Cours 2 : LES TDA PILE et FILE

- **Retour sur les listes : problèmes de pointeurs**
- Le TDA PILE
- Réalisations du TDA PILE
- Utilisations classiques des piles
- Le TDA FILE
- Réalisations du TDA PILE

L'étape précédente

- Au cours n°1 nous avons étudié les listes.
  - Nous avons d'abord présenté la philosophie des types de données abstraits puis nous avons insisté sur les différentes représentations des listes (avec ou sans en-tête, par tableau, simplement chaînées, doublement chaînées).
- Dans ce cours nous allons
  - revenir sur les listes (et plus généralement sur les conteneurs) en nous intéressant aux problèmes liés aux objets contenus :
    - sont-ils contenus par adresse ou par valeur ?
  - nous nous intéresserons à des listes particulières où lon se contente d'insertions et de suppressions aux extrémités de la liste.
    - On parle alors de pile et de file.

Problèmes de pointeurs

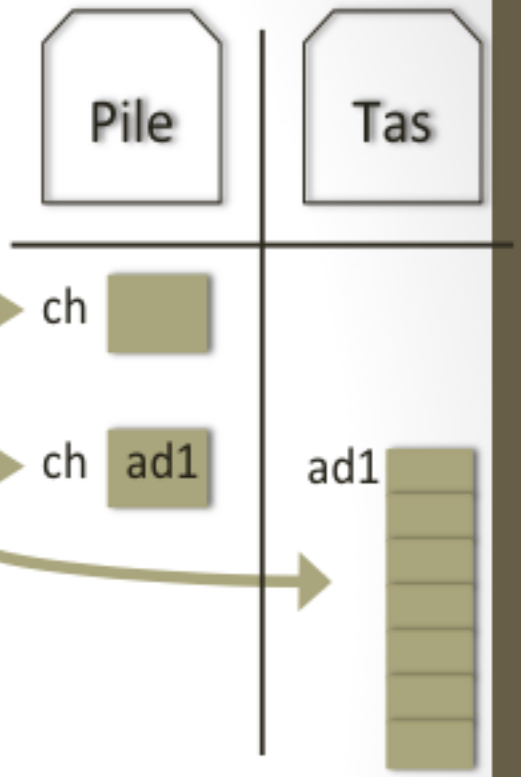
- Le B-A BA des pointeurs
  - Bien distinguer :
    - La déclaration d'un pointeur : allocation automatique (dans la pile d'exécution) d'une zone pour mémoriser une adresse d'un objet qui sera (un jour) alloué dans le tas
    - L'initialisation du pointeur : l'exécution d'une instruction malloc : allocation dynamique d'un objet dans le tas et retour de l'adresse où a été créé cet objet
- Sécurité/efficacité
- Fuites de mémoire
- Pointeurs fous (références pendantes)
- Stockage direct/ stockage indirect

Le B-A BA

```
void NOubliezJamaisCa (void) {
  char * ch ;
    /*allocation automatique d'une zone dans la pile pour
      contenir l'adresse d'un caractère */
  ch = (char *) malloc (5 * sizeof(char)) ;
    /*
    1/ allocation dynamique d'une zone dans le tas pour
      contenir 5 caractères
    2/ retour dans ch de l'adresse de cette zone */
  }
  /* sortie de la fonction et libération de ch
      mais pas de la zone pointée : fuite de
      mémoire à redouter
  */
```

Le B-A-BA

```
void NOubliezJamaisCa (void) {
   char * ch ;
      /*allocation automatique d'une zone dans la pile
        pour contenir l'adresse d'un caractère */
   ch = (char *) malloc (5 * sizeof(char)) ;
      /*
      1/ allocation dynamique d'une zone dans le tas
         pour contenir 5 caractères
      2/ retour dans ch de l'adresse de cette zone */
   }
   /* sortie de la fonction et libération de ch
       mais pas de la zone pointée : fuite de
       mémoire à redouter
   */
```



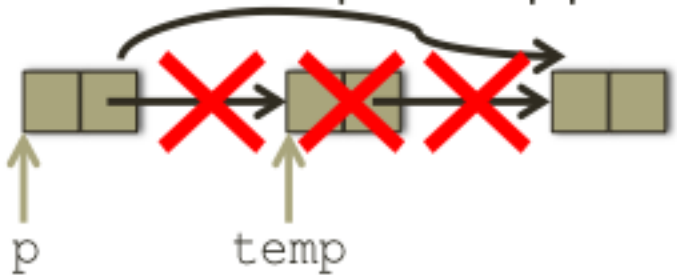
Le B-A BA

- **À savoir par cœur et à retenir jusqu'à la fin de sa vie de programmeur C**
  - En C la déclaration d'un pointeur ne fait que ce qu'elle peut faire : **réserver de la mémoire dans la pile pour une adresse** ;
  - en aucun cas elle n'alloue de la mémoire pour l'objet pointé.
- L'objet pointé doit :
  1. être alloué explicitement par le programmeur (malloc)
  2. être désalloué explicitement par le programmeur (free)

Sécurité/efficacité : exemple

```
bool ListeSupprimer (POSITION p, LISTE L)
{  /*o(1)*/
  cellule *  temp ;
  if (! ListeVide(L)) {
      temp = p->suivant;
       p->suivant=p->suivant->suivant;
     free(temp);  /*danger */
  }
}
```

Faut-il vérifier la validité de la position p passée en paramètre ?



Sécurité/efficacité : exemple

- **Problème de conception : Sécurité/ efficacité**
  - Dans `ListeSupprimer` la position est passée en paramètre par l'utilisateur ;
    - si la position n'est pas valide le `free(temp)` peut créer une catastrophe.
    - On peut donc se demander s'il ne serait pas prudent de vérifier cette position.
    - Mais alors `ListeSupprimer` deviendrait en o(n) dans le pire des cas.
      - Au pire (insertion en fin de liste ou position invalide) on parcourt la liste toute entière.
- **Problème :**
  - Si on fait confiance à la sagacité des programmeurs utilisateurs on gagne en efficacité mais on court un risque de vermines (bugs): on peut être sûr que quelqu'un se plantera un jour (loi dite "de l'emmer… maximum")
  - Si on est méfiant, on perd en efficacité mais on gagne en sécurité.
- **Conseil** : 2 temps
  - phases de mises au point : solution prudente, vérifier la validité de la position
  - phase opérationnelle : solution efficace, supprimer ces gardes fous.

La peste et le choléra

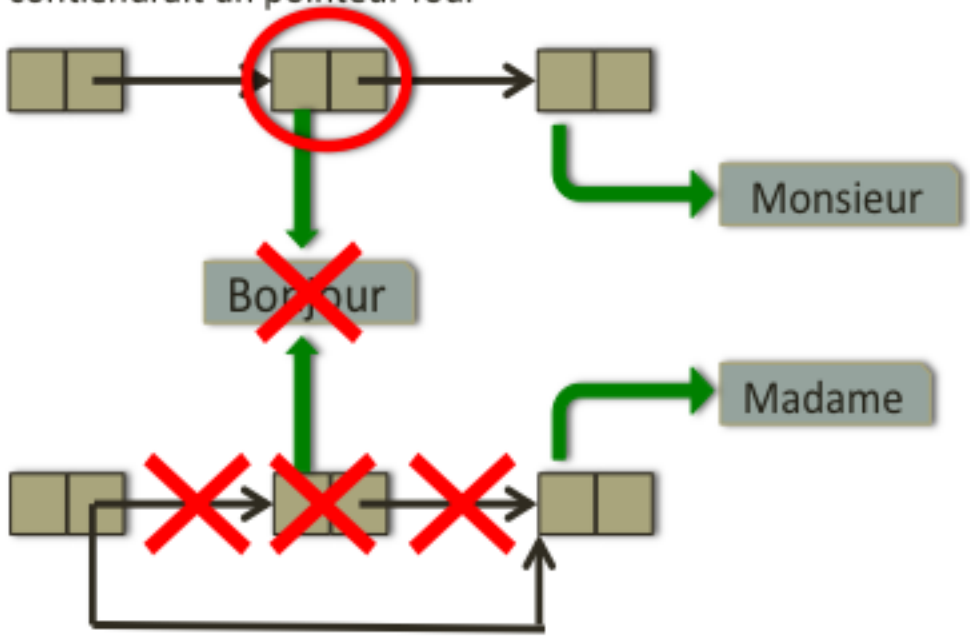
- Les deux fléaux qui guettent les programmes C :
- Les fuites de mémoire :
  - Des objets créés par des allocations dynamiques
    (des zones mémoires allouées dans le tas par un malloc)
    ne sont plus accessibles et n'ont pas été désalloués
- Les pointeurs fous (ou références pendantes) :
  - Le programme manipule des pointeurs vers une zone mémoire qui a déjà été désallouée

Fuite de mémoire

- Exemples de situations présentant des risques
  - Un conteneur mémorise des objets en stockage indirect et un objet est détaché
  - Un conteneur mémorise en stockage direct un objet qui possède des champs dynamiques (beurk, beurk)
  - Une fonction retourne un résultat créé en mémoire dynamique (programmation fonctionnelle)
- Solutions :
  - Les glanneurs de cellules (hélas pas en C)
  - Le compteur de référence (cf. C++)
  - La vigilance des programmeurs (ie. cata prévisibles)

Pointeur fou

- Exemples de situations présentant des risques
  - Des conteneurs en stockage indirect partagent des objets
    - **Exemple:**
      - Quand deux listes partagent un objet, si ListeSupprimer détruisait l'objet pointé (ce qui n'est pas le cas dans les programmes du cours 1), alors la deuxième liste contiendrait un pointeur fou.



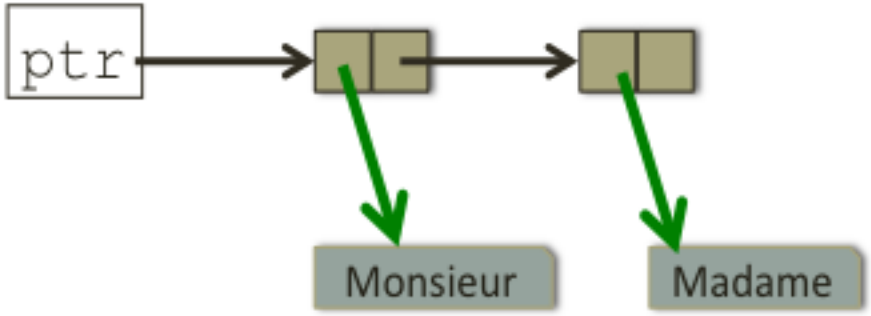
Pointeur fou

- Exemples d'erreur de programmation à se faire virer pour incompétence
  - Des variables de type pointeur n'ont pas été initialisées et sont manipulées (horreur)
    - **Exemple:**
      - typedef ELEMENT char * ;
      - ELEMENT e ; **cette instruction ne fait que ce qu'elle peut faire** elle réserve de la mémoire pour un pointeur sur un caractère, pas pour une chaîne de caractères.
  - Retour par adresse d'un objet alloué localement à une fonction

Stockage direct/indirect

- Stockage direct (par valeur)
  - Le TDA ELEMENT est un objet alloué automatiquement

- Stockage indirect (par adresse)
  - Le TDA ELEMENT est un pointeur sur un objet alloué en mémoire dynamique



Stockage direct/indirect

- Dans le cas du stockage indirect c'est à l'utilisateur de réserver de la mémoire pour l'objet pointé et de libérer cette mémoire.

On dit que la liste n'est pas propriétaire des objets qu'elle contient.

- Pour cette raison on ajoute deux primitives au TDA ELEMENT
  - `ELEMENT` **`ElementCreer`** `()` qui alloue de la mémoire pour l'élément vide et retourne un pointeur sur cette zone
  - `void` **`ElementDétruire`**`(ELEMENT x)` qui libère la mémoire pointée par ELEMENT

Stockage direct/indirect

- **Création d'une cellule**
  - `malloc(sizeof(cellule));`
    - cas du stockage direct : (ex ELEMENT est un entier) quand on crée une cellule on alloue de la mémoire pour un élément (ie pour l'objet stocké)
    - cas du stockage indirect : (ex ELEMENT est un pointeur sur une fiche) quand on crée une cellule on alloue de la mémoire pour un pointeur (**on n'alloue pas la mémoire pour la fiche mais seulement pour un pointeur sur une fiche**)
- **Destruction d'une cellule**
  - `free(unPointeurSurUneCellule)` libère la zone mémoire allouée pour la cellule
    - dans le cas du stockage direct : l'objet contenu dans la cellule est détruit (sauf s'il possède des champs dynamiques)
    - dans le cas du stockage indirect (ELEMENT est un pointeur) seuls les 2 pointeurs sont libérés et non la zone pointée QUI DOIT ALORS ËTRE ADOPTÉE pour qu'il n'y ait pas de fuites de mémoire

Stockage direct ou indirect

stockage	direct	indirect
objet mémorisé	lui-même	son adresse
utilisation	type simple ou objets pas trop gros	objets volumineux
un paramètre de type élément	passage par valeur	passage par adresse
objet dans plusieurs listes	dupliqué	partagé
allocation/ libération	automatique (lors de l'insertion, suppression)	dynamique (globale)

SLOGANS

- **en stockage direct un conteneur est propriétaire des objets qu'il contient ;**
- **en stockage indirect (généralement) un conteneur n'est pas propriétaire des objets qu'il contient.**

Règles de bonne conduite

- Un conteneur est propriétaire
  - De ses structures (cellules chaînées, tableau, etc.)
  - Pas des objets qu'il contient si ceux-ci sont alloués dans le tas
- Principes à retenir et à appliquer, en particulier en TP et en examen,
  - **En stockage direct** la structure du conteneur contient l'objet qui est donc créé et détruit en même temps que la structure. Pas de champ dynamique
  - **En stockage indirect** faire très attention à la durée de vie des objets stockés surtout s'ils sont partagés par plusieurs conteneurs
    - Si cela est le cas :
      - un seul conteneur est propriétaire des objets : celui qui a la durée de vie la plus longue ;
      - aucun des autres n'est propriétaire des objets.

Règles de bonne conduite

- En C et C++ se méfier du style fonctionnel qui retourne par adresse un objet alloué en mémoire dynamique
  - Résultat alloué dans la pile : viré pour incompétence
  - Résultat alloué dans la tas : fuite de mémoire possible
- Dans tout programme C /C++
  - A priori : commencer par penser à qui va allouer et qui va désallouer les objets manipulés par pointeur
  - A posteriori : vérifier en faisant l'inventaire des objets manipulés par pointeur et en explicitant qui les créé, qui les utilise et qui les détruit

Cours n°2 : LES TDA PILE et FILE

- Retour sur les listes : problèmes de pointeurs
- **Le TDA PILE**
- Réalisations du TDA PILE
- Utilisations classiques des piles
- Le TDA FILE
- Réalisations du TDA PILE

Le TDA PILE

- Définitions
  - Une pile est une liste de type "LIFO"
    (last in First out)
  - Une pile est une liste où toute insertion ou suppression se fait à une position particulière appelée le « sommet de la pile »
- Métaphore
  - La pile d'assiettes
- Utilisations classiques
  - Évaluations d'expressions arithmétiques
    - TD3, TP2, Horowitz p. 111
  - Pile système
    - Horowitz p 98
    - Aho (concepts ..., p. 339)
    - Aho (compilateurs, chapitre 7)

Les Primitives du TDA Pile

- `PILE` **`PileCreer`** `(int profondeur) ;`

  /* crée et retourne une pile en lui allouant de la mémoire dynamique
  La pile ainsi crée est pour l'instant vide
  Le paramètre entier désigne la profondeur maximale de la pile (pleine) */
- `void` **`PileDetruire`**`(PILE);`

  /* libère l'espace occupé par la Pile */
- `ELEMENT` **`PileSommet`**`(PILE);`

  /* retourne l'élément au sommet de la pile (sans le dépiler) */
- `ELEMENT` **`PileDepiler`**`(PILE);`

  /* retourne l'élément au sommet de la pile ET l'enlève de la pile) */
- `bool` **`PileEmpiler`**`(ELEMENT, PILE);`

  /* ajoute l'élément au sommet de la pile */
- `void` **`PileAfficher`** `(PILE ) ;`
- `void` **`PileRaz`**`(PILE) ;`
- `void` **`PileNettoyer`** `(PILE)`

Mises en œuvre du TDA PILE

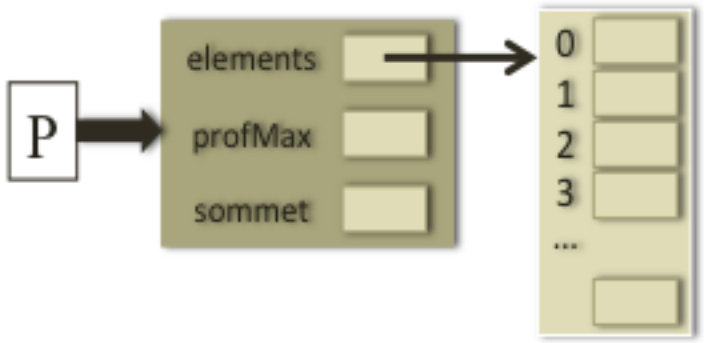
- On peut réaliser les primitives du TDA PILE à l'aide des primitives du TDA LISTE

```
typedef LISTE PILE
/* une pile est une liste */
bool PileEmpiler (ELEMENT x, Pile P) {
      return(ListeInserer(x, ListePremier(P), P) ;
      }
      etc…
```

- La Complexité dépend alors de la représentation des listes
  - Par cellules chaînées : en o(1)
  - Par cellules contiguës : en o(n)
    - (à cause des décalages quand on insère en tête…pas très malin),
- **Dès que l'on s'intéresse à la complexité des opérations on touche les limites de l'approche TDA** ;
  - pour implémenter efficacement les piles avec les listes il faut connaître la réalisation des listes
- il vaut mieux ne pas utiliser le TDA LISTE et définir les primitives à partir d'une SDF

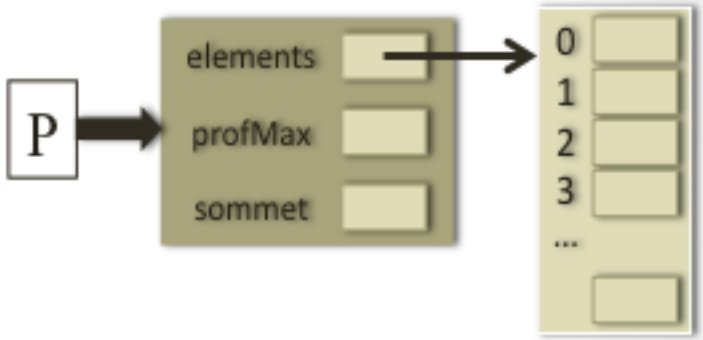
PILE par cellules contiguës

- Structure de données retenue :
  - Une pile est un pointeur sur une structure qui contient
    - L'indice du sommet de pile
    - La profondeur maximale de la pile
    - Un pointeur vers une zone dans le tas allouée pour pouvoir contenir un nombre d'éléments égal à la profondeur maximum



PILE par cellules contiguës

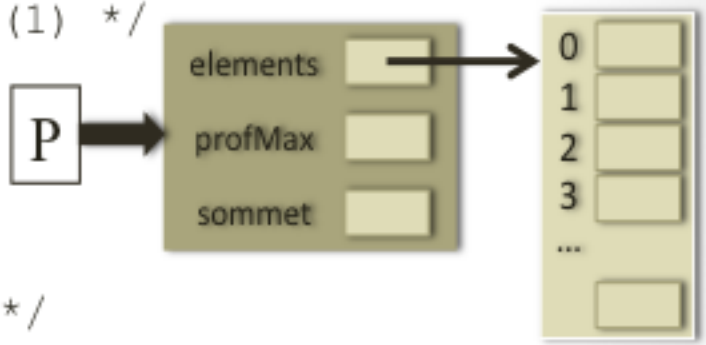
- Définition de la structure de données
  - ```
    typedef struct {
      ELEMENT *elements ;
      int sommet ;
      int profMax ;
    } pile,*PILE ;
    ```



- Définition des primitives
  - ```
    PILE PileCreer (int profondeur) {
      /* renvoie l'adresse d'une structure dont le champ
         éléments est un pointeur sur une zone mémoire de
         profondeur taille(ELEMENT) */
    PILE p;
    p=(PILE)malloc(sizeof(pile));
    if (! p) { printf("pb de mémoire") ; exit(0) ;}
    p->elements=(ELEMENT*)malloc(profondeur*sizeof(ELEMENT));
    p->sommet=0 ;
    p->profMax=profondeur ;
    return(p) ;
    }
    ```

PILE par cellules contiguës

- ```
  void PileDetruire(PILE P) { /* en o(1) */
    free(P->elements);
    free(P);
  }
  ```

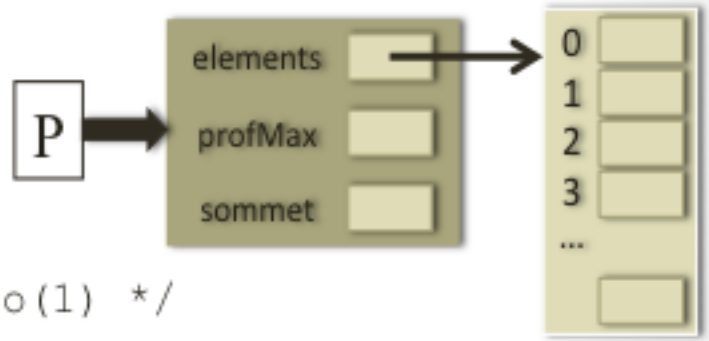



- ```
  bool PileVide(PILE P) { /* en o(1) */
    return(P->sommet==0);
  }
  ```

- ```
  ELEMENT PileDepiler(PILE P) {  /* en o(1) */
    ELEMENT elt ;
    if (!PileVide(P)) {
       ElementAffecter(&elt,(P->elements)[P->sommet]);
       P->sommet--;
       return(elt);
    }
    else return (ElementCreer());
  }
  ```
  ```

PILE par cellules contiguës

- ```
  bool PileEmpiler (ELEMENT elt, PILE P) { /* en o(1) */
    if(P->sommet < P->profMax)  {
       (P->sommet)++;
       ElementAffecter(&(P->elements)[P->sommet], elt);
       return(VRAI);
    }
    else {
       printf("Pile pleine\n");
       return(FAUX);
    }
  }
  ```
- ```
  ELEMENT PileSommet(PILE P) { /* en o(1) */
    if (! PileVide(P))
       return( (P->elements)[P->sommet] );
    else return (ElementCreer());
  }
  ```
- ```
  void PileAfficher (PILE P) { /* en o(n) */
    int i ;
    for (i = 1 ; i <= P -> sommet; i++)
       ElementAfficher(P -> elements [i]) ;
  }
  ```


  ```

Le TDA FILE

- Définitions :
  - Une file est une liste de type FIFO (First in First out)
  - Une file est une liste où les insertions s'effectuent à la fin et les suppressions au début

- Métaphore : file d'attente
- Utilisations classiques
  - Parcours d'arbres par niveau

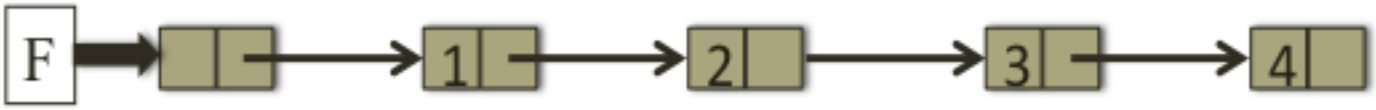
Les Primitives du TDA FILE

- `FFILE` **`FileCreer`** `(int profondeur) ;`
  /* crée et retourne une file vide en lui allouant dynamiquement de la mémoire */
- `void` **`FileDetruire`**`(FFILE);` /* libère la mémoire allouée pour la file */
- `int` **`FileVide`**`(FFILE);` /* teste si la file est vide */
- `ELEMENT` **`FileDebut`**`(FFILE);`
  /* retourne l'élément au début de la file sans le retirer de la file */
- `ELEMENT` **`FileSortir`**`(FFILE);`
  /* retourne l'élément au début de la file en le retirant de la file */
- `int` **`FileEntrer`**`(ELEMENT, FFILE);` /* place l'élément à la fin de la file */
- `void` **`FileAfficher`** `(FFILE ) ;` /* pour test et mise au point */
- `void` **`FileRaz`**`(FFILE) ;`

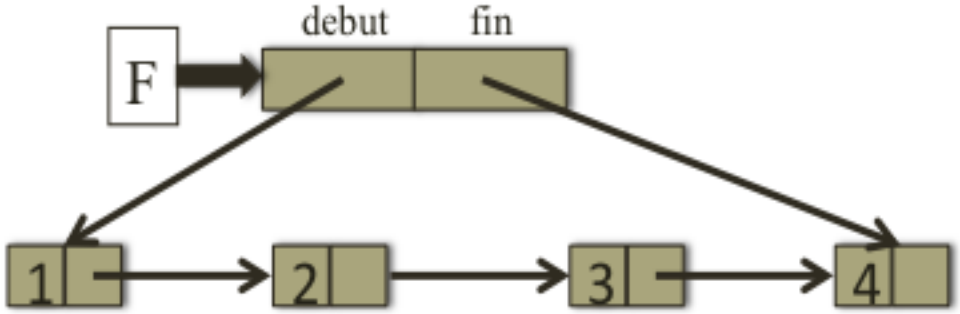
- **Remarque** : Comme pour les piles on peut réaliser les files à l'aide du TDA LISTE, mais on peut faire plus efficace.

FILE par cellules chaînées

- **Sol1** : une file est une liste (cf. TDA liste)
  - Entrer dans la file : insérer à la fin de la liste en o(n) (parcours de toutes les cellules pour trouver la fin); c'est mauvais

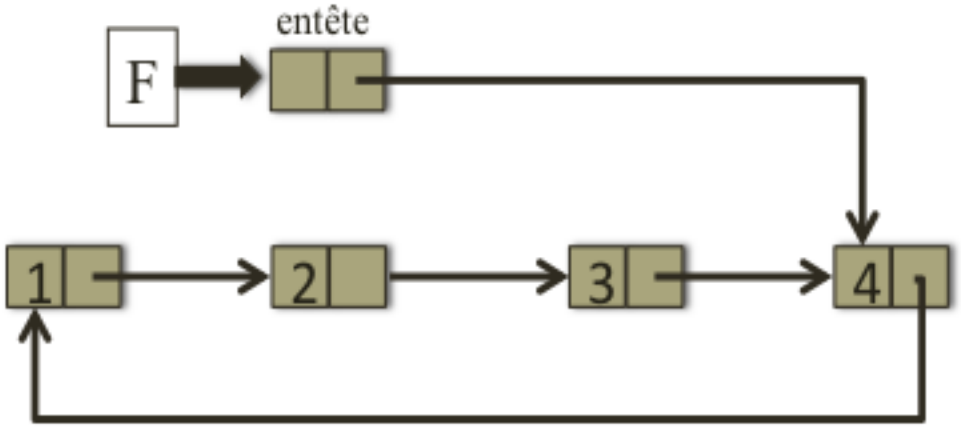


- **Sol2** : une file est un pointeur sur une structure
  - debut : pointeur sur la première cellule de la liste
  - Fin : pointeur sur la dernière
  - Toutes les opérations (sauf détruire sont en o(1))



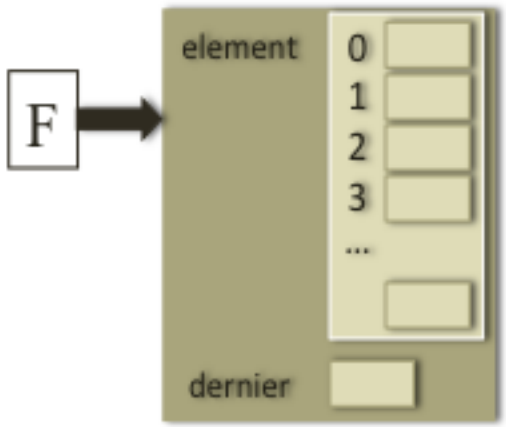
FILE par cellules chaînées

- **Sol3** : liste circulaire,
  une file est un pointeur sur une cellule d'en-tête dont le champ suivant pointe sur la dernière cellule de la file ; le champ suivant de cette dernière pointe sur la première (cf. TD 3)
  - Toutes les opérations (sauf détruire sont en o(1))



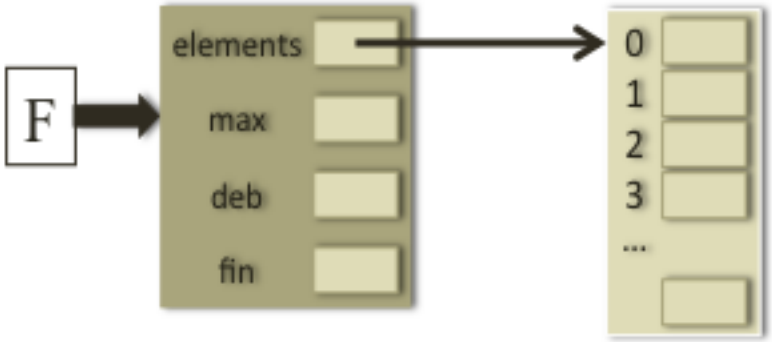
FILE par cellules contiguës

- **Sol 1** : une file est une liste (cf. TDA liste)
  - Sortir de la file : supprimer en tête de la liste en o(n) (à cause des décalages) ; c'est Mauvais



FILE par cellules contiguës

- **Sol 2** : tableau circulaire
  - Les éléments de la pile sont dans un tableau entre l'indice début et l'indice fin
  - Si fin devient supérieur à la taille du tableau, on remplit le tableau à partir de ces premières cases
  - Astuce pour distinguer une file vide d'une file pleine
    - File vide : `fin = (debut -1) % LongMax`
    - File pleine : `fin = (debut -2) % LongMax`



FILE par cellules contiguës

- ```
  typedef struct {
       ELEMENT *elements ;
       int debut ;
       int fin ;
       int longMax ;
   } file, *FFILE ;
  ```


- ```
  FFILE FileCreer (int profondeur){
  ```
  /* renvoie l'adresse d'une structure dont le champ éléments est un pointeur sur une zone mémoire de profondeur*taille(ELEMENT) */

```
  FFILE p;
  p=(FFILE)malloc(sizeof(file));

  if (! p) { printf("pb de mémoire") ; exit(0) ; }

  p->elements=(ELEMENT*)malloc(profondeur*sizeof(ELEMENT));
  p->debut= 0;
  p->fin = profondeur - 1 ;
  p->longMax= profondeur    ;
  return(p);
}
```