

# 实验三 红黑树和区间树

陈俊驰 PB18000051

## 一、实验内容及要求:

本次实验包含两个部分，红黑树和区间树，具体要求如下。

### 1) 红黑树

在该实验中，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 ( $K_1, K_2, \dots, K_n$ )，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  $n$  个结点，统计算法运行所需时间并画出时间曲线。然后，随机删除红黑树中  $n/4$  个结点，统计删除操作所需要时间并画出时间曲线图。

对于输入，从 input 文件夹中的 input.txt 文件读入，随机生成的正整数，用于构建红黑树。在本次实验中，选择生成共 100 个随机数，然后分别读取前 20~100 个作为输入。

对于输出，对于五个规模，分别构建红黑树，在 time1.txt 文件输出构建红黑树的插入操作所花费的时间，并在 inorder.txt 中输出构建好的红黑树的中序遍历序列；删除  $n/4$  个结点后，在 time2.txt 文件中输出删除红黑树结点的时间，并在 delete\_data.txt 中输出删除的结点关键字以及删除后的中序遍历序列。

### 2) 区间树

在该实验中，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。

对于输入，从 input 文件夹中的 input.txt 文件读入，每一行两个随机数据表示区间的左右端点，读取每行数据作为结点的 int 域。其中，右端点应大于左端点，所有区间取自  $[0,25][30,50]$  且左端点互异，不能有与  $(25,30)$  有重叠。

对于输出，将构建好的区间树的中序遍历序列以每行为左右端点及 max 域的格式输出到 inorder.txt 中，然后删除 3 个区间并将删除的数据及删除后中序遍历输出到 delete.txt 中，最后随机生成三个区间，其中一个取自  $(25,30)$ ，在区间树中进行搜索，将搜索到的重叠区间或者 Null 输出到 search.txt。

## 二、实验设备和环境:

本次实验所使用的实验设备为笔记本电脑，型号为 2018 版联想拯救者，处理器为 Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz 2.30 GHz，内存为 8.00GB。所使用的实验环境为 Windows 10，所使用的 IDE 为 Visual Studio 2019。

## 三、实验方法和步骤:

首先，根据实验文档的名称与结构要求建立根文件夹，文件夹名称为 36-陈俊驰-PB18000051-project3，在根文件夹下建立实验报告文件和 ex1、ex2 子文件夹，在子文件夹中创立三个子文件夹，分别为 input 文件夹，用于存储输入文件；src 文件夹，用于存储源程序，包含项目文件夹及 cpp 文件，由于所使用的 IDE 为 Visual Studio，每个源程序对应一个项目文件夹，由于相对路径问题，在执行

时需要打开该项目在 VS 中执行,而不能直接通过使用 Debug 文件夹中存储的 exe 文件; output 文件夹,其中为每个算法在 output 文件夹下构建一个子文件夹用于存储结果数据和时间结果。

### 1) 红黑树

0.首先根据要求生成 100 个互异的正整数,其中 search 函数用于在数组中搜索当前生成的随机数是否已经存在,从而保证互异

```
for (int n = 0; n <= 100; n++)
    b[n] = 0;
//b数组用于存储将要输出到input.txt文件的结果
while (i <= 100)
{
    a = rand();
    if (a > 0 && search(a, b) == false)
        //如果该随机数为正数,且a不在数组b中,则说明a为与已产生的数互异的一个正整数
        //并将a加入b数组
        b[i] = a;
        i++;
}
for (i = 1; i <= 100; i++)
    fprintf(fp, "%d ", b[i]);
//输出结果
```

图 1 生成 100 个互异的正整数

1.首先定义红黑树的结构体, RBTnode 为红黑树的节点; RBTtree 为红黑树,包括根与 nil 两部分。其中为了方便表示颜色,利用 enum 定义 red 和 black。

```
enum COLOR {
    red, black
};

struct RBTnode {
    enum COLOR color;
    int key;
    RBTnode* left;
    RBTnode* right;
    RBTnode* p;
};

struct RBTtree {
    RBTnode* root;
    RBTnode* nil;
};
```

图 2 定义红黑树结构体

2.打开对应的文件并初始化相关变量数组,其中 in 代表输入,其余四个代表插入删除的输出。input 数组用于存储输入的数据。

```

FILE* fpin = fopen("../..\\input\\input.txt", "r");
FILE* fporder = fopen("../..\\output\\inorder.txt", "w");
FILE* fptimeinser = fopen("../..\\output\\time1.txt", "w");
FILE* fpdel = fopen("../..\\output\\delete_data.txt", "w");
FILE* fptimedel = fopen("../..\\output\\time2.txt", "w");
int n = 20;
int input[101];
int del[101];
int i = 0;
input[0] = 0;
for (i = 1; i <= 100; i++)
{
    fscanf(fpin, "%d", &input[i]);
}
RBTtree* T;
RBTnode* z;
//打开文件以及初始化变量。读入input.txt的数据

```

图 3 初始化变量打开文件

3.然后，对不同规模  $n$  进行如下的操作，首先初始化一个根节点即为  $T.nil$  的空树作为初始；然后通过创建并利用  $rb\_insert$  插入  $n$  个关键字值为读入数据的结点，并输出其时间及中序遍历序列；然后，利用  $del$  数组作为  $input$  中第  $i$  个数据是否被删除的标志，选中  $n/4$  个值，找到其结点并利用  $rb\_delete$  删除，并输出删除时间以及中序遍历序列。

```

for (n = 20; n <= 100; n = n + 20)
{
    //对于20,40...100这五种情况进行处理
    T = (RBTtree*)malloc(sizeof(RBTtree));
    T->nil = (RBTnode*)malloc(sizeof(RBTnode));
    T->nil->color = black;
    T->nil->p = T->nil->left = T->nil->right = NULL;
    T->root = T->nil;
    //初始化树
    int i;
    auto start = system_clock::now();
    //开始计时
    for (i = 1; i <= n; i++)
    {
        //进行n个结点的插入操作
        z = (RBTnode*)malloc(sizeof(RBTnode));
        z->key = input[i];
        rb_insert(T, z);
    }
    auto end = system_clock::now();
    auto duration = duration_cast<nanoseconds>(end - start);
    //计算时间
    fprintf(fptimeinser, "对于%d规模花费了 %lf us\n", n, 1000000 * double(duration.count()) * nanoseconds::period::num / nanoseconds::period::den);
    fprintf(fporder, "规模为%d时中序遍历序列为:", n);
    inorder_tree_walk(T, T->root, fporder);
    fprintf(fporder, "\n");
    //对插入的时间及中序遍历结果输出
    int ndel = n / 4;
    int randnum;
    int key;
    RBTnode* tem=NULL;
    for (i = 1; i <= n; i++)
    {
        del[i] = 0;
        //利用del数组来标识input中的第i个数据是否被删除，是为1
        fprintf(fpdel, "当规模为%d时删除节点关键字为:", n);
        i = 1;
        start = system_clock::now();
        //开始计时
    }
}

```

```

while (i <= ndel)
{
    //删除ndel个结点
    randnum = rand() % (n+1);
    if (randnum <= 0)
        continue;
    //获得1~n之间的随机数
    if (del[randnum] == 1)
        continue;
    else
    {
        //如果序号对应数值的结点未被删除则删除
        key = input[randnum];
        tem = search(T, key, T->root);
        rb_delete(T, tem);
        del[randnum] = 1;
        i++;
        fprintf(fpdel, "%d ", key);
    }
}

end = system_clock::now();
duration = duration_cast<nanoseconds>(end - start);
//计算时间
fprintf(fptimedel, "对于%d规模删除花费了 %lf us\n", n, 1000000 * double(duration.count()) * nanoseconds::period::num / nanoseconds::period::den);
fprintf(fpdel, "\n此时中序遍历序列为: ");
inorder_tree_walk(T, T->root, fpdel);
fprintf(fpdel, "\n");
//输出删除结果及时间
}

```

图 4 对不同规模所进行的操作

4.部分展示如下的代码，其中插入删除 `rb_insert` 和 `rb_delete` 及相关代码由于过长就不具体展示，均按照课本所给出的伪代码的思想来书写。对于中序遍历，利用二叉树的规则来写。对于 `tree_minimum` 和 `search` 由于红黑树本质上也是一种二叉搜索树，故利用二叉搜索树的性质来书写。其中 `RBTnode` 和 `RBTtree` 分别为红黑树结点和树的结构体定义。

```

enum COLOR {
    red, black
};

struct RBTnode {
    enum COLOR color;
    int key;
    RBTnode* left;
    RBTnode* right;
    RBTnode* p;
};

struct RBTtree {
    RBTnode* root;
    RBTnode* nil;
};

void left_rotate(RBTtree* T, RBTnode* x) { ... }

void right_rotate(RBTtree* T, RBTnode* x) { ... }

void rb_insert_fixup(RBTtree* T, RBTnode* z) { ... }

void rb_insert(RBTtree* T, RBTnode* z) { ... }

void rb_transplant(RBTtree* T, RBTnode* u, RBTnode* v) { ... }

void rb_delete_fixup(RBTtree* T, RBTnode* x) { ... }

RBTnode* tree_minimum(RBTtree* T, RBTnode* z )
{
    //找到以z为根的子树中的最小结点
    RBTnode* x = z, * y = z->left;
    if (z == T->nil)
        return NULL;
    while (y != T->nil)
    {
        //一直向左知道遇到nil
        x = y;
        y = y->left;
    }
    return x;
}

void rb_delete(RBTtree* T, RBTnode* z) { ... }

void inorder_tree_walk(RBTtree* T, RBTnode* x, FILE* fp)
{
    //中序遍历输出结果
    if (x != T->nil)
    {
        inorder_tree_walk(T, x->left, fp);
        fprintf(fp, "%d ", x->key);
        inorder_tree_walk(T, x->right, fp);
    }
}

```

图 5 插入删除及相关函数代码实现

## 2) 区间树

0. 首先根据要求生成区间，其中通过 `search` 函数来保证不存在左端点，通过

对于随机生成的左端点的值不同情况进行讨论，保证右端点要大于左端点且不与(25,30)有重叠。

```
while (i <= 30)
{
    a = rand()%51;
    if (search(a, left))//保证不存在相同左端点
        continue;
    if (a >= 0 && a < 25)
        //对于取自[0, 25]的区间
        b = rand() % 26;
        while (b <= a)
            //保证b大于a
            b = rand() % 26;
        }
    left[i] = a;
    right[i] = b;
    i++;
}
else if (a >= 30 && a < 50)
    //对于取自[30, 50]的区间
    b = rand() % 21 + 30;
    while (b <= a)
    {
        b = rand() % 21 + 30;
    }
    left[i] = a;
    right[i] = b;
    i++;
}
else
    continue;
}
```

图 6 生成随机的输入区间

1.首先，定义区间树的结构体，`intnode` 表示区间树的结点；`inttree` 表示区间树，包含 `root` 和 `nil`；由于 C 语言本身不直接支持区间运算，故加入 `interval` 结构体用于定义区间，其包括区间上界 `high` 和下界 `low`。其中为了方便表示颜色，利用 `enum` 定义 `red` 和 `black`。

```

enum COLOR {
    red, black
};

struct interval {
    int low;
    int high;
};

struct intnode {
    enum COLOR color;
    interval inter;
    int max;
    intnode* left;
    intnode* right;
    intnode* p;
};

struct inttree {
    intnode* root;
    intnode* nil;
};

```

图 7 区间树结构体定义

2.打开文件，并初始化变量数组，在 **low** 和 **high** 数组中对应的存储从 **input.txt** 文件中输入的时间

```

FILE* fpin = fopen("../..\\input\\input.txt", "r");
FILE* fpinorder = fopen("../..\\output\\inorder.txt", "w");
FILE* fpdel = fopen("../..\\output\\delete_data.txt", "w");
FILE* fpsearch = fopen("../..\\output\\search.txt", "w");
int low[31];
int high[31];
int i, a;
intnode* x;
//打开文件并申请一些变量
for (i = 1; i <= 30; i++)
    fscanf(fpin, "%d%d", &low[i], &high[i]);
//读入输入的时间

```

图 8 初始化并读入数据

3.然后，对于所读入的数据，进行如下的操作，首先初始化一棵空的区间树 **T**，其根节点为 **T.nil**；声明一系列节点，并将输入的区间作为其区间域的 **low** 和 **high** 值初始化，然后调用 **rb\_insert** 插入，并输出中序遍历结果；然后随机选择并调用 **rb\_delete** 删除三个区间，并输出中序遍历结果及删除的节点值；最后随机生成 3 个区间，包括一个(25, 30)内的区间，进行搜索并输出搜索的结果。



```

inttree* T = (inttree*)malloc(sizeof(inttree));
T->nil = (intnode*)malloc(sizeof(intnode));
T->nil->inter.high = T->nil->inter.low = NULL;
T->nil->max = 0;
T->nil->color = black;
T->nil->p = T->nil->left = T->nil->right = NULL;
T->root = T->nil;
//初始化树
for (i = 1; i <= 30; i++)
{
    //构建并插入结点
    x = (intnode*)malloc(sizeof(intnode));
    x->inter.low = low[i];
    x->inter.high = high[i];
    rb_insert(T, x);
}
check(T, T->root); //测试使用, 检查max是否合规
inorder_tree_walk(T, T->root, fpinorder); //中序遍历
fprintf(fpdel, "删除的数据为: \n");
for (i = 1; i <= 3; i++)
{
    //随机删除三个区间
    a = rand() % 30 + 1;
    x = search(T, low[a], T->root);
    fprintf(fpdel, "%d %d %d\n", x->inter.low, x->inter.high, x->max);
    rb_delete(T, x);
}
fprintf(fpdel, "中序遍历序列为: \n");
check(T, T->root);
inorder_tree_walk(T, T->root, fpdel);

interval s1;
for (i = 1; i <= 2; i++)
{
    //随机生产查找两个区间
    s1.low = rand() % 50;
    while (true)
    {
        s1.high = rand() % 51;
        if (s1.high > s1.low)
            break;
    }
    fprintf(fpsearch, "第%d个搜索区间: %d %d\n", i, s1.low, s1.high);
    fprintf(fpsearch, "搜索结果为: ");
    interval_search(T, s1, fpsearch);
}
//生成并查找一个取自(25, 30)的区间
s1.low = rand() % 3 + 26;
while (true)
{
    s1.high = rand() % 4 + 26;
    if (s1.high > s1.low)
        break;
}
fprintf(fpsearch, "第3个搜索区间: %d %d\n", s1.low, s1.high);
fprintf(fpsearch, "搜索结果为: ");
interval_search(T, s1, fpsearch);

```

图 9 区间树对于规模 30 的处理



4. 由于原代码过长，故部分展示如下的代码。其中，由于区间树是在红黑树基础上进行扩展，故对于插入删除的基本算法保持不变，但需要加入对于 **max** 值的维护，在这里展示 **insert** 是如何维护 **max** 值的并展示一些自己加入的函数。主要加入的函数为 **maintain**，将节点 **x** 处的 **max** 值修改为符合规则的值；**overlap**，利用定义判断是否重叠；**check**，利用中序遍历检查每一个结点的 **max** 值是否合规。对于 **insert**，根据课本的思想，在第一阶段插入的过程中，变化将沿着插入位置向上传播，故我们在 **insert** 函数找寻路径的过程中对每一个祖先的 **max** 值进行修改；第二阶段，进行红黑性质恢复，仅有的结构改变是在于旋转，根据旋转后，**y** 是 **x** 的父亲，先对 **x** 再对 **y** 维护其 **max** 值。对于 **delete**，第一阶段中，为被删除节点的祖先和取代删除节点位置的结点的原祖先维护 **max** 值，第二阶段与插入同理。对于中序遍历，简单利用树的中序遍历思想即可。对于搜索，参照课本算法完成。

```
int max(int a, int b, int c) { ... }
void maintain(inttree* T, intnode* x)
{
    //维护x结点的max值
    if (x != T->nil)
        x->max = max(x->inter.high, x->left->max, x->right->max);
}

void left_rotate(inttree* T, intnode* x)
{
    //左旋
    intnode* y = x->right; //定义y
    x->right = y->left; //使y的左子树成为x的右子树
    if (y->left != T->nil)
        y->left->p = x;
    y->p = x->p; //y的父亲更改为x的父亲
    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; //将x放到y的left
    x->p = y; //使y成为x的父亲
    maintain(T, x);
    maintain(T, y);
    //维护x和y的max
}

void right_rotate(inttree* T, intnode* x) { ... }
void rb_insert_fixup(inttree* T, intnode* z)
{
    //插入红黑性质恢复
    intnode* y;
    while (z->p->color == red)
    {
        if (z->p == z->p->p->left)
        {
            y = z->p->p->right;
            if (y->color == red)
                //对应课本情况1
                z->p->color = black;
                y->color = black;
                z->p->p->color = red;
                maintain(T, z->p);
                maintain(T, z->p->p);
                z = z->p->p;
            }
            else
            {
                if (z == z->p->right)
                    //对应课本情况2
                    z = z->p;
                    left_rotate(T, z);
                //转换为情况3进行处理
                z->p->color = black;
                z->p->p->color = red;
            }
        }
    }
}
```

```

        right_rotate(T, z->p->p);
    }
}
else { ... }

T->root->color = black;
while (z != T->nil)
    //维护插入结点到根路径上的各结点的max
    maintain(T, z);
    z = z->p;
}

void rb_insert(inttree* T, intnode* z)
//红黑树插入
intnode* y = T->nil;
intnode* x = T->root;
while (x != T->nil)
    //按二叉搜索树规律找到插入位置
    x->max = (x->max > z->max ? x->max : z->max); //由于在插入的第一阶段要对每一个祖先修改，且只要将z.max与x.max比较即可
    y = x;
    if (z->inter.low < x->inter.low)
        x = x->left;
    else
        x = x->right;
}
//插入并着色
z->p = y;
if (y == T->nil) //空树
    T->root = z;
else if (z->inter.low < y->inter.low)
    y->left = z;
else
    y->right = z;
z->left = T->nil;
z->right = T->nil;
z->color = red;
maintain(T, z); //对插入的zmax进行赋值
rb_insert_fixup(T, z); //恢复红黑性质
}

void rb_transplant(inttree* T, intnode* u, intnode* v) { ... }

void rb_delete_fixup(inttree* T, intnode* x) { ... }

intnode* tree_minimum(inttree* T, intnode* z) { ... }

void rb_delete(inttree* T, intnode* z) { ... }

void inorder_tree_walk(inttree* T, intnode* x, FILE* fp) { ... }

```

```

void check(inttree* T, intnode* x)
{
    //通过中序遍历各结点并检测其max值是否符合规律
    if (x != T->nil)
    {
        check(T, x->left);
        if (x->max != max(x->inter.high, x->left->max, x->right->max))
        {
            printf("key %d error", x->inter.low);
        }
        check(T, x->right);
    }
}

int overlap(interval x, interval y)
{
    //检测区间x与y是否重叠
    if (x.low <= y.high && x.high >= y.low)
        return 1;
    else
        return 0;
}

void interval_search(inttree* T, interval i, FILE* fp)
{
    //搜索T中与区间i重叠的结点
    intnode* x = T->root;
    while (x != T->nil && !overlap(i, x->inter))
    {
        //利用max进行搜索
        if (x->left != T->nil && x->left->max >= i.low)
            x = x->left;
        else
            x = x->right;
    }
    //输出结果
    if (x == T->nil)
        fprintf(fp, "Null\n");
    else
        fprintf(fp, "%d %d\n", x->inter.low, x->inter.high);
}

intnode* search(inttree* T, int key, intnode* x)
{
    //查找一个inter.low为key的结点
    if (x == T->nil || key == x->inter.low)
        return x;
    if (key < x->inter.low)
        return search(T, key, x->left);
    else
        return search(T, key, x->right);
}

```

图 10 区间树具体代码实现

对于红黑树，由于需要计时，且在最初几次运行时，包括编译、地址申请等因素在内，耗时不稳定，故重复多次运行取稳定的结果，以增加结果的可靠性。而对于区间树，运行一次即可。最终获得两个实验的输出结果，对于实验结果进行比较分析并撰写实验报告。

#### 四、实验结果与分析：

##### 1) 红黑树

通过运行代码，获得 inorder、time1、delete\_data、time2 四个文件，通过 IDE 的调试检查生成的红黑树以及运行的输出结果，发现结果正确，成功实现了插入删除构建红黑树的要求。

对于 5 个规模，构建好的红黑树中序遍历如下，可见中序遍历序列符合二叉搜索树规则

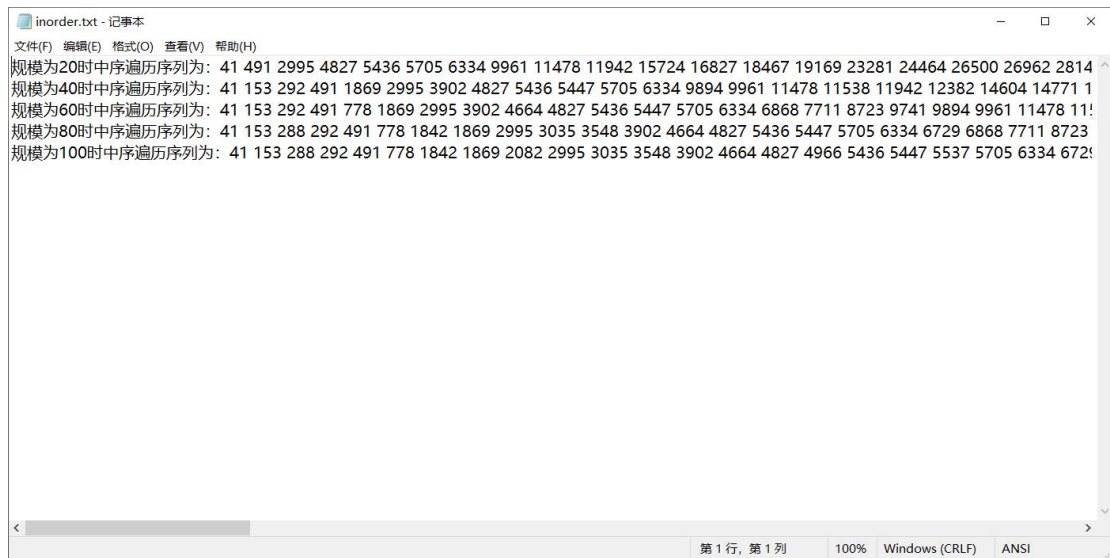


图 11 构建红黑树结果中序遍历

其运行时间如下图

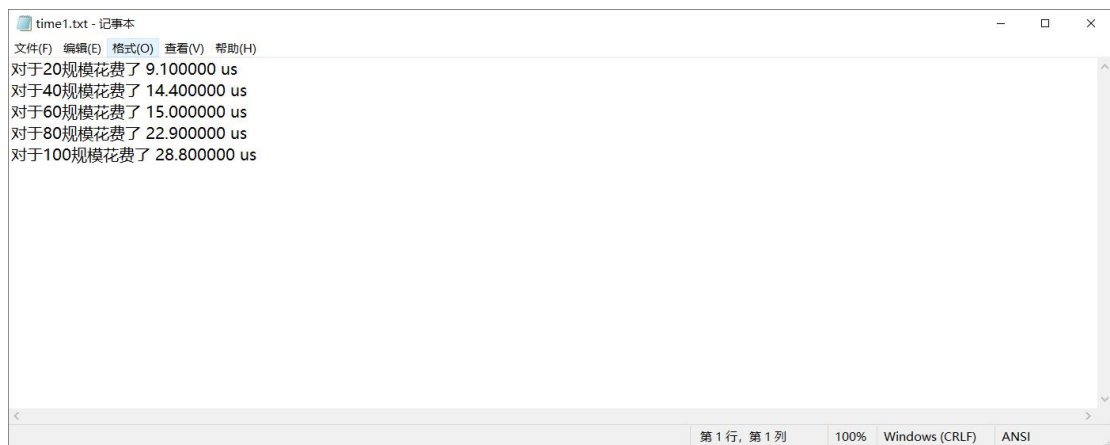


图 12 构建红黑树插入操作时间

根据上述构建红黑树时的插入操作所花费时间可以获得如下所示的运行时间-规模曲线图

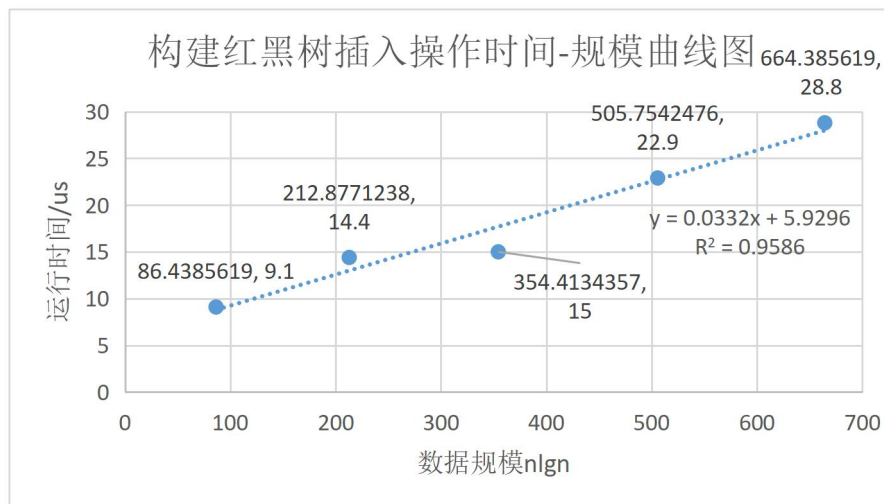
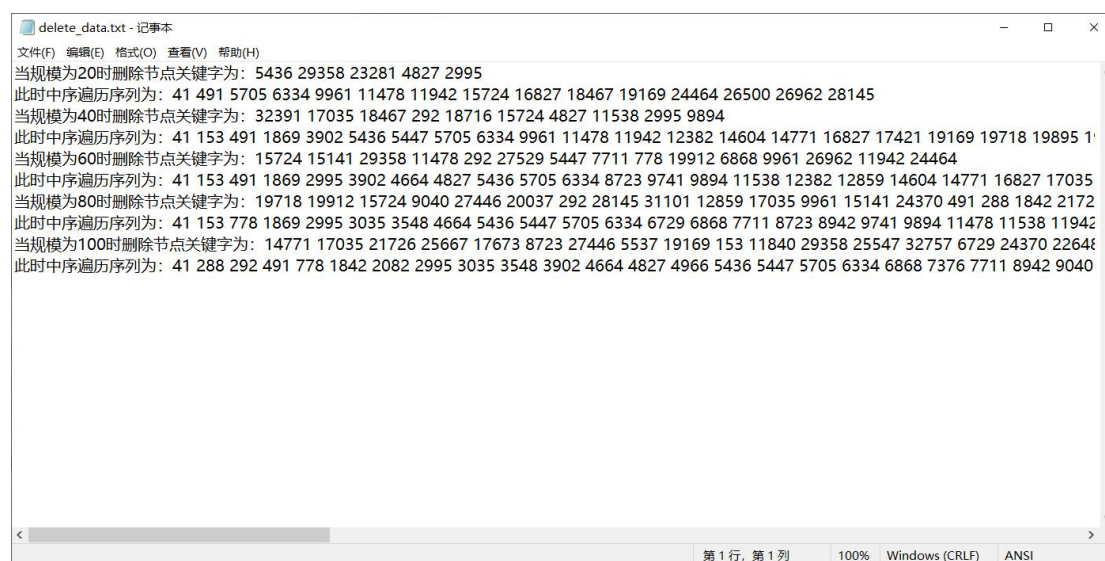


图 13 构建红黑树插入操作时间-规模曲线图

其中横坐标为数据规模  $n\lg n$ ，纵坐标为运行时间单位  $\mu s$ 。通过程序拟合，发现其运行时间与规模的关系符合表达式  $t=0.0332n\lg n+5.9296$  的函数，该函数为  $O(n\lg n)$ 。其中  $R^2=0.9586$ ，说明拟合程度一般，拟合结果不够精确。通过课本查询知，红黑树的插入操作插入一个节点运行时间为  $O(\lg n)$ ，由于我们将从一棵空树逐渐插入  $n$  个结点构成一棵红黑树，故总的运行时间应为  $O(\lg 1+\lg 2+\dots+\lg(n-1))=O(\lg((n-1)!))=O(n\lg n)$ ，实际复杂度与理论复杂度相同，运行结果较为准确。

此处的  $R^2$  数值不够大、对理论复杂度符合程度不够高，主要有以下几方面原因，第一点，我们获得的运行时间-规模关系对较少，导致拟合不够准确；第二点，理论的时间复杂度是针对数据规模  $n$  无穷大时，由此可以推测，当数据规模很大时，运行时间应当符合  $O(n\lg n)$ ，而在实验中由于数据规模很小，各低阶项常数项对于运行时间贡献更大，这从拟合的函数  $n\lg n$  系数仅为  $0.0332$  也可体现出，从而导致不够符合理论复杂度；最后一点，可以看出对于  $n=60$  时时间偏差最大，有可能存在其他程序干扰，导致计算机运算能力不稳定造成的。

对于五个规模的红黑树，删除操作所删除的结点及删除后的中序遍历序列如下图所示，可见删除后的中序遍历序列仍符合二叉搜索树的遍历规则



```
delete_data.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
当规模为20时删除结点关键字为: 5436 29358 23281 4827 2995
此时中序遍历序列为: 41 491 5705 6334 9961 11478 11942 15724 16827 18467 19169 24464 26500 26962 28145
当规模为40时删除结点关键字为: 32391 17035 18467 292 18716 15724 4827 11538 2995 9894
此时中序遍历序列为: 41 153 491 1869 3902 5436 5447 5705 6334 9961 11478 11942 12382 14604 14771 16827 17421 19169 19718 19895 1
当规模为60时删除结点关键字为: 15724 15141 29358 11478 292 27529 5447 7711 778 19912 6868 9961 26962 11942 24464
此时中序遍历序列为: 41 153 491 1869 2995 3902 4664 4827 5436 5705 6334 8723 9741 9894 11538 12382 12859 14604 14771 16827 17035
当规模为80时删除结点关键字为: 19718 19912 15724 9040 27446 20037 292 28145 31101 12859 17035 9961 15141 24370 491 288 1842 2172
此时中序遍历序列为: 41 153 778 1869 2995 3035 3548 4664 5436 5447 5705 6334 6729 6868 7711 8723 8942 9741 9894 11478 11538 11942
当规模为100时删除结点关键字为: 14771 17035 21726 25667 17673 8723 27446 5537 19169 153 11840 29358 25547 32757 6729 24370 22648
此时中序遍历序列为: 41 288 292 491 778 1842 2082 2995 3035 3548 3902 4664 4827 4966 5436 5447 5705 6334 6868 7376 7711 8942 9040
```

图 14 删除及删除后结果示意图

其运行时间如下图



图 15 删除操作运行时间

根据上述红黑树删除操作所花费时间可以获得如下所示的运行时间-规模曲线图

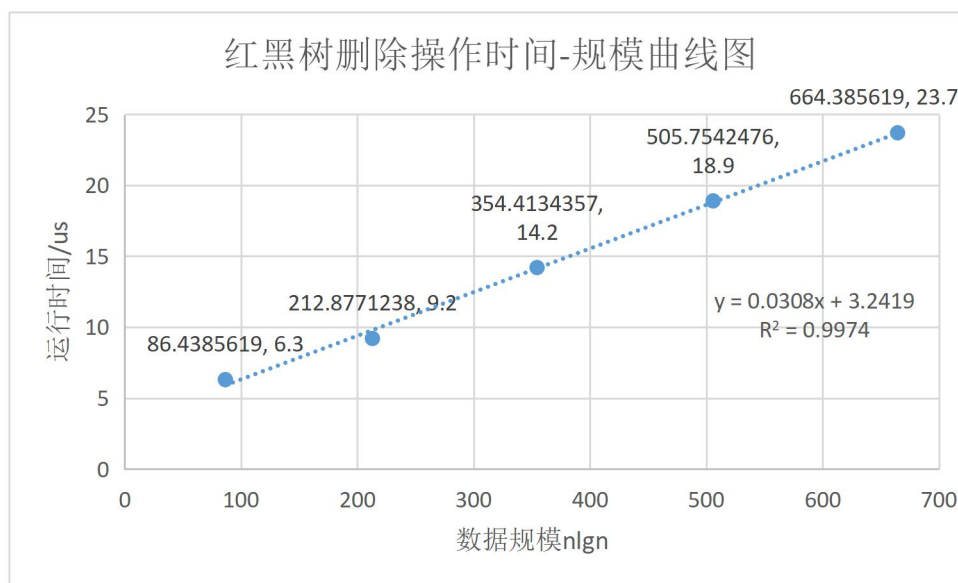


图 16 删除操作运行时间-规模曲线图

其中横坐标为数据规模  $nlg n$ ，纵坐标为运行时间单位  $us$ 。通过程序拟合，发现其运行时间与规模的关系符合表达式  $t=0.0308nlg n+3.2419$  的函数，该函数为  $O(nlg n)$ 。其中  $R^2=0.9974$ ，说明拟合程度很高，拟合结果较为精确。通过课本查知，红黑树的删除操作，删除一个结点运行时间为  $O(lg n)$ ，由于我们对于不同规模为  $n$  的数据删除掉  $n/4$  个结点，相较于原树的规模， $n/4$  个结点导致红黑树规模减小不可忽略，故总的运行时间应为  $O(lg n + lg(n-1) + \dots + lg(3n/4+1)) = O(lg(n!)) - O(lg((3n/4)!)) = O(lg(n!)) = O(nlg n)$ ，实际复杂度与理论复杂度相同，运行结果很准确，各数据都很好符合了理论复杂度，偏差很小。



## 2) 区间树

通过运行代码获得 `inorder.txt`、`delete_data.txt` 和 `search.txt`，经过检查各文件中的输出结果以及利用 `check` 函数去检查每一个结点 `max` 值，发现结果正确，成功实现了区间树的插入、删除、中序遍历、查找算法。

对于这 30 个区间插入后获得的区间树中序遍历序列结果如下，可以看出，其中序遍历的结果符合区间左端点从小到大的顺序，这也符合以左端点为 `key` 值二叉搜索树的中序遍历规律，`max` 值通过 `check` 检查正确。

```
inorder.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
0 3
1 25
2 23
3 4
4 16
5 22
7 13
8 18
12 23
13 24
14 15
15 21
16 24
17 23
18 23
22 25
30 39
31 47
32 38
33 43
37 43
39 44
40 41
41 43
42 50
43 48
44 48
45 48
48 50
49 50
第 1 行, 第 1 列 100% Windows (CRLF) UTF-8
```

图 17 区间树构建结果

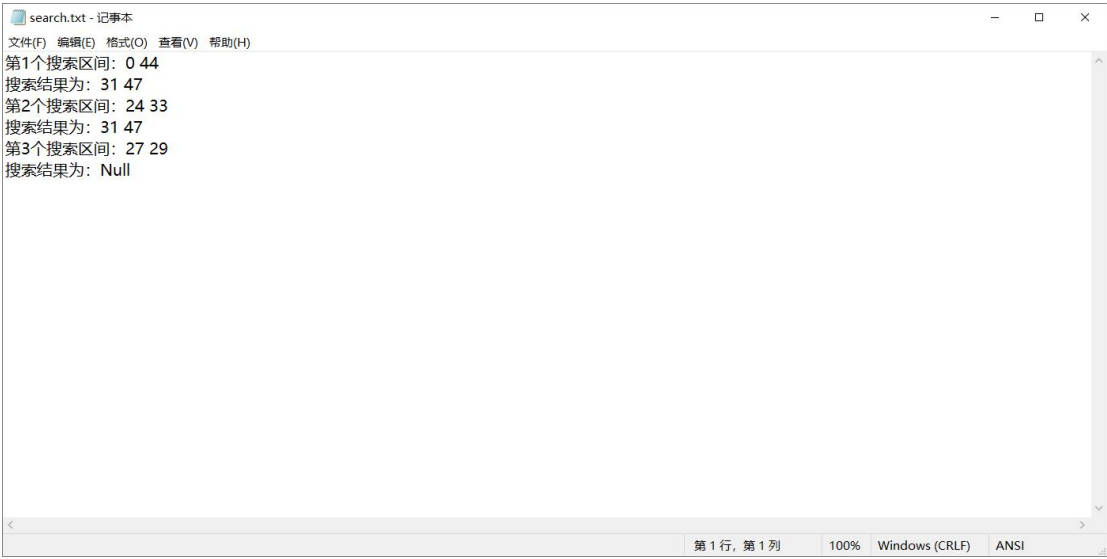
删除的三个随机的区间以及删除后的中序遍历结果如下，同理，通过检查序列以及 `max` 值结果正确。相较于构建时的中序遍历序列，不存在的区间恰好为删除的三组数据。

```
delete_data.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
删除的数据为:
45 48
16 16
30 39
中序遍历序列为:
0 3
1 25
2 23
3 4
5 22
7 13
8 18
12 23
13 24
14 15
15 21
16 24
17 23
18 23
22 25
31 47
32 38
33 43
37 43
39 44
40 41
41 43
42 50
43 48
44 48
48 50
49 50
第 1 行, 第 1 列 100% Windows (CRLF) UTF-8
```

图 18 删除节点及结果

三个随机生成的搜索区间及其搜索结果如下，其中前两个搜索区间的结果均

与该区间重叠符合要求，第三个搜索区间根据定义知必定不存在一个区间与之重接，故搜索结果为 **Null**，也符合要求。



```
search.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
第1个搜索区间: 0 44
搜索结果为: 31 47
第2个搜索区间: 24 33
搜索结果为: 31 47
第3个搜索区间: 27 29
搜索结果为: Null

第 1 行, 第 1 列 100% Windows (CRLF) ANSI
```

图 19 搜索结果

五、反省与总结：

本次实验中，复习了红黑树以及区间树的相关知识，并在实际编程中，对于这两种数据结构及其相应操作进行了运用与实践，加深了对于课本相关知识的理解。在红黑树实验中，通过对于伪代码的阅读及翻译为 C 语言代码，并参考课本的文字说明，对于红黑树的掌握有了很大的帮助；在区间树实验中，最初完全不知晓数据扩张应该在哪里对 **max** 值进行维护，通过参考网上的一些资料，并详细阅读红黑树扩张定理的证明，详细理解了对于插入删除的操作与恢复阶段，分别对于 **max** 值应当如何处理，更加具体实际的掌握了数据扩张的方法。此外，通过实验过程中的学习，加深了对于树、递归与迭代转换等基础知识的掌握。

在实验过程中，出现了由于对于红黑树边界问题、**T.nil** 理解不足导致出现了各项操作中访问不存在结点的情况，通过与伪代码比较、查找相关文字说明等方式解决。此外，还出现了由于条件判断导致的循环不合理终止的问题，最终通过反复检查以及利用断点 **debug** 等方式解决。在今后的实验中，应当更加细心认真，避免出现马虎导致的错误，此外，对于不了解不清楚的知识应当抓紧时间学习巩固。