Langage C : énoncé et corrigé des exercices

1.4 LISTE CHAINEE

Le premier exercice est corrigé.

Exercice 33 Ecrire un programme qui gère les listes chaînées. Pour cela, vous créerez un type de structure de liste chaînée dont les éléments sont des entiers. Vous créerez également un type de structure contenant trois pointeurs, *prem*, *der* et *cour* permettant d'accéder respectivement au premier et dernier élément, ainsi qu'à l'élément courant (cf. poly).

Le programme se composera de plusieurs fonctions :

- Une fonction de création *creer_liste* qui alloue la mémoire nécessaire à une liste et à la structure des trois pointeurs associés.
- Une fonction *insrer_apres* qui prend en paramètre une liste et un entier. Cette fonction insère l'entier dans la liste, après l'élément courant (pointé par *cour*).

Vous testerez votre programme sous UNIX, et fournirez un exemple ou deux d'exécution.

Dans les exercices qui suivent, on utilisera les types de données suivants :

```
typdef struct {
                char sNom[30];
                int iAnnee_naissance;
                char sTelephone[20];
              } personne;
```

```
typedef struct item
         { personne entree;
           struct item *suivant;
         } cellule;
```

```
typedef cellule *liste;
```

Exercice 34 Ecrire une fonction *int saisir(personne *individu)* saisissant au clavier un objet de type *personne*. Cette fonction retournera 0 si l'utilisateur entre un symbole spécial pour signifier qu'il n'a pas de données à saisir, 1 sinon.

Exercice 35 Ecrire une fonction *int est_identique(personne individu1, personne individu2)* retournant 1 si les deux objets *individu*1 et *individu*2 ont les mêmes valeurs.

Exercice 36 Ecrire une fonction *cellule *creer(personne individu)* retournant l'adresse d'un objet de type *cellule* dont les champs *entree.sNom*, *entree.iAnnee_naissance*, *entree.sTelephone* ont les valeurs respectives des champs correspondants de *individu*.

Exercice 37 Ecrire une fonction *void afficher(liste l)* affichant à l'écran une représentation lisible des éléments de la liste *l*.

Exercice 38 Ecrire une fonction *cellule *position(personne individu, liste l)* retournant l'adresse de la première cellule de *l* dont les valeurs des champs de *individu* sont identiques, et NULL si cette cellule n'existe pas.

Exercice 39 Dans cet exercice, on utilisera le type :

```
typedef struct info {
                     int iAge;
                     char* sNom;
                    }
```

```
typedef struct cellule
       {  struct info individu;
          struct cellule *suivant;
       } *liste;
```

```
/* Déclaration de variable */
liste prem;
int AGEMAX = 125;
```

Ecrire une fonction *separation(int iAgeMaxJeune)* qui, sans recopier de cellules, éclate la liste *prem* en deux listes : celle des individus d'âge inférieur à la valeur $iAgeMaxJeune$, dont l'adresse sera contenue dans une variable globale *jeunes*, et celle des individus d'âge supérieur ou égal à $iAgeMaxJeune$, dont l'adresse sera contenue dans une variable globale *vieux*.

Exercice 40 Dans cet exercice, on représente une liste d'entiers par une structure de liste chaînée circulaire telle que :

```
typedef struct cellule {
                         int iVal;
                         struct cellule *suivant;
                       } cellule;

typedef cellule *liste;
```

En raison de la circularité de la liste, dans aucune cellule, le champ *suivant* ne vaut $NULL$.

On suppose que la liste est donnée par l'adresse de la première cellule, et que l'ajout d'un nouvel élément se fait en tête de liste.

Ecrire les fonctions suivantes :

- Recherche d'un élément dans une liste circulaire (retournant 1 si l'élément appartient 0 sinon).
- Ajout d'un élément à une liste circulaire.
- Suppression d'un élément dans une liste circulaire.

Exercice 41 On reprend l'exercice précédent, en supposant maintenant que la liste est donnée par l'adresse de sa dernière cellule. L'ajout d'un élément se fait toujours en tête de liste. Reprendre les fonctions précédentes avec cette nouvelle configuration.

1.5 PILE ET FILE

Ces exercices sont corrigés.

Exercice 42 Ecrire un programme qui gère une pile à l'aide d'une liste chaînée. Pour cela, vous créerez un type de structure de pile à l'aide d'une liste chaînée dont le éléments sont des entiers. Le pointeur d'un élément de la pile pointe vers l'élément précédent.

Le programme se composera de plusieurs fonctions :

- Une fonction de création *creer_pile* qui retourne une pointeur de type *pile*, nul.
- Une fonction *vide* qui retourne 0 si la pile, passée en paramètre, est non vide, et un nombre différent de 0 dans le cas contraire.
- Une fonction *sommet* qui retourne le sommet de la pile passée en paramètres.
- Une fonction *empiler* qui empile l'entier $iVal$, passé en paramètre, à la pile p, également passée en paramètre.
- Une fonction *desempiler* qui supprime le sommet de la pile p, passée en paramètre. La mémoire occupée par le précédent sommet de la pile est libérée.
- Une fonction $affiche r\_recursive$ qui affiche le contenu de la pile de manière récursive.

Vous testerez votre programme sous UNIX, et fournirez un exemple ou deux d'exécution.

Exercice 43 Ecrire un programme qui gère une file de caractères en anneau continu (voir poly de cours). Pour cela, vous créerez un type de structure de file contenant quatre pointeurs sur des chaînes de caractères. Ces pointeurs représenteront respectivement le début et la fin de la zone mémoire allouée pour la file, ainsi que la tête et la queue de la file.

Le programme se composera de plusieurs fonctions :

- Une fonction de création $creer\_file$ qui retourne une file. Cette fonction alloue la zone mémoire nécessaire à la file, ainsi que les pointeurs de la file. A la création les pointeurs de tête et de début pointent au début de la zone mémoire. Les pointeurs de queue et de fin pointent à la fin de la zone. On passera en paramètre de la fonction la taille mémoire (c-à-d le nombre de caractères maximum) de la file.
- Une fonction *plusun* qui avance un pointeur de type $char*$, passé en paramètre, d'un espace mémoire dans la file.
- Une fonction *vide* qui retourne 0 si la file est non vide (un nombre supérieur à 0 sinon). Cette fonction fait appel à la fonction *plusun*. Une file est vide lorsque l'emplacement mémoire qui suit la queue de la file est l'adresse de la tête de file.
- Une fonction *pleine* qui retourne 0 si la file est non pleine (un nombre supérieur à 0 sinon). Cette fonction fait appel à la fonction *plusun*. Une file est pleine lorsque le deuxième emplacement mémoire qui suit la queue de la file est l'adresse de la tête de file. Une file est non pleine lorsqu'il y a au moins un emplacement mémoire vide entre la queue et la tête.
- Une fonction *lire* qui retourne le caractère situé à la tête de la file.
- Une fonction *avancer* qui permet de faire avancer la tête de la file d'un espace mémoire. Elle fait appel à *plusun*. Si la file est vide, la fonction affiche un message d'erreur[1].
- Une fonction *ajouter* qui ajoute un caractère en queue de la file (et déplace donc la queue d'un espace mémoire). Si la file est pleine, la fonction affiche un message d'erreur.
- Une fonction $affiche r$ qui affiche la file.

Vous testerez votre programme sous UNIX, et fournirez un exemple ou deux d'exécution.

1. Vous pouvez réaliser une fonction *erreur* qui prend en paramètre une chaîne de caractères (le message d'erreur) et l'affiche.

1.6 ARBRES BINAIRES

Dans tous les exercices qui suivent, on utilisera la représentation d'une expression arithmétique par un arbre binaire, au moyen des données suivantes :

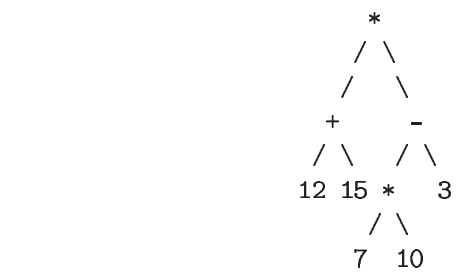
```
typedef union {                    typedef struct Sommet
                int iNombre;                { Type_valeur valeur;
                char cSigne;                  struct Sommet *fils_g, *fils_d;
              } Type_valeur;                } Sommet;

typedef Sommet *Arbre;
```

Exercice 44 Ecrire une fonction *Arbre construire_feuille(int iEntier)* retournant un arbre correspondant à l'expression arithmétique constituée d'un seul entier (c'est-à-dire un arbre contenant uniquement une feuille, sommet sans fils gauche ni fils droit).

Exercice 45 Ecrire une fonction *Arbre construire_noeud(char cOperateur, Arbre gauche, Arbre droit)* retournant un arbre dont la racine est la valeur *sOperateur* et dont les fils droit et gauche sont respectivement *droit* et *gauche*.

Exercice 46 Ecrire les fonctions *void infixe(Arbre tree), void prefixe(Arbre tree)* et *void postfixe(Arbre tree)* en affichant en notation classique (avec parenthèses), en notation polonaise préfixée et en notation polonaise postfixée l'expression arithmétique représentée par *tree*.

Par exemple, l'arbre :



En infixe, s'écrit (12+15)*((7*10)-3)
Le prefixe est *+1215-*7103
Le postfixe doit afficher 1215+710*3-*

Exercice 47 Ecrire une fonction *int eval(Arbre tree)* retournant la valeur de l'expression représentée par *tree*.

Exercice 48 Ecrire une fonction *Arbre construire(char* sExpression, int iIndiceDebut, int iIndiceFin)* qui construit l'arbre binaire correspondant à l'expression arithmétique se trouvant entre les indices *iIndiceDebut* et *iIndiceFin* de la chaîne *sExpression*. *sExpression* est écrite sous forme parenthèsée. Vous pouvez décomposer cette fonction en sous fonctions. Ecrire notamment une fonction *int indice_fin(char* sExpression, int iIndice)* qui calcule l'indice où se trouve une parenthèse fermante correspondant à *sExpression[iIndice]='('*.

EXERCICE 33

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

/* Déclaration d'un type élément de liste chaînée */
typedef struct lis
{
 /* Valeur de l'élément */
 int iValeur;
 /* Pointeur sur l'élément suivant */
 struct lis* suivant;
} *element;

/* Déclaration d'un type liste chaînée */
typedef struct t
{
 /* pointeurs sur le premier élément de la liste, sur le dernier */
 /* et sur l'élément courant                                     */
 element prem,der,cour;
} *liste;

/* Fonction de création d'une liste */
/* Paramètres d'entrée : Aucun      */
/* Paramètre de sortie : une liste  */
liste creer_liste()
{
  /* Déclaration d'une liste locale */
  liste Liste_Entier;

  /* Allocation mémoire de la liste */
  Liste_Entier = (liste)malloc(sizeof(struct t));
```

```
  /* Allocation mémoire des pointeurs prem, der, et cour de la liste    */
  /* au départ, la liste est vide, les pointeurs ne pointent sur rien.  */
  /* Ceci est une allocation multiple                                   */
  Liste_Entier->prem=Liste_Entier->der=Liste_Entier->cour = NULL;

  /* La liste est retournée */
  return Liste_Entier;
}

/* Fonction d'insertion après la position courante d'un élément         */
/* dans une liste                                                       */
/* Paramètres d'entrée :                                               */
/*                   liste Liste_Entier : liste où a lieu l'insertion   */
/*                   int iVal  : valeur de l'élément                    */
/* Paramètre de sortie : rien                                          */
void inserer_apres(liste Liste_Entier,int iVal)
{ /* Element à insérer */
  element Element_Liste = NULL;

  /* Allocation mémoire de l'élément */
  Element_Liste = (element)malloc(sizeof(struct lis));

  /* Initialisation de la valeur de e */
  Element_Liste->iValeur = iVal;

  /* S'il n'y a pas d'élément dans la liste */
  if (Liste_Entier->prem == NULL)
  { /* prem, der et cour pointent sur e */
    Liste_Entier->prem=Liste_Entier->der=Liste_Entier->cour =  Element_Liste;

    /* Il n'y a pas d'élément suivant */
    Element_Liste->suivant = NULL;

  } /* Fin de S'il n'y a pas d'élément dans la liste */

  /* Si la liste est non vide */
  else

    /* Si l'élément courant n'est pas le dernier */
    if(Liste_Entier->cour!=Liste_Entier->der)
    {
      /* Element_Liste pointe sur l'élément suivant de l'élément courant */
      Element_Liste->suivant = Liste_Entier->cour->suivant;

      /* Le suivant de l'élément courant est e */
      Liste_Entier->cour->suivant = Element_Liste;

      /* L'élément courant est Element_Liste */
      Liste_Entier->cour = Element_Liste;
    } /* Fin de Si l'élément courant n'est pas le dernier */
```

```
  /* Si l'élément courant est le dernier élément */
  else
  { /* Element_Liste n'a pas de suivant */
    Element_Liste->suivant = NULL;

    /* L'élément suivant de l'élément courant est Element_Liste */
    Liste_Entier->cour->suivant = Element_Liste;

    /* Element_Liste est l'élément courant et le dernier élément */
    Liste_Entier->cour = Liste_Entier->der = Element_Liste;
  } /* Fin de Si l'élément courant est le dernier élément */
}

/*****************************************************************************/
/*                            Programme principal                            */
/*****************************************************************************/
void main()
{ /* Déclaration d'une liste */
  liste l;
  /* Déclaration d'un compteur */
  int iCompteur;

  /* Création de la liste */
  l = creer_liste();

  /* Pour les entiers de 1 à 5 */
  for(iCompteur=1;iCompteur<=5;iCompteur++)
  { /* Ajout de l'entier dans la liste */
    inserer_apres(l,iCompteur);
    /* Affichage de l'élément courant */
    printf("Courant = %d\n",l->cour->iValeur);
  }

  /* Affichage de la liste */
  /* On positionne le pointeur courant sur le premier élément */
  l->cour = l->prem;
  printf("Liste dans sa totalité :");

  /* Tant qu'on est pas au dernier élément */
  while(l->cour!=l->der)
  { /* Affichage de la valeur de l'élément courant */
    printf("%d ",l->cour->iValeur);
    /* On passe au suivant */
    l->cour = l->cour->suivant;
  }

  /* Affichage du dernier élément */
  printf("%d ",l->cour->iValeur);
}
```

EXERCICE 42

Structure pile et opérations élémentaires en C

```
#include<stdio.h>

/* Définition d'une structure de pile */
typedef struct pi
{ /* valeur du sommet de la pile*/
  int iValeur;
  /* queue de la pile */
  struct pi *prec;
} * pile;

/* Fonction de création de pile         */
/* Paramètres d'entrée :  Aucun         */
/* Paramètres de sortie : La pile créée */
pile creer_pile()
{
 /* On retourne un pointeur nul */
 return NULL;
}

/* Fonction de test pour savoir si la pile est vide                        */
/* Paramètres d'entrée :  pile p : une pile                                */
/* Paramètres de sortie : un entier (0 si la pile est vide, >0 sinon)      */
int vide(pile p)
{
 /* On retourne la valeur du test p est NULL                   */
 /* Si p est NULL alors le test (p==NULL) sera différent de 0  */
 /* Si p est non NULL alors le test (p==NULL) sera égal à 0    */
 return (p==NULL);
}

/* Fonction qui retourne le sommet de la pile   */
/* Paramètres d'entrée :  pile p : une pile     */
/* Paramètres de sortie : le sommet de la pile  */
int sommet(pile p)
{ /* Si p est non vide */
  if(p)
   /* On retourne la valeur du sommet */
   return p->iValeur;
  /* Si p est vide on retourne la dernière valeur négative */
  /* possible pour un entier codé sur 2 octets             */
  else return -32768;
}
```

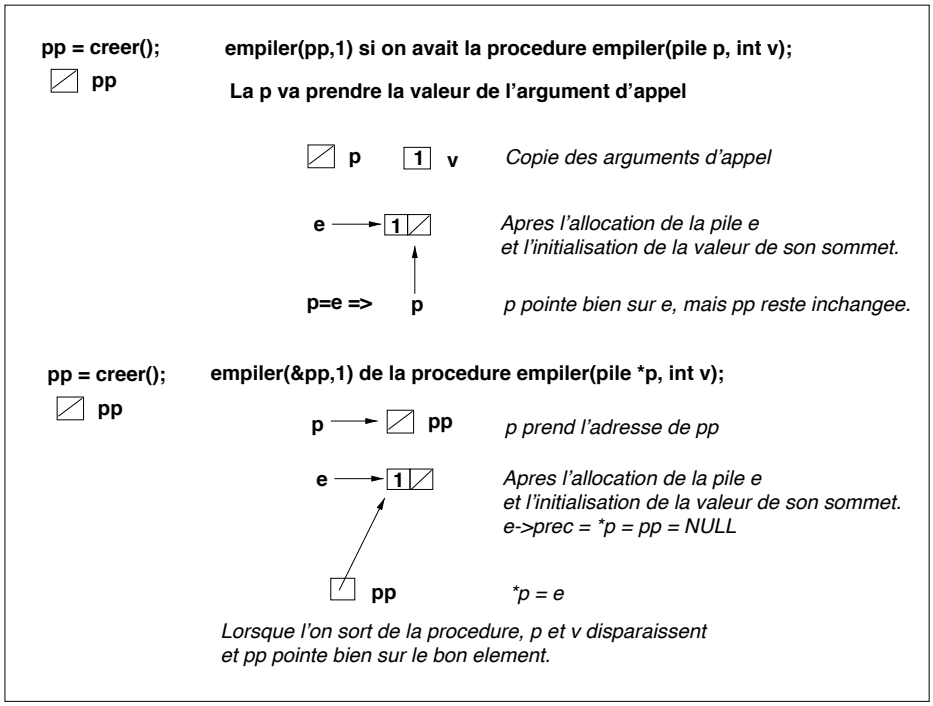


FIG. 2 – **Empilement d'une valeur dans une pile implémentée en liste chaînée.**

Fonction *empiler*

```
/* Fonction qui empile une valeur sur la pile              */
/* Paramètres d'entrée :                                   */
/*  pile *p  : un pointeur sur la pile qui va être modifiée */
/*  int iVal : la valeur à empiler                         */
/* Paramètres de sortie : rien                             */
void empiler(pile* p,int iVal)
{ /* Déclaration d'une pile locale à la fonction */
  pile e;

  /* Allocation mémoire de la pile locale */
  e = (pile)malloc(sizeof(struct pi));

  /* Initialisation de la valeur de la pile locale */
  e->iValeur = iVal;

  /* Ajout de la pile p à la queue de e */
  e->prec = *p;

  /* Mise à jour de la pile, p pointe sur le nouveau sommet de la pile */
  *p = e;
}
```

Pour les explications de la fonction *empiler*, voir figure 2.

Fonction désempiler

```
/* Fonction qui desempile la pile                              */
/* Paramètres d'entrée :                                       */
/*     pile* p : un pointeur sur la pile qui va être modifiée  */
/* Paramètres de sortie : rien                                 */
void desempiler(pile* p)
{ /* Déclaration d'une pile locale */
  pile q;

  /* Si la pile pointée par p est non vide */
  if(*p)
  { /* q est la pile pointée par p */
    q = *p;

    /* la pile pointée par p devient la queue de la pile */
    *p = (*p)->prec;

    /* Le mémoire occupée par q est libérée */
    free(q);
  }
}
```

Affichage récursif d'une pile L'affichage d'une pile peut se faire de manière récursive (voir figure 3). Une fonction récursive est une fonction qui s'appelle depuis son propre corps.
On remonte les appels en effectuant les actions écrites après les appels de récursivité.

```
/* Fonction qui affiche la pile de manière récursive      */
/* Paramètres d'entrée : pile p : la pile à afficher      */
/*   On ne transmet pas l'adresse de la pile, car elle ne */
/*   va pas être modifiée                                 */
/* Paramètres de sortie : rien                            */
void afficher_recursive(pile p)
{ /* Valeur du sommet de la pile à afficher */
  int iVal;

  /* Si la pile est non vide */
  if(p)
   {  /* iVal prend la valeur du sommet de la pile */
      iVal = p->iValeur;

      /* La pile devient la queue de p                                  */
      /* la pile passée en paramètres n'est pas modifiée  pour autant */
      /* puisque ce n'est pas l'adresse qui est passée en paramètres  */
      p = p->prec;

      /* Appel récursif de la fonction pour le reste de la pile */
      afficher_recursive(p);
      /* Affichage de la valeur du sommet */
      printf("%d ", iVal);
   }
}
```

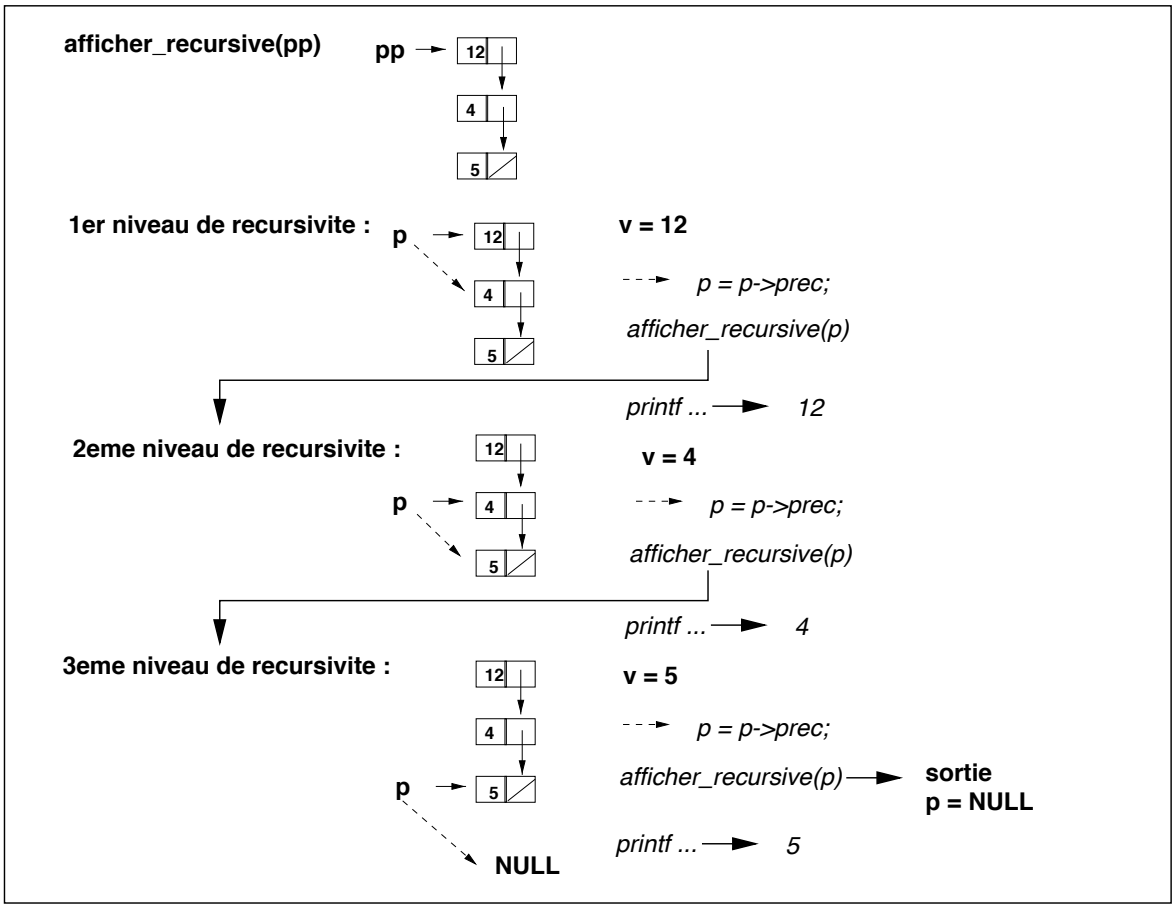


FIG. 3 – **Affichage récursif d'une pile implémentée en liste chaînée.**

Affichage d'une pile par échange de pointeurs

```
/* Fonction qui affiche la pile par échange de pointeurs  */
/* Paramètres d'entrée :                                  */
/*                      pile p : la pile à afficher       */
/*                      on ne transmet pas l'adresse de   */
/*                      la pile, car elle ne va pas être  */
/*                      modifiée                          */
/* Paramètres de sortie : rien                            */
void afficher_echange_pointeur(pile p)
{ /* Déclaration de deux piles locales */
  pile q,r;

  /* Initialisation de q */
  q = NULL;

  /* Tant que p est non nulle  */
  /* On va retourner la pile p */
  while(p)
   { /* Initialisation de r à l'élément précédente le sommet de p */
     r = p->prec;

     /* La queue de p devient q */
     p->prec = q;

     /* q devient la pile p modifiée précédemment */
     q = p;

     /* p devient la pile r initialisée précédemment*/
     p = r;
   }

  /* Tant que q est non nulle                 */
  /* On va afficher les éléments de la pile q */
  while(q)
  { /* Affichage de la valeur du sommet de q */
    printf("%d ",q->iValeur);

    /* r devient la queue de la pile q */
    r = q->prec;

    /* La queue de q devient p */
    q->prec = p;

    /* la pile p devient q */
    p = q;

    /* la pile q devient la pile r */
    q = r;
    }
}
```

Lorsque l'on fait appel à la fonction *afficher_echange_pointeur*, l'adresse de la pile n'est pas modifiée. Seul le contenu de la pile est touché (voir figure 4 page 163).

```
/*****************************************************************************/
/*                              Programme principal                          */
/*****************************************************************************/
void main()
{ /* Déclaration d'une pile */
  pile p;

  /* Création de la pile */
  p=creer_pile();

  /* Empilement de 8, 4 et 19 */
  empiler(&p,8);
  afficher_recursive(p);
  printf("\n");
  empiler(&p,4);
  afficher_recursive(p);
  printf("\n");
  empiler(&p,19);
  afficher_recursive(p);
  printf("\n");


  /* desempilement */
  desempiler(&p);
  afficher_echange_pointeur(p);
  printf("\n");
  desempiler(&p);
  afficher_echange_pointeur(p);
  printf("\n");
  desempiler(&p);
  afficher_echange_pointeur(p);
  if(vide(p)) printf("Pile vide\n");

}
```

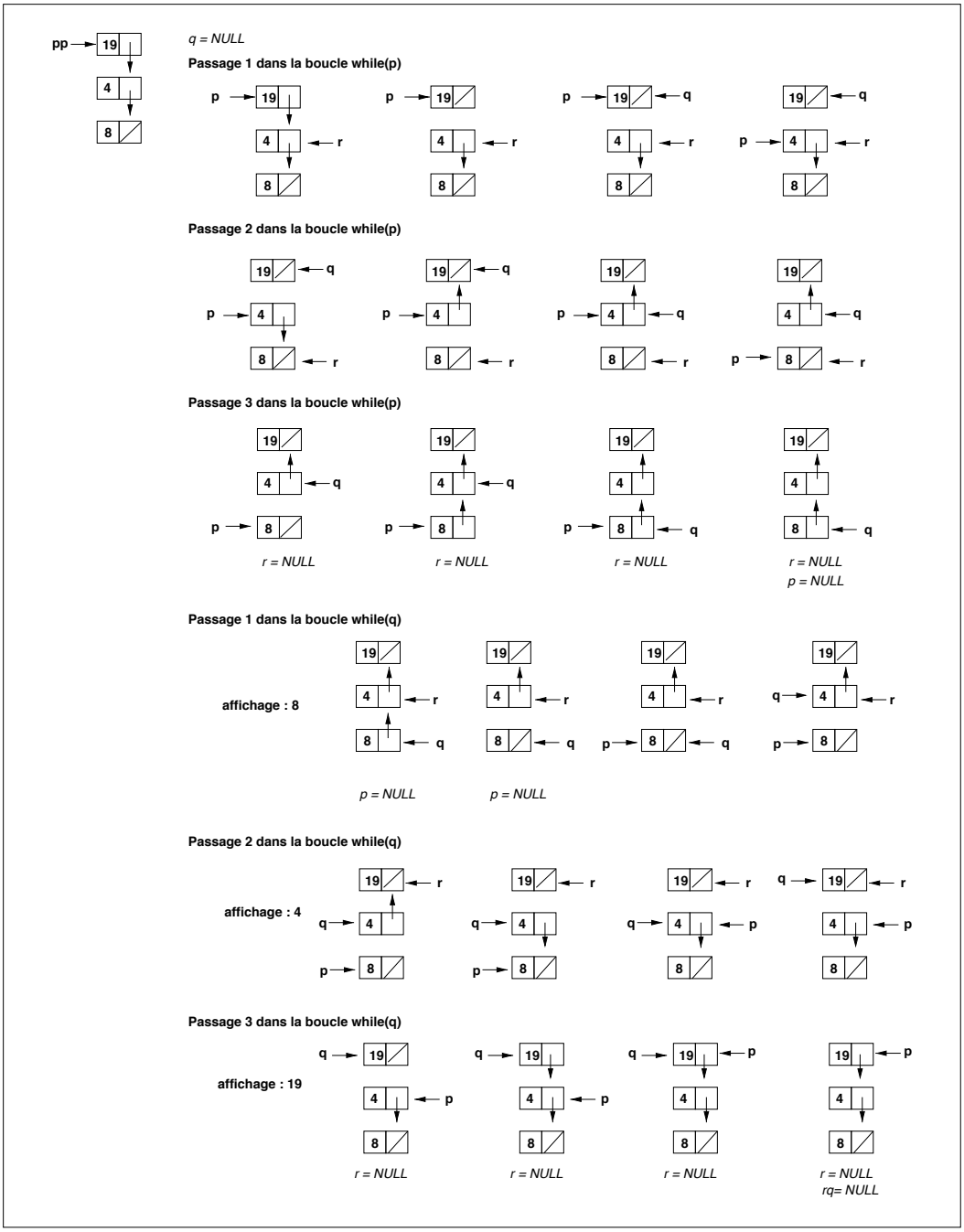


FIG. 4 – **Affichage par échange de pointeurs d'une pile implémentée en liste chaînée.**

EXERCICE 43

```
#include<stdio.h>

/* Définition d'un type file de caractères*/
typedef struct f
{ /* pointeurs sur le début et la fin de la zone mémoire de la file */
  char* cDebut;
  char* cFin;
  /* pointeurs sur la tête et la queue de la file                    */
  char* cTete;
  char* cQueue;
} *file;

/* Fonction de création de file                                              */
/* Paramètres d'entrée :                                                     */
/*          unsigned uTailleMemoire : longueur de la zone mémoire allouée */
/*                                    pour la file                           */
/* Paramètres de sortie : la file créée                                      */
file creer(unsigned uTailleMemoire)
{ /* Déclaration d'une file locale */
  file f;

  /* Allocation mémoire du pointeur sur la file */
  f = (file)malloc(sizeof(struct f));

  /* Allocation mémoire des pointeurs de début et de tête */
  f->cDebut = f->cTete = (char*)malloc(uTailleMemoire);

  /* Allocation mémoire des pointeurs de fin et de queue                    */
  /* On les placent à (n-1) espaces mémoire (de un octet) de la tête et     */
  /* du début                                                               */
  f->cFin = f->cQueue = f->cDebut + uTailleMemoire - 1;

  /* On retourne la file créée */
  return f;
}
```

```
/* Fonction de déplacement d'un espace mémoire d'un pointeur de la file   */
/* Paramètres d'entrée : file f : la file traitée                         */
/*                       char* cPointeur : le pointeur à avancer de 1     */
/* Paramètres de sortie : le pointeur avancé                              */
char* plusun(file f, char* cPointeur)
{ /* Si le pointeur est à la même adresse  */
  /* que la fin de zone mémoire de la file */
  if( cPointeur == f->cFin)
    /* On retourne le début de la zone mémoire */
    return f->cDebut;

  /* Si  le pointeur n'est pas à la même adresse que la fin de zone mémoire */
  else
     /* On avance le pointeur d'une case et on le retourne              */
     return ++cPointeur; /* Equivalent à cPointeur++; return cPointeur; */
}
/* Remarque si on appelle plusun(f,f->cQueue), f->cQueue n'est pas modifié  */
/* puisque ce n'est pas l'adresse de f->cQueue qui est passé en paramètre   */

/* Fonction de test permettant de savoir si la file est vide              */
/* Paramètres d'entrée : file f : la file traitée                         */
/* Paramètres de sortie : 0 si c'est faux, >0 sinon                       */
int vide(file f)
{ /* On retourne la valeur du test d'égalité entre l'emplacement qui suit */
  /* la queue de la file et l'adresse de la tête de la file               */
  return (plusun(f,f->cQueue) == f->cTete);
}

/* Fonction de test permettant de savoir si la file est pleine            */
/* Paramètres d'entrée : file f : la file traitée                         */
/* Paramètres de sortie : 0 si c'est faux, >0 sinon                       */
int pleine(file f)
{ /* On retourne la valeur du test d'égalité entre l'emplacement qui suit */
  /* la queue de la file  +1  et l'adresse de la tête de la file          */
  return (plusun(f,f->cQueue+1) == f->cTete);
}

/* Fonction de lecture de la tête de la file                              */
/* Paramètres d'entrée : file f : la file traitée                         */
/* Paramètres de sortie : la caractère de la tête de file                 */
char lire(file f)
{ /* Si la file est non vide                                              */
  /* NB : vide retourne 0 si la file est non vide, il faut donc tester si */
  /* vide(f) est différent de 0                                           */
  if (!vide(f))
  { /* on retrouve le contenu de la tête de file */
    return *(f->cTete);
  }
}
```

```
/* Fonction d'affichage d'un message d'erreur             */
/* Paramètres d'entrée : char *cMess : message à afficher */
/* Paramètres de sortie : rien                            */
void erreur(char* cMess)
{ /* Affichage sur l'écran du message                        */
  /* puts est une fonction prédéfinie du C ~printf("%s".,.. */
  /* mais passe automatiquement a la ligne                   */
  puts(cMess);
}

/* Fonction d'avancement de la tête d'un espace mémoire   */
/* Paramètres d'entrée : file f : la file traitée         */
/* Paramètres de sortie : rien                            */
void avancer(file f)
{ /* Si la file est vide */
  if(vide(f))
    /* Affichage d'un message d'erreur */
    erreur("file vide!");
  /* Si la file est non vide */
  else
    /* on déplace la tête d'une case mémoire */
    f->cTete = plusun(f,f->cTete);
}

/* Fonction d'ajout d'un élément dans la file                 */
/* Paramètres d'entrée : file f : la file traitée             */
/*                       char cCaractere : l'élément à ajouter */
/* Paramètres de sortie : rien                                */
void ajouter(file f, char cCaractere)
{ /* Si la file est pleine */
  if(pleine(f))
   /* Affichage d'un message d'erreur */
   erreur("File pleine!");

  /* Si la file n'est pas pleine */
  else
  { /* On déplace la queue de 1 */
    f->cQueue = plusun(f,f->cQueue);

    /* on ajoute la caractère à la queue de la file */
    *(f->cQueue) = cCaractere;
  }
}
```

```
/* Fonction d'affichage de la file                          */
/* Paramètres d'entrée : file f : la file traitée           */
/* Paramètres de sortie : rien                              */
void afficher(file f)
{ /* Déclaration d'un pointeur sur un caractère */
  char* cPointeur;

  /* le pointeur local pointe sur la tête de la file */
  cPointeur = f->cTete;

  /* si la file est vide */
  if(vide(f))
   /* on affiche () */
   puts("()");

  /* si la file est non vide */
  else
  { /* On affiche ( */
    puts("(");

    /* Boucle infinie */
    while(1)
    { /* On affiche le caractère pointé par le pointeur local */
      putchar(*cPointeur);

      /* Si Pointeur pointe sur la queue de la file */
      if(cPointeur == f->cQueue)
         /* on sort de la boucle */
         continue;
      /* Si Pointeur ne pointe sur la queue de la file,              */
      /* on déplace Pointeur de 1 (pas besoin de else car on sort de  */
      /*  la boucle si le test de if est vrai)                       */
      cPointeur  = plusun(f,cPointeur);
    } /* fin du while */

    /* On affiche ) */
    puts ("\n)");
  } /* fin du si la file est non vide (else) */
}
```

```
/*************************************************************************/
/*                          Programme principal                          */
/*************************************************************************/
void main()
{ /* Déclaration d'une file */
  file f;

  /* Création d'une file */
  f=creer(4);

  /* Lecture de la file */
  printf("%c\n",lire(f));

  /* Avancement dans la file */
  avancer(f);

  /* Ajout du caractère A */
  ajouter(f,'A');
  afficher(f);
  printf("\n");
  /* Ajout du caractère B */
  ajouter(f,'B');
  afficher(f);
   printf("\n");
  /* Ajout du caractère C */
  ajouter(f,'C');
  afficher(f);
  printf("\n");
  printf("%c\n",lire(f));

}
```