Project 4

Yue Hu, JieFu Zhang, YuYang Wu

2023.3.14

Introduction

Given an input N, print out the number of distinct structures of RB-Tree with exactly N internal nodes.

The definition of "distinct" is: Once the structures of 2 RB-Trees are not exactly the same, they will be viewed as different. Two symmetric but not identical RB-Trees will also be viewed as different/distinct.

The property of RB-Tree is:

(1) Every node is either red or black.

(2) The root is black.

(3) All the leaves are NULL nodes and are colored black.

(4) Each red node must have 2 black descends (may be NULL).

(5) All simple paths from any node x to a descendant leaf have the same number of black nodes.

Algorithm Specification

Algorithm 1

```
#include <iostream>
using namespace std;
int redBlackCount(int n){
    int B[n+1][n+1];//根节点为黑色
    int R[n+1][n+1];//根节点为红色
    for(int i=0;i<=n;i++){
        for(int j=0;j<=n;j++){
            R[i][j]=0;
            B[i][j]=0;
        }
    }
    B[1][2]=1;B[2][2]=2;B[3][3]=1;B[3][2]=1;R[3][2]=1;//初始化一些数据
便于计算
```

```
    for(int k=4;k<=n;k++){//总个数，最多n-1个
        for(int i=1;i<k-1;i++){//左子树个数，最多k-1个
            for(int j=0;j<=k;j++){//左右子树黑节点层数，最多i个
                R[k][j]+=B[i][j]*B[k-i-1][j];
                B[k][j+1]+=R[i][j]*B[k-i-1][j]+B[i][j]*R[k-i-1]
[j]+B[i][j]*B[k-i-1][j];
            }
        }
    }
    int num=0;
    for(int i=0;i<=n;i++){
        num+=B[n][i];
    }
    return num;
}
int main(){
    cout<<redBlackCount(6);
}
```

The algorithm works by using dynamic programming to build up the count of red-black trees for each number of nodes. It initializes two matrices, B and R, to keep track of the counts of black-rooted and red-rooted trees, respectively. It then iterates through each possible number of nodes, k, and for each k, iterates through each possible number of nodes in the left subtree, i. For each combination of k and i, it calculates the count of red-rooted trees with k nodes and j black nodes in the left subtree, as well as the count of black-rooted trees with k nodes and $j+1$ black nodes in the left subtree. It does this by using the counts of red- and black-rooted trees for the left and right subtrees of size i and $k-i-1$, respectively.

Algorithm 2

We use generating function to solve this problem. Suppose $T_h\left(x\right)$ is the generating function for RB-Tree of black height h with a black root, while $R_h\left(x\right)$ is the generating function for RB-Tree of black height h with a red root.
Firstly, an initial condition, $T_1\left(x\right)$ is like this:

$T_1\left(x\right) = x + 2\,x^2 + x^3.$

The meaning of this expression is: When the black height of the RB-Tree is 1, there exist totally $1+2+1=4$ different structures.The reason is that, if there are only 1 node in the tree, then the node must be black(1 structure); if there are 2 nodes in the tree, then the root must be black, and the leave must be red, which can be either the left child or the right child of the black root(2 structures); if there are 3 nodes in the tree, then the root must be black, and its 2 children are both red(1 structure).
Here comes the recursion formula:

$$T_{h+1}(x) = x\left(R_h(x) + T_h(x)\right)^2 = x\,T_h(x)^2\left(1 + x\,T_h(x)\right)^2 \ldots\ldots(1)$$

The meaning of this formula is: The root of the RB-Tree of black height $h+1$ must be black, thus it will have two children of black height h.Besides, each child of the root can be red or black. We also notice that:

$$R_h(x) = x\,T_h(x)^2 \quad \ldots\ldots(2)$$

The reason is: a subtree of black height h, with a red root, must have 2 children of black height h with a black root.

Thus, the counting problem has been transferred to a polynomial calculation problem. Since the black height h is of $O(logn)$, the number of polynomial addition and multiplication are also of $O(logn)$. For each black height h, we first calculate $T_h(x)$ using formula (1), then calculate $R_h(x)$ according to formula (2).

Finally, the number of structures for a tree with exactly n internal nodes, can be derived by traversing $T_1(x), T_2(x), \ldots, T_h(x)$ and accumulating the coefficient of x^n.If we use an array to represent each T_h, and the index of each element is exactly matched with the exponent of x, then we only need to sum $T_h[n]$ for h in range $[1, log_2(n+1)]$.

If we use NTT to solve this problem, the time complexity of each polynomial calculation can be reduced to $O(logn)$.

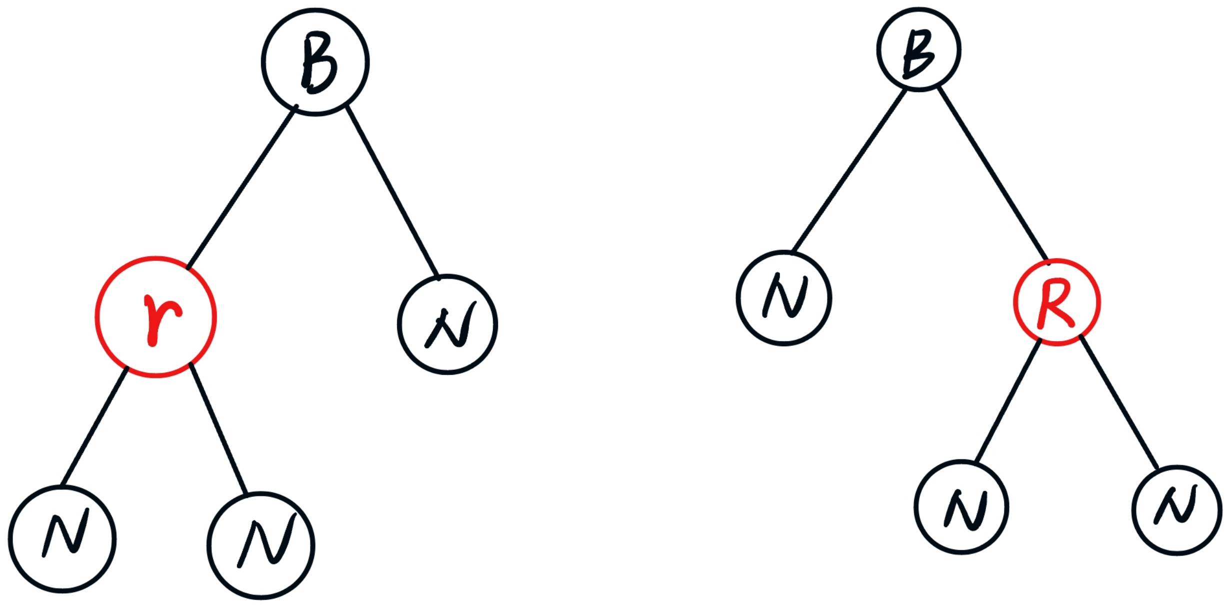
tips

NTT requires the module to have primary root. So we solve the problem with module 998244353, whose primary root is 3. For the same problem with module 1e9+7, we can use 3-module-NTT. But it's too complex for us, so we haven't tried that.

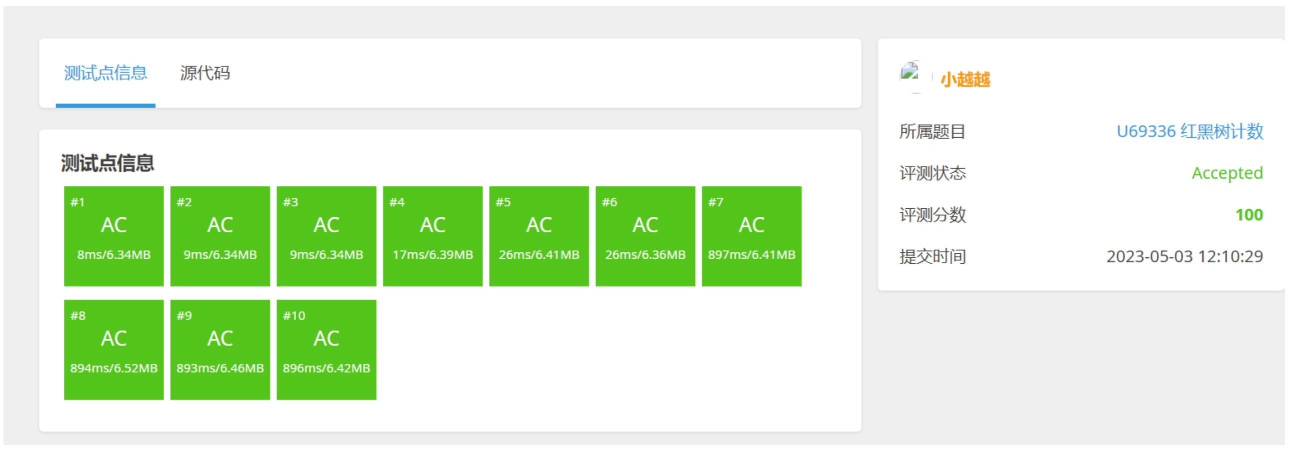
Test Results

Small Case Description

There are 2 red-black-trees when n=2. Note that mirror symmetric cases are not considered the same.



Algorithm 2 OJ result



测试点信息 源代码

小越越

所属题目 U69336 红黑树计数

评测状态 Accepted

评测分数 100

提交时间 2023-05-03 12:10:29

测试点信息

#1 AC 8ms/6.34MB	#2 AC 9ms/6.34MB	#3 AC 9ms/6.34MB	#4 AC 17ms/6.39MB	#5 AC 26ms/6.41MB	#6 AC 26ms/6.36MB	#7 AC 897ms/6.41MB
#8 AC 894ms/6.52MB	#9 AC 893ms/6.46MB	#10 AC 896ms/6.42MB				

Analysis

Algorithm 1

The time complexity of this algorithm is $O(\ n^3\ )$, since it involves three nested loops, each of which iterates up to n times. The space complexity is also $O(\ n^2\ )$, since it uses two matrices of size $n+1$ by $n+1$.

Algotithm 2

The optimal time complexity of this algorithm is $O(n\ (logn)\ ^2\ )$. The number of polynomial multiplication is $O(logn)$, while using NTT method, we can make the time complexity of each multiplication as $O(nlogn)$. The product of these two factors is $O(n\ (logn)\ ^2\ )$.

Declaration

I hereby declare that all the work done in this project is of my independent effort.