**Hashtable:**

**class HashtableDivision<T, V> {**

array table[10]

function put(key, value) {

index = hashDivision(key)

if (table[index] == null) {

table[index] = new ArrayList<Passenger<T, V>>()

}

for each passenger in table[index] {

if (passenger.key.equals(key)) {

passenger.value = value

return

}

}

table[index].add(new Passenger<T, V>(key, value))

**}**

**function get(key) {**

index = hashDivision(key)

if (table[index] != null) {

for each passenger in table[index] {

if (passenger.key.equals(key)) {

return passenger.value

}

}

}

return null

}

**}**

**class Passenger<T, V> {**

T key

V value

function Passenger(key, value) {

this.key = key

this.value = value

}

**}**

**function hashDivision(key) {**

hashCode = key.hashCode()

index = hashCode % 10

if (index < 0) {

index = index + 10

}

return index

**}**

**Priority Queue:**

**public void add(T element) {**

if (size == heap.length - 1) {

            resize();

        }

        heap[++size] = element;

        int pos = size;

        while (pos > 1 && heap[pos].compareTo(heap[pos/2]) > 0) {

            swap(pos, pos/2);

            pos /= 2;

        }

**}**

**public T peek() {**

if (size == 0) {

            return null;

        }

        return heap[1];

**}**

**public T poll() {**

    if (size == 0) {

            return null;

        }

        T result = heap[1];

        heap[1] = heap[size--];

        heapify(1);

        return result;

**}**

**private void swap(int i, int j) {**

        T temp = heap[i];

        heap[i] = heap[j];

        heap[j] = temp;

**}**

**private void resize() {**

        T[] newHeap = (T[]) new Comparable[heap.length \* 2];

        System.arraycopy(heap, 1, newHeap, 1, size);

        heap = newHeap;

**}**

**public int size() {**

  return size;

**}**

**public boolean isEmpty() {**

return size == 0;

**}**

**public <T extends Passenger> int comparePassengers(T c1, T c2) {**

   if (c1.getFirstClass() == c2.getFirstClass()) {

            if (c1.getFirstClass() == true) {

                if (c1.getPriority() != c2.getPriority()) {

                    if (c1.getPriority() > c2.getPriority()) then return 1 : if not true then return-1;

                } else if (c1.getMiles() != c2.getMiles()) {

                    if (c1.getMiles() > c2.getMiles()) then return 1 :if not true then return-1;

                } else if(c1.getArrivalTime().compareTo(c2.getArrivalTime()) != 0){

                    return c1.getArrivalTime().compareTo(c2.getArrivalTime());

                }

            } else if (c1.getArrivalTime().compareTo(c2.getArrivalTime()) != 0) {

                return c1.getArrivalTime().compareTo(c2.getArrivalTime());

            }

        } else {

            if (c1.getFirstClass() == true) then return 1 :if not true then return -1;

        }

        return 0;

**}**

**Queue**

**public class Queue<T> {**

    private Passenger<T> head;

    private Passenger<T> tail;

    private int size;

**public Queue() {**

        head = null;

        tail = null;

        size = 0;

**}**

**public int size() {**

        return size;

**}**

**public boolean isEmpty() {**

        return size == 0;

**}**

**public void add(T data) {**

        Node<T> newNode = new Node<>(data);

        if (isEmpty()) {

            head = newNode;

            tail = newNode;

        } else {

            tail.next = newNode;

            tail = newNode;

        }

        size++;

**}**

**public T poll() {**

        if (isEmpty()) {

            return null;

        } else {

            T data = head.data;

            head = head.next;

            size--;

            if (isEmpty()) {

                tail = null;

            }

            return data;

        }

**}**

**}**