Points off	1	2	3	4	5	6		Total off	Net Score

CS 314 – Final – Spring 2013

SOLUTION - SOLUTION - SOLUTION - SOLUTION - SOLUTION - SOLUTION - SOLUTION

Your UTEID _____

Instructions:

1. There are **6** questions on this test. The test is worth 80 points. Scores will be scaled to 300 for grade center.
2. You have 3 hours to complete the test.
3. You may not use any electronic devices while taking the test.
4. When writing a method, assume the preconditions of the method are met.
5. When writing a method you may add helper methods if you wish.
6. When answering coding questions, ensure you follow the restrictions of the question.
7. Test proctors will not answer any questions.
8. When you complete the test show the proctor your UTID, give them the test and any scratch paper, and please leave the room quietly.

1. (1 points each, 20 points total) Short answer. Place you answers on the attached answer sheet.
   a. If a question contains a syntax error or other compile error, answer "Compile error".
   b. If a question would result in a runtime error or exception answer "Runtime error".
   c. If a question results in an infinite loop answer "Infinite loop".
   d. Recall when asked for Big O your answer should be the most restrictive correct Big O function. For example Selection Sort has an average case Big O of $O(N^2)$, but per the formal definition of Big O it is correct to say Selection Sort also has a Big O of $O(N^3)$ or $O(N^4)$. I want the most restrictive, correct Big O function. (Closest without going under.)

A. What is returned by the method call `a(6)?`

```
public int a(int x) {
     if(x <= 3)
          return x * 2;
     else
          return x / 2 + a(x - 1) + a(x - 1);
}
```

B. What is the order (Big O) of method `b?` N = `data.length`

```
public int b(int[] data) {
     int total = 0;
     for(int i = 0; i < data.length; i++) {

          for(int j = data.length - 1; j > 0; j /= 2)
               if(data[j] == data[i])
                    total += data[i];

          for(int j = data.length - 1; j >= 0; j -= 2)
               if(data[j] == data[i] / 2)
                    total += data[j];
     }
     return total;
}
```

C. What is the worst case order (Big O) of method `search?` N = `list.size()`.
`list` is a Java `LinkedList`.

```
public int search(LinkedList<Integer> list, int tgt) {
     int low = 0;
     int high = list.size() - 1;
     while(low <= high) {
          int mid = (low + high) / 2;
          if(list.get(mid) == tgt)
               return mid;
          else if(list.get(mid) < tgt)
               low = mid + 1;
          else
               high = mid - 1;
     }
     return -1;
}
```

D. What is the order (Big O) of method `d?` N = `n`. The method uses the Java `TreeSet` class.

```
public TreeSet<Integer> d(int n) {
     TreeSet<Integer> result = new TreeSet<Integer>();
     final int LIMIT = n;
     for(int i = 0; i < LIMIT; i)
          result.add(i);
     return result;
}
```

E. What is the order (Big O) of method `e?` N = `n`.
The method uses the `BinarySearchTree` class from lecture and assignment 10.
Assume the `add` method uses the iterative algorithm to add, not the recursive algorithm.

```
public BinarySearchTree<Integer> e(int n) {
      BinarySearchTree <Integer> result = new BinarySearchTree <Integer>();
      final int LIMIT = n;
      for(int i = 0; i < LIMIT; i++)
            result.add(i);
      return result;
}
```

F. The following values are inserted in the order shown to a binary search tree using the traditional, naïve insertion algorithm. Draw the resulting tree.

```
7, 0, -1, 12, 7, 0, 5, 9
```

G. The following values are inserted in the order shown to a min heap using the algorithm demonstrated in class. Draw the resulting tree.

```
7, 0, 12, 0, 7, 20
```

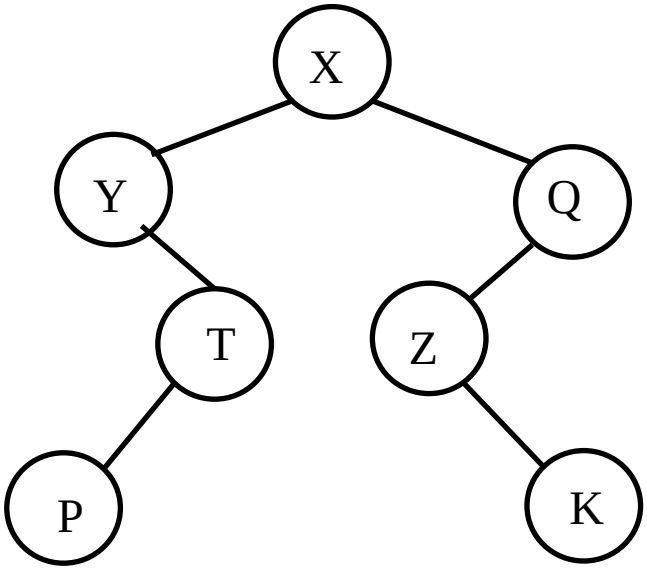
H. What is the worst case order (Big O) of method `h?`
N = `map.size()` and M = `data.length`. `map` is a Java `HashMap`.

```
public int h(HashMap<Integer, Integer> map, int[] data) {
      int result = 0;

      for(int i = 0; i < data.length; i++) {
            int temp = data[i];
            if(map.containsKey(temp))
                  result += map.get(temp);
      }
      return result;
}
```

I. Given the following timing data for the *SmoothSort* algorithm for sorting N distinct values in random order, what is the most likely average case order (big O) of the *SmoothSort* algorithm?

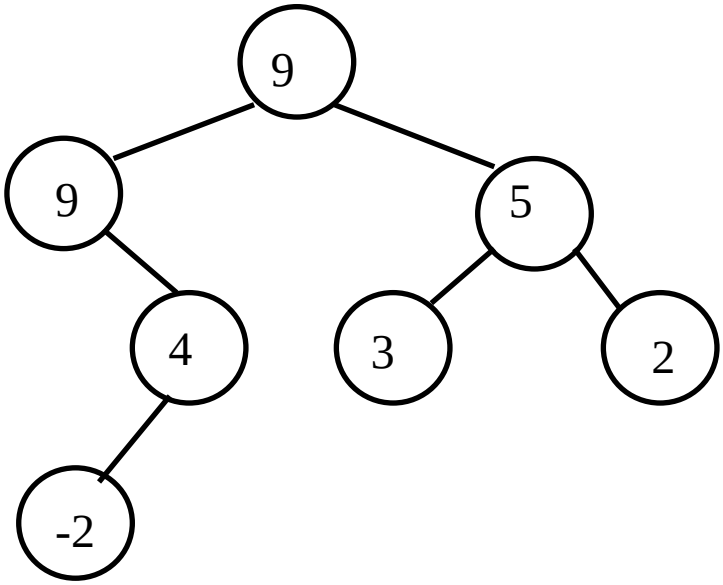
Num values	Time
1,000,000	5 seconds
2,000,000	10.5 seconds
4,000,000	22 seconds
8,000,000	46 seconds



J. What is the result of an in order traversal of the binary tree shown **above**?

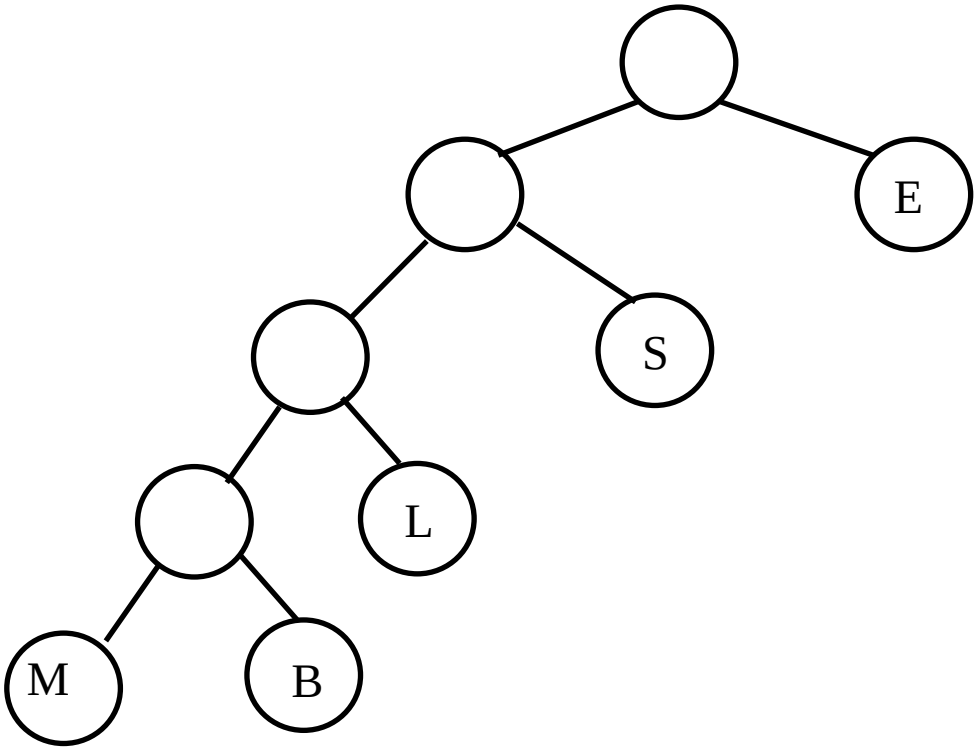
K. What is the result of a pre order traversal of the binary tree shown **above**?

L. Consider the following tree. It does not meet the requirements of a max heap. Why it is not a max heap? Be brief.



M. Suppose you want to encode 25 distinct characters names using a fixed length encoding scheme. What is the **minimum** number of bits per character required to have a unique code for each character?

N. Given the following Huffman code tree, what does the given bit stream decode to?



Given bit stream (spaces added for clarity): `0 0 0 1 1 0 0 1 0 0 1 1`

O. Consider the following class.

```
public class YetAnotherDataStructure<E> {

     // creates an empty YetAnotherDataStructure
     public YetAnotherDataStructure()

     public Iterator<E> iterator()

     // other methods and instance variables not shown
}
```

Why doesn't the following code compile? Be brief.

```
YetAnotherDataStructure<String> yads;
yads = new YetAnotherDataStructure<String>();
for(String str : yads)
     System.out.println(str);
```

P. Consider the following adjacency matrix for a directed graph.

A weighted edge exists from the vertex at the start of a row to the vertex at the top of a column if the value in the cell is greater than 0.

The cost of the edge is the integer value in the cell.

If a cell contains a 0 then no edge exists between the vertex at the start of a row and the vertex at the top of a column.

Draw the resulting graph on the answer sheet.

	A	B	C	D	E	F
A	0	0	10	2	0	3
B	0	0	10	0	1	0
C	0	0	0	0	0	0
D	0	1	0	0	5	0
E	9	0	0	0	0	0
F	0	0	0	0	5	0

Q. Given the graph from part P, what is the cost of the lowest cost path (shortest, weighted path) from vertex A to vertex E?

R Is the graph from part P a directed, acyclical graph? (Yes or No)

S. Of the four sorting algorithms we studied in class (Selection, Insertion, Quick, and Merge) which were stable?

T. Given an array of 1,000,000 distinct objects you are going to perform 100,000 searches, one at a time, using the linear search algorithm. Each element you are searching for **is present** in the array.

What is the expected, total number of time the `equals` method will be called when performing the 100,000 linear searches?

Give an actual number. Do not use Big O notation.

2. (ArrayLists and Sets - 12 points) Complete a constructor for the `UnsortedSet` class that accepts an `ArrayList` as a parameter and creates a new `UnsortedSet`. The original `ArrayList` **may have duplicate values**. Your constructor must ensure the `con` instance variable does not contain any duplicates.

The only methods you may use from the `ArrayList` class are:

```
public E get(int pos)
public int size()
public E remove(int pos) - remove value at given position
```

You may <u>not</u> use any other methods or constructors from the `ArrayList` class including iterators or the for-each loop.

The only other method you may use is the `Object` equals method. You may <u>not</u> use any other classes or methods including native arrays or other methods from the `UnsortedSet` class.

```
public class UnsortedSet<E> {

  private ArrayList<E> con; // contains elements of this UnsortedSet

  // pre: init != null, no elements of init == null
  public UnsortedSet(ArrayList<E> init) {
      con = new ArrayList(init);

      int index = 0;
      while(index < con.size()) {
            int dupIndex = 0;
            boolean unique = true;
            E current = con.get(index);
            while(dupIndex < index && unique) {
                  if(current.equals(con.get(dupIndex))
                        unique = false;
                  else
                        dupIndex++;
            }
            if(!unique)
                  con.remove(index);
            else
                  index++;
      }
}

// alt solution
      for(int i = 0; i < con.size(); i++) [
            E temp = con.get(i);
            // remove an occurrence of temp after index i
            for(int j = i + 1; j < con.size(); j++) {
                  if(con.get(j).equals(temp)) {
                        con.remove(j);
                        // back up j due to elements shifted left
                        j--;
            }
      }
```

Yet another solution. Very clever and CLEAN in that it starts from the back so no need to adjust loop control variable.

```
for(int i = con.size() - 1; i >= 0; i--) {
      E temp = con.get(i);
      for(int j = i - 1; j >= 0; j++)
            if(con.get(j).equals(temp))
                  con.remove(j);
}
```

Common problems:

- using disallowed methods or data structures
- not adjusting loop control variable
- removing things with init (or relying on init in any way)

Grading Criteria: 12 points

- outer loop to check all elements correct, 2 points
- inner loop to check other elements or remove duplicates, 3 points
- remove duplicate elements if found, 2 points
- use equals not ==, 2 points
- correctly adjusting loop control variables when removing, 3 points

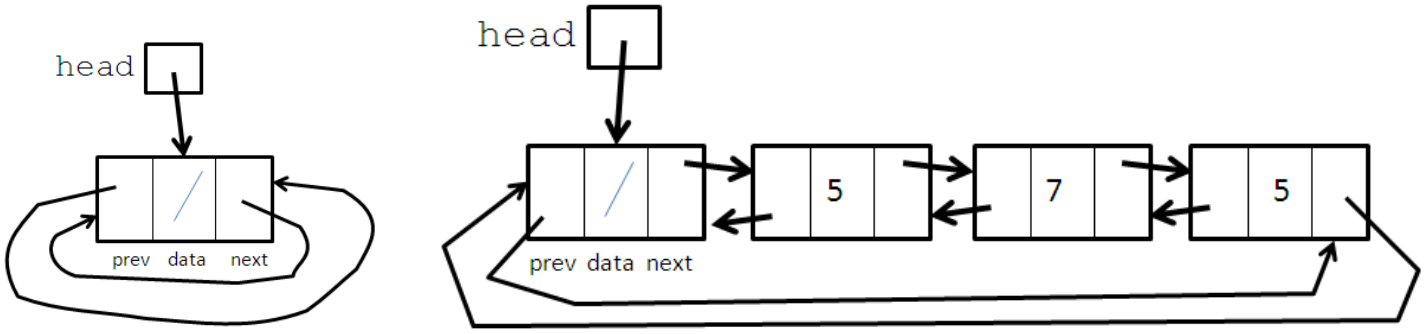
Other

- disallowed methods or data structures, -3 to -6
- alter list named init, -2
- off by one errors, -1
- wrong data type for values, -1
- [] instead of .get, -3

3. (Linked Lists - 12 points) Complete an instance method, `removePairs`, for a `LinkedList` class that stores integers. The method removes consecutive pairs of elements that equal a target `int`.

- You may not use any other methods in the `LinkedList` class unless you implement them yourself as a part of this question.
- Your solution must be O(1) space, meaning no matter how many elements are in the `LinkedList`, your solution always uses the same amount of space. In other words you can't use an auxiliary array or `List`. **Do not use recursion in your solution.**
- The `LinkedList` class uses doubly linked nodes.
- The list uses a dummy, header node. The next reference in the header node always refers to the second node in the list (unless the list is empty, in which case it refers to the header node) and the previous reference always refers to the last node in the list (unless the list is empty in which case it refers to the header node.)
- The list is circular. The last node 's next reference refers to the header node.
- **You may use the nested `Node` class. You may not use any other Java classes.**
- **Your method shall be as efficient as possible give the constraints of the question.**

The structure of the internal storage container for an empty `LinkedList` is shown below, on the left. (Arrows indicate the references stored by variables.)



The structure of the internal storarage container for a list that contains the values `[5, 7, 5]` is shown above on the right:

The method `removePairs` removes consecutive pairs of `int`s (and the nodes that contain them) that equal some target value from the linked structure of nodes.

For example if the initial list is `[2, 3, 3, 4, 6, 7]` and the target is `6` the resulting list is `[6, 7]`.

Initially the 3, 3 pair is removed. This changes the list to `[2, 4, 6, 7]`. The 2 and 4 are now consecutive elements and equal the target 6, so they must be removed.

Other examples:

```
[1, 1, 1, 1, 1, 1].removePairs(2) -> []
[1, 2, 0, 1].removePairs(2) -> []
[1, 1, 1, 1, 1, 1].removePairs(3) -> [1, 1, 1, 1, 1, 1].
[].removePairs(2) -> []
[1, 2, 2, 2, 1].removePairs(2) -> [1, 2, 2, 2, 1]
[3, 3, 3, 3, 3].removePairs(6) -> [3]
```

```
public class LinkedList {
     private Node head;
     private int size;

     // recall outer class can access private fields in nested class
     private static class Node {
          private Node prev;
          private int data;
          private Node next;
     }

     // pre: none
     // post: no consecutive pairs of elements equal tgt and
     //   size has been adjusted correctly
     public void removePairs(int tgt) {
          // SCOUT AND TRAIL NODE SOLUTION
          Node trail = header.next;
          Node scout = header.next.next;

          while(scout != header) {
               if(trail.data + scout.data == tgt)  {
                    size -= 2;
                    trail = trail.prev;
                    scout = scout.next;
                    trail.next = scout;
                    scout.prev = trail;
                    // special case if backed up to header
                    if(trail == header) {
                         trail = trail.next;
                         scout = scout.next;
                    }

               }
               else {
                    trail = trail.next;
                    scout = scout.next;
               }
          }
     }
```

Common problems:
- hard coding values, target
- assuming data is null (they were ints) or certain value for header
- not moving through list correctly
- not decrementing size
- efficiency, O(N^2) solutions instead of O(N)

```
// Look ahead solution
     public void removePairs(int tgt) {

          Node temp = header.next;
          while(temp.next != header) {
               if(temp.data + temp.next.data == tgt) {
                    size -= 2;
                    temp.prev.next = temp.next.next;
                    temp.next.next.prev = temp.prev;
                    if(temp.prev != header)
                         temp = temp.prev;

               }
               else
                    temp = temp.next;
     }
```

Grading Criteria: 12 points

- loop with correct stopping condition, 2 points
- check if consecutive values equal target, 1 point
- remove 2 nodes correctly if necessary
- adjust temp node(s) correctly when remove, 2 points
- move temp nodes correctly, 2 points
- adjust size correctly, 2 points
- handle special case when backed up to header node, 1 point

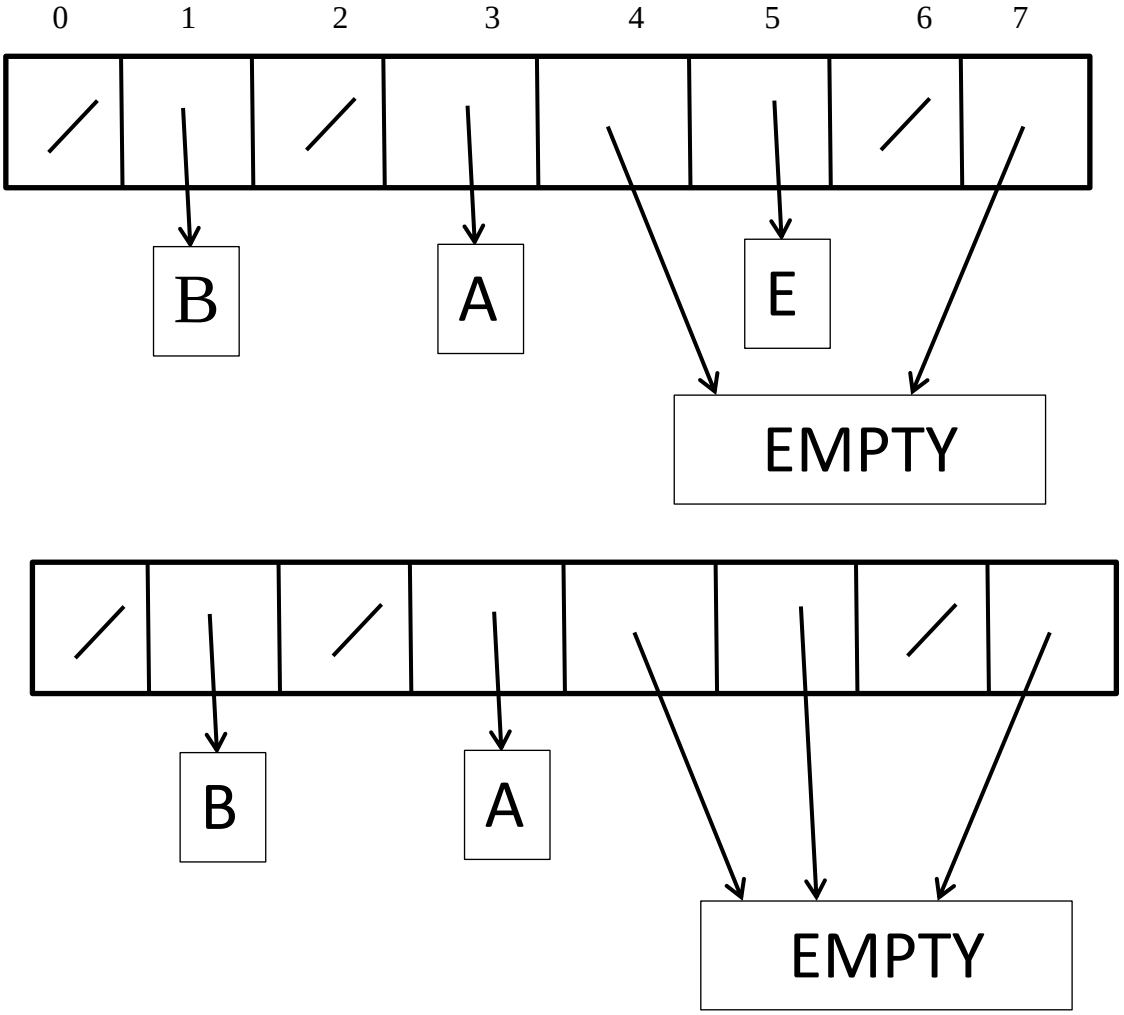
Other:

- destroy list - 5
- NPE -3
- efficiency, -2

4. (Hashtables - 12 points) Write the `remove` method for a hash table that uses *open address hashing*. In open address hashing each element in the array holds a reference to an actual object, `null`, or a reference to the `EMPTY` object if the element in the array stored an actual object at one point, but that object has subsequently been removed. Elements that refer to `EMPTY` could be used to store new values.

The hash table uses *linear probing*. Recall when adding or removing an element, linear probing moves down one spot at a time looking for an open spot (`null` or `EMPTY`) when adding or the target element when removing. The probe wraps around to the front of the array if necessary.

Considering the following example: Assume "E"'s hash code mod 8 (length of the array) is 3. When "E" was added to the hash table, there were already values at indices 3 and 4. The value at index 4 was subsequently removed. The second diagram shows the hash table after "E" has been removed.



Complete the `remove` method for the `HashTable` class on the next page. Recall the `hashCode` method from the object class, which returns the hash code for an object. You may not use any other methods or classes other than the `Object hashCode` and `equals` methods and the `Math.abs` method.

Your method shall be as efficient as possible give the constraints of the question.

```
public class HashTable<E> {

     private static final Object EMPTY = new Object();

     private int size; // number of elements in this HashTable
     private E[] con; // internal container

*/   pre: target != null
     post: target removed from this HashTable if it was present.
     Returns true if this HashTable changed as a result of this
     method call, false otherwise. Size updated properly. */
     public boolean remove(E target) {
          int index = target.hashCode();
          index = Math.abs(index) % con.length;
          int checked = 0;
          while(con[index] != null && checked < con.length) {
               if(target.equals(con[index])) {
                    con[index] = EMPTY;
                    size--;
                    return true;
               }
               index = (index + 1) % con.length;
               checked++;
          }
          return false;
     }
```

Comments:

- lots of linear searches from start of array. Showed complete misunderstanding of hash tables and how they work
- not calling hashcode method
- not taking abs of hashcode
- not taking remainder of con.length
- not wrapping during search

General Grading Criteria: 12 points

- call hashCode, 1 point
- abs of hashCode, 1 point
- % con.length, 1 point
- loop until found or hit null, 2 points
- stop if checked every element, 1 point
- check equals, 1 point (not okay to check hashCodes, == -1)
- remove by setting element equal to EMPTY, 2 points
- wrap index, 1 point
- return correct answer, 1 point
- size--, 1 point

Other:

- linear search with no use of hashcodes -7
- disallowed classes or methods, -3 to -6
- inefficient solutions, -3 to -8

5. (Red Black Trees - 12 points) This question has two parts.
Recall the path rule for a red black tree states ALL paths from the root node to the null children in the tree MUST contain the same number of BLACK nodes.
You may not use any other classes or methods except the `RBNode` class.
Your solutions shall be as efficient as possible given the constraints of the question.

Part A. 5 points. Complete a method that determines the number of black nodes in the path from the root of the tree to the node that contains the minimum value in the tree. Count the root and the node that contains the min if they are black.

```
public class RedBlackTree<E extends Comparable<E>> {

     private RBNode<E> root; // if size == 0, root == null
     private int size;

     // recall outer class can access private fields in nested class
     private static class RBNode<E> {

          private E data;

          // true if node is black, false if it is red
          private boolean isBlackNode;

          private RBNode<E> left, right;
     }

     // Complete the following method.
     // returns the number of black nodes in the path from the
     // root node to the node in the tree that contains the min
     private int blackNodesInRootToMinPath() {
          RBNode temp = root;
          int numBlackNodes = 0;
          while(temp != null) {
               if(temp.isBlackNode)
                    numBlackNodes++;
               temp = temp.left;
          }
          return numBlackNodes;
     }
```

General Grading criteria: 5 points

- temp node starting at root, 1 point
- loop until temp is null, 1 point
- check if black node and increment number of black nodes, 1 point
- move to left, 1 point
- return answer, 1 point

Other: -1 if inefficient in terms of time or space, -3 destroy tree, OBOE -1

Part B - 7 points. Complete an instance method for the `RedBlackTree` class that determines if the path rule is met.

- Call the `blackNodesInRootToMinPath` once.
- **You may not use any other classes or methods except the `RBNode` class and the `blackNodesInRootToMinPath`.**
- **Your solutions shall be as efficient as possible given the constraints of the question.**
- **Hint: You should add a helper method.**

```
// complete the following instance method for the RedBlackTree class.
private boolean pathRuleMet() {
     int magicNum = blackNodesInRootToMinPath;
     return helper(root, magicNum);
}

private boolean helper(RBNode<E> n, int goalBlackNodesInPath) {
     // success base case
     if(n == null && goalBlackNodesInPath == 0)
          return true;

     // simple failure base case
     else if(n == null || goalBlackNodesInPath <= 0
          return false;

     // recursive case
     else {
          if(n.isBlackBode)
               goalBlackNodesInPath--;
          return helper(n.left, goalBlackNodesInPath)
               && helper(n.right, goalBlackNodesInPath);
}
```

General Grading criteria: 7 points

- get magic number correctly, 1 point
- create and call correct helper method, 1 point
- success base case, 1 point
- simple failure base case, 1 point
- check if current node black in recursive case, 1 point
- make correct recursive calls and return correct answer, 2 points

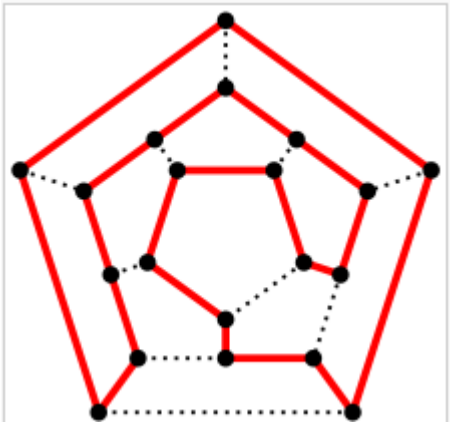
other:

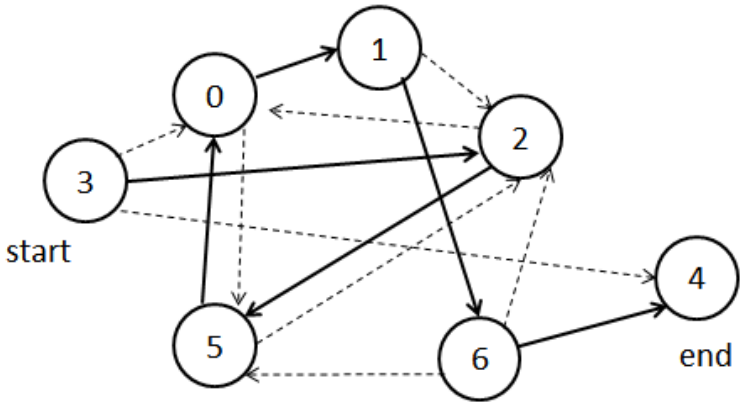
- not handling null or empty tree correctly, 1 point
- stopping early, 2 points (early return)

6. (Graphs - 12 points) Implement an instance method for a `Graph` class that determines if the `Graph` contains a *Hamiltonian Path*.

A Hamilton Path is a path through a graph where every vertex is visited **exactly** once.
Note: It is **not** necessary to visit every edge in the graph. The start and end vertices are different.

Consider the following Hamiltonian Paths. Used edges are solid and unused edges are dashed.





Recall the Graph class:

```
public class Graph {

    private Map<String, Vertex> vertices;

    private void clearAll() // calls reset on all Vertex objects in vertices

    private static class Vertex {

        private String name;
        private List<Edge> adjacent;
        private int scratch;
        private double distance;

        public void reset() {
            distance = INFINITY;
            prev = null;
            scratch = 0;
        }
    }

    private static class Edge {
        private Vertex dest;
        private double cost;
    }
```

You may use the Java `Map, List, and Iterator` methods (including for-each loops) in addition to the `Edge` and `Vertex` classes. Do not use any other methods from the `Graph` class or `Vertex` class unless you implement them yourself.

Do not use any auxiliary data structures other than iterators.

```
public boolean containsHamiltonianPath() {
     clearAll();
     for(String name : vertices.keySet())
          if(helper(name, 0))
               return true;
     return false;
}

private boolean helper(String currentVertex, int verticesVisited) {
     // complete this method

     Vertex current = vertices.get(currentVertex);

     // failure base case, already been here
     if(current.scratch == 1)
          return false;

     else {
          verticesVisited++;
          current.scrath = 1;
          // been to them all/
          if(verticesVisited == vertices.size())
               // return true;
          else {
               for(Edge e : current.adjacent)
                    if(helper(e.dest.name, verticesVisited)
                         return true;
          // undo changes in case another way exists
          current.scrath = 0;
          return false;
}
```

Common Problems:

- early return
- not undoing change
- clearing all (lose all past info)

Grading Criteria, 12 points

- success base case correct, 2 points
- failure base case, already visited, correct, 1 point
- increment vertices visited correctly, 1 point
- mark current vertex as visited, 1 point

- loop through current edges correctly, 2 points
- make recursive call, 2 points (-1 if use vertex instead of name)
- return true only if successful, 2 points
- unmark visited this node if failed all, 1 point
- return false after tried all options and failed, 1 point

Other:

- early return, - 5
- use of aux data structures, -2 to - 4
- no backtracking / GCE, -8

Question 1 Answer Sheet. Name_____ UTEID _____

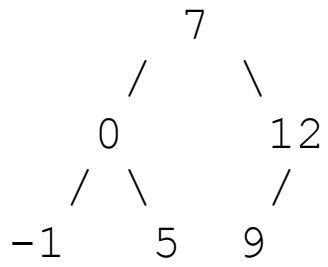
A. 63

B. $O(N^2)$

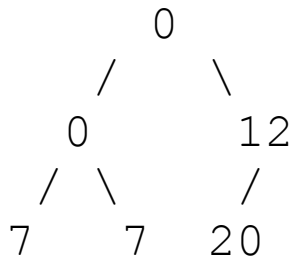
C. O(NlogN)

D. O(NlogN) or ERROR (any kind)

E. $O(N^2)$



F. _____



G. _____

H. O(M) NOT N O(N) = -1

I. O(NlogN)

J. Y P T X Z K Q

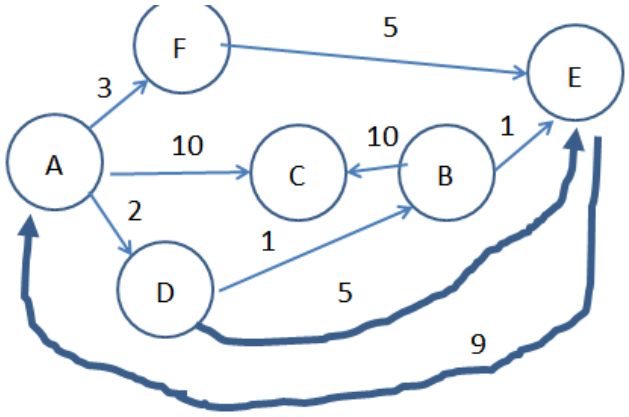
K. X Y T P Q Z K

L. Not a complete tree.

M. 5 bits

N. BELLE

O. Does not implement `Iterable`.



P. _____

Q. 4

R. NO

S. Insertion and Merge

T. 50,000,000,000 or 50 billion or $5x10^{10}$