Project 3 report

PB20111623 马子睿

实验内容与要求

实验内容

- 实现区间树的基本算法：
  - 随机生成30个正整数区间，以这30个正整数区 间的左端点作为关键字构建红黑树，先向一棵初始空的红黑树中依次插入 30个节点，然后随机选择其中3个区间进行删除，最后对随机生成的3个区间(其中一个区间取自(25,30))进行搜索。
  - 实现区间树的插入、删除、遍历和查找算法。

实验要求

- input/input.txt：
  - 输入文件中每行两个随机数据，表示区间的左右端点，其右端点值大于左端点值，总行数大于等于30。
  - 所有区间取自区间[0,25]或[30,50]且各区间左端点互异，和(25,30)无重叠。读取每行数据作为区间树的x.int域，并以其左端点构建红黑树，实现插入、删除、查找操作。
- output/inorder.txt:
  - 输出构建好的区间树的中序遍历序列，每行三个非负整数，分别为各节点int域左右端点和max域的值。
- delete_data.txt：
  - 输出删除的数据，以及删除完成后区间树的中序遍历序列。
- search.txt:
  - 对随机生成的3个区间(其中一个区间取自(25,30))进行搜索得到的结果，搜索成功则返回一个与搜索区间 重叠的区间，搜索失败返回NULL。

实验配置与环境

- 操作系统：Windows 11 专业版（version 21H2）
- 处理器：11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
- 时钟主频：2.80 GHz
- 编译器：gcc version 8.1.0

实验方法与步骤

实验方法

区间树是对红黑树数据结构的扩张，我们应首先维护好完备的红黑树，之后通过加入“区间”的属性，来实现对区间重叠的查找。当然，维护区间中max的值也需要在旋转、插入、删除等操作中完成，且并不破坏插入、删除操作的时间复杂度$O(logn)$

实验步骤

1. 构建区间树的数据结构

   首先是区间对象：

```
struct intval{
    int low;
    int high;
    intval(){}
    intval(int _low, int _high){ low = _low; high = _high;}
};
```

   其次是扩展的红黑树节点：

```
struct RBnode{
    int key;
    Color color;
    RBnode *lson;
    RBnode *rson;
    RBnode *p;
    int max;
    intval intv;
    RBnode(int _key, int _max){
        key = _key; max = _max;
        lson = rson = p = nullptr;
        color = BLACK; intv = intval(key, max);
    }
};
```

   这里利用了C++构造函数来进行了更为简单的构造。

   之后是区间树数据结构：

```
class RBTree{
private:
    RBnode *nil;

    bool is_overlap(intval a, intval b);

    int get_max(RBnode *z);
    void left_rotate(RBnode *z);
    void right_rotate(RBnode *z);

    void RB_insert_fixup(RBnode *z);

    RBnode* RB_tree_minimum(RBnode *z);
    void RB_trasplant(RBnode *u, RBnode *v);
    void RB_delete_fixup(RBnode *z);

public:
    RBnode *root;

    RBTree(){
        nil = new RBnode(-1, -1);
        root = nil;
```

```
    }

    void RB_insert(RBnode *z);
    bool RB_delete(RBnode *z);
    RBnode* RB_search(intval *intv, RBnode *r);
    intval *interval_search(intval i);

    void mid_trv(RBnode* r, ofstream &fout);
};
```

可以看出，这颗二叉树支持：

- 插入（包括旋转操作和插入调整）
- 删除（包括旋转操作和删除调整）
- 查找区间是否存在
- 查找重叠区间
- 中序遍历区间树

2. 编写区间树操作的各个方法

- 旋转：旋转操作基本与书中无异，但为了适应区间树的max域变化，需要加入常数时间的维护代码（这里以右旋为例）

```
void RBTree::right_rotate(RBnode *z){
    // fix z->lson
    RBnode *y = z->lson;
    z->lson = y->rson;
    if(y->rson != nil) y->rson->p = z;

    // fix y->p
    y->p = z->p;

    // fix z->p
    if(z->p == nil) root = y; // z is root
    else if(z == z->p->lson) z->p->lson = y;
    else z->p->rson = y;

    // fix y->rson
    y->rson = z;
    z->p = y;

    y->max = get_max(y);
    z->max = get_max(z);
}
```

在旋转过程之后，由于y和z的max域可能发生变化，因此我们必须更新这两个max域。

- 插入：插入操作也与书中基本无异，但在插入过程中，我们必须考虑插入路径上的max域变化。仔细思考可以得知：**只需要在插入路径上的每一个点和被插入点的max域比较一下（插入节点的max域被初始化为high），并将更大的值赋值给路径上的点即可。**这是因为，插入节点一定位于叶子结点，如果它的max比路径上所有的都大，那么路径上每个节点的max域自然应该更新成待插入节点的max域。

  这里给出插入的代码。插入的红黑修正代码与书中无异，详细代码请参考源文件。

```
void RBTree::RB_insert(RBnode *z){
    RBnode *y = nil;
    RBnode *x = root;
```

```
    // find location
    while(x != nil){
        y = x;
        x->max = max(x->max, z->max);
        if(z->key < x->key) x = x->lson;
        else x = x->rson;
    }
    // link z with rbtree
    z->p = y;
    if(y == nil) root = z;
    else if(z->key < y->key) y->lson = z;
    else y->rson = z;
    z->lson = z->rson = nil;
    z->max = get_max(z);
    z->color = RED;
    RB_insert_fixup(z);
}
```

- 删除：删除操作的实现基本与书中无异，但删除操作对于max域的维护应当格外小心。删除操作对max域的维护主要在调整之前：
  - 对于没有左孩子或没有右孩子的情况，我们需要对删除节点的父节点，和取代删除节点的节点的max域进行维护。这是因为，删除节点可能是最大的max域，一旦删除，其父必须要重新计算max。取代删除节点的节点将删除节点的左儿子接为自己的儿子，因此也需要重新维护max。
  - 对于左右孩子都有的情况，我们还是需要对删除节点的父节点，和取代删除节点的节点的max域进行维护，原因和上面相同。值得注意的是，这里我们**还需要对取代删除节点的y的父节点重新维护**，因为这个节点的儿子被提到上方，可能max域会有变化。具体操作如下：

```
bool RBTree::RB_delete(RBnode *z){
    if(z == nil) return false;

    RBnode *x;
    RBnode *y = z;
    Color y_orig_color = y->color;

    /* case 1: no lson, link directly*/
    if(z->lson == nil){
        x = z->rson;
        RB_trasplant(z, z->rson);
        if(z->p != nil) z->p->max = get_max(z->p);
        if(x != nil) x->max = get_max(x);
    }
    /* case 2: no rson, link directly */
    else if(z->rson == nil){
        x = z->lson;
        RB_trasplant(z, z->lson);
        if(z->p != nil) z->p->max = get_max(z->p);
        if(x != nil) x->max = get_max(x);
    }
    /* case 3: both has lson and rson */
    else {
        y = RB_tree_minimum(z->rson);
        y_orig_color = y->color;
        x = y->rson;
        if(y->p == z) x->p = y;
        else {
```

```
            RB_trasplant(y, y->rson);
            y->rson = z->rson;
            y->rson->p = y;
            if(y->p != nil) y->p->max = get_max(y->p);
        }
        RB_trasplant(z, y);
        y->lson = z->lson;
        y->lson->p = y;
        y->color = z->color;
        if(z->p != nil) z->p->max = get_max(z->p);
        if(y != nil) y->max = get_max(y);
    }
    if(y_orig_color == BLACK) RB_delete_fixup(x);
    delete z;
    return true;
}
```

其中，维护节点的max值时一定要小心，这里我有一些改变，详情见"遇到的问题"

之后我们依然需要为删除后的区间树进行调整。调整的代码与书中无异，请参见源代码。

- 遍历：实验要求的是中序遍历，我们按照普通的中序遍历来编写即可：

```
void RBTree::mid_trv(RBnode *r, ofstream &fout){
    if(r == nil) return;
    mid_trv(r->lson, fout);
    fout << r->intv.low << " " << r->intv.high << " " << r->max << endl;
    cout << r->intv.low << " " << r->intv.high << " " << r->max << endl;
    mid_trv(r->rson, fout);
}
```

由于我们需要输出到文件，因此加入一个文件流参数即可。

- 区间树的查找：

区间树需要能够对重叠区间进行查找。

```
intval* RBTree::interval_search(intval i){
    RBnode *x = root;
    while(x != nil && !is_overlap(i, x->intv)){
        if(x->lson != nil && x->lson->max >= i.low) x = x->lson;
        else x = x->rson;
    }
    if(x == nil) return nullptr;
    else return &(x->intv);
}
```

区间树通过max域来进行二分查找：如果当前的区间和待查找的区间无重叠，那么当左孩子的max域比需要查找的区间的左边界大时，证明待查找的区间一定会出现在左侧，那么就需要向左进行查找了。

对于区间是否重叠的判断，我们使用如下代码进行实现

```
bool RBTree::is_overlap(intval a, intval b){
    if(a.low>b.high||a.high<b.low) return false;
    else return true;
}
```

实验结果与分析

为了对代码进行验证，我做了一些测试：

1. 插入后全清空测试：

   我们创建一棵40节点的区间树，之后将其全部删除，最终应该得到一棵空树。事实上代码确实如此：

```
delete intvals:
6 11
32 46
43 50
35 45
44 49
20 21
2 4
41 43
35 35
39 39
32 38
2 14
37 41
21 24
6 15
11 22
49 50
32 37
47 50
14 20
46 47
35 46
45 47
4 14
34 47
19 24
38 40
6 8
48 48
3 17
42 42
13 22
36 44
10 12
16 22
4 23
0 21
2 7
2 18
40 46
middle traverse result:
```

   可以看到，删除后中序遍历为空。

2. 使用书中样例进行简单测试：

   执行结果如下：

```
0 3 3
5 8 10
6 10 10
8 9 23
15 23 23
16 21 30
17 19 20
19 20 20
25 30 30
26 26 26
delete intvals:
15 23
25 30
6 10
middle traverse result:
0 3 3
5 8 10
8 9 9
16 21 26
17 19 19
19 20 26
26 26 26
search intval: 25 30
result: 26 26
search intval: 5 45
result: 16 21
search intval: 22 24
result: NULL
```

可以看到删除之后的结果是正确的，同时对于查找的结果，删除了[15,23]的区间后，区间[22,24]查找是失败的，其他结果都是正确的。

3. 使用实验要求中的方法进行测试

实验要求我们生成一个不小于30节点的区间树，并对其中序遍历后，删除三个节点，再进行查找。生成输入数据的函数已经放在了src文件夹下，输入输出也放在了input和output文件夹下，故在此不再赘述。

遇到的问题

1. 查找区间是否存在的问题

在删除节点时，我们首先要在树中查找这个节点。这引入了一个问题：如果关键字相等，那么我们应该查询左子树还是右子树呢？一开始我并没有注意到这个问题，导致了删除出现了问题。之后我意识到，由于我们没有办法确定相等关键字究竟放在了左子树还是右子树，因此我们需要把两个子树都遍历一下：

```
RBnode* RBTree::RB_search(intval* intv, RBnode *r){
    if(r == nil || r->intv.low == intv->low && r->intv.high == intv->high ) return r;
    if(intv->low < r->key) return RB_search(intv, r->lson);
    else if(intv->low > r->key) return RB_search(intv, r->rson);
    else {
        RBnode * temp = RB_search(intv, r->lson);
        return (temp == nil) ? RB_search(intv, r->rson) : temp;
    }
}
```

实验总结

通过本次实验，我有了如下收获：

- 对红黑树和区间树有了更深刻的理解，能够独立编写区间树的操作。
- 更详细地了解了C++中的new特性
- 通过观察红黑树创建时的旋转和跳帧操作，理解了红黑树效率较高的原因