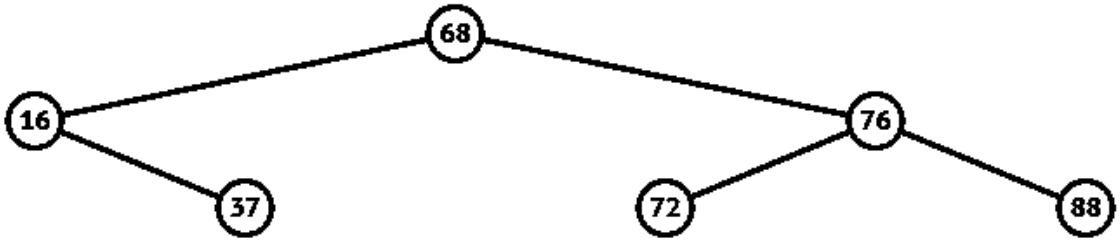
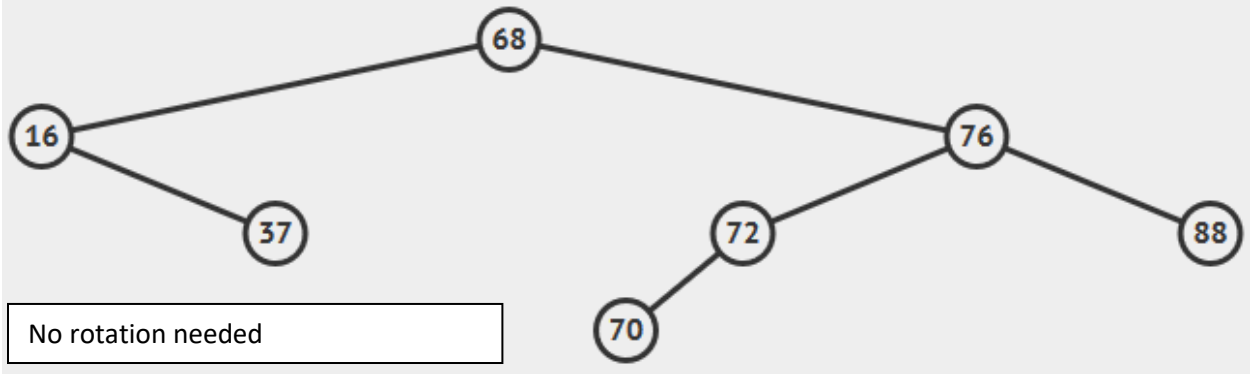
**Name**: _____

This is a **50-minute** exam. Notes, calculators, or electronic devices are not allowed. Please put away all phones and smart watches. Follow code style writing guidelines for the course (excluding requirements for code documentation). The back of each page may be used if you run out of room. Code will be partially graded on efficiency.

1. (25 pts) One of the problems with binary trees is that they can become unbalanced or skewed right or left. This reduces the efficiency of the data structure. To overcome this, we can use **AVL** trees. AVL trees maintain a balanced data structure through rotating nodes left or right when a node is inserted.

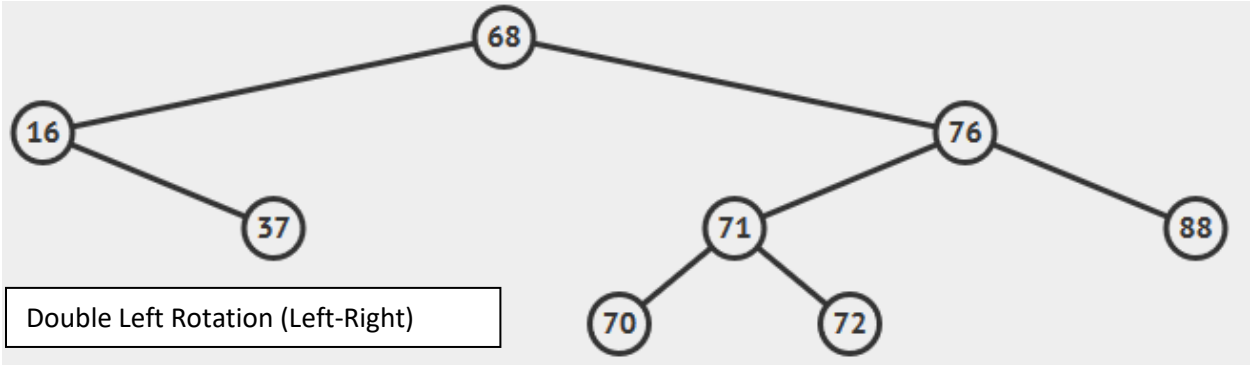



**a.** (5 pts) Using the AVL tree above, draw the result of inserting the number **70**



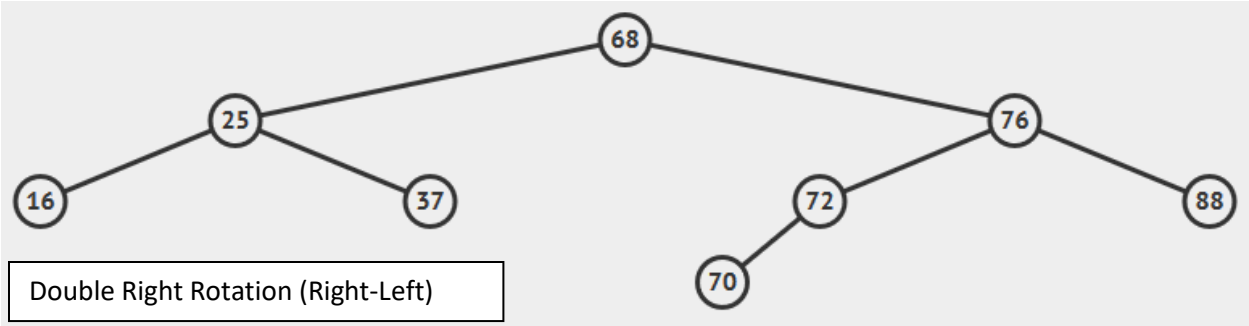

No rotation needed

**b.** **(**10 pts) Using the tree drawn in ***part A,*** draw the result of inserting **71**




Double Left Rotation (Left-Right)

c. **(**10 pts) Using the tree you drew for ***part A*** draw the result of inserting **25**




Double Right Rotation (Right-Left)

2. (40 pts) The method below takes a **double linked list** of integers. The DoubleLinkedListCell class is defined on the last page. You may assume that the linked list of numbers contains at least one number and is front of the linked list. Complete the method below to reverse alternate **k** cells in the linked list. This should work for any linked list of size n > 0.

   For example, if k=3 and the linked list contained 9 cells: 1 – 2 – 3 – 4 – 5 – 6 – 7 – 8 – 9, the resulting linked list should be: 3 – 2 – 1 – 4 – 5 – 6 – 9 – 8 – 7.

```
public void Reverse(DoubleLinkedListCell <int> list, int k){
     Queue<int> q = new Queue<int>();
     DoubleLinkedListCell<int> current = list;
     DoubleLinkedListCell<int> temp = new DoubleLinkedListCell<int>();
     bool end = false;

     while (current != null)
     {
         for (int i = 0; i < k; i++)
         {
             q.Enqueue(current.Data);
             if(current.Next != null)
                 current = current.Next;
             else
             {
                 end = true;
                 break;
             }
         }

         if(!end)
             current = current.Prev;
         while (q.Count > 0)
         {
             current.Data = q.Dequeue();
             if(current.Prev !=null)
                 current = current.Prev;
         }


         for (int i = 0; i < k*2 && current != null; i++)
         {
             current = current.Next;
         }

     }
}
```

3. (35 pts) Complete the method below that checks whether or not the given **binary search tree** is valid. The method should return **true** if the tree does not break any rules of a binary search tree or **false** otherwise. You may assume that the tree is non-null.

```
bool isBST(BinaryTreeNode<int> root)
 {
       //Note that this is a naive method, we should also check that all right
       //tree data is greater
     //than the largest value seen so far, and the opposite for the left tree data

     if (root == null)
         return true;

     if (root.LeftChild != null && root.Data < root.LeftChild.Data)
         return false;
     if (root.RightChild != null && root.Data > root.RightChild.Data)
         return false;

     return isBST(root.LeftChild) && isBST(root.RightChild);
 }
```

***This page is only a reference page and may be torn off the test.***

The **DoubleLinkedListCell<T>** class contains the following public members:

- **Data**: a property of type T that gets or sets the data stored in the cell.
- **Prev**: a property of type LinkedListCell<T> that gets or sets the next cell in the list.
- **Next**: a property of type LinkedListCell<T> that gets or sets the next cell in the list.
- A constructor that takes no parameters and constructs a cell whose **Data** property is the default value for type **T** and whose **Next** property is **null**.

The **BinaryTreeNode<T>** class is an immutable type containing the following public members:

- **Data**: a property of type T that gets the data stored in the node.
- **LeftChild**: a property of type **BinaryTreeNode<T>** that gets the left child of this node.
- **RightChild**: a property of type **BinaryTreeNode<T>** that gets the right child of this node.
- A constructor that takes a data item of type **T**, a left child of type **BinaryTreeNode<T>**, and a right child of type **BinaryTreeNode<T>**, and constructs a node with these components.

The starting tree for Question #1 for reference