GTU Department of Computer Engineering
CSE 222/505 - Spring 2022
Homework 7 Report

ÇAĞRI ÇAYCI
1901042629

1. System Requirements

• Binary Search Tree Class

To implement both toAVL method and toBinarySearchTree method, Binary Search Tree class should be implemented with some methods. Not all the methods of Binary Search Tree are required. Add and rotates methods are enough to implement toAVL and toBinarySearchTree method.

Binary Search Tree must keep address of the starting node of the Binary Search tree, number of elements on the Binary Search Tree and the information about Binary Search Tree is balanced or not. whether it is balanced or not should be initially set to false because it is not certain that binary search tree is balanced or not.

Binary Search Tree should have an inner class to keep nodes. Node class should keep data, right and left child, parent, height, and weight of the node. Height and weight information is required for toAVL method.

• Binary Tree class

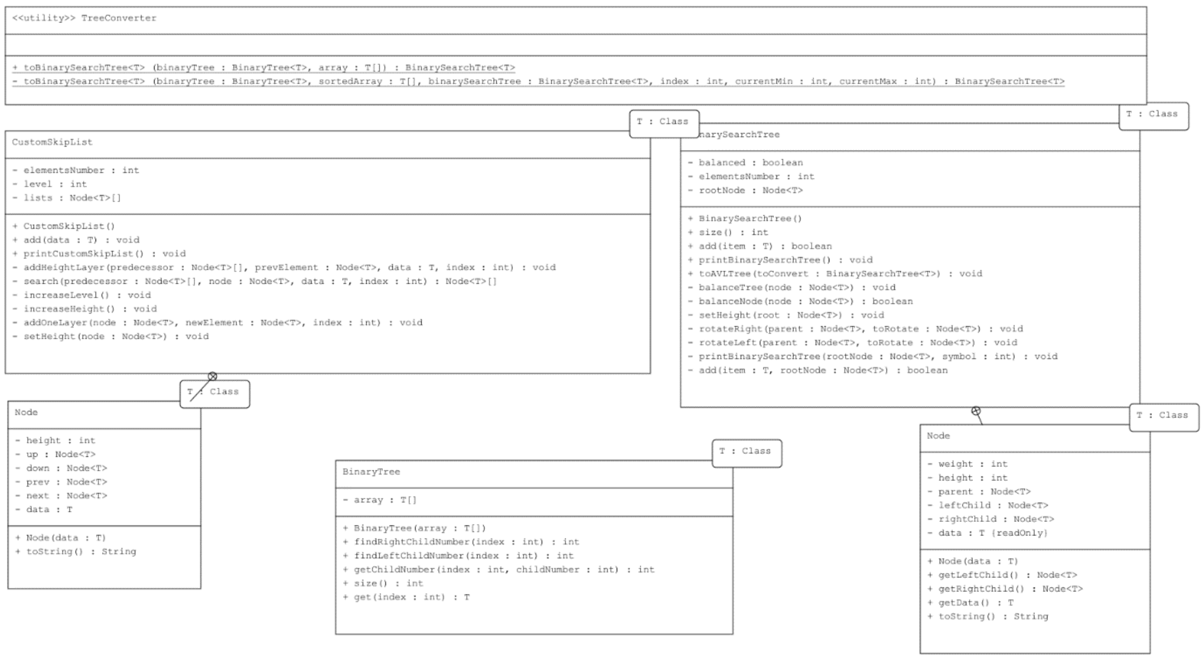
Binary Tree class should be implemented with some methods to get a Binary Tree as a parameter in toBinarySearchTree method. Not all the methods of Binary Tree are required. Get method is enough to implement toBinarySearchTree method.

• Custom Skip List

To implement Custom Skip List three data fields are required. Two integers to keep number of lists, number of elements. One Node array is required to keep roots of the lists. A constructor is required to create object of the Custom Skip List. An add method is required to add element to the Custom Skip List. A print method is required to print the Custom Skip List.

Custom Skip List should have an inner class to keep nodes. Node class should keep address of next, previous, up, and down nodes. Node class also should keep data and height of the node.

2. Class Diagrams


```
<<utility>> TreeConverter

+ toBinarySearchTree<T> (binaryTree : BinaryTree<T>, array : T[]) : BinarySearchTree<T>
- toBinarySearchTree<T> (binaryTree : BinaryTree<T>, sortedArray : T[], binarySearchTree : BinarySearchTree<T>, index : int, currentMin : int, currentMax : int) : BinarySearchTree<T>
```

```
CustomSkipList                                    T : Class
- elementsNumber : int
- level : int
- lists : Node<T>[]
+ CustomSkipList()
+ add(data : T) : void
+ printCustomSkipList() : void
- addHeightLayer(predecessor : Node<T>[], prevElement : Node<T>, data : T, index : int) : void
- search(predecessor : Node<T>[], node : Node<T>, data : T, index : int) : Node<T>[]
- increaseLevel() : void
- increaseHeight() : void
- addOneLayer(node : Node<T>, newElement : Node<T>, index : int) : void
- setHeight(node : Node<T>) : void
```

```
narySearchTree                                    T : Class
- balanced : boolean
- elementsNumber : int
- rootNode : Node<T>
+ BinarySearchTree()
+ size() : int
+ add(item : T) : boolean
+ printBinarySearchTree() : void
+ toAVLTree(toConvert : BinarySearchTree<T>) : void
- balanceTree(node : Node<T>) : void
- balanceNode(node : Node<T>) : boolean
- setHeight(root : Node<T>) : void
- rotateRight(parent : Node<T>, toRotate : Node<T>) : void
- rotateLeft(parent : Node<T>, toRotate : Node<T>) : void
- printBinarySearchTree(rootNode : Node<T>, symbol : int) : void
- add(item : T, rootNode : Node<T>) : boolean
```

```
Node                                              T : Class
- height : int
- up : Node<T>
- down : Node<T>
- prev : Node<T>
- next : Node<T>
- data : T
+ Node(data : T)
+ toString() : String
```

```
BinaryTree                                        T : Class
- array : T[]
+ BinaryTree(array : T[])
+ findRightChildNumber(index : int) : int
+ findLeftChildNumber(index : int) : int
+ getChildNumber(index : int, childNumber : int) : int
+ size() : int
+ get(index : int) : T
```

```
Node                                              T : Class
- weight : int
- height : int
- parent : Node<T>
- leftChild : Node<T>
- rightChild : Node<T>
- data : T {readOnly}
+ Node(data : T)
+ getLeftChild() : Node<T>
+ getRightChild() : Node<T>
+ getData() : T
+ toString() : String
```

3. Problem Solution Approach

- ### Creating a Binary Search Tree which has the same structure with a given Binary Tree with an array of elements

In the binary search tree, the insertion method works like this: the new element starts to be compared from the first node of the tree. If the new element is larger than the node, the comparison continues from the right child, if it is smaller, comparison continues from the left child. It is added when the element encounters null. So, to protect the structure of the Binary Tree, there are some procedures when adding elements. These procedures start from the root node of the Binary Tree. If node has only left child, it means that data of this node must be bigger than its child nodes. So, current max element of the array must be added to the Binary Search Tree, and index of the current max element on array must be decreased by one. If node has only right child, it means that data of this node must be less than its child nodes. So, current min element of the array must be added to the Binary Search Tree, and index of the current min element on array must be increased by one. If a node has both right and left child, array must be divided into two. First array size is children number of left child, second array size is children number of right child. The remaining element must be added to the Binary Search Tree.

```
Binary Tree             Binary Search Tree              Array
      0                                                 {1, 2, 3, 4}
        0
          0
            0

      0                         1                       {2, 3, 4}
        0
          0
            0

      0                         1                       {3, 4}
        0                         2
          0
            0

      0                         1                       {4}
        0                         2
          0                         3
            0

      0                         1                       {}
        0                         2
          0                         3
            0                         4
```

```
Binary Tree               Binary Search Tree                  Array
       0                                                      {1, 2, 3, 4}
     0
   0
0

       0                           4                          {1, 2, 3}
     0
   0
0

       0                           4                          {1, 2}
     0                           3
   0
0

       0                           4                          {1}
     0                           3
   0                           2
0

       0                           4                          {}
     0                           3
   0                           2
0                            1
```

```
Binary Tree             Binary Search Tree              Array
       0                                                {1, 2, 3, 4, 5}
     0   0
   0  0

       0                        4                       {1, 2, 3, 4, 5}
     0   0
   0  0

       0                        4                       {1, 2, 3}
     0   0                    2
   0  0

       0                        4                       {1}
     0   0                    2
   0  0                      1

       0                        4                       {3}
     0   0                    2
   0  0                      1 3

       0                        4                       {5}
     0   0                    2   5
   0  0    0                 1 3
```

- ## Converting a Binary Search Tree into AVL Tree

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes. So, when converting a Binary Search Tree to AVL tree, it is necessary to balance the Binary Search Tree. Two rotating operation is defined to balance a Binary Search tree. These are rotate right and rotate left. There are 4 different states when these operations used. State one is Left-Left. If weight of a node is less than -1 and weight of the left child of the node is less than 1, right rotate is applied on node. State two is Left-Right. If weight of a node is less than -1 and weight of the left child of the node is 1, first left rotate is applied on left child of the node. After right rotate is applied on node. State three is Right-Right. If weight of a node is bigger than 1 and weight of the right child of the node is bigger than -1, left rotate is applied on node. State four is Right-Left. If weight of a node is bigger than 1 and weight of the right child of the node is -1, right rotate is applied on right child of the node. After left rotate is applied on node.

Balancing operation is starts from most left child. After setting its height and weight, rotation operations are applied if they are necessary.

- ## Creating a Skip List

  - ### Setting height of a node

To set a node height, node must be added to the first list. After adding a new element, the distance of the element to the nearest right and left tall buildings is found. If there is no tall building to the right, the distance to the end of the list is found. If there is no tall building on the left side, its distance to the beginning of the list is found.

Then these distances are added together and divided by 10 to get a ratio. The height of the new element is found according to this ratio and can be as high as the skip list height.

  - ### Adding an element to Skip List

After the predecessors are found, adds the element first list. And sets its height. If its height is 1, terminates adding operation. If its height is bigger than 1, adds the element upper lists.

- **Searching for an element on Skip List**

A Node array is created with as heigh as skip list height. And fills the array with starting from top list. Finds predecessor of element on top list and it goes until predecessor of first list is found.

- **Increasing height of a node**

  Height of a node is increased by one every 10 times if height of the node is bigger than one. To increase the height of a node it is necessary to add the node as upper node, also it is necessary to add the node in upper list. So, height of every node of second level list must be increased.

4. Test Cases

toBinarySearchTree Method

Testing with a Binary Tree full of left child.

Testing with a Binary Tree full of right child.

Testing with a Binary Tree different structure.

toAVLTree Method

Testing with a Binary Search Tree full of left child.

Testing with a Binary Search Tree full of right child.

Testing with a Binary Search Tree filled randomly.

Custom Skip List

Testing with random values.

5. Results

Results are on output.txt file.