实验三 区间树

PB20000137 李远航

一、实验内容及要求

- 区间树
    - 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。

二、实验设备及环境

```
OS: Ubuntu 20.04 focal(on the Windows Subsystem for Linux)
Kernel: x86_64 Linux 5.10.102.1-microsoft-standard-WSL2
CPU: Intel Core i5-10200H @ 8x 2.4GHz
GPU: NVIDIA GeForce GTX 1650 Ti
g++ (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0
```

三、实验方法和步骤

1. 数据结构的设计
    - 数据域

    ```
    struct Interval
    {
        std::pair<int, int> in;
        int max;
        friend bool operator<(const Interval &a, const Interval &b);
        friend bool operator>(const Interval &a, const Interval &b);
        friend bool operator==(const Interval &a, const Interval &b);
    };
    ```

    - 树上的节点

    ```
    struct TreeNode
    {
        ForColor color;
        struct Interval key;
        struct TreeNode *left, *right, *parent;
    };
    ```

    - 区间树类

    ```
    class IntervalTree
    {
    private:
        struct TreeNode NILL = {black, {}, nullptr, nullptr, nullptr};
        struct TreeNode *root;
        void leftRotate(struct TreeNode *x);
        void rightRotate(struct TreeNode *y);
    ```

```
    void RBinsertFixup(struct TreeNode *z);
    void RBTransplant(struct TreeNode *u, struct TreeNode *v);
    void RBdeleteFixup(struct TreeNode *x);
    struct TreeNode *TreeMinmum(struct TreeNode *x);
    int overlap(std::pair<int, int> a, std::pair<int, int> b);
    int max_(int a, int b, int c);
    void updatemax(struct TreeNode *x);
    struct TreeNode *search(std::pair<int, int> key);
    void RBout(struct TreeNode *p, std::ofstream &outfile);
public:
    struct TreeNode *NIL = &NILL;
    IntervalTree();
    ~IntervalTree();
    void RBinsert(std::pair<int, int> key);
    void RBdelete(std::pair<int, int> key);
    struct TreeNode *Intervalsearch(std::pair<int, int> i);
    void print(std::ofstream &outfile) { RBout(root, outfile); };
};
```

2. 关键函数的实现

   - 旋转，以及旋转时维护 `max`

```
void IntervalTree::leftRotate(struct TreeNode *x)
{
    struct TreeNode *y = x->right;
    x->right = y->left;
    if (y->left != NIL)
        y->left->parent = x;
    y->parent = x->parent;
    if (x->parent == NIL)
        root = y;
    else if (x == x->parent->left)
        x->parent->left = y;
    else
        x->parent->right = y;
    y->left = x;
    x->parent = y;
    y->key.max = x->key.max;
    x->key.max = max_(x->key.in.second, x->left->key.max, x->right->key.max);
}
```

   - 插入节点及插入后红黑树性质的维护

```
void IntervalTree::RBinsert(std::pair<int, int> key)
{
    struct TreeNode *z = new TreeNode;
    struct Interval tmp = {key, key.second};
    z->key = tmp;
    z->color = red;
    z->parent = NIL;
    z->left = NIL;
    z->right = NIL;
    struct TreeNode *y = NIL;
    struct TreeNode *x = root;
    while (x != NIL)
    {
        x->key.max = std::max(x->key.max, z->key.max);
        y = x;
        if (z->key < x->key)
            x = x->left;
        else
            x = x->right;
    }
    z->parent = y;
    if (y == NIL)
        root = z;
    else if (z->key < y->key)
        y->left = z;
    else
        y->right = z;
    z->left = NIL;
    z->right = NIL;
    z->color = red;
    RBinsertFixup(z);
}

void IntervalTree::RBinsertFixup(struct TreeNode *z)
{
    struct TreeNode *y;
    while (z->parent->color == red)
    {
        if (z->parent == z->parent->parent->left)
        {
            y = z->parent->parent->right;
            if (y->color == red)
            {
                z->parent->color = black;
                y->color = black;
                z->parent->parent->color = red;
                z = z->parent->parent;
            }
            else
            {
                if (z == z->parent->right)
                {
                    z = z->parent;
                    leftRotate(z);
                }
                z->parent->color = black;
                z->parent->parent->color = red;
                rightRotate(z->parent->parent);
            }
        }
        else
        {
            y = z->parent->parent->left;
            if (y->color == red)
            {
                z->parent->color = black;
                y->color = black;
                z->parent->parent->color = red;
                z = z->parent->parent;
            }
            else
            {
                if (z == z->parent->left)
                {
                    z = z->parent;
                    rightRotate(z);
                }
                z->parent->color = black;
                z->parent->parent->color = red;
                leftRotate(z->parent->parent);
            }
        }
    }
    root->color = black;
}
```

- 节点的删除及维护红黑树的性质

```
void IntervalTree::RBdeleteFixup(struct TreeNode *x)
{
    struct TreeNode *w;
    while (x != root && x->color == black)
    {
        if (x == x->parent->left)
        {
            w = x->parent->right;
            if (w->color == red)
            {
                w->color = black;
                x->parent->color = red;
                leftRotate(x->parent);
                w = x->parent->right;
            }
            if (w->left->color == black && w->right->color == black)
            {
                w->color = red;
                x = x->parent;
            }
            else
            {
                if (w->right->color == black)
                {
                    w->left->color = black;
                    w->color = red;
                    rightRotate(w);
                    w = x->parent->right;
                }
                w->color = x->parent->color;
                x->parent->color = black;
                w->right->color = black;
                leftRotate(x->parent);
                x = root;
            }
        }
        else
        {
            w = x->parent->left;
            if (w->color == red)
            {
                w->color = black;
                x->parent->color = red;
                rightRotate(x->parent);
                w = x->parent->right;
            }
            if (w->left->color == black && w->right->color == black)
            {
                w->color = red;
                x = x->parent;
            }
            else
            {
                if (w->left->color == black)
                {
                    w->right->color = black;
                    w->color = red;
                    leftRotate(w);
                    w = x->parent->left;
                }
                w->color = x->parent->color;
                x->parent->color = black;
                w->left->color = black;
                rightRotate(x->parent);
                x = root;
            }
        }
    }
}

void IntervalTree::RBdelete(std::pair<int, int> key)
{
    struct TreeNode *z = search(key);
    if (z == NIL)
        return;
    struct TreeNode *y = z, *x;
    ForColor origin = y->color;
    if (z->left == NIL)
    {
        x = z->right;
        RBTransplant(z, z->right);
        updatemax(x->parent);
    }
    else if (z->right == NIL)
    {
        x = z->left;
        RBTransplant(z, z->left);
        updatemax(x->parent);
    }
    else
    {
        y = TreeMinmum(z->right);
        origin = y->color;
        x = y->right;
        if (y->parent == z)
            x->parent = y;
        else
        {
            RBTransplant(y, y->right);
            y->right = z->right;
            y->right->parent = y;
        }
        RBTransplant(z, y);
        y->left = z->left;
        y->left->parent = y;
        y->color = z->color;
        updatemax(x);
    }
    if (origin == black)
        RBdeleteFixup(x);
}
```

  - 重叠区间的查找

```

struct TreeNode *IntervalTree::Intervalsearch(std::pair<int, int> i)
{
    struct TreeNode *x = root;
    while (x != NIL && !overlap(i, x->key.in))
    {
        if (x->left != NIL && x->left->key.max >= i.first)
            x = x->left;
        else
            x = x->right;
    }
    return x;
}
```

四、实验结果和分析

- 随机的输入数据见 `./input/input.txt`
- 生成的区间树的中序遍历见 `./output/inorder.txt`
- 使用随机数获取需要删除和查找的区间
  - 在 `./output/delete_data.txt` 文件中有三组输出，每组输出第一行表示需要删除的区间，接下来为删除数据后区间树的中序遍历
  - 在 `./output/search.txt` 文件中同样有三组输出，每组输出第一行表示需要查找的区间，第二行为输出的结果

五、实验思考与反思

- 学习了红黑树的数据结构以及拓展
- 较为复杂的数据结构在亲身实现后会有更深的理解
- 本次实验中，设计的 `key` 值的数据结构过于复杂，给调试带来了很多的麻烦(需要展开变量)