|  |  |  |  |
| --- | --- | --- | --- |
| **学生学号** | 175064 | **实验课成绩** |  |

*河北工业大学*

**学 生 实 验 报 告 书**

|  |  |
| --- | --- |
| **实验课程名称** | 数据结构 |
| **开 课 学 院** | 国际教育学院 |
| **指导教师姓名** | 宛艳萍 |
| **学 生 姓 名** | 张心宇 |
| **学生专业班级** | 中法计172 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 2018 | -- | 2019 | 学年 | 第 | 二 | 学期 |

# 实验内容

【**问题描述**】

很多涉及图上操作的算法都是以图的遍历操作为基础的。试写一个程序，演示连通的无向图上，遍历全部结点的操作。

【基本要求】

以邻接多重表为存储结构，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。

【测试数据】

教科书图7.33。暂时忽略里程，起点为北京。

【实现提示】

设图的结点不超过30个，每个结点用一个编号表示（如果一个图有n个结点，则它们的编号分别为1，2，…，n）。通过输入图的全部边输入一个图，每个边为一个数对，可以对边的输入顺序作出某种限制。注意，生成树的边是有向边，端点顺序不能颠倒。

【选作内容】

（1） 借助于栈类型（自己定义和实现），用非递归算法实现深度优先遍历。

（2） 以邻接表为存储结构，建立深度优先生成树和广度优先生成树，再按凹入表或树形打印生成树。

（3） 正如习题7.8提示中分析的那样，图的路径遍历要比结点遍历具有更为广泛的应用。再写一个路径遍历算法，求出从北京到广州中途不过郑州的所有简单路径及其里程。

# 实验目的

（1） 掌握图的基本存储方法。

（2） 熟练掌握图的两种搜索路径的遍历方法。

# 程序清单

## Queue.h

//  
// Created by gorkr on 19-5-31.  
//  
  
#ifndef GRAPH\_QUEUE\_H  
#define GRAPH\_QUEUE\_H  
  
#endif //GRAPH\_QUEUE\_H  
  
  
#include <iostream>  
using std::cout;  
using std::endl;  
  
template<typename T,int size=30> // int size 非模板类型参数  
class Queue  
{  
public:  
 Queue();  
 bool isFull();  
 bool empty();  
 void EnQueue(T elem);  
 T DeQueue();  
 T get\_back();  
 T get\_front();  
  
private:  
 T \_arr[size];  
 int \_front;  
 int \_rear;  
};  
  
template<typename T,int size>  
Queue<T,size>::Queue()  
 :\_front(0)  
 ,\_rear(0){}  
  
template<typename T,int size>  
bool Queue<T, size>::empty() {  
 return \_rear==\_front;  
}  
  
template<typename T,int size>  
bool Queue<T, size>::isFull() {  
 return (\_rear + 1) % size == \_front;  
}  
  
template<typename T,int size>  
void Queue<T, size>::EnQueue(T elem) {  
 if (!isFull())  
 \_arr[\_rear++] = elem;  
 else  
 cout << "队列已经满了" << endl;  
}  
template<typename T,int size>  
T Queue<T, size>::DeQueue() {  
 if (!empty()) {  
 auto p = \_front;  
 ++\_front;  
 \_front %= size;  
 return \_arr[p];  
 }  
 else  
 cout << "队列已经空了" << endl;  
}  
template<typename T,int size>  
T Queue<T, size>::get\_front() {  
 return \_arr[\_front];  
}  
template<typename T,int size>  
T Queue<T, size>::get\_back() {  
 return \_arr[(\_rear - 1 + size) % size];  
}

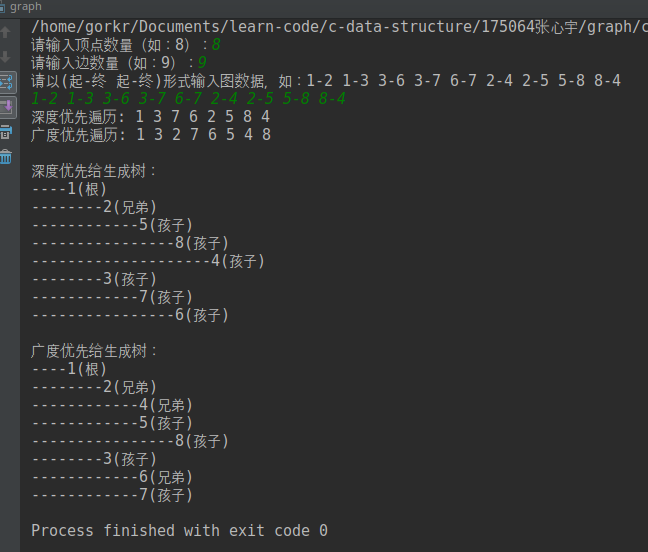
## Tree.h

//  
// Created by gorkr on 19-5-31.  
//  
  
#ifndef GRAPH\_TREE\_H  
#define GRAPH\_TREE\_H  
  
#endif //GRAPH\_TREE\_H  
#include <iostream>  
#include <stack>  
  
template <typename T>  
class Tree{  
public:  
 T data;  
 Tree \* firstchild, \* nextsibling;  
  
 Tree(T d, Tree \*fc, Tree \*nb):data(d),firstchild(fc),nextsibling(nb){};  
 Tree():data(NULL),firstchild(nullptr),nextsibling(nullptr){};  
 // 从当前节点向下打印二叉树  
 void dispTree();  
};  
  
  
template <typename T>  
void Tree<T>::dispTree() {  
 std::stack<Tree<T>\*> s;  
 std::stack<int> level;  
 std::stack<std::string> type;  
 int width = 4;  
 int above\_width=0;  
 Tree<T> \*p;  
 p=this;  
  
 // 先将根入栈  
 s.push(p);  
 level.push(width);  
 type.push("根");  
  
 while(s.size()){  
 p=s.top();s.pop();  
 above\_width = level.top(); level.pop();  
  
 for(int i=1; i<=above\_width; i++){  
 std::cout<<"-";  
 }  
 std::cout<<p->data<<"("<<type.top()<<")"<<std::endl;  
 type.pop();  
  
 if(p->firstchild!= nullptr){  
 s.push(p->firstchild);  
 level.push(above\_width+width);  
 type.push("孩子");  
 }  
 if(p->nextsibling!= nullptr){  
 s.push(p->nextsibling);  
 level.push(above\_width+width);  
 type.push("兄弟");  
 }  
  
 }  
  
}

## main.cpp

// 使用模板  
// size\_t 下标类型  
// std::function  
// 如何实现非模板类型动态赋值  
// 复习广度深度优先生成树概念。  
// 指针变量也和普通变量相同， 在函数内部给指针变量赋值，函数结束，也会丢失改变量  
  
// 可以写深度优先和广度优先 两种遍历方法， 将vist()函数作为参数，传入两种方法中  
  
#include <iostream>  
#include <string>  
#include <functional>  
#include <vector>  
#include "Queue.h"  
#include "Tree.h"  
  
  
const int MAX\_VERTEX\_NUM = 31; // 留个头  
enum class VisitIf{unvisited, visited};  
struct InData{  
 int head;  
 int end;  
};  
  
// 边  
struct EBox{ // edge的缩写吗？  
 VisitIf mark;  
 int ivex, jvex;  
 EBox \*ilink, \*jlink;  
 //int weight; // 里程  
 EBox(int iv,int jv):ivex(iv),jvex(jv),mark(VisitIf::unvisited),ilink(nullptr),jlink(nullptr){}  
};  
  
// 顶点  
struct VexBox{  
 // 顶点data  
 VisitIf mark;  
 int data; // 用下标当做data  
 //std::string city;  
 EBox \* firstedge;  
};  
  
class OLGraph{  
private:  
 std::vector<VexBox> adjmulist;  
 int vexnum, edgenum;  
  
 int FirstAdjVex(int v);  
 int NextAdjvex(int v, int w);  
 void DFS(int v, void (\*visit)(VexBox n));  
 void DFSTreeLoop(int v, Tree<int> \* t);  
  
public:  
  
 OLGraph(InData \* raw, int vn, int ln);  
 void DFSTraverse(void (\*visit)(VexBox n));  
 void DFSTree(Tree<int> \* t);  
 // 从第v个定点开始，建立深度优先生成树  
 void BFSTraverse(void (\*visit)(VexBox n));  
 void BFSTree(Tree<int> \* &t);  
  
 void SetUnvisit();  
   
 static void PrintVex(VexBox v);  
  
 ~OLGraph()= default;  
};  
  
void OLGraph::SetUnvisit() {  
 for(auto it=adjmulist.begin(); it!=adjmulist.end();++it){ // 初始化标志位  
 it->mark = VisitIf ::unvisited;  
 }  
}  
  
  
  
void OLGraph::DFSTree(Tree<int> \*t) {  
 SetUnvisit();  
 DFSTreeLoop(1,t);  
}  
  
void OLGraph::DFSTreeLoop(int v, Tree<int> \* t){  
 auto q = t;  
 adjmulist[v].mark=VisitIf::visited;  
 bool first = true;  
 for(int w=FirstAdjVex(v); w>0; w=NextAdjvex(v,w)){  
 if(adjmulist[w].mark==VisitIf::unvisited){  
 auto p = new Tree<int>(w, nullptr, nullptr);  
 if(first){  
 t->firstchild=p;  
 first = false;  
 }else{  
 while(bool(q->nextsibling))  
 q=q->nextsibling;  
 q->nextsibling=p;  
 }  
 DFSTreeLoop(w,p);  
 }  
 }  
}  
  
  
  
void OLGraph::BFSTree(Tree<int> \*&t){  
 SetUnvisit();  
 Queue<Tree<int>\*,MAX\_VERTEX\_NUM> que;  
 Tree<int>\* p;  
  
 adjmulist[1].mark=VisitIf::visited;  
 t =new Tree<int>(adjmulist[1].data, nullptr, nullptr);  
 que.EnQueue(t);  
 while(!que.empty()){  
 auto first = true;  
 p = que.DeQueue();  
 for(int w = FirstAdjVex(p->data); w>0; w = NextAdjvex(p->data,w)){  
 if(adjmulist[w].mark==VisitIf::unvisited){  
 adjmulist[w].mark=VisitIf::visited;  
 if(first){  
 p->firstchild = new Tree<int>(adjmulist[w].data, nullptr, nullptr);  
 first = false;  
 que.EnQueue(p->firstchild);  
 }else{  
 while(bool(p->nextsibling))p=p->nextsibling;  
 p->nextsibling=new Tree<int>(adjmulist[w].data, nullptr, nullptr);  
 que.EnQueue(p->nextsibling);  
 }  
  
 }  
  
 }  
 }  
}  
  
  
void OLGraph::BFSTraverse(void (\*visit)(VexBox n)) {  
 SetUnvisit();  
 Queue<size\_t,MAX\_VERTEX\_NUM> que;  
 for(size\_t i=1; i<=vexnum; i++){  
 if(adjmulist[i].mark==VisitIf::unvisited){  
 adjmulist[i].mark=VisitIf::visited;  
 visit(adjmulist[i]);  
 que.EnQueue(i);  
 while(!que.empty()){  
 auto p = que.DeQueue();  
 for(int w = FirstAdjVex(p); w>0; w=NextAdjvex(p,w)){  
 if(adjmulist[w].mark==VisitIf::unvisited){  
 adjmulist[w].mark=VisitIf::visited;  
 visit(adjmulist[w]);  
 que.EnQueue(w);  
 }  
 }  
 }  
 }  
 }  
  
}  
  
/\*\*  
 \* @param raw 一个边的数组  
 \* @param vn 顶点个数 ---------是不是可以没有 // 如果让下标对应存储data，可以用  
 \*/  
OLGraph::OLGraph(InData raw[], int vn, int ln):vexnum(vn),edgenum(ln){  
  
 for(int i=0;i<=vn;i++){  
 adjmulist.push\_back(VexBox{VisitIf::unvisited, i, nullptr});  
 }  
 // 头插！！  
 for(int i=0; i<edgenum; i++){  
 // 多用auto关键字，尽量使用更现代化的方式来 new  
 auto lin = new EBox(raw[i].head,raw[i].end);  
 lin->ilink=adjmulist[lin->ivex].firstedge;  
 lin->jlink=adjmulist[lin->jvex].firstedge;  
 adjmulist[lin->ivex].firstedge=adjmulist[lin->jvex].firstedge=lin;  
 }  
}  
  
  
void OLGraph::PrintVex(VexBox v){  
 std::cout << v.data<< ' ';  
}  
  
/\*\*  
 \*下标为v的节点的第一个临接节点  
 \* @param g 无向图对象  
 \* @param v 当前节点下标  
 \* @return int，第一个相邻节点下标  
 \*/  
int OLGraph::FirstAdjVex(int v) {  
 if(bool(this->adjmulist[v].firstedge))  
 return this->adjmulist[v].firstedge->ivex==v?this->adjmulist[v].firstedge->jvex:this->adjmulist[v].firstedge->ivex;  
 else return 0;  
  
}  
  
/\*\*  
 \* 下标为v的节点的w节点之后的临街点  
 \* @param g 无向图对象  
 \* @param v 在v上找临街点  
 \* @param w  
 \* @return int, 下一个临界点下标  
 \*/  
int OLGraph::NextAdjvex(int v, int w) {  
 EBox \*p = this->adjmulist[v].firstedge;  
 while(p->ivex!=w&&p->jvex!=w){  
 p = p->ivex==v?p->ilink:p->jlink;  
 }  
 p = p->ivex==v?p->ilink:p->jlink;  
 if(bool(p))  
 return p->ivex==v?p->jvex:p->ivex;  
 else return 0;  
  
}  
  
/\*\*  
 \* 递归访问一个回路  
 \* @param g 无向图对象  
 \* @param v 以v为入口  
 \* @param visit 访问函数  
 \*/  
void OLGraph::DFS(int v, void (\*visit)(VexBox n) ){  
 this->adjmulist[v].mark = VisitIf::visited;  
 visit(this->adjmulist[v]);  
 for(int w=FirstAdjVex(v);w > 0; w=NextAdjvex(v,w)){  
 if(this->adjmulist[w].mark == VisitIf::unvisited) DFS(w, visit);  
 }  
  
}  
  
void OLGraph::DFSTraverse(void (\*visit)(VexBox n)) {  
 SetUnvisit();  
  
 for(int i=1; i<=this->vexnum; i++){  
 if(this->adjmulist[i].mark==VisitIf::unvisited) DFS(i,visit);  
 }  
}  
  
  
void test\_graph(){  
 InData raw[]={{1,2},{1,3},{3,6},{3,7},{6,7},{2,4},{2,5},{5,8},{8,4}};  
 int vexnum=8, edgenum=9;  
 OLGraph map(raw, vexnum, edgenum);  
  
 std::cout<<"深度优先遍历: ";map.DFSTraverse(OLGraph::PrintVex);  
 std::cout<<"\n广度优先遍历: ";map.BFSTraverse(OLGraph::PrintVex);  
 std::cout<<std::endl<<"\n深度优先给生成树："<<std::endl;  
 auto t = new Tree<int>(1, nullptr, nullptr);  
 map.DFSTree(t);  
 t->dispTree();  
  
  
  
 std::cout<<std::endl<<"广度优先给生成树："<<std::endl;  
 //auto b = new Tree<int>(1, nullptr, nullptr);  
 Tree<int>\* b;  
 map.BFSTree(b);  
 b->dispTree();  
  
  
}  
  
void interface(){  
 int vexnum, edgenum;  
 std::cout << "请输入顶点数量（如：8）：";  
 std::cin >> vexnum;  
 std::cout << "请输入边数量（如：9）：";  
 std::cin >> edgenum;  
 auto raw = new InData [edgenum];  
 std::cout << "请以(起-终 起-终)形式输入图数据，如：1-2 1-3 3-6 3-7 6-7 2-4 2-5 5-8 8-4" << std::endl;  
 char ch;  
 auto p = raw;  
 while(true){  
 std::scanf("%d-%d%c",&p->head,&p->end, &ch);  
 if(ch=='\n')break;  
 p++;  
 }  
  
 // delete(raw);  
 OLGraph map(raw, vexnum, edgenum);  
 std::cout<<"深度优先遍历: ";map.DFSTraverse(OLGraph::PrintVex);  
 std::cout<<"\n广度优先遍历: ";map.BFSTraverse(OLGraph::PrintVex);  
  
 std::cout<<std::endl<<"\n深度优先给生成树："<<std::endl;  
 auto t = new Tree<int>(1, nullptr, nullptr);  
 map.DFSTree(t);  
 t->dispTree();  
  
 std::cout<<std::endl<<"广度优先给生成树："<<std::endl;  
 //auto b = new Tree<int>(1, nullptr, nullptr);  
 Tree<int>\* b;  
 map.BFSTree(b);  
 b->dispTree();  
  
}  
  
int main() {  
  
 //test\_graph();  
 interface();  
  
 return 0;  
}

# 运行结果



# 分析与思考

1. 数据结构真是太难了 😢
2. 使用栈结构改进递归。
3. C++中如果一个对象为nullpter且程序中对此对象调用属性或方法，会引发段错误。
4. 这个学期第一次使用C++，以后应该更多的使用oop，熟悉oop。