Is It A Red-Black Tree

Wu Xinbei

Date: 2020-10-30

Chapter 1: Introduction

Problem Description

1. Input Specification

The First Line: Number of Cases (≤ 30)

The Second Lind: Number of Nodes in Case 1 (≤ 30)

The Third Line: Preorder Traversal Sequence of the Tree in Case 1

……

While all the keys in a tree are positive integers, we use negative signs to represent red nodes.

Note: According to given Input Specification, there's no need for us to detect whether size of input overflows.

2. Goal

Tell **if it is a legal red-black tree.** and **output the results** for each given binary search tree.

3. Output Specification

Each line print "Yes" for valid Red-Black tree and "No" for invalid Red-Black Tree

Note: Output results together line by line rather than one by one while reading input

4. Properties of Red-Black Tree

- Balanced **Binary Search Tree**

  Note: Though we say Red-Black Tree is balanced binary tree, ***its balance is not equal to AVL*** *(balanced binary tree) . Its balance is a result of other properties. So there's no need to judge its balance specially.*
- Every node is either red or black.
- **The root is black.**
- Every leaf (NULL) is black.
- If a node is red, then both its children are black.

  Note: It means there' re ***no consecutive red nodes****.*
- For each node, **all simple paths from the node to descendant leaves contain the same number of black nodes**.

For example, the tree in Figure 1 is a valid red-black tree, while the ones in Figure 2 and 3 are not.

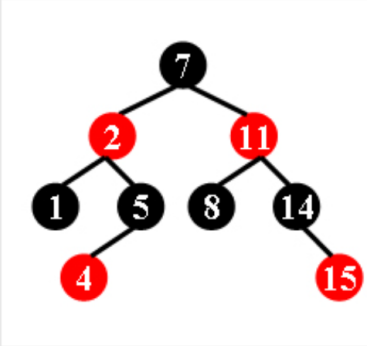


Figure 1

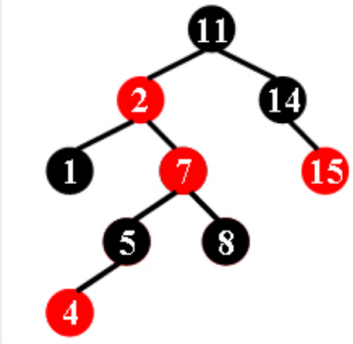


Figure 2

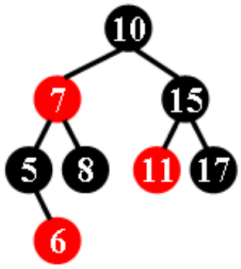


Figure 3

*Note: **NULL tree** satisfies all properties.*

Problem Analysis

We have to judge whether the tree is Red-Black Tree by given preorder sequence. To make problem easier, we can divide the problem into several part.

1. **Create a tree.**

   To make the whole program readable, create a tree first. Although it's workable to judge it without build a tree. While reading input, we have to judge whether its value is valid.

2. **Judge Red-Black Properties One by One**

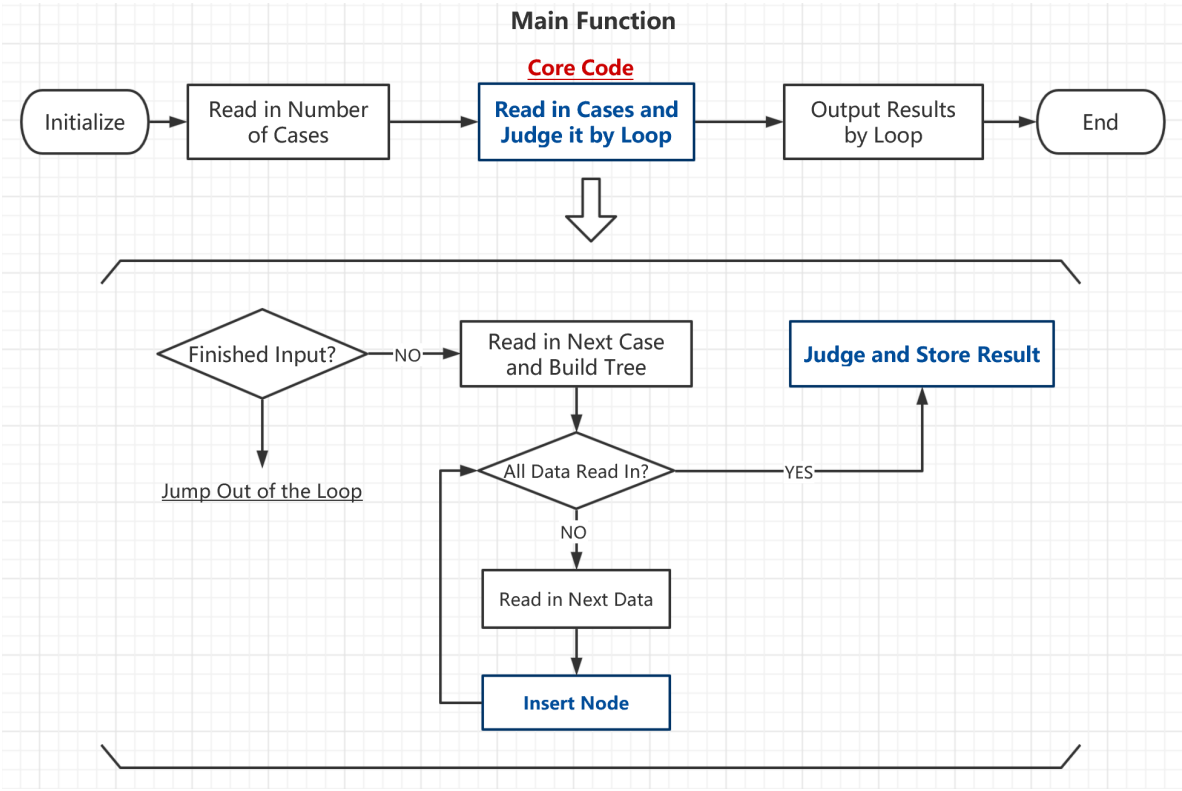
   - Null or not. If null, then it's valid.
   - Root should be Black.
   - Traverse to compare numbers of black nodes on all paths. If not equal, then it's invalid.
   - Traverse to detect whether there' re consecutive red nodes. If there is, then it's invalid.

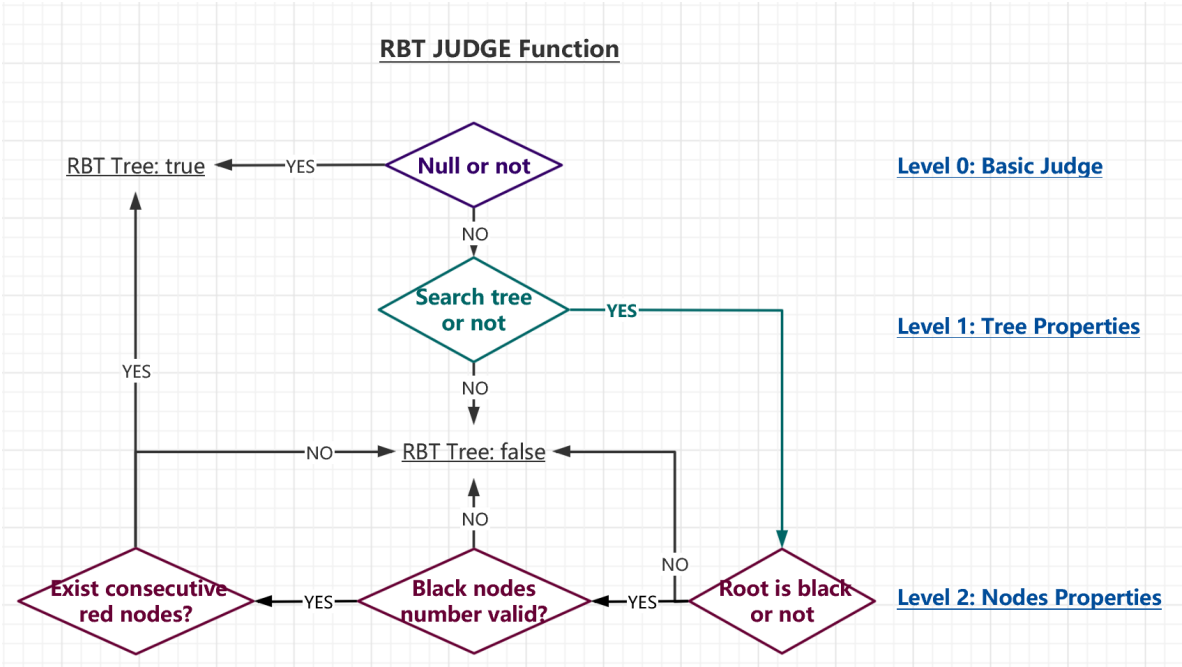
Chapter 2: Algorithm Specification

1. Main Algorithm: Recurse

To solve this problem related to **tree**, **link list** is the main data structure. While traversing and judge the tree, ***Recursive Algorithm*** makes the procedure much easier.

2. Overall of flow chart





3. Pseudo-code for Core Part Function

1. **Data Structure used to organize Binary tree**

```
struct treeNode {
    ElementType val; // used to store node's value
    bool isRed;      // mark the node is red or not
    bool isValid;    // mark the value of data is valid or not
    struct treeNode* leftT;     // link to left child
    struct treeNode* rightT;    // link to right child
};
```

Note: Node's color is determined by data input. But to make the whole program easier to read, I choose to sacrifice some space complexity. Condition for validity of node's value is the same.

2. **Pseudo-code for Tree-Building**

```
loop to read cases:
    declare a tree link list;
    loop to read data for certain case:
        insert node in tree {
            if root is null,
                allocate space, initialize it and continue;
            Compare node's value,
                recurse to find right position(null) to insert;
        }
        Detect the input data's validity;
    Judge whether it's Red-Black Tree;
    store result;
next part;
```

3. **Pseudo-code for Tree-Judging**

```
NULL or not?
    Yes, end Judge Procedure and return true Red-Black tree;
    No, continue;


Search tree or not?
    Traverse every node, is there any invalid data?
        No, end Judge Procedure and return false Red-Black tree;
        Yes, continue;


Root Color Black or not?
    No, end Judge Procedure and return false Red-Black tree;
    Yes, continue;


Numbers of Black Nodes valid or not?
    Recurse Exit: null node; return true Red-Black tree;
    Recurse to get numbers of black nodes of left and right children
        Recurse Exit: null node; return 0;
        if current node is black
            return max of black nodes number of left and right children
    Black nodes number of left child is equal to right child's?
        No, end Judge Procedure and return false Red-Black tree;
        Yes, continue;
    Recurse to traverse left tree and right tree.


Consecutive Red Nodes or Not?
    Recurse Exit: null node; return true Red-Black tree;
    if current node is red
        if it has child(left or right does not matter)
            if its child's node is red
                end Judge Procedure and return false Red-Black tree;
    Recurse to detect child tree;
```

Chapter 3: Testing Results

1. ## Min Input Case

    1. **No Case**

        Sample

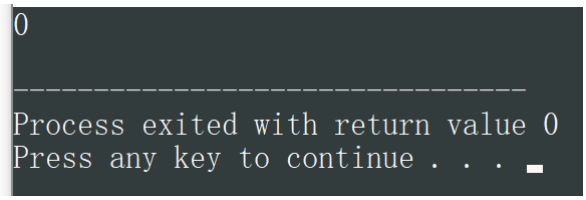
        ```
        0 // Number of Case
        ```

        Expected Output

        ```
        No Output
        ```

        Actual Output

        ```
        No Output
        ```

        Result: Pass

    2. **One Case Contains Null**

        Sample

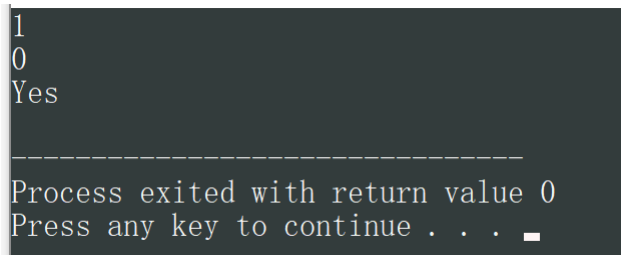
        ```
        1 // Number of Cases
        0 // Case1: Number of Node
        ```

        Expected Output

        ```
        Yes
        ```

        Actual Output

        ```
        Yes
        ```

        Result: Pass

    3. **One Case Contain Only One Node**

        Sample

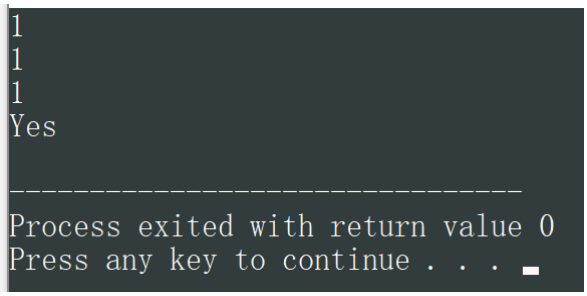
```
1 // Number of Case
1 // Case1: Nunber of Node
1 // Data
```

Expected Output

```
Yes
```

Actual Output

```
Yes
```



Result: Pass

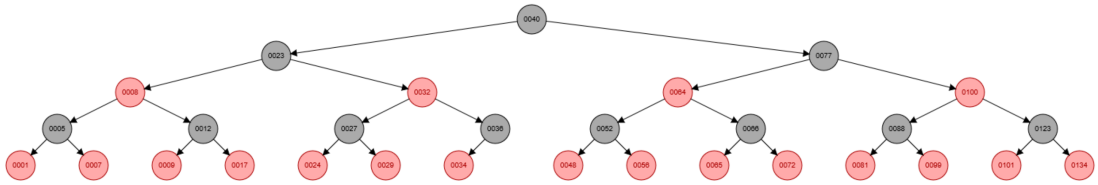
2. ## Max Input Case

Sample

```
30 // number of cases
30 // Case1: number of nodes
40 23 -8 5 -1 -7 12 -9 -17 -32 27 -24 -29 36 -34 77 -64 52 -48 -56 66 -65
-72 -100 88 -81 -99 123 -101 -134 // Data
30 // Case2: number of nodes
40 23 -8 5 -1 -7 12 -9 -17 -32 27 -24 -29 36 34 77 -64 52 -48 -56 66 -65 -72
-100 88 -81 -99 123 -101 -134 // Data
30 // Case3: number of nodes
99 -54 36 -16 1 21 -34 38 72 68 -55 -69 90 -88 -93 -289 188 162 -153 -166
222 -200 -256 365 324 -300 -400 384 484 -729 // Data
30 // Case4: number of nodes
99 -54 36 -16 1 21 -34 38 72 68 -55 -69 90 -88 -93 -289 188 162 -153 -166
222 -200 -256 365 324 -300 -400 384 484 729 // Data
30 // Case5: number of nodes
444 -246 222 90 -35 -123 234 333 270 -300 -389 381 435 -666 555 467 -489
-626 563 -590 645 888 871 -765 -875 -999 962 1001 -1000 -1234 // Data
30 // Case6: number of nodes
444 -246 222 90 -35 -123 234 333 270 -300 -389 381 435 -666 555 467 -489
-626 563 -590 645 888 871 -765 -875 -999 -962 1001 -1000 -1234 // Data
30 // Case7: number of nodes
45 32 21 12 23 -31 38 34 43 -39 80 -67 56 46 -54 65 73 70 76 -78 -89 85 83
87 93 91 -100 98 119 -120 // Data
30 // Case8: number of nodes
45 32 21 12 23 -31 38 34 43 -39 80 -67 56 45 -54 65 73 70 76 -78 -89 85 83
87 93 91 -100 98 119 -120 // Data
30 // Case9: number of nodes
8 4 2 1 3 6 5 7 -16 12 10 9 11 14 13 15 20 18 17 19 -24 22 21 23 26 25 -28
27 29 -30
30 // Case10: number of nodes
```

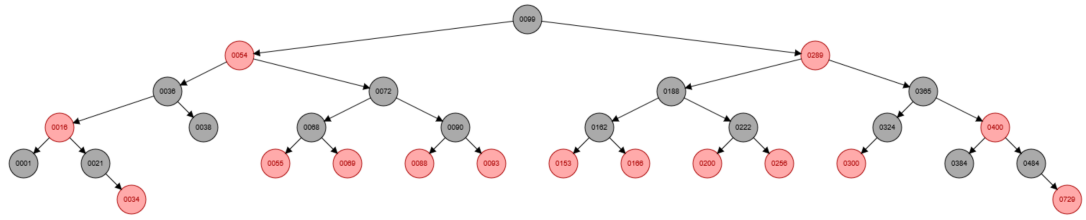
```
8 4 2 0 3 6 5 7 -16 12 10 9 11 14 13 15 20 18 17 19 -24 22 21 23 26 25 -28
27 29 -30
2 // Case11: number of nodes
1 -3 // Data
2 // Case12: number of nodes
1 3 // Data
12 // Case13: number of nodes
6 2 1 3 18 -12 9 16 -22 20 26 -30 // Data
13 // Case14: number of nodes
6 2 1 3 18 -12 9 16 -22 20 26 -30 28 // Data
30 // Case15: number of nodes
19 7 3 1 5 15 11 -9 -13 17 39 -27 23 21 25 35 31 -29 -33 37 -47 43 41 45 51
49 -55 53 57 -59 // Data
30 // Case16: number of nodes
-19 7 3 1 5 15 11 -9 -13 17 39 -27 23 21 25 35 31 -29 -33 37 -47 43 41 45 51
49 -55 53 57 -59 // Data
30 // Case17: number of nodes
88 44 22 1 -11 33 66 55 77 777 -333 111 99 -101 222 555 444 666 -2222 999
888 1111 -1001 4444 3333 -6666 5555 8888 -7777 -9999 // Data
30 // Case18: number of nodes
88 44 22 1 -11 33 66 55 77 777 333 111 99 -101 222 555 444 666 -2222 999 888
1111 -1001 4444 3333 -6666 5555 8888 -7777 -9999 // Data
30 // Case19: number of nodes
250 -67 17 -5 2 -1 -3 9 33 97 -73 69 -68 83 120 -110 -125 -1000 750 500 -375
-600 875 4000 -2000 1500 -1750 3000 8000 -6000 // Data
30 // Case20: number of nodes
250 -67 17 -5 2 -1 -3 -9 33 97 -73 69 -68 83 120 -110 -125 -1000 750 500
-375 -600 875 4000 -2000 1500 -1750 3000 8000 -6000 // Data
30 // Case21: number of nodes
478 -103 36 31 100 -44 268 234 -118 -249 -358 343 -315 465 -815 585 480 -514
-636 617 -613 -622 786 885 837 -825 -870 923 -911 -962 // Data
31 // Case22: number of nodes
478 -103 36 31 100 -44 268 234 -118 -249 -358 343 -315 465 -815 585 480 -514
-636 617 -613 -622 786 885 837 -825 -870 923 -911 -962 99 // Data
27 // Case23: number of nodes
8 4 2 1 3 6 5 7 12 10 9 11 -18 14 13 16 -15 -17 22 -20 19 21 -25 23 -24 26
-27 // Data
28 // Case24: number of nodes
8 4 2 1 3 6 5 7 12 10 9 11 -18 14 13 16 -15 -17 22 -20 19 21 -25 23 -24 26
-27 27
28 // Case25: number of nodes
8 4 2 1 3 6 5 7 16 -12 10 9 11 14 13 15 -20 18 17 19 24 -22 21 23 -26 25 27
-28// Data
28 // Case26: number of nodes
8 4 2 1 3 6 5 7 16 -12 10 9 11 14 13 15 -20 18 17 19 24 -22 21 23 -26 25 -27
-28
29 // Case27: number of nodes
8 4 2 1 3 6 5 7 16 -12 10 9 11 14 13 15 -20 18 17 19 24 -22 21 23 -26 25 28
-27 -29
29 // Case28: number of nodes
8 4 2 1 3 6 5 7 16 -12 10 9 11 14 13 15 0 18 17 19 24 -22 21 23 -26 25 28
-27 0
18 // Case29: number of nodes
21 3 1 -2 -8 5 13 -144 55 34 89 377 233 -987 610 2584 -1597 -4181 // Data
18 // Case30: number of nodes
21 3 1 -2 -8 5 13 -144 55 34 89 377 233 -987 610 2584 -1597 -4181 1 // Data
```

Case 1 Figure



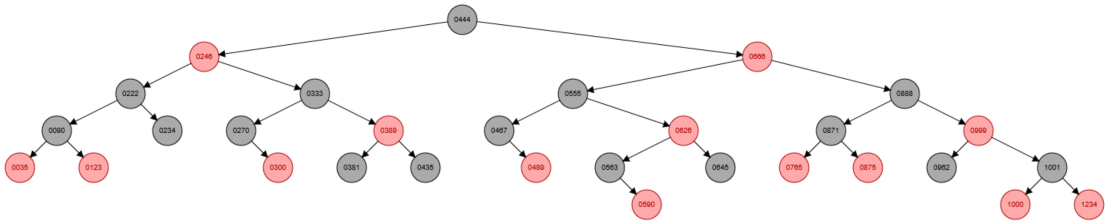
Case 2: Change "34” node’s color to black, keep other nodes the same as case 1.

Case 3 Figure



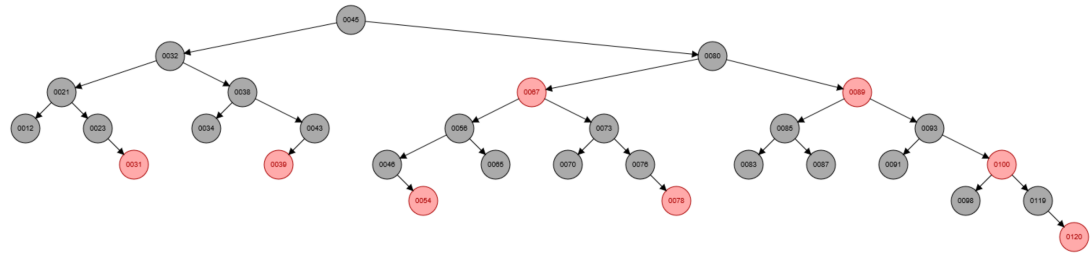
Case 4: Change "729” node’s color to black, keep other nodes the same as case 3.

Case 5 Figure



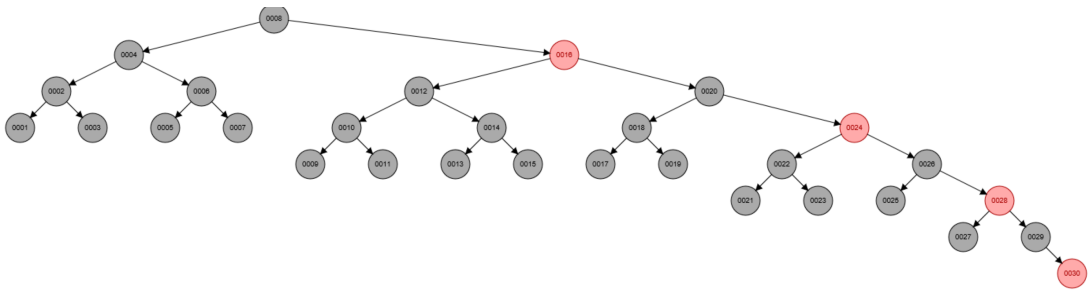
Case 6: Change "962” node’s color to red, keep other nodes the same as case 5.

Case 7 Figure



Case 8: Change "46” node to "45” node, keep other nodes the same as case 7.

Case 9 Figure

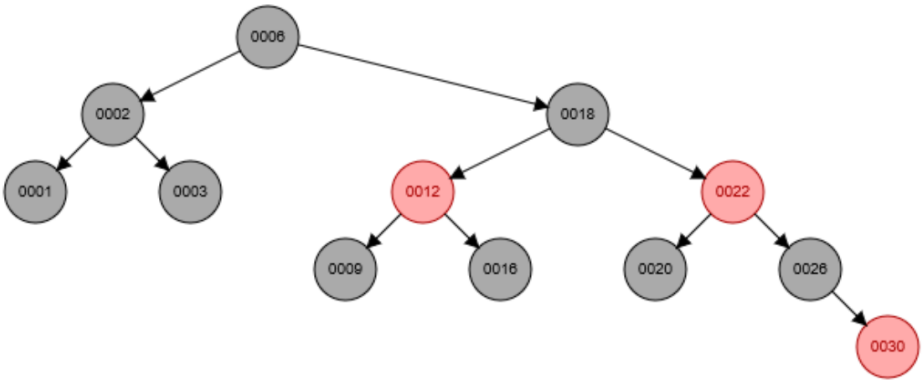


Case 10: Change "1” node’s value to "0”, keep other nodes the same as case 9.

Case 11 Figure

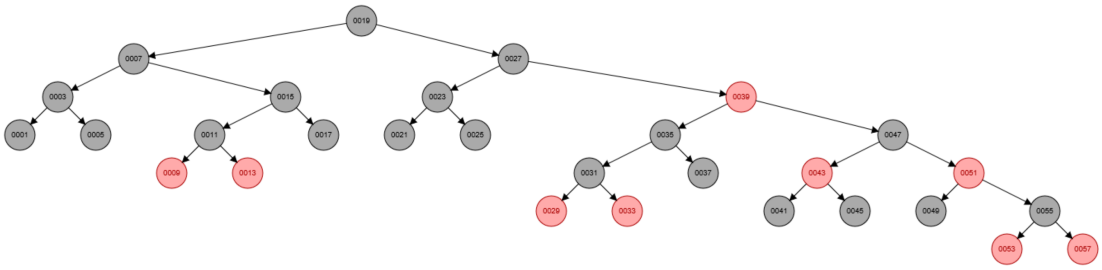
Case 12: Change '3' node' color to black, keep other nodes the same as case 11.

Case 13 Figure



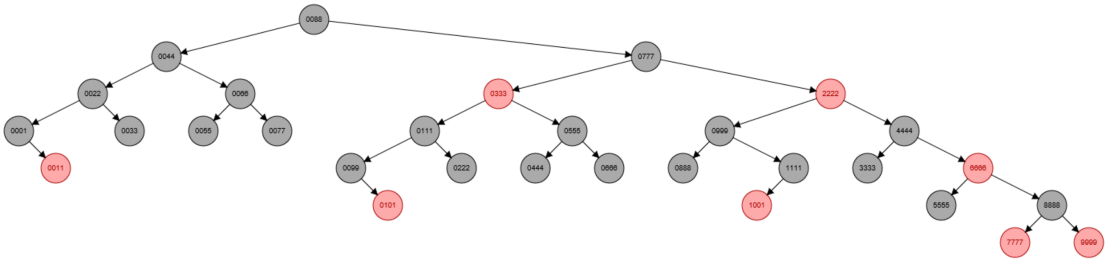
Case 14: Insert a black node "28" under node "30", keep other nodes the same as case 13.

Case 15 Figure



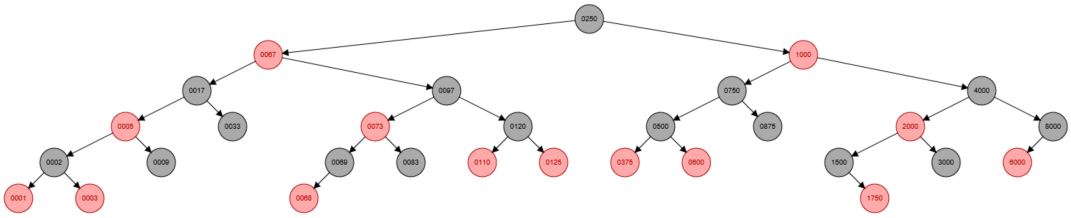
Case 16: Change root's color to red, keep other nodes the same as case 15.

Case 17 Figure



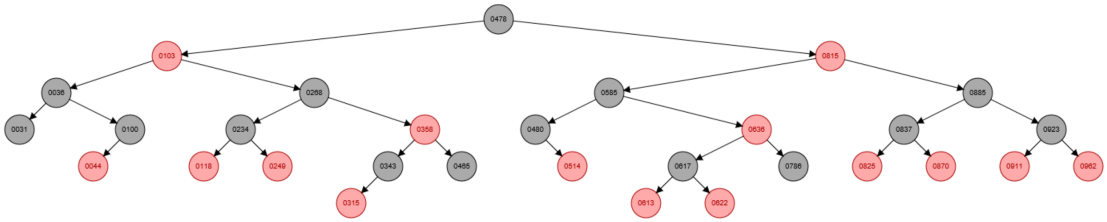
Case 18: Change "333" node to black, keep other nodes the same as case 17.

Case 19 Figure



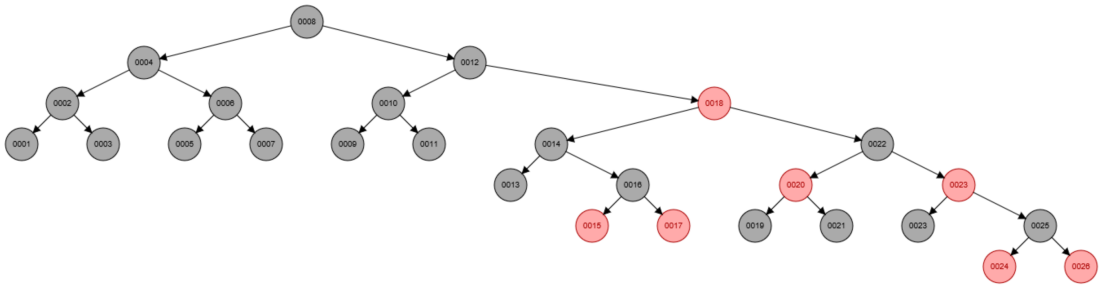
Case 20: Change “9” node to red, keep other nodes the same as case 19.

Case 21 Figure



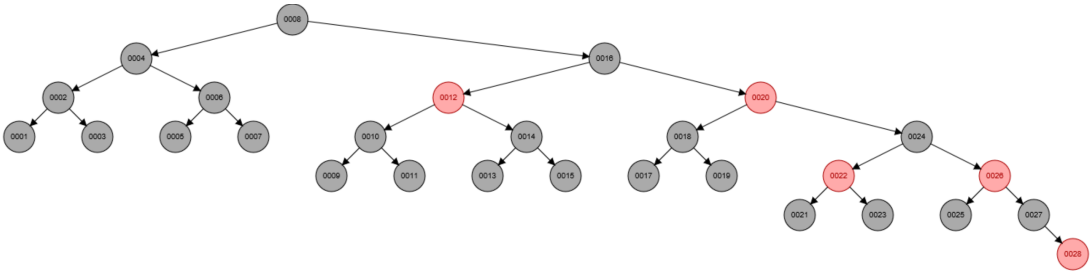
Case 22: Add node ”99"(overflow), keep other nodes the same as case 21.

Case 23 Figure



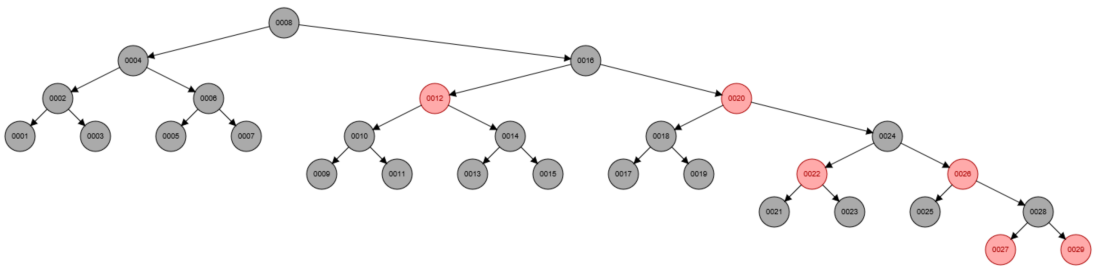
Case 24: Add node "27” (two nodes within same value), keep other nodes the same as case 23.

Case 25 Figure



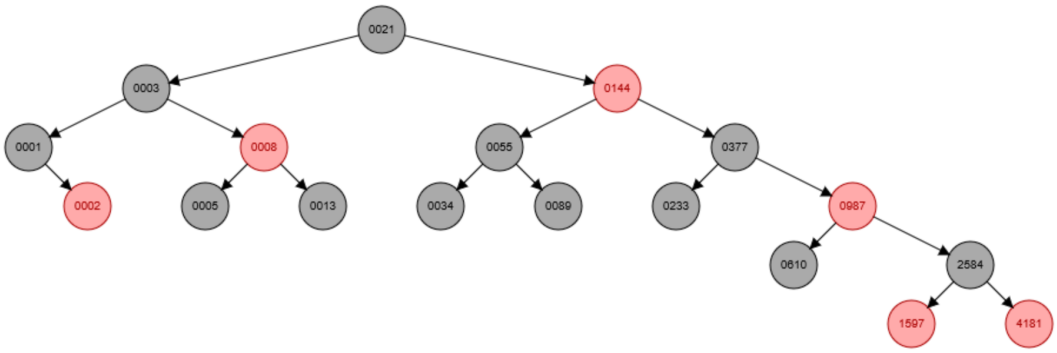
Case 26: Change "27” node’s color to red, keep other nodes the same as case 25.

Case 27 Figure



Case 28: Change "20” and “29” nodes’ value to "0”, keep other nodes the same as case 27.

Case 29 Figure



Case 30: Add node "1" (two nodes within same value), keep other nodes the same as case 29.

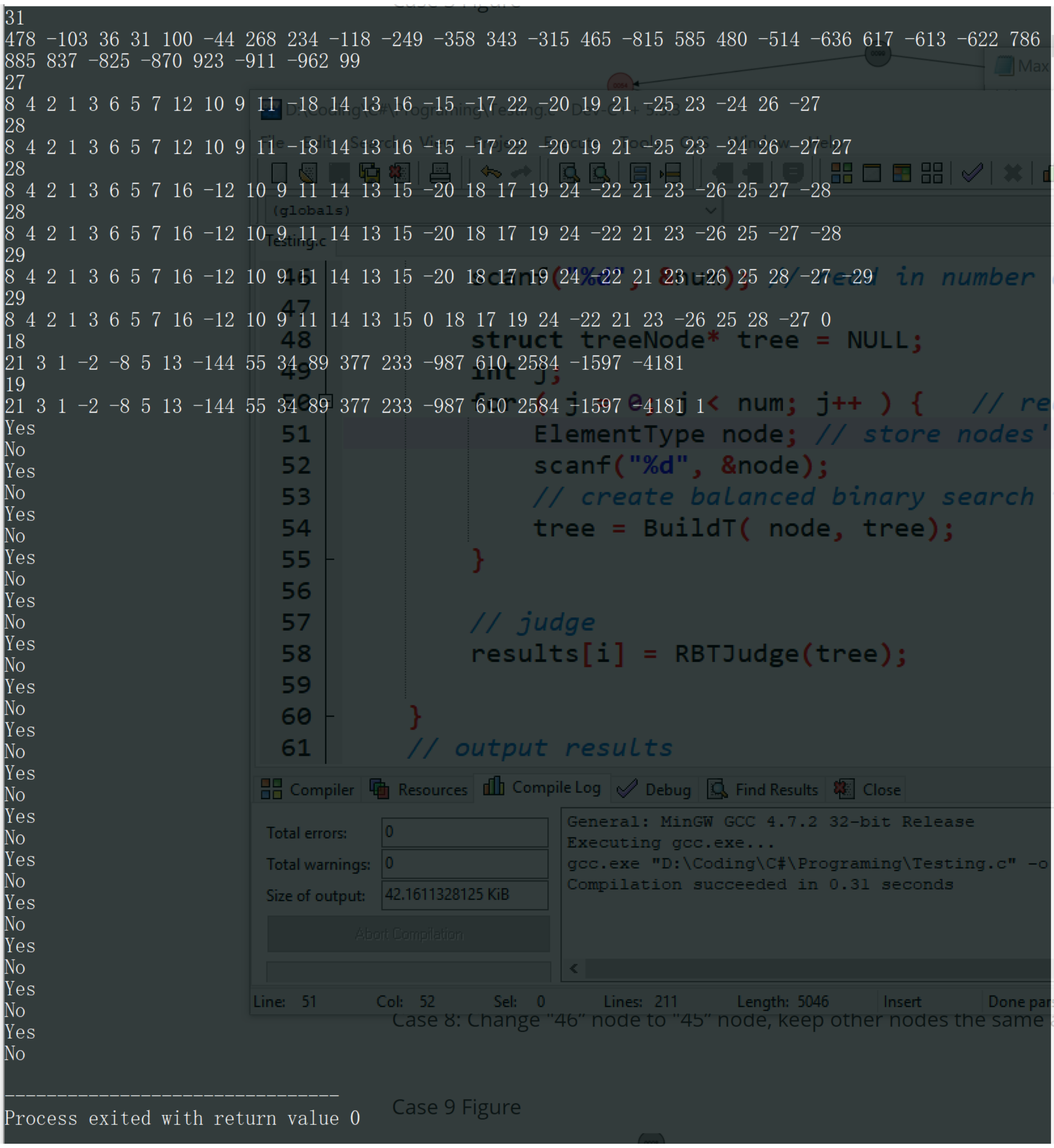
Expected Output

```
Yes // Case 1
No
Yes // Case 3
No
Yes // Case 5
No
Yes // Case 7
No
Yes // Case 9
No
Yes // Case 11
No
Yes // Case 13
No
Yes // Case 15
No
Yes // Case 17
No
Yes // Case 19
No
Yes // Case 21
No
Yes // Case 23
No
Yes // Case 25
No
Yes // Case 27
No
Yes // Case 29
No
```

Actual Output

```
Yes
No
Yes
No
```

```
Yes
No
Yes
No
Yes
No
Yes
No
Yes
No
Yes
No
Yes
No
Yes
No
Yes
No
Yes
No
Yes
No
Yes
No
Yes
No
Yes
No
```



Result: Pass

3. **Contain Null Tree**

Sample

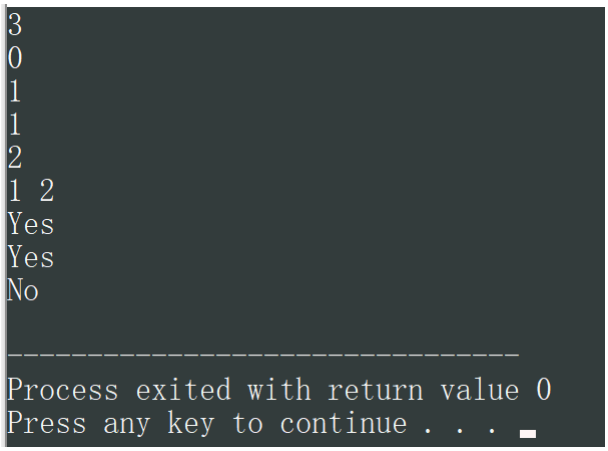
```
3 // Number of Cases
0 // Case1: Number of Nodes
1 // Case2: Number of Nodes
1 // Data
2 // Case3: Number of Nodes
1 2 // Data
```

Expected Output

```
Yes
Yes
No
```

Actual Output

```
Yes
Yes
No
```



Result: Pass

4. ## Invalid Search Tree

    1. **Input Same Value in One Case**

        Sample

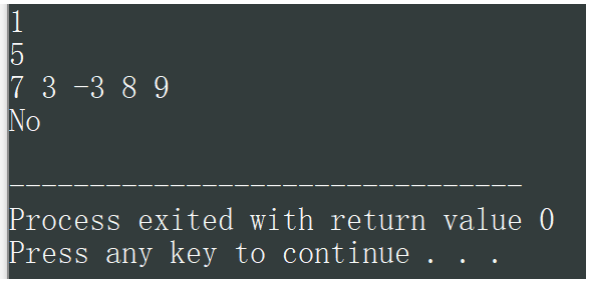
        ```
        1 // Number of Case
        5 // Case1: Number of Nodes
        7 3 -3 8 9 // Data
        ```

        Expected Output

        ```
        No
        ```

        Actual Output

        ```
        No
        ```

        Result: Pass

    2. **Input Data Contains '0'**

        Sample

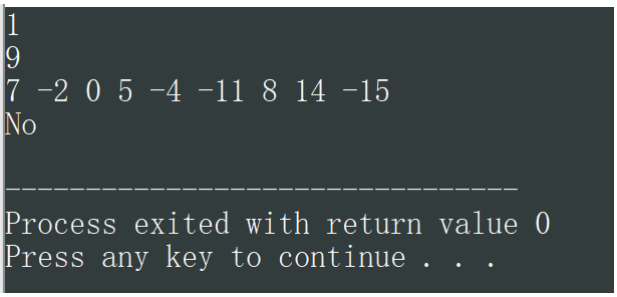
        ```
        1 // Number of Cases
        9 // Case1: Number of Nodes
        7 -2 0 5 -4 -11 8 14 -15 // Data
        ```

        Expected Output

```
No
```

Actual Output

```
No
```



Result: Pass

5. ## Numbers of Black Nodes Contract to Properties

Sample

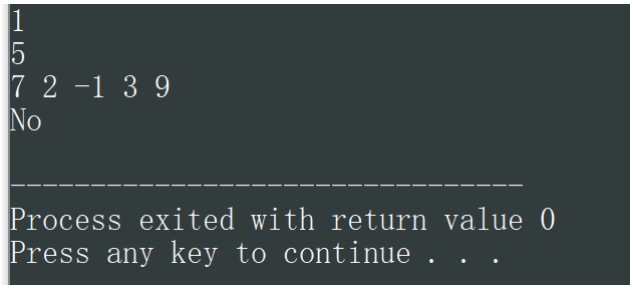
```
1 // Number of Cases
5 // Case1: Number of Nodes
7 2 -1 3 9 // Data
```

Expected Output

```
No
```

Actual Output

```
No
```



Result: Pass

6. ## Exist Consecutive Red Nodes

Sample

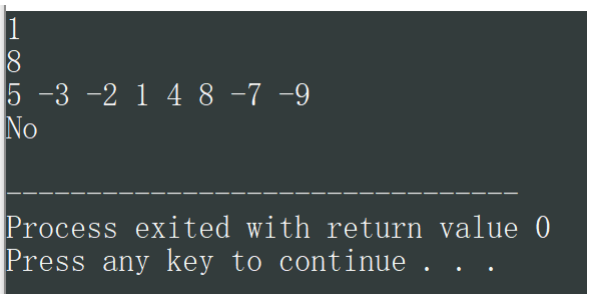
```
1 // Number of Cases
8 // Case1: Number of Nodes
5 -3 -2 1 4 8 -7 -9 // Data
```

Expected Output

```
No
```

Actual Output

```
No
```



Result: Pass

7. ## Given Sample

Sample

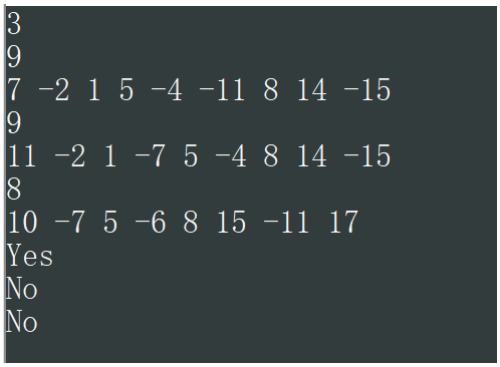
```
3 // Number of Cases
9 // Case1: Number of Nodes
7 -2 1 5 -4 -11 8 14 -15 // Data
9 // Case2: Nunber of Nodes
11 -2 1 -7 5 -4 8 14 -15 // Data
8 // Case3: Number of Nodes
10 -7 5 -6 8 15 -11 17 //Data
```

Expected Output

```
Yes
No
No
```

Actual Output

```
Yes
No
No
```



Result: Pass

8. ## Root Color is Red

Sample

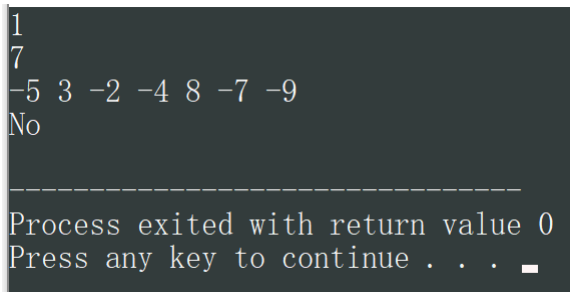
```
1 // Number of Cases
7 // Case1: Number of nodes
-5 3 -2 -4 8 -7 -9 // Data
```

Expected Output

```
No
```

Actual Output

```
No
```



Result: Pass

9. ## Correct Sample

Sample

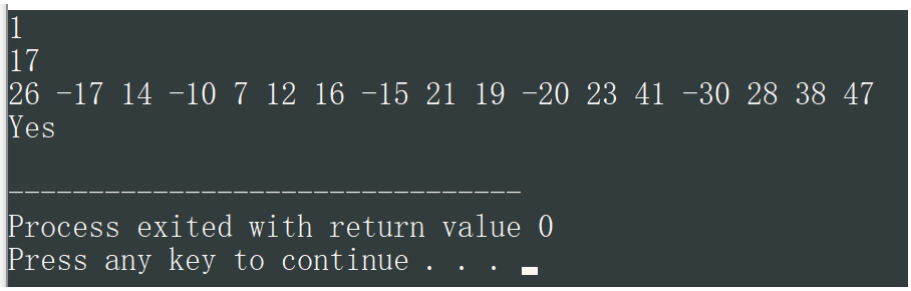
```
1 // Number of Cases
17 // Case1: Number of nodes
26 -17 14 -10 7 12 16 -15 21 19 -20 23 41 -30 28 38 47 // Data
```

Expected Output

```
Yes
```

Actual Output

```
Yes
```



Result: Pass

Final Result

The program works successfully for all testing samples.

Chapter 4: Analysis and Comments

Time Complexity

Firstly, assume the number of cases as **K** , the number of nodes for each case is **N**.

And it takes **O(N)** to traverse a tree within N nodes.

Secondly, compute each part's time complexity.

1. Main function

   It takes two-layers loop to read input, then it takes one-layer loop to print results.

   Assume it takes T(N) to build and judge tree. We'll compute it later.

   T(K) = O(K*T(N)) + O(K);

2. Build Tree

   The height of binary search tree can be varied **from log(N)** [Balanced]to **N**[Not balanced at all]. To insert a node, we have to recurse to find the suitable position. And input is random, so we cannot compute time complexity as a balanced tree. Thus,

   T(N) = O(N)

3. Judge Tree

   There' re several steps to judge whether the tree is Red-Black tree. They can be divided into several kinds as below

   1. O(1): Null Judge and Root Judge
   2. O(N): Search Tree Judge and Red-Nodes Judge. They only traverses the tree for one time.
   3. O(N^2): Number of Black-Nodes Judge. Because it counts every paths' black nodes while traversing.

   T(N) = 2 * O(1) + 2 * O(N) + O(N^2);

Thirdly, Compute the total time complexity.

T(K, N) = O(K * (O(N) + (2* O(1) + 2* O(N) + O(N^2)))) + O(N) = O(K* N^2) + O(N) = O(K* N^2);

Result: T(K, N) = O(K * N^2);

Space Complexity

Firstly, assume the number of cases as **K** , the number of nodes for each case is **N**.

And it takes **O(N)** to traverse a tree within N nodes.

Secondly, compute each part's time complexity.

1. Main function

   It takes two-layers loop to read input, then it takes one-layer loop to print results.

   Assume it takes T(N) to build and judge tree. We'll compute it later.

   S(K) = O(K*T(N)) + O(K);

2. Build Tree

It occupies certain space according to its number of nodes. Thus,

S(N) = O(N)

3. Judge Tree

There' re several steps to judge whether the tree is Red-Black tree. Only Search Tree Judge applies for a temporary link list. Other functions **do not** take any extra space.

S(N) = O(N) + O(N^2);

Thirdly, Compute the total time complexity.

T(K, N) = O(K * (O(N) + (O(N) + O(N^2))) + O(N) = O(K* N^2) + O(N) = O(K* N^2);

Result: S(K, N) = O(K * N^2);

Appendix: Source Code in C

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>

#define ElementType int
#define Max 30
#define MAX(a,b) (a>b)?a:b

// data structure used to organize tree
struct treeNode {
    ElementType val; // store node value
    bool isRed;      // mark node color
    bool isValid;    // mark node validity
    struct treeNode* leftT;  // link left child
    struct treeNode* rightT; // link right child
};
int results[Max]; // store results


struct treeNode* BuildT( ElementType data, struct treeNode *tree );
struct treeNode* NewNode( ElementType data, struct treeNode* node );

int SearchT( struct treeNode* tree );
int RBTJudge( struct treeNode *tree );
int NodeColor( struct treeNode* node );
int BlackNode( struct treeNode* tree );
int BlackNodeNum( struct treeNode* node );



/*---- Part 1: Main Function ----*/
/* Get input and Output results  */
int main() {

    // init and read input
    int cases; // store number of cases
    scanf("%d", &cases);
```

```
    int i;

    // read in cases and judge whether it's a Red-Black Tree
    for ( i = 0; i < cases; i++ ) {
        int num; // store number of nodes
        scanf("%d", &num); // read in number of nodes

        struct treeNode* tree = NULL;
        int j;
        for ( j = 0; j < num; j++ ) {   // read in nodes' value
            ElementType node; // store nodes' value
            scanf("%d", &node);
            // create balanced binary search tree by preorder sequence
            tree = BuildT( node, tree);
        }

        // judge
        results[i] = RBTJudge(tree);

    }
    // output results
    for ( i = 0; i < cases; i++ ) {
        if ( results[i] )
            printf ("Yes\n");
        else
            printf("No\n");
    }

    return 0;
}



/*---- Part 2: Build Tree ----*/
/*  Create a Binary Search Tree
    by Preorder sequence      */
struct treeNode* BuildT( ElementType data, struct treeNode* tree ) {
    // if tree is null, init it
    if ( !tree ) {
        tree = NewNode( data, tree);
        return tree;
    }
    // judge if input is valid
    if ( tree->val == abs(data) ) {
        tree->isValid = false;
    }
    // insert node in right position
    else if ( tree->val > abs(data) ) {
        tree->leftT = BuildT( data, tree->leftT );
    }
    else if ( tree->val < abs(data) ) {
        tree->rightT = BuildT( data, tree->rightT );
    }

    return tree;
}
```

```
struct treeNode* NewNode( ElementType data, struct treeNode* node ){
    // allocate new node space
    node = (struct treeNode *)malloc(sizeof(struct treeNode));
    // init new node and store data in
    node->leftT = NULL;
    node->rightT = NULL;
    node->val = abs(data);
    // judge if input is valid
    if ( !data ) // '0' is invalid
        node->isValid = false;
    else {
        // judge node's color
        node->isValid = true;
        if ( data < 0 )
            node->isRed = true;
        else
            node->isRed = false;
    }

    return node;
}



/*---- Part 3: Judge Tree ----*/
/*  Judge if the tree is a RBT
    by certain steps          */
int RBTJudge( struct treeNode *tree ) {

    // 1st, null tree is RedBlack Tree
    if ( !tree )
        return 1;

    // 2nd, it should be a Search Tree
    if ( !SearchT(tree) )
        return 0;

    // 3rd, root color should be Black
    if ( tree->isRed )
        return 0;

    // 4th, Recursively judge whether Numbers of Black Nodes are correct
    if ( !BlackNode(tree) )
        return 0;

    // 5th, Recursively judge whether Nodes' Colors are correct
    if ( !NodeColor(tree) )
        return 0;

    return 1;
}


// Judge Whether it's a Search Tree
int SearchT( struct treeNode* tree ) {
    struct treeNode* curT = tree; // point to Current Node
    if ( !curT )
        return 1;
```

```
    if ( !curT->isValid )
        return 0;
    // recursively traverse left child and right child
    return ( SearchT( curT->leftT) && SearchT(curT->rightT) );
}


// Judge whether Numbers of Black Nodes are Correct
int BlackNode( struct treeNode* tree ) {
    if ( !tree )
        return 1;
    // calculate Number of Black Nodes
    int leftNum, rightNum;
    leftNum = BlackNodeNum(tree->leftT);
    rightNum = BlackNodeNum(tree->rightT);
    // Compare
    if ( leftNum != rightNum )
        return 0;
    // Recurse
    return BlackNode(tree->leftT) && BlackNode(tree->rightT);
}


// Compute the Number of Black Nodes
int BlackNodeNum( struct treeNode* node ) {
    // Recursive export
    if ( !node )
        return 0;
    // Recurse to count Black Nodes
    int leftBlackNum, rightBlackNum;
    leftBlackNum = BlackNodeNum(node->leftT);
    rightBlackNum = BlackNodeNum(node->rightT);
    int blackNodeNum = MAX(leftBlackNum, rightBlackNum);
    // count current node
    if ( node->isRed )
        return blackNodeNum;
    return (1 + blackNodeNum);
}


// Detect if there're consecutive red nodes
int NodeColor( struct treeNode* node ) {
    // leaf(null) is black
    if ( !node )
        return 1;
    // if node ir red, then detect its' children
    if ( node->isRed ) {
        // if it has child and its child's node is red, invalid RBT
        if ( (node->leftT && node->leftT->isRed)
             || (node->rightT && node->rightT->isRed))
            return 0;
    }
    // Recurse left and right child
    return (NodeColor(node->leftT) && NodeColor(node->rightT));
}
```

Declaration

I hereby declare that all the work done in this project titled "Is is A Red-Black Tree” is of my independent effort.