Red-black Tree

Date: 2024-11-10

目录

Chapter 1: Introduction

红黑树是一种依靠为节点增加颜色属性并据此调整平衡的二叉树（将在数据结构类型部分继续介绍）。

`Internal Node`:对于一棵红黑树，通常意义上的“叶子”具有2个`NULL leaf`的孩子，且被认为是黑色，因此除了"NULL leaf“的节点都是`Internal Node`。

本次`Project`的**目的**在于探究：对于给定的`Internal Node`的数量，有多少种不同的红黑树。

整体流程包括：

1. 红黑树性质分析
2. 转移方程设计
3. 函数分析
4. 测试与结果分析

Chapter 2: Data Structure / Algorithm Specification

2.1 红黑树性质

2.1.1 基本性质

> 给出基本性质，为转移方程的设计提供依据。

在此介绍`Chapter 1`中**红黑树的定义**：

1. 每个节点是红色或者黑色的一种；
2. 根节点`root`是黑色；
3. 所有的`NULL leaf`被定义为黑色；
4. 每一个红色节点的两个孩子一定是黑色（包括`NULL leaf`），即**不能具有连续的红色节点**；
5. 对于任意一个节点，在其到叶子节点的所有简单路径中，经历的黑色节点数目均相同；

```
typedef struct AvlTreeNode{
    int key;
    int height;
    struct AvlTreeNode *left;
    struct AvlTreeNode *right;
}AvlTreeNode;
```

> 在项目实现中，并未实际用到上述的数据结构，而是利用红黑树的性质获得转移方程实现。在此给出结构体，使得描述更加清晰。

2.1.2 补充性质

> 推导补充性质，为约束循环条件的时间复杂度优化提供理论依据。

1. **`root`黑高为`BH`的红黑树至少有 $2^{BH}-1$ 个节点**

> **Proof**:
>
> 1. 根据红黑树的基本性质`5`，任意节点的`simple path`上的黑色节点数目相同，如果一棵红黑树只具有黑色节点，那么它一定是一棵完美二叉树。否则，在倒数第二层的节点的`BH`就不满足性质`5`。而`root`的黑高在此时恰好时BST的`height`，因此节点数为$2^{BH}-1$；
> 2. 在`1`的情况下，我们可以在任意两个黑色节点之间插入一个红色节点。红色节点不会破坏其他性质，因此节点数增多且仍旧为合理的红黑树；
> 3. 综上得证。

2. **`root`黑高为`BH`的红黑树至多有 $2^{2*BH}-1$ 个节点**

> **Proof**:
>
> 1. 根据补充性质`1`的思路延续，为了尽可能多地插入红色节点，且满足基本性质`4`,考虑从`root`开始各层节点是红黑相间的连续；
> 2. 为了尽可能得到多的红色节点，除了`NULL leaf`层之外，最底层一定是红色节点的层；
> 3. 因此不难发现，如果`root`的黑高为`BH`，此时整棵树的`height`为`2*BH`，且为完美二叉树；
> 4. 经过计算，发现此时最多有 $2^{2*BH}-1$ 个内点, 即 BH >= $\lceil log_4(N+1) \rceil$

2.2 转移方程

2.2.1 思路分析

考虑能否将问题转化为子问题并填表解决。对于`Internal Node`为1的红黑树，显然红黑树的种类只有1个；在构造具有`N`个`Internal Node`的红黑树时，将`root`的数目减去，剩余`N-1`个内点将分配到`root`的左右子树当中，只需要将左右子树的可能性相乘并累加不同规模的情况即可。

因此，接下来刻画获得`N`个内点种类的红黑树的**转移方程**:

1. 如上所述，除`root`之外的`N-1`个内点分配给左右子树。为了满足红黑树的基本性质`2`与`5`，`root`被标记为黑色，且左右子树的黑高`BH`须相等，由此可见我们在动态规划时需要考虑记录黑高`BH`；
2. 在`root`为黑色、黑高为`BH`的条件下，其左右子节点可以为红色或者黑色，一共有`4`种组合。假设左子树的根节点是黑色，那么其黑高应当为`BH-1`，这是因为黑高的定义是`simple path`上不包括自身的黑色节点数，而将其作为`root`的子树时，从新的`root`到叶子节点将经历子树的`old root`。同理，如果右子树的`root`为红色，那么其黑高应当为`BH`，这是因为在新的`root`的`simple path`上需要经过`BH`个黑色节点，而红色节点自然不会被纳入在列；
3. 经过步骤`2`的分析发现，转移方程中需要`root`为红色、“黑高”为`BH`、内点为`i`的数据，这要求我们在记录正常红黑树（`root`为黑色）的同时，记录`root`为红色的对称数据。

2.2.2 函数分析

由上述思路的得到以下转移方程的**核心**部分：

```
    for (int internal_nodes = 3; internal_nodes <= N; internal_nodes++) {
        for (int BH = (int)ceil(log(internal_nodes + 1) / log(4)); pow(2, BH) - 1
<= internal_nodes; BH++) {
            //compute the subproblems
            for (int left_size = pow(2,BH-1)-1 ; left_size < pow(4,BH)-1 &&
left_size < internal_nodes-1; left_size++) {
                int right_size = internal_nodes - 1 - left_size;

                red_root_dp[internal_nodes][BH] += black_root_dp[left_size][BH -
1] * black_root_dp[right_size][BH - 1];

                //when the subtree's root is black, its BH is actually its
black_height +1 ,so we use BH-1 for calculating BH
                RB_black_root = red_root_dp[left_size][BH] *
black_root_dp[right_size][BH - 1];
                BR_black_root = black_root_dp[left_size][BH - 1] *
red_root_dp[right_size][BH];
                BB_black_root = black_root_dp[left_size][BH - 1] *
black_root_dp[right_size][BH - 1];
                RR_black_root = red_root_dp[left_size][BH] *
red_root_dp[right_size][BH];

                black_root_dp[internal_nodes][BH] += RB_black_root +
BR_black_root + BB_black_root + RR_black_root;

            }
        }
    }
```

1. `Internal Node`数目为`1`,`2`时的情况由初始化给出，方便转移方程的书写；
2. 整体上包括三层遍历：最外层为内点数的递增，中间循环为给定节点数时遍历`root`可能的黑高情况，最内层的循环为遍历左右子树的`size`；
3. 为了减小时间复杂度，我们根据红黑树的性质，对循环的遍历区间作了精细的规定，接下来解释各层循环的取值区间：
    1. 最外层，根据动态规划，从节点为`3`开始递增到`N`；
    2. 中间层，对于给定给的`Internal Node`的数目，由`2.1.2`补充性质可知，`BH`介于$\lceil log_4(N+1) \rceil$与$\lceil log_2(N+1) \rceil$之间。我们根据内点数将`BH`的取值区间缩小范围；
    3. 最内层，由于转移方程用到了子树不同内点和黑高的数据，我们希望约束`left_size`左子树/右子树的取值规模。考虑到左右子树的黑高被约束在给定的`BH`与`BH-1`之间，因此最少的内点数$2^{BH-1}-1$，最多为$4^{BH}-1$与总内点数`Internal node`-1的较小值，由此我们得到`left_size`的遍历区间：

```
int left_size = pow(2,BH-1)-1 ; left_size < pow(4,BH)-1 && left_size <
internal_nodes-1;
```

4. 最内层的转移方程部分，思路已在`2.2.1`中给出，此处需要强调的是：由于`red_root`部分的计算只需要用到`BH-1`时的`black_root`数据，而黑高为`BH`的`black_root`组在计算时需要用到`red_root`的黑高为`BH`的数据，因此需要先更新`red_root`部分，确保计算顺序正确；

> 上述并非最佳优化结果，为了方便表述如此显示，在之后的章节将作出补充。(比如位移运算代替函数计算)

Chapter 3: Testing Results

3.1 测试目的及结果

3.1.1 正确性测试

1. **样例输入：**

```
please input the number of internal nodes:
5
black_root_dp of BH:2 is 8

red_root_dp of BH:2 is 6

black_root_dp of BH:3 is 0

red_root_dp of BH:3 is 0


--Result--
The number of different RBT with 5 internal nodes mod by 1000000007 is 8
```

结果显示符合预期

2. **边界情况：**

1. 令`N`为1,2,3，结果分别为`1`,`1`,`2`，符合预期；
2. 取最大规模：

```
please input the number of internal nodes:
500
black_root_dp of BH:5 is 4286746496

red_root_dp of BH:5 is 222380928

black_root_dp of BH:6 is 1837930112

red_root_dp of BH:6 is 221427200

black_root_dp of BH:7 is 1892399232

red_root_dp of BH:7 is 3065791872

black_root_dp of BH:8 is 1225491712

red_root_dp of BH:8 is 4228986368

black_root_dp of BH:9 is 0

red_root_dp of BH:9 is 0


--Result--
The number of different RBT with 500 internal nodes mod by 1000000007 is 652632960
```

成功输出。

3.1.2 时间复杂度测试

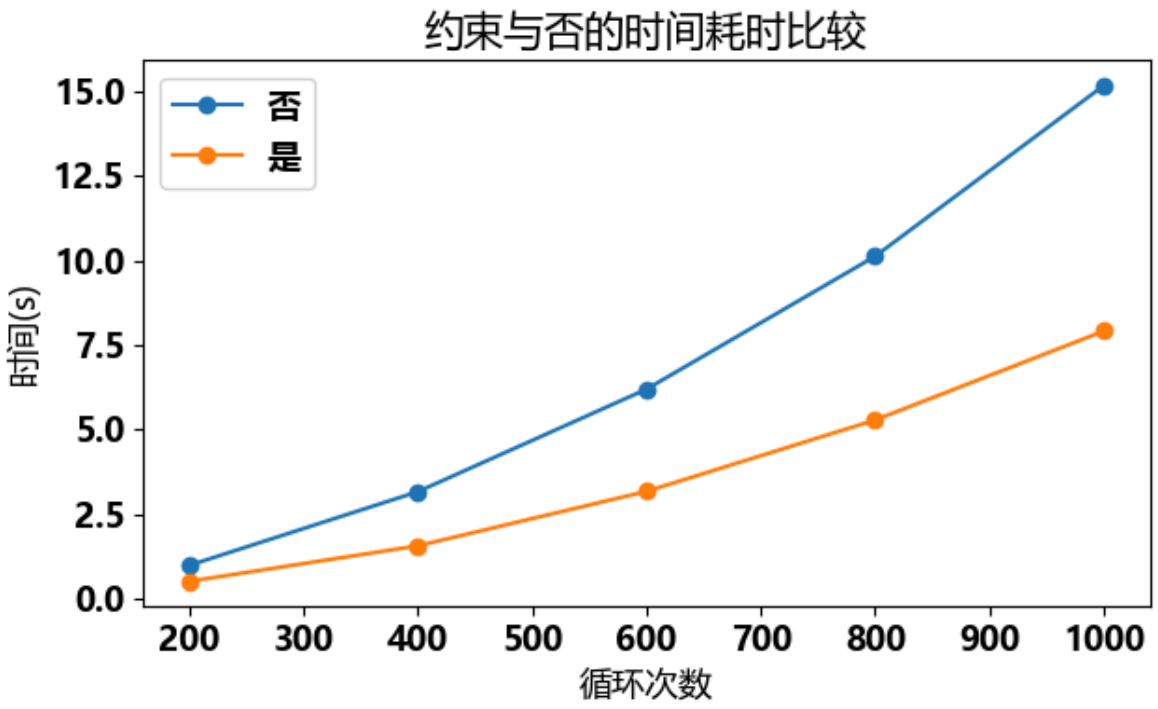
在`2.2.2`中我们分析了利用红黑树的性质并结合数学原理，可以得到`size`和`BH`之间更加精密的关系，从而缩小遍历区间。在此，我们通过运行条件**约束前后的代码**，得到不同的运行时间耗时。

具体的测试环境为：固定输入的`N`为`500`,遍历执行次数从`200`到`1000`（取等差数列，区间为`200`）：

- **Time Table:**

约束与否	否	是
200	0.98	0.51
400	3.16	1.56
600	6.19	3.17
800	10.12	5.28
1000	15.17	7.91

- **Plot:**



3.2 测试结果分析

1. 测试结果表明程序**通过了正确性测试**，且能够求解题目给出规模的上界；
2. 同时，在`3.1.2`中测试比较了引入`internal node`与`BH`分析前后的时间耗时，可见利用红黑树的特殊性质可以**有效约束循环**的求解范围，从而显著节约时间。

> 没有测试更大规模来探究具体的关系，这是由于程序中还有其他部分占了不可忽视的运行时间，比如数组的初始化、`math.h`中指数运算,`pow`运算的影响等。

在此给出约束前后的代码比较：

```
//约束后：
for (int internal_nodes = 3; internal_nodes <= N; internal_nodes++) {
            for (int BH = (int)ceil(log(internal_nodes + 1) / log(4)); 1 << BH -
1 <= internal_nodes; BH++)  {
```

```
                //compute the subproblems
             for (int left_size = pow(2, BH - 1) - 1; left_size < 1 << 2*BH - 1
&& left_size < internal_nodes - 1; left_size++) {
    ....

                }
            }
        }

//约束前：
       for (int internal_nodes = 3; internal_nodes <= N; internal_nodes++) {
             for (int BH = 1; pow(2, BH) - 1 <= internal_nodes; BH++) {
                //compute the subproblems
                 for (int left_size = 1; left_size < internal_nodes - 1;
left_size++) {
                ...
                }
            }
        }
```

Chapter 4: Analysis and Comments

4.1 时间复杂度分析

> 针对优化之后的版本进行分析

程序的核心部分是动态规划（DP）部分，其中嵌套的循环控制了程序的复杂度。逐步分析各个部分的复杂度：

- **外部循环（`internal_nodes`）**： 外部循环从 `internal_nodes = 3` 开始，到 `internal_nodes = N`，因此它的迭代次数为 O(N)。
- **第二层循环（`BH`）**： 第二层循环的范围是 `BH = (int)ceil(log(internal_nodes + 1) / log(4))` 到 `1 << BH - 1 <= internal_nodes`，这是一个关于 `internal_nodes` 的对数增长。由于 `log(4)` 是常数，因此该循环的迭代次数是 O(logN)；
- **第三层循环（`left_size`）**： 对于每一个 `internal_nodes` 和 `BH`，第三层循环遍历 `left_size`，其迭代次数范围是 `pow(2, BH - 1) - 1` 到 1<<2×BH −1 ，即 `left_size` 的范围大约是 2^{BH}。 而 `BH` 的最大值为 O(logN),因此该层循环的次数上界为O(N).

综上，DP的整体时间复杂度为：O$(N^2 \log N)$

4.2 空间复杂度分析

1. **动态规划数组 `black_root_dp` 和 `red_root_dp`**： 每个数组的大小是 `max_size`的平方，一般设置`max_size`为上界`N` = 500(+1)
2. 辅助参数为O(1),忽略不计；

综上，整体的空间复杂度为：O(N^2)

4.3 程序总体评价

4.3.1 正确性实现

通过`3.1.1`的测试情况，我们发现程序整体上实现了项目要求的功能，即根据给定的`internal node`的数目来确定红黑树的数量；

4.3.2 时间复杂度的改进

结合红黑树的基本性质`2.1.1`以及补充性质`2.1.2`，我们通过数学计算得到了内点数和`BH`之间内在的约束关系，从而限制了部分循环的区间，将时间复杂度从 O(N^3) 降低到了O$(N^2 \log N)$.

4.3.3 改进方向

- **空间复杂度**

该程序主要的问题在于空间复杂度较高。我们需要计算的数字比较大，因此开设的类型是`unsigned long int`，大小为`max_size`平方，且有两个这样的二维数组。

经过进一步分析，我们不难发现，在转移方程的计算过程当中，我们只需要两层的`BH`，分别是`BH-1`以及`BH`。这意味着我们可以用`BH_previous`以及`BH_current`来代替N维的数组计算。值得注意的是，这并不意味着我们可以用2维代替N维的`BH`数组，因为在最后计算`N`个内点的红黑树个数时，我们需要遍历所有可能的`BH`。结合之前的数学分析，我们可以知道，`BH`的取值范围大约为$\log_4(N+1)$到$\log_2(N+1)$，即长度约为$\log_2 N$. 这意味着我们可以用其来代替N维的空间。

综上分析，理论上可以优化空间复杂度为$O(N\log N)$，这可以成为之后的改进方向。

- **函数计算的时间成本**

在利用数学关系来约束循环区间时，我们采用了取对数和指数运算，在`math.h`中这伴随着较高的时间成本。

在上述的代码中，我们通过`1 << BH`的位移运算来代替`2^BH`的计算，节约了大量的时间，但是对数运算没有改进，后续可以考虑增加一个简单的函数来计算。

Appendix

Source Code

- `correctness_test`

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include <time.h>

#define max_size 501



int main() {
    int N; // number of the internal nodes;
    printf("please input the number of internal nodes:\n");
    scanf("%d", &N);

    if (N == 1 || N == 2) {
        printf("The number of different RBT with %d internal nodes is %d", N, N);
        return 0;
    }

    unsigned  long  int black_root_dp[max_size][max_size]; // dp[i][j] record RBT numbers when its internal nodes = i and black height = j
    unsigned  long  int red_root_dp[max_size][max_size]; //  same as the former, prepared for the former's calculation

    //initialize dp


    unsigned long int  RB_black_root, BR_black_root, BB_black_root, RR_black_root, count;

        for (int i = 0; i <= N; i++) {
        for (int j = 0; j <= N; j++) {
            black_root_dp[i][j] = 0;
            red_root_dp[i][j] = 0;
             }
         }
        black_root_dp[0][1] = 1; // Null leaf's BH = 1
            black_root_dp[1][1] = 1; // black root's BH = 1
            black_root_dp[2][1] = 2; //when it has two internal nodes and  black root 's BH = 1, there are two cases
            black_root_dp[2][2] = 0; // no situation for this case

            red_root_dp[0][1] = 1; // NULL leaf's BH = 1
            red_root_dp[1][1] = 1; // red root has its null leaf, so its BH = 1 and number of situaion = 1
            red_root_dp[2][1] = 0; // no consisent red nodes for RBT, so when BH = 1, number = 0
```

```
            red_root_dp[2][2] = 0; // same as the former

        //dynamical programming
        //BH : black height
        for (int internal_nodes = 3; internal_nodes <= N; internal_nodes++) {
            for (int BH = (int)ceil(log(internal_nodes + 1) / log(4)); 1 << BH -
1 <= internal_nodes; BH++)  {
                //compute the subproblems
                 for (int left_size = pow(2, BH - 1) - 1; left_size < 1 << 2*BH -
1 && left_size < internal_nodes - 1; left_size++) {
                    int right_size = internal_nodes - 1 - left_size;

                    red_root_dp[internal_nodes][BH] += black_root_dp[left_size]
[BH - 1] * black_root_dp[right_size][BH - 1];

                    //when the subtree's root is black, its BH is actually its
black_height +1 ,so we use BH-1 for calculating BH
                    RB_black_root = red_root_dp[left_size][BH] *
black_root_dp[right_size][BH - 1];
                    BR_black_root = black_root_dp[left_size][BH - 1] *
red_root_dp[right_size][BH];
                    BB_black_root = black_root_dp[left_size][BH - 1] *
black_root_dp[right_size][BH - 1];
                    RR_black_root = red_root_dp[left_size][BH] *
red_root_dp[right_size][BH];

                    black_root_dp[internal_nodes][BH] += RB_black_root +
BR_black_root + BB_black_root + RR_black_root;

                }
            }
        }

        //sum up all the RBT with N internal nodes
        count = 0;
        for (int i = (int)ceil(log(N + 1) / log(4)); 1 << i - 1 <= N; i++) {
            count += black_root_dp[N][i];

            printf("black_root_dp of BH:%d is %lu\n\n", i, black_root_dp[N][i]);
            printf("red_root_dp of BH:%d is %lu\n\n", i, red_root_dp[N][i]);
        }

        count = count % 1000000007;



    printf("\n--Result-- \nThe number of different RBT with %d internal nodes mod
by 1000000007 is %lu", N, count);

}
```

- `time_analyse`

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include <time.h>

#define max_size 501



int main() {
    int N; // number of the internal nodes;
    printf("please input the number of internal nodes:\n");
    scanf("%d", &N);

    if (N == 1 || N == 2) {
        printf("The number of different RBT with %d internal nodes is %d", N, N);
        return 0;
    }

    unsigned  long  int black_root_dp[max_size][max_size]; // dp[i][j] record RBT numbers when its internal nodes = i and black height = j
    unsigned  long  int red_root_dp[max_size][max_size]; //  same as the former, prepared for the former's calculation

    //initialize dp


    unsigned long int  RB_black_root, BR_black_root, BB_black_root, RR_black_root, count;
    int test_times ;
    int time_range[5] = {200,400,600,800,1000};

    clock_t start_time = clock();
    for (int i = 0; i < 5; i++) {
        test_times = time_range[i];
        for(int i = 0 ; i < test_times ;i ++){
        for (int i = 0; i <= N; i++) {
        for (int j = 0; j <= N; j++) {
            black_root_dp[i][j] = 0;
            red_root_dp[i][j] = 0;
             }
         }
        black_root_dp[0][1] = 1; // Null leaf's BH = 1
        black_root_dp[1][1] = 1; // black root's BH = 1
        black_root_dp[2][1] = 2; //when it has two internal nodes and  black root's BH = 1, there are two cases
        black_root_dp[2][2] = 0; // no situation for this case

        red_root_dp[0][1] = 1; // NULL leaf's BH = 1
        red_root_dp[1][1] = 1; // red root has its null leaf, so its BH = 1 and number of situaion = 1
```

```
        red_root_dp[2][1] = 0; // no consisent red nodes for RBT, so when BH = 1,
number = 0
        red_root_dp[2][2] = 0; // same as the former

        //dynamical programming
        //BH : black height
        for (int internal_nodes = 3; internal_nodes <= N; internal_nodes++) {
              // BH's range is calculated by RBT's property(please check out the
document for more information)
             for (int BH = 1; pow(2, BH) - 1 <= internal_nodes; BH++) {
                // compute the subproblems: add all situaition 's left_number *
right_number
                // Left_size's range is calculated by RBT's property(please check
document for more information)
                 for (int left_size = 1; left_size < internal_nodes - 1;
left_size++) {
                    int right_size = internal_nodes - 1 - left_size;

                     // for red root, sub_tree must be black_root, add them up
                    red_root_dp[internal_nodes][BH] += black_root_dp[left_size]
[BH - 1] * black_root_dp[right_size][BH - 1];

                   //when the subtree's root is black, its BH is actually its
black_height +1 ,so we use BH-1 for calculating BH
                    RB_black_root = red_root_dp[left_size][BH] *
black_root_dp[right_size][BH - 1]; // RB situation for black root case
                    BR_black_root = black_root_dp[left_size][BH - 1] *
red_root_dp[right_size][BH]; // BR situtation for black root case
                    BB_black_root = black_root_dp[left_size][BH - 1] *
black_root_dp[right_size][BH - 1]; // BB situation for black root case
                    RR_black_root = red_root_dp[left_size][BH] *
red_root_dp[right_size][BH]; // RR situation for black root case

                    //add up all situation numbers for black_root
                    black_root_dp[internal_nodes][BH] += RB_black_root +
BR_black_root + BB_black_root + RR_black_root;
                }
            }
        }

        //sum up all the RBT with N internal nodes
        count = 0;
        for (int i = 1; pow(i, 2) - 1 <= N; i++){
            count += black_root_dp[N][i];

            //printf("black_root_dp of BH:%d is %lu\n\n", i, black_root_dp[N]
[i]);
            //printf("red_root_dp of BH:%d is %lu\n\n", i, red_root_dp[N][i]);
        }

        count = count % 1000000007;
    }

    clock_t end_time = clock();// stop time record

   // calculate taken time
```

```
    double time_taken = (double)(end_time - start_time) / CLOCKS_PER_SEC;

    printf("Time taken for the computation of %d times: %f seconds\n",
test_times,time_taken);
    }

    printf("\n--Result-- \nThe number of different RBT with %d internal nodes mod
by 1000000007 is %lu", N, count);

}
```

- `improved_time_analyse`

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include <time.h>

#define max_size 501



int main() {
    int N; // number of the internal nodes;
    printf("please input the number of internal nodes:\n");
    scanf("%d", &N);

    //output the answer for the small size input
    if (N == 1 || N == 2) {
        printf("The number of different RBT with %d internal nodes is %d", N, N);
        return 0;
    }

    unsigned  long  int black_root_dp[max_size][max_size]; // dp[i][j] record RBT
numbers when its internal nodes = i and black height = j
    unsigned  long  int red_root_dp[max_size][max_size]; //  same as the former,
prepared for the former's calculation



    unsigned long int  RB_black_root, BR_black_root, BB_black_root,
RR_black_root, count;
    int test_times ;
    int time_range[5] = {200,400,600,800,1000};

    clock_t start_time = clock();
    for (int i = 0; i < 5; i++) {
        test_times = time_range[i];
        for(int i = 0 ; i < test_times ;i ++){
            //initialize dp
            for (int i = 0; i <= N; i++) {
            for (int j = 0; j <= N; j++) {
```

```
                black_root_dp[i][j] = 0;
                red_root_dp[i][j] = 0;
                }
            }
            black_root_dp[0][1] = 1; // Null leaf's BH = 1
            black_root_dp[1][1] = 1; // black root's BH = 1
            black_root_dp[2][1] = 2; //when it has two internal nodes and  black
root 's BH = 1, there are two cases
            black_root_dp[2][2] = 0; // no situation for this case

            red_root_dp[0][1] = 1; // NULL leaf's BH = 1
            red_root_dp[1][1] = 1; // red root has its null leaf, so its BH = 1
and number of situaion = 1
            red_root_dp[2][1] = 0; // no consisent red nodes for RBT, so when BH
= 1, number = 0
            red_root_dp[2][2] = 0; // same as the former

            //dynamical programming
            //BH : black height
            for (int internal_nodes = 3; internal_nodes <= N; internal_nodes++) {
                // BH's range is calculated by RBT's property(please check out
the document for more information)
                for (int BH = (int)ceil(log(internal_nodes + 1) / log(4)); 1 <<
BH - 1 <= internal_nodes; BH++)  {
                    // compute the subproblems: add all situaition 's left_number
* right_number
                    // Left_size's range is calculated by RBT's property(please
check document for more information)
                    for (int left_size = pow(2, BH - 1) - 1; left_size < 1 <<
2*BH - 1 && left_size < internal_nodes - 1; left_size++) {
                        int right_size = internal_nodes - 1 - left_size;

                        // for red root, sub_tree must be black_root, add them up
                        red_root_dp[internal_nodes][BH] +=
black_root_dp[left_size][BH - 1] * black_root_dp[right_size][BH - 1];

                        //when the subtree's root is black, its BH is actually
its black_height +1 ,so we use BH-1 for calculating BH
                        RB_black_root = red_root_dp[left_size][BH] *
black_root_dp[right_size][BH - 1]; // RB situation for black root case
                        BR_black_root = black_root_dp[left_size][BH - 1] *
red_root_dp[right_size][BH]; // BR situtation for black root case
                        BB_black_root = black_root_dp[left_size][BH - 1] *
black_root_dp[right_size][BH - 1]; // BB situation for black root case
                        RR_black_root = red_root_dp[left_size][BH] *
red_root_dp[right_size][BH]; // RR situation for black root case

                        //add up all situation numbers for black_root
                        black_root_dp[internal_nodes][BH] += RB_black_root +
BR_black_root + BB_black_root + RR_black_root;

                    }
                }
            }

            //sum up all the RBT with N internal nodes
```

```
            count = 0;
            // i's range is calculated by RBT's property(please check out the
document for more information)
            for (int i = (int)ceil(log(N + 1) / log(4)); 1 << i - 1 <= N; i++) {
                count += black_root_dp[N][i];

                //printf("black_root_dp of BH:%d is %lu\n\n", i, black_root_dp[N]
[i]);
                //printf("red_root_dp of BH:%d is %lu\n\n", i, red_root_dp[N]
[i]);
            }

        count = count % 1000000007;
    }

    clock_t end_time = clock(); // stop time record

    // calculate taken time
    double time_taken = (double)(end_time - start_time) / CLOCKS_PER_SEC;

    printf("Time taken for the computation of %d times: %f seconds\n",
test_times,time_taken);
    }

    printf("\n--Result-- \nThe number of different RBT with %d internal nodes mod
by 1000000007 is %lu", N, count);

}
```

References

> None.

Author List

`---`

Declaration

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