```
/*节点类*/
class Node
{
public:
    Node(char identifier = 0);

    char m_identifier;       //顶点编号
    bool m_isVisited;        //顶点访问标志位：true表示已经被访问
};
Node::Node(char identifier)
{
    m_identifier = identifier;
    m_isVisited = false;
}


/*图类*/
class Graph
{
public:
    Graph(int capacity);
    ~Graph();

    void resetNode();               //重置所有顶点的访问标志位为false，未访问
    bool addNode(Node *pNode);      //添加新顶点
    bool addEdgeForUndirectedGraph(int row, int col, int val = 1); //添加边以构造无向图，val表示权值，默认连通
    bool addEdgeForDirectedGraph(int row, int col, int val = 1);   //添加边以构造有向图，val表示权值，默认连通
    void printMatrix();   //打印邻接矩阵

    void depthFirstTraverse(int nodeIndex);   //深度优先遍历，指定第一个点
    void widthFirstTraverse(int nodeIndex);   //广度优先遍历，指定第一个点

private:
    bool getValueOfEdge(int row, int col, int &val);  //获取边权值
    void widthFirstTraverseImplement(vector<int> preVec);  //利用vector实现广度优先遍历

    int m_iCapacity;      //图容量，即申请的数组空间最多可容纳的顶点个数
    int m_iNodeCount;     //图的现有顶点个数
    Node *m_pNodeArray;   //存放顶点的数组
    int *m_pMatrix;       //为了方便，用一维数组存放邻接矩阵
};
Graph::Graph(int capacity)
{
    m_iCapacity = capacity;
    m_iNodeCount = 0;

    m_pNodeArray = new Node[m_iCapacity];
    m_pMatrix = new int[m_iCapacity*m_iCapacity];

    for (int i = 0;i < m_iCapacity*m_iCapacity;i++)  //初始化邻接矩阵
    {
        m_pMatrix[i] = 0;
    }
}
Graph::~Graph()
{
    delete []m_pNodeArray;
    delete []m_pMatrix;
}
void Graph::resetNode()
{
    for (int i = 0;i < m_iNodeCount;i++)
    {
        m_pNodeArray[i].m_isVisited = false;
    }
}
bool Graph::addNode(Node *pNode)
{
    if (pNode == NULL)
        return false;
    m_pNodeArray[m_iNodeCount].m_identifier = pNode->m_identifier;
    m_iNodeCount++;
    return true;
}
```

```
bool Graph::addEdgeForUndirectedGraph(int row, int col, int val)
{
    if (row < 0 || row >= m_iCapacity)
        return false;
    if (col < 0 || col >= m_iCapacity)
        return false;
    m_pMatrix[row*m_iCapacity + col] = val;
    m_pMatrix[col*m_iCapacity + row] = val;
    return true;
}
bool Graph::addEdgeForDirectedGraph(int row, int col, int val)
{
    if (row < 0 || row >= m_iCapacity)
        return false;
    if (col < 0 || col >= m_iCapacity)
        return false;
    m_pMatrix[row*m_iCapacity + col] = val;
    return true;
}
void Graph::printMatrix()
{
    for (int i = 0;i < m_iCapacity;i++)
    {
        for (int k = 0;k < m_iCapacity;k++)
            cout << m_pMatrix[i*m_iCapacity + k] << " ";
        cout << endl;
    }
}
void Graph::depthFirstTraverse(int nodeIndex)
{
    int value = 0;

    //访问第一个顶点
    cout << m_pNodeArray[nodeIndex].m_identifier << " ";
    m_pNodeArray[nodeIndex].m_isVisited = true;

    //访问其他顶点
    for (int i = 0;i < m_iCapacity;i++)
    {
        getValueOfEdge(nodeIndex, i, value);
        if (value != 0)   //当前顶点与指定顶点连通
        {
            if (m_pNodeArray[i].m_isVisited == true)  //当前顶点已被访问
                continue;
            else           //当前顶点没有被访问，则递归
            {
                depthFirstTraverse(i);
            }
        }
        else       //没有与指定顶点连通
        {
            continue;
        }
    }
}
void Graph::widthFirstTraverse(int nodeIndex)
{
    //访问第一个顶点
    cout << m_pNodeArray[nodeIndex].m_identifier << " ";
    m_pNodeArray[nodeIndex].m_isVisited = true;

    vector<int> curVec;
    curVec.push_back(nodeIndex);      //将第一个顶点存入一个数组
    widthFirstTraverseImplement(curVec);
}
void Graph::widthFirstTraverseImplement(vector<int> preVec)
{
    int value = 0;
    vector<int> curVec;    //定义数组保存当前层的顶点
    for (int j = 0;j < (int)preVec.size();j++)  //依次访问传入数组中的每个顶点
    {
        for (int i = 0;i < m_iCapacity;i++)  //传入的数组中的顶点是否与其他顶点连接
        {
            getValueOfEdge(preVec[j], i, value);
            if (value != 0)   //连通
            {
```

```
                if (m_pNodeArray[i].m_isVisited==true)  //已经被访问
                {
                    continue;
                }
                else   //没有被访问则访问
                {
                    cout << m_pNodeArray[i].m_identifier << " ";
                    m_pNodeArray[i].m_isVisited = true;

                    //保存当前点到数组
                    curVec.push_back(i);
                }
            }
        }
    }

    if (curVec.size()==0)   //本层次无被访问的点，则终止
    {
        return;
    }
    else
    {
        widthFirstTraverseImplement(curVec);
    }
}
bool Graph::getValueOfEdge(int row, int col, int &val)
{
    if (row < 0 || row >= m_iCapacity)
        return false;
    if (col < 0 || col >= m_iCapacity)
        return false;
    val = m_pMatrix[row*m_iCapacity + col];
    return true;
}
```