Name:

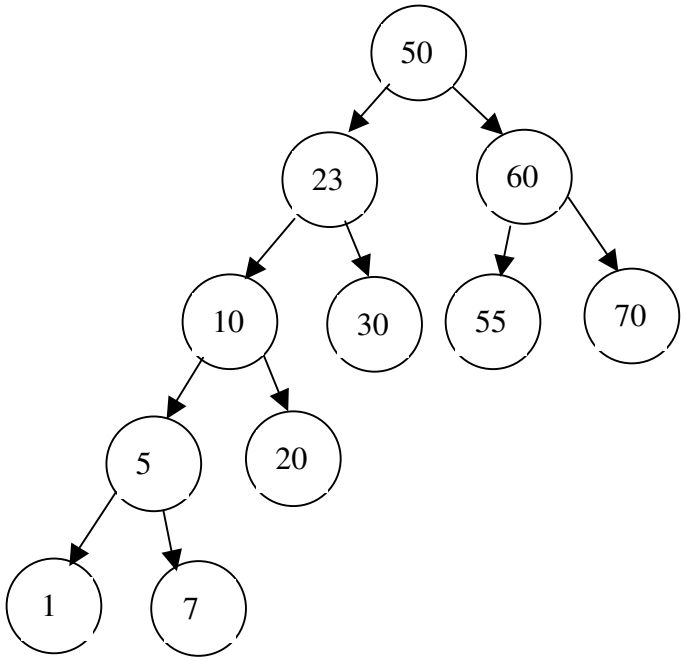
15-111 Test #3 (Sample - Revised)

1. Please describe the strategy behind the use of a hash table. In other words, "How do hash tables work?" Your answer should include a description of collision and the goal of collision resolution, but it should not describe the specific details of any particular collision resolution technique.

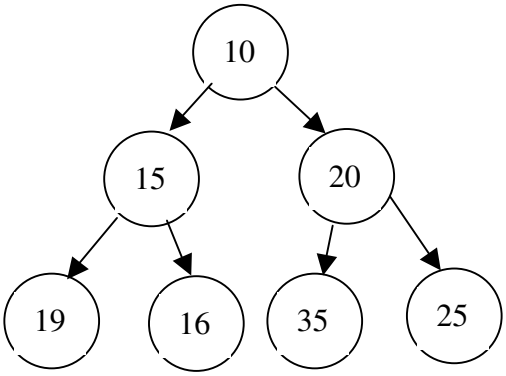
2. Please implement the following method for the BinarySearchTree class we used during lecture and in the assignment. If you would like to review the other methods, instance variables, &c, most of the rest of the class is attached.

```
/*
 * Recursive method which inserts a new element
 * into the tree. The tree must stay ordered after
 * the new element has been added
 */
private BinaryNode insert(BinaryNode root, Comparable data)
{
  /*
   * You're code here
   */
}
```

3. Please show the deletion of 23 from the BST below. Specifically, please draw figure(s) showing the tree after each step of the algorithm and label your figure(s) descriptively.

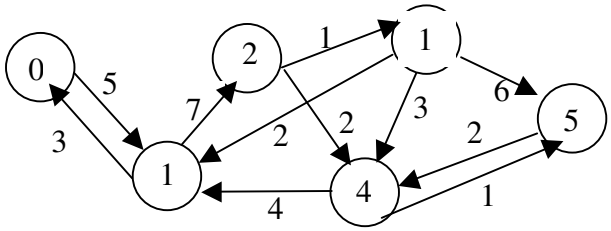


4. Please show the deleteMin() operation acting on the MinHeap below. Specifically, please draw figure(s) showing the tree after each step of the algorithm and label your figure(s) descriptively.



5. We discussed two different data structures used to represent graphs, the adjacency list and the adjacency matrix. Each representation has costs and benefits. Please explain the trade-offs between the two data structures and provide a comparatively good and bad application for each.

6. Please draw a picture that show the tree resulting from the Depth-First Search (DFS) rooted at Node-3 of the graph below.



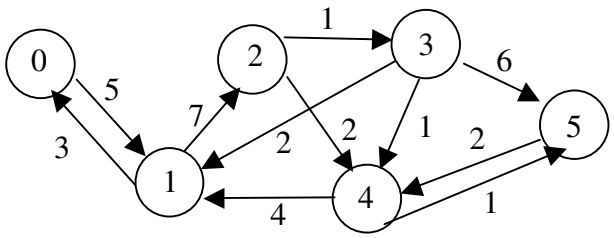
7. Please show the pseudo-code or Java-code for the DepthFirstTraversal() method. This method should simply output the number of each node as it is visited using a breadth-first traversal (like a breadth-first search).

Please write this method for an adjacency list representation of a graph. Please keep in mind the operations that an adjacency list is likely to support, as well as those that it is not likely to support. If you are uncomfortable estimating these operations, a skeleton of an AdjacencyList class is attached. It contains a sufficient set of operations to answer this question.

```
public static void DepthFirstTraversal (int NodeNumber);
```

8. In class we discussed Dijksta's algorithm. Please show the progression of Dijkstra's algorithm as it finds the shortest path from Node-2 to Node-5 within the graph below by doing two things.

a) Darken the edges of the graph that are included in the shortest path and number them in the order in which they become *known*.
b) Show the step-by-step progression of the algorithm by drawing the table used b1y the algorithm. You may redraw the table after each step, draw the table once and clearly mark the changes to the table for each step (in a way that we can understand them), or some combination of the two



9. In class we discussed Prim's algorithm for finding the *Minimum Spanning Tree.* It was extremely similar to Dijkstra's algorithm for finding the shortest path, but the algorithms do differ substantially at one critical step. How do the two algorithm differ? Why?

Please note: This question isn't asking for a defintion of a minimum spanning tree, nor is it asking for either algorithm in detail. Instead, it is specifically asking for and about the difference in the algorithms. If it helps, please assume that we know one algorithm or the other and explain to us how to adapt one algorithm to form the other and why we need to make the change.

```
/*
 * AdjacencyList public interface
 */

class AdjList
{
        private class Edge
        {
                public Edge(int vertex, int cost);
                public Edge(int vertex);
                public boolean equals(Edge other_edge);
        }

        public AdjList(int num_vertices);
        public addEdge(int vertex_from, int vertex_to, int cost);
        public removeEdge(int vertex_from, int vertex_to);
        public ListIterator getAdjacencies (int vertex)
        public Edge getEdge(int vertex_from, int vertex_to);
}
```

```
public class BinarySearchTree
{
  private class BinaryNode
  {
    private Comparable data;    // the data to be stored in the node
    private BinaryNode left;    // the left child of the current node
    private BinaryNode right;   // the right child of the current node

    public BinaryNode(Comparable data)
    {
      this.data = data;
      left = null;
      right = null;
    }

    public BinaryNode(Comparable data, BinaryNode left, BinaryNode right)
    {
        this.data = data;
        this.left = left;
        this.right = right;
      }

    private Comparable getData()
    {
      return data;
    }

    private BinaryNode getLeft()
    {
      return left;
    }

    private BinaryNode getRight()
    {
        return right;
      }

    private void setLeft(BinaryNode left)
    {
      this.left = left;
    }

    private void setRight(BinaryNode right)
    {
      this.right = right;
    }

    private boolean isLeaf()
    {
      return (left == null && right == null);
    }
  }

  private BinaryNode root;
```

```
public BinarySearchTree()
{
  // the tree will initially be empty, so set root to null
  root = null;
}


public void insert(Comparable data)
{
  if (null == data)
  {
    return;
  }

  root = insert(root, data);
}

/*
 * Recursive method which inserts a new element
 * into the tree. The tree must stay ordered after
 * the new element has been added
 */
private BinaryNode insert(BinaryNode root, Comparable data)
{
    /*
     * You're code here
     */
}

public Comparable find(Comparable data)
{
  if (null == data)
  {
    return null;
  }

  return find(root, data);
}

private Comparable find(BinaryNode root, Comparable data)
{
    /*
     * Not provided
     */
}

public void delete(Comparable data)
{
  if (null == data)
    return;

  root = delete(root, data);
}
```

```
private BinaryNode delete(BinaryNode root, Comparable data)
{
  if (null == root)
  {
    return null;
  }

  if (data.compareTo(root.getData()) == 0)
  {
    if (root.isLeaf())
    {
      return null;
    }

    if (root.getLeft() == null)
    {
      return root.getRight();
    }

    if (root.getRight() == null)
    {
      return root.getLeft();
    }

       Comparable replacementData = getRightmost(root.getLeft());

      return new BinaryNode(replacementData,
                  delete(root.getLeft(), replacementData),
                  root.getRight());
  }
    else if (data.compareTo(root.getData()) < 0)
  {
    root.setLeft(delete(root.getLeft(), data));
    return root;
  }
  else
  {
    root.setRight(delete(root.getRight(), data));
    return root;
  }
}

private Comparable getRightmost(BinaryNode root)
  {
    if (root == null)
      return null;

    if (root.getRight() == null)
    {
      return root.getData();
    }
    else
    {
      return getRightmost(root.getRight());
    }
}
```

```
  public void printInOrder()
  {
      if (null == root)
      {
        System.out.println("Tree is empty");
      }
      else
      {
        printInOrder(root);
      }
    }

  private void printInOrder(BinaryNode root)
  {
    if (null == root)
    {
      return;
    }

    printInOrder(root.getLeft());

    System.out.println(root.getData());

    printInOrder(root.getRight());
  }
}
```