



# 算法基础 Lab 3

## 红黑树插入算法

张芷苒

PB21081601

October 25, 2023

### Part 1: 实验要求

编码实现红黑树的插入算法，使得插入后依旧保持红黑性质。(即：实现教材 p178 页的 RB-INSERT, RB\_INSERT\_FIXUP 算法)

### Part 2: 算法设计思路

#### 2.1 定义红黑树结点

定义两个结构体 ‘RBNode’ ‘RBTree’ 以实现红黑树的数据结构：

```
1 typedef struct node
2 {
3     Colour color;
4     int key;
5     struct node *left, *right, *p;
6 } RBNode, *pRBNode;
7
8 typedef struct nodeTree
9 {
10     pRBNode root, NIL;
11 } RBTree, *pRBTree;
```

#### 2.2 红黑树初始化

初始化树，初始化根结点、NIL 结点。

```
1 void RBTInit(pRBTree T)
2 {
3     if (T == NULL)
4     {
5         return;
```

```

6     }
7     T->NIL = new RBNode;
8     T->NIL->color = BLACK;
9     T->root = T->NIL;
10    T->root->p = T->NIL;
11    return;
12 }

```

让 NIL 结点的颜色为 black, 并让根结点指向 NIL 结点的父结点指向 NIL。

## 2.3 插入结点

找到待插入元素的位置, 并将其颜色改为 red, 左右结点置为 NIL, 再进行红黑性质的保持。

```

1 void RBInsert(pRBTTree T, pRBNode z)
2 {
3     pRBNode y = T->NIL;
4     pRBNode x = T->root;
5     while (x != T->NIL)
6     {
7         y = x;
8         if (z->key < x->key)
9         {
10            x = x->left;
11        }
12        else
13        {
14            x = x->right;
15        }
16    }
17    z->p = y; // 插入根结点时让根节点父节点指向NIL
18    if (y == T->NIL)
19    {
20        T->root = z;
21    }
22    else if (z->key < y->key)
23    {
24        y->left = z;
25    }
26    else
27    {
28        y->right = z;

```

```

29 }
30 z->left = T->NIL;
31 z->right = T->NIL;
32 z->color = RED;
33 RBInsertFixup(T, z);
34 return;
35 }

```

## 2.4 红黑树的性质保持

在 2.3 插入结点后, 可能引起性质 4 的冲突 (插入结点的父亲结点也是红色), 这时需要进行以下操作 (以插入结点的父结点是 = 其祖父结点的左孩子为例):

1. case1:  $z$  的叔叔结点为红色, 则将父结点与叔叔结点变黑, 然后令  $z$  指向  $z$  的祖父结点, 矛盾上移;
2. case2:  $z$  的叔叔结点为黑色, 且  $z$  是其父节点的右孩子, 则  $z$  指向其父节点, 并且左旋  $z$ , 进入 case3;
3. case3:  $z$  的叔叔结点为黑色, 且  $z$  是其父节点的左孩子, 则将  $z$  的父亲节点赋为黑色, 并且右旋  $z$  的祖父节点。
4. case4, 5, 6: 同上, 是在以插入结点的父结点是其祖父结点的右孩子的情况下, 将 case1, 2, 3 中情况左右调换即可。

```

1 void RBInsertFixup(pRBTTree T, pRBNode z)
2 {
3     pRBNode y;
4     while (z->p->color == RED)
5     {
6         if (z->p == z->p->p->left)
7         {
8             y = z->p->p->right;
9             if (y->color == RED)
10            {
11                z->p->color = BLACK; // case1
12                y->color = BLACK;    // case1
13                z->p->p->color = RED; // case1
14                z = z->p->p;          // case1
15                cout << 1;
16            }
17            else

```

```

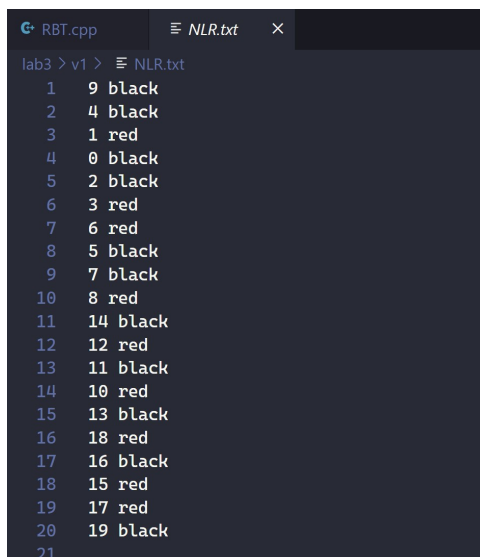
18     {
19         if (z == z->p->right)
20         {
21             z = z->p;           // case2
22             LeftRotate(T, z); // case2
23             cout << 2;
24         }
25         z->p->color = BLACK;    // case3
26         z->p->p->color = RED;   // case3
27         RightRotate(T, z->p->p); // case3
28         cout << 3;
29     }
30 }
31 else
32 {
33     y = z->p->p->left;
34     if (y->color == RED)
35     {
36         z->p->color = BLACK; // case4
37         y->color = BLACK;   // case4
38         z->p->p->color = RED; // case4
39         z = z->p->p;         // case4
40         cout << 4;
41     }
42     else
43     {
44         if (z == z->p->left)
45         {
46             z = z->p;           // case5
47             RightRotate(T, z); // case5
48             cout << 5;
49         }
50         z->p->color = BLACK;    // case6
51         z->p->p->color = RED;   // case6
52         LeftRotate(T, z->p->p); // case6
53         cout << 6;
54     }
55 }
56 }
57 T->root->color = BLACK;
58 return;
59 }

```

### Part 3: 实验结果分析

运行程序，可以得到以下结果：

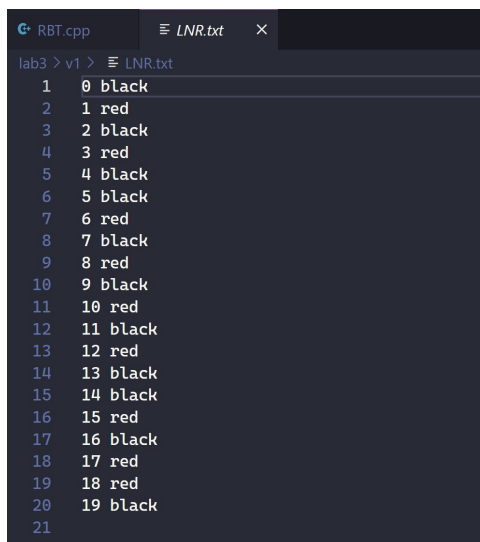
先序遍历结果：



```
lab3 > v1 > NLR.txt
1  9 black
2  4 black
3  1 red
4  0 black
5  2 black
6  3 red
7  6 red
8  5 black
9  7 black
10 8 red
11 14 black
12 12 red
13 11 black
14 10 red
15 13 black
16 18 red
17 16 black
18 15 red
19 17 red
20 19 black
21
```

图 1: 先序遍历结果

中序遍历结果：



```
lab3 > v1 > LNR.txt
1  0 black
2  1 red
3  2 black
4  3 red
5  4 black
6  5 black
7  6 red
8  7 black
9  8 red
10 9 black
11 10 red
12 11 black
13 12 red
14 13 black
15 14 black
16 15 red
17 16 black
18 17 red
19 18 red
20 19 black
21
```

图 2: 中序遍历结果

层次遍历结果：

```

RBT.cpp  LOT.txt
lab3 > v1 > LOT.txt
1  9 black
2  4 black
3  14 black
4  1 red
5  6 red
6  12 red
7  18 red
8  0 black
9  2 black
10 5 black
11 7 black
12 11 black
13 13 black
14 16 black
15 19 black
16 3 red
17 8 red
18 10 red
19 15 red
20 17 red
21

```

图 3: 后序遍历结果

实际结果:

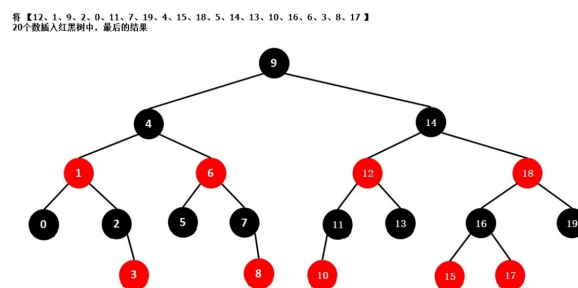


图 4: 实际结果

经过对比，以上结果均符合预期，由此可以认为算法正确。

## Part 4 实验总结

本次实验完成了快速排序算法的实现及其优化红黑树的插入算法。在实验过程中获得了以下收获：

- 完善了对红黑树的理解
- 加强了对数据结构实验的理解