```
#pragma warning(disable : 4996)
#include <iostream>
#include <vector>
#include <map>
#include <set>
#include <cmath>
#include <string>
using namespace std;

string characterArray[5] = {"age", "prescipt", "astigmatic", "tearRate", "conclusion"};
string CONCLUSION = "conclusion"; //以上修改可以解决别的问题
int depth = 0;
const enum treeType {
	ID3TYPE,
	C45TYPE,
	CART
};

struct node
{
	string character;                                          //分划子问题的属性
	vector<map<string, int>> statics; //当前节点剩余属性和数据
	map<int, node *> children;                    //保存儿子
	int DFSdepth;                                                    //方便显示打印 tab 数
	int preAnswer;                                                 //方便显示打印分支选项
};

class decisionTree
{
private:
	node root;
	void (*algorithmType)(vector<map<string, int>>, set<string>, node *);

	static inline double H(vector<map<string, int>> glassInfoDivide)
	{
		//熵值计算，向量中存储数据集
		map<int, double> division;
		double answer = 0;
		vector<int> statics;
		for (vector<map<string, int>>::iterator it = glassInfoDivide.begin(); it < glassInfoDivide.end(); ++it)
		{
			//提取出来数据
			statics.push_back((*it)[CONCLUSION]);
		}
		for (vector<int>::iterator it = statics.begin(); it < statics.end(); ++it)
		{
			//数据集分类
			if (division.find(*it) == division.end())
			{
				division.insert(pair<int, double>(*it, 0)); //不在表中插入表中，其实直接[]也行但是初始
化安全一点
			}
			division[*it]++;
		}
		for (map<int, double>::iterator it = division.begin(); it != division.end(); ++it)
		{
			double p = (*it).second / statics.size();
			answer += -p * log(p);
		}
		return answer;
	}
	static inline double H(vector<map<string, int>> InfoDivide, string characterSelect)
	{
```

```
        //得知当前信息后的熵值
        map<int, vector<int>> cDivision; //键为属性值，值为属性值下的数据集
        map<int, map<int, double>> sDivision;
        double answer = 0;
        vector<pair<int, int>> charactorAndStatics;

        for (vector<map<string, int>>::iterator it = InfoDivide.begin(); it < InfoDivide.end(); ++it)
        {
            //提取出来（信息属性-数据）对。该步可以省略，可以直接在提取时分类建表
            charactorAndStatics.push_back(pair<int, int>((*it)[characterSelect], (*it)[CONCLUSION]));
        }

        for (vector<pair<int, int>>::iterator it = charactorAndStatics.begin(); it < charactorAndStatics.end(); ++it)
        {
            //把提取出的数据按属性值分类
            if (cDivision.find((*it).first) == cDivision.end())
            {
                cDivision.insert(pair<int, vector<int>>((*it).first, vector<int>()));
            }
            cDivision[(*it).first].push_back((*it).second);
        }
        for (map<int, vector<int>>::iterator it = cDivision.begin(); it != cDivision.end(); ++it)
        {
            for (vector<int>::iterator it2 = (*it).second.begin(); it2 < (*it).second.end(); ++it2)
            {
                //再对各个属性值下的数据集分类
                if (sDivision[(*it).first].find(*it2) == sDivision[(*it).first].end())
                {
                    sDivision[(*it).first].insert(pair<int, int>(*it2, 0)); //(*it).first 是属性值，每个属性值
对应一个哈希表，哈希表中存了键为当前属性下数据的种类
                }
                sDivision[(*it).first][*it2]++; //和上面一样是一个当前属性 characterSelect 的值(*it).first，
该属性值下的*it2 数据类型的计数量
            }
            //(*it).second.size();
        }

        for (map<int, vector<int>>::iterator it = cDivision.begin(); it != cDivision.end(); ++it)
        {
            for (map<int, double>::iterator it2 = sDivision[(*it).first].begin(); it2 != sDivision[(*it).first].end();
++it2)
            {
                double p = (*it2).second / (*it).second.size();
                answer += -p * log(p) * (*it).second.size() / charactorAndStatics.size();
            }
            //(*it).second.size()各个属性值下的数据数
            //charactorAndStatics.size()所有属性值下的数据总数
            //(*it2).second 为某个属性值下某个数据值出现的数目
        }
        return answer;
    }

    static inline double SplitInformation(vector<map<string, int>> InfoDivide, string characterSelect)
    {
        //得知当前信息后的 SplitInformation
        map<int, vector<int>> cDivision; //键为属性值，值为属性值下的数据集
        map<int, map<int, double>> sDivision;
        double answer = 0;
        vector<pair<int, int>> charactorAndStatics;

        for (vector<map<string, int>>::iterator it = InfoDivide.begin(); it < InfoDivide.end(); ++it)
        {
            //提取出来（信息属性-数据）对。该步可以省略，可以直接在提取时分类建表
```

```
            charactorAndStatics.push_back(pair<int, int>((*it)[characterSelect], (*it)[CONCLUSION]));
        }

        for (vector<pair<int, int>>::iterator it = charactorAndStatics.begin(); it < charactorAndStatics.end(); ++it)
        {
            //把提取出的数据按属性值分类
            if (cDivision.find((*it).first) == cDivision.end())
            {
                cDivision.insert(pair<int, vector<int>>((*it).first, vector<int>()));
            }
            cDivision[(*it).first].push_back((*it).second);
        }

        for (map<int, vector<int>>::iterator it = cDivision.begin(); it != cDivision.end(); ++it)
        {
            double p = (*it).second.size();
            answer += -p * log(p) / charactorAndStatics.size();
            //(*it).second.size()各个属性值下的数据数
            //charactorAndStatics.size()所有属性值下的数据总数
        }
        return answer;
}

static void ID3(vector<map<string, int>> InfoDivide, set<string> charactersRemain, node *treeNode)
{
        treeNode->statics.resize(InfoDivide.size());
        treeNode->statics.assign(InfoDivide.begin(), InfoDivide.end()); //为节点赋数据
        int item = InfoDivide[0][CONCLUSION];
            //即(*(InfoDivide.begin()))[CONCLUSION]
        bool endFlag = false;
        for (vector<map<string, int>>::iterator it = InfoDivide.begin(); it < InfoDivide.end(); ++it)
        {
            if (item != (*it)[CONCLUSION])
            { //判断是否当前已经达到数据值均相同的情况
                endFlag = false;
                break;
            }
            else
            {
                endFlag = true;
            }
        }
        if (endFlag)
        {
            return;
        } //基础情况判断
        else
        {
            double initialH = 0;
            string characters2erase;
            double minHwithInfo = 1;
            map<int, vector<map<string, int>>> divideInfoDivide;

            initialH = H(InfoDivide); //计算初始未知额外信息时数据集的熵值
            //cout << initialH << endl;

            for (set<string>::iterator it = charactersRemain.begin(); it != charactersRemain.end(); ++it)
            {
                //选择当前信息增益最大的一个属性，也就是条件熵值最小的一个
                double tempH = H(InfoDivide, *it); //计算条件熵，initialH-tempH 为信息增益
                if (minHwithInfo > tempH)
                {
                    minHwithInfo = tempH;
```

```
                    characters2erase = *it;
                }
            }
            treeNode->character = characters2erase; //为节点赋属性

            for (vector<map<string, int>>::iterator it = InfoDivide.begin(); it < InfoDivide.end(); ++it)
            {
                //分划当前数据
                if (divideInfoDivide.find((*it)[characters2erase]) == divideInfoDivide.end())
                {
                    divideInfoDivide.insert(pair<int, vector<map<string, int>>>((*it)[characters2erase],
vector<map<string, int>>()));
                    treeNode->children.insert(pair<int, node *>((*it)[characters2erase], new node()));
                }
                divideInfoDivide[(*it)[characters2erase]].push_back(*it);

                for (vector<map<string, int>>::iterator it2 =
divideInfoDivide[(*it)[characters2erase]].begin(); it2 < divideInfoDivide[(*it)[characters2erase]].end(); ++it2)
                {
                    (*it2).erase(characters2erase);
                }
                //(*it)[characters2erase]为数据表每行的某个属性的取值
            }

            charactersRemain.erase(characters2erase); //去除已经判断过的属性
            //cout << characters2erase << endl;
            depth++;
            for (map<int, vector<map<string, int>>>::iterator it = divideInfoDivide.begin(); it !=
divideInfoDivide.end(); ++it)
            { //对每一部分再次使用 ID3
                treeNode->children[(*it).first]->DFSdepth = depth;
                treeNode->children[(*it).first]->preAnswer = (*it).first; //仅仅是用来打印的
                ID3((*it).second, charactersRemain, treeNode->children[(*it).first]);
            }
            depth--;
        }
    }

    static void C45(vector<map<string, int>> InfoDivide, set<string> charactersRemain, node *treeNode)
    {
        treeNode->statics.resize(InfoDivide.size());
        treeNode->statics.assign(InfoDivide.begin(), InfoDivide.end()); //为节点赋数据
        int item = InfoDivide[0][CONCLUSION];
        bool endFlag = false;
        for (vector<map<string, int>>::iterator it = InfoDivide.begin(); it < InfoDivide.end(); ++it)
        {
            if (item != (*it)[CONCLUSION])
            { //判断是否当前已经达到数据值均相同的情况
                endFlag = false;
                break;
            }
            else
            {
                endFlag = true;
            }
        }
        if (endFlag)
        {
            return;
        } //基础情况判断
        else
        {
            double initialH = 0;
```

```
		string characters2erase;
		double minRadioWithInfo = 65536; //取为最大，因为某属性无法区分数据时 SplitInformation 会到 0
		map<int, vector<map<string, int>>> divideInfoDivide;

		initialH = H(InfoDivide); //计算初始未知额外信息时数据集的熵值
		//cout << initialH << endl;

		for (set<string>::iterator it = charactersRemain.begin(); it != charactersRemain.end(); ++it)
		{
			//选择当前信息增益比率最大的一个属性
			double tempH = H(InfoDivide, *it);
			double ratio;
			double split = SplitInformation(InfoDivide, *it);
			if (split != 0 && tempH / split < minRadioWithInfo)
			{
				ratio = tempH / split;
			}
			else
			{ //至少保证了 characters2erase 不是空字符串
				ratio = 65535;
			}
			if (minRadioWithInfo > ratio)
			{
				minRadioWithInfo = ratio;
				characters2erase = *it;
			}
		}
		treeNode->character = characters2erase; //为节点赋属性

		for (vector<map<string, int>>::iterator it = InfoDivide.begin(); it < InfoDivide.end(); ++it)
		{
			//分划当前数据
			if (divideInfoDivide.find((*it)[characters2erase]) == divideInfoDivide.end())
			{
				divideInfoDivide.insert(pair<int, vector<map<string, int>>>((*it)[characters2erase], vector<map<string, int>>()));
				treeNode->children.insert(pair<int, node *>((*it)[characters2erase], new node()));
			}
			divideInfoDivide[(*it)[characters2erase]].push_back(*it);

			for (vector<map<string, int>>::iterator it2 = divideInfoDivide[(*it)[characters2erase]].begin(); it2 < divideInfoDivide[(*it)[characters2erase]].end(); ++it2)
			{
				(*it2).erase(characters2erase);
			}
			//(*it)[characters2erase]为数据表每行的某个属性的取值
		}

		charactersRemain.erase(characters2erase); //去除已经判断过的属性

			//cout << characters2erase << endl;
		depth++;
		for (map<int, vector<map<string, int>>>::iterator it = divideInfoDivide.begin(); it != divideInfoDivide.end(); ++it)
		{ //对每一部分再次使用 C4.5
			treeNode->children[(*it).first]->DFSdepth = depth;
			treeNode->children[(*it).first]->preAnswer = (*it).first; //仅仅是用来打印的
			C45((*it).second, charactersRemain, treeNode->children[(*it).first]);
		}
		depth--;
	}
}
```

```
public:
	decisionTree()
	{
		root.character = "";
		root.DFSdepth = 0;
	}
	decisionTree(vector<map<string, int>> InfoDivide, set<string> charactersRemain, treeType type)
	{
		root.character = ""; //叶节点此项必为空
		root.DFSdepth = 0;
		if (type == ID3TYPE)
		{
			algorithmType = &ID3;
		}
		else if (type == C45TYPE)
		{
			algorithmType = &C45;
		}
		algorithmType(InfoDivide, charactersRemain, &root);
	}

	void displayTree(node *child)
	{
		for (int i = 0; i < child->DFSdepth; ++i)
		{
			cout << "\t";
		}
		cout << child->preAnswer << " " << child->character << " " << child->statics.size() << endl;
		for (map<int, node *>::iterator it = child->children.begin(); it != child->children.end(); ++it)
		{
			displayTree((*it).second);
		}
	}
	void displayTree()
	{
		displayTree(&root);
	}

	int predict(map<string, int> info, node *now)
	{
		if (now->children.empty())
		{
			return (*(now->statics.begin()))[CONCLUSION]; //如果当前节点无儿子那么直接返回当前节点
数据集的结论数据
		}
		else
		{
			int maxChild = 0;
			for (map<int, node *>::iterator it = now->children.begin(); it != now->children.end(); ++it)
			{
				if ((*it).first == info[now->character])
				{
					return predict(info, (*it).second);
				}
				if ((*it).second->statics.size() > now->children[maxChild]->statics.size())
				{
					maxChild = (*it).first;
				}
			}
			//如果没有这个属性分支，则取最大的一个子集中的元素的数据
			return predict(info, now->children[maxChild]);
		}
```

```
	}
	int predict(map<string, int> info)
	{
		return predict(info, &root);
	}
};

int main()
{
	freopen("test.txt", "r", stdin);
	set<string> characters(characterArray, characterArray + sizeof(characterArray) / sizeof(string) - 1);
	vector<map<string, int>> glassInfo;
	map<string, int> glasses; //仅用于临时存储
	int trainNum = 24;				//多少个用来训练，剩余用来测试

	for (int i = 0; i < trainNum; ++i)
	{
		//根据输入数据对每个建立哈希表，即 vector 中每个元素为一行的数据
		for (int j = 0; j < sizeof(characterArray) / sizeof(string); ++j)
		{
			cin >> glasses[characterArray[j]];
		}
		glassInfo.push_back(glasses);
	}
	decisionTree newTree(glassInfo, characters, ID3TYPE);
	newTree.displayTree();

	//以上为训练以下为预测

	for (int i = trainNum; i < 24; ++i)
	{
		//根据输入数据对每个建立哈希表，即 vector 中每个元素为一行的数据
		for (int j = 0; j < sizeof(characterArray) / sizeof(string); ++j)
		{
			cin >> glasses[characterArray[j]];
		}
		glassInfo.push_back(glasses);
		cout << newTree.predict(glassInfo[i]) << endl;
	}
}
```