Trabalho 08

1. Um *deque* (abreviatura para *Double-Ended Queue*, do inglês) é uma estrutura de dados similar a uma fila, com a diferença fundamental de que é possível tanto inserir elementos quanto removê-los a partir dos dois lados. Consequentemente, ao invés de ter apenas as operações *push* e *pop*, um *deque* tem quatro operações: *push left*, *pop left*, *push right* e *pop right*. Essas operações realizam, respectivamente, inserção de um elemento pelo lado esquerdo, remoção de um elemento a partir do lado esquerdo, inserção de um elemento pelo lado direito e remoção de um elemento a partir do lado direito.

Implemente em Java uma estrutura de dados *deque* que oferece as quatro operações mencionadas acima e que, quando utilizada por múltiplas *threads* simultaneamente, deve satisfazer as seguintes propriedades:

P1 Nunca ocorre de duas *threads* realizarem uma operação (inserção ou remoção) em um mesmo lado ao mesmo tempo.

P2 Operações de inserção e remoção inserem e removem, respectivamente, exatamente um elemento do *deque*, exceto quando ele está vazio. Neste caso, a remoção de um elemento não modifica a estrutura de dados.

P3 Remoções sempre se aplicam aos elementos nas pontas do *deque*. Da mesma forma, inserções sempre ocorrem nas pontas do *deque* (propriedade básica de um deque).

P4 O programa não entra em *deadlock*.

P5 Sempre é possível para duas *threads* distintas realizar operações de inserção (*push left()* e *push right()*) simultaneamente nos dois lados do *deque*.

P6 Se duas *threads* distintas tentam realizar operações (remoção ou inclusão) em um mesmo lado, uma delas consegue.

Além disso, sua estrutura de dados deve, tanto quanto possível, permitir que remoções sejam realizadas a partir dos dois lados simultaneamente.

Não é permitido o uso de nenhuma estrutura de dados da biblioteca java.util.concurrent. É permitido, porém, o uso de travas explícitas do pacote java.util.concurrent.locks. É explicitamente permitido usar a classe java.util.LinkedList<E>, que implementa uma lista duplamente ligada em Java. Essa classe tem operações *addFirst*(), *addLast*(), *removeFirst*() e *removeLast*(), responsáveis por inserir um item no

início da lista, inserir um elemento no fim, remover o último elemento da lista e remover o primeiro, respectivamente. LinkedList não é uma classe segura para *threads*.

```java
/**
 * Interface that aims to represent the Double-Ended Queue.
 *
 * @param <T> The type of data that will be stored on queue.
 */
public interface Deque<T extends Object> {
    public void pushLeft(T item);
    public void pushRight(T item);
    public T popLeft();
    public T popRight();
}
```

```java
import java.util.LinkedList;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * Class that aims to represent a Compound Double-Ended Queue.
 *
 * @param <T>
 *            The type of data that will be stored on queue.
 */
public class CompoundDeque<T> implements Deque<T> {

    // locks that will be used for each side operation
    private Lock lockRight;
    private Lock lockLeft;

    // different lists to primary use on each side pop and push to store data
    private LinkedList<T> listRight;
    private LinkedList<T> listLeft;

    public CompoundDeque() {
        // initialization of locks and lists
        this.lockRight = new ReentrantLock();
        this.lockLeft = new ReentrantLock();
        this.listRight = new LinkedList<T>();
        this.listLeft = new LinkedList<T>();
    }

    @Override
    public void pushLeft(T item) {
```

```
    // acquire lock on left in order to add a new item on left list
    this.lockLeft.lock();
    this.listLeft.addFirst(item);
    this.lockLeft.unlock();
}

@Override
public void pushRight(T item) {
    // acquire lock on right in order to add a new item on right list
    this.lockRight.lock();
    this.listRight.addLast(item);
    this.lockRight.unlock();
}

@Override
public T popLeft() {
    return this.pop(this.lockLeft, this.listLeft, this.lockRight, this.listRight, true);
}

@Override
public T popRight() {
    return this.pop(this.lockRight, this.listRight, this.lockLeft, this.listLeft, false);
}

private T pop(Lock primaryLock, LinkedList<T> primaryList, Lock alternativeLock,
        LinkedList<T> alternativeList, boolean removeFirst) {
    T item = null;

    // acquire the primary lock in order to attempt the item retrieval on primary list
    primaryLock.lock();
    if (!primaryList.isEmpty()) {
        item = removeFromList(primaryList, removeFirst);
    }

    // if the item could not be retrieved on the primary list, the pop method will attempt to
    // retrieve it from alternative list
    if (item == null) {
        // acquire the alternative lock in order to attempt the item retrieval on alternative
        // list
        alternativeLock.lock();
        if (!alternativeList.isEmpty()) {
            item = removeFromList(alternativeList, removeFirst);
        }

        // rebalance the list, moving all elements from one side to another
        primaryList = alternativeList;
        alternativeList = new LinkedList<T>();
```

```
            // release the alternative lock
            alternativeLock.unlock();
        }

        // release the primary lock
        primaryLock.unlock();

        return item;
    }

    private T removeFromList(LinkedList<T> list, boolean removeFirst) {
        if (removeFirst) {
            return list.removeFirst();
        } else {
            return list.removeLast();
        }
    }
}
```

```
import java.util.Random;

public class DequeTestRunnable implements Runnable {

    private Deque<String> deque;

    public DequeTestRunnable(Deque<String> deque) {
        this.deque = deque;
    }

    @Override
    public void run() {
        Random random = new Random();
        while (true) {
            try {
                // just wait a little bit in order to validate behavior
                Thread.sleep(1000);
            } catch (InterruptedException e) {
            }

            int type = random.nextInt(4);
            int test = random.nextInt(10);
            switch (type) {
            case 0:
                this.deque.pushLeft("Test " + test);
                System.out.println("Left push: " + test);
```

```
            break;
         case 1:
            this.deque.pushRight("Test " + test);
            System.out.println("Right push: " + test);
            break;
         case 2:
            System.out.println("Left pop: " + this.deque.popLeft());
            break;
         case 3:
            System.out.println("Right pop: " + this.deque.popRight());
            break;
         default:
            break;
         }
      }
   }
}
```

```
public class DequeTest {
   private static final int N = 5;

   public static void main(String[] args) {
      Deque<String> deque = new CompoundDeque<String>();

      for (int i = 0; i < N; i++) {
         Thread thread = new Thread(new DequeTestRunnable(deque));
         thread.start();
      }
   }
}
```

2. Implemente uma solução para a questão 2 que: (i) empregue *hashing* conforme descrito na Seção 6.1.2.3; ou (ii) necessite de apenas uma lista ligada.

O teste da questão anterior pode ser alterado para instanciar a classe abaixo. A HashedDeque necessita na inicialização o valor que representa a quantidade de *buckets* a ser utilizado (ex: Deque<String> deque = new HashedDeque<String>(4)).

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
```

```java
/**
 * Class that aims to represent a Hashed Double-Ended Queue.
 *
 * @param <T>
 *            The type of data that will be stored on queue.
 */
public class HashedDeque<T> implements Deque<T> {

    // locks that will be used for each side operation
    private Lock lockRight;
    private Lock lockLeft;

    // index of the bucket that will be used to push or pop the proper value
    private int leftIndex;
    private int rightIndex;

    // buckets to store the data
    private List<LinkedList<T>> buckets;

    public HashedDeque(int numberOfBucket) {
        this.leftIndex = 0;
        this.rightIndex = 1;
        this.lockRight = new ReentrantLock();
        this.lockLeft = new ReentrantLock();
        this.buckets = new ArrayList<LinkedList<T>>();
        for (int i = 0; i < numberOfBucket; i++) {
            this.buckets.add(new LinkedList<T>());
        }
    }

    @Override
    public void pushLeft(T item) {
        this.lockLeft.lock();
        int index = this.leftIndex;
        this.buckets.get(index).addFirst(item);
        this.leftIndex = this.moveIndexToLeft(this.leftIndex);
        this.lockLeft.unlock();
    }

    @Override
    public void pushRight(T item) {
        this.lockRight.lock();
        int index = this.rightIndex;
        this.buckets.get(index).addLast(item);
        this.rightIndex = this.moveIndexToRight(this.rightIndex);
        this.lockRight.unlock();
    }
```

```
    @Override
    public T popLeft() {
        T item = null;
        this.lockLeft.lock();
        int index = this.moveIndexToRight(this.leftIndex);
        if (!this.buckets.get(index).isEmpty()) {
            item = this.buckets.get(index).removeFirst();

            if (item != null) {
                this.leftIndex = index;
            }
        }
        this.lockLeft.unlock();
        return item;
    }

    @Override
    public T popRight() {
        T item = null;
        this.lockRight.lock();
        int index = this.moveIndexToLeft(this.rightIndex);
        if (!this.buckets.get(index).isEmpty()) {
            item = this.buckets.get(index).removeLast();

            if (item != null) {
                this.rightIndex = index;
            }
        }
        this.lockRight.unlock();
        return item;
    }

    private int moveIndexToLeft(int index) {
        // if the index is already the first one, it must move to the last
        return index == 0 ? this.buckets.size() - 1 : index - 1;
    }

    private int moveIndexToRight(int index) {
        // if the index cross the value of the last one, it must move to the first
        return (index + 1) % this.buckets.size();
    }
}
```