tête. Dans ce cas très précis, on peut donc se passer de la variable de tête, et au contraire, s'appuyer sur la variable donnant le nombre d'éléments dans le tableau pour savoir si la file est vide ou non.

Dans le cas d'un tableau de taille fixe, les pointeurs de tête et de queue ne peuvent pas utiliser la valeur **NULL** comme indicateur de tableau vide, car cette valeur est égale à 0 (ce qui laisserait à penser que la queue est effectivement à la case 0). Plusieurs solutions sont possibles pour indiquer les cases où se trouvent la tête et la queue de la file, ainsi que le cas vide :

— On enregistre dans la structure de la file une variable servant à compter le nombre d'éléments présents (la queue peut donc prendre n'importe quelle valeur tant que la file est vide).

— On utilise un entier relatif pour indiquer la position, et -1 indique que la file est vide (l'ajout d'un élément décalera la tête et la queue à 0, c'est-à-dire la case où sera l'élément).

— On place la queue de la file sur la première case non utilisée, et l'accès au dernier élément se fait donc en retirant 1 au pointeur de sommet (ainsi, une queue à la case 0 indique que la file est vide). Attention : dans ce cas précis, un tableau plein aura un sommet hors des cases du tableau (il ne faudra donc *jamais* le déréférencer s'il atteinte une telle valeur).

— On représente complètement différemment la file : on décale les pointeurs de tête et de queue au fur et à mesure des insertions et suppressions ($+1$ / -1). Le pointeur de queue peut passer de la dernière case à la première, car les éléments actuellement dans la file se trouvent uniquement entre la tête et la queue. Vider ce tableau revient uniquement à passer les pointeurs à la valeur -1. Attention : dans ce cas précis, il est nécessaire de faire très attention à l'ordre de lecture des éléments entre la tête et la queue.

Cette autre façon de gérer la file est un tout petit peu plus complexe dans l'écriture des algorithmes de gestion, mais elle évite énormément de réécritures dans le tableau/en mémoire (inutile de lire une case, l'écrire ailleurs, et recommencer ainsi de suite lors de chaque *dequeue*).
Pour nos exercices, nous nous contenterons de démarrer les pointeurs de tête et de queue à -1 pour qu'ils disposent d'une adresse explicitement impossible à utiliser lorsque la file est vide.

6) Écrivez les différents algorithmes permettant de gérer une file à base de tableaux à taille fixe. Les algorithmes visés sont décrits dans l'énumération juste après cette question. Dans cette implémentation, les éléments seront tous des entiers strictement positifs (donc non nuls).

Les principales opérations se résument ainsi :

— Création : on alloue en mémoire le tableau (sauf s'il est statique), et on fixe la tête et la queue de la file à la valeur prévue pour démarrer (-1, 0, ou toute autre valeur choisie) [éventuellement, on met à jour le nombre d'objets dans le tableau en le fixant à 0].

— Enfiler : si le tableau est plein, on retourne une erreur, sinon, on ajoute le nouvel élément à gauche de la position du pointeur de queue, et on décale la queue de la file d'un cran à gauche [éventuellement, on met à jour le nombre d'objets dans le tableau]. Autre version : on ajoute le nouvel élément dans la case à gauche de la position du pointeur de queue modulo la taille du tableau, et on décale la queue de la file.

— Défiler : si la file est vide, on retourne une erreur, sinon, on décale l'ensemble des éléments vers la droite [éventuellement, on met à jour le nombre d'objets dans le tableau]. Autre version : on décale la tête de la file d'un cran à gauche.

— Vider : on fixe les pointeurs de tête et de queue à la valeur prévue pour démarrer (-1, 0, ou toute autre valeur choisie) [éventuellement, on met à jour le nombre d'objets dans le tableau en le fixant à 0].

— Taille : on renvoie le nombre d'éléments présents dans la file.

— TailleMax : on renvoie le nombre d'éléments maximum stockables dans la file.

— Tête : on renvoie le contenu de la case du pointeur de tête de file (le prochain élément qui sera défilé).

— Queue : on renvoie le contenu de la case du pointeur de queue de file (le dernier élément qui sera défilé).

3.2 Files avec pointeurs

Une implémentation à l'aide d'une liste chaînée permet d'exploiter la mémoire et d'être donc beaucoup plus flexible en terme de nombre maximum d'éléments.

```
struct file_p
  struct maillon_f  *head
  struct maillon_f  *tail
fin struct

struct maillon_f
  Elt               elt
  struct maillon_f  *next
fin struct
```

On y retrouve plusieurs fois la structure typique des listes chaînées (un élément et un pointeur vers l'élément suivant), ainsi que deux pointeurs indiquant respectivement la tête de la file (*head*) et la queue de la file (*tail*).
Deux cas particuliers concernent les files où le pointeur de tête et le pointeur de queue valent la même chose : la file vide où les pointeurs contiennent **NULL**, et la file contenant un seul élément vers lequel les deux pointeurs renvoient. Une file nouvellement créée se trouve dans l'état vide.

7) Écrivez les différents algorithmes permettant de gérer une file à base de pointeurs. Les algorithmes visés sont décrits dans l'énumération juste après cette question. Dans cette implémentation, les éléments seront tous des entiers strictement positifs (donc non nuls). Une liste vide correspond au pointeur **NULL**.

Les principales opérations se résument ainsi :

- Création : on alloue en mémoire la structure générale de la file, et on fixe les tête et queue de la file à **NULL**.
- Enfiler : on alloue en mémoire un nouvel élément, on met son pointeur *next* à **NULL**, puis, si la file est vide, on met les pointeurs de tête et de queue sur le nouvel élément, sinon, on met l'adresse du nouvel élément sur le pointeur *next* de l'élément pointé par la queue, et on met à jour le pointeur de queue sur le nouvel élément.
- Défiler : si la file est vide, on retourne une erreur, sinon, on récupère tout d'abord l'adresse de l'élément suivant celui en tête, puis, on libère l'élément en tête, puis, on met à jour le pointeur de tête de la file vers l'adresse de l'élément suivant. Si l'élément suivant est **NULL**, on met à jour le pointeur de queue également.
- Vider : on défile successivement tous les éléments jusqu'à obtenir la tête à **NULL** (ne pas oublier que l'opération qui défile met à jour la queue dans le cas **NULL**).
- Taille : on renvoie le nombre d'éléments présents dans la file.
- Tête : on renvoie le contenu de l'élément en tête de file (le prochain élément qui sera défilé).
- Queue : on renvoie le contenu de l'élément en queue de file (le dernier élément qui sera défilé).