**区间树上重叠区间的查找算法**

SA20225085 朱志儒

## 实验内容

对红黑树进行修改，使其成为一颗区间数，并实现区间树上的重叠区间查找算法。

程序输入：

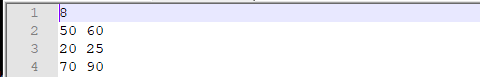
1、生成区间树

文件名： insert.txt

文件格式：第一行为待插入数据的个数，第二行之后每一行表示一个区间

注：1）初始时树应为空。

2）按顺序插入， 例如，对于下图的数据，插入顺序应为 [50, 60]，[20, 25]， [70, 90]



2、待查询区间应由控制台输入

程序输出：

控制台直接打印查找结果。

## 实验目的

1. 进一步熟悉C/C++语言和集成开发环境
2. 通过本实验加深对OS树和区间树的理解和运用

## 区间树的数据结构

1. **基本结构**

以红黑树为基础，对所有属于T的x，x包含区间int[x]的信息（低点和高点），key = low[int[x]]。

1. **附加信息**

红黑树中节点的附加信息是左子树高点、右子树高点和本节点高点三者的最大值，即

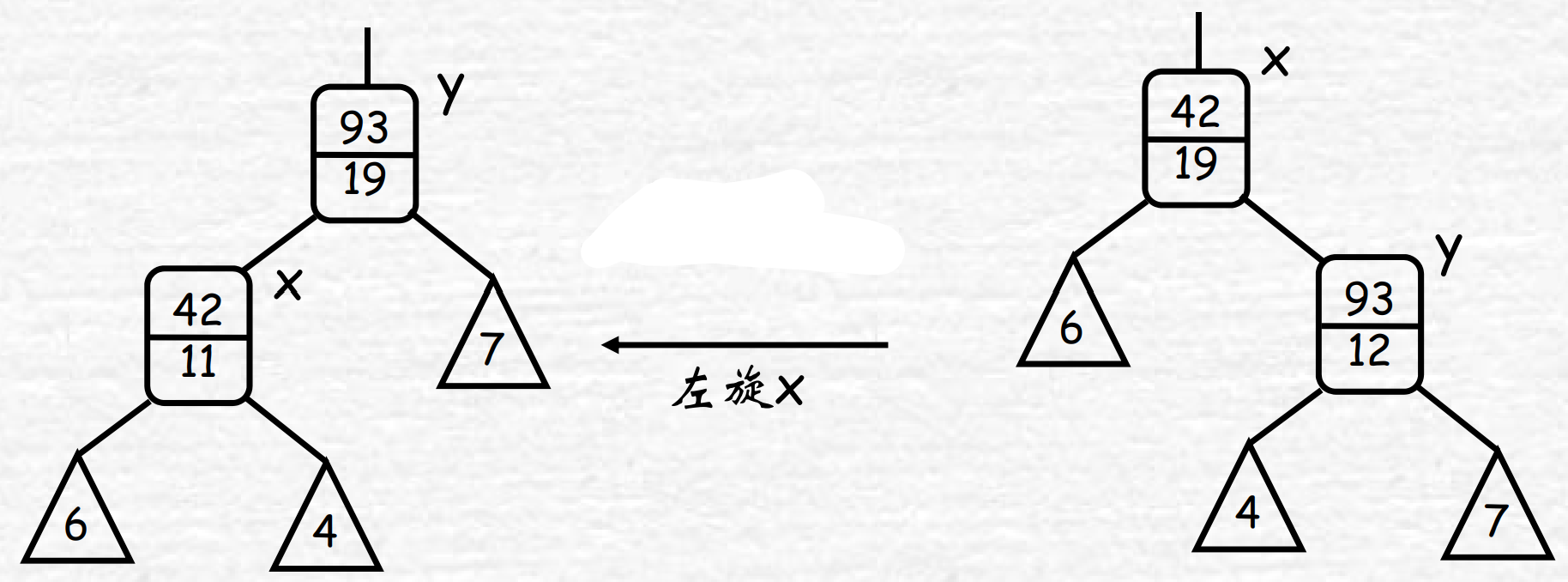
max[x] = max(high[int[x]], max[left[x]], max[right[x]]

1. **开发新操作**
2. **插入操作**

第一步，从根向下插入新节点，将搜索路径上所经历的每个节点的附加信息high取该节点的high与新节点的high中最大值。插入后，新节点的high取左子树高点、右子树高点和本节点高点三者的最大值。

第二步，采用变色和旋转方法，从叶子向上调整；变色不改变high；旋转可能改变high，因为旋转是局部操作，只有轴上的两个节点的high可能违反定义，所以，只需要在旋转操作后，对违反节点high进行修改。

其中，左旋操作：

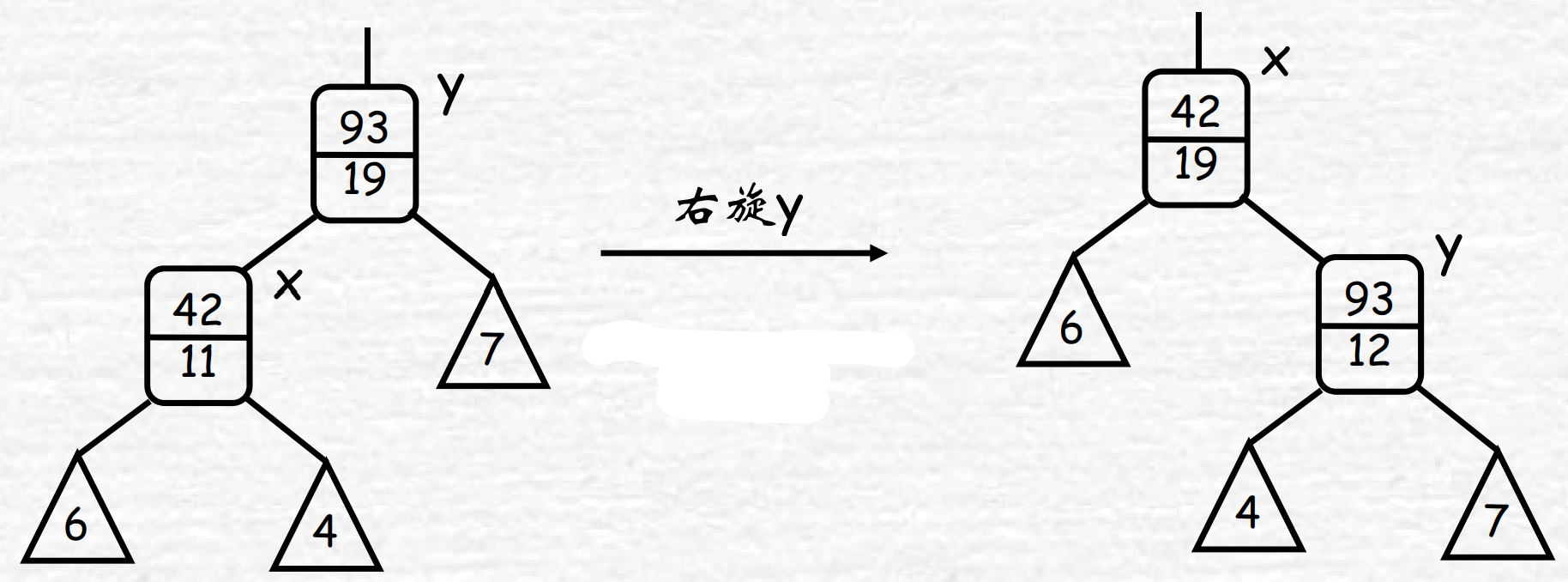


对附加信息high的修改：

high[y] = high[x];

high[x] = max(high[left[x]], high[right[x]]);

右旋操作：



对附加信息high的修改：

high[x] = high[y];

high[y] = max(high[left[y]], high[right[y]]);

1. **查找与给定区间重叠的区间**

基本思想：

x = root[T];

if x != nil[T] and i与int[x]不重叠

if x的左子树非空且左子树中最大高点>=low[i]

x = left[x];

else

x = right[x];

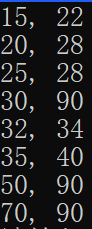
return x;

## 源码+注释

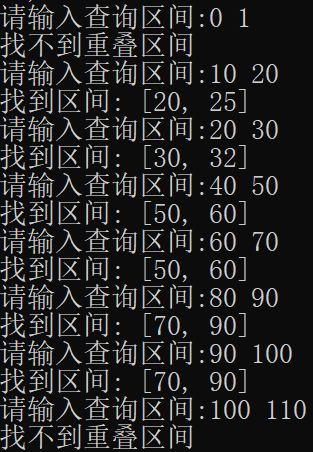
1. **enum** Color { RED, BLACK };
3. **struct** Interval {
4. **int** left, right;
5. Interval(**int** l, **int** r) :left(l), right(r) {}
6. };
8. **struct** TNode {
9. Color color;
10. Interval interval;
11. **int** key;
12. **int** high;
13. TNode\* left;
14. TNode\* right;
15. TNode\* parent;
16. TNode(**int** k, **int** h, Interval i):interval(i){
17. color = BLACK;
18. key = k;
19. high = h;
20. left = right = parent = 0;
21. }
22. };
24. **int** tripleMax(**int** x1, **int** x2, **int** x3=0) {
25. **return** max(x1, max(x2, x3));
26. }
28. **class** OSTree {
29. TNode\* root;
30. TNode\* nil;
31. **void** rotateLeft(TNode\*);
32. **void** rotateRight(TNode\*);
33. **void** insertFixup(TNode\*);
34. **public**:
35. OSTree();
36. TNode\* getRoot() { **return** root; }
37. **void** LNR(ostream&, TNode\*);
38. **void** insert(**int**, **int**);
39. Interval IntervalSearch(Interval);
40. };
42. OSTree::OSTree() {
43. nil = **new** TNode(-1, -1, Interval(-1, -1));
44. root = nil;
45. }
47. **void** OSTree::LNR(ostream& out, TNode\* p) {
48. **if** (p == nil)
49. **return**;
50. LNR(out, p->left);       //中序遍历左子树
51. out << p->key << ", " << p->high << endl;
52. LNR(out, p->right);      //中学遍历右子树
53. }
55. **void** OSTree::rotateLeft(TNode\* x) {
56. TNode\* y = x->right; //记录指向y节点的指针
57. x->right = y->left;       //y的左子节点连接到x的右
58. y->left->parent = x;
59. y->parent = x->parent;    //y连接到x的父节点
60. **if** (x->parent == nil) {  //x是根节点
61. root = y;           //修改树指针
62. }
63. **else** **if** (x == x->parent->left) {
64. x->parent->left = y;//x父节点左连接y
65. }
66. **else** {
67. x->parent->right = y;//x父节点右连接y
68. }
69. y->left = x; //x连接到y左
70. x->parent = y;
71. y->high = x->high;
72. x->high = tripleMax(x->left->high, x->right->high);
73. }
75. **void** OSTree::rotateRight(TNode\* y) {
76. TNode\* x = y->left;      //记录指向x节点的指针
77. y->left = x->right;       //x的右子节点连接到y的左
78. x->right->parent = y;
79. x->parent = y->parent;    //x连接到y的父节点
80. **if** (y->parent == nil) {  //y是根节点
81. root = x;           //修改树指针
82. }
83. **else** **if** (y == y->parent->left) {
84. y->parent->left = x;//y父节点左连接x
85. }
86. **else** {
87. y->parent->right = x;//y父节点右连接x
88. }
89. x->right = y;    //y连接到x右
90. y->parent = x;
91. x->high = y->high;
92. y->high = tripleMax(y->left->high, y->right->high);
93. }
95. **void** OSTree::insert(**int** left, **int** right) {
96. TNode\* z = **new** TNode(left, right, Interval(left, right));
97. TNode\* y = nil, \* x = root; //y用于记录：当前扫描节点的双亲节点
98. **while** (x != nil) {          //查找插入位置
99. y = x;
100. x->high = tripleMax(x->high, z->high);
101. **if** (z->key < x->key)   //z插入x的左边
102. x = x->left;
103. **else**
104. x = x->right;        //z插入x的右边
105. }
106. z->parent = y;               //y是z的双亲
107. **if** (y == nil)               //z插入空树
108. root = z;               //z是根
109. **else** **if** (z->key < y->key)
110. y->left = z;         //z是y的左子插入
111. **else**
112. y->right = z;            //z是y的右子插入
113. z->left = z->right = nil;
114. z->color = RED;
115. z->high = tripleMax(z->left->high, z->right->high, z->high);
116. insertFixup(z);
117. }
119. Interval OSTree::IntervalSearch(Interval x) {
120. TNode\* p = root;
121. **while** (p != nil && (x.left > p->interval.right || x.right < p->interval.left)) {
122. **if** (p->left != nil && p->left->high >= x.left)
123. p = p->left;
124. **else**
125. p = p->right;
126. }
127. **return** p->interval;
128. }
130. **void** OSTree::insertFixup(TNode\* z) {
131. **while** (z->parent->color == RED) {
132. //若z为根，则p[z]==nil[T]，其颜色为黑,不进入此循环
133. //若p[z]为黑，无需调整，不进入此循环
134. **if** (z->parent == z->parent->parent->left) { //z的双亲p[z]是其祖父p[p[z]]的左孩子
135. TNode\* y = z->parent->parent->right;   //y是z的叔叔
136. **if** (y->color == RED) {   //z的叔叔y是红色
137. y->color = BLACK;
138. z->parent->color = BLACK;
139. z->parent->parent->color = RED;
140. z = z->parent->parent;
141. }
142. **else** {  //z的叔叔y是黑色
143. **if** (z == z->parent->right) {  //z是双亲p[z]的右孩子
144. z = z->parent;
145. rotateLeft(z);  //左旋
146. }
147. //z是双亲p[z]的左孩子
148. z->parent->color = BLACK;
149. z->parent->parent->color = RED;
150. rotateRight(z->parent->parent);   //右旋
151. }
152. }
153. **else** {  //z的双亲p[z]是其祖父p[p[z]]的右孩子
154. TNode\* y = z->parent->parent->left;    //y是z的叔叔
155. **if** (y->color == RED) {   //z的叔叔y是红色
156. y->color = BLACK;
157. z->parent->color = BLACK;
158. z->parent->parent->color = RED;
159. z = z->parent->parent;
160. }
161. **else** {  //z的叔叔y是黑色
162. **if** (z == z->parent->left) {       //z是双亲p[z]的左孩子
163. z = z->parent;
164. rotateRight(z); //右旋
165. }
166. //z是双亲p[z]的右孩子
167. z->parent->color = BLACK;
168. z->parent->parent->color = RED;
169. rotateLeft(z->parent->parent);    //左旋
170. }
171. }
172. }
173. root->color = BLACK;
174. }
176. **int** main() {
177. ifstream infile("insert.txt");
178. OSTree ostree;
179. **int** n, left, right;
180. infile >> n;
181. **for** (**int** i = 0; i < n; ++i) {
182. infile >> left >> right;
183. ostree.insert(left, right);
184. }
185. cout << "OSTree的中序遍历：\n";
186. ostree.LNR(cout, ostree.getRoot());
187. **while** (**true**) {
188. cout << "请输入查询区间:";
189. cin >> left >> right;
190. Interval temp = ostree.IntervalSearch(Interval(left, right));
191. **if** (temp.left == -1 && temp.right == -1)
192. cout << "找不到重叠区间\n";
193. **else**
194. cout << "找到区间: [" << temp.left << ", " << temp.right << "]\n";
195. }
196. **return** 0;  }

## 算法测试结果

对生成的区间树中序遍历：



在控制台输入待查询区间：



## 实验过程中遇到的困难及收获

在本次实验中，为了方便判断i与int[x]两个区间是否重叠，我在红黑树的节点中添加了整个区间的信息，即起点low和终点high。这样每次新插入的节点代表的是输入中的一个区间，判断i与int[x]两个区间是否重叠时只需访问节点中的区间信息即可而不需要访问其他数据结构。