

Structure de données : Rapport sur les files

Rédigé et présenté par :

- Cédric Alonso
- Jade Hatoum

Année académique 2023 - 2024

SOMMAIRE

I. Structure générale

II. Simulation sans tests

III. Simulation avec tests

I. Structure générale

1. Algorithme explicité

La méthode `push()` est utilisée pour ajouter un nouvel élément de type T contenant une valeur spécifique à la fin de la file. L'algorithme se déroule comme suit :

- On vérifie si la file est pleine en appelant la méthode `isFull()`.
- Si la file est pleine, une exception `QueueException` est levée, indiquant que l'ajout n'est pas possible.
- Si la file n'est pas pleine, on ajoute l'élément à la position `end` dans le tableau `tab`.
- On met à jour l'indice de fin (`end`) en utilisant l'opération modulo (%) pour assurer le caractère circulaire de la file.
- On incrémente le nombre d'éléments (`nbElements`).
- L'algorithme garantit que l'ajout d'un nouvel élément respecte le principe FIFO de la file. Si la file est pleine, elle effectue une opération d'extension automatique de la capacité, permettant de doubler la taille du tableau statique en mémoire et de copier les données existantes dans le nouveau tableau. Cette opération assure que la file peut accueillir de nouveaux éléments sans perte de données ni débordement.
- Enfin, après l'ajout réussi de l'élément, l'indice de fin est mis à jour, et le nombre d'éléments est incrémenté, maintenant ainsi l'état cohérent de la file.

La méthode `pop()` est utilisée pour retirer et retourner l'élément situé au début de la file. L'algorithme se déroule comme suit :

- On vérifie si la file est vide en appelant la méthode `isEmpty()`.
 - Si la file est vide, une exception `QueueException` est levée, indiquant que le retrait n'est pas possible.
- Si la file n'est pas vide, on récupère la valeur de l'élément à la position `start` dans le tableau `tab`.
- On met à jour l'indice de début (`start`) en utilisant l'opération modulo (%) pour assurer le caractère circulaire de la file.
- On décrémente le nombre d'éléments (`nbElements`).
- On retourne la valeur de l'élément retiré (`value`).

L'algorithme garantit que le retrait d'un élément respecte le principe FIFO de la file. Si la file est vide, une exception est levée pour indiquer que l'opération de retrait n'est pas possible dans un état vide.

La méthode `top()` est utilisée pour accéder à l'élément situé au début de la file sans le retirer. L'algorithme se déroule comme suit :

- On vérifie si la file est vide en appelant la méthode `isEmpty()`.
- Si la file est vide, une exception `QueueException` est levée, indiquant que la lecture n'est pas possible dans un état vide.
- Si la file n'est pas vide, on retourne la valeur de l'élément à la position start dans le tableau tab.
- L'algorithme garantit que l'accès à l'élément de tête respecte le principe FIFO de la file. Si la file est vide, une exception est levée pour indiquer que l'opération de lecture n'est pas possible dans un état vide.

La méthode `toString()` est utilisée pour retourner une représentation sous forme de chaîne de caractères de la file. L'algorithme se déroule comme suit :

- On vérifie si la file est vide en appelant la méthode `isEmpty()`.
- Si la file n'est pas vide, on crée un objet `StringBuilder` pour construire la chaîne résultante.
- On ajoute le nombre d'éléments (`nbElements`) suivi du mot "élément(s)" ou "éléments" selon le nombre d'éléments dans la file.
- On ajoute les éléments de la file, en commençant par l'élément à la position start et en utilisant une boucle pour parcourir le tableau circulairement.
- On retourne la chaîne résultante.

La méthode `isFull()` est utilisée pour vérifier si la file est pleine. L'algorithme se déroule comme suit :

- On compare le nombre d'éléments (`nbElements`) avec la capacité maximale (`NBMAXELEMENTS`).
- Si le nombre d'éléments est égal à la capacité maximale, on retourne true (la file est pleine).
- Sinon, on retourne false.

La méthode `isEmpty()` est utilisée pour vérifier si la file est vide. L'algorithme se déroule comme suit :

- On compare le nombre d'éléments (`nbElements`) avec zéro.
- Si le nombre d'éléments est égal à zéro, on retourne true (la file est vide).
- Sinon, on retourne false.

2. Le code réalisé

```
package TP5.queue;

/**
 * La classe Queue représente une file générique implémentée à l'aide d'un tableau circulaire.
 * Elle permet d'ajouter des éléments à la fin de la file (push) et de retirer des éléments du début de la file (pop).
 * @param <T> Le type des éléments dans la file.
 *
 * @version 1.0
 * @author Cédric Alonso
 * @author Jade Hatoum
 */
5 usages  ± cedric.alonso
public class Queue<T> {

    4 usages
    private final T[] tab; // Tableau contenant les éléments de la file.
    5 usages
    private int start;      // Indice de début de la file.
    4 usages
    private int end;        // Indice de fin de la file.
    6 usages
    private final int NBMAXELEMENTS; // Capacité maximale de la file.
    8 usages
    private int nbElements; // Nombre d'éléments actuellement dans la file.

    /**
     * Constructeur de la classe Queue.
     *
     * @param nbMaxElements La capacité maximale de la file.
     */
    2 usages  ± cedric.alonso
    public Queue(int nbMaxElements) {
        this.NBMAXELEMENTS = nbMaxElements;
        this.tab = (T[]) new Object[NBMAXELEMENTS];
        this.start = 0;
        this.end = 0;
        this.nbElements = 0;
    }

    /**
     * Ajoute un élément à la fin de la file.
     *
     * @param value L'élément à ajouter.
     * @throws QueueException Si la file est pleine.
     */
    ± cedric.alonso
    public void push(T value) throws QueueException {
        if (this.isFull())
            throw new QueueException("Vous ne pouvez pas ajouter un élément (" + value + ") : La file est pleine !");
        this.tab[this.end] = value;
        this.end = (this.end + 1) % this.NBMAXELEMENTS;
        this.nbElements++;
    }

    /**
     * Retire et retourne l'élément du début de la file.
     *
     * @return L'élément retiré de la file.
     * @throws QueueException Si la file est vide.
     */
    ± cedric.alonso
    public T pop() throws QueueException {
        if (this.isEmpty())
            throw new QueueException("Vous ne pouvez pas retirer un élément : La file est vide !");
        T value = this.tab[this.start];
        this.start = (this.start + 1) % NBMAXELEMENTS;
        this.nbElements--;
        return value;
    }
}
```

```

    /**
     * Retourne l'élément du début de la file sans le retirer.
     *
     * @return L'élément du début de la file.
     * @throws QueueException Si la file est vide.
     */
no usages  ± Swiiip*
public T top() throws QueueException {
    if (this.isEmpty())
        throw new QueueException("Vous ne pouvez pas lire d'élément : La file est vide !");
    return this.tab[this.start];
}

/**
 * Retourne une représentation sous forme de chaîne de caractères de la file.
 *
 * @return Une chaîne de caractères représentant la file.
 */
± cedric.alonso +1
@Override
public String toString() {
    if (this.isEmpty())
        return "Il n'y a aucun élément dans la file." + System.lineSeparator();

    StringBuilder sb = new StringBuilder("Il y a ")
        .append(this.nbElements)
        .append(this.nbElements > 1 ? " éléments " : " élément ")
        .append("dans la file :")
        .append(System.lineSeparator());

    int cpt = this.nbElements;
    for (int i = this.start; cpt != 0; i = (i + 1) % NBMAXELEMENTS, cpt--)
        sb.append(this.tab[i]).append(System.lineSeparator());

    return sb.toString();
}

/**
 * Vérifie si la file est pleine.
 *
 * @return true si la file est pleine, false sinon.
 */
1usage  ± cedric.alonso*
public boolean isFull() {
    return this.nbElements == NBMAXELEMENTS;
}

/**
 * Vérifie si la file est vide.
 *
 * @return true si la file est vide, false sinon.
 */
± cedric.alonso*
public boolean isEmpty() {
    return this.nbElements == 0;
}

/**
 * Retourne le nombre d'élément dans la file
 *
 * @return nbElements le nombre d'élément dans la file
 */
no usages new*
public int getNbElements(){
    return this.nbElements;
}

```

3. Les tests et résultats de l'implémentation

```
package TP5.queue;

/**
 * La classe Main est une classe de démonstration illustrant l'utilisation de la classe Queue.
 * Elle crée une instance de la classe Queue, effectue des opérations de push et pop, et affiche les résultats.
 *
 * @version 1.0
 * @author Cédric Alonso
 * @author Jade Hatoum
 */
cedric.alonso +1
public class Main {

    /** File utilisée pour les démonstrations, avec une capacité maximale de 3 éléments. */
    4 usages
    private static final Queue<Integer> queue = new Queue<>( nbMaxElements: 3);

    /**
     * La méthode principale de la classe Main.
     *
     * @param args Les arguments de la ligne de commande.
     */
    cedric.alonso
    public static void main(String[] args) {
        push(new Integer[]{5, 7, 9, 10});

        pop();
        pop();
        pop();
        pop();
        pop();

        push(new Integer[]{55, 77, 99});
    }

    /**
     * Ajoute des valeurs à la file à l'aide de la méthode push et affiche le résultat.
     *
     * @param values Les valeurs à ajouter à la file.
     */
    2 usages cedric.alonso +1
    private static void push(Integer[] values) {
        for (Integer value : values) {
            try {
                queue.push(value);
                System.out.println("Valeur correctement ajoutée : " + value + " dans la file");
            } catch (QueueException e) {
                System.out.println(e.getMessage());
            }
        }
        System.out.println(System.lineSeparator() + queue);
    }

    /**
     * Retire un élément de la file à l'aide de la méthode pop et affiche le résultat.
     */
    5 usages cedric.alonso +1
    private static void pop() {
        try {
            Integer popValue = queue.pop();
            System.out.println("La valeur de l'élément retiré est : " + popValue);
        } catch (QueueException e) {
            System.out.println(e.getMessage());
        }
        System.out.println(queue);
    }
}
```

```
Valeur correctement ajoutée : 5 dans la file
Valeur correctement ajoutée : 7 dans la file
Valeur correctement ajoutée : 9 dans la file
Vous ne pouvez pas ajouter un élément (10) : La file est pleine !
```

```
Il y a 3 éléments dans la file :
5
7
9
```

```
La valeur de l'élément retiré est : 5
Il y a 2 éléments dans la file :
7
9
```

```
La valeur de l'élément retiré est : 7
Il y a 1 élément dans la file :
9
```

```
La valeur de l'élément retiré est : 9
Il n'y a aucun élément dans la file.
```

```
You ne pouvez pas retirer un élément : La file est vide !
Il n'y a aucun élément dans la file.
```

```
You ne pouvez pas retirer un élément : La file est vide !
Il n'y a aucun élément dans la file.
```

```
Valeur correctement ajoutée : 55 dans la file
Valeur correctement ajoutée : 77 dans la file
Valeur correctement ajoutée : 99 dans la file
```

```
Il y a 3 éléments dans la file :
55
77
99
```

```
Process finished with exit code 0
```

II. Simulation sans tests

1. Algorithme explicite

L'objectif de cet algorithme est de gérer le flux de patients arrivant dans une salle d'attente pour une auscultation chez un médecin. Le médecin ne peut examiner qu'un seul patient à la fois, ainsi le premier patient arrivé sera le premier à être traité.

Nous faisons donc appel à la pile de la classe *Queue* définie précédemment.

Deux classes ont été créées afin de résoudre cet énigme : la classe *Patient* portant le nom du patient *P#*, son temps d'arrivée *arrivalTime* et la durée d'auscultation *examinationDuration*. Puis, la classe *WaitingRoom*, qui portera une file de *Patient* appelée *waitingRoom*. Cette classe permettra à l'utilisateur de rajouter les *Patient* à la file à travers *addPatient*, mais surtout d'exécuter la simulation avec *simulate()*.

Cette dernière est le cœur de cet algorithme. Elle se déroule de la manière suivante :

- Avant tout, le temps d'attente total *totalWaitTime*, le temps global total *totalGlobalTime* et le temps écoulé jusqu'à présent *currentTime*, sont tous initialisés à 0 minutes.
- Dans une boucle while, nous itérons l'ensemble de la file *waitingRoom* tant qu'elle n'est pas vide :
 - Nous choisissons d'abord le *Patient* ayant arrivé le plus tôt dans la queue parmi les autres *Patient* dans la file, en faisant appel à notre méthode *getNextEarliestPatient()*.
 - Ce *Patient currentPatient* nous fournira son temps d'arrivée minimal *arrivalTime*.
 - C'est ici que *currentTime* sera mis à jour en prenant le dernier temps évalué, soit la dernière valeur de *currentTime*, donc la fin du temps d'examen du patient précédent, soit *arrivalTime*.
 - Le temps d'attente du *Patient waitTime* est calculé à partir de la différence du temps actuel *currentTime* et son temps d'arrivée *arrivalTime*.
 - Son temps global est donc la somme de ce temps d'attente *waitTime* et la durée de consultation, que nous obtenons à travers *getExaminationDuration()*.
 - Le début du temps d'examen *startExaminationTime* sera attribué le temps actuel à partir de *currentTime*.
 - La fin du temps d'examen *endExaminationTime* correspondra donc à la somme de *startExaminationTime* et la durée d'examen à nouveau.
 - *currentTime* est mis à jour une seconde fois, en prenant la valeur de *endExaminationTime*.
 - *startExaminationTime* et *endExaminationTime* sont utilisés afin d'afficher la période d'examen de *currentPatient*.

- Le temps d'attente global $totalWaitTime$ ajoutera $waitTime$, et le temps global $totalGlobalTime$ ajoutera $globalTime$ pour chaque patient dans la file d'attente $waitingRoom$.
- Finalement nous pouvons donc calculer la moyenne du temps global et du temps d'attente, en divisant $totalGlobalTime$ et $totalWaitTime$ par le nombre de patients $nbPatients$.
- Ces deux dernières et le temps total de simulation, donc $currentTime$, seront affichés.

Ainsi, l'ensemble des patients dans la file d'attente seront traités. Inutile de reenfiler les patients dans $waitingRoom$ puisque dans notre cas, ils ne sont ici que pour une seule consultation.

2. Le code réalisé

```
package TP5.simulation;

/**
 * La classe Patient modélise un patient avec des attributs tels que l'heure d'arrivée et la durée de l'examen.
 * Chaque patient se voit attribuer un nom unique commençant par "P1".
 *
 * @version 1.0
 * @author Cédric Alonso
 * @author Jade Hatoum
 */
11 usages ± Swiiip +1
class Patient {
    1 usage
    private static int idCpt = 0;
    2 usages
    private final String name;
    2 usages
    private final int arrivalTime;
    2 usages
    private final int examinationDuration;

    /**
     * Construit un patient avec l'heure d'arrivée et la durée d'examen spécifiées.
     *
     * @param arrivalTime      L'heure à laquelle le patient arrive.
     * @param examinationDuration La durée de l'examen pour le patient.
     */
    5 usages ± Swiiip +1
    public Patient(int arrivalTime, int examinationDuration) {
        this.name = "P" + ++idCpt;
        this.arrivalTime = arrivalTime;
        this.examinationDuration = examinationDuration;
    }

    /**
     * Récupère le nom du patient.
     *
     * @return Le nom du patient.
     */
    1 usage ± Swiiip
    public String getName() {
        return name;
    }

    /**
     * Récupère l'heure d'arrivée du patient.
     *
     * @return L'heure d'arrivée du patient.
     */
    3 usages ± cedric.alonso
    public int getArrivalTime() {
        return arrivalTime;
    }

    /**
     * Récupère la durée de l'examen du patient.
     *
     * @return La durée de l'examen du patient.
     */
    2 usages ± Swiiip
    public int getExaminationDuration() {
        return examinationDuration;
    }
}
```

```

package TP5.simulation;

import TP5.queue.Queue;
import TP5.queue.QueueException;

<*/>
 * La classe WaitingRoom modélise une salle d'attente où les patients sont traités dans l'ordre d'arrivée.
 * Elle permet d'ajouter des patients à la salle d'attente et de simuler le traitement de ceux-ci.
 *
 * @version 1.0
 * @author Cédric Alonso
 * @author Jade Hatoum
 */

2 usages ± Swiiiiip
class WaitingRoom {
    1 usage
    public static final int MAX_PATIENTS = 100;
    9 usages
    private final Queue<Patient> waitingRoom;

    <*/>
     * Constructeur de la salle d'attente.
    */
    1 usage ± Swiiiiip
    public WaitingRoom() {
        waitingRoom = new Queue<>(MAX_PATIENTS);
    }

    <*/
     * Ajoute un patient à la salle d'attente.
     *
     * @param patient Le patient à ajouter.
     * @throws QueueException Si la file est pleine.
     */
    5 usages ± Swiiiiip
    public void addPatient(Patient patient) throws QueueException {
        waitingRoom.push(patient);
    }

    <*/
     * Récupère et retourne le patient arrivant le plus tôt dans la salle d'attente.
     *
     * @return Le patient arrivant le plus tôt dans la salle d'attente.
     * @throws QueueException Si la salle d'attente est vide.
     */
    1 usage ± Swiiiiip
    public Patient getNextEarliestPatient() throws QueueException {
        Patient currentEarliestPatient = waitingRoom.pop();
        for(int i = 0; i < waitingRoom.getNbElements(); i++){
            Patient currentPatient = waitingRoom.pop();
            if(currentPatient.getArrivalTime() < currentEarliestPatient.getArrivalTime()) {
                waitingRoom.push(currentEarliestPatient);
                currentEarliestPatient = currentPatient;
            } else {
                waitingRoom.push(currentPatient);
            }
        }
        return currentEarliestPatient;
    }
}

```

```
/*
 * Simule le déroulement du traitement des patients en fonction de leur heure d'arrivée
 * et de la durée de leur examen.
 *
 * @throws QueueException Si la salle d'attente est vide.
 */
1usage ± Swimp
public void simulate() throws QueueException {
    int totalWaitTime = 0;
    int totalGlobalTime = 0;
    int nbPatients = waitingRoom.getNbElements();
    int currentTime = 0;

    while (!waitingRoom.isEmpty()) {
        Patient currentPatient = getNextEarliestPatient();
        int arrivalTime = currentPatient.getArrivalTime();

        currentTime = Math.max(currentTime, arrivalTime);

        int waitTime = currentTime - arrivalTime;
        int globalTime = waitTime + currentPatient.getExaminationDuration();

        int startExaminationTime = currentTime;
        int endExaminationTime = startExaminationTime + currentPatient.getExaminationDuration();

        System.out.println(startExaminationTime + " - " + endExaminationTime + " min : " + currentPatient.getName() + " is being examined.");
        currentTime = endExaminationTime;

        totalWaitTime += waitTime;
        totalGlobalTime += globalTime;
    }

    double averageGlobalTime = (double) totalGlobalTime / nbPatients;
    double averageWaitTime = (double) totalWaitTime / nbPatients;

    System.out.println("\nTotal Simulation Time: " + currentTime + " minutes.");
    System.out.println("\nAverage Global Time: " + averageGlobalTime + " minutes.");
    System.out.println("Average Waiting Time: " + averageWaitTime + " minutes.");
}
}
```

3. Les tests et résultats de l'implémentation

```
package TP5.simulation;

import TP5.queue.QueueException;

+ Swiiip +1
public class Main {

    + Swiiip
    public static void main(String[] args) throws QueueException {
        WaitingRoom waitingRoom = new WaitingRoom();

        waitingRoom.addPatient(new Patient( arrivalTime: 3, examinationDuration: 10));
        waitingRoom.addPatient(new Patient( arrivalTime: 0, examinationDuration: 6));
        waitingRoom.addPatient(new Patient( arrivalTime: 2, examinationDuration: 8));
        waitingRoom.addPatient(new Patient( arrivalTime: 4, examinationDuration: 9));
        waitingRoom.addPatient(new Patient( arrivalTime: 6, examinationDuration: 4));

        waitingRoom.simulate();
    }
}
```

0 - 6 min : P2 is being examined.
6 - 14 min : P3 is being examined.
14 - 24 min : P1 is being examined.
24 - 33 min : P4 is being examined.
33 - 37 min : P5 is being examined.

Total Simulation Time: 37 minutes.

Average Global Time: 19.8 minutes.
Average Waiting Time: 12.4 minutes.

Process finished with exit code 0

III. Simulation avec tests

1. Algorithme explicité

Dans ce cas, nous aurons à prendre en compte les tests sanguins, après la consultation du patient avec le médecin.

Nous savons que le premier chiffre dans la colonne “Blood Tests” représente le temps quand le test sanguin commence après le début de la consultation. Alors que le deuxième chiffre dans cette colonne représente la durée du test sanguin.

Cette fois-ci le temps d'attente du patient ne dépendra pas de la disponibilité du médecin, mais il sera fournis par l'utilisateur : aucune question de priorité du premier arrivé, les tests sanguins seront prêts à accueillir le patient dès que le temps d'attente s'écoule.

Aucun intérêt d'utiliser une seconde file donc !

D'abord, la classe *Patient* devra prendre le temps d'attente pour le test sanguin, ainsi que la durée de ce dernier en paramètre de son constructeur.

De plus, nous aurons à compléter notre code précédent, notamment la méthode *simulate()*. Pour ce faire, nous devrons juste relier les deux événements en traitant le patient dès la fin de sa consultation avec le médecin. Ce dernier aurait déjà été supprimé de la file d'attente de consultation lors d'un *pop()* afin d'être examiné par le médecin.

Nous devrons donc ajouter à la somme des temps d'attente, le temps d'attente pour le test sanguin, et pour le temps global, le temps d'attente de ce test ainsi que la durée du test sanguin. La moyenne encore une fois s'effectuera en divisant chaque somme totale par le nombre de patients, à la fin de la simulation.

Nous pouvons déduire que la moyenne du temps global avec les tests sanguins s'élève à :

$$= [19,8 * 5 + ((4+5) + (5+1) + (3+3) + (3+5) + (3+4))] / 5$$

$$= (99 + 135) / 5$$

$$= 27 \text{ minutes}$$

Quant à la nouvelle moyenne du temps d'attente :

$$= [12,4 * 5 + (4+5+3+3+3)] / 5$$

$$= (62 + 18) / 5$$

$$= 16 \text{ minutes}$$

Année académique 2023 - 2024