COM S 211/ENGRD 211 — May 15, 2003
Final Exam — 3:00 PM – 5:30 PM

Information:

Name (clearly *print* last, first, middle): _____

Net ID: _____

CU ID: _____

I have followed the rules of academic integrity on this exam (sign): _____

Instructions:

Failure to follow any instruction may result in a point deduction on your exam:

- Turn off all cell phones, beepers, pagers, and any other devices that will interrupt the exam.
- Remove all calculators, reference sheets, or any other material. This test is closed book.
- Fill out the information at the top of this exam.
- Skim the entire exam before starting any of the problems.
- Read each problem completely before starting it.
- Solve each problem using Java, except where indicated.
- Use only the given code in each problem and follow specifications on whether or not to use the API.
- Write your solutions directly on the test using blue/black pen or pencil. Clearly indicate which problem you are solving. You may write on the back of each sheet. If you need scrap paper, ask a proctor.
- Provide only one statement, expression, value, or comment per blank!
- Do not alter, add, or remove any code that surrounds the blanks and boxes.
- Do not supply multiple answers. If you do so, we will choose which one to grade.
- Follow good style! When possible, keep solutions general, avoid redundant code, use descriptive variables, use named constants, indent substructures, avoid breaking out of loops, and maintain other tenets of programming philosophy.
- Comment each control structure, major variable, method, and class (if used), briefly.
- Do not spend too much time on any single question and budget your time based on the amount of points.
- Do not work on bonus problems until you have thoroughly proofread all required (core-point) problems!
- Figure out any problem yourself before raising your hand so that we can avoid disturbing people in cramped rooms.

Core Points:

1. _____ (10 points) _____

2. _____ (10 points) _____

3. _____ (20 points) _____

4. _____ (5 points) _____

5. _____ (20 points) _____

6. _____ (25 points) _____

7. _____ (10 points) _____

Total: _____ /(100 points) _____

Reminders

CS211 API:

```
interface SearchStructure {
   void insert(Object o);    // stick into search structure
   void delete(Object o);    // remove objects equal to o from search structure
   boolean search(Object o); // search for object o in structure
   int size();               // number of items in structure
}

interface SeqStructure {
   void put(Object o);       // stick into sequence structure
   Object get();             // extract from sequence structure
   boolean isEmpty();        // return whether or not structure has items
   int size();               // number of items in structure
}
```

Java API:

`Comparable`

`int compareTo(Object o)`: Compares this object with the specified object for order.

`HashSet`

`boolean add(Object o)`: Adds the specified element to this set if it is not already present.
`void clear()`: Removes all of the elements from this set.
`boolean contains(Object o)`: Returns **`true`** if this set contains the specified element.
`boolean isEmpty()`: Returns true if this set contains no elements.
`Iterator iterator()`: Returns an iterator over the elements in this set.
`boolean remove(Object o)`: Removes the specified element from this set if it is present.
`int size()`: Returns the number of elements in this set (its cardinality).

`Iterator`

`boolean hasNext()`: Returns **`true`** if the iteration has more elements.
`Object next()`: Returns the next element in the iteration.
`void remove()`: Removes from the underlying collection the last element returned by the iterator (optional operation).

`Object`

`boolean equals(Object obj)`: Indicates whether some other object is “equal to” this one.

Problem 1 [10 points] *General Concepts*

Answer the following questions. Be concise and clear. You may use figures in your explanations.

1a [1 point] What are the three fundamental principles of object oriented programming?

1b [1 point] What is an abstract data type?

1c [1 point] Fill in the blank: A data structure is an _____ of an ADT.
Hint: The word we want begins with the letter "i".

1d [1 point] What is a search structure?

1e [1 point] What is a sequence structure?

1f [1 point] Why does a sequence structure usually make a poor search structure? Explain your answer in terms of the **`put`** and **`get`** operations.

1g [2 points] Explain why the worst-case asymptotic time complexity for the **`contains`** method in a binary search tree is $O(n)$.

1h [2 points] Is it true that our notions of Big-Oh and asymptotic complexity are valid for *all* mathematical functions? You must explain your answer for full credit. You may give examples to support your answer.

Problem 2 [10 points] *Asymptotic Complexity*

For Problems 2a and 2b, determine whether or not each of the following relationships is true. If the relationship is true, provide a witness pair to justify your answer. If the relationship is false, justify your answer.

2a [4 points] $2^{n+1} = O(2^n)$

2b [6 points] $n^n = O(2^n)$

Problem 3 [20 points] *Inner Classes*, *Iterators*, *Linked Lists*

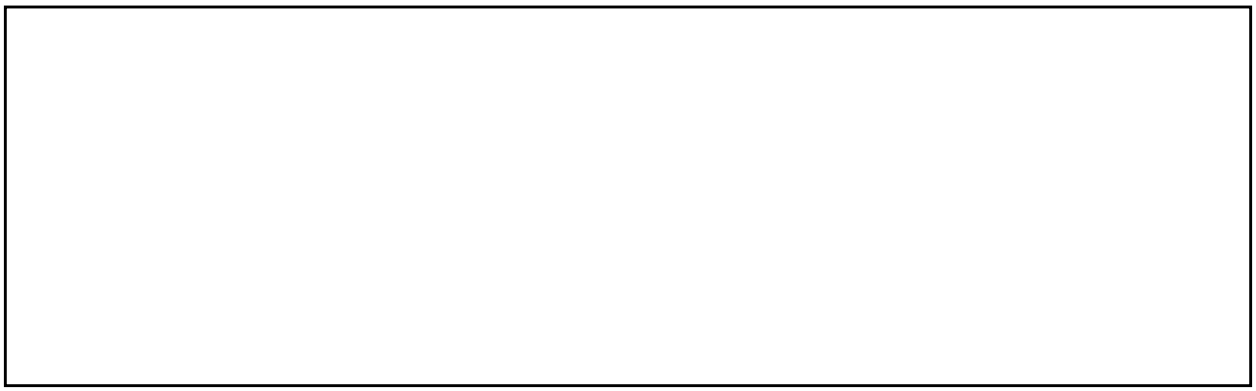
Background: The **remove** method of an iterator will remove the last item returned by **next**. For example, inside a loop for iteration, the **`next`** method might produce a "bad" value that a programmer might not want to be in the collection. If so, calling the iterator's **`remove`** method would remove that value.

Problem: You will complete method **`main`** in class **`CleanCircle`** to use a **`remove`** method that you will implement in inner class **`CircleIterator`**. In class **`CleanCircle`**, the user creates a doubly-linked circular list of a user-input length (from **`args[0]`**) with a sentinel node. Each node in the list contains a random integer, `0` to `3`, inclusive. The **`remove`** method will delete a node that contains the value `0`. For example, for the list `90210`, class **`CleanCircle`** would produce and display `921`. In the case of a list that contains all `0`s, all non-sentinel nodes are removed and an empty string is displayed.

Specifications, Assumptions, and Hints:

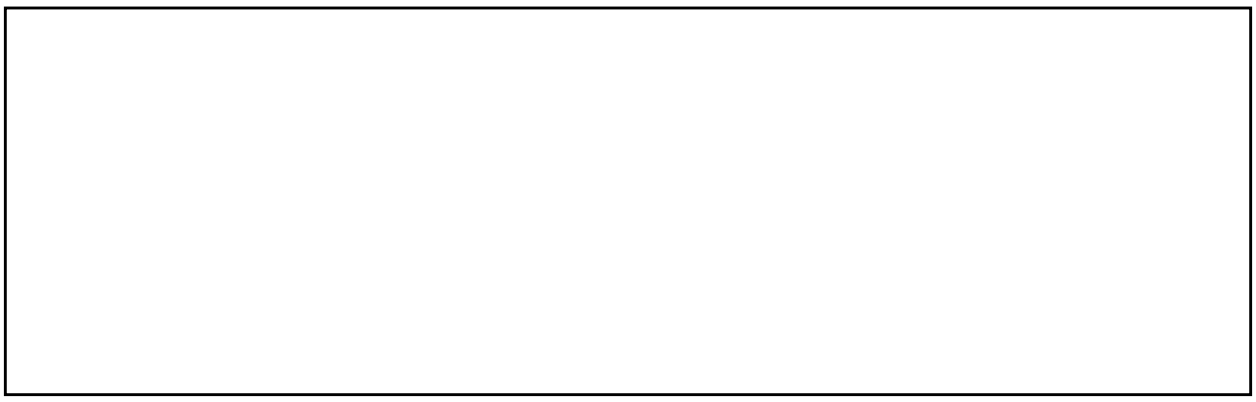
- You must use the **`CircleIterator`** inner class provided in class **`Circle`**.
- Using the previous links in class **`Node`** will *greatly* assist your solution.

```
import java.util.*;
public class CleanCircle {
   public static void main(String[] args) {

      // Create list with sentinel as first node:
         Circle c = new Circle(); // create Circle with sentinel
         Node s = c.sentinel;     // set reference s to sentinel node

      // Add nodes to list:
         Node n = s; // current node
         s.next = n; n.prev = s;
         for ( int i = 1 ; i <= Integer.parseInt(args[0]) ; i++ ) {
            Node tmp = new Node();
            n.next = tmp; tmp.prev = n; n = tmp;
         }
         n.next = s; s.prev = n;

      // Remove nodes with data of 0 from list, using CircleIterator's remove():
```



```
      // Display results:
         System.out.println(c);

   } // Method main

} // Class CleanCircle
```

```
class Node {
   public Node next; // next node
   public Node prev; // previous node
   public Object data = new Integer(MyMath.randInt(0,3));
   public String toString() { return ""+data; }
} // Class Node

class MyMath { /* code not shown */ }

class Circle {
   public Node sentinel;

   Circle() {
      sentinel = new Node();
      sentinel.next = sentinel;
      sentinel.prev = sentinel;
   }

   public class CircleIterator implements Iterator {
      private Node cursor; // current finger into list

      public CircleIterator( ) { cursor = sentinel.next; }

      public boolean hasNext( ) { return cursor != sentinel; }

      public Object next( ) {
         Object d = cursor.data;
         cursor = cursor.next;
         return d;
      }

      // Remove only a single node from list:
      public void remove( ) {
```



```
      }

   } // Class CircleIterator

   public String toString() { /* code not shown */ }

} // Class Circle
```

Problem 4 [5 points] *Code Analysis, Software Design*

Answer the following problems using Problem 3.

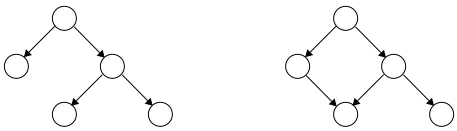
4a [2 points] What is the asymptotic time complexity of the **`main`** method in terms of *x*, where *x* represents **`Integer.parseInt(args[0])`**? Briefly explain how you determined your answer.

4b [1 point] What is the asymptotic time complexity of the **`remove`** method? Briefly explain how you determined your answer.

4c [2 points] An interesting twist we did not include is sorting of the list before removing the nodes with a `0`. How could you redesign the **`Circle`** class to make the removal process more efficient in terms of asymptotic time complexity?

Problem 5 [20 points] *Trees*, *Graphs*

Background: Each node in a tree has a unique path. If a node may be reached via multiple paths, then the underlying data structure is a graph but not a tree. See the figure below for example of both cases.



Problem: Write a method **`isTree`** that returns **`true`** if a suspected tree is indeed a tree. Otherwise, **`isTree`** returns **`false`** as in the case of a graph.

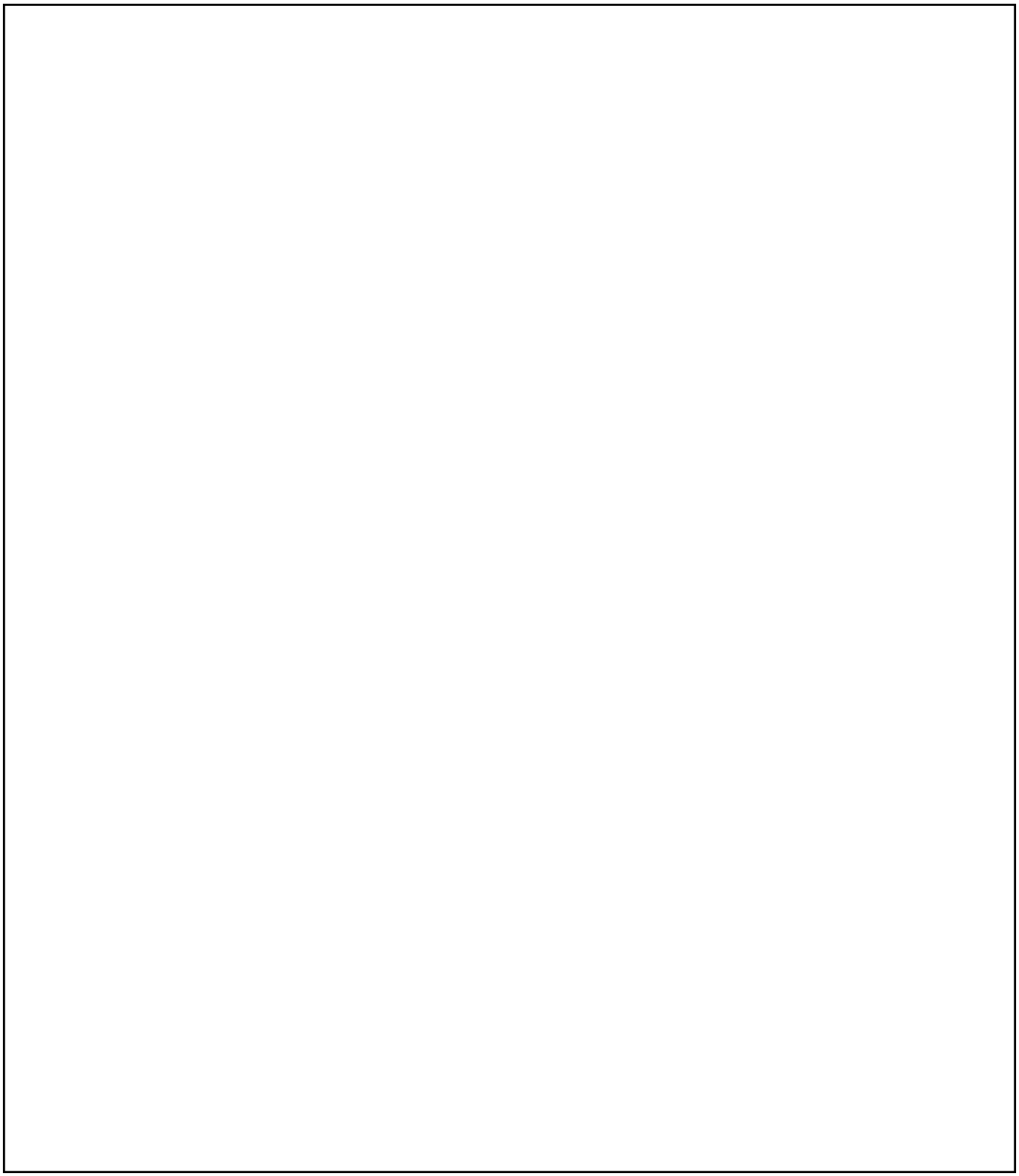
Specifications, Assumptions, and Hints:

- Refer to class **`TestTree`** for an example of how the classes and methods are used. Note that method **`main`** relies on a specific implementation of **`isTree`**.
- You must use classes **`BinaryNode`** and **`BinaryTree`**.
- You may write helper methods inside class **`BinaryTree`**, but you may not use any fields other than **`root`**.
- You might find the API's **`HashSet`** class, which is described on Page 2, very useful.

```
import java.util.*;

public class TestTree {
   public static void main(String[] args) {
      BinaryTree t = new BinaryTree();

      /* build tree: code not shown */

      System.out.println( t.isTree() );
   }
} // Class TestTree

class BinaryNode {
    public BinaryNode left;
    public BinaryNode right;
    public Object data;
    public BinaryNode(Object d) { data=d; }
} // Class BinaryNode
```

[Problem 4 continued on next page]

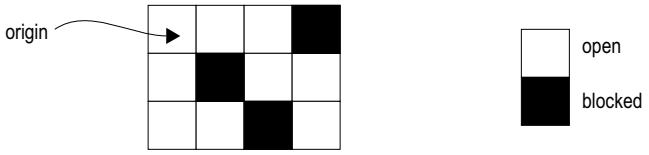
```
class BinaryTree {

   public BinaryNode root;

   // Methods to check if all nodes can be reached by only one path:
```



```
} // Class BinaryTree
```

Problem 6 [25 points] *Graphs*, *Graph Traversal*

Background: You have been given a special kind of robot to model with Java. Our robot moves *only* in perpendicular directions (north, south, east, and west) on a *rectangular* grid of spaces. Some spaces are open and some are blocked, but there is always at least one path to every open space, starting from the origin (upper, left space), which is always open. Not only *must* the robot stay in the grid, but the robot *must* stay in open spaces. The robot *must* visit every open space, starting at the origin, without getting stuck in a cycle. For instance, in the following grid



the robot branches in two directions from the origin, but will indeed reach all open spaces.

Problem: You need to complete method **`displayBFS`** in class **`TestRobot`** to write code that determines a robot's breadth-first search (BFS) traversal of a grid. The grid is represented as a two-dimensional array of **`int`**s, where **`0`** indicates an open space and anything else indicates a blocked space. Your BFS must handle *any* valid user-supplied grid, though we have given one example in method **`main`**. Class **`TestRobot`** uses classes **`Robot`** and **`Grid`**, as well as various data structures that are not shown.

Specifications, Assumptions, and Hints:

- Each instance of class **`Robot`** has a particular location in the grid. So, you need to create a new **`Robot`** object each time you make a unique move, which is a move to a space that the robot has not already visited.
- We do *not* keep track of the actual path the robot takes. The output lists all the visited nodes in an arbitrary order as determined by a search structure that stores the nodes. See the sample session for example output and variable **`bfs`** in method **`displayBFS`**.
- Refer to Page 2 to help with the **`toDo`** and **`bfs`** data structures.

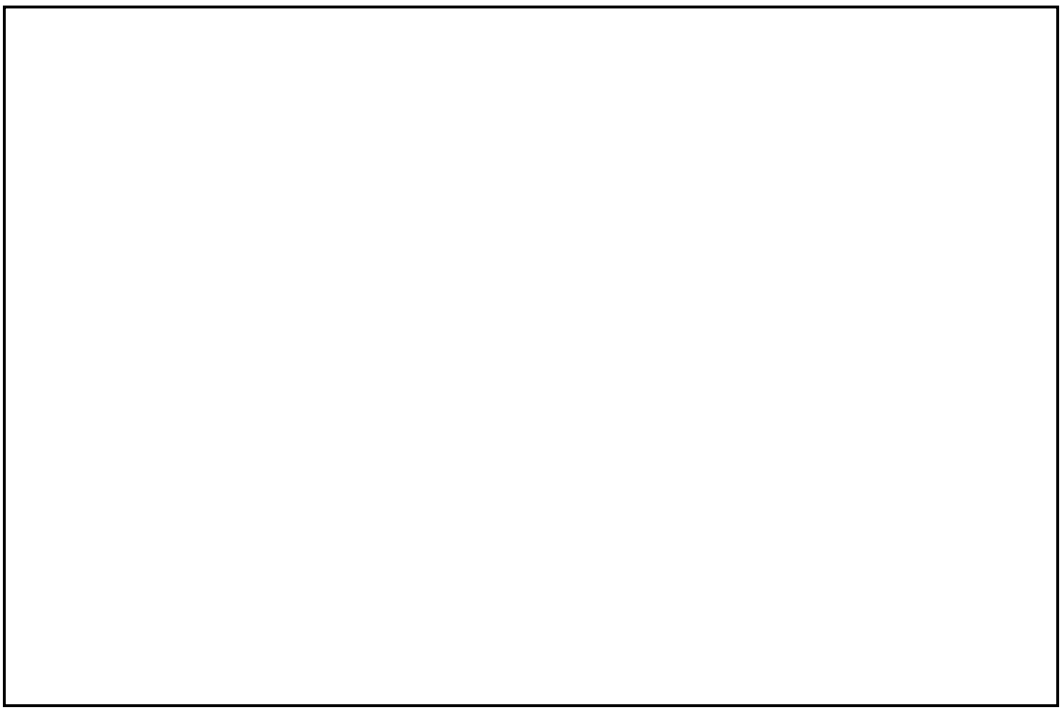
Sample Session:

The robot state is displayed as **`<rowcol>`**. For the given example, the program displays the following BFS:

```
[ <00> <10> <01> <20> <02> <21> <12> <13> <23> ]
```

```
public class TestRobot {

   public static void main(String[] args) {
      int[][] layout = new int[][] { {0,0,0,1}, {0,1,0,0}, {0,0,1,0} };
      Grid grid = new Grid(layout);
      Robot robot = new Robot(grid);
      displayBFS(robot);
   }

   public static void displayBFS(Robot robot) {

      SeqStructure toDo = new QueueAsList();// queue for processing nodes
      SearchStructure bfs = new BST();       // binary search tree for storing BFS

      // Initialize graph with origin as initial location of robot:
         toDo.put(robot);
         bfs.insert(robot);

         // continued on next page
```

```
      // Process each node and save in bfs until run out of moves:
         while(!toDo.isEmpty()) {

            // Get current robot state, which is the current node:

               Robot current = _____________________________ ;

            // Explore nodes that emanate from current node.
            // Generate each node by attempting to move robot in all directions.
            // Check if each node is legal and unvisited; if so, save node in BFS:

               String moves = "NSEW";

               for (int i = 0; i < moves.length(); i++) {

                  Robot next = _____________________________ ; // copy Robot state (node)

                  char m = moves.charAt(i);               // choose a new direction

                  boolean OK = _____________________________ ; // attempt to move Robot

                  // Was the attempt to move OK? If so, we have a node to process.
                  // Must then check if node has not been visited.
                  // If so, update toDo and bfs:
```

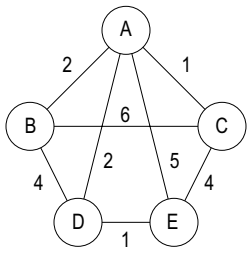


```
               } // end for

         } // end while

      // Display BFS nodes:

         System.out.println(bfs);

   } // Method displayBFS

} // Class TestRobot
```

```
class Robot implements Comparable {

   // Represent Robot state:
      public Grid grid;   // grid in which robot moves
      public int row,col; // current coordinate in grid; starts at origin

   // Create a new Robot which moves in grid:
      public Robot(Grid g) { grid = g; }

   // Attempt to move robot to open location in the direction m (N, S, E, or W).
   // Return false if location is blocked or attempting to move outside grid;
   // otherwise update the Robot state (row, col) and return true:
      public boolean move(char m) { /* code not shown */ }

   // Copy (clone) the current robot for use in generating new states:
      public Robot duplicate() {
         Robot r = new Robot(grid);
         r.row=row; r.col=col;
         return r;
      }

   // Stringify current robot state as current position in grid:
      public String toString() { return "<"+row+""+col+">"; }

   // Return true if two Robots have the same state; otherwise, return false:
      public boolean equals(Object o) { /* code not shown */ }

   // Provide way to compare to Robot states by checking if they are equal:
      public int compareTo(Object o) { /* code not shown */ }

} // Class Robot

class Grid {
   public int MINROW, MINCOL;
   public int MAXROW, MAXCOL;
   public int[][] grid;
   public Grid(int[][] grid) {
      this.grid=grid;
      MINROW = MINCOL = 0;
      MAXROW = grid.length-1;
      MAXCOL = grid[0].length-1;
   }
} // Class Grid
```

Problem 7 [10 points] Graphs

Recall that a *spanning tree* is a subset of a graph that is composed of edges such that each node is visited without forming a cycle. For this problem, you will use the following undirected weighted graph to generate different kinds of spanning trees:



7a [2 points] Should someone use an adjacency list or adjacency matrix to represent this graph? Justify your choice.

7b [2 points] Draw a breadth-first spanning tree rooted at A.

7c [2 points] Draw a depth-first spanning tree rooted at A.

7d [2 points] Draw a minimal spanning tree rooted at A.

7e [2 points] Draw a SSSP (single-source-shortest-path) tree rooted at A.