Cours: ASD 2 & Complexité

Chapitre 6: Les Files

Réalisé par:
Dr. Sakka Rouis Taoufik

I. Introduction

On appelle file d'attente (ou tout simplement file) un ensemble formé d'un nombre variable, éventuellement nul de données, sur lequel les opérations suivantes peuvent être effectuées :

- creer_file : permet de créer une file vide (création).
- file_vide : permet de tester la vacuité d'une file, ou si la file est vide ou non (consultation).
- enfiler : permet d'ajouter une donnée de type T à la file (modification).
- defiler : permet d'obtenir une nouvelle file (modification).
- premier : permet d'obtenir l'élément le plus ancien dan la file (consultation).

Ch 6: Les Files

I. Introduction

OPERATIONS ILLEGALES :

Il y a des opérations définies sur la SD file d'attente exigent des pré conditions :

- défiler exige que la file soit non vide.
- premier exige que la file soit non vide.

3

Ch 6: Les Files

II. Propriétés

Dans ce paragraphe on va citer (énumérer) les propriétés qui caractérisent la sémantique des opérations applicables sur la SD file d'attente.

- F1 : creer_file permet de créer une file vide.
- F2 : si un élément entré (enfiler) dans la file résultante est non vide.
- F3 : un élément qui entre (grâce à enfiler) dans la file d'attente devient immédiatement le premier : si la file vide sinon (file non vide) le premier reste inchangé.
- F4 : une entrée et une sortie successive sur une file vide la laissent vide.
- F5 :Une entrée et une sortie successive sur une file non vide peuvent être effectuées dans n'importe quel ordre.

4

Ch 6: Les Files

II. Propriétés

REMARQUE : ces cinq propriétés permettent de cerner la sémantique (comportement ou rôle) des opérations applicables sur la SD File.

En conclusion : La structure de File obéit à la loi FIFO : First In, First Out.

5

Ch 6: Les Files

III. Représentation physique

Pour pouvoir matérialiser (concrétiser, réaliser ou implémenter) une SD on distingue deux types de représentation :

- **Représentation chaînée** : les éléments d'une SD sont placés à des endroits quelconques (bien entendu dans la MC) mais chaînés (ou reliés).

  Idée : pointeur.

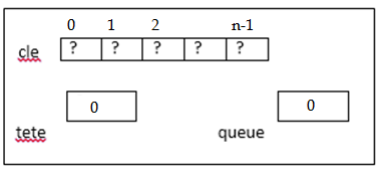
- **Représentation contiguë** : les éléments d'un SD sont rangés (placés) dans un espace contiguë.

  Idée : tableau.

6

TDA FILE concrétisé par une représentation contiguë

Il s'agit d'un tableau à deux points d'entrée : deux indices tête et queue.




```
define n 100
struct file {
        int cle[n] ;
        unsigned tete;
        unsigned queue;
};
```


➔ **Opération ou services exportés :**

/*Partie interface : file.h */

```
define n 100
struct file {
        int cle[n] ;
        unsigned tete;
        unsigned queue;
};
```

```
/*opération ou services exportés*/
void creer_file (struct file*) ;
/* ou bien    struct file* creer_file (void) ;  */
unsigned file_vide  (struct file) ;
/* ou bien unsigned file_vide (struct file *) ; */
int premier(struct file);
/* ou bien   int premier(struct file *);   */
void enfiler (int,struct file*);
void defiler (struct file*);
```

➔**Opération ou services exportés :**

/* Implémentation :file.c */

```
#include<assert.h>
#include "file.h"

void creer_file (struct file * f) {
        f→tete=0 ;
        f→ queue=0 ;
}

unsigned file_vide (struct file f) {
        return (f.tete==f.queue);
}
```



```
void enfiler (int info, struct file * f) {
        assert (f-> queue < N)
        f->cle [f->queue]=info ;
        f-> queue++;
}
void defiler (struct file *f) {
        assert (! file_vide (*f) );
        f -> tete++ ;
 }
int premier (struct file f) {
        assert (! file_vide (f) );
        reteurn (f.cle [ f.tele] );
 }
```

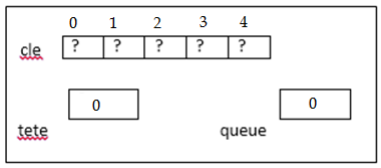
Ch 6: Les Files

III. Représentation physique

3.1 Représentation contiguë

➔Un problème posé par la représentation contiguë de la SD file :

Question : En partant de cette file vide (de taile n=5), exécuter la séquence suivante :



a) enfiler les éléments : 100 puis 20 puis 30 puis 40 puis 13.
b) defiler
c) enfiler 120 ;

11

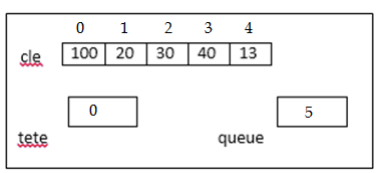
Ch 6: Les Files

III. Représentation physique

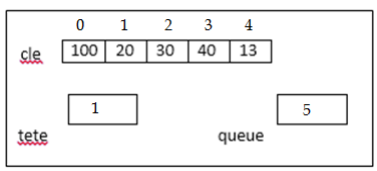
3.1 Représentation contiguë

➔Un problème posé par la représentation contiguë de la SD file :

a) Situation après les 5 enfilements



b) Situation après le défilement



c) L'état en position 0 est disponible, mais on ne peut pas enfiler de nouveau !!!

12

Ch 6: Les Files

III. Représentation physique

3.1 Représentation contiguë

➔Un problème posé par la représentation contiguë de la SD file :

Prob : On ne peut pas enfiler 120 après queue (en effet l'élément de position queue n'appartient pas au tableau clé. Et pourtant la file n'est pas pleine ??
Car le tableau est perçu d'une façon linéaire, il est parcourue de gauche à droite.

➔ Au bout de n enfilements, on ne peut plus ajouter des nouveaux éléments, **même si on fait des défilements**. Sachant que la valeur n est la taille du tableau cle.

13

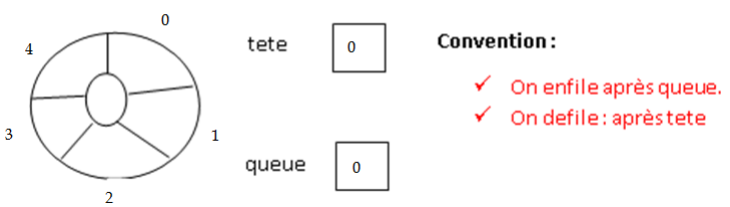
Ch 6: Les Files

III. Représentation physique

3.1 Représentation contiguë

➔Un problème posé par la représentation contiguë de la SD file :

Remède : un tableau circulaire c'est-à-dire : tete et queue **modulo n ;** avec n est la taille du tableau.
➔Il s'agit d'une perception logique et non physique.



14

Ch 6: Les Files

III. Représentation physique

3.1 Représentation contiguë

➔Un problème posé par la représentation contiguë de la SD file :

On va appliquer la séquence des actions a, b et c vue précédemment sur un tableau sur perçu d'une façon circulaire.

a)

enfilement 100 → queue=1
enfilement 20 → queue=2
enfilement 30 → queue=3
enfilement 40 → queue=4
enfilement 13 → queue+1 =5 ; or 5 > n-1 donc ➔ queue= 5 mod 5 = 0

b)

defiler → tete=1

c)

enfiler 120 → queue+1=0+1=1➔cette position est disponible.

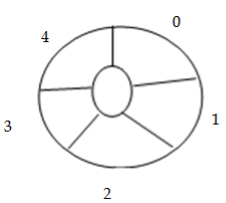
15

Ch 6: Les Files

III. Représentation physique

3.1 Représentation contiguë

Constatation :



tete 1

queue 1

tete=queue →file vide ou file pleine ??

➔d'où un deuxième problème

16

III. Représentation physique

3.1 Représentation contiguë

➔**Idée :** Au lieu de réserver un tableau (ici cle) de n éléments, on prévoit un tableau de taille n+1, en respectant la propriété suivante :

➔On utilise au plus n éléments : lorsque la file contient n éléments, la file est logiquement pleine mais pas physiquement.

➔La proposition tete=queue caractérise une file vide.

III. Représentation physique

3.1 Représentation contiguë

```
/*implementation :file.c */

#include<assert.h>
#include " file.h "


void creer_file (struct file*f){
        f→tete=0 ;
        f→ queue=0 ;
}

/*  RQ : n'importe quel indice
 compris entre 0 et n-1
pourra faire l'affaire.  */
```

```
unsigned file_vide (struct file f) {
        return  (f.tete==f.queue);
}

int premier (struct file f) {
        unsigned i;
        assert (!file_vide (f));
        i=f.tete+1 ;
        if (i>n-1)
        i=0;
        return(f.cle[i]);
}
```

Ch 6: Les Files

III. Représentation physique

3.1 Représentation contiguë

```
void enfiler (int info, struct file*f) {
        assert(f→tete != ((f→queue+1)%n)) ;
        /* au – deux cases vides */
        f→ queue++;
        if (f→queue > n-1)
        f→queue=0;
        f→cle[f→queue]=info ;
}
```

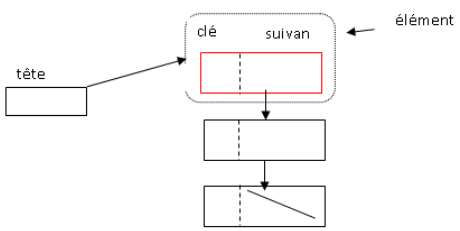
```
void defiler (struct file *f) {
        assert(!file_vide(*f));
        f→tete++ ;

        if (f→tete > n-1)
        f→tete=0 ;
}
```

19

Ch 6: Les Files

III. Représentation physique

3.1 Représentation chaînée



- **premier** : coût une indirection en partant du pointeur tête
- **defiler** : coût une indirection en partant du pointeur tête
- **enfiler** : coût il faut parcourir toute la file en partant du pointeur tête.

➔ Ceci nécessite plusieurs indirections. Ainsi, **il ne faut pas retenir la solution proposée**

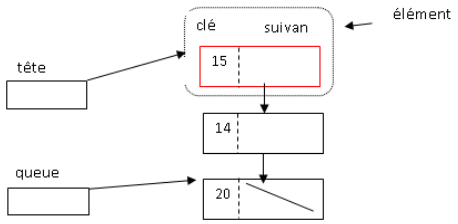
20

Ch 6: Les Files

III. Représentation physique

3.1 Représentation chaînée

Remède : on a besoin d'une représentation physique à deux points d'entrées : tête et queue.



- Le pointeur **tête** favorise l'implémentation efficace des opérations : **premier** et **defiler**.
- Le pointeur **queue** favorise l'implémentation efficace de l'opération **enfiler**.

Remarque: La SD file d'attente est structurée à deux points d'entrée. Par contre la SD pile est une structure à un seul point d'entrée.

21

Ch 6: Les Files

IV. Matérialisation de la SD File

Matérialisation de la SD File :
- **Objet abstrait (OA)**
- **Type de données Abstrait (TDA)**

- **Objet abstrait (OA)**→un seul exemplaire (ici une seul File)→ unique →implicite.
- **Type de données Abstrait (TDA)** →plusieurs exemplaires → il faut mentionner ou rendre explicite l'exemplaire courant.

22

IV. Matérialisation de la SD File

4.1 Structure de données File comme TDA

Dans cette partie, on va concrétiser la SD File sous forme d'un **Type de données Abstrait** (TDA) capable de gérer plusieurs exemplaires de la SD File et non pas un seul exemplaire.

IV. Matérialisation de la SD File

4.1 Structure de données File comme TDA

```
struct element {
        int cle ;
        struct element * suivant ;
};
struct file {
        struct element * tete ;
        struct element *queue ;
};

void cree_file (struct file * f){

        f->tete  =NULL ;
        f->queue=NULL;
}
```

```
int file_vide ( struct file *f ) {

    return  ( (f->queue)==NULL) &
              (f->tete)==NULL) ) ;

}


 int  premier (struct file *f ) {

        assert (! file_vide (f) ) ;
        return  f->tete-> cle ;
  }
```

IV. Matérialisation de la SD File

4.1 Structure de données File comme TDA

```
void enfiler (  int x , struct file * f ) {
        struct element * p ;
        p= (struct element * ) malloc (sizeof ( struct element )) ;
        p -> cle = x ;
        p->suivant = NULL  ;

        if ( file_vide (f) ) {
                f->tete = p ;
                f->queue = p ;
        }
        else {
                f->queue->suivant =p ;
                f->queue  =p ;
        }
}
```

Remarque : f->queue=p est exécutée systémiquement ou encore d'une façon inconditionnelle par conséquent elle ne dépend pas du schéma conditionnel.
Elle peut être simple comme suit :

```
if ( file_vide (f))
    f->tete=p ;
else
   f->queue->suivant=p ;
f->queue=p ;
```

25

IV. Matérialisation de la SD File

4.1 Structure de données File comme TDA

```
void defiler ( struct file * f ) {

        struct element * q ;
        assert (! file_vide (f) ) ;

        q = (f->tete) ;
        f->tete = f->tete->suivant ;

        free (q) ;
        if ( f-> tete == NULL ) /* un seul élément dans la file */
        f->queue = NULL ;

}
```

26

V. Exercices d'application

Exercice 1:

Implémenter la méthode« void defilerJusquaElem (struct file *f , int x) » permettant de défiler les éléments de la file jusqu'à attendre l'élément x ou bien jusqu'à la fin des éléments.

NB. L'élément x n'est pas défilé.

Exercice 2:

Montrer comment implémenter une file à partir de deux piles. Analyser le temps d'exécution des opérations de file.

V. Exercices d'application

Exercice 3:

On donne un ensemble d'éléments dont le nombre n'est pas connu a priori et variable (à cause des adjonctions et suppressions). Les éléments de cet ensemble sont classés en m catégories (m est connu a priori et fixe). Les éléments appartenant à une catégorie donnée sont gérées selon la stratégie FIFO. On vous demande de concevoir et réaliser en C un module permettant de gérer un tel ensemble.

V. Exercices d'application

Solution Exercice 3:

```
/* Partie Interface :ens.h  */
#define n 10
/* representation phisique File */
struct element {
        int cle ;
        struct element *suivant ;
} ;
struct file {
        struct element *tete ;
        struct element *queue ;
}
```

```
/* services exportés */
void creer( struct file * []) ;

void ajouter(struct file *[], unsigned , int) ;
/*le second paramètre indique la
catégorie et le troisième l'information à
ajouter à cette catégorie */

void supprimer( struct file*[], unseigned) ;
/* le second paramètre indique la
catégorie */
```

29

V. Exercices d'application

```
/* Partie Implementation :ens.c  */
/* on reutilise les services offerts par file.h */
#include "ens.h"
#include "file.h"

void creer( struct file * t [ ]) {
        unsigned i ;
        for(i=0 ; i<m ; i++)
                creer_file ( t [ i ] ) ;
}
void ajouter(struct file * t [ ], unsigned categorie, int x) {
        enfiler (x, t [categorie] ) ;
}
void supprimer(struct file * t [ ], unsigned categorie) {
        defiler ( t[categorie] ) ;
}
```

30