1. (20 Points) Multiple Choice:

A. (2 Points) In a graph, all ______ begin and end at the same vertex and do not pass through any other vertices more than once.
   a. Paths
   b. Simple paths
   c. Cycles
   d. Simple cycles

B. (2 Points) A connected undirected graph that has n vertices must have at least ______ edges.
   a. n
   b. n – 1
   c. n / 2
   d. n * 2

C. (2 Points) The maximum height of a binary search tree of n nodes is ______.
   a. n
   b. n – 1
   c. n / 2
   d. $\log_2(n + 1)$

D. (2 Points) In an array based representation of a complete binary tree, which of the following represents the left child of node tree[i]?
   a. tree[i+2]
   b. tree[i–2]
   c. tree[2*i+1]
   d. tree[2*i+2]

E. (2 Points) In an array based representation of a complete binary tree, which of the following represents the parent of node tree[i]?
   a. tree[i–2]
   b. tree[(i–1)/2]
   c. tree[2*i–1]
   d. tree[2*i–2]

F. (2 Points) The quicksort is _____ in the worst case.
   a. $O(n^2)$
   b. $O(n^3)$
   c. $O(n * \log_2 n)$
   d. $O(\log_2 n)$

G. (2 Points) A method in a subclass is said to ______ an inherited method if it has the same method declarations as the inherited method.
   a. Copy
   b. Override
   c. Overload
   d. Cancel

H. (2 Points) The ______ access modifier hides the members of a class from the class's clients but makes them available to a subclass and to another class within the same package.
   a. public
   b. private
   c. protected
   d. package access

I. (2 Points) Which of the following code fragments is used to delete the item at the front of a queue represented by a circular array?

```
a. front=MAX_QUEUE - front;
       --count;
b. front=front - back;
       --count;
c. front=(front+1)%MAX_QUEUE;
       --count;
d. front=(back+1)% MAX_QUEUE;
       --count;
```

(Answer: **c**)

J. (2 Points) If the array: {6, 2, 7, 13, 5, 4}is added to a stack, in the order given, which number will be the first number to be removed from the stack?
   a. 6
   b. 2
   c. 5
   d. 4

2. (20 Points) The corrected QuickSort Class:

```
import java.util.Vector;
public class QuickSort {
      public static <T extends Comparable<? super T>> void quickSort(Vector<T> theVector,
                                                                      int first, int last) {
            if (first < last) {
                  int pivotIndex = partition(theVector, first, last);
                  quickSort(theVector, first, pivotIndex - 1);
                  quickSort(theVector, pivotIndex + 1, last);
            } // end if
      } // end quicksort

      public static <T extends Comparable<? super T>> void choosePivot(Vector<T> theVector,
                                                                        int first, int last) {

            // The pivot will be the middle value of first, mid and last
            int mid = (first + last) / 2;
            T temp = theVector.elementAt(first);
            T f = theVector.elementAt(first);
            T m = theVector.elementAt(mid);
            T l = theVector.elementAt(last);

            if (((f.compareTo(m) <= 0) && (l.compareTo(m) >= 0)) ||
                ((f.compareTo(m) >= 0) && (l.compareTo(m) <= 0))) {
                  theVector.set(first, theVector.elementAt(mid));
                  theVector.set(mid, temp);
            } else if (((f.compareTo(l) <= 0) && (m.compareTo(l) >= 0)) ||
                        ((f.compareTo(l) >= 0) && (m.compareTo(l) <= 0))) {
                  theVector.set(first, theVector.elementAt(last));
                  theVector.set(last, temp);
            }
      } // end choosePivot

      public static <T extends Comparable<? super T>> int partition(Vector<T> theVector,
                                                                     int first, int last) {
            T tempItem;
            choosePivot(theVector, first, last);
            T pivot = theVector.elementAt(first); // reference pivot
            int lastS1 = first; // index of last item in S1
            for (int firstUnknown = first + 1; firstUnknown <= last; ++firstUnknown) {
                  if (theVector.elementAt(firstUnknown).compareTo(pivot) < 0) {
                        ++lastS1;
                        tempItem = theVector.elementAt(firstUnknown);
                        theVector.set(firstUnknown, theVector.elementAt(lastS1));
                        theVector.set(lastS1, tempItem);
                  } // end if
            } // end for
            tempItem = theVector.elementAt(first);
            theVector.set(first, theVector.elementAt(lastS1));
            theVector.set(lastS1, tempItem);
            return lastS1;
      } // end partition
}
```

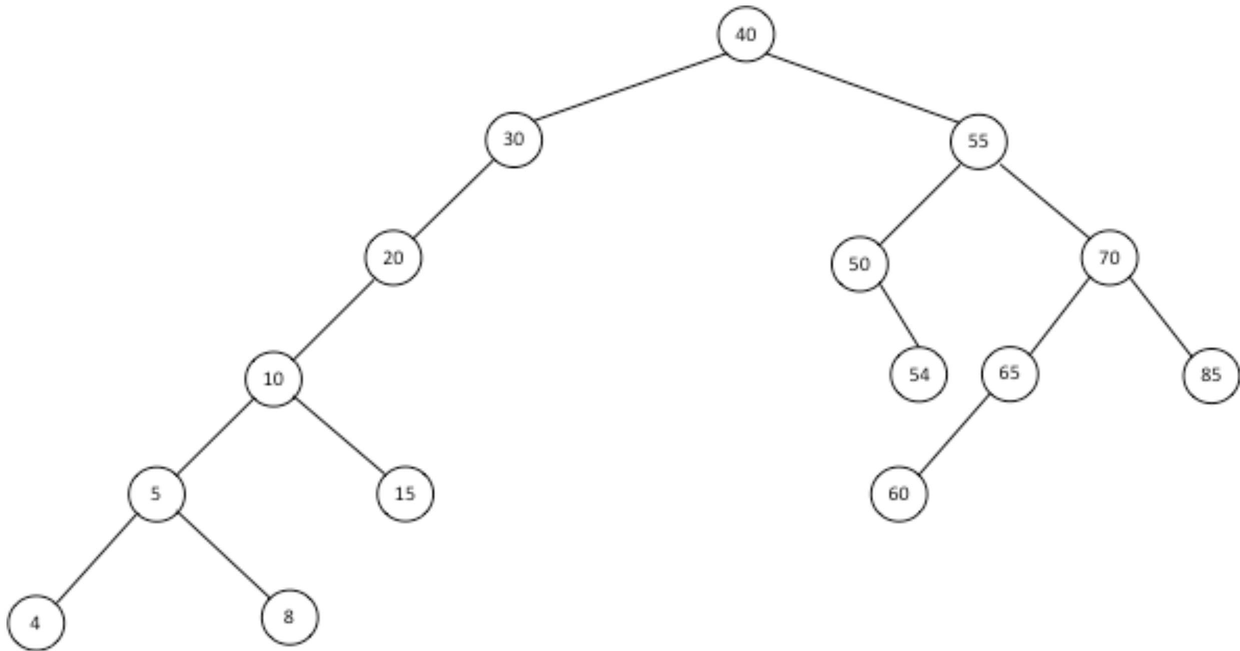
3. (50 Points) The correct BinarySearchTree implementation:

```
import java.util.Vector;

public class BinarySearchTree implements
BinarySearchTreeInterface {
   private TreeNode root = null;
   private Vector<TreeItem> v;

   @Override
   public boolean isEmpty() {
       return (root == null);
   }


   @Override
   public void makeEmpty() {
      root = null;
   }


   @Override
   public void insert(TreeItem item) {
      root = insertItem(root, item);
   }


   @Override
   public TreeItem retrieve(int key) {
       return retrieveItem(root, key);
   }


   @Override
   public Vector<TreeItem> inorder() {
      v = new Vector<TreeItem>();
      inorderTraversal(root);
       return v;
   }


   @Override
   public boolean equals(Object o) {
     boolean equals = true;
      BinarySearchTree other;
      Vector<TreeItem> v1, v2;

      if (o instanceof BinarySearchTree) {
         other = (BinarySearchTree) o;
         v1 = this.inorder();
         v2 = other.inorder();
         equals = v1.equals(v2);
      } else {
         equals = false;
      }

       return equals;
   }
```

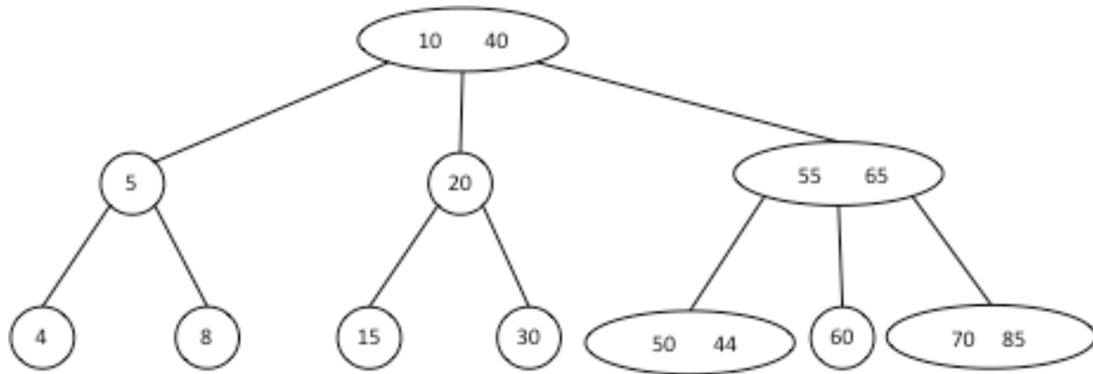
```
private TreeNode insertItem(TreeNode node, TreeItem newItem) {
      TreeNode newSubtree;

      if (node == null) {
         node = new TreeNode(newItem);
         return node;
      }

      if (newItem.compareTo(node.getItem()) < 0) {
         newSubtree = insertItem(node.getLeftChild(), newItem);
         node.setLeftChild(newSubtree);
         return node;
      } else {
         newSubtree = insertItem(node.getRightChild(), newItem);
         node.setRightChild(newSubtree);
         return node;
      }
   }



   private TreeItem retrieveItem(TreeNode node, int key) {
      TreeItem treeItem;
      TreeItem compareItem = new TreeItem(key);
      if (node == null) {
         treeItem = null;
      } else if (compareItem.compareTo(node.getItem()) == 0) {
         treeItem = node.getItem();
      } else if (compareItem.compareTo(node.getItem()) < 0) {
         treeItem = retrieveItem(node.getLeftChild(), key);
      } else {
         treeItem = retrieveItem(node.getRightChild(), key);
      }
      return treeItem;
   }


   private void inorderTraversal(TreeNode node) {
      if (node != null) {
         inorderTraversal(node.getLeftChild());
         v.add(node.getItem());
         inorderTraversal(node.getRightChild());
      }
   }
}
```

4. (20 Points) The following list of numbers: 40, 30, 55, 20, 10, 50, 70, 65, 5, 15, 4, 60, 85, 54, 8 inserted in the given order:

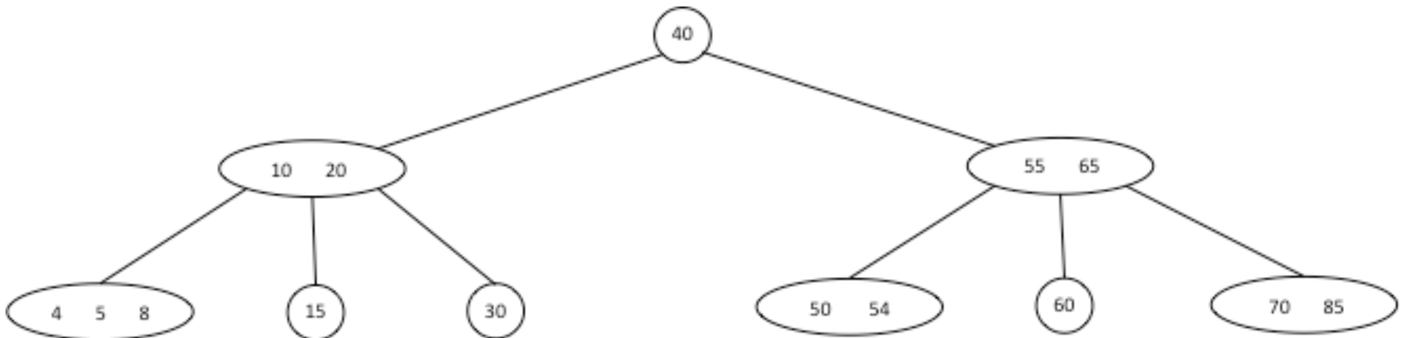
a. (5 Points) Into a Binary Search Tree produces the following tree:



b. (5 Points) Into a 2-3 Tree produces the following tree:



c. (5 Points) Into a 2-3-4 Tree produces the following tree:



d. (5 Points) The numbers in the following order produces a full Binary Search Tree:

40, 10, 5, 4, 8, 20, 15, 30, 60, 54, 50, 55, 70, 65, 85

1. (20 Points) Multiple Choice:

A. (2 Points) Which of the following is true about a simple cycle in a graph?
   a. It can pass through a vertex more than once
   b. It can not pass through a vertex more than once
   c. It begins at one vertex and ends at another vertex
   d. It passes through only one vertex

B. (2 Points) A tree with n nodes must contain ______ edges.
   a. n
   b. n – 1
   c. n / 2
   d. n * 2

C. (2 Points) Locating a particular item in a binary search tree of n nodes requires at most ______ comparisons.
   a. n
   b. n * 3
   c. n / 2
   d. n – (n / 2)

D. (2 Points) In an array based representation of a complete binary tree, which of the following represents the right child of node tree[i]?
   a. tree[i+2]
   b. tree[i–2]
   c. tree[2*i+1]
   d. tree[2*i+2]

E. (2 Points) In an array based representation of a complete binary tree, which of the following represents the parent of node tree[i]?
   a. tree[i–2]
   b. tree[(i–1)/2]
   c. tree[2*i–1]
   d. tree[2*i–2]

F. (2 Points) The quicksort is _____ in the worst case.
   e. $O(n^2)$
   f. $O(n^3)$
   g. $O(n * \log_2 n)$
   h. $O(\log_2 n)$

G. (2 Points) The keyword _______ is used in the class declaration of a subclass to indicate its superclass.
   a. inherits
   b. extends
   c. implements
   d. super

H. (2 Points) A class's _____ members can only be used by its own methods.
   a. public
   b. protected
   c. private
   d. package access

I. (2 Points) Which of the following is the code to insert a new node, referenced by `newNode`, into an empty queue represented by a circular linked list?

```
a. newNode.setNext(lastNode);
b. lastNode.setNext(lastNode);
   lastNode = newNode;
c. newNode.setNext(lastNode);
   newNode = lastNode;
d. newNode.setNext(newNode);
   lastNode = newNode;
```

(Answer: **d**)

J. (2 Points) If the array: {6, 21, 35, 3, 6, 2, 13} is added to a stack, in the order given, which of the following is the top of the stack?
   a. 2
   b. 6
   c. 35
   d. 13

2. (20 Points) The corrected MergeSort Class:

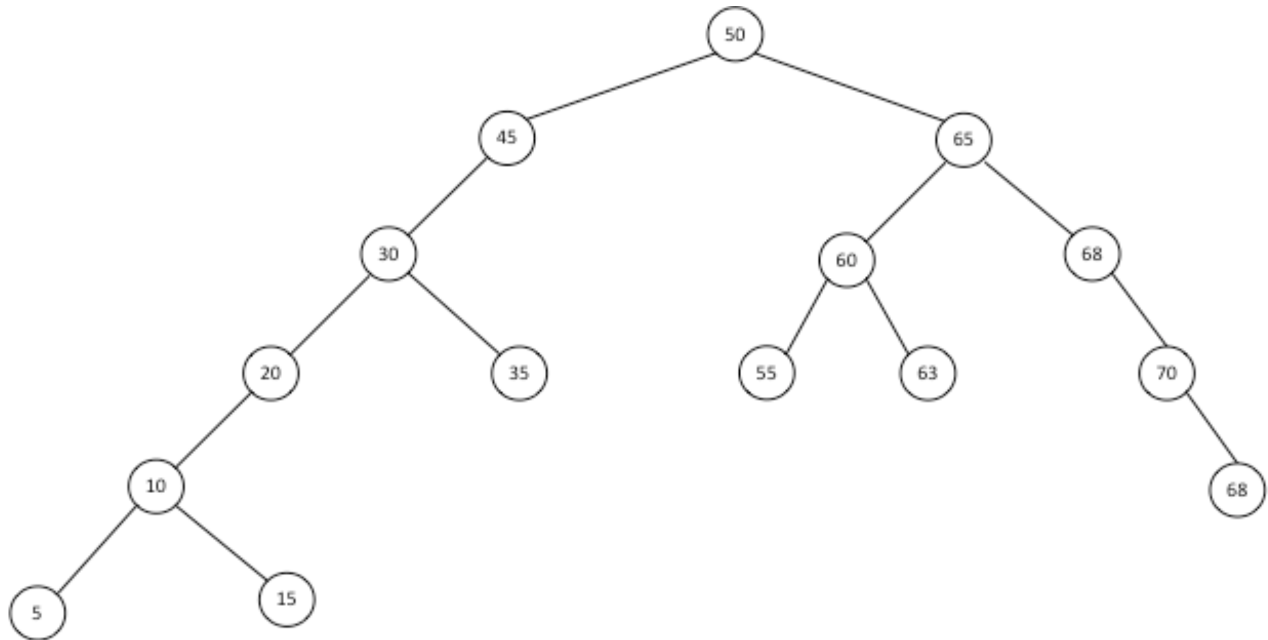
```
import java.util.Vector;
public class MergeSort {
     public static <T extends Comparable<? super T>> void sort(Vector<T> theVector ) {
        Vector<T> tempVector = new Vector<T>(theVector.size());
        for ( int i = 0 ; i < theVector.size() ; i++ ) {
            tempVector.add(null);
        }
        mergeSort(theVector, tempVector, 0, (theVector.size() - 1));
    }

    public static <T extends Comparable<? super T>> void mergeSort(Vector<T> theVector,
                                                                  Vector<T> tempVector,
                                                                  int first, int last) {
        if (first < last) {
            int mid = (first + last) / 2; // index of midpoint
            mergeSort(theVector, tempVector, first, mid);
            mergeSort(theVector, tempVector, mid + 1, last);
            merge(theVector, tempVector, first, mid, last);
        } // end if
    }

    public static <T extends Comparable<? super T>> void merge(Vector<T> theVector,
                                                              Vector<T> tempVector,
                                                              int first, int mid, int last) {
        int first1 = first;
        int last1 = mid;
        int first2 = mid + 1;
        int last2 = last;
        int index = first1;
        while ((first1 <= last1) && (first2 <= last2)) {
            if (theVector.elementAt(first1).compareTo(theVector.elementAt(first2)) < 0) {
                tempVector.set(index, theVector.elementAt(first1));
                first1++;
            } else {
                tempVector.set(index, theVector.elementAt(first2));
                first2++;
            } // end if
            index++;
        } // end while
        while (first1 <= last1) {
            tempVector.set(index, theVector.elementAt(first1));
            first1++;
            index++;
        } // end while
        while (first2 <= last2) {
            tempVector.set(index, theVector.elementAt(first2));
            first2++;
            index++;
        } // end while

        for (index = first; index <= last; ++index) {
            theVector.set(index, tempVector.elementAt(index));
        } // end for
    } // end merge
}
```

3. (50 Points) The correct BinarySearchTree implementation:

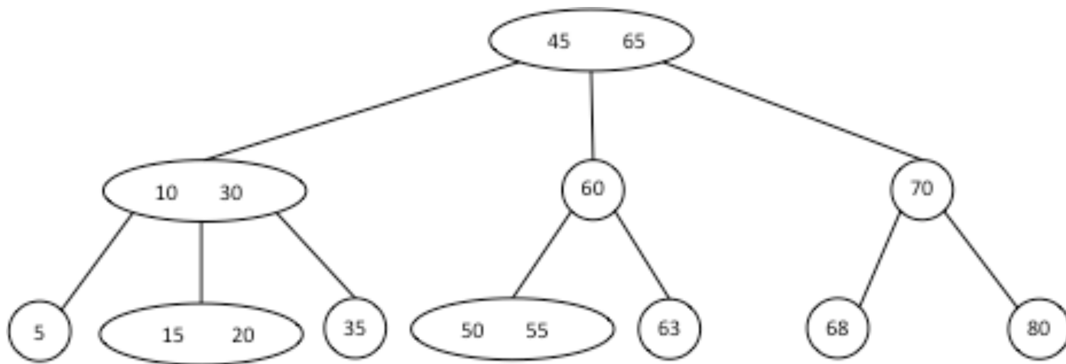
```
import java.util.Vector;

public class BinarySearchTree implements
BinarySearchTreeInterface {
   private TreeNode root = null;
   private Vector<TreeItem> v;

   @Override
   public boolean isEmpty() {
      return (root == null);
   }

   @Override
   public void makeEmpty() {
      root = null;
   }

   @Override
   public void insert(TreeItem item) {
      root = insertItem(root, item);
   }

   @Override
   public TreeItem retrieve(int key) {
      return retrieveItem(root, key);
   }

   @Override
   public Vector<TreeItem> inorder() {
      v = new Vector<TreeItem>();
      inorderTraversal(root);
      return v;
   }

   @Override
   public boolean equals(Object o) {
      boolean equals = true;
      BinarySearchTree other;
      Vector<TreeItem> v1, v2;

      if (o instanceof BinarySearchTree) {
         other = (BinarySearchTree) o;
         v1 = this.inorder();
         v2 = other.inorder();
         equals = v1.equals(v2);
      } else {
         equals = false;
      }

      return equals;
   }

private TreeNode insertItem(TreeNode node, TreeItem newItem) {
      TreeNode newSubtree;

      if (node == null) {
         node = new TreeNode(newItem);
         return node;
      }

      if (newItem.compareTo(node.getItem()) < 0) {
         newSubtree = insertItem(node.getLeftChild(), newItem);
         node.setLeftChild(newSubtree);
         return node;
      } else {
         newSubtree = insertItem(node.getRightChild(), newItem);
         node.setRightChild(newSubtree);
         return node;
      }
   }

   private TreeItem retrieveItem(TreeNode node, int key) {
      TreeItem treeItem;
      TreeItem compareItem = new TreeItem(key);
      if (node == null) {
         treeItem = null;
      } else if (compareItem.compareTo(node.getItem()) == 0) {
         treeItem = node.getItem();
      } else if (compareItem.compareTo(node.getItem()) < 0) {
         treeItem = retrieveItem(node.getLeftChild(), key);
      } else {
         treeItem = retrieveItem(node.getRightChild(), key);
      }
      return treeItem;
   }

   private void inorderTraversal(TreeNode node) {
      if (node != null) {
         inorderTraversal(node.getLeftChild());
         v.add(node.getItem());
         inorderTraversal(node.getRightChild());
      }
   }
}
```

4. (20 Points) The following list of numbers: 50, 45, 30, 65, 60, 35, 20, 55, 63, 10, 5, 68, 70, 80, 15 inserted in the given order:

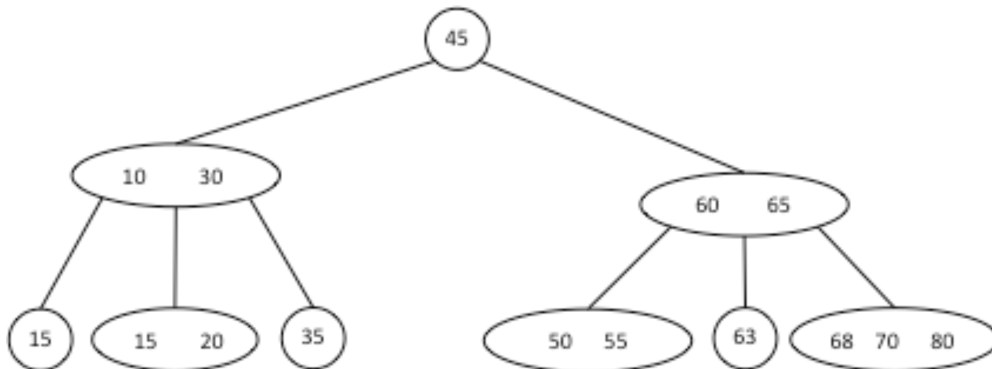
e. (5 Points) Into a Binary Search Tree produces the following tree:



f. (5 Points) Into a 2-3 Tree produces the following tree:



g. (5 Points) Into a 2-3-4 Tree produces the following tree:



h. (5 Points) The numbers in the following order produces a full Binary Search Tree:

50, 20, 10, 5, 15, 35, 30, 45, 65, 60, 55, 63, 70, 68, 85