Théorie des Langages - TP

Elena Tushkanova
elena.tushkanova@femto-st.fr

Les séances de TP de théorie des langages sont dédiées à la réalisation en `C` d'une application manipulant les automates finis. Il est pour cela indispensable de maîtriser les bases de la programmation `C` vue en L2, notamment les listes et les pointeurs. Pour plusieurs questions des indications (facultatives) sont données en annexe. Les codes donnés sont indicatifs et vous devez vous assurer de bien les comprendre !

Le nombre de séances (de 1h30) donné pour chaque partie est indicative. Vous devez cependant veiller à ne pas prendre de retard par un travail personnel, en dehors des heures de TP. Vous pouvez bien sûr aussi prendre de l'avance.

Un soin tout particulier devra être apporté à la réalisation de votre projet, notamment la mise en forme de votre code, la complexité des algorithmes proposés, le respect des indications données dans l'énoncé, et les commentaires associés à chaque partie de votre code. Il vous est également demandé d'utiliser les fichiers d'entête (fichier .h) de la programmation en `C`.

---

1 Mots et chaînes de caractères (1 séance)

Pour travailler sur les mots, nous allons utiliser la structure de *chaîne de caractères*. En `C`, une chaîne de caractères est un tableau de caractères `char` dont le dernier caractère est `'\0'`. Par exemple la chaîne de caractères *bonjour* sera codée par le tableau suivant :

b	o	n	j	o	u	r	\0

Une chaîne de caractères étant un tableau de caractères, elle se déclare naturellement comme un pointeur sur un caractère. De plus, de nombreuses fonctions sur les chaînes de caractères sont implémentées en `C` dans la bibliothèque `string.h`.

Que fait le programme suivant ? (Vous devez bien sûr également expliquer pourquoi il le fait).

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(){
        char* ch = malloc (12*sizeof(char));
        scanf("%s",ch);
        printf("vous avez tap\'e : %s\n",ch);
        while(*ch != '\0'){
                printf("%c\n",*ch);
                ch++;
        }
}
```

La bibliothèque `string.h` contient de nombreuses fonctions utiles. Par exemple, la fonction `strlen` calcule la longueur d'une chaîne et la fonction `strcat` concatène deux chaînes. Toutes ces fonctions gèrent parfaitement le caractère de fin de chaîne. Cependant, la gestion de la mémoire pour la taille des chaînes reste à faire *à la main*.

Voici un exemple de code utilisant les deux fonctions citées.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(){
        char* ch1 = malloc (24*sizeof(char));
        char* ch2 = malloc (12*sizeof(char));
        scanf("%s",ch1);
        scanf("%s",ch2);
        printf("la taille de ch1 est %d\n",strlen(ch1));
        printf("la taille de ch2 est %d\n",strlen(ch2));
        strcat(ch1,ch2);
        printf("maintenant la ch1 vaut %s\n",ch1);
}
```

- Écrire une fonction **concatAvecStringH** qui prend en entrée deux chaînes de caractères et qui les concatène en mettant la plus courte au début (si elles ont même taille, on les mettra dans un ordre quelconque). On utilisera pour ça les deux fonctions de `string.h` mentionnées ci-dessus.
- Écrire ensuite une fonction **concatSansStringH** qui réalise la même chose, sans utiliser cette fois-ci les fonctions de la biblithèque `string.h`.

2 Automates (2-3 séances)

Nous allons utiliser les structures de donnés suivantes pour coder un automate.

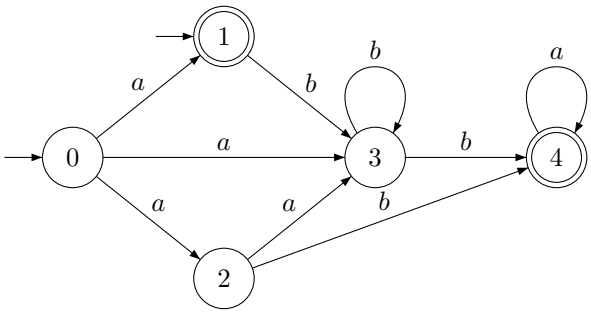
```
typedef struct s_liste {
        int state;
        struct s_liste* suiv;
} liste;


typedef struct {
         int size;
         int sizealpha;
         int* initial;
         int* final;
         liste*** trans;
} automate;
```

La première structure est une liste chaînée d'entiers. La seconde structure implante une reprśentation possible pour les automates. On trouve :

- le champ `size` donne le nombre d'états de l'automate. Par convention, les états seront obligatoirement numérotés de 0 à `size` - 1.
- le champ `sizealpha` donne la taille de l'alphabet. Par convention encore, on considèrera que les lettres sont les `sizealpha` premières lettres de l'alphabet en minuscule. Par exemple si `sizealpha` vaut 3, l'alphabet sera composé des lettres a, b et c.
- le champ `initial` est un tableau d'entiers de taille `size` qui contient à la i-ème case la valeur 1 si l'état i est initial, 0 sinon.
- le champ `final` est un tableau d'entiers de taille `size` qui contient à la i-ème case la valeur 1 si l'état i est final, 0 sinon.
- le champ `trans` est un tableau de taille `size` de `liste**`. La i-ème case de `trans` va coder toutes les transitions qui partent de l'état i. La case `trans[i]` contient donc un tableau de taille `sizealpha` de `liste*` qui contient la liste des états accessibles depuis i en lisant la $j+1$-ème lettre de l'alphabet. **Cette liste sera triée par ordre croissant et sans redondance**.

Considérons par exemple l'automate suivant :



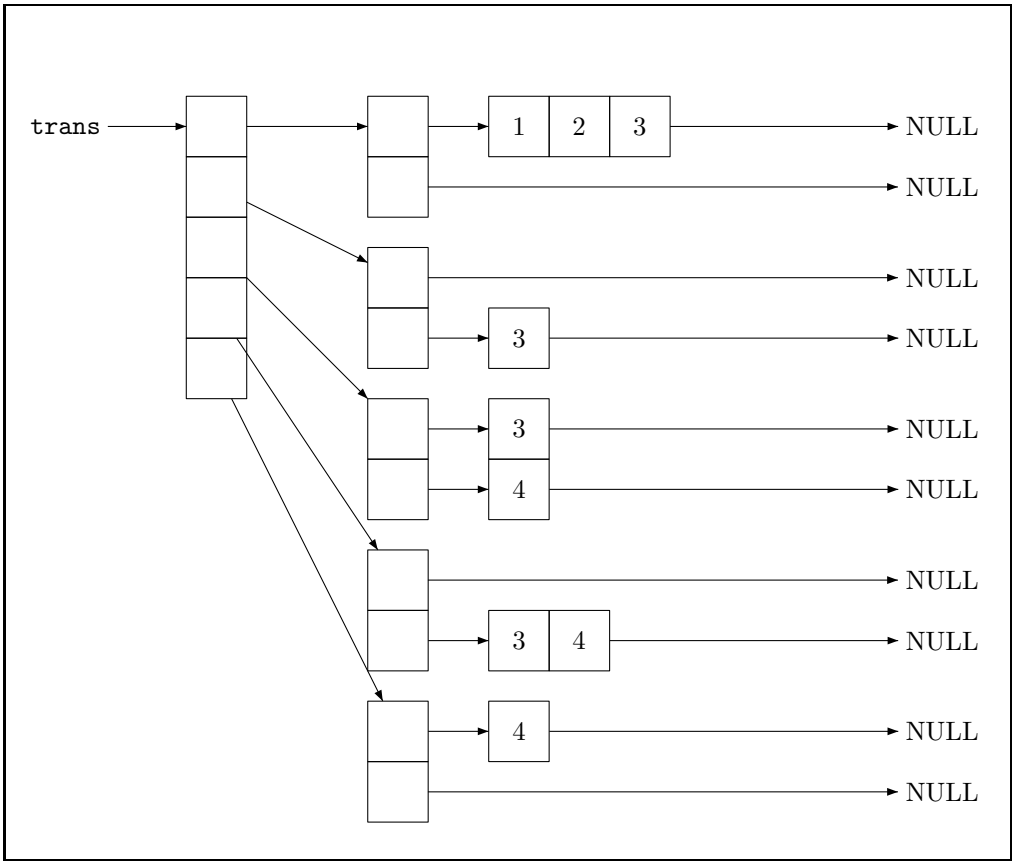
Il possède 5 états, le champ `size` devra donc contenir la valeur 5. On travaille avec les deux lettres a et b, donc le champ `sizealpha` aura la valeur 2. Il y a deux états initaux 0 et 1, le tableau `initial` sera donc

1	1	0	0	0

Il y a deux états finaux 1 et 4, le tableau `final` sera donc

0	1	0	0	1

Les transitions de cet automate seront codées comme suit :



- Ecrire une fonction **ajouteTransition** qui ajoute une transition à un automate. *On pourra pour cela (éventuellement) utiliser la fonction de la section 7.1 qui insère un entier dans une liste triée sans redondance.*
- Ecrire une fonction **construitAutomateExemple** qui retourne l'automate décrit dans l'exemple ci-dessus. *On pourra s'inspirer du code fourni en section 7.2.*
- Ecrire une fonction **afficheAutomate** qui affiche le contenu d'une structure d'automate. *Un exemple d'affichage est donné en section 7.3.*
- Ecrire une fonction **compteTransitions** qui retourne le nombre de transitions d'un automate.
- Ecrire une fonction **deterministe** qui retourne 1 si un automate est déterministe, 0 sinon.
- Ecrire une fonction **complet** qui retourne 1 si un automate est complet, 0 sinon.
- Ecrire une fonction **supprimeTransition** qui supprime une transition dans un automate.
- Ecrire une fonction **supprimeEtat** qui supprime un état dans un automate.
- Ecrire une fonction **completeAutomate** qui complète un automate.
- Ecrire une fonction **fusionEtats** qui fusionne deux états d'un automate.

3 Test du vide (2 séances)

Pour cette partie, vous devez vous rafraîchir la mémoire sur les notions de graphe et de parcours (en largeur ou en profondeur) vues en L2.

- Proposer une structure de graphe orienté par liste d'adjacence.
- En utilisant un parcours en profondeur, écrire une fonction **chemin** qui étant donnés deux sommets p et q d'un graphe, retourne 1 s'il existe un chemin de p vers q, 0 sinon. *On rappelle le pseudo-code du parcours en profondeur en section 7.4.*
- Écrire une fonction **automateToGraphe** qui étant donné un automate retourne son graphe sous-jacent, c'est-à-dire le graphe que l'on obtient à partir de l'automate sans tenir compte des états intiaux et finaux, ni des étiquettes des transitions.
- Justifier que le langage reconnu par un automate $\mathcal{A}$ est non vide si et seulement si il existe un chemin allant d'un état initial à un état final.
- Écrire une fonction **langageVide** qui retourne 1 si le langage reconnu par un automate est vide, 0 sinon.
- Écrire une fonction **supprimeNonCoAccessibles** qui supprime d'un automate les états qui ne sont pas co-accessibles.
- Écrire une fonction **supprimeNonAccessibles** qui supprime d'un automate les états qui ne sont pas accessibles.

4 Produit d'automates (1-2 séances)

Pour implémenter le produit d'automates, la difficulté réside dans le codage des états du produits par un seul entier (et non un couple). Supposons que $\mathcal{A}_1$ soit un automate à n états et $\mathcal{A}_2$ un automates à m états. L'automate produit de ces deux automates aura $n \times m$ états. On propose le codage suivant : l'état (p, q) du produit (théorique) sera codé en pratique par le numéro d'état $p * m + q$. Réciproquement, un état codé par l'entier r correspondra dans le produit théorique au couple $(r/m, r\%m)$. Un exemple de ce codage est donné en section 7.5.

- Écrire une fonction **produit** qui calcule le produit de deux automates.
- Écrire une fonction **intersectionVide** qui retourne 1 si l'intersection de deux langages donnés par des automates est vide, 0 sinon.

5 Déterminisation (2 séances)

Pour l'algorithme de déterminisation, le problème technique est la numérotation des états de l'automate déterminisé qui, théoriquement, sont des ensembles d'états. Pour cela il faut donner à chaque ensemble rencontré lors de la déterminisation un numéro et garder une table de correspondance à jour. Des indications sont données en section 7.6.

- Implémenter l'algorithme de déterminisation d'un automate par une fonction **determinise**.
- Écrire une fonction **inclusion** qui teste l'inclusion de langages donnés par des automates.

6 Minimisation (2 séances)

On s'intéresse ici à la minimisation d'un automate. D'abord par une méthode simple mais pas trés efficace, puis par l'algorithme vu en cours.

- En utilisant la question précédente, écrire une fonction **nerodeEquivalent** qui teste si deux états sont équivalents pour l'équivalence de Nerode.
- Écrire une fonction **minimiseNerode** qui minimise un automate en fusionnant les états Nerode-équivalents.
- Écrire une fonction **minimiseHopcroft** qui minimise un automate par l'algorithme de Hopcroft (celui vu en cours, qui s'appuie aussi sur l'équivalence de Nerode).

7 Annexes

7.1 Ajouter dans une liste triée

```
void ajouteListe(liste** l,int q){
        liste* ptl;
        liste* tmp;
        ptl=*l;
        if(!ptl){
                ptl=(liste*) malloc(sizeof(liste));
                ptl->state=q;
                ptl->suiv=NULL;
                *l=ptl;
                return;
        }
        if(ptl->state == q){
         return;
        }
        if(q< ptl->state){
                tmp=*l;
                *l=(liste*) malloc(sizeof(liste));
                (*l)->state=q;
                (*l)->suiv=tmp;
                return;
        }

        while(ptl->suiv && ptl->suiv->state <q){
                ptl=ptl->suiv;
        }
        if(!ptl->suiv){
                ptl->suiv=(liste*) malloc(sizeof(liste));
                ptl=ptl->suiv;
                ptl->state=q;
                ptl->suiv=NULL;
                return;
        }
        if(ptl->suiv->state==q){
                return;
        }
        tmp=ptl->suiv;
        ptl->suiv=(liste*) malloc(sizeof(liste));
        ptl=ptl->suiv;
        ptl->state=q;
        ptl->suiv=tmp;
}
```

7.2 Definir un automate en dur

```
automate A;
int i,j;
A.size=2;
A.sizealpha=2;
A.initial=(int*) malloc(A.size*sizeof(int));
A.initial[0]=1;
A.initial[1]=0;
A.final=(int*) malloc(A.size*sizeof(int));
A.final[0]=0;
A.final[1]=1;
A.trans=(liste***) malloc(A.size*sizeof(liste**));
for(i=0;i<A.size;i++){
        A.trans[i]=(liste**) malloc(A.sizealpha*sizeof(liste*));
```

```
        for(j=0;j<A.sizealpha;j++){
                A.trans[i][j]=NULL;
        }
}

ajouteTransition(A,0,0,'a');
ajouteTransition(A,0,1,'a');
ajouteTransition(A,1,1,'b');
afficheAutomate(A);
```

7.3 Exemple affichage

L'affichage ci-dessous correspond à l'exemple donné dans la fonction précédente.

```
Les etats initaux
 0
Les etats finaux
 1
Les Transitions
---------------------------------
Depuis l'etat 0
avec la lettre a :
0 1
avec la lettre b :

---------------------------------
Depuis l'etat 1
avec la lettre a :

avec la lettre b :
1
```

7.4 Parcours en profondeur

Soit un graphe (S, A) où S est l'ensemble des sommets et A l'ensemble des arètes. On a $A \subseteq S \times S$. Le pseudo-code suivant répond à la question de savoir s'il existe dans le graphe un chemin de p vers q en faisant un parcours en profondeur.

On utilise pour cela la fonction **VisiterPP**.

Données : S, A, p
Algorithme VisiterPP(S,A,p) :
    Colorier p en vert;
    Pour chaque fils r de p,
        Si r est rouge alors
            VisiterPP(S,A,r);
        FinSi
    FinPour

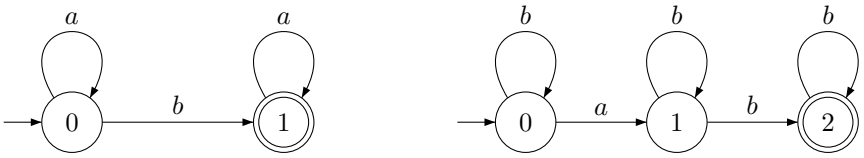
On peut maintenant donner l'algorithme.

Données : S, A, p et q
Algorithme :
    Colorier tous les sommets en rouge;
    VisiterPP(S,A,p);
    Retourner (couleur(q)==vert);

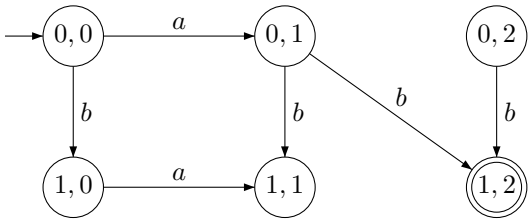
Il est à noter que l'algorithme proposé n'est pas optimal et qu'il est *souhaitable* de mettre la condition d'arrèt sur la couleur de q dans la fonction visite. En revanche, pour le test du vide, la coloration du graphe présentée est importante. A vous de voir pourquoi et de réflechir à la complexité de vos algorithmes.

7.5 Exemple de codage du produit

Considérons les deux automates suivants :



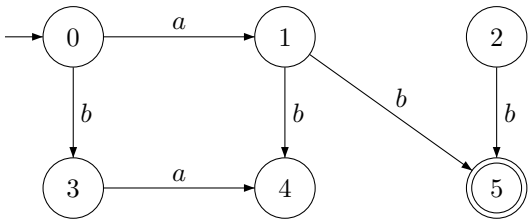
Le premier a deux états (donc $n = 2$) et le second a trois états (donc $m = 3$).
Le produit de ces deux automates est le suivant :



On obtient le codage suivant des états.

L'état	(p, q)	est codé par $p * m + q$.
L'état	$(0, 0)$	est codé par $0 * m + 0 = 0$.
L'état	$(0, 1)$	est codé par $0 * m + 1 = 1$.
L'état	$(0, 2)$	est codé par $0 * m + 2 = 2$.
L'état	$(1, 0)$	est codé par $1 * m + 0 = 3$.
L'état	$(1, 1)$	est codé par $1 * m + 1 = 4$.
L'état	$(1, 2)$	est codé par $1 * m + 2 = 5$.

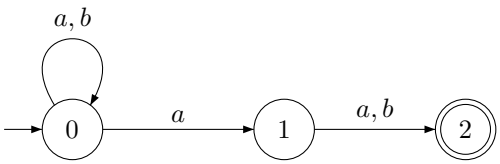
Ce qui nous donne au final l'automate :



Si l'on voulait réciproquement retrouver p et q à partir du codage dans le produit, on peut appliquer la formule donnée. Par exemple, à partir de l'état 4 du produit, on retrouve p en calculant $4/3$ (qui vaut 1). On retrouve q en calculant $4\%3$ qui vaut 1 aussi.

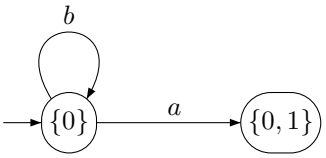
7.6 Déterminisation

Rappelons le principe de déterminisation sur l'automate suivant :



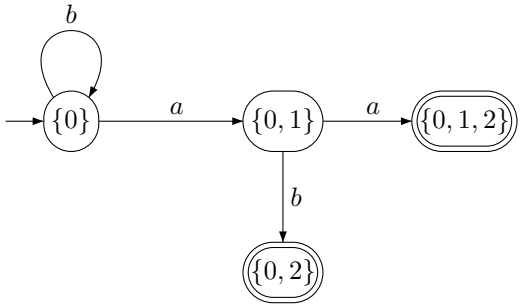
On part de l'état $\{0\}$ (seul état initial) et on construit les ensembles d'états accessibles :

Etape 1 De l'état $\{0\}$, en lisant a on arrive dans l'état $\{0, 1\}$ et en lisant b on reste dans l'état $\{0\}$. A ce stade

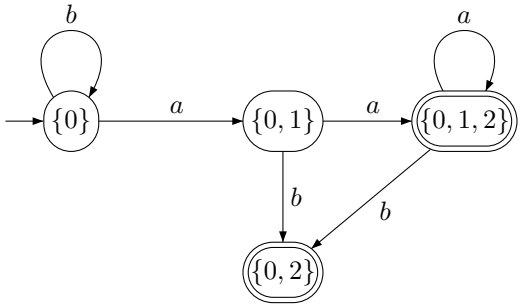


on a l'automate suivant :

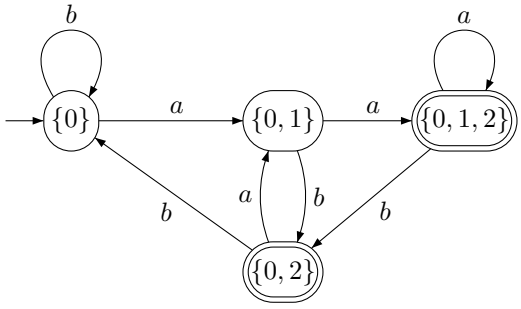
Etape 2 De l'état $\{0, 1\}$, en lisant a on arrive dans l'état $\{0, 1, 2\}$ et en lisant b arrive dans l'état $\{0, 2\}$. A ce stade on obtient l'automate suivant :



Etape 3 De l'état $\{0, 1, 2\}$, en lisant a on reste dans l'état $\{0, 1, 2\}$ et en lisant b arrive dans l'état $\{0, 2\}$. A ce stade on obtient l'automate suivant :



Etape 4 De l'état $\{0, 2\}$, en lisant a retourne dans l'état $\{0, 1\}$ et en lisant b revient dans l'état $\{0\}$. A ce stade on obtient l'automate suivant :



Etape 5 Tous les états on été traîetées, la procédure peut s'arréter.

La problématique est la similaire à celle du produit, il faut renommer les états. Cependant, contrairement au produit, on ne connaît pas *a priori*, le nombre d'état de l'aumtomate déterminisé. On va donc utiliser une file dans laquelle on ajoutera, au fur et à mesure de l'algorithme, les nouveaux états introduits. Pour cela on introduit les listes chaînées suivantes :

```
typedef struct s_iliste {
        int* val;
        int tailleVal;
```

```
        int state;
        struct s_iliste* suiv;
} iliste;
```

où `val` est un tableau du numéro des états, `tailleVal` est la taille de ce tableau (nombre d'éléments), `state` sera le numéro d'état utilisé dans le recodage.

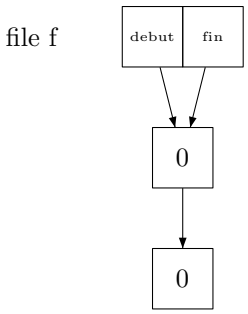
Classiquement, la file est définie avec deux pointeurs, l'un sur la tête, l'autre sur la queue.

```
typedef struct {
        iliste* debut;
        iliste* fin;
} ifile;
```

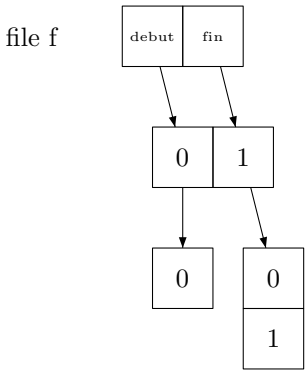
Au début de l'algorithme, On déclare une file vide. On ajoute l'état inital de l'automate déterminisé dans cette file. On calcule alors les états accessibles pour chaque lettre, en les ajoutant s'il n'existent pas dans cette file. On met à jour les listes de transitions.

Sur l'exemple ci-dessus cela donne :

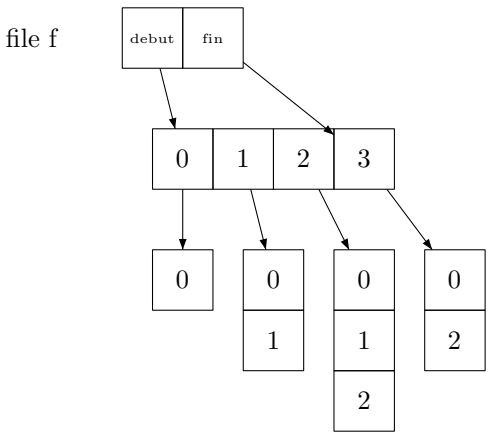
Etape 0 La file est la suivante :



Etape 1 On cherche ensuite les états accessible à partir de $\{0\}$. On obtient un nouvel état $\{0, 1\}$. La liste devient donc :



Etape 2 On ajoute les états $\{0, 1, 2\}$ et $\{0, 2\}$. La liste devient donc :



Etapes 3/4 On n'ajoute pas de nouveaux états.

Bien sûr, il faut de plus dans l'algorithme n'oublier ni les transitions, ni les états initiaux et finaux (qui peuvent être gérés à la fin). Afin de savoir quel état doit être traîté dans la file, il convient d'utiliser un pointeur.

Les fonctions suivantes pourront vous être utiles.

Pour tester si un élément est déjà dans la file.

```
int estDansFile(ifile f, int* pt, int n){
        int i;
        iliste* tmp;
        tmp=f.debut;
        while(tmp){
                if(tmp->tailleVal == n){
                        i=0;
                        while(i< n && pt[i]==tmp->val[i]){
                                i++;
                        }
                        if(i == n) {
                                return 1;
                        }
                }
                tmp=tmp->suiv;
        }
        return 0;
}
```

Pour insérer un élément dans la file.

```
void ajouteFile(ifile* f, int* pt,int n){
        int i;
        iliste* tmp;
        tmp= (iliste*) malloc(sizeof(iliste));
        tmp->val=(int*) malloc(n*sizeof(int));
        for(i=0;i<n;i++){
                tmp->val[i]=pt[i];
        }
        tmp->suiv=NULL;
        tmp->tailleVal=n;
        if(f->fin != NULL){
                tmp->state=f->fin->state+1;
                f->fin->suiv = tmp;
                f->fin= tmp;

                return;
        }
        tmp->state=0;
        f->fin= tmp;
        f->debut= tmp;
}
```

Pour le déboggage, fonction d'affichage d'une file :

```
void afficheFile(ifile f){
        iliste* pt;
        int i;
        int j=0;
        pt=f.debut;
        while(pt){
                printf("----------------------------\n");
                printf("Element %d \n",pt->state);
                for(i=0;i<pt->tailleVal;i++){
                        printf( " %d",pt->val[i]);
                }
                pt=pt->suiv;
                j++;
                printf("\n");
        }
}
```