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]

```
1  8
2  50 60
3  20 25
4  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]]。

2. 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]]

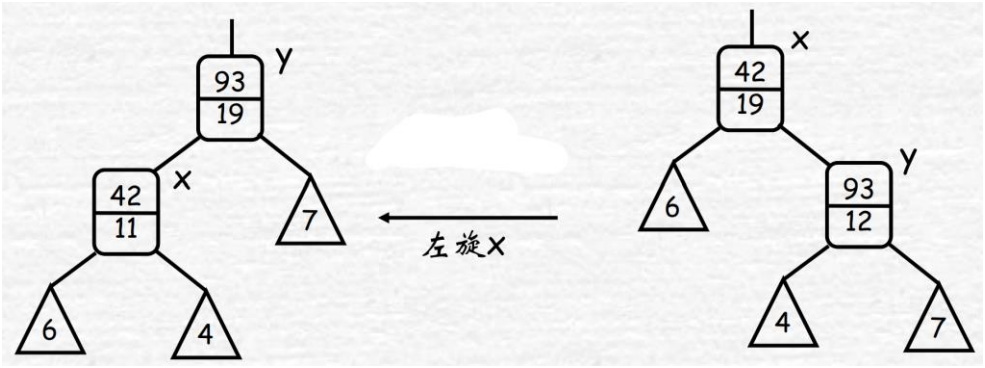
3. 开发新操作

(1) 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：

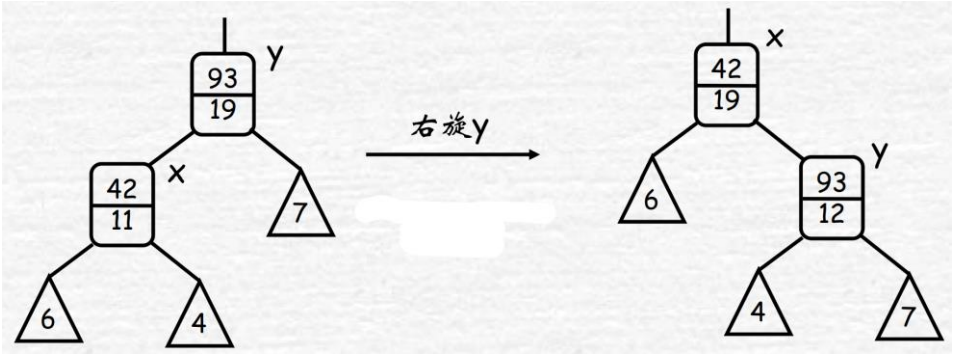


对附加信息 high 的修改：

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]]);

(2) 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

```
enum Color { RED, BLACK };

struct Interval {
    int left, right;
    Interval(int l, int r) :left(l), right(r) {}
};

struct TNode {
    Color color;
    Interval interval;
    int key;
    int high;
    TNode* left;
    TNode* right;
    TNode* parent;
    TNode(int k, int h, Interval i):interval(i){
        color = BLACK;
        key = k;
        high = h;
        left = right = parent = 0;
    }
};

```

```
int tripleMax(int x1, int x2, int x3=0) {
    return max(x1, max(x2, x3));
}

class OSTree {
    TNode* root;
    TNode* nil;
    void rotateLeft(TNode*);
    void rotateRight(TNode*);
    void insertFixup(TNode*);
public:
    OSTree();
    TNode* getRoot() { return root; }
    void LNR(ostream&, TNode*);
    void insert(int, int);
    Interval IntervalSearch(Interval);
};

OSTree::OSTree() {
    nil = new TNode(-1, -1, Interval(-1, -1));
    root = nil;
}

void OSTree::LNR(ostream& out, TNode* p) {
    if (p == nil)
        return;
    LNR(out, p->left);       //中序遍历左子树
    out << p->key << ", " << p->high << endl;
    LNR(out, p->right);      //中学遍历右子树
}

void OSTree::rotateLeft(TNode* x) {
    TNode* y = x->right; //记录指向 y 节点的指针
    x->right = y->left;       //y 的左子节点连接到 x 的右
    y->left->parent = x;
    y->parent = x->parent;    //y 连接到 x 的父节点
    if (x->parent == nil) {  //x 是根节点
        root = y;           //修改树指针
    }
    else if (x == x->parent->left) {
        x->parent->left = y;//x 父节点左连接 y
    }
    else {
        x->parent->right = y;//x 父节点右连接 y
```

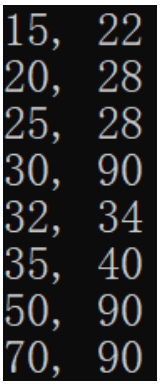
```
68.     }
69.     y->left = x; //x 连接到 y 左
70.     x->parent = y;
71.     y->high = x->high;
72.     x->high = tripleMax(x->left->high, x->right->high);
73. }
74.
75. void OSTree::rotateRight(TNode* y) {
76.     TNode* x = y->left;       //记录指向 x 节点的指针
77.     y->left = x->right;        //x 的右子节点连接到 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. }
94.
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. }
118.
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. }
129.
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
```

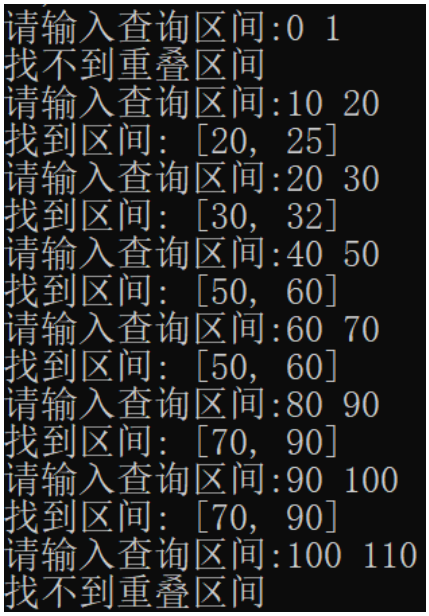
```
154.             TNode* y = z->parent->parent->left;     //y 是 z 的叔叔
155.             if (y->color == RED) {    //z 的叔叔 y 是红色
156.                 y->color = BLACK;
157.                 z->parent->color = BLACK;
158.                 z->parent->parent->color = RED;
159.                 z = z->parent->parent;
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. }
175.
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。