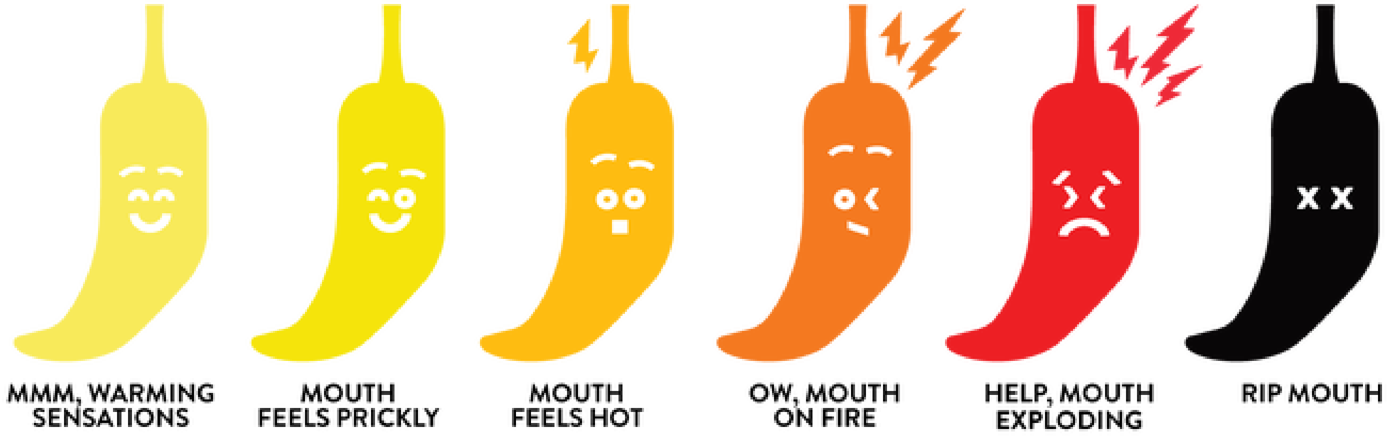
Assignment 3 BST Graded Assignment

Tasks

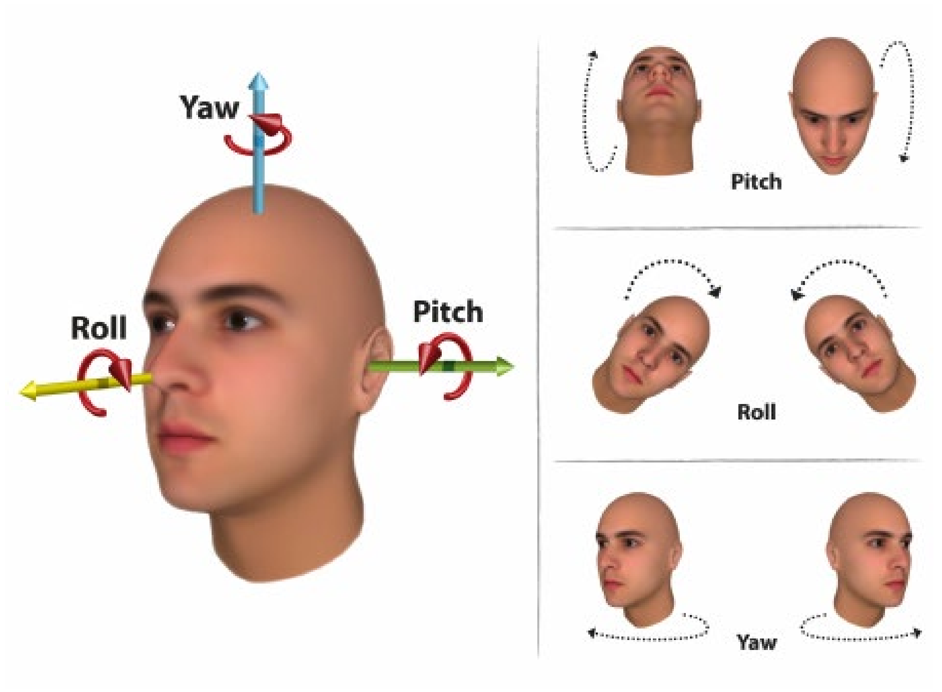
- inorder, postorder
- height, size
- search Min/Max
- exist
- successor
- Balancing (left/right rotation)

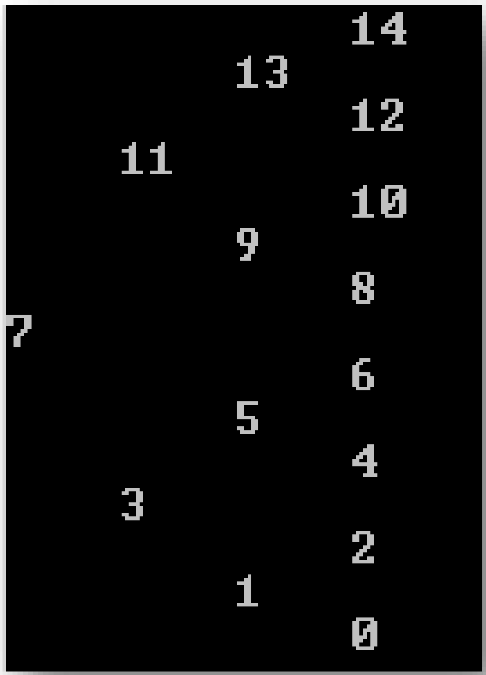
Your Tasks

Ranging from



When You Run the Code

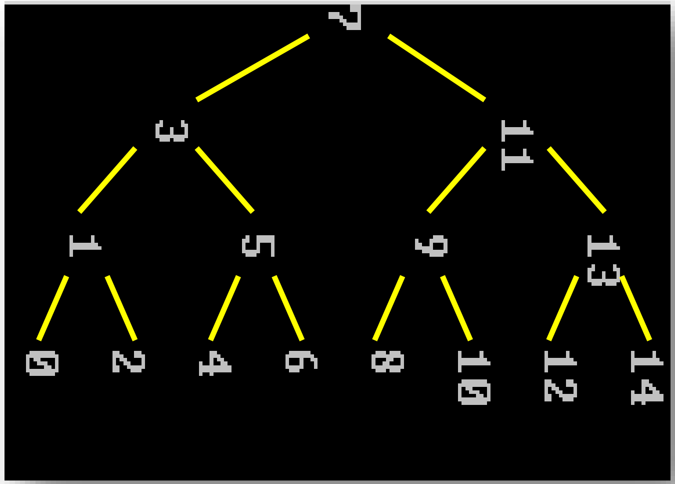




```
            14
        13
            12
    11
            10
        9
            8
7
            6
        5
            4
    3
            2
        1
            0
```

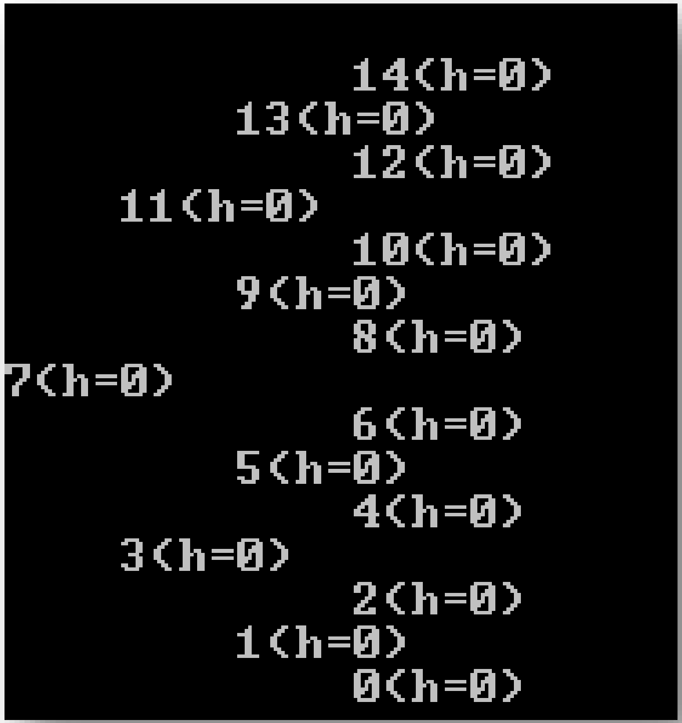
When You Run the Code

- It’s a binary search tree with a 90 angle rotation



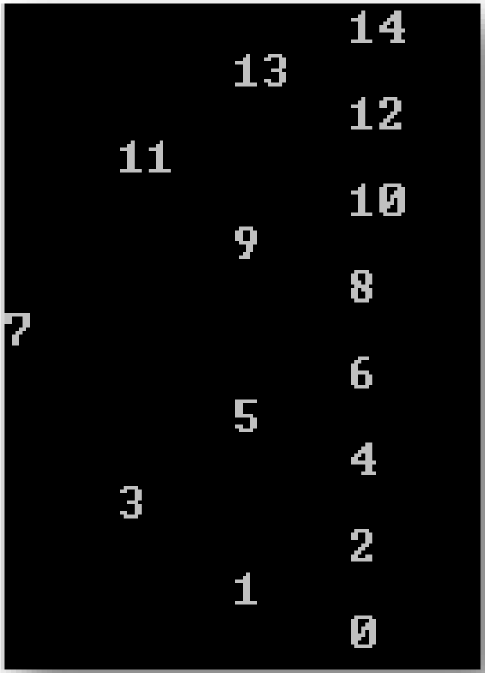
Print with Heights

`printTree(true)`



```
            14(h=0)
        13(h=0)
            12(h=0)
    11(h=0)
            10(h=0)
        9(h=0)
            8(h=0)
7(h=0)
            6(h=0)
        5(h=0)
            4(h=0)
    3(h=0)
            2(h=0)
        1(h=0)
            0(h=0)
```

`printTree(false)`

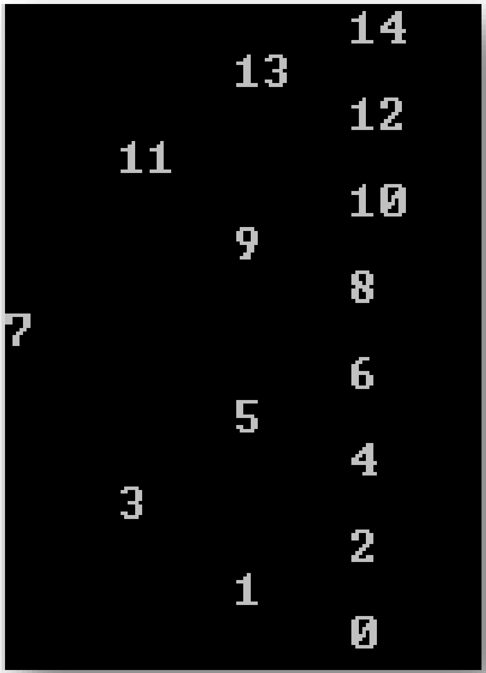


```
            14
        13
            12
    11
            10
        9
            8
7
            6
        5
            4
    3
            2
        1
            0
```

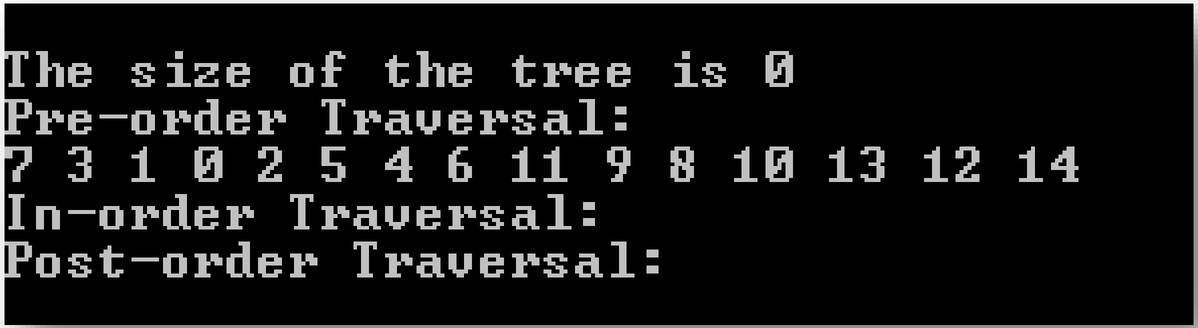
You will fix the heights later

Pre-order Traversal

- The size of the tree is the number of the nodes in the tree
  – It is not done yet
  – Your job
- Pre-order Traversal is done for you



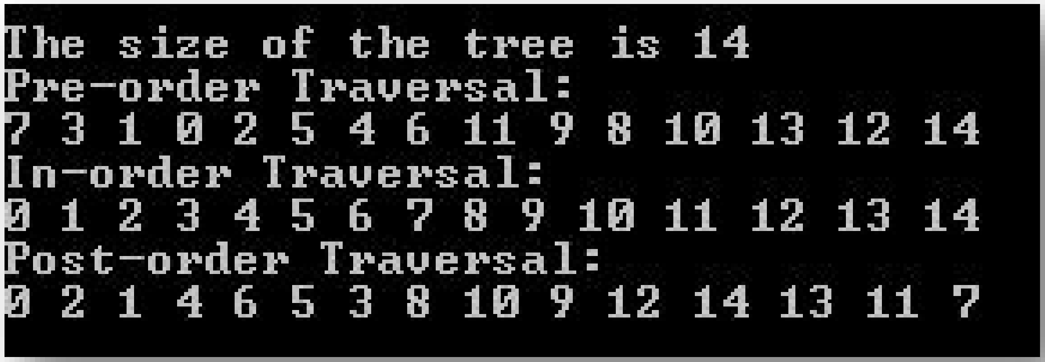
```
                14
           13
                12
      11
                10
           9
                8
7
                6
           5
                4
      3
                2
           1
                0
```



```
The size of the tree is 0
Pre-order Traversal:
7 3 1 0 2 5 4 6 11 9 8 10 13 12 14
In-order Traversal:
Post-order Traversal:
```

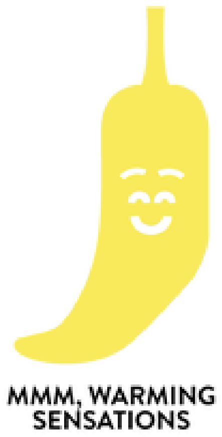
First Task

- Implement the two other traversals
  - In-order traversal
  - Post-order traversal
- If you implement them correctly:



```
The size of the tree is 14
Pre-order Traversal:
7 3 1 0 2 5 4 6 11 9 8 10 13 12 14
In-order Traversal:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
Post-order Traversal:
0 2 1 4 6 5 3 8 10 9 12 14 13 11 7
```

- Note that the in-order traversal is sorted

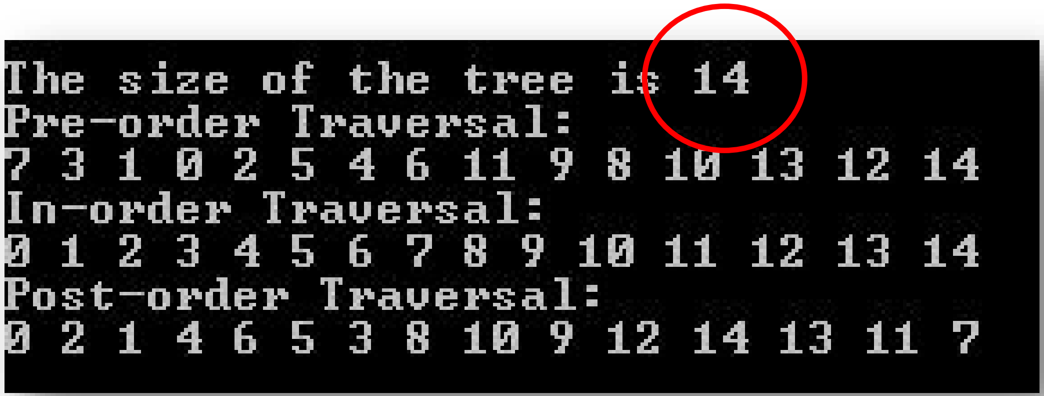


Second Task

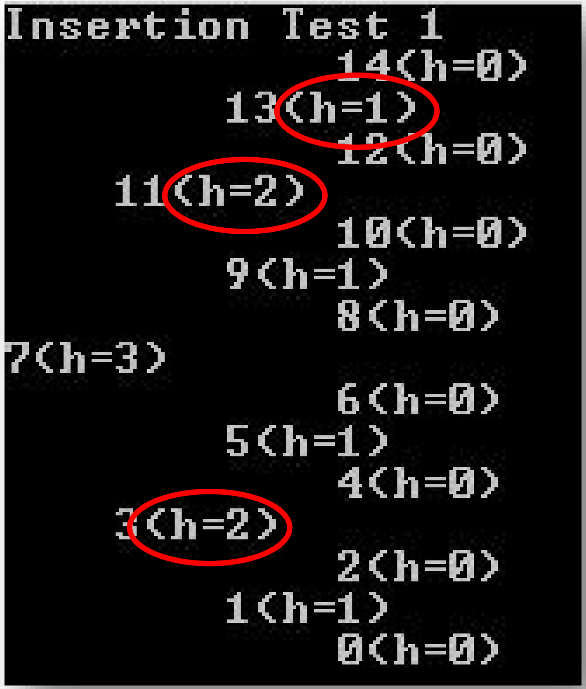
MOUTH
FEELS PRICKLY

- Fix
  - The size of the tree
    - `_size` in BinarySearchTree
  - And the height of each node
    - `_height` in TreeNode

OR 15?



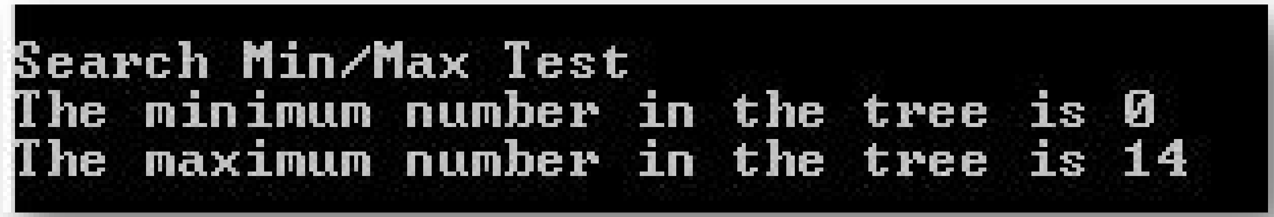
```
The size of the tree is 14
Pre-order Traversal:
7 3 1 0 2 5 4 6 11 9 8 10 13 12 14
In-order Traversal:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
Post-order Traversal:
0 2 1 4 6 5 3 8 10 9 12 14 13 11 7
```



```
Insertion Test 1
            14(h=0)
        13(h=1)
            12(h=0)
    11(h=2)
            10(h=0)
        9(h=1)
            8(h=0)
7(h=3)
            6(h=0)
        5(h=1)
            4(h=0)
    3(h=2)
            2(h=0)
        1(h=1)
            0(h=0)
```

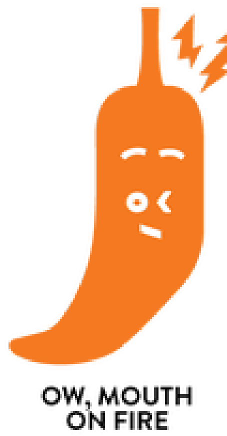
Third Task

- uncomment `testSearchMinMax()` in `main()`
- Implement the functions
  - `searchMin`
  - `searchMax`
- Sample output:

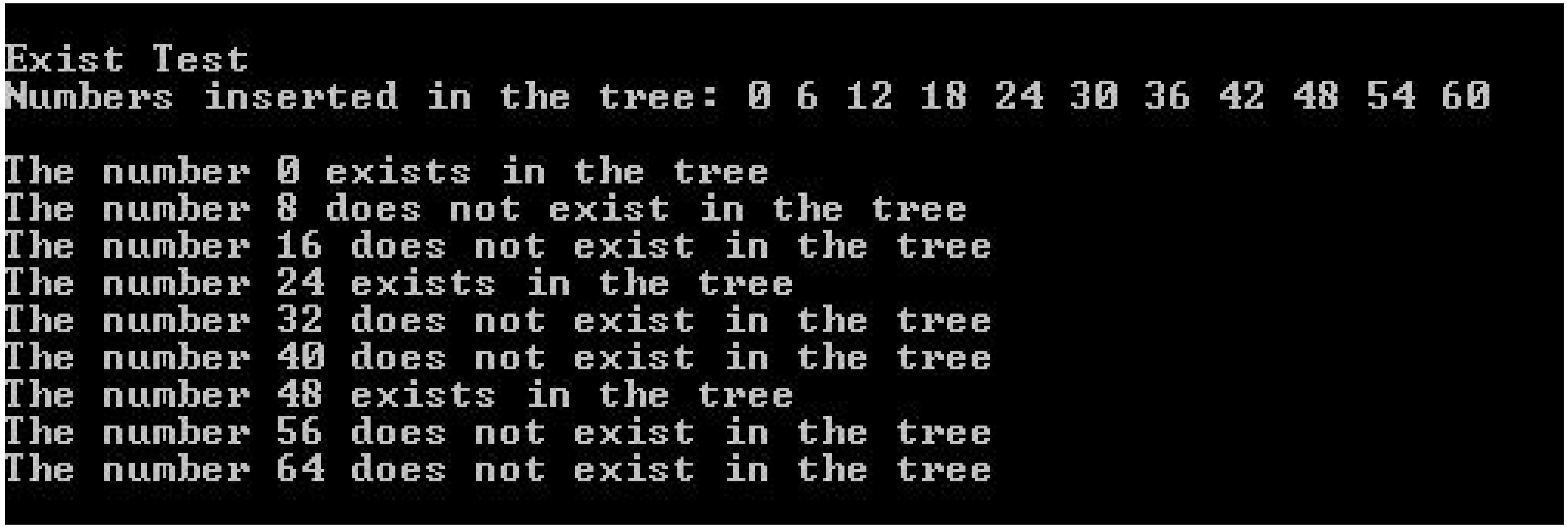


```
Search Min/Max Test
The minimum number in the tree is 0
The maximum number in the tree is 14
```

Forth Task

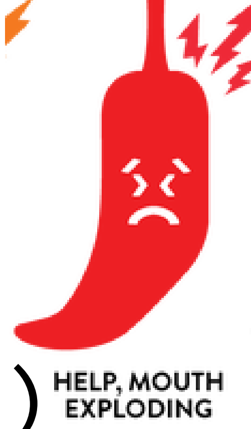


- uncomment `testExist()` in `main()`
- Implement the function **`exist()`** in BST such that the correct output will be

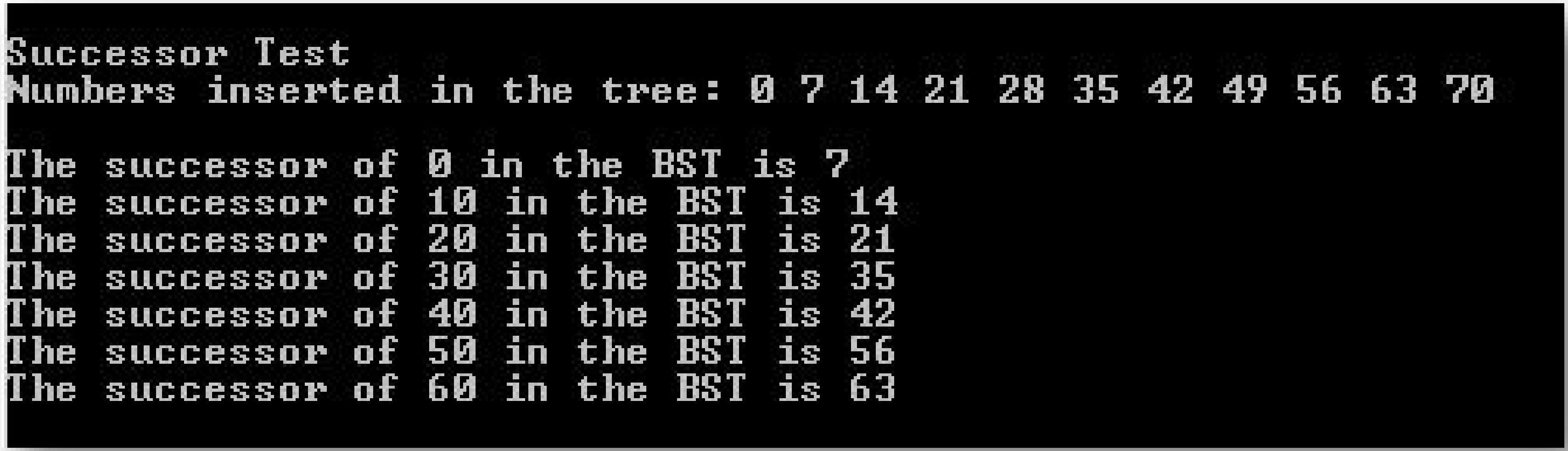


```
Exist Test
Numbers inserted in the tree: 0 6 12 18 24 30 36 42 48 54 60

The number 0 exists in the tree
The number 8 does not exist in the tree
The number 16 does not exist in the tree
The number 24 exists in the tree
The number 32 does not exist in the tree
The number 40 does not exist in the tree
The number 48 exists in the tree
The number 56 does not exist in the tree
The number 64 does not exist in the tree
```

Fifth Task

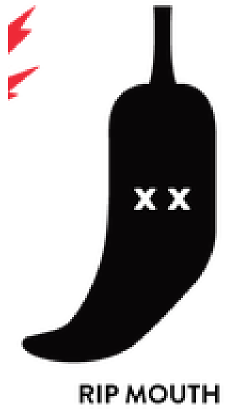


- uncomment `testSuccessor()` in `main()`
- implement the function successor(x) in BST
  – Such that it will return the successor of x



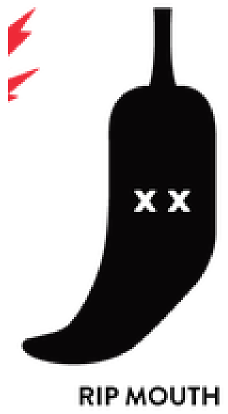
```
Successor Test
Numbers inserted in the tree: 0 7 14 21 28 35 42 49 56 63 70

The successor of 0 in the BST is 7
The successor of 10 in the BST is 14
The successor of 20 in the BST is 21
The successor of 30 in the BST is 35
The successor of 40 in the BST is 42
The successor of 50 in the BST is 56
The successor of 60 in the BST is 63
```

Sixth Task



- Balance the Tree
- Suggest that you make sure every before this is alright
- Make a backup of all the code before MESSING it up

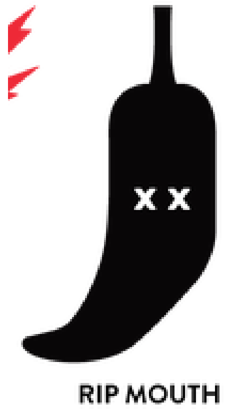
Sixth Task

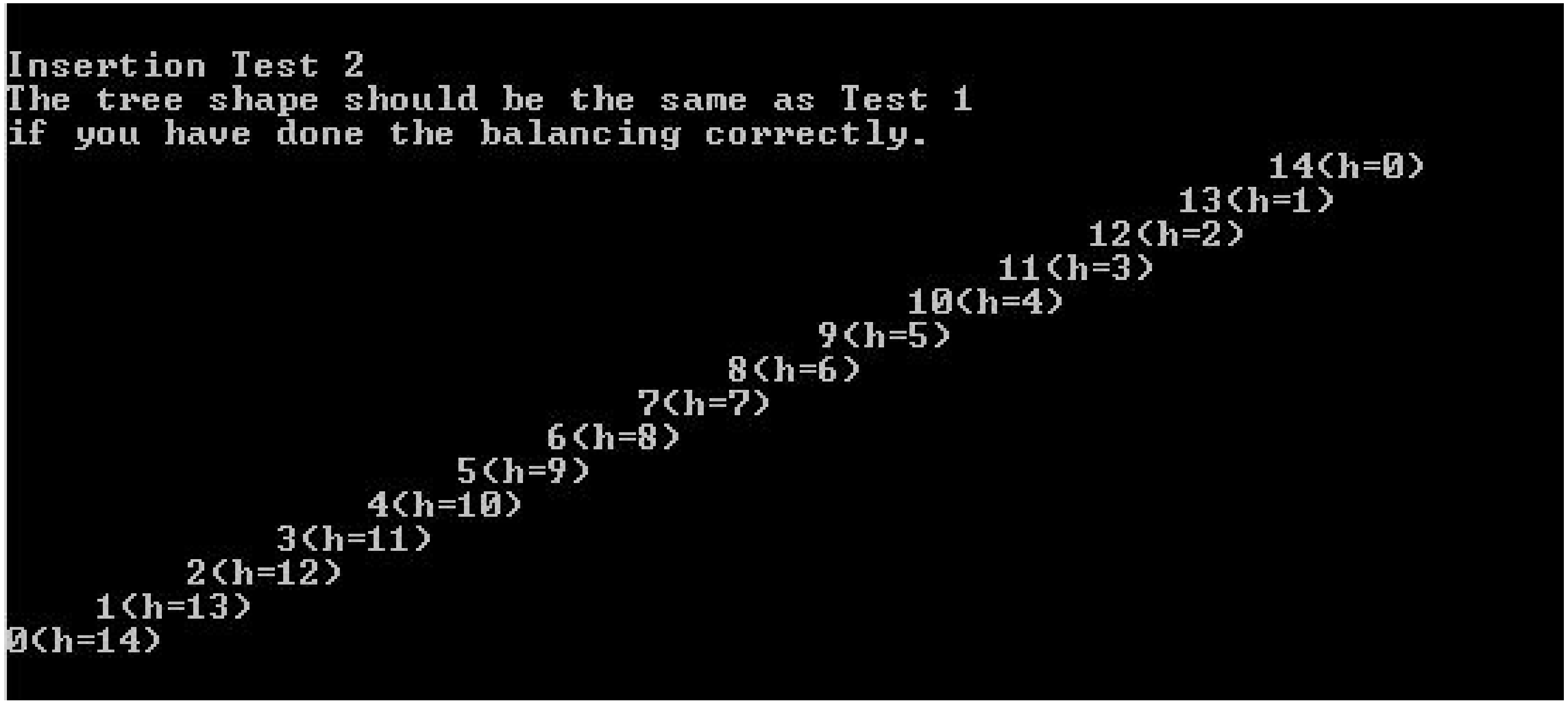


- Balance the Tree
- uncomment `testInsertion2 ()` in `main()`
- `Basically it should run, but because of the insertion order, the tree looks like…`

Sixth Task

- Balance the Tree

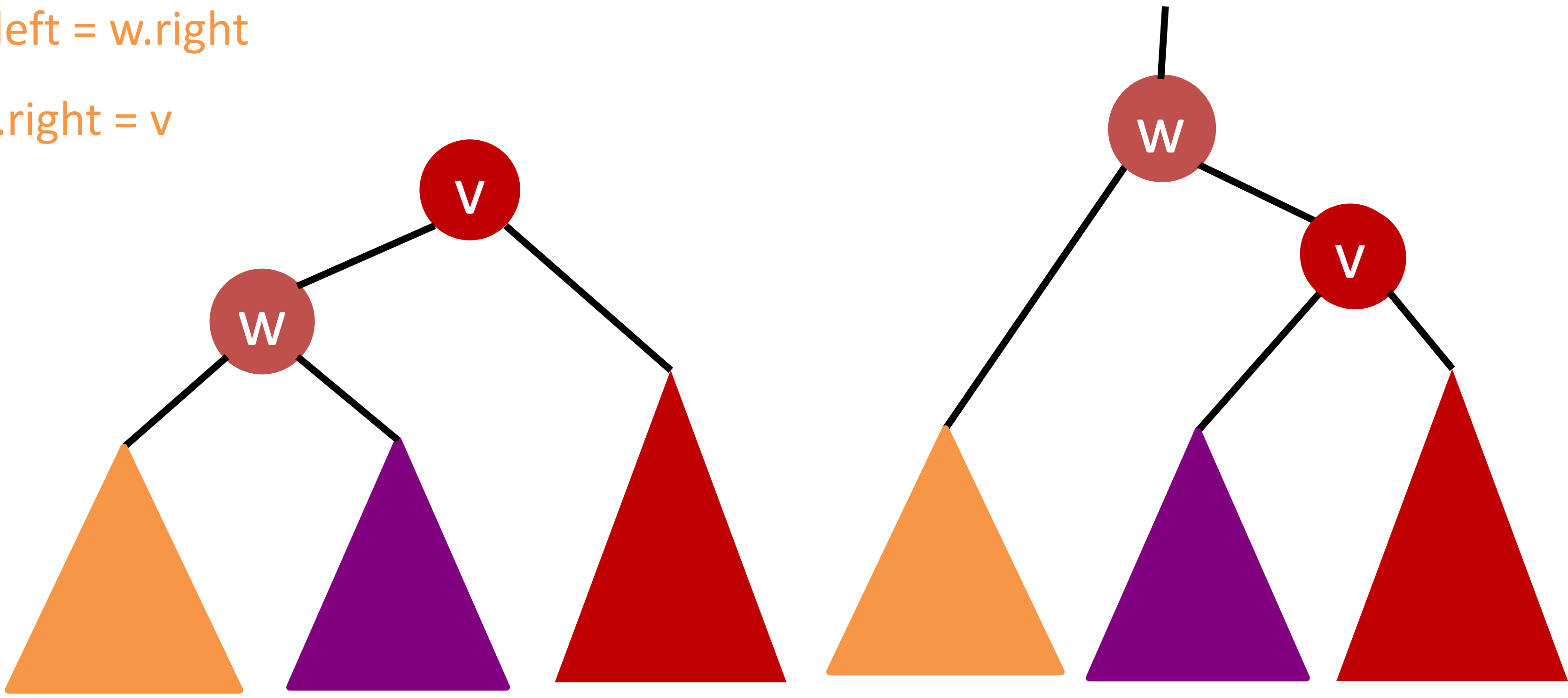




```
Insertion Test 2
The tree shape should be the same as Test 1
if you have done the balancing correctly.
                                                        14(h=0)
                                                    13(h=1)
                                                12(h=2)
                                            11(h=3)
                                        10(h=4)
                                    9(h=5)
                                8(h=6)
                            7(h=7)
                        6(h=8)
                    5(h=9)
                4(h=10)
            3(h=11)
        2(h=12)
    1(h=13)
0(h=14)
```

Rotations

```
right-rotate(v)          // assume v has left != null
    Let the parent of v is the pointer “parent"
    w = v.left
    parent = w
    v.left = w.right
    w.right = v
```



Rotations

Summary:

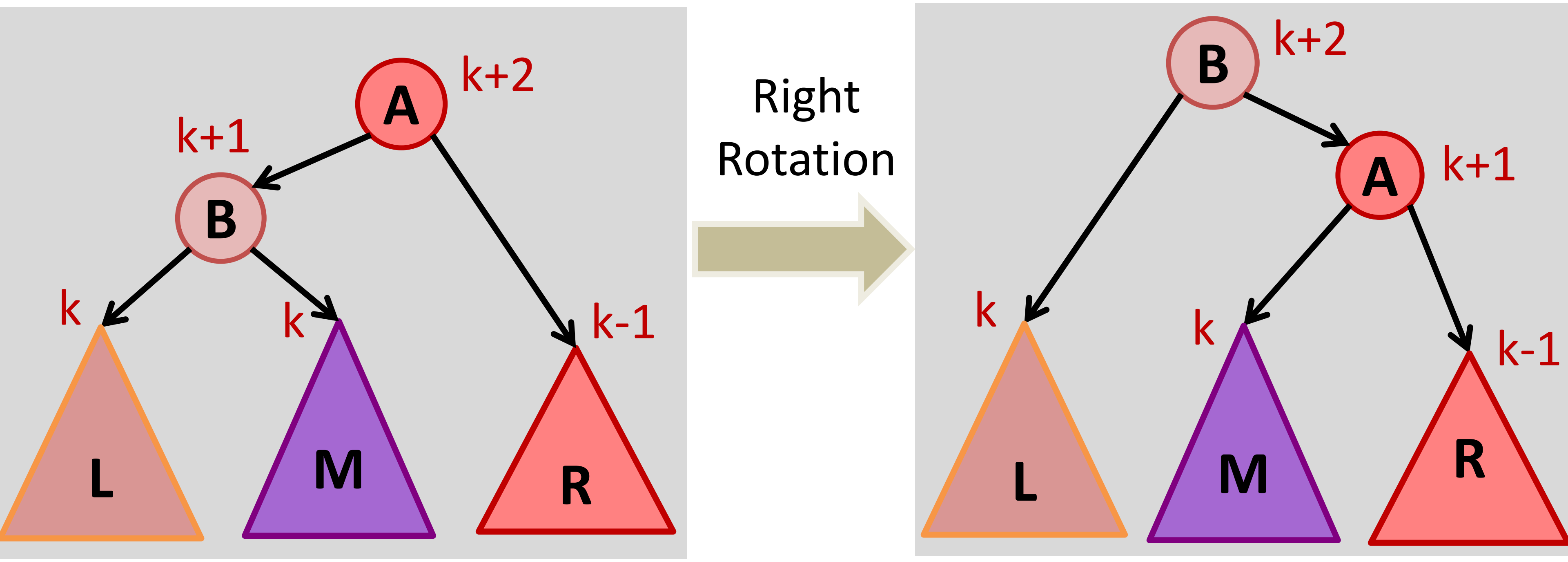
If v is out of balance and *left heavy*:

1. v.left is balanced: right-rotate(v)
2. v.left is left-heavy: right-rotate(v)
3. v.left is right-heavy: left-rotate(v.left)
   right-rotate(v)

If v is out of balance and right heavy:

Symmetric three cases....

Tree Rotations

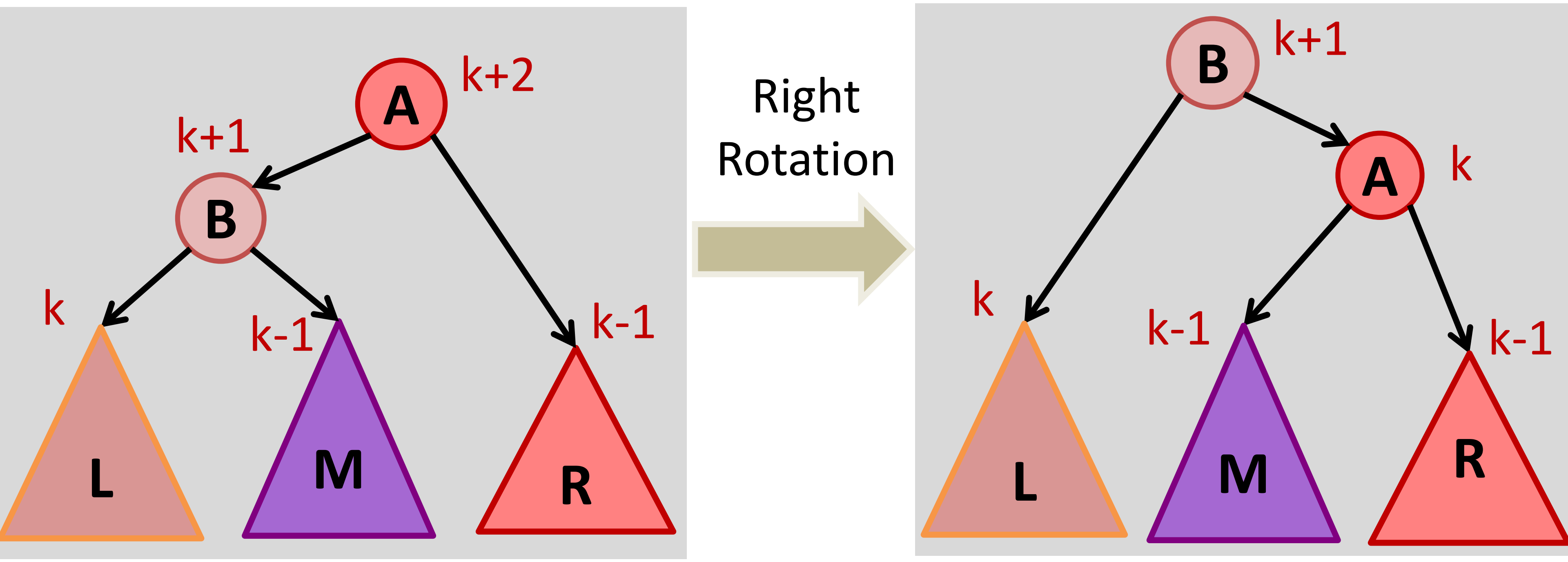


right-rotate:

Case 1: **B** is balanced : h(**L**) = h(**M**)

h(**R**) = h(**M**) – 1

Tree Rotations

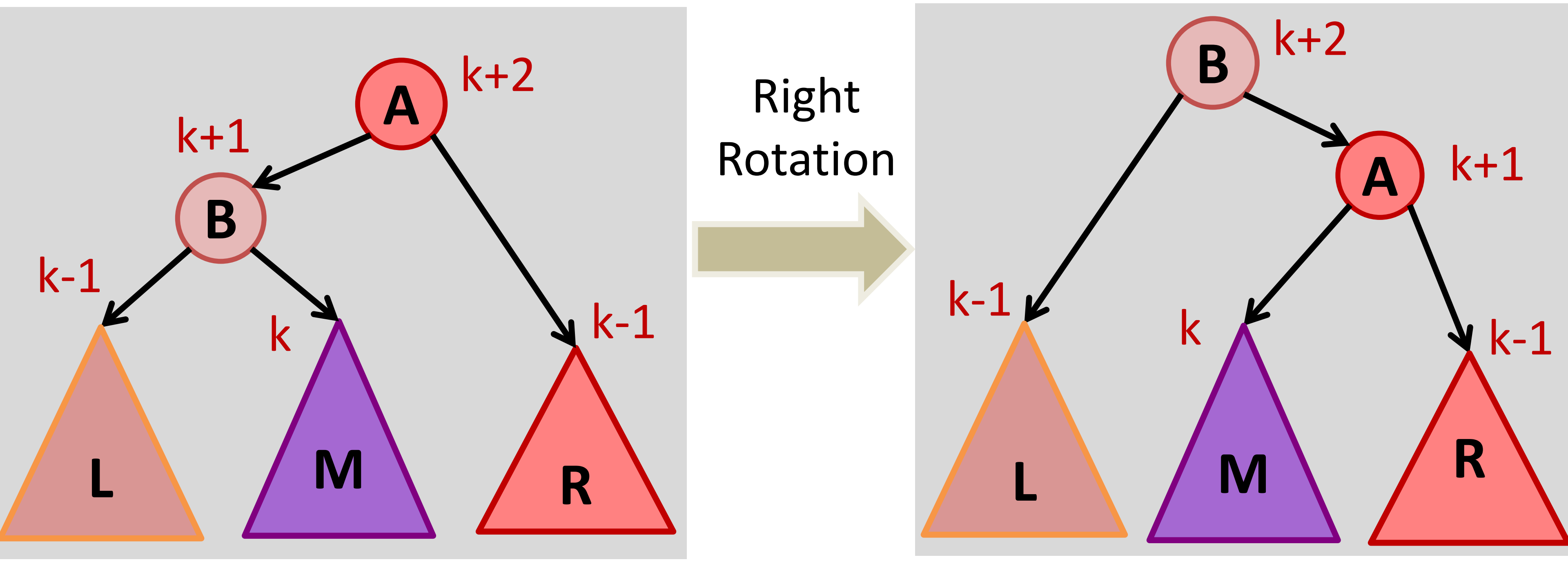


right-rotate:

Case 2: **B** is left-heavy: h(**L**) = h(**M**) +1

h(**R**) = h(**M**)

Tree Rotations

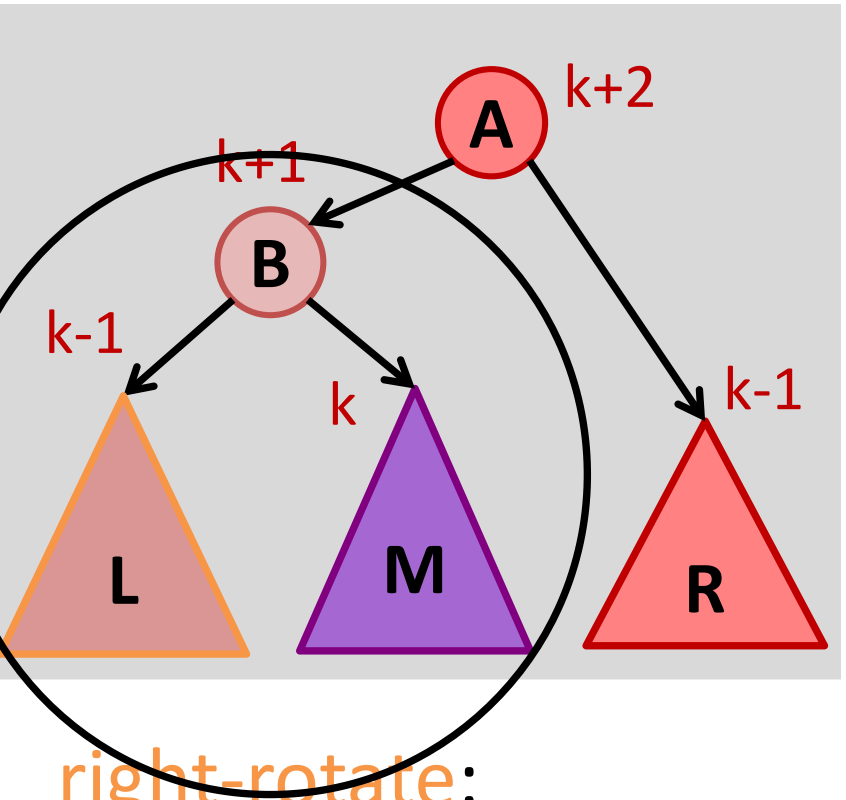


right-rotate:

Case 3: **B** is right-heavy: h(**L**) = h(**M**) – 1

h(**R**) = h(**L**)

Tree Rotations



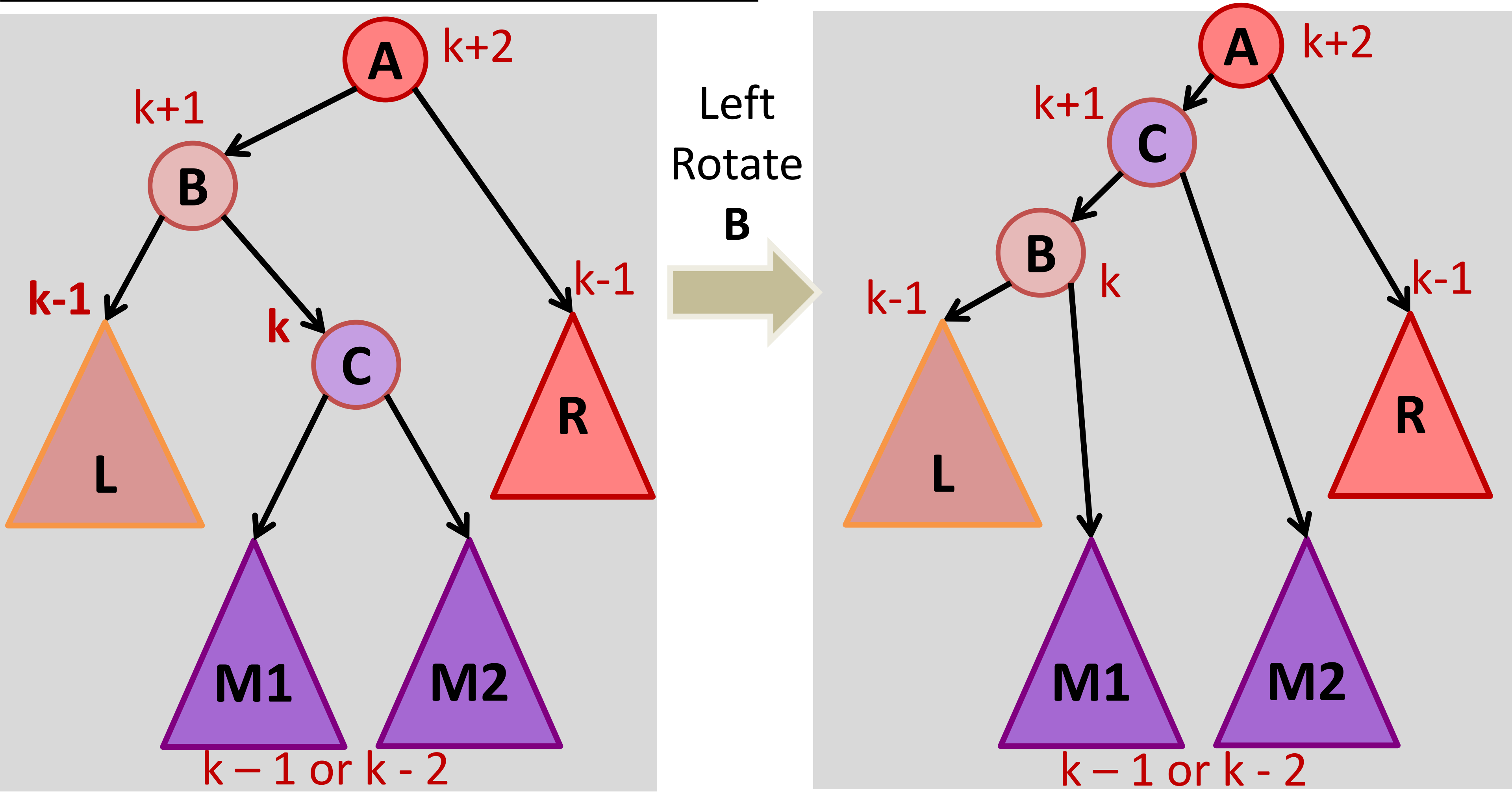
Let’s do something first before we right-rotate(A)

right-rotate:

Case 3: **B** is right-heavy: h(L) = h(**M**) – 1

h(**R**) = h(L)

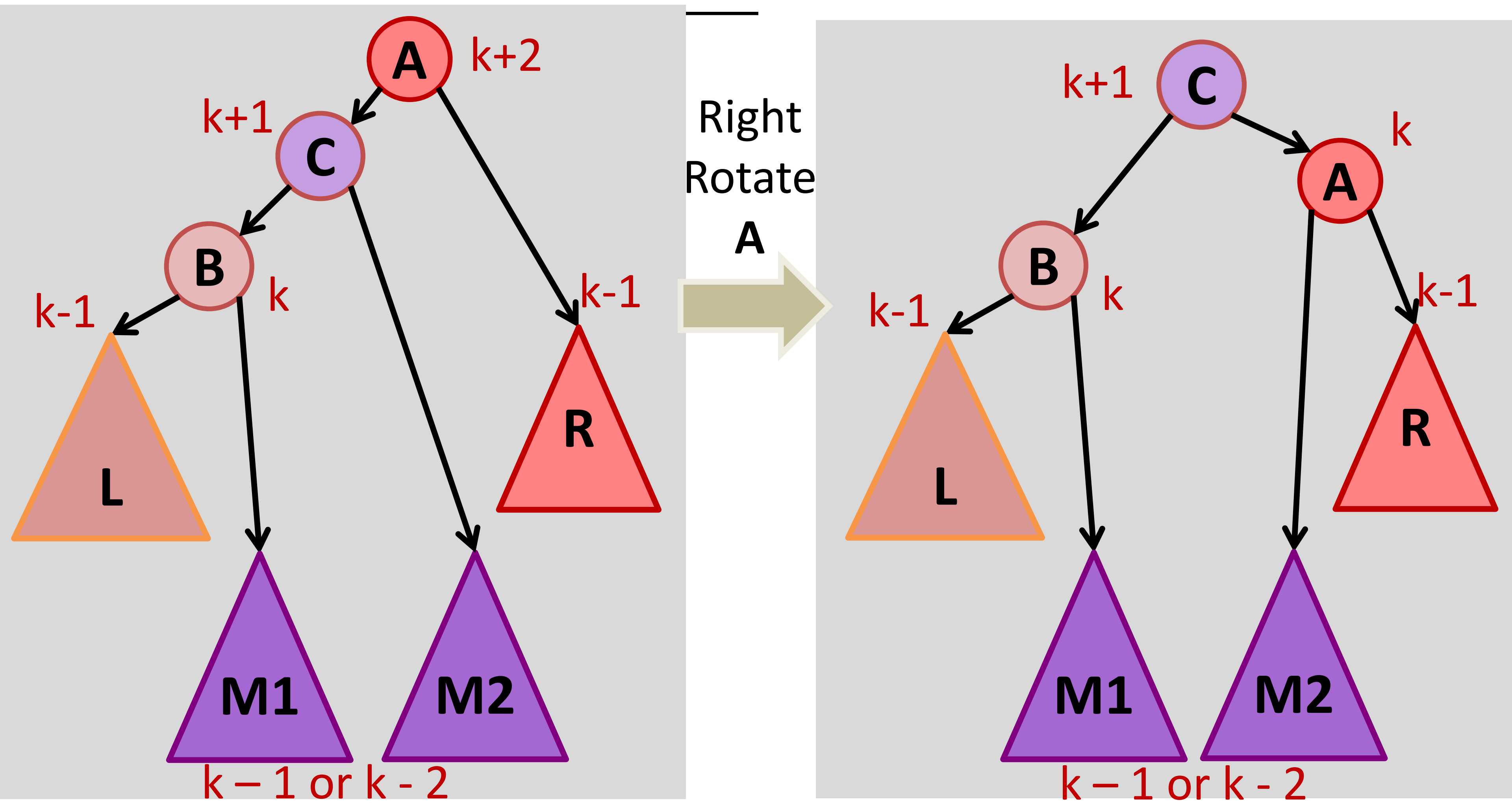
Tree Rotations



Left-rotate B

After left-rotate B: **A** and **C** still out of balance.

Tree Rotations



After right-rotate A: all in balance.

Rotations

Summary:

If v is out of balance and left heavy:

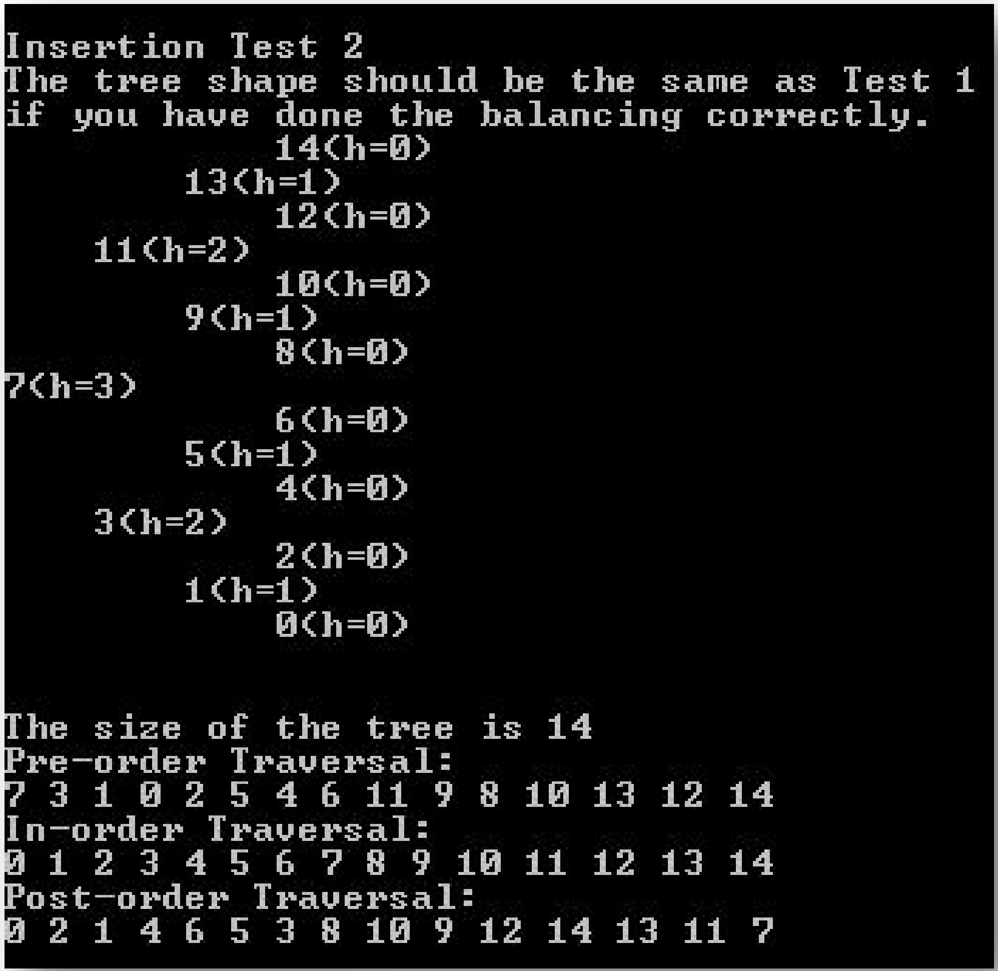
1. v.left is balanced: right-rotate(v)
2. v.left is left-heavy: right-rotate(v)
3. v.left is right-heavy: left-rotate(v.left)
   right-rotate(v)

If v is out of balance and right heavy:

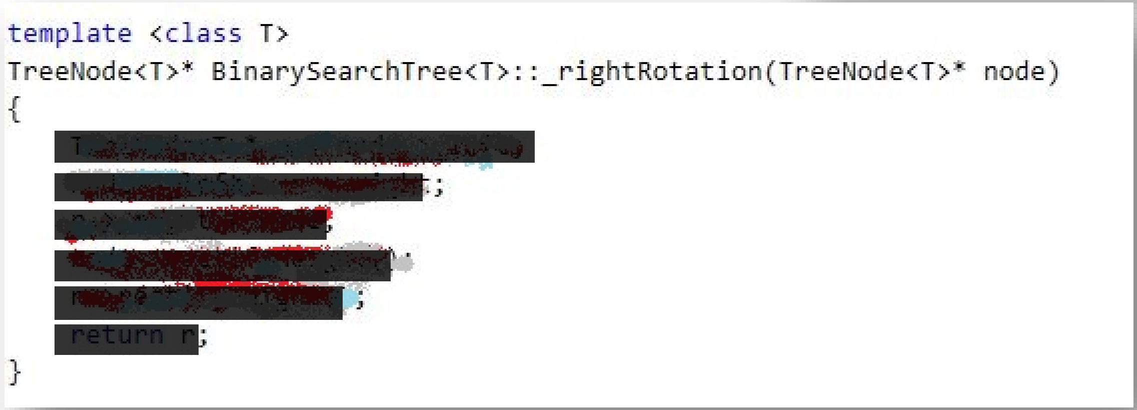
Symmetric three cases….

Sixth Task

- Balance the Tree
  - Sample output:
- In fact, it's less scary as it looks
  - My solution:



```
Insertion Test 2
The tree shape should be the same as Test 1
if you have done the balancing correctly.
            14(h=0)
        13(h=1)
            12(h=0)
    11(h=2)
            10(h=0)
        9(h=1)
            8(h=0)
7(h=3)
            6(h=0)
        5(h=1)
            4(h=0)
    3(h=2)
            2(h=0)
        1(h=1)
            0(h=0)


The size of the tree is 14
Pre-order Traversal:
7 3 1 0 2 5 4 6 11 9 8 10 13 12 14
In-order Traversal:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
Post-order Traversal:
0 2 1 4 6 5 3 8 10 9 12 14 13 11 7
```



```
template <class T>
TreeNode<T>* BinarySearchTree<T>::_rightRotation(TreeNode<T>* node)
{
    [redacted]
}
```