ITI 1521. Introduction à l'informatique II*

Marcel Turcotte
École d'ingénierie et de technologie de l'information

Version du 7 mars 2011

Résumé

– Files
  – ArrayQueue

*. Ces notes de cours ont été conçues afin d'être visualiser sur un écran d'ordinateur.

Définitions

Une **file** (*queue*) est un **type abstrait de données** linéaire tel que l'ajout de données se fait à une extrémité, l'**arrière** (*rear*) de la file, et le retrait à l'autre, l'**avant** (*front*).

Ces structures de données sont dites FIFO : *first-in first-out*.

`enqueue()` ⇒ Queue ⇒ `dequeue()`

Les deux opérations de base sont :

enqueue : l'ajout d'un élément à l'arrière de la file,

dequeue : le retrait d'un élément à l'avant de la file.

⇒ Les files sont donc des structures de données semblables aux files d'attente au supermarché, à la banque, au cinéma, etc.

TAD

```
public interface Queue<E> {
    public abstract boolean isEmpty();
    public abstract void enqueue( E o );
    public abstract E dequeue();
}
```

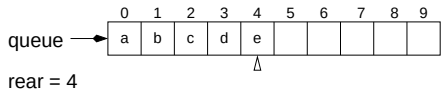
Implémentation à l'aide d'un tableau

```
public class ArrayQueue<E> implements Queue<E> {

    private E[] elems;

    public boolean isEmpty() { ... }
    public void enqueue( E o ) { ... }
    public E dequeue() { ... }

}
```

Suggestions pour la ou les variables d'instance supplémentaire ?

Implémenter une file à l'aide d'un tableau

Implémentation 1. L'avant de la file est fixe, en position 0 par exemple, et on utilise une variable qui pointe vers l'arrière de la file, *rear*.



⇒ Contrairement à l'implémentation des piles, l'implémentation des files à l'aide de tableaux va causer certains problèmes.

Implémentation à l'aide d'un tableau

```
public class ArrayQueue<E> implements Queue<E> {

    private E[] elems;
    private int rear;

    public boolean isEmpty() { ... }
    public void enqueue( E o ) { ... }
    public E dequeue() { ... }

}
```

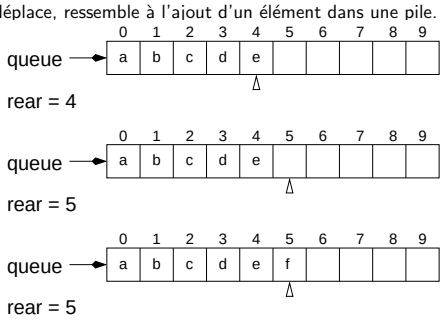
Remarques

– Tout comme nous l'avons fait pour les piles, nous pourrions :
  – faire pointer *rear* vers la première cellule libre ;
  – utiliser la partie haute du tableau ;
  – utiliser la technique du tableau dynamique.

⇒ Cependant les conclusions à tirer quand à l'efficacité des algorithmes d'insertion et de retrait demeureraient les mêmes.

Insertion (enqueue)

L'insertion d'un nouvel élément dans une file, lorsque l'avant de la file est fixe et l'arrière se déplace, ressemble à l'ajout d'un élément dans une pile.



⇒ *(i)* La variable `rear` est incrémentée de 1, puis *(ii)* la valeur est mise à la position `rear` du tableau.

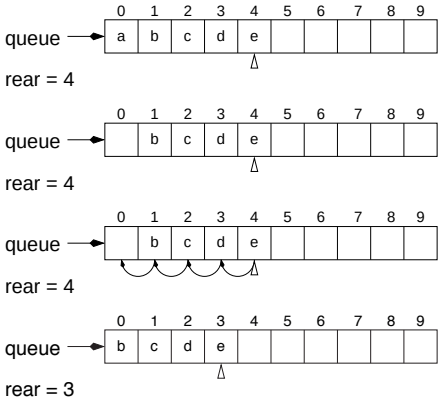
Retrait (dequeue)

Qu'en est-il du retrait d'un élément ?

Suite au retrait de l'élément, il faut déplacer tous les éléments d'une position vers la gauche afin que l'avant de la file demeure en position fixe, 0.

1. Sauvegarde dans une variable temporaire de la valeur qui se trouve à l'avant de la file ;
2. Pour i de 1 à *rear* déplacer la valeur à la position i vers la position i-1 ;
3. Initialiser la cellule *rear* du tableau ;
4. Décrémenter de 1 la variable *rear* ;
5. Retourner la valeur sauvegardée.

Retrait (dequeue)



Retrait (dequeue)

C'est donc dire que pour chaque retrait, il faut déplacer $n-1$ valeurs, s'il y avait n valeurs avant le retrait.

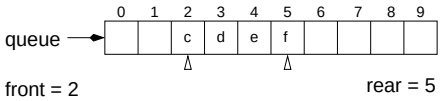
Plus il y a d'éléments dans la file, plus il y a de déplacements à faire. Si on double le nombre d'éléments dans la file, le retrait d'un élément va nécessiter 2 fois plus de déplacements.

Ce qui n'est pas le cas pour les ajouts, un ajout se fait toujours à la première position libre du tableau. L'insertion d'un élément dans une file contenant 2 fois plus de données ne nécessite pas plus de travail (opérations).

⇒ **Peut-on faire mieux ?**

Implémenter une file à l'aide d'un tableau

Implémentation 2. L'avant et l'arrière de la file se déplacent, il faut donc utiliser une variable qui pointe vers l'avant, *front*, et une autre qui pointe vers l'arrière, *rear*.



Implémentation à l'aide d'un tableau

```
public class ArrayQueue<E> implements Queue<E> {

    private E[] elems;
    private int front;
    private int rear;

    public boolean isEmpty() { ... }
    public void enqueue( E o ) { ... }
    public E dequeue() { ... }

}
```

Insertion (enqueue)

Qu'advient-il de l'insertion d'un élément ?

C'est comme avant :

1. Incrémenter la valeur de la variable *rear*,
2. Insérer la nouvelle valeur à la position *rear*.

Cette nouvelle implémentation nous permet de retirer un élément de façon efficace, c'est-à-dire que le temps nécessaire ne dépend plus du nombre d'éléments dans la file.

Mais le prix à payer est que lorsque la valeur de *rear* atteint la limite du tableau (mais que *front* n'est pas 0, autrement dit la file n'est pas pleine, elle ne s'est que déplacée vers la droite) il faut alors la repositionner à la gauche du tableau, c'est-à-dire qu'il faut déplacer tous ses éléments.

Plus la file contient d'éléments au moment de l'ajout, plus il y a d'éléments à dépalcer. S'il y a 2 fois plus d'éléments dans la file, il faudra déplacer 2 fois plus d'éléments.

Dans le cadre de l'implémentation 1, l'ajout d'éléments est efficace, mais le retrait est lent.

Dans le cadre de l'implémentation 2, c'est le contraire, l'ajout d'éléments est coûteux (lorsque la file s'est déplacée vers la droite), mais le retrait est efficace.

⇒ **Peut-on obtenir un retrait et un ajout qui soient tous le deux efficaces ?**

Retrait (dequeue)

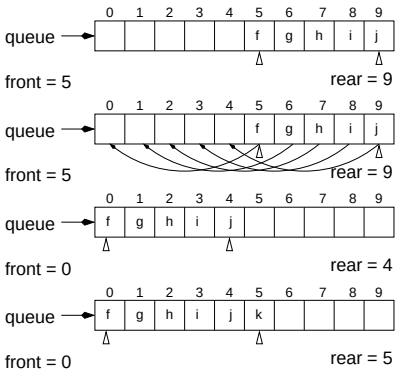
Afin de retirer un élément, on a maintenant *(i)* qu'à sauver la valeur qui se trouve à l'avant dans une variable temporaire, *(ii)* incrémenter avant et *(ii)* retourner la valeur sauvée.

Le retrait d'un élément se fait maintenant toujours en temps constant, c'est-à-dire que peut importe le nombre d'éléments présents dans la file, il n'y a que les trois opérations ci-dessus à faire (et possiblement initialiser la cellule libre à *null*, s'il s'agit d'une file de valeurs références).

Mission accomplie ?

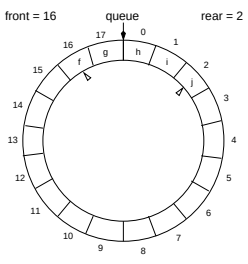
Anticipez-vous un problème avec une telle implémentation ?

C'est vrai, sauf si l'arrière atteint la fin du tableau et que l'avant n'est pas en position 0, il faut alors déplacer tous les éléments vers la gauche.



Implémenter une file à l'aide d'un tableau

Implémentation 3. Afin que les 2 opérations de base, le retrait et l'insertion, soient efficaces, nous utiliserons un **tableau circulaire**. L'avant et l'arrière de la file se déplaceront tous les deux, il faut donc utiliser une variable qui pointe vers l'avant, *front*, et une autre qui pointe vers l'arrière, *rear*.



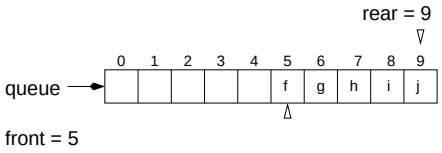
Ainsi, afin l'insérer une valeur il faut : *(i)* incrémenter la valeur *rear*, *(ii)* puis insérer la nouvelle valeur à la position *rear*.

De même, afin de retirer un élément il faut : sauver la valeur courante trouvée à la position *front*, re-initialiser cette position du tableau, incrémenter la valeur de *front* et retourner la valeur sauvée.

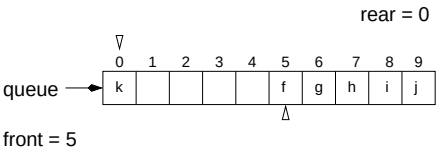
Tableau circulaire

Comment implémente-t-on un tableau circulaire ?

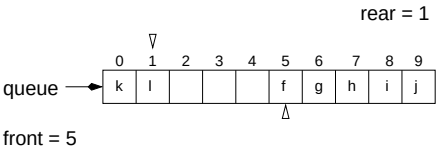
L'idée est la suivante, lorsque l'arrière de la file a atteint l'extrémité droite du tableau, et qu'il y a des cases libres dans sa partie gauche, alors on recommence à insérer des éléments au début du tableau.



front = 5



front = 5



front = 5

Diagramme de mémoire

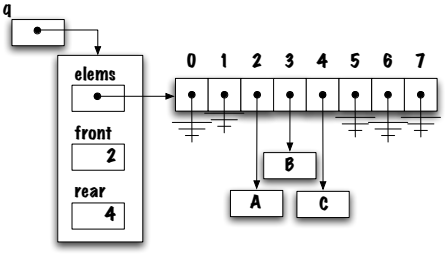
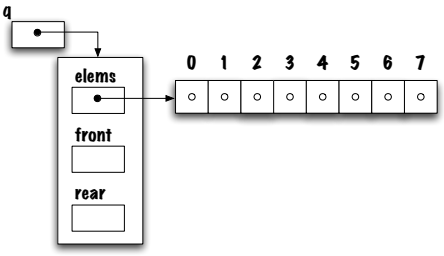


Diagramme de mémoire



Mais encore, comment écrit-on la méthode *enqueue* par exemple ?

On pourrait ajouter un test tel que celui-ci :

```
rear = rear + 1;
if ( rear == MAX_QUEUE_SIZE ) {
    rear = 0;
}
```

Alternative ?

On utilise plutôt l'arithmétique modulo qui simplifie l'écriture. [1],

```
rear = (rear + 1) % MAX_QUEUE_SIZE;
```

1. c'est ce qu'on faisait dans la classe Time afin de s'assurer que la valeur des heures revienne à zéro lorsque que sa valeur excédait 23.

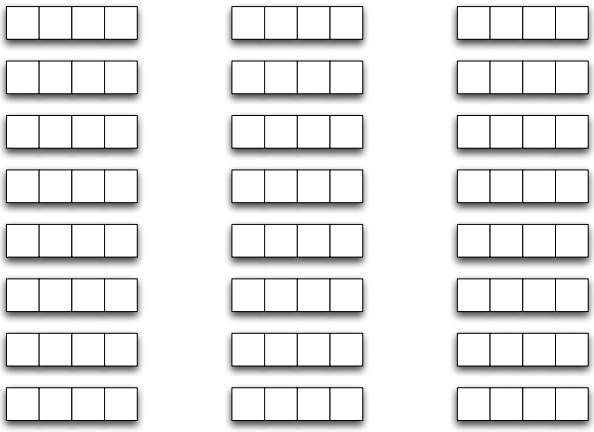
Insertion (enqueue)

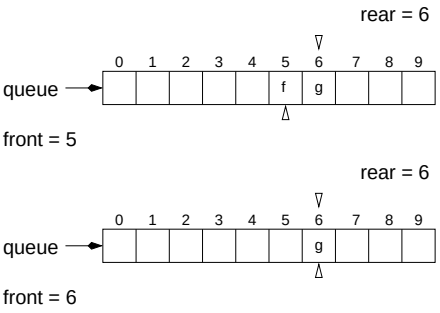
1. `rear = (rear+1) % MAX_QUEUE_SIZE;`
2. ajouter le nouvel élément à la position `rear`.

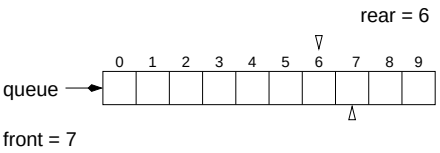
Retrait (dequeue)

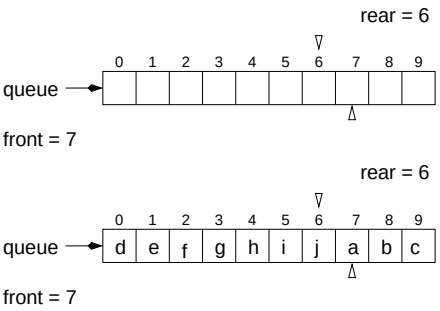
1. sauvegarder la valeur à l'avant de la file ;
2. re-initialiser la valeur à la position `front` du tableau ;
3. `front = (front+1) % MAX_QUEUE_SIZE;`
4. retourner la valeur sauvegardée.

⇒ Que se passe-t-il lorsque le pile est vide ?









On voit donc qu'il est impossible de distinguer une file vide d'une file pleine sur la base des variables *rear* et *front*.

Que faire ?

Il existe plusieurs alternatives possibles : détruire le tableau lorsque la file est vide, utiliser un booléen ou une sentinelle (-1) comme valeur pour *rear* ou *front*.

Une autre solution consiste à utiliser une variable d'instance afin de compter le nombre d'éléments dans la file et de choisir judicieusement les valeurs de *rear* et *front* : par exemple 0 et 1 ou encore `MAX_QUEUE_SIZE` et 0.

Nous utiliserons une sentinelle pour la valeur de l'index arrière (rear).

```
public class CircularQueue<E> implements Queue<E> {

    private static final int MAX_QUEUE_SIZE = 100;

    private E[] elems;
    private int front, rear;

    public CircularQueue() {
        elems = (E []) new Object[ MAX_QUEUE_SIZE ];
        front = 0; //
        rear = -1; // indique que la file est vide
    }

    // ...
```

```
public boolean isEmpty() {
    return ( rear == -1 ) ;
}
```

```
public boolean isFull() {


}
```

```
// fonctionne meme pour la file vide, pourquoi?

public void enqueue( E o ) {
    rear = ( rear+1 ) % MAX_QUEUE_SIZE;
    q[ rear ] = o;
}

public E peek() {
    return q[ front ];
}
```

```
public E dequeue() {

}
```

Insertion (enqueue)

Une autre façon de résoudre ce problème aurait été l'utilisation d'une variable d'instance pour compter les éléments.

1. `rear = ( rear+1 ) % MAX_QUEUE_SIZE`,
2. ajouter le nouvel élément à la position `rear`,
3. **incrémenter la valeur du compteur** `count`.

Retrait (dequeue)

1. sauvegarder la valeur à l'avant de la file,
2. re-initialiser la valeur à la position `front` du tableau,
3. `front = ( front+1 ) % MAX_QUEUE_SIZE`,
4. **décrémenter la valeur du compteur** `count`,
5. retourner la valeur sauvegardée.

empty()

1. `count == 0`

isFull()

1. `count == MAX_QUEUE_SIZE`