1、假设不带权有向图 G 采用邻接表存储，分别设计实现求解以下问题的算法。

(1)求出图 G 中每个顶点的入度。

(2)求出图 G 中每个顶点的出度。

(3)求出图 G 中出度最大的一个顶点，输出该顶点编号。

(4)计算图 G 中出度为 0 的顶点数。

(5)判断图 G 中是否存在边<i，j>

解：

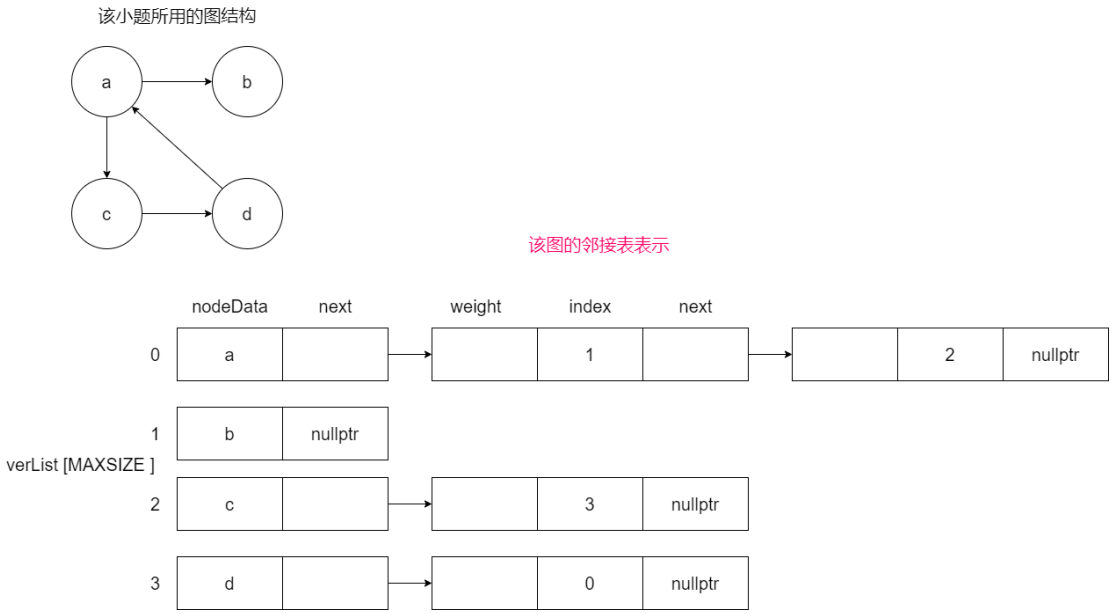


Figure1.1 第一小题的所用的图结构以及其邻接表

(1)求出图 G 中每个顶点的入度。

```
void HNodeIndegree(Graph G)
{
    int IndegreeCnt[G.vertexNum] = {0};
    for (int i = 0; i < G.vertexNum; i++)
    {
        Node *tmp = G.verList[i].next;
        while (tmp != nullptr)
        {
            IndegreeCnt[tmp->index]++;
            tmp = tmp->next;
        }
    }
    cout << "all node's indegree" << endl;
    for (int j = 0; j < G.vertexNum; j++)
    {
        cout << G.verList[j].nodeData << ':' << IndegreeCnt[j] << endl;
    }
}
```

Figure1.2 求出图 G 中每个顶点的入度代码

(2)求出图 G 中每个顶点的出度。

```cpp
void HNodeOutdegree(Graph G)
{
    int OutdegreeCnt[G.vertexNum] = {0};
    for (int i = 0; i < G.vertexNum; i++)
    {
        Node *tmp = G.verList[i].next;
        while (tmp != nullptr)
        {
            OutdegreeCnt[i]++;
            tmp = tmp->next;
        }
    }
    cout << "all node's outdegree" << endl;
    for (int j = 0; j < G.vertexNum; j++)
    {
        cout << G.verList[j].nodeData << ':' << OutdegreeCnt[j] << endl;
    }
}
```

Figure1.3 求出图 G 中每个顶点的出度代码

(3)求出图 G 中出度最大的一个顶点,输出该顶点编号。

```cpp
void MaxOutDegreeNode(Graph G)
{
    int index = -1; // 出度最大的数组下标
    int res = 0;    // 出度
    // 与HNodeOutdegree同
    // same code
    int OutdegreeCnt[G.vertexNum] = {0};
    for (int i = 0; i < G.vertexNum; i++)
    {
        Node *tmp = G.verList[i].next;
        while (tmp != nullptr)
        {
            OutdegreeCnt[i]++;
            tmp = tmp->next;
        }
    }
    // same code

    for (int j = 0; j < G.vertexNum; j++)
    {
        if (OutdegreeCnt[j] > res)
        {
            res = OutdegreeCnt[j];
            index = j;
        }
    }
    cout << "the max out degree is" << ':' << G.verList[index].nodeData << endl;
}
```

Figure1.4 求出图 G 中出度最大的一个顶点,输出该顶点编号代码

(4)计算图 G 中出度为 0 的顶点数。

```cpp
void ZeroOutdegree(Graph G)
{
    int cnt = 0;
    // same code
    int OutdegreeCnt[G.vertexNum] = {0};
    for (int i = 0; i < G.vertexNum; i++)
    {
        Node *tmp = G.verList[i].next;
        while (tmp != nullptr)
        {
            OutdegreeCnt[i]++;
            tmp = tmp->next;
        }
    }
    // same code
    for (int j = 0; j < G.vertexNum; j++)
    {
        if (OutdegreeCnt[j] == 0)
        {
            cnt++;
        }
    }
    cout << "the number of zero outdegree is" << ' ' << cnt;
}
```

Figure1.5 计算图 G 中出度为 0 的顶点数代码

(5)判断图 G 中是否存在边<i,j>

```cpp
bool IsArcExist(char a, char b, Graph G)
{
    int AIndex, BIndex; // 找到弧头和弧尾的数组下标
    for (int i = 0; i < G.vertexNum; i++)
    {
        if (G.verList[i].nodeData == a)
        {
            AIndex = i;
            continue;
        }
        if (G.verList[i].nodeData == b)
        {
            BIndex = i;
            continue;
        }
    }
    Node *tmp = G.verList[AIndex].next;
    while (tmp != nullptr)
    {
        if (tmp->index == BIndex)
        {
            return true;
        }
        tmp = tmp->next;
    }
    return false;
}
```

Figure1.6 判断图 G 中是否存在边<i,j>代码

2、分别以邻接矩阵和邻接表作为存储结构，实现以下图的基本操作：

① 增加一个新顶点 v，InsertVex(G, v)；

② 删除顶点 v 及其相关的边，DeleteVex(G, v)；

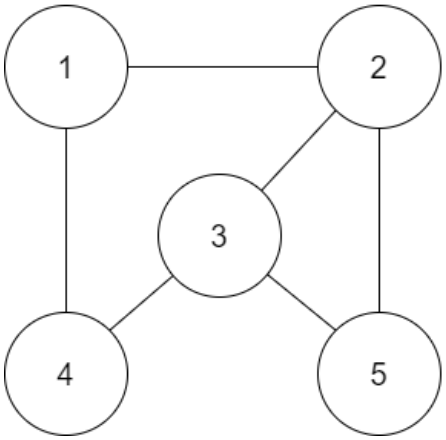
③ 增加一条边<v，w>，InsertArc(G, v, w)；

④ 删除一条边<v，w>，DeleteArc(G, v, w)。

解：

（一）：邻接矩阵为存储结构

该小题所用的图结构（初始）



该图的邻接矩阵表示（初始）

	①	②	③	④	⑤
①	0	1	0	1	0
②	1	0	1	0	1
③	0	1	0	1	1
④	1	0	1	0	0
⑤	0	1	1	0	0

Figure2.1 第二小题的所用的图结构以及其邻接矩阵

① 增加一个新顶点 v，InsertVex(G, v)；

```
// 添加顶点
void InsertVex(Graph &G, char c)
{
    G.vertexList[G.vertexNum].Vertex = c;          // 顶点信息存入顶点表中
    G.vertexList[G.vertexNum].index = G.vertexNum; // 顶点的在邻接矩阵的索引存入顶点表中                    // 顶点数量加一
    // 对新插入的顶点对应的矩阵初始化
    for (int i = 0; i <= G.vertexNum; i++)
    {
        G.Matrix[G.vertexNum][i] = 0;
        G.Matrix[i][G.vertexNum] = 0;
    }
    G.vertexNum++;
}
```

Figure2.2 增加一个新顶点 v，InsertVex(G, v)代码

② 删除顶点 v 及其相关的边，DeleteVex(G, v)；

```
// 删除顶点以及相关的边
void DeletVex(Graph &G, char c)
{
    int vexIndex = Locate(G, c);
    G.vertexNum- ;
    for (int i = 0; i < G.vertexNum; i++)
    {
        if (G.Matrix[vexIndex][i] == 1) // 有边
        {
            G.Matrix[vexIndex][i] = 0;
            G.arcNum--;
        }
        if (G.Matrix[i][vexIndex] == 1)
        {
            G.Matrix[i][vexIndex] = 0;
        }
    }
    G.vertexList[vexIndex].index = -1;
}
```

Figure2.3 删除顶点 v 及其相关的边，DeleteVex(G, v)代码

③ 增加一条边<v，w>，InsertArc(G, v, w)；

```
// 添加弧
void InsertArc(Graph &G, char tail, char head)
{
    int tailIndex = Locate(G, tail);
    int headIndex = Locate(G, head);
    if (headIndex == -1 || tailIndex == -1) // 输入的弧头或者弧尾不存在
    {
        return;
    }
    G.Matrix[tailIndex][headIndex] = 1;
    G.arcNum++;
    if (!G.isDireted)
    {
        G.Matrix[headIndex][tailIndex] = 1; // 若不是有向图则再添加一条对称边
    }
}
```

Figure2.4 增加一条边<v，w>，InsertArc(G, v, w)代码

④ 删除一条边<v，w>，DeleteArc(G, v, w)。

```cpp
// 删除弧
void DeleteArc(Graph &G, char tail, char head)
{
    int tailIndex = Locate(G, tail);
    int headIndex = Locate(G, head);
    if (headIndex == -1 || tailIndex == -1)
    {
        return;
    }
    G.Matrix[tailIndex][headIndex] = 0;
    G.arcNum--;
    if (!G.isDireted)
    {
        G.Matrix[headIndex][tailIndex] = 0; // 若不是有向图则再删除一条对称边
        G.arcNum--;
    }
}
```

Figure2.5 删除一条边<v，w>，DeleteArc(G, v, w)代码

（二）：邻接矩阵为存储结构

① 增加一个新顶点 v，InsertVex(G, v)；

```cpp
void InsertVex(Graph &G, char v)
{
    G.verList[G.vertexNum].nodeData = v;
    G.verList[G.vertexNum].next = nullptr;
    G.vertexNum++;
}
```

Figure2.6 增加一个新顶点 v，InsertVex(G, v)代码

② 删除顶点 v 及其相关的边，DeleteVex(G, v)；

```
void DeleteVex(Graph &G, char v)
{
    int index = Locate(v, G); // 该顶点的下标
    // 释放与该顶点有关的所有边
    Node *p = G.verList[index].next; // 临时指针p指向该顶点邻接表的第一个结点
    while (p != nullptr)
    {
        if (!G.isDireted) // G为无向图
        {
            Node *q = G.verList[p->index].next; // 临时指针q指向待删除结点所在邻接表中的第一个结点
            while (q->next->index != index)     // 一直找，直到找到q的next结点的index值为删除顶点的index
            {
                q = q->next;
            }
            Node *needDel = q->next; // 创建一个临时指针，指向待删除的结点
            q->next = needDel->next; // 保证邻接表不断
            free(needDel);           // 释放空间
        }
        G.verList[index].next = p->next;
        G.arcNum--;
        free(p);
    }
    // 释放顶点
    G.verList[index].nodeData = NULL;
    G.verList[index].next = nullptr;
    G.vertexNum--;
    return;
}
```

Figure2.7 删除顶点 v 及其相关的边，DeleteVex(G, v)代码

③ 增加一条边<v，w>，InsertArc(G, v, w)；

```
void InsertArc(Graph &G, char tail, char head)
{
    int TailIndex, HeadIndex;
    TailIndex = Locate(tail, G);
    HeadIndex = Locate(head, G);
    if (HeadIndex == -1 || TailIndex == -1) // 输入的弧头或者弧尾不存在
    {
        return;
    }
    // 无论G为有向图还是无向图
    Node *newNode = new Node;
    newNode->next = G.verList[TailIndex].next; // 头插法插入到邻接表中
    newNode->index = HeadIndex;
    G.verList[TailIndex].next = newNode;
    if (!G.isDireted) // G为无向图
    {
        Node *newNode = new Node;
        newNode->next = G.verList[HeadIndex].next; // 头插法插入到邻接表中
        newNode->index = TailIndex;
        G.verList[HeadIndex].next = newNode;
    }
}
```

Figure2.8 增加一条边<v，w>，InsertArc(G, v, w)代码

④ 删除一条边<v，w>，DeleteArc(G, v, w)。

```
void DeleteVex(Graph &G, char v)
{
    int index = Locate(v, G); // 该顶点的下标
    // 释放与该顶点有关的所有边
    Node *p = G.verList[index].next; // 临时指针p指向该顶点邻接表的第一个结点
    while (p != nullptr)
    {
        if (!G.isDireted) // G为无向图
        {
            Node *q = G.verList[p->index].next; // 临时指针q指向待删除结点所在邻接表中的第一个结点
            while (q->next->index != index)     // 一直找，直到找到q的next结点的index值为删除顶点的index
            {
                q = q->next;
            }
            Node *needDel = q->next; // 创建一个临时指针，指向待删除的结点
            q->next = needDel->next; // 保证邻接表不断
            free(needDel);           // 释放空间
        }
        G.verList[index].next = p->next;
        G.arcNum--;
        free(p);
    }
    // 释放顶点
    G.verList[index].nodeData = NULL;
    G.verList[index].next = nullptr;
    G.vertexNum--;
    return;
}
```

Figure2.9 删除一条边<v，w>，DeleteArc(G, v, w)代码

注：邻接矩阵的①②③④操作均在 HW10.1.cpp 中，可在 src 文件下查看相关源码，上诉所有代码均省略了前置函数如 Locate 等，均可以在 HW10.1.cpp 或 HW10.2.cpp 下查看。

3、已知如右图所示的有向图，请给出该图的

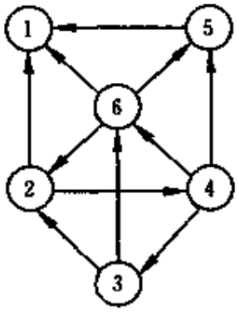
（1）每个顶点的入/出度；

（2）邻接矩阵；

（3）邻接表；

（4）逆邻接表；

（5）强连通分量



解：

（1）（下面将用①+来表示①顶点的入度，①-代表出度，其他顶点类推）

①+：3　①-：0；

②+：2　②-：2；

③+：1　③-：2；

④+：1　④-：3；

⑤+：2　⑤-：1；

⑥+：2　⑥-：3；

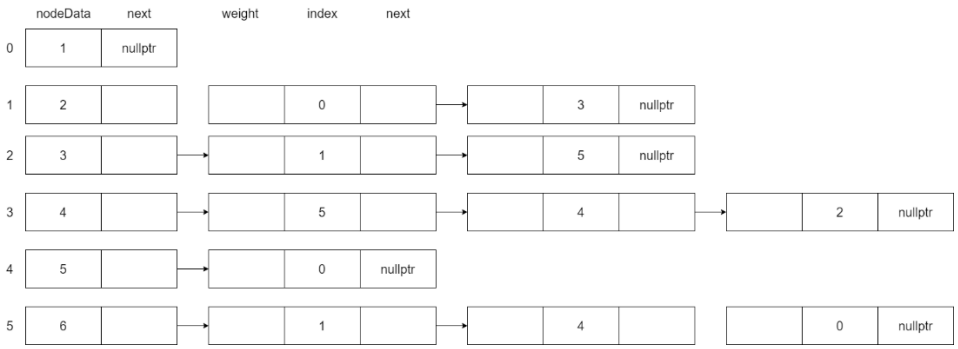
（2）

该图的邻接矩阵表示

	1	2	3	4	5	6
1	0	0	0	0	0	0
2	1	0	0	1	0	0
3	0	1	0	0	0	1
4	0	0	1	0	1	1
5	1	0	0	0	0	0
6	1	1	0	0	1	0

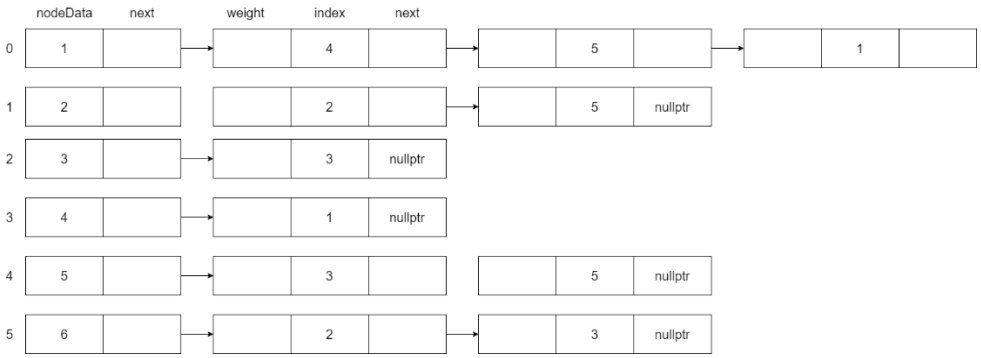
(3)

该图的邻接表表示

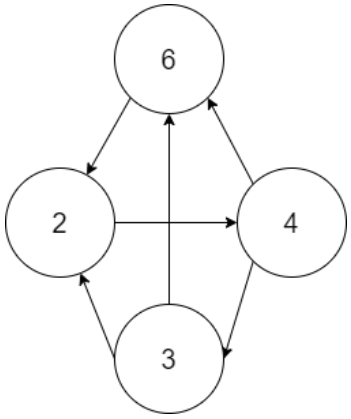


(4)

该图的逆邻接表表示



(5)



----------END----------