山东大学计算机科学与技术学院

数据结构与算法课程设计报告

学号：201700140056	姓名：李港	班级：17.4
上机学时：4		日期：2020.04.01
课程设计题目：森林与二叉树之间的转换		
软件环境：VS2019		

报告内容：

1.需求描述

1.1 问题描述

设计并实现森林与二叉树之间的转换算法。

1.2 基本要求

1. 构造并实现森林的 ADT 和二叉树 ADT，
    1. 森林 ADT 中应包括初始化、插入元素、删除元素，插入边，删除边，转换成二叉树，显示森林等基本操作
    2. 二叉树 ADT 中应包括初始化、插入根、插入指定元素的左孩子或右孩子，转换成森林，显示二叉树等基本操作。
2. 森林使用孩子链表表示，二叉树使用二叉链表表示，实现森林和二叉树结构。

1.3 输入说明

1. 输入设计：采用终端进行输入,采用 oj 的输入风格。
    1. 第一行三个数 K, M, N。
        1. K :初始化 ADT 的类型，0 表示接下来初始化森林，1 表示初始化二叉树。
        2. M :若 K=0，M 代表森林中树的个数. 若 K=1，M 一定为 1。
        3. N :初始化结点的个数，若 K=0，N 表示森林中所有树的结点的数目之和，若 K=1，N 表示二叉树中结点的个数。

    2. 接下来一行有 M 个数，分别代表初始的根结点，树中的结点均为正整数。

    3. 接下来 N 行，表示森林、二叉树中的结点信息。
        1. 若初始为森林，每行的格式为 A B [nodes]，表示结点 A 拥有 B 个孩子结点，孩子结点的集合为 nodes，i.e. 1 2 3 4 表示结点 1 拥有 2 个孩子，分别为结点 3，4。
        2. 若初始为二叉树，每行的格式为 A l r，表示结点 A 的左孩子是 l，右孩子是 r，若是某个孩子不存在，则其为值为 -1。
    4. 接下来一行一个数 Q (Q\leq 100Q≤100)，表示接下来操作的个数。
    5. 接下来 Q 行，每行的格式为 op [op_nums]，表示对当前森林/二叉树的操作，下面是各种操作的格式:
        1. 1 father node 表示为森林中树的结点 father 插入一个孩子结点 node ，若 father 为 -1 ，表示插入的是孤立结点。 注意同一父亲的孩子从左至右按大小升序保存。
        2. 2 father node 表示删除森林中的结点 node ，其中 father 是 node 的父亲结点，father 若为 -1 ，代表删除根结点。若待删除的结点 node 有孩子，则其所有孩子结点在删除后成为新的树，保留其原有的子树结构。
        3. 3 a b 表示在森林中的根结点 a，b 间插入一条边，其中 a 为 b 结点的父亲。
        4. 4 森林、二叉树转换，若当前为森林，则将森林转为二叉树，若当前为二叉树，则将二叉树转为森林。当森林转换二叉树时，将编号最小的根结点作为合并后的根结点，其余根结点按从小到大合并。二叉树转森林时，无须恢复原有的树结构。
        5. 5 pos father node 表示为二叉树的 father 结点插入一个孩子 node ，pos 的范围

为{0, 1}，0 表示插入的是 右孩子 ，1 表示插入的是 左孩子 . 数据保证 father 待插入的位置没有孩子。

6. 6 显示森林/二叉树。显示二叉树时，输出一行二叉树的前序遍历序列的异或值。显示森林时，按森林根大小的顺序升序输出一行森林中各棵树的结点遍历序列的异或值，元素间用空格分隔。

2. 输入数据样例

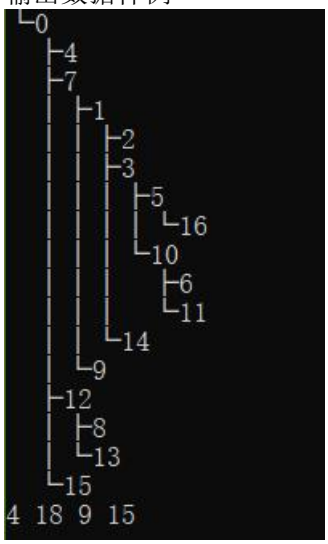
```
0 2 10
6 10
6 0
10 3 2 8 5
2 1 9
8 0
5 3 4 3 1
4 0
9 0
3 1 7
7 0
1 0
10
6
6
6
1 -1 11
1 11 12
1 6 13
6
6
4
6
```

1.4 输出说明

输出界面设计

输出操作结果信息，同时打印森林或二叉树的图形。

输出数据样例



2.分析与设计

2.1 问题分析

该项目可分解为两个问题：数据的表示与不同数据结构之间的转换。数据表示方面，我们可以采用不同或相同的类对它们进行表示。如果采用相同类进行表示，可以采用一个标志成员代表当前所处状态，从而在调用相应成员函数时进行二次分配。数据转化方面，可以参照标准的转换步骤进行编码。

2.2 主程序设计

主程序含有初始化和进一步操作这两个循环，首先读取内容进行二叉树或森林的初始化，然后循环读取操作码，进行相应的操作。对每种操作提供一个接口，每次操作之后更新可视化图形。

2.3 设计思路

本项目最开始分别采用树、二叉树、森林这三个类进行数据结构的管理，但在实践中发现多个类之间交互逻辑过多，于是在第二版中，本项目将所有结构使用同一个类进行管理极大降低了复杂度，提高了代码整洁性。

2.4 数据及数据类(型)定义

```
template <typename K,typename V>
class TNode {
private:
    TNode* _parent;//指向父节点
    TNode* _left;//指向左子节点
    TNode* _right;//指向右子节点

    //树的部分采用链表表示
    TNode* _sub;//指向树的第一个子节点
    TNode* _next;//指向树的兄弟节点

    int _index;//OJ 上输入的索引
    V _data;//数据
public:
    TNode ()
    TNode (int index, V data)
    ~TNode ()

    int getIndex ()
    V getData ()
    TNode<K, V>* getParent ()
    TNode<K, V>* getLChild ()
    TNode<K, V>* getRChild ()
    TNode<K, V>* getFirstChild ()
    TNode<K, V>* getNextSibling ()

    int BPreorder ()
    int TPreorder ()
    void BTreeToTree ()
    void TreeToBTree ()

    void setIndex (int index)
    void setData (V data)
    void setParenet (TNode* Node)
    void setLChild (TNode* Node)
    void setRChild (TNode* Node)
    void setFirstChild (TNode* Node)
    void setNextSibling (TNode* Node)
    void setChild (TNode* Node)
    TNode<K, V>* BiNodeSearch (int index)
    TNode<K, V>* Tsearch (int index)
    int NodeLeavesCount (int leaves)
    int getBTChildrenNum ()
    int deleteBAll ()
```

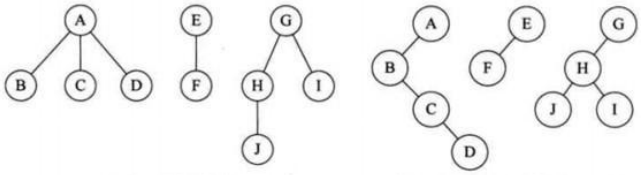
```
    int deleteTNode ()
};

template <typename K, typename V>
class Tree {
protected:
    TNode<K, V>* _root;   //指向根节点
    int _size;            //节点数目
public:
    Tree (K index, V data)
    Tree ()
    bool isEmpty ()
    bool setLeft (TNode<K, V>* node_in, K target)
    bool setRight (TNode<K, V>* node_in, K target)
    bool setSub (TNode<K, V>* node_in, K target)
    bool setSubTo (K nwe_target, V data, K target)
    void addEdge (int a, int b)
    void BTreeToTree ()
    void TreeToBTree ()
    bool deleteTreeNodeByIndex (int index)
    bool deleteTreeNodeByNode (TNode<K, V>* node_in)
    void BPreorder ()
    void TreePreorder ()
};
```

2.5.算法设计及分析

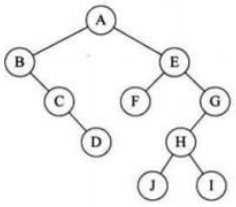
首先我们回顾一下树、二叉树、森林的相互转化

1. 森林转二叉树
    1. 把每棵树转换为二叉树
    2. 第一棵二叉树不动，从第二棵二叉树开始，一次把后一棵二叉树的根结点作为前一棵二叉树的根结点的右孩子，用线连接起来。
    3. 转换规则：兄弟相连，长兄为父，孩子靠左。



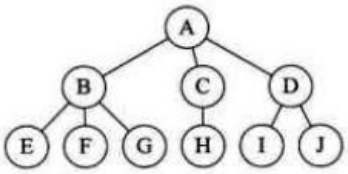
拥有三棵树的森林

步骤1:森林中每棵树转换为二叉树

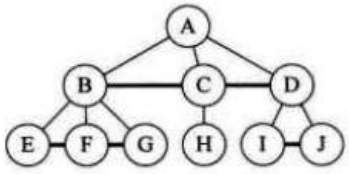


步骤2：将所有二叉树转换为一棵二叉树

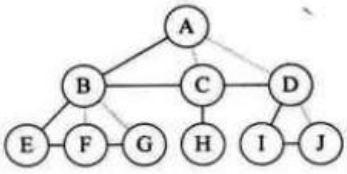
2. 树转二叉树
    1. 加线。在所有的兄弟结点之间加一条线。
    2. 去线。树中的每个结点，只保留它与第一个孩子结点的连线，删除其他孩子结点之间的连线。
    3. 调整。以树的根结点为轴心，将整个树调节一下（第一个孩子是结点的左孩子，兄弟转过来的孩子是结点的右孩子）



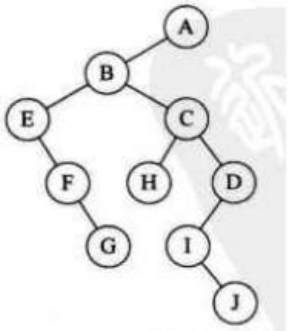
树



步骤1：给兄弟加线

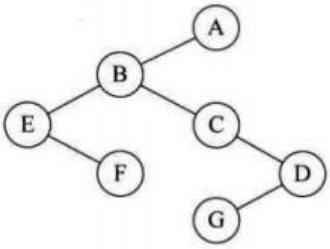


步骤2：给除长子外的孩子去线

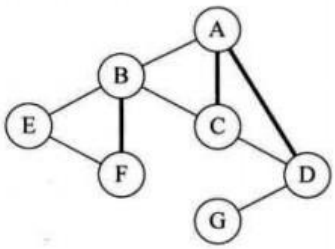


步骤3：层次调整

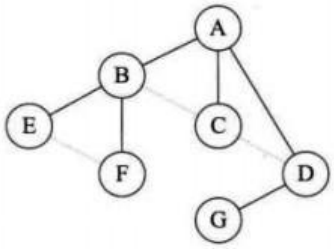
3. 二叉树转树
   1. 加线。若某结点 X 的左孩子结点存在，则将这个左孩子的右孩子结点、右孩子的右孩子的右孩子结点。。。都作为结点 X 的孩子。将结点 X 与这些右孩子结点用线连接起来。
   2. 去线。删除原二叉树中所有结点与其右孩子结点的连线。
   3. 层次调整。



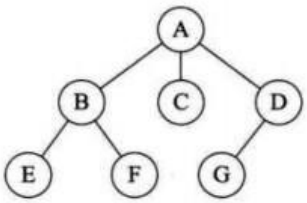
二叉树



步骤1：加线



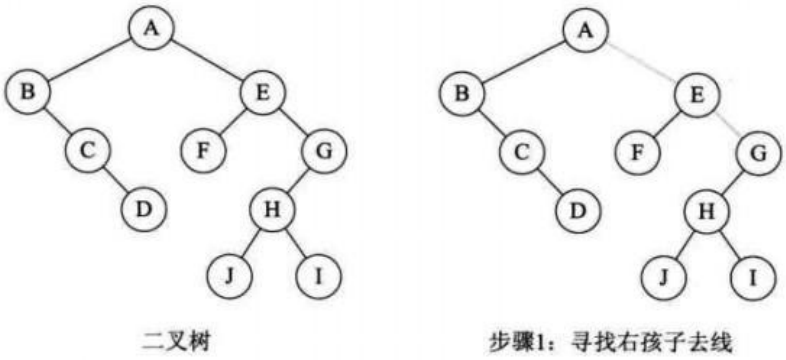
步骤2：去线



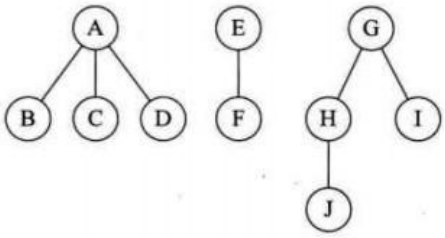
步骤3：层次调整

4. 二叉树转换为森林：

   假如一棵二叉树的根节点有右孩子，则这棵二叉树能够转换为森林，否则转换为一棵树。
   1. 从根节点开始，若右孩子存在，则把与右孩子结点的连线删除。再查看分离后的二叉树，若其根节点的右孩子存在，则连续删除。直到所有这些根结点与右孩子的连线都删除为止。
   2. 将每棵分离后的二叉树转换为树。



二叉树　　步骤1：寻找右孩子去线



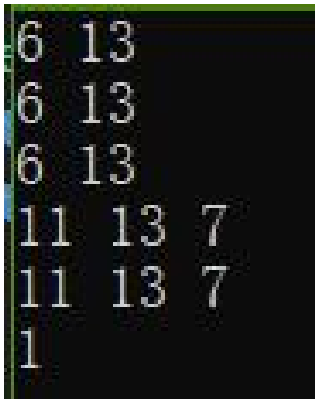
步骤2：将分离的二叉树转换成树

转换逻辑就是如此。

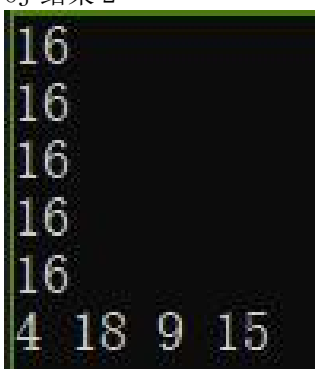
另外，在本项目中，采用成员变量_state 来表示当前结构正处于森林状态还是二叉树状态。

3. 测试

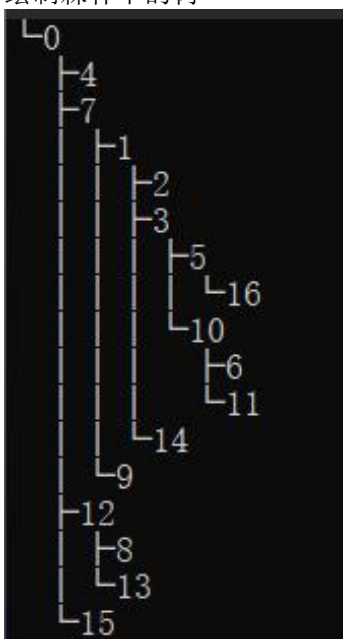
1. OJ 结果 1



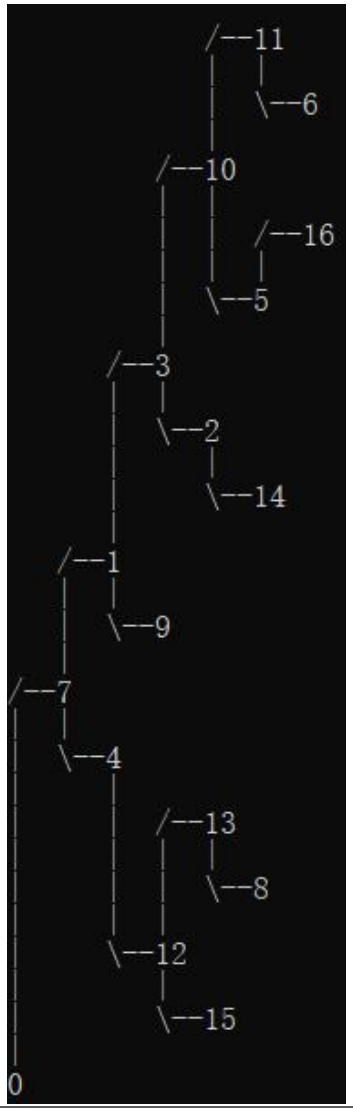
2. OJ 结果 2



3. 绘制森林中的树



4. 绘制二叉树



4. 分析与探讨

1. 抽象结构选择心得

   本项目设计经过两个阶段，第一阶段是将各种结构如树、森林、二叉树分开，但在实践中发现这种设计加大了不同类转换过程中交互行为的复杂度，因此在第二版中，本项目将三个类融合为一个类，将类间操作转化为类内操作，降低了程序复杂度。

2. 森林绘制总结：

   进行森林的绘制其实就是进行多个多叉树的绘制。由于树子节点使用链表存储，因此我们很轻易就可以通过链表打印出整棵树，采用递归过程进行嵌套打印，采用制表符绘制出树形结构。

3. 二叉树绘制总结

   为了达到“对称”的绘制效果，我采用了比绘制森林更复杂的机制来进行图形绘制。采用中序遍历进行绘制，这样就可以时竖直放置二叉树最左侧节点最先被遍历到，将二叉树平躺，中序遍历的顺序就是从上到下的顺序。

5. 附录：实现源代码

```
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

/*template <typename K,typename V>
class TNode {
private:
    TNode* _parent;//指向父节点
    TNode* _left;//指向左子节点
    TNode* _right;//指向右子节点

    //树的部分采用链表表示
    TNode* _sub;//指向树的第一个子节点
    TNode* _next;//指向树的兄弟节点

    int _index;//OJ 上输入的索引
    V _data;//数据
public:
    TNode ()
    TNode (int index, V data)
    ~TNode ()

    int getIndex ()
    V getData ()
    TNode<K, V>* getParent ()
    TNode<K, V>* getLChild ()
    TNode<K, V>* getRChild ()
    TNode<K, V>* getFirstChild ()
    TNode<K, V>* getNextSibling ()

    int BPreorder ()
    int TPreorder ()
    void BTreeToTree ()
    void TreeToBTree ()

    void setIndex (int index)
    void setData (V data)
    void setParenet (TNode* Node)
    void setLChild (TNode* Node)
    void setRChild (TNode* Node)
    void setFirstChild (TNode* Node)
    void setNextSibling (TNode* Node)
    void setChild (TNode* Node)
    TNode<K, V>* BiNodeSearch (int index)
```

```
    TNode<K, V>* Tsearch (int index)
    int NodeLeavesCount (int leaves)
    int getBTChildrenNum ()
    int deleteBAll ()
    int deleteTNode ()
};
*/
template <typename K,typename V>
class TNode {
private:
    TNode* _parent;//指向父节点
    TNode* _left;//指向左子节点
    TNode* _right;//指向右子节点

    /*树的部分采用链表表示*/
    TNode* _sub;//指向树的第一个子节点
    TNode* _next;//指向树的兄弟节点

    int _index;//OJ 上输入的索引
    V _data;//数据
public:
    TNode () : _index (0), _parent (0), _left (0), _right (0), _sub (0), _next (0), _data (0)
{}
    TNode (int index, V data) : _index (index), _data (data), _parent (0), _left (0), _right
(0), _sub (0), _next (0) {}
    ~TNode () {
        if (_left != nullptr) {
            _left->deleteBAll ();
            _left = nullptr;
        }

        if (_right != nullptr) {
            _right->deleteBAll ();
            _right = nullptr;
        }

        if (_sub != nullptr) {
            _sub->deleteTNode ();
            _sub = nullptr;
        }

        if (_next != nullptr) {
            _next->deleteTNode ();
            _next = nullptr;
        }
        _parent = nullptr;
    }


    int getIndex () { return _index; }
    V getData () { return _data; }
    TNode<K,V>* getParent () { return _parent; }
    TNode<K,V>* getLChild () { return _left; }
    TNode<K,V>* getRChild () { return _right; }
    TNode<K,V>* getFirstChild () { return _sub; }
    TNode<K,V>* getNextSibling () { return _next; }

    void setIndex (int index) { _index = index; }
    void setData (V data) { _data = data; }
    void setParenet (TNode* Node) { _parent = Node; }
    void setLChild (TNode* Node) {
        _left = Node;
        if (Node != nullptr) Node->setParenet (this);
    }
    void setRChild (TNode* Node) {
```

```
        _right = Node;
        if (Node != nullptr) Node->setParenet (this);
    }
    void setFirstChild (TNode* Node) { _sub = Node; }
    void setNextSibling (TNode* Node) { _next = Node; }
    void setChild (TNode* Node) {
        if (_sub == nullptr) {
            _sub = Node;
            Node->setParenet (this);
        }

        else if (Node->getIndex () < _sub->getIndex ()) {
            Node->setParenet (this);
            Node->setNextSibling (_sub);
            _sub = Node;
        } else {
            TNode<K,V>* ptemp_node = _sub;
            while (ptemp_node->getNextSibling () != nullptr &&
                ptemp_node->getNextSibling ()->getIndex () < Node->getIndex ())
                ptemp_node = ptemp_node->getNextSibling ();
            if (ptemp_node->getNextSibling () == nullptr) {
                ptemp_node->setNextSibling (Node);
                Node->setParenet (this);
            } else {
                Node->setParenet (this);
                Node->setNextSibling (ptemp_node->getNextSibling ());
                ptemp_node->setNextSibling (Node);
            }
        }
    }

    TNode<K,V>* BiNodeSearch (int index) {
        TNode<K,V>* temp_node = nullptr;
        if (_index == index) {
            return this;
        }
        if (_left != nullptr) {
            temp_node = _left->BiNodeSearch (index);
            if (temp_node != nullptr) {
                return temp_node;
            }
        }

        if (_right != nullptr) {
            temp_node = _right->BiNodeSearch (index);
            if (temp_node != nullptr) {
                return temp_node;
            }
        }
        return nullptr;
    }

    TNode<K,V>* Tsearch (int index) {
        TNode<K,V>* temp_node = nullptr;

        if (_index == index) {
            return this;
        }
        if (_sub != nullptr) {
            temp_node = _sub->Tsearch (index);
            if (temp_node != nullptr) {
                return temp_node;
            }
        }

        if (_next != nullptr) {
```

```
            temp_node = _next->Tsearch (index);
            if (temp_node != nullptr) {
                return temp_node;
            }
        }

        return nullptr;
    }

    int NodeLeavesCount (int leaves) {
        if (this->_left != nullptr)
            leaves = this->_left->NodeLeavesCount (leaves);

        if (this->_right != nullptr)
            leaves = this->_right->NodeLeavesCount (leaves);

        if (this->getLChild () == nullptr && this->getRChild () == nullptr)
            leaves++;

        return leaves;
    }

    int getBTChildrenNum () {
        int biCnt = 0;

        if (this->_left != nullptr)
            biCnt += this->_left->getBTChildrenNum ();

        if (this->_right != nullptr)
            biCnt += this->_right->getBTChildrenNum ();

        biCnt++;
        return biCnt;
    }


    int deleteBAll () {
        int Times = 0;
        if (this->_left != nullptr) {
            Times += this->_left->deleteBAll ();
            this->_left = nullptr;
        }

        if (this->_right != nullptr) {
            Times += this->_right->deleteBAll ();
            this->_right = nullptr;
        }

        Times++;
        delete this;
        return Times;
    }

    int deleteTNode () {
        int Times = 0;
        if (this->_sub != nullptr) {
            Times += this->_sub->deleteTNode ();
            this->_sub = nullptr;
        }

        if (this->_next != nullptr) {
            Times += this->_next->deleteTNode ();
            this->_next = nullptr;
        }

        Times++;
```

```
        delete this;
        return Times;
    }

    int BPreorder () {
        int res = this->getIndex ();
        if (this->getLChild () != nullptr)
            res ^= this->getLChild ()->BPreorder ();
        if (this->getRChild () != nullptr)
            res ^= this->getRChild ()->BPreorder ();
        return res;
    }

    int TPreorder () {
        int res = this->getIndex ();
        if (this->getFirstChild () != nullptr)
            res ^= this->getFirstChild ()->TPreorder ();
        if (this->getNextSibling () != nullptr)
            res ^= this->getNextSibling ()->TPreorder ();
        return res;
    }

    void BTreeToTree () {
        TNode<K,V>* pLeftNode = this->getLChild ();
        if (pLeftNode == nullptr)
            return;
        TNode<K,V>* pnode_in = pLeftNode;
        TNode<K,V>* pRNode;
        while (pnode_in != nullptr) {
            pRNode = pnode_in->getRChild ();
            pnode_in->setRChild (nullptr);
            this->setChild (pnode_in);
            pnode_in->BTreeToTree ();
            pnode_in = pRNode;
        }
    }

    void TreeToBTree () {
        TNode<K,V>* pFirstChild = this->getFirstChild ();
        TNode<K,V>* pNextSibling = this->getNextSibling ();

        this->setLChild (pFirstChild);
        this->setRChild (pNextSibling);
        this->setFirstChild (nullptr);
        this->setNextSibling (nullptr);

        if (pFirstChild != nullptr) pFirstChild->TreeToBTree ();
        if (pNextSibling != nullptr) pNextSibling->TreeToBTree ();
    }
};


/*
template <typename K, typename V>
class Tree {
protected:
    TNode<K, V>* _root;  //指向根节点
    int _size;           //节点数目
public:
    Tree (K index, V data)
    Tree ()
    bool isEmpty ()
    bool setLeft (TNode<K, V>* node_in, K target)
    bool setRight (TNode<K, V>* node_in, K target)
    bool setSub (TNode<K, V>* node_in, K target)
```

```
    bool setSubTo (K nwe_target, V data, K target)
    void addEdge (int a, int b)
    void BTreeToTree ()
    void TreeToBTree ()
    bool deleteTreeNodeByIndex (int index)
    bool deleteTreeNodeByNode (TNode<K, V>* node_in)
    void BPreorder ()
    void TreePreorder ()
};*/
template <typename K, typename V>
class Tree {
public:
    Tree ( K index, V data) : _root (new TNode<K,V> (index, data)), _size (1) {}
    Tree () : _root (new TNode<K,V> (0, 0)), _size (1){}
    ~Tree () {
        if (_root != nullptr) delete _root;
        _root = nullptr;
    }

    bool isEmpty () { return _root; }

    bool setLeft (TNode<K,V>* node_in, K target) {
        if (_root == nullptr) return false;
        TNode<K,V>* temp_node;
        temp_node = _root->BiNodeSearch (target);

        temp_node->setLChild (node_in);
        _size += node_in->getBTChildrenNum ();
        return true;
    }

    bool setRight (TNode<K,V>* node_in, K target) {
        if (_root == nullptr) return false;
        TNode<K,V>* temp_node;
        temp_node = _root->BiNodeSearch (target);

        temp_node->setRChild (node_in);

        return true;
    }

    bool setSub (TNode<K,V>* node_in, K target) {
        if (_root == nullptr)
            return false;

        TNode<K,V>* temp_node;
        temp_node = _root->Tsearch (target);
        temp_node->setChild (node_in);
        return true;
    }

    bool setSubTo (K nwe_target, V data, K target) {
        if (_root == nullptr)
            return false;

        TNode<K,V>* temp_node;
        temp_node = _root->Tsearch (target);

        if (temp_node != nullptr) {
            return addChild (nwe_target, data, temp_node);
        }

        return false;
    }

    void addEdge (int a, int b) {
```

```
        TNode<K,V>* nodeA, * nodeB, * temp_node;

        nodeA = _root->Tsearch (a);
        nodeB = _root->Tsearch (b);
        if (a < b) {
            temp_node = nodeA;
            while (temp_node->getNextSibling () != nodeB)
                temp_node = temp_node->getNextSibling ();

            temp_node->setNextSibling (nodeB->getNextSibling ());
            nodeB->setNextSibling (nullptr);
            nodeB->setParenet (nodeA);
            nodeA->setChild (nodeB);
        } else if (a > b) {
            if (_root->getFirstChild () == nodeB) {
                TNode<K,V>* NextSiblingNode = nodeB->getNextSibling ();
                _root->setFirstChild (NextSiblingNode);
                nodeB->setNextSibling (nullptr);
                nodeA->setChild (nodeB);
            } else {
                temp_node = _root->getFirstChild ();
                while (temp_node->getNextSibling () != nodeB)
                    temp_node = temp_node->getNextSibling ();
                temp_node->setNextSibling (nodeB->getNextSibling ());
                nodeB->setNextSibling (nullptr);
                nodeB->setParenet (nodeA);
                nodeA->setChild (nodeB);
            }
        }
    }

    bool deleteTreeNodeByIndex (int index) {
        TNode<K,V>* deleteNode = _root->Tsearch (index);

        if (deleteNode != nullptr) {
            if (deleteNode == _root) {
                cout << "deleteTreeNodeByIndex():" << index << "是根节点不能删除" << endl;
                return false;
            }
            return deleteTreeNodeByNode (deleteNode);
        }
        return false;
    }

    bool deleteTreeNodeByNode (TNode<K,V>* node_in) {
        if (node_in != nullptr) {
            TNode<K,V>* pFirstChildNode = node_in->getFirstChild ();
            if (pFirstChildNode != nullptr) {
                TNode<K,V>* pChildNode = pFirstChildNode;
                TNode<K,V>* pNextSiblingNode;
                while (pChildNode != nullptr) {
                    pNextSiblingNode = pChildNode->getNextSibling ();
                    _root->setChild (pChildNode);
                    pChildNode = pNextSiblingNode;
                }
            }

            node_in->setFirstChild (nullptr);
            TNode<K,V>* pParentNode = node_in->getParent ();
            TNode<K,V>* pCNode = pParentNode->getFirstChild ();
            if (pCNode == node_in) {
                pParentNode->setFirstChild (node_in->getNextSibling ());
            } else {
                while (pCNode->getNextSibling () != node_in) {
                    pCNode = pCNode->getNextSibling ();
                }
```

```
                pCNode->setNextSibling (node_in->getNextSibling ());
            }

            node_in->setNextSibling (nullptr);
            node_in->setFirstChild (nullptr);
            return true;
        }

        return false;
    }

    void BPreorder () {
        if (_root == nullptr) return;
        cout << _root->getLChild ()->BPreorder () << endl;
    }

    void TreePreorder () {
        if (_root == nullptr) return;

        TNode<K,V>* pFirstNode = _root->getFirstChild ();
        TNode<K,V>* pSiblingNode;

        int res = pFirstNode->getIndex ();
        if (pFirstNode->getFirstChild () != nullptr)
            res ^= pFirstNode->getFirstChild ()->TPreorder ();

        pSiblingNode = pFirstNode->getNextSibling ();
        cout << res << " ";

        while (pSiblingNode != nullptr) {
            res = pSiblingNode->getIndex ();
            if (pSiblingNode->getFirstChild () != nullptr)
                res ^= pSiblingNode->getFirstChild ()->TPreorder ();
            if (pSiblingNode->getNextSibling () != nullptr) {
                cout << res << " ";
            } else {
                cout << res;
            }
            pSiblingNode = pSiblingNode->getNextSibling ();
        }
        cout << endl;
    }

    void BTreeToTree () { _root->BTreeToTree (); }
    void TreeToBTree () { _root->TreeToBTree (); }

protected:
    TNode<K,V>* _root;//指向根节点
    int _size;//节点数目
};

#pragma warning(disable : 4996)
int main () {
    freopen ("in1.txt", "r", stdin);

    int i, j, K, M, N, Q, node, rootIndex, pos;
    Tree<int,int>* tree = new Tree<int,int>;

    TNode<int,int>* nodes = new TNode<int,int>[10000];
    for (i = 0; i < 5102; i++) {
        nodes[i].setIndex (i);
    }

    cin >> K >> M >> N;
```

```
    if (K == 0) {

        int* roots = new int[M];
        int A, B;
        for (i = 0; i < M; i++) cin >> roots[i];

        for (i = 0; i < N; i++) {
            cin >> A >> B;
            for (j = 0; j < B; j++) {
                cin >> node;
                nodes[A].setChild (&nodes[node]);
            }
        }

        for (i = 0; i < M; i++) {
            tree->setSub (&nodes[roots[i]], 0);
        }
    } else {
        int A, l, r;

        cin >> rootIndex;

        for (i = 0; i < N; i++) {
            cin >> A >> l >> r;
            nodes[A].setIndex (A);
            if (l == -1)
                nodes[A].setLChild (nullptr);
            else
                nodes[A].setLChild (&nodes[l]);

            if (r == -1)
                nodes[A].setRChild (nullptr);
            else
                nodes[A].setRChild (&nodes[r]);
        }
        tree->setLeft (&nodes[rootIndex], 0);
    }

    cin >> Q;
    int op, father, a, b;
    for (i = 0; i < Q; i++) {
        cin >> op;
        if (op == 1) {
            cin >> father >> node;
            if (father == -1) {
                tree->setSub (&nodes[node], 0);
            } else {
                tree->setSub (&nodes[node], father);
            }
        } else if (op == 2) {
            cin >> father >> node;
            tree->deleteTreeNodeByIndex (node);
        } else if (op == 3) {
            cin >> a >> b;
            tree->addEdge (a, b);
        } else if (op == 4) {
            if (K == 0) {
                tree->TreeToBTree ();
                K = 1;
            } else {
                tree->BTreeToTree ();
                K = 0;
            }
        } else if (op == 5) {
            cin >> pos >> father >> node;
            if (pos == 0)
```

```
                tree->setRight (&nodes[node], father);
            else
                tree->setLeft (&nodes[node], father);
        } else {
            if (K == 0)
                tree->TreePreorder ();
            else
                tree->BPreorder ();
        }
    }
    return 0;
}
```