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

Muhammed Sinan Pehlivanoğlu
1901042664

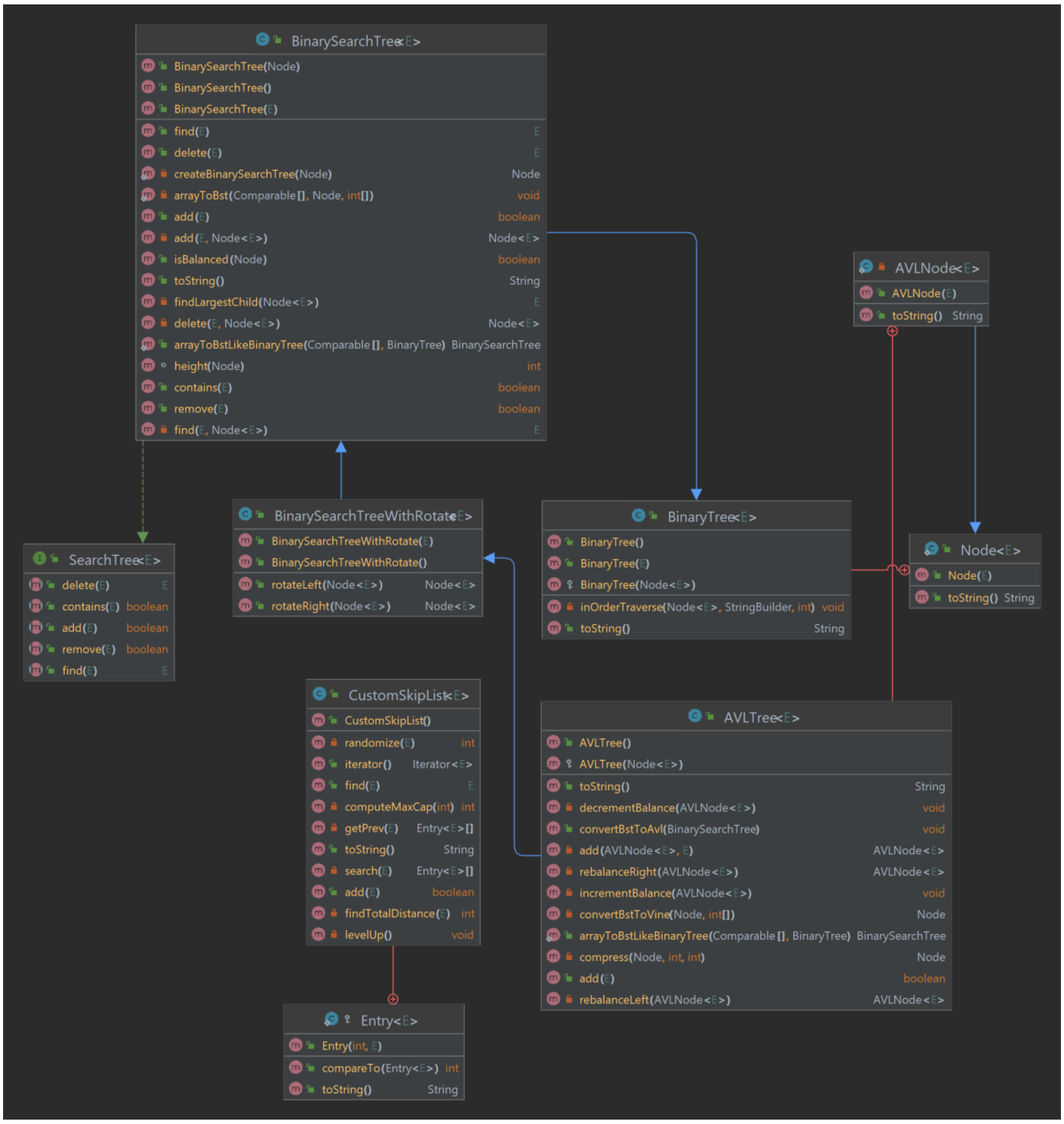
1. SYSTEM REQUIREMENTS

Operating System need to jdk and jre for start the program.
Stack capacity should not be less.
You need to enough space to store data's of program according to the how many you have objects.

2.PROBLEM SOLUTION APPROACH

- ***For first problem**, The new binary search tree is created like desired binary tree structure firstly. Then the items of array is sorted and added to new binary search tree using inorder traversal.*
- ***For second probLem**, Day-Stout Warren algorithm is used. I used the recursive version of algorithm. First the tree is converted to the vine. Like a right based chain. Then the vine is balanced begin with head node according to the mathematical algorithm used in Day-Stout Warren algorithm.*
- ***For third problem**, Default level of the CustomSkipList is 2. The level of the CSList is increased if the number of its element exceed the multiplies of 10. (in 11,21,31.. elements). İf the level of the CSList is increased. Then level of the all of the tall items is increased by one. The level of new inserted item is decided by randomize level algorithm in the problem.*

3.CLASS DIAGRAMS



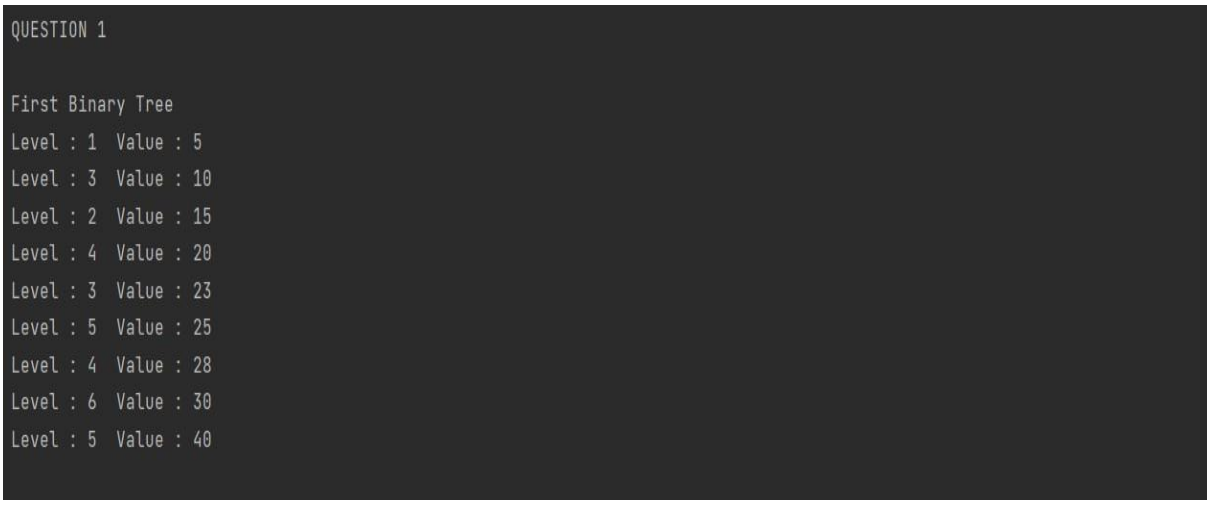
4.TEST CASES

- *Creating the binary search tree as the binary tree structure with desired array.*
- *Print the binary Search tree.*
- *Converting the binary search tree to balanced tree using rotation.*
- *Print the binary search tree.*
- *Check if the binary tree is balanced or not.*
- *Creating CustomSkip List and add the element into the Skip List.*
- *Print the Custom Skip List in each level increasing.*

5.RUNNING COMMANDS AND RESULTS

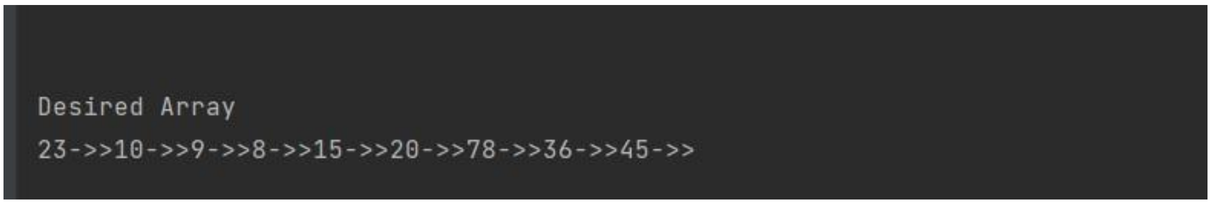
Q1 RESULT

First Binary Tree Structure



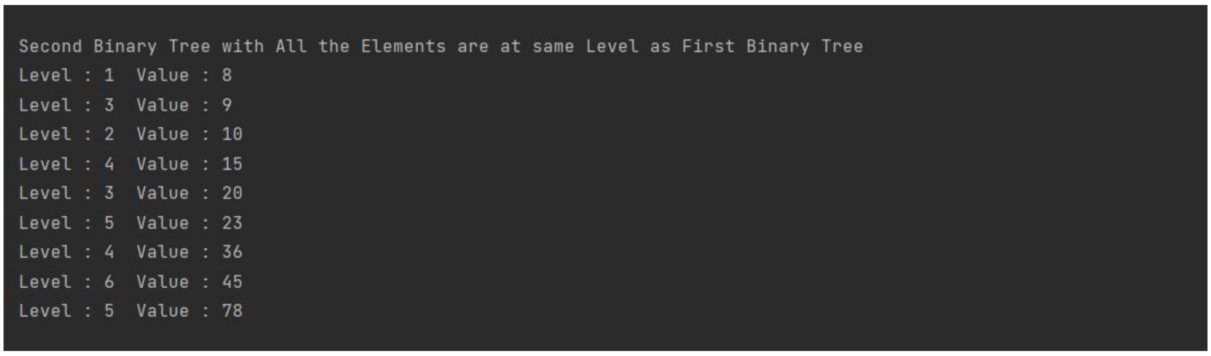
```
QUESTION 1

First Binary Tree
Level : 1  Value : 5
Level : 3  Value : 10
Level : 2  Value : 15
Level : 4  Value : 20
Level : 3  Value : 23
Level : 5  Value : 25
Level : 4  Value : 28
Level : 6  Value : 30
Level : 5  Value : 40
```

Desired array to be inserted to BinarySearch Tree



```
Desired Array
23->>10->>9->>8->>15->>20->>78->>36->>45->>
```

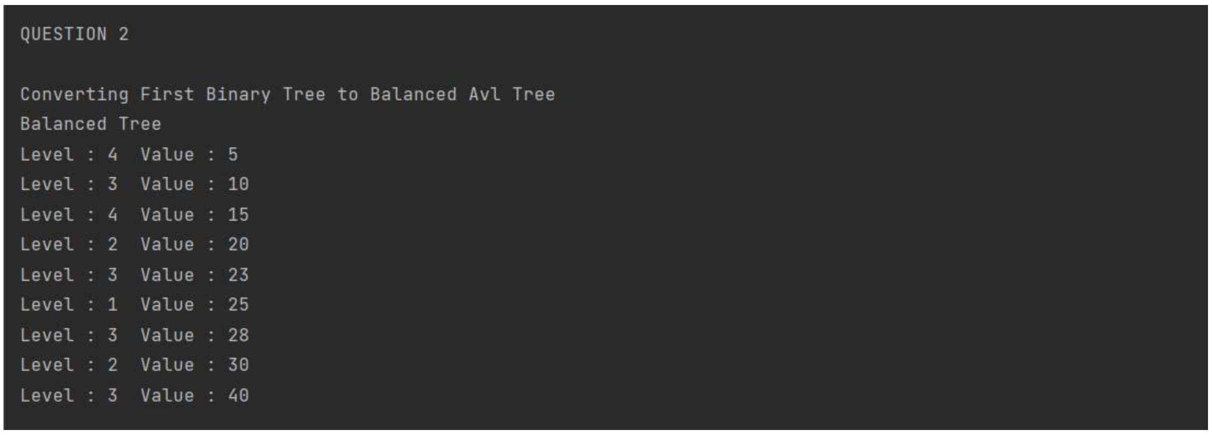
Second Binary Search. All the levels are same with the first binary tree.



```
Second Binary Tree with All the Elements are at same Level as First Binary Tree
Level : 1  Value : 8
Level : 3  Value : 9
Level : 2  Value : 10
Level : 4  Value : 15
Level : 3  Value : 20
Level : 5  Value : 23
Level : 4  Value : 36
Level : 6  Value : 45
Level : 5  Value : 78
```

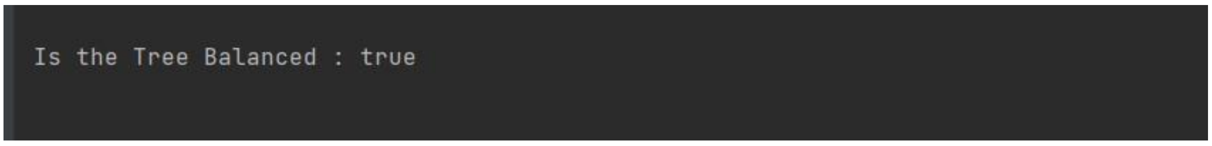
Q2 RESULT

Desired Binary Search Tree to be converted the balanced tree.



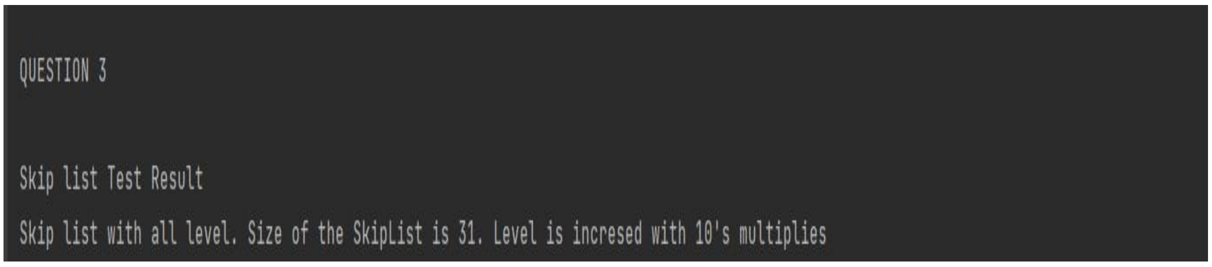
```
QUESTION 2

Converting First Binary Tree to Balanced Avl Tree
Balanced Tree
Level : 4  Value : 5
Level : 3  Value : 10
Level : 4  Value : 15
Level : 2  Value : 20
Level : 3  Value : 23
Level : 1  Value : 25
Level : 3  Value : 28
Level : 2  Value : 30
Level : 3  Value : 40
```

Checking the new tree is balanced or not.



```
Is the Tree Balanced : true
```

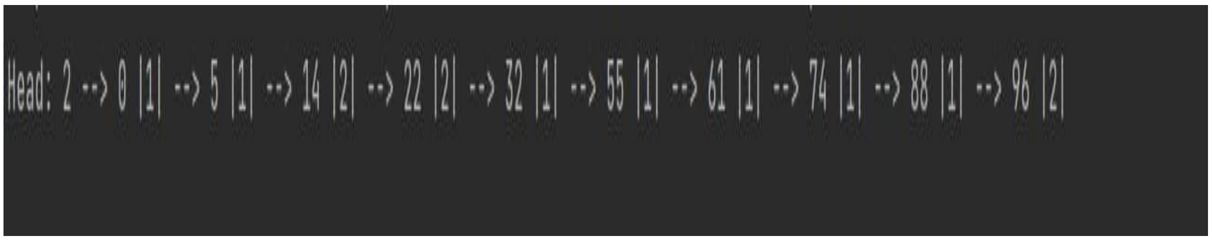
Q3 RESULT



```
QUESTION 3


Skip list Test Result
Skip list with all level. Size of the SkipList is 31. Level is incresed with 10's multiplies
```

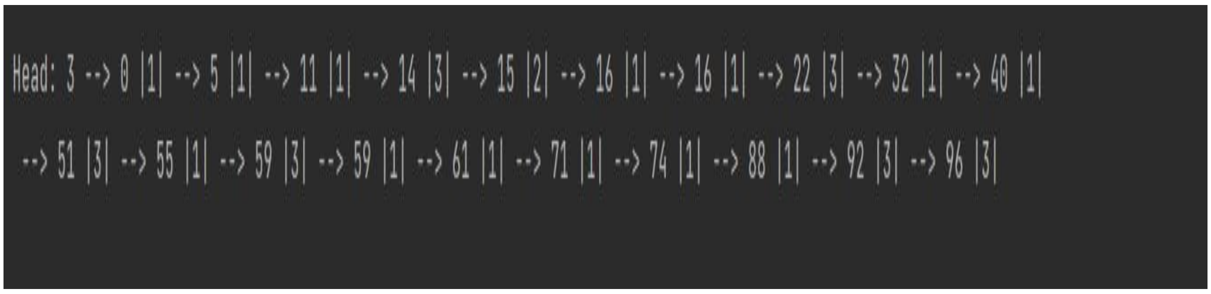
DATA|LEVEL| DESIGN

Default level skiplist. Number of the element is less than 11.



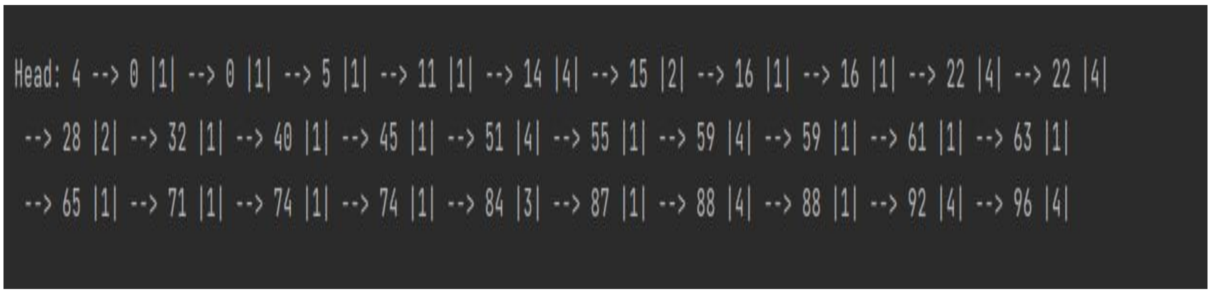
```
Head: 2 --> 0 |1| --> 5 |1| --> 14 |2| --> 22 |2| --> 32 |1| --> 55 |1| --> 61 |1| --> 74 |1| --> 88 |1| --> 96 |2|
```

Level of Skip list is increased. 3 level skiplist. Number of the element is less than 21.



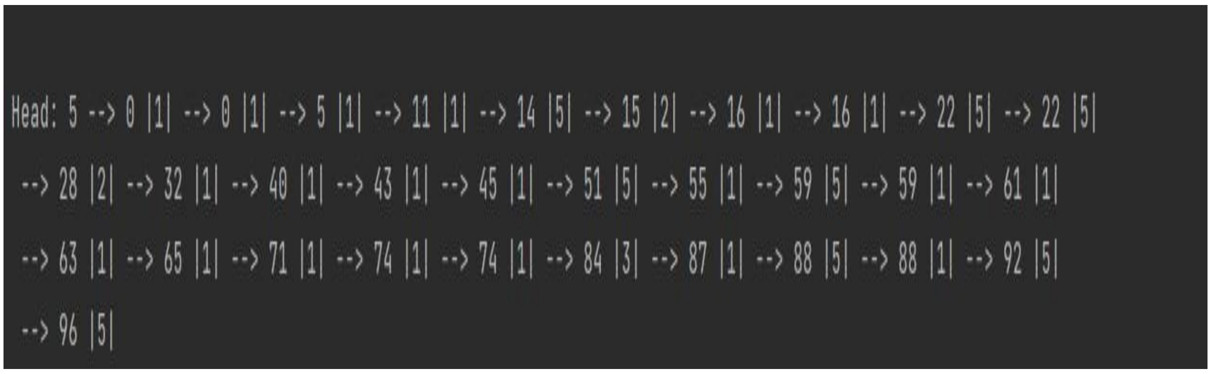
```
Head: 3 --> 0 |1| --> 5 |1| --> 11 |1| --> 14 |3| --> 15 |2| --> 16 |1| --> 16 |1| --> 22 |3| --> 32 |1| --> 40 |1|
 --> 51 |3| --> 55 |1| --> 59 |3| --> 59 |1| --> 61 |1| --> 71 |1| --> 74 |1| --> 88 |1| --> 92 |3| --> 96 |3|
```

Level of Skip list is increased. 4 level skiplist. Number of the element is less than 31.



```
Head: 4 --> 0 |1| --> 0 |1| --> 5 |1| --> 11 |1| --> 14 |4| --> 15 |2| --> 16 |1| --> 16 |1| --> 22 |4| --> 22 |4|
 --> 28 |2| --> 32 |1| --> 40 |1| --> 45 |1| --> 51 |4| --> 55 |1| --> 59 |4| --> 59 |1| --> 61 |1| --> 63 |1|
 --> 65 |1| --> 71 |1| --> 74 |1| --> 74 |1| --> 84 |3| --> 87 |1| --> 88 |4| --> 88 |1| --> 92 |4| --> 96 |4|
```

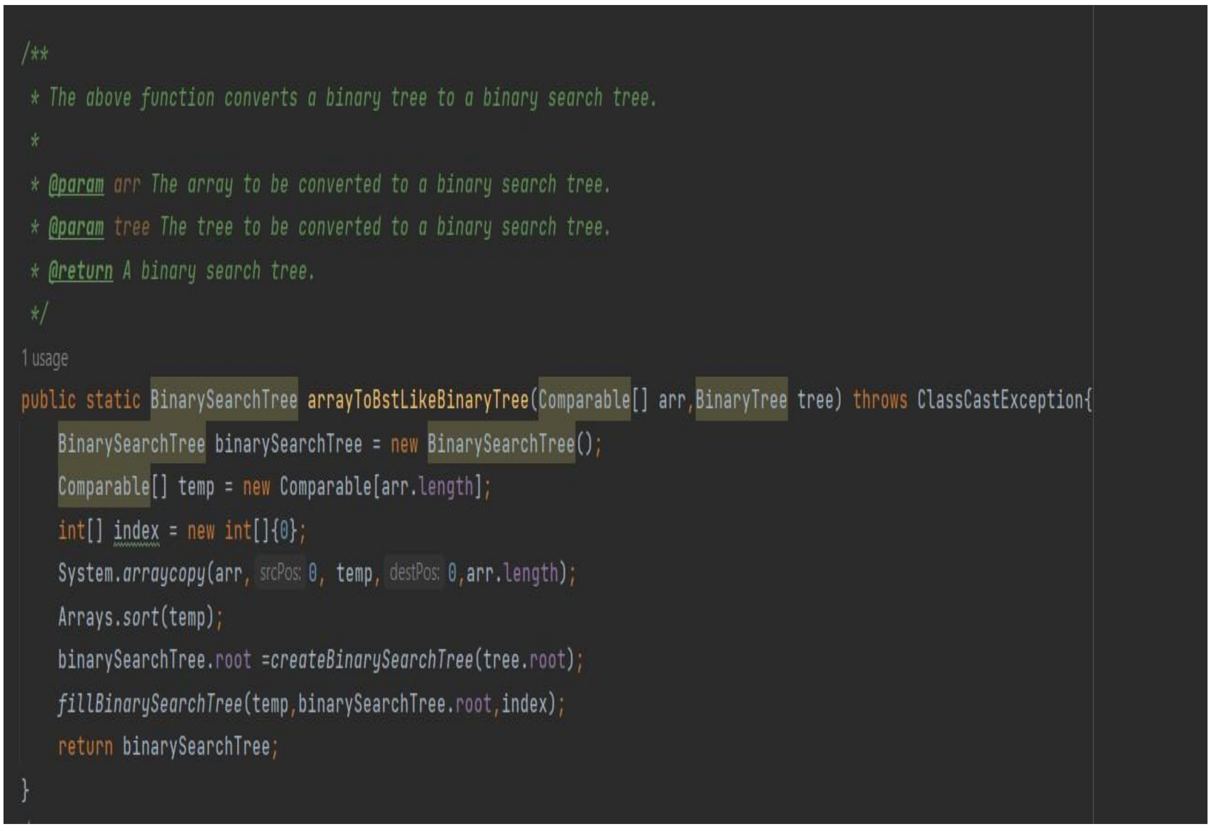
Level of Skip list is increased. 4 level skiplist. Number of the element is less than 41.



```
Head: 5 --> 0 |1| --> 0 |1| --> 5 |1| --> 11 |1| --> 14 |5| --> 15 |2| --> 16 |1| --> 16 |1| --> 22 |5| --> 22 |5|
 --> 28 |2| --> 32 |1| --> 40 |1| --> 43 |1| --> 45 |1| --> 51 |5| --> 55 |1| --> 59 |5| --> 59 |1| --> 61 |1|
 --> 63 |1| --> 65 |1| --> 71 |1| --> 74 |1| --> 74 |1| --> 84 |3| --> 87 |1| --> 88 |5| --> 88 |1| --> 92 |5|
 --> 96 |5|
```

6.TIME COMPLEXITY ANALYSIS

Q1:



```java
/**
 * The above function converts a binary tree to a binary search tree.
 *
 * @param arr The array to be converted to a binary search tree.
 * @param tree The tree to be converted to a binary search tree.
 * @return A binary search tree.
 */
1 usage
public static BinarySearchTree arrayToBstLikeBinaryTree(Comparable[] arr,BinaryTree tree) throws ClassCastException{
    BinarySearchTree binarySearchTree = new BinarySearchTree();
    Comparable[] temp = new Comparable[arr.length];
    int[] index = new int[]{0};
    System.arraycopy(arr, srcPos: 0, temp, destPos: 0,arr.length);
    Arrays.sort(temp);
    binarySearchTree.root =createBinarySearchTree(tree.root);
    fillBinarySearchTree(temp,binarySearchTree.root,index);
    return binarySearchTree;
}
```

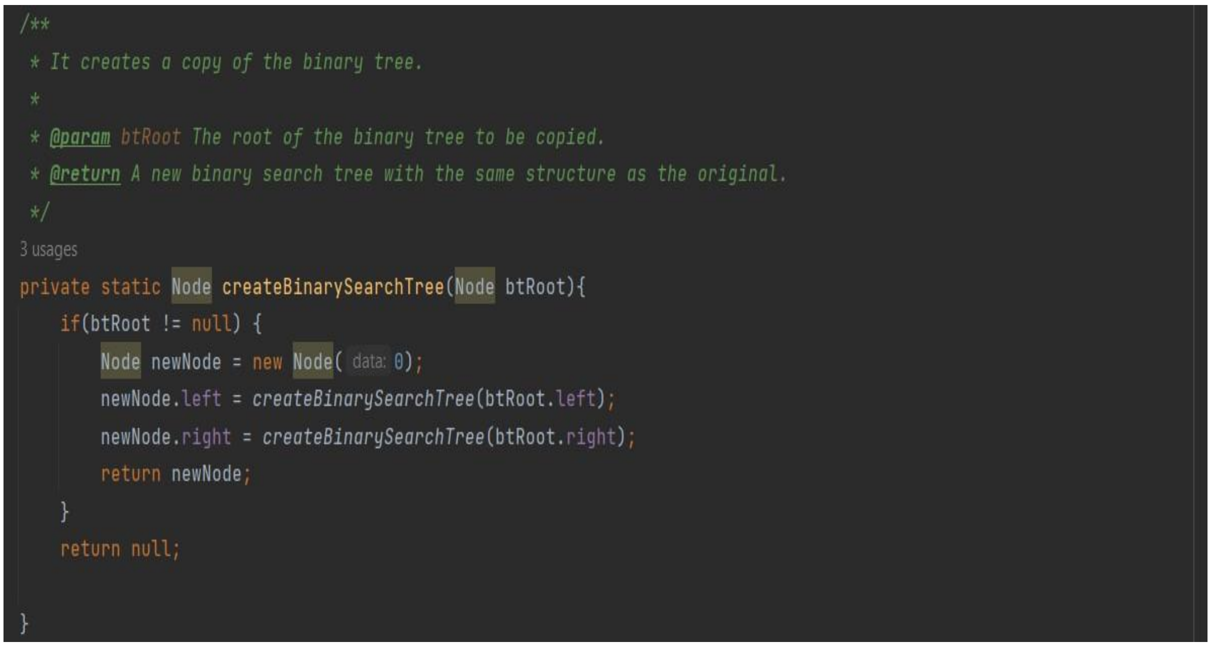
Array Copy : T(n) = θ (n)

Array Sort : MergeSort is used in java . T(n) = O(logn))

CreateBinarySearch : T(n) = O (nlogn). Using inorder traversal.

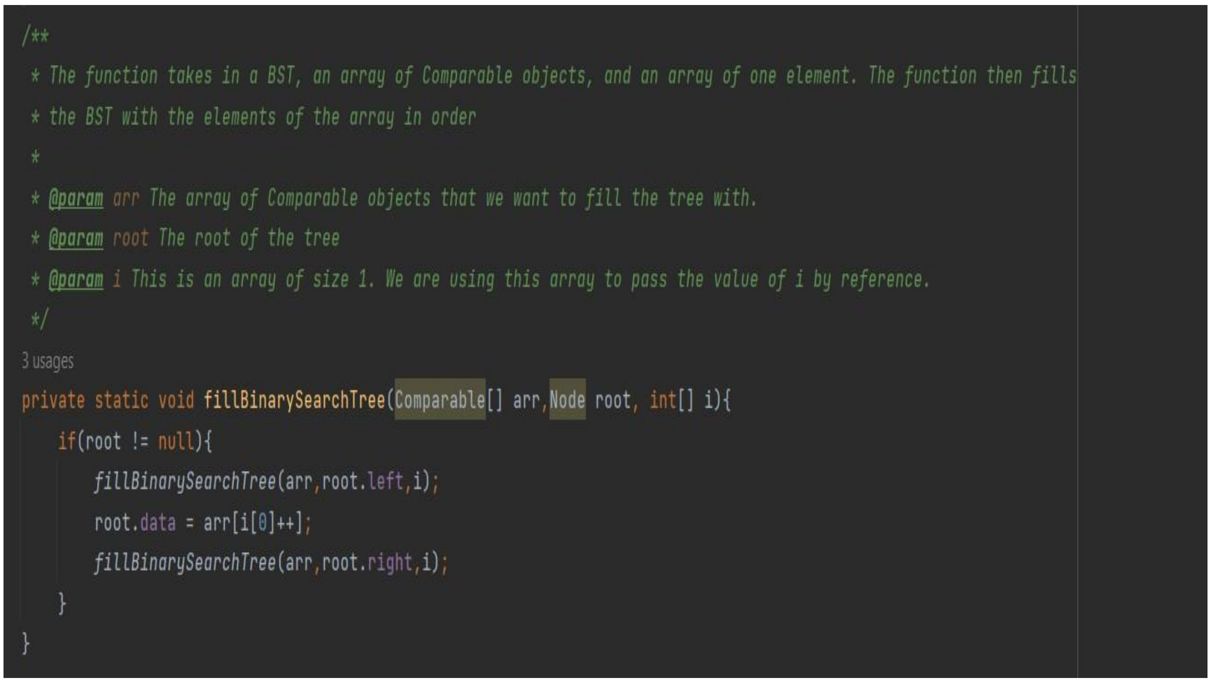
FillBinarySearchTree : T(n) = O (nlogn). Using inorder traversal.

So Time Complexitiy of Method is : T(n) = O(nlogn).



```java
/**
 * It creates a copy of the binary tree.
 *
 * @param btRoot The root of the binary tree to be copied.
 * @return A new binary search tree with the same structure as the original.
 */
3 usages
private static Node createBinarySearchTree(Node btRoot){
    if(btRoot != null) {
        Node newNode = new Node( data: 0);
        newNode.left = createBinarySearchTree(btRoot.left);
        newNode.right = createBinarySearchTree(btRoot.right);
        return newNode;
    }
    return null;

}
```

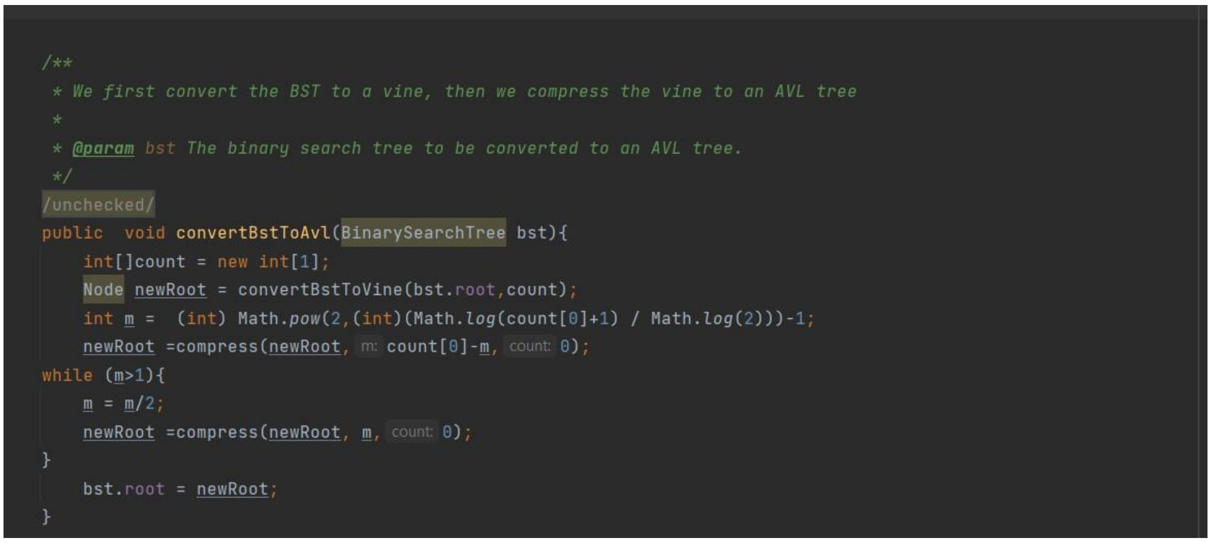
CreateBinarySearch : T(n) = θ (n). Using inorder traversal.



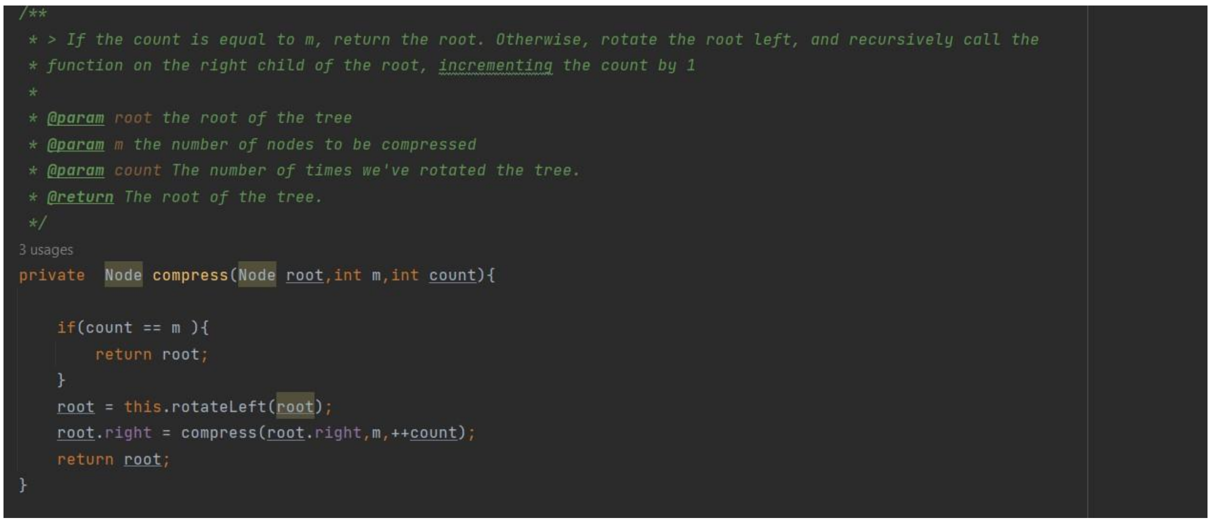
```java
/**
 * The function takes in a BST, an array of Comparable objects, and an array of one element. The function then fills
 * the BST with the elements of the array in order
 *
 * @param arr The array of Comparable objects that we want to fill the tree with.
 * @param root The root of the tree
 * @param i This is an array of size 1. We are using this array to pass the value of i by reference.
 */
3 usages
private static void fillBinarySearchTree(Comparable[] arr,Node root, int[] i){
    if(root != null){
        fillBinarySearchTree(arr,root.left,i);
        root.data = arr[i[0]++];
        fillBinarySearchTree(arr,root.right,i);
    }
}
```

FillBinarySearchTree : T(n) = θ (n). Using inorder traversal.

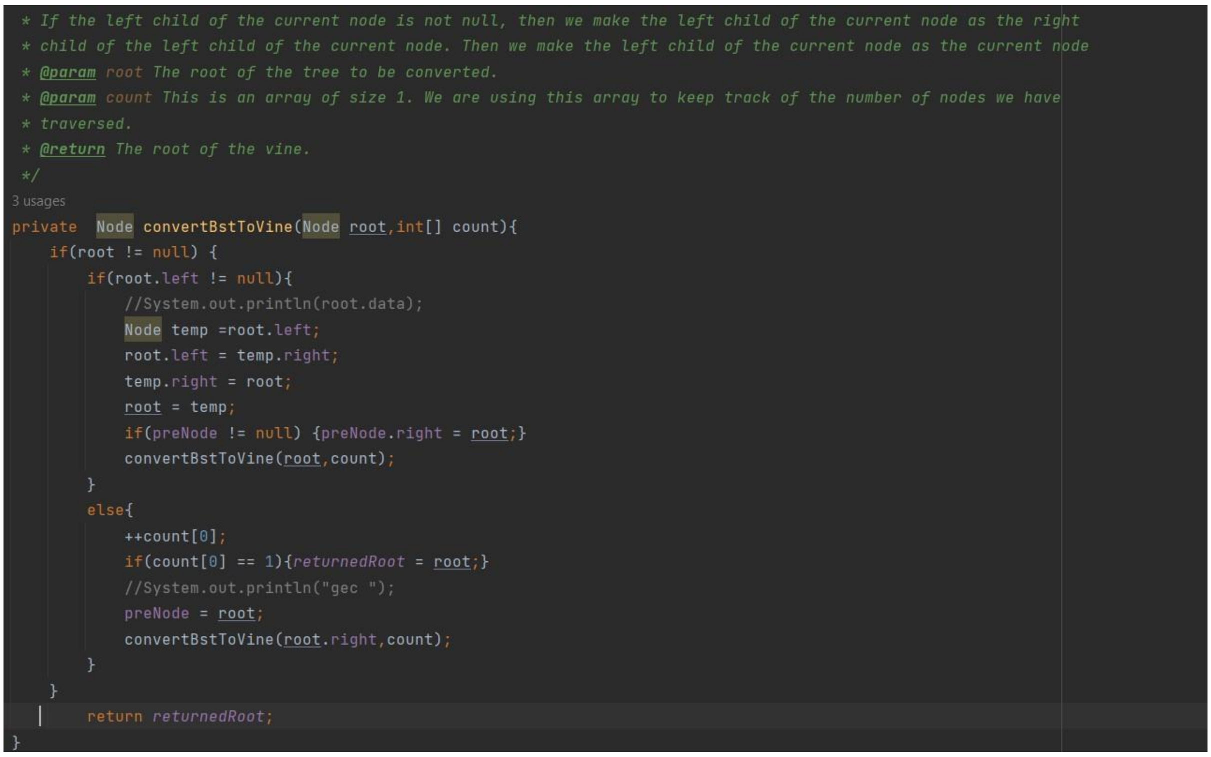
Q2:


```java
/**
 * We first convert the BST to a vine, then we compress the vine to an AVL tree
 *
 * @param bst The binary search tree to be converted to an AVL tree.
 */
/unchecked/
public  void convertBstToAvl(BinarySearchTree bst){
    int[]count = new int[1];
    Node newRoot = convertBstToVine(bst.root,count);
    int m =  (int) Math.pow(2,(int)(Math.log(count[0]+1) / Math.log(2)))-1;
    newRoot =compress(newRoot, m: count[0]-m, count: 0);
while (m>1){
    m = m/2;
    newRoot =compress(newRoot, m, count: 0);
}
    bst.root = newRoot;
}
```

Day-Warren Stout Algorithm is used. T(n) = O(n).


```java
/**
 * > If the count is equal to m, return the root. Otherwise, rotate the root left, and recursively call the
 * function on the right child of the root, incrementing the count by 1
 *
 * @param root the root of the tree
 * @param m the number of nodes to be compressed
 * @param count The number of times we've rotated the tree.
 * @return The root of the tree.
 */
3 usages
private  Node compress(Node root,int m,int count){

    if(count == m ){
        return root;
    }
    root = this.rotateLeft(root);
    root.right = compress(root.right,m,++count);
    return root;
}
```

T(n) = O(n). Since all the elements is chained right order.

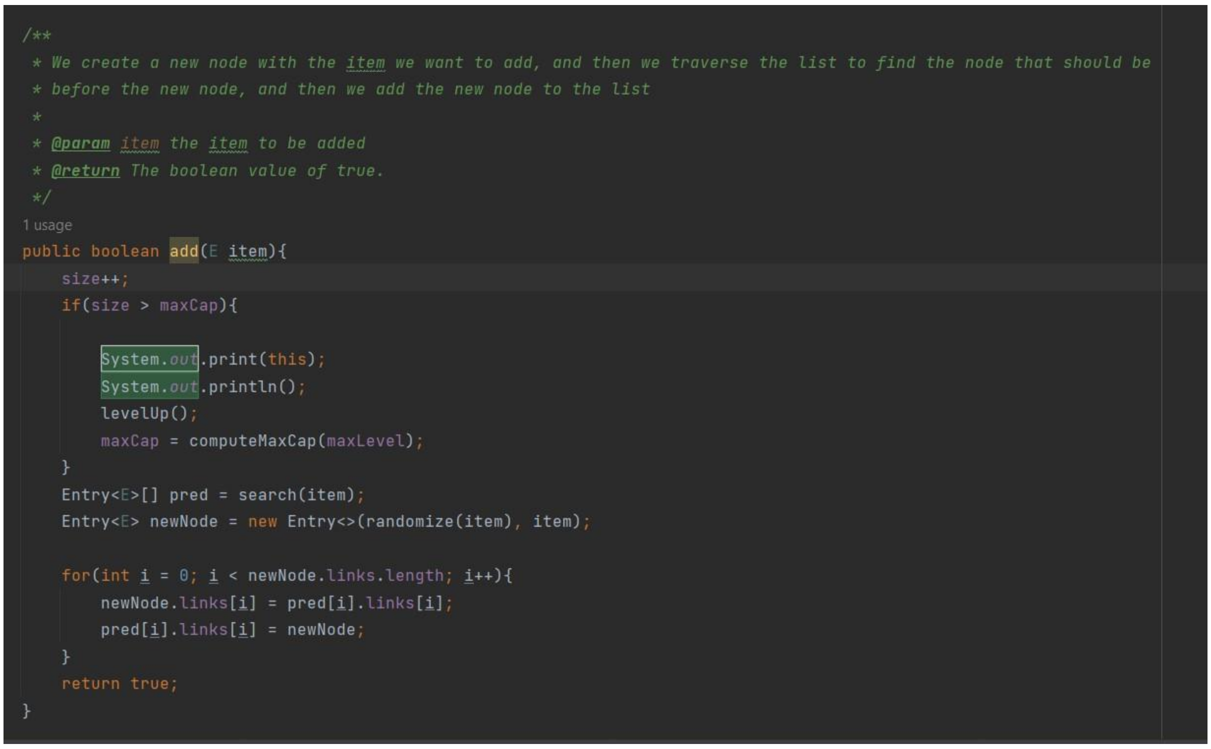


```
 * If the left child of the current node is not null, then we make the left child of the current node as the right
 * child of the left child of the current node. Then we make the left child of the current node as the current node
 * @param root The root of the tree to be converted.
 * @param count This is an array of size 1. We are using this array to keep track of the number of nodes we have
 * traversed.
 * @return The root of the vine.
 */
3 usages
private  Node convertBstToVine(Node root,int[] count){
    if(root != null) {
        if(root.left != null){
            //System.out.println(root.data);
            Node temp =root.left;
            root.left = temp.right;
            temp.right = root;
            root = temp;
            if(preNode != null) {preNode.right = root;}
            convertBstToVine(root,count);
        }
        else{
            ++count[0];
            if(count[0] == 1){returnedRoot = root;}
            //System.out.println("gec ");
            preNode = root;
            convertBstToVine(root.right,count);
        }
    }
        return returnedRoot;
}
```

Best case :O(logn),

Worst case : O(n). Entire binary search tree' structure will be vine.

Q3:



```
/**
 * We create a new node with the item we want to add, and then we traverse the list to find the node that should be
 * before the new node, and then we add the new node to the list
 *
 * @param item the item to be added
 * @return The boolean value of true.
 */
1 usage
public boolean add(E item){
    size++;
    if(size > maxCap){

        System.out.print(this);
        System.out.println();
        levelUp();
        maxCap = computeMaxCap(maxLevel);
    }
    Entry<E>[] pred = search(item);
    Entry<E> newNode = new Entry<>(randomize(item), item);

    for(int i = 0; i < newNode.links.length; i++){
        newNode.links[i] = pred[i].links[i];
        pred[i].links[i] = newNode;
    }
    return true;
}
```

Search : T(n) = O(logn).

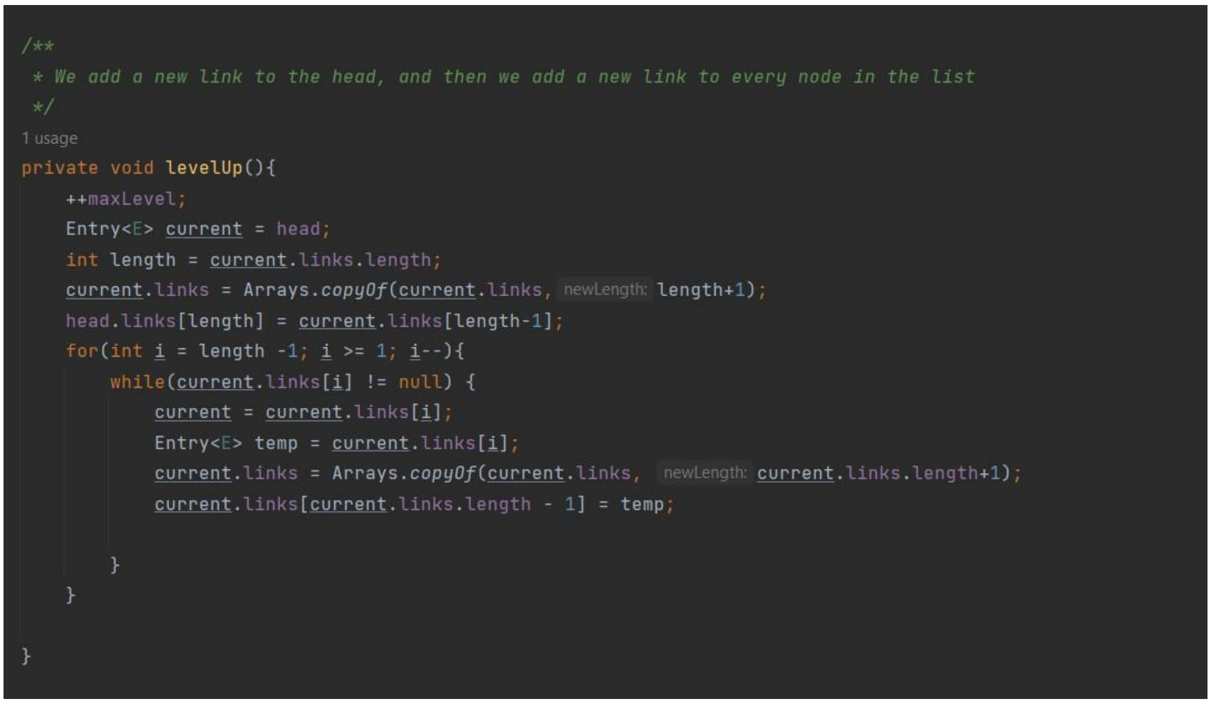
Level Up : T(n) = O(logn)

ComputeMaxCap : T(n) = O(1).

Randomize: T(n) = average O(n).

Adding element to the Skip List takes O(logn).

Extra Memory takes O(n).

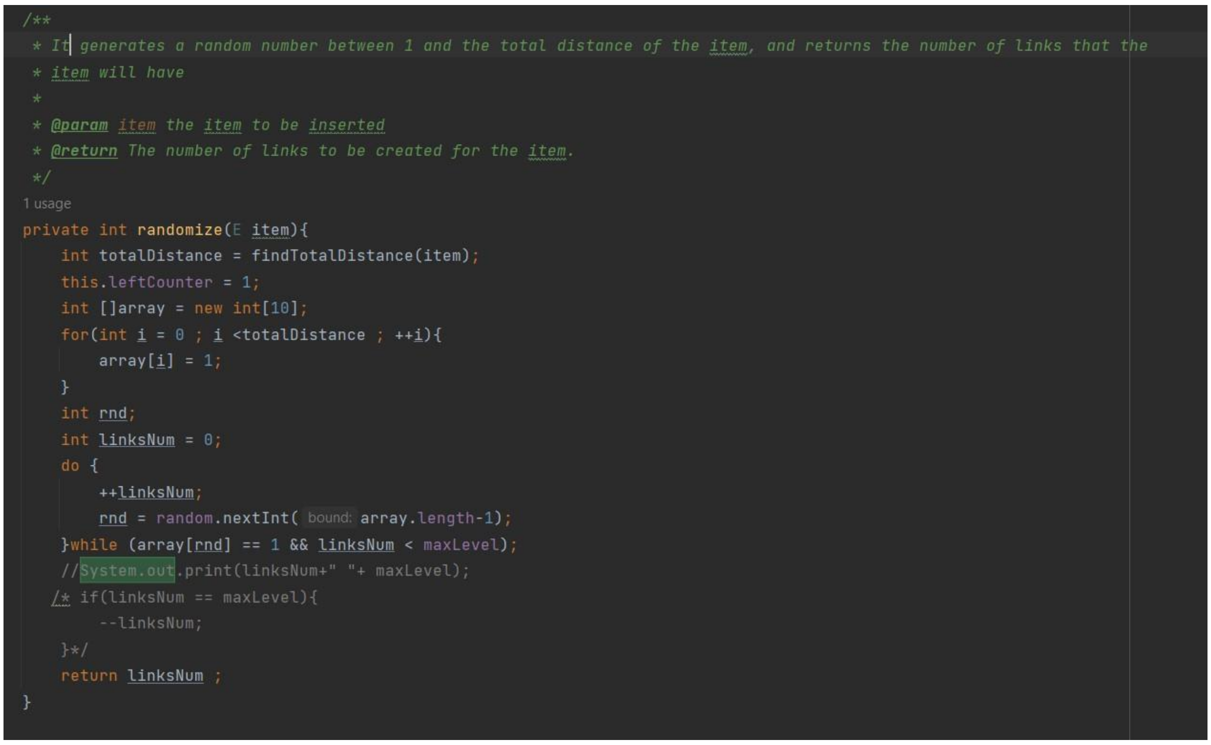


```java
/**
 * We add a new link to the head, and then we add a new link to every node in the list
 */
1 usage
private void levelUp(){
    ++maxLevel;
    Entry<E> current = head;
    int length = current.links.length;
    current.links = Arrays.copyOf(current.links, newLength: length+1);
    head.links[length] = current.links[length-1];
    for(int i = length -1; i >= 1; i--){
        while(current.links[i] != null) {
            current = current.links[i];
            Entry<E> temp = current.links[i];
            current.links = Arrays.copyOf(current.links, newLength: current.links.length+1);
            current.links[current.links.length - 1] = temp;

        }
    }

}
```

Copying links and update links number takes (Level of number x logn)

Level of number is constant . So it can be ignored.

So the Method takes :T(n) = O(logn).

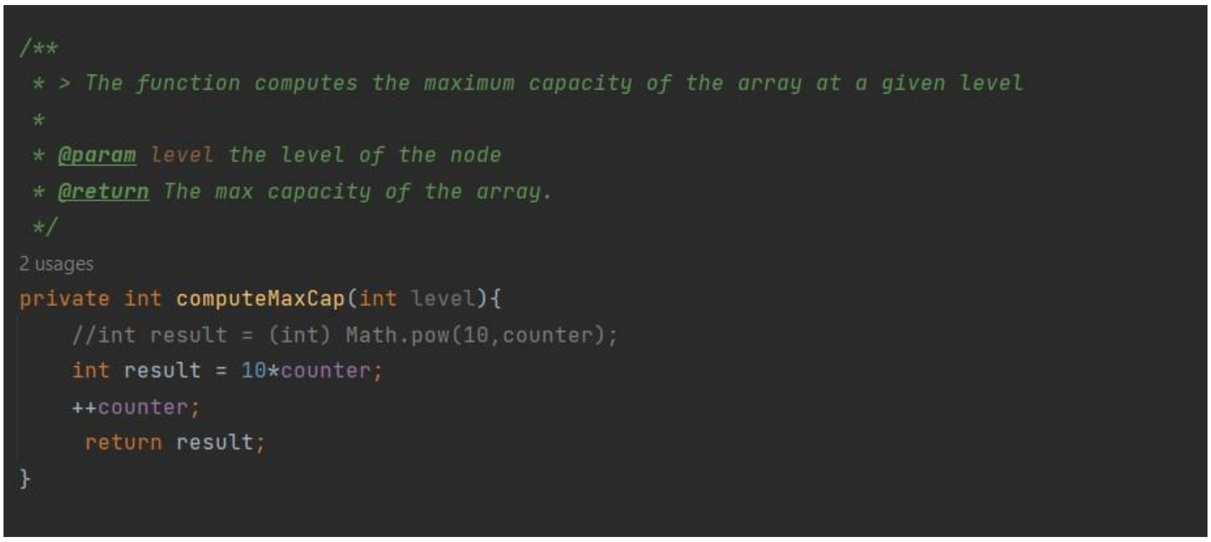


```
/**
 * It generates a random number between 1 and the total distance of the item, and returns the number of links that the
 * item will have
 *
 * @param item the item to be inserted
 * @return The number of links to be created for the item.
 */
1 usage
private int randomize(E item){
    int totalDistance = findTotalDistance(item);
    this.leftCounter = 1;
    int []array = new int[10];
    for(int i = 0 ; i <totalDistance ; ++i){
        array[i] = 1;
    }
    int rnd;
    int linksNum = 0;
    do {
        ++linksNum;
        rnd = random.nextInt( bound: array.length-1);
    }while (array[rnd] == 1 && linksNum < maxLevel);
    //System.out.print(linksNum+" "+ maxLevel);
   /* if(linksNum == maxLevel){
        --linksNum;
    }*/
    return linksNum ;
}
```

FindingTotalDistance to the right tall is O(n).

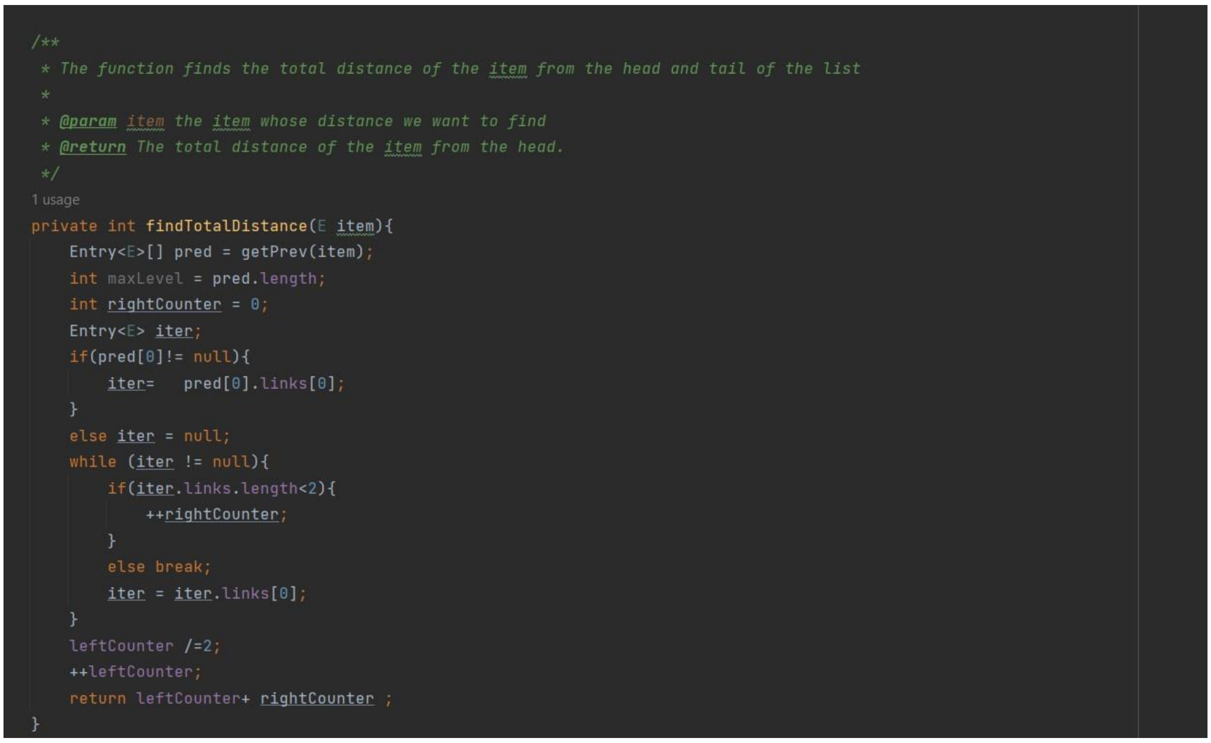
FindingTotalDistance to the right tall is amortised O(1).

So T(n) = O(n).



```
/**
 * > The function computes the maximum capacity of the array at a given level
 *
 * @param level the level of the node
 * @return The max capacity of the array.
 */
2 usages
private int computeMaxCap(int level){
    //int result = (int) Math.pow(10,counter);
    int result = 10*counter;
    ++counter;
     return result;
}
```

Compute max takes O(1).


```
/**
 * The function finds the total distance of the item from the head and tail of the list
 *
 * @param item the item whose distance we want to find
 * @return The total distance of the item from the head.
 */
1 usage
private int findTotalDistance(E item){
    Entry<E>[] pred = getPrev(item);
    int maxLevel = pred.length;
    int rightCounter = 0;
    Entry<E> iter;
    if(pred[0]!= null){
        iter=   pred[0].links[0];
    }
    else iter = null;
    while (iter != null){
        if(iter.links.length<2){
            ++rightCounter;
        }
        else break;
        iter = iter.links[0];
    }
    leftCounter /=2;
    ++leftCounter;
    return leftCounter+ rightCounter ;
}
```

FindingTotalDistance to the right tall is O(n).

FindingTotalDistance to the right tall is amortised O(1).

Finding total distance takes O(n).