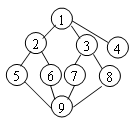
**第七章 图**

1. 首先将如下图所示的无向图给出其存储结构的邻接链表表示，然后写出对其分别进行深度，广度优先遍历的结果。

****

**1 2,3,4**

**2 1,5,6**

**3 1,7,8**

**4 1**

**5 2,9**

**6 2,9**

**7 3,9**

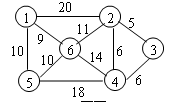
**8 3,9**

**9 5,6,7,8**

**Dfs 1,2,5,9,6,7,3,8,4**

**Bfs 1,2,3,4,5,6,7,8,9**

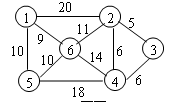
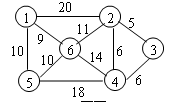
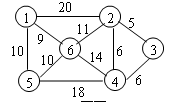
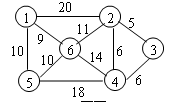
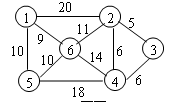
1. 已知一个无向图如下图所示，要求分别用Prim和Kruskal算法生成最小树（假设以①为起点，试画出构造过程）

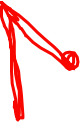
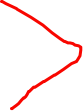
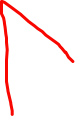
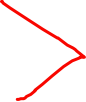
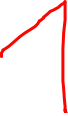
****

**Prim:**

1,6,5,2,3,4

Kruskal:





1. 已知如图所示的有向图，请给出该图的：



1. 每个顶点的入度、出度；

1 3

2 2,2

3 1,2

4 1,2

5 2,1

6 2,3

1. 邻接矩阵；

1

2 1

3 2 6

4 3 5 6

5 1

6 1 2 5

1. 邻接表；

1

2 1

3 2,6

4 3,5,6

5 1

6 1,2,5

1. 逆邻接表；

1 2,5,6

2 3,6

3 4

4

5 4,6

6 3,4

（5） 强连通分量。

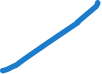
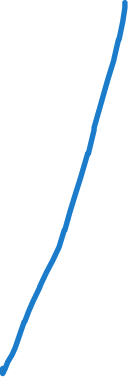
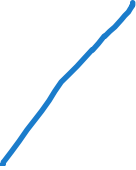
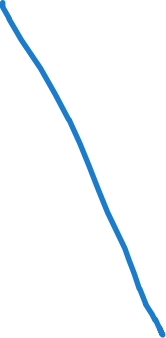
2,3,4,6

1. 已知世界六大城市为: 北京(Pe) 、纽约(N) 、巴黎(Pa) 、伦敦(L) 、东京(T) 、墨西哥(M), 下表给定了这六大城市之间的交通里程:

世界六大城市交通里程表( 单位: 百公里)



1. ．画出这六大城市的交通网络图;



1. ．画出该图的邻接表表示法;

PE: (N,109)(PA,82)(L,81) (T,21) (M,124)

N: (PE,109) (PA,58) (L,55) (T,108) (M,32)

PA: (PE,82) (N,58)(L,3)(T,97)(M,92)

L:(PE,81)(N,55)(PA,3)(T,95)(M,89)

T:(PE,21) (N,108) (PA,97) (L,95)(M,113)

M: (PE,124) (N,32)(PA,92)(L,89)(T,113)

（3）．画出该图按权值递增的顺序来构造的最小( 代价) 生成树.



1. 已知有向图有n 个顶点, 请写算法, 根据用户输入的偶对建立该有向图的邻接表。即接受用户输入的<vi,vj>( 以其中之一为0 标志结束), 对于每条这样的边, 申请一个结点, 并插入到单链表中, 如此反复, 直到将图中所有边处理完毕。

提示：先产生邻接表的n 个头结点（其结点数值域从1 到n）。

#include <iostream>  
#include <vector>  
  
using namespace std;  
  
class Node {  
public:  
 int value;  
 Node\* next;  
  
 Node(int val) : value(val), next(nullptr) {}  
};  
  
class DirectedGraph {  
public:  
 int num\_vertices;  
 vector<Node\*> adj\_list;  
  
 DirectedGraph(int num) : num\_vertices(num), adj\_list(num + 1, nullptr) {}  
  
 void addEdge(int start, int end) {  
 Node\* newNode = new Node(end);  
 newNode->next = adj\_list[start];  
 adj\_list[start] = newNode;  
 }  
};  
  
void buildDirectedGraph(DirectedGraph& graph) {  
 cout << "请输入顶点数n: ";  
 int n;  
 cin >> n;  
  
 DirectedGraph directedGraph(n);  
  
 cout << "请输入边的偶对，以0 0结束：" << endl;  
 while (true) {  
 int vi, vj;  
 cin >> vi >> vj;  
  
 if (vi == 0 || vj == 0) {  
 break;  
 }  
  
 directedGraph.addEdge(vi, vj);  
 }  
  
 graph = directedGraph;  
}  
  
void printAdjacencyList(const DirectedGraph& graph) {  
 cout << "邻接表：" << endl;  
 for (int i = 1; i <= graph.num\_vertices; ++i) {  
 cout << i << " -> ";  
 Node\* current = graph.adj\_list[i];  
 while (current) {  
 cout << current->value << " -> ";  
 current = current->next;  
 }  
 cout << "None" << endl;  
 }  
}  
  
int main() {  
 system("chcp 65001");  
 DirectedGraph directedGraph(0);  
 buildDirectedGraph(directedGraph);  
 printAdjacencyList(directedGraph);  
  
 return 0;  
}

1. 已知无向图采用邻接表存储方式，试写出删除边（ i ， j ）的算法。

#include <iostream>  
#include <vector>  
  
using namespace std;  
  
class Node {  
public:  
 int vertex;  
 Node\* next;  
  
 Node(int v) : vertex(v), next(nullptr) {}  
};  
  
class UndirectedGraph {  
public:  
 int numVertices;  
 vector<Node\*> adjacencyList;  
  
 UndirectedGraph(int n) : numVertices(n), adjacencyList(n + 1, nullptr) {}  
  
 void addEdge(int u, int v) {  
 Node\* newNodeU = new Node(v);  
 newNodeU->next = adjacencyList[u];  
 adjacencyList[u] = newNodeU;  
  
 Node\* newNodeV = new Node(u);  
 newNodeV->next = adjacencyList[v];  
 adjacencyList[v] = newNodeV;  
 }  
  
 void printGraph() {  
 cout << "Adjacency List:" << endl;  
 for (int i = 1; i <= numVertices; ++i) {  
 cout << i << " -> ";  
 Node\* current = adjacencyList[i];  
 while (current) {  
 cout << current->vertex << " -> ";  
 current = current->next;  
 }  
 cout << "Null" << endl;  
 }  
 }  
  
 void removeEdge(int i, int j) {  
 // Remove edge (i, j)  
 // Traverse the adjacency list of vertex i to find and remove the edge (i, j)  
 Node\* prevI = nullptr;  
 Node\* currentI = adjacencyList[i];  
  
 while (currentI && currentI->vertex != j) {  
 prevI = currentI;  
 currentI = currentI->next;  
 }  
  
 if (currentI) {  
 if (prevI) {  
 prevI->next = currentI->next;  
 } else {  
 adjacencyList[i] = currentI->next;  
 }  
 delete currentI;  
 }  
  
 // Traverse the adjacency list of vertex j to find and remove the edge (j, i)  
 Node\* prevJ = nullptr;  
 Node\* currentJ = adjacencyList[j];  
  
 while (currentJ && currentJ->vertex != i) {  
 prevJ = currentJ;  
 currentJ = currentJ->next;  
 }  
  
 if (currentJ) {  
 if (prevJ) {  
 prevJ->next = currentJ->next;  
 } else {  
 adjacencyList[j] = currentJ->next;  
 }  
 delete currentJ;  
 }  
 }  
};  
  
int main() {  
 UndirectedGraph graph(5);  
  
 graph.addEdge(1, 2);  
 graph.addEdge(1, 4);  
 graph.addEdge(2, 3);  
 graph.addEdge(2, 5);  
 graph.addEdge(3, 5);  
  
 cout << "Before removing edge (2, 3):" << endl;  
 graph.printGraph();  
  
 graph.removeEdge(2, 3);  
  
 cout << "\nAfter removing edge (2, 3):" << endl;  
 graph.printGraph();  
  
 return 0;  
}