山东大学 计算机科学与技术 学院

数据结构与算法 课程实验报告

学号：201700130033	姓名： 武学伟	班级： 2017 级 2 班
实验题目：搜索树		
实验学时：2	实验日期： 2018.12.10	

实验目的：

1.掌握二叉搜索树结构的定义、描述方法、操作实现。

软件环境：

Win10home，sublime

1. 实验内容（题目内容，输入要求，输出要求）

输入描述：

输入第一行一个数字 m ， 表示有 m 个操作。

输入的每一行有两个数字 a，b。

当输入的第一个数字 a 为 0 时，输入的第二个数字 b 表示向搜索树中插入 b；

当输入的第一个数字 a 为 1 时，输入的第二个数字 b 表示向搜索树中查找 b；

当输入的第一个数字 a 为 2 时，输入的第二个数字 b 表示向搜索树中删除 b；

当输入的第一个数字 a 为 3 时，输入的第二个数字 b 表示查找搜索树中名次为 b 的元素；

当输入的第一个数字 a 为 4 时，输入的第二个数字 b 表示删除搜索树中名次为 b 的元素；

请注意，查询与删除操作中，待查询的元素也需要比较。

查找（删除）操作中，如果未找到，或者插入操作，已存在，输出 0 ，不需要输出异或和。

查找（删除）第 b 大，如果不存在，输出 0。

删除操作中，如果当前元素有两个孩子，替换的为右子树中最小的，如果只有一个孩子，直接用该孩子替换当前元素，如果没有孩子，直接删除。

输出描述：

对于输入中的每一种操作，输出执行操作的过程中依次比较的元素值的 异或和。

提示：

查找和删除第 k 大的元素时，可以先把第 k 的元素找到，再按照该元素查找和删除

2. 数据结构与算法描述 （整体思路描述，所需要的数据结构与算法）

数据结构：

1） 二叉树、二叉搜索树

算法：

名次查找可以利用带索引的二叉搜索树，加入成员 leftSize，用来标记名次。

1） 查找：

定义一个指针 p，从根节点开始，节点值大于查找值，去找左孩子，否则去右孩子，直至找到需要查找的元素。在每次循环的时候，进行节点的值的异或运算。

```
E output = 0;                    //用于输出比较节点的异或
binaryTreeNode<E> *p = root;
while (p != NULL)
{
    output ^= p->element;
    if (theElement < p->element)
        p = p->leftChild;
    else
        if (theElement > p->element)
            p = p->rightChild;
        else
        {
            cout << output << endl;
            return;
        }
}
```

2） 插入：

先进行一次查找的判断，然后确定元素是否存在，以免修改了错误的 leftSize 的值。

先进行依次查找是 bool 型的函数：

```
template <class E>
bool binarySearchTree<E>::find2(const E& theElement) con
{
    binaryTreeNode<E> *p = root;
    while (p != NULL)
    {
        if (theElement < p->element)
            p = p->leftChild;
        else if (theElement > p->element)
            p = p->rightChild;
        else
            return true;
    }
    return false;
}
```

然后利用 p 和 pp 来找到插入的正确位置：

pp 是 p 的父节点

```
binaryTreeNode<E> *p = root;
binaryTreeNode<E> *pp = NULL;
```

同时在循环中进行异或运算

```
while (p != NULL)                //值不存在，找到正确的插入位置
{
    output ^= p->element;    //与p节点的值做异或运算
    pp = p;
    if (theElement < p->element)
    {
        p->leftSize++;
        p = p->leftChild;
    }
    else
        if (theElement > p->element)
            p = p->rightChild;
}
```

然后判断根节点是否存在，不存在直接设置为根节点，存在则判断一下与 pp 的大小关系决定插入的左右孩子位置。

3) 删除：

同样首先判断是否存在，不存在直接退出。

然后找到元素值，异或运算在查找的循环中进行。

然后分情况讨论，当元素有两个孩子时，将其转化为一个孩子的情况，在 p 的右子树中寻找最小值替换，转化为一个孩子或者没有孩子的情况，这时按照删除一个孩子或者没有孩子的情况删除。

建立一个节点，等于删除节点的孩子，当删除的节点是根节点，直接替换，否则判断 p 是 pp 的左孩子还是右孩子，选择合适的替换位置。

若没有孩子，直接删除就可以。

4) 按名次查找：

在查找的基础上进行修改，参数是 rank。

比较 rank 与当前元素 leftSize+1 的值，等于就停止循环，并输出结果，大于就跳至右子树，并让 rank 减去 leftSize+1，小于直接跳至左子树。

```
if (rank < p->leftSize+1)
    p = p->leftChild;
else
    if (rank > p->leftSize+1)
    {
        rank -= (p->leftSize+1);
        p = p->rightChild;
    }
    else
    {
        cout << output << endl;
        return;
    }
```

5） 按名次删除：

首先判断是否存在该元素，然后获取元素的值作为参数直接调用 erase 函数。

判断存在可以直接修改按名次查找的函数：

```
/*查找到对应的元素值，用于按名次的删除*/
template <class E>
E binarySearchTree<E>::rankfind2(int rank)
{
    if (rank < 1 || rank > treeSize)
        return 0;
    binaryTreeNode<E> *p = root;
    while (p != NULL)
    {
        if (rank == p->leftSize+1)
            return p->element;
        else if (rank > p->leftSize+1)
        {
            rank -= (p->leftSize+1);
            p = p->rightChild;
        }
        else if (rank < p->leftSize+1)
            p =  p->leftChild;
    }
    return 0;              //没有查询到
}
```

然后调用函数

```
if (rank > treeSize || rank < 1)
    cout << 0 << endl;
erase(rankfind2(rank));     //先查找元素再调用删除函数
```

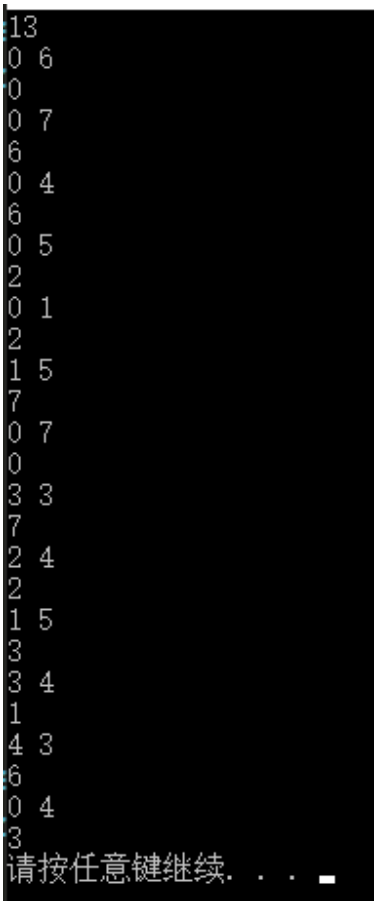
3. 测试结果（测试输入，测试输出，结果分析）

测试一

输入：

13
0 6
0 7
0 4
0 5
0 1

1 5
0 7
3 3
2 4
1 5
3 4
4 3
0 4
输出:
0
6
6
2
2
7
0
7
2
3
1
6
3



4. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）

结果分析：

数据正确。

实验原本打算利用函数直接计算某个节点的右子树的节点数，作为名次排序的依据。但是这样每一次访问一个元素就需要调用一次函数，并且函数的时间复杂度是 O(n)，n 是节点数，然后查找与删除的时间复杂度均是 O(logn)，所以不如直接将 lestSize 作为节点的成员，省去了计算节点数的时间。

在按名次删除的时候，调用了一个按名次查找的 int 型函数，其中没有找到的返回值为零，若是树中有元素值为 0 的话，删除就会出错，本题中没有，所以不用注意，否则就必须在原本删除的操作中修改，循环中的比较就需要改为名次的比较，并且先行查找的函数类型改为 bool 型，删除时先判断是否为真，不为真直接退出。

5. 附录：实现源代码（本实验的全部源程序代码，程序风格清晰易理解，有充分的注释）

```
/*Ex10_1.cpp*/
#include <iostream>
using namespace std;

template <class T>
struct binaryTreeNode
{
    T element;
    int leftSize;          //补充左子树的节点数，方便进行名次的排序
    binaryTreeNode<T> *leftChild, *rightChild;
    binaryTreeNode()
    {
        leftSize = 0;
        leftChild = rightChild = NULL;
    }
    binaryTreeNode(const T& theElement):element(theElement)
    {
        leftSize = 0;
        leftChild = rightChild = NULL;
    }
    binaryTreeNode(const T& theElement,const int& theLeftSize,
binaryTreeNode *theLeftChild, binaryTreeNode
*theRightChild):element(theElement)
    {
        leftSize = theLeftSize;
        leftChild = theLeftChild;
        rightChild = theRightChild;
    }
};
```

```
template <class E>
class binarySearchTree
{
    public:
        binarySearchTree()
        {
            root = NULL;
            treeSize = 0;
        }
        ~binarySearchTree() {;}
        bool empty() const {return treeSize == 0;}
        int size() const {return treeSize;}
        void find(const E& theElement) const;      //查找
        bool find2(const E& theElement) const;
        void insert(const E& theElement);      //插入
        void erase(const E& theElement);           //删除
        void rankfind(int rank);        //查找名次对应的元素
        E rankfind2(int rank);
        void rankerase(int rank);       //删除名次对应的元素
    private:
        binaryTreeNode<E> *root; //根节点
        int treeSize;                          //树的节点数
};

template <class E>
void binarySearchTree<E>::find(const E& theElement) const
{
    E output = 0;            //用于输出比较节点的异或和
    binaryTreeNode<E> *p = root;
    while (p != NULL)
    {
        output ^= p->element;
        if (theElement < p->element)
            p = p->leftChild;
        else
            if (theElement > p->element)
                p = p->rightChild;
            else
            {
                cout << output << endl;
                return;
            }
    }
```

```
    if (p == NULL)
        cout << 0 << endl;//没有查询到直接输出 0
}

template <class E>
bool binarySearchTree<E>::find2(const E& theElement) const
{
    binaryTreeNode<E> *p = root;
    while (p != NULL)
    {
        if (theElement < p->element)
            p = p->leftChild;
        else if (theElement > p->element)
            p = p->rightChild;
        else
            return true;
    }
    return false;
}

template <class E>
void binarySearchTree<E>::insert(const E& theElement)
{
    if (find2(theElement))
    {
        cout << 0 << endl;
        return;
    }                           //先查找一次，以免未找到的时候修改了错
误的 leftSize
    E output = 0;               //用于输出比较节点的异或和
    binaryTreeNode<E> *p = root;
    binaryTreeNode<E> *pp = NULL;
    while (p != NULL)           //值不存在，找到正确的插入位置
    {
        output ^= p->element;    //与 p 节点的值做异或运算
        pp = p;
        if (theElement < p->element)
        {
            p->leftSize++;
            p = p->leftChild;
        }
        else
            if (theElement > p->element)
                p = p->rightChild;
```

```
    }
    binaryTreeNode<E> *newNode = new binaryTreeNode<E>
(theElement);
    if (root != NULL)          //插入新值，若树不为空，与 pp 链接
        if (theElement < pp->element)
            pp->leftChild = newNode;
        else
            pp->rightChild = newNode;
    else                       //树为空，作为根节点
        root = newNode;
    cout << output << endl;
    treeSize++;
}

template <class E>
void binarySearchTree<E>::erase(const E& theElement)
{
    if (!find2(theElement))
    {
        cout << 0 << endl;
        return;
    }
    E output = 0;
    binaryTreeNode<E> *p = root;
    binaryTreeNode<E> *pp = NULL;
    output ^= p->element;
    while (p != NULL && p->element != theElement)    //找到对应元素
    {
        pp = p;
        if (theElement < p->element)
        {
            p->leftSize--;
            p = p->leftChild;
        }
        else
            p = p->rightChild;
        output ^= p->element;
    }
    if (p->leftChild != NULL && p->rightChild != NULL)   //同时存在
左孩子与右孩子
    {
        //在 p 的右子树中寻找最小值
        binaryTreeNode<E> *s = p->rightChild;
        binaryTreeNode<E> *ps = p;      //s 的父节点
```

```
        while (s->leftChild != NULL)
        {
            ps = s;
            s->leftSize--;
            s = s->leftChild;
        }
        //将最小元素移动到 p
        binaryTreeNode<E> *q = new binaryTreeNode<E> (s->element,
p->leftSize, p->leftChild, p->rightChild);
        if (pp == NULL)
            root = q;
        else if (p == pp->leftChild)
            pp->leftChild = q;
        else
            pp->rightChild = q;
        if (ps == p)
            pp = q;
        else
            pp = ps;
        delete p;
        p = s;
    }
    binaryTreeNode<E> *c;
    if (p->rightChild != NULL)
        c = p->rightChild;
    else
        c = p->leftChild;
    if (p == root)
        root = c;
    else
    {
        if (p == pp->leftChild)
            pp->leftChild = c;
        else
            pp->rightChild = c;
    }
    treeSize--;
    delete p;
    cout << output << endl;
}

template <class E>
void binarySearchTree<E>::rankfind(int rank)
{
```

```
    if (rank < 1 || rank > treeSize)
    {
        cout << 0 << endl;
        return;
    }               //错误的 rank，直接退出
    int output = 0;
    binaryTreeNode<E> *p = root;
    while (p != NULL)
    {
        output ^= p->element;
        if (rank < p->leftSize+1)
            p = p->leftChild;
        else
            if (rank > p->leftSize+1)
            {
                rank -= (p->leftSize+1);
                p = p->rightChild;
            }
            else
            {
                cout << output << endl;
                return;
            }
    }
    if (p == NULL)
        cout << 0 << endl;
    return;
}

/*查找到对应的元素值，用于按名次的删除*/
template <class E>
E binarySearchTree<E>::rankfind2(int rank)
{
    if (rank < 1 || rank > treeSize)
        return 0;
    binaryTreeNode<E> *p = root;
    while (p != NULL)
    {
        if (rank == p->leftSize+1)
            return p->element;
        else if (rank > p->leftSize+1)
        {
            rank -= (p->leftSize+1);
            p = p->rightChild;
```

```
        }
        else if (rank < p->leftSize+1)
            p =  p->leftChild;
    }
    return 0;          //没有查询到
}

template <class E>
void binarySearchTree<E>::rankerase(int rank)
{
    if (rank > treeSize || rank < 1)
        cout << 0 << endl;
    erase(rankfind2(rank));      //先查找元素再调用删除函数
}

int main()
{
    binarySearchTree<int> bsTree;
    int num;
    cin >> num;
    int choice;
    int input;
    for (int i=0; i<num; i++)
    {
        cin >> choice;
        switch(choice)
        {
            case 0:
                cin >> input;
                bsTree.insert(input);
                break;
            case 1:
                cin >> input;
                bsTree.find(input);
                break;
            case 2:
                cin >> input;
                bsTree.erase(input);
                break;
            case 3:
                cin >> input;
                bsTree.rankfind(input);
                break;
            case 4:
```

```
                cin >> input;
                bsTree.rankerase(input);
                break;
            default:
                break;
        }
    }
    return 0;
}
```