Chapitre 3 :
Les piles et les files

Université Mohamed 1er
Faculté des sciences
Oujda

Les piles

- **Définition** :
  - Une **pile** (***stack***) est une structure de données linéaire dans laquelle les opérations d'insertion et de suppression sont réalisées à une **extrémité fixée** de la liste, appelée « **sommet de la pile** ».
  - Dans une **pile**, le **dernier élément entré** (***inséré***) est le **premier sorti** (***supprimé***).
  - Le terme « ***Last In, First Out*** » ou ***LIFO*** indique cette **stratégie** utilisée avec une **pile**.

Les piles

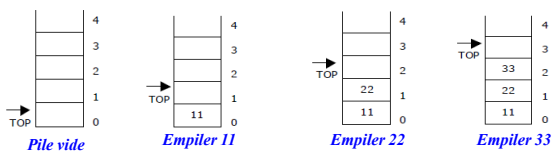
- **Insertion dans une pile** :
  - L'opération « **insérer** » dans une pile est souvent appelée « **empiler** » (**push**).
- **Suppression dans une pile** :
  - L'opération « **supprimer** » dans une pile est souvent appelée « **dépiler** » (**pop**).
- **Sommet de la pile** :
  - L'opération « **retourner sommet** » dans une pile est souvent appelée « **sommet** » (**top**).

Les piles

- **Applications des piles** :
  - Utilisées par les compilateurs pour vérifier le bon parenthésage.
  - Utilisées pour évaluer une expression postfixée.
  - Utilisées pour convertir une expression infixée en une expression postfixée/préfixée.
  - Dans une procédure ou une fonction récursive, tous les arguments intermédiaires et toutes les valeurs de retour sont stockées dans une pile.
  - Durant l'appel d'une fonction, les arguments et les adresses de retour sont empilées dans la pile et dépilées de cette pile après une instruction « return ».

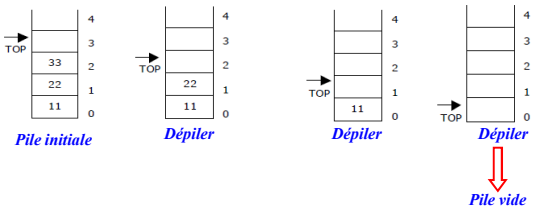
Les piles

- **Opération « empiler » :**



Les piles

- Opération « dépiler » :



Les piles

- Représentation contiguë avec un tableau (exemple : une pile d'entiers) :

```
#define STACKSIZE 100
typedef struct StackType
{
  int stack[STACKSIZE];
  int stackPointer;  // initialisé à 0
} StackType;
int isEmpty(StackType P)
{
  return P.stackPointer == 0;
}
```


Les piles

- Représentation contiguë avec un tableau (suite):

```
int isFull(StackType P)
{
  return P.stackPointer == STACKSIZE - 1;
}
void push(StackType P, int x)
{
  if(isFull(P))
    printf("Error : the stack is full !!!");
  else
    P.stack[stackPointer++] = x;
}
```


Les piles

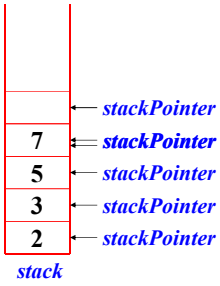
- Représentation contiguë avec un tableau (suite):

```
void pop(StackType P)  {
  if(isEmpty(P))
    printf("Error : the stack is empty !!!");
  else
    stackPointer--;
}
void top(StackType P) {
  if(isEmpty(P)) {
    printf("Error : the stack is empty !!!");
    return 0; }
  else  return stack[stackPointer - 1];  }
```


Les piles

- Représentation contiguë avec un tableau (suite):

```
main( )  {
  StackType P;
  push(P, 2);
  push(P, 3);
  push(P, 5);
  push(P, 7);
  printf("%d\n", top(P));
  pop(P);
  printf("%d\n", top(P));
}
```



Les piles

- Représentation non contiguë (exemple : une pile d'entiers) :

```
typedef struct Node
{
  int data;
  struct Node *next;
} Node;

typedef Node *stack;
```

Les piles

- **Représentation non contiguë (suite)** :

```
int isEmpty(stack P)
{
  return P == NULL;
}
void push(stack *P, int x)
{
  Node *newNode = (Node*) malloc(sizeof(Node));

  newNode->data = x;
  newNode->next = *P;
  *P = newNode;
}
```


Les piles

- **Représentation non contiguë (suite)** :

```
void pop(stack *P)
{
  if(isEmpty(*P))
    printf("Error : the stack is empty\n");
  else
  {
    Node *temp = *P;
    *P = (*P)->next;
    free(temp);
  }
}
```


Les piles

- **Représentation non contiguë (suite)** :

```
int top(stack P)
{
  if(isEmpty(P))
  {
    printf("Error : the stack is empty\n");
    return 0;
  }
  else
    return P->data;
}
```


Les piles

- **Représentation non contiguë (suite)** :

```
main( ) {
  stack P;
  push(&P, 2);
  push(&P, 3);
  push(&P, 5);
  push(&P, 7);
  printf("%d\n", top(P));
  pop(&P);
  printf("%d\n", top(P));
}
```


Les files

- **Définition** :
  - Une **file** (***queue***) est une structure de données linéaire dans laquelle **l'opération d'insertion** est réalisée à une **extrémité** de la liste (**la queue**), et **l'opération de suppression** est réalisée à **l'autre extrémité** de la liste (**la tête**).
  - Dans une **file**, le **premier élément entré** (***inséré***) est le **premier sorti** (***supprimé***).
  - Le terme « ***First In, First Out*** » ou ***FIFO*** indique cette **stratégie** utilisée avec une **file**.

Les files

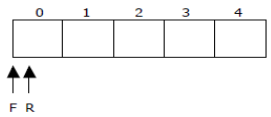
- **Insertion dans une file** :
  - L'opération « **insérer** » dans une **file** est souvent appelée « **enfiler** » (***enqueue***).
- **Suppression dans une pile** :
  - L'opération « **supprimer** » dans une **file** est souvent appelée « **défiler** » (***dequeue***).

Les files

- **Illustration d'une file** :
  - Considérons une file qui peut contenir au maximum 5 éléments.
  - La queue est repérée par un pointeur appelé « **Rear** » (= en arrière).
  - La tête est repérée par un pointeur appelé « **Front** » (= en avant).

Les files

- **Illustration d'une file** :
  - Initialement, la file est vide :



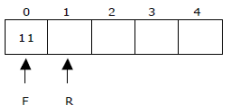
F(Front) = 0

R(Rear) = 0

File vide

Les files

- **Illustration d'une file** :
  - Maintenant, insérons 11 (en queue) :

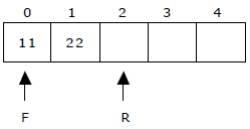


Front = 0

Rear = Rear + 1 = 1

Les files

- **Illustration d'une file** :
  - Ensuite, insérons 22 à la file :

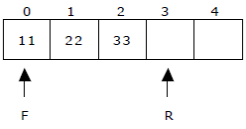


Front = 0

Rear = Rear + 1 = 2

Les files

- **Illustration d'une file** :
  - Encore une fois, insérons l'élément 33 à la file.

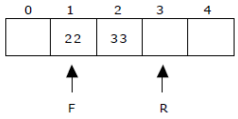


Front = 0

Rear = Rear + 1 = 3

Les files

- **Illustration d'une file** :
  - Maintenant, retirons un élément de la file.
  - L'élément à retiré est l'élément qui se trouve en tête de la file (***Front***).
  - L'état de la file sera alors :

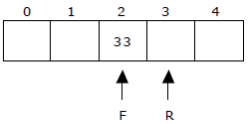


Front = Front + 1 = 1

Rear = 3

Les files

- **Illustration d'une file** :
  - Retirons encore un élément de la file.
  - L'élément à retiré est l'élément qui se trouve en tête de la file (l'élément 22) :

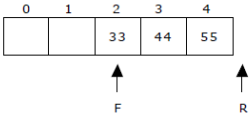


Front = Front + 1 = 2

Rear = 3

Les files

- **Illustration d'une file** :
  - Maintenant, insérons les éléments 44 et 55 :



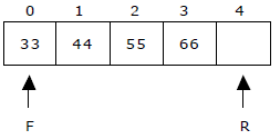
Front = 2

Rear = 5

Les files

- **Illustration d'une file** :
  - Maintenant, il est impossible d'insérer un élément 66 **même s'il y a deux positions vacantes en tête de la file**.
  - Pour régler ce problème, on peut **décaler les éléments de la file en tête** (**Front**) de celle-ci pour créer des positions vacantes en queue de la file : les valeurs de **Front** et **Rear** doivent être **ajustées correctement**.
  - L'élément 66 peut maintenant être inséré en queue.

Les files



Front = 0

Rear = 4

Les files

- **Représentation d'une file par un tableau** :

```
# include <stdio.h>

# define SIZE 5

typedef struct File
{
  int contenu[SIZE];
  int tete, queue;  /* Front et Rear */
} File;
```

Les files

- Opérations (sans ajustement des positions vacantes) :

```
void initialiser(File* F)
{
  F->tete = F->queue = 0;
}
int estVide(File F)
{
  return (F.queue == 0);
}
```


Les files

- Opérations (sans ajustement des positions vacantes) :

```
int estPleine(File F)
{
  return (F.queue == SIZE);
}
void enfiler(File* F, int x)
{
  if(estPleine(*F))
    printf("\n\n Erreur : la file est vide !!!");
  else
    F->contenu[F->queue++] = x;
}
```


Les files

- Opérations (sans ajustement des positions vacantes) :

```
void defiler(File* F)
{
  if(estVide(*F))
    printf("\n\n Erreur : la file est vide !!!");
  else
    F->tete++;
}
```


Les files

- Opérations (sans ajustement des positions vacantes) :

```
void afficher(File F)
{
  if(!estVide(F)) {
      int i;
      for(i = F.tete; i < F.queue; i++)
          printf("%d ", F.contenu[i]);
      puts("");    }
}
```


Les files

- Opérations (sans ajustement des positions vacantes) :

```
main ( )  {
  File F;
  initialiser(&F);
  enfiler(&F, 11);  afficher(F);  enfiler(&F, 22);
  afficher(F);      enfiler(&F, 33);  afficher(F);
  defiler(&F);      afficher(F);      defiler(&F);
  afficher(F);      enfiler(&F, 44);  afficher(F);
  enfiler(&F, 55);  afficher(F);  enfiler(&F, 66);
  afficher(F);  system("pause");   }
```


Les files

- Opérations (avec ajustement des positions vacantes) :
  - Solution par décalage à gauche :
    - ✓ Tous les éléments **reculent d'une place lorsque le premier est servi** (supprimé).
    - ✓ Le pointeur de tête (***Front***) est alors **inutile**, puisqu'il est toujours égal à 0.
    - ✓ Par conséquent, la file est gérée par un tableau et un seul pointeur (***Rear***) et **lorsqu'un élément est supprimé**, les **autres éléments restants** sont **décalés à gauche** et la valeur de « ***Rear*** » est décrémentée.

Les files

- Opérations (avec ajustement des positions vacantes) :
  - Solution par décalage à gauche :

```
# include <stdio.h>
# define SIZE 5
typedef struct File
{
  int contenu[SIZE];
  int queue; /* Pas besoin de Front (tete) */
} File;
```


Les files

- Opérations (avec ajustement des positions vacantes) :
  - Solution par décalage à gauche :

```
void initialiser(File* F) {
  F->queue = 0;  }
int estVide(File F)   {
  return (F.queue == 0);
}
int estPleine(File F)  {
  return (F.queue == SIZE);
}
```


Les files

- Opérations (avec ajustement des positions vacantes) :
  - Solution par décalage à gauche :

```
void enfiler(File* F, int x)
{
  if(estPleine(*F))  {
    printf("File pleine : ");
    printf("impossible d'enfiler %d\n", x);  }
  else
    F->contenu[F->queue++] = x;
}
```


Les files

- Opérations (avec ajustement des positions vacantes) : Solution par décalage à gauche

```
void defiler(File* F)   {
  if(estVide(*F))
    printf("File vide : impossible de defiler la file\n");
  else   {
    int i;    /* décalage des éléments restants vers la tête */
    for(i = 0; i < F->queue - 1; i++)
      F->contenu[i] = F->contenu[i + 1];
    F->queue--;  /* reculer la queue */        }
}
```


Les files

- Opérations (avec ajustement des positions vacantes) :
  - Solution par décalage à gauche :

```
void afficher(File* F)
{
  if(!estVide(F))  {
      int i;
      for(i = 0; i < F.queue; i++)
          printf("%d ", F.contenu[i]);
      puts("");    }
}
```


Les files

- Opérations (avec ajustement des positions vacantes) :
  - Solution par tableau « circulaire » :
    - ✓ **Efficace** : car elle **minimise la place nécessaire dans le tableau**, tout **en évitant les opérations de décalage**.
    - ✓ Lorsque la tête (**Front**) devient égale à la queue (**Rear**) (**modulo N**), la file est soit vide, soit pleine.
    - ✓ On est donc amené à distinguer les 2 possibilités en utilisant deux indicateurs booléens « **vide** » et « **pleine** ».

Les files

- **Solution par tableau circulaire :**
  - Au départ la file est vide :
    - ✓ *pleine := 0 et vide := 1*
  - La procédure « **enfiler**(x) » :
    - ✓ Si la file est **pleine** : impossible d'enfiler x.
    - ✓ Sinon (la file n'est pas pleine), il y a deux cas :
      - Si *queue* < *N*, alors on **enfile** x dans la case **queue**, et l'on **incrémente queue**.

Les files

- **Solution par tableau circulaire :**
  - Si *queue* = *N*, alors on **enfile** x dans la case **0**, et l'on **incrémente queue**.
  - ✓ On peut regrouper les deux cas en un seul :
    - *queue := queue % N;*
    - On **enfile** x dans la case **queue**, et l'on **incrémente queue**.
  - ✓ La **file** est certainement **non vide** :
    - *vide := 0*

Les files

- **Solution par tableau circulaire :**
  - ✓ Finalement, on teste si la **file est pleine** :
    - *pliene := (queue < N et tete == queue) ou (queue = N et tete = 0)*

Les files

- **Solution par tableau circulaire :**
  - La procédure « **défiler**() » :
    - ✓ Si la file est **vide** : impossible d'enfiler x.
    - ✓ Sinon (la file n'est pas vide), il y a deux cas :
      - Si *tete* < *N*, alors on l'on **incrémente**.
      - Si *tete* = *N*, alors **tete** := 1.
    - ✓ On peut regrouper les deux cas en un seul :
      - *tete := tete % N;*
      - on **incrémente tete**

Les files

- **Solution par tableau circulaire :**
  - ✓ La **file** est certainement **non pleine** :
    - *pleine := 0*
  - ✓ On teste si la file **est vide** :
    - *vide := (tete < N et tete == queue) ou (tete = N et queue = 0)*

Les files

- **Solution par tableau circulaire :**

```
#include <stdio.h>
#define SIZE 5
typedef struct File
{
  int contenu[SIZE];
  int tete, queue;
  int vide, pleine;
} File;
```

Les files

- **Solution par tableau circulaire :**

```
void initialiser(File* F) {
  F->tete = F->queue = 0;
  F->vide = 1;
  F->pleine = 0;  }
int estVide(File F)  {
  return F.vide;  }
int isFull(File F)  {
  return F.pleine;
}
```


Les files

- **Solution par tableau circulaire :**

```
void enfiler(File* F, int x)  {
  if(estPleine(*F))
    printf("Erreur : la file est pleine !!!\n");
  else  {
    F->queue = F->queue % SIZE;
    F->contenu[F->queue++] = x;
    F->vide = 0;
    F->pleine = (F->tete == 0 && F->queue ==
    SIZE)  ||  (F->queue < SIZE && F->tete ==
    F->queue);  } }
```


Les files

- **Solution par tableau circulaire :**

```
void defiler(File* F)  {
  if(estVide(*F))
    printf("Erreur : la file est vide !!!\n");
  else  {
    F->tete = F->tete % SIZE;
    F->tete++;
    F->pleine = 0;
    F->vide = (F->tete == SIZE && F->queue == 0)
   || (F->tete < SIZE && F->tete == F->queue);  }
}
```


Les files

- **Solution par tableau circulaire :**

```
void afficher(File F)
{
  int i;
  if(estVide(F))
    printf("La file est vide\n");
  else
  {
    if(F.tete < F.queue)
    {
      for(i = F.tete; i < F.queue; i++)
```

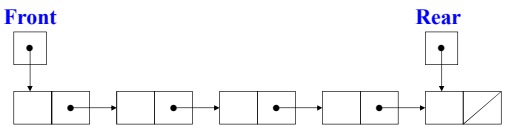

Les files

- **Solution par tableau circulaire :**

```
        printf("%d ", F.contenu[i]);
      puts("");  }
    else if(F.tete >= F.queue)  {
      for(i = F.tete; i < SIZE; i++)
        printf("%d ", F.contenu[i]);
      for(i = 0; i < F.queue; i++)
        printf("%d ", F.contenu[i]);
      puts("");  }
  }
}
```


Les files

- **Représentation d'une file par une liste chaînée** :
  - C'est une représentation classique de liste chaînée, dans laquelle on utilise un **pointeur supplémentaire** : le **pointeur vers la dernière cellule**, ce qui permet d'accélérer l'algorithme d'insertion à la fin de la liste.



Les files

- Représentation d'une file par une liste chaînée :

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

typedef struct Noeud
{
  int info;
  struct Noeud* suiv;
} Noeud;
```


Les files

- Représentation d'une file par une liste chaînée :

```
typedef struct File
{
  Noeud* tete;
  Noeud* queue;
} File;
```


Les files

- Représentation d'une file par une liste chaînée :

```
void initialiser(File* F)
{
  F->tete = NULL;
  F->queue = NULL;
}
int estVide(File F)
{
  return F.tete == NULL;
}
```


Les files

- Représentation d'une file par une liste chaînée :

```
void enfiler(File* F, int x)
{
  Noeud* temp = (Noeud*) malloc(sizeof(Noeud));

  if(temp == NULL)
    printf("Erreur : impossible d'enfiler %d\n", x);
  else
  {
    temp->info = x;
    temp->suiv = NULL;
```


Les files

- Représentation d'une file par une liste chaînée :

```
    if(estVide(*F))
      F->tete = F->queue = temp;
    else
    {
      F->queue->suiv = temp;
      F->queue = temp;
    }
    printf("L'element %d est enfile\n", x);
  }
}
```


Les files

- Représentation d'une file par une liste chaînée :

```
void defiler(File* F)  {
  if(estVide(*F))
    printf("Erreur : la file est vide !!!\n");
  else
  {
    Noeud* temp = F->tete;
    printf("L'element %d est defile\n", temp->info);
    F->tete = F->tete->suiv;
    free(temp);
  }
}
```

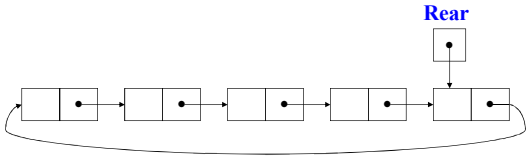
Les files

- **Représentation d'une file par une liste chaînée** :

```
void afficher(File F)
{
  Noeud* temp;
  temp = F.tete;
  while(temp != NULL)
  {
    printf("%d ", temp->info);
    temp = temp->suiv;
  }
  puts("");
}
```


Les files

- **Représentation d'une file par une liste circulaire**:
  - Dans ce cas, la file sera représentée par un **pointeur vers la dernière cellule**.



Les files

- **Représentation d'une file par une liste circulaire**:

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

typedef struct Noeud
{
  int info;
  struct Noeud* suiv;
} Noeud;

typedef Noeud* File;
```


Les files

- **Représentation d'une file par une liste circulaire**:

```
void initialiser(File* F)
{
  *F = NULL;
}

int estVide(File F)
{
  return F == NULL;
}
```


Les files

- **Représentation d'une file par une liste circulaire**:

```
void enfiler(File* F, int x) {
  Noeud* temp = (Noeud*) malloc(sizeof(Noeud));
  temp->info = x;
  if(estVide(*F)) {
    *F = temp;
    temp->suiv = temp; }
  else {
    temp->suiv = (*F)->suiv;
    (*F)->suiv = temp;
    *F = temp;          }
}
```


Les files

- **Représentation d'une file par une liste circulaire**:

```
void defiler(File* F)  {
  if(estVide(*F))
    printf("Erreur : la file est vide !!!\n");
  else {
    if((*F)->suiv == *F) {
      free(*F);
      initialiser(F);      }
    else    {
      Noeud* temp = (*F)->suiv;
      (*F)->suiv = temp->suiv;
      free(temp);  } }
}
```

Les files

- **Représentation d'une file par une liste circulaire**:

```
void afficher(File F)  {
 if(estVide(F))
   puts("File vide");
 else  {
    Noeud* temp = F->suiv;
    while(temp != F)  {
      printf("%d ", temp->info);
      temp = temp->suiv;  }
    printf("%d\n", temp->info); }
}
```