**TD Algorithmique: Pile, File, Tas**

Dans les exercices suivants, le travail demandé consiste à implementer les opérations d'une pile et celles d'une file en C++. Toutes les opérations doivent être testées pour assurer leur bon fonctionnement.

## 1. Pile

Une pile est une liste équipée de deux opérations qui ajoutent un élément à la fin de la liste et suppriment un élément à la fin de la liste. Si on a une pile de disques, chaque disque placé en dernier dans la pile sera au-dessus de tous les autres disques et sera supprimé en premier. En raison du principe « dernier entré, premier sorti », la pile est également connue sous le nom de liste LIFO (Last In First Out) et la dernière position de la liste est appelée le sommet de la pile.

### 1.1. Implémenter une pile à l'aide de tableaux

Les opérations d'une pile fonctionnent de la manière suivante:

- Ajouter un élément à la pile équivaut à ajouter un élément à la fin du tableau.
- Supprimer un élément de la pile équivaut à supprimer un élément de la fin du tableau.
- La pile déborde lors de l'ajout d'un élément à un tableau déjà plein.
- La pile est vide lorsque le nombre d'éléments réellement contenus dans le tableau est égale à 0.

```
program StackByArray;
const
max = 10000;
var
Stack: array[1..max] of Integer;
Top: Integer; {Stocker l'index du dernier élément de la pile}
procedure StackInit; {Initialiser une pile vide}
begin
Top := 0;
end;
procedure Push(V: Integer); {Insérer une valeur V dans la pile}
begin
if Top = max then WriteLn('Stack is full') {Si la pile est pleine, aucun élément supplémentaire ne peut être ajouté}
```

```
else
begin
Inc(Top); Stack[Top] := V; {Sinon, ajouter un élément à la fin du tableau}
end;
end;
function Pop: Integer; {Supprimer une valeur de la pile, renvoyez-la au résultat de la fonction}
begin
if Top = 0 then WriteLn('Stack is empty') {Si la pile est vide, elle ne peut pas être récupérée}
else
begin
Pop := Stack[Top]; Dec(Top); {Récupérer le dernier élément du tableau}
end;
end;
begin
StackInit;
〈Test〉; {Quelques commandes pour tester le fonctionnement de la pile}
end
```

**1.2. Implémenter une pile à l'aide d'une liste chaînée LIFO**

Une liste chaînée peut être également utilisée pour implémenter une pile. Les opérations d'une pile comme celles dans la section précédente sont illustrées par le programme suivant.

```
program StackByLinkedList;
type
PNode = ^TNode; {Pointeur vers un nœud de la liste}
TNode = record {Structure d'un nœud de la liste}
Value: Integer;
Link: PNode;
end;
var
Top: PNode; {Pointeur vers la tête}
procedure StackInit; {Initialiser une pile vide}
```

```
begin
Top := nil;
end;
procedure Push(V: Integer); {Insérer la valeur V dans la pile ⇔ Ajouter un nouveau nœud contenant V et ajoutez-le à la liste}
var
P: PNode;
begin
New(P); P^.Value := V; {Créer un nouveau nœud}
P^.Link := Top; Top := P; {Insérez ce nœud dans la liste}
end;
function Pop: Integer; {Récupérer une valeur de la pile, renvoyez-la au résultat de la fonction}
var
P: PNode;
begin
if Top = nil then WriteLn('Stack is empty')
else
begin
Pop := Top^.Value;
P := Top^.Link; {Conserver le nœud suivant Top^ (le nœud inséré dans la liste avant le nœud Top^)}
Dispose(Top); Top := P; {Libérer la mémoire allouée à Top^, metter à jour Top}
end;
end;
begin
StackInit;
〈Test〉; {Quelques commandes pour tester le fonctionnement de la pile}
end.
```

**2. File**

Une file (file d'attente) est une liste équipée de deux opérations qui ajoutent un élément à la fin de la liste (Rear) et supprimez un élément au début de la liste (Front).

Nous pouvons imaginer une file d'attente comme un groupe de personnes faisant la queue pour acheter des billets : celui qui fait la queue en premier va pouvoir acheter des billets en premier. En raison de ce principe « premier entré, premier sorti », la file d'attente est également appelée FIFO (First In First Out).

**2.1. Implémenter une file d'attente à l'aide de tableaux**

Nous avons deux indices Front et Rear: Front stocke l'indice du premier élément, Rear stockent l'indice du dernier élément. Les opérations d'une file fonctionnent de la manière suivante:

- Initialiser la file vide : Front := 1 et Rear := 0.
- Pour ajouter un élément à la file d'attente, nous augmentons Rear de 1 et mettons cet élément à l'indice Rear.
- Pour supprimer un élément de la file, nous prenons la valeur à l'indice Front et augmentons Front de 1.
- Lorsque Rear augmente jusqu'à la fin du tableau, le tableau est pleine et ne peut plus être ajouté.
- Lorsque Front > Rear, cela signifie que la file d'attente est vide.

```
program QueueByArray;
const
max = 10000;
var
Queue: array[1..max] of Integer;
Front, Rear: Integer;
procedure QueueInit; {Initialiser une file d'attente vide}
begin
Front := 1; Rear := 0;
end;
procedure Push(V: Integer); {Ajouter V dans la file}
begin
if Rear = max then WriteLn('Overflow')
else
```

```
begin
Inc(Rear);
Queue[Rear] := V;
end;
end;
function Pop: Integer; {Prendre une valeur de la file d'attente, renvoyez-la au résultat de la fonction}
begin
if Front > Rear then WriteLn('Queue is Empty')
else
begin
Pop := Queue[Front];
Inc(Front);
end;
end;
begin
QueueInit;
〈Test〉; {Donnez quelques commandes pour tester le fonctionnement de la file}
end.
```

**2.2. Implémenter une file d'attente à l'aide d'une liste circulaire**

En examinant le programme d'une pile avec un tableau de taille maximale de 10000 éléments, nous voyons que si on fait 6000 Push, puis 6000 Pop, puis 6000 Push, il n'y a pas de problème. La raison est que l'indice Top augmente jusqu'à 6000 puis diminue à 0, puis augmente à nouveau jusqu'à 6000. Mais pour l'implémentation de la file d'attente comme ci-dessus, vous rencontrerez l'erreur de débordement de tableau, car à chaque Push, l'indice à la fin de la file d'attente augmente toujours. Le problème est qu'il n'y a que des éléments aux positions Front et Rear qui appartient à la file d'attente.

Pour résoudre ce problème, nous décrivons la file d'attente avec une liste circulaire (représentée par un tableau où une liste) en considérant que les éléments sont disposés autour du tableau dans une certaine direction. Les éléments situés sur l'arc allant de la position Front à la position Rear sont des éléments de la file d'attente. Il faut avoir une variable supplémentaire n qui stocke le nombre d'éléments dans la file d'attente. L'ajout d'un élément à la file d'attente est fait en déplaçant Rear d'une position, puis y mettons

la nouvelle valeur. La suppression d'un élément de la file d'attente est faite en prenant la valeur à la position Front et en déplaçant Front dans la bonne direction.

Notez que lors des opérations Push et Pop, vous devez vérifier si la file d'attente déborde ou si elle est vide, la variable n doit donc être mise à jour (Ici, nous utilisons la variable supplémentaire n pour faciliter la compréhension, mais en réalité nous n'avons besoin que de deux variables Front et Rear pour vérifier si la file d'attente déborde).

```
program QueueByCList;
const
max = 10000;
var
Queue: array[0..max - 1] of Integer;
i, n, Front, Rear: Integer;
procedure QueueInit; {Initialiser une file d'attente vide}
begin
Front := 0; Rear := max - 1; n := 0;
end;
procedure Push(V: Integer); {Ajouter V dans la file}
begin
if n = max then WriteLn('Queue is Full')
else
begin
Rear := (Rear + 1) mod max; {Rear se déplace sur un cercle}
Queue[Rear] := V;
Inc(n);
end;
end;
function Pop: Integer; {Prendre un élément de la file d'attente, renvoyez-le au résultat de la fonction}
begin
if n = 0 then WriteLn('Queue is Empty')
else
begin
```

```
Pop := Queue[Front];
Front := (Front + 1) mod max; {Front se déplace sur un cercle}
Dec(n);
end;
end;
begin
QueueInit;
〈Test〉; {Donnez quelques commandes pour tester le fonctionnement de la file}
end
```

**2.3. Implémenter une file d'attente à l'aide d'une liste chaînée FIFO**

Comme dans l'implémentation d'une pile à l'aide d'une liste chaînée LIFO, nous ne vérifions pas si une file d'attente déborde dans le cas où elle est décrite par une liste chaînée FIFO.

```
program QueueByLinkedList;
type
PNode = ^TNode; {Pointeur vers un nœud de liste}
TNode = record {Structure d'un nœud}
Value: Integer;
Link: PNode;
end;
var
Front, Rear: PNode; {Deux pointeurs vers le premier nœud et le dernier nœud de la liste}
procedure QueueInit; {Initialiser une file d'attente vide}
begin
Front := nil;
end;
procedure Push(V: Integer); {Ajouter V dans la file}
var
P: PNode;
begin
New(P); P^.Value := V; {Créer un nouveau nœud}
```

```
P^.Link := nil;
if Front = nil then Front := P {Insérer le nœud dans la file}
else Rear^.Link := P;
Rear := P; {Le nœud devient le dernier élément, mettre à jour Rear}
end;
function Pop: Integer; {Prendre un élément de la file, renvoyez-le au résultat de la fonction}
var
P: PNode;
begin
if Front = nil then WriteLn('Queue is empty')
else
begin
Pop := Front^.Value;
P := Front^.Link; {Garder le nœud suivant Front^ (le nœud est inséré dans la file juste après Front^)}
Dispose(Front); Front := P; {Libérer la mémoire allouée à Front^, mette à jour Front}
end;
end;
begin
QueueInit;
〈Test〉; {Donnez quelques commandes pour tester le fonctionnement de la file}
end.
```

**3. Tas binaire**

Un tas est une structure de données arborescente, et tous les nœuds de cet arbre sont disposés dans un certain ordre, ascendant ou décroissant.

Supposons que nous ayons A comme nœud parent de B. Si la valeur du nœud A est supérieure à la valeur du nœud B alors cette relation s'applique également à l'ensemble de l'arbre. Cela signifie que la valeur du nœud B sera supérieure à la valeur de son nœud enfant et ainsi de suite, cet ordre est appliqué à l'ensemble de l'arborescence.

Le nombre maximum d'enfants d'un nœud dans un tas dépend du type de tas. En réalité, nous avons de nombreux types de tas, mais le type le plus couramment utilisé est le tas binaire.

- Tas-Max: la valeur de chaque nœud est supérieure ou égale aux valeurs de ses enfants, garantissant que le nœud racine contient la valeur maximale. Lorsque vous descendez dans l'arborescence, les valeurs diminuent.
- Tas-Min: la valeur de chaque nœud est inférieure ou égale aux valeurs de ses enfants, garantissant que le nœud racine contient la valeur minimale. Lorsque vous descendez dans l'arborescence, les valeurs augmentent.

Nous considérons les opérations suivantes d'un tas:

- Insérer : ajoute un nouvel élément au tas tout en conservant la propriété du tas.
- Extraire Max/Min : supprime l'élément maximum ou minimum du tas et le renvoie.

Considérons maintenant une suite de nombres entiers 27, 12, 8, 45, 17, 63, 100, 5, 75, 90, 26, 32, 188, 95 et un tas-max.

3.1. Dessiner le tas après chaque ajout de ces nombres dans leur ordre.

3.2. Dessiner le tas après chaque suppression de ces nombres du dernier tas obtenu en 3.1.

3.3. Implémenter les deux opérations Insérer et Extraire Max et testez-les.

3.4. Les mêmes questions que celles dans 3.1, 3.2, 3.3 mais avec un tas-min.