

# 红黑树插入算法实验报告

SA20225085 朱志儒

## 实验内容

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

节点属性：

```
TNode = {  
    Color: red / black,  
    Key: int,  
    Left: TNode*,  
    Right: TNode*,  
    P: TNode*  
}
```

程序输入：

文件名： insert.txt

文件格式： 第一行为待插入数据的个数，第二行为待插入的数据（int 类型， 空格分割）

注： 1) 初始时红黑树应为空。

2) 按顺序插入， 例如，对于下图的数据，插入顺序应为 20, 10, 14



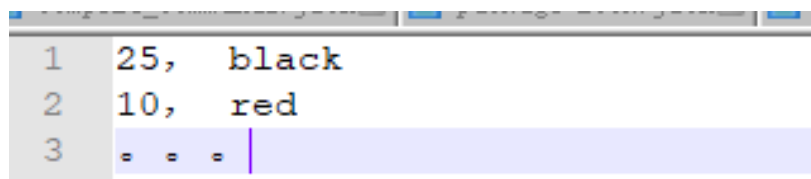
1	3
2	20 10 14

程序输出： 将插入完成后的红黑树进行“先序遍历（NLR）”和“中序遍历（LNR）”并将相应的遍历序列输出到文件中。

文件名： LNR.txt 中序遍历序列结果

NLR.txt 先序遍历序列结果

格式： 每一行对应一个节点的信息（key, color）



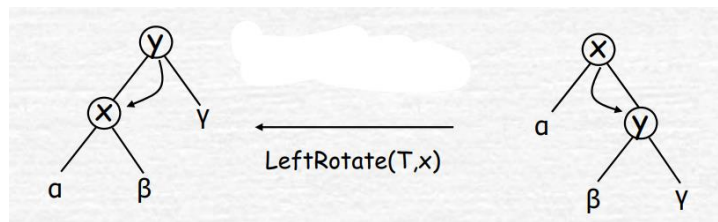
1	25, black
2	10, red
3	- - -

## 实验目的

1. 进一步熟悉 C/C++ 语言的集成开发环境
2. 通过本实验加深对红黑树插入的理解和运用

## 算法设计思路

### 1. 左旋算法

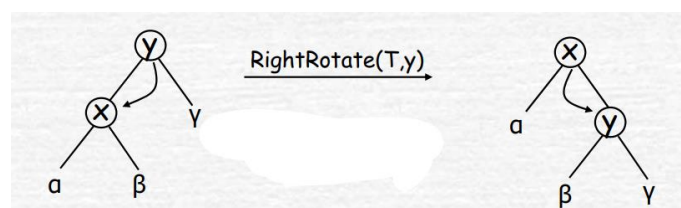


伪代码:

rotateLeft(T, x):

```
y = right[x];          //记录指向 y 节点的指针
right[x] = left[y];     //y 的左子节点连接到 x 的右
p[left[y]] = x;
p[y] = p[x];           //y 连接到 x 的父节点
if p[x] == nil[T] then  //x 是根节点
    root[T] = y;        //修改树指针
else if x = left[p[x]] then
    left[p[x]] = y;     //x 父节点左连接 y
else
    right[p[x]] = y;    //x 父节点右连接 y
left[y] = x;           //x 连接到 y 左
p[x] = y;
```

### 2. 右旋算法



伪代码:

rotateRight(T, y):

```
x = left[y];           //记录指向 x 节点的指针
left[y] = right[x];    //x 的右子节点连接到 y 的左
p[right[x]] = y;
p[x] = p[y];           //x 连接到 y 的父节点
if p[y] == nil then    //y 是根节点
    root[T] = x;       //修改树指针
else if y == left[p[y]] then
    left[p[y]] = x;    //y 父节点左连接 x
else
    right[p[y]] = x;   //y 父节点右连接 x
right[x] = y;          //y 连接到 x 右
p[y] = x;
```

### 3. 插入算法

伪代码:

```
y = nil[T];           //y 用于记录: 当前扫描节点的双亲节点
x = root[T];          //从根开始扫描
while x != nil[T] do  //查找插入位置
    y = x;
    if key[z] < key[x] then
        x = left[x];  //z 插入 x 的左边
    else
        x = right[x]; //z 插入 x 的右边
p[z] = y;              //y 是 z 的双亲
if y == nil[T] then    //z 插入空树
    root[T] = z;       //z 是根
else if key[z] < key[y] then
    left[y] = z;       //z 是 y 的左子插入
else
```

```
right[y] = z;    //z 是 y 的右子插入
left[z] = right[z] = nil[T];
color[z] = red;
RBInsertFixup(T, z);
```

#### 4. 调整算法

case1~3 为  $z$  的双亲  $p[z]$  是其祖父  $p[p[z]]$  的左孩子。

case1:  $z$  的叔叔  $y$  是红色, 则将  $p[z]$  和  $y$  变黑,  $p[p[z]]$  变红,  $z = p[p[z]]$ ,  $z$  上溯, 若红色传播到根, 将根涂黑, 则树的黑高增 1,  $z$  是  $p[z]$  的左、右孩子均一样处理。

case2:  $z$  的叔叔  $y$  是黑色, 且  $z$  是双亲  $p[z]$  的右孩子, 则  $z = p[z]$ , 左旋  $z$ , 变成 case3。

case3:  $z$  的叔叔  $y$  是黑色, 且  $z$  是双亲  $p[z]$  的左孩子, 则  $p[z]$  和  $p[p[z]]$  变色, 右旋  $p[p[z]]$ 。

case4~6 为  $z$  的双亲  $p[z]$  是其祖父  $p[p[z]]$  的右孩子。

case4:  $z$  的叔叔  $y$  是红色, 则将  $p[z]$  和  $y$  变黑,  $p[p[z]]$  变红,  $z = p[p[z]]$ ,  $z$  上溯, 若红色传播到根, 将根涂黑, 则树的黑高增 1,  $z$  是  $p[z]$  的左、右孩子均一样处理。

case5:  $z$  的叔叔  $y$  是黑色, 且  $z$  是双亲  $p[z]$  的左孩子, 则  $z = p[z]$ , 右旋  $z$ , 变成 case6。

case6:  $z$  的叔叔  $y$  是黑色, 且  $z$  是双亲  $p[z]$  的右孩子, 则  $p[z]$  和  $p[p[z]]$  变色, 左旋  $p[p[z]]$ 。

## 源码及注释

```
1.  enum Color { RED, BLACK };
2.
3.  struct TNode {
4.      Color color;
5.      int key;
6.      TNode* left;
7.      TNode* right;
8.      TNode* parent;
9.      TNode(int k){
10.         color = BLACK;
11.         key = k;
12.         left = right = parent = 0;
13.     }
14. };
15.
16. class redBlackTree {
17. public:
18.     TNode* root;
19.     TNode* nil;
20.     void rotateLeft(TNode*);
21.     void rotateRight(TNode*);
22.     void insertFixup(TNode*);
23.     redBlackTree();
24.     void insert(TNode*);
25.     void LNR(ofstream&, TNode*);
26.     void NLR(ofstream&, TNode*);
27. };
28.
29. redBlackTree::redBlackTree() {
30.     nil = new TNode(-1);
31.     root = nil;
32. }
33.
34. void redBlackTree::rotateLeft(TNode* x) {
35.     TNode* y = x->right; //记录指向 y 节点的指针
36.     x->right = y->left;    //y 的左子节点连接到 x 的右
37.     y->left->parent = x;
38.     y->parent = x->parent; //y 连接到 x 的父节点
39.     if (x->parent == nil) { //x 是根节点
40.         root = y;          //修改树指针
41.     }
42.     else if (x == x->parent->left) {
```

```

43.         x->parent->left = y; //x 父节点左连接 y
44.     }
45.     else {
46.         x->parent->right = y; //x 父节点右连接 y
47.     }
48.     y->left = x; //x 连接到 y 左
49.     x->parent = y;
50. }
51.
52. void redBlackTree::rotateRight(TNode* y) {
53.     TNode* x = y->left;        //记录指向 x 节点的指针
54.     y->left = x->right;         //x 的右子节点连接到 y 的左
55.     x->right->parent = y;
56.     x->parent = y->parent;     //x 连接到 y 的父节点
57.     if (y->parent == nil) {    //y 是根节点
58.         root = x;             //修改树指针
59.     }
60.     else if (y == y->parent->left) {
61.         y->parent->left = x; //y 父节点左连接 x
62.     }
63.     else {
64.         y->parent->right = x; //y 父节点右连接 x
65.     }
66.     x->right = y;             //y 连接到 x 右
67.     y->parent = x;
68. }
69.
70. void redBlackTree::insert(TNode* z) {
71.     TNode* y = nil, * x = root; //y 用于记录: 当前扫描节点的双亲节点
72.     while (x != nil) {           //查找插入位置
73.         y = x;
74.         if (z->key < x->key)      //z 插入 x 的左边
75.             x = x->left;
76.         else
77.             x = x->right;         //z 插入 x 的右边
78.     }
79.     z->parent = y;               //y 是 z 的双亲
80.     if (y == nil)               //z 插入空树
81.         root = z;              //z 是根
82.     else if (z->key < y->key)
83.         y->left = z;             //z 是 y 的左子插入
84.     else
85.         y->right = z;            //z 是 y 的右子插入
86.     z->left = z->right = nil;

```

```

87.     z->color = RED;
88.     insertFixup(z);
89. }
90.
91. void redBlackTree::LNR(ofstream& outfile, TNode* p) {
92.     if (p == nil)
93.         return;
94.     LNR(outfile, p->left);      //中序遍历左子树
95.     outfile << p->key << ", " << (p->color == RED ? "red" : "black") << endl
96.         ;
97.     LNR(outfile, p->right);     //中序遍历右子树
98. }
99. void redBlackTree::NLR(ofstream& outfile, TNode* p) {
100.    if (p == nil)
101.        return;
102.    outfile << p->key << ", " << (p->color == RED ? "red" : "black") << endl
103.        ;
104.    NLR(outfile, p->left);       //先序遍历左子树
105.    NLR(outfile, p->right);      //先序遍历右子树
106. }
107. void redBlackTree::insertFixup(TNode* z) {
108.    while (z->parent->color == RED) {
109.        //若 z 为根, 则 p[z]==nil[T], 其颜色为黑, 不进入此循环
110.        //若 p[z]为黑, 无需调整, 不进入此循环
111.        if (z->parent == z->parent->parent->left) { //z 的双亲 p[z]是其祖父
112.            //p[p[z]]的左孩子
113.            TNode* y = z->parent->parent->right; //y 是 z 的叔叔
114.            if (y->color == RED) { //z 的叔叔 y 是红色
115.                y->color = BLACK;
116.                z->parent->color = BLACK;
117.                z->parent->parent->color = RED;
118.                z = z->parent->parent;
119.            }
120.            else { //z 的叔叔 y 是黑色
121.                if (z == z->parent->right) { //z 是双亲 p[z]的右孩子
122.                    z = z->parent;
123.                    rotateLeft(z); //左旋
124.                }
125.                //z 是双亲 p[z]的左孩子
126.                z->parent->color = BLACK;
127.                z->parent->parent->color = RED;
128.                rotateRight(z->parent->parent); //右旋

```

```

128.         }
129.     }
130.     else { //z 的双亲 p[z]是其祖父 p[p[z]]的右孩子
131.         TNode* y = z->parent->parent->left; //y 是 z 的叔叔
132.         if (y->color == RED) { //z 的叔叔 y 是红色
133.             y->color = BLACK;
134.             z->parent->color = BLACK;
135.             z->parent->parent->color = RED;
136.             z = z->parent->parent;
137.         }
138.         else { //z 的叔叔 y 是黑色
139.             if (z == z->parent->left) { //z 是双亲 p[z]的左孩子
140.                 z = z->parent;
141.                 rotateRight(z); //右旋
142.             }
143.             //z 是双亲 p[z]的右孩子
144.             z->parent->color = BLACK;
145.             z->parent->parent->color = RED;
146.             rotateLeft(z->parent->parent); //左旋
147.         }
148.     }
149. }
150. root->color = BLACK;
151. }
152.
153. int main() {
154.     ifstream infile("insert.txt");
155.     ofstream outfile_NLR("NLR.txt");
156.     ofstream outfile_LNR("LNR.txt");
157.     redBlackTree rbtree;
158.     int n, key;
159.     infile >> n;
160.     for (int i = 0; i < n; ++i) {
161.         infile >> key;
162.         TNode* node = new TNode(key);
163.         rbtree.insert(node);
164.     }
165.     rbtree.LNR(outfile_LNR, rbtree.root);
166.     rbtree.NLR(outfile_NLR, rbtree.root);
167.     return 0;
168. }

```

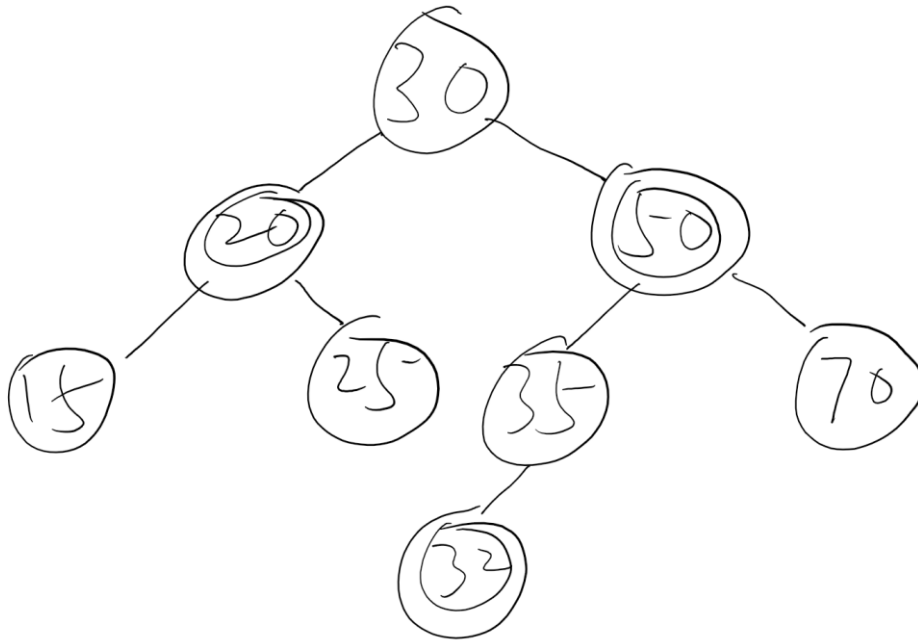


## 算法正确性测试

输入 (insert.txt):

```
1 8
2 50 20 70 15 30 25 35 32
```

根据输入得到的红黑树（一环表示黑色，双环表示红色）:



中序遍历序列结果 (LNR.txt):

```
1 15, black
2 20, red
3 25, black
4 30, black
5 32, red
6 35, black
7 50, red
8 70, black
```

先序遍历序列结果 (NLR.txt):

```
1 30, black
2 20, red
3 15, black
4 25, black
5 50, red
6 35, black
7 32, red
8 70, black
```

显然整个算法是正确的。

## 实验过程中遇到的困难及收获

通过本次实验，我学习了左旋和右旋的算法，了解了红黑树的插入算法以及调整算法，并将这些算法实现，对这些算法有更加深刻的认识。

本次实验的 RB-INSERT-FIXUP 算法中的旋转有点类似于 AVL 树的旋转，不过，两者区别还是比较大的，AVL 树要求左右子树的高度差小于等于 1，而红黑树要求每个节点到其后代叶子的左右路径含有同样多的黑节点。显然 AVL 树的条件比红黑树要更加严格。