Midterm 2

Graded

Student

Swabir Bwana

Total Points

96.5 / 100 pts

Question 1

Part 1 - Conceptual **19.5** / 20 pts

1.1 **(no title)** **3.5** / 4 pts

✔ **– 0.5 pts** imprecise / slightly incorrect runtime

□(□) time, where □ is the height of the tree – and if the tree is balanced, then □ = □(log□), leading to a search time complexity of □(log□)

✔ **– 0 pts** For unbalanced BST, it is not necessarily O(n). Rather, the runtime is O(h). Note that the height of a tree can be less than the number of nodes in the tree even when the tree is unbalanced.

1.2 **(no title)** **4** / 4 pts

✔ **– 0 pts** Correct

1.3 **(no title)** **4** / 4 pts

✔ **+ 4 pts** Identified both differences correctly

1.4 **(no title)** **4** / 4 pts

✔ **– 0 pts** Correct

1.5 **(no title)** **4** / 4 pts

✔ **– 0 pts** Correctly explains both properties

Question 2

(no title) **20** / 20 pts

2.1 **(no title)** **5** / 5 pts

✔ **– 0 pts** Correct error identification, fix, and tree

1 mostly correct, but note that in terms of syntax, 10 and 1 should be in ( )

2.2 **(no title)** **5** / 5 pts

✔ **– 0 pts** Correct

2.3 **(no title)** **5** / 5 pts

✔ **– 0 pts** correct output + correct order of cost statement

2.4 **(no title)** **5** / 5 pts

✔ **– 0 pts** Correct drawing of hash and tree maps, and explanations

Question 3

(no title) **30** / 30 pts

3.1 **(no title)** **6** / 6 pts

- ✔ **+ 1 pt** correct class signature implementing the interface
- ✔ **+ 1.5 pts** array for containing elements of the queue
- ✔ **+ 1.5 pts** front, rear, and size
- ✔ **+ 1 pt** correct constructor signature method
- ✔ **+ 1 pt** correct initialization

+ 0 pts no answer

+ 1 pt front and size (but no rear)

+ 0.5 pts inclusion of just one variable referring to index of array

+ 1 pt front and rear (but no size, rear is not also size)

+ 0.5 pts constructor should not take a parameter because SimpleQueue constructor does not take parameter

+ 0.5 pts declares class but missing "implements SimpleQueue"

+ 0.5 pts variable tracking size of array

+ 0.5 pts declares an array (but incorrect declaration)

+ 1 pt array declaration includes the element of an array but should be: private T[] array

+ 1 pt should be an array of type T and not int

+ 0.75 pts slightly incorrect class declaration

+ 0.75 pts no need to put T[] in front of array again when initializing in constructor

+ 0.5 pts initialization of array contains elements for initializing an array but should be arr (or whatever name of array is) = new T[initial capacity] (should have an initial capacity/size variable)

+ 1 pt rear and size but no front

+ 0.5 pts variable tracking the rear

+ 0.5 pts variable tracking the front

+ 0.5 pts tracked size programmatically

3.2 **(no title)** **14** / 14 pts

- ✔ **– 0 pts** Correct

3.3 **(no title)** **6** / 6 pts

- ✔ **+ 1 pt** throws correct exception
- ✔ **+ 1 pt** correct runtime
- ✔ **+ 1 pt** correct item to be returned
- ✔ **+ 1 pt** modulo f (front)
- ✔ **+ 1 pt** increase f (front)
- ✔ **+ 1 pt** correct update of size

+ 0 pts no response/nothing of substance

+ 0.5 pts throws exception but does not specify which kind

+ 0.5 pts returns null instead of throwing exception

+ 2 pts inefficient implementation with shift assuming an efficient implementation (you are capped at 2 points if you do this)

+ 2 pts correct implementation but with a linked list instead of an array (instructions specify you should use an array) (you are capped at 2 points if you do this)

+ 0.5 pts overall captures the wrap around, but check and update of f has some minor mistakes

3.4 **(no title)** **2** / 2 pts

- ✔ **– 0 pts** Correct
- ✔ **– 0 pts** Run time should be O(1)/Theta(1)

3.5 **(no title)** **2** / 2 pts

- ✔ **– 0 pts** Correct

Question 4

(no title) **27** / 30 pts

4.1 **(no title)** **2** / 3 pts

+ **0 pts** Incorrect or Missing Solution

✔ + **1 pt** Used a Map ADTs

✔ + **1 pt** Associates (from)-(to) cities as key to a list of floats of the coordinates

+ **1 pt** O(1) runtime for access and retrieval

+ **1 pt** Used a Graph ADT

4.2 **(no title)** **1** / 3 pts

+ **0 pts** Incorrect or Missing Solution

+ **1.5 pts** A reasonable data structure to store route information

+ **1.5 pts** Correct constructor

✔ + **1 pt** Correct Strings and arrays, but missing maps and constructor

+ **1 pt** Slightly incorrect data structure

+ **1 pt** Slightly incorrect constructor

4.3 **(no title)** **8** / 8 pts

✔ – **0 pts** Correct

4.4 **(no title)** **8** / 8 pts

+ **2 pts** Correct handling of no route found

+ **2 pts** Correct handling of reversed

+ **3 pts** Correct printing of coordinates

+ **1 pt** Correct formatting: i.e, output is exactly the same or very similar to the test

+ **0 pts** Incorrect

+ **2 pts** Partial credit given for printing out all coordinates, but forgot to reverse the order when the route is reversed.

+ **1 pt** Partial credit given for printing out the from & to, but do not print out the places in between

+ **0.5 pts** Partial credit given for half-correct formatting

✔ + **8 pts** Correct

– **6 pts** incorrect or insufficient pseudocode

– **1 pt** Partially correct handling of reversed (missing how to tell if coordinates need to be flipped?)

4.5 **(no title)** **8** / 8 pts

✔ **– 0 pts** Correct

CS10 Fall 2025 Midterm 2

Print your name: SWABIR MOHAMED BWANA

Exam objective

Welcome! This exam is designed to help you see how you're doing with the course material and to give you practice with technical interview-style questions. Take a deep breath—you've got this!

Exam overview

- Duration: 2 hours
- Scope: All material up to and including Day16.
- Format: On paper and closed-book. No computers or external resources are needed.
- Goal: To assess your conceptual understanding, analysis, and problem-solving skills.

What to expect on the exam

The questions will be similar in style to what you've seen in lectures, recitations, and problem sets. You can expect to find:

- Short free-answer questions.
- Exercises where you understand or fix existing code.
- Problems on developing and analyzing algorithms.
- Questions on designing object-oriented (OOP) code.

How You'll Be Graded

Your answers will be evaluated on three main criteria:

- **Correctness**: Does your solution work as intended?
- **Efficiency**: Is your approach well-designed and not overly complex?
- **Clarity**: Is your reasoning and code easy to understand?

Pro-Tip: Writing comments in your code is highly encouraged! It helps us understand your thought process and can earn you partial credit, even if the final code isn't perfect.

Rules & Logistics

Before You Start

- Please check that your exam packet has all the pages.
- Raise your hand at any point if you have a clarifying question.

Allowed Materials

- You may use one 8.5" x 11" note sheet (both sides are okay).
- Notes must be handwritten or typed in a 10-point font or larger.

Coding Questions

- All code must be written in Java.
- You can use standard Java objects (like ArrayList) unless a question specifically asks you to implement a data structure from scratch.
- Be sure to write down any helper methods or new instance variables your solution needs.
- Class declarations necessary for answering related questions are given at the back of the exam.

When You're Finished

- Please bring your completed paper exam to an instructor.

Honor Code

All work on this exam must be your own. The word and spirit of the Honor Code are in full effect. You may not work with a partner, share information, or use any resources other than what is explicitly allowed.

I read, understood, will follow the instructions and honor code, and ask for any questions or doubts.
Signature

Grade

Part	Possible	Actual
1	20	
2	20	
3	30	
4	30	
Total	**100**	

Part 1 - Conceptual [20 points]

a) [4 points] 1) What is the defining property of a Binary Search Tree? 2) Explain the primary benefit of this property in terms of the runtime efficiency of the tree's core operations (search, insertion, deletion).

. The left node < parent & right node > parent. (Generally left node < right node BST is beneficial in that it makes the runtime efficiency for the tree's core operations O(log n), or sometimes O(log h) where h is the height of the tree depending on the nature of the search.

b) [4 points] Evaluate the following claim: 'Any subtree of a valid Red-Black Tree is itself a valid Red-Black Tree.' Is this claim true or false? Justify your answer by identifying which specific property (or properties) of a Red-Black Tree could be violated.

false. Any subtree can mean ~~the~~ a red sub-node, ~~of~~ from a red parent node. That violates the rules. because the root is always black.
Plus, you can't have a red parent with a red child node.

c) [4 points] Explain two primary differences between a List and a Set. Your answer must address how each ADT handles **duplicate elements** and the **ordering of elements.**

1) List is ordered while Set is unordorded.
2) List can have multiple duplicates while a Set can't, it has unique elements.

d) [4 points] 1) Define the load factor of a hash table and give its formula in terms of n keys and m slots. 2) What fundamental assumption about our hash function allows us to use the load factor to estimate the number of collisions?

load factor $\lambda = \frac{n}{m}$ → but we operate with '%' (modulus). λ tells us how many keys to store in a slot. e.g. If there are 5 keys & 5 slots. $5/5 = 1$, so 1 key/slot.

The assumption is that we want even distribution of the keys in the slots.

e) [4 points] The data structure underlying Java's `PriorityQueue` is a min-heap. For a binary tree to be a valid min-heap, it must satisfy two critical invariants. Name and describe both of these invariants.

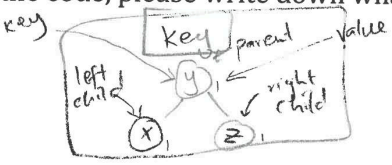
1) It has to be a complete tree meaning there should be no spaces. After one level (horizontal) is completely filled, that's when you move to the next horizontal level. The filling takes place from left to right.

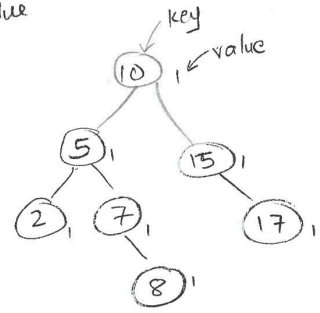
2) Since it's a min-heap, the hierarchical structure will be ordered such that the parent of every node will be less than its children. i.e The lowest value will be at the top.

Part 2 – Understanding/fixing existing code [20 points]

a) [5 points] After the execution of this code:

- Illustrate the final Binary Search Tree (BST, Attached code F)). Make it clear what is the the root, the left children and the right children, along with each of their keys and values.
- If there is any error in the code, please write down what type of error it is and fix it.





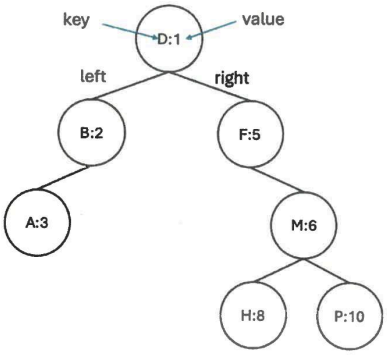

```
public static void main(String[] args){
    BST<Integer,Integer> t;
①   t.insert(10, 1);
②   t.insert(5, 1);
    t.insert(2, 1);
    t.insert(7, 1);
    t.insert(8, 1);
    t.insert(15, 1);
    t.insert(17, 1);
}
```

Error: It will throw a NullExreption Error

Fix: BST<Integer, Integer> t = new BST<insert 10, 1>;

b) [5 points] After the execution of this method, added to the BST.java (Attached code F)), at the root of the tree:

```
public List<V> question(){
    List<V> res = new ArrayList<>();
    if (hasLeft()) res.addAll(left.question());
    if (hasRight()) res.addAll(right.question());
    res.add(value);
    System.out.println(key + " " + res);
    return res;
}
```



- write down the output for the tree on the right.

Post order

L R N

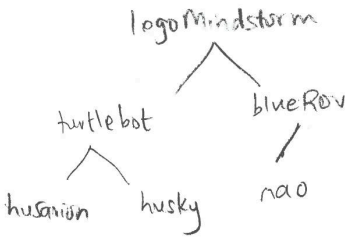
A {3}
B {3,2}
H {8}
P {8,10}
M {8,10,6}
F {8,10,6,5}
D {3,2,8,10,6,5,1}

The next question concerns the following code:

```
public class RobotSKU implements Comparable<RobotSKU>{
        private String name;
        private int id;
        private int cost;

        public RobotSKU(String name, int id, int cost) {
                this.name = name;
                this.id = id;
                this.cost = cost;
        }
        public String toString(){ return name; }

        @Override
        /**
        * Returns a negative integer, zero, or a positive integer when this object is less than,
        * equal to, or greater than the specified object (o), respectively. This is used by the priority
        queue to sort items.
        */
        public int compareTo(RobotSKU o) {
                return this.cost - o.cost;
        }
        public static void main(String[] args){
                PriorityQueue<RobotSKU> pq = new PriorityQueue<RobotSKU>();
                pq.add(new RobotSKU("husarion", 1, 2000));
                pq.add(new RobotSKU("husky", 3, 10000));
                pq.add(new RobotSKU("turtlebot", 2, 500));
                pq.add(new RobotSKU("nao", 4, 12000));
                pq.add(new RobotSKU("blueROV", 6, 1400));
                pq.add(new RobotSKU("legoMindstorm", 5, 85));

                while (!pq.isEmpty()) {
                    System.out.println(pq.poll());
                }
        }
}
```

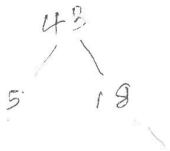
c) [5 points] How are the RobotSKU objects sorted in the priority queue above (priority queue, Attached code K) – remember that Java's implementation of a PriorityQueue uses a min-heap internally)? What is the output after running the main function?

Array → legoMindstorm, turtlebot, blueROV, husarion, husky, nao (in ascending order)



d) [5 points] Consider the following code (Map, attached code N)):

```
public static void main(String[] args) {
        HashMap<Integer, String> hash = new HashMap<Integer, String>();
        TreeMap<Integer, String> tree = new TreeMap<Integer, String>();

        //enter the *same* data into HashMap and TreeMap
        hash.put(5, "Bob");       tree.put(5, "Bob");
        hash.put(18, "Charlie");  tree.put(18, "Charlie");
        hash.put(19, "Dave");     tree.put(19, "Dave");
        hash.put(4, "Alice");     tree.put(4, "Alice");

        //print
        System.out.println("HashMap " + hash);
        System.out.println("TreeMap " + tree);
}
```



Which produces the following output:

```
HashMap {18=Charlie, 19=Dave, 4=Alice, 5=Bob}
TreeMap {4=Alice, 5=Bob, 18=Charlie, 19=Dave}
```

Assume the TreeMap uses either a Red-Black Tree or a 2-3-4 Tree (your choice) as the underlying data structure. Assume the HashMap uses a Hash Table with linear probing that begins with an initial capacity of 16 and uses hash function *h=(key % table size).*

Bob – 5
Charlie – 18
Dave – 19
Alice – 4

- Draw the data structures underlying both
  - the HashMap and
  - the TreeMap (indicating colors or 2/3/4 nodes) after data is entered,
- and then explain why, even though both are Maps, the output is in a different order.
  - Be sure to name the ordering of tree traversal. ("In order" traversal)

HashMap

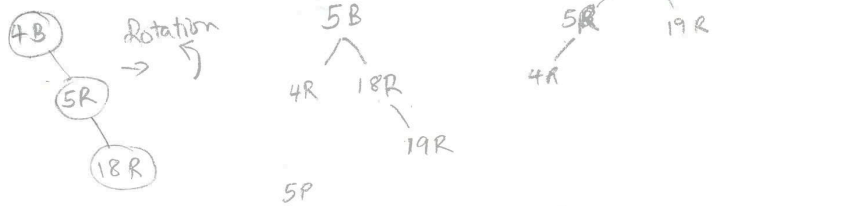
Bob 5%16 = 5
Charlie 18%16 = 2
Dave 19%16 = 3
Alice 4%16 = 4

Hash Table

indices	key
0	-
1	-
2	Charlie
3	Dave
4	Alice
5	Bob
6	-
⋮	⋮
15	

TreeMap uses the redblack tree logic. Which uses "in order" format

Key: B stands for Black node.
R stands for red node.



Explanation

- HashMap stores the Key in the hashtable randomly based on the % operation and places the keys in their respective indices. That's why the output is randomized.
- TreeMap uses redblackTree (Inorder Traversal) and since it's always a balanced tree, the output comes out sorted in ascending order.

Part 3 – Algorithm development/analysis [30 points]

Write a class that implements the SimpleQueue interface we saw in class (Attached Code I)), with an Array (instead of a linked list or ArrayList). The class should support growing the array as needed.

In addition, comment on the *worst* runtime complexity of each operation given this implementation (discuss amortized complexity if appropriate).

a) [6 points] Define the class with attributes and constructor

```
public class ArrayQueue<T> implements SimpleQueue<T> {
    Private int f=0, r=0, size=0
              ↑front  ↑rear
    private T[] queue;
    private final int initialCap = 10; ← Initial capacity
    // constructor that creates an empty queue.
    public ArrayQueue(){
        queue = (T[]) new Object [initialCap];

    }
```

b) [14 points] Write the enqueue method and its runtime complexity

```
Public void enqueue(T item){
    if (queue.length == size) {
        T[] copy = (T[]) new Object [size * 2]
        // then copy
        int i = f;
        int j = 0;
        while (j<size) {
            copy[j] = queue[i];
            i = (i+1) % queue.length;
            j++;
        }
        f = 0
        r = size
        queue = copy;
    }
    queue[r] = item;
    r = (r+1) % queue.length;
    size++
}
// The runtime complexity → O(1) (amortized)
```

This is to double the size of the array to accommodate more elements.

c) [6 points] Write the deque method and its runtime complexity

```
Public T deque() throws Exception {
    if (isEmpty()) throw new Exception ("The Queue is empty");
    T item = queue[f];
    f = (f+1) % queue.length;
    size--;
    return--;
}          // The runtime complexity is O(1)
```

d) [2 point] Write the front method and its runtime complexity

```
Public T front() throws Exception {
    if (isEmpty()) throw new Exception ("empty queue");
    return queue[f];

}   // The runtime complexity → O(1)
```

e) [2 point] Write the isEmpty method and its runtime complexity

```
Public boolean isEmpty() {
    return size == 0

}
```

Runtime Complexity → O(1)

Part 4 Problem solving with OOP [30 points]

Context: a logistics company is seeking to optimize the routing of its trucks to save on fuel and tire wear. As a preliminary step, the company is recording the routes that its drivers are currently taking.

Your objective: write a java class, `BestRouteFinder` that will:

- parse the route log file and
- save only the *best* (shortest) routes that the drivers have found between pairs of cities.

Details: The routes are stored in a file called `routes.csv`. Here are a few lines from the beginning of the file:

```
# Via Route 120 (5.7 miles distance)
Lebanon-Hanover,5.7,43.642819,-72.251549,43.665059,-72.248473,43.703083,-72.288667
#
# Via Route 10 south (7.5 miles distance)
Hanover-Lebanon,7.5,43.703083,-72.288667,43.649791,-72.310425,43.642819,-72.251549
#
# Via I-91 (exit 10A) and I-89 (exit 19). 11.5 miles distance.
Hanover-Lebanon,11.5,43.703083,-72.288667,43.705593,-72.305679,43.639948,-72.338526,43.649047,-72.248817,
43.642819,-72.251549
#
# Via I-91 (exit 13)
WRJ-Hanover,7.4,43.648890,-72.319431,43.639912,-72.338507,43.705593,-72.305679,43.703083,-72.288667
#
# Via route 10 (direct)
Hanover-WRJ,4.5,43.703083,-72.288667,43.648890,-72.319431
```

In general, note that routes are stored in the following format in the file:

```
<from>-<to>,<distance>,<fromLat>,<fromLong>,[<viaLat>,<viaLong>,...,]<toLat>,<toLong>
```

- Note that there can be a variable number of waypoints along the route between the starting and ending coordinates. But all route entries must have at least the starting and ending latitude and longitude coordinates.
- The lines starting with the pound symbol # are comments and can be ignored.

Your code must implement the following interface:

```
public interface RouteFinder {
        /**
        * Adds a route to the map if the route is not existing or it is shorter than the existing route
        * @param to the destination city of the route
        * @param from the starting city of the route
        * @param distanceAndCoords the distance (first item) and coordinates (rest of items) of the route
        * @return true if the route was added, false if it was not added
        */
        boolean addRouteIfShortest(String to, String from, float[] distanceAndCoords);

        /**
        * Print the best route from one location to another. If existing, will print the coordinates
        * in an order reflecting travel from the starting location to the ending location,
        * even if the coordinates are stored in a different order internally. If no route is found between
        * the two cities, prints "No route found between <from> and <to>".
        * @param from The starting city
        * @param to The ending city
        */
        void printBestRoute(String from, String to);
}
```

Symmetry specification: the routes should be treated as symmetrical for this problem. Therefore, if run on the data in the `routes.csv` file above, this line of code

```
finder.printBestRoute("WRJ", "Hanover");
```

should print the following to the console (note the ordering of the coordinates):

```
Best route from WRJ to Hanover (4.5 miles):
  (43.648890,-72.319431) ->
  (43.703083,-72.288667)
```

Exercises

a) [3 points] Thinking about requirements of the problem, how should the route data be stored in your `BestRouteFinder` class? More specifically, what ADT will you use and why? What will you use as the keys, values, or entries (as appropriate, depending on the ADT you choose)?

I would use a nested map ADT. So the key for the outer map (in string format) would be the "from" cities. Then its value would be another map (hence nested). For the inner map, the key would be the destination cities, in String format as well. Then its values would be an array of the distance and coordinates. This is because, with this format I can easily map the three components together, and quickly retrieve them easily

b) [3 points] Write the code for the instance variables and constructor. Note that there should be no routes stored in the object initially.

```
public class BestRouteFinder implements RouteFinder {
    // TODO: add instance variable(s) and constructor
```

```
private String mapFrom;
private String mapTo;
private float[] distanceAndCoords;
```

c) [8 points] Now write the logic for adding a route to the route finder.

- Be sure to account for the *symmetry specification* as mentioned above.
- Remember that the route should be added if
  - it is not existing
  - or if it is shorter than the existing one
- Note the boolean return type.

```
/**
* Adds a route to the map if it is not existing or shorter than the existing route
* @param to the destination city of the route
* @param from the starting city of the route
* @param distanceAndCoords the distance (first item) and coordinates (rest of items) of the route
* @return true if the route was added, false if it was not added
*/
public boolean addRouteIfShortest(String to, String from, float[] distanceAndCoords){
       // Your code here

    String mapfrom = from;
    String mapTo = to
    if (routes.containsKey(to) && routes.get(to).containsKey(from)) {
            mapfrom = to;
            mapTo = from;

    }
    if (!routes.containskey(mapfrom)) {
        routes.put(mapfrom, new Hashmap<> ());

    }
    if (!routes.get(mapfrom).containsKey(mapTo) ||
    routes.get(mapfrom).get(mapTo)[0] >distanceAndCoords[0]) {
        routes.get(mapfrom).put(mapTo, distanceAndCoords);
        return true;

    }
    if (!routes.containsKey(mapfrom)) {
        routes.put(mapfrom, new Hashmap<> ());

    }

    return false;


}
```

d) [8 points] Now write the logic for printing the best route between the specified cities.

- The output should match the behavior & format specified in the *symmetry specification.* Therefore, a route stored as WRJ to Lebanon should count when a query comes in for Lebanon to WRJ – but the order of printing the waypoints should be reversed to account for the direction of travel.
- *Hint*: Consider detecting when the stored route is in reverse order of the query, and passing off to helper functions to print the waypoints appropriately.

```
/**
* Print the best route from one location to another. If existing, will print the coordinates
* in an order reflecting travel from the starting location to the ending location,
* even if the coordinates are stored in a different order internally. If no route is found between
* the two cities, prints "No route found between <from> and <to>".
* @param from The starting city
* @param to The ending city
*/
public void printBestRoute(String from, String to) {
int // Your code here stateFound = 0 ← not found
String mapFrom = from,
String mapTo = to;
if (routes.containsKey(from)) {
    if (routes.get(from).containsKey(to))
        stateFound = 1; ← found

  else {
    if (routes.containsKey(to) && routes.get(to).containsKey(from)) {
        mapFrom = to
        mapTo = from
        stateFound = 2; ← reversed.
if (stateFound == 0) {
    System.out.println("Route could not be found between " + from + " and " + to);
    return;
}
System.out.println("The best route found was from " + from + " to " + to + " (" +
+ routes.get(mapFrom).get(mapTo)[0] + " miles);");
int i = 1
int incr = 2; continuation
}

// Feel free to write helper methods as needed
if (stateFound == 2) {
    i = routes.get(mapFrom).get(mapTo).length - 2;
    incr *= -1
}
```

continues on page 14

e) [8 points] Now write the logic for parsing the `routes.csv` file. You may assume that the city names do not have any whitespace or commas in them.

- *Hint 1:* the `split` method on the String class (Attached code C)) is useful to split a string into substrings based on a separator character. For example, `"abc:def".split(":")` will result in a 2-item string array `["abc", "def"]`
- *Hint 2:* you can convert a string `"1.245"` into a float `1.245` using the method `Float.parseFloat(String s)` - see Attached code B) .

```
public void main(String[] args) {
    String csvPath = "routes.csv";

    // TODO instantiate the BestRouteFinder class
    BestRouteFinder finder = new BestRouteFinder();


    // Read the file, parse best route from each line
    try {
        BufferedReader reader = new BufferedReader(new FileReader(csvPath));
        String line;
        while ((line = reader.readLine()) != null) {
            // TODO: parse line (be sure to handle comments),
            //       and add route if shortest, if not, just print "Route not added"

            if (line.charAt(0) == '#')
                continue;
            String[] s = line.split(",");
            float[] d = new float[s.length-1];
            for (int i=0; i< d.length; i++)
                d[i] = Float.parseFloat(s[i+1]);

            String[] fromTo = s[0].split("_");
            if (!finder.addRouteIfShortest(fromTo[1], fromTo[0], d))
                System.out.println("Route not added");
        }
        reader.close();
    } catch (IOException e) {
        System.err.println("Error reading file: " + e.getMessage());
    }

    finder.printBestRoute("Lebanon", "Hanover");
  } // End of main
} // End of BestRouteFinder class
```

(additional page – please clearly mark what question your answer is referring to)

Continuation of 4d

(blank page, usable if needed, please clearly mark what question your answer is referring to)

```
int numCoord= (routes.get(mapFrom).get(mapTo).length - 1)/2 ;
for (int counter = 0; counter < numCoord - 1 ; counter ++) {
   System.out.println ("(" + routes.get(mapFrom).get(mapTo)[i] +
   "," + routes.get(mapFrom).get(mapTo)[i+1] + ") => ");
   i += incr
}
System.out.println ("(" + routes.get(mapFrom).get(mapTo)[i] +
   "," + routes.get(mapFrom).get mapTo [i+1] + ")");
```

Attached code

A) Main method signature

```
public static void main(String[] args) {
}
```

B) Float methods

```
public final class Float extends Number {          //some built-in methods from Float
  public static float parseFloat(String s);        //given a string returns the float number
}
```

C) String methods

```
//some built-in methods of String objects
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence,
               Constable, ConstantDesc {
  public int length();
  public char charAt(int index);
  public boolean equals(Object anObject);
  public String[] split(String regex);
}
```

D) Comparing

```
public interface Comparator<T> {
  int compare(T o1, T o2);     // <0 if o1 "<" o2; >0 if o1 ">" o2; ==0 if o1 "==" o2
}

public interface Comparable<T> {
  int compareTo(T o2);                  // <0 if this "<" o2; >0 if this ">" o2; ==0 if this "==" o2
}

// As virtual method
(T o1, T o2) -> o1.variable - o2.variable;
```

E) Binary tree

```
public class BinaryTree<E> {
          private BinaryTree<E> left, right;                // children; can be null
          E data;

          public BinaryTree(E data) {
                    this.data = data; this.left = null; this.right = null;
          }
          public BinaryTree(E data, BinaryTree<E> left, BinaryTree<E> right) {
                    this.data = data; this.left = left; this.right = right;
          }
          public boolean hasLeft()                          //true if has left child
          public boolean hasRight()                 //true if has right child
          public BinaryTree<E> getLeft()                    //return left child
          public BinaryTree<E> getRight()                   //return right child
          public E getData()                        //get data for this vertex
          public boolean isInner()                          //true if inner node (not leaf)
          public boolean isLeaf()                           //true if leaf node (not inner node)
          public int size()                         //number of nodes (inner and leaf)
          public int height()                       //longest path to leaf
          public boolean equalsTree(BinaryTree<E> t2) //same structure and data?
          public ArrayList<E> fringe()              //leaves, in order from left to right
          public String toString()                          //return String representation of tree
}
```

F) Binary search tree

```
public class BST<K extends Comparable<K>,V> {
        private K key;
        private V value;
        private BST<K,V> left, right;

        public BST(K key, V value) {
                this.key = key; this.value = value;
        }
        public BST(K key, V value, BST<K,V> left, BST<K,V> right) {
                this.key = key; this.value = value;
                this.left = left; this.right = right;
        }
        public boolean isLeaf()                                  //true if leaf node
        public boolean hasLeft()                                 //true if has left child
        public boolean hasRight()                        //true if has right child
        public V find(K search) throws InvalidKeyException //find search key in tree
        public K min()                                           //find min value in tree recursively
        public K minIter                                         //find min value in tree iteratively
        public void insert(K key, V value)       //insert key and value into tree
        public BST<K,V> delete(K search) throws InvalidKeyException //delete node with key
        public String toString()                                 //string representation of tree
```

G) Stack interface

```
public interface SimpleStack<T> {
        public void push(T element);             //add element to top of stack
        public T pop() throws Exception;                 //remove element from top of stack
        public T peek() throws Exception;                //get element on top of stack, but do not
remove
        public boolean isEmpty();                        //true if stack empty
}
```

H) Singly Linked Stack

```
public class SLLStack<T> implements SimpleStack<T> {
        private Element top;                     // top of the stack

        /**
         * The linked elements
         */
        private class Element {
                private T data;
                private Element next;

                public Element(T data, Element next) {
                        this.data = data;
                        this.next = next;
                }
        }

        //methods to implement Stack interface not shown for brevity
}
```

I) Queue interface

```
public interface SimpleQueue<T> {
        public void enqueue(T item);             //add item to rear of queue
        public T dequeue() throws Exception;     //remove item from front of queue
        public T front() throws Exception;               //get item at front, do not remove
        public boolean isEmpty();                        //true if queue empty

}
```

J) Singly Linked List Queue

```
public class SLLQueue<T> implements SimpleQueue<T> {
        private Element head;          // front of the linked list
        private Element tail;          // tail of the linked list

        /**
         * The linked elements
         */
        private class Element {
                private T data;
                private Element next;

                public Element(T data) {
                        this.data = data;
                        this.next = null;
                }
        }

        //methods to implement Queue interface not shown for brevity
}
```

K) Priority Queue

```
public class PriorityQueue<E extends Comparable<E>> {
        public PriorityQueue();                  //constructor
        public boolean isEmpty();                //true if priority queue is empty, false otherwise
        public void insert(E element);           //add item to priority queue
        public E minimum();             //get min value, but do not remove
        public E remove();              //remove and return the min value from queue
        public int size();              //number of elements in priority queue
}
```

L) Simple List interface

```
//largely mirrors Java's built-in List, but Java's version has an add method without an index parameter
public interface List<E> {                       //implementation: GrowingArray, SinglyLinked
  public int size();                             //number of elements in list
  public boolean isEmpty();                      //true if List is empty
  public void add(int idx, E item);              //add item at index idx
  public void remove(int idx);                   //delete item at index idx
  public E get(int idx);                                 //return item at index idx, do not remove
  public void set(int idx, E item);              //update item at index idx
}
```

M) Set interface

```
public interface Set<E> {                                //implementations: TreeSet, HashSet
public int size();                          //number of items in Set
public boolean isEmpty();                   //true is Set is empty
public boolean add(E e);                    //add e to Set, return true if e already in Set
public boolean remove(E e);                 //remove e from Set, return true is e in Set
        public boolean contains(E e);       //true if Set contains e
        public Iterator<E> iterator()       //returns an iterator over Set
}
```

N) Map interface

```
public interface Map<K,V> {                          //implementations: TreeMap, HashMap
public int size();                          //number of items in Map
public boolean isEmpty();                   //true if Map is empty
public V put(K key, V value);               //associate key->value, return old value
public V remove(K key);                     //remove key's association, return old value
public boolean containsKey(K key);          //true if key in Map
public boolean containsValue(V value);      //true if value in Map
public V get(K key);                        //get key's associated value
Set<K> keySet();                            //all the keys
Set<Map.Entry<K,V>> entrySet();             //all the key->value pairs
}
```