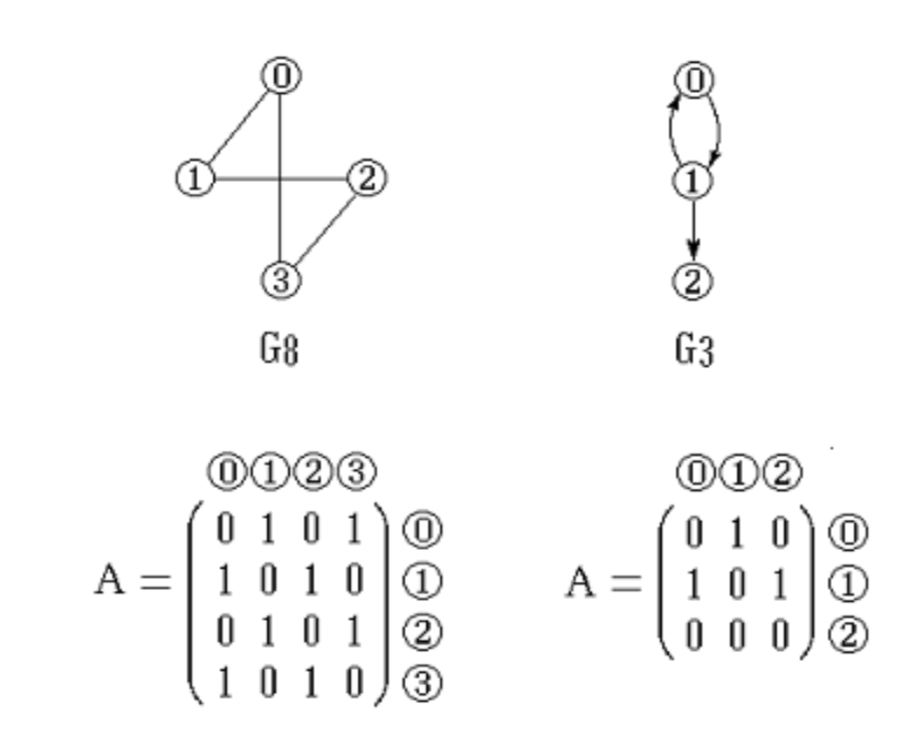
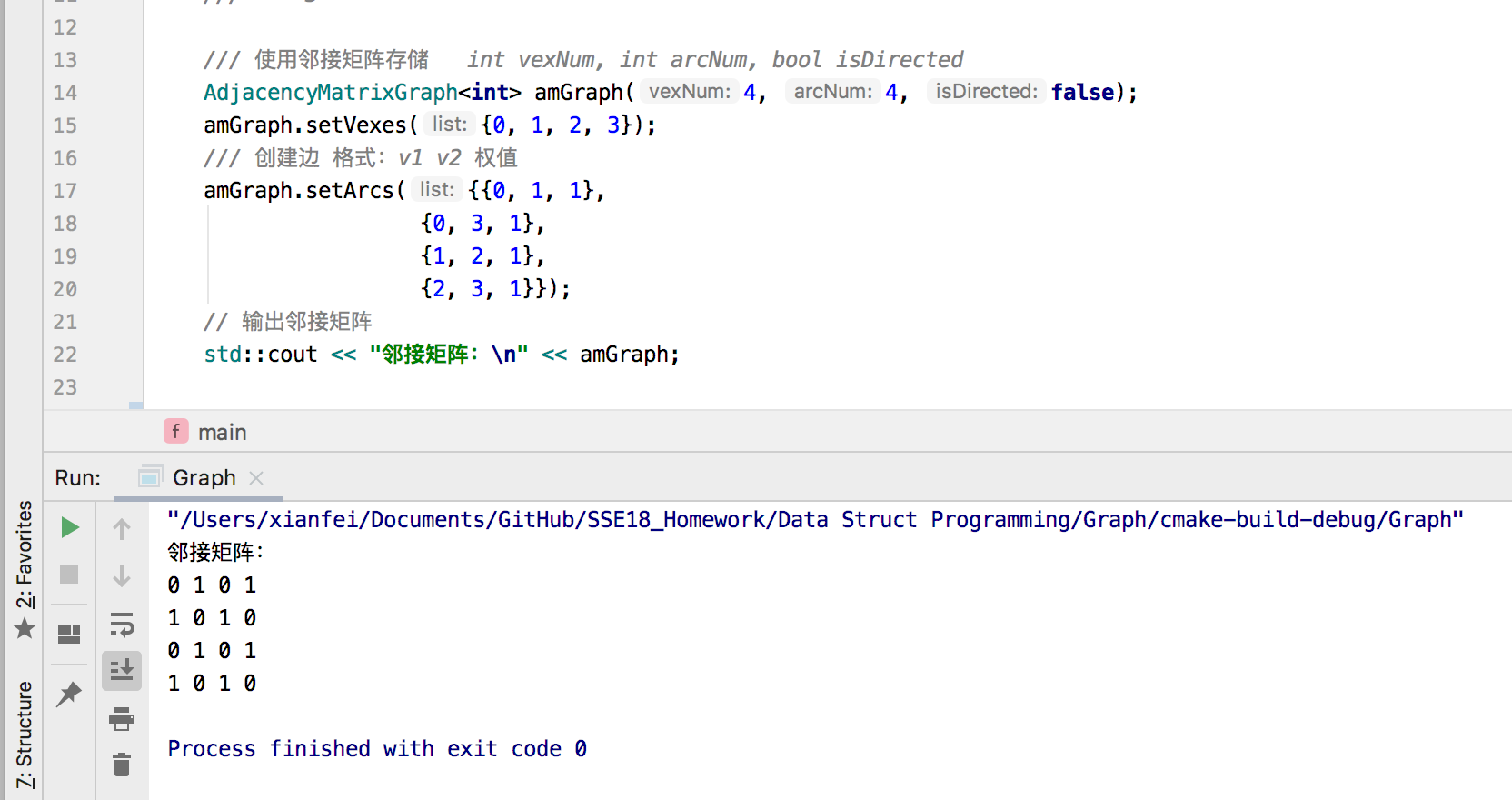
1. 邻接矩阵创建图：

测试的图：PPT 中的无向图和有向图

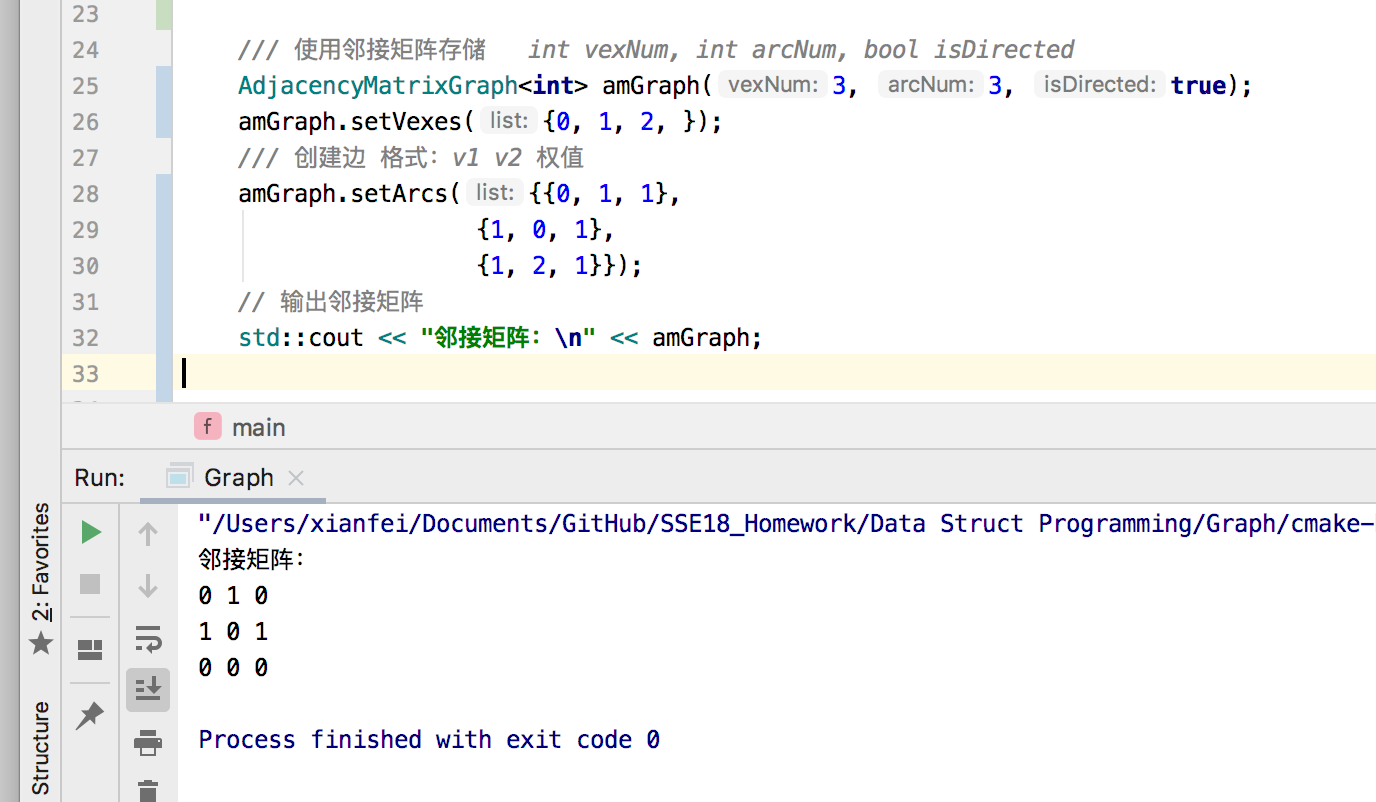


测试代码及输出：

左图：



右图：

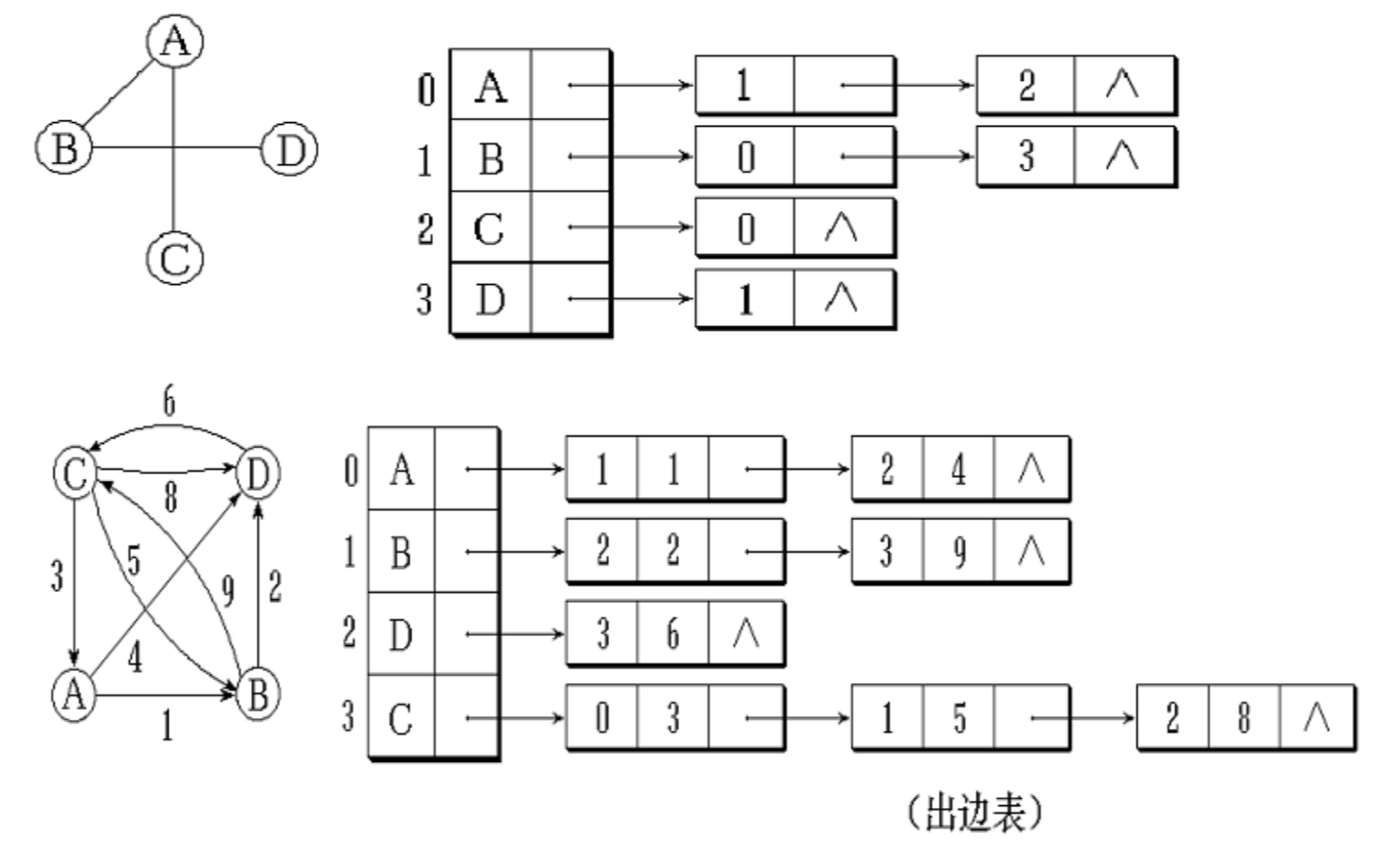


结果：与PPT上的示例完全相同。

心得：使用长度和宽度都为顶点个数的二维数组存储，如果是无向图则需要把两个方向的都要存上。期中无向图的邻接矩阵关于对角线对称。

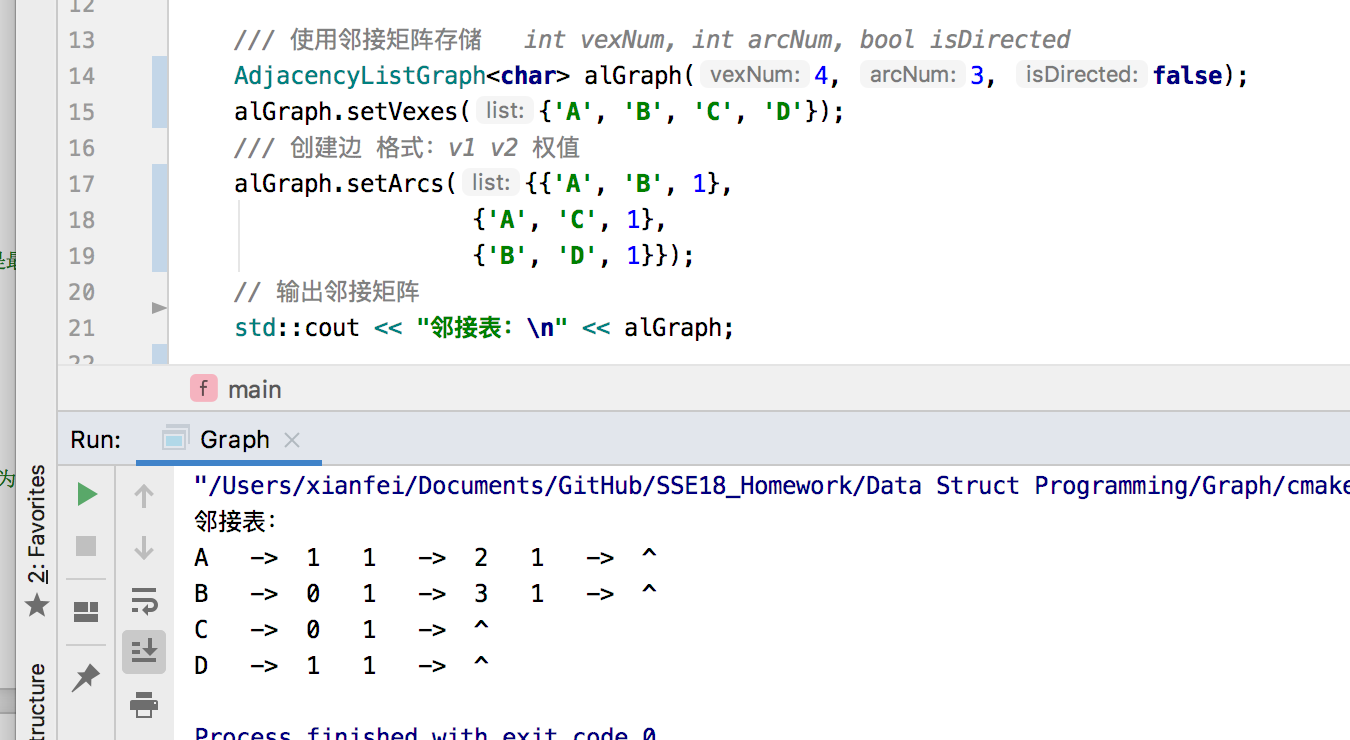
1. 邻接表创建图

测试的图：PPT 中的无向图和有向图

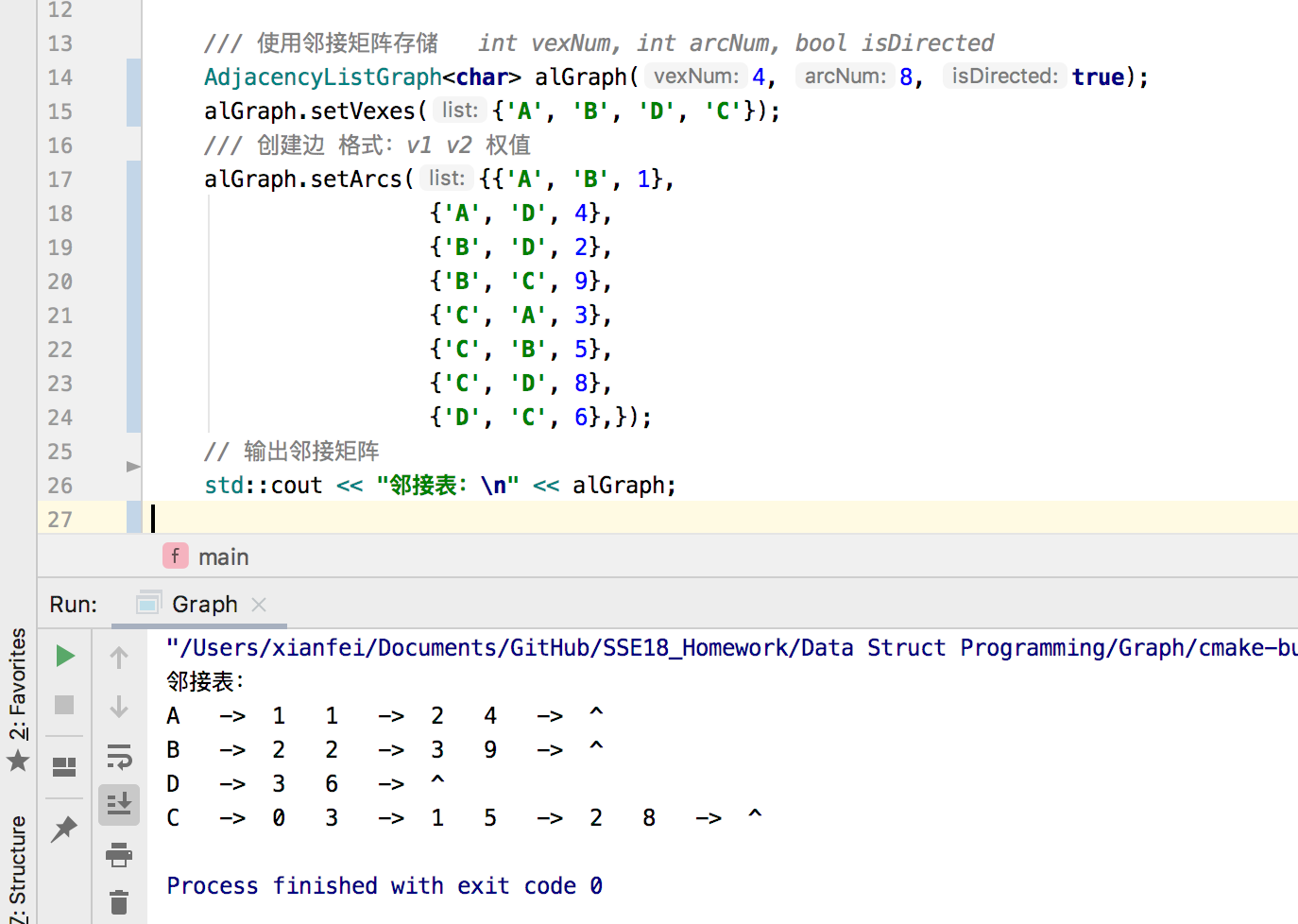


测试代码及输出：

上图：



下图：

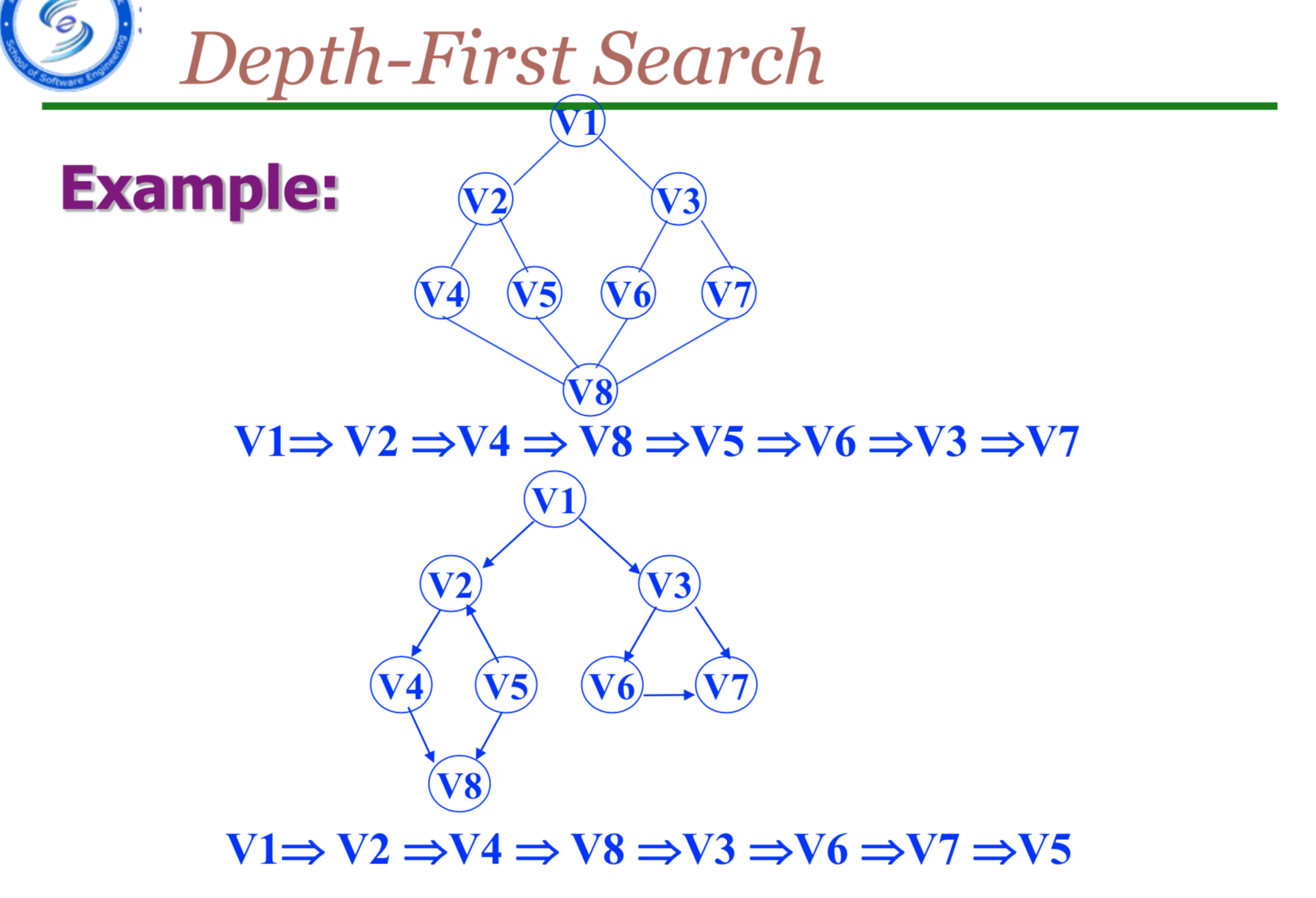
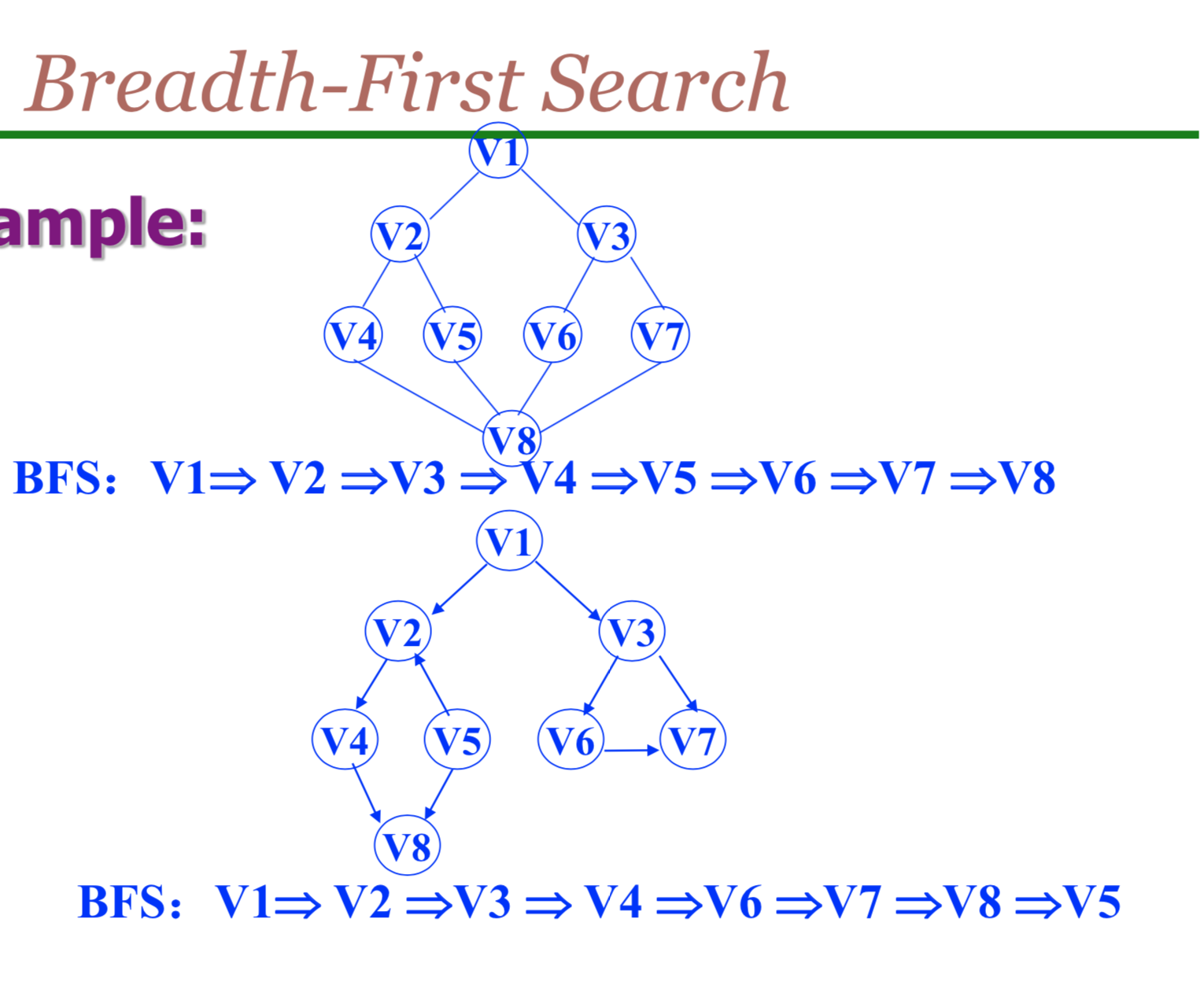


结果：起初与PPT上的结果不同，之后调换顶点顺序后与PPT的邻接表相同

心得：输入的顺序不一样 输出的顺序就不一样。每张图所对应的邻接表不唯一。

1. 邻接表存储的矩阵的深度优先搜索与宽度优先搜索

测试的图：PPT 中的无向图和有向图

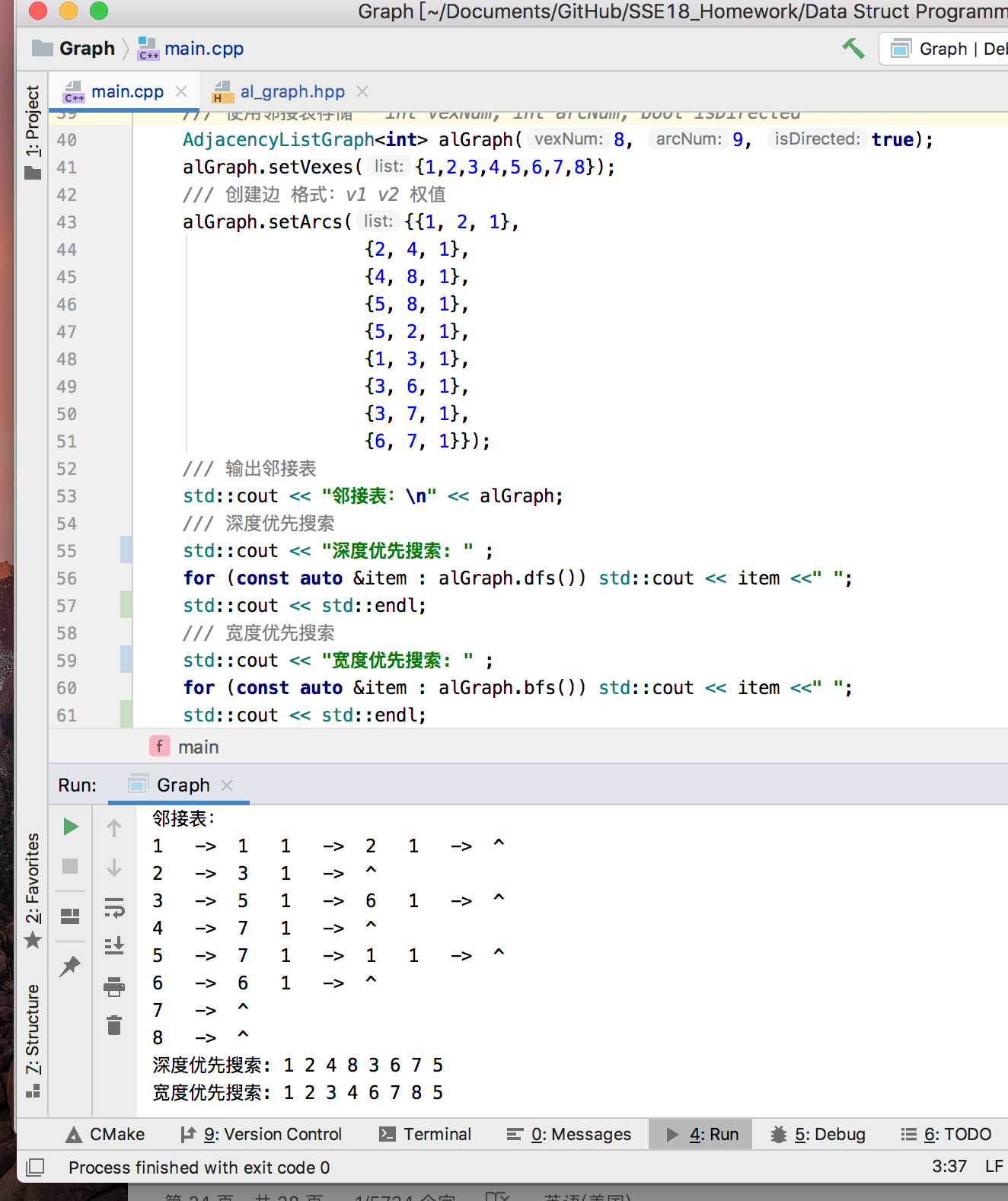
 

测试代码及输出：

上图：



下图：

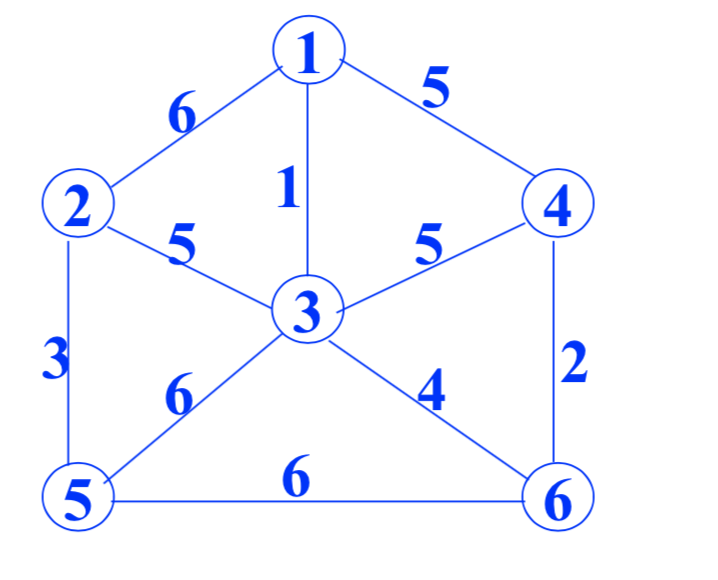


结果：与PPT上一致

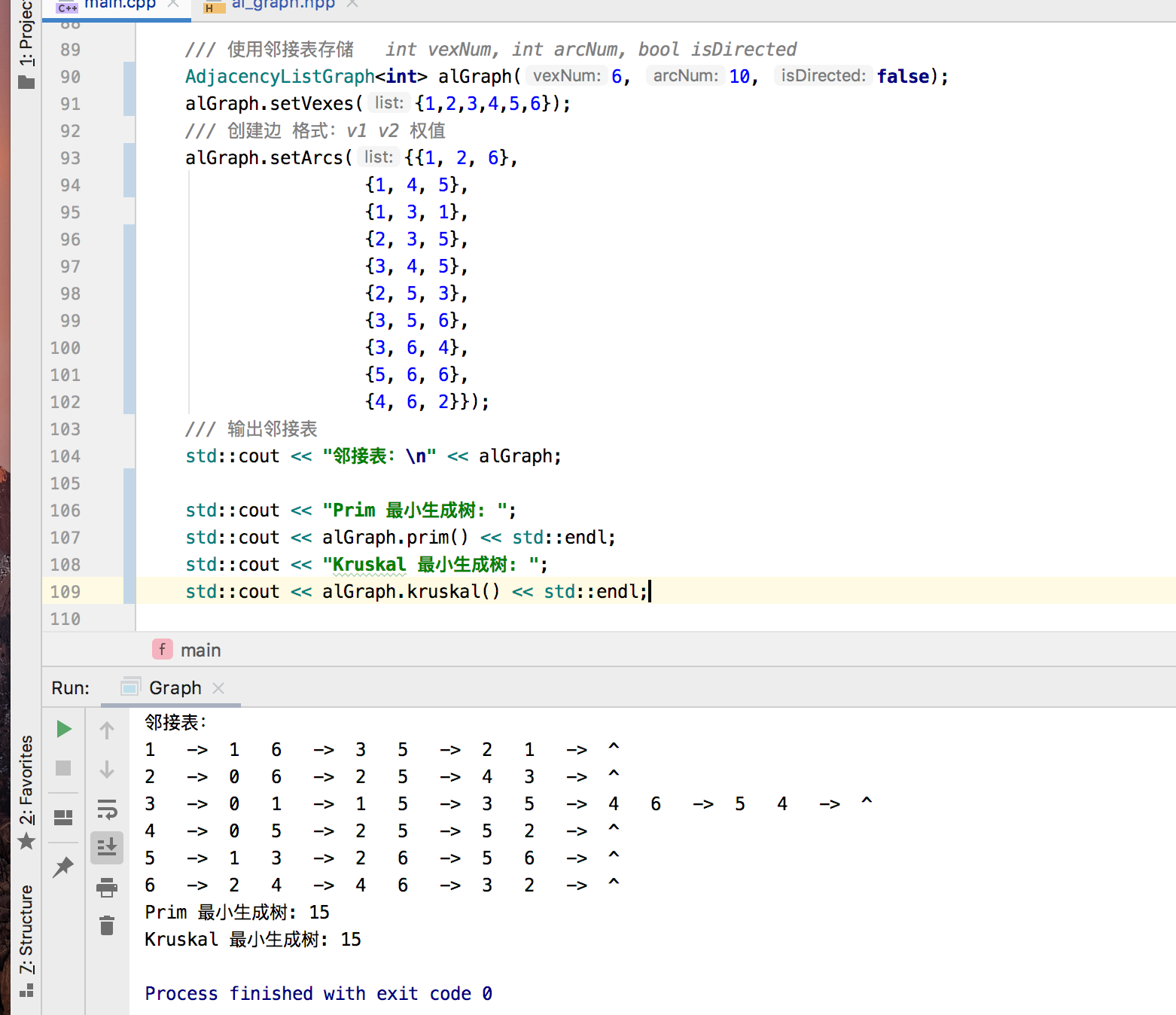
心得：BFS和DFS都可以通过使用stack或queue结构实现非递归调用。

1. 最小生成树——Prim和Kruskal算法。

测试的图：



测试代码及结果截图：



结果：与肉眼观察一致

心得：最小生成树方法有点难度，在于查找最小的权重以及判断两个集合的关系。

附录：代码

文件：main.cpp

#include **<iostream>**#include **"am\_graph.hpp"**#include **"al\_graph.hpp"  
  
int** main() {  
  
 */// 0  
 /// /|  
 /// 1-+-2  
 /// |/  
 /// 3  
  
// /// 使用邻接矩阵存储 int vexNum, int arcNum, bool isDirected  
// AdjacencyListGraph<char> alGraph(4, 8, true);  
// alGraph.setVexes({'A', 'B', 'D', 'C'});  
// /// 创建边 格式：v1 v2 权值  
// alGraph.setArcs({{'A', 'B', 1},  
// {'A', 'D', 4},  
// {'B', 'D', 2},  
// {'B', 'C', 9},  
// {'C', 'A', 3},  
// {'C', 'B', 5},  
// {'C', 'D', 8},  
// {'D', 'C', 6},});  
// // 输出邻接矩阵  
// std::cout << "邻接表：\n" << alGraph;  
  
// /// 使用邻接矩阵存储 int vexNum, int arcNum, bool isDirected  
// AdjacencyMatrixGraph<int> amGraph(3, 3, true);  
// amGraph.setVexes({0, 1, 2, });  
// /// 创建边 格式：v1 v2 权值  
// amGraph.setArcs({{0, 1, 1},  
// {1, 0, 1},  
// {1, 2, 1}});  
// // 输出邻接矩阵  
// std::cout << "邻接矩阵：\n" << amGraph;  
  
  
// /// 使用邻接表存储 int vexNum, int arcNum, bool isDirected  
// AdjacencyListGraph<int> alGraph(8, 9, true);  
// alGraph.setVexes({1,2,3,4,5,6,7,8});  
// /// 创建边 格式：v1 v2 权值  
// alGraph.setArcs({{1, 2, 1},  
// {2, 4, 1},  
// {4, 8, 1},  
// {5, 8, 1},  
// {5, 2, 1},  
// {1, 3, 1},  
// {3, 6, 1},  
// {3, 7, 1},  
// {6, 7, 1}});  
// /// 输出邻接表  
// std::cout << "邻接表：\n" << alGraph;  
// /// 深度优先搜索  
// std::cout << "深度优先搜索: " ;  
// for (const auto &item : alGraph.dfs()) std::cout << item <<" ";  
// std::cout << std::endl;  
// /// 宽度优先搜索  
// std::cout << "宽度优先搜索: " ;  
// for (const auto &item : alGraph.bfs()) std::cout << item <<" ";  
// std::cout << std::endl;  
  
// /// 使用邻接表存储 int vexNum, int arcNum, bool isDirected  
// AdjacencyListGraph<int> alGraph(8, 10, false);  
// alGraph.setVexes({1,2,3,4,5,6,7,8});  
// /// 创建边 格式：v1 v2 权值  
// alGraph.setArcs({{1, 2, 1},  
// {1, 3, 1},  
// {2, 4, 1},  
// {2, 5, 1},  
// {3, 6, 1},  
// {3, 7, 1},  
// {4, 8, 1},  
// {5, 8, 1},  
// {6, 8, 1},  
// {7, 8, 1}});  
// /// 输出邻接表  
// std::cout << "邻接表：\n" << alGraph;  
// /// 深度优先搜索  
// std::cout << "深度优先搜索: " ;  
// for (const auto &item : alGraph.dfs()) std::cout << item <<" ";  
// std::cout << std::endl;  
// /// 宽度优先搜索  
// std::cout << "宽度优先搜索: " ;  
// for (const auto &item : alGraph.bfs()) std::cout << item <<" ";  
// std::cout << std::endl;  
  
  
 /// 使用邻接表存储 int vexNum, int arcNum, bool isDirected* AdjacencyListGraph<**int**> alGraph(6, 10, **false**);  
 alGraph.setVexes({1,2,3,4,5,6});  
 */// 创建边 格式：v1 v2 权值* alGraph.setArcs({{1, 2, 6},  
 {1, 4, 5},  
 {1, 3, 1},  
 {2, 3, 5},  
 {3, 4, 5},  
 {2, 5, 3},  
 {3, 5, 6},  
 {3, 6, 4},  
 {5, 6, 6},  
 {4, 6, 2}});  
 */// 输出邻接表* std::cout << **"邻接表：\n"** << alGraph;  
  
 std::cout << **"Prim 最小生成树: "**;  
 std::cout << alGraph.prim() << std::endl;  
 std::cout << **"Kruskal 最小生成树: "**;  
 std::cout << alGraph.kruskal() << std::endl;  
  
 **return** 0;  
}  
  
**int** a(){  
 */// 使用邻接表存储 int vexNum, int arcNum, bool isDirected* AdjacencyListGraph<**int**> alGraph(8, 9, **true**);  
 alGraph.setVexes({1,2,3,4,5,6,7,8});  
 */// 创建边 格式：v1 v2 权值* alGraph.setArcs({{1, 2, 1},  
 {2, 4, 1},  
 {4, 8, 1},  
 {5, 8, 1},  
 {5, 2, 1},  
 {1, 3, 1},  
 {3, 6, 1},  
 {3, 7, 1},  
 {6, 7, 1}});  
 */// 输出邻接表* std::cout << **"邻接表：\n"** << alGraph;  
 */// 深度优先搜索* std::cout << **"深度优先搜索: "** << std::endl;  
 **for** (**const auto** &item : alGraph.dfs()) std::cout << item <<**" "**;  
 */// 宽度优先搜索* std::cout << **"宽度优先搜索: "** << std::endl;  
 **for** (**const auto** &item : alGraph.bfs()) std::cout << item <<**" "**;  
 **return** 0;  
}

文件：al\_graph.hpp

*//  
// Created by xianfei on 2019/11/18.  
//*#ifndef **GRAPH\_AL\_GRAPH\_HPP**#define **GRAPH\_AL\_GRAPH\_HPP**#include **<stdexcept>**#include **<initializer\_list>**#include **<ostream>**#include **<tuple>**#include **<vector>**#include **"queue.hpp"  
  
template**<**typename** VertexType>  
**class** AdjacencyListGraph {  
 **struct** ArcNode {  
 **int** adjVex;*/\*邻接点域\*/* **int** weight;*/\*表示边上信息的域info\*/* ArcNode \*next = **nullptr**;*/\*指向下一个邻接点的指针域\*/* ArcNode(**int** adjVex, **int** weight) : adjVex(adjVex), weight(weight) {}  
 };  
  
 **bool** \_isDirected = **false**;  
 **int** \_vexNum = 0, \_arcNum = 0;  
 **struct** \_vNode {  
 VertexType v;  
 ArcNode \*next = **nullptr**;  
 } \*\_vexList = **nullptr**;  
  
**private**: *// dfs 的递归调用函数* **template**<**typename** Function>  
 **void** \_dfs(**int** n, **bool** \*visited, Function visit) {  
 visited[n] = **true**;  
 visit(\_vexList[n].v);  
 **auto** ptr = \_vexList[n].next;  
 **if** (ptr) {  
 **while** (ptr->next != **nullptr**) {  
 **if** (!visited[ptr->adjVex])\_dfs(ptr->adjVex, visited, visit);  
 ptr = ptr->next;  
 }  
 **if** (!visited[ptr->adjVex])\_dfs(ptr->adjVex, visited, visit);  
 }  
 }  
  
**public**: *// 返回vector的深度优先搜素算法* std::vector<VertexType> dfs() {  
 **bool** \*visited = **new bool**[\_vexNum];  
 std::vector<VertexType> result;  
 **for** (**int** i = 0; i < \_vexNum; ++i)visited[i] = **false**;  
 **for** (**int** i = 0; i < \_vexNum; ++i)  
 **if** (!visited[i])  
 \_dfs(i, visited, [&result](VertexType v) -> **void** { result.push\_back(v); });  
 **delete**[] visited;  
 **return** result;  
 }  
  
 std::vector<VertexType> bfs() {  
  
 std::vector<VertexType> result;  
 **bool** \*visited = **new bool**[\_vexNum];  
 **for** (**int** i = 0; i < \_vexNum; ++i)visited[i] = **false**;  
 Queue<**int**> queue;  
 **for** (**int** i = 0; i < \_vexNum; ++i) {  
 **if** (!visited[i]) {  
 queue.push(i);  
 result.push\_back(\_vexList[i].v);  
 visited[i] = **true**;  
 } **else continue**;  
 **while** (!queue.empty()) {  
 **auto** ptr = \_vexList[queue.pop()].next;  
 **if** (ptr) {  
 **while** (ptr->next != **nullptr**) {  
 **if** (!visited[ptr->adjVex]) {  
 visited[ptr->adjVex] = **true**;  
 queue.push(ptr->adjVex);  
 result.push\_back(\_vexList[ptr->adjVex].v);  
 }  
 ptr = ptr->next;  
 }  
 **if** (!visited[ptr->adjVex]) {  
 visited[ptr->adjVex] = **true**;  
 queue.push(ptr->adjVex);  
 result.push\_back(\_vexList[ptr->adjVex].v);  
 }  
 }  
 }  
 }  
  
 **delete**[] visited;  
 **return** result;  
 }  
  
 **int** prim(){  
 **bool** \*visited = **new bool**[\_vexNum];  
 **for** (**int** i = 0; i < \_vexNum; ++i)visited[i] = **false**;  
 std::vector<\_vNode> vexs;  
 **int** willPushed= 0,length=0;  
 **for**(**int** i=0;i<\_vexNum;i++){  
 vexs.push\_back(\_vexList[willPushed]);  
 visited[willPushed]= **true**;  
 ArcNode \*node = vexs.back().next;  
 **int** minWeight=-1,minIndex=-1;  
 **for**(;;){  
 **if**(node== **nullptr**)**break**;  
 **if**(node->weight>minWeight && !visited[node->adjVex]){  
 minWeight=node->weight;  
 minIndex=node->adjVex;  
 }  
 node = node->next;  
 }  
 willPushed=minIndex;  
 **if**(minWeight==-1)**continue**;  
 length+=minWeight;  
 }  
 **if**(vexs.size()<\_vexNum)printf(**"No spanning tree\n"**);  
 **delete**[] visited;  
 **return** length;  
 }  
   
  
**public**:  
  
 AdjacencyListGraph(**int** vexNum, **int** arcNum, **bool** isDirected) : \_vexNum(vexNum), \_arcNum(arcNum),  
 \_isDirected(isDirected) {  
 \_vexList = **new** \_vNode[vexNum];  
 }  
  
 **virtual** ~AdjacencyListGraph() {  
 **delete**[] \_vexList;  
 }  
  
 **int** locateVex(VertexType target) {  
 **for** (**int** i = 0; i < \_vexNum; i++) {  
 **if** (\_vexList[i].v == target)**return** i;  
 }  
 **return** -1;  
 }  
  
 **void** setVexes(std::initializer\_list<VertexType> list) {  
 **int** i = -1;  
 **for** (**const auto** &item : list) {  
 **if** (i + 1 == \_vexNum)**throw** std::out\_of\_range(**""**);  
 \_vexList[++i].v = item;  
 }  
 }  
  
 **void** setArcs(std::initializer\_list<std::tuple<VertexType, VertexType, **int**>> list) {  
 **for** (**const auto** &[v1, v2, weight] : list) {  
 **int** i = locateVex(v1), j = locateVex(v2);  
 **auto** ptr = \_vexList[i].next;  
 **if** (ptr) {  
 **while** (ptr->next != **nullptr**)ptr = ptr->next;  
 ptr->next = **new** ArcNode(j, weight);  
 } **else** \_vexList[i].next = **new** ArcNode(j, weight);  
 **if** (!\_isDirected) {  
 ptr = \_vexList[j].next;  
 **if** (ptr) {  
 **while** (ptr->next != **nullptr**)ptr = ptr->next;  
 ptr->next = **new** ArcNode(i, weight);  
 } **else** \_vexList[j].next = **new** ArcNode(i, weight);  
 }  
 }  
 }  
  
 **friend** std::ostream &**operator**<<(std::ostream &os, **const** AdjacencyListGraph &graph) {  
 **for** (**int** i = 0; i < graph.\_vexNum; i++) {  
 os << graph.\_vexList[i].v << **"\t->\t"**;  
 **auto** ptr = graph.\_vexList[i].next;  
 **if** (ptr) {  
 **while** (ptr->next != **nullptr**) {  
 os << ptr->adjVex << **'\t'** << ptr->weight << **"\t->\t"**;  
 ptr = ptr->next;  
 }  
 os << ptr->adjVex << **'\t'** << ptr->weight << **"\t->\t^\t"**;  
 } **else** os << **"^\t"**;  
 os << std::endl;  
 }  
 **return** os;  
 }  
  
};  
  
#endif *//GRAPH\_AL\_GRAPH\_HPP*

文件：**am\_graph.hpp**

*//  
// Created by xianfei on 2019/11/18.  
//*#ifndef **GRAPH\_AM\_GRAPH\_HPP**#define **GRAPH\_AM\_GRAPH\_HPP**#include **<stdexcept>**#include **<cstring>**#include **<ostream>**#include **<tuple>**#include **<initializer\_list>  
  
template** <**typename** VertexType>  
**class** AdjacencyMatrixGraph{  
 **int** \_vexNum=0,\_arcNum=0;  
 **bool** \_isDirected = **false**;  
 **int** \*\*arcMatrix = **nullptr**;  
 VertexType \*vexList = **nullptr**;  
**public**:  
 AdjacencyMatrixGraph(**int** vexNum, **int** arcNum, **bool** isDirected) : \_vexNum(vexNum), \_arcNum(arcNum), \_isDirected(isDirected) {  
 arcMatrix = **new int**\*[vexNum];  
 **for** (**int** i=0;i<vexNum;i++) {  
 arcMatrix[i]=**new int**[vexNum];  
 memset(arcMatrix[i],0, **sizeof**(**int**)\*vexNum);  
 }  
 vexList = **new** VertexType[vexNum];  
 memset(vexList,0, **sizeof**(**int**)\*vexNum);  
 }  
  
 **virtual** ~AdjacencyMatrixGraph() {  
 **for** (**int** i=0;i<\_vexNum;i++) **delete** [] arcMatrix[i];  
 **delete** [] arcMatrix;  
 **delete** [] vexList;  
 }  
  
 **int** locateVex(VertexType target){  
 **for** (**int** i=0;i<\_vexNum;i++) {  
 **if**(vexList[i]==target)**return** vexList[i];  
 }  
 **return** -1;  
 }  
  
 **void** setVexes(std::initializer\_list<VertexType> list){  
 **int** i=-1;  
 **for** (**const auto** &item : list) {  
 **if**(i+1==\_vexNum)**throw** std::out\_of\_range(**""**);  
 vexList[++i]=item;  
 }  
 }  
  
 **void** setArcs(std::initializer\_list<std::tuple<**int**,**int**,**int**>> list){  
 **for** (**const auto** &[v1,v2,weight] : list) {  
 **int** i=locateVex(v1),j=locateVex(v2);  
 arcMatrix[i][j] = weight;  
 **if**(!\_isDirected)arcMatrix[j][i] = weight;  
 }  
 }  
  
 **friend** std::ostream &**operator**<<(std::ostream &os, **const** AdjacencyMatrixGraph &graph) {  
 **for**(**int** i=0;i<graph.\_vexNum;i++){  
 **for**(**int** j=0;j<graph.\_vexNum;j++){  
 os << graph.arcMatrix[i][j] << **' '**;  
 }  
 os << std::endl;  
 }  
 **return** os;  
 }  
};  
  
#endif *//GRAPH\_AM\_GRAPH\_HPP*

文件：stack.hpp

*//  
// Created by xianfei on 2019.  
//*#ifndef STACK\_H  
#define STACK\_H  
  
#include **<stdexcept>**#include **<cstdlib>  
  
template** <**class** T>  
**class** Stack{  
 **const int** DEFAULT\_SIZE=10;  
 T\* \_array=(T\*)malloc(DEFAULT\_SIZE \* **sizeof**(T));  
 **int** size=0;  
 **int** capacity=DEFAULT\_SIZE;  
**public**:  
 **virtual** ~Stack(){  
 free(\_array);  
 }  
 T top(){  
 **if**(size==0)**throw** std::out\_of\_range(**"stack is empty"**);  
 **return** \_array[size - 1];  
 }  
 **void** push(**const** T &num){  
 **if**(size>=capacity-1){  
 \_array=(T\*)realloc(\_array, 2 \* capacity \* **sizeof**(T));  
 **if**(!\_array)**throw** std::overflow\_error(**"stack overflow"**);  
 capacity\*=2;  
 }  
 \_array[size++]=num;  
 }  
 T pop(){  
 **if**(size==0)**throw** std::out\_of\_range(**"stack is empty"**);  
 **return** \_array[--size];  
 }  
 **bool** empty(){ **return** size==0;}  
};  
  
  
#endif *//STACK\_H*

文件：queue.hpp

*//  
// Created by xianfei on 2019.  
//*#ifndef **QUEUE\_H**#define **QUEUE\_H**#include **<stdexcept>  
  
template** <**class** T>  
**class** Queue{  
 **struct** Node{  
 Node \*next= **nullptr**;  
 T data;  
 };  
 Node\* \_rear= **nullptr**;  
 Node\* \_front= **nullptr**;  
**public**:  
 **virtual** ~Queue(){  
 **while**(\_front != **nullptr**)pop();  
 }  
 **bool** empty(){ **return** \_front == **nullptr**;}  
 **void** push(**const** T& data){  
 Node\* newNode = **new** Node;  
 **if**(\_rear == **nullptr**){  
 newNode->data=data;  
 \_rear= \_front=newNode;  
 **return**;  
 }  
 newNode->data=data;  
 \_rear->next=newNode;  
 \_rear=newNode;  
 }  
 T front(){  
 **if**(\_front == **nullptr**)**throw** std::out\_of\_range(**"queue is empty"**);  
 **return** \_front->data;  
 }  
 T pop(){  
 Node\* willDequeue = \_front;  
 **if**(willDequeue== **nullptr**)**throw** std::out\_of\_range(**"queue is empty"**);  
 \_front = \_front->next;  
 T willReturn=willDequeue->data;  
 **if**(willDequeue == \_rear)\_rear=**nullptr**;  
 **delete**(willDequeue);  
 **return** willReturn;  
 }  
};  
  
  
#endif *//QUEUE\_H*