Chris Ceron

CSCI 230 – Friday

Project 4

Eclipse Oxygen

File Names

AbstractHashMap.java

AbstractMap.java

AbstractPriorityQueue.java

AdaptablePriorityQueue.java

AdjacencyMatrixGraph.java

DefaultComparator.java

Driver.java

Entry.java

HeapAdaptablePriorityQueue.java

HeapPriorityQueue.java

Map.java

PriorityQueue.java

ProbeHashMap.java

Input Files:

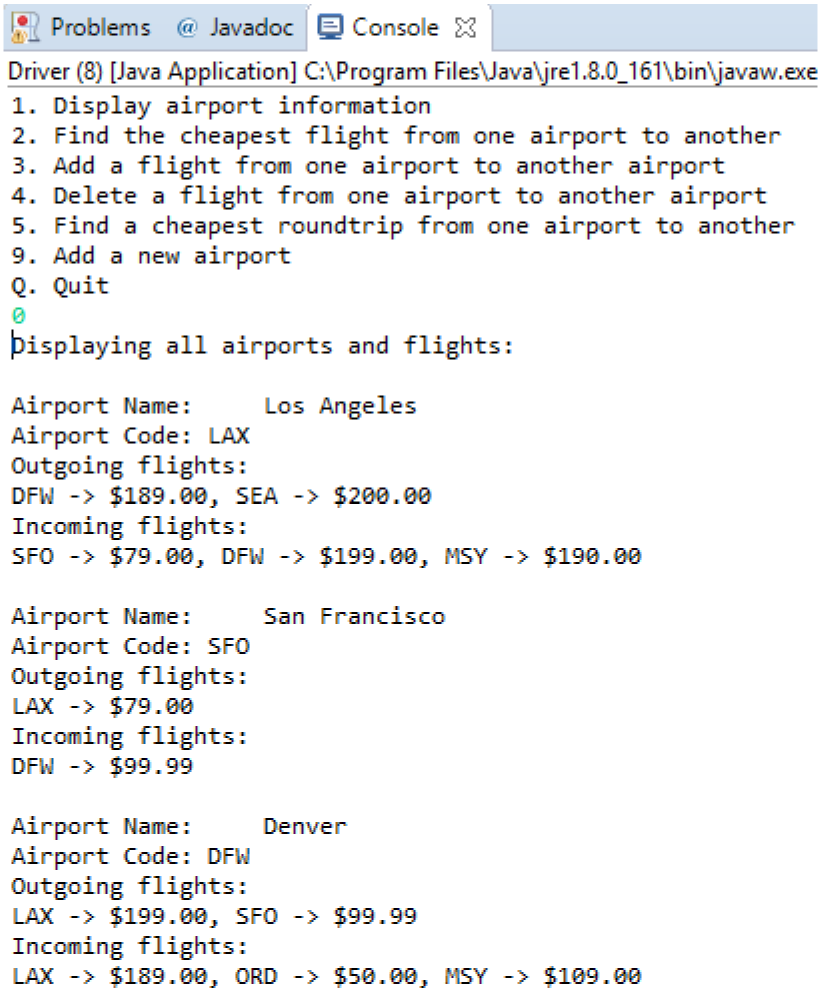
P4Airports.txt

P4Flights.txt

Notes

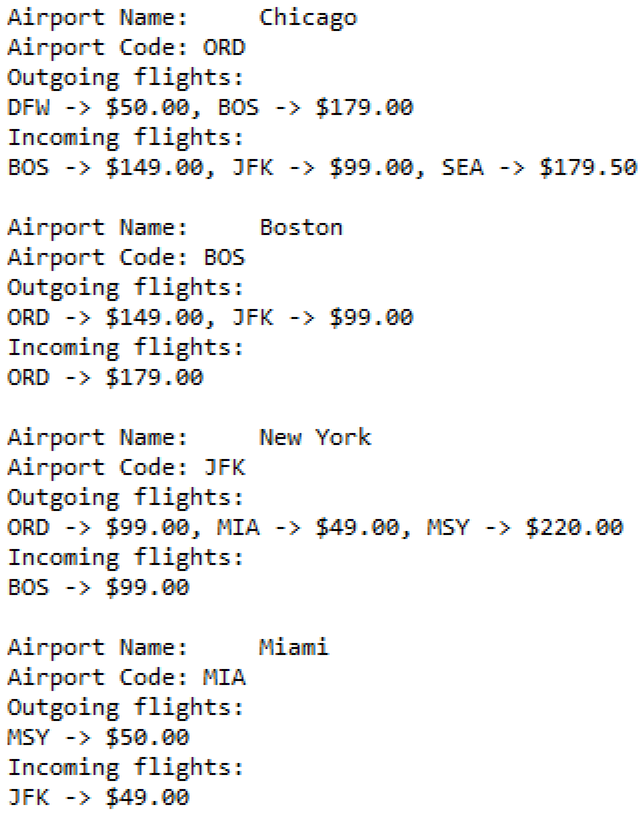
The status of my program is complete and has thoroughly been tested. I did not have any issues completing the project as much of it was done through labs. The hardest part of the project was to keep track of the previous vertex within the shortest path algorithm to reconstruct the path from the start vertex to the end vertex. I believe my use of the hash map to complete this task is very efficient as it takes no time to find the previous vertex and reconstruct the path. I also completed **extra credit option 9** which is to include the option to add an airport.

Output – Mandatory Test Cases

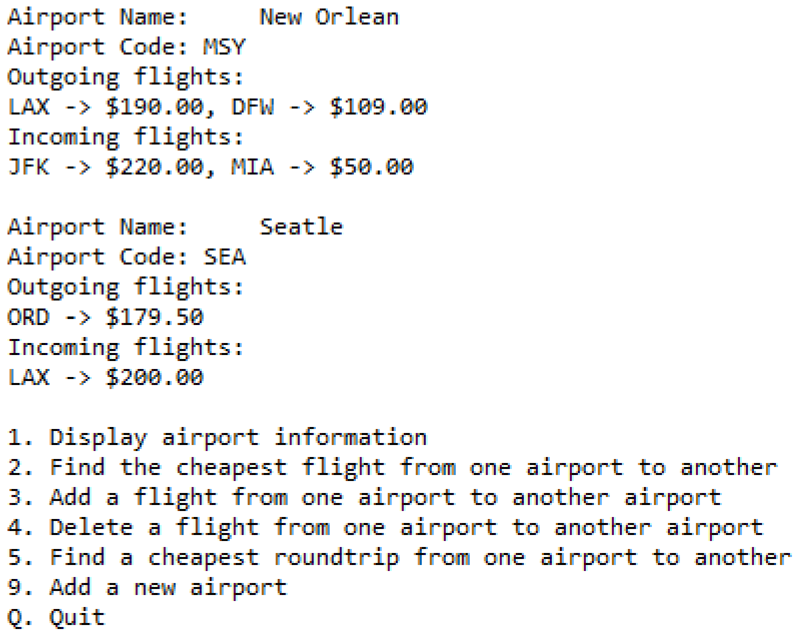


Driver (8) [Java Application] C:\Program Files\Java\jre1.8.0_161\bin\javaw.exe

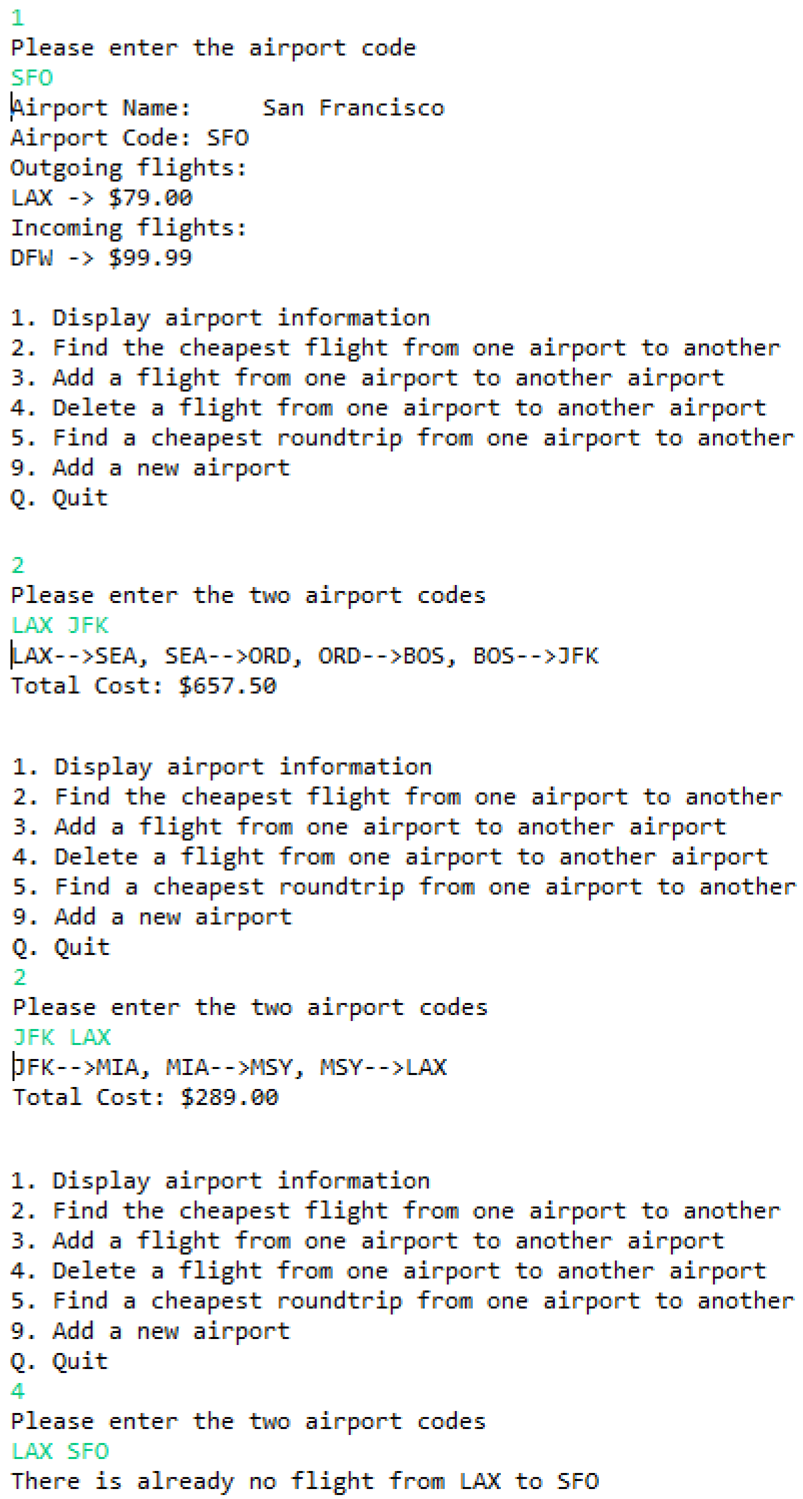
```
1. Display airport information
2. Find the cheapest flight from one airport to another
3. Add a flight from one airport to another airport
4. Delete a flight from one airport to another airport
5. Find a cheapest roundtrip from one airport to another
9. Add a new airport
Q. Quit
0
Displaying all airports and flights:

Airport Name:     Los Angeles
Airport Code: LAX
Outgoing flights:
DFW -> $189.00, SEA -> $200.00
Incoming flights:
SFO -> $79.00, DFW -> $199.00, MSY -> $190.00

Airport Name:     San Francisco
Airport Code: SFO
Outgoing flights:
LAX -> $79.00
Incoming flights:
DFW -> $99.99

Airport Name:     Denver
Airport Code: DFW
Outgoing flights:
LAX -> $199.00, SFO -> $99.99
Incoming flights:
LAX -> $189.00, ORD -> $50.00, MSY -> $109.00
```



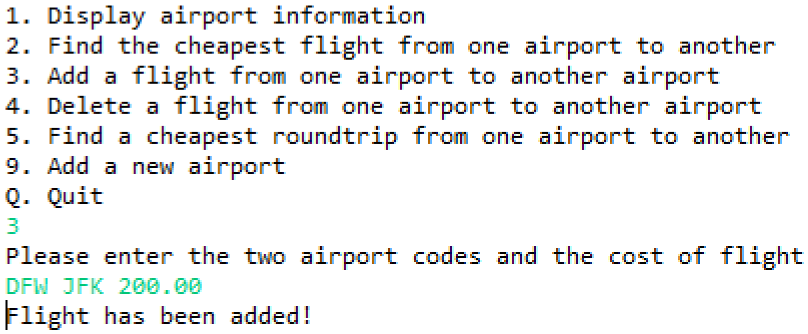
```
Airport Name:     Chicago
Airport Code: ORD
Outgoing flights:
DFW -> $50.00, BOS -> $179.00
Incoming flights:
BOS -> $149.00, JFK -> $99.00, SEA -> $179.50

Airport Name:     Boston
Airport Code: BOS
Outgoing flights:
ORD -> $149.00, JFK -> $99.00
Incoming flights:
ORD -> $179.00

Airport Name:     New York
Airport Code: JFK
Outgoing flights:
ORD -> $99.00, MIA -> $49.00, MSY -> $220.00
Incoming flights:
BOS -> $99.00

Airport Name:     Miami
Airport Code: MIA
Outgoing flights:
MSY -> $50.00
Incoming flights:
JFK -> $49.00
```



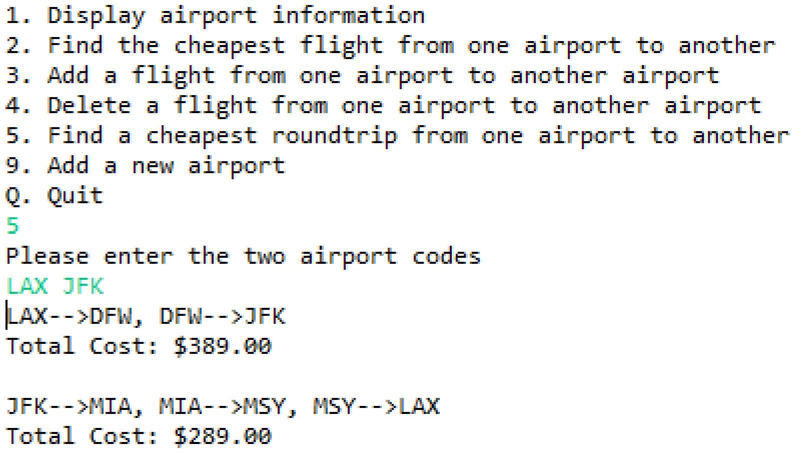
```
Airport Name:     New Orlean
Airport Code: MSY
Outgoing flights:
LAX -> $190.00, DFW -> $109.00
Incoming flights:
JFK -> $220.00, MIA -> $50.00

Airport Name:     Seatle
Airport Code: SEA
Outgoing flights:
ORD -> $179.50
Incoming flights:
LAX -> $200.00

1. Display airport information
2. Find the cheapest flight from one airport to another
3. Add a flight from one airport to another airport
4. Delete a flight from one airport to another airport
5. Find a cheapest roundtrip from one airport to another
9. Add a new airport
Q. Quit
```



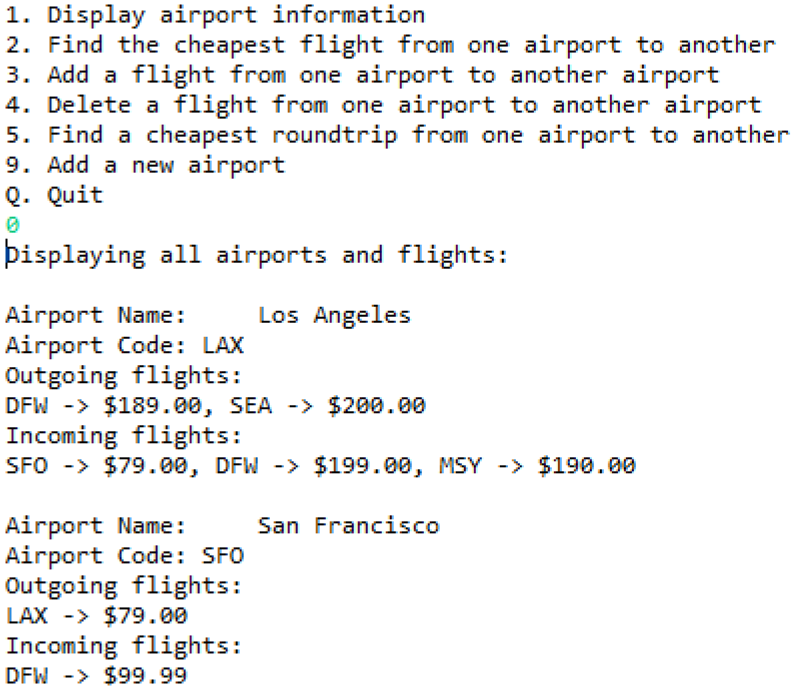
```
1
Please enter the airport code
SFO
Airport Name:     San Francisco
Airport Code: SFO
Outgoing flights:
LAX -> $79.00
Incoming flights:
DFW -> $99.99

1. Display airport information
2. Find the cheapest flight from one airport to another
3. Add a flight from one airport to another airport
4. Delete a flight from one airport to another airport
5. Find a cheapest roundtrip from one airport to another
9. Add a new airport
Q. Quit

2
Please enter the two airport codes
LAX JFK
LAX-->SEA, SEA-->ORD, ORD-->BOS, BOS-->JFK
Total Cost: $657.50


1. Display airport information
2. Find the cheapest flight from one airport to another
3. Add a flight from one airport to another airport
4. Delete a flight from one airport to another airport
5. Find a cheapest roundtrip from one airport to another
9. Add a new airport
Q. Quit
2
Please enter the two airport codes
JFK LAX
JFK-->MIA, MIA-->MSY, MSY-->LAX
Total Cost: $289.00


1. Display airport information
2. Find the cheapest flight from one airport to another
3. Add a flight from one airport to another airport
4. Delete a flight from one airport to another airport
5. Find a cheapest roundtrip from one airport to another
9. Add a new airport
Q. Quit
4
Please enter the two airport codes
LAX SFO
There is already no flight from LAX to SFO
```



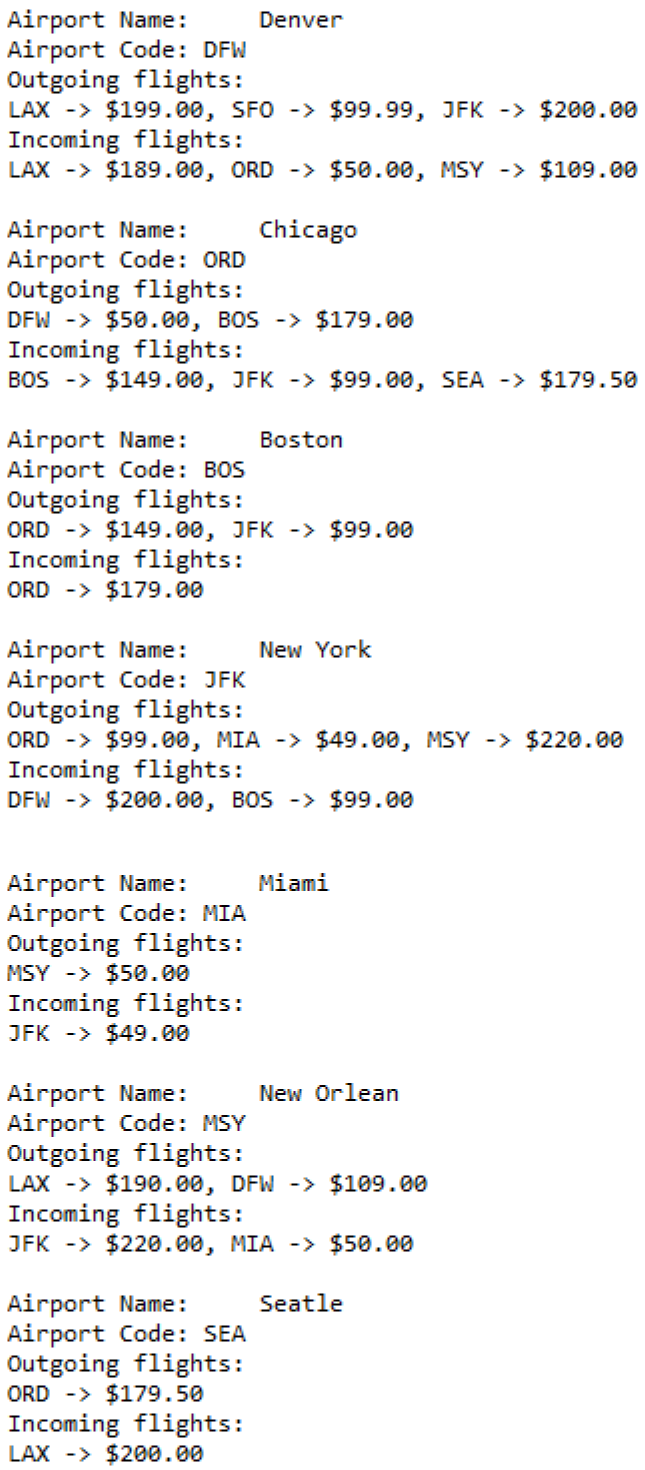
```
1. Display airport information
2. Find the cheapest flight from one airport to another
3. Add a flight from one airport to another airport
4. Delete a flight from one airport to another airport
5. Find a cheapest roundtrip from one airport to another
9. Add a new airport
Q. Quit
3
Please enter the two airport codes and the cost of flight
DFW JFK 200.00
Flight has been added!
```



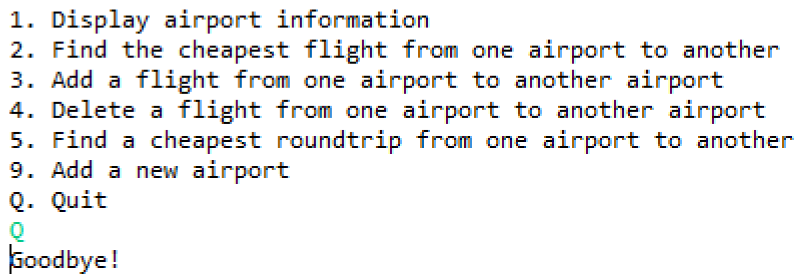
```
1. Display airport information
2. Find the cheapest flight from one airport to another
3. Add a flight from one airport to another airport
4. Delete a flight from one airport to another airport
5. Find a cheapest roundtrip from one airport to another
9. Add a new airport
Q. Quit
5
Please enter the two airport codes
LAX JFK
LAX-->DFW, DFW-->JFK
Total Cost: $389.00

JFK-->MIA, MIA-->MSY, MSY-->LAX
Total Cost: $289.00
```



```
1. Display airport information
2. Find the cheapest flight from one airport to another
3. Add a flight from one airport to another airport
4. Delete a flight from one airport to another airport
5. Find a cheapest roundtrip from one airport to another
9. Add a new airport
Q. Quit
0
Displaying all airports and flights:

Airport Name:     Los Angeles
Airport Code: LAX
Outgoing flights:
DFW -> $189.00, SEA -> $200.00
Incoming flights:
SFO -> $79.00, DFW -> $199.00, MSY -> $190.00

Airport Name:     San Francisco
Airport Code: SFO
Outgoing flights:
LAX -> $79.00
Incoming flights:
DFW -> $99.99
```

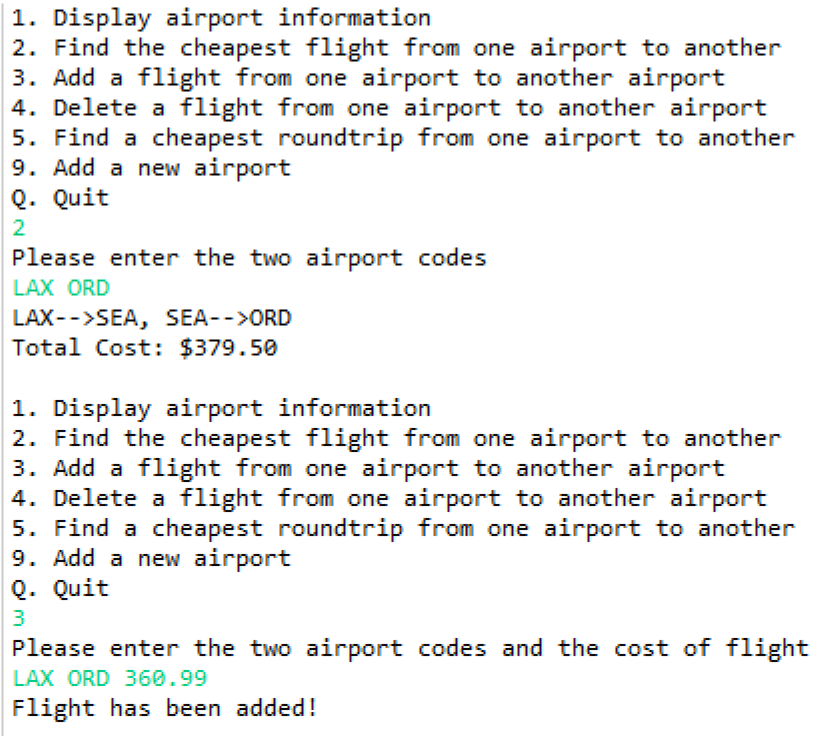


```
Airport Name:     Denver
Airport Code: DFW
Outgoing flights:
LAX -> $199.00, SFO -> $99.99, JFK -> $200.00
Incoming flights:
LAX -> $189.00, ORD -> $50.00, MSY -> $109.00

Airport Name:     Chicago
Airport Code: ORD
Outgoing flights:
DFW -> $50.00, BOS -> $179.00
Incoming flights:
BOS -> $149.00, JFK -> $99.00, SEA -> $179.50

Airport Name:     Boston
Airport Code: BOS
Outgoing flights:
ORD -> $149.00, JFK -> $99.00
Incoming flights:
ORD -> $179.00

Airport Name:     New York
Airport Code: JFK
Outgoing flights:
ORD -> $99.00, MIA -> $49.00, MSY -> $220.00
Incoming flights:
DFW -> $200.00, BOS -> $99.00


Airport Name:     Miami
Airport Code: MIA
Outgoing flights:
MSY -> $50.00
Incoming flights:
JFK -> $49.00

Airport Name:     New Orlean
Airport Code: MSY
Outgoing flights:
LAX -> $190.00, DFW -> $109.00
Incoming flights:
JFK -> $220.00, MIA -> $50.00

Airport Name:     Seatle
Airport Code: SEA
Outgoing flights:
ORD -> $179.50
Incoming flights:
LAX -> $200.00
```

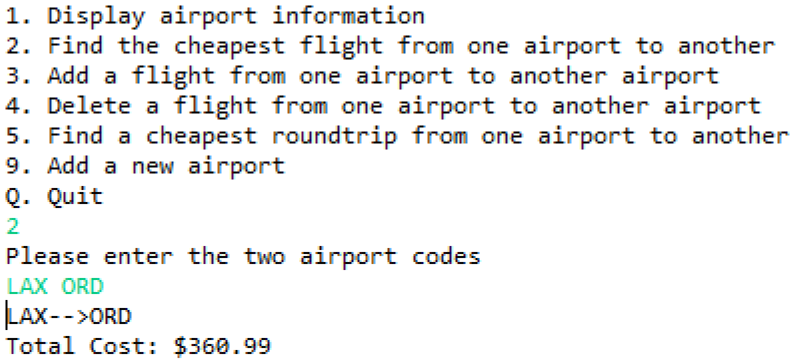


```
1. Display airport information
2. Find the cheapest flight from one airport to another
3. Add a flight from one airport to another airport
4. Delete a flight from one airport to another airport
5. Find a cheapest roundtrip from one airport to another
9. Add a new airport
Q. Quit
Q
Goodbye!
```

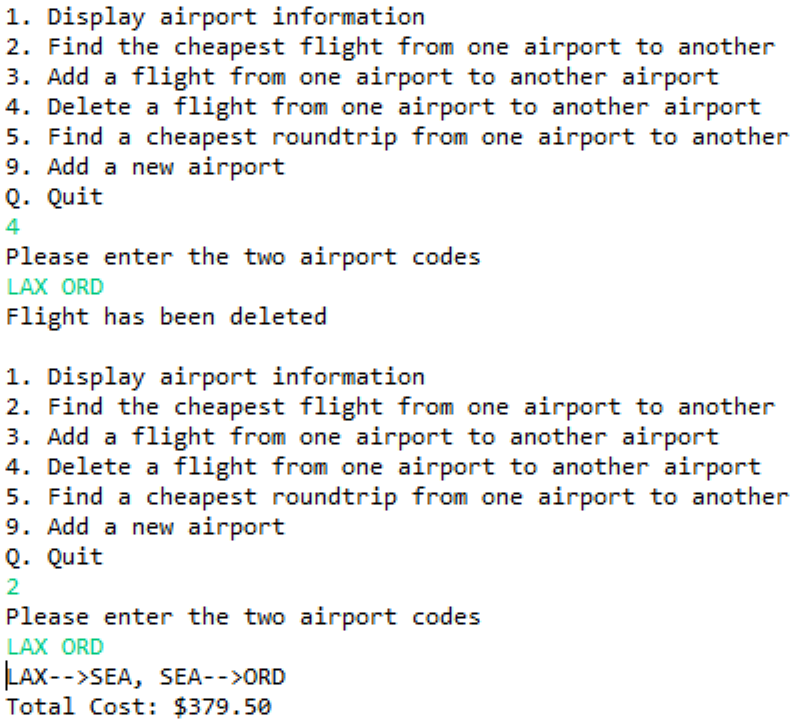
Output – Additional



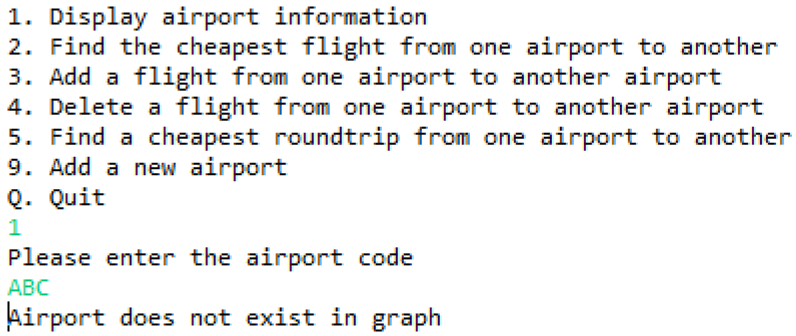
```
1. Display airport information
2. Find the cheapest flight from one airport to another
3. Add a flight from one airport to another airport
4. Delete a flight from one airport to another airport
5. Find a cheapest roundtrip from one airport to another
9. Add a new airport
Q. Quit
2
Please enter the two airport codes
LAX ORD
LAX-->SEA, SEA-->ORD
Total Cost: $379.50

1. Display airport information
2. Find the cheapest flight from one airport to another
3. Add a flight from one airport to another airport
4. Delete a flight from one airport to another airport
5. Find a cheapest roundtrip from one airport to another
9. Add a new airport
Q. Quit
3
Please enter the two airport codes and the cost of flight
LAX ORD 360.99
Flight has been added!
```



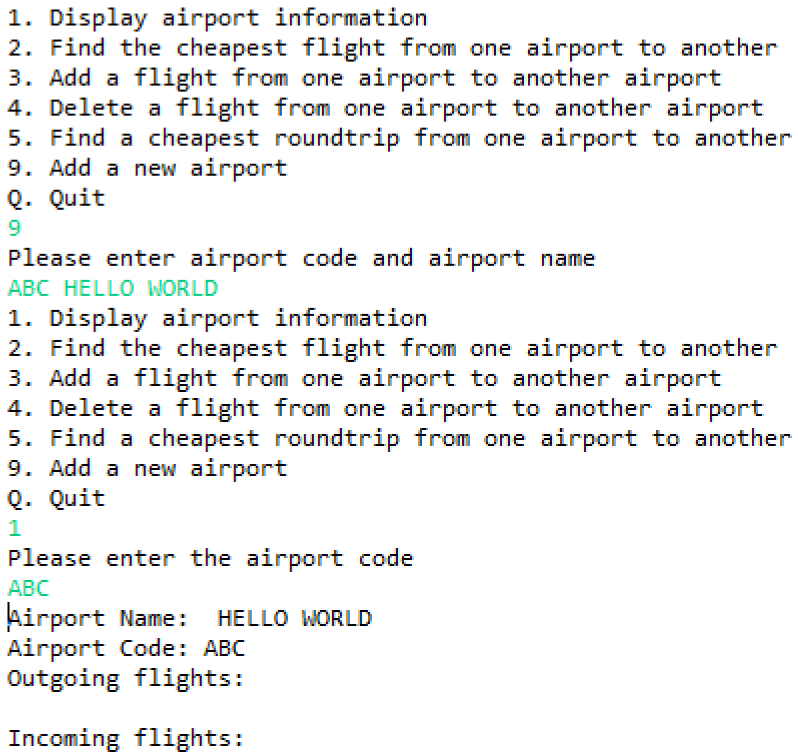
```
1. Display airport information
2. Find the cheapest flight from one airport to another
3. Add a flight from one airport to another airport
4. Delete a flight from one airport to another airport
5. Find a cheapest roundtrip from one airport to another
9. Add a new airport
Q. Quit
2
Please enter the two airport codes
LAX ORD
LAX-->ORD
Total Cost: $360.99
```



```
1. Display airport information
2. Find the cheapest flight from one airport to another
3. Add a flight from one airport to another airport
4. Delete a flight from one airport to another airport
5. Find a cheapest roundtrip from one airport to another
9. Add a new airport
Q. Quit
4
Please enter the two airport codes
LAX ORD
Flight has been deleted

1. Display airport information
2. Find the cheapest flight from one airport to another
3. Add a flight from one airport to another airport
4. Delete a flight from one airport to another airport
5. Find a cheapest roundtrip from one airport to another
9. Add a new airport
Q. Quit
2
Please enter the two airport codes
LAX ORD
LAX-->SEA, SEA-->ORD
Total Cost: $379.50
```



```
1. Display airport information
2. Find the cheapest flight from one airport to another
3. Add a flight from one airport to another airport
4. Delete a flight from one airport to another airport
5. Find a cheapest roundtrip from one airport to another
9. Add a new airport
Q. Quit
1
Please enter the airport code
ABC
Airport does not exist in graph
```



```
1. Display airport information
2. Find the cheapest flight from one airport to another
3. Add a flight from one airport to another airport
4. Delete a flight from one airport to another airport
5. Find a cheapest roundtrip from one airport to another
9. Add a new airport
Q. Quit
9
Please enter airport code and airport name
ABC HELLO WORLD
1. Display airport information
2. Find the cheapest flight from one airport to another
3. Add a flight from one airport to another airport
4. Delete a flight from one airport to another airport
5. Find a cheapest roundtrip from one airport to another
9. Add a new airport
Q. Quit
1
Please enter the airport code
ABC
Airport Name:  HELLO WORLD
Airport Code: ABC
Outgoing flights:

Incoming flights:
```

Source Code

```
import java.io.File;

import java.io.FileNotFoundException;

import java.util.Scanner;


public class Driver

{

                              public static void main(String []args) throws FileNotFoundException

                              {

                               AdjacencyMatrixGraph<String, Double> expedia = new
AdjacencyMatrixGraph<String, Double>();

                               int userInput;


                               File file = new File("P4Airports.txt");

                               Scanner input = new Scanner(file);

                               Scanner scan = new Scanner(System.in);


                               while(input.hasNext())

                               {

                                          int index = input.nextInt();

                                          String airportCode = input.next();

                                          String airportName = input.nextLine();


                                          expedia.addVertex(airportCode, airportName, index);

                               }


                               input = new Scanner(new File("P4Flights.txt"));

                               while(input.hasNext())

                               {
```

```
    int row = input.nextInt();
    int column = input.nextInt();
    double weight = input.nextDouble();

    expedia.addEdge(row, column, weight);
}

userInput = displayMenu(scan);
while(userInput != 7)
{
    if(userInput == 0)
    {
        System.out.println("Displaying all airports and flights:\n");
        for(int i = 0; i < expedia.getVerticies(); i++)
            expedia.printAllAirport(i);

    }
    else if(userInput == 1)
    {
        System.out.println("Please enter the airport code");
        String code = scan.nextLine();
        expedia.printAirport(code);
    }
    else if(userInput == 2)
    {
        System.out.println("Please enter the two airport codes");
        String airport1 = scan.next();
        String airport2 = scan.next();
```

```
                    expedia.shortestPathLengths(airport1, airport2);
                    airport1 = scan.nextLine();
                }
                else if(userInput == 3)
                {
                    System.out.println("Please enter the two airport codes and the cost of flight");
                    String airport1 = scan.next();
                    String airport2 = scan.next();
                    double cost = scan.nextDouble();

                    expedia.addEdge(airport1, airport2, cost);

                    airport1 = scan.nextLine();

                }
                else if (userInput == 4)
                {
                    System.out.println("Please enter the two airport codes");
                    String airport1 = scan.next();
                    String airport2 = scan.next();

                    expedia.removeEdge(airport1, airport2);
                    airport1 = scan.nextLine();
                }
                else if(userInput == 5)
                {
                    System.out.println("Please enter the two airport codes");
                    String airport1 = scan.next();
                String airport2 = scan.next();


                expedia.shortestPathLengths(airport1, airport2);
                expedia.shortestPathLengths(airport2, airport1);
                airport1 = scan.nextLine();
            }
            else if(userInput == 9)
            {
                System.out.println("Please enter airport code and airport
name");
                String code = scan.next();
                String name = scan.nextLine();


                expedia.addVertex(code, name);
            }
            else System.out.println("Invalid selection");


            userInput = displayMenu(scan);
         }


         System.out.println("Goodbye!");
         /*
         Map<String, Double> cloud = algorithms.shortestPathLengths(expedia, "LAX");
         System.out.println("Shortest path to JFK: " + cloud.get("JFK"));
         */
        }


        public static int displayMenu(Scanner scan)
        {
         System.out.println("1. Display airport information");
         System.out.println("2. Find the cheapest flight from one airport to another");
         System.out.println("3. Add a flight from one airport to another airport");
         System.out.println("4. Delete a flight from one airport to another airport");
         System.out.println("5. Find a cheapest roundtrip from one airport to another");
         System.out.println("9. Add a new airport");
         System.out.println("Q. Quit");

         String input = scan.nextLine();

         switch(input)
         {
            case "0": return 0;
            case "1" : return 1;
            case "2": return 2;
            case "3": return 3;
            case "4": return 4;
            case "5": return 5;
            case "9": return 9;
            case "Q": return 7;
            default: return -1;
         }

        }
}

import java.util.ArrayList;
import java.util.Random;
```

```
public abstract class AbstractHashMap<K,V> extends AbstractMap<K,V> {
 protected int n = 0;                // number of entries in the dictionary
 protected int capacity;             // length of the table
 private int prime;                  // prime factor
 private long scale, shift;          // the shift and scaling factors

 /** Creates a hash table with the given capacity and prime factor. */
 public AbstractHashMap(int cap, int p) {
  prime = p;
  capacity = cap;
  Random rand = new Random();
  scale = rand.nextInt(prime-1) + 1;
  shift = rand.nextInt(prime);
  createTable();
 }

 /** Creates a hash table with given capacity and prime factor 109345121. */
 public AbstractHashMap(int cap) { this(cap, 109345121); }  // default prime

 /** Creates a hash table with capacity 17 and prime factor 109345121. */
 public AbstractHashMap() { this(17); }                    // default capacity

 // public methods
 /**
  * Tests whether the map is empty.
  * @return true if the map is empty, false otherwise
  */
 @Override
 public int size() { return n; }
```

```
/**
 * Returns the value associated with the specified key, or null if no such entry exists.
 * @param key  the key whose associated value is to be returned
 * @return the associated value, or null if no such entry exists
 */
@Override
public V get(K key) { return bucketGet(hashValue(key), key); }

/**
 * Removes the entry with the specified key, if present, and returns
 * its associated value. Otherwise does nothing and returns null.
 * @param key  the key whose entry is to be removed from the map
 * @return the previous value associated with the removed key, or null if no such entry exists
 */
@Override
public V remove(K key) { return bucketRemove(hashValue(key), key); }

/**
 * Associates the given value with the given key. If an entry with
 * the key was already in the map, this replaced the previous value
 * with the new one and returns the old value. Otherwise, a new
 * entry is added and null is returned.
 * @param key    key with which the specified value is to be associated
 * @param value  value to be associated with the specified key
 * @return the previous value associated with the key (or null, if no such entry)
 */
@Override
public V put(K key, V value) {
  V answer = bucketPut(hashValue(key), key, value);
  if (n > capacity / 2)              // keep load factor <= 0.5
    resize(2 * capacity - 1);        // (or find a nearby prime)
  return answer;
}

// private utilities
/** Hash function applying MAD method to default hash code. */
private int hashValue(K key) {
  return (int) ((Math.abs(key.hashCode()*scale + shift) % prime) % capacity);
}

/** Updates the size of the hash table and rehashes all entries. */
private void resize(int newCap) {
  ArrayList<Entry<K,V>> buffer = new ArrayList<>(n);
  for (Entry<K,V> e : entrySet())
    buffer.add(e);
  capacity = newCap;
  createTable();                    // based on updated capacity
  n = 0;                            // will be recomputed while reinserting entries
  for (Entry<K,V> e : buffer)
    put(e.getKey(), e.getValue());
}

// protected abstract methods to be implemented by subclasses
/** Creates an empty table having length equal to current capacity. */
protected abstract void createTable();

/**
```

```
   * Returns value associated with key k in bucket with hash value h.
   * If no such entry exists, returns null.
   * @param h  the hash value of the relevant bucket
   * @param k  the key of interest
   * @return   associate value (or null, if no such entry)
   */
  protected abstract V bucketGet(int h, K k);

  /**
   * Associates key k with value v in bucket with hash value h, returning
   * the previously associated value, if any.
   * @param h  the hash value of the relevant bucket
   * @param k  the key of interest
   * @param v  the value to be associated
   * @return   previous value associated with k (or null, if no such entry)
   */
  protected abstract V bucketPut(int h, K k, V v);

  /**
   * Removes entry having key k from bucket with hash value h, returning
   * the previously associated value, if found.
   * @param h  the hash value of the relevant bucket
   * @param k  the key of interest
   * @return   previous value associated with k (or null, if no such entry)
   */
  protected abstract V bucketRemove(int h, K k);
}
import java.util.Iterator;
/**
```

```java
 * An abstract base class to ease the implementation of the Map interface.
 *
 * The base class provides three means of support:
 * 1) It provides an isEmpty implementation based upon the abstract size() method.
 * 2) It defines a protected MapEntry class as a concrete implementation of the
 *    entry interface
 * 3) It provides implemenations of the keySet and values methods, based upon use
 *    of a presumed implementation of the entrySet method.
 *
 * @author Michael T. Goodrich
 * @author Roberto Tamassia
 * @author Michael H. Goldwasser
 */
public abstract class AbstractMap<K,V> implements Map<K,V> {

  /**
   * Tests whether the map is empty.
   * @return true if the map is empty, false otherwise
   */
  @Override
  public boolean isEmpty() { return size() == 0; }

  //---------------- nested MapEntry class ----------------
  /**
   * A concrete implementation of the Entry interface to be used
   * within a Map implementation.
   */
  protected static class MapEntry<K,V> implements Entry<K,V> {
    private K k;  // key
    private V v;  // value

    public MapEntry(K key, V value) {
      k = key;
      v = value;
    }

    // public methods of the Entry interface
    public K getKey() { return k; }
    public V getValue() { return v; }

    // utilities not exposed as part of the Entry interface
    protected void setKey(K key) { k = key; }
    protected V setValue(V value) {
      V old = v;
      v = value;
      return old;
    }

    /** Returns string representation (for debugging only) */
    public String toString() { return "<" + k + ", " + v + ">"; }
  } //----------- end of nested MapEntry class -----------

  // Provides support for keySet() and values() methods, based upon
  // the entrySet() method that must be provided by subclasses
  //---------------- nested KeyIterator class ----------------
  private class KeyIterator implements Iterator<K> {
    private Iterator<Entry<K,V>> entries = entrySet().iterator();   // reuse entrySet
    public boolean hasNext() { return entries.hasNext(); }
    public K next() { return entries.next().getKey(); }             // return key!
    public void remove() { throw new UnsupportedOperationException("remove not
supported"); }
  } //----------- end of nested KeyIterator class -----------

  //---------------- nested KeyIterable class ----------------
  private class KeyIterable implements Iterable<K> {
    public Iterator<K> iterator() { return new KeyIterator(); }
  } //----------- end of nested KeyIterable class -----------

  /**
   * Returns an iterable collection of the keys contained in the map.
   *
   * @return iterable collection of the map's keys
   */
  @Override
  public Iterable<K> keySet() { return new KeyIterable(); }

  //---------------- nested ValueIterator class ----------------
  private class ValueIterator implements Iterator<V> {
    private Iterator<Entry<K,V>> entries = entrySet().iterator();   // reuse entrySet
    public boolean hasNext() { return entries.hasNext(); }
    public V next() { return entries.next().getValue(); }           // return value!
    public void remove() { throw new UnsupportedOperationException("remove not
supported"); }
  } //----------- end of nested ValueIterator class -----------

  //---------------- nested ValueIterable class ----------------
  private class ValueIterable implements Iterable<V> {
    public Iterator<V> iterator() { return new ValueIterator(); }
  } //----------- end of nested ValueIterable class -----------

  /**
   * Returns an iterable collection of the values contained in the map.
   * Note that the same value will be given multiple times in the result
   * if it is associated with multiple keys.
   *
   * @return iterable collection of the map's values
   */
  @Override
  public Iterable<V> values() { return new ValueIterable(); }
}


import java.util.Comparator;

public abstract class AbstractPriorityQueue<K,V> implements PriorityQueue<K,V> {
  //---------------- nested PQEntry class ----------------
  /**
   * A concrete implementation of the Entry interface to be used within
   * a PriorityQueue implementation.
   */
  protected static class PQEntry<K,V> implements Entry<K,V> {
    private K k;  // key
    private V v;  // value

    public PQEntry(K key, V value) {
      k = key;
      v = value;
    }

    // methods of the Entry interface
    public K getKey() { return k; }
    public V getValue() { return v; }

    // utilities not exposed as part of the Entry interface
    protected void setKey(K key) { k = key; }
    protected void setValue(V value) { v = value; }
  } //----------- end of nested PQEntry class -----------

  // instance variable for an AbstractPriorityQueue
  /** The comparator defining the ordering of keys in the priority queue. */
  private Comparator<K> comp;

  /**
   * Creates an empty priority queue using the given comparator to order keys.
   * @param c comparator defining the order of keys in the priority queue
   */
  protected AbstractPriorityQueue(Comparator<K> c) { comp = c; }

  /** Creates an empty priority queue based on the natural ordering of its keys. */
  protected AbstractPriorityQueue() { this(new DefaultComparator<K>()); }

  /** Method for comparing two entries according to key */
  protected int compare(Entry<K,V> a, Entry<K,V> b) {
    return comp.compare(a.getKey(), b.getKey());
  }

  /** Determines whether a key is valid. */
  protected boolean checkKey(K key) throws IllegalArgumentException {
    try {
      return (comp.compare(key,key) == 0);  // see if key can be compared to itself
    } catch (ClassCastException e) {
      throw new IllegalArgumentException("Incompatible key");
    }
  }

  /**
   * Tests whether the priority queue is empty.
   * @return true if the priority queue is empty, false otherwise
   */
  @Override
  public boolean isEmpty() { return size() == 0; }
}
```

```
public interface AdaptablePriorityQueue<K,V> extends PriorityQueue<K,V> {

  /**
   * Removes the given entry from the priority queue.
   *
   * @param entry  an entry of this priority queue
   * @throws IllegalArgumentException if e is not a valid entry for the priority queue.
   */
  void remove(Entry<K,V> entry) throws IllegalArgumentException;

  /**
   * Replaces the key of an entry.
   *
   * @param entry  an entry of this priority queue
   * @param key    the new key
   * @throws IllegalArgumentException if e is not a valid entry for the priority queue.
   */
  void replaceKey(Entry<K,V> entry, K key) throws IllegalArgumentException;

  /**
   * Replaces the value of an entry.
   *
   * @param entry  an entry of this priority queue
   * @param value  the new value
   * @throws IllegalArgumentException if e is not a valid entry for the priority queue.
   */
  void replaceValue(Entry<K,V> entry, V value) throws IllegalArgumentException;
}
```

```
import java.text.DecimalFormat;

import java.util.ArrayList;


public class AdjacencyMatrixGraph<V,E>

{

                    private ArrayList<String> vertex = new ArrayList<String>();

                    private ArrayList<String> vertexName = new ArrayList<String>();

                    private Map<String, String> previous;

                    private Map<String, Double> cloud;


                    double [][]edgeMatrix = new double[10][10];

                    int numberOfEdges = 0;
```

```
public AdjacencyMatrixGraph() {}

public void addVertex(String v, String vName, int index)
{
 if(vertex.contains(v))
        System.out.println("Vertex is already in the graph");
 else
        {
                vertex.add(index, v);
                vertexName.add(index, vName);
        };

}
public void addVertex(String v, String vName)
{
 if(vertex.contains(v))
        System.out.println("Vertex is already in the graph");
 else
        {
                vertex.add(v);
                vertexName.add(vName);
        };

}
public void addEdge(String u, String v, double weight)
{
 if(!vertex.contains(u))
 System.out.println("Vertex " + u + " is not in the graph");
 else if(!vertex.contains(v))
         System.out.println("Vertex " + v + " is not in the graph");


 else
 {
         int row = vertex.indexOf(u);
         int column = vertex.indexOf(v);


         numberOfEdges++;
         edgeMatrix[row][column] = weight;
         System.out.println("Flight has been added!\n");
 }
}
public void addEdge(int u, int v, double weight)
{
 numberOfEdges++;
 edgeMatrix[u][v] = weight;
}
public boolean hasEdge(String u, int v)
{
 int row = vertex.indexOf(u);
 if(edgeMatrix[row][v] != 0)
         return true;
 else return false;
}
public void printVertex()
{
 System.out.println(vertex);
```

```
}
public ArrayList<String> vertices() {return vertex;}


public int getVerticies(){return vertex.size();}


public double getEdge(int u, int v)
{
  return edgeMatrix[u][v];
}


public int getNumEdges() {return numberOfEdges;}
public int getEdgeSize() {return edgeMatrix.length;}


public void removeEdge(String row, String column)
{
  if(!vertex.contains(row))
          System.out.println("Vertex " + row + " is not in the graph");


  else if(!vertex.contains(column))
                  System.out.println("Vertex " + column + " is not in the graph");


  else if(edgeMatrix[vertex.indexOf(row)][vertex.indexOf(column)] == 0)
          System.out.println("There is already no flight from " + row + " to " +
column);


  else {
          edgeMatrix[vertex.indexOf(row)][vertex.indexOf(column)] = 0;
          System.out.println("Flight has been deleted\n");
  }
```

```
}

public void printEdge()
{
 for(int i = 0; i < edgeMatrix.length; i++)
 {
        for(int j = 0; j < edgeMatrix.length; j++)
        {
                if(edgeMatrix[i][j] == 1)
                        {
                                System.out.println(vertex.get(i) + " to " +
vertex.get(j));
                        }
        }
 }
}
public String indexOf(int i) {return vertex.get(i);}
public double getEdgeWeight(String u, String i)
{
 return edgeMatrix[vertex.indexOf(u)][vertex.indexOf(i)];
}
public void printTable()
{

 System.out.print("  ");
 for(int i = 0; i < vertex.size(); i++)
        System.out.print(vertex.get(i) + " ");

 System.out.println();
```

```
             for(int i = 0; i < vertex.size(); i++)
             {
                 System.out.print(vertex.get(i) + " ");
                 for(int j = 0; j < vertex.size(); j++)
                 {
                     System.out.print(edgeMatrix[i][j] + " ");
                 }
                 System.out.println();
             }
            }

            public void printAllAirport(int i)
            {
             String airportCode = vertex.get(i);
             printAirport(airportCode);
            }

            public void printAirport(String airportCode)
            {
             DecimalFormat df = new DecimalFormat("0.00");
             Boolean commaNeeded = false;

             if(vertex.contains(airportCode))
             {
                 System.out.println("Airport Name: " +
vertexName.get(vertex.indexOf(airportCode)));
                 System.out.println("Airport Code: " + airportCode);

                 System.out.println("Outgoing flights:");
```

```
                int row, column;
                row = column = vertex.indexOf(airportCode);

                for(int i = 0; i < vertex.size(); i++)
                {
                    if(edgeMatrix[row][i] > 0)
                    {
                        if(commaNeeded) System.out.print(", ");
                        System.out.print(vertex.get(i) + " -> $" +
df.format(edgeMatrix[row][i]));
                        commaNeeded = true;
                    }
                }

                System.out.println("\nIncoming flights:");
                commaNeeded = false;

                for(int i = 0; i < vertex.size(); i++)
                {


                    if(edgeMatrix[i][column] > 0)
                    {
                        if(commaNeeded) System.out.print(", ");
                        System.out.print(vertex.get(i) + " -> $" +
df.format(edgeMatrix[i][column]));
                        commaNeeded = true;
                    }

                }
```

```
        System.out.println("\n");
 }
 else System.out.println("Airport does not exist in graph");
}

public void shortestPathLengths(String src, String dest) {
  // d.get(v) is upper bound on distance from src to v
  Map<String, Double> d = new ProbeHashMap<>();
  // map reachable v to its d value
  cloud = new ProbeHashMap<>();
  // pq will have vertices as elements, with d.get(v) as key
  AdaptablePriorityQueue<Double, String> pq;
  pq = new HeapAdaptablePriorityQueue<>();
  // maps from vertex to its pq locator
  Map<String, Entry<Double,String>> pqTokens;
  pqTokens = new ProbeHashMap<>();
  previous = new ProbeHashMap<>();

  // for each vertex v of the graph, add an entry to the priority queue, with
  // the source having distance 0 and all others having infinite distance
  for (String v : vertices()) {
   if (v.equals(src))
    d.put(v,0.0);
   else
    d.put(v, Double.MAX_VALUE);
   pqTokens.put(v, pq.insert(d.get(v), v));      // save entry for future updates
  }
  // now begin adding reachable vertices to the cloud
  while (!pq.isEmpty()) {
   Entry<Double, String> entry = pq.removeMin();
   double key = entry.getKey();
   String u = entry.getValue();
   cloud.put(u, key);                  // this is actual distance to u
   pqTokens.remove(u);                 // u is no longer in pq
   for (int i = 0; i < edgeMatrix.length; i++) {

         if(hasEdge(u, i))
         {

                String v = vertex.get(i);
                if (cloud.get(v) == null) {
      // perform relaxation step on edge (u,v)
      double wgt = getEdgeWeight(u, v);
      if (d.get(u) + wgt < d.get(v)) {        // better path to v?
       d.put(v, d.get(u) + wgt);              // update the distance
       pq.replaceKey(pqTokens.get(v), d.get(v));   // update the pq entry
       previous.put(v,u);
     }
     }
    }
   }
  }
  printShortestPath(src, dest);
 }

public void printShortestPath(String src, String dest)
{
 ArrayList<String> cheapest = new ArrayList<>();
```

```
            String traversal = dest;

            boolean commaNeeded = false;

            double totalCost = 0;

            DecimalFormat df = new DecimalFormat("0.00");


            while(traversal != null)

            {

                cheapest.add(traversal);

                traversal = previous.get(traversal);

            }

            for(int i = cheapest.size()-1; i > 0; i--)

            {

                if(commaNeeded) System.out.print(", ");

                System.out.print(cheapest.get(i) + "-->" + cheapest.get(i-1));

                commaNeeded = true;


           totalCost+=edgeMatrix[vertex.indexOf(cheapest.get(i))][vertex.indexOf(cheapest
.get(i-1))];

            }

            System.out.println("\nTotal Cost: $" + df.format(totalCost) + "\n");




           }

}
```

```
import java.util.Comparator;

/**
 * Comparator based on the compareTo method of a Comparable element type.
 *
 * @author Michael T. Goodrich
 * @author Roberto Tamassia
 * @author Michael H. Goldwasser
 */
public class DefaultComparator<E> implements Comparator<E> {
  /**
   * Compares two elements.
   *
   * @return a negative integer if <tt>a</tt> is less than <tt>b</tt>,
   * zero if <tt>a</tt> equals <tt>b</tt>, or a positive integer if
   * <tt>a</tt> is greater than <tt>b</tt>
   */
  @SuppressWarnings({"unchecked"})
  public int compare(E a, E b) throws ClassCastException {
    return ((Comparable<E>) a).compareTo(b);
  }
}


public interface Entry<K,V> {
  /**
   * Returns the key stored in this entry.
   * @return the entry's key
   */
  K getKey();

  /**
   * Returns the value stored in this entry.
   * @return the entry's value
   */
  V getValue();
}


import java.util.Comparator;

class HeapAdaptablePriorityQueue<K,V> extends HeapPriorityQueue<K,V>
                                    implements AdaptablePriorityQueue<K,V> {

  //---------------- nested AdaptablePQEntry class ----------------
  /** Extension of the PQEntry to include location information. */
  protected static class AdaptablePQEntry<K,V> extends PQEntry<K,V> {
    private int index;          // entry's current index within the heap
    public AdaptablePQEntry(K key, V value, int j) {
      super(key, value);        // this sets the key and value
      index = j;                // this sets the new field
    }
    public int getIndex() { return index; }
    public void setIndex(int j) { index = j; }
  } //----------- end of nested AdaptablePQEntry class -----------

  /** Creates an empty adaptable priority queue using natural ordering of keys. */
  public HeapAdaptablePriorityQueue() { super(); }

  /**
   * Creates an empty adaptable priority queue using the given comparator to order
keys.
   * @param comp comparator defining the order of keys in the priority queue
   */
```

```java
public HeapAdaptablePriorityQueue(Comparator<K> comp) { super(comp);}

// protected utilites
/**
 * Validates an entry to ensure it is location-aware.
 * @param entry an entry instance
 * @return the entry cast as an AdaptablePQEntry instance
 * @throws IllegalArgumentException if the given entry was not valid
 */
protected AdaptablePQEntry<K,V> validate(Entry<K,V> entry)
                                throws IllegalArgumentException {
  if (!(entry instanceof AdaptablePQEntry))
    throw new IllegalArgumentException("Invalid entry");
  AdaptablePQEntry<K,V> locator = (AdaptablePQEntry<K,V>) entry;   // safe
  int j = locator.getIndex();
  if (j >= heap.size() || heap.get(j) != locator)
    throw new IllegalArgumentException("Invalid entry");
  return locator;
}

/** Exchanges the entries at indices i and j of the array list. */
@Override
protected void swap(int i, int j) {
  super.swap(i,j);                                        // perform the swap
  ((AdaptablePQEntry<K,V>) heap.get(i)).setIndex(i);      // reset entry's index
  ((AdaptablePQEntry<K,V>) heap.get(j)).setIndex(j);      // reset entry's index
}

/** Restores the heap property by moving the entry at index j upward/downward.*/
protected void bubble(int j) {
  if (j > 0 && compare(heap.get(j), heap.get(parent(j))) < 0)
    upheap(j);
  else
    downheap(j);                   // although it might not need to move
}
// public methods

/**
 * Inserts a key-value pair and return the entry created.
 * @param key     the key of the new entry
 * @param value   the associated value of the new entry
 * @return the entry storing the new key-value pair
 * @throws IllegalArgumentException if the key is unacceptable for this queue
 */
@Override
public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
  checkKey(key);                   // might throw an exception
  Entry<K,V> newest = new AdaptablePQEntry<>(key, value, heap.size());
  heap.add(newest);                // add to the end of the list
  upheap(heap.size() - 1);         // upheap newly added entry
  return newest;
}

/**
 * Removes the given entry from the priority queue.
```

```
   *
   * @param entry  an entry of this priority queue
   * @throws IllegalArgumentException if e is not a valid entry for the priority
queue.
   */
  @Override
  public void remove(Entry<K,V> entry) throws IllegalArgumentException {
    AdaptablePQEntry<K,V> locator = validate(entry);
    int j = locator.getIndex();
    if (j == heap.size() - 1)        // entry is at last position
      heap.remove(heap.size() - 1);  // so just remove it
    else {
      swap(j, heap.size() - 1);      // swap entry to last position
      heap.remove(heap.size() - 1);  // then remove it
      bubble(j);                     // and fix entry displaced by the swap
    }
  }

  /**
   * Replaces the key of an entry.
   *
   * @param entry  an entry of this priority queue
   * @param key    the new key
   * @throws IllegalArgumentException if e is not a valid entry for the priority
queue.
   */
  @Override
  public void replaceKey(Entry<K,V> entry, K key)
                         throws IllegalArgumentException {
    AdaptablePQEntry<K,V> locator = validate(entry);
    checkKey(key);                   // might throw an exception
    locator.setKey(key);             // method inherited from PQEntry
    bubble(locator.getIndex());      // with new key, may need to move entry
  }

  /**
   * Replaces the value of an entry.
   *
   * @param entry  an entry of this priority queue
   * @param value  the new value
   * @throws IllegalArgumentException if e is not a valid entry for the priority
queue.
   */
  @Override
  public void replaceValue(Entry<K,V> entry, V value)
                           throws IllegalArgumentException {
    AdaptablePQEntry<K,V> locator = validate(entry);
    locator.setValue(value);         // method inherited from PQEntry
  }
}
```

```
import java.util.ArrayList;
import java.util.Comparator;

/**
 * An implementation of a priority queue using an array-based heap.
 *
 * @author Michael T. Goodrich
 * @author Roberto Tamassia
 * @author Michael H. Goldwasser
 */
public class HeapPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
  /** primary collection of priority queue entries */
  protected ArrayList<Entry<K,V>> heap = new ArrayList<>();

  /** Creates an empty priority queue based on the natural ordering of its keys. */
  public HeapPriorityQueue() { super(); }

  /**
   * Creates an empty priority queue using the given comparator to order keys.
   * @param comp comparator defining the order of keys in the priority queue
   */
  public HeapPriorityQueue(Comparator<K> comp) { super(comp); }

  /**
   * Creates a priority queue initialized with the respective
   * key-value pairs.  The two arrays given will be paired
   * element-by-element. They are presumed to have the same
   * length. (If not, entries will be created only up to the length of
   * the shorter of the arrays)
   * @param keys an array of the initial keys for the priority queue
   * @param values an array of the initial values for the priority queue
   */
  public HeapPriorityQueue(K[] keys, V[] values) {
    super();
    for (int j=0; j < Math.min(keys.length, values.length); j++)
      heap.add(new PQEntry<>(keys[j], values[j]));
    heapify();
  }

  // protected utilities
  protected int parent(int j) { return (j-1) / 2; }     // truncating division
  protected int left(int j) { return 2*j + 1; }
  protected int right(int j) { return 2*j + 2; }
  protected boolean hasLeft(int j) { return left(j) < heap.size(); }
  protected boolean hasRight(int j) { return right(j) < heap.size(); }

  /** Exchanges the entries at indices i and j of the array list. */
  protected void swap(int i, int j) {
    Entry<K,V> temp = heap.get(i);
    heap.set(i, heap.get(j));
    heap.set(j, temp);
  }

  /** Moves the entry at index j higher, if necessary, to restore the heap property. */
  protected void upheap(int j) {
    while (j > 0) {            // continue until reaching root (or break statement)
      int p = parent(j);
      if (compare(heap.get(j), heap.get(p)) >= 0) break; // heap property verified
      swap(j, p);
      j = p;                              // continue from the parent's location
    }
  }

  /** Moves the entry at index j lower, if necessary, to restore the heap property. */
  protected void downheap(int j) {
    while (hasLeft(j)) {              // continue to bottom (or break statement)
      int leftIndex = left(j);
      int smallChildIndex = leftIndex;     // although right may be smaller
      if (hasRight(j)) {
          int rightIndex = right(j);
          if (compare(heap.get(leftIndex), heap.get(rightIndex)) > 0)
            smallChildIndex = rightIndex;  // right child is smaller
      }
      if (compare(heap.get(smallChildIndex), heap.get(j)) >= 0)
        break;                            // heap property has been restored
      swap(j, smallChildIndex);
      j = smallChildIndex;                // continue at position of the child
    }
  }

  /** Performs a bottom-up construction of the heap in linear time. */
  protected void heapify() {
    int startIndex = parent(size()-1);    // start at PARENT of last entry
    for (int j=startIndex; j >= 0; j--)   // loop until processing the root
      downheap(j);
  }

  // public methods
```

```
/**
 * Returns the number of items in the priority queue.
 * @return number of items
 */
@Override
public int size() { return heap.size(); }

/**
 * Returns (but does not remove) an entry with minimal key.
 * @return entry having a minimal key (or null if empty)
 */
@Override
public Entry<K,V> min() {
  if (heap.isEmpty()) return null;
  return heap.get(0);
}

/**
 * Inserts a key-value pair and return the entry created.
 * @param key     the key of the new entry
 * @param value   the associated value of the new entry
 * @return the entry storing the new key-value pair
 * @throws IllegalArgumentException if the key is unacceptable for this queue
 */
@Override
public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
  checkKey(key);      // auxiliary key-checking method (could throw exception)
  Entry<K,V> newest = new PQEntry<>(key, value);
  heap.add(newest);                     // add to the end of the list
  upheap(heap.size() - 1);              // upheap newly added entry
  return newest;
}

/**
 * Removes and returns an entry with minimal key.
 * @return the removed entry (or null if empty)
 */
@Override
public Entry<K,V> removeMin() {
  if (heap.isEmpty()) return null;
  Entry<K,V> answer = heap.get(0);
  swap(0, heap.size() - 1);             // put minimum item at the end
  heap.remove(heap.size() - 1);         // and remove it from the list;
  downheap(0);                          // then fix new root
  return answer;
}

/** Used for debugging purposes only */
private void sanityCheck() {
  for (int j=0; j < heap.size(); j++) {
    int left = left(j);
    int right = right(j);
    if (left < heap.size() && compare(heap.get(left), heap.get(j)) < 0)
      System.out.println("Invalid left child relationship");
    if (right < heap.size() && compare(heap.get(right), heap.get(j)) < 0)
      System.out.println("Invalid right child relationship");
  }
 }

}

public interface Map<K,V> {

  /**
   * Returns the number of entries in the map.
   * @return number of entries in the map
   */
  int size();

  /**
   * Tests whether the map is empty.
   * @return true if the map is empty, false otherwise
   */
  boolean isEmpty();

  /**
   * Returns the value associated with the specified key, or null if no such entry
exists.
   * @param key  the key whose associated value is to be returned
   * @return the associated value, or null if no such entry exists
   */
  V get(K key);

  /**
   * Associates the given value with the given key. If an entry with
   * the key was already in the map, this replaced the previous value
   * with the new one and returns the old value. Otherwise, a new
   * entry is added and null is returned.
   * @param key    key with which the specified value is to be associated
   * @param value  value to be associated with the specified key
   * @return the previous value associated with the key (or null, if no such entry)
   */
  V put(K key, V value);

  /**
   * Removes the entry with the specified key, if present, and returns
   * its associated value. Otherwise does nothing and returns null.
   * @param key  the key whose entry is to be removed from the map
   * @return the previous value associated with the removed key, or null if no such
entry exists
   */
  V remove(K key);

  /**
   * Returns an iterable collection of the keys contained in the map.
   *
   * @return iterable collection of the map's keys
   */
  Iterable<K> keySet();

  /**
   * Returns an iterable collection of the values contained in the map.
   * Note that the same value will be given multiple times in the result
   * if it is associated with multiple keys.
   *
   * @return iterable collection of the map's values
   */
  Iterable<V> values();

  /**
   * Returns an iterable collection of all key-value entries of the map.
   *
   * @return iterable collection of the map's entries
   */
  Iterable<Entry<K,V>> entrySet();
}


public interface PriorityQueue<K,V> {

  /**
   * Returns the number of items in the priority queue.
   * @return number of items
   */
  int size();

  /**
   * Tests whether the priority queue is empty.
   * @return true if the priority queue is empty, false otherwise
   */
  boolean isEmpty();

  /**
   * Inserts a key-value pair and returns the entry created.
   * @param key     the key of the new entry
   * @param value   the associated value of the new entry
   * @return the entry storing the new key-value pair
   * @throws IllegalArgumentException if the key is unacceptable for this queue
   */
  Entry<K,V> insert(K key, V value) throws IllegalArgumentException;

  /**
   * Returns (but does not remove) an entry with minimal key.
   * @return entry having a minimal key (or null if empty)
   */
  Entry<K,V> min();

  /**
   * Removes and returns an entry with minimal key.
   * @return the removed entry (or null if empty)
   */
  Entry<K,V> removeMin();
}


import java.util.ArrayList;
```

```
public class ProbeHashMap<K,V> extends AbstractHashMap<K,V> {
  private MapEntry<K,V>[] table;        // a fixed array of entries (all initially
null)
  private MapEntry<K,V> DEFUNCT = new MapEntry<>(null, null);   //sentinel

  // provide same constructors as base class
  /** Creates a hash table with capacity 17 and prime factor 109345121. */
  public ProbeHashMap() { super(); }

  /** Creates a hash table with given capacity and prime factor 109345121. */
  public ProbeHashMap(int cap) { super(cap); }

  /** Creates a hash table with the given capacity and prime factor. */
  public ProbeHashMap(int cap, int p) { super(cap, p); }

  /** Creates an empty table having length equal to current capacity. */
  @Override
  @SuppressWarnings({"unchecked"})
  protected void createTable() {
    table = (MapEntry<K,V>[]) new MapEntry[capacity];   // safe cast
  }

  /** Returns true if location is either empty or the "defunct" sentinel. */
  private boolean isAvailable(int j) {
    return (table[j] == null || table[j] == DEFUNCT);
  }

  /**
   * Searches for an entry with key equal to k (which is known to have
   * hash value h), returning the index at which it was found, or
   * returning -(a+1) where a is the index of the first empty or
   * available slot that can be used to store a new such entry.
   *
   * @param h the precalculated hash value of the given key
   * @param k the key
   * @return index of found entry or if not found, value -(a+1) where a is index of
first available slot
   */
  private int findSlot(int h, K k) {
    int avail = -1;                               // no slot available (thus far)
    int j = h;                                    // index while scanning table
    do {
      if (isAvailable(j)) {                       // may be either empty or defunct
        if (avail == -1) avail = j;               // this is the first available
slot!
        if (table[j] == null) break;              // if empty, search fails
immediately
      } else if (table[j].getKey().equals(k))
        return j;                                 // successful match
      j = (j+1) % capacity;                       // keep looking (cyclically)
    } while (j != h);                             // stop if we return to the start
    return -(avail + 1);                          // search has failed
  }
```

```
/**
 * Returns value associated with key k in bucket with hash value h.
 * If no such entry exists, returns null.
 * @param h  the hash value of the relevant bucket
 * @param k  the key of interest
 * @return   associate value (or null, if no such entry)
 */
@Override
protected V bucketGet(int h, K k) {
  int j = findSlot(h, k);
  if (j < 0) return null;                   // no match found
  return table[j].getValue();
}

/**
 * Associates key k with value v in bucket with hash value h, returning
 * the previously associated value, if any.
 * @param h  the hash value of the relevant bucket
 * @param k  the key of interest
 * @param v  the value to be associated
 * @return   previous value associated with k (or null, if no such entry)
 */
@Override
protected V bucketPut(int h, K k, V v) {
  int j = findSlot(h, k);
  if (j >= 0)                               // this key has an existing entry
    return table[j].setValue(v);
  table[-(j+1)] = new MapEntry<>(k, v);     // convert to proper index
  n++;
  return null;
}

/**
 * Removes entry having key k from bucket with hash value h, returning
 * the previously associated value, if found.
 * @param h  the hash value of the relevant bucket
 * @param k  the key of interest
 * @return   previous value associated with k (or null, if no such entry)
 */
@Override
protected V bucketRemove(int h, K k) {
  int j = findSlot(h, k);
  if (j < 0) return null;                   // nothing to remove
  V answer = table[j].getValue();
  table[j] = DEFUNCT;                       // mark this slot as deactivated
  n--;
  return answer;
}

/**
 * Returns an iterable collection of all key-value entries of the map.
 *
 * @return iterable collection of the map's entries
 */
@Override
```

```
  public Iterable<Entry<K,V>> entrySet() {
    ArrayList<Entry<K,V>> buffer = new ArrayList<>();
    for (int h=0; h < capacity; h++)
      if (!isAvailable(h)) buffer.add(table[h]);
    return buffer;
  }
}
```