

Trent University COIS 2020

Test Case for Assignment 3

Date: December 6 2023

Version 1.0

Teammates

David Chan, 0767384
Mohammad Abdur Rakib, 0685509
Chengjun Yin, 0695866

1. Executive Summary:

The project aims to deepen understanding and implementation skills of a self-adjusting binary tree known as the splay tree. The tree adjusts itself after each operation through a process called splaying without storing and additional information per node. The project is splitted into three parts:

1. To comprehend the principles behind self-adjusting binary search trees. Specifically splay trees.
2. To use a Slack class to track access path within the tree during various operations.
3. To implement and test deep coping and equality checking of splay trees using recursion.

2. Introduction:

The project introduces the properties of splay trees, where the last accessed node is moved to the root through a series of rotations during operations like Insert, Remove, or Contains. This procedure helps in ensuring that frequently accessed elements can be reached quickly.

Part A: Re-implementation

Access Method: A non-recursive implementation of the Access method that retrieves the nodes along the access path in reverse order is done using a Stack. The method is central to tracing the path and is utilized for splaying operations.

Splay Method: The Splay method is re-implemented to perform rotations. Unlike the classic method where the first rotation happens early on, in this assignment, the splay method requires the first rotation to be the last rotation.

Part B: Deep Copy:

Clone Method: A deep copy of the splay tree is created using a Clone method, which constructs a replica from the top down following a preorder traversal.

Equals Method: To verify the accuracy of cloning, an override for the Equals method is implemented to access if two trees are exact copies.

Part C: Undo

Undo Method: The Undo method is devised to reverse the insertion process by relocating the root node inserted most recently to a left position through reverse rotations and then removing it. This method relies on trial and error using different rotations, leveraging the Clone method to reset to the prior state is necessary.

3. Source code:

SplayTree.cs:

```
using System;
using COIS2020H_Assignment3_DavidMohammadChengCRY;

namespace SplayTree
{
    //-----

    public interface ISearchable<T>
    {
        void Insert(T item);
        bool Remove(T item);
        bool Contains(T item);
    }

    class SplayTree<T> : ISearchable<T> where T : IComparable
    {
        private Node<T> root;           // Reference to the root of a splay tree
        public Node<T> Root
        {
            get
            {
                return root;
            }
        }

        // Constructor
        // Initializes an empty splay tree
        // Time complexity: O(1)

        public SplayTree()
        {
            root = null;                // Empty splay tree
        }

        // Returns the nodes in reverse order along the access path from the root to the last node
        // accessed.
        // In the case of a successful insertion, the last node contains the inserted item.
        private Stack<Node<T>> Access(T item)
        {
            // stack for storing nodes in reverse order along the access path
            Stack<Node<T>> accessPath = new Stack<Node<T>>();
        }
    }
}
```

```
Node<T> temp = root;
// traverse until item is found or a leaf node is reached
while (temp != null)
{
    accessPath.Push(temp);
    // get temp node right child
    if (item.CompareTo(temp.Item) > 0)
    {
        temp = temp.Right;
    }
    // get temp node left child
    else if (item.CompareTo(temp.Item) < 0)
    {
        temp = temp.Left;
    }
    // item = temp.Item
    else
    {
        break;
    }
}
return accessPath;
}

// RightRotate
// Rotates the splay tree to the right around Node p
// Returns the new root of the (sub)tree
// Worst case time complexity: O(1)

private Node<T> RightRotate(Node<T> p)
{
    Node<T> q = p.Left;

    p.Left = q.Right;
    q.Right = p;

    return q;
}

// RotateLeft
// Rotates the splay tree to the left around Node p
// Returns the new root of the (sub)tree
// Worst case time complexity: O(1)

private Node<T> LeftRotate(Node<T> p)
{
    Node<T> q = p.Right;
```

```
p.Right = q.Left;
q.Left = p;

return q;
}

// splay the last node in accessPath to the root
private void Splay(T item, Stack<Node<T>> accessPath)
{
    // if the number of elements in accessPath is less than 2, no rotation is needed
    if (accessPath.Count < 2)
    {
        return;
    }

    // get last accessedNode, which should be a constant
    Node<T> lastAccessedNode = accessPath.Peek();

    // handle last accessed item at odd depth
    if ((accessPath.Count - 1) % 2 == 1)
    {
        lastAccessedNode = accessPath.Pop();
        Node<T> parentNode = accessPath.Pop();
        if (lastAccessedNode.Equals(parentNode.Left))
        {
            parentNode.Left = lastAccessedNode.Right;
            lastAccessedNode.Right = parentNode;
        }
        else if (lastAccessedNode.Equals(parentNode.Right))
        {
            parentNode.Right = lastAccessedNode.Left;
            lastAccessedNode.Left = parentNode;
        }
        else
        {
            throw new Exception("ERROR: unable to find parent of last accessed node.");
        }
    }

    if (accessPath.Count > 0)
    {
        Node<T> grandParentNode = accessPath.Peek();
        if (parentNode.Equals(grandParentNode.Left))
        {
            grandParentNode.Left = lastAccessedNode;
        }
        else if (parentNode.Equals(grandParentNode.Right))
        {
            grandParentNode.Right = lastAccessedNode;
        }
    }
}
```

```
        else
        {
            throw new Exception("ERROR: unable to find parent of the parentNode");
        }
    }
    else
    {
        root = lastAccessedNode;
    }
}
else
{
    accessPath.Pop();
}

// now the count of accessPath must be even, rotate the last accessed node two levels up
while (accessPath.Count > 0)
{
    // get the next two nodes to be involved in the rotation
    Node<T> parentNode = accessPath.Pop();
    Node<T> grandParentNode = accessPath.Pop();

    if (parentNode.Equals(grandParentNode.Left))
    {
        // do right right rotation
        if (lastAccessedNode.Equals(parentNode.Left))
        {
            RightRotate(grandParentNode);
            RightRotate(parentNode);
        }
        // do left right rotation
        else if (lastAccessedNode.Equals(parentNode.Right))
        {
            LeftRotate(parentNode);
            grandParentNode.Left = lastAccessedNode;
            RightRotate(grandParentNode);
        }
        else
        {
            throw new Exception("ERROR: unable to find parent of last accessed node.");
        }
    }
    else if (parentNode.Equals(grandParentNode.Right))
    {
        // do right left rotation
        if (lastAccessedNode.Equals(parentNode.Left))
        {

```

```
        RightRotate(parentNode);
        grandParentNode.Right = lastAccessedNode;
        LeftRotate(grandParentNode);
    }
    // do left left rotation
    else if (lastAccessedNode.Equals(parentNode.Right))
    {
        LeftRotate(grandParentNode);
        LeftRotate(parentNode);
    }
    else
    {
        throw new Exception("ERROR: unable to find parent of last accessed node.");
    }
}
else
{
    throw new Exception("ERROR: unable to find parent of the parent node.");
}

// access lastAccessedNode to grandGrandParentNode's child
if (accessPath.Count > 0)
{
    Node<T> grandGrandParentNode = accessPath.Peek();
    if (grandParentNode.Equals(grandGrandParentNode.Left))
    {
        grandGrandParentNode.Left = lastAccessedNode;
    }
    else if (grandParentNode.Equals(grandGrandParentNode.Right))
    {
        grandGrandParentNode.Right = lastAccessedNode;
    }
    else
    {
        throw new Exception("ERROR: unable to find parent of the grandParentNode");
    }
}

root = lastAccessedNode;
}
}

// Inserts an item into a splay tree
public void Insert(T item)
{
    Node<T> insertNode = new Node<T>(item);
    copy0 = ((SplayTree<T>)Clone()).root;
    // directly set item as root if root is null
    if (root == null)
```



```
{
    root = insertNode;
    return;
}

// get access path to item
Stack<Node<T>> accessPath = Access(item);

// splay the last accessed item to the root
Splay(item, accessPath);

// old root becomes insertNode right child, left child of the old root becomes
insertNode left child
if (root.Item.CompareTo(item) > 0)
{
    insertNode.Left = root.Left;
    insertNode.Right = root;
    root.Left = null;
    root = insertNode;
    //Console.WriteLine("Insert success");
}
// old root becomes insertNode left child, right child of the old root becomes
insertNode right child
else if (root.Item.CompareTo(item) < 0)
{
    insertNode.Right = root.Right;
    insertNode.Left = root;
    root.Right = null;
    root = insertNode;
    //Console.WriteLine("Insert success");
}
}

// Remove an item from a splay tree
public bool Remove(T item)
{
    // get access path to item
    Stack<Node<T>> accessPath = Access(item);

    // splay the last accessed node to the root
    Splay(item, accessPath);

    // remove success
    if (root.Item.CompareTo(item) == 0)
    {
        // left subtree of the root
        SplayTree<T> cloneTree = (SplayTree<T>)Clone();
        cloneTree.root = cloneTree.root.Left;
        Node<T> largestNode = cloneTree.root;
```

```
// find largest of the left subtree
while (largestNode.Right != null)
{
    largestNode = largestNode.Right;
}

// splay the largest left subtree node to its root
Stack<Node<T>> accessPathToLeftSubtreeLargestNode =
cloneTree.Access(largestNode.Item);
cloneTree.Splay(accessPathToLeftSubtreeLargestNode.Peek().Item,
accessPathToLeftSubtreeLargestNode);

// make the left subtree as the root and assign the old root's right child as the new
root's right child
cloneTree.root.Right = root.Right;
root = cloneTree.root;

Console.WriteLine("Remove success");
return true;
}
else
{
    Console.WriteLine("Remove fail, item not presented in the tree");
    return false;
}
}

// Public Contains
// Returns true if the item is found in the root; false otherwise
public bool Contains(T item)
{
    if (root == null) // Empty splay tree
        return false;
    else
    {
        // look for the path to the item
        Stack<Node<T>> accessPath = Access(item);
        // splay the item to the root
        Splay(item, accessPath);

        // return true if the root of the splay tree is item, false otherwise
        return item.CompareTo(root.Item) == 0;
    }
}

// Returns a deep copy of the current splay tree
public object Clone()
{

```

```
SplayTree<T> cloneTree = new SplayTree<T>();
cloneTree.root = (Node<T>)CloneTree(root);
return cloneTree;
}

// recursive function to create a new copy of every node of root
// use preorder to traverse and copy the root
private object CloneTree(Node<T> node)
{
    if (node == null)
    {
        return null;
    }
    // preorder traversal
    Node<T> temp = new Node<T>(node.Item);
    temp.Left = (Node<T>)CloneTree(node.Left);
    temp.Right = (Node<T>)CloneTree(node.Right);
    return temp;
}

// check is t the same as root
// expect t is in type splayTree
public override bool Equals(Object t)
{
    try
    {
        // will throw Exception if t is not in type SplayTree<T>
        t = (SplayTree<T>)t;
    }
    catch (Exception)
    {
        Console.WriteLine("object is not a node");
        return false;
    }
    // true if equal
    return EqualsTree(root, ((SplayTree<T>)t).root);
}

// recursive function to is every node of root and copy the same
// use preorder to traverse and check
private bool EqualsTree(Node<T> root, Node<T> copy)
{
    if ((root == null && copy == null))
    {
        return true;
    }
    else if (root == null || copy == null)
    {
        return false;
    }
}
```

```
}
return root.Item.Equals(copy.Item) && EqualsTree(root.Left, copy.Left) &&
EqualsTree(root.Right, copy.Right);
}

//Undo insertion
//Assumption: last operation is a successful insert
public SplayTree<T> Undo()
{
    // make copy of current Splay tree
    SplayTree<T> copy = (SplayTree<T>)Clone();

    // Assume root.Right is the original last accessed node
    SplayTree<T> answer1;
    try
    {
        SplayTree<T> copy1 = (SplayTree<T>)copy.Clone();

        Node<T> copy1LastAccessedNode = copy1.root.Left;
        if (copy1LastAccessedNode == null)
        {
            throw new Exception("root has no left child");
        }
        //turn the root back to leaf
        copy1LastAccessedNode.Right = copy1.root.Right;
        copy1.root = copy1LastAccessedNode;
        if (copy1.Size() <= 1) return copy1;
        SplayTree<T> copy1Copy = (SplayTree<T>)copy1.Clone();

        answer1 = UndoTree(copy1, copy.root.Left.Item, copy);
    }
    catch (Exception e)
    {
        //Console.WriteLine(e.Message);
        answer1 = null;
    }

    // Assume root.Left is the original last accessed node
    SplayTree<T> answer2;
    try
    {
        SplayTree<T> copy2 = (SplayTree<T>)Clone();
        Node<T> copy2LastAccessedNode = copy2.root.Right;
        if (copy2LastAccessedNode == null)
        {
            throw new Exception("root has no right child");
        }
        copy2LastAccessedNode.Left = copy2.root.Left;
        copy2.root = copy2LastAccessedNode;
    }
```

```

        if (copy2.Size() <= 1) return copy2;
        SplayTree<T> copy2Copy = (SplayTree<T>)copy2.Clone();

        answer2 = UndoTree(copy2, copy.root.Right.Item, copy);
    }
    catch (Exception e)
    {
        answer2 = null;
    }

    // return answer
    if (answer1 != null) return answer1;
    if (answer2 != null) return answer2;

    // throw Exception when unable to find any answer
    throw new Exception("Undo failed");
}

// four condition: rr, ll, rl, lr
// last two condition: r, l
public SplayTree<T> UndoTree(SplayTree<T> o, T item, SplayTree<T> originalTree)
{
    SplayTree<T> copyO = (SplayTree<T>)o.Clone();

    //start reverses rotation
    SplayTree<T> r1 = UndoRightRight(copyO, item, originalTree);
    SplayTree<T> r2 = UndoLeftLeft(copyO, item, originalTree);
    SplayTree<T> r3 = UndoRightLeft(copyO, item, originalTree);
    SplayTree<T> r4 = UndoLeftRight(copyO, item, originalTree);
    SplayTree<T> r5 = UndoLeft(copyO, item, originalTree);
    SplayTree<T> r6 = UndoRight(copyO, item, originalTree);
    List<SplayTree<T>> candidate = new List<SplayTree<T>>();
    SplayTree<T> mock = new SplayTree<T>();
    mock.root = copyO;
    candidate.Add(mock);
    if (r1 != null) candidate.Add(r1);
    if (r2 != null) candidate.Add(r2);
    if (r3 != null) candidate.Add(r3);
    if (r4 != null) candidate.Add(r4);
    if (r5 != null) candidate.Add(r5);
    if (r6 != null) candidate.Add(r6);
    if (candidate.Count > 0)
    {
        return candidate[0];
    }

    return null;
}

```

```
public SplayTree<T> CheckLeafNodeAndEqual(SplayTree<T> copyO, T item,
SplayTree<T> originalTree)
{
    SplayTree<T> copy1 = (SplayTree<T>)copyO.Clone();
    copy1.Insert(originalTree.root.Item);
    if (copy1.Equals(originalTree))
    {
        List<SplayTree<T>> candidate = new List<SplayTree<T>>();
        SplayTree<T> mock = new SplayTree<T>();
        mock.root = copyO;
        return mock;
    }
    // wrong attempt
    return null;
}

public Node<T> traverseToItemParent(SplayTree<T> copy, T item)
{
    //SplayTree<T> copy = (SplayTree<T>)o.Clone();
    Node<T> temp = copy.root;
    while (true)
    {
        // when item is root
        if (temp.Item.Equals(item))
        {
            break;
        }
        if (item.CompareTo(temp.Item) > 0)
        {
            if (item.Equals(temp.Right.Item))
            {
                break;
            }
            temp = temp.Right;
        }
        else if (item.CompareTo(temp.Item) < 0)
        {
            if (item.Equals(temp.Left.Item))
            {
                break;
            }
            temp = temp.Left;
        }
        else
        {
            throw new Exception("PPP");
        }
    }
    return temp;
}
```

```
}

public SplayTree<T> UndoRightRight(SplayTree<T> o, T item, SplayTree<T>
originalTree)
{
    SplayTree<T> copyO = (SplayTree<T>)o.Clone();
    // traverse to parent of the target item
    Node<T> temp = traverseToItemParent(copyO, item);

    // reverse rr rotation
    try
    {
        Node<T> itemNode;

        // reverse rr
        if (item.CompareTo(temp.Item) > 0)
        {
            itemNode = temp.Right;
        }
        else if (item.CompareTo(temp.Item) < 0)
        {
            itemNode = temp.Left;
        }
        else
        {
            itemNode = temp;
        };
        if (!(itemNode.Right != null && itemNode.Right.Right != null)) return null;

        Node<T> temp2 = itemNode.Right.Left;
        Node<T> temp3 = itemNode.Right.Right.Left;
        Node<T> itemNode2 = itemNode.Right;
        Node<T> itemNode3 = itemNode.Right.Right;
        itemNode2.Left = itemNode;
        itemNode.Right = temp2;
        itemNode2.Right = temp3;
        itemNode3.Left = itemNode2;

        if (item.CompareTo(temp.Item) > 0)
        {
            temp.Right = itemNode3;
        }
        else if (item.CompareTo(temp.Item) < 0)
        {
            temp.Left = itemNode3;
        }
        else
        {
            copyO.root = itemNode3;
        }
    }
}
```

```
};

// reached leaf node
if (itemNode.Left == null && itemNode.Right == null)
{
    return CheckLeafNodeAndEqual(copyO, item, originalTree);
}
}
catch (Exception e)
{
    //Console.WriteLine(e.Message);
    return null;
}

// check
SplayTree<T> copy2 = (SplayTree<T>)copyO.Clone();
copy2.Insert(item);
if (copy2.Equals(originalTree))
{
    //continue reverses rotation
    SplayTree<T> r1 = UndoRightRight(copyO, item, originalTree);
    SplayTree<T> r2 = UndoLeftLeft(copyO, item, originalTree);
    SplayTree<T> r3 = UndoRightLeft(copyO, item, originalTree);
    SplayTree<T> r4 = UndoLeftRight(copyO, item, originalTree);
    SplayTree<T> r5 = UndoLeft(copyO, item, originalTree);
    SplayTree<T> r6 = UndoRight(copyO, item, originalTree);
    List<SplayTree<T>> candidate = new List<SplayTree<T>>();
    SplayTree<T> mock = new SplayTree<T>();
    mock.root = copyO;
    candidate.Add(mock);
    if (r1 != null) candidate.Add(r1);
    if (r2 != null) candidate.Add(r2);
    if (r3 != null) candidate.Add(r3);
    if (r4 != null) candidate.Add(r4);
    if (r5 != null) candidate.Add(r5);
    if (r6 != null) candidate.Add(r6);

    if (candidate.Count > 0)
    {
        return candidate[0];
    }
}
else
{
    //wrong attempt
    return null;
}
return null;
}
```



```
public SplayTree<T> UndoLeftLeft(SplayTree<T> o, T item, SplayTree<T> originalTree)
{
    SplayTree<T> copyO = (SplayTree<T>)o.Clone();
    // traverse to parent of the target item
    Node<T> temp = traverseToItemParent(copyO, item);

    // reverse ll rotation
    try
    {
        Node<T> itemNode;

        // reverse ll
        if (item.CompareTo(temp.Item) > 0)
        {
            itemNode = temp.Right;
        }
        else if (item.CompareTo(temp.Item) < 0)
        {
            itemNode = temp.Left;
        }
        else
        {
            itemNode = temp;
        }
    }

    if (!(itemNode.Left != null && itemNode.Left.Left != null)) return null;

    Node<T> temp2 = itemNode.Left.Right;
    Node<T> temp3 = itemNode.Left.Left.Right;
    Node<T> itemNode2 = itemNode.Left;
    Node<T> itemNode3 = itemNode.Left.Left;
    itemNode2.Right = itemNode;
    itemNode.Left = temp2;
    itemNode2.Left = temp3;
    itemNode3.Right = itemNode2;

    if (item.CompareTo(temp.Item) > 0)
    {
        temp.Right = itemNode3;
    }
    else if (item.CompareTo(temp.Item) < 0)
    {
        temp.Left = itemNode3;
    }
    else
    {
        copyO.root = itemNode3;
    }
};
```

```
// reached leaf node
if (itemNode.Left == null && itemNode.Right == null)
{
    return CheckLeafNodeAndEqual(copyO, item, originalTree);
}
}
catch (Exception e)
{
    //Console.WriteLine(e.Message);
    return null;
}

// check
SplayTree<T> copy2 = (SplayTree<T>)copyO.Clone();
copy2.Insert(item);
if (copy2.Equals(originalTree))
{
    //continue reverses rotation
    SplayTree<T> r1 = UndoRightRight(copyO, item, originalTree);
    SplayTree<T> r2 = UndoLeftLeft(copyO, item, originalTree);
    SplayTree<T> r3 = UndoRightLeft(copyO, item, originalTree);
    SplayTree<T> r4 = UndoLeftRight(copyO, item, originalTree);
    SplayTree<T> r5 = UndoLeft(copyO, item, originalTree);
    SplayTree<T> r6 = UndoRight(copyO, item, originalTree);

    List<SplayTree<T>> candidate = new List<SplayTree<T>>();

    if (r1 != null) candidate.Add(r1);
    if (r2 != null) candidate.Add(r2);
    if (r3 != null) candidate.Add(r3);
    if (r4 != null) candidate.Add(r4);
    if (r5 != null) candidate.Add(r5);
    if (r6 != null) candidate.Add(r6);
    SplayTree<T> mock = new SplayTree<T>();
    mock.root = copyO;
    candidate.Add(mock);
    if (candidate.Count > 0)
    {
        return candidate[0];
    }
}
else
{
    //wrong attempt
    return null;
}
return null;
}
```

```
SplayTree<T> UndoLeftRight(SplayTree<T> o, T item, SplayTree<T> originalTree)
{
    SplayTree<T> copyO = (SplayTree<T>)o.Clone();
    // traverse to parent of the target item
    Node<T> temp = traverseToItemParent(copyO, item);

    // reverse lr rotation
    try
    {
        Node<T> itemNode;

        // reverse lr
        if (item.CompareTo(temp.Item) > 0)
        {
            itemNode = temp.Right;
        }
        else if (item.CompareTo(temp.Item) < 0)
        {
            itemNode = temp.Left;
        }
        else
        {
            itemNode = temp;
        }
    };

    Node<T> temp2 = itemNode.Right.Left;
    Node<T> temp3 = itemNode.Left.Right;
    Node<T> itemNode2 = itemNode.Right;
    Node<T> itemNode3 = itemNode.Left;
    itemNode2.Left = itemNode;
    itemNode.Right = temp2;
    itemNode3.Right = itemNode;
    itemNode2.Left = itemNode3;
    itemNode.Left = temp3;

    if (item.CompareTo(temp.Item) > 0)
    {
        temp.Right = itemNode2;
    }
    else if (item.CompareTo(temp.Item) < 0)
    {
        temp.Left = itemNode2;
    }
    else
    {
        copyO.root = itemNode2;
    }
};
```

```

        // reached leaf node
        if (itemNode.Left == null && itemNode.Right == null)
        {
            return CheckLeafNodeAndEqual(copyO, item, originalTree);
        }
    }
    catch (Exception e)
    {
        //Console.WriteLine(e.Message);
        return null;
    }

    // check
    SplayTree<T> copy2 = (SplayTree<T>)copyO.Clone();
    copy2.Insert(item);
    if (copy2.Equals(originalTree))
    {
        //continue reverses rotation
        SplayTree<T> r1 = UndoRightRight(copyO, item, originalTree);
        SplayTree<T> r2 = UndoLeftLeft(copyO, item, originalTree);
        SplayTree<T> r3 = UndoRightLeft(copyO, item, originalTree);
        SplayTree<T> r4 = UndoLeftRight(copyO, item, originalTree);
        SplayTree<T> r5 = UndoLeft(copyO, item, originalTree);
        SplayTree<T> r6 = UndoRight(copyO, item, originalTree);
        List<SplayTree<T>> candidate = new List<SplayTree<T>>();
        if (r1 != null) candidate.Add(r1);
        if (r2 != null) candidate.Add(r2);
        if (r3 != null) candidate.Add(r3);
        if (r4 != null) candidate.Add(r4);
        if (r5 != null) candidate.Add(r5);
        if (r6 != null) candidate.Add(r6);
        SplayTree<T> mock = new SplayTree<T>();
        mock.root = copyO;
        candidate.Add(mock);
        if (candidate.Count > 0)
        {
            return candidate[0];
        }
    }
    else
    {
        //wrong attempt
        return null;
    }
    return null;
}

SplayTree<T> UndoRightLeft(SplayTree<T> o, T item, SplayTree<T> originalTree)
{

```

```
SplayTree<T> copyO = (SplayTree<T>)o.Clone();
// traverse to parent of the target item
Node<T> temp = traverseToItemParent(copyO, item);

// reverse rl rotation
try
{
    Node<T> itemNode;

    // reverse rr
    if (item.CompareTo(temp.Item) > 0)
    {
        itemNode = temp.Right;
    }
    else if (item.CompareTo(temp.Item) < 0)
    {
        itemNode = temp.Left;
    }
    else
    {
        itemNode = temp;
    }
};

Node<T> temp2 = itemNode.Left.Right;
Node<T> temp3 = itemNode.Right.Left;
Node<T> itemNode2 = itemNode.Left;
Node<T> itemNode3 = itemNode.Right;
itemNode2.Right = itemNode;
itemNode.Left = temp2;
itemNode3.Left = itemNode;
itemNode2.Right = itemNode3;
itemNode.Right = temp3;

if (item.CompareTo(temp.Item) > 0)
{
    temp.Right = itemNode2;
}
else if (item.CompareTo(temp.Item) < 0)
{
    temp.Left = itemNode2;
}
else
{
    copyO.root = itemNode2;
};

// reached leaf node
if (itemNode.Left == null && itemNode.Right == null)
{
```

```

        return CheckLeafNodeAndEqual(copyO, item, originalTree);
    }
}
catch (Exception e)
{
    //Console.WriteLine(e.Message);
    return null;
}

// check
SplayTree<T> copy2 = (SplayTree<T>)copyO.Clone();
copy2.Insert(item);
if (copy2.Equals(originalTree))
{
    //continue reverses rotation
    SplayTree<T> r1 = UndoRightRight(copyO, item, originalTree);
    SplayTree<T> r2 = UndoLeftLeft(copyO, item, originalTree);
    SplayTree<T> r3 = UndoRightLeft(copyO, item, originalTree);
    SplayTree<T> r4 = UndoLeftRight(copyO, item, originalTree);
    SplayTree<T> r5 = UndoLeft(copyO, item, originalTree);
    SplayTree<T> r6 = UndoRight(copyO, item, originalTree);
    List<SplayTree<T>> candidate = new List<SplayTree<T>>();
    if (r1 != null) candidate.Add(r1);
    if (r2 != null) candidate.Add(r2);
    if (r3 != null) candidate.Add(r3);
    if (r4 != null) candidate.Add(r4);
    if (r5 != null) candidate.Add(r5);
    if (r6 != null) candidate.Add(r6);
    SplayTree<T> mock = new SplayTree<T>();
    mock.root = copyO;
    candidate.Add(mock);
    if (candidate.Count > 0)
    {
        return candidate[0];
    }
}
else
{
    //wrong attempt
    return null;
}
return null;
}

SplayTree<T> UndoLeft(SplayTree<T> o, T item, SplayTree<T> originalTree)
{
    SplayTree<T> copyO = (SplayTree<T>)o.Clone();
    // traverse to parent of the target item
    Node<T> temp = traverseToItemParent(copyO, item);

```

```
// reverse l rotation
try
{
    Node<T> itemNode;
    // reverse l
    if (item.CompareTo(temp.Item) > 0)
    {
        itemNode = temp.Right;
    }
    else if (item.CompareTo(temp.Item) < 0)
    {
        itemNode = temp.Left;
    }
    else
    {
        itemNode = temp;
    };

    Node<T> temp2 = itemNode.Left.Right;
    Node<T> itemNode2 = itemNode.Left;
    itemNode2.Right = itemNode;
    itemNode.Left = temp2;

    if (item.CompareTo(temp.Item) > 0)
    {
        temp.Right = itemNode2;
    }
    else if (item.CompareTo(temp.Item) < 0)
    {
        temp.Left = itemNode2;
    }
    else
    {
        copyO.root = itemNode2;
    };

    // reached leaf node
    if (itemNode.Left == null && itemNode.Right == null)
    {
        return CheckLeafNodeAndEqual(copyO, item, originalTree);
    }
    else
    {
        return null;
    }
}
catch (Exception e)
{
}
```

```
//Console.WriteLine(e.Message);
return null;
}
}

SplayTree<T> UndoRight(SplayTree<T> o, T item, SplayTree<T> originalTree)
{
    SplayTree<T> copyO = (SplayTree<T>)o.Clone();
    // traverse to parent of the target item
    Node<T> temp = traverseToItemParent(copyO, item);

    // reverse l rotation
    try
    {
        Node<T> itemNode;
        // reverse l
        if (item.CompareTo(temp.Item) > 0)
        {
            itemNode = temp.Right;
        }
        else if (item.CompareTo(temp.Item) < 0)
        {
            itemNode = temp.Left;
        }
        else
        {
            itemNode = temp;
        }
        };

        Node<T> temp2 = itemNode.Right.Left;
        Node<T> itemNode2 = itemNode.Right;
        itemNode2.Left = itemNode;
        itemNode.Right = temp2;

        if (item.CompareTo(temp.Item) > 0)
        {
            temp.Right = itemNode2;
        }
        else if (item.CompareTo(temp.Item) < 0)
        {
            temp.Left = itemNode2;
        }
        else
        {
            copyO.root = itemNode2;
        }
        };

        // reached leaf node
        if (itemNode.Left == null && itemNode.Right == null)
```



```
        {
            return CheckLeafNodeAndEqual(copyO, item, originalTree);
        }
        else
        {
            return null;
        }
    }
    catch (Exception e)
    {
        //Console.WriteLine(e.Message);
        return null;
    }
}
```

```
// Public Size
// Returns the number of items in an splay tree
// Time complexity: O(n)
```

```
public int Size()
{
    return Size(root);    // Calls the private, recursive Size
}
```

```
// Size
// Calculates the size of the tree rooted at node
// Time complexity: O(n)
```

```
private int Size(Node<T> node)
{
    if (node == null)
        return 0;
    else
        return 1 + Size(node.Left) + Size(node.Right);
}
private Node<T> copyO;
```

```
// Public Print
// Outputs the items of an splay tree in sorted order
// Time complexity: O(n)
```

```
public void Print()
{
    int indent = 0;

    Print(root, indent);    // Calls private, recursive Print
    Console.WriteLine();
}
```

```
// Private Print
// Outputs items using an inorder traversal
// Time complexity: O(n)

private void Print(Node<T> node, int indent)
{
    if (node != null)
    {
        Print(node.Right, indent + 3);
        Console.WriteLine(new String(' ', indent) + node.Item.ToString());
        Print(node.Left, indent + 3);
    }
}
}
```

Node.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace COIS2020H_Assignment3_DavidMohammadChengCRY
{
    public class Node<T> where T : IComparable
    {
        // Read/write properties

        public T Item { get; set; }
        public Node<T> Left { get; set; }
        public Node<T> Right { get; set; }

        public Node(T item)
        {
            Item = item;
            Left = Right = null;
        }
    }
}
```

Program.cs

```
using COIS2020_Assignment3_DavidChengMohammaid_06122023Version;
using SplayTree;

class Program
{
    static void Main(string[] args)
    {
        // test for part A
        // including insert, remove, contains
        // rotations included left, right, left-left, right-right, left-right, right-left
        Console.WriteLine("Part A:\n");
        Program.TestPartA();

        // test for clone and equals method
        Console.WriteLine("\nPart B:\n");
        Program.TestCloneCreatesDeepCopy();

        // test for undo method
        Console.WriteLine("\nPart C:\n");
        Program.TestInsertUndoInsertSequence();
    }

    // Right Rotation Test
    public static void TestPartA()
    {
        SplayTree<int> tree = new SplayTree<int>();
        int[] items = { 10, 20, 30, 40, 50 };
        // involve insert, left and left-left rotation
        for (int i = 0; i < items.Length; i++)
        {
            Console.WriteLine("Insert " + items[i]);
            tree.Insert(items[i]);
            tree.Print();
        }

        // test for contain, right and right-right rotation
        bool contains10 = tree.Contains(20);
        Console.WriteLine("Contains 20: " + contains10);
        if (!contains10 || tree.Root.Item != 20)
        {
            Console.WriteLine("ERROR: contains 20 should have returned true or item of root !=
20");
        }
        Console.WriteLine("Resultant tree: ");
        tree.Print();

        // test for right-left rotation
```

```
Console.WriteLine("\nTest for right-left rotation:");
Console.WriteLine("Insert " + 25);
tree.Insert(25);
tree.Print();

Console.WriteLine("Remove " + 40);
if (!tree.Remove(40))
{
    Console.WriteLine("ERROR: remove 40 should have returned true");
}
Console.WriteLine("Resultant tree: ");
tree.Print();

// test for contain with non-exist value
Console.WriteLine("\nTest for contain with non-exist value:");
bool contains40 = tree.Contains(40);
Console.WriteLine("Contains 20: " + contains10);
if (contains40)
{
    Console.WriteLine("ERROR: contains 40 should have returned false");
}
Console.WriteLine("Resultant tree: ");
tree.Print();

// test for delete non-exist value
Console.WriteLine("\nTest for delete non-exist value: ");
Console.WriteLine("Remove " + 40);
if (tree.Remove(40))
{
    Console.WriteLine("ERROR: remove 40 should have returned false");
}
Console.WriteLine("Resultant tree: ");
tree.Print();

// preparation for testing afterwards
Console.WriteLine("\nTest for testing afterwards: ");
Console.WriteLine("Contains 10: " + tree.Contains(10));
Console.WriteLine("Resultant tree: ");
tree.Print();

Console.WriteLine("Contains 50: " + tree.Contains(50));
Console.WriteLine("Resultant tree: ");
tree.Print();

// test for left-right rotation
Console.WriteLine("\nTest for left-right rotation ");
Console.WriteLine("Insert 26");
tree.Insert(26);
Console.WriteLine("Resultant tree: ");
```

```
        tree.Print();
    }

    public static void TestCloneCreatesDeepCopy()
    {
        // Create and populate the original splay tree
        SplayTree<int> originalTree = new SplayTree<int>();
        originalTree.Insert(10);
        originalTree.Insert(5);
        originalTree.Insert(20);

        // Clone the tree
        SplayTree<int> cloneTree = (SplayTree<int>)originalTree.Clone();

        // Check if the root the same for every node
        if (originalTree.Root.Item != cloneTree.Root.Item
            || originalTree.Root.Left.Item != cloneTree.Root.Left.Item
            || originalTree.Root.Right.Item != cloneTree.Root.Right.Item)
        {
            Console.WriteLine("ERROR CLONE: The clone tree is not exactly the same as the
original tree!");
        }

        // Check if the value of cloneTree the same as the original tree through the Equals method
        implemented by ourselves
        bool same = originalTree.Equals(cloneTree);
        if (!same)
        {
            Console.WriteLine("ERROR CLONE: The clone tree is not exactly the same as the
original tree!");
        }
        Console.WriteLine("Clone method PASSED: the clone tree contains the same value as the
original tree");

        // Since the clone tree is a deep copy of the original tree, changing the value of the clone
        tree
        // should have no effect on the original tree
        // first we insert a value to the clone
        cloneTree.Insert(10000);
        // stillSame should have returned false
        bool stillSame = originalTree.Equals(cloneTree);
        if (stillSame)
        {
            Console.WriteLine("ERROR CLONE DEEP COPY: Clone tree still the same as
original tree after inserting!");
        }
        Console.WriteLine("Clone Deep copy property PASSED: modification on clone tree didn't
affect the original tree");
    }
}
```

```
}

public static void TestInsertUndoInsertSequence()
{
    // Create an empty splay tree
    SplayTree<int> splayTree = new SplayTree<int>();

    // Insert items
    splayTree.Insert(10);
    splayTree.Insert(20);
    splayTree.Insert(30);

    // Make a deep copy of the tree to compare later
    SplayTree<int> copyAfterSuccessfulInsert = (SplayTree<int>)splayTree.Clone();
    SplayTree<int> copy = splayTree.Undo();
    // Assuming the Clone method from the previous scenario is defined

    // Perform Undo to revert the insert
    Console.WriteLine("Original Tree:");
    splayTree.Print();
    Console.WriteLine("\nUndo of the original tree: ");
    copy.Print();

    // Insert the same item again
    copy.Insert(30);
    Console.WriteLine("\nUndo tree after inserting 30: ");
    splayTree.Print();

    // Assert: Use the Equals method to verify the tree was reproduced //correctly //after the
    insert-undo-insert sequence
    if (!splayTree.Equals(copyAfterSuccessfulInsert))
    {
        Console.WriteLine("ERROR UNDO: Insert undo insert didn't product the same tree!");
    }
    Console.WriteLine("Undo method PASSED: Insert undo insert product the same tree!");
}
}
```

4. Test Cases of Part A, B and C:

Testing Part A:

Test Case #	Task	Test to be performed	Expected Result	Actual result	Pass/Fail
1	Insert successful	Inserting 1, 3, 5, 7. First inserting 1 then 3 then 5 and lastly 7	<pre> 7 5 3 1 </pre>	<pre> Insert 1 1 Insert 3 3 1 Insert 5 5 3 1 Insert 7 7 5 3 1 </pre>	Pass/Fail

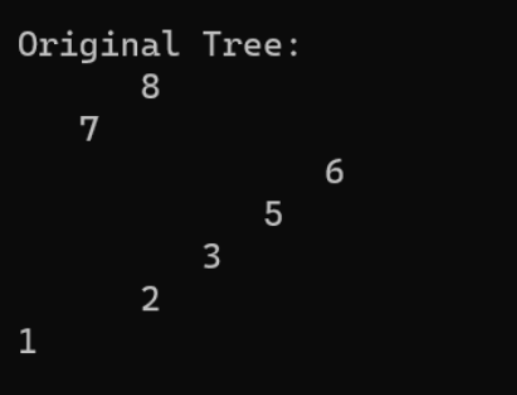
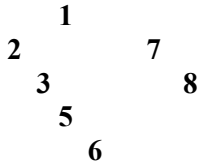
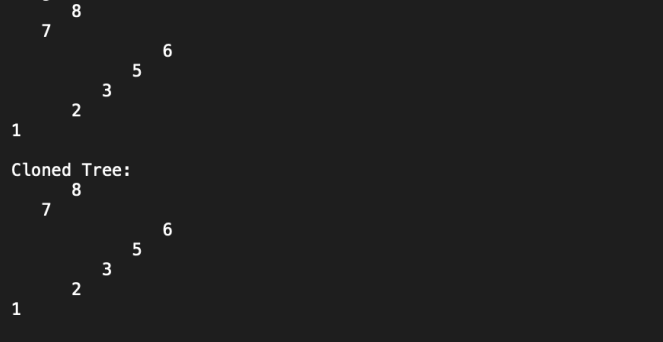

2	Last insert in the root	Now inserting 2 in the previous tree and 2 will be at the root through rotation.	<pre> 2 / \ 1 5 \ / \ 3 7 </pre>	<pre> Insert 2 7 / \ 5 3 / \ 2 1 </pre>	Pass/Fail
3	Successfully removing an item	Removing item 4 from the tree below: <pre> Original Tree: 8 / \ 7 6 / \ / \ 5 4 3 2 / \ 1 </pre>	<pre> 3 / \ 2 7 / \ / \ 1 5 6 8 </pre> <p>Here 4 will first go to the root then it will be removed and the next highest item which is 3 will be on the root.</p>	<pre> Removing item 4: Remove success 8 / \ 7 6 / \ / \ 3 5 2 1 </pre>	Pass/Fail
4	Unsuccessful remove	Removing item 40 from the tree below: <pre> 8 / \ 7 6 / \ / \ 3 5 2 1 </pre>	<p>Here as 40 is not in the tree, 8 is the next highest priority item and the tree will be rearranged to make 8 the root.</p> <pre> 8 / \ 7 3 / \ / \ 1 2 5 6 </pre>	<pre> Removing item 40: Remove fail, item not presented in the tree 8 / \ 7 6 / \ / \ 3 5 2 1 </pre>	Pass/Fail

5	Unsuccessful retrieve (with contain())	<p>Checking if item 30 is in the tree below:</p> <pre> 8 7 6 5 3 2 1 </pre>	<p>As item 30 doesn't contain in the tree, the root will be the next highest priority item which is 8. Therefore the tree will remain the same and it will show "Flase" message.</p>	<pre> Checking if item 30 is in the tree: False 8 7 6 5 3 2 1 </pre>	Pass/Fail
6	Successfully retrieving an item (with contain())	<p>Checking if item 1 is in the tree below:</p> <pre> 8 7 6 5 3 2 1 </pre>	<p>As item 1 contains in the tree, the root will be 1 and a "True" message will be shown.</p> <pre> 1 7 8 2 3 5 6 3 5 6 </pre>	<pre> Checking if item 1 is in the tree: True 8 7 6 5 3 2 1 </pre>	Pass/Fail

7	Successfully rotated the tree	To perform left-right rotation to insert 26 from Resultant tree to another resultant tree	The result: <pre> 50 / \ 30 26 / \ 25 20 \ / 10</pre>	<pre>contains 601 tree Resultant tree: 50 30 25 20 \ / 10 Test for left-right rotation Insert 26 Resultant tree: 50 / \ 30 26 / \ 25 20 \ / 10</pre>	Pass/Fail
---	-------------------------------	---	---	--	-----------

8	Successfully rotated the tree	To perform right-left rotation to insert 25 from Resultant tree to another resultant tree	The result: <pre> 50 40 30 25 20 10</pre>	<pre>Resultant tree: 50 40 30 20 10 Test for right-left rotation: Insert 25 50 40 30 25 20 10</pre>	Pass/Fail
---	-------------------------------	---	--	---	-----------

Testing Part B:

Test Case #	Task	Test to be performed	Expected Result	Actual result	Pass/Fail
1	Cloning tree	Cloning a tree to make a identical tree of the tree below: 	The cloned tree should looks exactly like the original tree. 	Checking if the cloned tree is equal to the original tree: True Original Tree:  Cloned Tree: 	Pass/Fail

Testing Part C:

Test Case #	Task	Test to be performed	Expected Result	Actual result	Pass/Fail
1	For carrying out an item from previous part by Undo	To create an output that creates a copy using splaytree's undo, as recording the last version of the splay tree	<p>The original tree:</p> <pre> 30 20 10 </pre> <p>The undo original tree:</p> <pre> 20 10 </pre> <p>Undo tree after inserting 30:</p> <pre> 30 20 10 </pre>	<pre> Part C: Original Tree: 30 20 10 Undo of the original tree: 20 10 Undo tree after inserting 30: 30 20 10 Undo method PASSED: Insert undo insert product the same tree! </pre>	Pass/Fail

5. Conclusion:

We had successfully completed part A, B, an C's requirements and show the result and test case in this document.