ALP2 Blatt 10

(Mi14h)
Tutor: Can Göktas

Tobias Lohse
Okan Biskin

Für Alle Aufgaben:

```java
import java.util.*;
import java.lang.*;

public class EmptyQueueException extends Exception {
  public EmptyQueueException() {
    super();
  }
}

public class NoMoreElementsException extends NoSuchElementException {
  public NoMoreElementsException() {
    super();
  }
}
```

Aufgabe 1

```java
public class ListQueue<E> implements Queue<E>, Iterable<E> {
  // ATTRIBUTES
  private ListNode<E> head;
  private ListNode<E> tail;

  // CONSTRUCTORS
  public ListQueue() {
    this.head = null;
    this.tail = null;
  }

  // LISTNODE CLASS
  class ListNode<E> {
    // ATTRIBUTES
    E element;
    ListNode<E> next;

    // CONSTRUCTORS
    ListNode(E element, ListNode<E> next) {
      this.element = element;
      this.next = next;
    }
    ListNode(E element) {
      this(element, null);
    }
    ListNode() {
      this(null, null);
    }
  }

  // QUEUE METHODS
  public void enqueue( E element ) {
  /** Adds an element at the end of the list. */
    if( this.empty() ){
      this.tail = this.head = new ListNode<E>(element);
    } else {
      this.tail = this.tail.next = new ListNode<E>(element);
    }
  }
  public E dequeue() throws EmptyQueueException {
  /** Removes and returns the first element of the queue. */
    if( this.empty() ){
      throw new EmptyQueueException();
    } else {
      E element = this.head.element;
```

```
      this.head = this.head.next;
      return element;
    }
  }
  public E head() throws EmptyQueueException {
  /** returns the first element of the queue. */
    if( this.empty() ){
      throw new EmptyQueueException();
    } else {
      return this.head.element;
    }
  }
  public boolean empty() {
  /** checks wether the queue is empty. */
    return (this.head == null);
  }
  public String toString(){
    String out = "LISTQUEUE = | ";
    for( E element : this ){
      out += element + " | ";
    }
    return out;
  }

  // ITERABLE METHODS
  public Iterator<E> iterator(){
    return new QueueIterator<E>(head);
  }

  class QueueIterator<E> implements Iterator<E> {
    // ATTRIBUTES
    ListNode<E> current;

    // CONSTRUCTORS
    QueueIterator(ListNode<E> head) {
      this.current = head;
    }

    // METHODS
    public boolean hasNext() {
      return (current != null);
    }
    public E next() {
      if( !this.hasNext() ){
        throw new NoMoreElementsException();
      } else {
        E element = this.current.element;
        this.current = this.current.next;
        return element;
      }
    }
    public void remove() {
      throw new UnsupportedOperationException();
    }
  }
}

public class TestListQueue {
  public static void main(String[] args) {
    // TESTS
    ListQueue<String> strings = new ListQueue<String>();
    System.out.println("empty? " + strings.empty());
    System.out.println(strings);
    strings.enqueue("Iterators");
    strings.enqueue("are");
    strings.enqueue("useful");
    System.out.println("empty? " + strings.empty());
    System.out.println(strings);
```

```
    String sentence = "";
    for( String element : strings ){
      sentence += " " + element ;
    }
    System.out.println(sentence + ".");

    try {
      System.out.println("removed: " + strings.dequeue());
      System.out.println("removed: " + strings.dequeue());
      System.out.println("next: " + strings.head());
    } catch(Exception e) {
      System.out.println("Empty Queue!");
    }
    System.out.println(strings);
  }
}
```

Aufgabe 2

```
public class ArrayQueue<E> implements Queue<E>, Iterable<E> {
  // ATTRIBUTES
  private E[] sArray;
  private int head;
  private int tail;
  private int max;

  // CONSTRUCTORS
  public ArrayQueue(){
    // this.sArray = (E[]) new Object[2];
    this.sArray = (E[]) new Object[5];
    this.max = sArray.length-1;
    this.head = 0;
    this.tail = 0;
  }

  // QUEUE METHODS
  public void enqueue( E element ) {
  /** Adds an element at the end of the list. */
    if( this.full() ){
      this.resizeSArray();
    }
    this.sArray[this.tail] = element;
    this.tail = nextPosition(this.tail);
  }
  public E dequeue() throws EmptyQueueException {
  /** Removes and returns the first element of the queue. */
    if( this.empty() ){
      throw new EmptyQueueException();
    } else {
      E element = sArray[head];
      this.head = nextPosition(this.head);
      return element;
    }
  }
  public E first() throws EmptyQueueException {
  /** returns the first element of the queue. */
    if( this.empty() ){
      throw new EmptyQueueException();
    } else {
      return sArray[head];
    }
  }
  public boolean empty() {
  /** checks wether the queue is empty. */
    return (this.head == this.tail);
  }
  private boolean full() {
```

```
  /** checks wether the array is full. */
    return (this.head == 0 && this.tail == this.max) || (this.head == this.tail+1) ;
  }
  private int nextPosition(int position) {
    if( position == this.max ){
      return 0;
    } else {
      return position+1;
    }
  }
  private void resizeSArray() {
    E[] temp = (E[]) new Object[this.sArray.length*2];
    for( int i=0; i<this.max; i++ ){
      if( i+this.head <= this.max ){
        temp[i] = this.sArray[i+this.head];
      } else {
        temp[i] = this.sArray[i+this.head-this.max-1];
      }
    }
    this.sArray = temp;
    this.head = 0;
    this.tail = this.max;
    this.max = (this.max+1)*2-1;
  }
  public String toString(){
    String out = "ARRAYQUEUE = | ";
    for( E element : this ){
      out += element + " | ";
    }
    return out;
  }

  // ITERABLE METHODS
  public Iterator<E> iterator(){
    return new QueueIterator<E>(head, sArray);
  }

  class QueueIterator<E> implements Iterator<E> {
    // ATTRIBUTES
    int current;
    private E[] sArray;

    // CONSTRUCTORS
    QueueIterator(int head, E[] sArray) {
      this.current = head;
      this.sArray = sArray;
    }

    // METHODS
    public boolean hasNext() {
      return (current != tail);
    }
    public E next() {
      if( !this.hasNext() ){
        throw new NoSuchElementException();
      } else {
        E element = sArray[this.current];
        this.current = nextPosition(this.current);
        return element;
      }
    }
    public void remove() {
      throw new UnsupportedOperationException();
    }
  }
}

public class TestArrayQueue {
```

```
  public static void main(String[] args) {
    // TESTS
    ArrayQueue<String> strings = new ArrayQueue<String>();
    System.out.println("empty? " + strings.empty());
    System.out.println(strings);
    strings.enqueue("Iterators");
    strings.enqueue("are");
    strings.enqueue("really");
    strings.enqueue("useful");
    System.out.println("empty? " + strings.empty());
    System.out.println(strings);

    String sentence = "";
    for( String element : strings ){
      sentence += " " + element ;
    }
    System.out.println(sentence + ".");

    try {
      System.out.println("removed: " + strings.dequeue());
      System.out.println("removed: " + strings.dequeue());
      System.out.println("removed: " + strings.dequeue());
      System.out.println("next: " + strings.first());
    } catch(Exception e) {
      System.out.println("Empty Queue!");
    }
    System.out.println(strings);

    strings.enqueue("iterators");
    // System.out.println(strings);
    strings.enqueue("are");
    strings.enqueue("really");
    // System.out.println(strings);
    strings.enqueue("useful");
    strings.enqueue("are");
    // System.out.println(strings);
    strings.enqueue("they");
    strings.enqueue("not");
    System.out.println(strings);
  }
}
```

Aufgabe 3

```
public class PriorityQueue <P extends Comparable<P>,D> implements Queue<D,P> {
  // ATTRIBUTES
  private TreeNode<P,D> root;
  private int size;

  // CONSTRUCTOR
  PriorityQueue() {
    this.root = null;
    this.last = null;
    this.size = 0;
  }

  class TreeNode<P,D> {
    // ATTRIBUTES
    P priority;
    D data;
    TreeNode<P,D> left;
    TreeNode<P,D> right;
    TreeNode<P,D> parent;

    // CONSTRUCTOR
    TreeNode(P priority, D data, TreeNode<P,D> left,
             TreeNode<P,D> right, TreeNode<P,D> parent) {
      this.priority = priority;
```

```
    this.data = data;
    this.left = left;
    this.right = right;
    this.parent = parent;
  }
  TreeNode(P priority, D data, TreeNode<P,D> parent) {
    this(priority, data, null, null, parent);
  }

  // METHODS
  public String toString() {
    return this.data + " : P" + this.priority;
  }
}

// METHODS
public static char[] getNodePath(int n) {
  String binString = Integer.toBinaryString(n);
  char[] binChars = binString.substring(1, binString.length()).toCharArray();
  return binChars;
}
public TreeNode<P,D> getNode(int n) {
/** Returns node at given position. */
  TreeNode<P,D> node = this.root;
  for( char c : getNodePath(n) ){
    if( c == '0' ){
      node = node.left;
    } else if( c == '1' ){
      node = node.right;
    }
  }
  return node;
}
public String toString() {
  String out = "PriorityQueue = | ";
  if( this.empty() ){
    out += "EMPTY";
  }
  for( int i=1; i<=size; i++ ){
    TreeNode<P,D> node = this.getNode(i);
    out += node + " | ";
  }
  return out;
}

// QUEUE METHODS
public void enqueue(P priority, D data) {
/** Adds a node and reheapifies. */
  if( this.empty() ){
    this.size++;
    this.root = new TreeNode<P,D>(priority, data, null);
  } else {
    this.size++;
    TreeNode<P,D> parent = this.getNode((this.size)>>1);
    assert parent != null;
    if( ((this.size)&1) == 0 ){
      parent.left = new TreeNode<P,D>(priority, data, parent);
    } else {
      parent.right = new TreeNode<P,D>(priority, data, parent);
    }
    assert this.getNode(this.size) != null;
    this.heapifyUp(this.getNode(this.size));
  }
}
public D dequeue() throws EmptyQueueException{
/** Removes and returns the element with highest priority and reheapifies. */
  if( this.empty() ){
    throw new EmptyQueueException();
```

```
  } else {
    assert this.root != null;
    D data = this.root.data;
    TreeNode<P,D> last = this.getNode(this.size);
    this.swapContent(this.root, last);
    this.heapifyDown(this.root);
    last = null;
    this.size--;
    return data;
  }
}
public D highest() throws EmptyQueueException {
/** Returns the element with highest priority. */
  if( this.empty() ){
    throw new EmptyQueueException();
  } else {
    assert this.root != null;
    return this.root.data;
  }
}
public void clear() {
/** Clears cue. */
  for( int i=size; i>=1; i-- ){
    TreeNode<P,D> last = this.getNode(i);
    last = null;
    this.size--;
  }
  assert this.empty() && this.root == null && this.root.left == null;
}
public boolean empty() {
/** checks wether the queue is empty. */
  return (this.size == 0);
}

// HEAP METHODS
private void heapifyDown(TreeNode<P,D> node) {
/** Ensures the heap condition from a given node on down. */
TreeNode<P,D> biggest;
  if( node.left != null && node.priority.compareTo(node.left.priority) < 0 ) {
    biggest = node.left;
  } else {
    biggest = node;
  }
  if( node.right != null && biggest.priority.compareTo(node.right.priority) < 0 ) {
    biggest = node.right;
  }
  if( biggest.priority != node.priority ){
    this.swapContent(node, biggest);
    assert node.priority.compareTo(biggest.priority) < 0;
    this.heapifyDown(biggest);
  }
}
private void heapifyUp(TreeNode<P,D> node) {
/** Ensures the heap condition from a given node on up. */
  if( node.parent != null && node.priority.compareTo(node.parent.priority) > 0 ) {
    this.swapContent(node, node.parent);
    assert node.priority.compareTo(node.parent.priority) < 0;
    this.heapifyUp(node.parent);
  }
}
public boolean testHeap() {
  boolean test = true;
  for( int i=1; i<=this.size; i++ ){
    test = test && ( this.getNode(i).left == null ||
              this.getNode(i).priority.compareTo(this.getNode(i).left.priority) >= 0 );
    test = test && ( this.getNode(i).right == null ||
              this.getNode(i).priority.compareTo(this.getNode(i).right.priority) >= 0 );
  }
```

```
    return test;
  }
  private void swapContent(TreeNode<P,D> a, TreeNode<P,D> b) {
  /** Swaps two given nodes in the tree. */
    P aPriority = a.priority;
    D aData = a.data;
    P bPriority = b.priority;
    D bData = b.data;
    a.priority = bPriority;
    a.data = bData;
    b.priority = aPriority;
    b.data = aData;
  }
}

public class SimulateMessageTraffic {
  // ATTRIBUTES
  public PriorityQueue<Integer, String> queue = new PriorityQueue<Integer, String>();
  public Random rand = new Random();
  private int messageNumber = 0;

  // METHODS
  public String nextMessage() {
    this.messageNumber++;
    return "message #" + messageNumber;
  }
  public int randomPriority() {
    return rand.nextInt(9);
  }
  public void addMessage() {
    int priority = this.randomPriority();
    String data = this.nextMessage() + " P:"+priority;
    // String data = this.nextMessage();
    this.queue.enqueue(priority, data);
  }
  public String removeMessage() {
    try {
      return this.queue.dequeue();
    } catch( EmptyQueueException e ){
      return "    empty queue";
    }
  }

  public static void main(String[] args) {
    // TESTS
    SimulateMessageTraffic simulator = new SimulateMessageTraffic();
    for( int i=0; i<25; i++ ){
      simulator.addMessage();
    }
    System.out.println(simulator.queue);
    System.out.println(simulator.queue.testHeap());
    System.out.println("\n");

    simulator.queue.clear();
    System.out.println(simulator.queue.empty());
    int iterations = 100;
    Random rand = new Random();
    for( int i=0; i<iterations; i++ ){
      boolean newmessage = rand.nextBoolean();
      if( newmessage ){
        simulator.addMessage();
        System.out.println("    added message");
      } else {
        System.out.println(simulator.removeMessage());
      }
    }
  }
}
```