中国科学技术大学计算机学院

算法基础实验报告

实验三

红黑树和区间树

学　　号：　**PB18000203**

姓　　名：　　汪洪韬

专　　业：计算机科学与技术

指导老师：　　顾乃杰

中国科学技术大学计算机学院

2020 年 12 月 17 日

一、 实验内容

1. 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。

2. 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。

二、 实验设备和环境

1.实验设备：PC 机；

2.实验环境：Visual Studio 2019

三、　实验方法和步骤

1. 红黑树

红黑树的左旋（右旋类似）

```c
void LEFT_ROTATE(RBRoot* T, RBTree* s) {//left rotate in center of s
    RBTree* y = s->right;
    s->right = y->left;
    if (y->left != NIL)
        y->left->p = s;
    y->p = s->p;
    if (s->p == NIL)
        T->node = y;//if s's parent is NIL, set y as a root
    else {
        if (s->p->left == s)
            s->p->left = y;
        else
            s->p->right = y;
    }
    y->left = s;
    s->p = y;
}
```

红黑树的构建与插入

```c
void RB_insert(RBRoot* T, RBTree *s) {
    RBTree* x = T -> node, * y = NIL;
    while (x != NIL) {//find the insert position
        y = x;
        if (s->key < x->key)
            x = x->left;
        else
            x = x->right;
    }
    s->p = y;
    if (y == NIL)//if y is NULL, set s as a root node
```

```c
        T->node = s;
    else if (s->key < y->key)
        y->left = s;
    else
        y->right = s;
    RB_insert_fix(T, s);//fixup the color
}
```

```c
void RB_insert_fix(RBRoot* T, RBTree* s) {//fixup the color after insert a new node
    RBTree *y, *p, *gp;
    while ((p = s->p) && p->color == RED) {
        gp = p->p;
        if (p == gp->left) {//parent node is the leftchild of grandparent node
            {
                RBTree* u = gp->right;
                if (u && u->color == RED) {//case1: uncle node is red
                    u->color = BLACK;
                    p->color = BLACK;
                    gp->color = RED;
                    s = gp;
                    continue;
                }
            }
            if (p->right == s) {//case2: uncle node is black, s is the rightchild
                RBTree* t;
                LEFT_ROTATE(T, p);
                t = p;
                p = s;
                s = t;
            }
            //case3: uncle is black, s is the leftchild
            p->color = BLACK;
            gp->color = RED;
            RIGHT_ROTATE(T, gp);
        }
        else {//symmetric with the case above
```

```
            {
                RBTree* u = gp->left;
                if (u && u->color == RED) {
                    u->color = BLACK;
                    p->color = BLACK;
                    gp->color = RED;
                    s = gp;
                    continue;
                }
            }
            if (p->left == s) {
                RBTree* t;
                RIGHT_ROTATE(T, p);
                t = p;
                p = s;
                s = t;
            }
            p->color = BLACK;
            gp->color = RED;
            LEFT_ROTATE(T, gp);
        }
    }
    T->node->color = BLACK;
}
```

红黑树的删除

```
void RB_deleteVer(RBRoot *T, RBTree *s) {//delete a node
    RBTree* c, * p;
    int color;
    if ((s->left != NIL) && (s->right != NIL)) {//s has two children
        RBTree* re = s;
        re = re->right;
        while (re->left != NIL)
            re = re->left;
        if (s->p != NIL) {//s is not root
            if (s->p->left == s)
                s->p->left = re;
            else
                s->p->right = re;
        }
        else//s is root
            T->node = re;
        c = re->right;
        p = re->p;
        color = re->color;
        if (p == s)
            p = re;
        else {
            if (c) //child is not NULL
                c->p = p;
            p->left = c;
            re->right = s->right;
            s->right->p = re;
        }
        re->p = s->p;
        re->color = s->color;
        re->left = s->left;
        s->left->p = re;
        if (color == BLACK)
            RBTree_delete_fix(T, c, p);
        free(s);
        return;
    }
    if (s->left != NIL)
        c = s->left;
    else
        c = s->right;
    p = s->p;
    color = s->color;
    if (c)
        c->p = p;
    if (p) {
        if (p->left == s)
            p->left = c;
        else
            p->right = c;
    }
```

```c
    else
        T->node = c;
    if (color == BLACK)
        RBTree_delete_fix(T, c, p);
    free(s);
}
```

```c
void RBTree_delete_fix(RBRoot *T,RBTree *s,RBTree *p) {//fixup the color after delete
    RBTree *x;
    while ((!s || s->color == BLACK) && (s != T->node)) {
        if (p->left == s) {
            x = p->right;
            if (x->color == RED) {//case1: s's brother is red
                x->color = BLACK;
                p->color = RED;
                LEFT_ROTATE(T, p);
                x = p->right;
            }
            if ((!x->left || x->left->color == BLACK) && (!x->right || x->right->color == BLACK)) {
                //case2: s's brother w is black, and w's two children is black
                x->color = RED;
                s = p;
                p = s->p;
            }
            else {
                if (!x->right || x->right->color == BLACK) {
                    //case3: s's brother is black,and w's leftchild is red, rightchild is black
                    x->left->color = BLACK;
                    x->color = RED;
                    RIGHT_ROTATE(T, x);
                    x = p->right;
                }
                //case4: s's brother is black,and w's rightchild is red
                x->color = p->color;
                p->color = BLACK;
                x->right->color = BLACK;
                LEFT_ROTATE(T, p);
                s = T->node;
                break;
            }
        }
        else {//symmetric with the case above
            x = p->left;
            if (x->color == RED) {
                x->color = BLACK;
                p->color = RED;
                RIGHT_ROTATE(T, p);
                x = p->left;
            }
            if ((!x->left || x->left->color == BLACK) && (!x->right || x->right->color == BLACK)) {
                x->color = RED;
                s = p;
                p = s->p;
            }
            else {
                if (!x->left || x->left->color == BLACK) {
                    x->right->color = BLACK;
                    x->color = RED;
                    LEFT_ROTATE(T, x);
                    x = p->left;
                }
                x->color = p->color;
                p->color = BLACK;
                x->left->color = BLACK;
                RIGHT_ROTATE(T, p);
                s = T->node;
                break;
            }
        }
    }
    if (s)
        s->color = BLACK;
}
```

中序打印节点

```c
void RB_print(Tree T, FILE* fp) {//print RBTree's nodes in midorder
    if (T != NIL) {
        RB_print(T->left, fp);
        fprintf(fp, "%2d ", T->key);//print key
        if (T->color == RED)//print color
            fprintf(fp, "RED\n");
        else
            fprintf(fp, "BLACK\n");
        RB_print(T->right, fp);
    }
}
```

主函数

主函数采用随机生成不同数据规模（20，40，60，80，100）的节点信息的方式生成红黑树，并将随机生成的过程重复多次（100 次），以期望获得更好的运行时花费时间。主函数具体略。

2. 区间树

上述操作与红黑树类似，只是加入了 max 域的维护操作以及搜索。

Max 域的维护

```c
void IN_max_fix(INRoot* T, INTree *s) {//fix the max after delete a node
    while (s != NIL) {
        s->max = MAX2(s->left->max, s->right->max);
        s = s->p;
    }
}
```

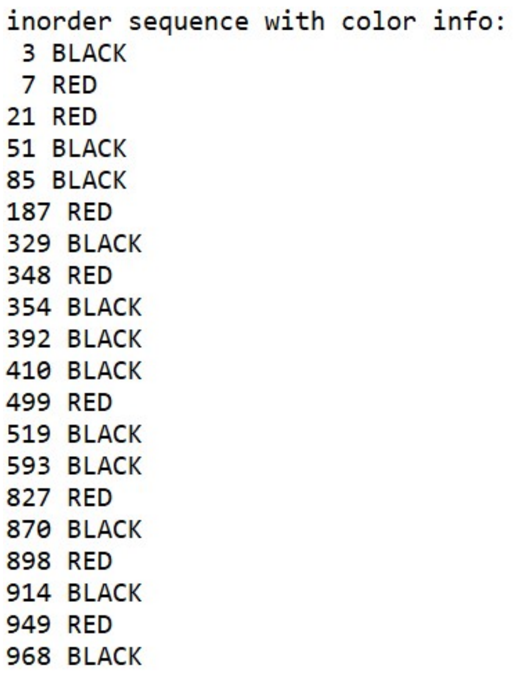
区间树的搜索

```c
void IN_search(INRoot* T,int *l, int *h, FILE *fp) {
    INTree* x = T->node;
    int i;
    for (i = 0; i < 3; i++) {
        x = T->node;
        while (x != NIL && (l[i] > x->high || h[i] < x->key)) {
            if (x->left != NIL && x->left->max >= l[i])
                x = x->left;
            else
                x = x->right;
        }
        if (x == NIL)
            fprintf(fp, "[%d, %d]is not found!\n", l[i], h[i]);
        else
            fprintf(fp, "[%d, %d] is overlaped by [%d, %d]\n", l[i], h[i], x->key, x->high);
    }
}
```

四、 实验结果与分析

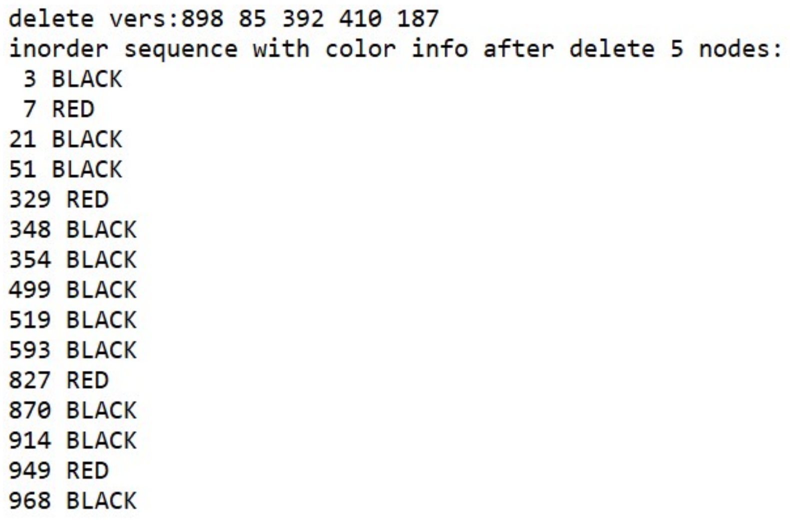
1. 红黑树的构建结果与删除结果（以数据规模为 20 时为例）

构建结果：



```
inorder sequence with color info:
 3 BLACK
 7 RED
21 RED
51 BLACK
85 BLACK
187 RED
329 BLACK
348 RED
354 BLACK
392 BLACK
410 BLACK
499 RED
519 BLACK
593 BLACK
827 RED
870 BLACK
898 RED
914 BLACK
949 RED
968 BLACK
```

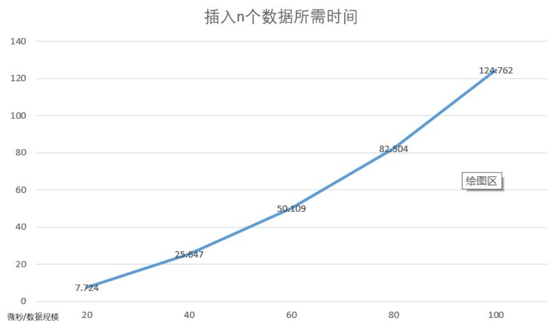
删除结果：

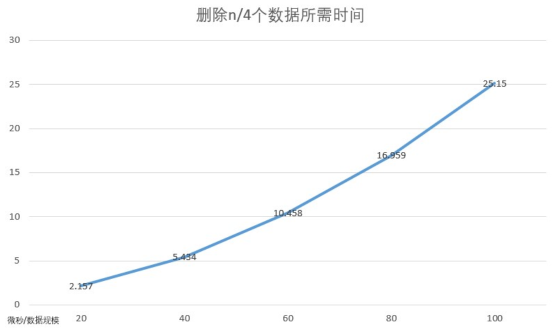


```
delete vers:898 85 392 410 187
inorder sequence with color info after delete 5 nodes:
 3 BLACK
 7 RED
21 BLACK
51 BLACK
329 RED
348 BLACK
354 BLACK
499 BLACK
519 BLACK
593 BLACK
827 RED
870 BLACK
914 BLACK
949 RED
968 BLACK
```

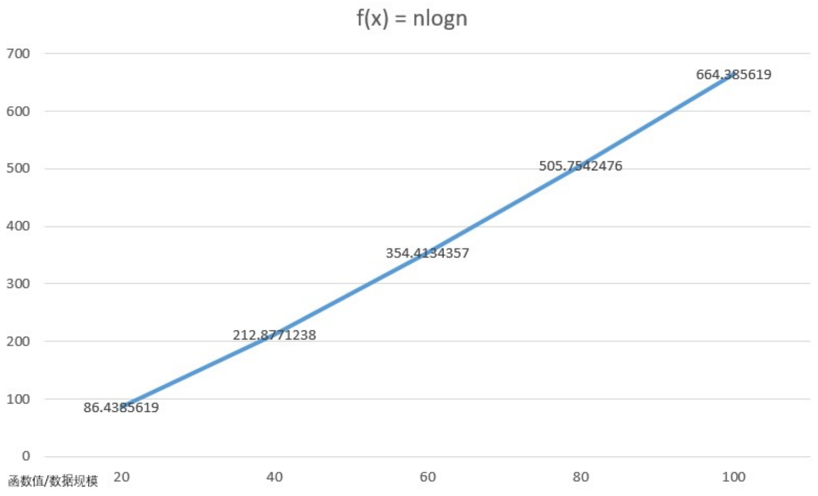
2. 红黑树操作的时间复杂度分析

具体输出见 output 文件夹下的 time1.txt 与 time2.txt 文件，下面根据所得到的时间信息来分析时间复杂度是否正确。



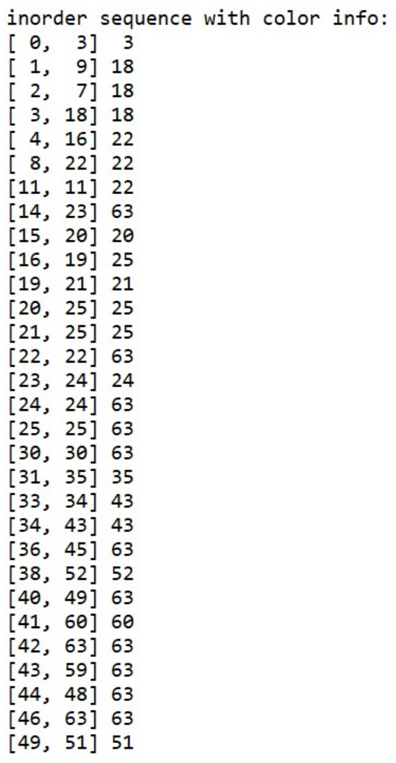


与所期望的时间复杂度 O(nlogn)相比基本拟合。（不是非常拟合的原因可能是在较小数据规模下，一些在较大数据规模可忽略的操作也会对操作花费时间有较大影响，导致操作总的花费时间会大于其理论值，导致曲线前半部分曲率较低）



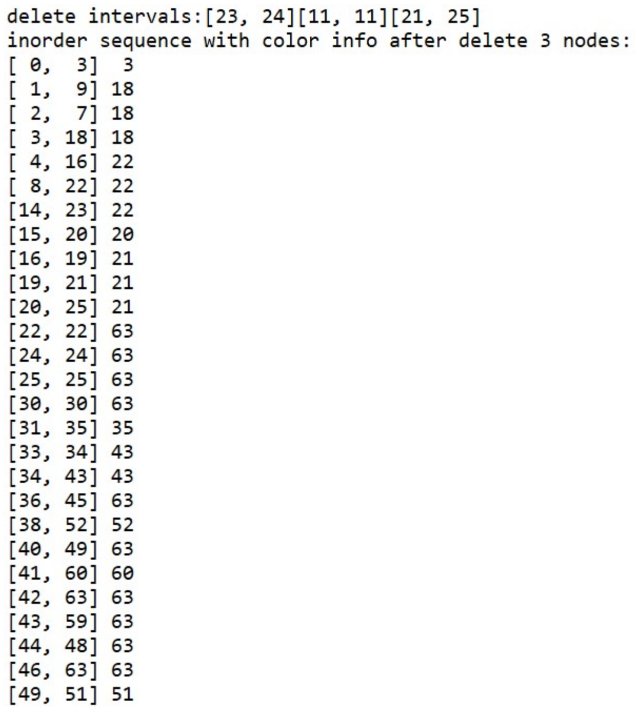
3. 区间树的构建、删除与搜索

构建结果：



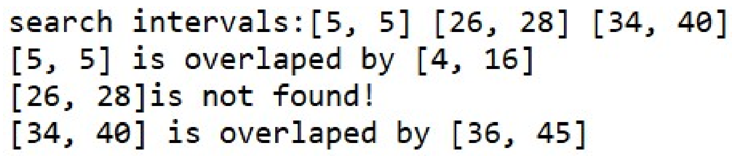
```
inorder sequence with color info:
[ 0,  3]  3
[ 1,  9] 18
[ 2,  7] 18
[ 3, 18] 18
[ 4, 16] 22
[ 8, 22] 22
[11, 11] 22
[14, 23] 63
[15, 20] 20
[16, 19] 25
[19, 21] 21
[20, 25] 25
[21, 25] 25
[22, 22] 63
[23, 24] 24
[24, 24] 63
[25, 25] 63
[30, 30] 63
[31, 35] 35
[33, 34] 43
[34, 43] 43
[36, 45] 63
[38, 52] 52
[40, 49] 63
[41, 60] 60
[42, 63] 63
[43, 59] 63
[44, 48] 63
[46, 63] 63
[49, 51] 51
```

删除结果：



```
delete intervals:[23, 24][11, 11][21, 25]
inorder sequence with color info after delete 3 nodes:
[ 0,  3]  3
[ 1,  9] 18
[ 2,  7] 18
[ 3, 18] 18
[ 4, 16] 22
[ 8, 22] 22
[14, 23] 22
[15, 20] 20
[16, 19] 21
[19, 21] 21
[20, 25] 21
[22, 22] 63
[24, 24] 63
[25, 25] 63
[30, 30] 63
[31, 35] 35
[33, 34] 43
[34, 43] 43
[36, 45] 63
[38, 52] 52
[40, 49] 63
[41, 60] 60
[42, 63] 63
[43, 59] 63
[44, 48] 63
[46, 63] 63
[49, 51] 51
```

删除后的搜索结果：



```
search intervals:[5, 5] [26, 28] [34, 40]
[5, 5] is overlaped by [4, 16]
[26, 28]is not found!
[34, 40] is overlaped by [36, 45]
```