实验3 红黑树和区间树

实验设备和环境

macOS Mojave 10.14.6 MacBook Pro (Retina, 13-inch, Early 2015) 处理器 2.9 GHz Intel Core i5

IDEA Clion

实验3.1 红黑树

实验要求

- 实现红黑树的基本算法， 分别对整数 n = 20、40、60、80、100，随机生成 n 个互异的正整数（K1, K2, K3, ..., Kn）, 以这 n 个正整数作为结点的关键字，向一棵初始空的红黑树中依次插入n个节点，统计算法运行所需时间 ，画出时间曲线。
- 随机删除红黑树中 n/4 个结点，统计删除操作所需时间，画出时间曲线图。

■实验3.1 红黑树
- □ex1/input/
  - ● input.txt：随机生成的n个正整数，用于构建红黑树。
- □ex1/output/
  - ● inorder.txt:：构建好的红黑树的中序遍历序列。
  - ● time1.txt：构建红黑树的插入操作所花费时间。
  - ● delete_data.txt ： 红黑树删除的结点关键字及删除后的中序遍历序列。
  - ● time2.txt：删除红黑树结点的时间。
- □同行数据间用空格隔开，其中n=20，40，60，80，100。

实验过程

关于红黑树，我根据课本构造了 RBTNode 和 RBT 两种数据结构。

```
typedef struct rbtnode{
    bool color;
    int key;
    struct rbtnode *left, *right, *p;
} RBTNode;
typedef struct rbt{
    RBTNode *nil, *root;
} RBT;
```

main.c 的主体是 n 分别为 20，40，60，80，100 的循环体，对于不同的 n，操作的不同仅与 n 有关。

```
for(int i = 0; i < 5; i++){
    n = (i+1)*20;
    ...
}
```

所以接下来仅针对循环体内的结构介绍。

首先，构造一颗初始化好的红黑树：

```
T = (RBT *)malloc(sizeof(RBT));
T->nil = (RBTNode *)malloc(sizeof(RBTNode));
T->nil->color = BLACK;
T->nil->key = -1;
T->nil->left = T->nil->right = T->nil->p = T->nil;
T->root = T->nil;
```

关于输入，需要随机生成 n 个**互不相同**的数。我使用了 Python 中的 random.sample。

```
import random
f = open('./input/input.txt', 'w')

for i in range(0, 5):
    n = 20*i+20
    L = random.sample(range(256), n)
    for num in L:
        f.write(str(num)+" ")
    f.write('\n')

f.close()
```

在 main.c 中读入 input.txt，存在全局数组 array 中：

```
#define MAX_LEN 100

int array[MAX_LEN];

void rand_input(int n, FILE *fp){
    for(int i = 0; i < n; i++){
        fscanf(fp, "%d", &array[i]);
    }
    return;
}
```

准备好输入之后，可以开始计时，仍使用 time.h 提供的 clock() 函数。

```
start_t = clock();
for(int j = 0; j < n; j++){
    RB_INSERT_KEY(T, array[j]);
}
end_t = clock();
```

插入函数 RB_INSERT_KEY() 的实现是核心，基本是依据课本的伪代码实现的。实现 RB_INSERT_KEY() 需要构造如下的子函数：

```
void LEFT_ROTATE(RBT *T, RBTNode *x);
void RIGHT_ROTATE(RBT *T, RBTNode *x);
void RB_INSERT(RBT *T, RBTNode *z);
void RB_INSERT_FIXUP(RBT *T, RBTNode *z);
```

因为这四个函数课本都有较完整的代码，我就不仔细解释了，仅把代码贴在下面：

```
void LEFT_ROTATE(RBT *T, RBTNode *x){
    RBTNode *y = x->right;
    x->right = y->left;
    if(y->left != T->nil)
        y->left->p = x;
    y->p = x->p;
    if(x->p == T->nil)
        T->root = y;
    else if(x == x->p->left)
        x->p->left = y;
    else
        x->p->right = y;
    y->left = x;
    x->p = y;
}

void RIGHT_ROTATE(RBT *T, RBTNode *x){
    RBTNode *y = x->left;
```

```
    x->left = y->right;
    if(y->right != T->nil)
        y->right->p = x;
    y->p = x->p;
    if(x->p == T->nil)
        T->root = y;
    else if(x == x->p->left)
        x->p->left = y;
    else
        x->p->right = y;
    y->right = x;
    x->p = y;
}

void RB_INSERT(RBT *T, RBTNode *z){
    RBTNode *y = T->nil, *x = T->root;
    while(x != T->nil){
        y = x;
        if(z->key < x->key)
            x = x->left;
        else
            x = x->right;
    }
    z->p = y;
    if(y == T->nil)
        T->root = z;
    else if(z->key < y->key)
        y->left = z;
    else
        y->right = z;
    z->left = T->nil;
    z->right = T->nil;
    z->color = RED;
    RB_INSERT_FIXUP(T, z);
}

void RB_INSERT_FIXUP(RBT *T, RBTNode *z){
    RBTNode *y;
    //while(z != T->root & z->p->color == RED){
    while(z->p->color == RED){
        if(z->p == z->p->p->left){
            y = z->p->p->right;
            if(y->color == RED){
                z->p->color = BLACK;
                y->color = BLACK;
                z->p->p->color = RED;
                z = z->p->p;
            }
            else {
```

```
                if(z == z->p->right){
                    z = z->p;
                    LEFT_ROTATE(T, z);
                }
                z->p->color = BLACK;
                z->p->p->color = RED;
                RIGHT_ROTATE(T, z->p->p);
            }
        }
        else{ // wait to check
            y = z->p->p->left;
            if(y->color == RED){
                z->p->color = BLACK;
                y->color = BLACK;
                z->p->p->color = RED;
                z = z->p->p;
            }
            else {
                if (z == z->p->left) {
                    z = z->p;
                    RIGHT_ROTATE(T, z);
                }
                z->p->color = BLACK;
                z->p->p->color = RED;
                LEFT_ROTATE(T, z->p->p);
            }
        }
    }
    T->root->color = BLACK;
}
```

要比课本伪代码多做一步的就是从 key 到结点的转换，这个在 RB_INSERT_KEY 中处理：

```
void RB_INSERT_KEY(RBT *T, int key){
    RBTNode *z = (RBTNode *)malloc(sizeof(RBTNode));
    z->key = key;
    z->left = z->right = z->p = T->nil;
    z->color = RED; // wait to check
    RB_INSERT(T, z);
}
```

实验要求把插入 n 个结点的耗时写入 time1.txt，这个与之前的实验实现有相似之处，就不展开讲了。关于 inorder.txt 的处理，实现如下：

```
void InOrderTraverse(RBTNode *T, RBTNode *nil, FILE *fp){
    char buf[BUF_LEN];
    if(T != nil){
        InOrderTraverse(T->left, nil, fp);
```

```
        printf("%d ", T->key);
        memset(buf, '0', BUF_LEN * sizeof(char));
        sprintf(buf, "%d ", T->key);
        fprintf(fp, "%s", buf);
        InOrderTraverse(T->right, nil, fp);
    }
    return;
}

// main
printf("n = %d insert\n", n);
InOrderTraverse(T->root, T->nil, fp3);
printf("\n");
fprintf(fp3, "\n");
```

到此与插入有关的实现已经介绍完了，下面是删除 n/4 个结点的实验部分。

要求是“随机删除红黑树中 n/4 个结点”，仍然使用 Python 中的 random.sample（隐含了不相同的要求），但思路略有不同。从下面的代码中可以看出，我是生成了 n 范围内的 n/4 个索引值。

```
import random
f = open('./input/delete_index.txt', 'w')

for i in range(0, 5):
    n = 5*i+5
    L = random.sample(range(n*4), n) # 索引
    for num in L:
        f.write(str(num)+" ")
    f.write('\n')

f.close()
```

只需在 main.c 中将索引转化为 array[index] 即可确定下来要删除的 key。

```
void rand_delete(int n, FILE *fp){
    int index;
    for(int i = 0; i < n; i++){
        fscanf(fp, "%d", &index);
        delete_data[i] = array[index];
    }
    return;
}
```

与 insert 类似，main 中的处理核心就是 RB_DELETE_KEY() 函数：

```
rand_delete(n/4, fp5);
start_t = clock();
for(int j = 0; j < n/4; j++){
```

```
    RB_DELETE_KEY(T, delete_data[j]);
}
end_t = clock();
total_t = (double) (end_t - start_t) / CLOCKS_PER_SEC;
memset(buf, '0', BUF_LEN * sizeof(char));
sprintf(buf, "%f\n", total_t);
fprintf(fp6, "%s", buf);
// delete_data.txt
printf("n = %d delete\n", n);
InOrderTraverse(T->root, T->nil, fp4);
printf("\n");
fprintf(fp4, "\n");
```

下面列举的四个函数，课本比较完整的伪代码，我就不仔细解释了。

```
void RB_TRANSPLANT(RBT *T, RBTNode *u, RBTNode *v);
RBTNode *TREE_MINIMUM(RBT *T, RBTNode *x);
void RB_DELETE(RBT *T, RBTNode *z);
void RB_DELETE_FIXUP(RBT *T, RBTNode *x);

void RB_TRANSPLANT(RBT *T, RBTNode *u, RBTNode *v){
    if(u->p == T->nil)
        T->root = v;
    else if(u == u->p->left)
        u->p->left = v;
    else
        u->p->right = v;
    //if(v!=T->nil)
    v->p = u->p;
    return;
}

RBTNode *TREE_MINIMUM(RBT *T, RBTNode *x){
    while(x->left != T->nil)
        x = x->left;
    return x;
}

void RB_DELETE(RBT *T, RBTNode *z){
    RBTNode *y = z, *x;
    bool y_original_color = y->color;
    if(z->left == T->nil){
        x = z->right;
        RB_TRANSPLANT(T, z, z->right);
    }
    else if(z->right == T->nil){
        x = z->left;
        RB_TRANSPLANT(T, z, z->left);
```

```
    }
    else{
        y = TREE_MINIMUM(T, z->right);
        y_original_color = y->color;
        x = y->right;
        if(y->p == z)
            x->p = y;
        else{
            RB_TRANSPLANT(T, y, y->right);
            y->right = z->right;
            y->right->p = y;
        }
        RB_TRANSPLANT(T, z, y);
        y->left = z->left;
        y->left->p = y;
        y->color = z->color;
    }
    if(y_original_color == BLACK)
        RB_DELETE_FIXUP(T, x);
    return;
}

void RB_DELETE_FIXUP(RBT *T, RBTNode *x){
    RBTNode *w;
    while(x != T->root & x->color == BLACK){
        if(x == x->p->left){
            w = x->p->right;
            if(w->color == RED){
                w->color = BLACK;
                x->p->color = RED;
                LEFT_ROTATE(T, x->p);
                w = x->p->right;
            }
            if(w->left->color == BLACK & w->right->color == BLACK){
                w->color = RED;
                x = x->p;
            }
            else{
                if(w->right->color == BLACK){
                    w->left->color = BLACK;
                    w->color = RED;
                    RIGHT_ROTATE(T, w);
                    w = x->p->right;
                }
                w->color = x->p->color;
                x->p->color = BLACK;
                w->right->color = BLACK;
                LEFT_ROTATE(T, x->p);
                x = T->root;
```

```
            }
        }
        else {
           w = x->p->left;
            if(w->color == RED){
                w->color = BLACK;
                x->p->color = RED;
                RIGHT_ROTATE(T, x->p);
                w = x->p->left;
            }
            if(w->left->color == BLACK & w->right->color == BLACK){
                w->color = RED;
                x = x->p;
            }
            else{
                if(w->left->color == BLACK){
                    w->right->color = BLACK;
                    w->color = RED;
                    LEFT_ROTATE(T, w);
                    w = x->p->left;
                }
                w->color = x->p->color;
                x->p->color = BLACK;
                w->left->color = BLACK;
                RIGHT_ROTATE(T, x->p);
                x = T->root;
            }
        }
    }
    x->color = BLACK;
    return;
}
```

考虑到实验要求，构造了这两个函数，RB_SEARCH() 针对 key 搜索结点，RB_DELETE_KEY() 调用 RB_SEARCH() 和 RB_DELETE() 删除结点：

```
RBTNode *RB_SEARCH(RBT *T, RBTNode *x, int key){
    if(x == T->nil || key == x->key)
        return x;
    else if(key < x->key)
        return RB_SEARCH(T, x->left, key);
    else
        return RB_SEARCH(T, x->right, key);
}

void RB_DELETE_KEY(RBT *T, int key){
    RBTNode *p = RB_SEARCH(T, T->root, key);
    RB_DELETE(T, p);
}
```

值得一提的是，虽然实验没有要求，但我认为需要考虑对于树结点的释放，利用了后序遍历，实现如下：

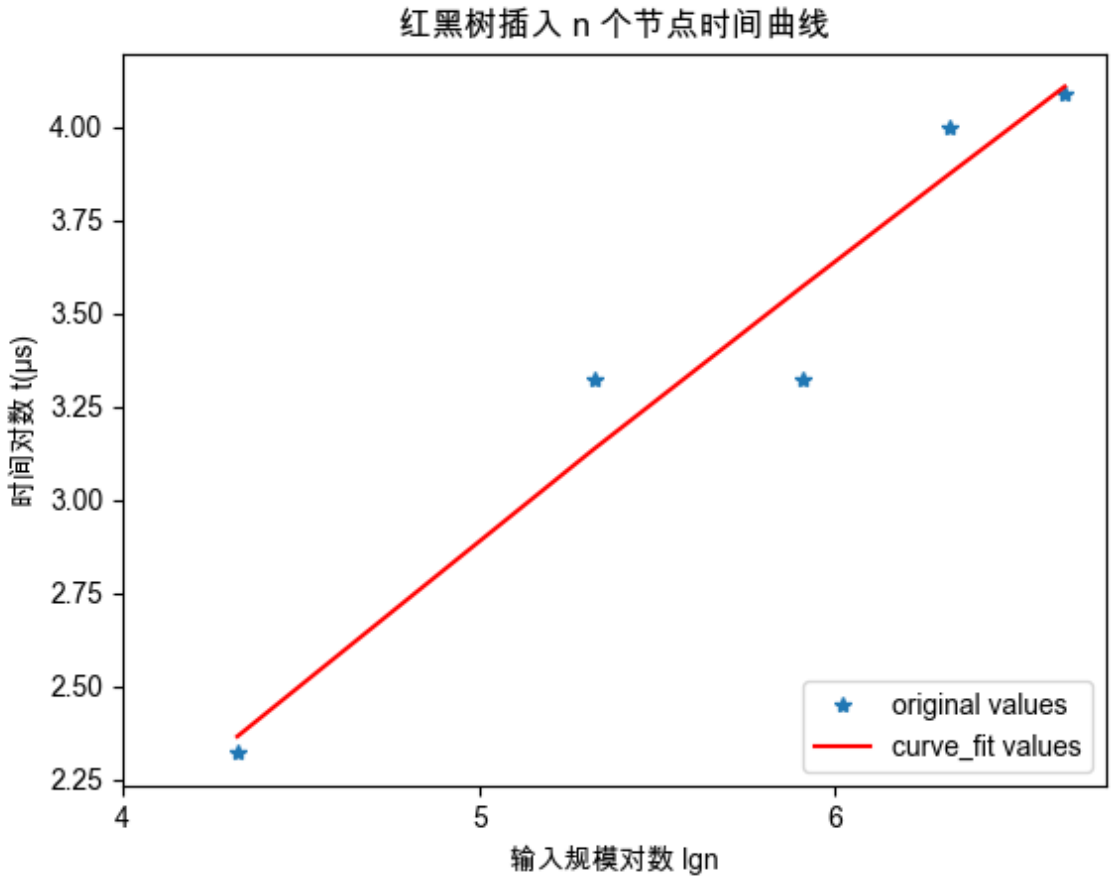
```
void PostOrderTraverse(RBTNode *T, RBTNode *nil){
    if(T != nil){
        PostOrderTraverse(T->left, nil);
        PostOrderTraverse(T->right, nil);
        free(T);
    }
    return;
}

PostOrderTraverse(T->root, T->nil);
free(T->nil);
```

实验结果

比较实际复杂度和理论复杂度是否相同，给出分析。

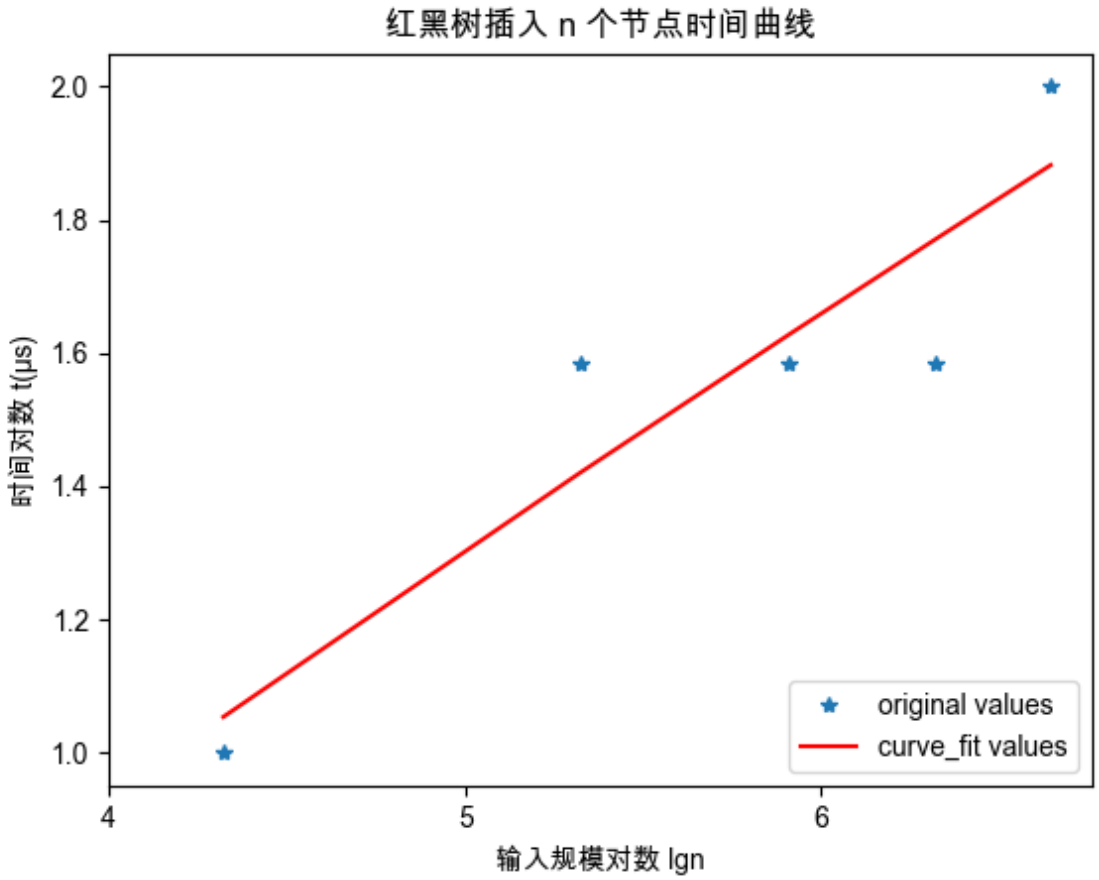
插入操作是 O(lgn) 的时间复杂度，time1 是 O(lg1+lg2+...lgn) = O(n!) = O(nlgn)

横坐标为 [20, 40, 60, 80, 100] 的对数，时间为换算成微秒后取的对数。考虑到误差，可以认为实际复杂度和理论复杂度（拟合）相同。



删除操作我计时时没有剔除删除前查找结点的步骤，因为查找和删除都是 O(lgn) 的时间复杂度。因为 time2 计时是 n/4 *O(lgn) 所以理论复杂度应该是 O(nlgn)。

横坐标为 [20, 40, 60, 80, 100] 的对数，时间为换算成微秒后取的对数。考虑到误差，可以认为实际复杂度和理论复杂度（拟合）相同。



实验3.2 区间树

实验要求

- 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，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。实现区间树的插入、删除、遍历和查找算法。

■实验3.2 区间树
- □ex2/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，不要和(25,30)有重叠。
    - 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。
- □ex2/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。
- □同行数据间用空格隔开

实验过程

对于区间树的要求，引入了新的数据结构 Interval：

```
typedef struct Interval {
    int low, high;
} Interval;
```

把 key 替换为 interval，加入 max：

```
typedef struct rbtnode{
    Interval interval;
    int max;
    ...
} RBTNode;
```

对于 interval 和 max 属性也要做初始化，区间信息从 input.txt 中获得，不妨把 max 初始化为 interval.high。

```
RBTNode *INTERVAL_INIT_KEY(RBT *T, FILE *fp){
    RBTNode *z = (RBTNode *)malloc(sizeof(RBTNode));
    fscanf(fp, "%d", &z->interval.low);
    fscanf(fp, "%d", &z->interval.high);
    z->left = z->right = z->p = T->nil;
    z->color = RED;
    z->max = z->interval.high;
    return z;
}
```

维护 max 属性要分为两个部分，左右旋后需平衡一次，整体还需平衡一次，具体如下： 1）需要三个数取最大的函数

```
int Max(int a, int b, int c){
    if(a > b)
        b = a;
    if(b > c)
        return b;
    else
        return c;
}
```

2) 在 LEFT_ROTATE() 和 RIGHT_ROTATE() 中补充，例如

```
void LEFT_ROTATE(RBT *T, RBTNode *x){
    ...
    // about max
    x->max = Max(x->interval.high, x->left->max, x->right->max);
    y->max = Max(y->interval.high, y->left->max, y->right->max);
}
```

3）在 RB_INSERT() 函数的尾部，对整体的 max 做一次维护：

```
void RB_INSERT(RBT *T, RBTNode *z){
    ...
    MAX_FIXUP(T, z);
    RB_INSERT_FIXUP(T, z);
}

void MAX_FIXUP(RBT *T, RBTNode *z){
    while(z != T->nil){
        z->max = Max(z->interval.high, z->left->max, z->right->max);
        z = z->p;
    }
}
```

至此，插入的算法已经处理好了。下面介绍搜索。

搜索需要引入 Overlap() 和 INTERVAL_SEARCH()，和红黑树中的搜索 key 的逻辑不同。这里终于用到了前面对于 max 属性的小心维护。

```
bool Overlap(Interval a, Interval b){
    if(a.low < b.low)
        if(a.high < b.low)
            return false;
        else
            return true;
    else if(a.low == b.low)
        return true;
    else
    if(b.high < a.low)
        return false;
    else
        return true;
}

RBTNode *INTERVAL_SEARCH(RBT *T, Interval i){
    RBTNode *x = T->root;
    while(x != T->nil && !Overlap(x->interval, i)){
        if(x->left != T->nil && x->left->max >= i.low)
            x = x->left;
        else
```

```
            x = x->right;
        }
        return x;
}
```

删除逻辑与红黑树几乎一致，只需将 key 替换为 interval.low，加入 max 的维护。

```
void RB_DELETE(RBT *T, RBTNode *z){
    ...
    MAX_FIXUP(T, x);
    if(y_original_color == BLACK)
        RB_DELETE_FIXUP(T, x);
    return;
}
```

和实验 3.1 类似，也利用了后序遍历对树结点进行了释放。

关于实验结果的部分代码在下一部分介绍。

实验结果

三个文件的生成

1. input.txt 插入 30 个节点，右端点大于左端点，左端点互异，所有区间取自 [0, 25] ∪ [30, 50]。

构造输入数据的代码如下：

```
import random
f = open('./input/input.txt', 'w')

list1 = list(range(0, 25))
list2 = list(range(30, 50))
list3 = list(range(0, 26))
list4 = list(range(30, 51))

list5 = list1+list2

L = random.sample(list5, 30)    # 左端点互异
for left in L:
    if left in list1:
        right = random.choice(list3)
        while right <= left:        # 右端点更大
            right = random.choice(list3)
    else:
        right = random.choice(list4)
        while right <= left:        # 右端点更大
            right = random.choice(list4)
    f.write(str(left)+" "+str(right)+'\n')

```

```
f.close()
```

main() 中只需读入数据、初始化结点、插入树即可：

```
RBTNode *x;
for(int i = 0; i < 30; i++){
    x = INTERVAL_INIT_KEY(T, fp1);
    RB_INSERT(T, x);
}
InOrderTraverse(T->root, T->nil, fp2);
```

2. search.txt 随机生成三个区间，失败返回 Null。要包含一个应该返回 Null 的样例。

构造随机区间的代码如下：

```
import random
f = open('./input/search_interval.txt', 'w')

list1 = list(range(0, 26))
list2 = list(range(30, 50))
list3 = list1+list2
list4 = list(range(30, 51))
list5 = list1+list4

L = random.sample(list3, 2) # 索引
for left in L:
    f.write(str(left)+" ")
    right = random.choice(list5)
    while right <= left:        # 右端点更大
        right = random.choice(list5)
    f.write(str(right)+'\n')

f.write(str(26)+" "+str(28)+"\n")
f.close()
```

查询失败就写入 Null，否则写入查到的区间：

```
Interval interval;
for(int i = 0; i < 3; i++){
    fscanf(fp6, "%d", &interval.low);
    fscanf(fp6, "%d", &interval.high);
    x = INTERVAL_SEARCH(T, interval);
    if(x == T->nil)
        fprintf(fp5, "Null\n");
    else
        fprintf(fp5, "%d %d\n", x->interval.low, x->interval.high);
}
```

3. delete_data.txt 随机选择三个区间进行删除，输出删除的数据以及删除后的中序遍历序列。

随机选择区间的代码如下：

```
import random
f1 = open('./input/delete_data.txt', 'w')
f2 = open('./input/input.txt', 'r')

L = random.sample(range(30), 3) # 索引
print(L)
input_data = []
for line in f2.readlines():
    print(line)
    l = line.split(" ")
    input_data.append(l[0])
print(input_data)

for index in L:
    f1.write(str(input_data[index])+" ")
f1.write("\n")

f1.close()
f2.close()
```

删除其实依据的是区间左边界，读入的数据只需要是三个 interval.low。

```
for(int i = 0; i < 3; i++){
    fscanf(fp3, "%d", &interval.low);
    RBTNode *p = RB_SEARCH(T, T->root, interval.low);
    fprintf(fp4, "%d %d\n", p->interval.low, p->interval.high);
    RB_DELETE(T, p);
}
InOrderTraverse(T->root, T->nil, fp4);
```

I/O

input/input.txt 满足对于输入的限制：

```
5 15
1 5
37 50
9 10
16 24
42 50
10 16
36 46
47 50
33 47
```

```
30 33
22 24
13 24
6 12
32 41
0 2
12 21
21 22
23 24
46 47
18 23
34 35
3 7
15 18
35 36
38 39
4 13
24 25
39 43
19 23
```

output/inorder.txt low high max

```
0 2 2
1 5 13
3 7 13
4 13 13
5 15 24
6 12 12
9 10 12
10 16 24
12 21 21
13 24 24
15 18 18
16 24 50
18 23 23
19 23 23
21 22 22
22 24 25
23 24 25
24 25 25
30 33 41
32 41 41
33 47 50
34 35 35
35 36 46
36 46 46
37 50 50
```

```
38 39 39
39 43 50
42 50 50
46 47 50
47 50 50
```

input/search_interval.txt

```
41 50
34 46
26 28
```

output/search.txt 查找到的内容满足区间树要求

```
33 47
33 47
Null
```

input/delete_data.txt 需删除的结点

```
34 46 10
```

output/delete_data.txt 前 3 行是选择删除的三个区间，后 27 行是剩下的内容的中序输出

```
34 35
46 47
10 16
0 2 2
1 5 13
3 7 13
4 13 13
5 15 24
6 12 12
9 10 12
12 21 21
13 24 24
15 18 18
16 24 50
18 23 23
19 23 23
21 22 22
22 24 25
23 24 25
24 25 25
30 33 41
32 41 41
33 47 50
```

```
35 36 46
36 46 46
37 50 50
38 39 39
39 43 50
42 50 50
47 50 50
```