

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

```
1 OS: Ubuntu 20.04 focal(on the Windows Subsystem for Linux)
2 Kernel: x86_64 Linux 5.10.102.1-microsoft-standard-WSL2
3 CPU: Intel Core i5-10200H @ 8x 2.4GHz
4 GPU: NVIDIA GeForce GTX 1650 Ti
5 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

```
1 struct Interval
2 {
3     std::pair<int, int> in;
4     int max;
5     friend bool operator<(const Interval &a, const Interval &b);
6     friend bool operator>(const Interval &a, const Interval &b);
7     friend bool operator==(const Interval &a, const Interval &b);
8 };
```

- 树上的节点

```
1 struct TreeNode
2 {
3     ForColor color;
4     struct Interval key;
5     struct TreeNode *left, *right, *parent;
6 };
```

- 区间树类

```
1 class IntervalTree
2 {
3 private:
4     struct TreeNode NIL = {black, {}, nullptr, nullptr, nullptr};
5     struct TreeNode *root;
6     void leftRotate(struct TreeNode *x);
7     void rightRotate(struct TreeNode *y);
```

```

8     void RInsertFixup(struct TreeNode *z);
9     void RBTransplant(struct TreeNode *u, struct TreeNode *v);
10    void RBdeleteFixup(struct TreeNode *x);
11    struct TreeNode *TreeMinum(struct TreeNode *x);
12    int overlap(std::pair<int, int> a, std::pair<int, int> b);
13    int max_(int a, int b, int c);
14    void updatemax(struct TreeNode *x);
15    struct TreeNode *search(std::pair<int, int> key);
16    void RBout(struct TreeNode *p, std::ofstream &outfile);
17    public:
18        struct TreeNode *NIL = &NILL;
19        IntervalTree();
20        ~IntervalTree();
21        void RInsert(std::pair<int, int> key);
22        void RBdelete(std::pair<int, int> key);
23        struct TreeNode *Intervalsearch(std::pair<int, int> i);
24        void print(std::ofstream &outfile) { RBout(root, outfile); };
25    };

```

2. 关键函数的实现

- 旋转，以及旋转时维护 `max`

```

1     void IntervalTree::leftRotate(struct TreeNode *x)
2     {
3         struct TreeNode *y = x->right;
4         x->right = y->left;
5         if (y->left != NIL)
6             y->left->parent = x;
7         y->parent = x->parent;
8         if (x->parent == NIL)
9             root = y;
10        else if (x == x->parent->left)
11            x->parent->left = y;
12        else
13            x->parent->right = y;
14        y->left = x;
15        x->parent = y;
16        y->key.max = x->key.max;
17        x->key.max = max_(x->key.in.second, x->left->key.max, x->right->key.max);
18    }

```

- 插入节点及插入后红黑树性质的维护

```

1     void IntervalTree::RInsert(std::pair<int, int> key)
2     {
3         struct TreeNode *z = new TreeNode;
4         struct Interval tmp = {key, key.second};
5         z->key = tmp;
6         z->color = red;
7         z->parent = NIL;
8         z->left = NIL;
9         z->right = NIL;
10        struct TreeNode *y = NIL;
11        struct TreeNode *x = root;
12        while (x != NIL)

```

```

13     {
14         x->key.max = std::max(x->key.max, z->key.max);
15         y = x;
16         if (z->key < x->key)
17             x = x->left;
18         else
19             x = x->right;
20     }
21     z->parent = y;
22     if (y == NIL)
23         root = z;
24     else if (z->key < y->key)
25         y->left = z;
26     else
27         y->right = z;
28     z->left = NIL;
29     z->right = NIL;
30     z->color = red;
31     RInsertFixup(z);
32 }
33
34 void IntervalTree::RInsertFixup(struct TreeNode *z)
35 {
36     struct TreeNode *y;
37     while (z->parent->color == red)
38     {
39         if (z->parent == z->parent->parent->left)
40         {
41             y = z->parent->parent->right;
42             if (y->color == red)
43             {
44                 z->parent->color = black;
45                 y->color = black;
46                 z->parent->parent->color = red;
47                 z = z->parent->parent;
48             }
49             else
50             {
51                 if (z == z->parent->right)
52                 {
53                     z = z->parent;
54                     leftRotate(z);
55                 }
56                 z->parent->color = black;
57                 z->parent->parent->color = red;
58                 rightRotate(z->parent->parent);
59             }
60         }
61         else
62         {
63             y = z->parent->parent->left;
64             if (y->color == red)
65             {
66                 z->parent->color = black;
67                 y->color = black;
68                 z->parent->parent->color = red;

```

```

69         z = z->parent->parent;
70     }
71     else
72     {
73         if (z == z->parent->left)
74         {
75             z = z->parent;
76             rightRotate(z);
77         }
78         z->parent->color = black;
79         z->parent->parent->color = red;
80         leftRotate(z->parent->parent);
81     }
82 }
83 }
84 root->color = black;
85 }

```

- 节点的删除及维护红黑树的性质

```

1  void IntervalTree::RBdeleteFixup(struct TreeNode *x)
2  {
3      struct TreeNode *w;
4      while (x != root && x->color == black)
5      {
6          if (x == x->parent->left)
7          {
8              w = x->parent->right;
9              if (w->color == red)
10             {
11                 w->color = black;
12                 x->parent->color = red;
13                 leftRotate(x->parent);
14                 w = x->parent->right;
15             }
16             if (w->left->color == black && w->right->color == black)
17             {
18                 w->color = red;
19                 x = x->parent;
20             }
21             else
22             {
23                 if (w->right->color == black)
24                 {
25                     w->left->color = black;
26                     w->color = red;
27                     rightRotate(w);
28                     w = x->parent->right;
29                 }
30                 w->color = x->parent->color;
31                 x->parent->color = black;
32                 w->right->color = black;
33                 leftRotate(x->parent);
34                 x = root;
35             }
36         }
37     }
38 }

```

```

37         else
38         {
39             w = x->parent->left;
40             if (w->color == red)
41             {
42                 w->color = black;
43                 x->parent->color = red;
44                 rightRotate(x->parent);
45                 w = x->parent->right;
46             }
47             if (w->left->color == black && w->right->color == black)
48             {
49                 w->color = red;
50                 x = x->parent;
51             }
52             else
53             {
54                 if (w->left->color == black)
55                 {
56                     w->right->color = black;
57                     w->color = red;
58                     leftRotate(w);
59                     w = x->parent->left;
60                 }
61                 w->color = x->parent->color;
62                 x->parent->color = black;
63                 w->left->color = black;
64                 rightRotate(x->parent);
65                 x = root;
66             }
67         }
68     }
69 }
70
71 void IntervalTree::RBdelete(std::pair<int, int> key)
72 {
73     struct TreeNode *z = search(key);
74     if (z == NIL)
75         return;
76     struct TreeNode *y = z, *x;
77     ForColor origin = y->color;
78     if (z->left == NIL)
79     {
80         x = z->right;
81         RBTransplant(z, z->right);
82         updatemax(x->parent);
83     }
84     else if (z->right == NIL)
85     {
86         x = z->left;
87         RBTransplant(z, z->left);
88         updatemax(x->parent);
89     }
90     else
91     {
92         y = TreeMinum(z->right);

```

```

93         origin = y->color;
94         x = y->right;
95         if (y->parent == z)
96             x->parent = y;
97         else
98         {
99             RBTransplant(y, y->right);
100             y->right = z->right;
101             y->right->parent = y;
102         }
103         RBTransplant(z, y);
104         y->left = z->left;
105         y->left->parent = y;
106         y->color = z->color;
107         updatemax(x);
108     }
109     if (origin == black)
110         RBdeleteFixup(x);
111 }

```

- 重叠区间的查找

```

1
2  struct TreeNode *IntervalTree::Intervalsearch(std::pair<int, int> i)
3  {
4      struct TreeNode *x = root;
5      while (x != NIL && !overlap(i, x->key.in))
6      {
7          if (x->left != NIL && x->left->key.max >= i.first)
8              x = x->left;
9          else
10             x = x->right;
11     }
12     return x;
13 }

```

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)

