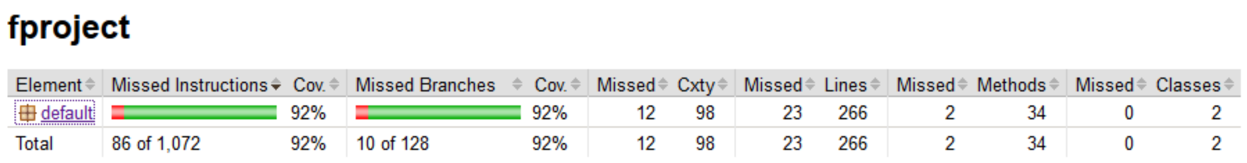
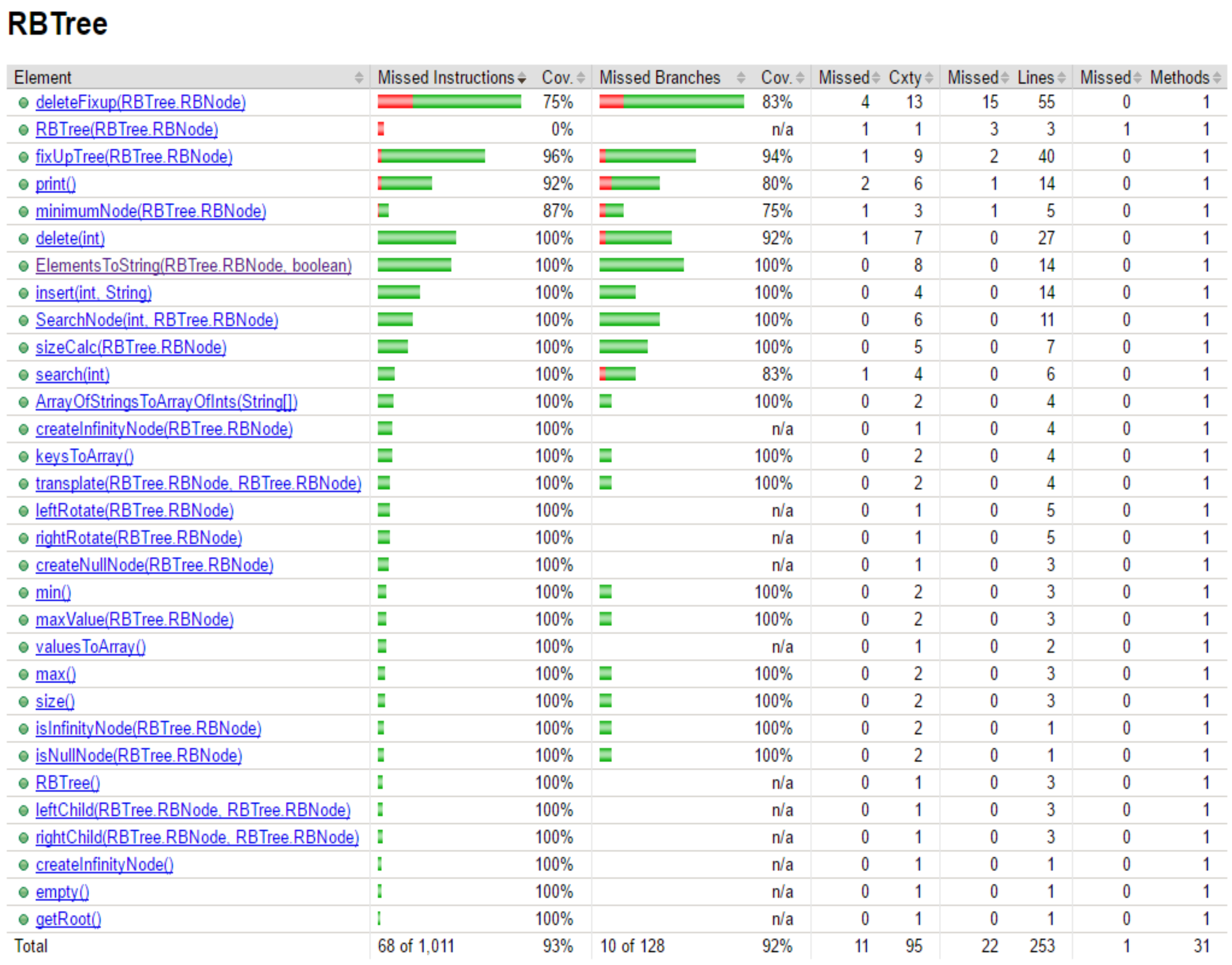
Project:	Software Testing Project
Team:	6
Class:	CSE 4321; Spring 2016
Date:	05/05/2016
Contributors:	John Green, John Sapp

Code Coverage Report:



fproject

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
default		92%		92%	12	98	23	266	2	34	0	2
Total	86 of 1,072	92%	10 of 128	92%	12	98	23	266	2	34	0	2



RBTree

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
deleteFixup(RBTree.RBNode)		75%		83%	4	13	15	55	0	1
RBTree(RBTree.RBNode)		0%		n/a	1	1	3	3	1	1
fixUpTree(RBTree.RBNode)		96%		94%	1	9	2	40	0	1
print()		92%		80%	2	6	1	14	0	1
minimumNode(RBTree.RBNode)		87%		75%	1	3	1	5	0	1
delete(int)		100%		92%	1	7	0	27	0	1
ElementsToString(RBTree.RBNode, boolean)		100%		100%	0	8	0	14	0	1
insert(int, String)		100%		100%	0	4	0	14	0	1
SearchNode(int, RBTree.RBNode)		100%		100%	0	6	0	11	0	1
sizeCalc(RBTree.RBNode)		100%		100%	0	5	0	7	0	1
search(int)		100%		83%	1	4	0	6	0	1
ArrayOfStringsToArrayOfInts(String[])		100%		100%	0	2	0	4	0	1
createInfinityNode(RBTree.RBNode)		100%		n/a	0	1	0	4	0	1
keysToArray()		100%		100%	0	2	0	4	0	1
transplate(RBTree.RBNode, RBTree.RBNode)		100%		100%	0	2	0	4	0	1
leftRotate(RBTree.RBNode)		100%		n/a	0	1	0	5	0	1
rightRotate(RBTree.RBNode)		100%		n/a	0	1	0	5	0	1
createNullNode(RBTree.RBNode)		100%		n/a	0	1	0	3	0	1
min()		100%		100%	0	2	0	3	0	1
maxValue(RBTree.RBNode)		100%		100%	0	2	0	3	0	1
valuesToArray()		100%		n/a	0	1	0	2	0	1
max()		100%		100%	0	2	0	3	0	1
size()		100%		100%	0	2	0	3	0	1
isInfinityNode(RBTree.RBNode)		100%		100%	0	2	0	1	0	1
isNullNode(RBTree.RBNode)		100%		100%	0	2	0	1	0	1
RBTree()		100%		n/a	0	1	0	3	0	1
leftChild(RBTree.RBNode, RBTree.RBNode)		100%		n/a	0	1	0	3	0	1
rightChild(RBTree.RBNode, RBTree.RBNode)		100%		n/a	0	1	0	3	0	1
createInfinityNode()		100%		n/a	0	1	0	1	0	1
empty()		100%		n/a	0	1	0	1	0	1
getRoot()		100%		n/a	0	1	0	1	0	1
Total	68 of 1,011	93%	10 of 128	92%	11	95	22	253	1	31

Summary and Discussion:

This project was completed using the methods and techniques discussed during the course of the semester. The first step taken was the creation of block tables for each method in the red black tree. The block tables provided the nodes for the control flow graph. The edges of the control flow graph are based on the logic of the code. Combined, the control flow graph represents a clear view of all possible data paths. From the control flow graph, edge coverage paths can be determined. The paths can be further expanded to complete test paths. With test paths established, input values to satisfy the test paths can be generated. A Junit test, using the generated input values, is created to test the code logic for faults. These tests simplify the identification of faults by catching semantics errors that do not generate system errors. The final step in the process employs the use of code coverage software. The monitoring of code coverage provides valuable information on the analysis of the Junit tests. A high coverage rate implies a verbose testing regime.

While test paths are useful at determining input values, the values are still heavily reliant on the skill of the test writer. Creating input values to traverse all the test paths varies in difficulty relative to the complexity of the method. Determining the correct assertions for the test path requires a firm understanding of the semantics of the program. The usefulness of method simplification is very apparent, in regards to test generation. The breakup of methods into simple forms allows for simplification of test generation and targeted testing.

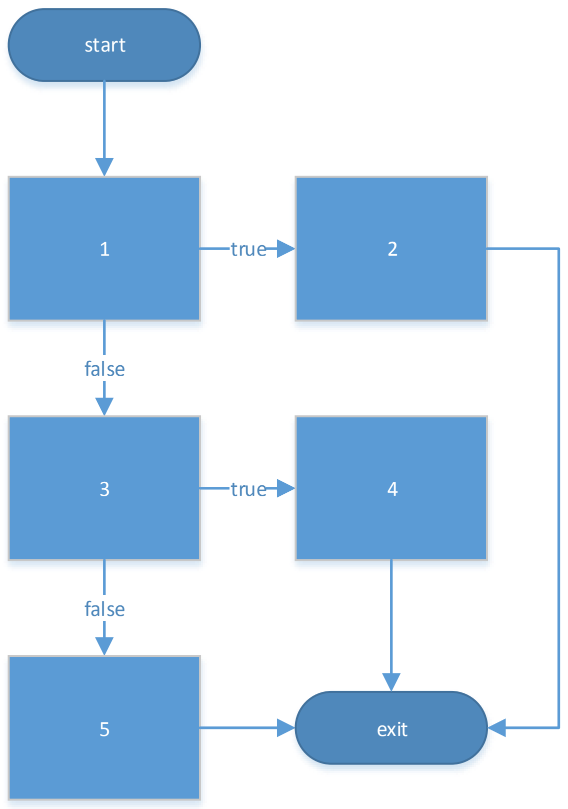
```
1   :**
2    * O(log(n))
3    * public String search(int k)
4    *
5    * returns the value of an item with key k if it exists in the tree
6    * otherwise, returns null
7    */
8   public String search(int k)
9   {
10          RBNode currNode = SearchNode(k,this.Root);
11          //return null if the tree is empty
12          if(currNode == null){
13                  return null;
14          }
15
16          //if not found return null
17          if(isNullNode(currNode) || currNode.Key != k){
18                  return null;
19          }
20
21          //return the value of the node with the key = k
22          return currNode.Value;
23  }
24
```

Block	Lines	Entry	Exit
1	10,12	10	12
2	13	13	13
3	17	17	17
4	18	18	18
5	22	22	22

Test path: t1 = [1,2]
T2 = [1,3,4]
T3 = [1,3,5]

Test data:

```
//check empty tree
Assert.assertEquals("Check for null return on empty tree", rbTree.search(100), null);

//check for value not in tree
rbTree.insert(100, "One hundred");
Assert.assertEquals("Check for null when value is not in tree", rbTree.search(50), null);

//check for value found to right
rbTree.insert(150, "One hundred and fifty");
Assert.assertEquals("Check for value of search return", rbTree.search(150), "One
hundred and fifty");
```


start
1
true
2
false
3
true
4
false
5
exit

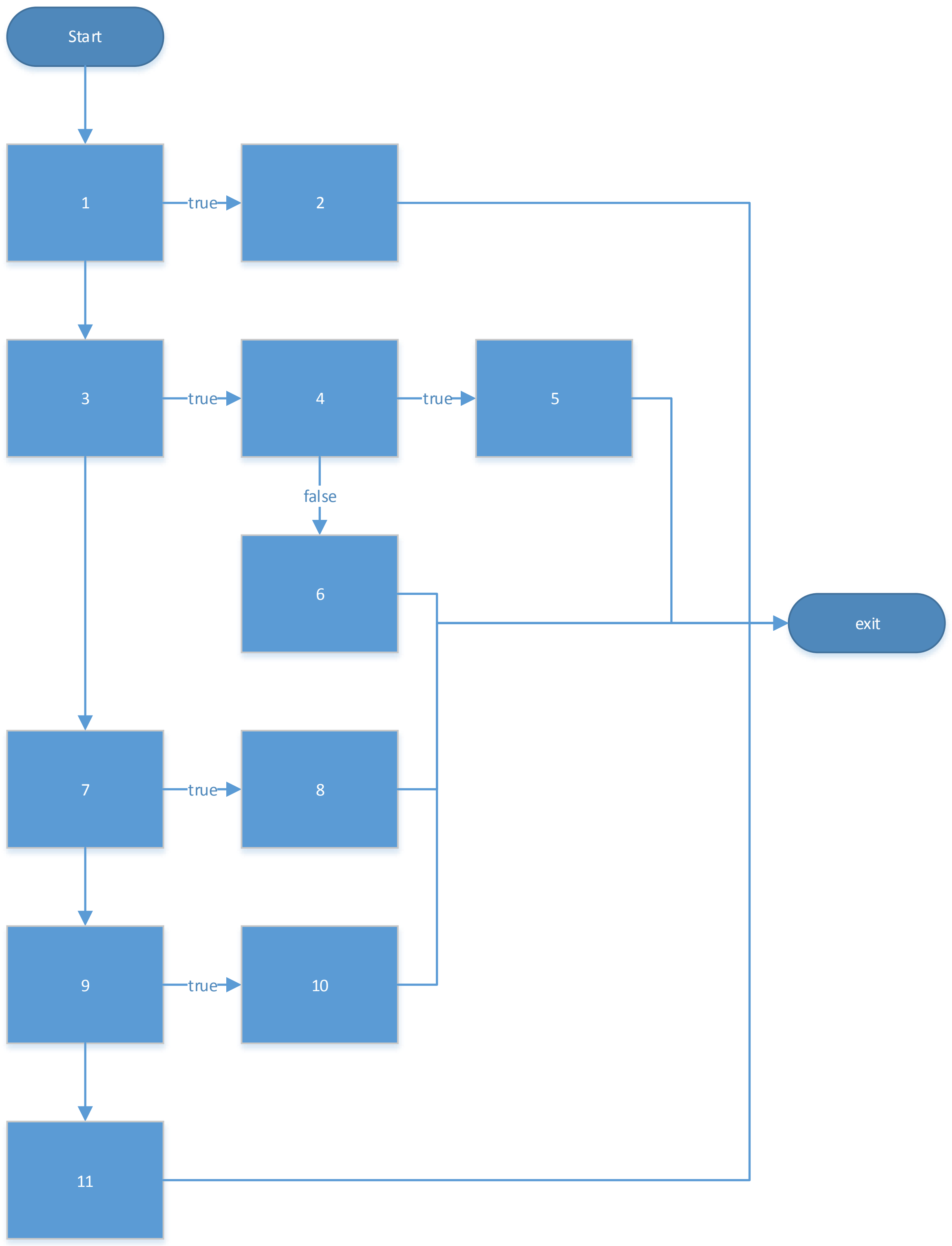
```
/**
 * O(log(n))
 * search for node
 * @param k - the key to look for
 * @param node - the node to start looking from
 * @return if found return the node with the same key and if not found returns the nearest node for the location
 * the node was supposed to be found. if the parent is the infinity node and not part of the tree
 * returns null
 */
private RBNode SearchNode(int k,RBNode node){
		//if the node that was given is the sentinel recursively call the function with the tree root
		if (isInfinityNode(node)){
				return SearchNode(k,node.Left);
		}

		if(isNullNode(node)){
				//the node is not exist in the tree. we will return null if the tree is empty.
				if(isInfinityNode(node.Parent)){
						return null;
				}else{
				//we could not find the node so we will return the parent of the imaginary node if it was exist in the tree
						return node.Parent;
				}
		}

		if(node.Key > k){
				//if the current node key is bigger then the key we look for
				//recursively call searchNode with the left child of the current node
				return SearchNode(k,node.Left);
		}

		if(node.Key < k){
				//if the current node key is lower then the key we look for
				//recursively call searchNode with the right child of the current node
				return SearchNode(k,node.Right);
		}

		//if the current node key is not bigger and not lower it equals to the key we are looking for
		return node;
}
```

Block	Lines	Entry	Exit
1	12	12	12
2	13	13	13
3	16	16	16
4	18	18	18
5	19	19	19
6	22	22	22
7	26	26	26
8	29	29	29
9	32	32	32
10	35	35	35
11	39	39	39

Test path:	t1 = [1,2]
	T2 = [1,3,4,5]
	T3 = [1,3,4,6]
	T4 = [1,3,7,8]
	T5 = [1,3,7,9,10]
	T6 = [1,3,7,9,11]

Test data:

```
rbTree.insert(150, "One hundred and fifty");
rbTree.insert(100, "One hundred");
rbTree.insert(250, "Two hundred and fifty");
rbTree.insert(50, "Fifty");
rbTree.insert(200, "Two hundred");
rbTree.insert(75, "Seventy-five");
rbTree.insert(300, "Three hundred");
rbTree.insert(25, "Twenty-five");
```


Start
1
true
2
3
true
4
true
5
false
6
exit
7
true
8
9
true
10
11

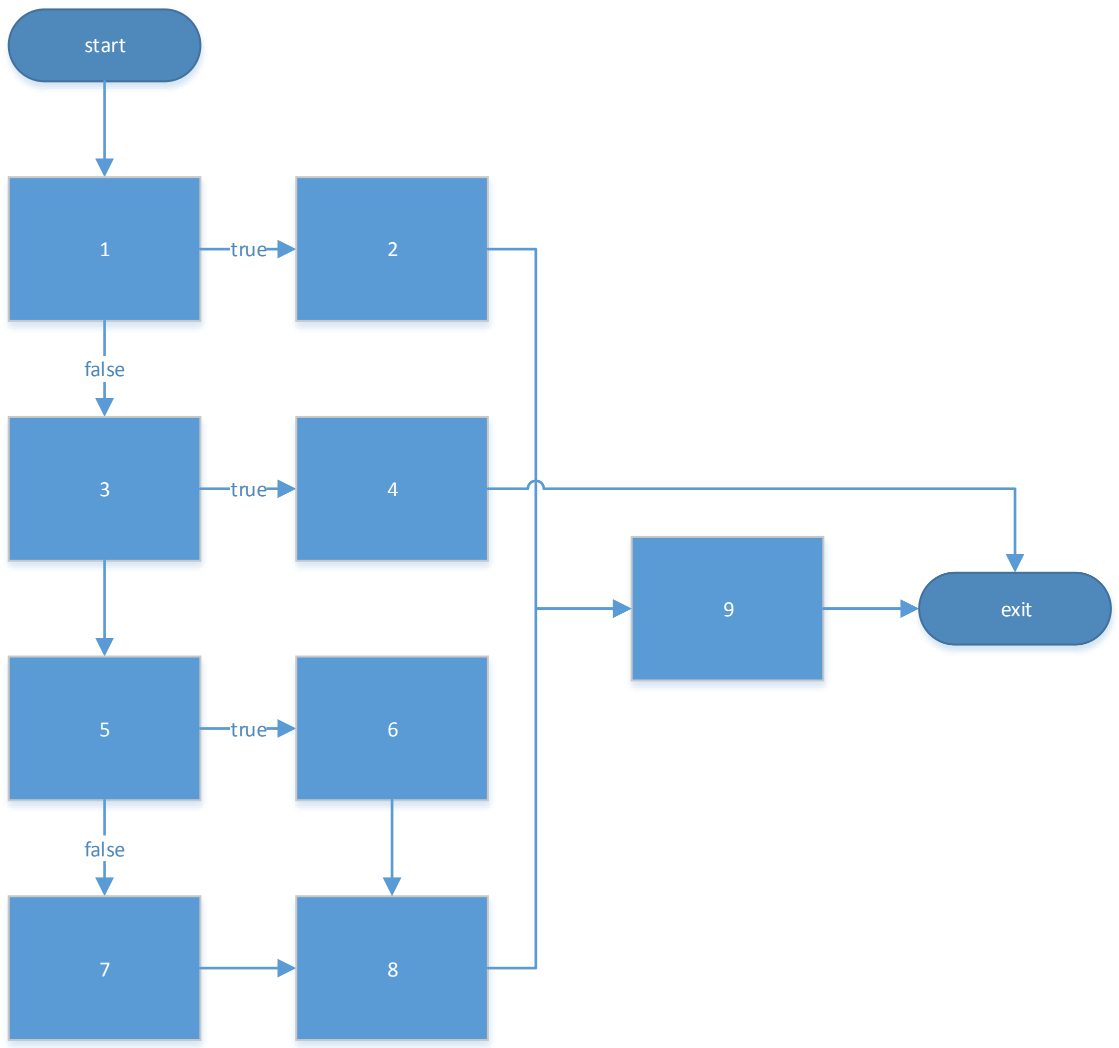
```
/**
 * O(log(n))
 * public int insert(int k, String v)
 *
 * inserts an item with key k and value v to the red black tree.
 * the tree must remain valid (keep its invariants).
 * returns the number of color switches, or 0 if no color switches were necessary.
 * returns -1 if an item with key k already exists in the tree.
 */
public int insert(int k, String v) {
		//find the place we want to insert the new node
		RBNode y = SearchNode(k,this.Root);
		//create new node which y is its parent
		RBNode z = new RBNode(v,k,y);
		//counter counts the number of color changes
		int counter = 0;
		if(y == null){
			//empty tree
			this.Root.Left = z;
			z.Black = true;
			z.Parent = this.Root;
		}else{
			if(y.Key == z.Key){
				//an item with the key k is  already exist in the tree
				return -1;
			}

			if(z.Key < y.Key){
				//z will be a left child
				y.Left = z;
			}else{
				//z will be a right child
				y.Right = z;
			}

			//calling the method that will check if the tree is valid and if not fix it
			counter = fixUpTree(z);
		}

		return counter;
}
```

Block	Lines	Entry	Exit
1	12,14,16,17	11	17
2	19,20,21	18	21
3	23	23	23
4	25	25	25
5	28	28	28
6	30	30	30
7	33	33	33
8	37	37	37
9	40	40	40

Test path: t1 = [1,2,9]
T2 = [1,3,4]
T3 = [1,3,5,6,8,9]
T4 = [1,3,5,7,8,9]

Test data:

```
//check initial insert
rbTree.insert(100, "One hundred");
Assert.assertEquals("Check for value for root node", rbTree.getRoot().Value, "One
hundred");

//check left insert
rbTree.insert(50, "Fifty");
Assert.assertEquals("Check for value of left node", rbTree.getRoot().Left.Value,
"Fifty");
Assert.assertEquals("Check value of left node parent",
rbTree.getRoot().Left.Parent.Value, "One hundred");

//check right insert
rbTree.insert(200, "Two hundred");
Assert.assertEquals("Check for value of right node", rbTree.getRoot().Right.Value, "Two
hundred");
Assert.assertEquals("Check value of right node parent",
rbTree.getRoot().Right.Parent.Value, "One hundred");

//check right right insert
rbTree.insert(250, "Two hundred and fifty");
Assert.assertEquals("Check for value of right right node",
rbTree.getRoot().Right.Right.Value, "Two hundred and fifty");
Assert.assertEquals("Check value of right right node parent",
rbTree.getRoot().Right.Right.Parent, rbTree.getRoot().Right);

//check right left insert
rbTree.insert(150, "One hundred and fifty");
Assert.assertEquals("Check for value of right left node",
rbTree.getRoot().Right.Left.Value, "One hundred and fifty");
Assert.assertEquals("Check value of right left node parent",
rbTree.getRoot().Right.Left.Parent, rbTree.getRoot().Right);

//check left right insert
rbTree.insert(75, "Seventy-five");
Assert.assertEquals("Check for value of left right node",
rbTree.getRoot().Left.Right.Value, "Seventy-five");
Assert.assertEquals("Check value of left right node parent",
rbTree.getRoot().Left.Right.Parent, rbTree.getRoot().Left);

//check left left insert
rbTree.insert(25, "Twenty-five");
Assert.assertEquals("Check for value of left left node",
rbTree.getRoot().Left.Left.Value, "Twenty-five");
Assert.assertEquals("Check value of left left node parent",
rbTree.getRoot().Left.Left.Parent, rbTree.getRoot().Left);

rbTree.insert(300, "Three hundred");
rbTree.insert(325, "Three hundred and twenty-five");

//check for duplicate value
Assert.assertEquals("Check for return value of duplicate insert", rbTree.insert(25,
"Twenty-five"), -1);
```


start
1
true
2
false
3
true
4
9
exit
5
true
6
false
7
8

```
/**
 * O(log(n))
 * keep the red and black rules. this method will be called when the rules were been
 * compromised and needs to be fix.
 * @param z - the node to start the fix from
 * @return the number of color changes that occured while fixing the tree
 */
private int fixUpTree (RBNode z){
        //counter will count the number of color changes
        int counter = 0;
        //run until there is no problem with the red rule
        while(!z.Parent.Black){
                if(z.Parent == z.Parent.Parent.Left){
                        //z parent is a left child
                        RBNode y = z.Parent.Parent.Right;
                        if(!y.Black){
                                //case 1: z'w parent and uncle are red
                                z.Parent.Black = true;
                                y.Black = true;
                                z.Parent.Parent.Black = false;
                                z = z.Parent.Parent;

                                //case 1 cost 3 color changes
                                counter += 3;
                        }else{
                                if(z == z.Parent.Right){
                                        //case 2: z is a right child and its uncle is red. need to left rotate
                                        z = z.Parent;
                                        leftRotate(z);
                                }
                                //case 3: z is a left child and its uncle is red. need to right rotate
                                z.Parent.Black = true;
                                z.Parent.Parent.Black = false;
                                rightRotate(z.Parent.Parent);
                                counter += 2;
                        }
                }else{
                        //z parent is a right child
                        RBNode y = z.Parent.Parent.Left;
                        if(!y.Black){
                                //case 1: z'w parent and uncle are red
                                z.Parent.Black = true;
                                y.Black = true;
                                z.Parent.Parent.Black = false;
                                z = z.Parent.Parent;

                                //case 1 cost 3 color changes
                                counter += 3;
                        }else{
                                if(z == z.Parent.Left){
                                        //case 2: z is a left child and its uncle is red. need to right rotate
                                        z = z.Parent;
                                        rightRotate(z);
                                }
                                //case 3: z is a right child and its uncle is red. need to left rotate
                                z.Parent.Parent.Black = false;
                                leftRotate(z.Parent.Parent);

                                counter++;
                        }
                }
            }
            if(!this.Root.Left.Black){
                counter++;
                this.Root.Left.Black = true;
            }

            return counter;
        }
```

Block	Lines	Entry	Exit
1	10	9	10
2	12	12	12
3	13	13	13
4	15,16	15	16
5	18,19,20,21,24	18	24
6	26	26	26
7	28,29	28	29
8	32,33,34,35	32	35
9	39,40	39	40
10	42,43,44,45,48	42	48
11	50	50	50
12	52,53	52	53
13	56,57,59	56	59
14	63	63	63
15	64,65	64	65
16	68	68	68

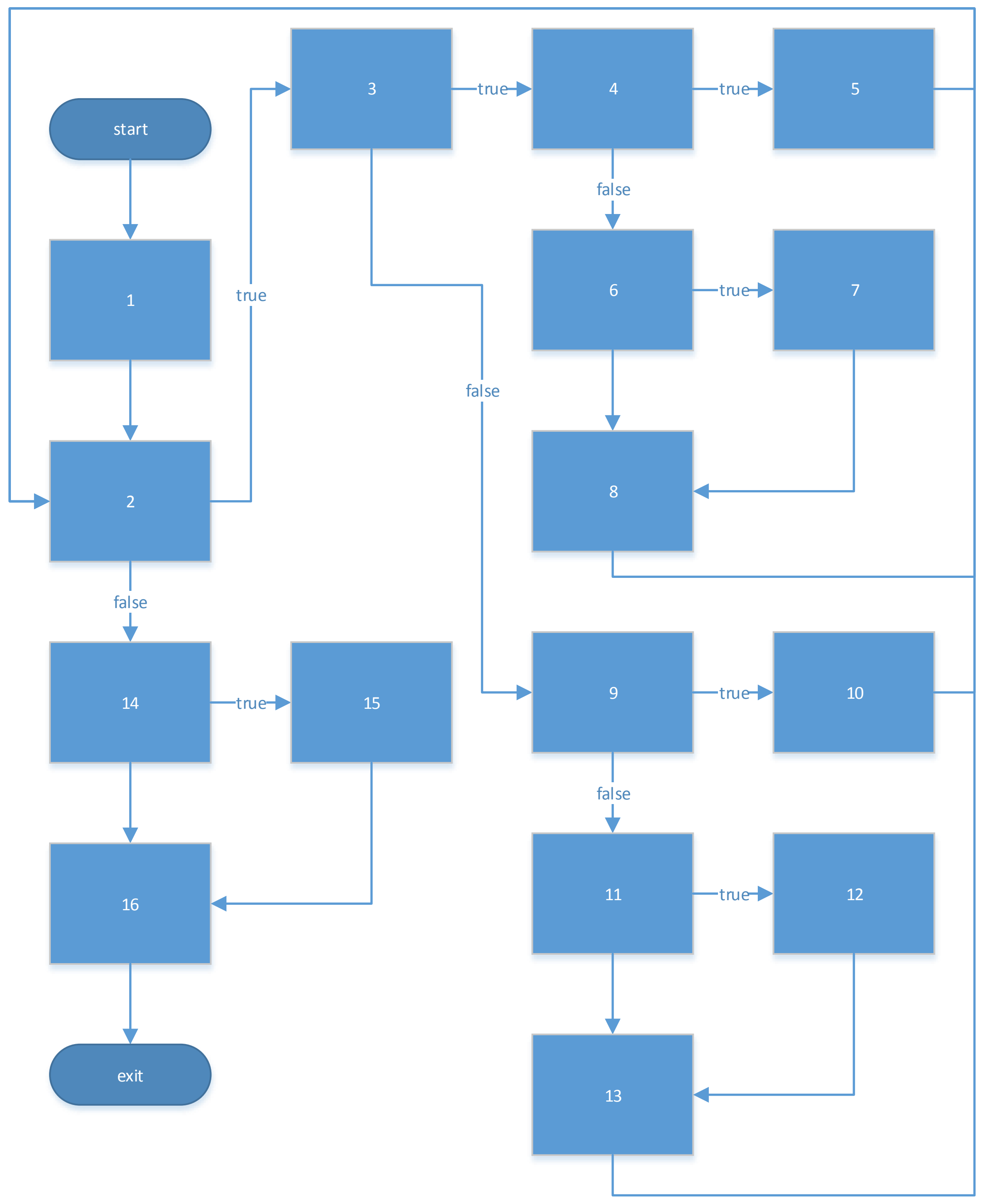
Test path: T1 = [1,2,3,4,5,2,14,15,16]
T2 = [1,2,3,4,6,7,8,2,14,16]
T3 = [1,2,3,4,6,8,2,14,16]
T4 = [1,2,3,9,10,2,14,16]
T5 = [1,2,3,9,11,12,13,2,14,16]
T6 = [1,2,3,9,11,13,2,14,16]

Corrections: z.Parent.Black = true; insert between line 55 & 56 to apply color change prior to rotation

counter += 2; replace line 59 as two colors have been changed

Test data:

```
RBTree.RBNode rootNode = rbTree.new RBNode("One hundred", 100, null, null,
rbTree.getRoot().Parent);
rbTree.getRoot().Parent.Left = rootNode;
rootNode.Left = rbTree.createNullNode(rootNode);
rootNode.Right = rbTree.createNullNode(rootNode);


RBTree.RBNode rightNode = rbTree.new RBNode("One hundred and fifty", 150, null, null
rootNode);
rootNode.Right = rightNode;
rightNode.Left = rbTree.createNullNode(rightNode);
rightNode.Right = rbTree.createNullNode(rightNode);

rbTree.fixUpTree(rightNode);

RBTree.RBNode rightRightNode = rbTree.new RBNode("Two hundred and fifty", 250, null,
null, rightNode);
rightNode.Right = rightRightNode;
rightRightNode.Left = rbTree.createNullNode(rightRightNode);
rightRightNode.Right = rbTree.createNullNode(rightRightNode);


rbTree.fixUpTree(rightRightNode);

RBTree.RBNode rightRightRightNode = rbTree.new RBNode("Three hundred and fifty", 350
null, null, rightRightNode);
rightRightNode.Right = rightRightRightNode;
rightRightRightNode.Left = rbTree.createNullNode(rightRightRightNode);
rightRightRightNode.Right = rbTree.createNullNode(rightRightRightNode);

rbTree.fixUpTree(rightRightRightNode);
rbTree.insert(225, "Two hundred and twenty-five");
rbTree.insert(325, "Three hundred and twenty-five");
rbTree.insert(230, "Two hundred and thirty");
rbTree.insert(300, "Three hundred");
rbTree.insert(400, "Four hundred");
rbTree.insert(450, "Four hundred and fifty");
Assert.assertEquals("Check value of root", rbTree.getRoot().Value, "Two hundred and
fifty");
Assert.assertEquals("Check value of left node of root", rbTree.getRoot().Left.Value,
"One hundred and fifty");
Assert.assertEquals("Check value of right node of root", rbTree.getRoot().Right.Valu
"Three hundred and twenty-five");
Assert.assertEquals("Check value of right node of right node of root",
rbTree.getRoot().Right.Right.Value, "Four hundred");
```


start
1
2
true
3
true
4
true
5
false
6
true
7
false
8
false
14
true
15
16
exit
9
true
10
false
11
true
12
13

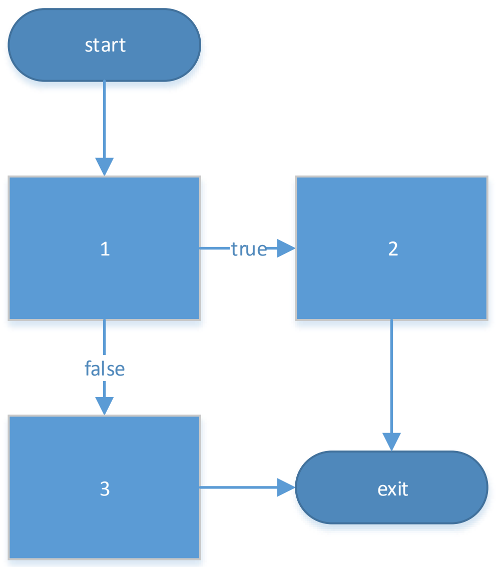
```
1   /**
2    * O(1)
3    * put y instead of x
4    * @param x - the original child
5    * @param y - the child after the change
6    */
7   private void transplate(RBNode x, RBNode y){
8           if (x == x.Parent.Left){
9                   leftChild(x.Parent,y);
10          }else{
11                  rightChild(x.Parent,y);
12          }
13  }
14
```

Block	Lines	Entry	Exit
1	8	8	8
2	9	9	9
3	11	11	11

Test path: T1 = [1,2]
T2 = [1,3]

Test data:

```
RBTree.RBNode rootNode = rbTree.new RBNode("One hundred", 100, null, null,
rbTree.getRoot().Parent);
rbTree.getRoot().Parent.Left = rootNode;
rootNode.Left = rbTree.createNullNode(rootNode);
rootNode.Right = rbTree.createNullNode(rootNode);


RBTree.RBNode rightNode = rbTree.new RBNode("One hundred and fifty", 150, null, null,
rootNode);
rootNode.Right = rightNode;
rightNode.Left = rbTree.createNullNode(rightNode);
rightNode.Right = rbTree.createNullNode(rightNode);


RBTree.RBNode rightRightNode = rbTree.new RBNode("Two hundred and fifty", 250, null,
null, rightNode);
rightNode.Right = rightRightNode;
rightRightNode.Left = rbTree.createNullNode(rightRightNode);
rightRightNode.Right = rbTree.createNullNode(rightRightNode);


rbTree.leftRotate(rbTree.getRoot());
```


start
1
true
2
false
3
exit

```
/**
 * O(log(n))
 * public int delete(int k)
 *
 * deletes an item with key k from the binary tree, if it is there;
 * the tree must remain valid (keep its invariants).
 * returns the number of color switches, or 0 if no color switches were needed.
 * returns -1 if an item with key k was not found in the tree.
 */
public int delete(int k)
{
	int counter = 0;
	RBNode z = SearchNode(k, this.Root);
	if(z.Key != k){
		//item with the key k could not be found
		return -1;
	}

	RBNode x;
	RBNode y = z;
	boolean isBlackOriginalY = y.Black;

	//z is the node we want to delete
	if(isNullNode(z.Left)){
		x = z.Right;
		transplate(z,z.Right);
	}else if(isNullNode(z.Right)){
		x = z.Left;
		transplate(z,z.Left);
	}else{
		y = minimumNode(z.Right);
		isBlackOriginalY = y.Black;
		x = y.Right;
		if(y.Parent == z){
			x.Parent = y;
		}else{
			transplate(y,y.Right);
			y.Right = z.Right;
			y.Right.Parent = y;
		}
		transplate(z,y);
		y.Left = z.Left;
		y.Black = z.Black;
	}

	if(isBlackOriginalY){
		//we have a problem with the black rule that needs to be fixed
		counter = deleteFixup(x);
	}

	return counter;
}
```

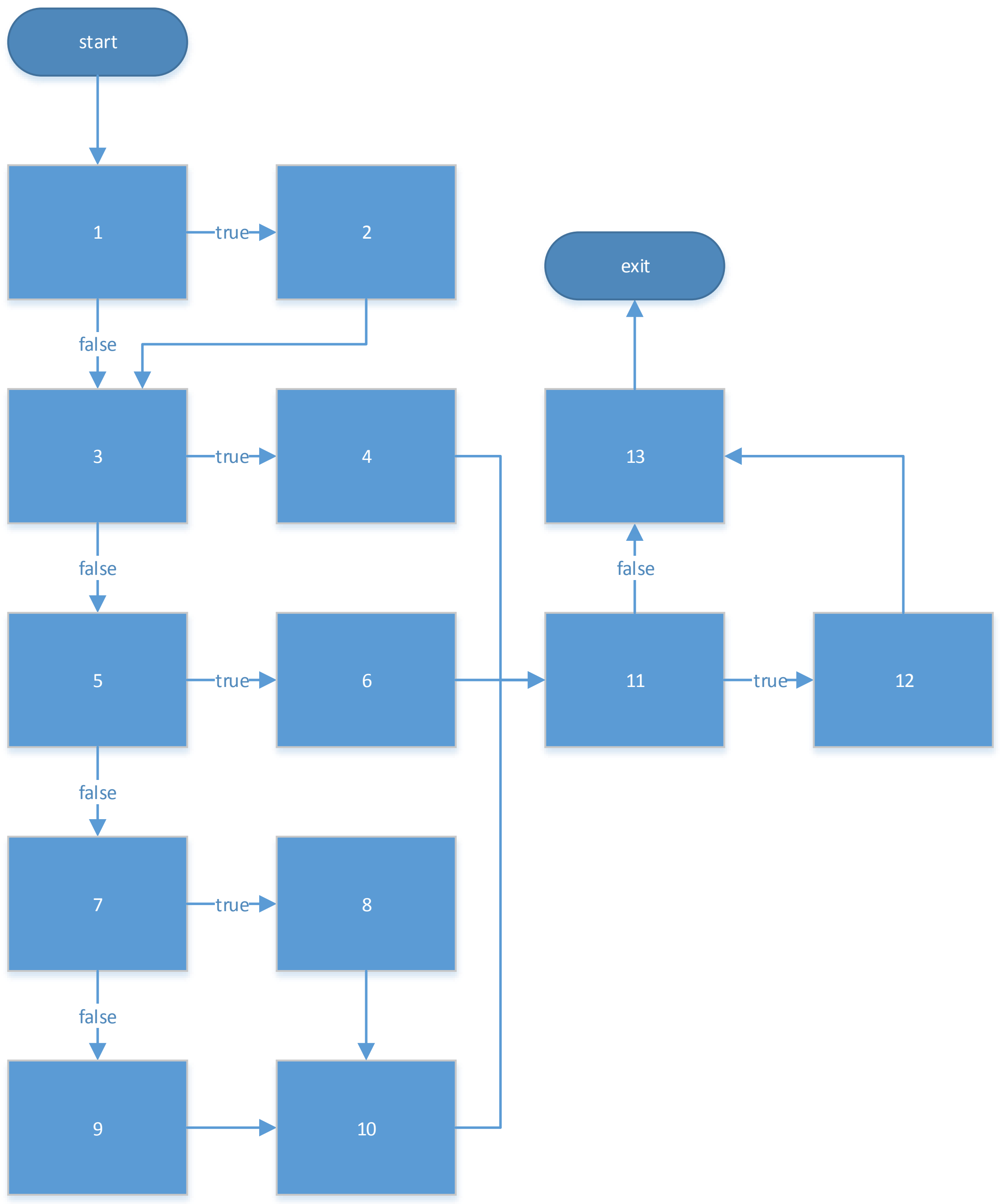
Block	Lines	Entry	Exit
1	12,13,14	12	14
2	16	16	16
3	19,20,21,24	19	24
4	25,26	25	26
5	27	27	27
6	28,29	28	29
7	31,32,33,34	31	34
8	35	35	35
9	37,38,39	37	39
10	41,42,43	41	43
11	46	46	46
12	48	48	48
13	51	51	51

Test path: T1 = [1,2,3,4,11,12,13]
T2 = [1,3,5,6,11,13]
T3 = [1,3,5,7,8,10,11,13]
T4 = [1,3,5,7,9,10,11,13]

Corrections: if(z == null || z.Key != k){ replace line 14 to check for null (an empty tree)
y.Left.Parent = y; insert between lines 42 & 43 to set proper parent after deletion

Test data:

```
Assert.assertEquals("Check delete on empty tree", rbTree.delete(100), -1);

rbTree.insert(100, "One hundred");
rbTree.delete(100);
Assert.assertEquals("Delete one node", rbTree.empty(), true);

rbTree.insert(100, "One hundred");
rbTree.insert(150, "One hundred and fifty");
rbTree.delete(100);
Assert.assertEquals("Check after left null right child delete", rbTree.getRoot().Value,
"One hundred and fifty");

rbTree.insert(50, "Fifty");
rbTree.delete(150);
Assert.assertEquals("Check after right null left child  delete",
rbTree.getRoot().Value, "Fifty");


rbTree.insert(25, "Twenty-five");
rbTree.insert(100, "One hundred");
rbTree.delete(50);
Assert.assertEquals("Check after right and left child delete", rbTree.getRoot().Value,
"One hundred");
Assert.assertEquals("Check after right and left child delete",
rbTree.getRoot().Left.Value, "Twenty-five");
Assert.assertEquals("Check after right and left child delete",
rbTree.getRoot().Right.Value, "N");
Assert.assertEquals("Check after right and left child delete",
rbTree.getRoot().Left.Parent.Value, "One hundred");
Assert.assertEquals("Check after right and left child delete",
rbTree.getRoot().Right.Parent.Value, "One hundred");


rbTree.insert(150, "One hundred and fifty");
rbTree.insert(75, "Seventy-five");
rbTree.insert(125, "One hundred and twenty-five");
rbTree.delete(100);
Assert.assertEquals("Check after right and left child delete", rbTree.getRoot().Value,
"One hundred and twenty-five");
Assert.assertEquals("Check after right and left child delete",
rbTree.getRoot().Left.Value, "Twenty-five");
Assert.assertEquals("Check after right and left child delete",
rbTree.getRoot().Right.Value, "One hundred and fifty");
Assert.assertEquals("Check after right and left child delete",
rbTree.getRoot().Left.Parent.Value, "One hundred and twenty-five");
Assert.assertEquals("Check after right and left child delete",
rbTree.getRoot().Right.Parent.Value, "One hundred and twenty-five");
```


start
1
true
2
exit
false
3
true
4
13
false
5
true
6
11
true
12
false
7
true
8
false
9
10

```java
/**
 * O(log(n))
 * fix the red and black rules after deleting a node.
 * @param x - the node to start the fixing from
 * @return number of color changes made while fixing the tree
 */
private int deleteFixup(RBNode x){
		//number of color changes
		int counter = 0;
		//run until x is the tree root and as long as x is black
		while(x != this.Root.Left && x.Black){
			if(x == x.Parent.Left){
				//x is a left child
				RBNode w = x.Parent.Right;
				if(!w.Black){
					//case 1
					w.Black = true;
					x.Parent.Black = false;
					leftRotate(x.Parent);
					w = x.Parent.Right;
					counter += 2;
				}
				if(w.Left.Black && w.Right.Black){
					//case 2
					w.Black = false;
					x = x.Parent;
					counter += 1;
				}else
				{
					if(w.Right.Black){
						//case 3
						w.Left.Black = true;
						w.Black = false;
						rightRotate(w);
						w = x.Parent.Right;
						counter +=2;
					}
					//case 4
					w.Black = x.Parent.Black;
					x.Parent.Black = true;
					w.Right.Black = true;
					leftRotate(x.Parent);
					x = this.Root.Left;
					counter += 1;
				}
			}else{
				//x is a right child
				RBNode w = x.Parent.Left;
				if(!w.Black){
					//case 1
					w.Black = true;
					x.Parent.Black = false;
					rightRotate(x.Parent);
					w = x.Parent.Left;
					counter += 2;
				}
				if(w.Left.Black && w.Right.Black){
					//case 2
```

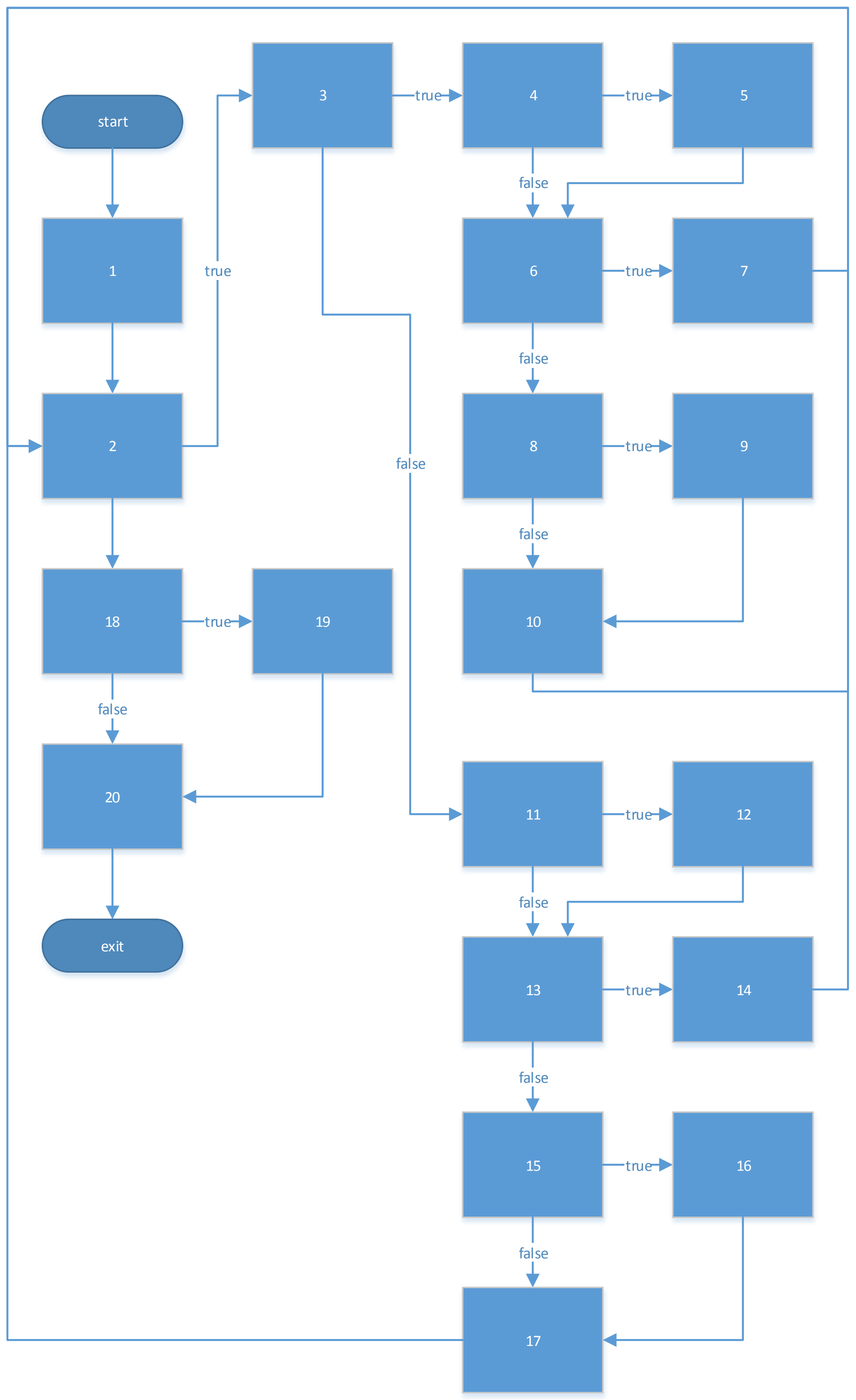
```
                w.Black = false;
                x = x.Parent;
                counter += 1;
            }else
            {
                if(w.Left.Black){
                    //case 3
                    w.Right.Black = true;
                    w.Black = false;
                    leftRotate(w);
                    w = x.Parent.Left;
                    counter +=2;
                }
                //case 4
                w.Black = x.Parent.Black;
                x.Parent.Black = true;
                w.Left.Black = true;
                rightRotate(x.Parent);
                x = this.Root.Left;
                counter +=1;
            }
        }
    }
    if(x.Black == false){
        x.Black = true;
        counter += 1;
    }

    return counter;
}
```

Block	Lines	Entry	Exit
1	9	9	9
2	11	11	11
3	12	12	12
4	14,15	14	15
5	17,18,19,20,21	17	21
6	23	23	23
7	25,26,27	25	27
8	30	30	30
9	32,33,34,35,36	32	36
10	39,40,41,42,43,44	39	44
11	48,49	48	49
12	51,52,53,54,55	51	55
13	57	57	57
14	59,60,61	59	61
15	64	64	64
16	66,67,68,69,70	66	70
17	73,74,75,76,77,78	73	78
18	82	82	82
19	83,84	83	84
20	87	87	87

Test path: T1 = [1,2,18,19,20]
T2 = [1,2,3,4,5,6,7,2,18,20]
T3 = [1,2,3,4,6,8,9,10,2,18,20]
T4 = [1,2,3,4,6,8,10,2,18,20]
T5 = [1,2,3,11,12,13,14,2,18,20]
T6 = [1,2,3,11,13,15,16,17,2,18,20]
T6 = [1,2,3,11,13,15,17,2,18,20]

Test data:

```
//create initial tree
createTree();

//right child case 4
rbTree.delete(100);
Assert.assertEquals("Check deleted 100", null, rbTree.search(100));

//left child case 2
rbTree.delete(25);
Assert.assertEquals("Check deleted 25", null, rbTree.search(25));


rbTree.insert(25, "Twenty-five");
rbTree.insert(100, "One hundred");
rbTree.insert(325, "Three hundred and twenty-five");

//left child case 4
rbTree.delete(25);
Assert.assertEquals("Check deleted 25", null, rbTree.search(25));
Assert.assertEquals("Check root", "One hundred and fifty", rbTree.getRoot().Value);
rbTree.insert(30, "Thirty");
rbTree.insert(20, "Twenty");
rbTree.insert(40, "Forty");
rbTree.insert(35, "Thirty-five");
rbTree.insert(50, "Fifty");
rbTree.delete(20);
Assert.assertEquals("Check deleted 20", null, rbTree.search(20));

rbTree.insert(30, "Thirty");
rbTree.insert(20, "Twenty");
rbTree.insert(50, "Fifty");
rbTree.insert(10, "Ten");
rbTree.insert(5, "Five");
rbTree.insert(3, "Three");
rbTree.delete(50);
Assert.assertEquals("Check deleted 50", null, rbTree.search(50));
```


start
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
exit
true
false

```
1    /**
2     *  O(log(n))
3     * public String min()
4     *
5     * Returns the value of the item with the smallest key in the tree,
6     * or null if the tree is empty
7     */
8    public String min()
9    {
10             return minimumNode(this.Root.Left).Value;
11   }
12
```

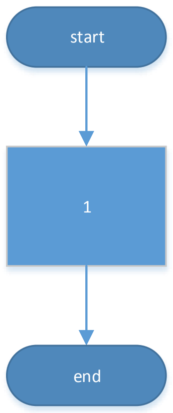
Block	Lines	Entry	Exit
1	10	10	10

Test path: T1 = [1]

Corrections: if(this.empty()){return null;} insert between lines 9 & 10 check for empty tree

Test data:

```
Assert.assertEquals("Check for empty tree", rbTree.min(), null);

rbTree.insert(100, "One hundred");
Assert.assertEquals("Check tree with one value", rbTree.min(), "One hundred");

rbTree.insert(50, "Fifty");
Assert.assertEquals("Check for one insert to left", rbTree.min(), "Fifty");

rbTree.insert(200, "Two hundred");
rbTree.insert(250, "Two hundred and fifty");
rbTree.insert(350, "Three hundred and fifty");

//Check insert after rotation
rbTree.insert(25, "Twenty-five");
Assert.assertEquals("Check for value of right node after rotation", rbTree.min(),
"Twenty-five");
```



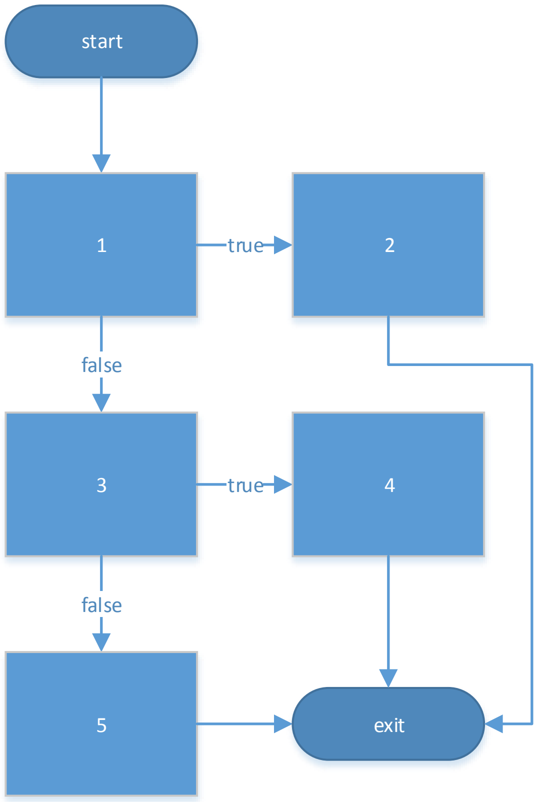
```
1   /**
2    *  O(log(n))
3    * return the node with the minimal key in the tree
4    * @param node - the node to start looking from
5    * @return the minimal node of the tree
6    */
7   private static RBNode minimumNode(RBNode node){
8           if(isNullNode(node)){
9                   return null;
10          }
11
12          //next smaller node is null node so this is the minimal node
13          if(isNullNode(node.Left)){
14                  return node;
15          }
16
17          return minimumNode(node.Left);
18  }
19
```

Block	Lines	Entry	Exit
1	8	8	8
2	9	9	9
3	13	13	13
4	14	14	14
5	17	17	17

Test path: T1 = [1,2]
T2 = [1,3,4]
T3 = [1,3,5]

Test data:

```
Assert.assertEquals("Check for empty tree", rbTree.min(), null);

rbTree.insert(100, "One hundred");
Assert.assertEquals("Check tree with one value", rbTree.min(), "One hundred");

rbTree.insert(50, "Fifty");
Assert.assertEquals("Check for one insert to left", rbTree.min(), "Fifty");

rbTree.insert(200, "Two hundred");
rbTree.insert(250, "Two hundred and fifty");
rbTree.insert(350, "Three hundred and fifty");

//Check insert after rotation
rbTree.insert(25, "Twenty-five");
Assert.assertEquals("Check for value of right node after rotation", rbTree.min(),
"Twenty-five");
```



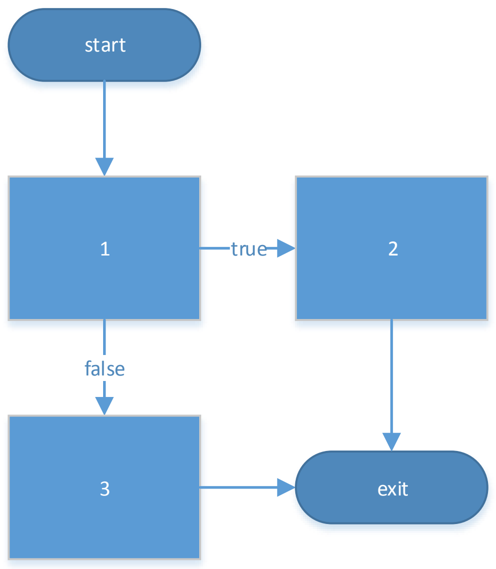
```
1   /**
2    *  O(log(n))
3    * public String max()
4    *
5    * Returns the value of the item with the largest key in the tree,
6    * or null if the tree is empty
7    */
8   public String max()
9   {
10          if(this.empty()){
11                  return null;
12          }
13
14          return maxValue(this.Root.Left);
15  }
16
```

Block	Lines	Entry	Exit
1	10	10	10
2	11	11	11
3	14	14	14

Test path: T1 = [1,2]
T2 = [1,3]

Test data:

```
Assert.assertEquals("Check for empty tree", rbTree.max(), null);

rbTree.insert(100, "One hundred");
Assert.assertEquals("Check tree with one value", rbTree.max(), "One hundred");

rbTree.insert(200, "Two hundred");
Assert.assertEquals("Check for one insert to right", rbTree.max(), "Two hundred");

rbTree.insert(50, "Fifty");
rbTree.insert(25, "Twenty-five");
rbTree.insert(10, "Ten");

//Check insert after rotation
rbTree.insert(300, "Three hundred");
Assert.assertEquals("Check for value of right node after rotation", rbTree.max(),
"Three hundred");
```


start
1
true
2
false
3
exit

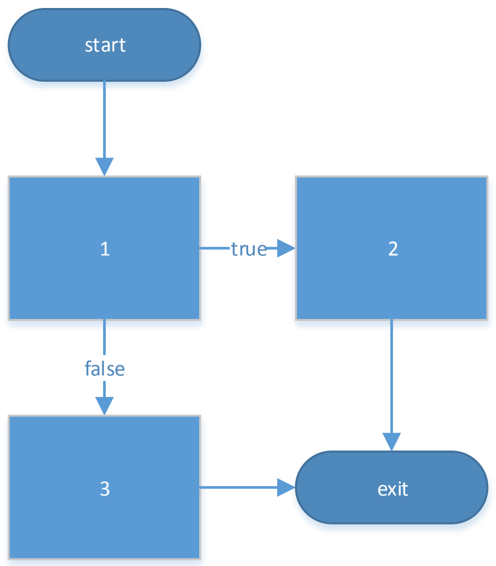
```
1   /**
2    *  O(log(n))
3    * Returns the value of the item with the largest key in the tree,
4    * @param node the node to start looking from
5    * @return the value of the minimal node in the tree
6    */
7   public static String maxValue(RBNode node){
8           //next node is null node so return this node
9           if(isNullNode(node.Right)){
10                  return node.Value;
11          }
12
13          return maxValue(node.Right);
14  }
15
```

Block	Lines	Entry	Exit
1	9	9	9
2	10	10	10
3	13	13	13

Test path: T1 = [1,2]
T2 = [1,3]

Test data:

```
Assert.assertEquals("Check for empty tree", rbTree.max(), null);

rbTree.insert(100, "One hundred");
Assert.assertEquals("Check tree with one value", rbTree.max(), "One hundred");
```

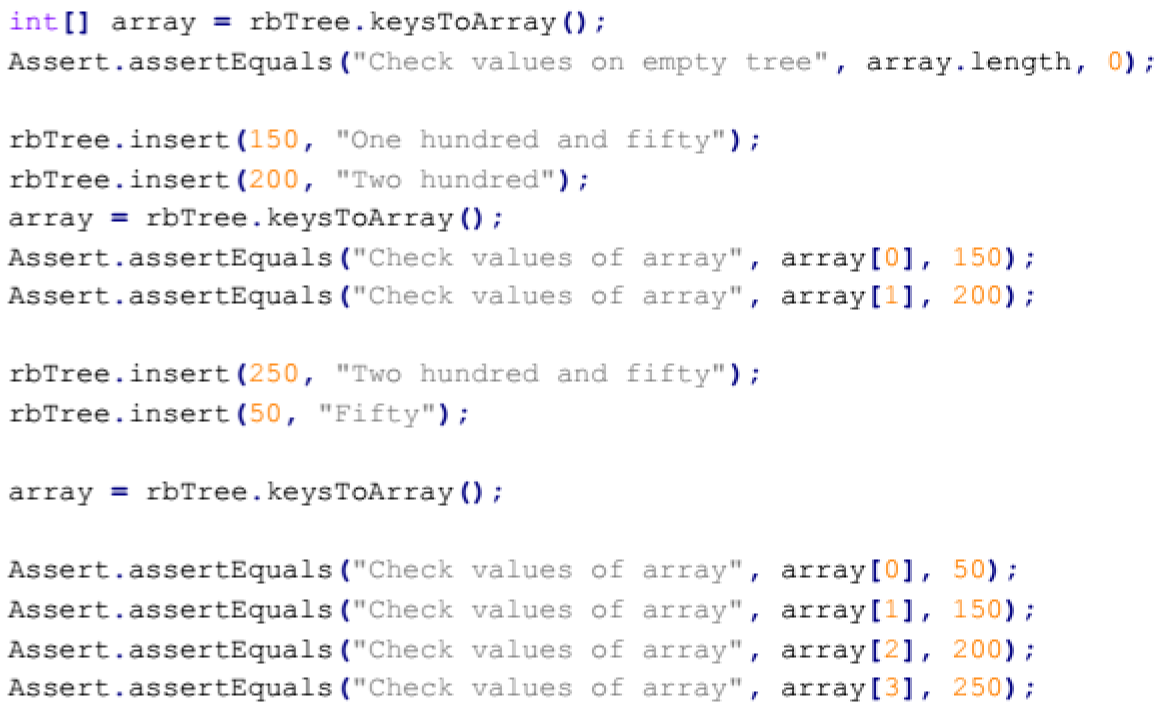


```
1   /**
2    *  O(n)
3    * public int[] keysToArray()
4    *
5    * Returns a sorted array which contains all keys in the tree,
6    * or an empty array if the tree is empty.
7    */
8   public int[] keysToArray()
9   {
10          String keysString = ElementsToString(this.Root,true);
11          //keysString is a string of all the keys in the tree separated by ","
12          if(keysString.equals("")){
13                  return new int[0];
14          }
15          return ArrayOfStringsToArrayOfInts(keysString.split(","));
16  }
17
```

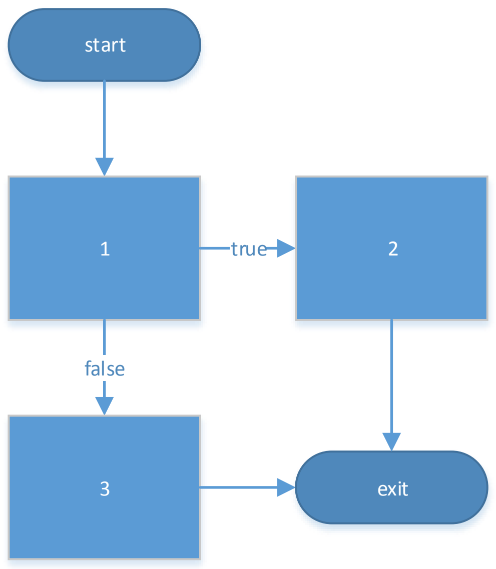
Block	Lines	Entry	Exit
1	10,12	10	12
2	13	13	13
3	15	15	15

Test path: T1 = [1,2]
T2 = [1,3]

Test data:



```
int[] array = rbTree.keysToArray();
Assert.assertEquals("Check values on empty tree", array.length, 0);

rbTree.insert(150, "One hundred and fifty");
rbTree.insert(200, "Two hundred");
array = rbTree.keysToArray();
Assert.assertEquals("Check values of array", array[0], 150);
Assert.assertEquals("Check values of array", array[1], 200);

rbTree.insert(250, "Two hundred and fifty");
rbTree.insert(50, "Fifty");

array = rbTree.keysToArray();

Assert.assertEquals("Check values of array", array[0], 50);
Assert.assertEquals("Check values of array", array[1], 150);
Assert.assertEquals("Check values of array", array[2], 200);
Assert.assertEquals("Check values of array", array[3], 250);
```


start
1
true
2
false
3
exit

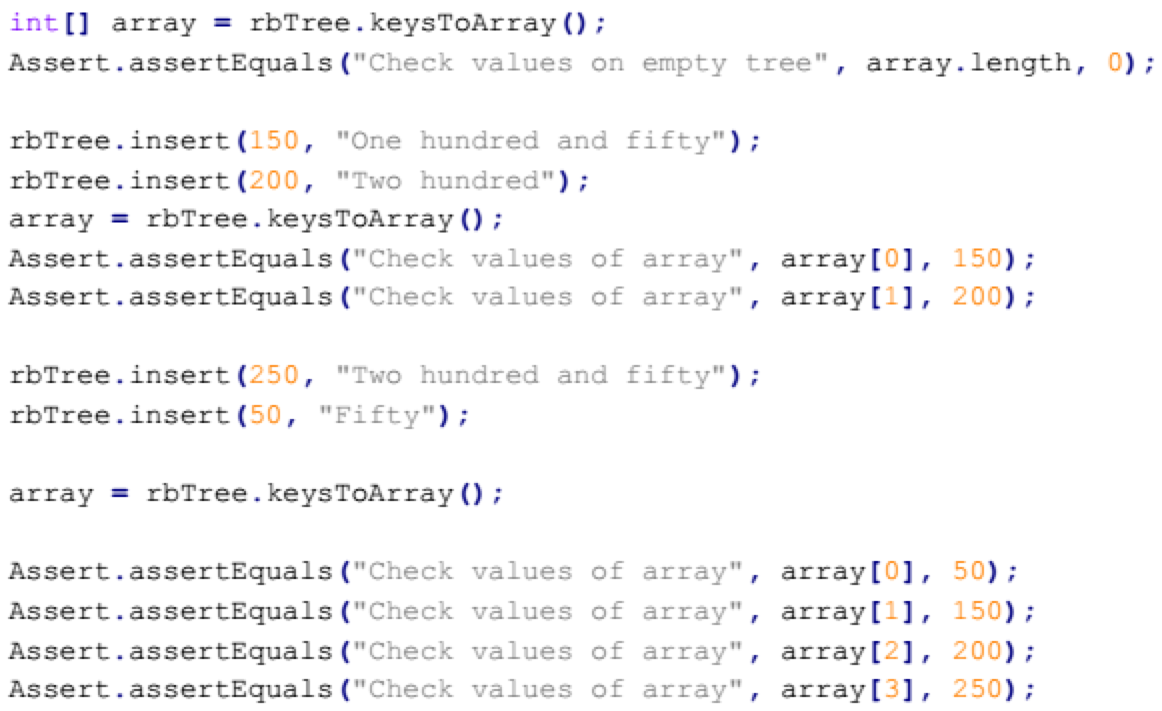
```
1    /**
2     * O(strArr length)
3     * @param strArr - array of string values (that can be parsed into integers)
4     * @return array of the same values cast as int
5     */
6    private int[] ArrayOfStringsToArrayOfInts(String[] strArr){
7            int[] arr = new int[strArr.length];
8            for(int i=0; i<= strArr.length;i++){
9                    arr[i] = Integer.parseInt(strArr[i]);
10           }
11           return arr;
12   }
13
```

Block	Lines	Entry	Exit
1	7	7	7
2	8	8	8
3	9	9	9
4	11	11	11

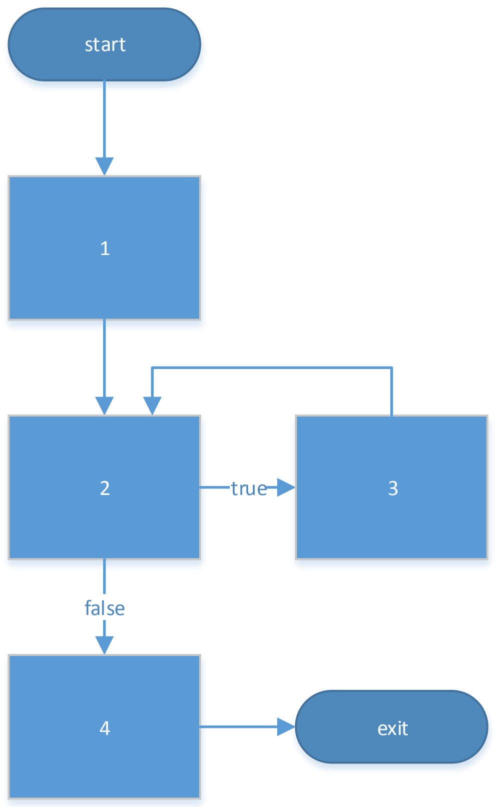
Test path: T1 = [1,2,3,2,4]

Corrections: for(int i=0; i< strArr.length;i++){ change on line 8 <= to < to avoid index out of range

Test data:



```
int[] array = rbTree.keysToArray();
Assert.assertEquals("Check values on empty tree", array.length, 0);

rbTree.insert(150, "One hundred and fifty");
rbTree.insert(200, "Two hundred");
array = rbTree.keysToArray();
Assert.assertEquals("Check values of array", array[0], 150);
Assert.assertEquals("Check values of array", array[1], 200);

rbTree.insert(250, "Two hundred and fifty");
rbTree.insert(50, "Fifty");

array = rbTree.keysToArray();

Assert.assertEquals("Check values of array", array[0], 50);
Assert.assertEquals("Check values of array", array[1], 150);
Assert.assertEquals("Check values of array", array[2], 200);
Assert.assertEquals("Check values of array", array[3], 250);
```


start
1
2
true
3
false
4
exit

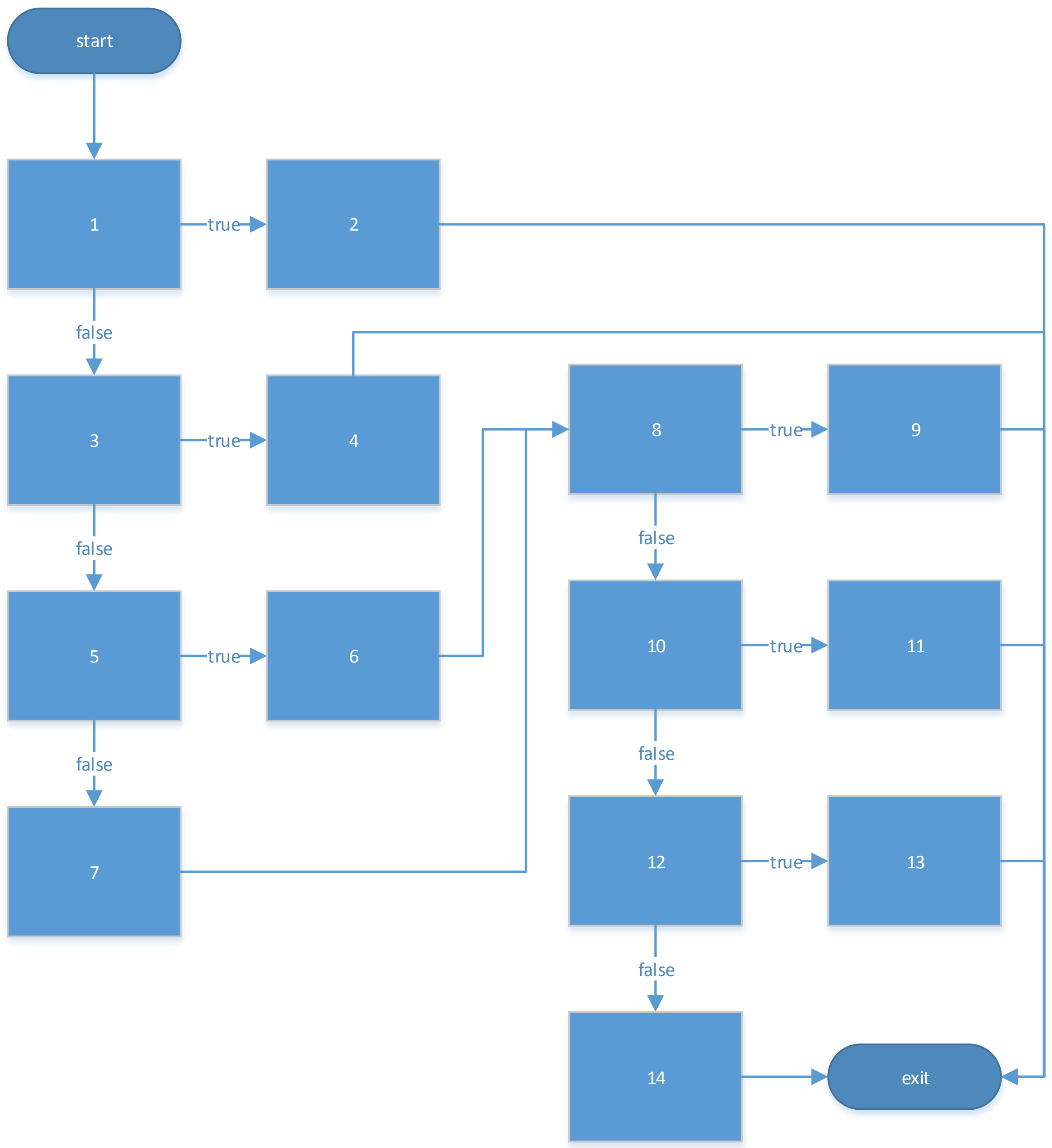
```
/**
 * O(n)
 * @param node - the node we start from
 * @param key - if true returns the string with keys elements else return the string with value elements
 * @return a string with values\keys seperated with ,
 */
private String ElementsToString(RBNode node,boolean key){
		if(isNullNode(node)){
			return "";
		}

		//if called with the infinity node recursively call with the real root
		if(isInfinityNode(node)){
			return ElementsToString(node.Left,key);
		}

		String str;
		if(key){
			//str will be the key
			str = Integer.toString(node.Key);
		}else{
			//str will be the value
			str = node.Value;
		}

		//str is the element needs to be insert in the returning string
		if(!isNullNode(node.Left) && !isNullNode(node.Right)){
			//return all the elements, bigger and smaller then the current node
			return ElementsToString(node.Left,key) + "," + str + "," + ElementsToString(node.Right,key);
		}

		if(!isNullNode(node.Right)){
			//there is no lower elements return the current element + all the elements that bigger
			return str + "," + ElementsToString(node.Right,key);
		}

		if(!isNullNode(node.Left)){
			//there is no higher elements. return all the elements that are lower than the current element
			return ElementsToString(node.Left,key) + "," + str;
		}

		return str;
}
```

Block	Lines	Entry	Exit
1	8	8	8
2	9	9	9
3	13	13	13
4	14	14	14
5	17,18	17	18
6	20	20	20
7	23	23	23
8	27	27	27
9	29	29	29
10	32	32	32
11	34	34	34
12	37	37	37
13	39	39	39
14	42	42	42

Test path: T1 = [1,2]
T2 = [1,3,4]
T3 = [1,3,5,6,8,9]
T4 = [1,3,5,7,8,10,11]
T5 = [1,3,5,6,8,10,12,13]
T6 = [1,3,5,6,8,10,12,14]

Test data:

```
int[] array = rbTree.keysToArray();
Assert.assertEquals("Check values on empty tree", array.length, 0);

rbTree.insert(150, "One hundred and fifty");
rbTree.insert(200, "Two hundred");
array = rbTree.keysToArray();
Assert.assertEquals("Check values of array", array[0], 150);
Assert.assertEquals("Check values of array", array[1], 200);

rbTree.insert(250, "Two hundred and fifty");
rbTree.insert(50, "Fifty");

array = rbTree.keysToArray();

Assert.assertEquals("Check values of array", array[0], 50);
Assert.assertEquals("Check values of array", array[1], 150);
Assert.assertEquals("Check values of array", array[2], 200);
Assert.assertEquals("Check values of array", array[3], 250);
```


start
1
true
2
false
3
true
4
false
5
true
6
false
7
8
true
9
false
10
true
11
false
12
true
13
false
14
exit

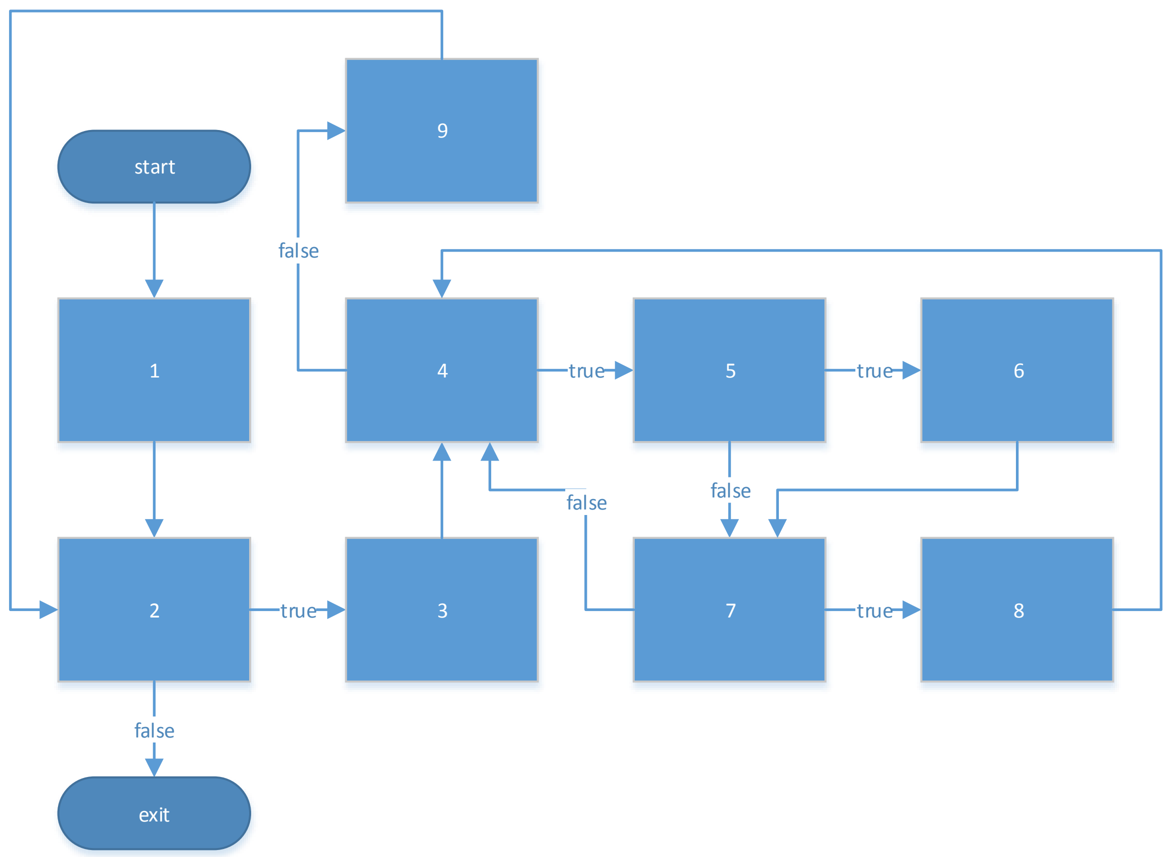
```
1   /**
2    * O(n)
3    * print out this RBTree level by level.
4    */
5   public void print() {
6           Queue<RBNode> queue = new LinkedList<>();
7           queue.add(this.Root);
8           while (!queue.isEmpty()) {
9                   int size = queue.size();
10                  for (int i = 0; i < size; i++) {
11                          RBNode curNode = queue.poll();
12                          System.out.print(curNode.Key + " " + curNode.Value + " " + (curNode.Black? "black" : "red")+" | ");
13                          if (curNode.Left != null) {
14                                  queue.add(curNode.Left);
15                          }
16                          if (curNode.Right != null) {
17                                  queue.add(curNode.Right);
18                          }
19                  }
20                  System.out.print('\n');
21          }
22  }
23
```

Block	Lines	Entry	Exit
1	6,7	6	7
2	8	8	8
3	9	9	9
4	10	10	10
5	11,12,13	11	13
6	14	14	14
7	16	16	16
8	17	17	17
9	20	20	20

Test path: T1 = [1,2]
T2 = [1,2,3,4,5,6,7,8,4,9,2]
T3 = [1,2,3,4,5,7,4,9,2]

Test data:

```
rbTree.insert(30, "Thirty");
rbTree.insert(20, "Twenty");
rbTree.insert(50, "Fifty");
rbTree.insert(10, "Ten");
rbTree.insert(5, "Five");
rbTree.insert(3, "Three");
rbTree.delete(50);
rbTree.print();
```



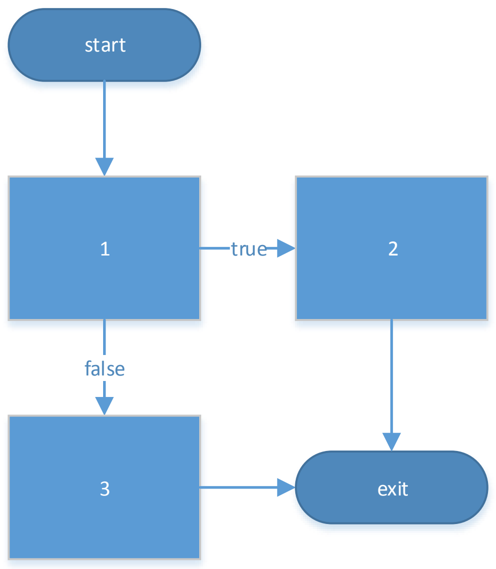
```
1	/**
2	 * O(log(n))
3	 * public int size()
4	 *
5	 * Returns the number of nodes in the tree.
6	 *
7	 * precondition: none
8	 * postcondition: none
9	 */
10	public int size()
11	{
12		if(this.empty()){
13			return 0;
14		}
15
16		return sizeCalc(this.Root.Left);
17	}
18
```

Block	Lines	Entry	Exit
1	12	12	12
2	13	13	13
3	16	16	16

Test path: T1 = [1,2]
T2 = [1,3]

Test data:

```
Assert.assertEquals("Check for empty tree size", rbTree.size(), 0);

rbTree.insert(100, "One hundred");
Assert.assertEquals("Check one node", rbTree.size(), 1);

rbTree.insert(50, "Fifty");
rbTree.insert(150, "One hundred and fifty");
rbTree.insert(200, "Two hundred");

Assert.assertEquals("Check after inserts", rbTree.size(), 4);
```


start
1
true
2
false
3
exit

```
1   /**
2    * O(log(n))
3    * @param node the node to start from
4    * @return the amount of all nodes "under" that node
5    */
6   public static int sizeCalc(RBNode node){
7           //count from both sides
8           if(!isNullNode(node.Left) && !isNullNode(node.Right)){
9                   return 2 + sizeCalc(node.Left) + sizeCalc(node.Right);
10          }
11
12          //count with higher side
13          if(!isNullNode(node.Left)){
14                  return 1 + sizeCalc(node.Left);
15          }
16
17          //count with lower side
18          if(!isNullNode(node.Right)){
19                  return 1 + sizeCalc(node.Right);
20          }
21
22          return 1;
23  }
24
```

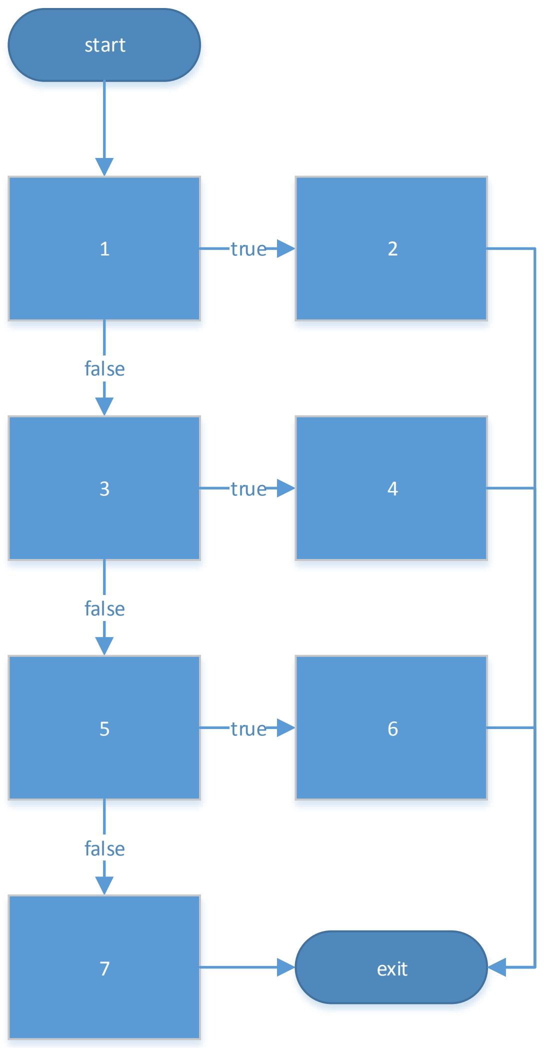
Block	Lines	Entry	Exit
1	8	8	8
2	9	9	9
3	13	13	13
4	14	14	14
5	18	18	18
6	19	19	19
7	22	22	22

Test path: T1 = [1,2]
T2 = [1,3,4]
T3 = [1,3,5,6]
T4 = [1,3,5,7]

Corrections: return 1 + sizeCalc(node.Left) + sizeCalc(node.Right); change line 9 to return 1 as only 1 node is counted

Test data:

```
rbTree.insert(100, "One hundred");
Assert.assertEquals("Check root element", rbTree.sizeCalc(rbTree.getRoot()), 1);

rbTree.insert(50, "Fifty");
Assert.assertEquals("Check root element", rbTree.sizeCalc(rbTree.getRoot()), 2);
```


start
1
true
2
false
3
true
4
false
5
true
6
false
7
exit