GIT Department of Computer Engineering
CSE 222/505 - Spring 2021
Homework 7 Report

CANBERK ARICI
171044062

System Requirements

1. Binary Tree -

A binary tree is a tree-type non-linear data structure with a maximum of two children for each parent. Every node in a binary tree has a left and right reference along with the data element. The node at the top of the hierarchy of a tree is called the root node. The nodes that hold other sub-nodes are the parent nodes.

Terminologies associated with Binary Trees and Types of Trees -

- **Node**: It represents a termination point in a tree.
- **Root**: A tree’s topmost node.
- **Parent**: Each node (apart from the root) in a tree that has at least one sub-node of its own is called a parent node.
- **Child**: A node that straightway came from a parent node when moving away from the root is the child node.
- **Leaf Node**: These are external nodes. They are the nodes that have no child.
- **Internal Node**: As the name suggests, these are inner nodes with at least one child.
- **Depth of a Tree**: The number of edges from the tree’s node to the root is.
- **Height of a Tree**: It is the number of edges from the node to the deepest leaf. The tree height is also considered the root height.

2. Binary Search Tree -

A binary search tree is a tree (data structure) that either has one child, two children(left child and right child) or no child at all. It is a data structure that is used to store the piece of data in a sorted manner and results in efficient searching. The famous Binary Search Algorithm is actually the in order traversal of a binary search tree and so, it derives its name from the Algorithm.

The most important property of a BST that identifies it is:-

- The left child of the tree should be less than the root and the right child of the tree should be greater than or equal to the root! That means the extreme leftest leaf node will be the smallest number or data that tree has and the extreme rightist leaf node will have the greatest/the largest data of the tree.

- The subtrees should also follow the leaf order property that means The right child of the left subtree cannot be greater than the root and the left child of the right subtree cannot be lesser than the root.

3. AVL Tree -

AVL trees are height balancing binary search trees. AVL tree checks the height of the left and the right subtrees and assures that the difference is not more than 1. This difference is called the Balance Factor.BalanceFactor = height(left-subtree) − height(right-subtree)
If the difference in the height of left and right subtrees is more than 1, the tree is balanced using some rotation techniques.

AVL Rotations
To balance itself, an AVL tree may perform the following four kinds of rotations −

1. Left rotation
2. Right rotation
3. Left-Right rotation
4. Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2.

4. Red-Black Tree -

Rudolf Bayer developed the Red–Black tree as a special case of his B-tree Leo Guibas and Robert Sedgewick refined the concept and introduced the color convention.

A Red–Black tree maintains the following invariants:

1. A node is either red or black.
2. The root is always black.
3. A red node always has black children. (A null reference is considered to refer to a black node.)
4. The number of black nodes in any path from the root to a leaf is the same.

5. 2-3 Tree -

A 2-3 Tree is a tree data structure where every node with children has either two children and one data element or three children and two data elements. A node with 2 children is called a 2-NODE and a node with 3 children is called a 3-NODE.

Nodes on the outside of the tree i.e. the leaves have no children and have either one or two data elements. All its leaves must be on the same level so that a 2-3 tree is always height balanced. A 2-3 tree is a special case of a B-Tree of order 3. Below is an example of a 2-3 tree.

Properties of 2-3 Trees

- Every internal node in the tree is a 2-node or a 3-node i.e it has either one value or two values.
- A node with one value is either a leaf node or has exactly two children. Values in left subtree < value in node < values in right subtree.
- A node with two values is either a leaf node or has exactly 3 children. It cannot have 2 children. Values in left subtree < first value in node < values in middle subtree < second value in node < value in right subtree.
- All leaf nodes are at the same level.

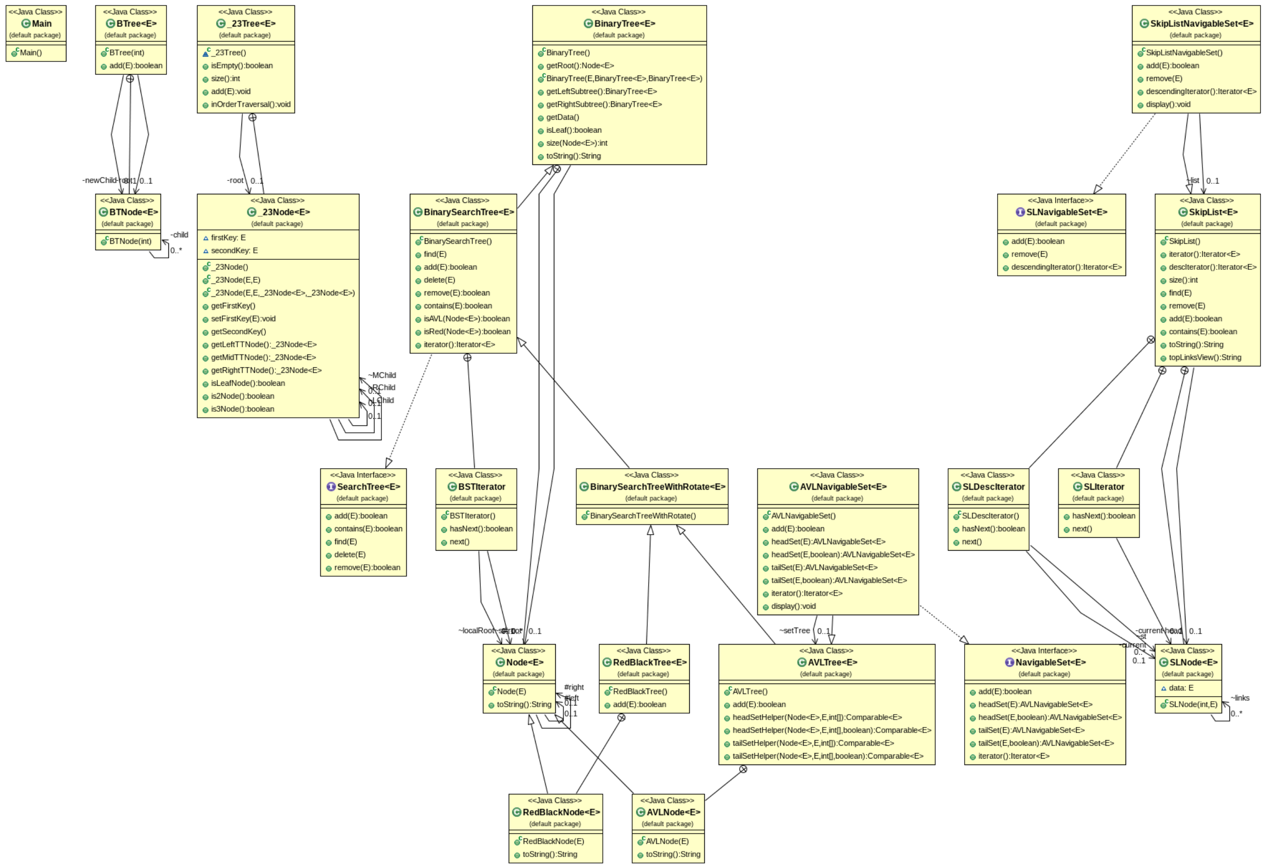
6. B-Tree -

In the 2–3 tree, a 2-node has two children and a 3-node has three children. In the B-tree, the maximum number of children is the order of the B-tree, which we will represent by the variable order . Other than the root, each node has between order /2 and order children. The number of data items in a node is 1 less than the number of children. The data items in each node are in increasing order. B-trees were developed to store indexes to databases on disk storage. Disk storage is broken into blocks, and the nodes of a B-tree are sized to fit in a block, so each disk access to the index retrieves exactly one B-tree node. The time to retrieve a block is large compared to the time required to process it in memory, so by making the tree nodes as large as possible, we reduce the number of disk accesses required to find an item in the index. Assuming a block can store a node for a B-tree of order 200, each node would store at least 100 items. This would enable 100 4 or 100 million items to be accessed in a B-tree of height 4.

7. Skip List -

The skip-list is another data structure that can be used as the basis for the NavigableSet or NavigableMap and as a substitute for a balanced tree. Like a balanced tree, it provides for ***O(log n) search, insert, and remove***. It has the additional advantage over the Red–Black tree-based TreeSet in that concurrent references are easier to achieve. With the TreeSet class, if two threads have iterators to the set and one thread makes a modification to the set, then the iterators are invalid and will throw the ConcurrentModificationException when next referenced. A skip-list is a list of lists. Each node in a list contains a data element with a key, and the elements in each list are in increasing order by key. Unlike the usual list node, which has a single forward link to the next node, the nodes in a skip list have a varying number of forward links. The number of such links is determined by the level of a node. A level-m node has m forward links. When a new data element is inserted in a skip-list, a new node is inserted to store the element. The Node's level is chosen randomly in such a way that approximately 50 percent are level 1 (one forward link), 25 percent are level 2 (two forward links), 12.5 percent are level 3, and so on.

CLASS DIAGRAM



Solution Approach Part 1

a.NavigableSet using SkipList methods

```
abstract interface SLNavigableSet<E extends Comparable<E>> {
   public boolean add(E item);

   public E remove(E item);

   public Iterator<E> descendingIterator();
}

public class SkipListNavigableSet<E extends Comparable<E>> extends
SkipList<E> implements SLNavigableSet<E> {

   SkipList<E> list = new SkipList<E>();

   public boolean add(E item) {
       boolean added = list.add(item);
       return added;
   }

   public E remove(E item) {
       E removed = list.remove(item);
       return removed;
   }

   public Iterator<E> descendingIterator() {
       return list.descIterator();
   }

   public void display(){
       System.out.println(list.toString());
   }

}
```

a) ***Add method*** - I added the given item as parameter to the SkipList we Created for the Set. **O(log2(N))**
b) ***Remove method*** - I removed the given item as a parameter to the SkipList we Created for the Set. **O(log2(N))**
c) ***Descending Iterator*** - I take the Elements in the SkipList and add them to a stack and then Override the Iterators methods by checking stack size and popping the Stack top to get the elements in Descending Order **O(N).**

```
private class SLDescIterator implements Iterator<E> {

       private SLNode<E> current = head;
       Stack<SLNode<E>> st;

       public SLDescIterator() {
            st = new Stack<SLNode<E>>();
            System.out.println();
            while (current != null) {
                st.push(current);
                current = current.links[0];
            }
       }

       @Override
       public boolean hasNext() {
            return (st.size() > 1);
       }

       @Override
       public E next() {
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
            SLNode<E> curr = st.pop();
            return curr.data;
       }
   }
```

b. NavigableSet using AVL methods

```
// Interface
abstract interface NavigableSet<E extends Comparable<E>> {

   public boolean add(E item);

   public AVLNavigableSet<E> headSet(E item);

   public AVLNavigableSet<E> headSet(E item, boolean inclusive);

   public AVLNavigableSet<E> tailSet(E item);

   public AVLNavigableSet<E> tailSet(E item, boolean inclusive);

   public Iterator<E> iterator();
}

public class AVLNavigableSet<E extends Comparable<E>> extends
AVLTree<E> implements NavigableSet<E> {

   // Tree for the Navigable Set Implementation
   AVLTree<E> setTree = new AVLTree<E>();

   // adds the item to the tree
   public boolean add(E item) {
       boolean added = setTree.add(item);
       return added;
   }

   // Gives the HeadSet -- elements smaller than the item
   public AVLNavigableSet<E> headSet(E item) {
       int[] sz = new int[1];
       Comparable<E>[] array =
setTree.headSetHelper(setTree.root, item, sz);

       AVLNavigableSet<E> newSet = new AVLNavigableSet<E>();

       for (int i = 0; i < sz[0]; i++) {
           newSet.add((E) (array[i]));
       }
```

```
        return newSet;
    }

    // Gives the HeadSet -- elements smaller or equal to than the
item
    public AVLNavigableSet<E> headSet(E item, boolean inclusive) {
        int[] sz = new int[1];
        Comparable<E>[] array;

        if (!inclusive) {
            array = setTree.headSetHelper(setTree.root, item, sz);
        } else {
            array = setTree.headSetHelper(setTree.root, item, sz,
inclusive);
        }

        AVLNavigableSet<E> newSet = new AVLNavigableSet<E>();

        for (int i = 0; i < sz[0]; i++) {
            newSet.add((E) (array[i]));
        }

        return newSet;
    }

    // Gives the Tailset -- elements greater  to than the item

    public AVLNavigableSet<E> tailSet(E item) {
        int[] sz = new int[1];
        Comparable<E>[] array =
setTree.tailSetHelper(setTree.root, item, sz);

        AVLNavigableSet<E> newSet = new AVLNavigableSet<E>();

        for (int i = 0; i < sz[0]; i++) {
            newSet.add((E) (array[i]));
        }

        return newSet;
    }

    // Gives the Tailset -- elements greater or equal to than the
```

```
item

    public AVLNavigableSet<E> tailSet(E item, boolean inclusive) {
        int[] sz = new int[1];
        Comparable<E>[] array;

        if (!inclusive) {
            array = setTree.tailSetHelper(setTree.root, item, sz);
        } else {
            array = setTree.tailSetHelper(setTree.root, item, sz,
inclusive);
        }

        AVLNavigableSet<E> newSet = new AVLNavigableSet<E>();

        for (int i = 0; i < sz[0]; i++) {
            newSet.add((E) (array[i]));
        }

        return newSet;
    }

    public Iterator<E> iterator() {
        return setTree.iterator();
    }

    public void display() {
        System.out.println(setTree.toString());
    }

}
```

a) ***Add method*** - We add the given item as parameter to the AVL-Tree we Created for the Set. **O(log2(N))**
b) ***HeadSet*** - We Implemented the Headset and HeadSet Inclusive. **O((N)) as we traverse the Tree to Find Elements.**

c) ***TailSet*** - We Implemented the TailSet and TailSet Inclusive. **O((N)) as we traverse the Tree to FInd Elements**.

- ***SINCE THE INORDER TRAVERSAL OF THE SEARCH TREES GIVES A SORTED ASCENDING ORDER WE CAN TRAVERSE THE TREE IN THAT ORDER TO GET THE ELEMENTS WHICH ARE SMALLER [HEADSET] , SMALLER EQUAL TO [HEADSET (INCLUSIVE)] , GREATER [TAILSET] and GREATER THAN EQUAL TO [TAILSET (INCLUSIVE)] ...***

```
private void getHeadSet(Node<E> localRoot, Comparable<E>[] array, int[]
idx, E item) {
      // Similarly for the Tail Set Function
      // Function is Overloaded with a Boolean value if passed

   if (localRoot == null) {
     return;
   }
   getHeadSet(localRoot.left, array, idx, item);

      // < HeadSEt
      // <= HeadSet Inclusive
      // > TailSet
      // >= TailSet Inclusive
                                                 // Important
   if (localRoot.data.compareTo(item) < 0) {
     array[idx[0]] = localRoot.data;
     idx[0]++;
   }



   getHeadSet(localRoot.right, array, idx, item);
 }

      // Returns the Array from which a new Tree can be Created
```

```java
public Comparable<E>[] headSetHelper(Node<E> localRoot, E item, int[]
sz) {
   int treeSize = size(localRoot);
   Comparable<E>[] array = (Comparable<E>[]) new Comparable[treeSize +
2];

   int idx[] = new int[1];
   idx[0] = 0;
   getHeadSet(localRoot, array, idx, item);

   sz[0] = idx[0];
   return array;
 }
```

d) ***Iterator*** - Iterator traverses the BST in sorted order(increasing). I have implemented the iterator using a stack data structure. **Operations are O(1)**

```java
// Iterator class for BST iteration
 private class BSTIterator implements Iterator<E> {
   Node<E> localRoot = root;
   Stack<Node<E>> st;

   public BSTIterator() {

     // Initialises the Stack
     st = new Stack<Node<E>>();

     //Pushes all the left nodes to the BST

     while (localRoot != null) {
       st.push(localRoot);
       localRoot = localRoot.left;
     }

   }

   @Override
   public boolean hasNext() {
```

```java
        return !st.isEmpty();
    }

    @Override
    public E next() {
        // Pops the current NOde and returns its data
        Node<E> node = st.pop();
        E data_popped = node.data;

        // if it has right subtree go to that and push all its left nodes
to the stack as well....
        if (node.right != null) {
            node = node.right;
            while (node != null) {
                st.push(node);
                node = node.left;
            }
        }
        return data_popped;
    }
}
```

Solution Approach Part 2

c. Checking if a given BST is Height Balanced Like AVL Tree

We Check if the Balance factor at any node > 1 or < -1 then we say it's not a AVL balanced BST else it's a AVL Balanced BST.
Time Complexity = O(n) as we check heights and for each node we do an O(1) operation checking if it's balanced..

```
// Return the Height of the Tree
private int heightofBST(Node<E> root) {

   int leftST_height, rightST_height;

   if (root == null) {
     return 0;
   }

   leftST_height = heightofBST(root.left);
   rightST_height = heightofBST(root.right);

   if (leftST_height > rightST_height) {
     return (1 + leftST_height);
   } else {
     return (1 + rightST_height);
   }

 }
      // Check sif the Tree is AVL balanced or not
 public boolean isAVL(Node<E> root) {
   int leftST_height, rightST_height;

   if (root == null) {
     return true;
   }
```

```
    // calculates left and right height
  leftST_height = heightofBST(root.left);
  rightST_height = heightofBST(root.right);

    // absolite difference
  int difference = Math.abs(leftST_height - rightST_height);

  if (difference <= 1 && isAVL(root.left) && isAVL(root.right)) {
    return true;
  }
  return false;
}
```

d. Checking if a given BST is Height Balanced Like Red Black Tree (isRed)

In a Red-Black Tree, the maximum height of a node is at most twice the minimum height (The four Red-Black tree properties make sure this is always followed). We check if for every node, length of the longest leaf to node path has not more than twice the nodes on shortest path from node to leaf.

" maxHeight <= 2 * minHeight at each node "

Time Complexity = O(N) as we traverse the Tree and check all nodes and left and right subtree heights..

```
/*
  * Checks Red Black Balancing of Tree
  * For every node, length of the longest leaf to node path has not more than twice the nodes on shortest path from node to leaf.
  */
```

```java
private boolean isRedBlackBalancedUtil(Node<E> root, int[] maxHeight,
int[] minHeight) {

   // Base case
   if (root == null) {
     maxHeight[0] = minHeight[0] = 0;
     return true;
   }

   // To store max and min heights of left subtree
   int[] leftST_maxHeight = new int[1];
   int[] leftST_minHeight = new int[1];
   leftST_maxHeight[0] = 0;
   leftST_minHeight[0] = 0;

   // To store max and min heights of right subtree
   int[] rightST_minHeight = new int[1];
   int[] rightST_maxHeight = new int[1];
   rightST_maxHeight[0] = 0;
   rightST_minHeight[0] = 0;

   // Check if left subtree is balanced,
   // also set leftST_maxHeight and leftST_minHeight
   if (!isRedBlackBalancedUtil(root.left, leftST_maxHeight,
leftST_minHeight))
     return false;

   // Check if right subtree is balanced,
   // also set rightST_maxHeight and rightST_minHeight
   if (!isRedBlackBalancedUtil(root.right, rightST_maxHeight,
rightST_minHeight))
     return false;

   // Set the max and min heights
   // of this node for the parent call

   // System.out.println(leftST_minHeight + " " + rightST_minHeight);
   maxHeight[0] = Math.max(leftST_maxHeight[0], rightST_maxHeight[0]) +
1;
```

```java
    minHeight[0] = Math.min(leftST_minHeight[0], rightST_minHeight[0]) +
1;

    // See if this node is Red Black balanced
    if (maxHeight[0] <= 2 * minHeight[0])
      return true;

    return false;
  }
  /*
   * Returns if root node is RED.
   * Checks the Height Balanced Properites of Red Black Tree
   *
   */
  public boolean isRed(Node<E> root) {
    int[] maxHeight = new int[1];
    int[] minHeight = new int[1];
    maxHeight[0] = 0;
    minHeight[0] = 0;
    return isRedBlackBalancedUtil(root, maxHeight, minHeight);
  }
```

Test Cases

“ PART-1 ”

Test Case#	Description
1	Descending Iteration (Skip List) NavigableSet and Removal from NavigableSet

```
/* A. NavigableSet using SkipList */

	SkipListNavigableSet<Integer> ns1 = new
SkipListNavigableSet<Integer>();

	/* Add Method */
	int p1_arr1[] = new int[10];

	for (int i = 0; i < 10; i++) {
	    p1_arr1[i] = rand.nextInt(1000000);
	    ns1.add(p1_arr1[i]);
	}

	System.out.println("Navigable Set 1 : ");
	ns1.display();
	System.out.println();

	System.out.println("Descending Iteration Navigable Set 1 :
");
	Iterator<Integer> ns1_itr1 = ns1.descendingIterator();

	while (ns1_itr1.hasNext()) {
	    System.out.println("Data: " + ns1_itr1.next());
	}
	System.out.println();
```

```java
        for (int i = 3; i < 6; i++) {
            System.out.println("To Remove : " + p1_arr1[i] + " <->
Removed :" + ns1.remove(p1_arr1[i]));
        }

        System.out.println("To Remove : " + (-30000) + " <->
Removed :"
                + (ns1.remove(-30000) == null ? "Item Not Present"
: ns1.remove(-30000)));

        System.out.println();
        System.out.println("Navigable Set 1 after Removal : ");
        ns1.display();
        System.out.println();
```

Output -

```
Navigable Set 1 :
[153707, 259354, 415019, 503613, 568455, 692695, 788023, 791506, 869482, 994312]

Descending Iteration Navigable Set 1 :

Data: 994312
Data: 869482
Data: 791506
Data: 788023
Data: 692695
Data: 568455
Data: 503613
Data: 415019
Data: 259354
Data: 153707

To Remove : 791506 <-> Removed :791506
To Remove : 503613 <-> Removed :503613
To Remove : 415019 <-> Removed :415019
To Remove : -30000 <-> Removed :Item Not Present

Navigable Set 1 after Removal :
[153707, 259354, 568455, 692695, 788023, 869482, 994312]
```

Test Case	Description
2	Headset, Tailset, Headset Inclusive, Tailset Inclusive, Iterating NavigableSet, Iterating Headset of Navigable Set

```
      /* B. NavigableSet using AVLTrees */

       AVLNavigableSet<Integer> ns2 = new AVLNavigableSet<Integer>();

       /* Add Method */
       int p1_arr2[] = new int[10];

       for (int i = 0; i < 10; i++) {
           p1_arr2[i] = rand.nextInt(1000000);
           ns2.add(p1_arr2[i]);
       }

       /* Inorder Traversal Should give a Sorted Set */
       System.out.println("Navigable Set 2 : ");
       ns2.display();

       AVLNavigableSet<Integer> head = ns2.headSet(p1_arr2[4]);

       System.out.println();
       System.out.println("HeadSet of Value " + p1_arr2[4] + " is :\
n");
       head.display();
       System.out.println();

       AVLNavigableSet<Integer> tail = ns2.tailSet(p1_arr2[4]);

       System.out.println("TailSet of Value " + p1_arr2[4] + " is :\
n");
       tail.display();
       System.out.println();

       AVLNavigableSet<Integer> headinc = ns2.headSet(p1_arr2[4],
```

```
true);

        System.out.println("HeadSet Inclusive of Value " + p1_arr2[4] +
" is :\n");
        headinc.display();
        System.out.println();

        AVLNavigableSet<Integer> tailinc = ns2.tailSet(p1_arr2[4],
true);

        System.out.println("TailSet Inclusive of Value " + p1_arr2[4] +
" is :\n");
        tailinc.display();
        System.out.println();

        System.out.println("Iterating Navigable Set 2 : ");

        Iterator<Integer> ns2_itr1 = ns2.iterator();

        while (ns2_itr1.hasNext()) {
            System.out.println("Data: " + ns2_itr1.next());
        }
        System.out.println();

        System.out.println("Iterating HeadSet of Navigable Set : ");

        Iterator<Integer> ns2_itr2 = head.iterator();

        while (ns2_itr2.hasNext()) {
            System.out.println("Data: " + ns2_itr2.next());
        }
        System.out.println();
```

Output: ***Note - Bal : Balance of AVL trees node***

```
Navigable Set 2 :
[[Bal: 1, Item: 48446]
, [Bal: 0, Item: 255108]
, [Bal: 0, Item: 272415]
, [Bal: 0, Item: 310322]
, [Bal: -1, Item: 404964]
, [Bal: -1, Item: 440849]
, [Bal: 0, Item: 658636]
, [Bal: 0, Item: 683742]
, [Bal: 0, Item: 849395]
, [Bal: 0, Item: 983244]
, ]

HeadSet of Value 683742 is :

[[Bal: 0, Item: 48446]
, [Bal: 0, Item: 255108]
, [Bal: 0, Item: 272415]
, [Bal: 0, Item: 310322]
, [Bal: 0, Item: 404964]
, [Bal: 0, Item: 440849]
, [Bal: 0, Item: 658636]
, ]

TailSet of Value 683742 is :

[[Bal: 1, Item: 849395]
, [Bal: 0, Item: 983244]
, ]
```

```
HeadSet Inclusive of Value 683742 is :

[[Bal: 0, Item: 48446]
, [Bal: 0, Item: 255108]
, [Bal: 0, Item: 272415]
, [Bal: 1, Item: 310322]
, [Bal: 0, Item: 404964]
, [Bal: 1, Item: 440849]
, [Bal: 1, Item: 658636]
, [Bal: 0, Item: 683742]
, ]

TailSet Inclusive of Value 683742 is :

[[Bal: 0, Item: 683742]
, [Bal: 0, Item: 849395]
, [Bal: 0, Item: 983244]
, ]

Iterating Navigable Set 2 :
Data: 48446
Data: 255108
Data: 272415
Data: 310322
Data: 404964
Data: 440849
Data: 658636
Data: 683742
Data: 849395
Data: 983244
```

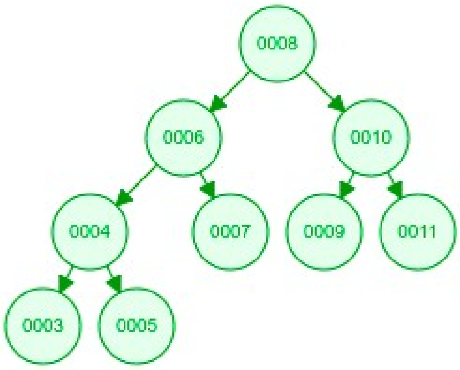
```
Iterating HeadSet of Navigable Set :
Data: 48446
Data: 255108
Data: 272415
Data: 310322
Data: 404964
Data: 440849
Data: 658636
```

“ PART-2 ”

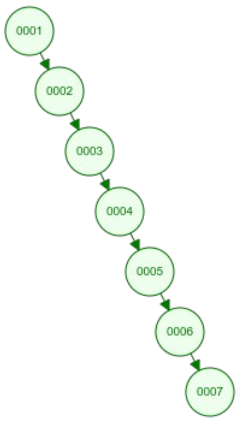
Test Case	Description
3	Check whether tree is AVL

```
/*
         * A. Checking if a given BST is an AVL Balanced BST or Not
         */

        int[] p2_arr1 = { 8, 6, 10, 4, 7, 9, 11, 3, 5 };

        BinarySearchTree<Integer> p2_bst1 = new
BinarySearchTree<Integer>();

        for (int i = 0; i < 9; i++) {
            p2_bst1.add(p2_arr1[i]);
        }

        System.out.println("Binary Search Tree 1 : ");
        System.out.println(p2_bst1.toString());
        System.out.println();
        System.out.println("Checking if Its AVL Balanced Tree : ");
        System.out.println(p2_bst1.isAVL(p2_bst1.getRoot()));
        System.out.println();

        // -----------------------------------------------------//

        int[] p2_arr2 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

        BinarySearchTree<Integer> p2_bst2 = new
BinarySearchTree<Integer>();

        for (int i = 0; i < 10; i++) {
            p2_bst2.add(p2_arr2[i]);
        }

        System.out.println("Binary Search Tree 2 : ");
        System.out.println(p2_bst2.toString());
```

```
System.out.println();
System.out.println("Checking if Its AVL Balanced Tree : ");
System.out.println(p2_bst2.isAVL(p2_bst2.getRoot()));
System.out.println();
```



BST1 - Height Balanced



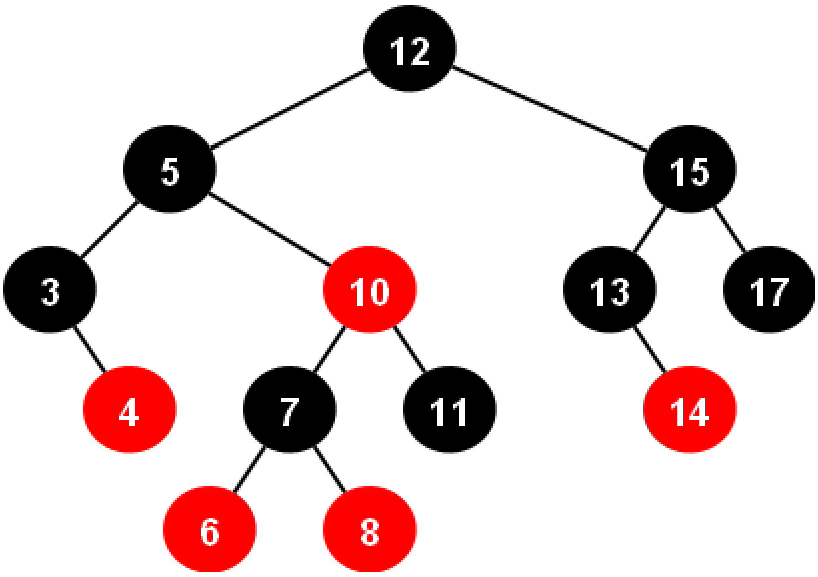
BST2 - Not Height Balanced

```
Binary Search Tree 1 :
[3, 4, 5, 6, 7, 8, 9, 10, 11, ]

Checking if Its AVL Balanced Tree :
true

Binary Search Tree 2 :
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ]

Checking if Its AVL Balanced Tree :
false
```

Test Case	Description
4	Check if a tree is Red Black Tree

```java
/*
         * A. Checking if a given BST has Properties of RedBlack tree
with Root as Red
         */

        int[] p2_arr3 = { 12, 5, 15, 3, 10, 13, 17, 4, 7, 11, 14, 6,
8 };

        BinarySearchTree<Integer> p2_bst3 = new
BinarySearchTree<Integer>();

        for (int i = 0; i < 13; i++) {
            p2_bst3.add(p2_arr3[i]);
        }

        System.out.println("Binary Search Tree 3 : ");
        System.out.println(p2_bst3.toString());
        System.out.println();
        System.out.println("Checking if Its a Red Black [Balanced]
Tree : ");
        System.out.println(p2_bst3.isRed(p2_bst3.getRoot()));
        System.out.println();
```

```
        int[] p2_arr4 = { 10, 5, 100, 50, 40, 150, 32, 1, 160, 170,
200 };

        BinarySearchTree<Integer> p2_bst4 = new
BinarySearchTree<Integer>();

        for (int i = 0; i < 11; i++) {
            p2_bst4.add(p2_arr4[i]);
        }

        System.out.println("Binary Search Tree 4 : ");
        System.out.println(p2_bst4.toString());
        System.out.println();
        System.out.println("Checking if Its a Red Black [Balanced]
Tree : ");
        System.out.println(p2_bst4.isRed(p2_bst4.getRoot()));
        System.out.println();
```



Output:

```
Binary Search Tree 3 :
[3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 15, 17, ]

Checking if Its a Red Black [Balanced] Tree :
true

Binary Search Tree 4 :
[1, 5, 10, 32, 40, 50, 100, 150, 160, 170, 200, ]

Checking if Its a Red Black [Balanced] Tree :
false
```

Test Case	Description
5	Insertion numbers as many as entered number

```
Scanner scan = new Scanner(System.in);

        System.out.print("Enter the number of Integers to Insert : ");
        int numTimes = scan.nextInt();

        System.out.println("\n\n\t\t\t\tNumber to be Inserted are " +
numTimes + "\n");

        long[][] stats = new long[5][2]; // store the stats of all 5
data structures
        int ds = 0;
```

```
        // Binary Search Trees
        int timer = 0;
        while (timer++ < 1) {
            int counter = 0;

            BinarySearchTree<Integer> _dataStruct = new
BinarySearchTree<Integer>();

            Set<Integer> set = new LinkedHashSet<Integer>();
            while (set.size() < numTimes) {
                set.add(rand.nextInt(1000000));
            }

            int p3_arr1[] = new int[numTimes];
            int k = 0;
            Iterator<Integer> itr = set.iterator();
            while (itr.hasNext()) {
                p3_arr1[k++] = itr.next();
            }

            long startTime = System.nanoTime();
            // adding num times
            for (int i = 0; i < numTimes; i++) {
                _dataStruct.add(p3_arr1[i]);
            }
            long endTime = System.nanoTime();
            stats[ds][counter++] = (endTime - startTime);

            Set<Integer> set1 = new LinkedHashSet<Integer>();
            while (set1.size() < 100) {
                // Will be distinct from previous set
                set1.add(rand.nextInt(100000) + 1000000);
            }

            int p3_arr2[] = new int[100];
            k = 0;
            Iterator<Integer> itr2 = set1.iterator();
            while (itr2.hasNext()) {
                p3_arr2[k++] = itr2.next();
            }

            startTime = System.nanoTime();
```

```java
            // Adding 100
            for (int i = 0; i < 100; i++) {
                _dataStruct.add(p3_arr2[i]);
            }

            endTime = System.nanoTime();

            stats[ds++][counter++] = (endTime - startTime);
        }
```

Similarly Same Code is applied to all other 4 Data Structures.

```java
System.out.println("(ns)\tTime\t  Extra Time\n");
        for (int i = 0; i < 5; i++) {
            if (i == 0) {
                System.out.print("BST \t");
            } else if (i == 1) {
                System.out.print("RBT \t");
            } else if (i == 2) {
                System.out.print("23T \t");
            } else if (i == 3) {
                System.out.print("BT \t");
            } else if (i == 4) {
                System.out.print("SL \t");
            }
            System.out.println(stats[i][0] + "    " + stats[i][1]);
        }

        System.out.println("\nSimulation Completed !!");
```

Stats in the Form of a Table

	10K integers									
	BST		RBT		23T		BT		SL	
NanoSeconds	Time	ExtraTime	Time	ExtraTime	Time	ExtraTime	Time	ExtraTime	Time	ExtraTime
1	11856696	79822	9861388	93039	11292716	35985	9831597	44063	15851370	61226
2	7698243	84282	9359942	111894	11436734	34443	9763450	40229	17323304	97256
3	7531418	85026	9003682	143890	15266404	67449	22585824	67299	17600467	77448
4	7761482	73423	8682211	91749	10655253	34725	9082933	38390	18524081	79096
5	7971076	65489	8935225	88559	11114900	35945	9791863	35809	17698223	86351
6	7935743	151173	8709052	138215	15294443	50897	13806877	64476	21857849	87692
7	7921227	68972	9061990	90313	10442517	36334	9574727	49000	18188154	81094
8	7709428	81276	9122758	87681	10771092	36907	9409325	38610	17811491	85975
9	8616770	43709	8375646	90732	11731157	37874	9110339	37263	16453167	63074
10	7850035	81291	8685097	93548	13215582	40505	8281653	42597	16905818	75869
Average Running Time	8285211.8	81446.3	8979699.1	102962	12122079.8	41106.4	11123858.8	45773.6	17821392.4	79508.1

	20K Integers									
	BST		RBT		23T		BT		SL	
NanoSeconds	Time	ExtraTime	Time	ExtraTime	Time	ExtraTime	Time	ExtraTime	Time	ExtraTime
1	13674966	84343	19913307	65445	16139071	77386	13034475	52020	22899399	53720
2	13279944	70662	15531558	63346	15041520	67531	13305407	63301	22959534	53404
3	14103427	83172	14180053	44090	11691757	50285	11719065	56970	20859830	55445
4	13481626	79669	15776374	65045	14925238	87043	11325503	54387	22688519	47650
5	13884445	82388	15820778	59688	14324857	55868	11243260	48343	21941321	56214
6	23339080	108879	23207862	44995	11311811	52422	11551074	41689	29173775	103679
7	14478056	99059	23349633	75321	12366613	48489	12168042	45268	29430297	142249
8	13891555	71173	17669385	56317	22884934	57203	17946230	54343	30560893	78961
9	13548207	75489	15569801	59682	14300306	55160	11111875	56595	21154021	61403
10	13561985	88066	18888948	62380	16391189	68988	13172823	52267	20894874	52573
Average Running Time	14724329.1	84290	17990769.9	59630.9	14937729.6	62037.5	12657775.4	52518.3	24256246.3	70529.8

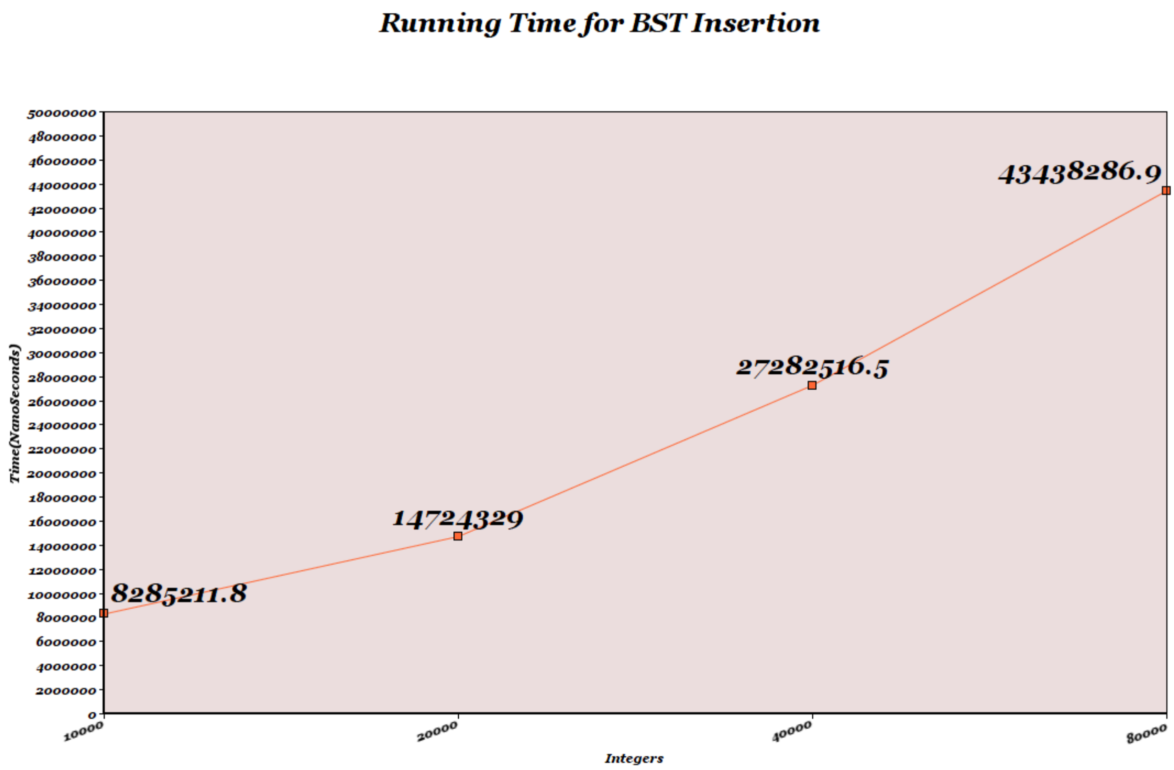
	40K Integers									
	BST		RBT		23T		BT		SL	
NanoSeconds	Time	ExtraTime	Time	ExtraTime	Time	ExtraTime	Time	ExtraTime	Time	ExtraTime
1	26108814	26839	22721261	47614	25307227	57054	24222580	50981	48437013	47196
2	26343087	32730	22954045	46671	25713059	55223	23806600	47651	49490997	44225
3	27192416	66108	26632053	48263	28599301	32620	20303486	35093	42589224	44249
4	26251486	37714	24764392	47098	28765608	38939	27299135	38169	41269354	41056
5	27429784	37711	25495926	48416	26600354	34983	23475460	45674	62755424	135988
6	26076189	37587	24564946	49973	27164381	34068	22895713	46381	59737755	44846
7	27492520	28555	25398390	50079	28647786	33039	27019893	36788	51106515	44545
8	30926169	36140	24519080	44976	23681843	55203	23229428	45188	61283157	103421
9	28805309	30661	25267230	47123	25930282	52700	24346819	48952	54207802	41477
10	26199391	33721	30188503	44405	23385086	52224	23841103	45678	42825893	44301
Average Running Time	27282516.5	36776.6	25250582.6	47461.8	26379492.7	44605.3	24044021.7	44055.5	51370313.4	59130.4

	80K Integers									
	BST		RBT		23T		BT		SL	
NanoSeconds	Time	ExtraTime	Time	ExtraTime	Time	ExtraTime	Time	ExtraTime	Time	ExtraTime
1	35962087	20553	41624971	24114	81590749	17702	82631678	34987	123313837	45439
2	34601224	19403	39554255	26605	64111041	15522	50646653	62258	118460858	40086
3	37599276	19237	57303158	26615	66447159	23480	79270044	33013	126989028	52817
4	39830399	20454	43415588	28703	63872263	15455	76895990	36058	116730563	37912
5	47282212	30328	42225302	32940	69463349	16119	89021137	89379	121942580	54373
6	36640554	21987	45292816	24668	64364977	16576	81437199	34310	118075297	53790
7	58781886	37591	78161604	43324	67257108	15645	51011952	33525	126910324	43381
8	49605086	17088	39003473	32129	60534144	22390	76566189	35139	132316057	42201
9	46802506	17121	40251461	28419	67927054	26630	73686964	34268	125959840	43952
10	47277639	38892	41292777	28553	70813709	17618	78109934	34943	128104429	47573
Average Running Time	43438286.9	24265.4	46812540.5	29607	67638155.3	18713.7	73927774	42788	123880281.3	46152.4

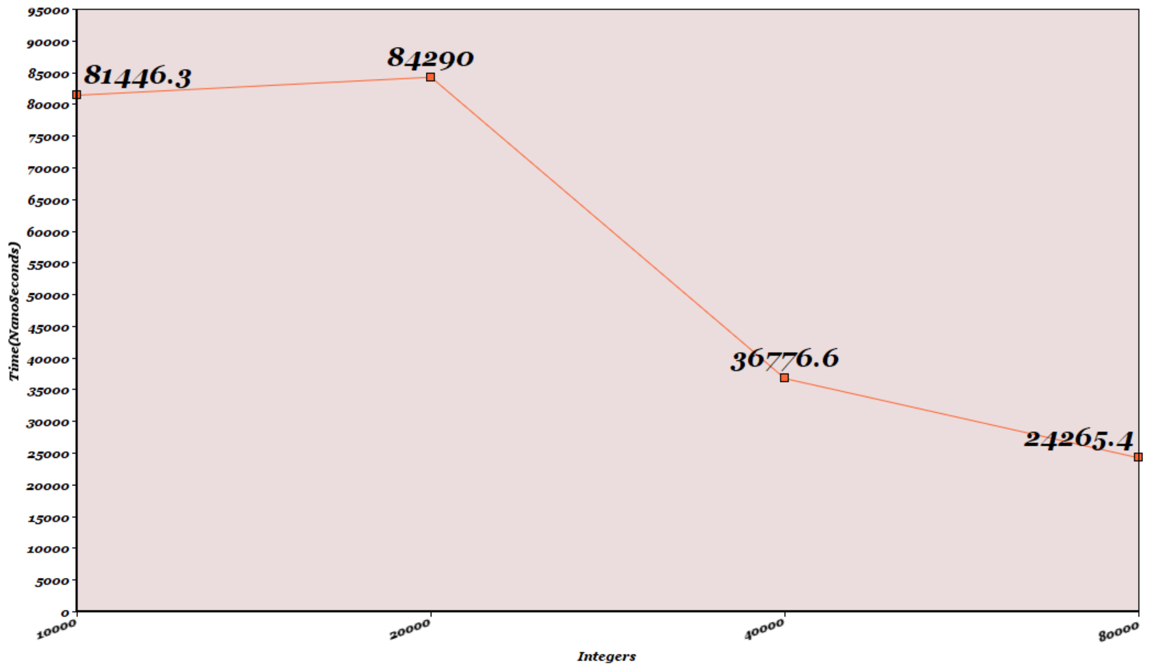
“ Graphs and Analysis of Average Run Times ”

Here We will Compare the Average Running Times of all the Data Structures in Insertions and Conclude about the Characteristics shown and Complexities.

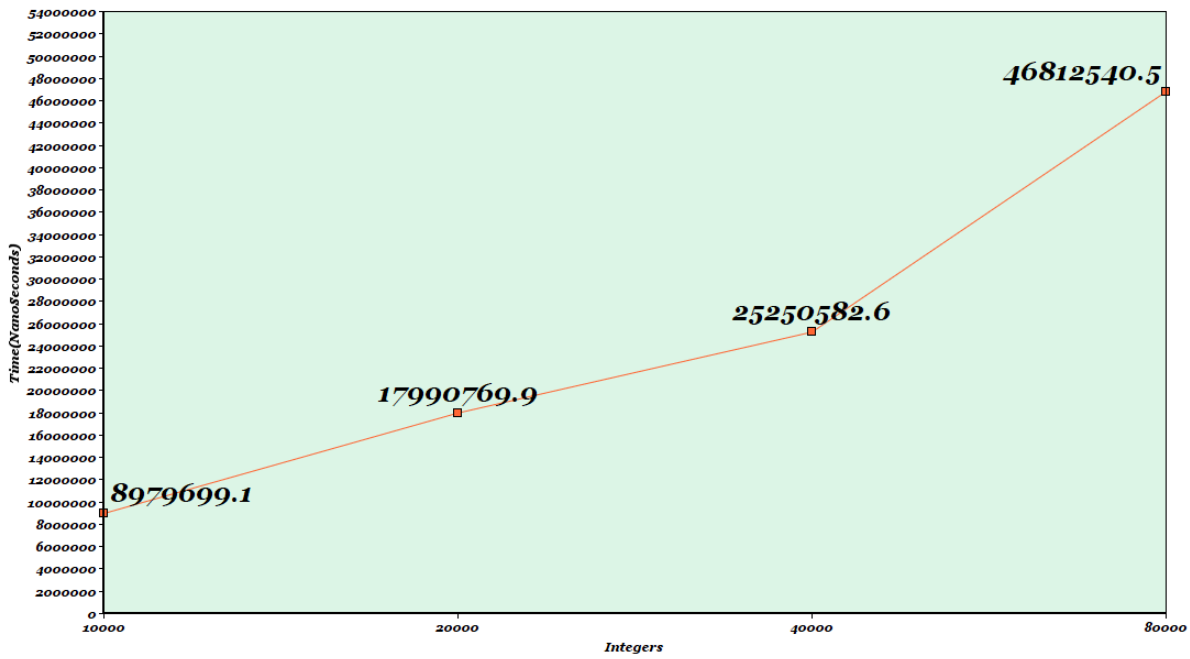
1. ***Graph1 - Avg Running Time of DataStructure***
2. ***Graph2 - Avg Running Time of inserting Extra 100 Elements to the Existing Structure***



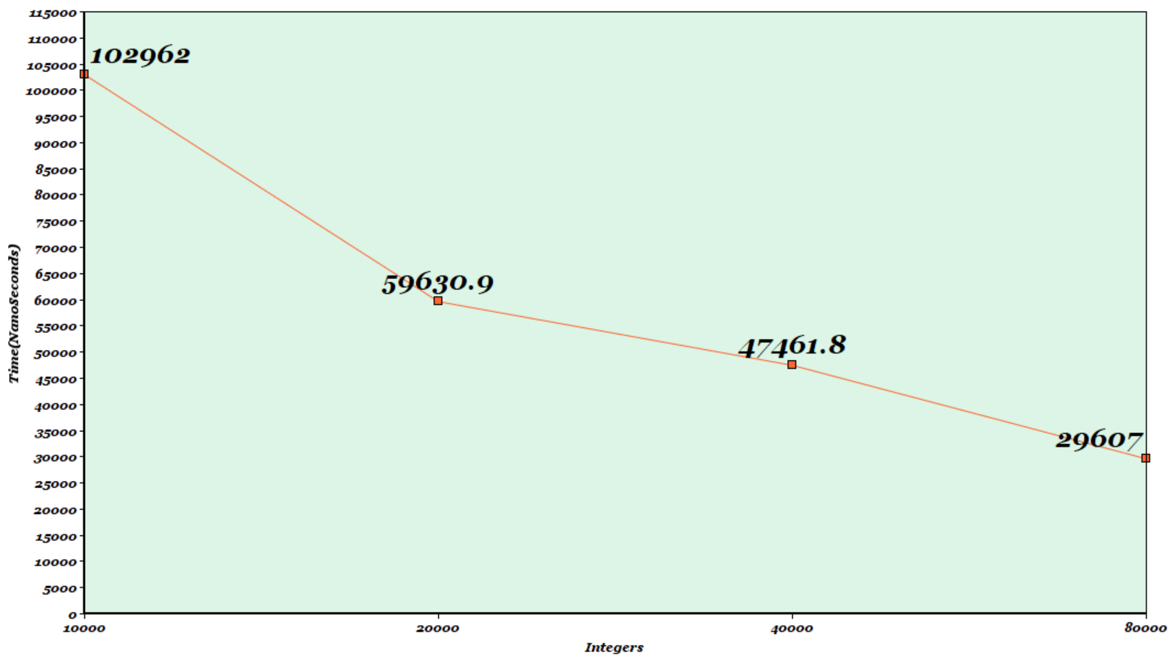
Running Time for BST Extra Insertion



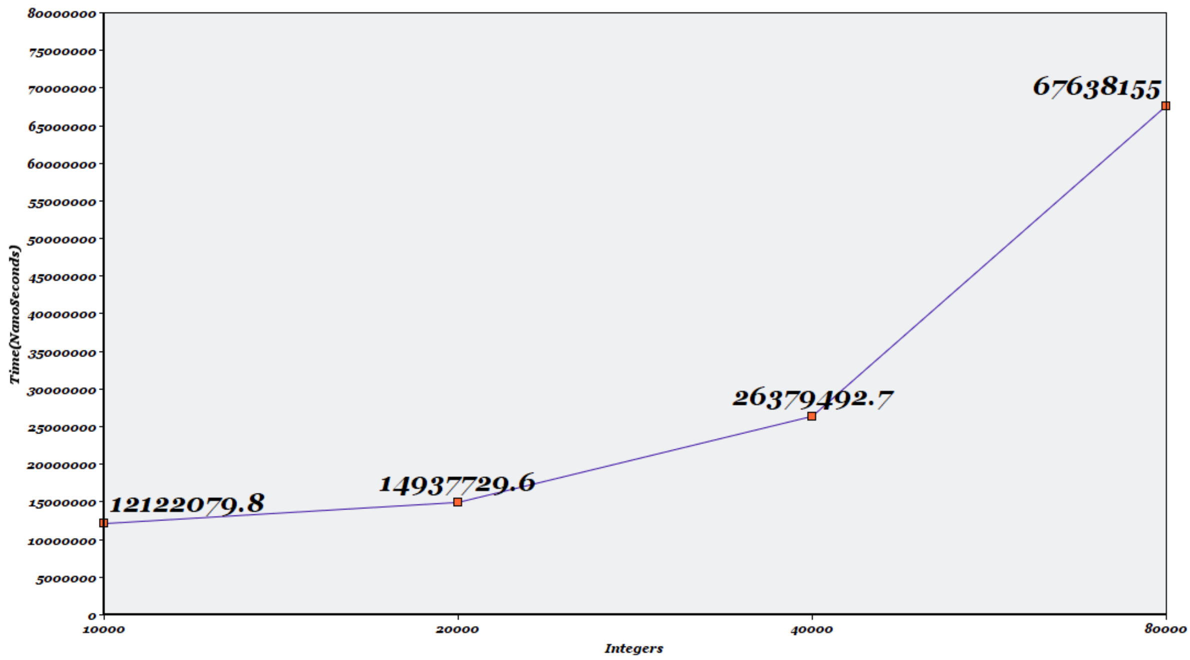
Running Time for Red-Black Tree Insertion



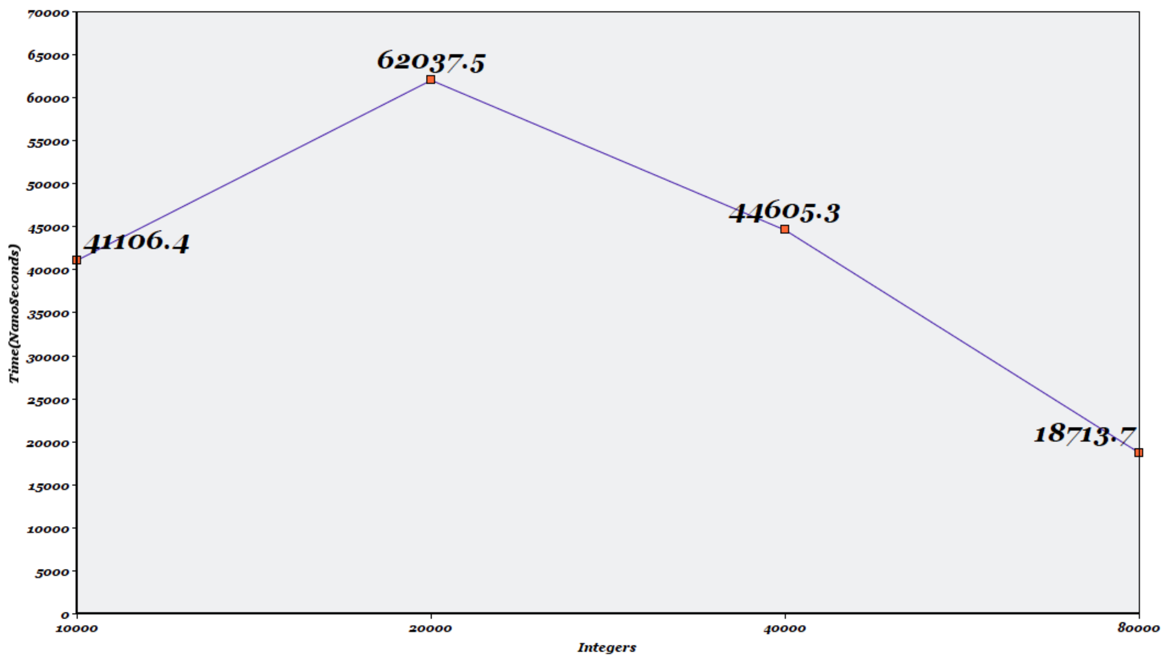
Running Time for Red-Black Tree Extra Insertion



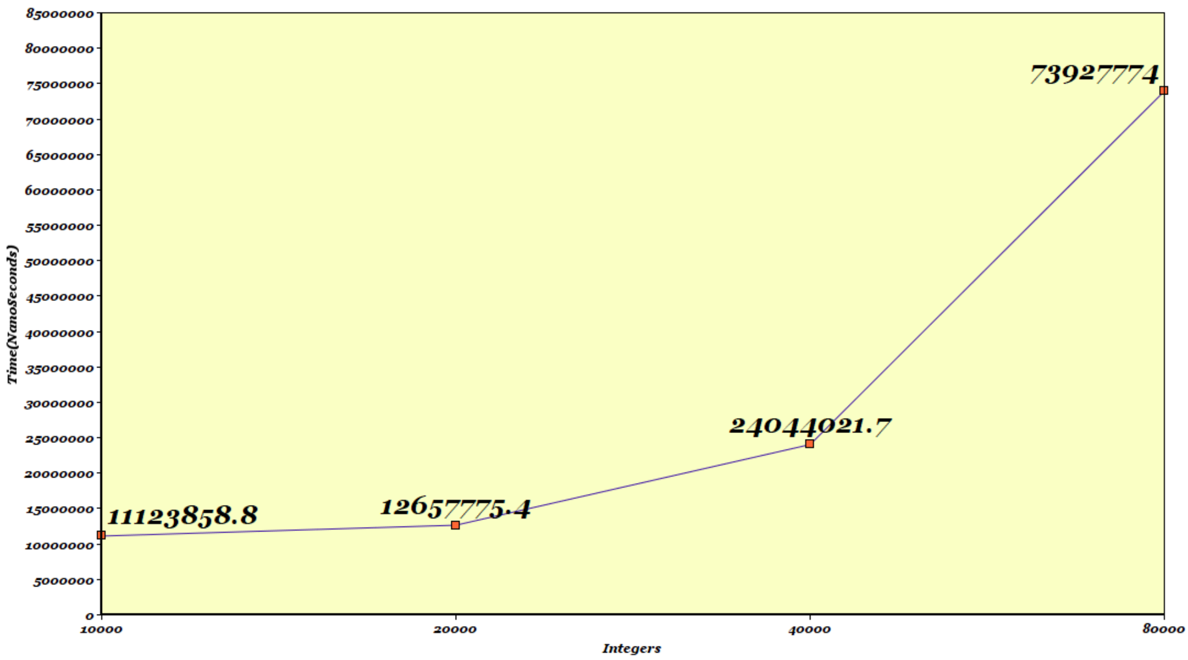
Running Time for 2-3 Tree Insertion



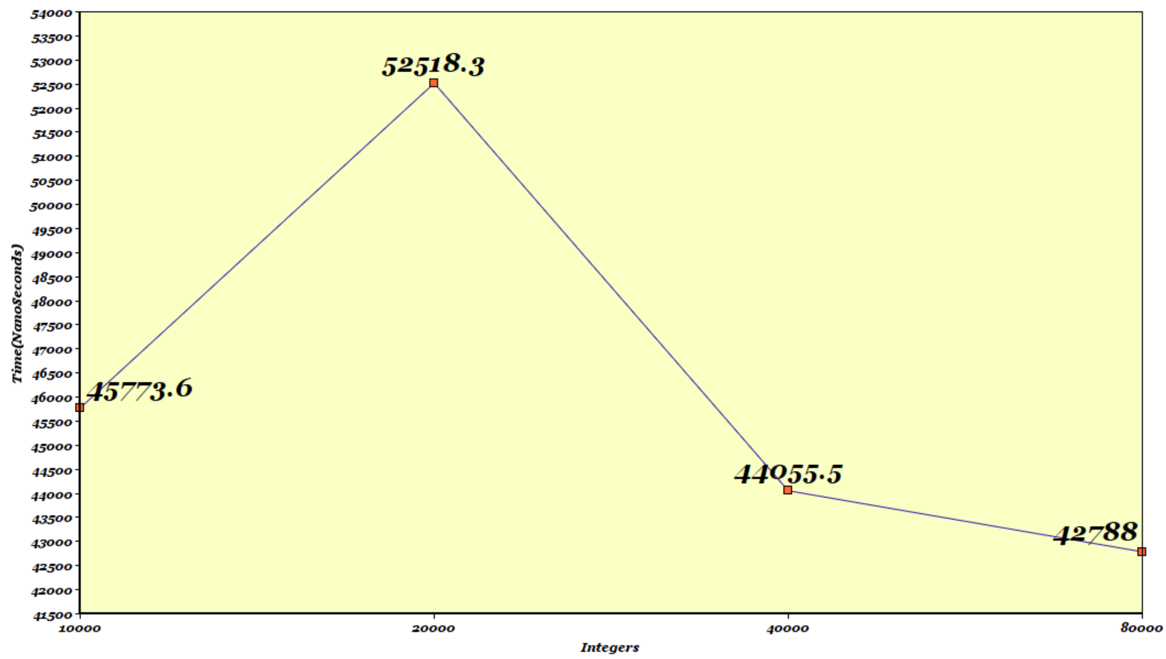
Running Time for 2-3 Tree Extra Insertion



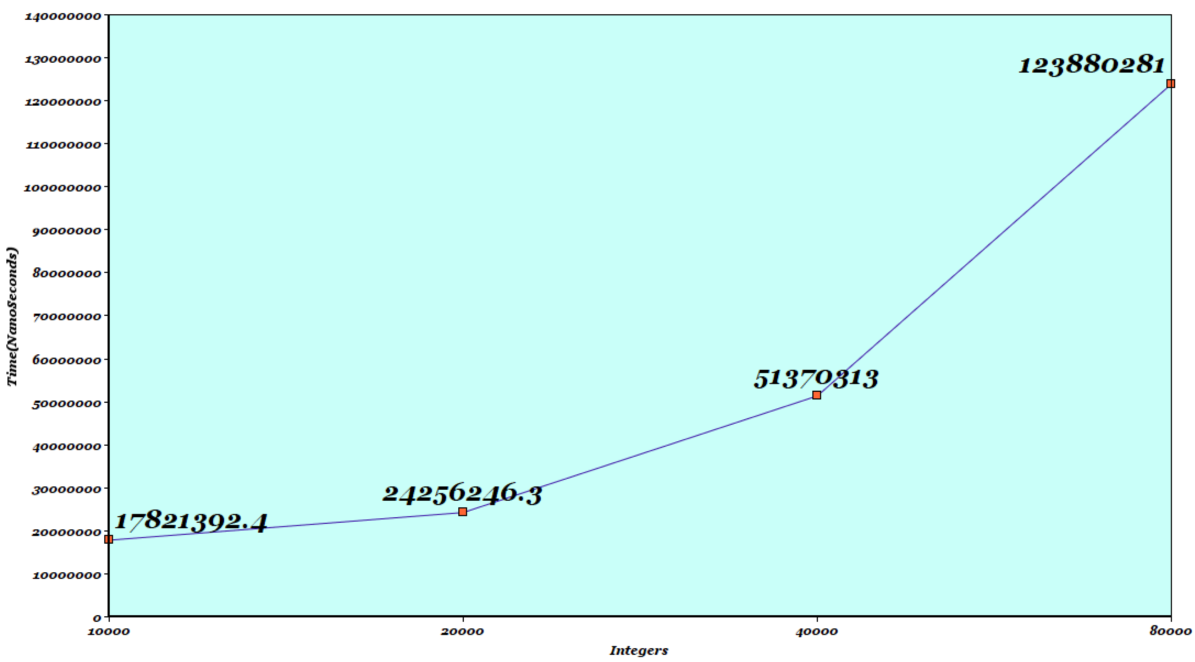
Running Time for B-Tree Insertion



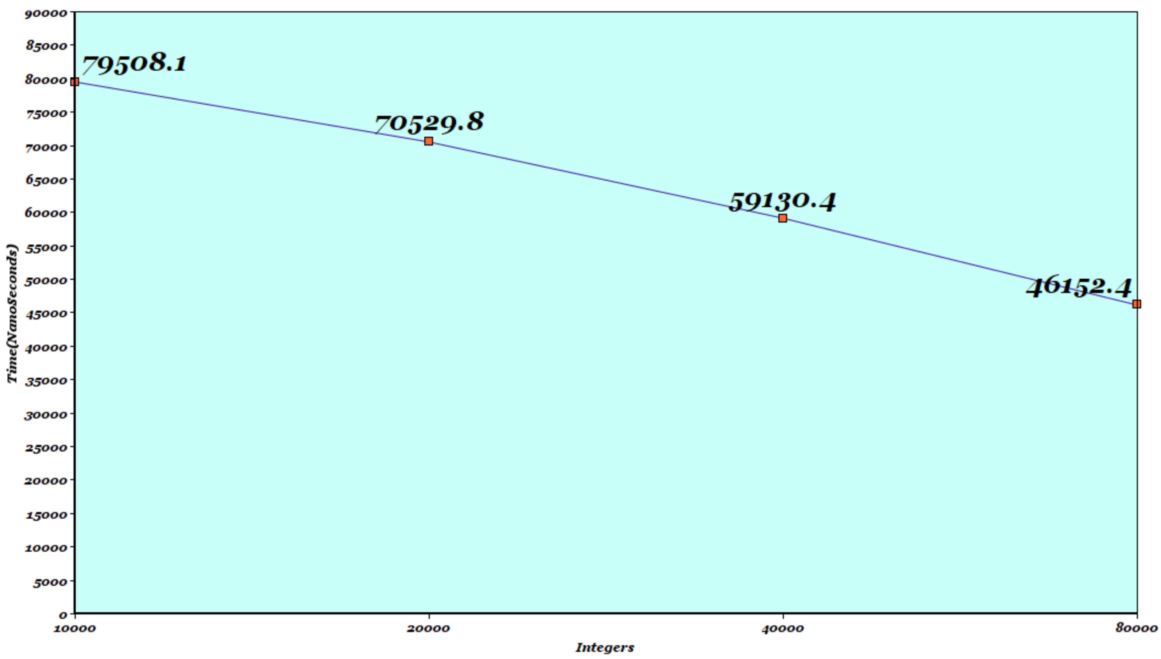
Running Time for B-Tree Extra Insertion



Running Time for SkipList Insertion



Running Time for SkipList Extra Insertion



Note : Here B-Tree is taken of the Order = 4.

Conclusion and Results

- **We can see that with an insertion of a considerable amount of integers to each of the data structures, We find that the average running time increases with the increase in the number of the integers inserted.**

- **Now the main thing is what factor the running time increases, we can approximately calculate it.**

BST === 8285.. -> 14724… -> 27282… -> 43438..
RedBlack === 897.. -> 1799.. -> 2725.. -> 4681..
SkipList === 178… -> 242… -> 513… -> 1238…
……..

- **Here we can observe that the number of Running time is approximately getting double for each data structure and since the number of integers that we are inserting is also getting doubled**

So , For Each Double in the number of Integers to be inserted, the running time of the Insertion also gets doubled

- ***<u>Hence, Time Complexity is Average Case is O(log2(n))..</u>***
- ***<u>Worst Case Time Complexity is still O(height) of the tree..</u>***

- **The Insertion of Extra integers whose count is 100 is almost random as the Tree has been already structured by that time and depending upon what random Values are getting inserted into the trees.**
- **For a small range of Insertion We cannot predict the overall time complexity of the Data Structure.**

RUNNING COMMANDS AND RESULTS

1)Navigable Set(Skiplist) Display

```
System.out.println("Navigable Set 1 : ");
nsl.display();
System.out.println();
```

```
Navigable Set 1 :
[29182, 129961, 150306, 162609, 166999, 448320, 501634, 539119, 640643, 941606]
```

2)Descending Iteration Navigable Set(Skiplist)

```
System.out.println("Descending Iteration Navigable Set 1 : ");
Iterator<Integer> nsl_itrl = nsl.descendingIterator();

while (nsl_itrl.hasNext()) {
    System.out.println("Data: " + nsl_itrl.next());
}
System.out.println();
```

```
Descending Iteration Navigable Set 1 :

Data: 941606
Data: 640643
Data: 539119
Data: 501634
Data: 448320
Data: 166999
Data: 162609
Data: 150306
Data: 129961
Data: 29182
```

3)Removal from Navigable Set(Skiplist)

```
for (int i = 3; i < 6; i++) {
    System.out.println("To Remove : " + pl_arrl[i] + " <-> Removed :" + nsl.remove(pl_arrl[i]));
}

System.out.println("To Remove : " + (-30000) + " <-> Removed :"
        + (nsl.remove(-30000) == null ? "Item Not Present" : nsl.remove(-30000)));

System.out.println();
System.out.println("Navigable Set 1 after Removal : ");
nsl.display();
System.out.println();
```

```
To Remove : 150306 <-> Removed :150306
To Remove : 162609 <-> Removed :162609
To Remove : 640643 <-> Removed :640643
To Remove : -30000 <-> Removed :Item Not Present

Navigable Set 1 after Removal :
[29182, 129961, 166999, 448320, 501634, 539119, 941606]
```

4) Display Navigable Set(AVL Tree)

```
System.out.println("Navigable Set 2 : ");
ns2.display();
```

```
Navigable Set 2 :
[[Bal: 0, Item: 97111]
, [Bal: -1, Item: 140078]
, [Bal: 0, Item: 276453]
, [Bal: 1, Item: 286074]
, [Bal: 0, Item: 374774]
, [Bal: 0, Item: 666107]
, [Bal: 0, Item: 682708]
, [Bal: -1, Item: 856135]
, [Bal: -1, Item: 929943]
, [Bal: 0, Item: 998072]
, ]
```

5)HeadSet of Value

```
AVLNavigableSet<Integer> head = ns2.headSet(p1_arr2[4]);

System.out.println();
System.out.println("HeadSet of Value " + p1_arr2[4] + " is :\n");
head.display();
System.out.println();
```

```
HeadSet of Value 276453 is :

[[Bal: 1, Item: 97111]
, [Bal: 0, Item: 140078]
, ]
```

6)TailSet of Value

```
AVLNavigableSet<Integer> tail = ns2.tailSet(p1_arr2[4]);

System.out.println("TailSet of Value " + p1_arr2[4] + " is :\n");
tail.display();
System.out.println();
```

```
TailSet of Value 276453 is :

[[Bal: 0, Item: 286074]
, [Bal: 0, Item: 374774]
, [Bal: 0, Item: 666107]
, [Bal: 0, Item: 682708]
, [Bal: 0, Item: 856135]
, [Bal: 0, Item: 929943]
, [Bal: 0, Item: 998072]
, ]
```

7)HeadSet Inclusive of Value

```
AVLNavigableSet<Integer> headinc = ns2.headSet(p1_arr2[4], true);

System.out.println("HeadSet Inclusive of Value " + p1_arr2[4] + " is :\n");
headinc.display();
System.out.println();
```

```
HeadSet Inclusive of Value 276453 is :

[[Bal: 0, Item: 97111]
, [Bal: 0, Item: 140078]
, [Bal: 0, Item: 276453]
, ]
```

8)TailSet Inclusive of Value

```
AVLNavigableSet<Integer> tailinc = ns2.tailSet(p1_arr2[4], true);

System.out.println("TailSet Inclusive of Value " + p1_arr2[4] + " is :\n");
tailinc.display();
System.out.println();
```

```
TailSet Inclusive of Value 276453 is :

[[Bal: 0, Item: 276453]
, [Bal: 0, Item: 286074]
, [Bal: 0, Item: 374774]
, [Bal: 1, Item: 666107]
, [Bal: 0, Item: 682708]
, [Bal: 1, Item: 856135]
, [Bal: 1, Item: 929943]
, [Bal: 0, Item: 998072]
, ]
```

9)Iterating HeadSet of Navigable Set

```
System.out.println("Iterating HeadSet of Navigable Set : ");

Iterator<Integer> ns2_itr2 = head.iterator();

while (ns2_itr2.hasNext()) {
    System.out.println("Data: " + ns2_itr2.next());
}
System.out.println();
```

```
Iterating HeadSet of Navigable Set :
Data: 97111
Data: 140078
```

10)Iterating Navigable Set

```
System.out.println("Iterating Navigable Set 2 : ");

Iterator<Integer> ns2_itr1 = ns2.iterator();

while (ns2_itr1.hasNext()) {
    System.out.println("Data: " + ns2_itr1.next());
}
System.out.println();
```

```
Iterating Navigable Set 2 :
Data: 97111
Data: 140078
Data: 276453
Data: 286074
Data: 374774
Data: 666107
Data: 682708
Data: 856135
Data: 929943
Data: 998072
```

11)Insertion as many as given number

```
Enter the number of Integers to Insert : 100


                                Number to be Inserted are 100

(ns)    Time      Extra Time

BST     254923    173723
RBT     185140    193051
23T     216573    308627
BT      346365    205862
SL      335358    344208
```

12) Is tree an AVL Tree

```java
int[] p2_arr1 = { 8, 6, 10, 4, 7, 9, 11, 3, 5 };

BinarySearchTree<Integer> p2_bst1 = new BinarySearchTree<Integer>();

for (int i = 0; i < 9; i++) {
    p2_bst1.add(p2_arr1[i]);
}

System.out.println("Binary Search Tree 1 : ");
System.out.println(p2_bst1.toString());
System.out.println();
System.out.println("Checking if Its AVL Balanced Tree : ");
System.out.println(p2_bst1.isAVL(p2_bst1.getRoot()));
System.out.println();
```

```
Binary Search Tree 1 :
[3, 4, 5, 6, 7, 8, 9, 10, 11, ]

Checking if Its AVL Balanced Tree :
true
```

13)Is tree a Red Black Tree?

```java
int[] p2_arr3 = { 12, 5, 15, 3, 10, 13, 17, 4, 7, 11, 14, 6, 8 };

BinarySearchTree<Integer> p2_bst3 = new BinarySearchTree<Integer>();

for (int i = 0; i < 13; i++) {
    p2_bst3.add(p2_arr3[i]);
}

System.out.println("Binary Search Tree 3 : ");
System.out.println(p2_bst3.toString());
System.out.println();
System.out.println("Checking if Its a Red Black [Balanced] Tree : ");
System.out.println(p2_bst3.isRed(p2_bst3.getRoot()));
System.out.println();
```

```
Binary Search Tree 3 :
[3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 15, 17, ]

Checking if Its a Red Black [Balanced] Tree :
true
```

*** All parts are implemented.**