天津大学

# 数据结构实验报告

实验名称：图

学院名称 智能与计算学部

专　　业 软件工程

学生姓名 陈昊昆

学　　号 3021001196

年　　级 2021 级

班　　级 软工 3 班

时　　间 2023 年 5 月 18 日

# 1. 实验内容

实现使用邻接表存储的无向图。边节点类 EdgeNode 、顶点节点类 VertexNode 和图类 MyGraph ， 邻接表采用线性表存储，起点节点使用数组存储，同一起点节点的边节点使用链表存储，完成以下 功能：
边节点类的初始化和销毁、私有属性存取等基本功能
顶点节点类的初始化和销毁、私有属性存取等基本功能
图类的初始化和销毁、私有属性存取等基本功能
图的带参数初始化 MyGraph(int, int, char*, int*, int*) ，
参数列表为：
顶点数目 int
边数目 int
顶点名称数组 char* ，
长度等于顶点数目，不会出现重复的名称
边的起点顶点下标数组 int* （从 0 开始，无向图每条边只出现一次），长度等于边数目 边的终点顶点下标数组 int* （从 0 开始），长度等于边数目

图的打印， string printGraph() ，打印图的邻接表，返回一个 string 类型字符串

实现图的深度优先搜索（DFS）和广度优先搜索（BFS）算法，输出对应的顶点序列

测试时，调用 string graph_DFS_Traverse() 和 string graph_BFS_Traverse() 函数， 返回对应图的顶点序列 string 字符串
DFS 与 BFS 结果不唯一，在遇到多种情况时，请按照邻接表中链表的存储顺序进行输出。也可以不使用这四个辅助函数，根据自己的习惯自定义实现算法的中间函数。测试时 不会调用这四个辅助函数，但请注意不要出现编译错误。

# 2. 程序实现

```cpp
#include "myGraph.h"
#include <iostream>
#include <sstream>
#include <string>
#include <queue>
#include <stack>

using namespace std;

EdgeNode::EdgeNode(){
    dest = -1;
    link = nullptr;
}
```

```
EdgeNode::EdgeNode(int dest, EdgeNode* link){
    this->dest = dest;
    this->link = link;
}

EdgeNode::~EdgeNode(){
    dest = -1;
    link = nullptr;
}

int EdgeNode::getDest(){
    return dest;
}

EdgeNode* EdgeNode::getNext(){
    return link;
}

void EdgeNode::setDest(int dest){
    this->dest = dest;
}

void EdgeNode::setNext(EdgeNode* link){
    this->link = link;
}

VertexNode::VertexNode(){
    data = ' ';
    firstEdge = nullptr;
}

VertexNode::VertexNode(char data, EdgeNode* firstEdge){
    this->data = data;
    this->firstEdge = firstEdge;
}

VertexNode::~VertexNode(){
    data = ' ';
    firstEdge = nullptr;
}

char VertexNode::getData(){
    return data;
}
```

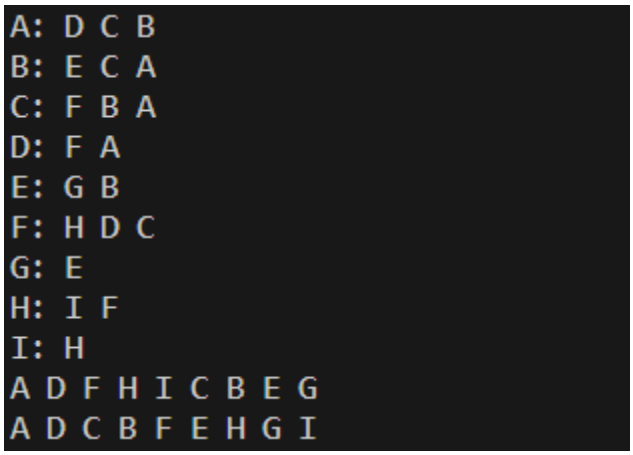
```
EdgeNode* VertexNode::getFirstEdge(){
    return firstEdge;
}

void VertexNode::setData(char data){
    this->data = data;
}

void VertexNode::setFirstEdge(EdgeNode* firstEdge){
    this->firstEdge = firstEdge;
}

MyGraph::MyGraph(){

}

MyGraph::MyGraph(int nodeNum, int edgeNum, char* nodeList, int*
edgeStartList, int* edgeEndList) {
    this->nodeNum = nodeNum;
    this->edgeNum = edgeNum;
    VexList = new VertexNode[nodeNum];

    for (int i = 0; i < nodeNum; i++) {
        VexList[i].setData(nodeList[i]);
        VexList[i].setFirstEdge(nullptr);
    }

    for (int i = edgeNum - 1; i > -1; i--) {
        int start = edgeStartList[i];
        int end = edgeEndList[i];

        EdgeNode* newEdgeNode = new EdgeNode(end, nullptr);

        EdgeNode* p = VexList[start].getFirstEdge();
        if(p == nullptr){
            VexList[start].setFirstEdge(newEdgeNode);
        }
        else{
            while(p->getNext() != nullptr){
                p = p->getNext();
            }
            p->setNext(newEdgeNode);
        }

        EdgeNode* newEdgeNode2 = new EdgeNode(start, nullptr);
```

```
        EdgeNode* q = VexList[end].getFirstEdge();
        if(q == nullptr){
            VexList[end].setFirstEdge(newEdgeNode2);
        }
        else{
            while(q->getNext() != nullptr){
                q = q->getNext();
            }
            q->setNext(newEdgeNode2);
        }
    }
}

MyGraph::~MyGraph(){
    nodeNum = 0;
    edgeNum = 0;
    delete VexList;
}

int MyGraph::getNodeNum(){
    return nodeNum;
}

int MyGraph::getEdgeNum(){
    return edgeNum;
}

string MyGraph::printGraph() {
    string output;

    for (int i = 0; i < nodeNum; ++i) {
        output += VexList[i].getData();
        output += ": ";

        EdgeNode* currentEdge = VexList[i].getFirstEdge();
        while (currentEdge != nullptr) {
            output += VexList[currentEdge->getDest()].getData();

            if (currentEdge->getNext() != nullptr) {
                output += ' ';
            }

            currentEdge = currentEdge->getNext();
        }
```

```
        if (i < nodeNum - 1) {
            output += '\n';
        }
    }

    return output;
}



string MyGraph::graph_DFS_Traverse(){
    bool* visited = new bool [nodeNum];
    string result = "";
    for(int i = 0; i < nodeNum; i++){
        visited[i] = false;
    }
    stack <VertexNode> stk;
    stk.push(VexList[0]);
    while(!stk.empty()){
        int index = -1;
        for(int i = 0; i < nodeNum; i++){
            if(VexList[i].getData() == stk.top().getData()) index = i;
        }
        if(!visited[index]){
            result += stk.top().getData();
            result += " ";
            visited[index] = true;
            stk.pop();

            stack <VertexNode> stk2;
            EdgeNode* p = VexList[index].getFirstEdge();
            if(p == nullptr){}
            else{
                do{
                    stk2.push(VexList[p->getDest()]);
                    p = p->getNext();
                } while(p!= nullptr);

            }

            while(!stk2.empty()){
                VertexNode v;
                v = stk2.top();
                stk2.pop();
```

```
                stk.push(v);
            }
        }
        else{
            stk.pop();
        }

    }
    return result;


}


string MyGraph::graph_BFS_Traverse(){
bool* visited = new bool [nodeNum];
    string result = "";
    for(int i = 0; i < nodeNum; i++){
        visited[i] = false;
    }
    queue <VertexNode> q;
    q.push(VexList[0]);
    while(!q.empty()){
        int index = -1;
        for(int i = 0; i < nodeNum; i++){
            if(VexList[i].getData() == q.front().getData()) index = i;
        }
        if(!visited[index]){
            result += q.front().getData();
            result += " ";
            visited[index] = true;
            EdgeNode* p = VexList[index].getFirstEdge();
            if(p == nullptr){}
            else{
                do{
                    q.push(VexList[p->getDest()]);
                    p = p->getNext();
                } while(p!= nullptr);

            }
        }
        q.pop();
    }
    return result;
}
```

## 3. 实验结果




## 4. 实验中遇到的问题及解决方法

实现了图的构造和析构,理解了邻接表构造过程。构造邻接表需要对边集合进行排序,然后按顺序构建链表。

实现了两种遍历方式:BFS 和 DFS。理解了两者的实现原理和区别。BFS 适合在图中寻找最短路径,DFS 适合于深度优先探索图结构。

BFS 使用队列实现,DFS 使用栈实现。两种数据结构对应两种遍历方式。理解数据结构与算法的配合。

实现过程中注意到一些细节,如标记访问过的顶点,处理没有边的顶点等。这些在实现图遍历时需要注意。