Université Mohamed Premier **Filière SMI**
Faculté des Sciences, Oujda **Semestre 4**
Département d'Informatique **Année 2016/2017**

Corrigé de l'examen de Structures de données
Session Ordinaire – Juillet 2017

Exercice 1

1. Quatre exemples de structures de données linéaires : **les tableaux, les listes chaînées, les piles et les files**.
2. On ne peut pas avoir dans une structure C nommée « X », un champ de type « X », **parce que pour une telle structure, le compilateur ne va pas savoir calculer sa taille**. Ce problème est résolu en C **en utilisant un pointeur vers une structure « X »**.
3. La définition d'une pile : **c'est une structure de données linéaire dont laquelle l'insertion et la suppression s'effectuent dans la même extrémité (le sommet de la pile) de la structure. Une pile utilise la stratégie LIFO (Last In, First Out) où le dernier élément inséré est le premier retiré.**

   La définition d'une file : **c'est une structure de données linéaire dont laquelle la suppression s'effectue dans une extrémité (la tête) de la structure et l'insertion s'effectue dans l'autre extrémité de la structure (la queue). Une file utilise la stratégie FIFO (First In, First Out) où le premier élément inséré est le premier retiré (servi).**
4. Etant donné une liste L doublement chaînée circulaire et triée. Donnons la complexité dans le pire des cas de chacun des algorithmes suivants :
   a. Le calcul du plus petit élément de L : **O(1)**.
   b. Le calcul du plus grand élément de L : **O(1)**.
   c. La suppression d'un élément de L d'adresse donnée : **O(1)**.
   d. La recherche d'un élément x dans L : **O(n)**.

Exercice 2

1. Le programme principal demandé :

```
main() {
  int i;
  Node *first, *temp, *pre;
  first = (Node*) malloc(sizeof(Node));
  first->data = 0;
  first->next = NULL;
  head = pre = first;
```

```
    for(i = 1; i <= 5; i++)  {
      temp = (Node*) malloc(sizeof(Node));
      temp->data = i * i;
      temp->next = NULL;
      pre->next = temp;
      pre = pre->next;    }
  }
```

2. Une procédure qui affiche dans l'ordre, les éléments impairs de la liste **head** :

```
  void afficherImpairs()  {
    Node *temp = head;
    while(temp != NULL)    {
      if(temp->data % 2 == 1)
        printf("%d ", temp->data);
      temp = temp->next;    }
  }
```

3. Une procédure récursive qui supprime tous les éléments pairs de la liste **head** :

```
  void supprimerPairs(List *pL)   {
    if(*pL != NULL)    {
      if((*pL)->data % 2 == 0)  {
        Node *temp = *pL;
        *pL = (*pL)->next;
        free(temp);    }
      supprimerPairs(&(*pL)->next);  }
  }
  Appel : supprimerPairs(&head) ;
```

4. Une procédure qui teste si la liste **head** contient un élément x :

```
  int contient(int x)   {
    int trouve = 0;
    Node *temp = head;
    while(!trouve && temp != NULL)   {
      if(temp->data == x)
        trouve = 1;
      else
        temp = temp->next;     }
    return trouve;
```

```
}
```

5. Une procédure qui permet de transformer la liste **head** en une liste circulaire :

```
void rendreCirculaire()  {
  if(head != NULL)    {
    Node *temp = head;
    while(temp->next != NULL)
          temp = temp->next;
    temp->next = head;  }
}
```

6. Une procédure qui affiche la longueur de **head** comme étant une liste circulaire :

```
int longueur()   {
  int count = 0;
  if(head != NULL)    {
    Node *temp = head;
    count = 1;
    while(temp->next != head)  {
      temp = temp->next;
      count++;  }
  }
  return count;
}
```

Exercice 3

1. La taille actuelle de la pile *S* est : **t = p - q, avec p est le nombre des empilements, et q est le nombre des dépilements réussis (sans échec). Comme p = 25 et q = 10 – 3 = 7, Alors t = 25 – 7 = 18**.

2. Une fonction C qui fusionne deux piles triées A et B :

```
Pile fusion(Pile A, Pile B)  {
  Pile C, D;
  C = create_empty_stack();
  D = create_empty_stack();
  while(!isEmpty(A) && !isEmpty(B))  {
    int x = top(A);
    int y = top(B);
    if(x <= y)    {
      push(x, &C);
```

```
            pop(&A); }
          else {
            push(y, &C);
            pop(&B); }
        }
        if(!isEmpty(A)) {
         while(!isEmpty(A)) {
           push(top(A), &C);
           pop(&A); } }
        if(!isEmpty(B)) {
         while(!isEmpty(B)) {
           push(top(B), &C);
           pop(&B); } }
        while(!isEmpty(C)) {
           push(top(C), &D);
           pop(&C); }
        return D;
      }
```

Exercice 4

Une fonction C qui retourne le minimum d'une file Q (supposée non vide) :

```
int getMin(File Q) {
  int min, s, i;
  min = dequeue(&Q);
  enqueue(min, &Q);
  s = size(Q);
  for(i = 1; i < s; i++) {
    int x = dequeue(&Q);
    if(x < min)
      min = x;
    enqueue(x, &Q); }
  return min;
}
```