Red-black Tree

Chapter1: Introduction

1.1 Background

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 (maybe 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.

Here comes the question: given a positive N, how many distinct red-black trees are there that consist of exactly N internal nodes?

1.2 Project Overview

We are going to solve the above question by **dynamic programming**. We will classify the red-black trees by the number of internal nodes and the black height. We will use two **2D arrays** to store the number of trees with a black root and a red root, respectively. We will iterate through all possible black heights and calculate the number of trees with a black root and a red root for each number of internal nodes and black height. Finally, we will sum up the number of trees with a black root at all possible heights to get the final result.

Chapter2: Algorithm Specification

2.1 Data Structure and Variables

We use 2 2D arrays to store the number of trees with a black root and a red root, respectively. We also define some constants for the maximum number of nodes, a large prime for modulo operations, and the maximum height based on the possible maximum black height.

1. **Constants**:
    - NODES_MAX: Maximum number of nodes
    - OVERFLOW: A large prime for modulo operations to prevent integer overflow
    - HEIGHT_MAX: Adjusted maximum height based on possible maximum black height
2. **Global Variables**:
    - BlackDP: A 2D array to store the count of trees with a black root, classified by number of nodes and black height. Rows represent the number of nodes, and columns represent the black height, and the value at each cell represents the number of distinct trees with a black root.

- RedDP: A 2D array to store the count of trees with a red root, classified by number of nodes and black height. Rows represent the number of nodes, and columns represent the black height, and the value at each cell represents the number of distinct trees with a red root.

> Though an actual Balck-Red Tree can't have a red root, we still need to consider the case of a red root in the dynamic programming process. This is because the **subtrees of a red root tree can have a black root**, and the black height of the left subtree and right subtree must be the same.

```
// Constants
const int NODES_MAX = 505;      // Maximum number of nodes
const int OVER = 1000000007;  // A large prime for modulo operations to prevent integer overflow
const int HEIGHT_MAX = 30;      // Adjusted maximum height based on possible maximum black height

// Global variables for dynamic programming results
vector<vector<long long> > BlackDP(NODES_MAX, vector<long long>(HEIGHT_MAX, 0)); // Count of trees with a black root
vector<vector<long long> > RedDP(NODES_MAX, vector<long long>(HEIGHT_MAX, 0));   // Count of trees with a red root
```

2.2 Dynamic Programming

1. initialize base cases for the dynamic programming table
2. iterate through all possible black heights
3. for each number of nodes, calculate the number of trees with a black root and a red root at the current black height
4. calculate the final result by summing up the number of trees with a black root at all possible heights

```
int main() {
    int n = 0;  // Input total number of nodes
    cin >> n;

    // Initialize base cases, root node don't count into BH, NIL count into BH
    BlackDP[1][1] = 1;  // 1 for node number, 1 for black height, 1 for distinct trees number
    BlackDP[2][1] = 2;  // 2 for node number, 1 for black height, 2 for distinct trees number
    RedDP[1][1] = 1;    // 1 for node number, 1 for black height, 1 for distinct trees number

    // Dynamic programming to fill the table
    for (int i = 3; i <= n; i++) {
        for (int j = static_cast<int>(log2(i + 1) / 2); j <= static_cast<int>(log2(i + 1) * 2) + 1; j++) {
            for (int k = 1; k < i; ++k) {
                // For each Black Root Tree, its subtrees can have 1 black root or 1 red root
                // And the black height of left subtree and right subtree must be the same
                BlackDP[i][j] = (BlackDP[i][j] + (((BlackDP[k][j - 1] + RedDP[k][j]) *
                                                   (BlackDP[i - 1 - k][j - 1] + RedDP[i - 1 - k][j])) % OVER)) % OVER;
                // For each Red Root Tree, its subtrees must have black root
                // And the black height of left subtree and right subtree must be the same
                RedDP[i][j] = (RedDP[i][j] + ((BlackDP[k][j - 1] * BlackDP[i - 1 - k][j - 1]) % OVER)) % OVER;
            }
```

```
        }
    }

    // Calculate the final result for trees with a black root at all possible heights
    long long totalTrees = 0;
    for (int i = 0; i < HEIGHT_MAX; i++) {
        totalTrees = (totalTrees + BlackDP[n][i]) % OVER;
    }

    // Output the final result
    cout << totalTrees << endl;

    return 0;
}
```

Chapter3: Test cases

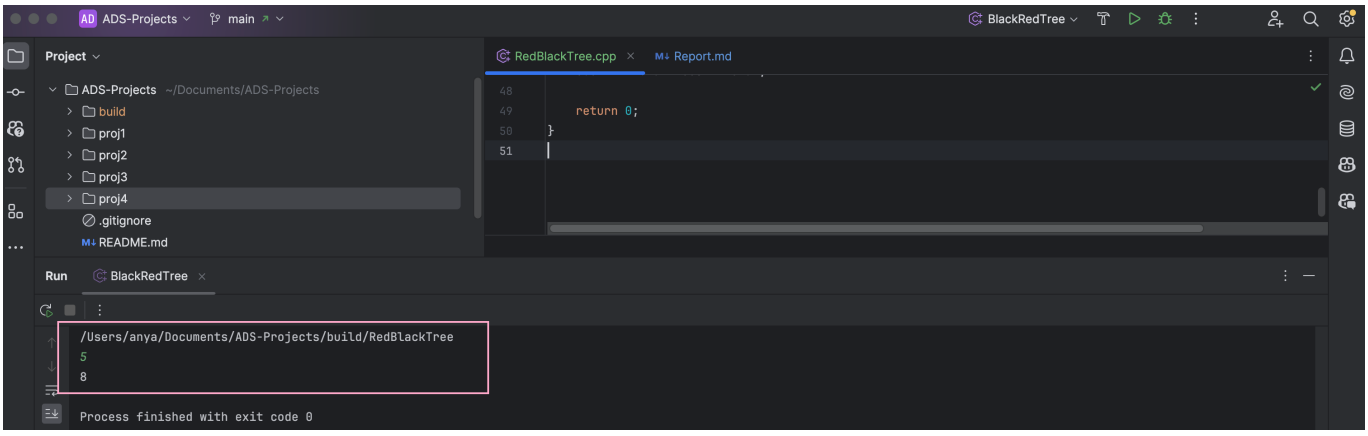
3.1 Normal Test Cases

Input:

```
5
```

Output:

```
8
```



3.2 Large Test Cases

Input:

```
100
```

Output:

```
167844408
```

```
/Users/anya/Documents/ADS-Projects/build/RedBlackTree
100
167844408

Process finished with exit code 0
```

Input:

```
500
```

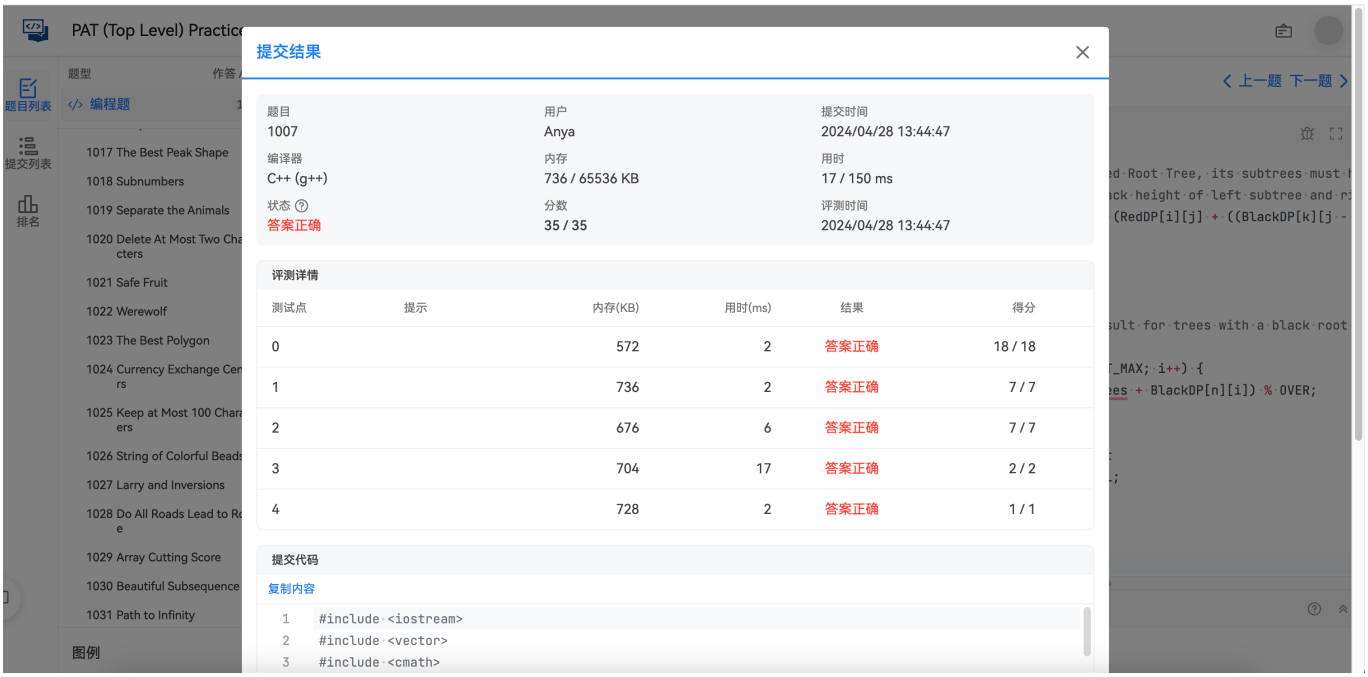
Output:

```
905984258
```

```
/Users/anya/Documents/ADS-Projects/build/RedBlackTree
500
905984258

Process finished with exit code 0
```

PTA TestCases



Chapter4: Analysis and Commnets

4.1 Time Complexity

The **time complexity** of the dynamic programming solution is $O(N^2 \log N)$, where N is the total number of nodes. The outer loop iterates through all possible numbers of nodes, and the inner loop iterates through all possible black heights. The calculation of the number of trees with a black root and a red root at each black height takes $O(N)$ time. Therefore, the overall time complexity is $O(N^2 \log N)$.

4.2 Space Complexity

The **space complexity** of the dynamic programming solution is $O(N \log N)$, where N is the total number of nodes. We use two 2D arrays to store the number of trees with a black root and a red root, respectively. The size of each array is $N \times \log N$. Therefore, the overall space complexity is $O(N \log N)$.

4.3 Comments

The dynamic programming solution is an efficient way to solve the problem of counting the number of distinct red-black trees with a given number of internal nodes. By classifying the trees based on the number of nodes and black height, we can calculate the number of trees with a black root and a red root at each black height. The final result is obtained by summing up the number of trees with a black root at all possible heights. The time complexity of the solution is $O(N^2 \log N)$, and the space complexity is $O(N \log N)$.

There's still room for improvement in the code. For example, we can use a more efficient way to calculate the logarithm of the number of nodes and optimize the modulo operations to prevent integer overflow. We can use FFT to speed up the multiplication of large numbers and reduce the time complexity of the solution. We can also optimize the space complexity by using a more compact data structure to store the dynamic programming results. The thought of optimization comes from PTA testcases when we failed to run it in time with python, though using cpp we can pass it with normal algorithm.