目录

二叉树的建立与遍历

1. 需求分析

二叉树的基本操作包括了建立与遍历。

（1）输入的形式和输入值的范围：输入先序序列，用英文空格表示第一级 null，输入的数据应为字符。

（2）输出的形式：输出三次遍历的结果

（3）程序所能达到的功能：能够根据先序序列建立一棵二叉树，能够进行三种遍历并打印输出遍历序列。

（4）测试数据：

- 依次输入 A B C_ _ D E_G_ _F_ _ _（‘_’表示空格字符）；

2. 概要设计

（1）抽象数据类型：

ADT BinaryTree{

数据对象 $D=\{a_i \mid a_i \in Elemset, i=1,2,...,n, n\geq 0\}$

数据关系 R:

若 D 为空则为空树；

若 D 不为空则必有一个唯一的根结点，剩余结点可被分为若干不相交的集合，每个集合都是根的子树。

基本操作

InitBiTree(&T);

操作结果：构造空二叉树 T。

DestroyBiTree(&T);

初始条件：二叉树 T 存在。

操作结果：销毁二叉树 T

CreateBiTree(&T,definition);

初始条件：definition 给出二叉树 T 的定义。

操作结果：按 definition 构造二叉树 T。

PreOrdTraverse(&T,Visit());

初始条件：二叉树 T 存在，Visit 是对结点操作的应用函数。

操作结果：先序遍历 T，对每个结点调用函数 Visit 一次且仅一次。一旦 Visit（）失败，则操作失败。

InOrdTraverse(&T,Visit());

初始条件：二叉树 T 存在，Visit 是对结点操作的应用函数。

操作结果：中序遍历 T，对每个结点调用函数 Visit 一次且仅一次。一旦 Visit（）失败，则操作失败。

PostOrdTraverse(&T,Visit());

初始条件：二叉树 T 存在，Visit 是对结点操作的应用函数。

操作结果：后序遍历 T，对每个结点调用函数 Visit 一次且仅一次。一旦 Visit（）失败，则操作失败。

}ADT BinaryTree

（2）本程序包含的函数：

主函数 main( )

创建二叉树函数 Create( )

先序遍历函数 PreOrdTraverse( )

中序遍历函数 InOrdTraverse( )

后序遍历函数 PostOrdTraverse( )

打印函数 PrintNode( )

删除结点函数 DeleteNode( )

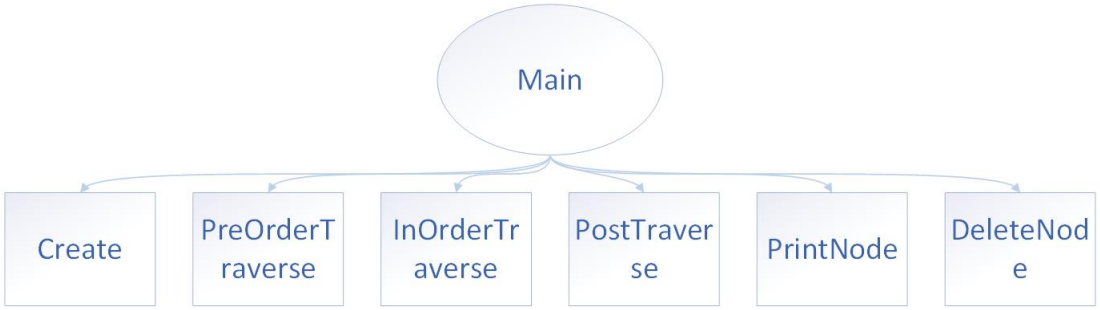


图 1 函数模块及关系图

使用 C++模板类实现二叉树。确定所需提供的可供用户及外部程序方位的公共接口的函数原型，以及相关实现细节部分。

- 构造函数与析构函数
- 基本操作的公共接口，包括引用型操作和修改型操作。
- 数据成员。使用一个指针保存二叉树的根结点的存储位置及状态

3. 详细设计

（1）公共接口。根据需求分析的结果，进一步加入对实现细节的考虑，

例如，对于三种遍历方式，为了减少用户使用复杂度将三个遍历函数整合到一个遍历函数 Traverse( )中。

```
template <typeName T>
class LinkedBiTree
{
private:
    BinaryTreeNode<T> *Root;//根节点

    BinaryTreeNode<T>* PreCreate(T *&PreOrd);
    void PreOrdTraverse(BinaryTreeNode<T>*& Node, void
(*Visit)(BinaryTreeNode<T>*& Node));
    void InOrdTraverse(BinaryTreeNode<T>*& Node, void
(*Visit)(BinaryTreeNode<T>*& Node));
    void PostOrdTraverse(BinaryTreeNode<T>*& Node, void
(*Visit)(BinaryTreeNode<T>*& Node));

public:
    enum { PREORD = 1,INORD=2,POSTORD=3};

    LinkedBiTree(){Root=nullptr;}//构造函数
    ~LinkedBiTree(){PostOrdTraverse(Root, DeleteNode);}//析构函数
    void Create(T *PreOrd){
    Root=PreCreate(PreOrd);
    }
    void Traverse(int mode=PREORD, void (*Visit)(BinaryTreeNode<T>*& Node)
= PrintNode);
};
```

（2）数据成员：类中有一个指针成员 Root 用于标识二叉树的根结点，属于类的实现细节，不能被用户直接访问，为私有成员。有一个枚举类型常量用于用户指示遍历方式和 Traverse( )配合使用，为公有成员。

（3）主要操作的实现。

```
template<typeName T>
BinaryTreeNode<T>* LinkedBiTree<T>::PreCreate(T*& PreOrd)
{
```

```
    if (PreOrd != nullptr)
    {
        static int index = 0;//静态指针用于标记当前指向的序列元素
        if (PreOrd[index] == " ")
        {
            index++;//验证之后指针才能移动
            return nullptr;
        }

        BinaryTreeNode<T>* Node = new BinaryTreeNode<T>;
        Node->data = PreOrd[index++];
        Node->Left = PreCreate(PreOrd);
        Node->Right = PreCreate(PreOrd);
        return Node;
    }
    return nullptr;
}

template<typeName T>
inline void LinkedBiTree<T>::PreOrdTraverse(BinaryTreeNode<T>*& Node,
void(*Visit)(BinaryTreeNode<T>*& Node))
{
    if (Node != nullptr)
    {
        Visit(Node);
        PreOrdTraverse(Node->Left,Visit);
        PreOrdTraverse(Node->Right,Visit);
    }
}

template<typeName T>
inline void LinkedBiTree<T>::InOrdTraverse(BinaryTreeNode<T>*& Node,
void(*Visit)(BinaryTreeNode<T>*& Node))
{
    if (Node != nullptr)
    {
        InOrdTraverse(Node->Left, Visit);
        Visit(Node);
        InOrdTraverse(Node->Right, Visit);
    }
}

template<typeName T>
inline void LinkedBiTree<T>::PostOrdTraverse(BinaryTreeNode<T>*& Node,
```

```
void(*Visit)(BinaryTreeNode<T>*& Node))
{
	if (Node != nullptr)
	{
		PostOrdTraverse(Node->Left, Visit);
		PostOrdTraverse(Node->Right, Visit);
		Visit(Node);
	}
}

template<typeName T>
inline void LinkedBiTree<T>::Traverse(int mode,
void(*Visit)(BinaryTreeNode<T>*& Node))
{
	switch (mode)
	{
		case PREORD:PreOrdTraverse(Root, Visit); break;//先序遍历
		case INORD:InOrdTraverse(Root, Visit); break;//中序遍历
		case POSTORD:PostOrdTraverse(Root, Visit); break;//后续遍历

		default:
			break;
	}
}
```

4. 调试分析

无法正常创建一棵树，经跟踪调试发现是由于先序序列输入错误没有终止符导致递归调用时访问越界。于是在主函数增加了判定条件，只有当空格数=元素数+1 时才进行树的创建，否则进行重复输入。

（1）分析程序代码的质量

- 正确性。在一定的数据范围内，只要输入的是正确的先序序列，该程序能实现所需功能，所以正确性是没有问题的。
- 稳定性。该程序通过 string 类对宽字符输入进行了处理，对于错误的输入会通过循环重新要求输入，故稳定性没有问题。

5. 使用说明

二叉树模板类的全部实现内容均包含在一个独立的 C++头文件 linkedbitree.h 中，要在程序中使用这个类只需包含该头文件即可。这个类实现了创建、遍历等操作。

6.运行结果

依次输入 A B C_ _ D E_G_ _F_ _ _（‘_’表示空格字符），程序运行结果如下：

7 心得体会

（1）对二叉树的递归遍历有了较深刻的认识。进行递归函数的设计时要充分考虑到共同使用的输入参数和返回值。

（2）带有默认参数的形参后面不能有不带默认参数的形参。

计算机目录树的基本操作

1. 需求分析

计算机目录树的基本操作包括了建立目录，修改目录结构，查询和删除等操作。

对目录树的操作实际上就是对二叉树的操作，一个分支为子目录，一个分支为同级目录。

测试数据：

- 输入 mkDir　/Home/My/qq
- 输入 mvDir　/Home/My　/
- 输入 fdDir　/Home
- 输入 fdDir　-c　/
- 输入 fdDir　-f　/Home
- 输入 quick　/My/qq
- 输入 dedesktop
- 输入 rmDir　/My
- 输入 rmDir　-r　/My
- 输入 q

2. 概要设计

（1）抽象数据类型：

ADT DirTree{

数据对象 $D=\{a_i \mid a_i \in Elemset, i=1,2,...,n, n\geq 0\}$

数据关系 R:

若 D 为空则为空树；

若 D 不为空则必有一个唯一的根结点，剩余结点可被分为若干不相交的集合，每个集合都是根的子树。

基本操作

MakeDir(&T,Path);

操作结果：建立目录树

MoveDir(&T,source,target);

初始条件：目录存在

操作结果：移动目录至目标目录

FindDir(&T,Path,param);

初始条件：目录存在

操作结果：查询目录或查询目录的父子目录

ReMoveDir(&T,Path);

初始条件：目录存在且为叶子节点

操作结果：删除目录

Quick(&T,Path);

初始条件：目录存在

操作结构：桌面下创建快捷方式

}ADT DirTree

（2）本程序包含的函数：

主函数 main( )

菜单函数 showMenu( )

创建目录 MakeDir( )

查找目录 FindDir( )

移动目录 MoveDir( )

删除目录 reMoveDir( )

创建快捷方式 quick( )

匹配函数 Search( )

获取父目录 getfPath( )

删除函数 reMove( )

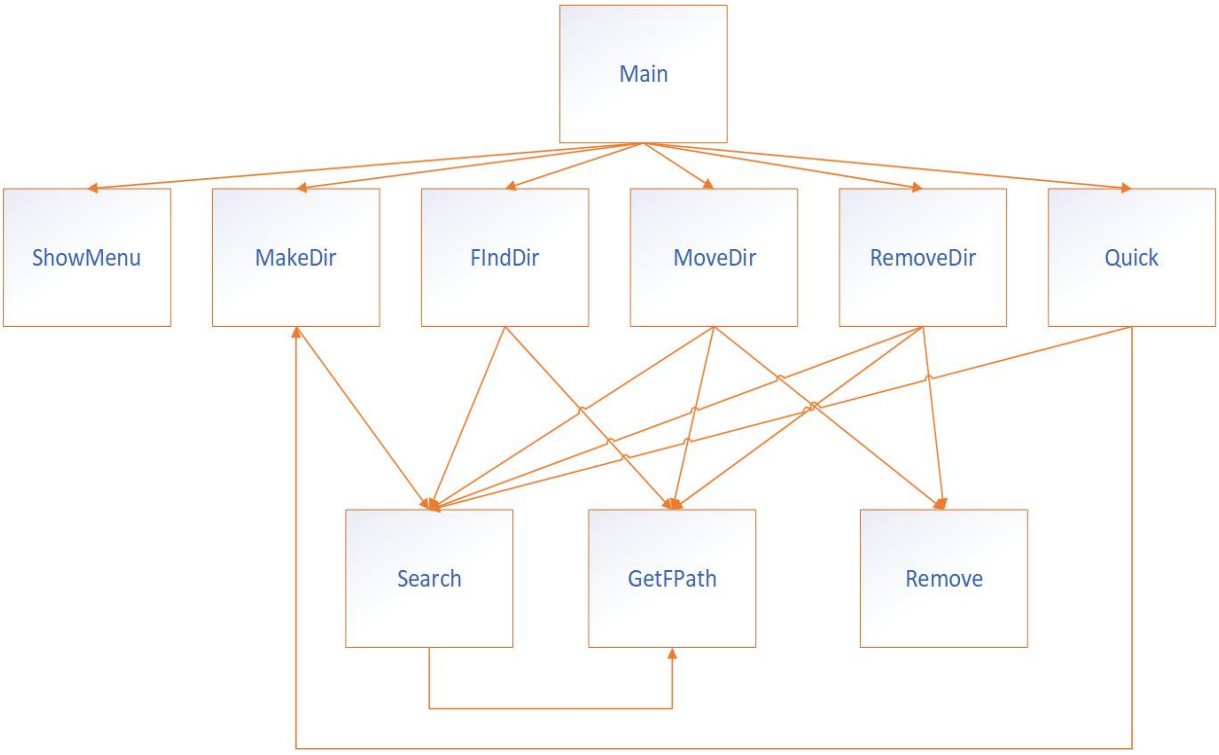


图 2 函数模块及关系图

3. 详细设计

（1）公共接口。根据需求分析的结果，进一步加入对实现细节的考虑，例如在考虑到各操作系统平台对目录操作的习惯，使用输入命令和参数来进行交互。得到目录树类的公共接口如下：

```
class DirTree
{
public:
    DirTree();
    ~DirTree();

    void MakeDir(string Name);                          //创建函数
    void MoveDir(string source, string target);//移动函数
    void reMoveDir(string Path, string param="");   //删除函数
    void FindDir(string Path, string param = ""); //查找函数
    void quick(string Path,string copyloc);       //快捷函数
private:
    Directory* Root;//根目录

    bool Search(Directory* &start, string &Path,int &Flag);     //匹配函数
    bool reMove(Directory* pDir,string Name,string param="");//删除子目录
    void getfPath(string& source, string& finalS);                  //获取父目录
};
```

（2）数据成员：类中有一个数据成员根目录，只能通过成员函数访问，不能直接访问。

（3）主要操作的实现：

```
inline void DirTree::MakeDir(string Name)
{
    int Flag = -1;//Flag 的使用参照 Search 函数说明
    Directory* q=Root;//q 为 p 的上级目录

    //查找匹配目录是否创建，若目录未完全创建就继续完成目录树创建
    if(!Search(q, Name,Flag))
    {
        //字符串流暂存输入字符串
        stringstream ss;
        ss << Name;
```

```
            //按/逐个创建目录
            while (getline(ss, Name, '/'))
            {
                if (Name != "")
                {
                    //创建目录
                    Directory* p = new Directory;
                    p->Name = Name;
                    p->subDir = nullptr;
                    p->nextBrother = nullptr;

                    //插入树中，若 Flag 为 1 说明插入同级目录，为 0 说明插入子目
录
                    if (Flag)
                    {
                        q->nextBrother = p;
                        Flag = 0;//只需插入一次同级目录，之后都是子目录
                    }
                    else
                    {
                        q->subDir = p;
                    }

                    q = p;//q 保留本级目录
                }
            }
        }
}

inline void DirTree::MoveDir(string source, string target)
{
    //目标目录包含源目录说明为源目录或它的子目录
    if (target.Find(source) != string::npos)
    {
        cout << endl << "目标目录不能为本身或其子目录";
        return;
    }

    //找源目录的父目录，先将最终目录单独存储

    string finalS;//存储最终目录
    getfPath(source, finalS);
```

```
    Directory* sDir = Root;
    int sFlag = -1;
    if (!Search(sDir, source, sFlag))
    {
        cout << endl << "未找到要移动的目录的父目录";
        return;
    }

    //确定目标目录位置
    Directory* tDir = Root;
    int tFlag = -1;
    if (!Search(tDir, target, tFlag))
    {
        cout << endl << "未找到目标目录";
        return;
    }

    if (tDir == sDir)
    {
        cout << endl << "目标目录与源目录的父级目录相同，无需移动";
        return;
    }

    //下一个循环查找是否有同名目录
    source = finalS;
    tFlag = -1;//重定义 tFlag 用于存储用户选择
    for (Directory* tsub = tDir->subDir; tsub != nullptr; tsub = tsub->nextBrother)
    {
        if (tsub->Name == source)
        {
            cout << endl << "存在同名目录，是否覆盖(1-是，0-否):";
            while (!(cin >> tFlag) || (tFlag != 0 && tFlag != 1))//输入字母或不为 0
和 1 都会进入循环
            {
                cout << endl << "请重新输入(1-是，0-否):";
                cin.clear();//清除错误标记，重新打开输入流，但是输入流中依旧
保留着之前的不匹配的类型
                cin.ignore(std::numeric_limits<int>::max(), '\n');//忽略输入流中的
一行
            }
            break;
        }
    }
```

```
    if (tFlag == 1)
    {
        //若覆盖则删除被覆盖结点
        reMove(tDir, source, "-r");
    }

    if (!(tFlag == 0))
    {
        //ssub 指向源目录的同级目录
        Directory* ssub = sDir->subDir;

        //Flag 用于标记是否进入第一个循环
        int Flag = 0;

        //如果第一个子目录不是要移动的目录就循环查找后面的目录
        if (ssub->Name != source)
        {
            Flag = 1;
            do
            {
                sDir = ssub;
                ssub = sDir->nextBrother;

            } while (ssub != nullptr && ssub->Name != source);
        }

        //如果循环结束还未找到
        if (ssub == nullptr)
        {
            cout << endl << "未找到要移动的目录";
            return;
        }

        //若第一个或循环后找到了目录
        if (ssub->Name == source)
        {
            //拆链，若经历过循环 Flag=1，sDir 指向同级目录；未经历过循环
Flag=0，sDir 指向父级目录
            Flag ? sDir->nextBrother = ssub->nextBrother : sDir->subDir =
ssub->nextBrother;

            //连接
            ssub->nextBrother = tDir->subDir;
            tDir->subDir = ssub;
```

```
            cout << "移动目录成功" << endl;
            return;
        }
    }

}

inline void DirTree::reMoveDir(string Path, string param)
{
    //找目录的父目录，先将最终目录单独存储
    string finalS;
    getfPath(Path, finalS);

    //定位父目录
    int Flag = -1;
    Directory* fDir = Root;
    if (!Search(fDir, Path, Flag))
    {
        cout << endl << "未找到要删除的目录的父目录";
        return;
    }

    //删除父目录下的最终目录
    Path = finalS;
    if(reMove(fDir, Path, param))
    {
        cout << "删除成功" << endl;
    }

    return;
}

inline void DirTree::FindDir(string Path, string param)
{
    //查找父目录
    if (param == "-f")
    {
        string finalS;
        getfPath(Path, finalS);

        finalS = Path;//拷贝父目录
        Directory* Dir = Root;
        int Flag = -1;
```

```
        if (Search(Dir, Path, Flag))
        {
            Path == "" ? finalS = "/" : 0;//根目录是空目录和根目录的父目录

            cout << "该目录的父目录为：" << finalS << endl;
        }
        else
        {
            cout << endl << "未找到父目录";
        }

        return;
    }

    //查找子目录
    if (param == "-c")
    {
        Directory* Dir = Root;
        string PathB = Path;
        int Flag = -1;

        //若匹配失败
        if (!Search(Dir, Path, Flag))
        {
            cout << endl << "未找到该目录";
            return;
        }

        //若无子目录
        if (Dir->subDir == nullptr)
        {
            cout << endl << "该目录下没有子目录";
            return;
        }

        //循环输出子目录
        cout << PathB << "  的子目录：" << endl;
        for (Dir = Dir->subDir; Dir != nullptr; Dir = Dir->nextBrother)
        {
            cout << Dir->Name << endl;
        }

        return;
```

```
    }

    //查找输入目录
    Directory* Dir = Root;
    string PathB = Path;
    int Flag = -1;

    if (!Search(Dir, Path, Flag))
    {
        cout << endl << "未找到该目录";
    }
    else
    {
        cout << PathB << "  查找成功" << endl;
    }

    return;
}

//函数名称：快捷函数
//功能描述：在目录 copyloc 下创建目录 Path 的快捷方式

{
    //定位源路径
    Directory* Dir = Root;
    int dFlag = -1;
    string finalS=Path;
    if (!Search(Dir, finalS, dFlag))
    {
        cout << endl << "快捷方式的源目录不存在";
        return;
    }

    //创建快捷方式拷贝
    copyloc.append("/");
    copyloc.append(finalS);
    MakeDir(copyloc);

    //定位拷贝路径
    Directory* copyDir=Root;
    int cFlag = -1;
    if (!Search(copyDir, copyloc, cFlag))
    {
        cout << endl << "快捷方式的父目录不存在";
```

```
            return;
        }

        //连接快捷方式和源目录的子目录
        copyDir->subDir = Dir->subDir;

        cout << "创建成功" << endl;
        return;
}
```

4. 调试分析

（1）调试过程中遇到的主要问题及解决过程。

由于对自定义的匹配函数中的循环终止条件和后置条件状态分析不清楚，一刀切的返回方式导致其他调用它的函数在功能上都出现了错误，不能对目录树按照需求修改结构甚至发生异常。原来代码如下

```
    Directory* p = start->subDir;
    while (p != nullptr && getline(ss, Path, '/'))
    {
        if (Path != "")
        {
            //在本级目录循环匹配
            while (p != nullptr)
            {
                if (p->Name == Path)
                {
                    break;
                }
                start = p;
                p = p->nextBrother;
            }
            //若匹配成功，进入下一级
            if (p != nullptr)
            {
                start = p;
                p = start->subDir;
            }
        }
    }
```

由于 start 作为输出参数使用，但是循环结束时 start 有时指向最后一个匹配的父目录，有时指向本级目录链表的最后一个目录，Path 的状态也不唯一，没有加以区分，导致后续调用出错。后来添加了 Flag 进行区分，函数得以正确执行。

（2）分析程序代码的质量

- 正确性。在一定的数据范围内，该程序能实现所需功能，所以正确性是没有问题的。
- 稳定性。该程序能够处理输入数据非法以及匹配失败的情况，稳定性目前没有问题。

5. 使用说明

目录树类及部分操作函数的全部实现内容均包含在一个独立的C++头文件 Dirtree.h 中，要在程序中使用这个类只需包含该头文件即可。这个头文件实现了创建目录、查找目录、删除目录、移动目录、创建快捷方式等操作。目录与子目录之间要使用‘/’或‘//’符号分隔，空格的数量需要严格控制。

6. 测试程序的运行结果

(1) 输入 mkDir　/Home/My/qq

(2) 输入 mvDir　/Home/My　/

(3) 输入 fdDir　/Home

(4) 输入 fdDir　-c　/

(5) 输入 fdDir　-f　/Home

(6) 输入 quick　/My/qq

(7) 输入 dedesktop

(8) 输入 rmDir　/My

(9) 输入 rmDir　-r　/My

(10)　输入 q，退出程序。

发现结果均如预期

Huffman 编码

需求分析

需求：

设某编码系统共有 n 个字符，使用频率分别为 w1, w2, …, wn，设计一个不等长的编码方案，使得该编码系统的空间效率最好。

基本要求：

一个完整的系统应具有以下功能。

(1) I：初始化（Initialization）。从终端读入字符集大小 n，以及 n 个字符和 n 个权值，建立赫夫曼树，并将它存于文件 hfmTree 中。

(2) E：编码（Encoding）。利用已建好的赫夫曼树对文件 ToBeTran 中的正文进行编码，然后将结果存入文件 CodeFile 中。

(3) D：译码（Decoding）。利用已建好的赫夫曼树将文件 CodeFile 中的代码进行译码，结果存入文件 TextFile 中。

(4) P：打印代码文件（Print）。将文件 CodeFile 以紧凑格式显示在终端上，每行 50 个字符。同时将此字符形式的编码文件写入文件 CodePrint 中。

(5) T：打印赫夫曼树（Tree printing）。将已在内存中的赫夫曼树以直观的方式显示在终端上，同时将此字符形式的赫夫曼树写入文件 TreePrint 中。

测试数据:

由读者依据软件工程的测试技术自己确定。注意测试边界数据。

实现提示：

利用赫夫曼编码树求得最佳的编码方案。

(1) 文件 CodeFile 的基类型可以设为字节型。

(2) 用户界面可以设计为“菜单”方式，除显示上述功能符号外，还应显示“Q”（Quit），表示退出运行。请用户键入一个选择功能符。此功能执行完毕后再显示此菜单，直至某次用户选择了“E” 为止。在程序的一次执行过程中，第一次执行 I、D 或 C 命令之后，赫夫曼树已经在内存了，不必再读入。每次执行时不一定执行 I 命令，因为文件 hfmTree 可能早已建好。

概要设计

抽象数据类型：

ADT BinaryTree{

数据对象 D：D 是具有相同特性的数据元素的集合

数据关系 R：

若 D = Φ,则 R = Φ，称 BinaryTree 为空二叉树

若 D ≠ Φ,则 R = {H}，H 有如下二元关系：

（1）在 D 中存在唯一的称为根的数据元素 root,它在关系 H 下无前驱；

（2）若 D-{root} ≠ Φ,则存在 D - {root} = {Dl,Dr},且 Dl ∩ Dr = Φ；

（3）若 Dl ≠ Φ,则 Dl 中存在唯一的元素 xl，<root,xl>∈H,且存在 Dl 上的关系 Hl⊂H；

若 Dr ≠ Φ,则 Dr 中存在唯一的元素 xr，<root,xr>∈H,且存在 Dr 上的关系 Hr⊂H；

H = {<root,xl>，<root,xr>，Hl,Hr}；

（4）(Dl,{Hl})是一棵符合本定义的二叉树，称为根的左子树，

(Dr,{Hr})是一棵符合本定义的二叉树，称为根的右子树；

```
typedef struct{
    int weigth;          //权值
    int parent;          //父母
    int lchild;          //左儿子
    int rchild;          //右儿子
}HTNode,*HuffmanTree;     //哈夫曼树结构
```

基本操作 P:

InitBiTree(&T);

操作结果:构造空的二叉树 T。

DestoryBiTree(&T);

初始条件:二叉树 T 存在。

操作结构:销毁二叉树 T。

CreateBiTree(&T,definition);

初始条件:definition 给出二叉树 T 的定义。

操作结果:按 definition 构造二叉树 T。

ClearBiTree(&T);

初始条件:二叉树 T 存在。

操作结果:将二叉树清为空树。

BiTreeEmpty(T);

初始条件:二叉树 T 存在。

操作结果:若 T 为空二叉树，返回 TRUE，否则返回 FALSE。

BiTreeDepth(T);

初始条件:二叉树 T 存在。

操作结果:返回 T 的深度。

Root(T);

初始条件:二叉树 T 的根存在。

操作结果:返回 T 的根。

Value(T,e);

初始条件:二叉树 T 存在，e 是 T 中某个结点。

操作结果:返回 e 的值。

Assign(T,&e,value);

初始条件:二叉树 T 存在，e 是 T 中的某个结点。

操作结果:结点 e 赋值为 value。

Parent(T,e);

初始条件:二叉树 T 存在，e 是 T 中的结点。

操作结果:若 e 是 T 的非根结点，则返回它的双亲，否则返回"空"。

LeftChild(T,e);

初始条件:二叉树 T 存在，e 是 T 中的某个结点。

操作结果:返回 e 的左孩子。若 e 无左孩子，则返回"空"。

RightChild(T,e);

初始条件:二叉树 T 存在，e 是 T 中的某个结点。

操作结果:返回 e 的右孩子。若 e 无右孩子，则返回"空"。

LeftSibling(T,e);

初始条件:二叉树 T 存在，e 是 T 中的某个结点。

操作结果:返回 e 的左兄弟。若 e 是 T 的左孩子或无左兄弟，则返回"空"。

RightSibling(T,e);

初始条件:二叉树 T 存在，e 是 T 中的某个结点。

操作结果:返回 e 的右兄弟。若 e 是 T 的右孩子或无右兄弟，则返回"空"。

InsertChild(T,p,LR,c);

初始条件:二叉树 T 存在，p 指向 T 中的某个结点，LR 为 0

或 1，非空二叉树 c 与 T 不想交且右子树为空。

操作结果:根据 LR 为 0 或 1，插入 c 为 T 中 p 所指结点的左或右子树。p 所指结点的原有左或右子树则成为 c 的右子树。

DeleteChile(T,p,LR);

初始条件:二叉树 T 存在，p 指向 T 中的某个结点，LR 为 0 或 1.

操作结果:根据 LR 为 0 或 1，删除 T 中 p 所指结点的左或右子树。

PreOrderTraverse(T,visit());

初始条件:二叉树 T 存在，visit()是对结点操作的应用函数。

操作结果:先序遍历 T，对每个结点调用函数 visit 一次且仅一次。一旦 visit 失败，则操作失败。

InOrderTraverse(T,visit());

初始条件:二叉树 T 存在，visit()是对结点操作的应用函数。

操作结果:中序遍历 T，对每个结点调用函数 visit 一次且仅一次。一旦 visit 失败，则操作失败。

PostOrderTraverse(T,visit());

初始条件:二叉树 T 存在，visit()是对结点操作的应用函数。

操作结果:后序遍历 T，对每个结点调用函数 visit 一次且仅一次。一旦 visit 失败，则操作失败。

LevelOrderTraverse(T,visit());

初始条件:二叉树 T 存在，visit()是对结点操作的应用函数。

操作结果:层序遍历 T，对每个结点调用函数 visit 一次仅且一次。一旦 visit 失败，则操作失败。

}ADT BinaryTree

本程序包含的函数：

主函数 main( )

InitBiTree(&T)

DestoryBiTree(&T)

CreateBiTree(&T,definition)

ClearBiTree(&T)

BiTreeEmpty(T)

BiTreeDepth(T)

Root(T)

Value(T,e)

Assign(T,&e,value)

Parent(T,e)

LeftChild(T,e)

RightChild(T,e)

LeftSibling(T,e)

RightSibling(T,e)

InsertChild(T,p,LR,c)

DeleteChile(T,p,LR)

PreOrderTraverse(T,visit())

InOrderTraverse(T,visit())

PostOrderTraverse(T,visit())

LevelOrderTraverse(T,visit())

详细设计

准备工作

定义哈夫曼树结构

```
typedef struct {
    int weigth;          //权值
    int parent;          //父母
    int lchild;          //左儿子
    int rchild;          //右儿子
    //静态三叉链表
}HTNode, * HuffmanTree;     //哈夫曼树结构
```

选择权值最小的

```
void select(HuffmanTree p, int n, int* a, int* b)
{
    *a = 0;
    *b = 0;//两个指针先初始化
```

```
for (int z = 1; z <= n; z++)//遍历
{
 if (p[z].parent == 0 && *a == 0)//如果父节点为空，且*a 还没赋值
 {
      *a = z;
      continue;
 }
 if (p[z].parent == 0)
 {
      *b = z;
      break;
 }
}
if (p[*a].weigth >= p[*b].weigth)//a 的权值大于等于 b
{
 int i = *a;
 *a = *b;
 *b = i;//a,b 互换
}
for (int m = 1; m <= n; m++)//遍历
{
 if (p[m].parent != 0)//如果父节点已经不为 0 了
```

```
        {
            continue;//跳过 （舍弃掉）
        }
        if (p[*a].weigth > p[m].weigth && *a != m && *b != m)
        {
            *b = *a;
            *a = m;
        }
        else if (p[*b].weigth > p[m].weigth && *b != m && *a != m)
        {
            *b = m;
        }
    }
}
```

建立哈夫曼树

```
void HuffmanCoding(HuffmanTree* HT, int* w, int n)
{
    if (n <= 1)
    {
     return;//如果不超过 1 个字符，直接返回
    }
```

```
int m = 2 * n - 1;//n 个叶子节点，需要结合 n-1 次，共有 2n-1 个节点
*HT = (HuffmanTree)malloc((m + 1) * sizeof(HTNode));//为树分配空间 （第一个位置不用）
int i = 1;//用于 for 循环
HuffmanTree p = *HT + 1;//指向已有的树的后一个空间（用于遍历树）
for (; i <= n; ++i, ++p, ++w)//遍历
{
 //对前 n 个节点位置，用来放有效的编码
 p->lchild = 0;
 p->parent = 0;
 p->rchild = 0;
 p->weigth = *w;//初始化
}
for (; i <= m; ++i, ++p)
{
 //n 之后的位置 ，放用来结合的空根
 p->lchild = 0;
 p->parent = 0;
 p->rchild = 0;
 p->weigth = 0;
```

```
    }
    for (int k = n + 1; k <= m; ++k)
    {
     //n+1 的空根开始遍历，直到所有的根遍历完成
     int s1, s2;
     select(*HT, k - 1, &s1, &s2);//k-1 代表的是选择的上限位置
     (*HT)[s1].parent = k;
     (*HT)[s2].parent = k;//把选中的两个权值最小的置为 k 的左右孩子
     (*HT)[k].lchild = s1;
     (*HT)[k].rchild = s2;
     (*HT)[k].weigth = (*HT)[s1].weigth + (*HT)[s2].weigth;//修改 k 的
权值
    }
}
```

编码

```
void coding(char*** p, int n, HuffmanTree t)
{
    *p = (char**)malloc((n + 1) * sizeof(char*));//分配空间
    char* cd;
    cd = (char*)malloc(n * sizeof(char));//分配空间
    cd[n - 1] = '\0'; //字符串
```

```
    for (int i = 1; i <= n; ++i)//遍历
    {
     int start = n - 1;
     for (int c = i, f = t[i].parent; f != 0; c = f, f = t[f].parent)
     {
         if (t[f].lchild == c)
         {
             //左孩子就赋值为 0
             cd[--start] = '0';
         }
         else
         {
             //右孩子就赋值为 1
             cd[--start] = '1';
         }
     }
     (*p)[i] = (char*)malloc((n - start) * sizeof(char));
     strcpy((*p)[i], &cd[start]);
    }
    free(cd);
}
```

译码

```
void decoding(FILE* r, FILE* w, HuffmanTree t, int n, char* a)       //译码
{
    int num = 2 * n - 1;//根节点
    for (int i; fscanf(r, "%1d", &i) != EOF;)
    {
     if (i == 0)
     {
         int j = t[num].lchild;//0 就是左孩子
         if (t[j].lchild == 0)//如果已经没有子树了（此处不一定为左孩子，因为左右成对出现）
         {
             fprintf(w, "%c", a[j - 1]);//输出
             num = 2 * n - 1;//重新给 num
         }
         else
         {
             num = j;//如果还有左孩子，又继续从 j 的位置往下
         }
     }
     else if (i == 1)
```

```
    {
        int j = t[num].rchild;//1 就是右孩子
        if (t[j].rchild == 0)//如果已经没有子树了
        {
            fprintf(w, "%c", a[j - 1]);//输出
            num = 2 * n - 1;
        }
        else
        {
            num = j;
        }
    }
  }
}
```

主要操作的实现

打印哈夫曼树

```
void visit(HuffmanTree t, FILE* w, int n)                    //打印哈夫曼树
{
   HuffmanTree p = t + 1;
   for (int m = 0; m < 2 * n - 1; m++)
```

```
    {
     printf("%d\t%d\t%d\t%d\n",  p->weigth,  p->parent,  p->lchild,
p->rchild);
     fprintf(w,  "%d  %d  %d  %d\n",  p->weigth,  p->parent,  p->lchild,
p->rchild);
     ++p;
    }
}
```

读取边

```
void read_weigth(int* w, char* ch, int number)                    //读取边
{
    char q;
    for (int n = 0; n < number; n++)
    {
     scanf("%d%c", &w[n], &ch[n]);//权值-字符
     scanf("%c", &q);//空格
     //权值存在 w,字符存在 ch
    }
}
```

将字符串转化为二进制代码

```
void found(FILE* r1, FILE* w2, char** p, int n, char* zh)        //将字符串转化为二进制代码
{
    //r1 为原文，w2 为编码加密之后的文,r2 为哈夫曼树
    char ch;
    //     fscanf(r2,"%s",zh);//先读取全部的编码
    for (; (fscanf(r1, "%c", &ch)) != EOF;)
    {//读取原文
     int i = 0;
     while (zh[i] != ch)//找到对应的编码
     {
          i++;
     }
     fprintf(w2, "%s", p[i + 1]);//写到 r2 文件
    }
}
```

调试分析

遇到的主要问题：

如何定义哈夫曼树的数据结构，需要存储哪些信息（例如权值、双亲、左右孩子、字符等）；

如何实现哈夫曼编码和译码的算法，需要用到哪些数据结构（例如数组、堆、队列等）；

如何处理边界情况，例如空字符集、单个字符、相同权值的字符等；

如何评估编码效率，需要计算哪些指标（例如平均码长、压缩比等）；

如何优化代码性能，需要考虑哪些因素（例如时间复杂度、空间复杂度、递归深度等）。

改进设想：

使用自适应霍夫曼编码，它可以动态地调整二叉树的结构，以适应字符出现频率的变化，从而提高编码效率。

使用算术编码，它可以将整个字符序列看作一个分数，然后用二进制数来表示这个分数，从而实现更高的压缩比。

使用混合编码，它可以结合不同的编码方法，例如霍夫曼编码和游程长度编码，以利用不同类型的数据特征，从而提高编码性能。

分析程序代码的质量：

正确性：在一定数据范围内，本程序可以正确实现所有功能，有正确性。

健壮性和安全性：代码中没有进行错误处理或异常处理，例如分配空间失败、输入文件为空或不存在、输入数据不符合要求等情况。可以考虑添加一些判断语句或异常处理机制来增强代码的健壮性和安全性。

高效性：利用了霍夫曼编码算法的优点，能够根据字符出现频率分配最优前缀编码，从而实现无损压缩和解压缩。使用动态内存分配和释放来避免浪费空间，循环和递归来遍历霍夫曼树，并利用了文件操作来读写数据，没有使用不必要的变量或语句来增加复杂度或运行时间。

可扩展性：可以适应不同大小和种类的输入数据，只要提供相应的霍夫曼树和原始字符数组即可；可方便地修改或增加功能，比如添加错误处理、日志记录、进度显示等；与其他模块或程序协作，比如使用其他算法来生成或解析霍夫曼树等。

灵活性：可以在不同平台和环境下运行，只要有支持 C 语言和文件操作的编译器和系统即可；可根据用户需求或偏好进行定制或优化，比如调整内存分配策略、编码长度、输出格式等。

心得体会：

通过实现以上的代码，我对哈夫曼编码有了更深入的理解和体会。哈夫曼编码是一种无损数据压缩算法，它根据输入字符的出现频率，为每个字符分配不同长度的编码。这样，出现频率高的字符就可以用更少的比特表示，从而减少了文件的总大小。哈夫曼编码的核心是构建一棵哈夫曼树，它是一种特殊的前缀码树，即任何一个字符的编码都不是另一个字符编码的前缀，这样可以避免在解码过程中产生歧义。在我的代码中，我用一个结构体数组来存储哈夫曼树的节点信息，包括字符、频率、父节点、左孩子和右孩子，用一个优先队列来存储每个节点的指针，并按照频率从小到大排序，用一个循环来构建哈夫曼

树，每次从优先队列中取出两个最小频率的节点，将它们作为新节点的左右孩子，并将新节点加入到优先队列中。当优先队列中只剩下一个节点时，就是哈夫曼树的根节点。在编码函数中，我用一个二维字符数组来存储每个字符的编码，并用一个一维字符数组来临时存储编码过程中生成的字符串。循环来遍历每个字符，从叶子节点开始向上回溯，根据父节点和孩子节点之间的关系来确定编码是 0 还是 1，并将其插入到字符串的前面。当回溯到根节点时，就得到了该字符的完整编码，并将其复制到二维数组中。在译码函数中，用两个文件指针分别指向输入文件和输出文件，并用一个整数变量来记录当前访问到的哈夫曼树节点。用一个循环来读取输入文件中的每一位数字，并根据数字是 0 还是 1 来判断应该向左还是向右移动到下一个节点等等.....

在实现的过程中，我对哈夫曼编码和二叉树的结构都有了更深刻的认识和体会，对数据结构这门课程也有了更深刻的认识，感到收益良多。

1. 测试结果

运行结果：

```
请选择: I:初始化    E:编码    D:译码    P:打印代码文件    T:打印赫夫曼树    Q:退出
I
请输入字符集大小: 5
请输入5个权值和字符: 1a 2b 4c 8d 9e
初始化成功, 结果存在文件'hfmTree.txt'中!

请选择: I:初始化    E:编码    D:译码    P:打印代码文件    T:打印赫夫曼树    Q:退出
T
weigth  parent  lchild  rchild
1       6       0       0
2       6       0       0
4       7       0       0
8       8       0       0
9       9       0       0
3       7       1       2
7       8       6       3
15      9       7       4
24      0       5       8
打印赫夫曼树成功, 结果存在文件'TreePrint.txt'中
```

图的深度遍历与广度遍历

一．需求分析

以邻接表为存储结构，实现连通无向图的深度优先和广度优先遍历。以用户指定的结点为起点，分别输出每种遍历下的结点访问序列和相应生成树的边集。

在主程序中提供下列菜单：

(1)“1”代表图的建立；

(2)“2”代表深度优先遍历图；

(3)“3”代表广度优先遍历图；

(4)“0”代表结束。

题目分析：

（1) 输入的形式和输入值的范围：字符与 int 型数字

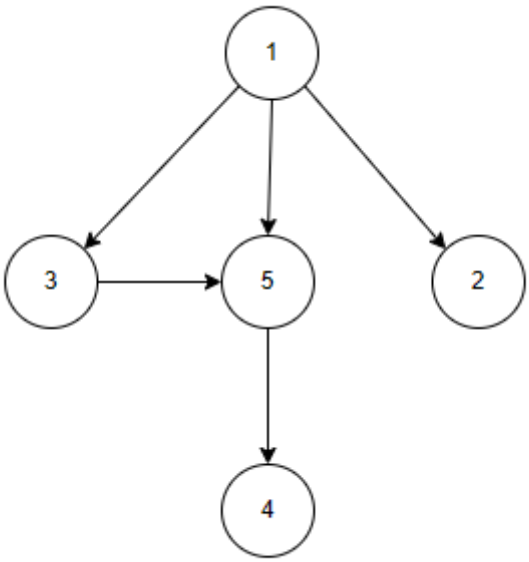
（2) 输出的形式：输出对应遍历的结果字符串。

（3) 程序所能达到的功能：

- 建立图
- 深度优先遍历图
- 广度优先遍历图

（4) 测试数据：

构建如下的图



二．设计思路

（1） 抽象数据类型：

ADT Graph{

数据对象：V 是具有相同特性的数据元素的集合，称为顶点集。

数据关系：

R={VR}

VR =（<v，w>|v，w 属于 v 且 P（v，w）<v，w>表示从 v 到 w 的弧）

基本操作：

CreateGraph（&G，v，VR）

初始条件：V 是图的顶点集，VR 是图中弧的集合。

操作结果：按 V 和 VR 的定义构造图 G

DestroyGraph（&G）

初始条件：图 G 存在。

操作结果：销毁图。

G LocateVex（G，u）

初始条件：图 G 存在，u 和 G 中顶点有相同特征。

操作结果：若 G 中存在顶点 u，则返回该顶点在图中的位置；否则返回其他信息。

GetVex（G，v）

初始条件：图 G 存在.v 是 G 中某个顶点。

操作结果：返回 v 的值。

PutVex（&G，v，value）

初始条件：图 G 存在，v 是 G 中某个顶点。

操作结果：对 v 赋值 value 。

InsertVex（&G，v）

初始条件：图 G 存在，v 和图中顶点有相同特征。

操作结果：在图 G 中增添新顶点 v

DeleteVex（&G，v）

初始条件：图 G 存在，v 是 G 中某个顶点。

操作结果：删除 G 中顶点 v 及其相关的弧

InsertArc（&G，v，w）

初始条件：图 G 存在，v 和 w 是 G 中两个顶点。

操作结果：在 G 中增添弧<v，w>，若 G 是无向图，则还增添对称弧<w，v>

DeleteArc（&G，v，w）

初始条件：图 G 存在，v 和 w 是 G 中两个顶点。

操作结果：在 G 中删除弧<v，w>、若 G 是无向图，则还删除对称弧<w，v.

DFSTraverse（G）

初始条件：图 G 存在。

操作结果：对图进行深度优先遍历，在遍历过程中对每个顶点访问一次。

BFSTraverse（G）

初始条件：图 G 存在。

操作结果：对图进行广度优先遍历，在遍历过程中对每个顶点访问一次。

}ADT Graph

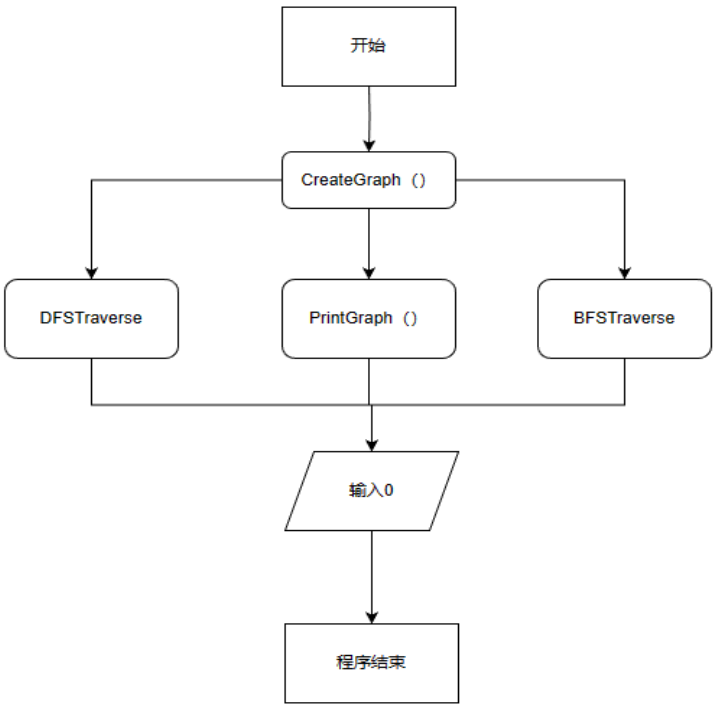


图 1：程序结构示意图

（2） 主程序的流程：

主程序 int main（）

建立图 void createGraph（）

深度优先遍历 void DFSTraverse（）

广度优先遍历 void DFSTraverse（）

三． 详细设计

(1) 主要操作的实现：

//函数名称：建立图的邻接表

//函数功能描述：新建一个图的邻接表

//预备条件：已有初始化的图

//返回后的处理：无

//返回值：无

```
//函数的输入参数：已有初始化图的地址
//函数的输出参数：无
//函数的抽象算法：无
//调用关系：无
void createGraph(Graph &G){
    int i,j,k; //辅助变量
    ALnode *p1,*p2; //辅助结点
    cout<<"输入图的顶点数:";
    cin>>G.n;
    cout<<"输入图的边数:";
    cin>>G.e;
    cout<<endl; //换行
    cout<<"输入图的各顶点(存储序号从 1 开始)："<<endl;
    for(i=0;i<G.n;i++){   //生成有 n 个顶点的顶点表
        cout<<"第"<<i+1<<"个顶点信息：";
        cin>>G.adjlist[i].data; //顶点数据存入表头
        G.adjlist[i].firstal=NULL; //边表头指针域置为空
    }
    cout<<endl; //换行
    cout<<"输入图中的边，顶点序号从 1 开始:"<<endl;
    for(k=0;k<G.e;k++){
        cout<<endl; //换行
        cout<<"输入第"<<k+1<<"条边:"<<endl;
        cout<<"输入出发顶点的序号：";
        cin>>i;
        i--;
        cout<<"输入指向顶点的序号:";
        cin>>j;
        j--;
```

```
            //邻接表存储连接
            p1=(ALnode *)malloc(sizeof(ALnode)); //分配存储空间
            p1->adjvex=j; //指向顶点的序号存入邻接点数据域
            p1->next=G.adjlist[i].firstal; //新的结点的指针域置为空
            G.adjlist[i].firstal=p1; //新结点信息依次存入邻接表中

            p2=(ALnode *)malloc(sizeof(ALnode)); //分配存储空间
            p2->adjvex=i; //指向顶点的序号存入邻接点数据域
            p2->next=G.adjlist[j].firstal; //新的结点的指针域置为空
            G.adjlist[j].firstal=p2; //新结点信息依次存入邻接表中
        }
}


//函数名称：输出邻接表
//函数功能描述：输出已有图的邻接表
//预备条件：已有初始化的图
//返回后的处理：无
//返回值：无
//函数的输入参数：已初始化的图
//函数的输出参数：无
//函数的抽象算法：无
//调用关系：无
void printGraph(Graph G){
    int i; //辅助变量
    ALnode *p; //辅助结点
    cout<<"邻接表中的存储内容如下所示："<<endl;
    for(i=0;i<G.n;i++){
        cout<<i<<' '<<G.adjlist[i].data; //输出表头结点的数据
```

```
            p=G.adjlist[i].firstal; //指向下一结点
            while(p!=NULL){
                cout<<"--->"<<p->adjvex<<' '; //顺次输出结点信息
                p=p->next;
            }
            cout<<endl; //换行
        }
}

//函数名称：图的深度优先遍历
//函数功能描述：输出深度优先遍历已有的图
//预备条件：已有初始化的图
//返回后的处理：无
//返回值：无
//函数的输入参数：已初始化的图，起点结点
//函数的输出参数：无
//函数的抽象算法：无
//调用关系：无
void DFSTraverse(Graph G,int v){

    ALnode *p; //辅助结点
    cout<<G.adjlist[v].data<<' '; //输出顶点信息
    visited[v] = 1;
    p=G.adjlist[v].firstal; //访问第 v 个顶点
    while(p!=NULL){
        if(visited[p->adjvex]==0){
            DFSTraverse(G,p->adjvex);
        }
        p=p->next;
    }
```

```
}

//函数名称：图的广度优先遍历
//函数功能描述：输出广度优先遍历已有的图
//预备条件：已有初始化的图
//返回后的处理：无
//返回值：无
//函数的输入参数：已初始化的图，起点结点
//函数的输出参数：无
//函数的抽象算法：无
//调用关系：无
void BFSTraverse(Graph G,int v){

    int i,j,visited[MAX]; //辅助变量、标志数组
    ALnode *p; //辅助结点
    int queue[MAX],front=0,rear=0; //定义循环队列
    for(i=0;i<G.n;i++){
        visited[i]=0; //标志数组信息初始化
    }
    cout<<G.adjlist[v].data<<' '; //输出顶点信息
    visited[v]=1; //对应顶点的标志置为 1
    rear=(rear+1)%MAX; //队尾指针后移
    queue[rear]=v; //查找的顶点对应序号入队列
    //循环遍历
    while(front!=rear){
        front=(front+1)%MAX; //队头指针后移
        j=queue[front]; //从队列中取出顶点对应序号
        p=G.adjlist[j].firstal; //取对应序号的顶点信息
        while(p!=NULL){
            if(visited[p->adjvex]==0){
```

```
                    visited[p->adjvex]=1;

cout<<"("<<p->adjvex<<","<<G.adjlist[p->adjvex].data<<")"<<' '; //输出顶点信息
                    rear=(rear+1)%MAX; //队尾指针后移
                    queue[rear]=p->adjvex; //查找的顶点对应序号入队列
                }
                p=p->next;
            }
        }
    }


    void show(){
        cout<<"请选择你要执行的操作："<<endl;
        cout<<"0.退出"<<endl;
        cout<<"1.创建有向图（采用邻接表存储结构）"<<endl;
        cout<<"2.深度优先遍历"<<endl;
        cout<<"3.广度优先遍历"<<endl;
    }

    //主函数
    int main(){
        Graph G; //定义图结构变量
        int v1,v2,choose;
        show();
        cin>>choose;
        while(choose!=0){
            switch(choose){
```

```
case 1:{
        createGraph(G); //创建有向图
        printGraph(G); //输出
        break;
}
case 2:{
        cout<<"输入从哪个顶点开始遍历(输入序号)：";
        cin>>v1;
        v1--;
        DFSTraverse(G,v1);
        for(int i=0;i<G.n;i++){
              visited[i]=0; //标志数组信息初始化
        }
        cout<<endl;
        break;
}
case 3:{
        cout<<"输入从哪个顶点开始遍历(输入序号)：";
        cin>>v2;
        v2--;
        BFSTraverse(G,v2);
        for(int i=0;i<G.n;i++){
              visited[i]=0; //标志数组信息初始化
        }
        cout<<endl;
        break;
}
default:{
        cout<<"输入错误，请重新选择！"<<endl;
        show();
```

```
                }
            }
            show();
            cin>>choose;
        }
    }
```

四．调试分析

（1）调试过程中遇到的困难及分析

图的结构：首先应该定义表头结点，然后是边表结点，最后是图的结构。

边集的表示：DFS,BFS 搜索生成的边集需要定义一个结构 Arc，存储搜索结点的顺序。

图的创建：利用头插法，将表头结点和边表结点联系起来。

深度优先：需要利用 visit 数组来判断该节点是否被访问过，然后利用递归来实现 DFS。

广度优先：需要定义一个队列的数据结构，同时也要利用 visit 数组来判断结点是否又被访问过，同时也需要递归来实现。

（2）对设计和编码的讨论和分析。

该程序实现了图的操作。分析程序代码的质量，主要从以下几个方面考虑。

正确性。在一定的数据范围内，该程序能实现所需功能，所以正确性是没有问题的。

健壮性。在一定的数据输入范围内，该程序能较好的实现图的操作。但是如果输入数据非法，该程序还是可能会产生一些预想不到的输出结构，或是不做任何处理。所以，该程序的健壮性有待进一步的提高。要综合考虑一些情况，当输入有误时，应返回一个表示错误的值，并中止程序的执行，以便在更高的抽象层次上进行处理。

五．使用说明

根据对话框中的提示，可分别输入：

“1”：创建有向图

“2”：实现深度优先遍历

“3”：实现广度优先遍历

“0”：退出程序

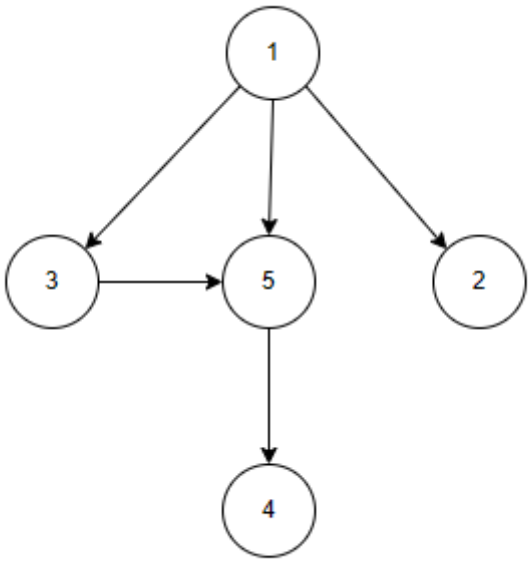
输入“1”后有二级操作：

输入顶点数—输入边数—依次输入各个顶点的信息-依次输入各个有向边。

在执行 1 后方可执行 2、3 操作。

六．运行结果

输入 1，并按以下图输入该图的顶点和边的信息



再输入 2、3，得到结果如下：

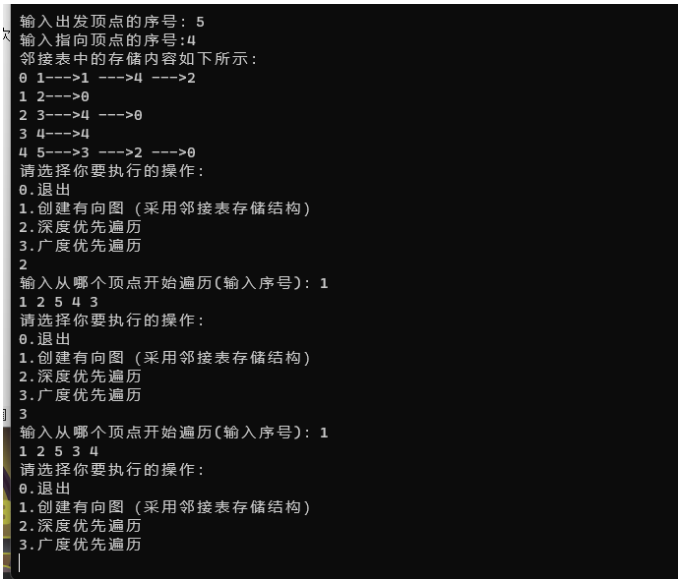


图 2：任务一结果图

可以看到，输出结果与与其结果相吻合，证明了该程序的正确性与可用性。

七．心得体会

（1） 更加深刻地体会到了递归地调用逻辑，以及对将队列等结构用于递归中的使用实例的更真切的了解。

（2） 学会了如何实现图的两种遍历方式。

最小生成树的造价问题

一、 需求分析

若在 n 个城市之间建通信网络，只需架设 n−1 条线路即可。如何以最低的经济代价建设这个通信网是一个网的最小生成树问题。以邻接表或邻接矩阵为存储结构，利用 Prim 算法或 Kruskal 算法求网的最小生成树。

（1） 输入的形式与输入值的范围：

输入集为 ascII 码。输入内容为所选的操作、图的顶点数、边数以

及边上的权值。

（2） 输出的形式：

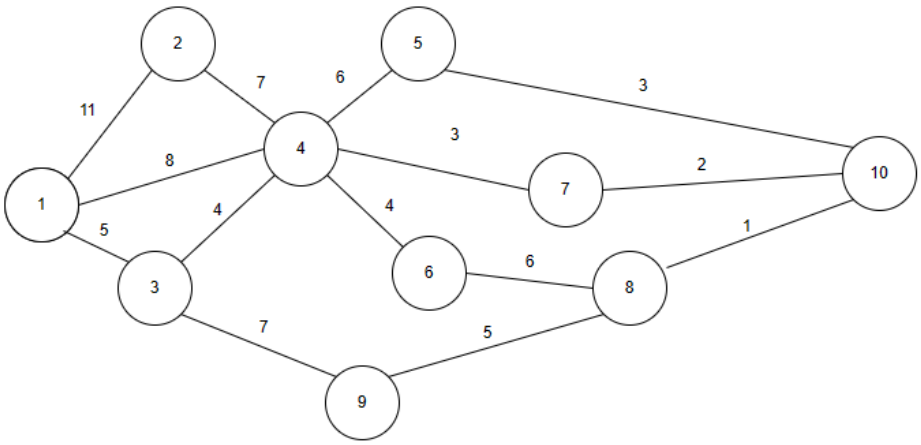
所求图的最小生成树和最小生成树的权值。

（3） 程序所能达到的功能：

- 创建一个无向图
- 对已有无向图求最小生成树
- 对已有无向图求最小生成树的权值

（4） 测试数据

构建以下的图



二、 设计思路

（1） 抽象数据类型：

ADT Graph{

数据对象：V 是具有相同特性的数据元素的集合，称为顶点集。

数据关系：

R={VR}

VR =（<v，w>|v，w 属于 v 且 P（v，w）<v，w>表示从 v 到 w 的弧）

基本操作：

CreateGraph（&G，v，VR）

初始条件：V 是图的顶点集，VR 是图中弧的集合。

操作结果：按 V 和 VR 的定义构造图 G

DestroyGraph（&G）

初始条件：图 G 存在。

操作结果：销毁图。

G LocateVex（G，u）

初始条件：图 G 存在，u 和 G 中顶点有相同特征。

操作结果：若 G 中存在顶点 u，则返回该顶点在图中的位置；否则返回其他信息。

GetVex（G，v）

初始条件：图 G 存在.v 是 G 中某个顶点。

操作结果：返回 v 的值。

PutVex（&G，v，value）

初始条件：图 G 存在，v 是 G 中某个顶点。

操作结果：对 v 赋值 value 。

InsertVex（&G，v）

初始条件：图 G 存在，v 和图中顶点有相同特征。

操作结果：在图 G 中增添新顶点 v

DeleteVex（&G，v）

初始条件：图 G 存在，v 是 G 中某个顶点。

操作结果：删除 G 中顶点 v 及其相关的弧

InsertArc（&G，v，w）

初始条件：图 G 存在，v 和 w 是 G 中两个顶点。

操作结果：在 G 中增添弧<v，w>，若 G 是无向图，则还增添对称弧<w，v>

DeleteArc（&G，v，w）

初始条件：图 G 存在，v 和 w 是 G 中两个顶点。

操作结果：在 G 中删除弧<v，w>、若 G 是无向图，则还删除对称弧<w，v.

DFSTraverse（G）

初始条件：图 G 存在。

操作结果：对图进行深度优先遍历，在遍历过程中对每个顶点访问一次。

BFSTraverse（G）

初始条件：图 G 存在。

操作结果：对图进行广度优先遍历，在遍历过程中对每个顶点访问一次。

}ADT Graph

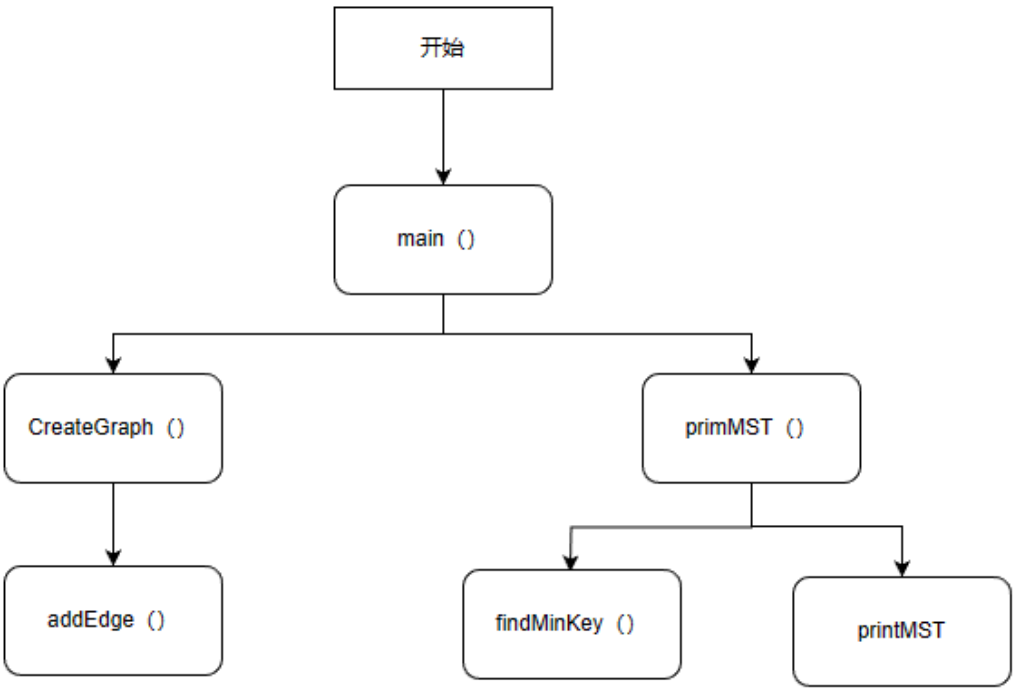


图 3：任务二程序结构示意图

（2） 主程序的流程：

主程序 int main（）

建立图 void CreateGraph（）

添加边 void addEdge（）

求最小生成树 void primMST（）

寻找最小键值 findMinKey（）

打印最小生成树 printMST（）

三、 详细设计

（1） 主要操作的实现：

```
//函数名称：建立图的邻接表(添加顶点)
//函数功能描述：新建一个图的邻接表
//预备条件：已有初始化的图
//返回后的处理：无
//返回值：无
//函数的输入参数：已有初始化图的地址，顶点数目
//函数的输出参数：无
//函数的抽象算法：无
//调用关系：无
void CreateGraph(Graph* graph, int numNodes) {
    graph->numNodes = numNodes;
    for (int i = 0; i < numNodes; i++) {
        for (int j = 0; j < numNodes; j++) {
            graph->adjacencyMatrix[i][j] = INF; //  初始化所有边的权值为无穷大
        }
    }
}
//函数名称：建立图的邻接表（添加边）
//函数功能描述：新建一个图的邻接表
//预备条件：已有初始化的图
//返回后的处理：无
//返回值：无
//函数的输入参数：已有初始化图的地址，边的端点及其权值
//函数的输出参数：无
//函数的抽象算法：无
//调用关系：无
```

```
void addEdge(Graph* graph, int src, int dest, int weight) {
    graph->adjacencyMatrix[src][dest] = weight;
    graph->adjacencyMatrix[dest][src] = weight; // 由于是无向图，所以双向边的权值相同
}

//函数名称：查找键值最小的节点
//函数功能描述：查找键值最小的节点
//预备条件：已有建立好的图
//返回后的处理：无
//返回值：无
//函数的输入参数：已有初始化图的地址，键值数组，结点数
//函数的输出参数：无
//函数的抽象算法：无
//调用关系：无
int findMinKey(int key[], bool mstSet[], int numNodes) {
    int minKey = INF;
    int min = -1;

    for (int i = 0; i < numNodes; i++) {
        if (!mstSet[i] && key[i] < minKey) {
            minKey = key[i];
            min = i;//标记未访问最小键值所在的位置
        }
    }

    return min;//返回最小键值位置
}

// 输出最小生成树
```

```
//函数名称：求解最小生成树
//函数功能描述：使用 Prim 算法求解最小生成树
//预备条件：已有建立好的图
//返回后的处理：无
//返回值：无
//函数的输入参数：已有初始化图的地址
//函数的输出参数：无
//函数的抽象算法：无
//调用关系：findMinKey（），printMST（）
void printMST(int parent[], int numNodes, int adjacencyMatrix[MAX_NODES][MAX_NODES]) {
    cout << "最小生成树的边及其权值："<<endl;
    for (int i = 1; i < numNodes; i++) {//从第二个开始输出双亲结点和权值
        cout << "("<<parent[i] + 1 <<"->" << i + 1 << ") 权值为："<< adjacencyMatrix[i][parent[i]]<<endl;
        add+=adjacencyMatrix[i][parent[i]];
    }
    cout << "总代价为：";
    cout << add<<endl;
}

//函数名称：求解最小生成树
//函数功能描述：使用 Prim 算法求解最小生成树
//预备条件：已有建立好的图
//返回后的处理：无
//返回值：无
//函数的输入参数：已有初始化图的地址
//函数的输出参数：无
//函数的抽象算法：无
//调用关系：findMinKey（），printMST（）
```

```
void prim(Graph* graph) {
    int parent[MAX_NODES]; // 存储最小生成树中每个节点的父节点
    int key[MAX_NODES]; // 存储节点的键值
    bool mstSet[MAX_NODES]; // 标记节点是否已加入最小生成树

    for (int i = 0; i < graph->numNodes; i++) {
        key[i] = INF; // 初始化所有节点的键值为无穷大
        mstSet[i] = false; // 初始化所有节点的标记为 false
    }
    key[0] = 0; // 将第一个节点作为起始节点
    parent[0] = -1; // 根节点没有父节点

    for (int count = 0; count < graph->numNodes; count++) {
        int u = findMinKey(key, mstSet, graph->numNodes);
// 选择键值最小的节点加入最小生成树
        mstSet[u] = true; // 将该节点标记为已加入最小生成树

        for (int v = 0; v < graph->numNodes; v++) {
            if (graph->adjacencyMatrix[u][v] != INF && !mstSet[v] &&
graph->adjacencyMatrix[u][v] < key[v]) {
                // 更新节点 v 的键值和父节点
                key[v] = graph->adjacencyMatrix[u][v];
                parent[v] = u;
            }
        }
    }
    printMST(parent, graph->numNodes, graph->adjacencyMatrix);
}
```

四、 调试分析

（1） 调试过程中遇到的困难及分析：

Prim 算法的实现：在实现过程利用 findMinKey 函数寻找最小键值，并利用数组将已连接的结点记录起来以实现集合的功能。

（2） 对设计和编码的讨论和分析：

该程序实现了图的建立和最小生成树的寻找，分析代码质量，可从以下几个方面考虑：

正确性。在一定的数据范围内，该程序能实现所需功能，所以正确性是没有问题的。

健壮性。在一定的数据输入范围内，该程序能较好的实现图的操作。但是如果输入数据非法，该程序还是可能会产生一些预想不到的输出结构，或是不做任何处理。所以，该程序的健壮性有待进一步的提高。考虑到某些特殊的输入，该程序可能出现死循环导致的非正常运行情况。要综合考虑一些情况，当输入有误时，应返回一个表示错误的值，并中止程序的执行，以便在更高的抽象层次上进行处理。

五、 使用说明

根据对话框中的提示，可分别输入：

“1”：创建有向图

“2”：获取最小生成树

“0”：退出程序

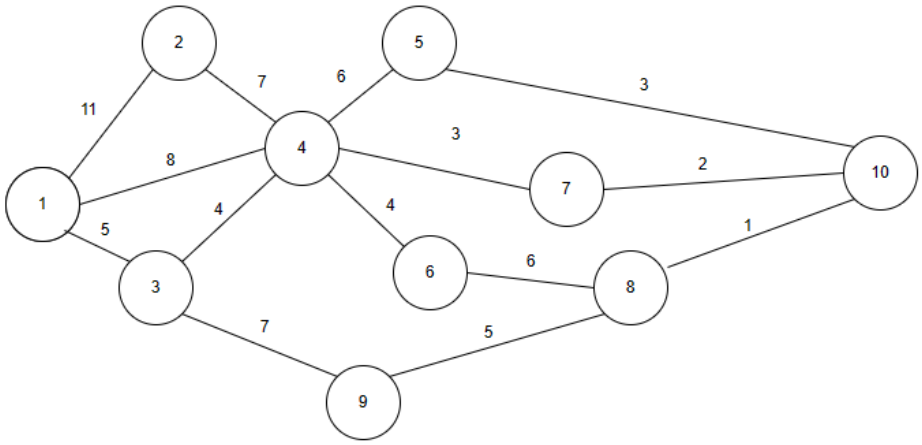
输入“1”后有二级操作：

输入顶点数—输入边数-依次输入各个有向边及其权值。

在执行 1 后方可执行 2 操作。

六、 运行结果

输入 1，并按以下图输入该图的顶点和边的信息



再输入 2，得到结果如下：

```
请输入第7条边的信息：3 9 7
请输入第8条边的信息：9 8 5
请输入第9条边的信息：8 10 1
请输入第10条边的信息：4 7 3
请输入第11条边的信息：4 5 6
请输入第12条边的信息：5 10 3
请输入第13条边的信息：6 8 6
请输入第14条边的信息：4 6 4
操作菜单：
0.退出
1.创建有向图（采用邻接表存储结构）
2.获得最小生成树
请选择你要执行的操作：2
最小生成树的边及其权值：
(4->2) 权值为：7
(1->3) 权值为：5
(3->4) 权值为：4
(4->5) 权值为：6
(4->6) 权值为：4
(4->7) 权值为：3
(10->8) 权值为：1
(8->9) 权值为：5
(5->10) 权值为：3
总代价为：38
操作菜单：
0.退出
1.创建有向图（采用邻接表存储结构）
2.获得最小生成树
请选择你要执行的操作：0
请按任意键继续. . .
```

图 4：任务二结果示意图

再输入 0，程序结束。

可以看到，输出结果与与其结果相吻合，证明了该程序的正确性与可用性。

七、 心得体会

（1） 学会了 Prim 算法的实现，深刻地理解到 Prim 算法的具体实现。

导游问题

一、 需求分析

给出一张某公园的导游图，游客通过终端查询可得：

1. 每个景点的名称和相应的介绍
2. 从某一景点到另一景点的最短路径
3. 游客从大门进入后一条最佳的游玩路径，使得游客可以不重复地游玩所有景点

题目分析：

（1） 输入的形式与输入值的范围：

（2） 输入集为 ascII 码。输入内容为所选的操作、图的顶点数、边数以及边上的权值。

（3） 输出的形式：

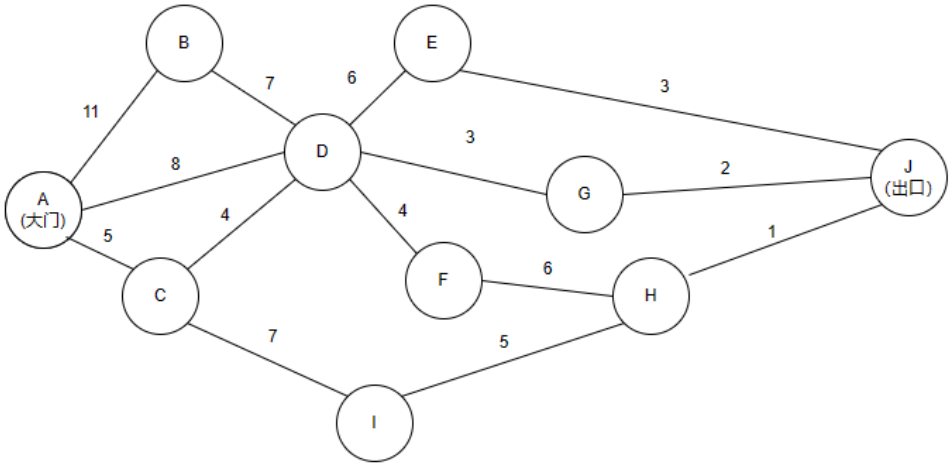
顶点的信息和两个顶点之间的最短路径

（4） 程序所能达到的功能：

- 创建一个无向图
- 查询顶点的信息
- 查询两个顶点之间的最短路径

（5） 测试数据

构建以下的图



二、 设计思路

（1） 抽象数据类型

ADT Graph{

数据对象：V 是具有相同特性的数据元素的集合，称为顶点集。

数据关系：

R={VR}

VR =（<v，w>|v，w 属于 v 且 P（v，w）<v，w>表示从 v 到 w 的弧）

基本操作：

CreateGraph（&G，v，VR）

初始条件：V 是图的顶点集，VR 是图中弧的集合。

操作结果：按 V 和 VR 的定义构造图 G

DestroyGraph（&G）

初始条件：图 G 存在。

操作结果：销毁图。

G LocateVex（G，u）

初始条件：图 G 存在，u 和 G 中顶点有相同特征。

操作结果：若 G 中存在顶点 u，则返回该顶点在图中的位置；否则返回其他信息。

GetVex（G，v）

初始条件：图 G 存在.v 是 G 中某个顶点。

操作结果：返回 v 的值。

PutVex（&G，v，value）

初始条件：图 G 存在，v 是 G 中某个顶点。

操作结果：对 v 赋值 value 。

InsertVex（&G，v）

初始条件：图 G 存在，v 和图中顶点有相同特征。

操作结果：在图 G 中增添新顶点 v

DeleteVex（&G，v）

初始条件：图 G 存在，v 是 G 中某个顶点。

操作结果：删除 G 中顶点 v 及其相关的弧

InsertArc（&G，v，w）

初始条件：图 G 存在，v 和 w 是 G 中两个顶点。

操作结果：在 G 中增添弧<v，w>，若 G 是无向图，则还增添对称弧<w，v>

DeleteArc（&G，v，w）

初始条件：图 G 存在，v 和 w 是 G 中两个顶点。

操作结果：在 G 中删除弧<v，w>、若 G 是无向图，则还删除对称弧<w，v.

DFSTraverse（G）

初始条件：图 G 存在。

操作结果：对图进行深度优先遍历，在遍历过程中对每个顶点访问一次。

BFSTraverse（G）

初始条件：图 G 存在。

操作结果：对图进行广度优先遍历，在遍历过程中对每个顶点访问一次。

}ADT Graph

（2） 主程序的流程：

主函数 main（）

构创/析构函数 MatrixUDG（）

获取顶点序号 GetPosition（）

利用 Dijkstra 算法获取最短路径 Dijkstra（）

获取最佳游览路径 BestPath（）

获取顶点信息 GetImformation（）

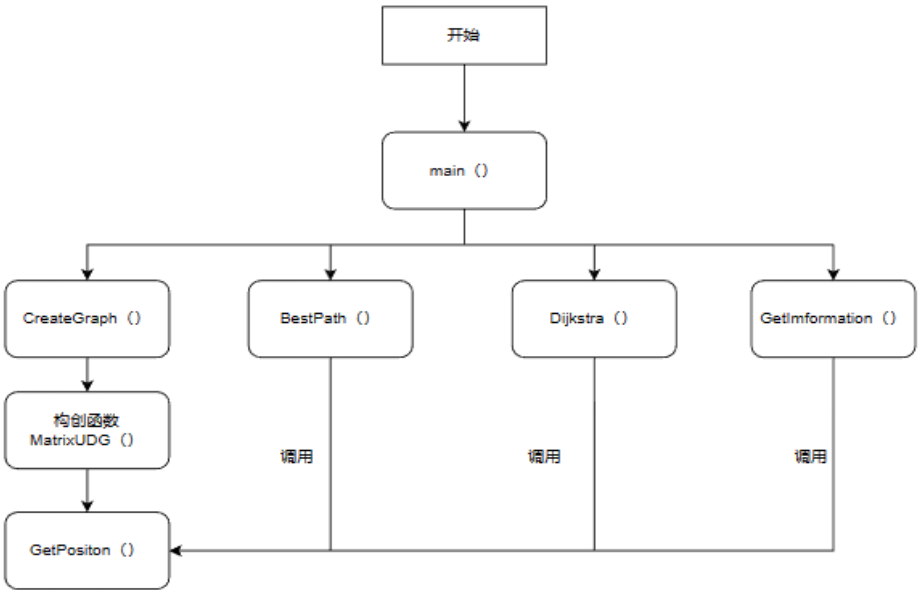


图 5：任务三程序结构示意图

使用 C++模板类实现链表。确定所需提供的可供用户及外部程序方位的公共接口的函数原型，以及相关实现细节部分。

- 构造函数与析构函数
- 基本操作的公共接口，包括引用型操作和修改型操作
- 数据成员。一个 int 数组寄存顶点集合，一个 int 数组记录某顶点的直接前驱，一个 string 数组寄存顶点信息，一个二维 int 数组寄存该图的邻接矩阵，两个 int 分别寄存顶点数和边数。

三、 详细设计

（1） 公共接口。

根据需求分析的结果，进一步加入对实现细节的考虑，例如 delete 函数删除元素，在实现细节中，进一步细分为按位置删除和按元素的值删除。另外，加上 C++类设计中的正规函数（包括构造函数、析构函数等），得到的该无向图类的公共接口如下：

/类名称：MatrixUDG

//定义该类的目的：便于完成求图种两个顶点的最短路径，获取顶点信息的功能，并保持对象的稳定

//类属性：无

```
constexpr auto MAX = 100;
class MatrixUDG {
private:
    char mVexs[MAX];            // 顶点集合
    int mVexNum;                // 顶点数
    int mEdgNum;                // 边数
    int mMatrix[MAX][MAX];   // 邻接矩阵
    int path[MAX];          //path[j]用于存放 j 结点的前驱
    string imformation[MAX];//顶点信息
public:
    MatrixUDG();// 创建图（自己输入数据）
    ~MatrixUDG();// 析构函数
    void print();// 打印矩阵
    void Dijkstra(char i,char j);//Dijkstra 算法
    void BestPath();//获取最短路径
    void GetImformation(char i);//获取结点信息
private:
    // 读取一个输入字符
    char readChar();
    // 返回 ch 在 mMatrix 矩阵中位置
    int getPosition(char ch);
    void find(int x,int p);
};
```

（2） 数据成员：

一个 int 数组寄存顶点集合，一个 int 数组记录某顶点的直接前驱，一个 string 数组寄存顶点信息，一个二维 int 数组寄存该图的邻接矩阵，两个 int 分别寄存顶点数和边数。

（3） 主要操作的实现：

```
//构造函数
```

```
//函数名称：创建无向图
//函数功能描述：创建无向图
//预备条件：已有初始化的无向图对象
//返回后的处理：无
//返回值：无
//函数的输入参数：无
//函数的输出参数：无
//函数的抽象算法：无
//调用关系：GetPositon（）、readchar（）
MatrixUDG::MatrixUDG() {
    char c1, c2;
    int i,j, p1, p2;
    // 输入顶点数和边数
    cout << "请输入顶点数目：";
    cin >> mVexNum;
    cout << "请输入边数目: ";
    cin >> mEdgNum;
    if (mVexNum < 1 || mEdgNum < 1 || (mEdgNum > (mVexNum *
(mVexNum - 1)))) {
        cout << "输入错误！请重试！" << endl;
        return;
    }

    // 初始化顶点
    for (i = 0; i < mVexNum; ++i) {
        cout << "顶点(" << i << ")及其信息:";
        mVexs[i] = readChar();
        cin >> imformation[i];
    }
```

```
        // 初始化边
        // 先把所有起始顶点到其他顶点的边初始化为最大值
        for (i = 0; i < mVexNum; i++)
            for (j = 0; j < mVexNum; ++j)
                mMatrix[i][j] = 99999;

        for (i = 0; i < mEdgNum; ++i) {
            // 读取边的起始顶点和结束顶点
            cout << "边(" << i << ")及其权值:";
            c1 = readChar();
            c2 = readChar();

            p1 = getPosition(c1);
            p2 = getPosition(c2);
            if (p1 == -1 || p2 == -1) {
                cout << "输入错误！请重试！" << endl;
                return;
            }
            int weight;
            cin >> weight;
            mMatrix[p1][p2] = weight;
            mMatrix[p2][p1] = weight; //无向图
            mMatrix[p1][p1] = 0;
            mMatrix[p2][p2] = 0;
        }
    }

    /*
     * 析构函数
     */
```

```
MatrixUDG::~MatrixUDG(){

}

//函数名称：获取顶点信息
//函数功能描述：获取顶点信息
//预备条件：已有已创建的无向图
//返回后的处理：无
//返回值：无
//函数的输入参数：所需求的顶点
//函数的输出参数：所需求的顶点的信息
//函数的抽象算法：无
//调用关系：GetPositon（）
void MatrixUDG::GetImformation(char i){
    int p1;
    if(p1=getPosition(i)==-1)
        cout << "该点不在地图上！请确认后重试！" <<endl;
    else
    cout << imformation[p1]<<endl;
}

//函数名称：获取最短路径
//函数功能描述：通过 Dijkstra 算法获取顶点信息
//预备条件：已有已创建的无向图
//返回后的处理：无
//返回值：无
//函数的输入参数：所需求的两个顶点
//函数的输出参数：所需求的两个顶点之间的最短路径
//函数的抽象算法：无
```

```
//调用关系：GetPositon（）、find（）
void MatrixUDG::Dijkstra(char i,char j){
    int p1,p2;
    p1 = getPosition(i);
    p2 = getPosition(j);
    int dis[mVexNum];
    int book[mVexNum];



    for(int k=0;k<mVexNum;k++){
        dis[k]=mMatrix[p1][k];
        book[k]=0;
    }
    book[p1]=1;
    dis[p1]=0;

    for(int k=0;k<mVexNum;k++){
        int min=99999,u=p1;
        for(int r=0;r<mVexNum-1;r++){
            if((!book[r])&&min>dis[r]){
                min=dis[r];
                u=r;
            }
        }
        book[u]=1;
        for(int r=0;r<mVexNum;r++){
            if((!book[r])&&(dis[r]>dis[u]+mMatrix[u][r])){
                dis[r]=dis[u]+mMatrix[u][r];
                path[r]=u;
            }
```

```
            }
        }
        find(p2,p1);

    }

    //函数名称：获取结点序号
    //函数功能描述：通过结点的名字获取结点序号（在 mMatrix 矩阵中的位置）
    //预备条件：已有已创建的无向图
    //返回后的处理：无
    //返回值：无
    //函数的输入参数：所需求的个顶点的名字
    //函数的输出参数：返回 ch 在 mMatrix 矩阵中的位置
    //函数的抽象算法：无
    //调用关系：无
    int MatrixUDG::getPosition(char ch) {
        int i;
        for (i = 0; i < mVexNum; ++i)
            if (mVexs[i] == ch)
                return i;
        return -1;
    }

    //函数名称：读入字符
    //函数功能描述：读入字符
    //预备条件：无
    //返回后的处理：无
    //返回值：无
```

```
//函数的输入参数：无
//函数的输出参数：输入的字符
//函数的抽象算法：无
//调用关系：无
char MatrixUDG::readChar() {
    char ch;
    do {
        cin >> ch;
    } while (!((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z')));
    return ch;
}

//函数名称：打印邻接矩阵
//函数功能描述：打印邻接矩阵
//预备条件：已有已创建的无向图
//返回后的处理：无
//返回值：无
//函数的输入参数：无
//函数的输出参数：无
//函数的抽象算法：无
//调用关系：无
void MatrixUDG::print(){
    int i, j;
    cout << "Martix Graph:" << endl;
    for (i = 0; i < mVexNum; i++){
        for (j = 0; j < mVexNum; j++)
            cout << mMatrix[i][j] << " ";
        cout << endl;
    }
}
```

```
//函数名称：寻找最短路径
//函数功能描述：通过两个结点序号寻找两个结点之间的最短路径并打印
//预备条件：已有已创建的无向图
//返回后的处理：无
//返回值：无
//函数的输入参数：所需求的两个顶点
//函数的输出参数：所需求的两个顶点之间的最短路径
//函数的抽象算法：无
//调用关系：find（）
void MatrixUDG::find(int x,int p){
    if(path[x]==p){
        cout << mVexs[p];
    }
    else{
        find(path[x],p);
    }
    cout << "-->" << mVexs[x];
    return ;
}
```

（1） 调试过程中遇到的困难及分析：

Dijsktra 算法的实现：在实现过程利用 book、dis 数组寻找最小路径，并以书上的思路完成算法。

对于输入的顶点名称和输入的边，为了避免输入出现意外的错误，选着采用 readChar（）函数稳定输入，提高程序的稳定性。

（2） 对设计和编码的讨论和分析：

该程序实现了图的建立和最小路径的寻找，分析代码质量，可从以下几个方面考虑：

正确性。在一定的数据范围内，该程序能实现所需功能，所以正确性是没有问题的。

健壮性。在一定的数据输入范围内，该程序能较好的实现图的操作。但是如果输入数据非法，该程序还是可能会产生一些预想不到的输出结构，或是不做任何处理。所以，该程序的健壮性有待进一步的提高。考虑到某些特殊的输入，该程序可能出现死循环导致的非正常运行情况。要综合考虑一些情况，当输入有误时，应返回一个表示错误的值，并中止程序的执行，以便在更高的抽象层次上进行处理。

四、 使用说明

根据对话框中的提示，程序开始时要依次输入：

输入顶点数—输入边数-依次输入各个顶点及其信息-依次输入各个有向边及其权值。

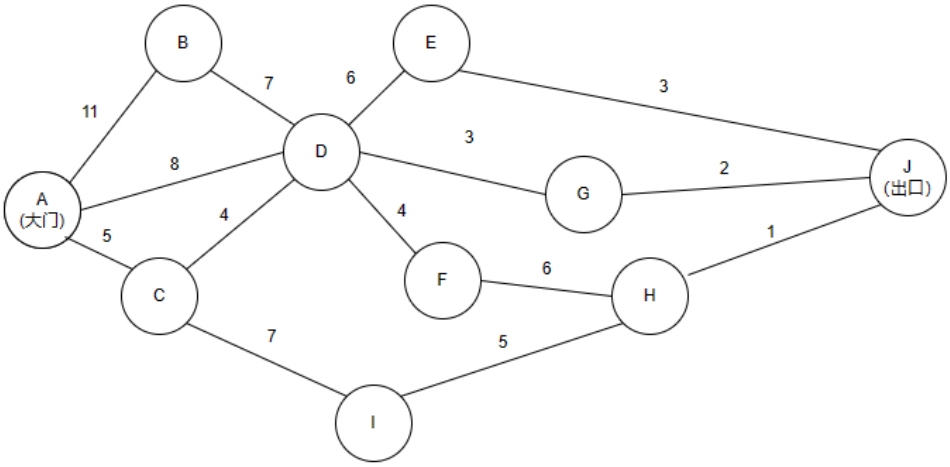
“1”：查询结点信息

“2”：查询最短路径

“0”：退出程序

五、 测试结果

程序开始时，按以下图输入该图的顶点和边的信息



输入 1，再输入 B，查询 B 景点的信息；

输入 2，再输入 A J，查询 A、J 之间的最短路径，得到结果如下：

图 6：任务三运行结果

再输入 0，程序结束。

可以看到，输出结果与与其结果相吻合，证明了该程序的正确性与可用性。

六、 心得体会

纸上得来终觉浅，绝知此事要躬行