Is It A Red-Black Tree

Name: a smart student

Date: 2023-11-7

Chapter 1: Introduction

The red-black tree is a self-balancing binary search tree that adds a storage bit to each node to represent the color of the node, which can be either red or black. By maintaining some basic properties, the red-black tree ensures balanced performance during operations such as insertion, deletion, and search.

The red-black tree has the following properties:

1. Every node is either red or black.
2. The root node is black.
3. Every leaf node (NIL node, empty node) is black.
4. If a node is red, then its children must be black.
5. All paths from any node to its descendant leaf nodes contain the same number of black nodes.

The design of the red-black tree aims to maintain the balance of the tree, ensuring stable performance during operations such as insertion and deletion. Compared to a regular binary search tree, the red-black tree has a more balanced height, which means that the time complexity for searching, inserting, and deleting remains stable and does not degrade due to tree imbalance.

The significance of this program lies in its implementation of insertion and validation functions for red-black trees, which can be used to determine whether an input sequence of nodes satisfies the properties of a red-black tree. Through this program, we can verify whether a given node sequence forms a valid red-black tree, thus helping to ensure the correctness and stability of the data structure in practical applications.

In summary, the red-black tree is an important data structure with self-balancing properties that enable it to play a crucial role in many application scenarios. The program's is able to verify whether a tree is a red-black tree when you input a tree and the number of its nodes.

Chapter 2: Algorithm Specification

pseudo-code

```
Algorithm Insert(root, a):
    if node[a][0] <= node[root][0]:
        if node[root][1]:
            Insert(node[root][1], a)
        else:
            node[root][1] = a
    else:
        if node[root][2]:
            Insert(node[root][2], a)
        else:
            node[root][2] = a

Algorithm Judge(root):
    if not node[root][1] and not node[root][2]:
        if not color[root]:
            return 1
        else:
            return 0

    if color[root]:
        if color[node[root][1]] or color[node[root][2]]:
            return -1

    lc = 0, rc = 0
    if node[root][1]:
        lc = Judge(node[root][1])
        if lc < 0:
            return -1

    if node[root][2]:
        rc = Judge(node[root][2])
        if rc < 0:
            return -1

    if lc == rc:
        return lc + not color[root]
    else:
        return -1

Algorithm main():
    Input K
    for k = 1 to K:
        Input N
        flag = 0
        Initialize node and color arrays
        for i = 0 to N-1:
            Input data
            if data < 0:
                if i == 0:
                    flag = 1
                Store node value and mark as red
            else:
                Store node value
            if i > 0:
                Insert(0, i)

        if flag or Judge(0) == -1:
            Output "No"
        else:
            Output "Yes"
```

Data Structure

Array of Arrays for Tree Node Information:

- Example: `int node[35][3]`
- Explanation: This 2D array is used to store the information of each node in the tree. Each row represents a node, and the three columns store the node's value, left child index, and right child index, respectively. For example, `node[2][0]` stores the value of the third node, `node[2][1]` stores the index of its left child, and `node[2][2]` stores the index of its right child.

Array for Node Colors:

- Example: `int color[35]`
- Explanation: This 1D array is used to store the color (black or red) of each node in the tree. For example, `color[4]` stores the color of the fifth node in the tree.

Tree

- explanation：I use the arrays in order to store the informations of the trees. The details are as above.

Chapter 3: Testing case

the case as the project gives me

Sample Input：

```
3
9
7 -2 1 5 -4 -11 8 14 -15
9
11 -2 1 -7 5 -4 8 14 -15
8
10 -7 5 -6 8 15 -11 17
```

Sample Output:

```
Yes
No
No
```

Output:

```
Yes
No
No
```

some case generated randomly

Input：

```
3
5
-5 -10 15 -20 25
0
1
5
```

Expected output:

```
No
Yes
Yes
```

Output：

```
No
Yes
Yes
```

these case contains a tree which is not a red-black tree, a tree with no node and a tree with just one node.

the test result in the public problem set

the problem is found in pintia, a platform which shares great number of problems, and I have asured the the problem is exactly the same as the project.

And following is the result

测试点	提示	内存(KB)	用时(ms)	结果	得分
0		332	4	答案正确	15 / 15
1		320	3	答案正确	6 / 6
2		324	3	答案正确	1 / 1
3		512	4	答案正确	8 / 8

if you want to personally confirm it, you can click this——the problem

the current status

all the test cases are passed.

Chapter 4: Analysis and comments

Analysis

Time Complexity Analysis:

- The time complexity of the `Insert` function depends on the shape of the tree and is O(N) in the worst case, where N is the number of nodes, as it needs to traverse the entire tree.
- Similarly, the time complexity of the `Judge` function also depends on the shape of the tree and is O(N) in the worst case, as it needs to traverse the entire tree.
- Overall, since both the `Insert` and `Judge` functions require traversing the entire tree, the overall time complexity of the program can be considered as O(N^2), where N is the number of nodes.

Space Complexity Analysis:

- The entire program uses two global arrays, `node` and `color`, whose space usage is proportional to the number of nodes, making the space complexity O(N).

Comments

Evaluation:

1. **Clear Program Structure:** The program has a high level of modularity, with well-defined functions, making it easy to read and understand.
2. **Recursive Implementation:** The use of recursion for binary tree insertion and validation makes the code concise and easy to understand.
3. **Basic Input Validation:** The program performs basic checks on input data to avoid potential illegal input scenarios.

Improvement Suggestions:

1. **Performance Optimization:** While the time complexity of red-black tree insertion and validation operations is low, further algorithm optimization could be considered, such as using iteration instead of recursion and introducing balance factors.
2. **Error Handling:** When the input data is invalid, the program directly outputs "No." Enhanced error messages could be added to assist users in identifying issues.
3. **Dynamic Memory Allocation:** Consider using dynamic memory allocation to support more flexible tree structures and avoid fixed array size limitations.
4. **Modularity:** Encapsulate global variables in structures to prevent global variable pollution and enhance code portability.

Appendix： Source code（in C）

```c
#include <stdio.h>
#include <string.h>

#define INF 99999999

// Define global variables
int K, N;
int node[35][3];  // Tree node information, each node has value and left and right
children
int color[35];    // Node color, 0 for black, 1 for red

// Insert a node into the tree
void Insert(int root, int a) {
    if (node[a][0] <= node[root][0]) {  // If the value of the node to be inserted is less
than or equal to the value of the current node
        if (node[root][1])                  // If the current node has a left child
            Insert(node[root][1], a);       // Recursively insert into the left subtree
        else
            node[root][1] = a;              // Otherwise, set it as the left child of the
current node
    } else {
        if (node[root][2])                  // If the current node has a right child
            Insert(node[root][2], a);       // Recursively insert into the right subtree
        else
            node[root][2] = a;              // Otherwise, set it as the right child of the
current node
    }
}

// Check if the properties of a red-black tree are satisfied
int Judge(int root) {
    if (!node[root][1] && !node[root][2]) {  // If it is a leaf node
        if (!color[root])                     // If it is a black node
            return 1;                         // Return height 1
        else
            return 0;                         // Otherwise, return height 0
    }

    if (color[root]) {                        // If the current node is red
        if (color[node[root][1]] || color[node[root][2]])  // If the child nodes have red
nodes
            return -1;                       // Return -1 indicating adjacent red nodes
    }

    int lc = 0, rc = 0;

    if (node[root][1]) {                      // If there is a left child
        lc = Judge(node[root][1]);            // Recursively check the left subtree
        if (lc < 0)
```

```
            return -1;                          // If the left subtree does not satisfy the
red-black tree property, directly return -1
    }

    if (node[root][2]) {                        // If there is a right child
        rc = Judge(node[root][2]);              // Recursively check the right subtree
        if (rc < 0)
            return -1;                          // If the right subtree does not satisfy the
red-black tree property, directly return -1
    }

    if (lc == rc)
        return lc + !color[root];               // If the heights of the left and right
subtrees are equal, return the height of the left subtree plus the color of the current
node (0 or 1)
    else
        return -1;                              // Otherwise, return -1 indicating that the
red-black tree property is not satisfied
}

int main() {
    scanf("%d", &K);  // Input the number of test cases
    for (int k = 0; k < K; k++) {
        scanf("%d", &N);  // Input the number of nodes for the current test case
        int flag = 0;
        int data;
        memset(node, 0, sizeof(node));  // Initialize the tree node information
        memset(color, 0, sizeof(color));  // Initialize the node colors
        for (int i = 0; i < N; i++) {
            scanf("%d", &data);
            if (data < 0) {
                if (i == 0)
                    flag = 1;                 // If the first node is red, set the flag to 1
                node[i][0] = -data;           // Store the node value (remove the negative
sign)
                color[i] = 1;                 // Mark the node as red
            } else {
                node[i][0] = data;            // Store the node value
            }
            if (i)
                Insert(0, i);               // Insert the current node into the tree
        }
        if (flag || Judge(0) == -1)
            printf("No\n");                 // Output "No" indicating that the red-black
tree property is not satisfied
        else
            printf("Yes\n");                // Output "Yes" indicating that the red-black
tree property is satisfied
    }
    return 0;
}
```

Declaration

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