Red-black Tree

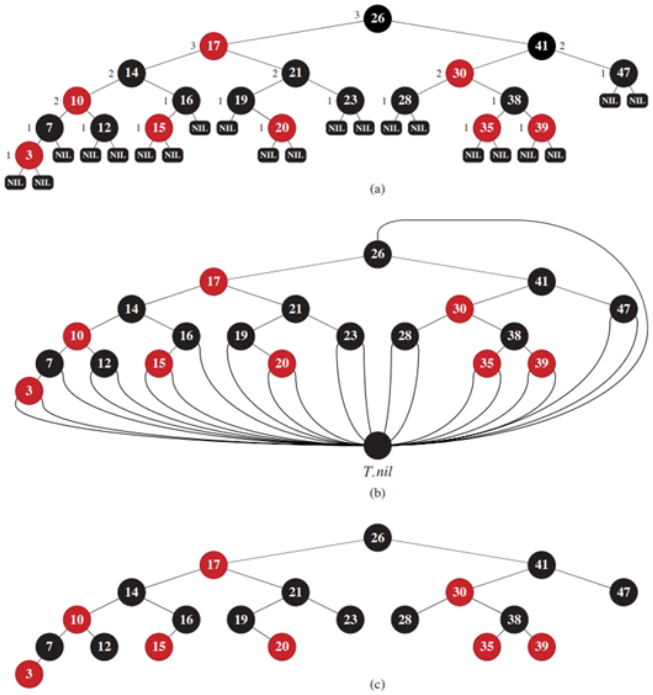
ADS Group 5
Li Jingxiong Sun ruoyv Li Qiuyv

date: 2024-05-01

Chapter 1: Introduction

There is a kind of binary tree named **red-black tree** in the data structure. It has the following 5 properties:

- (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.



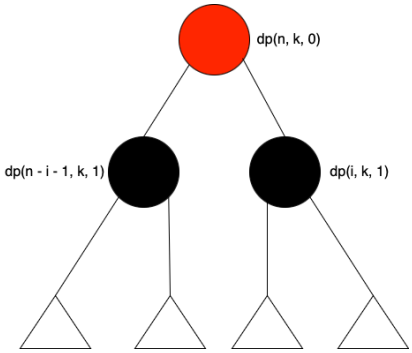
We call a non-NULL node an **internal node**. From property (5) we can define the **black-height** of a red-black tree as the number of nodes on the simple path from the root (excluding the root itself) to any NULL leaf (including the NULL leaf). And we can derive that a red-black tree with black height H has at least $2^H - 1$ internal nodes.

given a positive *N*, we want to compute how many distinct red-black trees are there that consist of exactly *N* internal nodes.

Chapter 2: Algorithm Specification

2.1 Auxiliary red root tree

From property(2) the root of a red-black tree must be black, we design a red-rooted tree as an auxiliary data structure. The root of the red-rooted tree is a red node, and the rest of the properties are exactly the same as those of the red-black tree. From this we can define the **black height** of the red root tree by analogy: consider the root node of the red root tree as black, so it is exactly the same as the original black height definition.



Use a two-dimensional array to store this structure. The first index is the total number of nodes and the second index is the black height of the tree.

```
RedBlackTree -> RBT[n][h]: n - node number                    1 ~ MAX_N
                           h - black height                   1 ~ MAX_HEIGHT
RedRootTree  -> RRT[n][h]: n - node number                    1 ~ MAX_N
                           h - black height                   1 ~ MAX_HEIGHT
                           (Assume that the red node is black)
```

2.2 Dynamic programming ideas

Follow the course's four-step approach:

- **Characterize an optimal solution**

Empty trees return 0 directly, the subtrees all have at least one node. $k >= 1$. First, initialize some base cases and then we can ensure that each subtree split has at least one node.

```
    RBT[0][0] = 0;      // n == 0
    RBT[1][1] = 1;      // n == 1
    RRT[1][1] = 1;      // n == 1
    RBT[2][1] = 2;      // n == 2
```

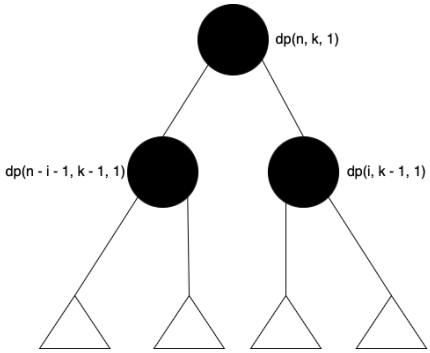
To construct a red-black tree with n(n>=3) nodes and black height bh, you need to construct two subtrees first. Assume that the left subtree has k nodes and the right subtree has $n-k-1$ nodes. Both subtrees need to satisfy the black height $bh-1$. The root node of the subtree can be either red or black.

- **Recursively define the optimal values**

We maintain an array of red-black trees by analyzing different situations.

1. If all the root of the subtree is black, then the number of possible trees is

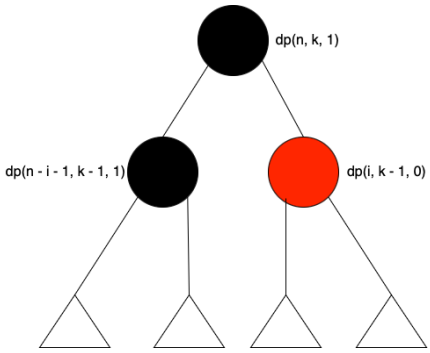
$$\sum_{k=1}^{n-2} RBT[k][bh-1] * RBT[n-k-1][bh-1]$$



2. If the left subtree root is black and the right subtree root is red, the number of possible trees is

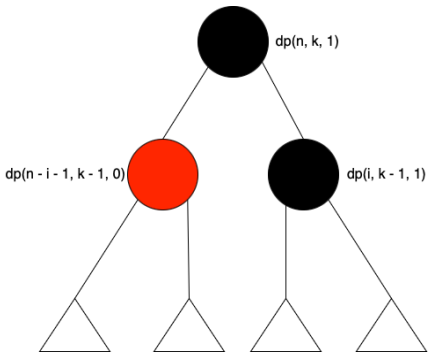
$$\sum_{k=1}^{n-2} RBT[k][bh-1] * RRT[n-k-1][bh]$$

Since the black height of our red root tree is calculated assuming that the root node is black, the actual black height of the red root tree with a black height of bh is $bh-1$. So we use $RRT[n-k-1][bh]$.



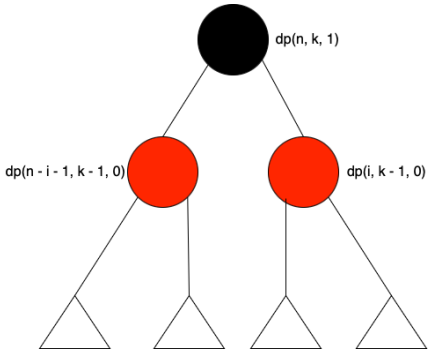
3. If the left subtree root is red and the right subtree root is black, the number of possible trees is

$$\sum_{k=1}^{n-2} RRT[k][bh] * RBT[n-k-1][bh-1]$$



4. If all the root of the subtree is red, then the number of possible trees is

$$\sum_{k=1}^{n-2} RRT[k][bh] * RRT[n-k-1][bh]$$

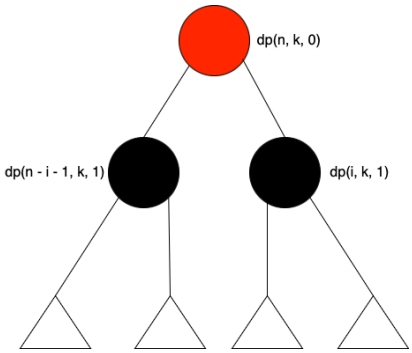


Add the four cases and simplify using the multiplication rule

$$RBT[n][bh] = \sum_{k=1}^{n-2}((RBT[k][bh-1] + RRT[k][bh]) * (RBT[n-k-1][bh-1] + RRT[n-k-1][bh]))$$

At the same time, we need to maintain the red root tree array. The children of the red node must be black nodes, so there is only one situation as follows.

$$RRT[n][bh] = \sum_{k=1}^{n-2} RBT[k][bh-1] * RBT[n-k-1][bh-1]$$



- **Compute the values in some order**

Use a triple loop, the first loop is the number of nodes, and build the solution space from small to large. The second layer loops through the red-black tree black height. The third loop divides the left and right subtrees by k.

```
for i=3 to N:
    for j=min_bh to max_bh:
        for k=1 to i-2:
            update RBT[i][j]
            update RRT[i][j]
```

i starts from 3, k starts from 1 to ensure that the left and right subtrees must have at least one node, corresponding to the base case. Reduce the amount of calculation by calculating the range of black height of the red-black tree.

$$h \leq 2 \times bh$$
$$2^{bh} - 1 \leq n$$
$$\left\lfloor \frac{\log_2(n+1)}{2} \right\rfloor \leq bh \leq \log_2(n+1)$$

- **Reconstruct the solving strategy**

If you need to build a specific red-black tree, you need to modify the iterated array to store more information.

Chapter 3: Testing Results

Pass PTA test case. **result mod 1000000007.**

评测详情

测试点	提示	内存(KB)	用时(ms)	结果	得分
0		400	2	答案正确	18 / 18
1		412	2	答案正确	7 / 7
2		400	4	答案正确	7 / 7
3		416	12	答案正确	2 / 2
4		288	2	答案正确	1 / 1

N	**50**	**100**	**150**	**200**	**250**
result	37914925	167844408	742395690	728464467	940174255
N	**300**	**350**	**400**	**450**	**500**
result	527048662	341330501	79324284	431612116	905984258

Chapter 4: Analysis and Comments

- Time complexity

The problem of the number of red-black trees is solved by dynamic programming. The core part of the algorithm is the triple loop, but it is obvious that the number of traversals of the second loop is $2log(N+1)$ of the black height of the tree.

So the overall time complexity is

$$T(N) = O(N^2 logN)$$

- Space complexity

The program only uses two two-dimensional arrays to store the number of red-black trees. The first index of the array is the number of nodes N, and the second index is the black height. So the overall space complexity is

$$S(N) = O(NlogN)$$

- Optimization

The number of triple loops can still be reduced. The range of black height is related to the way the number of subtree nodes is divided. If the subtree nodes are not divided evenly, the black height will be smaller and there will be fewer types of red-black trees. At this time, the range of the loop should be reduced.

Appendix: Source Code

```
#include <iostream>
#include <cmath>
#define MAX_N 501           // the max number of nodes
#define MAX_HEIGHT 16       // the max height of red black tree
#define MOD 1000000007      // find the remainder of the result
using namespace std;
/*
    RedBlackTree -> RBT[n][h]: n - node number                      1 ~ MAX_N
                               h - black height                     1 ~ MAX_HEIGHT
    RedRootTree  -> RRT[n][h]: n - node number                      1 ~ MAX_N
                               h - black height                     1 ~ MAX_HEIGHT
                               (Assume that the red node is black)


    base case: n = 1
        RBT[1][1] = 1
        RRT[1][1] = 1

    base case: n = 2
        RBT[2][1] = 2

    iterative: for n=3, 4, ..., N
        RBT[n][h] = RBT[k][h-1] * RBT[n-k-1][h-1]   for k=1, 2, ..., n-2
                  + RBT[k][h-1] * RRT[n-k-1][h]     for k=1, 2, ..., n-2
                  + RRT[k][h] * RBT[n-k-1][h-1]     for k=1, 2, ..., n-2
                  + RRT[k][h] * RRT[n-k-1][h]       for k=1, 2, ..., n-2
        RRT[n][h] = RBT[k][h-1] * RBT[n-k-1][h-1]   for k=1, 2, ..., n-2
*/
long long RBT[MAX_N][MAX_HEIGHT];       // red black tree
long long RRT[MAX_N][MAX_HEIGHT];       // red root tree
int main()
{
    // define some variables
    int i, j, k, N;
    int min_bh, max_bh;
    long long res = 0;
    // input the number of nodes (a positive integer N < 500)
    cin >> N;
```

```cpp
    // initialize base case
    RBT[1][1] = 1;
    RBT[2][1] = 2;
    RRT[1][1] = 1;
    // dynamic progamming loops
    // start with 3 nodes ensure each subtree to have a node
    for(i=3; i<=N; i++){
        // compute black height range
        min_bh = log2(i + 1) / 2;
        max_bh = log2(i + 1);
        // for fixed number of nodes, loop over height
        for(j=min_bh; j<=max_bh; j++){
            // for fixed height, loop over the number of nodes in the subtree
            for(k=1; k<i-1; k++){
                // the roots of the subtrees are all black
                RBT[i][j] += RBT[k][j-1] * RBT[i-k-1][j-1];
                RBT[i][j] %= MOD;
                // the root of the left subtree is black, and the root of the right subtree is red
                RBT[i][j] += RBT[k][j-1] * RRT[i-k-1][j];
                RBT[i][j] %= MOD;
                // the root of the left subtree is red, and the root of the right subtree is black
                RBT[i][j] += RRT[k][j] * RBT[i-k-1][j-1];
                RBT[i][j] %= MOD;
                // the roots of the subtrees are all black
                RBT[i][j] += RRT[k][j] * RRT[i-k-1][j];
                RBT[i][j] %= MOD;

                // the roots of the red root tree are all black
                RRT[i][j] += RBT[k][j-1] * RBT[i-k-1][j-1] % MOD;
                RRT[i][j] %= MOD;
            }
        }
    }
    // for N nodes, add up the number of trees of different heights
    for(i=0; i<=max_bh; i++){
        res += RBT[N][i];
        res %= MOD;
    }
    // ouput the result
    cout << res << endl;
    return 0;
}
```

Author List

Li Jingxiong Sun Ruoyv Li Qiuyv : Complete code and report writing together.

Declaration

We hereby declare that all the work done in this project titled Red-black Tree is of our independent effort as a group.