2019

Compte rendu de TP : Faites la queue

Java avancé – TP n°5

Jonathan CRÉTÉ – Filière Informatique

Exercice 1 - Fifo

1) Cette représentation peut poser problème, car si la tête et la queue correspondent au même indice, il n'est pas facile de détecter si cela veux dire que la file est pleine ou vide.

Comment doit-on faire pour détecter si la file est pleine ou vide ?

Pour détecter si la fifo est vide ou pleine, on peut mettre le premier indice à -1 ou bien utiliser un compteur qui compte les éléments dans la file.

2) Écrire une classe Fifo générique (avec une variable de type E) dans le package fr.umlv.queue prenant en paramètre le nombre maximal d'éléments que peut stocker la structure de données. Pensez à vérifier les préconditions.

Voir la classe Fifo.java

```
public class Fifo <E>  implements Iterable<E>{

		private final E [] fifo;
		private int tail;  // position prochaine element a inserer
(pointeur)
		private int head; // position du premier element (pointeur)
		private int size; // compter les element pour tester la
taille
		private int maxSize;

		public void shouldHasAnPositiveCapacityNotZero(int capacity)
{
			if (capacity < 0 || capacity == 0) {
				throw new IllegalArgumentException();
			}
		}

		/**
		 * @param fifo
		 */
		@SuppressWarnings("unchecked")
		public Fifo(int capacity) {
			this.shouldHasAnPositiveCapacityNotZero(capacity);
			this.fifo = (E[]) new Object[capacity];
			this.head = 0;
			this.tail = 0;
			this.size = 0;
			this.maxSize = capacity;
		}
...
```

3) Écrire la méthode offer qui ajoute un élément de type E dans la file. Pensez à vérifier les préconditions sachant que, notamment, on veut interdire le stockage de null.

Comment détecter que la file est pleine ?

Que faire si la file est pleine ?

```
/**
		 * add an E element into queue
		 * @param e element to add
```

```
             */
            public void offer (E e ) {
                  Objects.requireNonNull(e);
                  if(this.size == this.maxSize) {
                        throw new IllegalStateException("queue is full");
                  }
                  fifo[tail] = e;
                  tail = (tail + 1) % fifo.length; // re-calcule de la
position de tail, retour en tÃªte de fifo (0)
                  this.size++;
            }
```

En testant si la taille de la fifo n'est pas égale à la taille maximale étant à la capacity à l'initialisation.

Si la fifo est pleine , on lève une **IlegalStateException**.

4) Écrire une méthode poll qui retire un élément de type E de la file. Penser à vérifier les préconditions.

Que faire si la file est vide ?

```
/**
             * remove an E element of fifo
             * @return
             */
            public E poll() {
                  if(this.size == 0) {
                        throw new IllegalStateException("queue is
empty");
                  }
                  var eltToRemove = fifo[head];
                  head = (head + 1) % fifo.length;
                  this.size--;
                  return eltToRemove;
            }
```

Si la file est vide, on lève tout de suite une IllegalStateException.

5) Ajouter une méthode d'affichage qui affiche les éléments dans l'ordre dans lequel ils seraient sortis en utilisant poll. L'ensemble des éléments devra être affiché entre crochets ('[' et ']') avec les éléments séparés par des virgules (suivies d'un espace).

Note: attention à bien faire la différence entre la file pleine et la file vide.

Note 2: Il existe une classe StringJoiner qui est ici plus pratique à utiliser qu'un StringBuilder !

```java
@Override
public String toString() {
	var st =  new StringJoiner(", ", "[", "]");
	int i = head;
	for(int j = 0; j < this.size; j++  ) {
		st.add(fifo[i].toString());
		i =   (i +1) % fifo.length;
	}
	return st.toString();
}
```

6) Rappelez ce qu'est un memory leak en Java et assurez-vous que votre implantation n'a pas ce comportement indésirable.

Une *Memory leak* est fuite de mémoire dans un programme qui alloue trop régulièrement de nouveaux objets en mémoire, la cause la plus classique d'un tel bogue est l'absence de désallocation (de libération) de l'espace utilisé, lorsque ces objets ne sont plus référencés.

Ce n'est pas le cas ici

7) Ajouter une méthode size et une méthode isEmpty.

```java
/**
  * return size of fifo
  * @return
  */
  public int size() {
	return size;
  }

/**
 * test if fifo is empty
 * @return
 */
 public boolean isEmpty() {
	return size == 0;
 }
```

8) Rappelez quel est le principe d'un itérateur.

Quel doit être le type de retour de la méthode iterator() ?

-- Utilisation d'un iterateur --

```
for(Iterator<MyObject> iter = myList.iterator(); iter.hasNext();) {
	MyObject element = iter.next();
	// To do...
 }
```

On peut utiliser un itérateur sur une structure de donnée dès que l'on souhaite modifier ou supprimer des éléments de celle-ci .Relativement performant, un itérateur est assez souple,

et peut nous permettre d'utiliser plusieurs conditions d'arrêts et d'effectuer plusieurs pas d'un coup. La méthode iterator() doit retourner un **E**, étant l'etat dans le parcours de la structure de donnée.

9) Implanter la méthode iterator().

Note: ici, pour simplifier le problème, on considérera que l'itérateur ne peut pas supprimer des éléments pendant son parcours.

```
public Iterator<E> iterator() {
      return new Iterator<E>(){
            private int iteratorCounter;
            private int index = head;

            @Override
            public boolean hasNext() {
                  // test if we have next
                  //return (iteratorCounter < size);
                  return iteratorCounter != size;
            }

            @Override
            public E next() {
               if(!hasNext()) {
                  throw new NoSuchElementException();
               }
               var tmp = fifo[index];
               this.index =  (index +1) % fifo.length;
               iteratorCounter++;
               return tmp;
            }
      };

}
```

On utilise une classe interne de méthode ici. De cette manière, ont redéfini les méthodes *hasNext(...) et next(...) de l'interface Iterator.*

10) Rappeler à quoi sert l'interface Iterable.

Faire en sorte que votre file soit Iterable.

L'interface Iterable permet de rendre iterable notre structure de données. La structure Fifo est donc composée d'éléments que le code appelant pourra parcourir.

Exercice 2 - ResizeableFifo

1) Indiquer comment agrandir la file si celle-ci est pleine et que l'on veut doubler sa taille. Attention, il faut penser au cas où le début de la liste a un indice qui est supérieur à l'indice indiquant la fin de la file.

Implanter la solution retenue dans une nouvelle classe ResizeableFifo.

Note: il existe les méthodes Arrays.copyOf et System.arraycopy.

Dans un premier temps, on implémente la méthode grow(...) dans le cas ou la file est pleine et que l'on veut doubler sa taille :

```
@SuppressWarnings("unchecked")
    private void grow() {
        int newSize = 2 * this.maxSize;
        E[] tmpFifo = (E[]) new Object[newSize];
        System.arraycopy(fifo, head, tmpFifo, 0, size - head);
        System.arraycopy(fifo, 0, tmpFifo, size - head, size - tail);
        this.maxSize = newSize;
        fifo = tmpFifo;
        head = 0;
        tail = size;
    }
```

Dès l'ajout d'un élément dans la file, si le nombre d'éléments est égal à la capacité maximale de la file. On appelle la méthode *grow().*

2) En fait, il existe déjà une interface pour les files dans le JDK appelée java.util.Queue.

Sachant qu'il existe une classe AbstractQueue qui fournit déjà des implantations par défaut de l'interface Queue indiquer

quelles sont les méthodes supplémentaires à implanter;

quelles sont les méthodes dont l'implantation doit être modifiée;

quelles sont les méthodes que l'on peut supprimer.

Faire en sorte que la classe ResizableFifo implante l'interface Queue.

Voir ResizeableFifo.java

Méthodes à implanter :

- ➔ public E peek()

Méthodes dont l'implantation doit être modifiée :

- ➔ public boolean offer(E e)
- ➔ public E poll()

Méthodes à supprimer :

- ➔ l'Ancienne méthode offer(E e)

Conclusion

Durant ce TP de JAVA avancé, j'ai pu implémenter une autre structure de donnée qui demande de réfléchir davantage, à la frontière du cours d'algorithmique de première année.

Je comprendre bien mieux les types paramétrés. J'ai perdu un peu de temps à comprendre qu'il fallait utiliser une classe interne de méthode pour la méthode *iterator(). J'ai compris que le code appelant , à savoir une boucle foreach, par exemple, utilise un itérateur pour parcourir notre file.*

Comme dit dans le précédent rapport je dois continuer de travailler sur l'implémentation d'une structure de donnée, de manière être plus efficace, avec par exemple le TP noté 2017 sur le Container.