实验 3 红黑树和顺序统计树

1. 实验要求

实验 1：实现红黑树的基本算法， 对 n 的取值分别为 12、24、36、48、60，随机生成 n 个互异的正整数（$K_1, K_2, K_3, \ldots\ldots, K_n$）作为节点的关键字，向一棵初始空的红黑树中依次插入这 n 个节点，统计算法运行所需时间 ，画出时间曲线。（红黑树采用三叉链表）

实验 2：对上述生成的红黑树，找出树中的第 n/3 小的节点和第 n/4 小的节点，并删除这两个节点，统计算法运行所需时间 ， 画出时间曲线。

算法实现:

a)本次实验需要实现红黑树部分基本算法主要包括如下：

1) 实现红黑树左旋操作 LEFT-ROTATE(T, x) 实现右旋操作 RIGHT-ROTATE(T, x)

2)实现红黑树插入节点的操作 RB-INSERT(T, z)以及插入之后修正为红黑书的的算法 RB-INSERT-FIXUP(T, x)（在函数实现过程中对于 case1 case2 case3 的三部分代码要注释清楚）

3)实现红黑树删除节点的操作 RB-DELETE(T, z)以及插入之后修正为红黑书的的算法 RB-DELETE-FIXUP(T, x) （在函数实现过程中对于 case1 case2 case3 case4 的四部分代码要注释清楚）

4)实现按要求数据构建顺序统计树的操作

5)实现遍历输出构建好的红黑树的操作

6)实现查找顺序统计树的第 i 小关键字的操作 OS-SELECT(T.root,i)

b) 为了验证第二个实验的正确性，要求编写一个检测程序，使用中位数一章的线性时间的选择算法 Select(a,p,r,i)在输入数据找到找到第 n/3 小的节点和第 n/4 小节点，与 OS-SELECT(T.root,i)的结果 delete_data 进行对比检查

2. 实验环境

编译环境：gcc (Ubuntu 5.4.0-6ubuntu1~16.04.4) 5.4.0 20160609
机器内存：5.8G
时钟主频：2.10GHz
注：在 Ubuntu16.04 虚拟机下进行实验

3. 实验过程

注：方便起见，本次实验将 ex1 和 ex2 一起实现。

1. 随机生成 60 个互异的正整数，存放在 input/input.txt 中。

为了生成互异的随机数，先建立一个大小为 60 的数组。每次生成一个随机正整数（此处取 1~1000），把它与之前已经生成过并存放到数组中的随机数进行比较。若有相同的，则重新生成一个随机数，再将它与之前的数进行比较，直到生成一

个与之前的数都互异的正整数，把它存放在数组中。
然后再将数组中的数输入到 input.txt 文件中，即可。
注：以下为实验 1 内容
2. 建立一个空的红黑树（顺序统计树）
3. 从 input.txt 中读取前 n 个数据，并将它们一一插入红黑树中。
在此过程中进行计时。由于需要每插入十个数据记录一次时间，所以使用 st_t/ed_t 记录整个插入过程的开始/结束时间，对每插入十个数据，再用 st/ed 记录开始/结束时间，将每次的 difftime(ed, st)和总的 difftime(ed_t, st_t)输入到 output/size{n}/time1.txt 中。
4. 将生成的红黑树的先序/中序/后序遍历结果分别输出到 output/size{n}/preorder.txt, inorder.txt, postorder.txt 中。
注：以下为实验 2 内容
5. 使用 OS-SELECT 函数找到顺序统计树中 rank 为 n/3 和 n/4 的结点。
6. 将这两个结点删除，分别记录时间并输出到 output/size{n}/time2.txt 中，再将这两个结点的结点信息记录到 output/size{n}/delete_data.txt 中。
7. 删除红黑树。释放所有结点的内存空间。
8. 为了验证 OS-SELECT 函数的正确性。从 input.txt 读出 n 个数据存放在数组中，用线性时间算法 select 找到第 n/3 小和(n-1)/4 小的数据，与 delete_data.txt 中的数据进行比较。

4. 实验关键代码截图（结合文字说明）

根据实验要求分别做说明：

a)本次实验需要实现红黑树部分基本算法主要包括如下：

1) 实现红黑树左旋操作 LEFT-ROTATE(T, x) 实现右旋操作 RIGHT-ROTATE(T, x)

这两个算法几乎按照课本来，故对于实现方法不再加以赘述。但为了避免 left-rotate/right-rotate 函数的错误使用，如对 T->nil 结点进行左旋右旋操作，或左旋操作的 x->right 为 T->nil 这种错误，首先判断 x/y 结点是否为 T->nil，如不是，再继续进行操作。另外，由于本次实验中实现顺序统计树，结点中多了一个 size 域，故 left-rotate/right-rotate 中还需要对结点的 size 域进行调整。由于左旋/右旋操作并没有删除或添加结点，经过分析易知，这两个操作只改变了 x 和 y 的 size。只需使 y->size=x->size; x-size=x->left-size+x->right->size+1;即可。

```
void left_rotate(RBTree T, RBNode *x)
{
    if(x == T->nil) return;
    RBNode *y = x->right; //set y
    if(y == T->nil) return;
    x->right = y->left; //turn y's left subtree into x's right subtree
    if(y->left != T->nil)
        y->left->parent = x;
    y->parent = x->parent;
    if(x->parent == T->nil)
        T->root = y;
    else if(x == x->parent->left)
        x->parent->left = y;
    else
        x->parent->right = y;
    y->left = x;
    x->parent = y;
    y->size = x->size;
    x->size = x->left->size + x->right->size +1;
}
```

```
void right_rotate(RBTree T, RBNode *x)
{
    if(x == T->nil) return;
    RBNode *y = x->left;
    if(y == T->nil) return;
    x->left = y->right;
    if(y->right != T->nil)
        y->right->parent = x;
    y->parent = x->parent;
    if(x->parent == T->nil)
        T->root = y;
    else if(x == x->parent->left)
        x->parent->left = y;
    else
        x->parent->right = y;
    y->right = x;
    x->parent = y;
    y->size = x->size;
    x->size = x->left->size + x->right->size +1;
}
```

2)实现红黑树插入节点的操作 RB-INSERT(T, z)以及插入之后修正为红黑书的的算法 RB-INSERT-FIXUP(T, x)

Insert 和 insert-fixup 函数的算法均与课本一致，此处不再赘述。

函数 insert-fixup 中的三种情况，以 z->parent==z->parent->parent->left 为例，实现如下图所示：

```
if(z->parent == z->parent->parent->left)
{
    y = z->parent->parent->right;
    if(y->color == red) //case 1
    {
        z->parent->color = black;
        y->color = black;
        z->parent->parent->color = red;
        z = z->parent->parent;
    }
    else
    {
        if(z == z->parent->right) //case 2
        {
            z = z->parent;
            left_rotate(T,z);
        }
        //case 3
        z->parent->color = black;
        z->parent->parent->color = red;
        right_rotate(T,z->parent->parent);
    }

}
```

此处主要讲一下对于附加的 size 域，这两个函数中作出的调整。由于函数

inset-fixup 中，除了调用 left-rotate/right-rotate 函数外，没有改变任何一个结点的子结点，而左旋/右旋函数中已经对 size 域进行了调整，故 insert-fixup 函数中无需进行任何对于 size 域的操作。

Insert 函数中，在调用 insert-fixup 前，首先将插入的结点 z 的 size 初始化为 1，插入 z 后，对 z 的所有祖先节点，将其 size 域加一。如下图所示：

```
z->size = 1;
RBNode *temp = z->parent;
while(temp!=T->nil)
{
    temp->size++;
    temp = temp->parent;
}
RB_insert_fixup(T,z);
```

3)实现红黑树删除节点的操作 RB-DELETE(T, z)以及插入之后修正为红黑书的的算法 RB-DELETE-FIXUP(T, x)

同上，由于思路与课本一致，此处不对算法思想加以赘述。在函数 delete-fixup 中，以 x==x->parent->left 为例，遇到的四种情况的不同实现如下图：

```
if(x == x->parent->left)
{
    w = x->parent->right;
    if(w->color == red) //case 1
    {
        w->color = black;
        x->parent->color = red;
        left_rotate(T,x->parent);
        w = x->parent->right;
    }
    if(w->left->color == black && w->right->color == black) //case 2
    {
        w->color = red;
        x = x->parent;
    }
    else if(w->right->color == black) //case 3
    {
        w->left->color = black;
        w->color = red;
        right_rotate(T,w);
        w = x->parent->right;
    }
    //case 4
    w->color = x->parent->color;
    x->parent->color = black;
    w->right->color = black;
    left_rotate(T,x->parent);
    x = T->root;
}
```

对于 size 域的调整，由于除了调用 left-rotate/right-rotate 函数，delete-fixup 函数中没有改变任何结点的 left/right 域，故 delete-fixup 函数无需对任何结点的 sie 域进行调整。在 delete 函数中，若 z 的左子树或右子树为空时，只需调用 transplant 函数，transplant 函数中对于 size 域的调整下文再详述；若 z 的做右子

树均不为空，在用 z 的后继 y 代替 z 后，显然 y 及其祖先结点的 size 域均发生改变，操作如下：

```
RB_transplant(T,z,y);
y->left = z->left;
y->left->parent = y;
y->color = z->color;
RBNode *temp = y;
while(y!=T->nil)
{
    y->size = y->left->size + y->right->size + 1;
    y = y->parent;
}
```

对于 transplant(T,u,v)函数，在 transplant 执行完后，将 v 及其祖先结点的 size 域皆作出调整即可，如下图：

```
void RB_transplant(RBTree T, RBNode *u, RBNode *v)
{
    RBNode *temp = u->parent;

    if(u->parent == T->nil)
    {
        T->root = v;
    }
    else if(u == u->parent->left)
    {
        u->parent->left = v;
    }
    else
    {
        u->parent->right = v;
    }
    if(v!=T->nil) v->parent = u->parent;
    while(temp!=T->nil)
    {
        temp->size = temp->left->size + temp->right->size +1;
        temp = temp->parent;
    }

}
```

4)实现按要求数据构建顺序统计树的操作

在 insert/delete 的同时修改 size 域，实现顺序统计树。

5)实现遍历输出构建好的红黑树的操作

遍历操作的实现使用递归，较为简单：先序遍历中，首先输出当前结点的信息，在按顺序对左子树/右子树进行先序遍历；中序遍历中，首先中序遍历左子树，再输出当前结点信息，最后对右子树进行中序遍历；后序遍历中，首先按顺序对左子树/右子树进行后序遍历，最后输出当前结点的信息。

```
void RBTree_print_preorder(RBTree T, RBNode *x, FILE *stream)
{
    if(x == T->nil) return;

    RBNode_print(T,x,stream);
    RBTree_print_preorder(T,x->left,stream);
    RBTree_print_preorder(T,x->right,stream);
}

void RBTree_print_inorder(RBTree T, RBNode *x, FILE *stream)
{
    if(x == T->nil) return;
    RBTree_print_inorder(T,x->left,stream);
    RBNode_print(T,x,stream);
    RBTree_print_inorder(T,x->right,stream);
}

void RBTree_print_postorder(RBTree T, RBNode *x, FILE *stream)
{
    if(x == T->nil) return;
    RBTree_print_postorder(T,x->left,stream);
    RBTree_print_postorder(T,x->right,stream);
    RBNode_print(T,x,stream);
}
```

6)实现查找顺序统计树的第 i 小关键字的操作 OS-SELECT(T.root,i)

```
RBNode *OS_select(RBNode *x,int i)
{
    int r = x->left->size+1;
    if(i == r)
        return x;
    if(i < r)
        return OS_select(x->left,i);
    return OS_select(x->right,i-r);
}
```

使用递归查询，较为简单。

b) 为了验证第二个实验的正确性，要求编写一个检测程序，使用中位数一章的线性时间的选择算法 Select(a,p,r,i)在输入数据找到找到第 n/3 小的节点和第 n/4 小节点，与 OS-SELECT(T.root,i)的结果 delete_data 进行对比检查

首先读取 input.txt 前 n 个数据，存在一个数组中，对它使用 random-select 取第 n/3 和 n/4 小的结点，再与 delete_data.txt 的数据进行比较即可。下面简单叙述以下 random-select 的算法：

该算法借鉴了 random-quicksort 中的 random-partition 思想，随机指定数组 A[p..r]中的一个数 x，经过 partition 后，数组 A[p..r]中所有比 x 小的存放在 x 左边，比 x 大的存放在 x 右边，返回 x 的新下标 q，显然，x 是 A[p..r]中第 q-p+1 小的数。将 q-p+1 和 i 进行比较，若 i 较小，则下次进行 select 时，只需考虑数组 A[p..q-1]中的元素，否则，只需考虑数组 A[q+1..r]中的元素。这样以来，select 操作的时间复杂度是线性的。

```
int random_select(int n, int A[n], int p, int r, int i)
{
    if(p==r) return A[p];
    int q = random_partition(n,A,p,r);
    int k = q-p+1;
    if(i==k) return A[q];
    else if(i<k) return random_select(n,A,p,q-1,i);
    else return random_select(n,A,q+1,r,i-k);
}
```

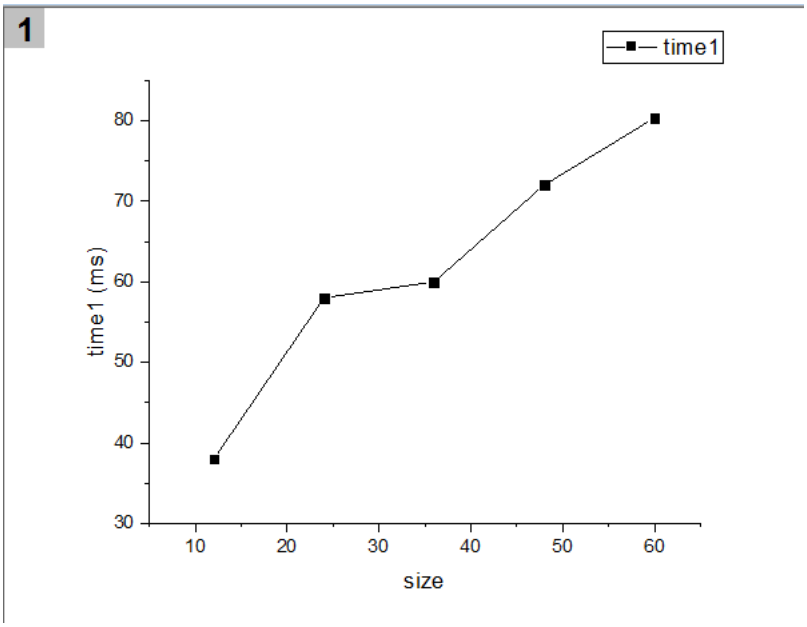
5. 实验结果、分析（结合相关数据图表分析）

1. 实验结果经过人工排查，每次 insert 和 delete 后的结果均正确。通过 select 函数的比较，OS_SELECT 函数也正确运行。

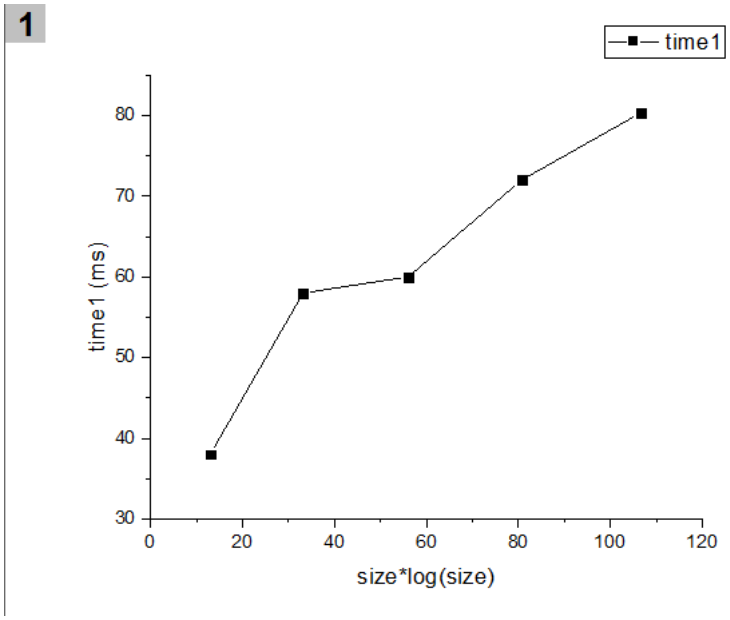
2. 算法所用时间上，总共取 3 组实验运行时间，具体在该目录下 time.xlsx 中有记录，此处列出这 3 组实验所用的平均时间：

	average time/ms	
size	time1	time2
12	38.00	5.67
24	58.00	5.00
36	60.00	6.67
48	72.00	8.33
60	80.33	5.33

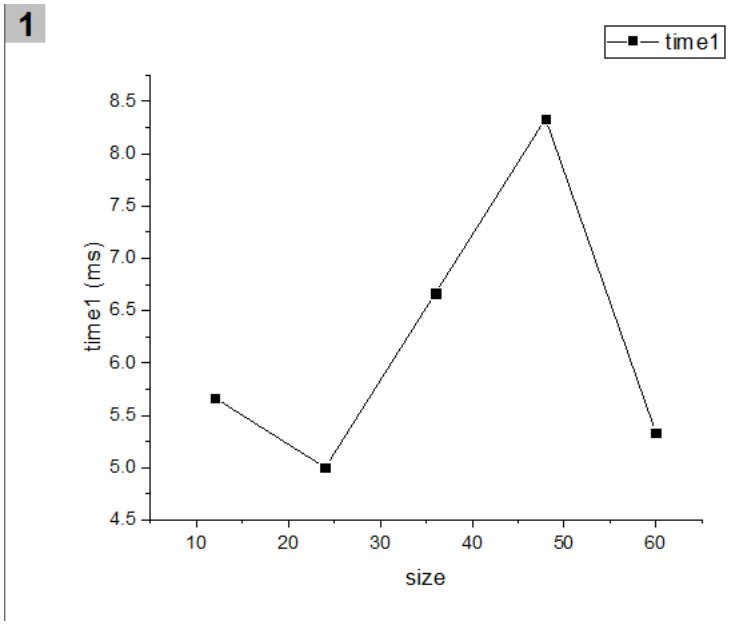
对于实验 1，可以看出 time1 几乎是线性增长的。由 time1-size 图，我们可以看出，当 size=24 时数据其实是异常的，但由于这三次实验中 size=24 时数据差距很小，故并不是某次实验误差太大的原因，具体原因暂时不明。



考虑到，经过数学证明，time1 的时间复杂度应为 O(nlogn)，故作下图 time1-nlogn。可看出除了 n=24 时的异常数据外，几乎是呈线性关系的。



对于实验 2，由于删除两个结点所需时间太短，且所求时间的精度只为毫秒，故存在一定误差，数据浮动较大，无法作出准确的判断。



6. 实验心得

利用红黑树进行顺序统计的效率非常高。红黑树中插入、删除、查找的时间复杂度均为 O(logn)。