**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 / blcak，

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

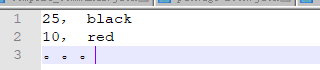


程序输出： 将插入完成后的红黑树进行 “先序遍历（NLR）” 和 “中序遍历（LNR）” 并将相应的遍历序列输出到文件中。

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）

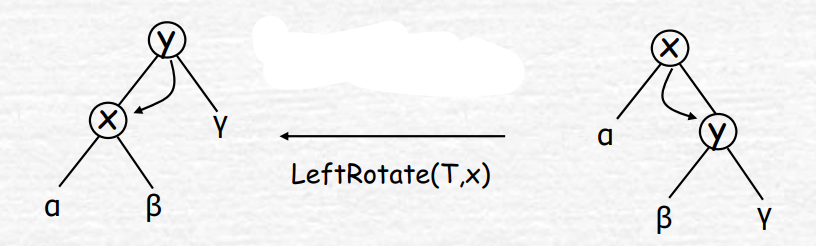


## 实验目的

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[x]; //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

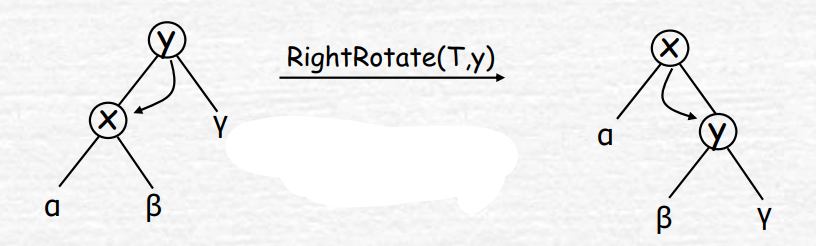
else

right[p[x]] = y; //x父节点右连接y

left[y] = x; //x连接到y左

p[x] = y;

1. **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;

1. **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);

1. **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。

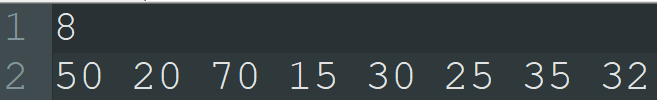
case6：z的叔叔y是黑色，且z是双亲p[z]的右孩子，则p[z]和p[p[z]]变色，左旋p[p[z]]。

## 源码及注释

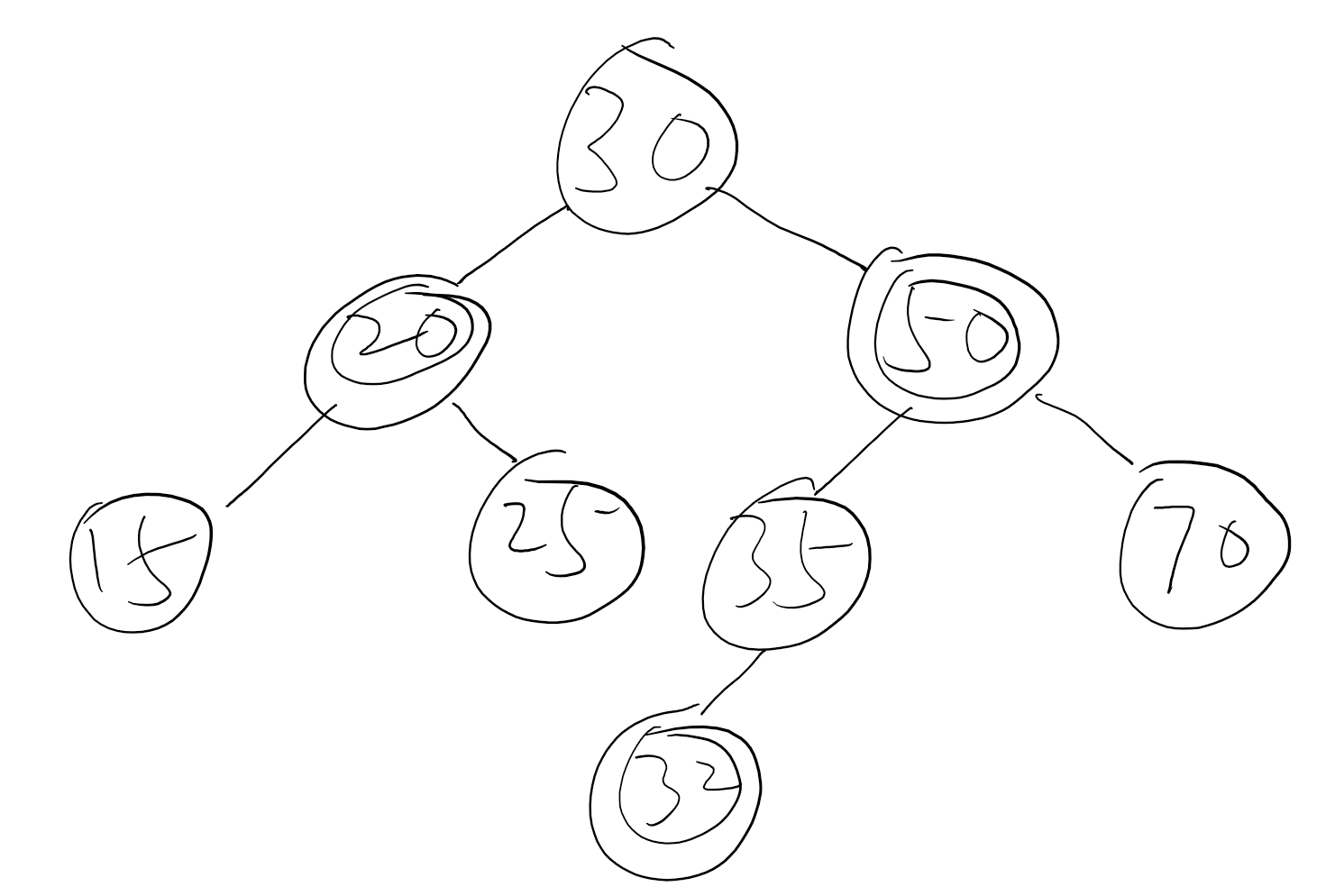
1. **enum** Color { RED, BLACK };
3. **struct** TNode {
4. Color color;
5. **int** key;
6. TNode\* left;
7. TNode\* right;
8. TNode\* parent;
9. TNode(**int** k){
10. color = BLACK;
11. key = k;
12. left = right = parent = 0;
13. }
14. };
16. **class** redBlackTree {
17. **public**:
18. TNode\* root;
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. };
29. redBlackTree::redBlackTree() {
30. nil = **new** TNode(-1);
31. root = nil;
32. }
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. }
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. }
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. }
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. LNR(outfile, p->right);      //中学遍历右子树
97. }
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. NLR(outfile, p->left);       //先序遍历左子树
104. NLR(outfile, p->right);      //先序遍历右子树
105. }
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p[p[z]]的左孩子
112. TNode\* y = z->parent->parent->right;   //y是z的叔叔
113. **if** (y->color == RED) {   //z的叔叔y是红色
114. y->color = BLACK;
115. z->parent->color = BLACK;
116. z->parent->parent->color = RED;
117. z = z->parent->parent;
118. }
119. **else** {  //z的叔叔y是黑色
120. **if** (z == z->parent->right) {  //z是双亲p[z]的右孩子
121. z = z->parent;
122. rotateLeft(z);  //左旋
123. }
124. //z是双亲p[z]的左孩子
125. z->parent->color = BLACK;
126. z->parent->parent->color = RED;
127. 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. }
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

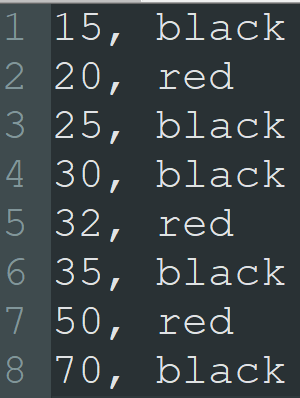
输入（insert.txt）：



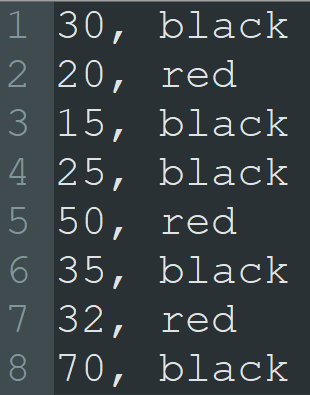
根据输入得到的红黑树（一环表示黑色，双环表示红色）：



中序遍历序列结果（LNR.txt）：



先序遍历序列结果（NLR.txt）：



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。