Travaux Pratiques N°14

Les Files

1. Définition

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 : 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).

Opérations illégales : 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.

2. 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.

Illustration : File non vide : 5 6 1
Cas 1 : 5 6 1
enfiler 8 ->5 6 1 8
defiler->6 1 8
Cas 2 : 8 6 1
defiler -> 6 1
enfiler 8->6 1 8

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

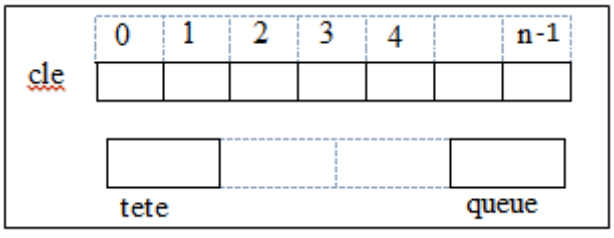
3. Représentation physique

On distingue deux types de représentations physiques :

-représentation contiguë
-représentation chaînée

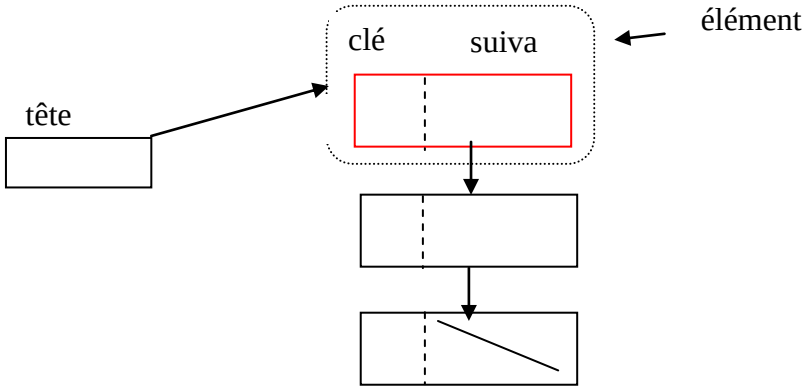
3.1 Représentation contiguë

TDA FILE concrétisé par une représentation contiguë
Il s'agit d'un tableau à deux points d'entrée : deux indices tête et queue.



Constante N 100
Type
File= struct
Cle :tableau de N entier
Tete :entier
Queue : entier
FinStruct

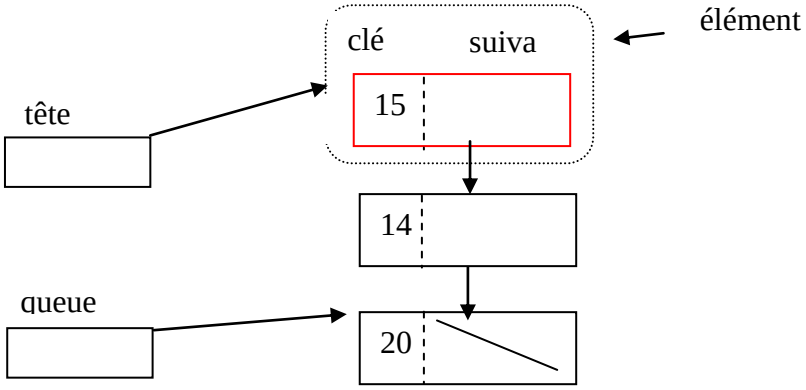
3.2 Représentation chaînée



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**
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.



Traduction en C :

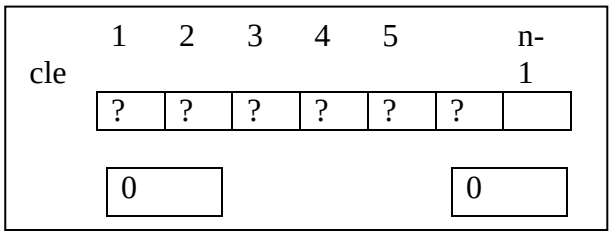
```
struct element {
  int cle ;
  struct element * suivant ;
};
struct file  {
  struct element* tete ;
  struct element* queue ;
};
```

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.

4. Matérialisation de la SD File sous forme d'un TDA file concrétisé par une représentation contiguë

Partie interface : file.h
/*représentation chaînée*/
define n 100
struct file {
int cle[n] ;
unsigned tete;
unsigned queue;
};
	/*opération ou services exportés*/
	void creer_file (struct file*) ;
	/* struct file* creer_file (void) ;/*/
	unsigned file_vide (struct file) ;
	/* unsigned file_vide (struct file *) ; */
	int premier (struct file);
	/* int premier (struct file *);*/
	void enfiler (int, struct file*);
	void defiler (struct file*);/
	Partie implémentation: file.c
	…

Le problème posé par la représentation contigüe de la SD file :

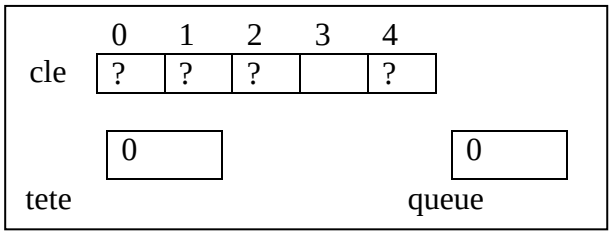


Question : En partant de cette file vide, exécuter la séquence suivante :

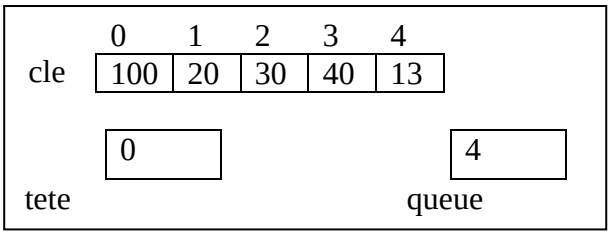
enfiler les éléments : 100, 20, 30, 40, 13 ;
defiler
enfiler 120

Résultat :

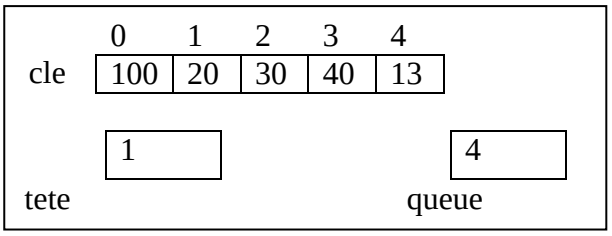
a) situation initiale



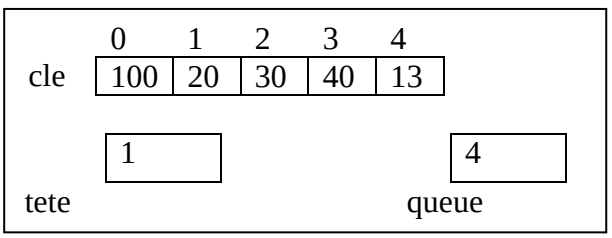
b) Situation après les 5 enfilements



c) situation après le défilement



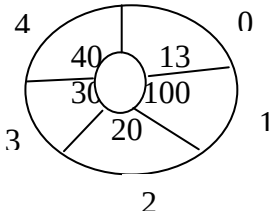
d) Situation finale
L'état en position 0 est disponible



On ne peut enfiler 120 après queue (en effet l'élément de position queue+1(4+1=5) 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 n est la taille du tableau cle.

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

Situation initiale : file vide



tete 0

queu 0

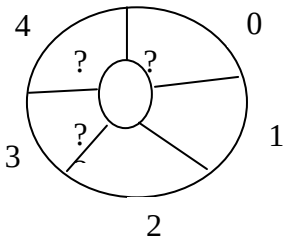
Convention : on enfile après queue.
On defile : après tete

enfilement 100 queue=1
enfilement 20 queue=2
enfilement 30 queue=3
enfilement 40 queue=4
enfilement 13 (queue +1> 4)➔queue=0

defiler➔100 tete=1

enfiler 120 queue+1=0+1=1➔cette position est disponible. L'élément dans cette position est déjà traitée en (b))

Constatation :



tete 0

queue 0

tete=queue →file vide ou file pleine

➔d'où un problème

➔Il s'agit d'une proposition ambiguë

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.

Implémentation :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]);
}

void enfiler (int info, struct file *f) {
        f -> queue++;

        if (f -> queue>n-1)
        f -> queue=0;

        assert (f -> tete!=f ->  queue) ;
        f -> cle[f -> queue]=info ;
}
void defiler(struct file *f) {
        assert (!file_vide(*f));
        f-> tete++ ;

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

Exercice d'application

Exercice 1 :

Matérialiser en C la structure de données File sous forme d'un TDA (Type de Données Abstrait) en adoptant une représentation chaînée.

****

4Matérialisation de la SD File comme type de donnée abstrait

```
/*représentation chainée*/
Types
        Cellule = Struct
        cle : entier
        Suiv : ^Cellule
        FinStruct

        File = Struct
        tete : ^Cellule
        queue : ^Cellule
        FinStruct
Procedure creer_File (var F :File )
Début
        F.tete←Nil
        F.queue←Nil
Fin Proc

Fonction File_vide (F :File): boolean
Debut
        File_vide←( F.queue=Nil)
fin Fn

Procédure enfiler (x : Entier ; Var F : File)
Var
        P : ^Cellule
Début
        Allouer(P)
        P^.cle← x
        P^.Suiv←Nil

        Si File_vide(F) alors
                F.tête← P
                F.queue← P
        sinon
                F.queue^.Suiv←P
                F.queue← P
        FinSi
Fin Proc
```

```
Procédure défiler (Var F : File)
Var
        P : ^Cellule
Début
        Assure (non File_vide(F))
        P←F.Tête
        F.Tête←F.Tête^.Suiv
        Libérer(P)
        Si F.Tête = NIL alors
           F.queue ← NIL
        FinSi
Fin Proc

Fonction premier (F : File) : entier
Début
        Si (File_vide(F)) Alors
        Ecrire(" impossible, la file est vide" )
        Sinon
        premier←F.tête^.cle
        FinSi
Fin Fn
```