* Implémentons d’une file avec une liste
  + On peut voir la file comme un cas particulier d’une liste.
  + Enfile //insère à la fin
  + Defile //premier + getElement() + supprime
  + estVide //estVide
  + Vider //supprimeTout
  + DefileSansEnlever //premier + getElement()
* Structure données abstraite (TDA)
  + Définition
  + Spécification vs implémentation
  + File
  + Pile

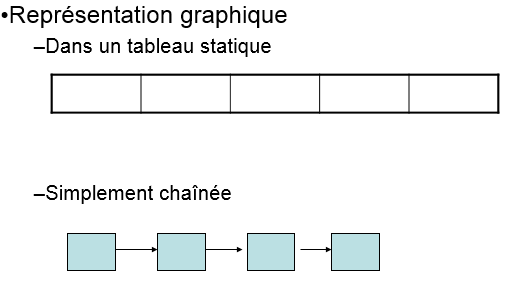
**Structure de données**

* Structure de données :
  + Conteneur de données du même type.
* Type de données abstrait
  + Une structure de données qui utilise l’encapsulation
* Deux aspects
  + Services offerts (spécification des méthodes)
  + Implémentation

**Implémentation**

* + Tableau statique
    - Le nombre d’éléments maximum est limité.
    - L’espace mémoire est réservé totalement à la compilation.
  + Chaînage dynamique
    - Le nombre d’éléments est illimité.

L’espace mémoire est réservé à l’exécution sur demande



* File (FIFO; First In First Out)
  + Structure de données où on insère à la fin et on retire au début (dessus)
  + Même principe que la file d’attente dans la vie courante
    - File à la banque, à l’épicerie, …
  + Fortement utilisé en informatique
    - Imprimante, réseau, …
* **Service d’une File**
  + Enfile //Insère à la fin de la file
  + Defile //Enlève et retourne le premier de la

//file

* + estVide //retourne si la file est vide
  + Vider //Enlève tous les éléments
  + DefileSansEnlever //Retourne l’élément du dessus

//sans l’enlever de la file

2.2 **Les files**

Dans le cas d'une file on fait les adjonctions à une extrémité, les accès et les suppressions à l'autre extrémité.

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.

Les files sont aussi appelées FIFO pour First-In-First-Out, c.à.d premier-entré-premier-sorti. Les opérations sur les files sont : tester si la file est vide ; accéder au premier élément de la file ; ajouter un élément dans la file ; retirer le premier élément de la file.

2.2.1 Représentation contiguë des files

Dans ce cas on doit conserver l'indice i du premier élément et l'indice j de la première case libre après la file. On fait progresser ces indices modulo la taille *lmax* du tableau. Le seul point délicat est la détection des débordements.

|  |  |  |
| --- | --- | --- |
| partie vide | éléments de la file | partie vide |

0 i j lmax-1 i < j

|  |  |  |
| --- | --- | --- |
| de la file | partie vide | éléments |

0 j i lmax-1

i > j

|  |  |
| --- | --- |
| de la file | éléments |

0 i=j lmax-1

i = j Tableau est plein

typedef struct file

{

T donnée[n] ;

unsigned tête ;

unsigned fin ;

}file ;

2.2.2 Représentation chaînée des files

Dans le cas d'une représentation chaînée, soit on a deux pointeurs, tête et dernier, vers le premier et le dernier élément de la file, soit on utilise le pointeur qui suit le dernier élément pour repérer le premier élément. On a donc une représentation circulaire.

Manipulation d'une file (méthode avec 2 pointeurs)

**Définition de file en C**

typedef struct poste

{

T info;

struct poste\* suivant;

} poste;

typedef poste\* ptr\_poste;

typedef struct file

{

ptr\_poste tête;

ptr\_poste fin;

int nb\_elt; /\* pas indispensable mais conseillé \*/

}file;

1. Initialisation d'une file juste déclarée

Résultat de type file

Entête : file init( )

Algorithme

{

v.l. file f ;

f.tete 🡨 NULL

f.fin 🡨 NULL

f.nb\_elt 🡨 0

return f

}

1. Tester si la file est vide

Donnée : file f

Résultat : de type booléen

Entête en C : int file\_vide (file f)

Algorithme file vide

{

v.l. booléen vide

SI f.nb\_elt=0 ALORS

vide 🡨 vrai

SINON

vide 🡨 faux

return vide

}

3) Mettre un élément à la fin de la file (enfiler)

Donnée : T elt\_à\_enf

Donnée modifiée : file \*ff

Entête void enfiler(T elt\_a\_enf, file \*ff);

Algorithme enfiler

{

v. l. ptr\_poste courant

reserver(courant)

courant🡪info 🡨 x

courant🡪suivant 🡨 NULL

SI ff🡪tete = NULL ALORS

{

ff🡪tete 🡨 courant

ff🡪fin 🡨 courant

}

SINON

{

ff🡪fin🡪suivant 🡨 courant

ff🡪fin 🡨 courant

}

ff🡪nb\_elt 🡨 ff🡪nb\_elt + 1

}

)

4) Retirer un élément de la tête de la file (défiler)

Donnée modifiée : file \*ff, T \*elt\_def

Résultat : de type booléen

Entête  : int defiler(file\* ff, T \*elt\_dep);

Algorithme

{v. l. ptr\_poste courant, booléen ok

SI non file\_vide(\*ff) ALORS

{

ok 🡨vrai

\*elt\_def 🡨 ff🡪tete🡪info

ff🡪nb\_elt 🡨 ff🡪nb\_elt – 1

courant🡨 ff🡪tete /\* cette opération permettra de libérer l'espace mémoire \*/

ff🡪tete 🡨 ff🡪tete🡪suivant

liberer(courant)

SI ff🡪nb\_elt = 0 ALORS

ff🡪fin 🡨NULL /\* si la file contenait un seul élément, elle devient vide, et ff🡪fin devient NULL, ainsi que ff🡪tete \*/

}

SINON

ok 🡨 faux

}

return ok

}

Rmq : Une file se représente beaucoup mieux en dynamique (représentation chaînée).

Utilisation des files : tous les exemples où l'on dispose d'une ressource et l'où on souhaite :

* que plusieurs programmes l'utilisent
* que plusieurs données soient manipulées par cette ressource

Ex. Le buffer d'imprimante est géré en file d'attente

3.3 Exemple de gestion d'un buffer clavier en liste circulaire (le dernier poste d'une liste circulaire est rattaché au premier)

On a un ensemble des données homogènes gérées sur un principe de lecture/écriture concurrente : lecture et écriture peuvent se produire de manière simultanée mais sont conditionnées par l'autre opération.

Une lecture ne peut être faite que si une écriture a déjà été faite, mais une écriture ne peut être faite que si la lecture a été effectuée sous peine de perdre l'information.

Plus généralement : il s'agit d'une file d'attente particulière où l'on a pas besoin de produire toutes les données pour commencer à les consommer.

Le nombre de postes est fixé au départ et ne varie plus au cours du programme.

On sait qu'on consomme en moyenne un nombre de postes à chaque fois qu'on en produit.

lecteur écrivain

Il y en a deux indicateurs : lecteur et écrivain

Il faut écrire deux algorithmes :

* lire : c'est prendre l'information référencée par lecteur, à condition que cela soit possible et déplacer lecteur sur le poste suivant
* écrire : c'est mettre une information référencée par écrivain si cela est possible, et déplacer écrivain sur le poste suivant

Rmq. On ne peut pas lire si rien n'a été écrit ou si l'information a été déjà lue.

On ne peut pas écrire, si le poste contient une information encore non-lue.

Exemple

écrivain

**.**

**:**

**a**

**y**

**p**

**o**

**c**

lecteur

C\> copy a: .

le fait de taper *RC* entame la lecture

On tape *dir* très rapidement

**d**

**.**

**:**

**a**

**y**

**p**

**r**

**i**

écrivain lecteur

Modèle statique

typedef struct elt

{

T info ;

booléen àlire ;

} elt ;

àlire = vrai : on peut lire, mais pas écrire

àlire = faux : on peut écrire, mais pas lire

àlire est un flag (drapeau).

typedef struct liste\_circ

{

elt donnée [n] ;

int lecteur ;

int écrivain ;

}liste\_circ

Donnée : int n

Donnée modifiée liste\_circ \*l

Entête : void init(liste\_circ \*l);

{ v.l. int i

POUR i 🡨 0 à n-1 FAIRE

l🡪donnée[i].àlire 🡨 faux

l🡪lecteur 🡨 0

l🡪ecrivain 🡨 0

}

Donnée : int n

Donnée modifiée : T \*elt\_à\_lire, liste\_circ \*l

Résultat de type booléen

Entête : int lecture(liste\_circ \*l, T \*elt\_à\_lire)

{

SI l🡪donnée[l🡪lecteur].àlire = faux ALORS

ok 🡨 faux

SINON

{

ok 🡨 vrai

\*elt\_à\_lire **🡨** l🡪donnée[l🡪lecteur].info

l🡪donnée[l🡪lecteur].àlire 🡨 faux

l🡪lecteur 🡨 (l🡪lecteur + 1) mod n

}

return ok

}

Donnée : int n, T elt\_ecr

Donnée modifiée liste\_circ \*l

Résultat de type booléen

Entête : int écriture(T elt\_ecr, liste\_circ \*l);

{

SI l🡪donnée[l🡪ecrivain].àlire = vrai ALORS

ok 🡨 faux

SINON

{

ok 🡨 vrai

l🡪donnée[l🡪ecrivain].info 🡨 elt\_ecr

l🡪donnée[l🡪ecrivain].àlire 🡨 vrai

l🡪ecrivain 🡨 (l🡪ecrivain + 1) mod n

}

return ok

}

Modèle dynamique

typedef struct elt

{

T info ;

booléen àlire

struct elt \*suivant

} elt

typedef ptr\_elt \*elt

typedef struct liste\_circ

{

ptr\_elt lire

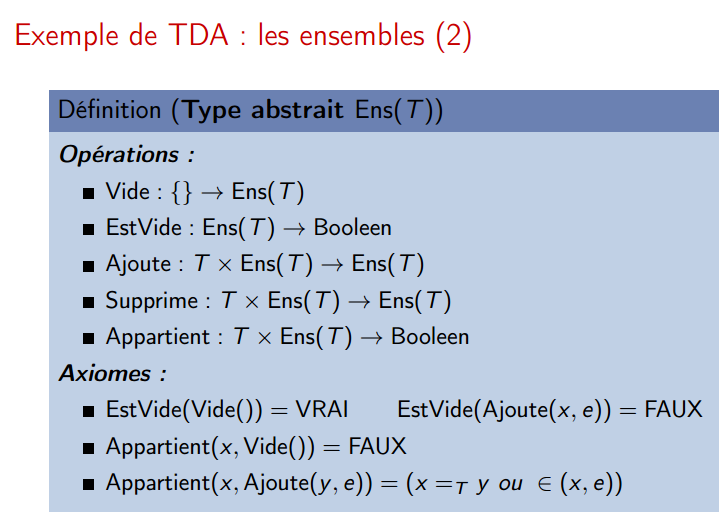
ptr\_elt ecrire

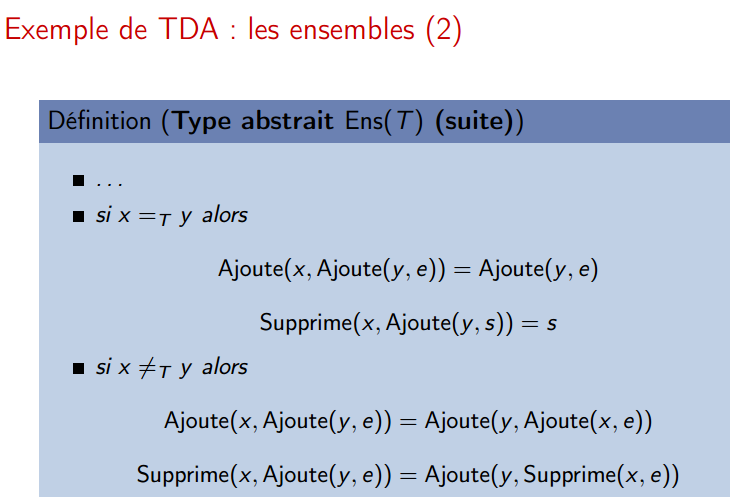
} liste\_circ ;

Rmq. Rien dans les types n'implique qu'après le dernier on passe sur le premier, ceci devra être dit dans les algorithmes.

En statique il s’agit de l’opération modulo N

En dynamique il s’agit du passage au suivant





1. **FILES**

Les files constituent un autre cas particulier des listes où les éléments sont insérés à une extrémité (la queue de la file) et la suppression se fait à l’autre extrémité (la tête de la file). Les files sont aussi appelées structures FIFO (First In First Out) c'est-à-dire « le premier qui arrive c’est le premier qui sort ». Les opérations classiques pour les files sont similaires à celles pour les piles sauf que cette fois-ci les insertions se font à la queue de la file au lieu de la tête. Il convient aussi de changer la terminologie précédente (utilisée pour les piles) dans le cas des files. Les opérations classiques sur les files peuvent être décrites comme suit :

* + **InitialiserFile**  : permet de créer une file vide.
  + **TeteDeFile** : retourne le premier élément de la file.
  + **QueueDeFile** : retourne le dernier élément de la file.
  + **Enfiler** : insère un nouvel élément à la fin de la file (EnQueue).
  + **Defiler** : supprime le premier élément de la file (DeQueue).
  + **FileVide** : renvoie « vrai » si la file est vide et « faux » sinon.

**III.1. Implémentation d’une file à l’aide d’un tableau**

Pour des raisons d’efficacité, une file est généralement implémentée à l’aide d’un tableau « circulaire ». On garde l’indice de tete et de queue de file. Si l’un deux « arrive » à la fin du tableau, il retourne à l’indice de départ.

**Exercice :** Proposer une implémentation d’une file à l’aide d’un tableau circulaire et donner les détails des routines correspondantes.

**III.2. Implémentation d’une file à l’aide d’une liste chaînée**

Comme pour les piles, les routines d’implémentation d’une liste à l’aide d’une liste chaînée restent valables sur les files qui peuvent être considérées comme un cas particulier des listes. Cependant, on peut utiliser le fait que les insertions se font toujours à la queue de la liste pour rendre la procédure ***Enfiler*** plus efficace et aboutir à des économies en termes de temps. En effet, au lieu de parcourir la liste du début jusqu'à la fin à chaque fois qu’on veut ajouter un élément, c'est-à-dire enfiler un élément, on peut garder un pointeur sur la queue de la liste (dernier élément).   
Bien entendu, on doit aussi garder un pointeur sur la tête de celle-ci qui est utilisé pour la fonction ***TeteDeFile*** et aussi la procédure ***Defiler***.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | TeteDeFile | 15 | 39 | 11 |  | QueueDeFile |
|  |  | ←⎯⎯⎯⎯⎯⎯ | | |  |  |

↓

Tête de File

Queue de File

15

39

11

null

Les détails d’implémentation peuvent être décrits comme suit :

**Applications**

L’application la plus classique est la file d’attente, et elle sert beaucoup en simulation. Elle est aussi très utilisée aussi bien dans la vie courante que dans les systèmes informatiques. Par exemple, elle modélise la file d’attente des clients devant un guichet, les travaux en attente d’exécution dans un système de traitement par lots, ou encore les messages en attente dans un commutateur de réseau téléphonique. On retrouve également les files d’attente dans les programmes de traitement de transactions telle que les réservations de sièges d’avion ou de billets de théâtre.

