算法基础 Lab 4

区间树上重叠区间的查找算法

张芷苒

PB21081601

November 5, 2023

Part 1: 实验要求

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。

Part 2: 区间树的数据结构

区间树的结点 x 包括区间 (interval)、max 值，并且每一个结点的 key 值为 x.int.low 。

```
//定义红黑树的颜色：红或黑
enum NodeColour {
    RED,
    BLACK
};

//定义区间结构体，有低位和高位
struct IntervalSpan {
    int lowPoint;
    int highPoint;
};

//定义红黑树的节点结构
struct TreeNode {
    NodeColour nodeColour;
    IntervalSpan span;
    int keyValue;
    int maxValue;
    TreeNode *leftChild, *rightChild, *parent;
};

//定义红黑树结构
struct TreeStructure {
    TreeNode *treeRoot, *NILNode;
```

```
};
```

Part 3: 算法设计思路

区间树由红黑树扩展而来，二者大部分操作均比较相似。对于本实验而言，主要区别如下：

在树初始化的时候，要将 tree.NILNode 的 max 值赋为负无穷，可以省去很多繁琐的判断。

```
void InitializeTree(TreeStructure &tree) {
    tree.NILNode = new TreeNode;
    tree.NILNode->nodeColour = BLACK;
    tree.NILNode->maxValue = INT_MIN;
    tree.treeRoot = tree.NILNode;
    tree.treeRoot->parent = tree.NILNode;
}
```

每插入一个结点后，该节点会影响从此结点到根结点一条路径上的所有结点的 max 值，所以每插入一个结点都要进行一次自底向上的 max 值更新：

```
void AdjustMaxValue(TreeStructure &tree, TreeNode *z) {
    TreeNode *currentNode = z;
    while (currentNode != tree.NILNode) {
        currentNode->maxValue = max(max(currentNode->leftChild->maxValue, currentNode->
        currentNode = currentNode->parent;
    }
}

void InsertNode(TreeStructure &tree, TreeNode *z) {
    TreeNode *temp1 = tree.NILNode;
    TreeNode *temp2 = tree.treeRoot;

    while (temp2 != tree.NILNode) {
        temp1 = temp2;
        if (z->keyValue < temp2->keyValue) {
            temp2 = temp2->leftChild;
        } else {
            temp2 = temp2->rightChild;
        }
    }
```

```

    z->parent = temp1;

    if (temp1 == tree.NILNode) {
        tree.treeRoot = z;
    } else if (z->keyValue < temp1->keyValue) {
        temp1->leftChild = z;
    } else {
        temp1->rightChild = z;
    }

    z->leftChild = tree.NILNode;
    z->rightChild = tree.NILNode;
    z->nodeColour = RED;

    AdjustMaxValue(tree, z);
    InsertFix(tree, z);
}
```

左旋和右旋也会改变 max 值，但只会改变左旋结点及其对应的孩子结点的 max ，所以只需要对于旋转之后的两个点进行 max 值的更新 (先对孩子结点更新，再对父亲结点更新)。代码详见源码，在此不赘述。

重叠区间的查找算法:
区间重叠的判断条件是：对两个闭区间 x, y, 若 $x.low > y.high$ 或 $y.low > x.high$，则区间不重叠，否则重叠。

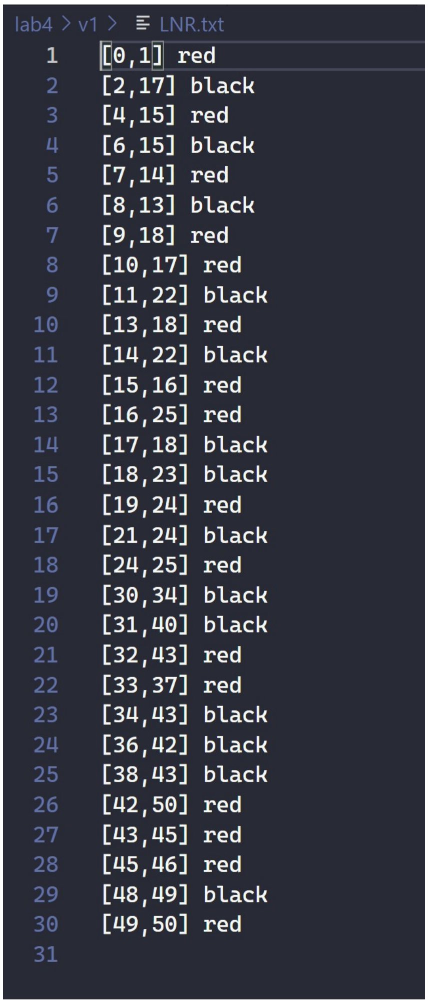
```
// 判断两个区间是否有重叠
int IsOverlap(IntervalSpan x, IntervalSpan y)
{
    return !(x.highPoint < y.lowPoint || x.lowPoint > y.highPoint);
}

// 查找与查询区间重叠的区间
TreeNode *SearchOverlapInterval(TreeStructure &tree, IntervalSpan querySpan)
{
    TreeNode *currentNode = tree.treeRoot;
    while (currentNode != tree.NILNode && !IsOverlap(querySpan, currentNode
->span)) {
        if (currentNode->leftChild != tree.NILNode && currentNode->leftChild
->maxValue >= querySpan.lowPoint) {
```

```
            currentNode = currentNode->leftChild;
        } else {
            currentNode = currentNode->rightChild;
        }
    }
    return currentNode;
}
```

Part 4: 实验结果分析

运行程序，可以得到以下输出结果：

实际结果：



lab4 > v1 > LNR.txt

```
1  [0,1] red
2  [2,17] black
3  [4,15] red
4  [6,15] black
5  [7,14] red
6  [8,13] black
7  [9,18] red
8  [10,17] red
9  [11,22] black
10 [13,18] red
11 [14,22] black
12 [15,16] red
13 [16,25] red
14 [17,18] black
15 [18,23] black
16 [19,24] red
17 [21,24] black
18 [24,25] red
19 [30,34] black
20 [31,40] black
21 [32,43] red
22 [33,37] red
23 [34,43] black
24 [36,42] black
25 [38,43] black
26 [42,50] red
27 [43,45] red
28 [45,46] red
29 [48,49] black
30 [49,50] red
31
```

图 1: 输出结果

进行查询，可以得到以下查询结果：

图 2: 查询结果

以上结果符合预期，由此可以认为算法正确。

Part 5 实验总结

在实验过程中获得了以下收获：

- 对于数据结构的扩张——区间树有了更加深刻的理解
- 在扩张的过程中维护扩张的性质需要保证时间复杂度仍为 $O(logn)$

下附：证明此算法单次查找的时间复杂度为 $O(min(n, klogn))$, **其中** k **为输出的区间数**。

在红黑树中进行区间搜索函数的时间复杂度取决于树的高度以及搜索过程中的操作。
红黑树的性质确保了树的高度为 $O(\log n)$，其中 n 是树中节点的数量。搜索操作每向下一步，要么是向左子树，要么是向右子树进行，因此，单次搜索的时间复杂度是 $O(\log n)$。
然而，我们搜索的不仅是单个节点；我们可能需要找到多个与给定区间有重叠的区间。在最坏的情况下，这意味着遍历树的一个分支，可能涉及所有的 k 个重叠的区间，其中 k 是与查询区间重叠的区间数量。在这种情况下，复杂度会增加到 $O(k)$。
因此，函数的时间复杂度是 $O(\log n)$ 加上找到所有 k 个重叠区间的复杂度 $O(k)$。
总的时间复杂度 $T(n)$ 可以表示为：

$$T(n) = O(\log n) + O(k)$$

但我们必须考虑实际操作中，当 k 变得很大时，$O(k)$ 可能会超过 $O(\log n)$。因此，最终的时间复杂度取决于这两者中的较大值。所以，我们有：

$$T(n) = O(\max(\log n, k))$$

此表达式表示复杂度受到 $\log n$ 或 k 中较大者的限制。