GIT Department of Computer Engineering

CSE 222/505 - Spring 2021

Homework 7 Report

Muhammed Bedir ULUCAY

1901042697

1 - System Requirements:

Part 1:

```java
public class NavigableSetSkipList <E extends Comparable<E>> implements NavigableSetSkipListInterface<E>

    private SkipList<E>  skipList = null;
```

Class & Instance

```java
@Override
public boolean insert(E e)
```

Adding new element to instance

```java
@Override
public E delete(E e)
```

Removing an element from instance

```java
/**Implemented in skiplist class*/
@SuppressWarnings("unchecked")
@Override
public Iterator<E> descendingIterator()
```

Returning descending iterator

```java
public class NavigableSetAvlTree<E extends Comparable<E>> implements NavigableSetAvlTreeInterface<E>

    private MyAvlTree<E> avlTree = null;
```

Class & Instance

```java
@Override
public boolean insert(E e)
```

Adding new element to instance

```java
@Override
public Iterator<E> iterator()
```

Returning inorder iterator

```java
@Override
public MyAvlTree<E> headSet(E e)
```

Returning headset of getting value

```java
@Override
public MyAvlTree<E> tailSet(E e)
```

Returning tailset of getting value

Part 2:

```
public boolean isRedBlackTree(BinarySearchTree<E> tree)
    return isBalancedRedBlack(tree.root,0,0);
```

```
private boolean isBalancedRedBlack(BinaryTree.Node<E> node,Integer maxH,Integer minH)
```

Reurning the result if the getting tree is obey the rules of red black tree return true other wise false

```
public boolean isAVL(BinarySearchTree<E> tree)
    return isBalanced(tree.root);
```

```
private boolean isBalanced(BinaryTree.Node<E> node)
```

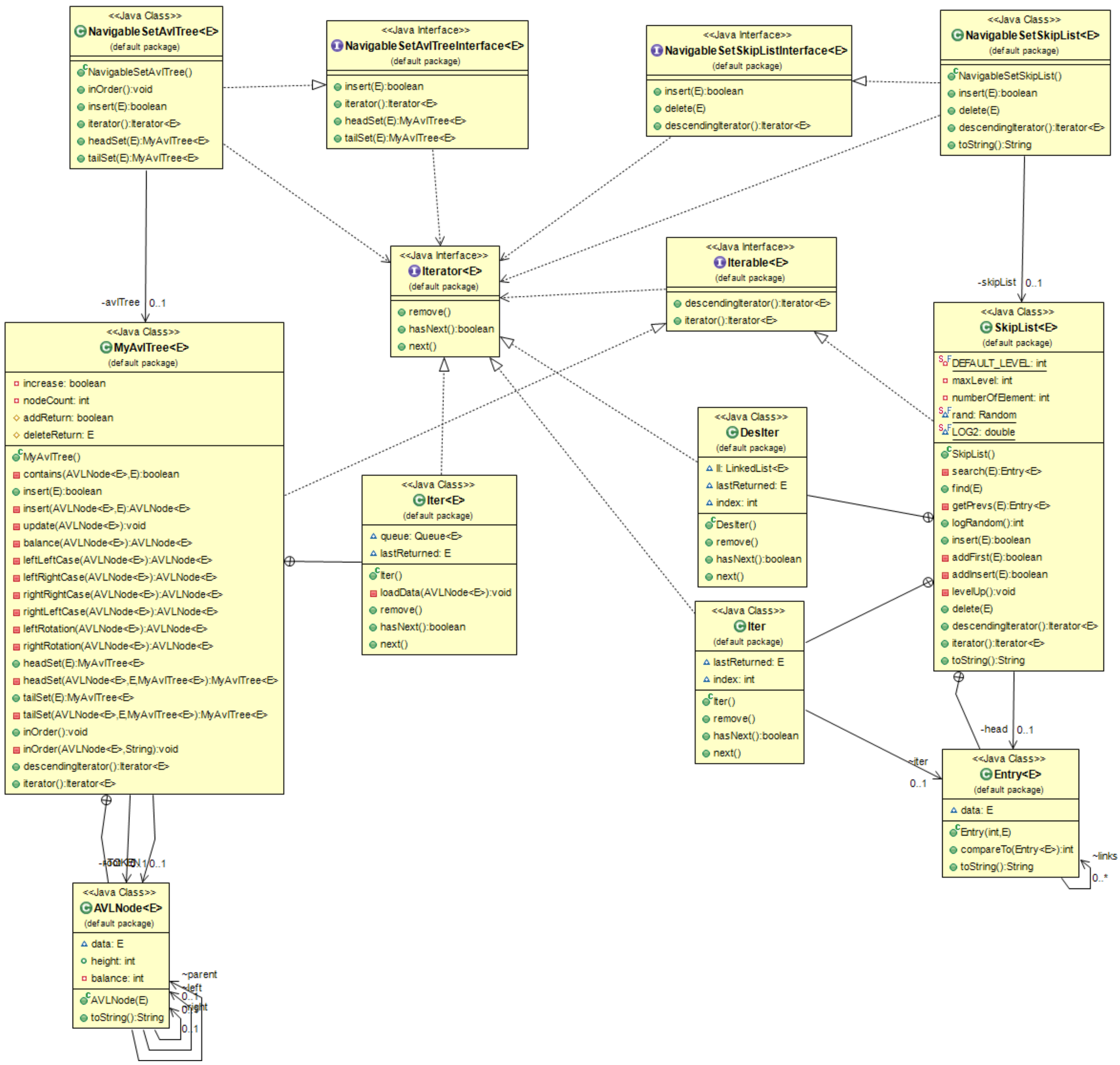
Reurning the result if the getting tree is obey the rules of avl tree return true other wise false

Part 3:

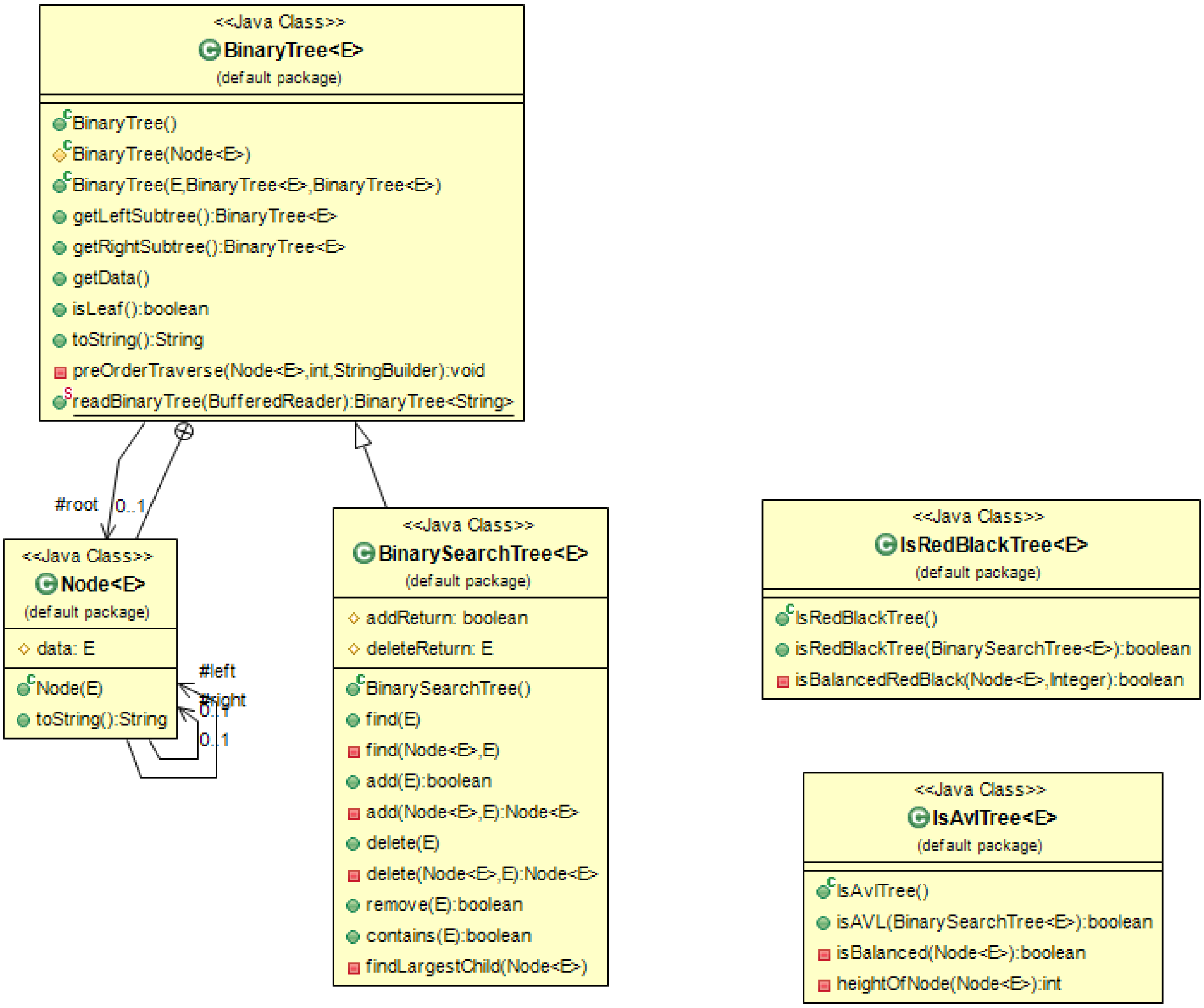
Number of set size 10K,20K,40K,80K creater not repeating and fill the created 200 instance then compare the running time results.

2 – Class Diagrams:

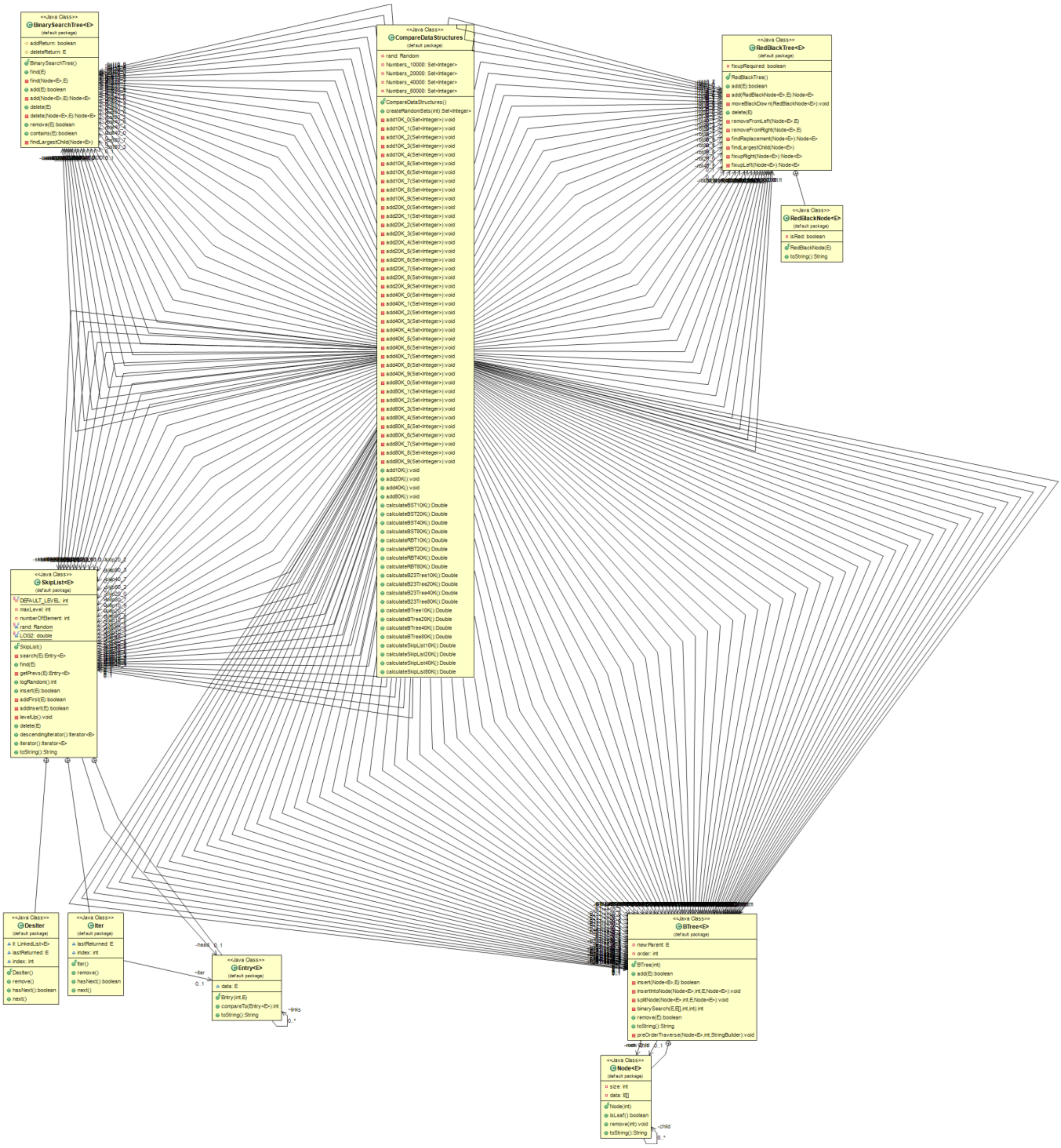
Part 1 Class Diagram:



Part 2 Class Diagram:



Part 3 Class Diagram:



3 – Problem Solution Approach

Problem solution steps are;

– Specify the problem requirements

– Analyze the problem

– Design an algorithm and Program

– Implement the algorithm

– Test and verify the program

– Maintain and update the program

Part 1 Solution Approach:

Created an instance of skip list and avl tree in the navigble set class for each and using this instance reach the other class's method and store the data in the instant.

Part 2 Solution Approach:

The basic role of this tree like avl and red-black tree is garranti to us adding searching and removing operations are hapenning in O(log n) time. We need to apply a few rule for to do that.

For avl tree:

The different between left and right branch for each node must 1 or 0 other wise.It cannot accept as Avl Tree.

There are a fer rotate operations according to the statuation.

For Red Black Tree:

According to the avl tree the different between branch height is minHeight*1 >= maxHeight. So that in red black tree does not require rotate operations like avl tree.

To A tree be an red black tree it needs to obey the following rules;

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 isconsidered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Part 3 Solution Approach:

Comparing the following data structure for different size

- Binary search tree implementation in the book
- Red-Black tree implementation in the book
- 2-3 tree implementation in the book
- B-tree implementation in the book
- Skip list implementation in the book

Measure the time for each instance of data structure (total 200 instance) and compare the time result and increasing rate according to the getting numbers.

4 – Test Cases

Part 1 Test Case:

NavigableSetSkipList:

- Inserting new element
- Deleting an element
- Using descending iterator

NavigableSetAvlTree:

- Inserting new element
- Checking inorder iterator
- Checking tailSet iterator
- Checking headSet iterator

Part 2 Test Case:

IsRedBlackTree:

- Checking function works properly for rule of rbt
- for true and false case

IsAvlTree:

- Checking function works properly for rule of avl tree
- For true and false case

Part 3 Test Case:

CompareDataStructures:

- Comparison between following data structure for 10K,20K,40K and 80K element
- Binary search tree implementation in the book
- Red-Black tree implementation in the book
- 2-3 tree implementation in the book
- B-tree implementation in the book
- Skip list implementation in the book
- **And their analysis**

5 – Runnig Command & Results

Part 1:

A) NavigableSetSkipList:

Inserting:

```
a) Navigable Set using skip list
Default size = 2
Data: null Level: 2
0 -> null
1 -> null
```

```
add some random numbers
Data: null Level: 4
0 -> Data: 35 Level: 1
1 -> Data: 98 Level: 4
2 -> Data: 98 Level: 4
3 -> Data: 98 Level: 4
Data: 35 Level: 1
0 -> Data: 98 Level: 4
Data: 98 Level: 4
0 -> Data: 119 Level: 3
1 -> Data: 119 Level: 3
2 -> Data: 119 Level: 3
3 -> Data: 484 Level: 4
Data: 119 Level: 3
0 -> Data: 358 Level: 3
1 -> Data: 358 Level: 3
2 -> Data: 358 Level: 3
Data: 358 Level: 3
0 -> Data: 415 Level: 1
1 -> Data: 462 Level: 2
2 -> Data: 484 Level: 4
Data: 415 Level: 1
0 -> Data: 462 Level: 2
Data: 462 Level: 2
0 -> Data: 484 Level: 4
1 -> Data: 484 Level: 4
Data: 484 Level: 4
```

```
Data: 484 Level: 4
0 -> Data: 579 Level: 3
1 -> Data: 579 Level: 3
2 -> Data: 579 Level: 3
3 -> Data: 738 Level: 4
Data: 579 Level: 3
0 -> Data: 645 Level: 1
1 -> Data: 664 Level: 2
2 -> Data: 738 Level: 4
Data: 645 Level: 1
0 -> Data: 664 Level: 2
Data: 664 Level: 2
0 -> Data: 738 Level: 4
1 -> Data: 738 Level: 4
Data: 738 Level: 4
0 -> Data: 852 Level: 3
1 -> Data: 852 Level: 3
2 -> Data: 852 Level: 3
3 -> null
Data: 852 Level: 3
0 -> Data: 860 Level: 2
1 -> Data: 860 Level: 2
2 -> null
Data: 860 Level: 2
0 -> null
1 -> null
```

Descending Iterator:

```
Iterate on the Navigable set skip list
860 => 852 => 738 => 664 => 645 => 579 => 484 => 462 => 415 => 358 => 119 => 98 => 35 => Null
```

Deleting element:

```
remove 860
remove 484
remove 35
Using delete
nssl.delete(first);
nssl.delete(middle);
nssl.delete(last);
Print again using iterator
852 -> 738 -> 664 -> 645 -> 579 -> 462 -> 415 -> 358 -> 119 -> 98 -> Null
```

B) NavigableSetAvlTree:

Inserting:

```
b) Navigable set using avl tree
Insert some numbers
Inorder print
hllll(17,0)
hlll(102,0)
hlllr(148,0)
hll(198,-1)
hllr(250,0)
hl(348,1)
hlrll(475,0)
hlrl(476,-1)
hlr(479,1)
hlrrl(484,1)
hlrrlr(550,0)
hlrr(640,-1)
hlrrr(657,0)
h(658,-1)
hrll(687,0)
hrl(701,0)
hrlr(741,0)
hr(803,1)
hrrl(827,0)
hrr(855,1)
hrrrl(880,0)
hrrr(911,0)
hrrrr(921,0)
```

Inorder iterator:

```
Print avl navigable set using iterator as inorder
17
102
148
198
250
348
475
476
479
484
550
640
657
658
687
701
741
803
827
855
880
911
921
```

Tail Set:

```
tail set of number 479
Print tail set as inorder
hlll(484,0)
hll(550,0)
hllr(640,0)
hl(657,0)
hlrl(658,0)
hlr(687,0)
hlrr(701,0)
h(741,0)
hrll(803,0)
hrl(827,0)
hrlr(855,0)
hr(880,0)
hrr(911,1)
hrrr(921,0)
```

Head Set:

```
head set of number 479
Print head set as inorder
hll(17,0)
hl(102,0)
hlr(148,0)
h(198,1)
hrl(250,0)
hr(348,1)
hrr(475,1)
hrrr(476,0)
```

Part 2:

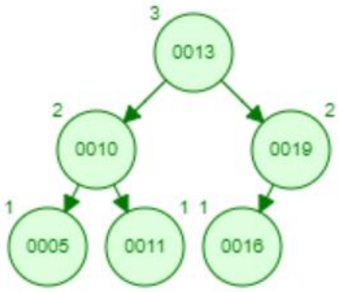
A) IsAvlTree:

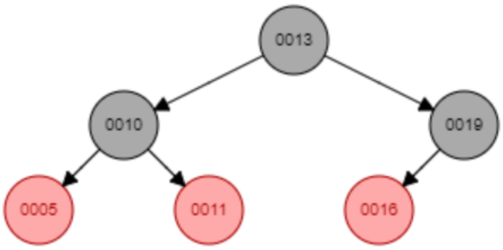
```java
/**The different between left and right branch for each node
 * Must be less than two other wise.It cannot accept as Avl Tree*/
public boolean isAVL(BinarySearchTree<E> tree) {
    return isBalanced(tree.root);
}

/**Is balanced avl tree control function*/
private boolean isBalanced(BinaryTree.Node<E> node){

    if (node == null)
        return true;
    // Between left and right tree the different of high must be 1 or 0
    // Otherwise the tree is not available for being avl tree
    if (Math.abs(heightOfNode(node.left) - heightOfNode(node.right)) <= 1
        && isBalanced(node.left)
        && isBalanced(node.right)) {
        return true;
    }
    return false;
}

/**Calculating the height of tree node*/
private int heightOfNode(BinaryTree.Node<E> node){
    if (node == null)
        return 0;
    return 1 + Math.max(heightOfNode(node.left), heightOfNode(node.right));
}
```

B) IsRedBlackTree:

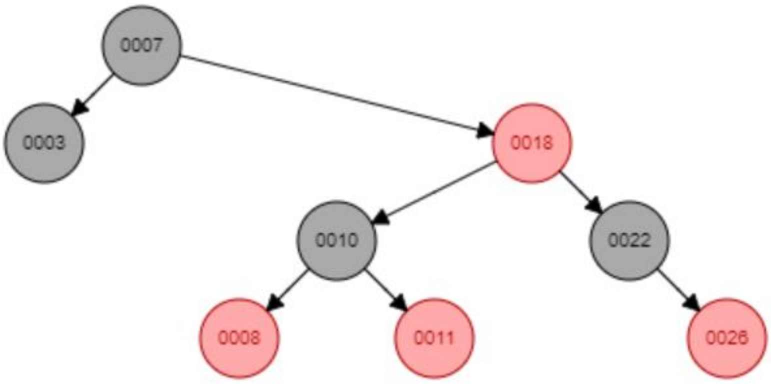
```
/**Wrapper method for
 * isBalancedRedBlack(BinaryTree.Node<E> ,Integer)*/
public boolean isRedBlackTree(BinarySearchTree<E> tree) {

    return isBalancedRedBlack(tree.root,0,0);
}

/**equation must be maxH <= min*2 if this equation is not true
* Then this tree cannot be red-black tree
* If the equation is provided that means the tree balance enough for red black tree*/
private boolean isBalancedRedBlack(BinaryTree.Node<E> node,Integer maxH,Integer minH) {

    if(node == null) {
        maxH = minH = 0;
        return true;
    }
    Integer leftMaxH = 0, leftMinH = 0;
    Integer rightMaxH =0, rightMinH =0;

    if(isBalancedRedBlack(node.left, leftMaxH, leftMinH) == false)
        return false;
    if(!isBalancedRedBlack(node.right, rightMaxH , rightMinH) == false)
        return false;

    if(maxH <= minH*2)
        return true;

    return false;
}
```

Tests For Part 2:

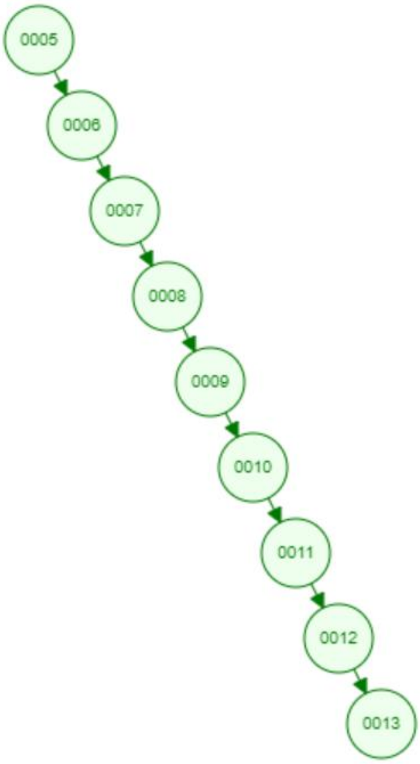





```
Create an binary tree avl tree with following values
Insert 13, 10, 19, 5 ,11 , 16
Is avl = true
Is rbt = true
```




```
Create another bst with following values
Insert 7, 3 , 18, 10, 22, 8, 11, 26
Is avl = false
Is rbt = true
```




```
Create another bst with following values
Insert 5, 6, 7, 8, 9, 10, 11, 12, 13
Is avl = false
Is rbt = false
```

Part 3:

Results:

Binary Search Tree:

```
BST_10K_0 : 148000
BST_10K_1 : 209000
BST_10K_2 : 261500
BST_10K_3 : 155800
BST_10K_4 : 198000
BST_10K_5 : 139300
BST_10K_6 : 232200
BST_10K_7 : 288700
BST_10K_8 : 142000
BST_10K_9 : 111900
Average for BST 10K avg = 188640.0
----------------------
BST_20K_0 : 205600
BST_20K_1 : 177300
BST_20K_2 : 97900
BST_20K_3 : 127700
BST_20K_4 : 83400
BST_20K_5 : 81700
BST_20K_6 : 105400
BST_20K_7 : 91200
BST_20K_8 : 90600
BST_20K_9 : 101900
Average for BST 20K avg = 116270.0
----------------------
```

```
BST_40K_0 : 99700
BST_40K_1 : 110700
BST_40K_2 : 112700
BST_40K_3 : 101300
BST_40K_4 : 105600
BST_40K_5 : 126500
BST_40K_6 : 98900
BST_40K_7 : 101800
BST_40K_8 : 115500
BST_40K_9 : 98900
Average for BST 40K avg = 107160.0
----------------------
BST_80K_0 : 115700
BST_80K_1 : 127100
BST_80K_2 : 105200
BST_80K_3 : 131100
BST_80K_4 : 121400
BST_80K_5 : 157300
BST_80K_6 : 131800
BST_80K_7 : 184300
BST_80K_8 : 251100
BST_80K_9 : 237900
Average for BST 80K avg = 156290.0
============================
```

Red-Black Tree:

```
RBT_10K_0 : 206700
RBT_10K_1 : 157400
RBT_10K_2 : 156100
RBT_10K_3 : 113000
RBT_10K_4 : 123900
RBT_10K_5 : 103900
RBT_10K_6 : 159800
RBT_10K_7 : 120200
RBT_10K_8 : 128000
RBT_10K_9 : 106500
Average for RBT 10K avg = 137550.0
----------------------
RBT_20K_0 : 114900
RBT_20K_1 : 137500
RBT_20K_2 : 100100
RBT_20K_3 : 109400
RBT_20K_4 : 94100
RBT_20K_5 : 93300
RBT_20K_6 : 135800
RBT_20K_7 : 96300
RBT_20K_8 : 95100
RBT_20K_9 : 117900
Average for RBT 20K avg = 109440.0
----------------------
```

```
RBT_40K_0 : 114900
RBT_40K_1 : 123800
RBT_40K_2 : 117100
RBT_40K_3 : 93800
RBT_40K_4 : 92900
RBT_40K_5 : 126500
RBT_40K_6 : 102400
RBT_40K_7 : 113500
RBT_40K_8 : 105900
RBT_40K_9 : 176500
Average for RBT 40K avg = 116730.0
----------------------
RBT_80K_0 : 166700
RBT_80K_1 : 160000
RBT_80K_2 : 179700
RBT_80K_3 : 140700
RBT_80K_4 : 132900
RBT_80K_5 : 129000
RBT_80K_6 : 116000
RBT_80K_7 : 215800
RBT_80K_8 : 243400
RBT_80K_9 : 201900
Average for RBT 80K avg= 168610.0
----------------------
```

2-3 BTree:

```
23BTree_10K_0 : 127000
23BTree_10K_1 : 122600
23BTree_10K_2 : 120800
23BTree_10K_3 : 114100
23BTree_10K_4 : 111500
23BTree_10K_5 : 100300
23BTree_10K_6 : 129700
23BTree_10K_7 : 150300
23BTree_10K_8 : 100500
23BTree_10K_9 : 170500
23BTree for 10K avg = 124730.0
----------------------
23BTree_20K_0 : 216900
23BTree_20K_1 : 174200
23BTree_20K_2 : 123100
23BTree_20K_3 : 133200
23BTree_20K_4 : 125900
23BTree_20K_5 : 157400
23BTree_20K_6 : 113100
23BTree_20K_7 : 157600
23BTree_20K_8 : 113300
23BTree_20K_9 : 120600
23BTree for 20K avg = 143530.0
----------------------
```

```
23BTree_40K_0 : 157900
23BTree_40K_1 : 166100
23BTree_40K_2 : 150900
23BTree_40K_3 : 141500
23BTree_40K_4 : 142000
23BTree_40K_5 : 152500
23BTree_40K_6 : 150800
23BTree_40K_7 : 151900
23BTree_40K_8 : 161800
23BTree_40K_9 : 189000
23BTree for 40K avg = 156440.0
----------------------
23BTree_80K_0 : 243600
23BTree_80K_1 : 256000
23BTree_80K_2 : 233500
23BTree_80K_3 : 270300
23BTree_80K_4 : 243200
23BTree_80K_5 : 260900
23BTree_80K_6 : 223700
23BTree_80K_7 : 309100
23BTree_80K_8 : 352100
23BTree_80K_9 : 301600
23BTree for 80K avg = 269400.0
----------------------
```

BTree (Order = 111):

```
BTree_10K_0 : 202800
BTree_10K_1 : 175200
BTree_10K_2 : 170600
BTree_10K_3 : 168400
BTree_10K_4 : 203400
BTree_10K_5 : 173900
BTree_10K_6 : 195800
BTree_10K_7 : 182800
BTree_10K_8 : 109000
BTree_10K_9 : 174200
BTree for 10K avg = 175610.0
----------------------
BTree_20K_0 : 210300
BTree_20K_1 : 230900
BTree_20K_2 : 132800
BTree_20K_3 : 186200
BTree_20K_4 : 126100
BTree_20K_5 : 156300
BTree_20K_6 : 162600
BTree_20K_7 : 150600
BTree_20K_8 : 216800
BTree_20K_9 : 348500
BTree for 20K avg = 192110.0
----------------------
```

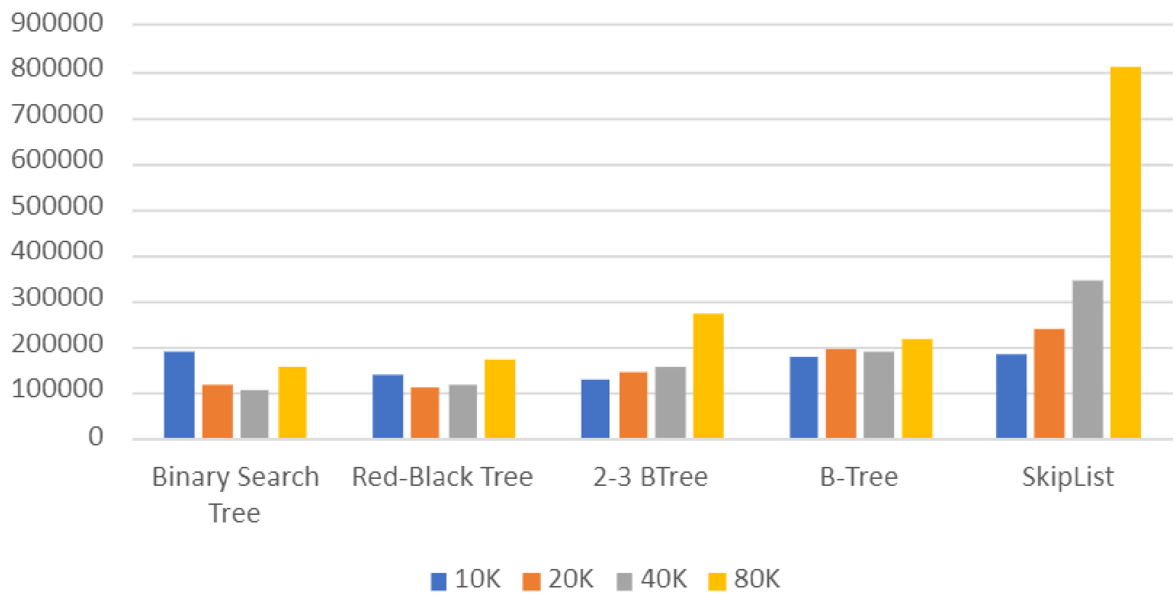
```
BTree_40K_0 : 208400
BTree_40K_1 : 215900
BTree_40K_2 : 146200
BTree_40K_3 : 178800
BTree_40K_4 : 219900
BTree_40K_5 : 198600
BTree_40K_6 : 152800
BTree_40K_7 : 235800
BTree_40K_8 : 166700
BTree_40K_9 : 166500
BTree for 40K avg = 188960.0
----------------------
BTree_80K_0 : 176000
BTree_80K_1 : 259200
BTree_80K_2 : 181000
BTree_80K_3 : 237200
BTree_80K_4 : 198100
BTree_80K_5 : 283100
BTree_80K_6 : 183800
BTree_80K_7 : 223100
BTree_80K_8 : 215100
BTree_80K_9 : 222900
BTree for 80K avg = 217950.0
----------------------
```

Skip List:

```
SkipList_10K_0 : 169100
SkipList_10K_1 : 164300
SkipList_10K_2 : 149700
SkipList_10K_3 : 167400
SkipList_10K_4 : 200300
SkipList_10K_5 : 188700
SkipList_10K_6 : 217700
SkipList_10K_7 : 166000
SkipList_10K_8 : 184100
SkipList_10K_9 : 202200
Skip List for 10K avg = 180950.0
-----------------------
SkipList_20K_0 : 292000
SkipList_20K_1 : 258800
SkipList_20K_2 : 217000
SkipList_20K_3 : 253400
SkipList_20K_4 : 225500
SkipList_20K_5 : 214700
SkipList_20K_6 : 211700
SkipList_20K_7 : 238500
SkipList_20K_8 : 219100
SkipList_20K_9 : 248100
Skip List for 20K avg = 237880.0
-----------------------
```

```
-----------------------
SkipList_40K_0 : 351500
SkipList_40K_1 : 355100
SkipList_40K_2 : 305800
SkipList_40K_3 : 334100
SkipList_40K_4 : 313900
SkipList_40K_5 : 356800
SkipList_40K_6 : 358200
SkipList_40K_7 : 373600
SkipList_40K_8 : 332600
SkipList_40K_9 : 348100
Skip List for 40K avg = 342970.0
-----------------------
SkipList_80K_0 : 473500
SkipList_80K_1 : 535800
SkipList_80K_2 : 482800
SkipList_80K_3 : 552700
SkipList_80K_4 : 489500
SkipList_80K_5 : 496000
SkipList_80K_6 : 483400
SkipList_80K_7 : 1475000
SkipList_80K_8 : 1540400
SkipList_80K_9 : 1551300
Skip List for 80K avg = 808040.0
```

Analysis Results:



The more prefable data structure for each size is red-black tree or binary search tree.

Running Time Comparison:

For 10K element:

2-3BTree < Red-Black Tree < Binary Search Tree < BTree < SkipList

For 20K element:

Red-Black Tree < Binary Search Tree < 2-3BTree < BTree < SkipList

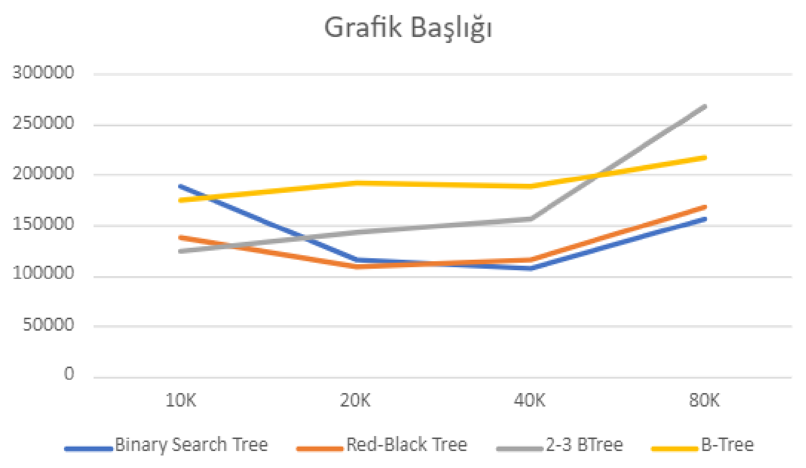
For 40K element:

Binary Search Tree < Red-Black Tree < 2-3BTree < BTree < SkipList

For 40K element:

Binary Search Tree < Red-Black Tree < 2-3BTree < BTree < SkipList

Increasing Rate Comparision:



Skip List has much more increasing rate compare to the others.

SkipList > 23BTree > BTree > Binary Search Tree = Red-Black Tree