Name-Abhi Mehta

Reg No.-221080001

Laboratory-08

Experiment: Implement a priority queue Model an element and its priority as a key-value composite known as an entry using Linked Positional List. Define the priority queue ADT to support the following methods:

•insert(k, v):Creates an entry with key k and value v in the priority queue.

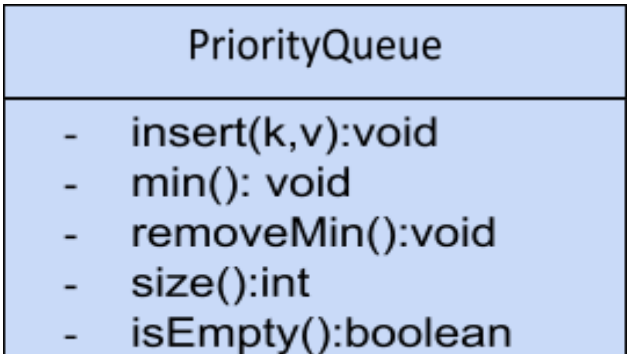
•min():Returns (but does not remove) a priority queue entry(k,v) having minimal key; returns null if the priority queue is empty.

•removeMin():Removes and returns an entry(k,v) having minimal key from the priority queue; returns null if the priority queue is empty.

•size():Returns the number of entries in the priority queue.

•isEmpty():Returns a Boolean indicating whether the priority queue is empty.

Class Diagram:



Java Constructs used in the program-

- The Priority Queue implementation in Java relies on the concept of Entry, where each Entry comprises an integer key and a corresponding string value. The key serves as the determinant of priority within the Queue, with the minimum key value designating the highest priority.

- This implementation leverages a Linked Positional List, a doubly-linked list employing the Position interface. The Linked Positional List constructs the queue of entries through its functions like addLast and remove, facilitating the identification and removal of the minimum entry.

- The use of Generic classes ensures flexibility across various classes such as Entry, PositionalList, Position, AbstractPriorityQueue, PriorityQueue, and Comparator.

- AbstractPriorityQueue serves as the class defining essential functions within the PriorityQueue interface. A Default comparator, imported from the java.util.Comparator module, facilitates the comparison of keys between different entries.

- Control structures, such as while loops, are employed in the display function to iteratively assess whether an entry is null, facilitating the loop's execution until the condition becomes false. If-else statements

play a crucial role in handling conditional blocks, checking for null values, and comparing different entries to determine the minimum entry for removal.

- The main function utilizes a switch-case construct to execute corresponding operations as specified in the menu, enhancing the clarity and organization of the code.

Code:

```
import java.util.Comparator;
import java.util.Scanner;

// Entry class representing an element and its priority
class Entry<K, V> {
    private K key;
    private V value;

    public Entry(K key, V value) {
        this.key = key;
        this.value = value;
    }
```

```
    public K getKey() {
        return key;
    }

    public V getValue() {
        return value;
    }
}

// PositionalList interface representing a linked positional list
interface PositionalList<E> {
    int size();

    boolean isEmpty();

    Position<E> first();

    Position<E> last();

    Position<E> before(Position<E> p) throws IllegalArgumentException;

    Position<E> after(Position<E> p) throws IllegalArgumentException;
```

```
    Position<E> addFirst(E e);

    Position<E> addLast(E e);

    Position<E> addBefore(Position<E> p, E e) throws
IllegalArgumentException;

    Position<E> addAfter(Position<E> p, E e) throws
IllegalArgumentException;

    E set(Position<E> p, E e) throws IllegalArgumentException;

    E remove(Position<E> p) throws IllegalArgumentException;
}

// Position interface
interface Position<E> {
    E getElement() throws IllegalStateException;
}

// AbstractPriorityQueue class implementing PriorityQueue interface
```

```
abstract class AbstractPriorityQueue<K, V> implements
PriorityQueue<K, V> {
   private EntryComparator<K> comparator;

   protected AbstractPriorityQueue(Comparator<K> comparator) {
      this.comparator = new EntryComparator<>(comparator);
   }

   protected AbstractPriorityQueue() {
      this(new DefaultComparator<>());
   }

   public boolean isEmpty() {
      return size() == 0;
   }

   private static class DefaultComparator<E> implements
Comparator<E> {
      public int compare(E a, E b) throws ClassCastException {
         return ((Comparable<E>) a).compareTo(b);
      }
   }
```

```
    protected int compare(Entry<K, ?> a, Entry<K, ?> b) {
        return comparator.compare(a, b);
    }

    private static class EntryComparator<K> implements
Comparator<Entry<K, ?>> {
        private Comparator<K> keyComparator;

        public EntryComparator(Comparator<K> keyComparator) {
            this.keyComparator = keyComparator;
        }

        @Override
        public int compare(Entry<K, ?> e1, Entry<K, ?> e2) {
            return keyComparator.compare(e1.getKey(), e2.getKey());
        }
    }
}
// PriorityQueue interface
interface PriorityQueue<K, V> {
    int size();
```

```
    boolean isEmpty();

    Entry<K, V> insert(K key, V value);

    Entry<K, V> min();

    Entry<K, V> removeMin();

    void display();
}

// LinkedPositionalList class implementing PositionalList
class LinkedPositionalList<E> implements PositionalList<E> {
    private Node<E> header;
    private Node<E> trailer;
    private int size = 0;

    public LinkedPositionalList() {
        header = new Node<>(null, null, null);
        trailer = new Node<>(null, header, null);
        header.setNext(trailer);
    }
```

```
// Generic Node class to hold Entry<K, V>
private static class Node<E> implements Position<E> {
    private Entry<E, ?> entry;
    private Node<E> prev;
    private Node<E> next;

    public Node(Entry<E, ?> entry, Node<E> prev, Node<E> next) {
        this.entry = entry;
        this.prev = prev;
        this.next = next;
    }

    public Node<E> getPrev() {
        return prev;
    }

    public Node<E> getNext() {
        return next;
    }

    public void setPrev(Node<E> prev) {
```

```
    this.prev = prev;
}

public void setNext(Node<E> next) {
    this.next = next;
    if (next != null) {
        next.setPrev(this);
    }
}

@Override
public E getElement() throws IllegalStateException {
    if (prev == null || next == null) {
        System.out.println("Position no longer valid");
        return null;
    }
    return (E) entry.getValue();
}

public void setElement(E e) {
    this.entry = (Entry<E, ?>) e;
}
```

```
}

@Override
public int size() {
   return size;
}

@Override
public boolean isEmpty() {
   return size == 0;
}

@Override
public Position<E> first() {
   if (isEmpty()) {
      return null;
   }
   return header.getNext();
}

@Override
public Position<E> last() {
```

```
        if (isEmpty()) {
            return null;
        }
        return trailer.getPrev();
    }

    @Override
    public Position<E> before(Position<E> p) throws
IllegalArgumentException {
        Node<E> node = validate(p);
        return node.getPrev();
    }

    @Override
    public Position<E> after(Position<E> p) throws
IllegalArgumentException {
        Node<E> node = validate(p);
        return node.getNext();
    }

    @Override
    public Position<E> addFirst(E e) {
        return addBetween(e, header, header.getNext());
```

```
    }

    @Override
    public Position<E> addLast(E e) {
        return addBetween(e, trailer.getPrev(), trailer);
    }

    @Override
    public Position<E> addBefore(Position<E> p, E e) throws
IllegalArgumentException {
        Node<E> node = validate(p);
        return addBetween(e, node.getPrev(), node);
    }

    @Override
    public Position<E> addAfter(Position<E> p, E e) throws
IllegalArgumentException {
        Node<E> node = validate(p);
        return addBetween(e, node, node.getNext());
    }

    @Override
    public E set(Position<E> p, E e) throws IllegalArgumentException {
```

```
    Node<E> node = validate(p);
    E oldValue = node.getElement();
    node.setElement(e);
    return oldValue;
}

@Override
public E remove(Position<E> p) throws IllegalArgumentException {
    Node<E> node = validate(p);
    Node<E> prevNode = node.getPrev();
    Node<E> nextNode = node.getNext();
    prevNode.setNext(nextNode);
    nextNode.setPrev(prevNode);
    size--;
    E removedElement = node.getElement();
    node.setPrev(null);
    node.setNext(null);
    node.setElement(null);
    return removedElement;
}

private Position<E> addBetween(E e, Node<E> pred, Node<E> succ) {
```

```
        Node<E> newNode = new Node<>(new Entry<>(e, null), pred,
succ);
        pred.setNext(newNode);
        succ.setPrev(newNode);
        size++;
        return newNode;
    }

    private Node<E> validate(Position<E> p) throws
IllegalArgumentException {
    if (!(p instanceof Node)) {
        throw new IllegalArgumentException("Invalid position");
    }
    Node<E> node = (Node<E>) p;
    Node<E> current = header.getNext();

    while (current != null) {
        if (current == node) {
            return node;
        }
        current = current.getNext();
    }
```

```
    throw new IllegalArgumentException("Position is no longer in the
list");
}



}
// LinkedPriorityQueue class extending AbstractPriorityQueue
class LinkedPriorityQueue<K, V> extends AbstractPriorityQueue<K, V> {
    public PositionalList<Entry<K, V>> list = new LinkedPositionalList<>();
    public EntryComparator<K> comparator = new EntryComparator<>();
    int size;
    @Override
    public int size() {
        return list.size();
    }

    @Override
    public boolean isEmpty() {
        return list.isEmpty();
    }

    @Override
```

```
public Entry<K, V> insert(K key, V value) {
   Entry<K, V> newest = new Entry<>(key, value);
   list.addLast(newest);
   size++;
   return newest;
}

@Override
public Entry<K, V> min() {
   if (isEmpty()) {
      return null;
   } else {
      Position<Entry<K, V>> minPos = findMin();
      return minPos.getElement();
   }
}

@Override
public Entry<K, V> removeMin() {
   if (isEmpty()) {
      return null;
   }
```

```
    Position<Entry<K, V>> minPos = findMin();
    Node<K, V> minNode = validate(minPos);
    Node<K, V> prevNode = minNode.getPrev();
    Node<K, V> nextNode = minNode.getNext();
    prevNode.setNext(nextNode);
    nextNode.setPrev(prevNode);
    minNode.setPrev(null);
    minNode.setNext(null);
    size--;
    return minPos.getElement();
}

private Node<K, V> validate(Position<Entry<K, V>> minPos){
if (!(minPos instanceof Node)) {
    throw new IllegalArgumentException("Invalid position");
}

Node<K, V> node = (Node<K, V>) minPos;
Position<Entry<K, V>> current = list.first();
while (current != null) {
    Node<K, V> currentNode = (Node<K, V>) current;
    if (currentNode == node) {
```

```
            return node;
        }
        current = list.after(current);
    }
    return node;
}
```

```
    // Display function to print the elements in the priority queue and return the minimum entry
    public void display() {
    Position<Entry<K, V>> current = list.first();
    while (current != null) {
        Entry<K, V> entry = current.getElement();
        if (entry != null) {
            System.out.println("Key: " + entry.getKey() + ", Value: " + entry.getValue());
        }
        current = list.after(current);
    }
    }
```

```
   private Position<Entry<K, V>> findMin() {
      Position<Entry<K, V>> minPos = list.first();
      if (minPos == null) {
         throw new IllegalArgumentException("Position is no longer in the
list");
      }

      Position<Entry<K, V>> walk = list.after(minPos);
      while (walk != null) {
            if (minPos == null || comparator.compare(walk.getElement(),
minPos.getElement()) < 0) {
               minPos = walk;
            }
            else{
               System.out.print("Position is no longer in the list");
            }
            walk = list.after(walk);
      }
      return minPos;
```

```
    }
}


// EntryComparator class for comparing entries based on their keys
class EntryComparator<K> implements java.util.Comparator<Entry<K, ?>> {
    @Override
    public int compare(Entry<K, ?> e1, Entry<K, ?> e2) {
        if (e1 != null && e2 != null) {
            return ((Comparable<K>) e1.getKey()).compareTo(e2.getKey());
        } else {
            return -1;
        }
    }
}


// Example usage with user input
public class PriorityQueueExample {
    public static void main(String[] args) {
        PriorityQueue<Integer, String> priorityQueue = new LinkedPriorityQueue<>();
```

```
        Scanner sc = new Scanner(System.in);
        System.out.println("Menu\n"+"1.Insert Key\n"+"2.Minimum\n"+"3.Remove Minimum\n"+"4.Size\n"+"5.Empty or not\n"+"6.Display\n"+"Exit");
        boolean bool=true;
        while (bool) {
            System.out.print("Enter choice:");
            int choice=sc.nextInt();
            switch(choice){
                case 1:
                    System.out.print("Enter key (or 'exit' to stop): ");
                    int key = sc.nextInt();
                    System.out.print("Enter value: ");
                    String value = sc.next();
                    priorityQueue.insert(key, value);
                    break;
                case 2:
                    Entry<Integer, String> minEntry = priorityQueue.min();
                    if (minEntry != null) {
                        System.out.println("Min: " + minEntry.getValue());
                    }
                    else{
                        System.out.println("Priority queue is invalid");
```

```
    }
    break;
case 3:
    priorityQueue.removeMin();
    System.out.println("Removed Minimum.");
    break;
case 4:
    System.out.println("Size: " + priorityQueue.size());
    break;
case 5:
    System.out.println(priorityQueue.isEmpty());
    break;
case 6:
    priorityQueue.display();
    break;
case 7:
    System.out.println("Exit.");
    bool=false;
    break;
default:
    System.out.println("Enter valid option");
    break;
```

```
        }
      }
    }
}
```

Errors encountered-

- Correct parameterization is crucial when creating a generic class, ensuring that the Node<Entry<k,v>> structure is employed to prevent errors.

- All classes, both for the Priority Queue and the Linked Positional List, must be meticulously crafted, as they complement each other in the implementation.

- Precision in specifying data types is paramount, with a strict adherence to the fixed data types for keys and values—integer and string, respectively.

- Proper use of the "extend" and "implement" keywords is necessary in relevant contexts to avert errors.

- Attention to detail in the inclusion of semicolons and adherence to syntax rules is fundamental, as these basics contribute significantly to error prevention in the code

Learning-

I've gained insights into the intriguing concept of utilizing one data structure, like the Positional Linked List, to implement another, such as the Priority Queue. It's fascinating to discover that a Priority Queue, with its unique functions like min() and removeMin(), can be built upon a different underlying structure. The process involves the creation of several specific classes, including Position, Node, AbstractPriorityQueue, PriorityQueue, PositionalList, LinkedPositionalList, and others. Witnessing the synergy between these distinct data structures has been a captivating learning experience, showcasing how diverse elements can seamlessly complement each other.

.

Conclusion-

In summary, our experimentation leads us to the conclusion that there exist multiple approaches to implement a Priority Queue, and we successfully employed the technique of utilizing a Positional Doubly Linked List to execute the specified operations with efficiency.