《算法设计与分析》上机报告

姓名：	廖洲洲	学号：	PB17081504	日期：	2019.11.30
上机题目：	红黑树维护算法及其区间树应用				

实验环境：
CPU：Intel Core i7-8550U；内存:8G；操作系统：Win 10；
软件平台: JetBrains CLion；

一、算法设计与分析：

题目一:

实现红黑树的基本算法，对于 n=20,40,60,80 分别生成 n 个随机正整数，将 n 个正整数作为关键字当作红黑树的结点，从空树依次插入 n 个节点。对于生成的红黑树，找出树中第 n/4 和 n/2 小的节点，并删除。

(一)算法思想

1. 左、右旋转中二叉树的性质不变
2. 插入算法步骤
   a) 将 z 节点按 BST 树规则插入红黑树中，z 是叶子节点
   b) 将 z 涂红
   c) 调整使其满足红黑树的性质
3. 插入的调整算法：通过旋转和改变颜色，自下而上调整（z 进行上溯），使树满足红黑树
   a) 若 z 为根，将其涂黑
   b) 若 z 为非根，则 p[z]存在
      i. 若 p[z]为黑，无需调整
      ii. 若 p[z]为红，违反性质 4，则需调整（具体调整算法见书 P181
4. 删除算法
   a) z 为叶子
   b) z 只有一个孩子(非空)
   c) z 的两个孩子均非空（具体情况讨论见书 P184）
5. 删除后的调整算法
   a) 若 x 是根，直接移去多余一层黑色(树黑高减 1)，终止
   b) 若 x 原为红，将 y 的黑色涂到 x 上，终止
   c) 若 x 非根节点，且为黑色，则 x 为双黑。通过变色、旋转使多余黑色向上传播，直到某个红色节点或传到根（具体情况分析见书 P186）
6. 查找第 n 小的节点
   a) 采用中序遍历的思想
   b) 每次从左子树中调用中序遍历返回后 n 减 1，当 n 减为 0 时说明当前节点即为要查找的节点
   c) 找到要查找的节点后结束递归，返回查找的节点即可

题目二：

实现区间树的基本算法，随机取 n=100 个 0~500 间随机正整数，将其作为关键字，从空树插入区间树的节点。对于生成的区间树，找出树中第 n/4 和 n/2 小的节点，并删除。

(一)算法思想

1. 基本结构
   以红黑树为基础，对树中节点 x，x 包含区间 int[x]的信息(低点和高点)，其中 key=low[int[x]]。
2. 附加信息
   每个结点 x 中除了自身区间信息外，还包含一个值 x.max，它是以 x 为根的子树中所有区间端点的最大值。
3. 对信息的维护
   max[x]=max(high[int[x]], max[left[x]], max[right[x]])
4. 对信息的维护在各算法动作中的维护细节
   a) 对左旋操作的维护：在原左旋操作最后加上
      y->max=x->max;
      x->max=mymax(x->high,x->left->max,x->right->max);
   b) 对左旋操作的维护：在原左旋操作最后加上
      x->max=y->max;
      y->max=mymax(y->high,y->left->max,y->right->max);
   c) 插入算法
      从根向下插入新节点，将搜索路径上所经历的每个节点的 max 进行更新。其中 y->max=max(y->max,z->high)。
   d) 插入调整算法
      其中变色不改变 max，只有改变树的结构的旋转操作会改变 max。故只需要在旋转操作中完成对 max 的修改即可。
   e) 删除算法
      物理上删除 y，在删除 y 时从 y 上溯至根，将所经历的节点的 max 进行更新。
      其中更新按 x->max=max(x->high,x->left->max,x->right->max)进行。
   f) 删除调整算法
      其中变色不改变 max，只有改变树的结构的旋转操作会改变 max。故只需要在旋转操作中完成对 max 的修改即可。
5. 其他操作基本与红黑树算法一致

二、核心代码：

题目一:

1. 节点域和树

```
struct node{
    int key;
    int color;//0为黑，1为红
    struct node *left;
    struct node *right;
    struct node *p;
};
struct RB_tree{
    struct node *root;
    struct node *nil;
};
```

2. 左旋（右旋与左旋完全对称）

```
int left_rotate(struct node *x){//对x左旋,x的右孩子不为nil
    struct node *y=x->right;//step1
    x->right=y->left;y->left->p=x;//step2
    y->p=x->p;//step3
    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;//step4
}
```

3. 插入算法

```
int rb_insert(struct node *z){//插入z
    struct node *y=T.nil;//y用于记录当前扫描节点的双亲节点
    struct node *x=T.root;//从根开始扫描
    while(x!=T.nil){
        y=x;
        if(z->key < x->key)
            x=x->left;
        else
            x=x->right;
    }
    z->p=y;//y是z的双亲
    if(y==T.nil)//z插入空树,z为根
        T.root=z;
    else if(z->key<y->key)//z是y的左子插入
        y->left=z;
    else//z是y的右子插入
        y->right=z;
    z->right=T.nil;z->left=T.nil;
    z->color=1;//将z涂红
    rb_insert_fixup(z);
}
```

4. 插入调整算法

```
int rb_insert_fixup(struct node *z){//调整
    struct node *y;
    while(z->p->color==1){
        //若z为根, z.p.color=black,不进入循环
        //若z父节点为黑, 无需调整, 不进入循环
        if(z->p==z->p->p->left){//case1,2,3, 双亲是祖父的左孩子
            y=z->p->p->right;//y是z的叔叔
            if(y->color==1) {//case1,z的叔叔是红色
                y->color=0;z->p->color=0;
                z->p->p->color=1;
                z=z->p->p;
            }
            else{ //case 2 or 3
                if(z==z->p->right){//case2
                    z=z->p;//上溯到双亲
                    left_rotate(z);
                }//case 3
                z->p->color=0;z->p->p->color=1;
                right_rotate(z->p->p);
            }
        }
        else{...}//对称的case4, 5, 6
    }
    T.root->color=0;
```

5. 删除算法

```
struct node* rb_delete(struct node *z){//删除z结点
    struct node *y,*x;
    if(z->left==T.nil || z->right==T.nil)//case 1,2
        y=z;
    else
        y=treesuccessor(z);//y是z的中序后继
    //此时，y统一是x的双亲节点，且是要删除的节点
    //x是待连接到y.p的节点，下面要确定x
    if(y->left != T.nil)
        x=y->left;
    else
        x=y->right;
    //以下用x取代y
    x->p=y->p;
    if(y->p==T.nil)
        T.root=x;
    else if(y==y->p->left)
        y->p->left=x;
    else
        y->p->right=x;
    if(y!=z){
        z->key=y->key;
    }
    if(y->color==0)//y是红点，删除不影响，y是黑点，需要调整
        rb_delete_fixup(x);
    return y;//返回实际删除的节点
```

6. 删除调整算法

```
int rb_delete_fixup(struct node *x){
    struct node *w;
    while(x!=T.root && x->color==0){
        if(x==x->p->left){//x是双亲的左孩子
            w=x->p->right;//w是x的兄弟
            if(w->color==1){//x的兄弟是红色,case1
                w->color=0;
                x->p->color=1;
                left_rotate(x->p);
                w=x->p->right;
            }//case1转换为case2,3,4
            if(w->left->color==0&&w->right->color==0){//case2,w两个孩子为黑
                w->color=1;
                x=x->p;
            }//x上移
            else {//case 3,4
                if(w->right->color==0){//case3,w右子为黑，左子为红
                    w->left->color=0;
                    w->color=1;
                    right_rotate(w);
                    w=x->p->right;
                }//case3转为case4
```

```
            else {//case 3,4
                if(w->right->color==0){//case3,w右子为黑，左子为红
                    w->left->color=0;
                    w->color=1;
                    right_rotate(w);
                    w=x->p->right;
                }//case3转为case4
                w->color=x->p->color;
                x->p->color=0;
                w->right->color=0;
                left_rotate(x->p);
                x=T.root;
            }
        }
        else{//x是双亲的右孩子
...
    }//x是双亲的右孩子
    x->color=0;
}
```

7. 查找算法

```
struct node* rb_search_order(struct node *x,int *n,struct node **p){//查找第n小的节点,返回节点p
    //注意：在这里进行节点的保存时不能使用*p，而是要使用**p，因为传进函数的是一个临时指针变量，仅能对指针指向的内容进行修改，
    //对该指针进行修改没有用，在函数体外的指针不会变，故需要传进该指针变量的地址，才能保存下查找到的节点地址
    if(x!=T.nil && *n>=0){
        if(*p!=T.nil)
            return *p;
        rb_search_order(x->left,n,p);
        *n=(*n)-1;
        //printf("%d(%d) ",x->key,*n);
        if(*n==0) {
            *p = x;
        }
        rb_search_order(x->right,n,p);
    }
    return T.nil;
}
```

题目二：

1. 节点域和树

```
struct node{
    int key;//保存该节点区间的低端点
    int high;//保存该节点区间的高端点
    int max;//以该节点为根的子树中所有区间的端点的最大值
    int color;//0为黑，1为红
    struct node *left;
    struct node *right;
    struct node *p;
};
struct RB_tree{
    struct node *root;
    struct node *nil;
};
```

2. 旋转算法

```
int left_rotate(struct node *x){//对x左旋,x的右孩子不为nil
    struct node *y=x->right;//step1
    x->right=y->left;y->left->p=x;//step2
    y->p=x->p;//step3
    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;//step4
    y->max=x->max;
    x->max=mymax(x->high,x->left->max,x->right->max);
}
```

3. 插入算法

```
int rb_insert(struct node *z){//插入z，在往下寻找的时候修改max
    struct node *y=T.nil;//y用于记录当前扫描节点的双亲节点
    struct node *x=T.root;//从根开始扫描
    while(x!=T.nil){//在查找插入的位置时，对节点的max进行修改
        y=x;
        y->max=mymax2(y->max,z->high);//z要插入以y为根的子树，修改其max
        if(z->key < x->key)
            x=x->left;
        else
            x=x->right;
    }
    z->p=y;//y是z的双亲
    if(y==T.nil) {//z插入空树,z为根
        T.root = z;
        y->max=y->high;//插入空树修改区间大小为区间高端点
    }
    else if(z->key<y->key)//z是y的左子插入
        y->left=z;
    else//z是y的右子插入
        y->right=z;
    z->right=T.nil;z->left=T.nil;
    z->color=1;//将z涂红
    rb_insert_fixup(z);
}
```

4. 插入调整算法不用修改
5. 删除算法
6. 在原算法基础上增加从底向上修改 max

```
    x=x->p;
    while(x!=T.nil){//从x向上修改
        x->max=mymax(x->high,x->left->max,x->right->max);
        x=x->p;
    }
```

7. 其他算法基本不变

三、结果与分析：

题目一:

```
 Random number:75 52 63 89 80 62 46 90 63 18 53 23 21 61 14 55 88 30 30 79
inorder walk:14 18 21 23 30 30 46 52 53 55 61 62 63 63 75 79 80 88 89 90
Search:5th lowest number:30
10th lowest number:55
inorder walk:14 18 21 23 30 46 52 53 61 62 63 63 75 79 80 88 89 90
```

```
 Random number:8 27 53 17 54 21 44 93 76 9 69 96 0 53 36 24 81 30 55 42 39 56 43 52
 62 7 37 66 51 68 61 88 38 89 77 25 82 61 53 88
inorder walk:0 7 8 9 17 21 24 25 27 30 36 37 38 39 42 43 44 51 52 53 53 53 54 55 56 61 61 62 66 68 69 76 77 81 82 88 88
89 93 96
Search:10th lowest number:30
20th lowest number:53
inorder walk:0 7 8 9 17 21 24 25 27 36 37 38 39 42 43 44 51 52 53 53 54 55 56 61 61 62 66 68 69 76 77 81 82 88 88 89 93
96
```

分析：

1. 对于插入操作
   - 调整算法的时间：O(logn)
   - 整个插入算法的时间：O(logn)
   - 调整算法中至多使用 2 个旋转
2. 对于删除操作
   - 因为含有 n 个节点的红黑树的高度为 O(logn),故不调用调整操作时时间为 O(logn)。
   - 调整操作时间为 O(logn)。
   - 调整操作最多做 3 次旋转
3. 总得来说红黑树在最坏情况下基本动态集合操作的时间复杂度为 O(logn)。

题目二：

```
Please input the number of the nodes:8
 Random number:[244,399] [400,430] [205,487] [297,315] [408,493] [347,459] [63,298]
 [27,321]
inorder walk:[27,321](321) [63,298](487) [205,487](487) [244,399](493) [297,315](459) [347,459](459) [400,430](493) [408
,493](493)
Search:2th lowest number:[63,298]
4th lowest number:[244,399]
inorder walk:[27,321](487) [205,487](487) [297,315](493) [347,459](459) [400,430](493) [408,493](493)
```

```
 Please input the number of the nodes:20
  Random number:[218,472] [167,444] [314,367] [386,481] [141,390] [272,287] [254,433
 ] [461,485] [60,279] [288,301] [92,110] [137,339] [431,471] [256,496] [462,472] [16,106] [60,87] [471,495] [298,402] [71
 ,94]
 inorder walk:[16,106](106) [60,279](279) [60,87](94) [71,94](94) [92,110](444) [137,339](339) [141,390](444) [167,444](4
 44) [218,472](496) [254,433](496) [256,496](496) [272,287](496) [288,301](402) [298,402](402) [314,367](496) [386,481](4
 81) [431,471](495) [461,485](485) [462,472](495) [471,495](495)
 Search:5th lowest number:[92,110]
 10th lowest number:[254,433]
 inorder walk:[16,106](106) [60,279](279) [60,87](94) [71,94](94) [137,339](444) [141,390](390) [167,444](444) [218,472](
 496) [256,496](496) [272,287](496) [288,301](402) [298,402](402) [314,367](496) [386,481](481) [431,471](495) [461,485](
 485) [462,472](495) [471,495](495)
```

分析：

1. 我们在插入和删除下对附加信息的维护均是有效维护，有效维护保证扩充前后的基本操作的渐近时间不变。
2. 因此区间树在最坏情况下基本动态集合操作的时间复杂度也为 O(logn)。

四、备注：

有可能影响结论的因素：

```
struct node* rb_search_order(struct node *x,int *n,struct node **p){//查找第n小的节点,返回节点p
    //注意：在这里进行节点的保存时不能使用*p，而是要使用**p，因为传进函数的是一个临时指针变量，仅能对指针指向的内容进行修改，
    //对该指针进行修改没有用，在函数体外的指针不会变，故需要传进该指针变量的地址，才能保存下查找到的节点地址
```

在进行查找操作时，希望通过一个指针变量 p 来保存查找到的节点。但是在这里进行节点的保存时不能使用 struct node *p，而是要使用**p，因为传进函数的是一个临时指针变量，仅能对指针指向的内容进行修改，对该指针进行修改没有用，在函数体外的指针不会变，故需要传进该指针变量的地址，才能修改该指针变量指向的地址，实现对查找到的节点地址的保存

总结：

1. 通过本次实验，加深了红黑树的理解。通过动手实现各个操作，发现了很多以前没有考虑到的细节。
2. 通过本次实验，对指针操作又有了进一步理解。
3. 了解了红黑树的运用和其各种优秀的性能。

附录（源代码）

算法源代码（C/C++/JAVA 描述）

1. 红黑树

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
struct node{
        int key;
        int color;//0 为黑，1 为红
        struct node *left;
        struct node *right;
        struct node *p;
};
struct RB_tree{
        struct node *root;
        struct node *nil;
};
struct RB_tree T;
int left_rotate(struct node *x){//对 x 左旋,x 的右孩子不为 nil
        struct node *y=x->right;//step1
        x->right=y->left;y->left->p=x;//step2
```

```
        y->p=x->p;//step3
        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;//step4
    }
    int right_rotate(struct node *y){//对 y 右旋,y 的左孩子不为 nil
        struct node *x=y->left;//step1
        y->left=x->right;x->right->p=y;//step2
        x->p=y->p;//step3
        if(y->p==T.nil)
            T.root=x;
        else if(y==y->p->left)
            y->p->left=x;
        else
            y->p->right=x;
        x->right=y;y->p=x;
    }
    int rb_insert_fixup(struct node *z){//调整
        struct node *y;
        while(z->p->color==1){
            //若 z 为根，z.p.color=black,不进入循环
            //若 z 父节点为黑，无需调整，不进入循环
            if(z->p==z->p->p->left){//case1,2,3，双亲是祖父的左
    孩子
                y=z->p->p->right;//y 是 z 的叔叔
                if(y->color==1) {//case1,z 的叔叔是红色
                    y->color=0;z->p->color=0;
                    z->p->p->color=1;
                    z=z->p->p;
                }
                else{ //case 2 or 3
                    if(z==z->p->right){//case2
                        z=z->p;//上溯到双亲
                        left_rotate(z);
                    }//case 3
                    z->p->color=0;z->p->p->color=1;
                    right_rotate(z->p->p);
                }
            }
            else{//case 4,5,6,双亲是祖父的右孩子
```

```
                y=z->p->p->left;//y 是 z 的叔叔
                if(y->color==1) {//case4,z 的叔叔是红色
                    y->color=0;z->p->color=0;
                    z->p->p->color=1;
                    z=z->p->p;
                }
                else{ //case 5 or 6,z 的叔叔是黑色
                    if(z==z->p->left){//case5,z 是双亲的左孩子
                        z=z->p;//上溯到双亲
                        right_rotate(z);
                    }//case 6
                    z->p->color=0;z->p->p->color=1;
                    left_rotate(z->p->p);
                }
            }//对称的 case4，5，6
        }
        T.root->color=0;
    }
    int rb_insert(struct node *z){//插入 z
        struct node *y=T.nil;//y 用于记录当前扫描节点的双亲节点
        struct node *x=T.root;//从根开始扫描
        while(x!=T.nil){
            y=x;
            if(z->key < x->key)
                x=x->left;
            else
                x=x->right;
        }
        z->p=y;//y 是 z 的双亲
        if(y==T.nil)//z 插入空树,z 为根
            T.root=z;
        else if(z->key<y->key)//z 是 y 的左子插入
            y->left=z;
        else//z 是 y 的右子插入
            y->right=z;
        z->right=T.nil;z->left=T.nil;
        z->color=1;//将 z 涂红
        rb_insert_fixup(z);
    }
    int rb_inorder(struct node *x){ //中序遍历
        if(x!=T.nil){
            rb_inorder(x->left);
            printf("%d ",x->key);
            rb_inorder(x->right);
```

```
        }
    }
    struct node *rb_search_key(struct node *x,int k){//按关键字进行查询
        if(x==T.nil || k==x->key)
            return x;
        if(k < x->key)
            return rb_search_key(x->left,k);
        else
            return rb_search_key(x->right,k);
    }
    struct node* rb_search_order(struct node *x,int *n,struct node **p){//查找第 n 小的节点,返回节点 p
        //注意：在这里进行节点的保存时不能使用*p，而是要使用**p，因为传进函数的是一个临时指针变量，仅能对指针指向的内容进行修改，
        //对该指针进行修改没有用，在函数体外的指针不会变，故需要传进该指针变量的地址，才能保存下查找到的节点地址
        if(x!=T.nil && *n>=0){
            if(*p!=T.nil)
                return *p;
            rb_search_order(x->left,n,p);
            *n=(*n)-1;
            //printf("%d(%d) ",x->key,*n);
            if(*n==0) {
                *p = x;
            }
            rb_search_order(x->right,n,p);
        }
        return T.nil;
    }

    struct node *treesuccessor(struct node *x){//寻找 x 的中序后继
        if(x==T.nil) return x;
        if(x->right!=T.nil){
            x=x->right;
            while(x->left!=T.nil)
                x=x->left;
            return x;
        }
        struct node *y=x->p;
        while(y!=T.nil && x==y->right){
            x=y;
            y=y->p;
```

```
        }
        return y;
    }
    int rb_delete_fixup(struct node *x){
        struct node *w;
        while(x!=T.root && x->color==0){
            if(x==x->p->left){//x 是双亲的左孩子
                w=x->p->right;//w 是 x 的兄弟
                if(w->color==1){//x 的兄弟是红色,case1
                    w->color=0;
                    x->p->color=1;
                    left_rotate(x->p);
                    w=x->p->right;
                }//case1 转换为 case2,3,4

    if(w->left->color==0&&w->right->color==0){//case2,w  两个孩子为黑
                    w->color=1;
                    x=x->p;
                }//x 上移
                else {//case 3,4
                    if(w->right->color==0){//case3,w 右子为黑，左子为红
                        w->left->color=0;
                        w->color=1;
                        right_rotate(w);
                        w=x->p->right;
                    }//case3 转为 case4
                    w->color=x->p->color;
                    x->p->color=0;
                    w->right->color=0;
                    left_rotate(x->p);
                    x=T.root;
                }
            }
            else{//x 是双亲的右孩子
                w=x->p->left;//w 是 x 的兄弟
                if(w->color==1){//x 的兄弟是红色,case1
                    w->color=0;
                    x->p->color=1;
                    right_rotate(x->p);
                    w=x->p->left;
                }//case1 转换为 case2,3,4
```

```
if(w->left->color==0&&w->right->color==0){//case2,w 两个孩子为黑
                    w->color=1;
                    x=x->p;
                }//x 上移
                else {//case 3,4
                    if(w->left->color==0){//case3
                        w->right->color=0;
                        w->color=1;
                        left_rotate(w);
                        w=x->p->left;
                    }//case3 转为 case4
                    w->color=x->p->color;
                    x->p->color=0;
                    w->left->color=0;
                    right_rotate(x->p);
                    x=T.root;
                }
            }
        }//x 是双亲的右孩子
        x->color=0;
    }
    struct node* rb_delete(struct node *z){//删除 z 结点
        struct node *y,*x;
        if(z->left==T.nil || z->right==T.nil)//case 1,2
            y=z;
        else
            y=treesuccessor(z);//y 是 z 的中序后继
        //此时，y 统一是 x 的双亲节点，且是要删除的节点
        //x 是待连接到 y.p 的节点，下面要确定 x
        if(y->left != T.nil)
            x=y->left;
        else
            x=y->right;
        //以下用 x 取代 y
        x->p=y->p;
        if(y->p==T.nil)
            T.root=x;
        else if(y==y->p->left)
            y->p->left=x;
        else
            y->p->right=x;
        if(y!=z){
            z->key=y->key;
```

```
    }
    if(y->color==0)//y 是红点，删除不影响，y 是黑点，需要调整
        rb_delete_fixup(x);
    return y;//返回实际删除的节点
}


int main() {
    int i,num;
    struct node *p;
    struct node *x;
    srand(time(NULL));
    T.nil=(struct node *)malloc(sizeof(struct node));
    T.nil->color=0;//nil.color=black
    T.root=T.nil;
    printf("Please input the number of the nodes: ");
    scanf("%d",&num);
    printf("Random number:");
    for(i=0;i<num;i++){
        p=(struct node *)malloc(sizeof(struct node));
        p->key=rand()%100;
        printf("%d ",p->key);
        rb_insert(p);
    }
    printf("\ninorder walk:");
    rb_inorder(T.root);
    printf("\nSearch:");
    int tmp=num/4;
    p=T.nil;
    rb_search_order(T.root,&tmp,&p);
    printf("%dth lowest number:%d\n",num/4,p->key);
    tmp=num/2;
    x=T.nil;
    rb_search_order(T.root,&tmp,&x);
    printf("%dth lowest number:%d\n",num/2,x->key);
    //p=treesuccessor(p);
    //printf("%d ",p->key);
    rb_delete(p);
    rb_delete(x);
    printf("inorder walk:");
    rb_inorder(T.root);
    //p=rb_search_order(T.root,num/2);
    //rb_delete(p);
```

```
        return 0;
    }
2.  区间树
    #include <stdio.h>
    #include <stdlib.h>
    #include <time.h>
    #include <math.h>
    struct node{
        int key;//保存该节点区间的低端点
        int high;//保存该节点区间的高端点
        int  max;//以该节点为根的子树中所有区间的端点的最大值
        int color;//0 为黑，1 为红
        struct node *left;
        struct node *right;
        struct node *p;
    };
    struct RB_tree{
        struct node *root;
        struct node *nil;
    };
    struct RB_tree T;
    int mymax(int a,int b,int c){
        int max;
        if(a>b)
            max=a;
        else
            max=b;
        if(c>max)
            max=c;
        return max;
    }
    int mymax2(int a,int b){
        if(a>b)
            return a;
        else
            return b;
    }
    int left_rotate(struct node *x){//对 x 左旋,x 的右孩子不为 nil
        struct node *y=x->right;//step1
        x->right=y->left;y->left->p=x;//step2
        y->p=x->p;//step3
        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;//step4
        y->max=x->max;
        x->max=mymax(x->high,x->left->max,x->right->max);
}
int right_rotate(struct node *y){//对 y 右旋,y 的左孩子不为 nil
        struct node *x=y->left;//step1
        y->left=x->right;x->right->p=y;//step2
        x->p=y->p;//step3
        if(y->p==T.nil)
            T.root=x;
        else if(y==y->p->left)
            y->p->left=x;
        else
            y->p->right=x;
        x->right=y;y->p=x;
        x->max=y->max;
        y->max=mymax(y->high,y->left->max,y->right->max);
}
int rb_insert_fixup(struct node *z){//调整,调整中只要旋转会改
变树的结构
        struct node *y;
        while(z->p->color==1){
            //若 z 为根，z.p.color=black,不进入循环
            //若 z 父节点为黑，无需调整，不进入循环
            if(z->p==z->p->p->left){//case1,2,3，双亲是祖父的左
孩子
                y=z->p->p->right;//y 是 z 的叔叔
                if(y->color==1) {//case1,z 的叔叔是红色
                    y->color=0;z->p->color=0;
                    z->p->p->color=1;
                    z=z->p->p;
                }
                else{ //case 2 or 3
                    if(z==z->p->right){//case2
                        z=z->p;//上溯到双亲
                        left_rotate(z);
                    }//case 3
                    z->p->color=0;z->p->p->color=1;
                    right_rotate(z->p->p);
                }
```

```
            }
            else{//case 4,5,6,双亲是祖父的右孩子
                y=z->p->p->left;//y 是 z 的叔叔
                if(y->color==1) {//case4,z 的叔叔是红色
                    y->color=0;z->p->color=0;
                    z->p->p->color=1;
                    z=z->p->p;
                }
                else{ //case 5 or 6,z 的叔叔是黑色
                    if(z==z->p->left){//case5,z 是双亲的左孩子
                        z=z->p;//上溯到双亲
                        right_rotate(z);
                    }//case 6
                    z->p->color=0;z->p->p->color=1;
                    left_rotate(z->p->p);
                }
            }
        }
        T.root->color=0;
}
int rb_insert(struct node *z){//插入 z，在往下寻找的时候修改 max
        struct node *y=T.nil;//y 用于记录当前扫描节点的双亲节点
        struct node *x=T.root;//从根开始扫描
        while(x!=T.nil){//在查找插入的位置时，对节点的 max 进行修改
            y=x;
            y->max=mymax2(y->max,z->high);//z 要插入以 y 为根的子树，修改其 max
            if(z->key < x->key)
                x=x->left;
            else
                x=x->right;
        }
        z->p=y;//y 是 z 的双亲
        if(y==T.nil) {//z 插入空树,z 为根
            T.root = z;
            y->max=y->high;//插入空树修改区间大小为区间高端点
        }
        else if(z->key<y->key)//z 是 y 的左子插入
            y->left=z;
        else//z 是 y 的右子插入
            y->right=z;
```

```
        z->right=T.nil;z->left=T.nil;
        z->color=1;//将 z 涂红
        rb_insert_fixup(z);
    }
    int rb_inorder(struct node *x){ //中序遍历
        if(x!=T.nil){
            rb_inorder(x->left);
            printf("[%d,%d](%d) ",x->key,x->high,x->max);
            rb_inorder(x->right);
        }
    }
    struct node *rb_search_key(struct node *x,int k){//按关键字进行
    查询
        if(x==T.nil || k==x->key)
            return x;
        if(k < x->key)
            return rb_search_key(x->left,k);
        else
            return rb_search_key(x->right,k);
    }
    struct node* rb_search_order(struct node *x,int *n,struct node
    **p){//查找第 n 小的节点,返回节点 p
        //注意：在这里进行节点的保存时不能使用*p，而是要使
    用**p，因为传进函数的是一个临时指针变量，仅能对指针指
    向的内容进行修改，
        //对该指针进行修改没有用，在函数体外的指针不会变，
    故需要传进该指针变量的地址，才能保存下查找到的节点地址
        if(x!=T.nil && *n>=0){
            if(*p!=T.nil)
                return *p;
            rb_search_order(x->left,n,p);
            *n=(*n)-1;
            //printf("%d(%d) ",x->key,*n);
            if(*n==0) {
                *p = x;
            }
            rb_search_order(x->right,n,p);
        }
        return T.nil;
    }

    struct node *treesuccessor(struct node *x){//寻找 x 的中序后继
        if(x==T.nil) return x;
        if(x->right!=T.nil){
```

```
            x=x->right;
            while(x->left!=T.nil)
                x=x->left;
            return x;
        }
        struct node *y=x->p;
        while(y!=T.nil && x==y->right){
            x=y;
            y=y->p;
        }
        return y;
}
int rb_delete_fixup(struct node *x){//调整只要旋转会改变子树结构
        struct node *w;
        while(x!=T.root && x->color==0){
            if(x==x->p->left){//x 是双亲的左孩子
                w=x->p->right;//w 是 x 的兄弟
                if(w->color==1){//x 的兄弟是红色,case1
                    w->color=0;
                    x->p->color=1;
                    left_rotate(x->p);
                    w=x->p->right;
                }//case1 转换为 case2,3,4

if(w->left->color==0&&w->right->color==0){//case2,w 两个孩子为黑
                    w->color=1;
                    x=x->p;
                }//x 上移
                else {//case 3,4
                    if(w->right->color==0){//case3,w 右子为黑，左子为红
                        w->left->color=0;
                        w->color=1;
                        right_rotate(w);
                        w=x->p->right;
                    }//case3 转为 case4
                    w->color=x->p->color;
                    x->p->color=0;
                    w->right->color=0;
                    left_rotate(x->p);
                    x=T.root;
                }
```

```
            }
            else{//x 是双亲的右孩子
                w=x->p->left;//w 是 x 的兄弟
                if(w->color==1){//x 的兄弟是红色,case1
                    w->color=0;
                    x->p->color=1;
                    right_rotate(x->p);
                    w=x->p->left;
                }//case1 转换为 case2,3,4

if(w->left->color==0&&w->right->color==0){//case2,w 两个孩子为黑
                    w->color=1;
                    x=x->p;
                }//x 上移
                else {//case 3,4
                    if(w->left->color==0){//case3
                        w->right->color=0;
                        w->color=1;
                        left_rotate(w);
                        w=x->p->left;
                    }//case3 转为 case4
                    w->color=x->p->color;
                    x->p->color=0;
                    w->left->color=0;
                    right_rotate(x->p);
                    x=T.root;
                }
            }
        }
        x->color=0;
}
struct node* rb_delete(struct node *z){//删除 z 结点，从实际删除的 y 节点向上修改 max
        struct node *y,*x;
        if(z->left==T.nil || z->right==T.nil)//case 1,2
            y=z;
        else
            y=treesuccessor(z);//y 是 z 的中序后继
        //此时，y 统一是 x 的双亲节点，且是要删除的节点
        //x 是待连接到 y.p 的节点，下面要确定 x
        if(y->left != T.nil)
            x=y->left;
        else
```

```
        x=y->right;
    //以下用 x 取代 y
    x->p=y->p;
    if(y->p==T.nil)
        T.root=x;
    else if(y==y->p->left)
        y->p->left=x;
    else
        y->p->right=x;
    if(y!=z){
        z->key=y->key;
        z->high=y->high;
        z->max=y->max;
    }
    x=x->p;
    while(x!=T.nil){//从 x 向上修改

x->max=mymax(x->high,x->left->max,x->right->max);
        x=x->p;
    }
    if(y->color==0)//y 是红点，删除不影响，y 是黑点，需要
调整
        rb_delete_fixup(x);
    return y;//返回实际删除的节点
}


int main() {
    int i,num,high;
    struct node *p;
    struct node *x;
    srand(time(NULL));
    T.nil=(struct node *)malloc(sizeof(struct node));
    T.nil->color=0;//nil.color=black

T.nil->high=0;T.nil->key=0;T.nil->max=0;T.nil->left=T.nil;T.nil-
>right=T.nil;
    T.root=T.nil;
    printf("Please input the number of the nodes: ");
    scanf("%d",&num);
    printf("Random number:");
    for(i=0;i<num;i++){
        p=(struct node *)malloc(sizeof(struct node));
        p->key=rand()%500;
```

```
            while(p->key==0)
                p->key=rand()%500;
            high=rand()%500;
            while(high<=p->key)
                high=rand()%500;
            p->high=high;
            p->max=high;
            printf("[%d,%d] ",p->key,p->high);
            rb_insert(p);
        }
        printf("\ninorder walk:");
        rb_inorder(T.root);
        printf("\nSearch:");
        int tmp=num/4;
        p=T.nil;
        rb_search_order(T.root,&tmp,&p);
        printf("%dth                                    lowest
number:[%d,%d]\n",num/4,p->key,p->high);
        tmp=num/2;
        x=T.nil;
        rb_search_order(T.root,&tmp,&x);
        printf("%dth                                    lowest
number:[%d,%d]\n",num/2,x->key,x->high);
        //p=treesuccessor(p);
        //printf("%d ",p->key);
        rb_delete(p);
        rb_delete(x);
        printf("inorder walk:");
        rb_inorder(T.root);
        //p=rb_search_order(T.root,num/2);
        //rb_delete(p);
        return 0;
    }
```