Lab4 实验报告

一、设计思路

Student 类：

包括 level、课程数 course、用于存储学分的数组 credit[]和用于存储成绩的数组 grade[]，以及用于获得两个数组的方法 makeArray()和用于计算 level 的方法 getLevel()。

Worker 类：

只包括工资 salary 和 level 两个属性。

Node 类：

作为二叉搜索树的节点，类中有名称 name、level 和左子节点 leftchild 及右子节点 rightchild，包含用于输出节点 displayNode()方法和构造节点的方法。

Binarytree 类：

包含根节点，寻找节点的 findNode()方法、插入节点的 insertNode()方法、删除节点的 deleteNode()方法、先序遍历的 preorderTraverse(Node treenode)方法。

二、与参考类结构的对比

1、 参考类结构有用于获取 level 的接口，将获得数据的过程和计算 level 的过程分别封装在类的方法中

```
interface Rankable{
    int getRank();
    void intputRank(Scanner s);
}
//参考类结构中的定义
    while(treesize>0)
    {
        judge=out.nextInt();
        if(judge==1)
        {
            Student stu=new Student();
            stu.course=out.nextInt();
            stu.makeArray();
            for(int i=0;i<stu.course;i++)
            {
                stu.credit[i]=out.nextInt();
                stu.grade[i]=out.nextInt();
            }
            Node thenode=new Node("Student",stu.getLevel(),null);
            tree.insertNode(thenode);
        }
        else if(judge==2)
```

```
            {
                Worker theworker=new Worker();
                theworker.salary=out.nextInt();
                theworker.level=theworker.salary/100;
                Node thenode2=new Node("Worker",theworker.level,null);
                tree.insertNode(thenode2);
            }
            treesize--;
        }
```

//我定义的类的代码

由于没有封装所以这段代码显得杂乱且不易阅读

2、 参考类结构中定义了 CourseScore 类来储存学分和成绩，而我的代码在 Student 类中使用两个数组分别来存储成绩

```
class CourseScore{
        int Credit;
        int score;
        CourseScore(int Credit,int score){
            this.Credit=Credit;
            this.score=score;
        }
    }
ArrayList<CourseScore> courseList;
```

//参考类结构中的代码

```
class Student{
    int course;
    int level;
    int grade[];
    int credit[];
    public void makeArray()
    {
        this.grade=new int[this.course];
        this.credit=new int[this.course];
    }
}
```

//我的代码

建议的类的结构使用一个类型存储成绩让 Student 类结构更加清晰，也便于写 getLevel 方法，我的代码中使用了两个独立的数组，层次不分明。

3、 参考类的结构中在 Node 类的构造中使用了泛型类，使得 Student 类型和 Worker 能够通用的插入二叉搜索树中

```
class Node<T extends Rankable>
{
    Node preNode;
    T data;
    Node leftChild;
```

```
        Node rightChild;

        public Node(T t,Node preNode) {
            this.preNode = preNode;
            this.data=t;
        }
    }
```

//建议类的结构中 Node 中 data 的数据类型可以是多种类型，使得代码易于改写，便于添加新的类型的节点

```
    class Node{
     String name;
     int level;
     Node leftchild;
     Node rightchild;
     Node lastnode;
     Node(){
         this.name="";
         this.level=0;
     }
     Node(String name,int level,Node thelastnode){
         this.name=name;
         this.level=level;
         this.lastnode=thelastnode;
     }
     public void displayNode()
     {
         System.out.println(this.name+":"+this.level);
     }
}
```

//我实现的代码

由于没有使用泛型类，用 String+int 来形成节点，不便于添加新的类型的节点，也没有显示出类内部的联系。Student 类、Worker 类和 Node 类之间是孤立的。

三、优化类的结构

无

四、源代码

```
 import java.util.*;
 public class Test {
     public static void main(String[] arg)
     {
     Scanner out=new Scanner(System.in);
     int treesize=out.nextInt();
     int judge;
```

```
        Binarytree tree=new Binarytree();
        while(treesize>0)
        {
            judge=out.nextInt();
            if(judge==1)
            {
                Student stu=new Student();
                stu.course=out.nextInt();
                stu.makeArray();
                for(int i=0;i<stu.course;i++)
                {
                    stu.credit[i]=out.nextInt();
                    stu.grade[i]=out.nextInt();
                }
                Node thenode=new Node("Student",stu.getLevel(),null);
                tree.insertNode(thenode);
            }
            else if(judge==2)
            {
                Worker theworker=new Worker();
                theworker.salary=out.nextInt();
                theworker.level=theworker.salary/100;
                Node thenode2=new Node("Worker",theworker.level,null);
                tree.insertNode(thenode2);
            }
            treesize--;
        }
        tree.preorderTraverse(tree.root);
        }
}
class Student{
        int course;
        int level;
        int grade[];
        int credit[];
        public void makeArray()
        {
            this.grade=new int[this.course];
            this.credit=new int[this.course];
        }
        public int getLevel()
        {
            if(this.course==0)
                return 0;
```

```
            int sum=0;
            int sum2=0;
            for(int i=0;i<this.course;i++)
            {
                sum=sum+this.grade[i]*this.credit[i];
                sum2=sum2+credit[i];
            }
            this.level=sum/sum2;
            return this.level;
        }
}
class Worker{
        int salary;
        int level;

}
class Node{
        String name;
        int level;
        Node leftchild;
        Node rightchild;
        Node lastnode;
        Node(){
            this.name="";
            this.level=0;
        }
        Node(String name,int level,Node thelastnode){
            this.name=name;
            this.level=level;
            this.lastnode=thelastnode;
        }
        public void displayNode()
        {
            System.out.println(this.name+":"+this.level);
        }
}
class Binarytree{
        Node root;
        Binarytree(){
            this.root=null;
        }

        public Node findNode(Node treenode)
        {
```

```
        if(this.root==null)
            return this.root;
        Node operate=this.root;
        while(operate!=null)
        {
            if(operate.level>treenode.level&&operate.leftchild!=null)
            {
                operate=operate.leftchild;
            }
            else if(operate.level<=treenode.level&&operate.rightchild!=null)
            {
                operate=operate.rightchild;
            }
            else
                return operate;
        }
        return operate;
    }
    public void insertNode(Node treenode)
    {
        Node thenode=this.findNode(treenode);
        if(thenode==null)
        {
            this.root=new Node(treenode.name,treenode.level,null);
            return ;
        }
        if(treenode.level>=thenode.level)
        {
            thenode.rightchild=new Node(treenode.name,treenode.level,thenode);
        }
        else if(treenode.level<thenode.level)
            thenode.leftchild=new Node(treenode.name,treenode.level,thenode);
        else
            System.out.println("error");
    }
    public Node findPrecursor(Node treenode)
    {
        Node operate=treenode.leftchild;
        while(operate.rightchild!=null)
        {
            operate=operate.rightchild;
        }
        return operate;
    }
```

```
public boolean deleteNode(Node treenode)
{
    if(this.root==null)
        return false;
    Node operate=this.root;
    while(operate!=null)
    {
        if(operate.level>treenode.level&&operate.leftchild!=null)
        {
            operate=operate.leftchild;
        }
        else if(operate.level<treenode.level&&operate.rightchild!=null)
        {
            operate=operate.rightchild;
        }
        else
if(operate.level==treenode.level&&operate.name.equals(treenode.name))
        {
            if(operate.leftchild==null||operate.rightchild==null)
            {
                if(operate.leftchild==null&&operate.rightchild==null)
                {
                    if(operate==operate.lastnode.rightchild)
                        operate.lastnode.rightchild=null;
                    else
                        operate.lastnode.leftchild=null;
                    operate=null;
                }
                else if(operate.leftchild==null)
                {
                    if(operate==operate.lastnode.rightchild)
                        operate.lastnode.rightchild=operate.rightchild;
                    else
                        operate.lastnode.leftchild=operate.rightchild;
                    operate.rightchild.lastnode=operate.lastnode;
                    operate=null;
                }
                else
                {
                    if(operate==operate.lastnode.rightchild)
                        operate.lastnode.rightchild=operate.leftchild;
                    else
                        operate.lastnode.leftchild=operate.leftchild;
                    operate.leftchild.lastnode=operate.lastnode;
```

```
                    operate=null;
                }
            }
            else
            {
                Node precursor=this.findPrecursor(operate);
                operate.level=precursor.level;
                operate.name=precursor.name;
                if(precursor.lastnode.leftchild==precursor)
                    precursor.lastnode.leftchild=precursor.leftchild;
                else
                    precursor.lastnode.rightchild=precursor.leftchild;
                precursor.leftchild.lastnode=precursor.lastnode;
                precursor=null;
            }
            return true;
        }
        else
            return false;
    }
    return false;
}
public void preorderTraverse(Node treenode)
{
    treenode.displayNode();
    if(treenode.leftchild!=null)
        preorderTraverse(treenode.leftchild);
    if(treenode.rightchild!=null)
        preorderTraverse(treenode.rightchild);
}
}
```