Normal Performance Measurement (POW)

ZHONGWEI YU

Date: 2023-11-8

Chapter 1: Introduction

This problem deals with the properties and judgments of red-black trees, which are self-balancing binary search trees that have the following properties:

a. Each node is either red or black.

b. The root node must be black.

c. Every leaf node (NIL node) is black.

d. If a node is red, then its children must be black.

e. A simple path from any node to each of its leaves has the same number of black nodes.

The problem asks to judge whether a given binary search tree satisfies the properties of a red-black tree. The input consists of several test cases, and the first line of each test case is a positive integer K, indicating that there are K use cases in total. The second line of each use case is a positive integer N, representing the total number of nodes in the binary tree; The next line is the pre-ordered traversal sequence of the binary tree, separated by Spaces. In the sequence, red nodes are represented by negative signs. The output is "Yes" for each test case if the given tree is a red-black tree, and "No" otherwise.

This problem requires some understanding of the properties of red-black trees and pre-ordered traversal, as well as writing a program to determine whether a given binary search tree fits the definition of a red-

black tree.

Chapter 2: Algorithm Specification

1. Structure Definition

Define a **structure TreeNode** to represent a node in the binary tree with fields for key, left and right child pointers, and a color flag.

Define **type aliases** for **BinTree** as pointers to **struct TreeNode**.

2. Function Prototypes

BuildTree(int N): Builds a binary tree with N nodes based on user input.

createNode(int key, int color): Creates a new tree node with the given key and color.

freeTree(BinTree T): Frees the memory allocated for the tree nodes.

isRedBlackTree(BinTree T, int blackCount, int* pathBlackCount): Checks if the given tree satisfies the properties of a red-black tree.

3. Input/Output:

Read the number of test cases **K**.

For each test case, read the number of nodes **N** and their **keys/colors** to build the tree.

Output "**Yes**" if the constructed tree is a valid red-black tree or "**No**"

otherwise.

4. Function

a) BuildTree

Initialize an empty tree **T**.

For each input node, create a new node and insert it into the tree based on its key and color.

b) CreateNode

Allocate memory for a new tree node.

Initialize the node's key, color, and child pointers.

Return the newly created node.

c) FreeTree

Recursively free the memory for each node in the tree using post-order traversal.

d) IsRedBlackTree

Check the properties of a red-black tree for the given tree T:

Property 1: Every node is either red or black.

Property 2: The root node is always black.

Property 3: Every leaf (NULL) is black.

Property 4: If a node is red, then both its children are black.

Property 5: All simple paths from a node to descendant leaves contain the same number of black nodes.

Update the path black count while traversing the tree and checking properties recursively.

e) Main

Read the number of test cases K.

For each test case, build the tree, check if it's a red-black tree, and output the result.

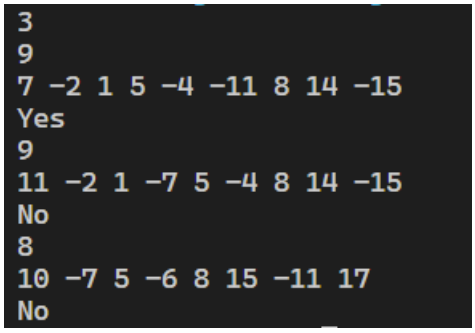
Free the memory of each constructed tree.

Chapter 3: Testing Results

1. Sample Input

```
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
```

Screenshot of Terminal:

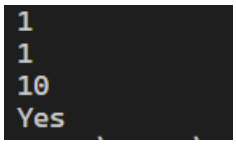


2. Designed Input

In this example, there is only one node and the value of the node is 10. This sample is used to test the ability of the program to handle a case with only one node.

```
Input:
1
1
10
```

Screenshot of Terminal:



3. random inputs

Repeat the input of a large number of random trees to verify the general applicability of the program

```
Input:
16
9
7 -2 1 5 -4 -11 8 14 -15
7
5 -3 1 4 -20 15 30
5
8 -6 1 7 9
9
11 -2 1 -7 5 -4 8 14 -15
8
10 -7 5 -6 8 15 -11 17
6
1 2 -3 -4 6 5
7
-1 2 4 -5 -7 3 6
```

```
5
-1 2 4 5 3
7
-1 2 4 -5 -7 3 6
8
-1 2 4 -5 7 3 6 8
6
-1 2 4 -5 3 6
4
-1 2 -3 4
9
-1 2 4 -5 -7 3 6 8 9
5
-1 2 3 -4 5
6
-1 2 4 -5 3 6
8
-1 2 -3 4 -5 6 -7 8
```

Screenshot of Terminal:

```
16
9
7 -2 1 5 -4 -11 8 14 -15
Yes
7
5 -3 1 4 -20 15 30
Yes
5
8 -6 1 7 9
Yes
9
11 -2 1 -7 5 -4 8 14 -15
No
8
10 -7 5 -6 8 15 -11 17
No
6
1 2 -3 -4 6 5
No
7
-1 2 4 -5 -7 3 6
No
5
-1 2 4 5 3
No
7
-1 2 4 -5 -7 3 6
No
8
-1 2 4 -5 7 3 6 8
No
6
```

```
-1 2 4 -5 3 6
No
4
-1 2 -3 4
No
9
-1 2 4 -5 -7 3 6 8 9
No
5
-1 2 3 -4 5
No
6
-1 2 4 -5 3 6
No
8
-1 2 -3 4 -5 6 -7 8
No
```

The results are generally in line with expectations(all pass).

Chapter 4: Analysis and Comments

1. **Time Complexity**:
    a) **BuildTree： O(NlogN)**, where N is the number of nodes in the tree. In each insertion, we need to traverse the path from the root to the leaf node, which takes a maximum of logN iterations in the worst case. Therefore, the overall time complexity is O(NlogN).
    b) **freeTree: O(N)**, where N is the number of nodes in the tree. Each node needs to be freed, resulting in a time complexity of O(N).
    c) **isRedBlackTree**: **O(N)**, where N is the number of nodes in the tree. Each node needs to be traversed once, resulting in a time complexity of O(N).

So the overall time complexity of the program is **O(NlogN)**, where N is the number of nodes in the tree, as the overall time complexity is dominated by the maximum time complexity among all function, which is **O(NlogN)**.

2. **Space Complexity**
    a) **BuildTree**: O(N), where N is the number of nodes in the tree,

as it need to create space for N nodes.

b) **freeTree**: **O(logN)**, where N is the number of nodes in the tree. During recursive calls, there can be a maximum of logN function call stack frames at the same time.

c) **isRedBlackTree**: **O(logN)**, where N is the number of nodes in the tree. During recursive calls, there can be a maximum of logN function call stack frames at the same time.

The overall space complexity of the program is **O(N)**, where N is the number of nodes in the tree. as the overall space complexity is dominated by the maximum space complexity among all function, which is **O(N)**.

In summary, the program has a time complexity of **O(NlogN)** and a space complexity of **O(N)**.

3. Comments on further possible improvements:

Have some ideas for improvement, but have no energy to implement.

a) The current implementation of the **isRedBlackTree** function uses a **recursive** approach. However, recursive function calls can consume a significant amount of memory, especially for large trees. Implementing an iterative approach using a **stack** or **queue** maybe

can help reduce the space complexity and potentially improve the performance of the function.

b) The traversal algorithm in the **isRedBlackTree** function uses recursive inorder traversal to check the color and path of each node. If it use an iterative approach for inorder traversal, using a stack to simulate the recursion process, maybe it can reduce function call overhead and improve the performance and memory efficiency of the program.

Appendix: Source Code (in C)


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

typedef struct TreeNode *BinTree;
struct TreeNode{
    int key;
    BinTree Left;
    BinTree Right;
    int color; //color=1 --> red, color=0 --> black.
};

BinTree BuildTree(int N);
BinTree createNode(int key, int color);
void freeTree(BinTree root); //free memory of tree
int isRedBlackTree(BinTree T, int blackCount, int* pathBlackCount);
//judge the tree T is red-black tree or not
```

```c
int abs(int n){     //just an absolute value function
    int bits = sizeof(int) * 8 - 1;
    return (n^(n>>bits)) - (n>>bits);
}

int main() {
    int K = 0,N = 0;
    scanf("%d",&K);
    while (K--){    //check every tree, one by one

    //build tree
        scanf("%d", &N);
        BinTree T = NULL;
        T = BuildTree(N);

    //judge tree
        int pathBlackCount = 0; //count the black node of the path
from a node to its leaf node
        if (
            T == NULL
            || (T->color == 1)
            || !isRedBlackTree(T, 0, &pathBlackCount)
        )
            if(K==0) printf("No"); //to make no '\n' after the last
one output
            else printf("No\n");
        else
            if(K==0) printf("Yes");   //to make no '\n' after the
last one output
            else printf("Yes\n");

        freeTree(T);    //free memory of this tree
    }

    return 0;
}
```

```
BinTree BuildTree(int N){
        BinTree T = NULL;
        int key = 0, color = 0;
        for (int i = 0; i < N; ++i) {

            //scan key and color of each node
            scanf("%d", &key);
            if (key < 0) {
                color = 1;  //red node
                key = -key;  //absolute value
            } else {
                color = 0;  //black node
            }

            //insert new node into the tree
            if (!T) {
                T = createNode(key, color); //creat root
            }else{  //creat son node
                BinTree current = T;    //the current node
                BinTree parent = NULL;  //the parent of current
node, after the current enter its subtree

                //let current reach the leaf node of the BST, while
recording its parent
                while (current != NULL) {
                    parent = current;
                    if (key < current->key)
                        current = current->Left;
                    else
                        current = current->Right;
                }

                //insert the new node of the BST
                if (key < parent->key)
                    parent->Left = createNode(key, color);
                else
                    parent->Right = createNode(key, color);
            }
```

```
        }

    return T;
}


BinTree createNode(int key, int color) {
    BinTree newNode = (BinTree)malloc(sizeof(struct TreeNode));
    newNode->key = key;
    newNode->Left = NULL;
    newNode->Right = NULL;
    newNode->color = color;
    return newNode;
}

// free memory
void freeTree(BinTree T) {
    if (!T) return;

    // free the memory, begin from the leaf node
    freeTree(T->Left);
    freeTree(T->Right);
    free(T);
}

// Red-Black-Tree judge fuction
int isRedBlackTree(BinTree T, int blackCount, int* pathBlackCount)
{
    if (T == NULL) {
        (*pathBlackCount)++;    //as every leaf (NULL) is black
(property 3)
        return 1;
    }

    //test property 4: If a node is red(1), then both its children
are black(0).
    if (T->color == 1)
        if (
            (T->Left && T->Left->color == 1)
```

```
            || (T->Right && T->Right->color == 1)
        )
            return 0;

    //count the number of black nodes in the path of the left and
right subtrees of the current node
    int leftPathBlackCount = 0, rightPathBlackCount = 0;
    int leftIsRedBlack = isRedBlackTree(T->Left, blackCount,
&leftPathBlackCount);
    int rightIsRedBlack = isRedBlackTree(T->Right, blackCount,
&rightPathBlackCount);

    /* test property 5: For each node, all simple paths from the
node to descendant leaves
     contain the same number of black nodes. */
    if (leftIsRedBlack && rightIsRedBlack && leftPathBlackCount ==
rightPathBlackCount){
        *pathBlackCount = T->color == 0 ? leftPathBlackCount + 1 :
leftPathBlackCount;  //we need to add the current node if it is
black
        return 1;
    }else{
        return 0;
    }
}
```

Declaration

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