Experiment—Interval Tree Search

PB18010496 杨乐园

Introduction

通过对红黑树进行数据扩张与修改，使其成为一颗区间树，并实现区间树上的重叠区间的查找算法。其中，区间树的生成，要求初始时树为空，再依照文件$insert.txt$内信息逐节点按顺序依次插入生成树中，并完善交互界面信息，待查询的区间应由控制台输入，并直接在控制台打印查找结果。

Purpose

实验目的：熟悉并掌握针对红黑树数据结构的扩张，学习区间重叠的三分律，并编程掌握区间树的查找算法。

Data structure of interval tree and node

节点的数据结构为：

$$
\begin{aligned}
TNode = \{ & \\
& color : red/color; \\
& key : int; \\
& max : int; \\
& low, high : int; \\
& left : TNode*; \\
& right : TNode*; \\
& parent : TNode*; \\
\} &
\end{aligned}
$$

区间树的数据结构为：

$$
\begin{aligned}
RBTree = \{ & \\
& root : TNode*; \\
& nil : TNode*; \\
\} &
\end{aligned}
$$

其中，设区间树为T，则对$\forall x \in T$，节点x关键字key为相应区间的低点，即$x.key = x.int.low$；而附加信息max为当前节点x的区间的高点与其孩子节点的max的最大值，即$x.max = max\{x.int.high, x.left.max, x.right.max\}$，故由定理14.1即知附加信息的有效性。

Algorithm

首先我们先将附加信息max的维护函数写好，即当某节点的孩子节点发生改变时，其附加信息max的修正：

```
//附加信息的维护，返回max值；
int attach(TNode* x)
{
    int temp = x->high;
    if (x->max != error && x->left->max > temp)//左孩子max更大；
        temp = x->left->max;
    if (x->max != error && x->right->max > temp)//右孩子max更大；
        temp = x->right->max;
    return temp;
}
```

其次我们修改相应左旋与右旋操作，在变换过程中将附加信息max做好相应维护：

```
//左旋，输入树T,待旋转的节点x；
void LeftRotate(RBTree* T, TNode* x)
{
    TNode* y = x->right;     //y为x的右孩子；
    x->right = y->left;
    x->max = attach(x);      //x的孩子节点发生改变，故维护x.max；
    if(y->left!=T->nil)
        y->left->parent = x;
    y->parent = x->parent;   //y的父母节点指向x的父母节点；
    if (x->parent == T->nil)
        T->root = y;
    else if (x == x->parent->left)
        x->parent->left = y;
    else
        x->parent->right = y;
    y->parent->max = attach(y->parent);//y.parent的孩子节点改变，维护；
    y->left = x;
    y->max = attach(y);      //维护y节点；
    x->parent = y;
}

//右旋；
void RightRotate(RBTree* T, TNode* y)
{
    TNode* x = y->left;
    y->left = x->right;
    y->max = attach(y);
    if (x->right != T->nil)
        x->right->parent = y;
    x->parent = y->parent;
    if (y->parent == T->nil)
        T->root = x;
    else if (y == y->parent->right)
        y->parent->right = x;
    else
        y->parent->left = x;
    x->parent->max = attach(x->parent);
    x->right = y;
    x->max = attach(x);
    y->parent = x;
}
```

再者，由于生成区间树的必要，需要写好区间树的插入与维护算法，这直接与红黑树的插入与维护算法无异，如下：

```
//插入算法；
void RBInsert(RBTree* T, int k)
{
    TNode* y = T->nil; //y记录当前扫描节点的双亲结点；
    TNode* x = T->root;//从根开始扫描
    TNode* z = (TNode*)malloc(sizeof(TNode));
    z->color = black;
    z->key = k;
    while (x != T->nil)
    {
        y = x;
        if (z->key < x->key)
            x = x->left;
        else
            x = x->right;
    }
    z->parent = y;      //y是z的双亲
    if (y == T->nil)    //z插入空树，故z是根；
        T->root = z;
    else if (z->key < y->key)
        y->left = z;
    else
        y->right = z;
    z->left = T->nil;  //初始化z；
    z->right = T->nil;
    z->color = red;
    RBInsertFixup(T, z);
}
```

```
//插入的修正；
void RBInsertFixup(RBTree* T, TNode* z)
{
    while (z->parent->color == red)
        //若z为根，则z->parent=T->nil，颜色黑，不进入此循环；
        //若z->parent为黑，则无需调整，同样不进入循环；
    {
        if (z->parent == z->parent->parent->left)//case1,2,3
        {
            TNode* y = z->parent->parent->right; //y是z的叔叔；
            if (y->color == red)                 //case1
            {
                z->parent->color = black;
                y->color = black;
                z->parent->parent->color = red;
                z = z->parent->parent;
            }
            else                                 //case2 or case3，y为黑
            {
                if (z == z->parent->right)       //case2
                {
                    z = z->parent;
                    LeftRotate(T, z);
                }                                //以下为case3
```

```
                z->parent->color = black;
                z->parent->parent->color = red;
                RightRotate(T, z->parent->parent);
            }
        }
        else                                    //case4,5,6
        {
            TNode* y = z->parent->parent->left; //y是z的叔叔;
            if (y->color == red)                //case4
            {
                z->parent->color = black;
                y->color = black;
                z->parent->parent->color = red;
                z = z->parent->parent;
            }
            else                                //case5 or case6，y为黑
            {
                if (z == z->parent->left)       //case6
                {
                    z = z->parent;
                    RightRotate(T, z);
                }                               //以下为case6
                z->parent->color = black;
                z->parent->parent->color = red;
                LeftRotate(T, z->parent->parent);
            }
        }
    }
    T->root->color = black;
}
```

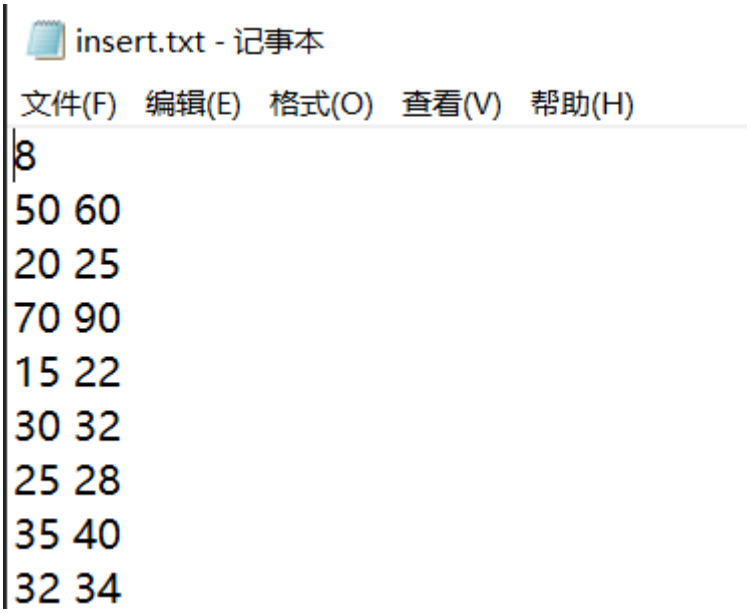
最后，实现重叠区间查找的算法，代码如下：

```
//判定是否重叠，若重叠则返回False;
bool overlap(TNode* x, int low, int high)
{
    if (x->low <= high && low <= x->high) //区间重叠;
        return false;
    return true;
}

//查找重叠区间;
TNode* search(RBTree* T, int low, int high)
{
    TNode* x = T->root;  //从根开始查找;
    while (x != T->nil && overlap(x, low, high))
    {
        if (x->left != T->nil && x->left->max >= low)
            x = x->left; //到x的左子树继续查找;
        else
            x = x->right;//到x的右子树继续查找;
    }
    return x;            //返回x=nil或重叠区间的指针节点
}
```

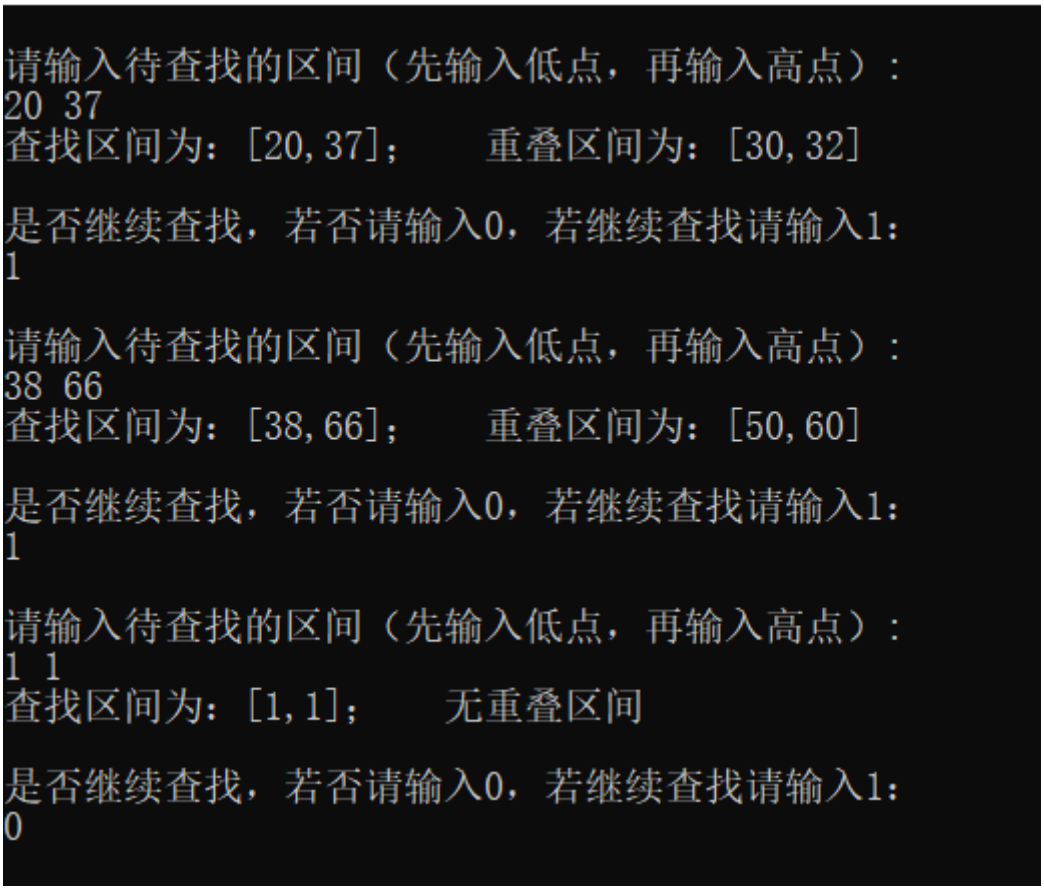
Results

通过运行程序建立如下区间树：



其中第一行为区间树节点总数，如下每一行分别为某个区间的低点与高点。

接下来对部分区间进行查询，结果如下：



通过对运行结果与实际结果的比对，我们可以看到查询结果的正确性，故算法设计与代码完成正确无误。

Code

具体完整代码，参看附件文件$Intervaltree$。