第16章

图(GRAPHS)

- 16.1　基本概念
- 16.2　应用和更多的概念
- 16.3　特性
- 16.4　抽象数据类型graph
- 16.5　无权图的描述
- 16.6　有权图的描述
- 16.7　类实现
- 16.8　图的遍历
- 16.9　应用

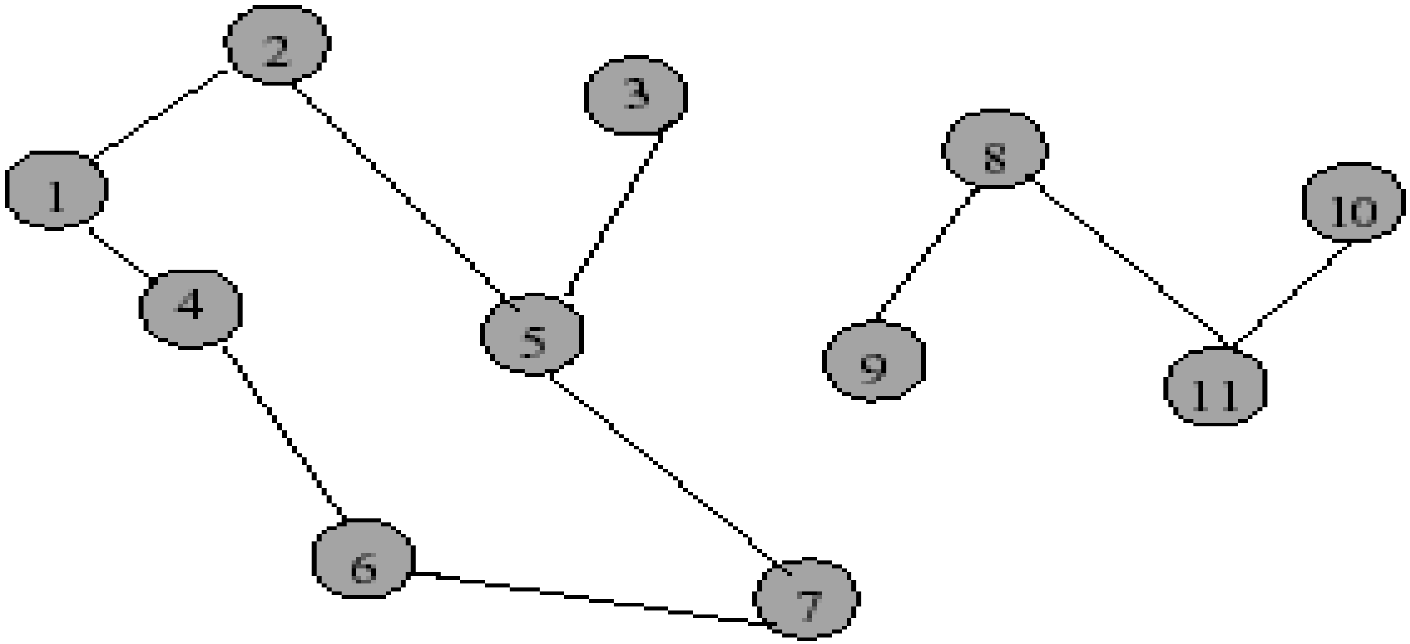
16.1 基本概念

- G = (V,E)
- V 是顶点集. E是边集.
- **顶点**也叫作**节点**(nodes)和**点**(points).
- E中的每一条**边**连接V中两个不同的顶点。边也叫作**弧**(arcs)或**连线**(lines) 。可以用**(*i*, *j*)**来表示一条边，其中 *i*和 *j*是*E*所连接的两个顶点。
- **无向边(**undirected edge): (*i*,*j*)和(*j*,*i*)是一样的。
- **有向边**(directed edge):(*i*,*j*)和(*j*,*i*)是不同的。

图

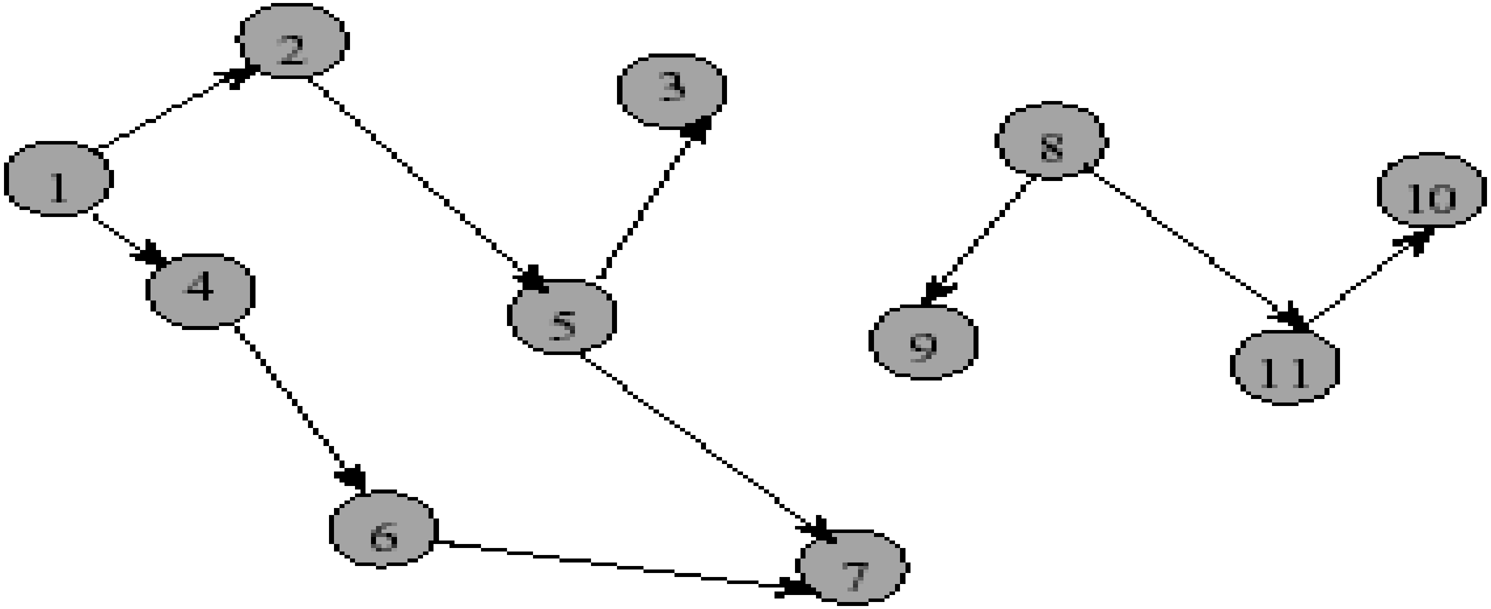
- **无向图**(Undirected graph) →图中所有的边都是无向边 .
- **有向图**(Directed graph) →图中所有的边都是有向边.
- 通常把无向图称为图；有向图称为有向图。

无向图



- 当(*i*,*j*)是图中的边时，
- 顶点*i*和*j*是**邻接的**(adjacent)。(*j*是i的**邻接点**； *i*是j的**邻接点.**)
- 边(*i*,*j*)**关联于**(incident)顶点*i*和*j*。

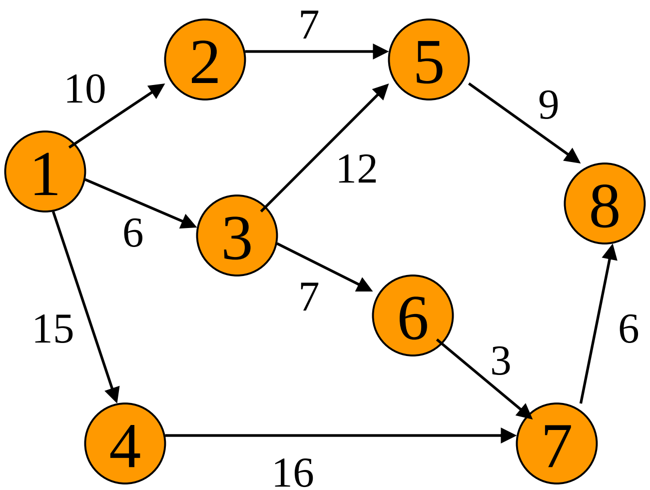
有向图

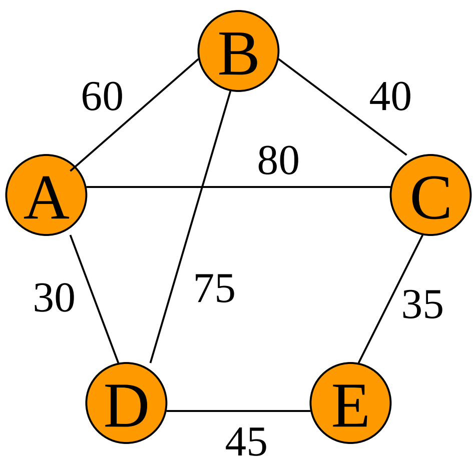


- 在有向图中，
  - **有向边(*i*, *j*)** 是关联至(incident to)顶点***j*** 而关联于(incident from)顶点***i***。
  - **顶点*i*** 邻接至(adjacent to)顶点***j***，顶点***j***邻接于(adjacent from)顶点***i***。

- **网络**(Network):**加权有向图**(Weight digraph)或**加权无向图**(Weight graph)，每一条边都赋予一个权值或耗费。
- 所有图都可以看作网络的一种特殊情况:
  - 一个无向(有向)图可以被看作是一个所有边具有相同权的无向(有向)网络。

网络(Network):





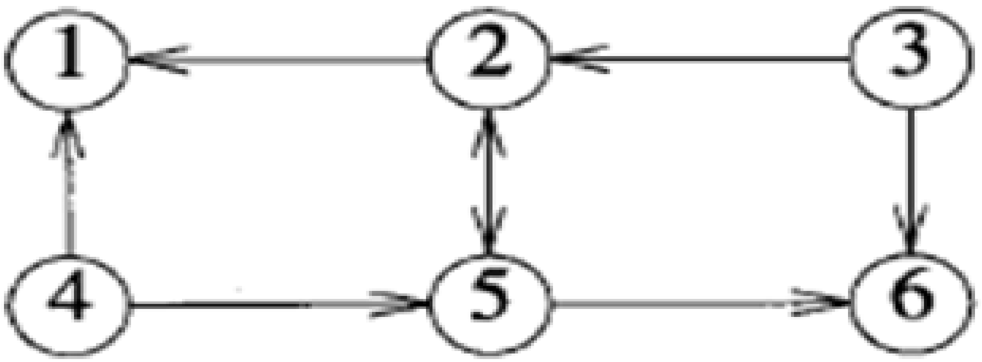
16.2 应用和更多的概念

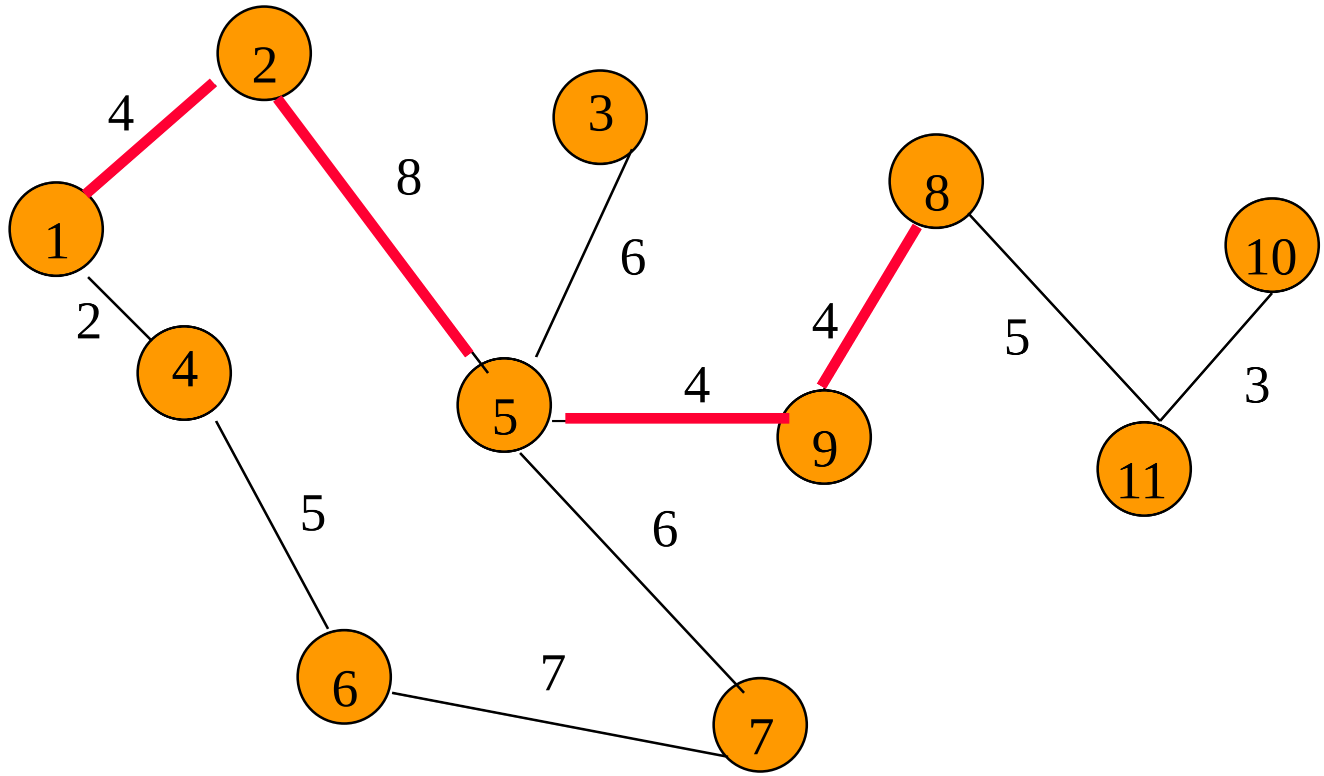
- 例16-1 路径问题

- 例16-2 生成树

例16-1　路径问题

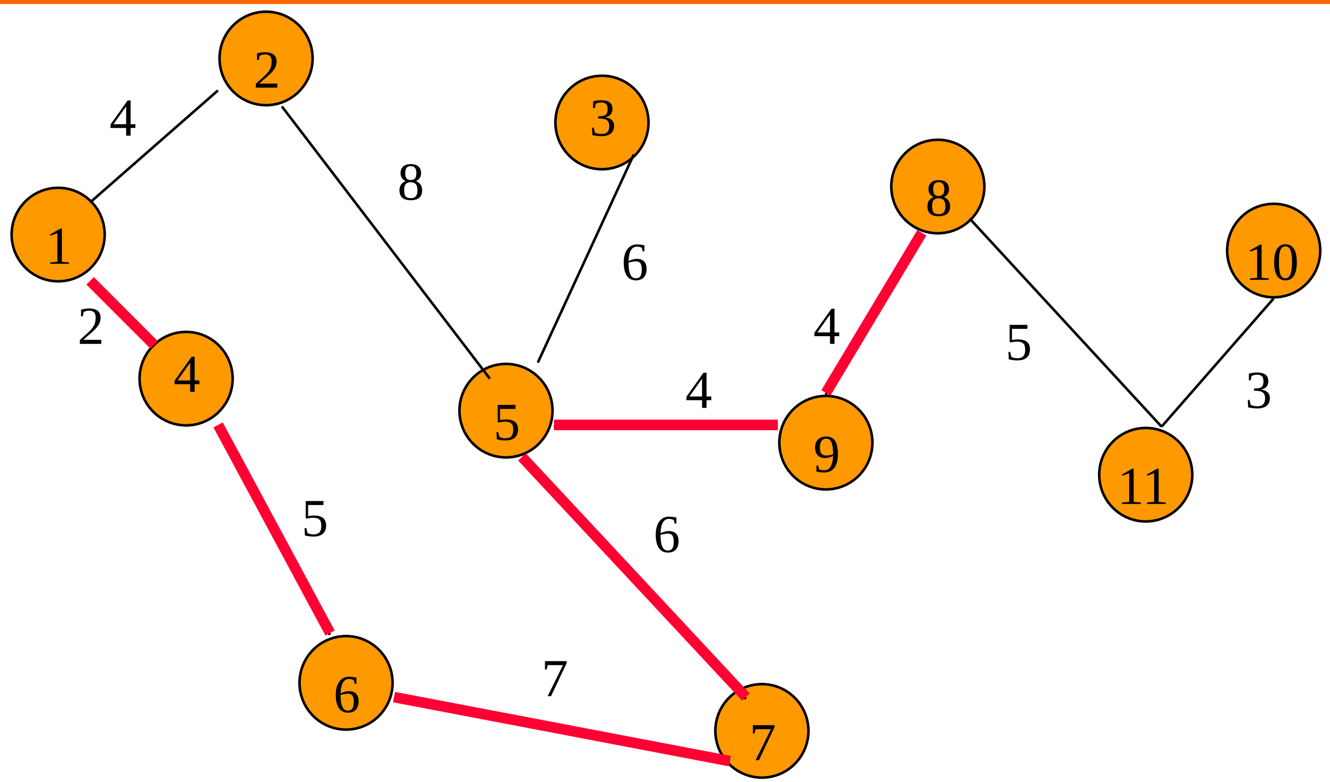
- 当且仅当对于每一个$j(1\le j\le k)$，边(i_j,i_{j+1})都在E中时，**顶点序列P=i_1, i_2, i_3, …i_k**,是图或有向图$G=(V, E)$中一条从i_1到i_k的**路径**。
- **简单路径**是这样一条路径：除第一个和最后一个顶点以外，路径中其他所有顶点均不同.
- **路径的长度**是路径上所有边的长度之和。
- **顶点*i* 到*j* 的最短路径**.





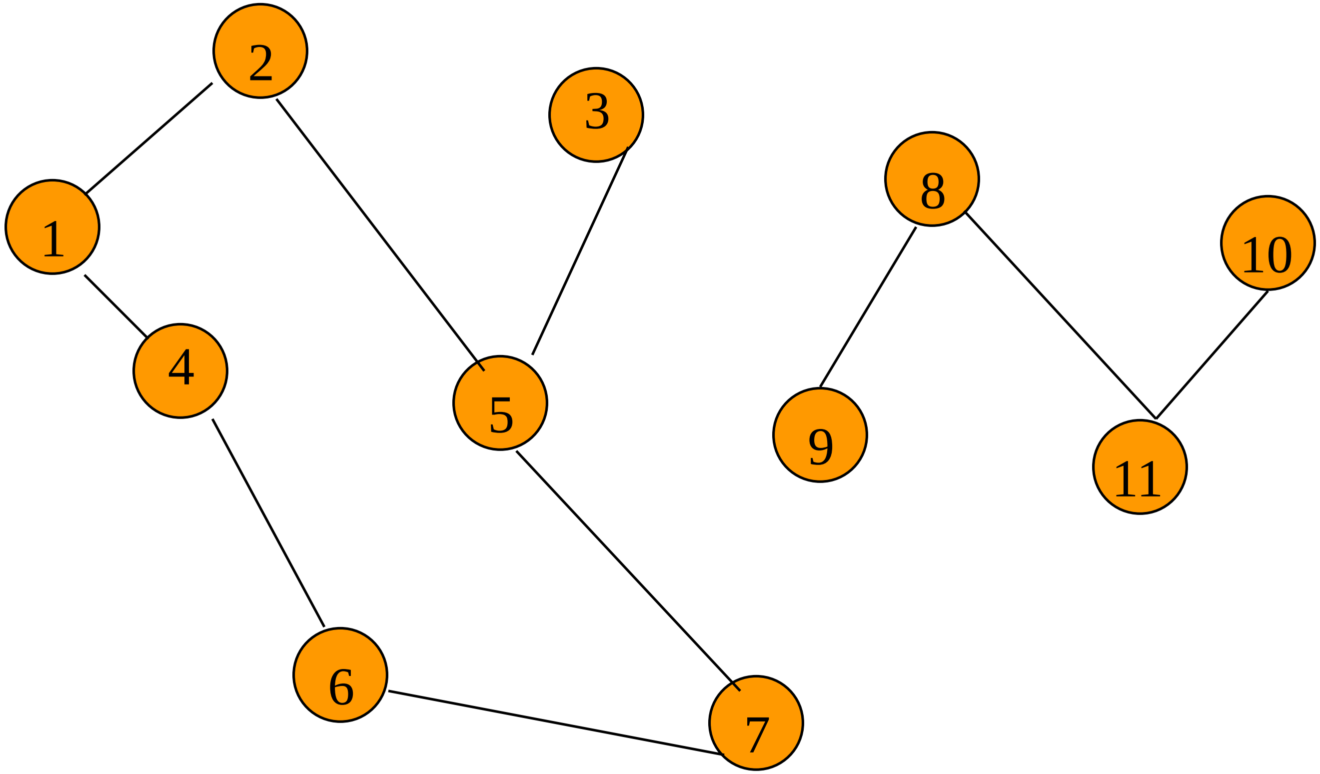
从 1 到 8的一条路径.

路径长度是：20.



从1到8的另一路径.

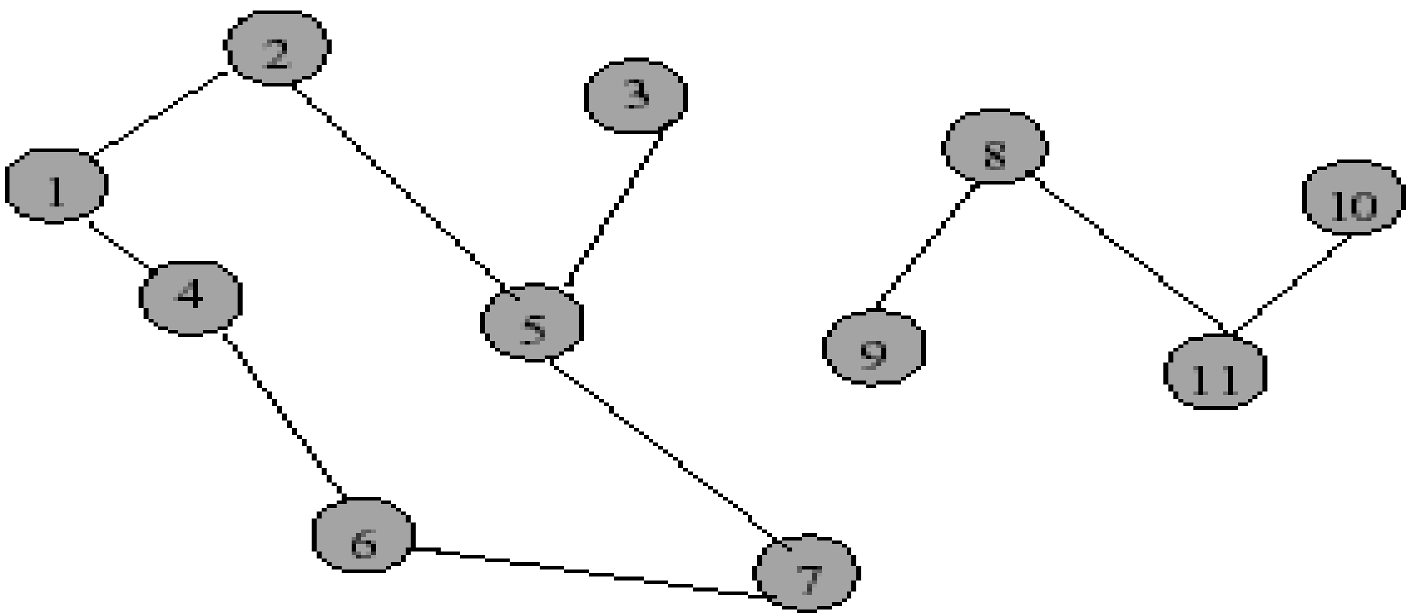
路径长度是： 28.



从2 到 9没有路径。

例16-2 生成树

- 设*G*= (*V*,*E*)是一个无向图，当且仅当*G*中每一对顶点之间有一条路径时，可认为*G*是连通图(Connected Graph)。

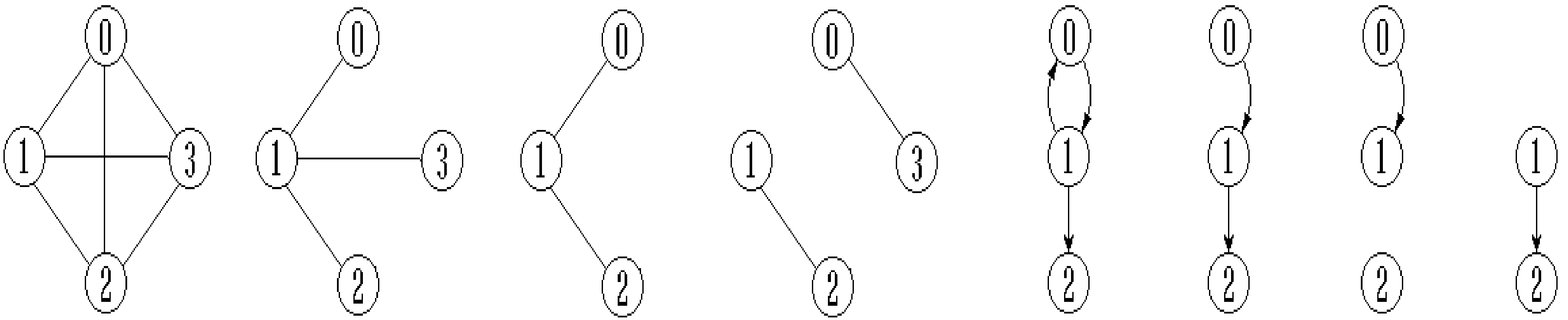


例16-2　生成树

- *H*是**图*G*的子图**(subgraph)的充要条件是，*H*的顶点和边的集合是*G*的顶点和边的集合的子集。
- **环路**(Cycle/回路):起始节点与结束节点是同一节点的简单路径。
- **树**(Tree):**没有环路**的**无向连通图**是一棵树。
- **树的耗费**(cost of tree/树的代价)：　树的耗费是所有边的耗费(weights/costs)之和 .
- **图G的生成树**(Spanning Tree of G):一棵包含*G*中所有顶点并且是*G*的子图的树是*G*的生成树。
- 一个*n*节点的连通图必须至少有*n*-1条边 .
- ⇒如果图*G*有 n 个顶点, 那么**图*G*的生成树有 n 个顶点和有*n*-1条边。**

子图

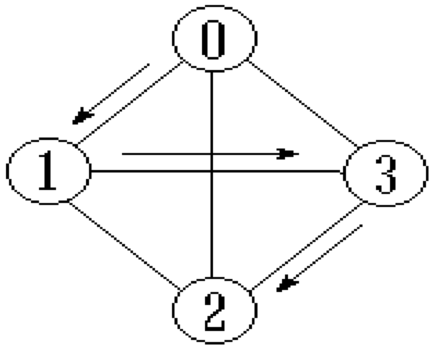
- 图H是图G的子图(subgraph)的充要条件是，H的顶点和边的集合是G的顶点和边的集合的子集。



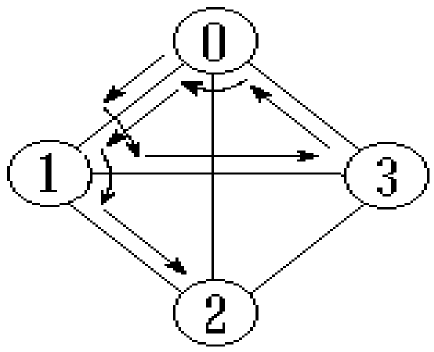
(a) G1　子图　子图　子图　(b) G3　子图　子图　子图

环路(回路)

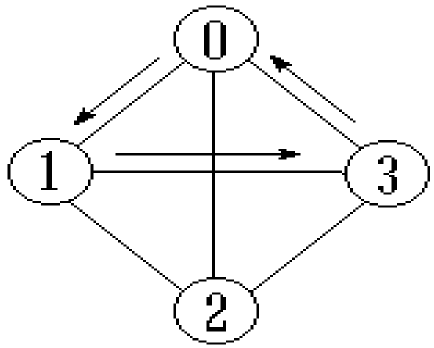
- 环路(cycle)的起始节点与结束节点是同一节点。



(a) 简单路径



(b) 非简单路径



(c) 回路

- 简单路径组成的回路称为简单回路。

生成树

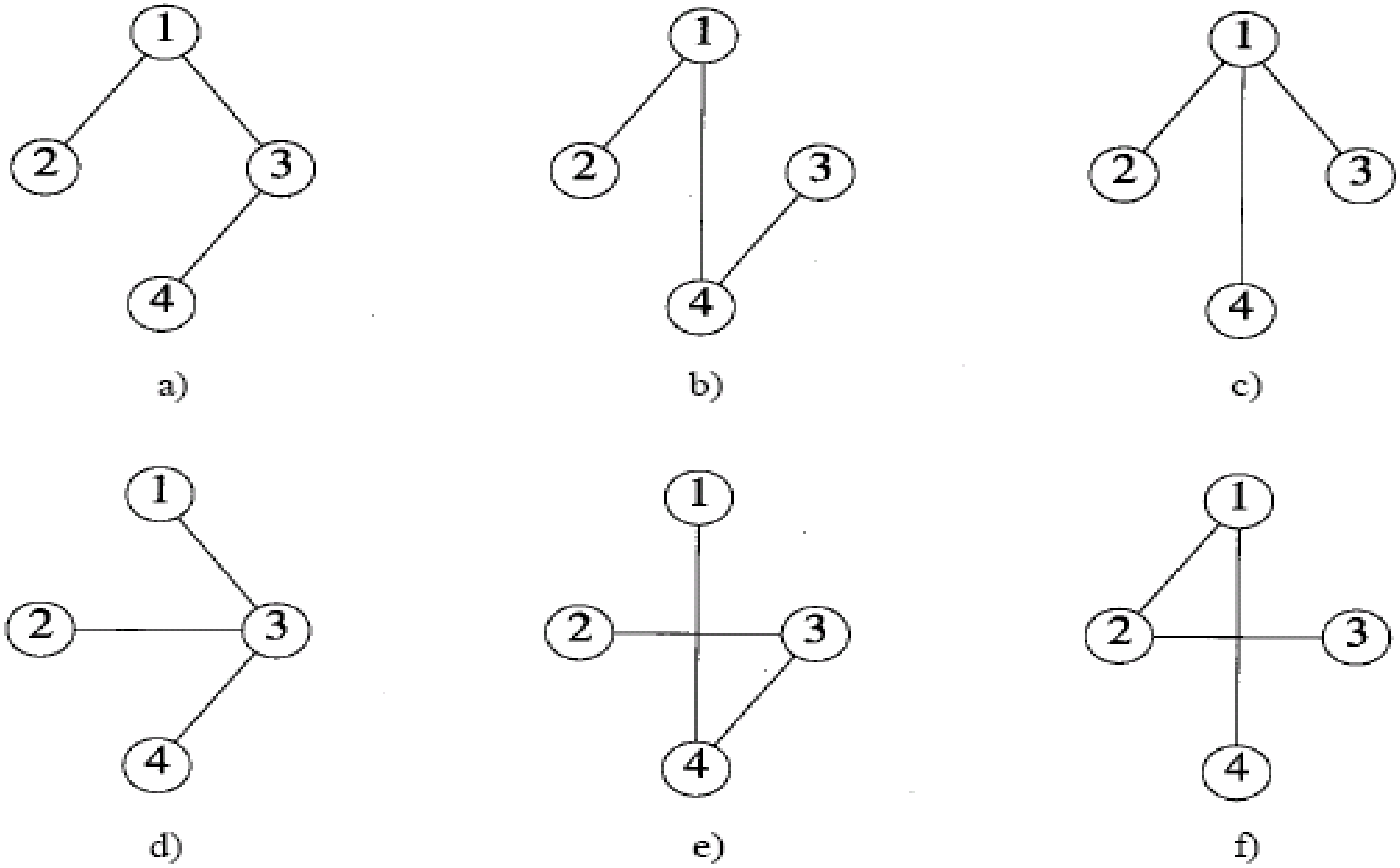
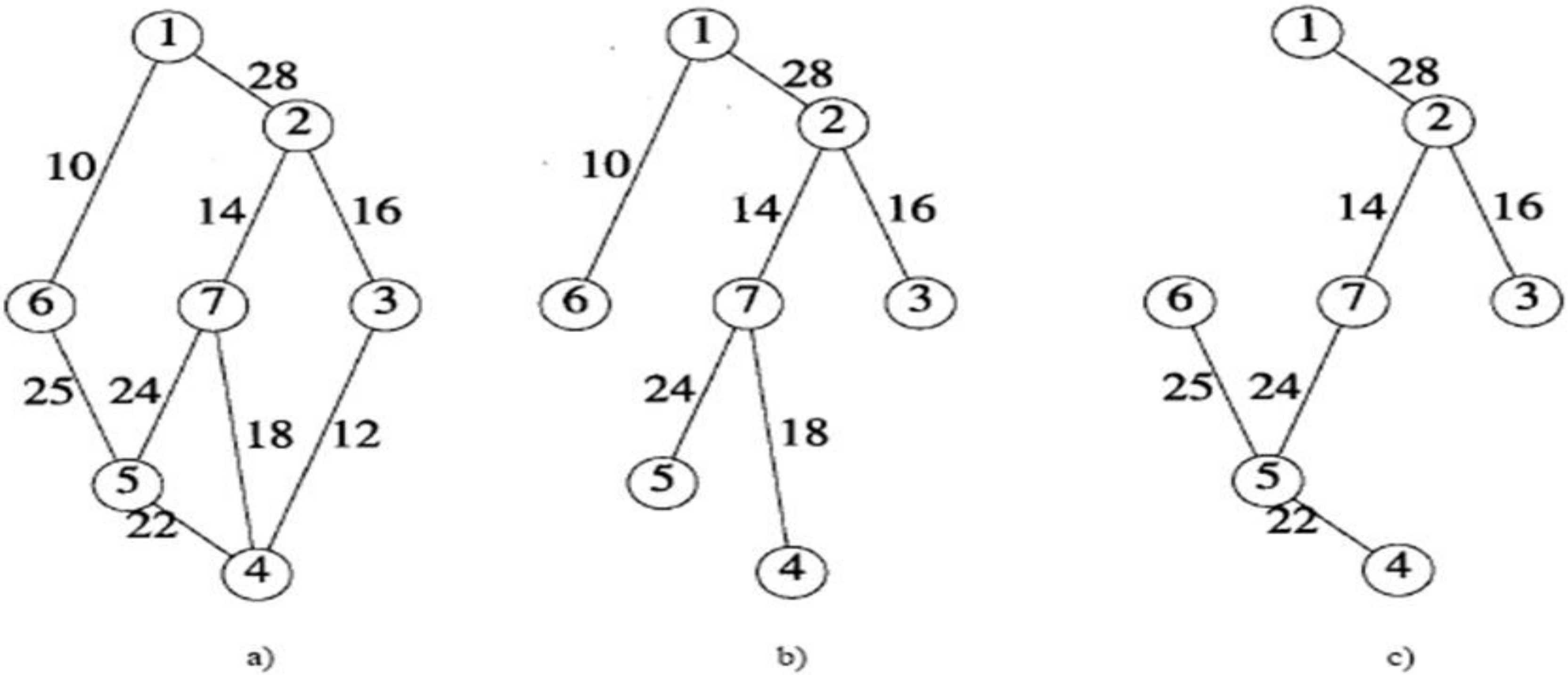


图12-3　图12-1a 的生成树

生成树

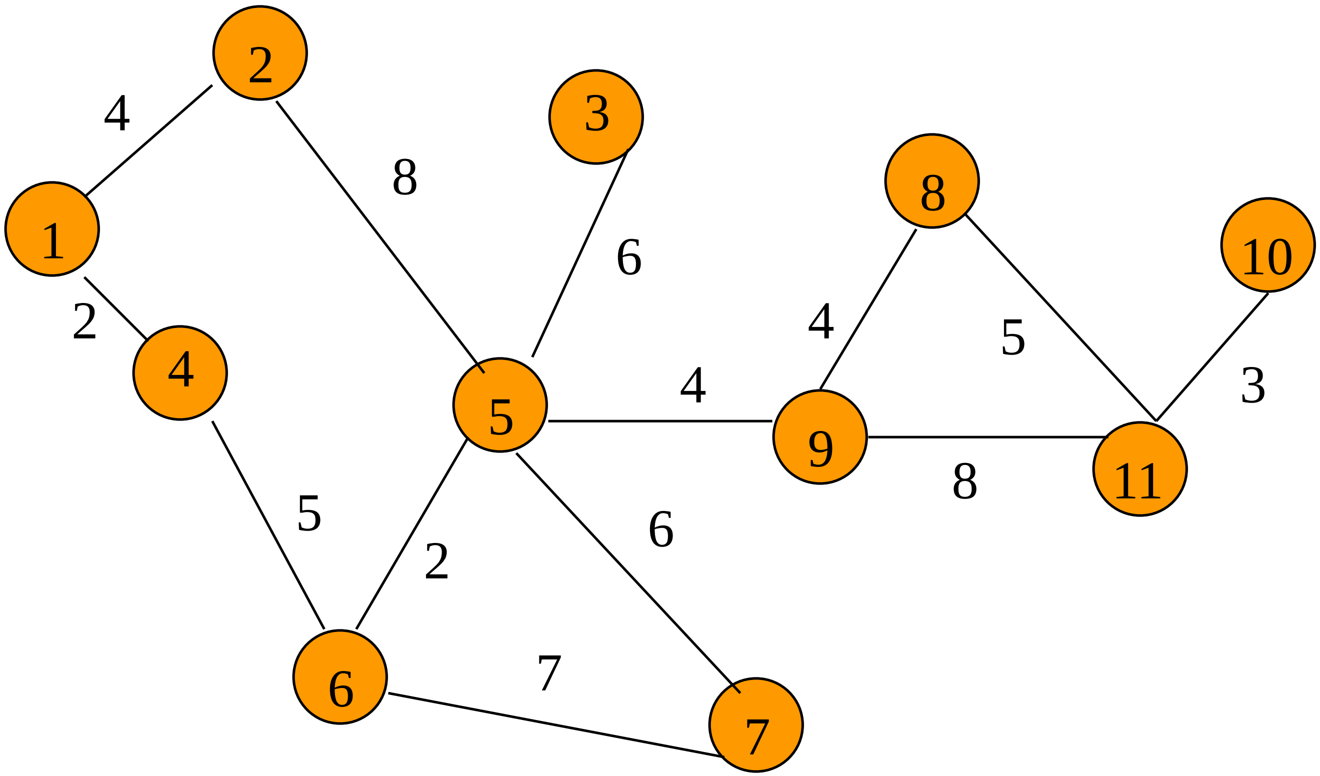
- 因此当通信网络的每条链路具有相同的建造费用时，在任意一棵生成树上建设所有的链路可以将网络建设费用减至最小，并且能保证每两个城市之间存在一条通信路径。
- 如果链路具有不同的耗费，那么需要在一棵最小耗费生成树(生成树的耗费是所有边的耗费之和)上建立链路。

生成树示例

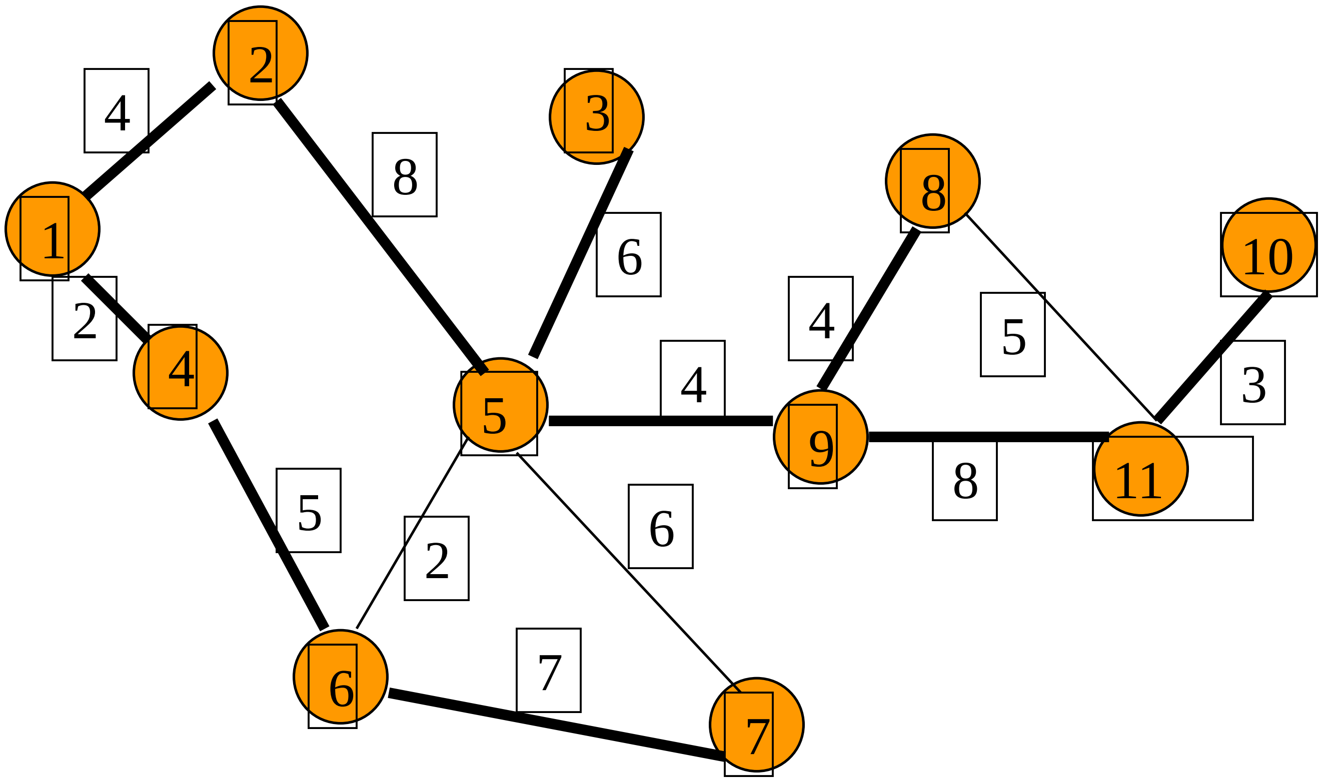


- 图a的两棵生成树:图b和图c
- 图b生成树耗费 = 100;.图c生成树耗费 = 129.
- **最小耗费生成树（最小代价生成树）：**生成树耗费（代价）达到最小的生成树.

生成树示例

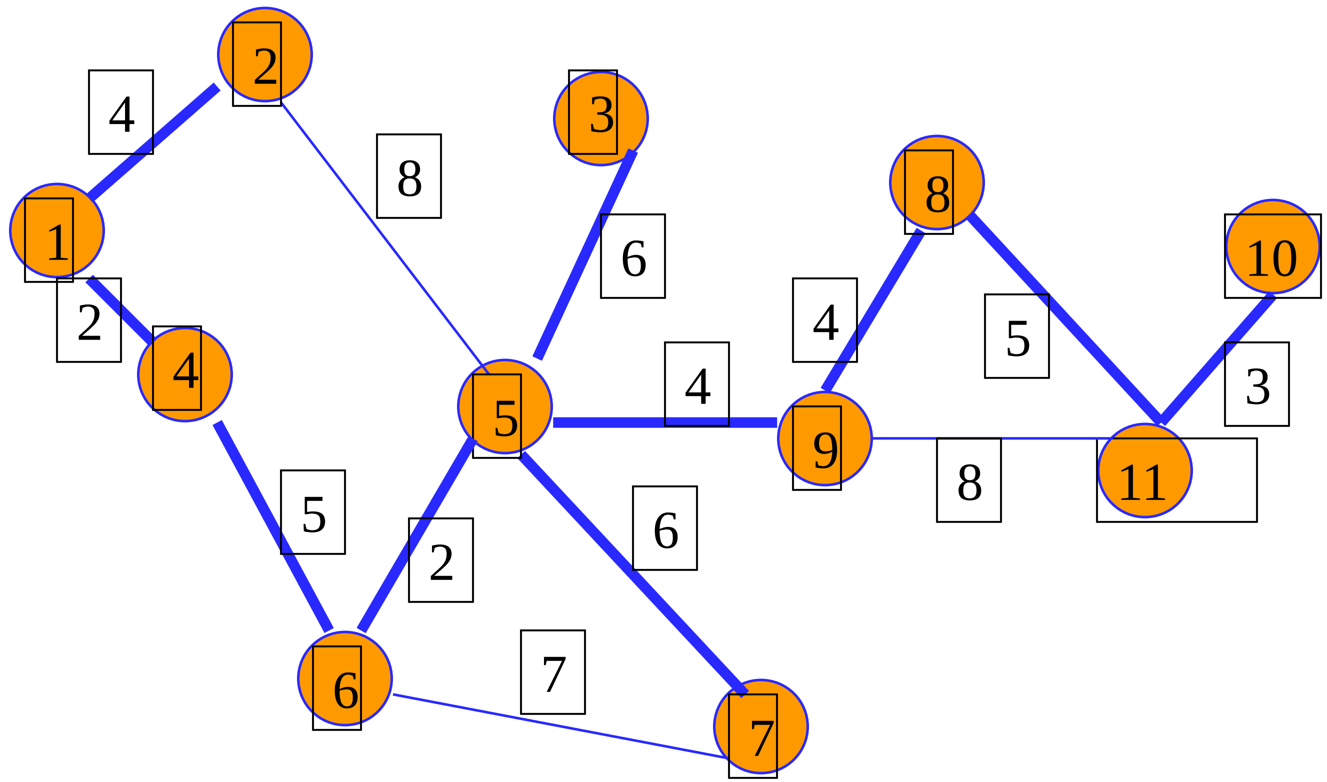


生成树示例



- **生成树耗费 = 51.**

生成树示例



生成树耗费= 41.

最小耗费生成树（最小代价生成树）： 生成树耗费（代价）达到最小的生成树.

例16-3

- 会议，此次大会上的所有发言人都只会说英语，而参加会议的其他人说的语言是{L1, L2, ..., Ln}之一。翻译小组能够将英语与其他语言互译。

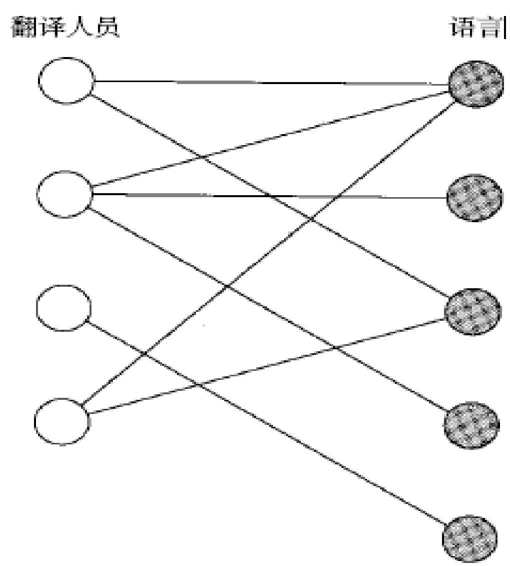


图12-5　翻译人员和语言

二分图

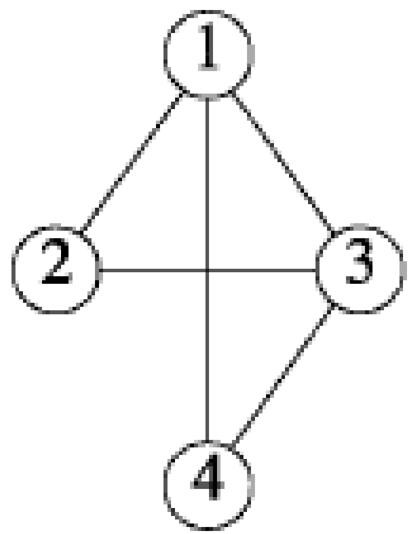
- 可以将顶点集合分成两个子集A和B，这样每条边在A中有一个端点，在B中有一个端点，具有这种特征的图叫作二分图(bipartite graph)

覆盖

- 当且仅当B中的每个顶点至少与A中一个顶点相连时，A的一个子集A' 覆盖集合B（或简单地说，A' 是一个覆盖）。
- 覆盖A' 的大小即为A' 中的顶点数目。
- 当且仅当A' 是覆盖B的子集中最小的时，A' 为最小覆盖。
- 在二分图中寻找最小覆盖的问题为二分覆盖(bipartite-cover)问题。
  - 二分覆盖问题是N P-复杂问题。
  - 可以利用贪婪算法寻找一种快速启发式方法。
  - 一种可能是分步建立覆盖A'，每一步选择A中的一个顶点加入覆盖。顶点的选择利用贪婪准则：从A中选取能覆盖B中还未被覆盖的元素数目最多的顶点。

16.3 特性

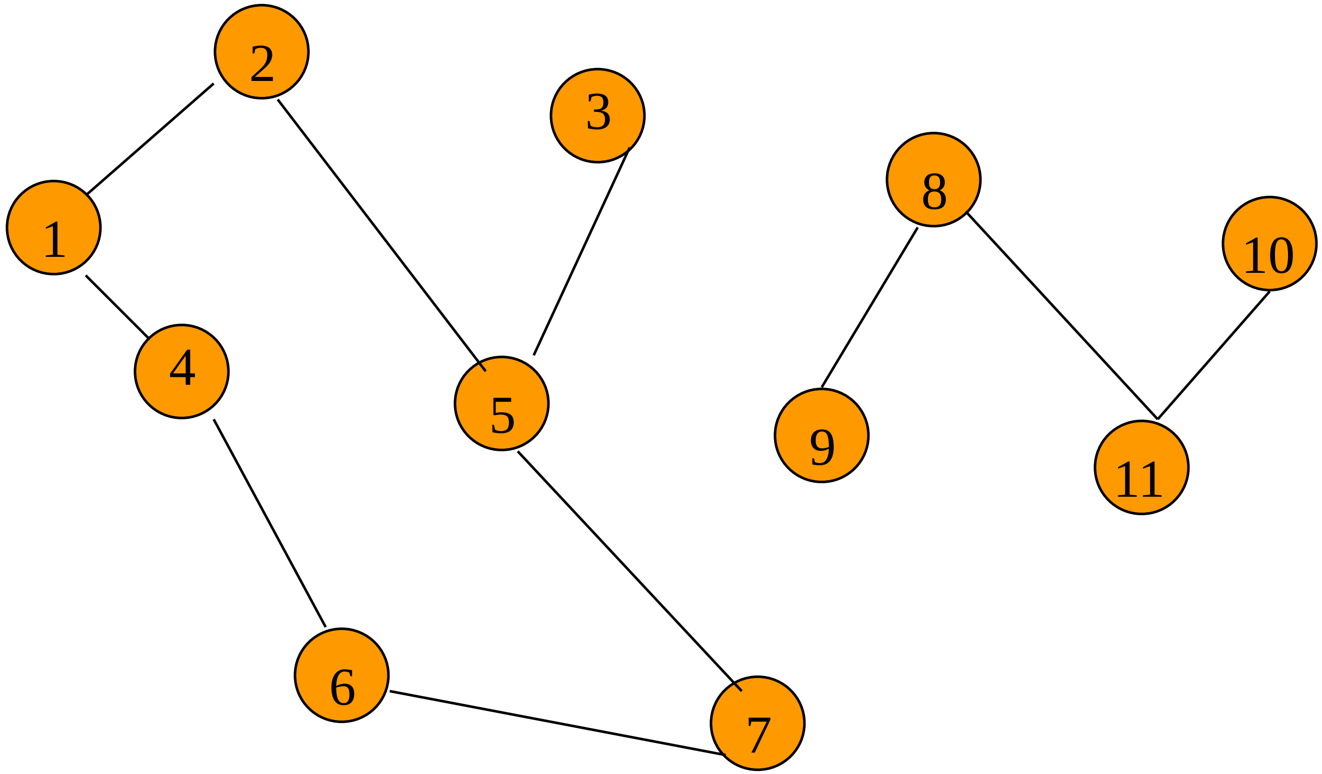
设*G*是一个**无向图，顶点*i*的度**(degree)d_i是与顶点*i*相连的边的个数。



$d_2 = 2, d_4 = 2, d_3 = 3$

16.3 特性

设*G*是一个无向图，顶点*i*的度(degree)d_i是与顶点*i*相连的边的个数。



$d_2= 2, d_5= 3, d_3 = 1$

16.3 特性

- 特性 16-1:
- 设G=(V,E) 是一无向图.令 |V|=n; |E| =e; d_i=顶点i的度，则.
- (a) $\sum_{i=1}^{n} d_i$ =2e.
- (b) $0 \le e \le n*(n-1)/2$.

证明:

(a) 无向图的每一条边与两个顶点相连

$\Rightarrow$顶点的度之和等于边的数量(e)的2倍 = 2e

(b) $0 \le d_i \le n-1$

$\Rightarrow 0 \le \sum_{i=1}^{n} d_i \le n*(n-1)$

$\Rightarrow 0 \le e \le n*(n-1)/2$

单选题 1分

设置

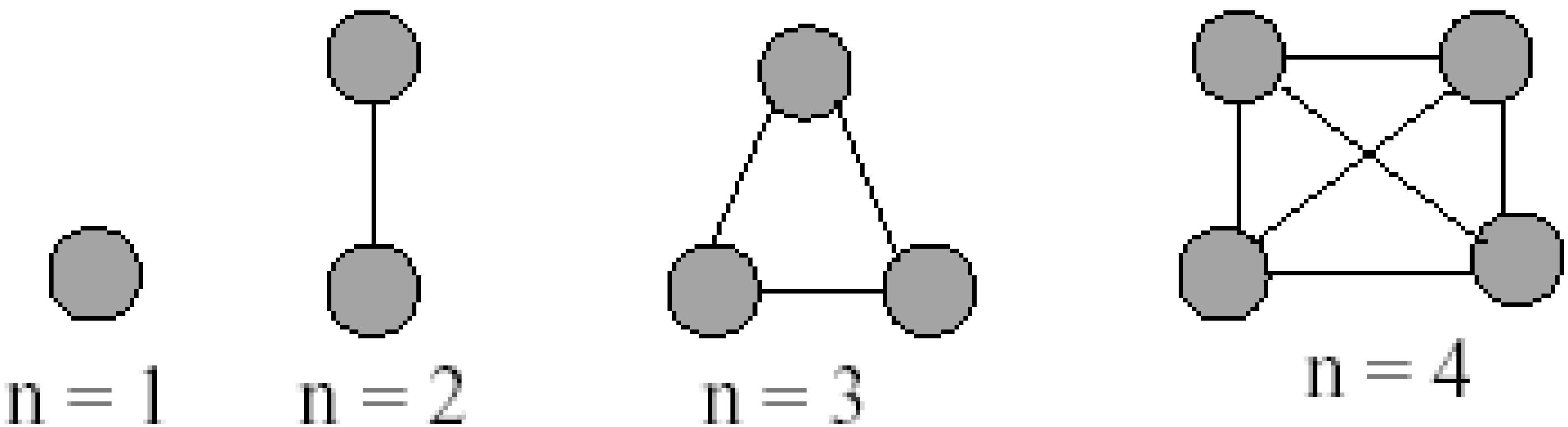
设G是任意无向图，度数为奇数的顶点。有偶数个还是奇数个?

- A 奇数
- B 偶数
- C 不确定

提交

完全(无向)图

- 一个具有*n* 个顶点， n(n-1)/2 条边的图是一个完全(无向)图.



若某无向图一共有16条边，并且有3个度为4的顶点，4个度为3的顶点，其余顶点的度均小于3，则该无向图至少有多少个顶点？至多有多少个节点？

A 11，15

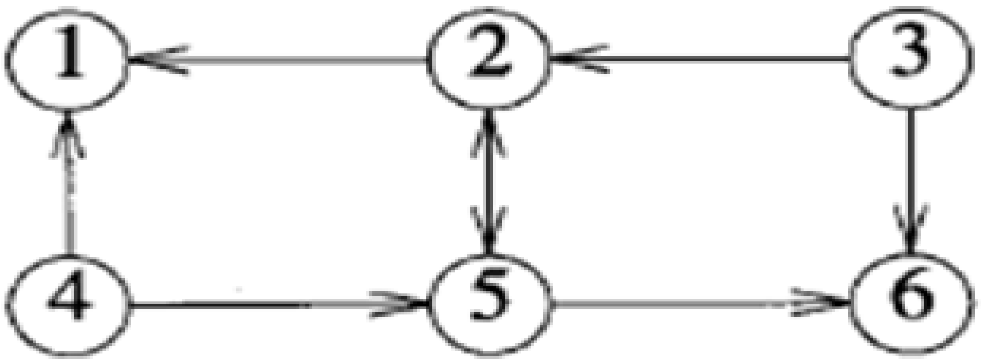
B 7，16

提交

连通图

- 设G= (V,E)是一个无向图，当且仅当G中每一对顶点之间有一条路径时，可认为G是连通的(connected)。
- 每一个n（n≥2）顶点的连通图至少包含n-1 条边。

顶点i的入度



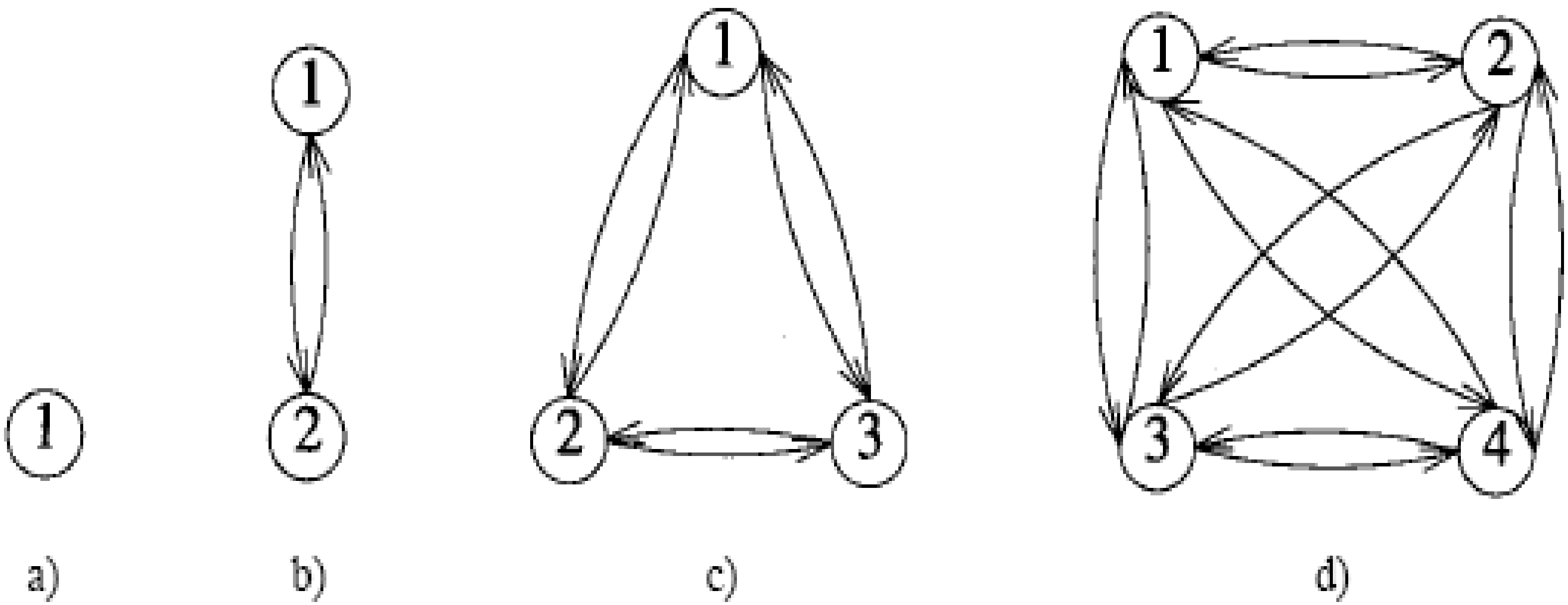
- 设G是一个**有向图，顶点i的入度**(in-degree) $\mathbf{d_i^{in}}$ 是指关联至**顶点i**的边的数量。
- $d_1^{in} = 2$, $d_3^{in} = 0$

- **顶点i 的出度**(out-degree) $\mathbf{d_i^{out}}$是指关联于该顶点的边的数量。
- $d_2^{out} = 2$, $d_6^{out} = 0$

- **特性 16-2:**
- 设G=(V,E) 是一有向图.令|V|=n; |E| =e;
- (a) $0 \le e \le n*(n-1)$.
- (b) $\sum_{i=1}^{n} d_i^{in} = \sum_{i=1}^{n} d_i^{out} = e$.

完全有向图

- 一个具有*n* 个顶点， n(n-1)条边的图是一个完全有向图.



n=1　　*n=2*　　*n=3*　　*n=4*

强连通有向图

- 一个有向图是强连通(strongly connected)的充要条件是：对于每一对不同顶点i 和j，从i 到j 和从j 到i 都有一个有向路径。
- 1）对于每一个n（n≥2），都存在一个包含n 条边的强连通有向图。
- 2）每一个n（n≥2）顶点的强连通有向图至少包含n 条边。

16.4　抽象数据类型graph

- ***graph：无向图、有向图、加权无向图、加权有向图***

抽象数据类型 *graph*
{实例
　　顶点集合V和边集合E。
　操作:
　　numberOfVertices()：返回图中顶点的数目
　　numberOfEdges()：返回图中边的数目
　　ExistsEdge (*i,j*)：如果边(*i,j*)存在,返回true，否则返回false
　　insertEdge (*theEdge*)：向图中添加边*theEdge*
　　eraseEdge (*i,j*)：删除边(*i,j*)
　　degree(*i*)：返回顶点*i*的度，只能用于无向图
　　inDegree(*i*)：返回顶点*i*的入度
　　outDegree(*i*)：返回顶点*i*的出度
}

抽象类graph

```
template <class T>
class  graph
{public:
  ……
 //ADT方法操作:
 virtual int numberOfVertices() const=0;
 virtual int numberOfEdges() const=0;
 virtual bool existsEdge (int,int) const=0;
 virtual void insertEdge (edge<T> *) const=0;
 virtual void eraseEdge (int,int) const=0;
 virtual int degree(int) const=0;
 virtual int inDegree(int) const=0;
 virtual int outDegree(int) const=0;
 //其他方法
 virtual bool directed() const=0;//当且仅当是有向图时，返回值是true
 virtual bool weighted() const=0; //当且仅当是加权图时，返回值是true
 virtual vertexIterator<T>* iterator(int) = 0;//访问指定顶点的邻接顶点
}
```

抽象类edge

```
template <class T>
class  edge
{public:
  ……
 //ADT方法操作:
 virtual int vertex1 ();
 virtual int vertex2 ();
 virtual int weight();
}
```

抽象类vertexIterator

```
template <class T>
class  vertexIterator
{public:
  ……
  //ADT方法操作:
 virtual int next()=0;
 virtual int next(T&)=0;
}
```

令 g 是指向一个图的一个指针。我们可以用下面的语句创建顶点 5 的一个迭代器：

```
vertexIterator<T> *vertex5Iterator=iterator(5);
```

接下来的调用语句 vertex5Iterator->next() 将返回一个与顶点 5 相邻的顶点，假设是 j，这时（5，j）是图的一条边。如果没有相邻的顶点，返回值是 0。当 g 指向的是一个加权图时，假设调用语句 vertex5Iterator->next(w) 的返回值是 j，这时 w 成为边（5，j）的权。

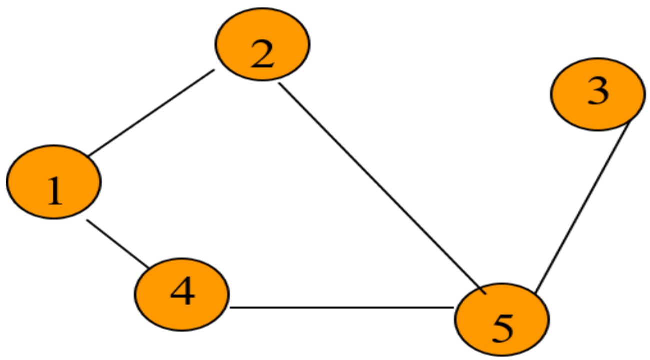
16.5 无权图的描述

- 邻接矩阵(adjacency matrix)
- 邻接链表(linked-adjacency-lists)
- 邻接数组

邻接矩阵

- 无权图G=(V,E)， 令|V|=n; 假设： V={1,2,3,…,n}
- G的邻接矩阵：**0/1 n×n矩阵A,**
- G 是一无向图
  - $A(i,j) = \begin{cases} 1 & \text{如果}(i,j) \in E \text{ 或 } (j,i) \in E . \\ 0 & \text{其它} \end{cases}$
- G是一有向图
  - $A(i,j) = \begin{cases} 1 & \text{如果}(i,j) \in E. \\ 0 & \text{其它} \end{cases}$

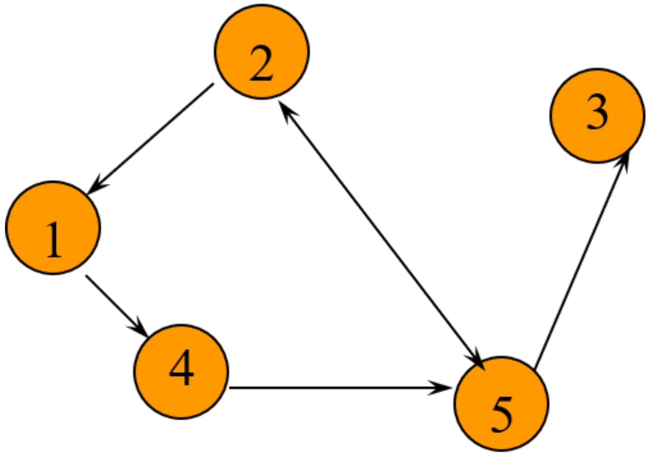
无向图的邻接矩阵-特性1/2



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

- 对于*n*顶点的无向图，有
  - *1. A*(*i*,*i*)=0， 1≤*i*≤*n*
  - 2. 邻接矩阵是**对称的**， 即 (A(i,j) = A(j,i)， 1≤*i*≤*n*， 1≤*j*≤*n*)
  - 3. $$\sum_{j=1}^{n} A(i,j) = \sum_{j=1}^{n} A(j,i) = \mathrm{d}_i$$

有向图的邻接矩阵-特性



$$
\begin{array}{c|ccccc}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 0 & 0 & 1 & 0 \\
2 & 1 & 0 & 0 & 0 & 1 \\
3 & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 0 & 0 & 1 \\
5 & 0 & 1 & 1 & 0 & 0
\end{array}
$$

- 对于n顶点的有向图，有：

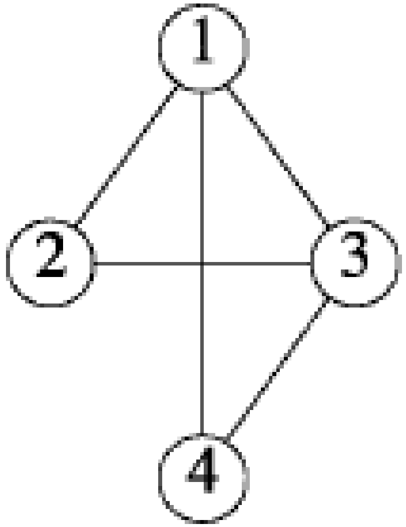
$$\sum_{j=1}^{n} A(i,j) = \mathrm{d}_{\mathrm{i}}^{\mathrm{out}};\sum_{j=1}^{n} A(j,i) = \mathrm{d}_{\mathrm{i}}^{\mathrm{in}};1 \le i \le n$$

邻接矩阵的存储

- 使用$(n+1)\times(n+1)$的布尔型数组a，映射$A(i,j)=1$，当且仅当$a[i][j]$=true
  - 需要$(n+1)^2$字节空间
- 减少存储空间：
  - 采用$n\times n$数组$a[n][n]$，映射$A(i,j)=1$，当且仅当$a[i-1][j-1]$=true
    - 需要n^2字节，比前一种减少了$2n+1$个字节
  - 不储存所有对角线元素(都是零)
    - 减少n个字节空间
  - 以上减少存储空间的办法，代码容易出错，某些操作代价大。
- 对无向图，只需要存储上三角(或下三角)的元素

16.5.2　邻接链表

- **顶点i的邻接表**(adjacency list)：是一个**邻接于顶点i**的顶点的线性表,包含了顶点i的所有邻接点。



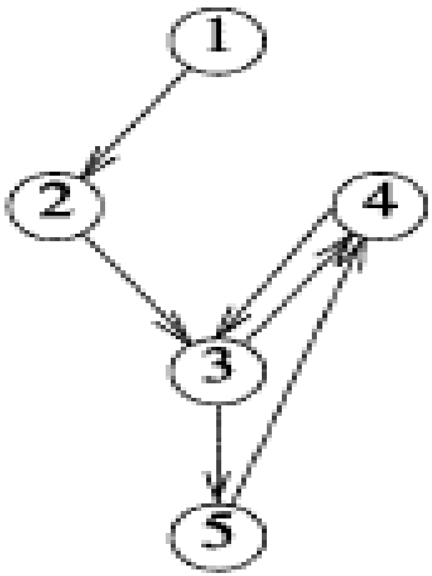
顶点1的邻接表 = (2，3,4)

顶点2的邻接表 = (1,3)

顶点3的邻接表 = (1，2，4)

顶点4的邻接表 = (1,3)

有向图邻接表示例



顶点1的邻接表 = (2)

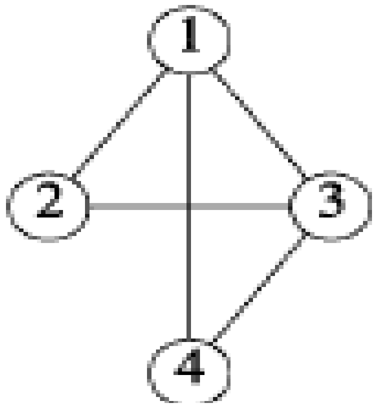
顶点2的邻接表 = (3)

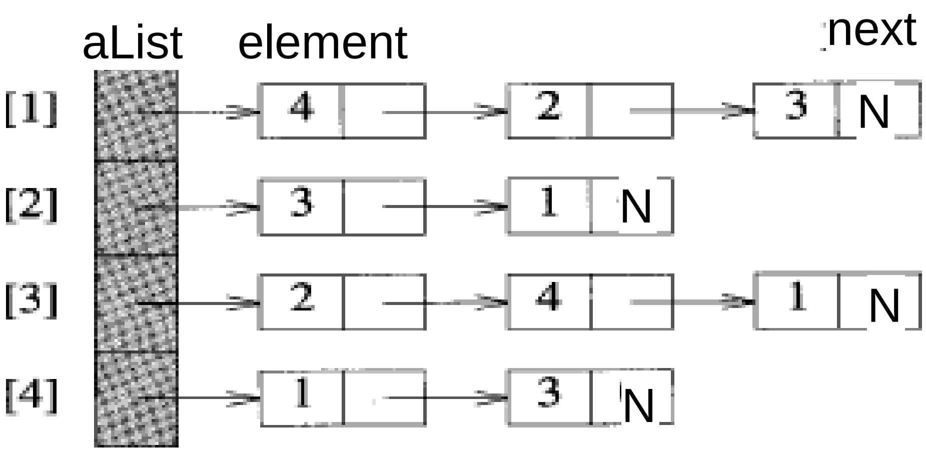
顶点3的邻接表 = (5,4)

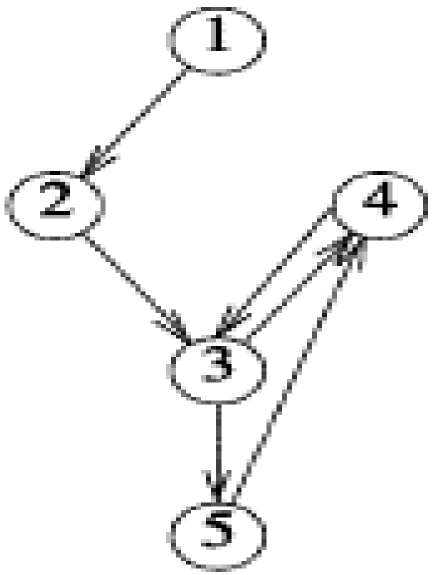
顶点4的邻接表 = (3)

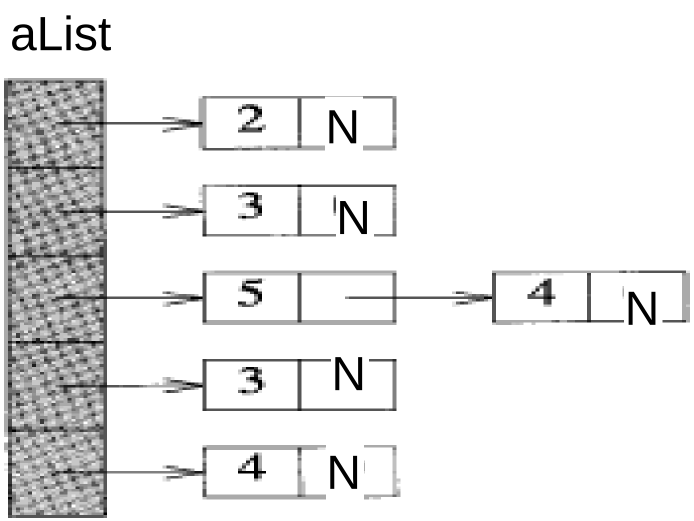
顶点5的邻接表 = (4)

邻接链表

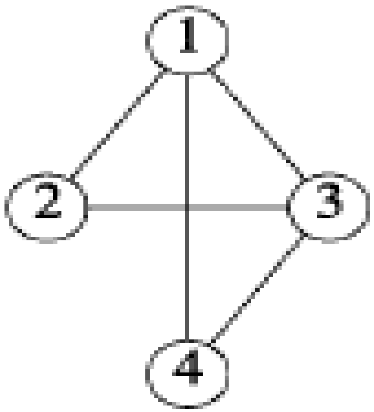

1
2
3
4

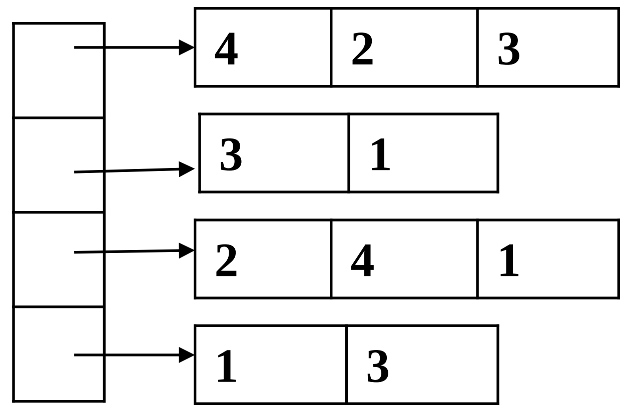

aList
element
next
[1]
[2]
[3]
[4]
4
2
3
N
3
1
N
2
4
1
N
1
3
N

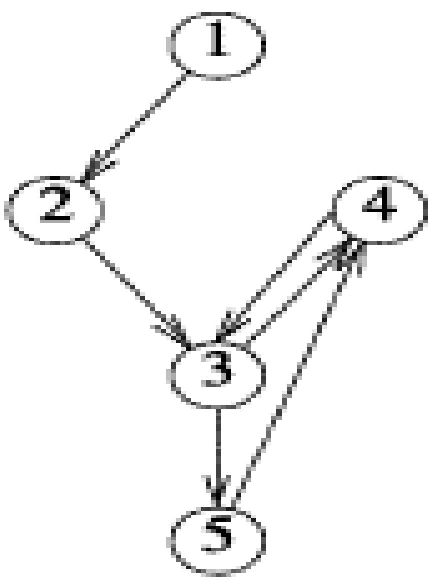

1
2
3
4
5

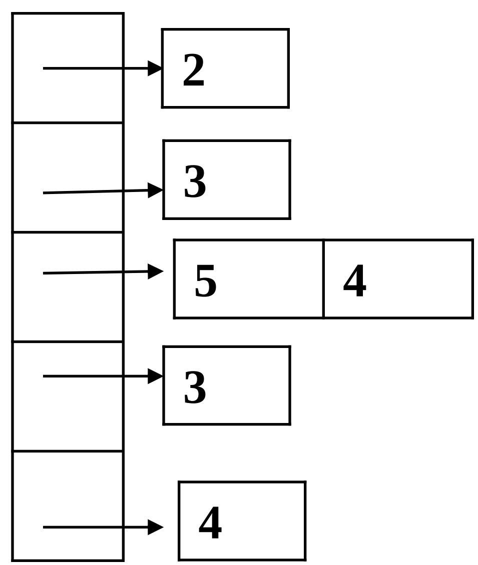

aList
2
N
3
N
5
4
N
3
N
4
N

邻接数组







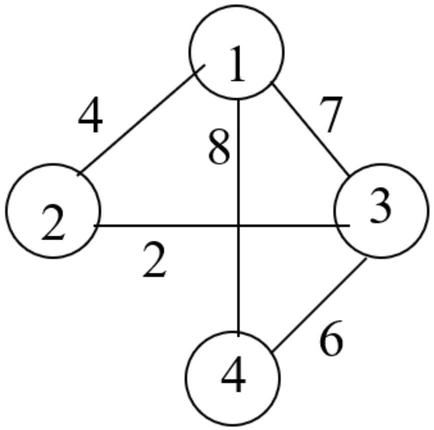


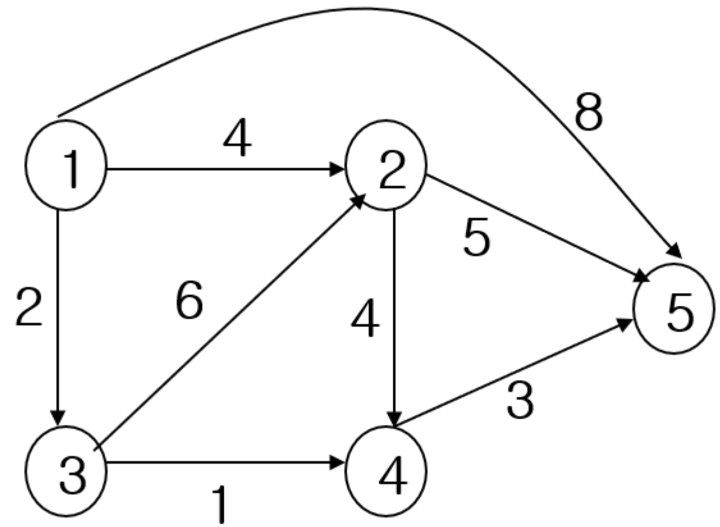
- 在邻接数组中，每个顶点的邻接表使用一个数组描述。

16.6 加权图(网络)描述

- 耗费/代价/成本(cost)邻接矩阵:
- G 是一加权无向图
  - $$C(i,j) = \begin{cases} \text{边 }(i,j)\text{的耗费(或权)} & \text{如果}(i,j) \in E \text{ 或 } (j,i) \in E \\ -\text{ 或 } \infty & \text{其它} \end{cases}$$
- G是一加权有向图
  - $$C(i,j) = \begin{cases} \text{边 }(i,j)\text{的耗费(或权)} & \text{如果}(i,j) \in E. \\ -\text{ 或 } \infty & \text{其它} \end{cases}$$
- -或 ∞: 用noEdge表示

加权图邻接矩阵描述示例

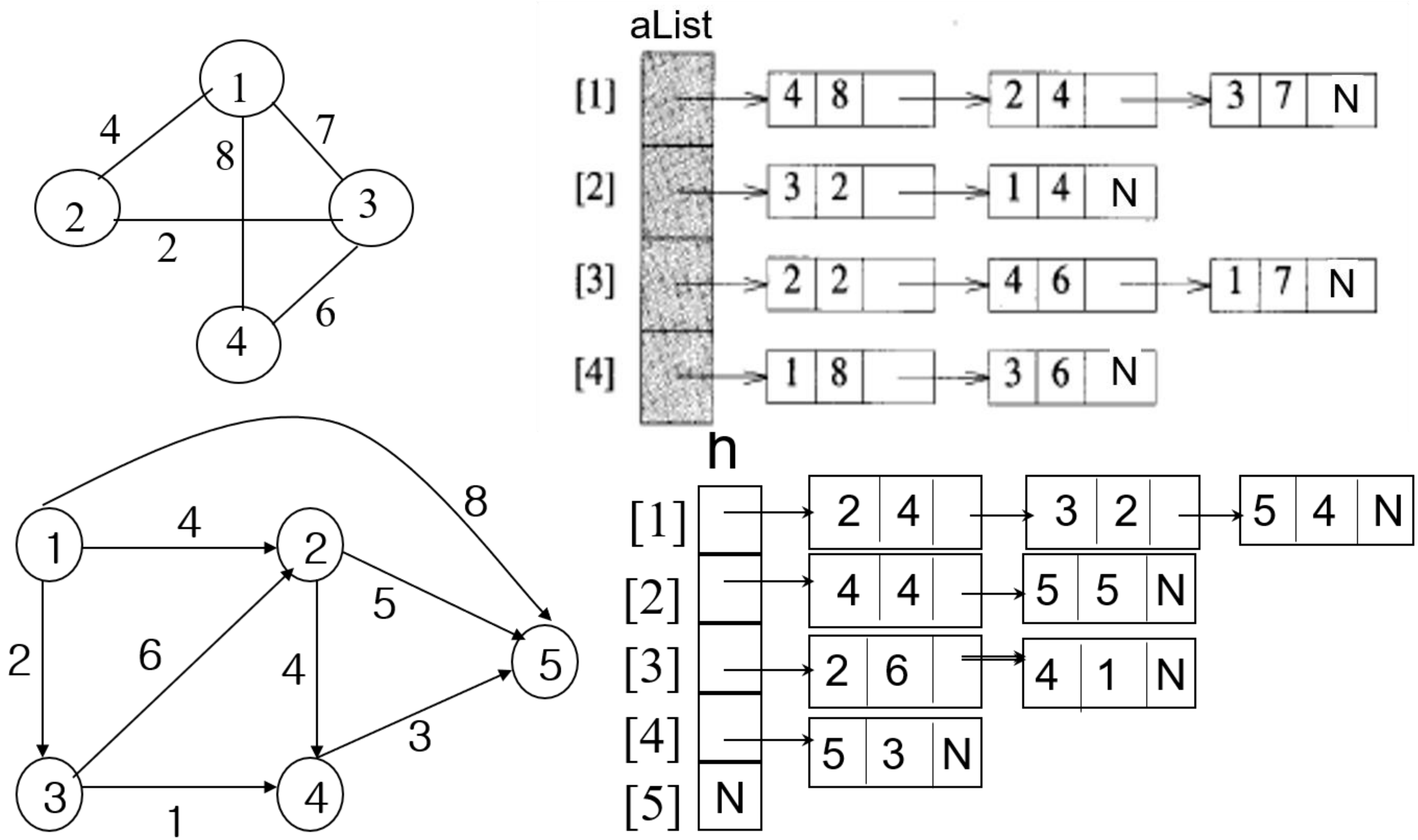




$$
\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}
\begin{array}{c}
\begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\
\begin{bmatrix}
\infty & 4 & 7 & 8 \\
4 & \infty & 2 & \infty \\
7 & 2 & \infty & 6 \\
8 & \infty & 6 & \infty
\end{bmatrix}
\end{array}
$$

$$
\begin{bmatrix}
\infty & 4 & 2 & \infty & 8 \\
\infty & \infty & \infty & 4 & 5 \\
\infty & 6 & \infty & 1 & \infty \\
\infty & \infty & \infty & \infty & 3 \\
\infty & \infty & \infty & \infty & \infty
\end{bmatrix}
$$

加权图邻接链表描述


aList
[1] 4 8 → 2 4 → 3 7 N
[2] 3 2 → 1 4 N
[3] 2 2 → 4 6 → 1 7 N
[4] 1 8 → 3 6 N
h
[1] 2 4 → 3 2 → 5 4 N
[2] 4 4 → 5 5 N
[3] 2 6 → 4 1 N
[4] 5 3 N
[5] N

16.7 类实现

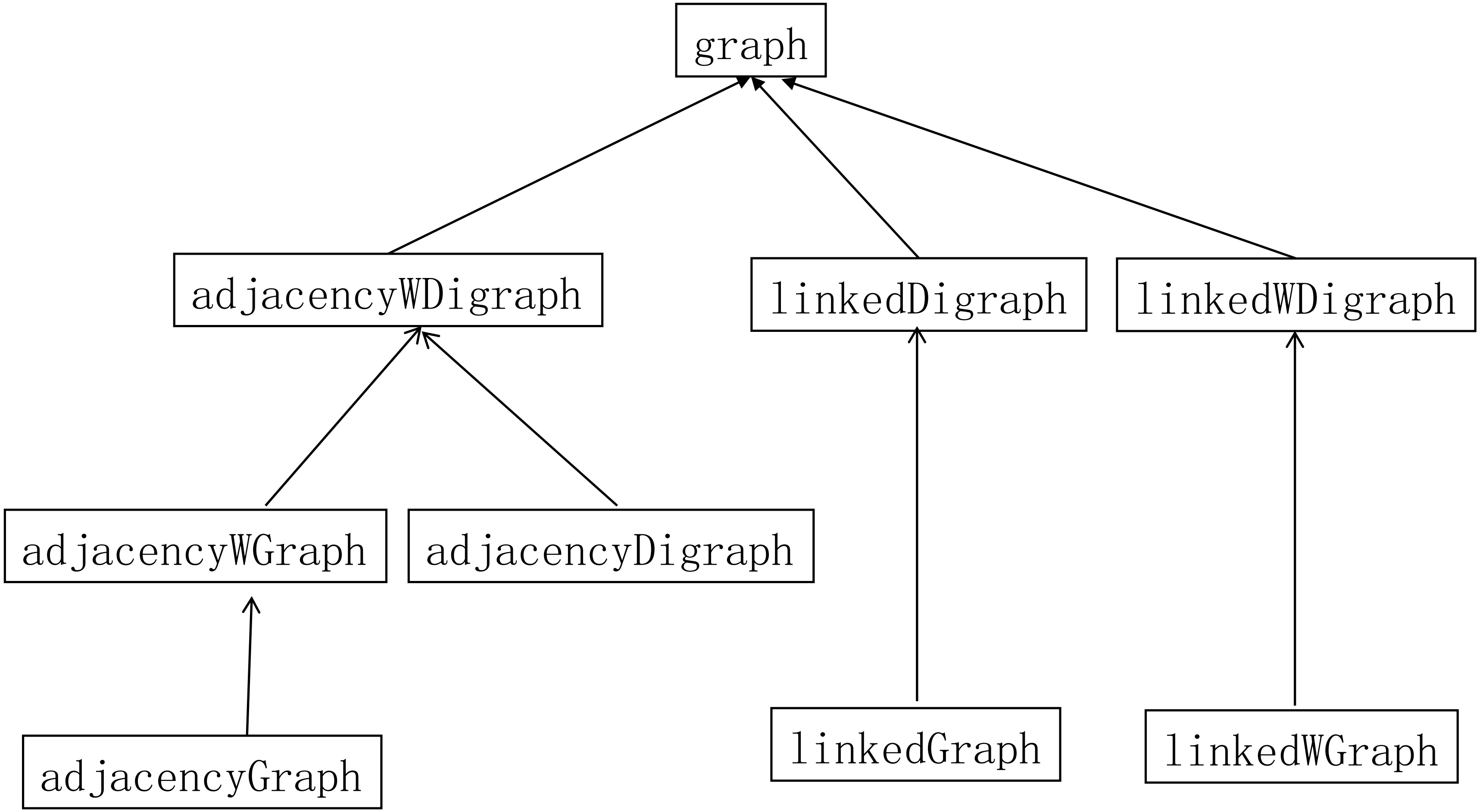
- 图的描述：
  - 邻接矩阵Adjacency Matrix
  - 邻接链表Linked Adjacency Lists
  - 邻接数组
  - 3 种描述
- 图的类型
  - 有向图、无向图。
  - 加权有向图、加权无向图。
  - 4种类型
- 3 x 4 = 12 个类

- 邻接矩阵(Adjacency Matrix)
  - 邻接矩阵描述的无向图(adjacencyGraph)
  - 邻接矩阵描述的加权无向图(adjacencyWGraph)
  - 邻接矩阵描述的有向图(adjacencyDigraph)
  - 邻接矩阵描述的加权有向图(adjacencyWDigraph)
- 邻接链表(Linked Adjacency Lists)
  - 邻接链表描述的无向图(linkedGraph)
  - 邻接链表描述的加权无向图(linkedWGraph)
  - 邻接链表描述的有向图(linkedDigraph)
  - 邻接链表描述的加权有向图(linkedWDigraph)

- **无权**有向图和无向图可以看作每条边的权是1的加权有向图和无向图。

- 无向图,边(i, j)存在可以看作边(i, j) 和边(j,i)都存在的有向图。

类的派生结构


graph
adjacencyWDigraph
linkedDigraph
linkedWDigraph
adjacencyWGraph
adjacencyDigraph
adjacencyGraph
linkedGraph
linkedWGraph

16.7.2　邻接矩阵类

```
template <class T>
class adjacencyWDigraph : public graph<T>
{
   protected:
      int n;            //顶点数目
      int e;            //边数
      T **a;             // 邻接数组
      T noEdge;          // 表示不存在的边
```

adjacencyWDigraph类

```
adjacencyWDigraph(int numberOfVertices=0, T theNoEdge=0)
    {// 构造函数.
     // 确认顶点数的合法性
       if (numberOfVertices < 0)  throw   ..........
       n = numberOfVertices;
       e = 0;
       noEdge = theNoEdge;
       make2dArray(a, n + 1, n + 1);
       for (int i = 1; i <= n; i++)
          // 初始化邻接矩阵
          fill(a[i], a[i] + n + 1, noEdge);
    }
~adjacencyWDigraph() {delete2dArray(a, n + 1);}
```

adjacencyWDigraph::insertEdge

```
void insertEdge(edge<T> * theEdge)
  {// 在图中插入边theEdge; 如果该边已经存在，
   //则theEdge中的权值更新该边权值
    int v1 = theEdge->vertex1();
    int v2 = theEdge->vertex2();
    if (v1 < 1 || v2 < 1 || v1 > n || v2 > n || v1 == v2)
       {……//输出出错信息，抛出异常}
    if (a[v1][v2] == noEdge)  // 新的边
          e++;
    a[v1][v2] = theEdge->weight();
   }
```

adjacencyWDigraph :: eraseEdge

```
void eraseEdge(int i, int j)
    {// 删除边(i,j).
      if (i >= 1 && j >= 1 && i <= n && j <= n && a[i][j] !=
noEdge)
      {
        a[i][j] = noEdge;
        e--;
      }
```

复杂度?

adjacencyWDigraph :: checkVertex

```
void checkVertex(int theVertex) const
{// 确认是有效顶点
    if (theVertex < 1 || theVertex > n)
    {
        ostringstream s;
        s << "no vertex " << theVertex;
        throw illegalParameterValue(s.str());
    }
}
```

复杂度？

adjacencyWDigraph :: outDegree

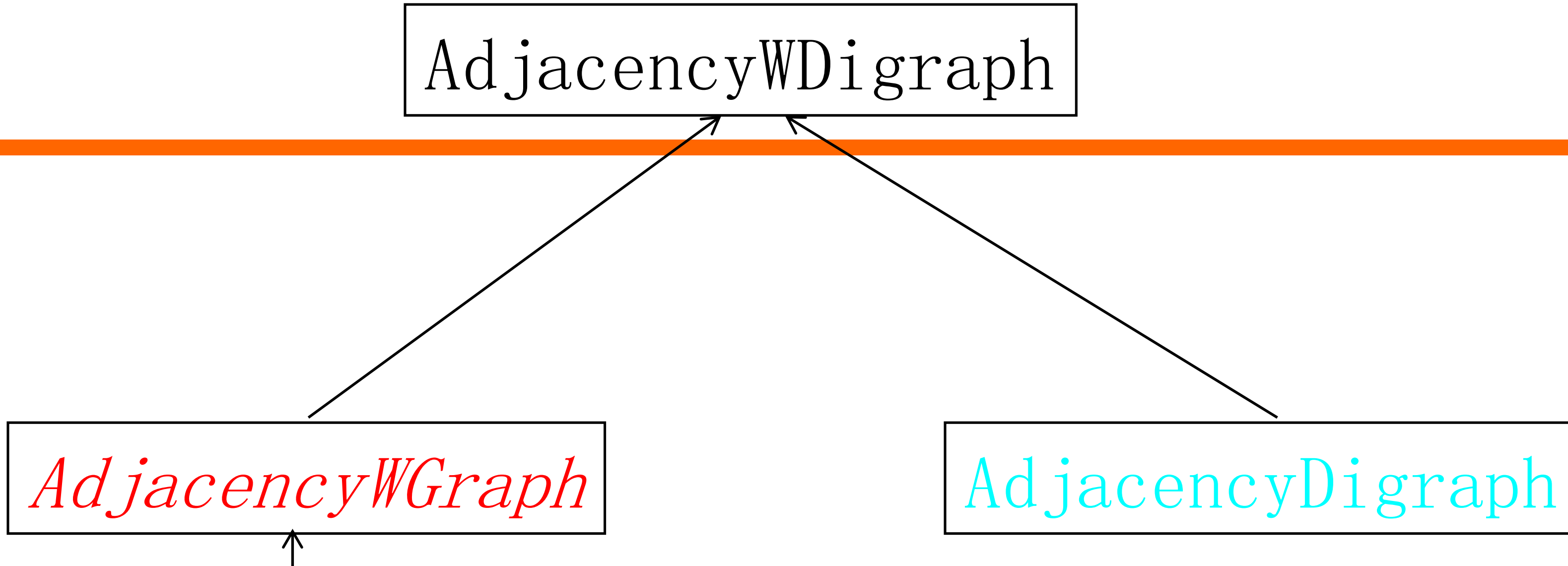
```
int outDegree(int theVertex) const
{// 返回顶点 theVertex 的出度
   checkVertex(theVertex);

   // 计数关联于顶点 theVertex 的边数
   int sum = 0;
   for (int j = 1; j <= n; j++)
      if (a[theVertex][j] != noEdge)
         sum++;

   return sum;
}
```

复杂度？

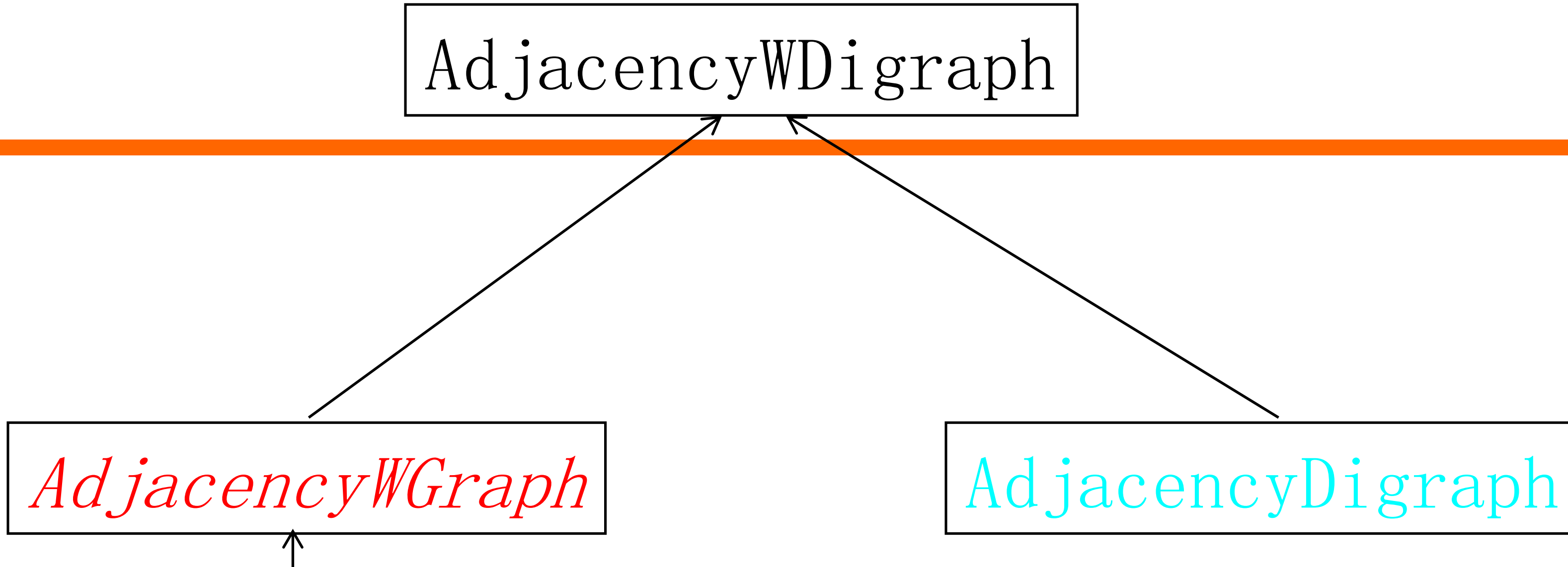
adjacencyWDigraph :: inDegree

```
int inDegree(int theVertex) const
{// 返回顶点 theVertex 的入度
   checkVertex(theVertex);

   // 计数关联至顶点 theVertex 的边数
   int sum = 0;
   for (int j = 1; j <= n; j++)
      if (a[j][theVertex] != noEdge)
         sum++;

   return sum;
}
```

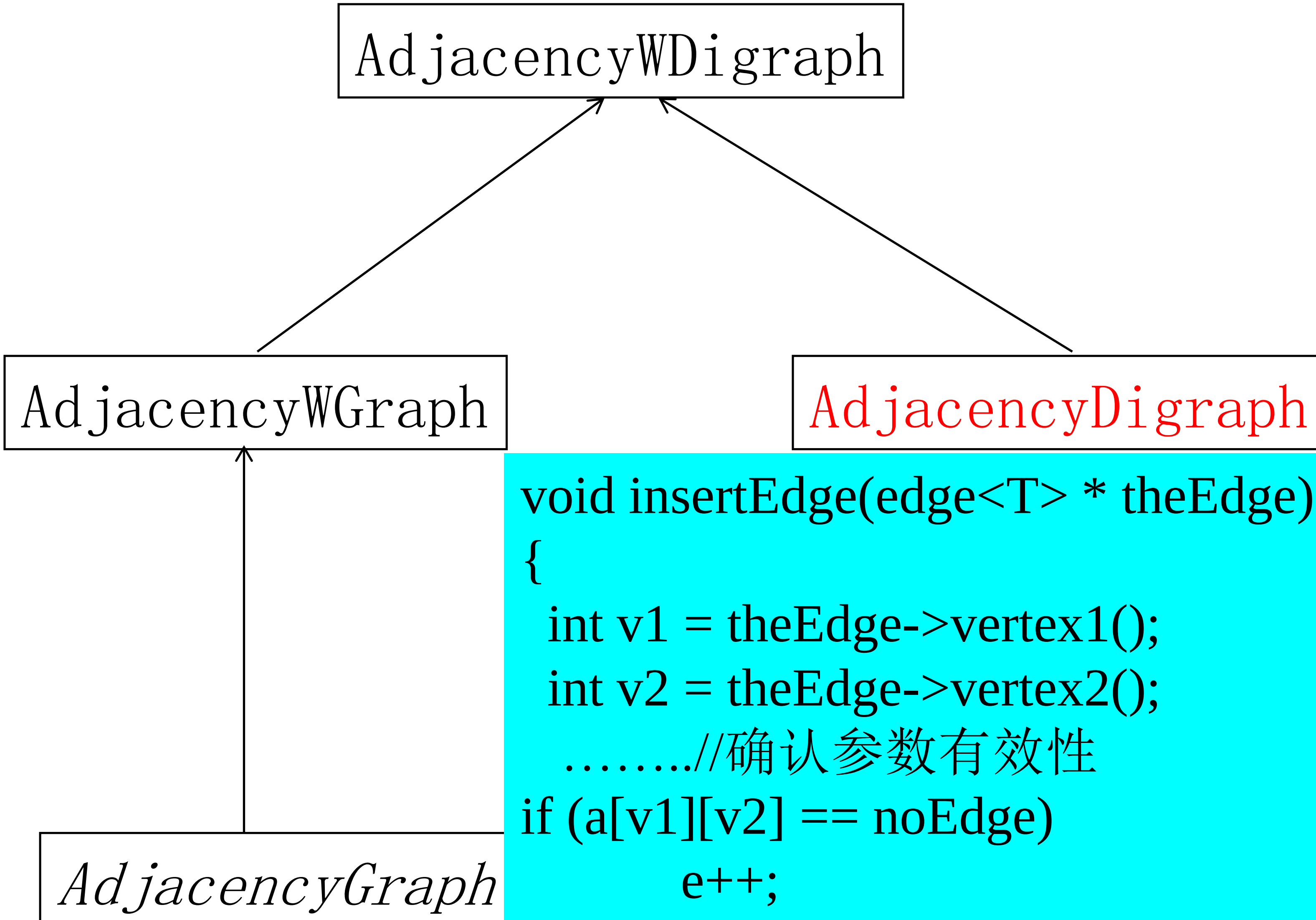
复杂度?



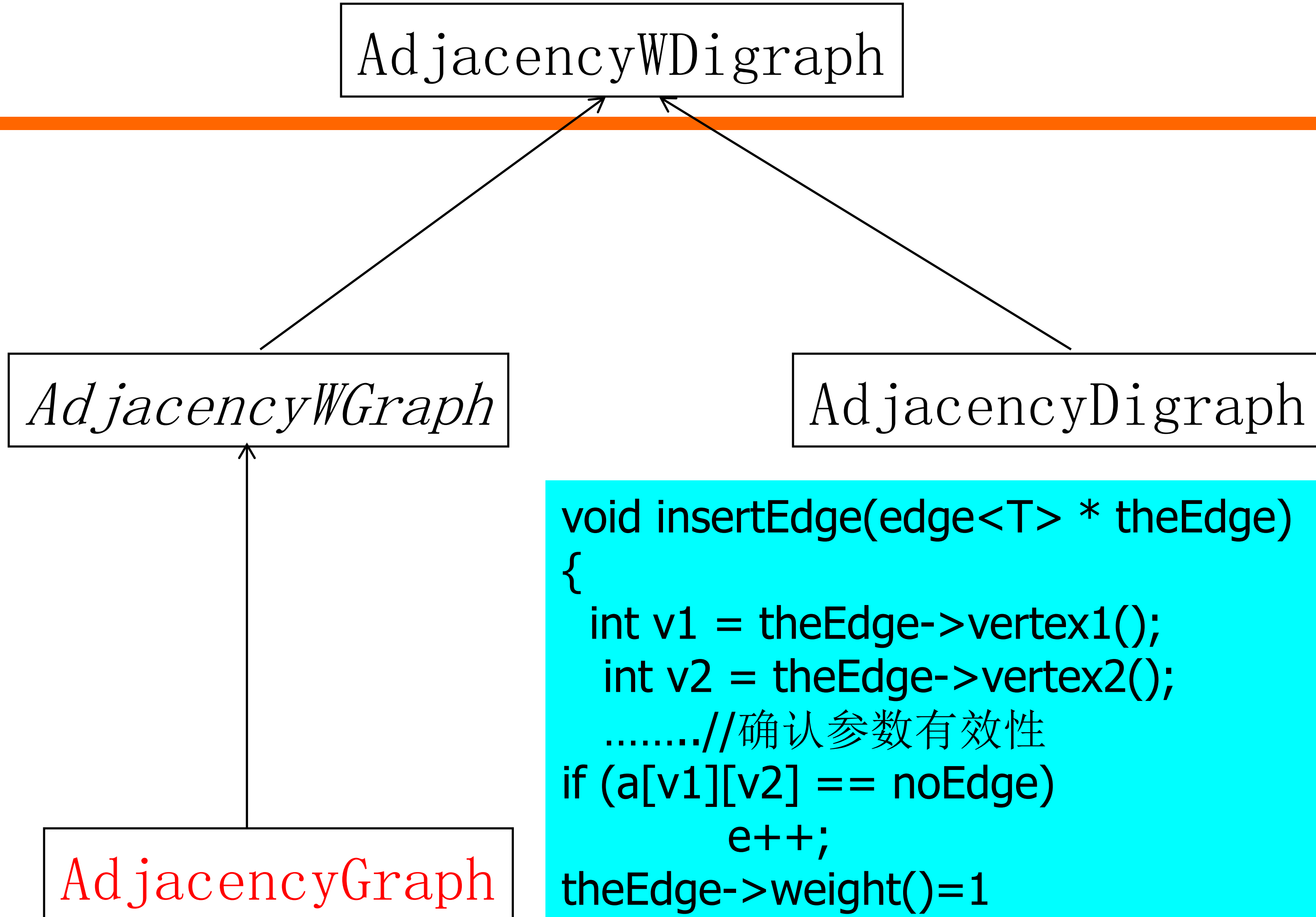

```
void insertEdge(edge<T> * theEdge)
{
  int v1 = theEdge->vertex1();
   int v2 = theEdge->vertex2();
   ……...//确认参数有效性
if (a[v1][v2] == noEdge)
          e++;
a[v1][v2] = theEdge->weight();
a[v2][v1] = theEdge->weight();
}
```




```
void eraseEdge(int i, int j)
    {// 删除边(i,j).
      if (i >= 1 && j >= 1 && i <= n && j <= n && a[i][j] != noEdge)
      {
        a[i][j] = noEdge;
        a[j][i] = noEdge;
        e--;
      }
```




```
void insertEdge(edge<T> * theEdge)
{
  int v1 = theEdge->vertex1();
  int v2 = theEdge->vertex2();
   ………//确认参数有效性
if (a[v1][v2] == noEdge)
         e++;
theEdge->weight()=1
a[v1][v2] = theEdge->weight();
```




```
void insertEdge(edge<T> * theEdge)
{
  int v1 = theEdge->vertex1();
   int v2 = theEdge->vertex2();
   ……..//确认参数有效性
if (a[v1][v2] == noEdge)
            e++;
theEdge->weight()=1
a[v1][v2] = theEdge->weight();
a[v2][v1] = theEdge->weight();
}
```

邻接矩阵的遍历函数

```
class myIterator : public vertexIterator<T>
{
   public:
      myIterator(T* theRow, T theNoEdge, int numberOfVertices)
      {
         row = theRow;
         noEdge = theNoEdge;
         n = numberOfVertices;
         currentVertex = 1;
      }

      ~myIterator() {}
```

邻接矩阵的遍历函数

```
int next(T& theWeight)
{// 返回下一个顶点。若不存在，则返回 0
 // 赋权值 theWeight = 边的权值
 // 寻找下一个邻接的顶点
   for (int j = currentVertex; j <= n; j++)
      if (row[j] != noEdge)
      {
         currentVertex = j + 1;
         theWeight = row[j];
         return j;
      }

         // 不存在下一个邻接的顶点
         currentVertex = n + 1;
         return 0;
    }
```

$$\begin{array}{cc} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} \infty & 4 & 7 & 8 \\ 4 & \infty & 2 & \infty \\ 7 & 2 & \infty & 6 \\ 8 & \infty & 6 & \infty \end{bmatrix} \end{array}$$

邻接矩阵的遍历函数

```
int next([redacted])
{// 返回下一个顶点。若不存在，则返回 0
 [redacted]
 // 寻找下一个邻接的顶点
   for (int j = currentVertex; j <= n; j++)
      if (row[j] != noEdge)
      {
         currentVertex = j + 1;
         [redacted]
         return j;
      }

      // 不存在下一个邻接的顶点
      currentVertex = n + 1;
      return 0;
 }
```

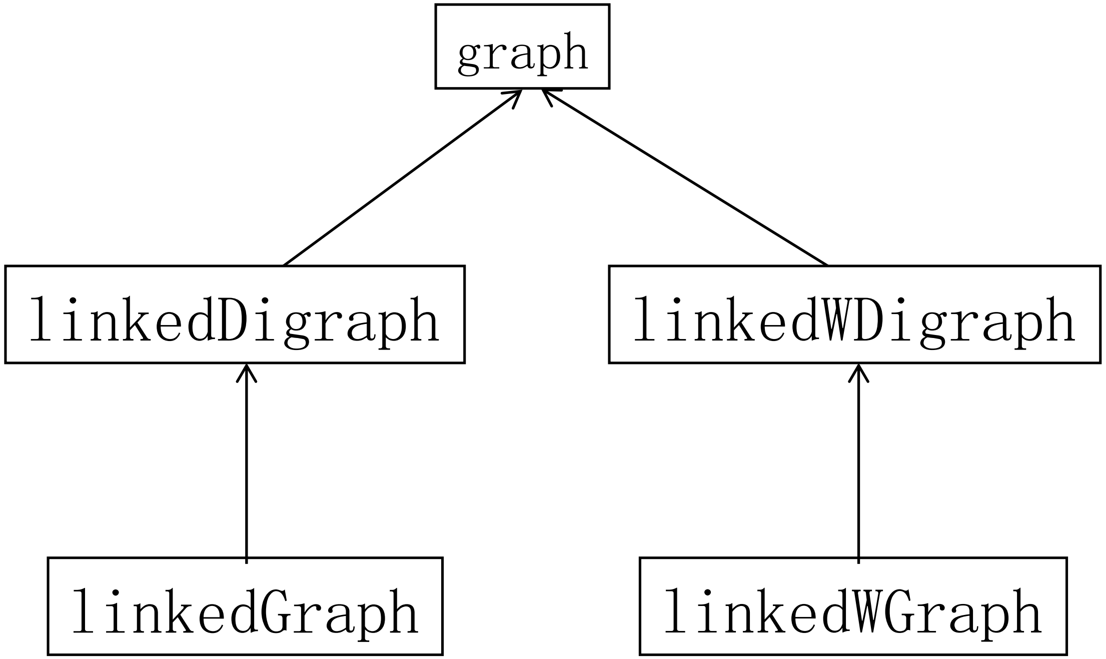
$$
\begin{array}{c c}
 & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \left[\begin{array}{cccc} \infty & 4 & 7 & 8 \\ 4 & \infty & 2 & \infty \\ 7 & 2 & \infty & 6 \\ 8 & \infty & 6 & \infty \end{array}\right]
\end{array}
$$

邻接矩阵的遍历函数

```
    protected:
        T* row;              //邻接矩阵的行
        T noEdge;            //theRow[i] == noEdge，当且仅当没有关联于顶点 i 的边
        int n;               //顶点数
        int currentVertex;
    };

    myIterator* iterator(int theVertex)
    {// 返回顶点 theVertex 的迭代器
        checkVertex(theVertex);
        return new myIterator(a[theVertex], noEdge, n);
    }
};
```

邻接链表类

- 邻接链表描述的无向图(linkedGraph)
- 邻接链表描述的加权无向图(linkedWGraph)
- 邻接链表描述的有向图(linkedDigraph)
- 邻接链表描述的加权有向图(linkedWDigraph)



扩展链表*graphChain*

class graphChain

{

graphChain 是类*Chain* 的扩展

bool empty() const {return listSize = = 0;}

int size() const {return listSize;}

T& get(int theIndex) const;

int indexOf(const T& theElement) const;

void erase(int theIndex);

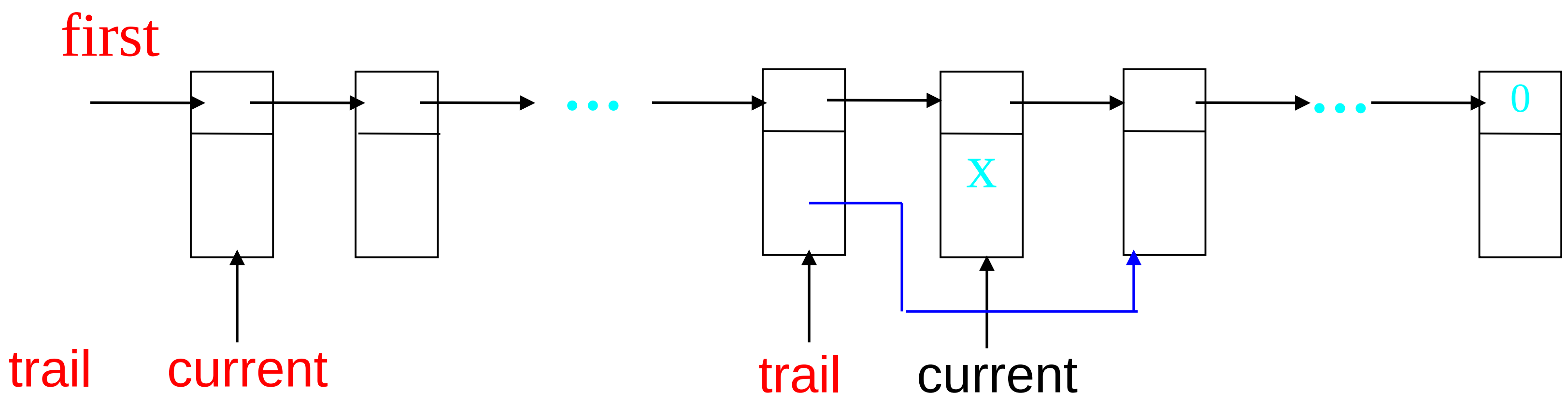
eraseElement(theVertex)：删除顶点等于theVertex的元素

void insert(int theIndex, const T& theElement);

void output(ostream& out) const;

扩展链表*graphChain*

- eraseElement

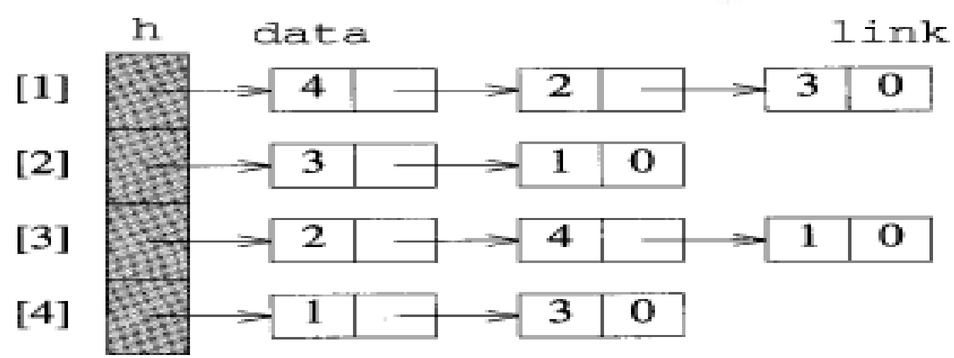


邻接链表类linkedDigraph

```
class linkedDigraph : public graph<bool>
  {
    protected:
      int n;                        // 顶点数目
      int e;                        // 边数目
      graphChain<int> *aList;    // 邻接链表
  ……..
```

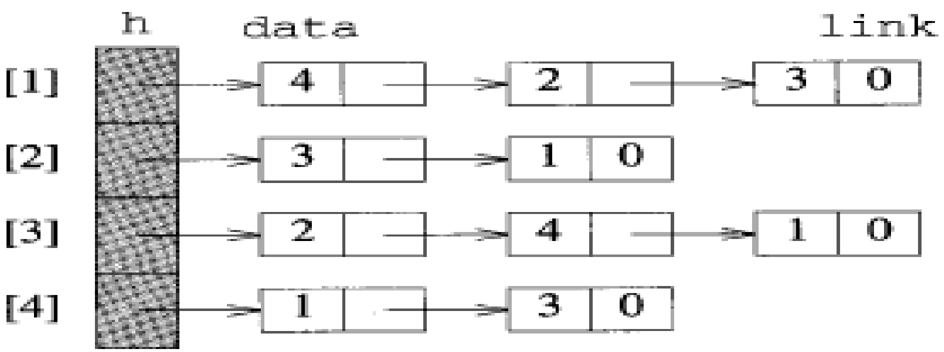
邻接链表类*linkedDigraph*

```
linkedDigraph(int numberOfVertices = 0)
{// 构造函数
    if (numberOfVertices < 0)
        throw illegalParameterValue
            ("number of vertices must be >= 0");
    n = numberOfVertices;
    e = 0;
    aList = new graphChain<int> [n + 1];
}

~linkedDigraph() {delete [] aList;}
```



邻接链表类*linkedDigraph*

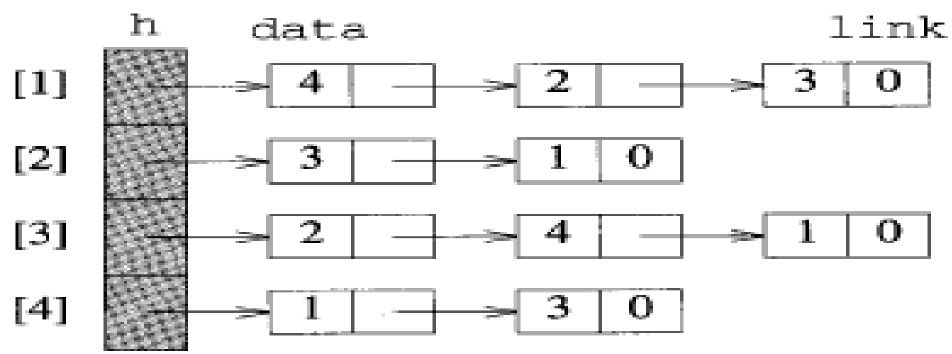
```
bool existsEdge(int i, int j) const
{// 返回 true，当且仅当 (i,j) 是一条边
   if (i < 1 || j < 1 || i > n || j > n
       || aList[i].indexOf(j) == -1)
      return false;
   else
      return true;
}
```



复杂度？

邻接链表类*linkedDigraph*

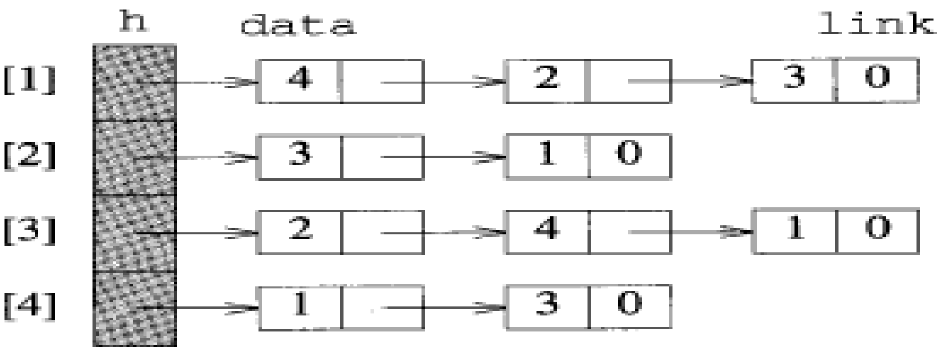
```
void insertEdge(edge<bool> *theEdge)
{// 插入一条边
   // 设置 V1 和 V2，并检验其合法性，此处代码与 adjacencyDigraphino 相同

   if (aList[v1].indexOf(v2) == -1)
   {// 新边

      aList[v1].insert(0, v2);
      e++;
   }
}
```



复杂度？

邻接链表类*linkedDigraph*

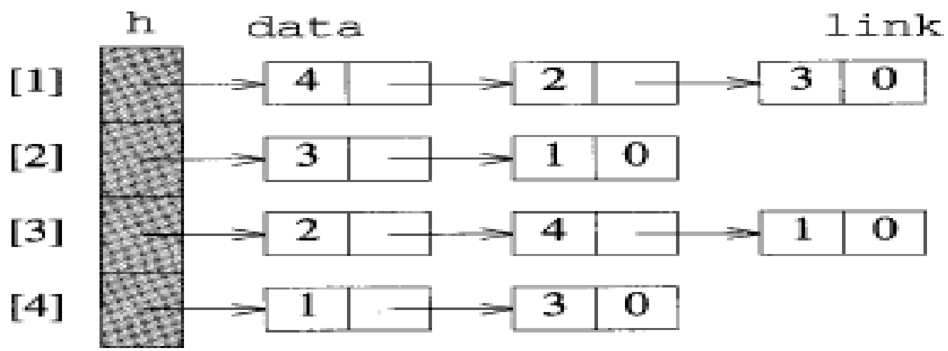
```
void eraseEdge(int i, int j)
{
    if (i >= 1 && j >= 1 && i <= n && j <= n)
    {
        int *v = aList[i].eraseElement(j);
        if (v != NULL)  //边 (i,j) 存在
            e--;
    }
}
```



复杂度?

邻接链表类*linkedDigraph*

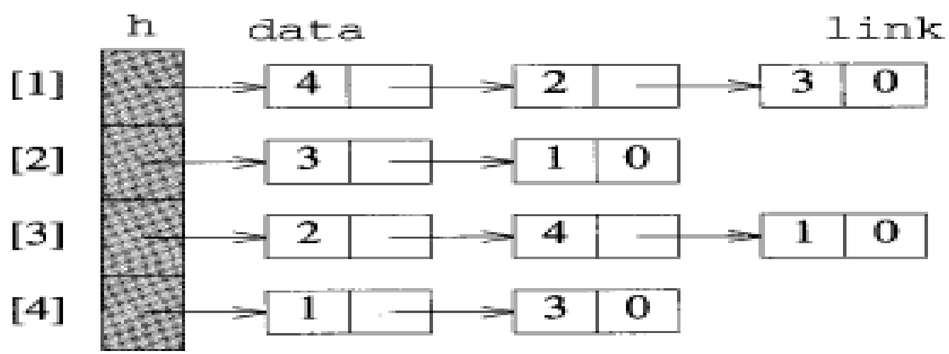
```
int outDegree(int theVertex) const
{// 返回顶点 theVertex 的出度
    checkVertex(theVertex);
    return aList[theVertex].size();
}
```



复杂度?

邻接链表类*linkedDigraph*

```
int inDegree(int theVertex) const
{
   checkVertex(theVertex);

   // 计数顶点 theVertex 的入边
   int sum = 0;
   for (int j = 1; j <= n; j++)
      if (aList[j].indexOf(theVertex) != -1)
         sum++;

   return sum;
}
```



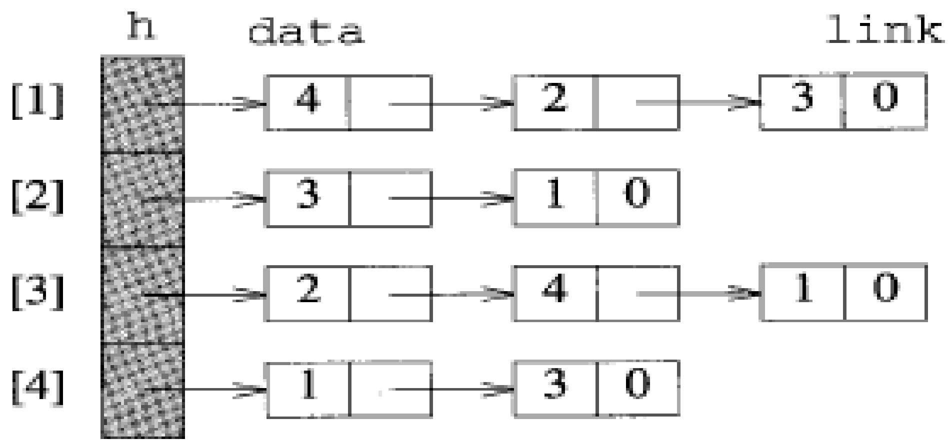
复杂度?

16.8　图的遍历

- 图的许多函数(寻找路径，寻找生成树，判断无向图是否连通等)都要求**从一个给定的顶点开始，访问能够到达的所有顶点。**
- 当且仅当存在一条从*v* 到*u* 的路径时，顶点*v* 可到达顶点*u*。
- 图的搜索：从一个已知的顶点开始，搜索所有可以到达的顶点。
- 两种搜索方法：
  - 广度优先搜索 (BFS----Breadth-First Search).（又称宽度优先搜索）
  - 深度优先搜索(DFS----Depth-First Search).

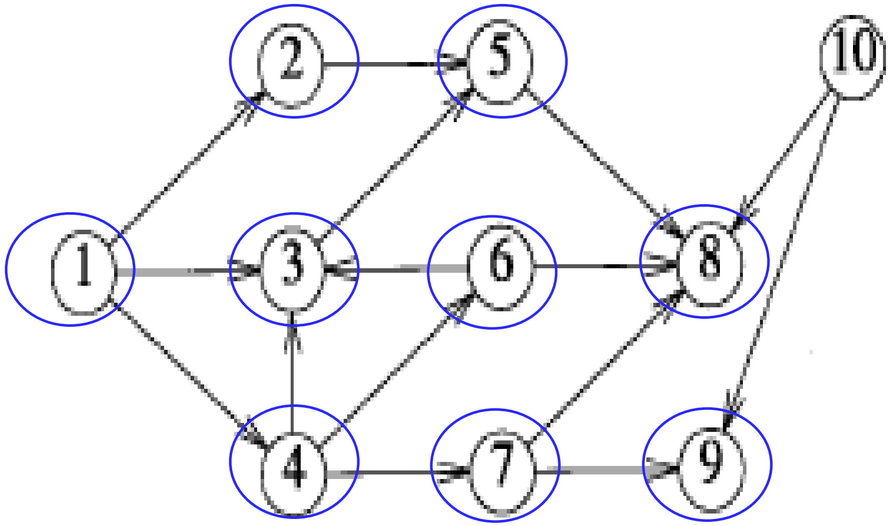
- 图的搜索：可以使用图的遍历函数。

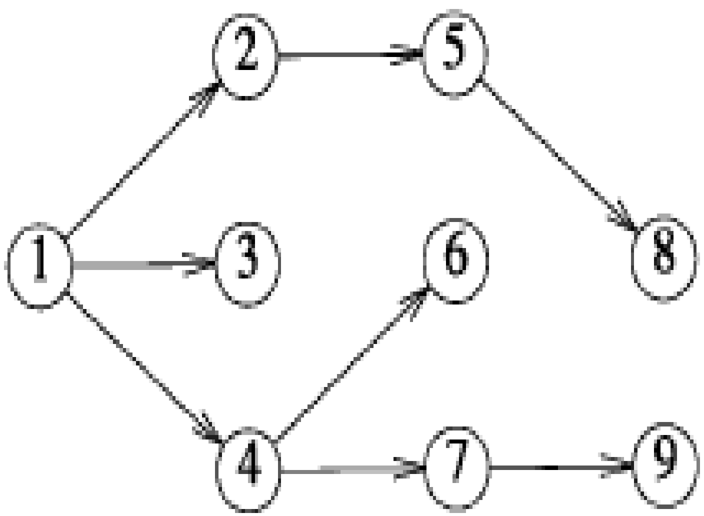
$$
\begin{array}{c|cccc}
 & 1 & 2 & 3 & 4 \\
\hline
1 & \infty & 4 & 7 & 8 \\
2 & 4 & \infty & 2 & \infty \\
3 & 7 & 2 & \infty & 6 \\
4 & 8 & \infty & 6 & \infty
\end{array}
$$



宽度优先搜索BFS

- 从一个顶点开始，识别所有可到达顶点的方法叫作宽度优先搜索(Breadth -FirstSearch, BFS)。
- 这种搜索可使用队列来实现。





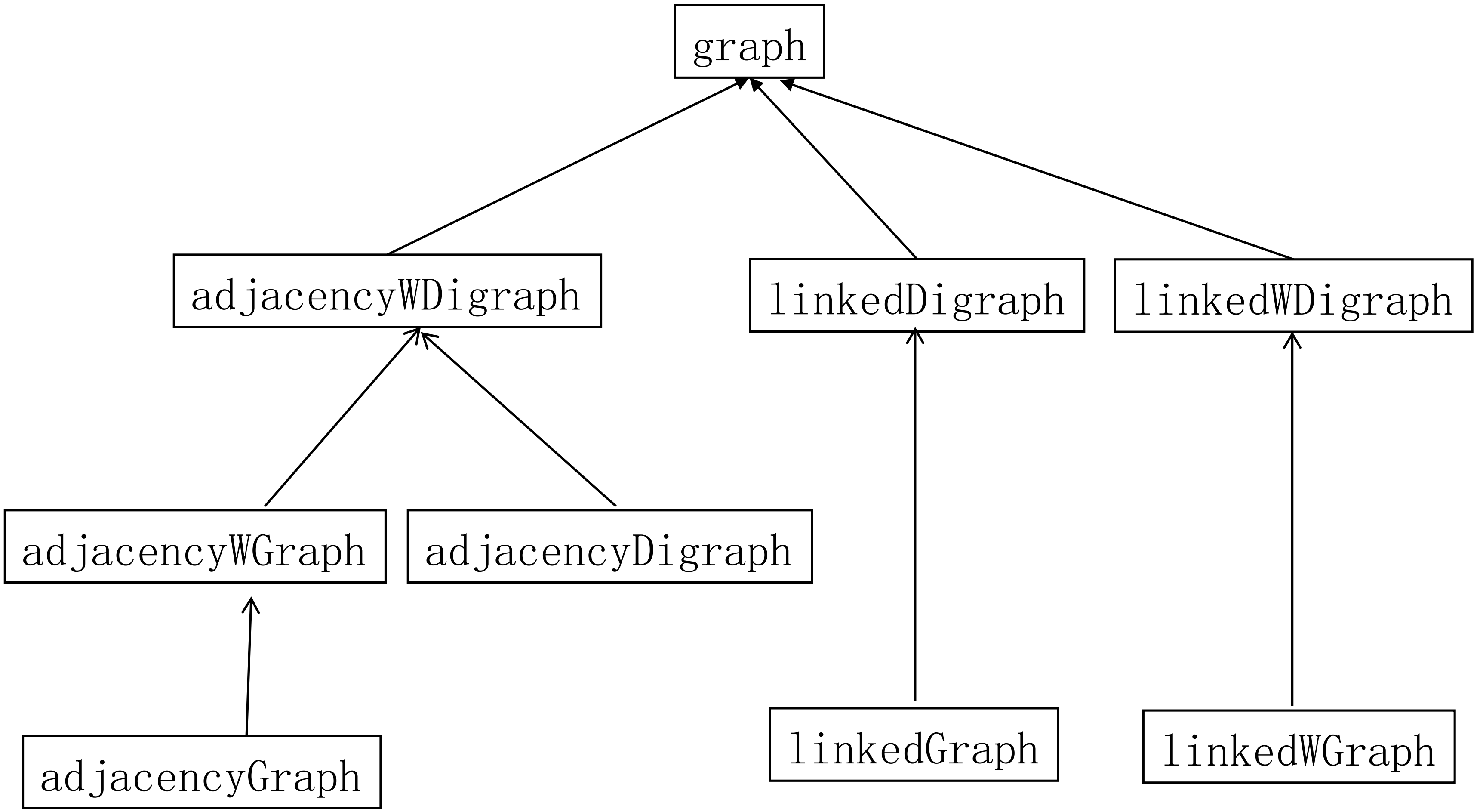
1, 2, 3, 4, 5，6，7， 8，9

宽度优先搜索伪码

```
breadthFirstSearch(v)
//从顶点v 开始的宽度优先搜索
把顶点v标记为已到达顶点;
初始化队列Q，其中仅包含一个元素v;
while (Q不空)
    {从队列中删除顶点w;
     令u 为邻接于w 的顶点;
     while (u!=NULL)
        {if ( u 尚未被标记) {
             把u 加入队列;
             把u 标记为已到达顶点; }
         u = 邻接于w 的下一个顶点;
        }
    }
```

宽度优先搜索特性

- 定理16-1
  - 设G是一个任意类型的图，*v*是G中的任意顶点。上述breadthFirstSearch(v)的伪代码能够标记从*v*出发可以到达的所有顶点(包括顶点*v*)。


graph
adjacencyWDigraph
linkedDigraph
linkedWDigraph
adjacencyWGraph
adjacencyDigraph
adjacencyGraph
linkedGraph
linkedWGraph

graph::bfs实现

```
void bfs (int v, int reach[], int label)
{   //广度优先搜索，  reach[i] = label用来标记顶点i
     arrayQueue<int> q(10);
     reach[v] = label;
     q.push(v);
     while (!q.empty())
       {int w=q.front(); // 获取一个已标记的顶点
         q.pop(); // 从队列中删除一个已标记过的顶点
         // 标记所有邻接自w的没有到达的顶点
         vertexIterator<T> *iw=iterator(w);//顶点w的迭代器
         int u;
         while ( u = iw->next()!=0)//w的下一个邻接点u
         //访问w的下一个邻接点u
         if (reach[u]==0) //u未到达过
              {q.push(u); reach[u] = label; //标记顶点u为已到达}
       delete iw;
      }
  }
```

方法graph::bfs复杂性分析

- 从顶点*v* 出发，可到达的每一个顶点都被加上标记，且每个顶点只加入到队列中一次，也只从队列中删除一次。
- 当一个顶点从队列中删除时，需要考察它的邻接点
  - 当使用邻接矩阵时，它的邻接矩阵中的行只遍历一次。 $\Theta(n)$.
  - 当使用邻接链表时，它的邻接链表只遍历一次。 Θ(顶点的出度) .
- 总时间(如果有*s* 个顶点被标记)
  - 当使用邻接矩阵时， $\Theta(sn)$
  - 当使用邻接链表时， $\Theta(\Sigma d_i^{out})$ (对于无向图／网络来说，顶点的出度就等于它的度。)

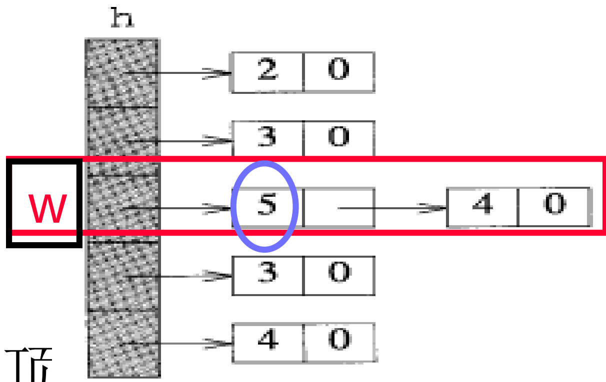
为AdjacencyWDigraph定制的BFS的实现

```
void bfs (int v, int reach[], int label)
{   //广度优先搜索， reach[i] = label用来标记顶点i
    arrayQueue<int> q(10);
    reach[v] = label;
    q.push(v);
    while (!q.empty()) {
        int w=q.front(); // 获取一个已标记的顶点
        q.pop(); // 从队列中删除一个已标记过的顶点
        //标记所有没有到达的 ,w的邻接点
        for (int u = 1; u <= n; u++)
        if (a[w][u] != noEdge && reach[u]==0) { //u未到达过
            q.push(u);
            reach[u] = label;}
    }
}
```

$$
\begin{array}{c|cccc|}
 & 1 & 2 & 3 & 4 \\
\hline
1 & \infty & 4 & 7 & 8 \\
2 & 4 & \infty & 2 & \infty \\
3 & 7 & 2 & \infty & 6 \\
4 & 8 & \infty & 6 & \infty
\end{array}
$$

为linkedDigraph定制的BFS的实现

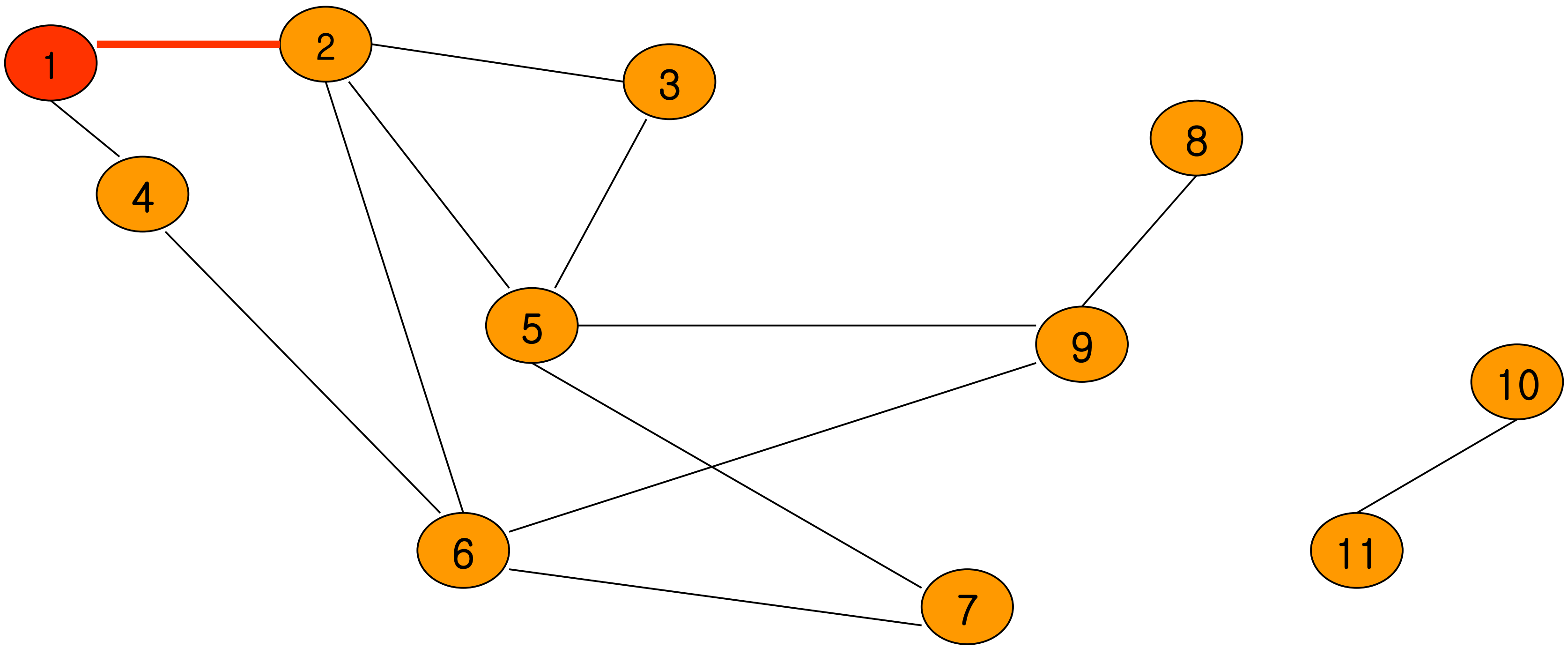
```
void bfs (int v, int reach[], int label)
{   //广度优先搜索，  reach[i] = label用来标记顶点i
     arrayQueue<int> q(10);
     reach[v] = label;
     q.push(v);
     while (!q.empty())
       {int w=q.front(); // 获取一个已标记的顶点
          q.pop(); //从队列中删除一个已标记过的顶
          //标记所有 没有到达的、邻接自w的顶点
          // 使用指针u沿着邻接表进行搜索
          for (ChainNode<int> *u = aList[w].FirstNode;
               u!=NULL; u = u->next)
               {if (reach[u->element]==0) {// 一个尚未到达的顶点
                    q.push(u->element);
                    reach[u->element] = label;}
               }
     }
}
```



深度优先搜索

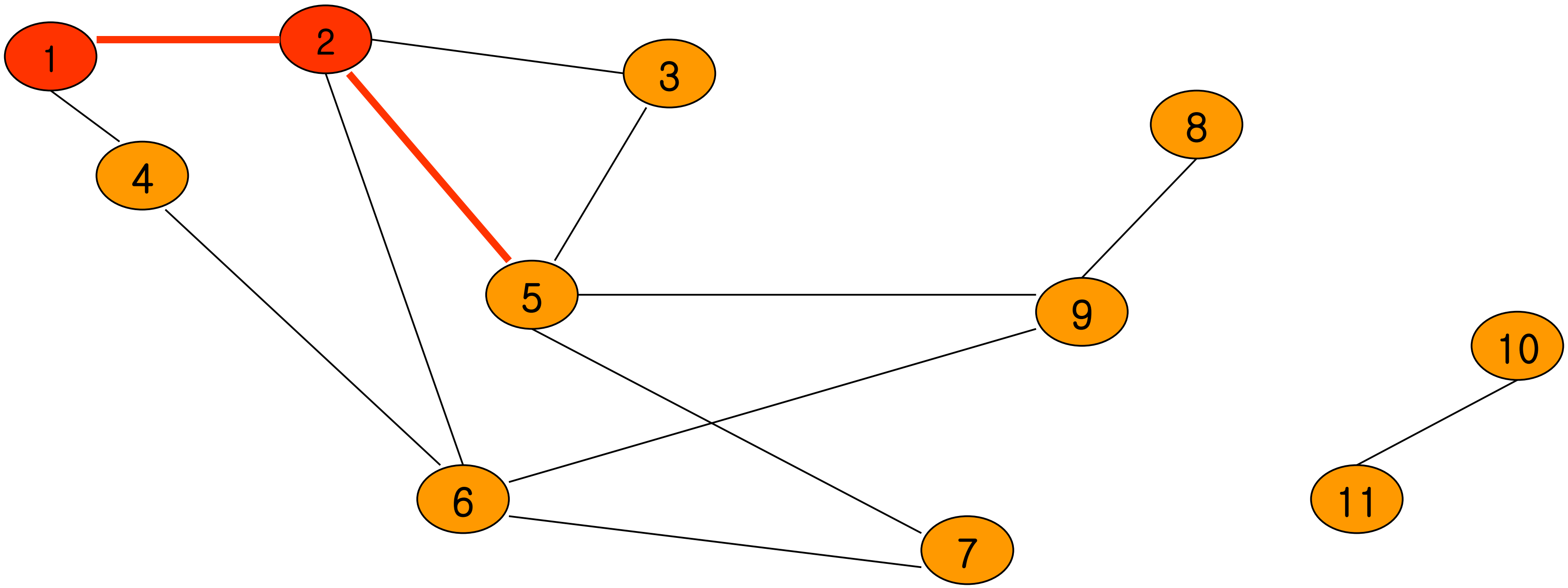
- 深度优先搜索(DFS—Depth-First Search)
- 从顶点v 出发，首先将v 标记为已到达顶点，然后选择一个与v 邻接的尚未到达的顶点u，如果这样的u 不存在，搜索中止。假设这样的u 存在，那么从u 又开始一个新的DFS。当从u 开始的搜索结束时，再选择另外一个与v 邻接的尚未到达的顶点，如果这样的顶点不存在，那么搜索终止。而如果存在这样的顶点，又从这个顶点开始DFS，如此循环下去。

深度优先搜索示例



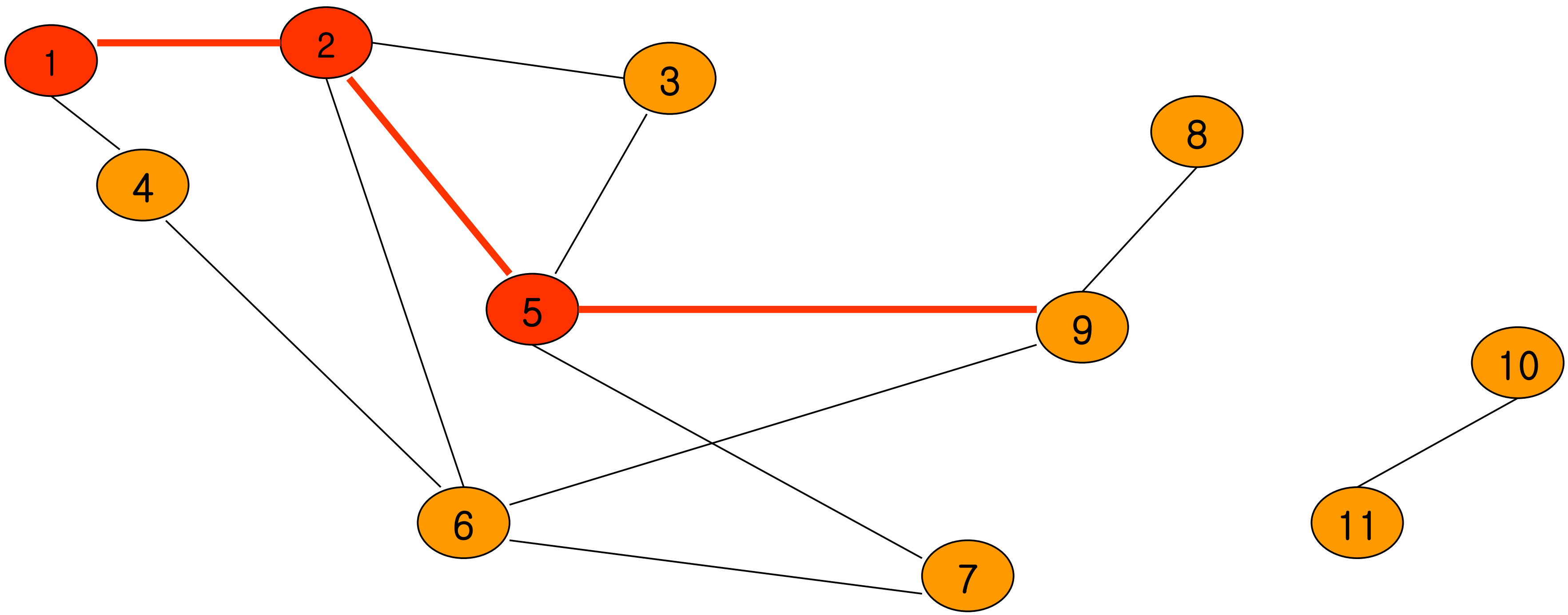
- 从顶点1开始.
- 将顶点1标记为已到达顶点，从顶点2 或4开始进行DFS

深度优先搜索示例



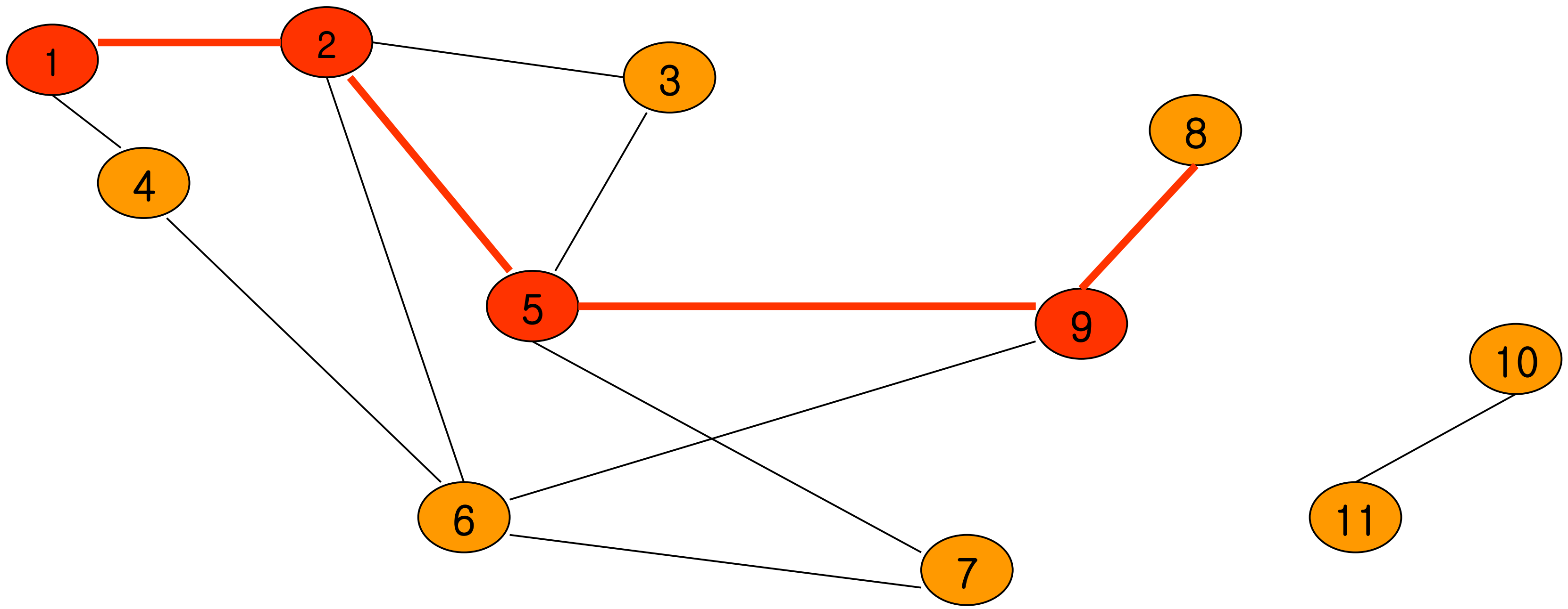
- 从顶点2开始.
- 将顶点2标记为已到达顶点，从顶点3，5 或6开始进行DFS。

深度优先搜索示例



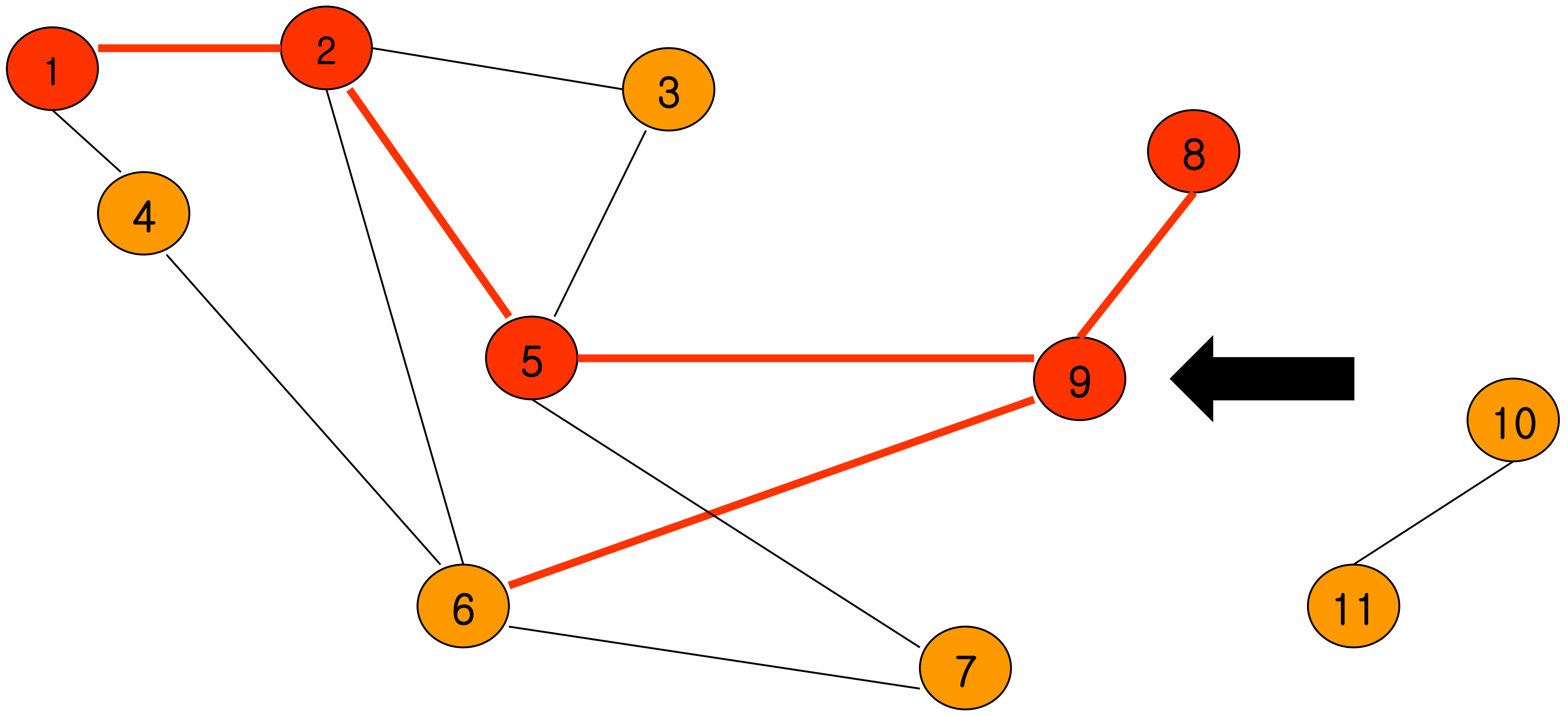
- 将顶点5标记为已到达顶点，从顶点3，7 或9开始进行DFS。

深度优先搜索示例



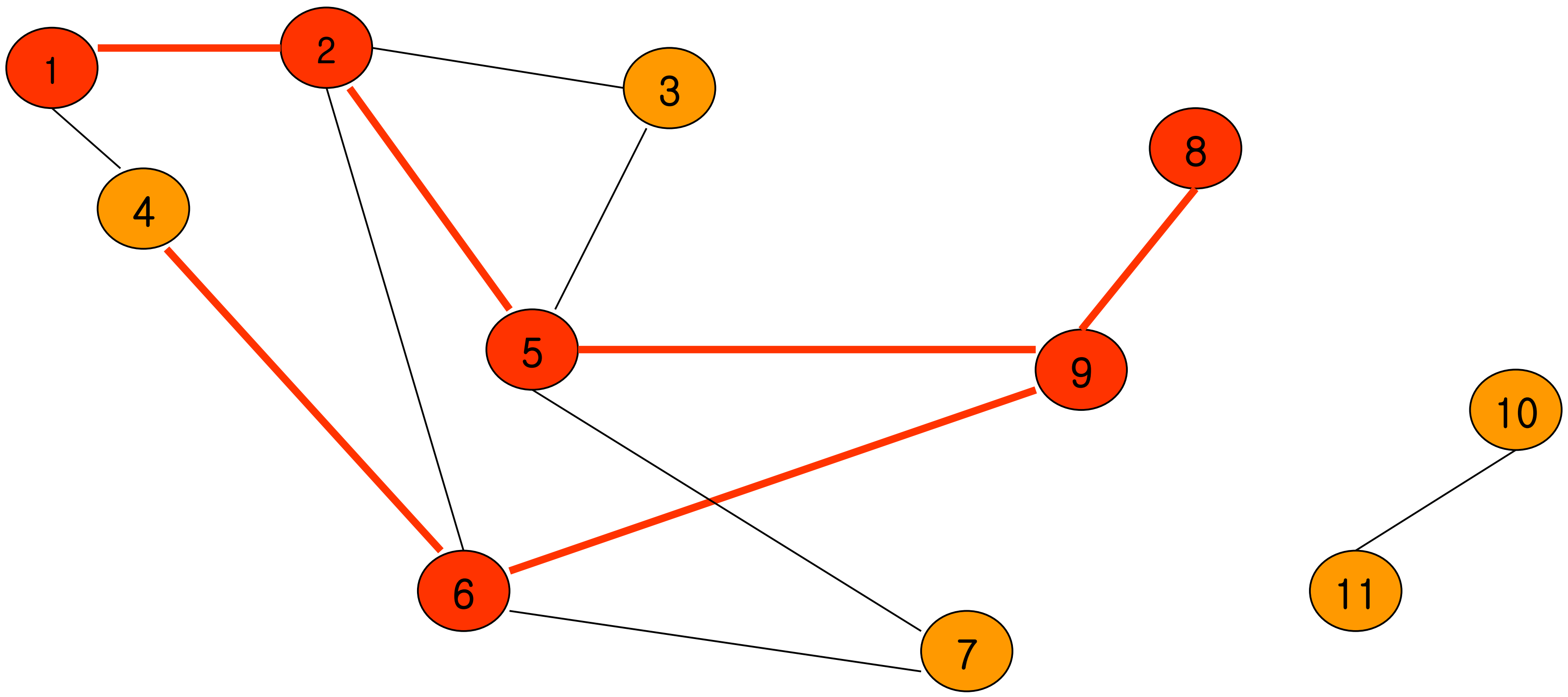
- 将顶点9标记为已到达顶点，从顶点6 或8开始进行DFS。

深度优先搜索示例



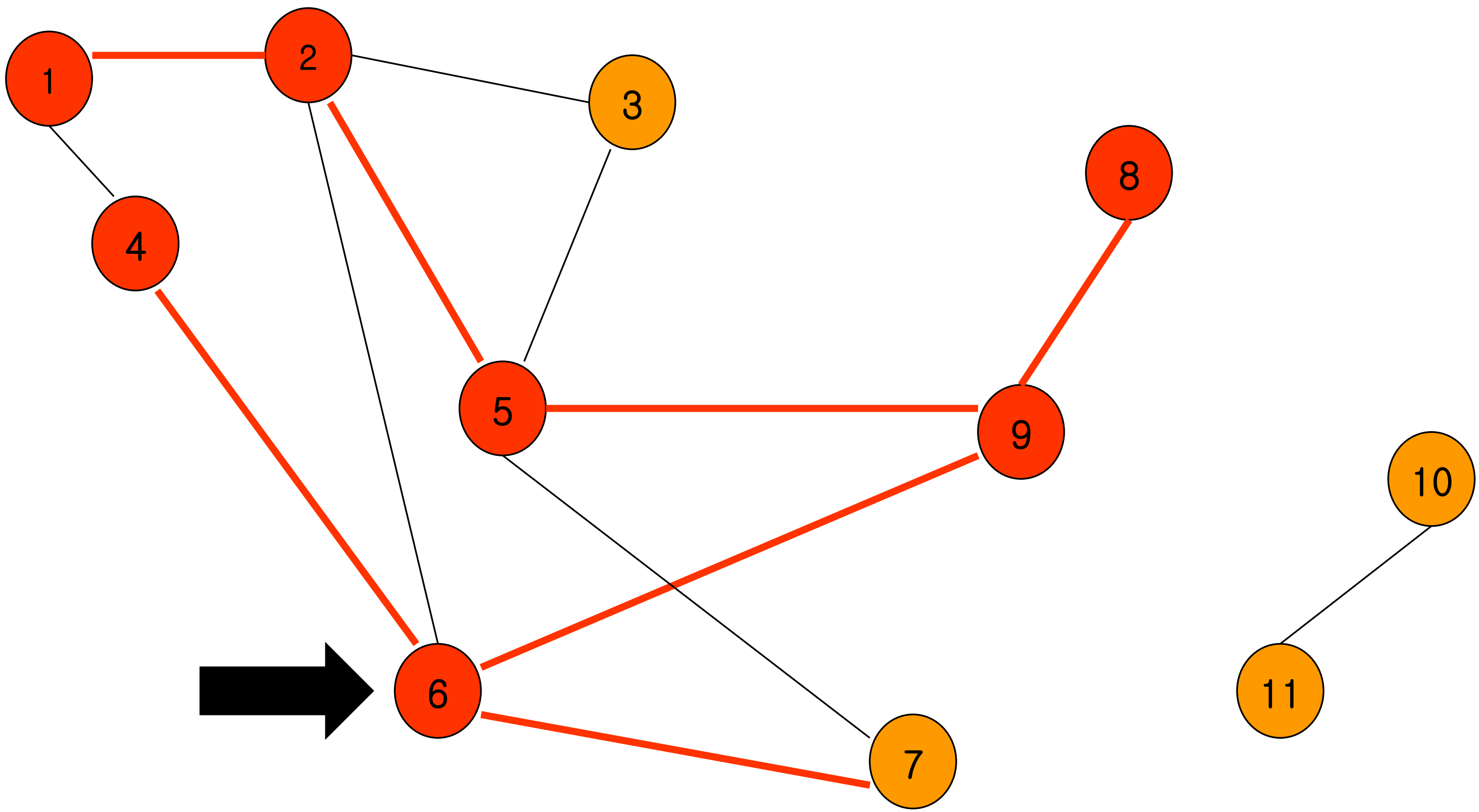
- 将顶点8标记为已到达顶点，返回到顶点9.
- 从顶点6 开始进行DFS。

深度优先搜索示例

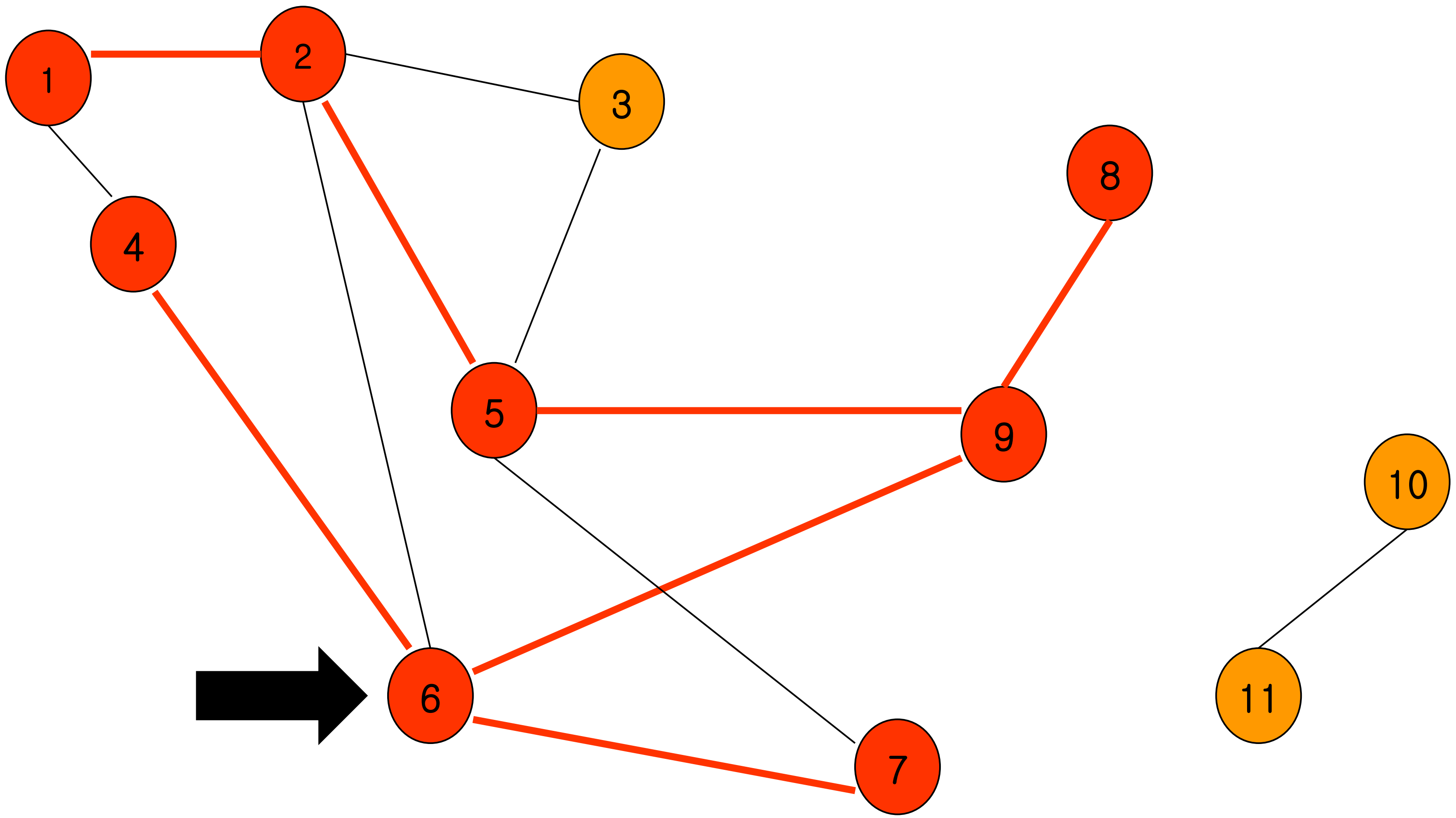


- 将顶点6标记为已到达顶点，从顶点4 或7开始进行DFS。

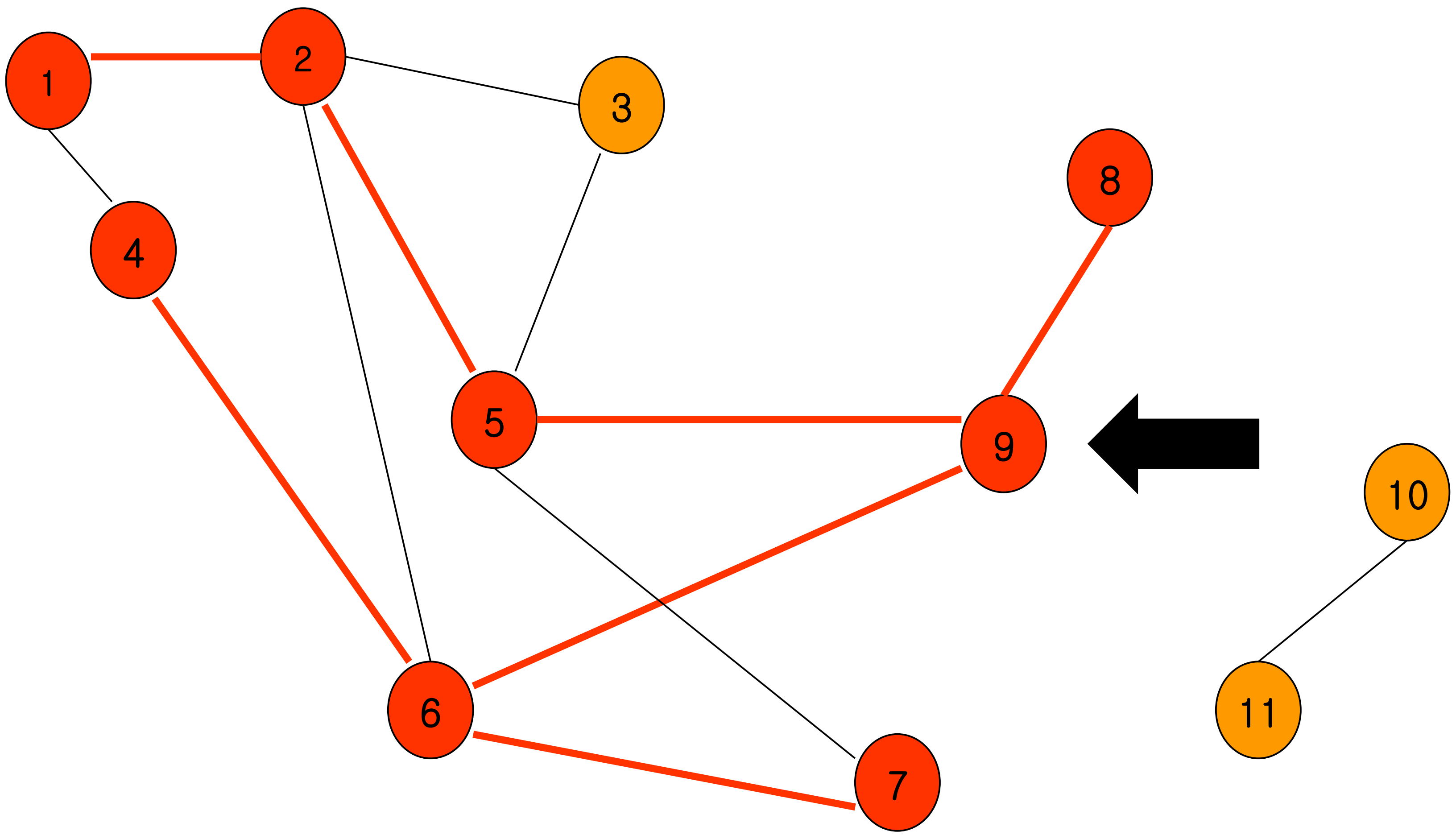
深度优先搜索示例



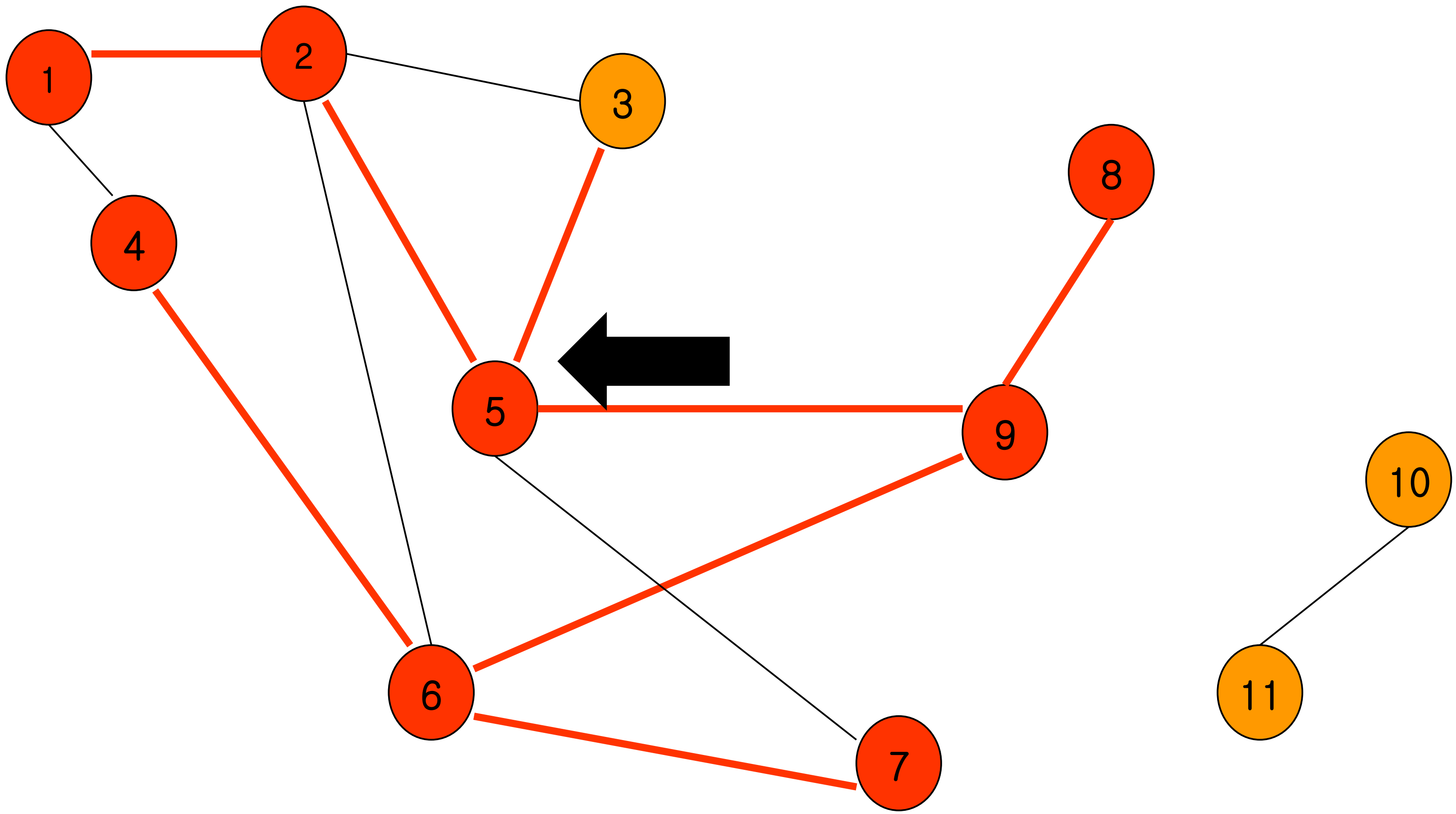
深度优先搜索示例



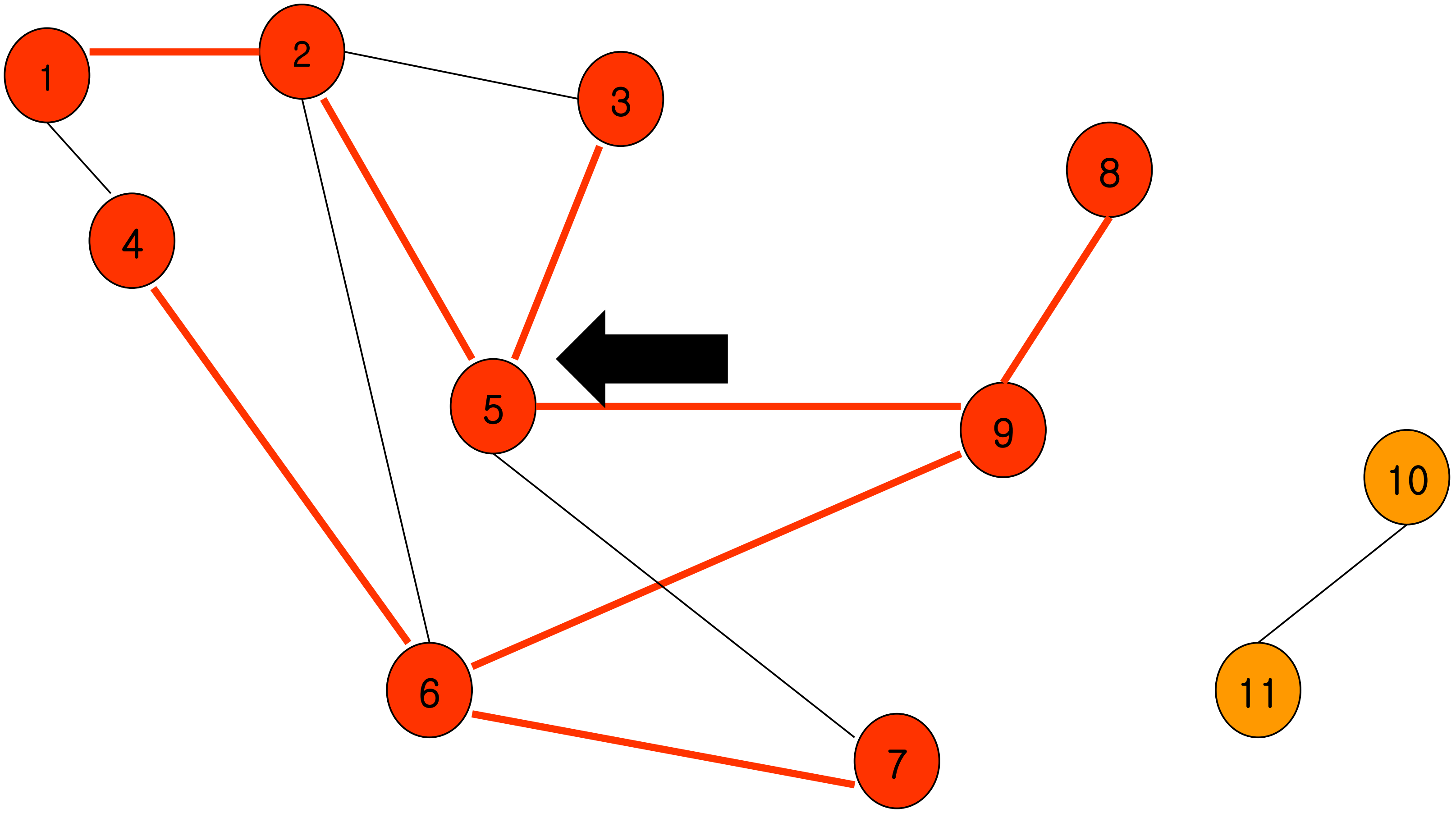
深度优先搜索示例



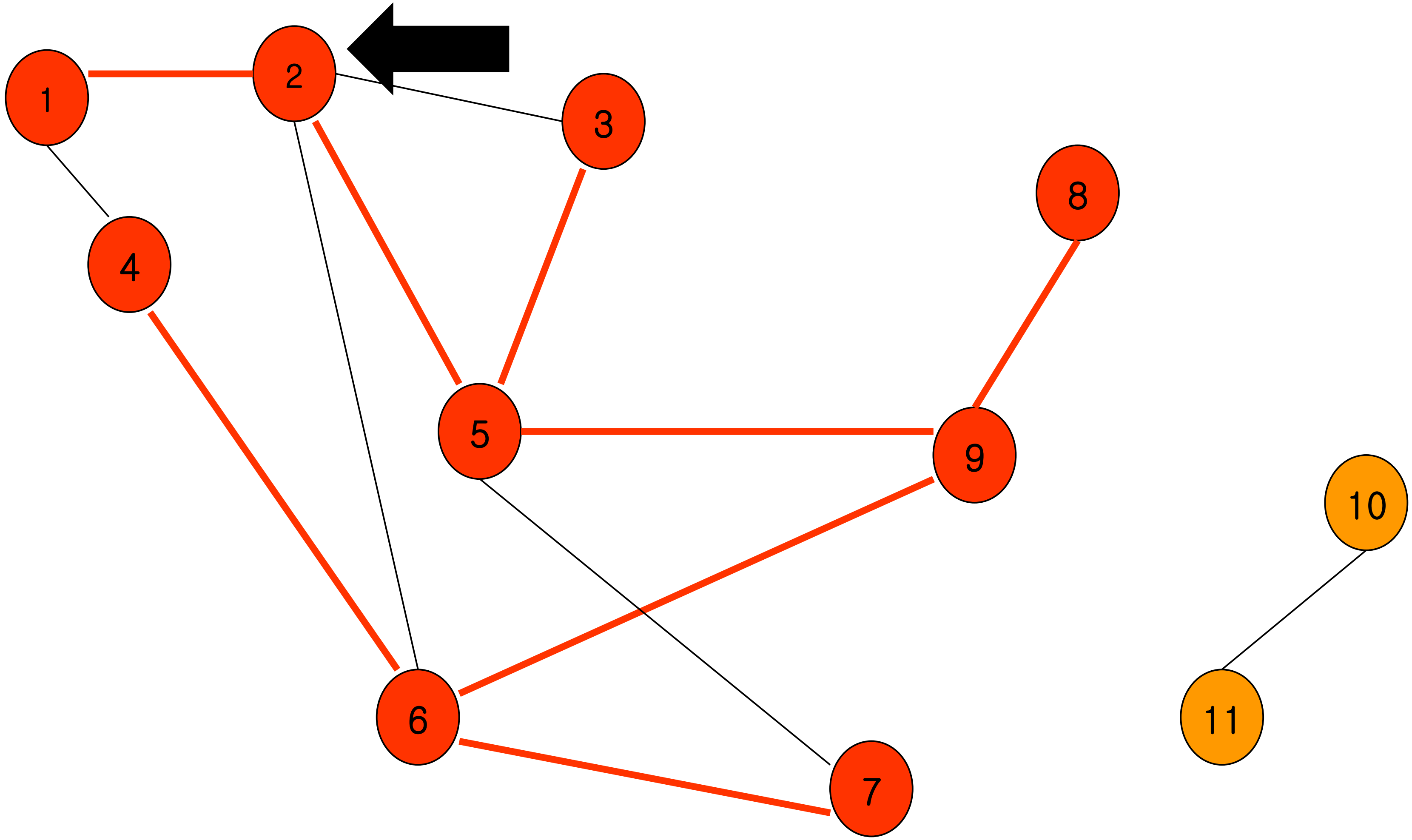
深度优先搜索示例



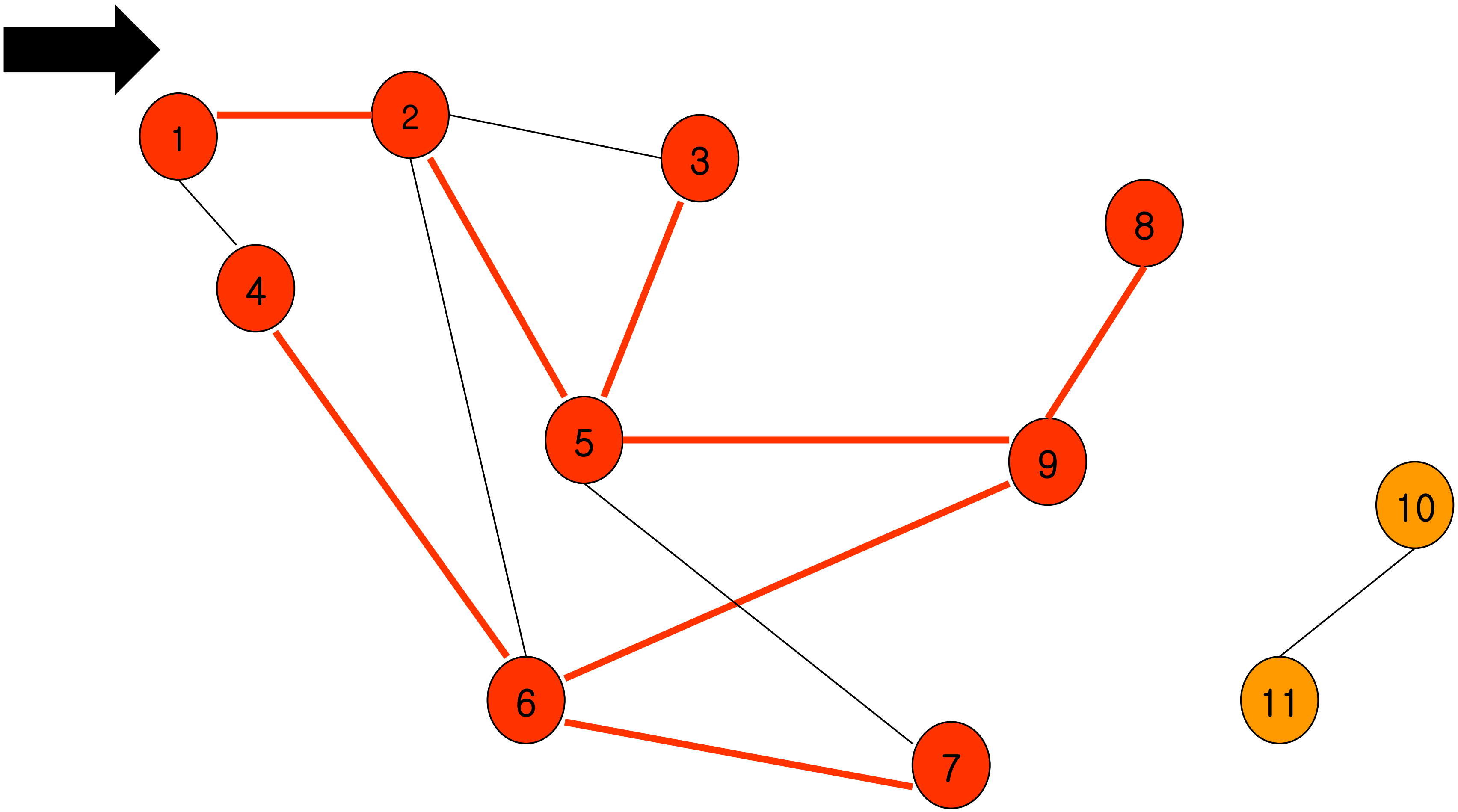
深度优先搜索示例



深度优先搜索示例



深度优先搜索示例



```
void dfs(int v, int reach[], int label)
{// dfs—— graph的public 成员方法
//深度优先搜索， reach[i] = label用来标记顶点i
     graph<T>::reach=reach;
     graph<T>::label=label;
     rDFs(v);//执行dfs
}
void  rDFs (int v)
{// dfs— 保护成员方法，深度优先搜索递归方法
     reach[v] = label;
     vertexIterator<T> *iv=iterator(v);//顶点v的迭代器
     int u;
     while ( u = iv->next()!=0)  //v的下一个邻接点u
           //访问v的下一个邻接点u
          if (reach[u]==0)   //u未到达过
                   rDFs (u);
      delete iv;
}
```

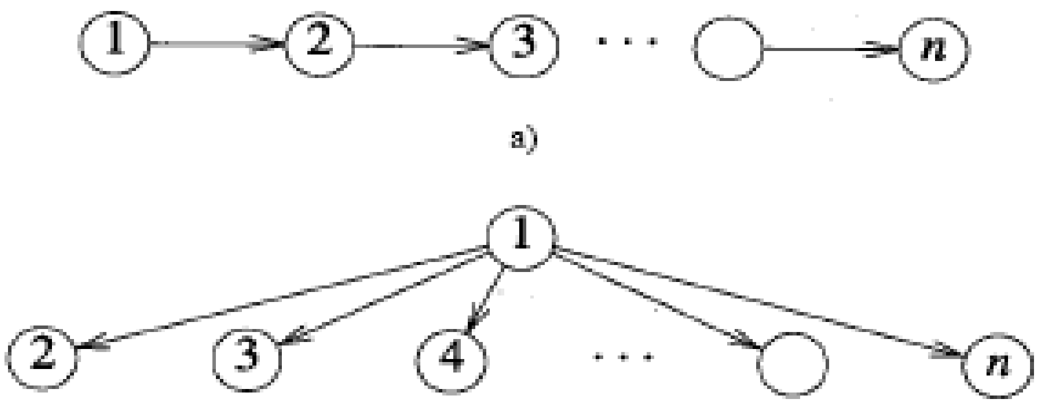
深度优先搜索的实现

深度优先搜索特性

- 定理16-2
  - 设*G*是一个任意类型的图，v是G中任意顶点。depthFirstSearch(v)可以标记所有从顶点v可到达的顶点(包括v) 。

方法graph::dfs的复杂性分析

- dfs与bfs有相同的时间和空间复杂性。



a)
b)

- (a) depthFirstSearch(1)的最坏情况(占用空间最大);
  breadthFirstSearch(1)的最好情况(占用空间最小)
- (b) depthFirstSearch(1)的最好情况
  breadthFirstSearch(1)的最坏情况

16.9 应用

- 16.9.1　寻找路径
- 16.9.2　连通图及其构件
- 16.9.3　生成树

16.9.1 寻找路径

- 找一条从顶点theSourse 到达顶点theDestination 的路径
  - 从顶点theSourse开始搜索(宽度或深度优先)且到达顶点theDestination时终止搜索

    如何得到路径?
  - 路径是一顶点序列，path[0]记录路径中的边数(路径长度) ;用数组path[1:path[0]+1]记录路径中的顶点; length记录顶点的个数;path[1]= theSourse；path[length]= theDestination

Path[0]=v

Path[length]=w

graph::rFindPath实现

```
bool rFindPath(int s)
{//寻找顶点s到达顶点destination的路径，s≠destination
  //找到一条路径，返回true; 否则，返回false
      reach[s] = 1; //将s标记为已到达顶点
      vertexIterator<T>* is = iterator(s);
      int u;
      while ((u = is->next()) != 0)
       {// 访问s的一个未到达邻接点
        if (reach[u] == 0)   // u 未到达
          {// 移到顶点u
           path[++length] = u; // 将 u 加入路径
           if (u == destination || rFindPath(u))
                 return true;
           // 从u 到 destination没有路径
          length--;    // 从路径中删除 u
          }
       }
      delete is;
      return false;
   }
```

Graph::findPath实现1/2

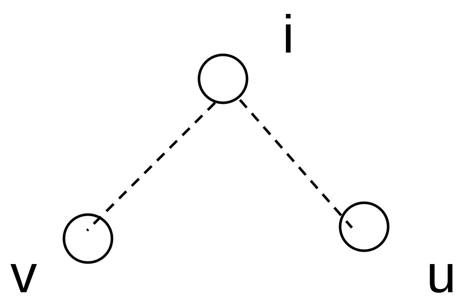
```
int* findPath(int theSource, int theDestination)
   {//寻找一条顶点theSourse到达顶点theDestination的路径，
    // 返回一个数组path， path[0]表示路径长度，
    //path[1] 开始表示路径，如果路径不存在，返回NULL.

    //为调用递归函数 rFindPath(theSourse)初始化
      int n = numberOfVertices();
      path = new int [n + 1];
      path[1] = theSource;      // 路径中的第一个顶点
      length = 1;               // 当前路径长度+ 1
      destination = theDestination;
      reach = new int [n + 1];
      for (int i = 1; i <= n; i++)
            reach[i] = 0;
```

Graph::findPath实现2/2

```
// 搜索路径
    if (theSource == theDestination || rFindPath(theSource))
      // 找到一条路径
      path[0] = length - 1;
    else
      {  //路径不存在
         delete [] path;
         path = NULL;
       }

    delete [] reach;
    return path;
  }
```

16.9.2 连通图及其构件

- 一个无向图是连通图吗?
- 从任意顶点开始执行DFS或BFS
- 一个无向图是连通图 *当且仅当* 所有顶点被标记为已到达顶点.
  - 所有n个顶点被标记为已到达顶点.
  - ⇒任意两个顶点u和v之间存在路径.



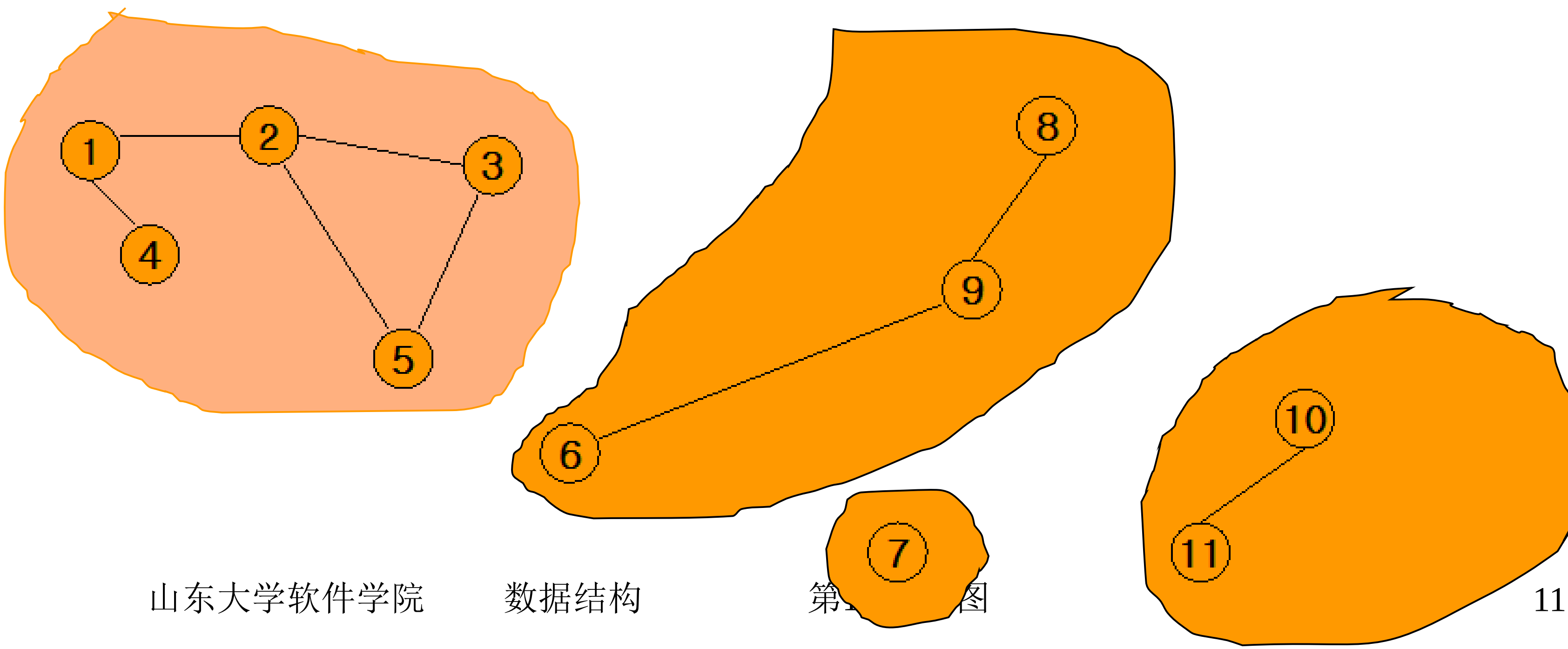
判断无向图是否是连通图的实现

```
bool connected()
{ //当且仅当图是连通的，则返回true
   if (directed()) throw  ……//如果图不是无向图， 抛出异常;
  int n =numberofVertices();//图中顶点数
  //置所有顶点为未到达顶点
  int *reach = new int [n+1];
  for (int i = 1; i <= n; i++)
        reach[i] = 0;
  //对从顶点1出发可到达的顶点进行标记
   dfs(1, reach, 1);
   //检查是否所有顶点都已经被标记
   for (int i = 1; i <= n; i++)
      if (reach[i]==0) return false;
   return true;
}
```

连通构件

从顶点i 可到达的顶点的集合C与连接C中顶点的边称为连通构件(connected component)。

构件标记问题：给无向图中的顶点做标记，两个顶点具有相同的标记，*当且仅当* 两个顶点属于同一个构件。



标记连通构件的实现

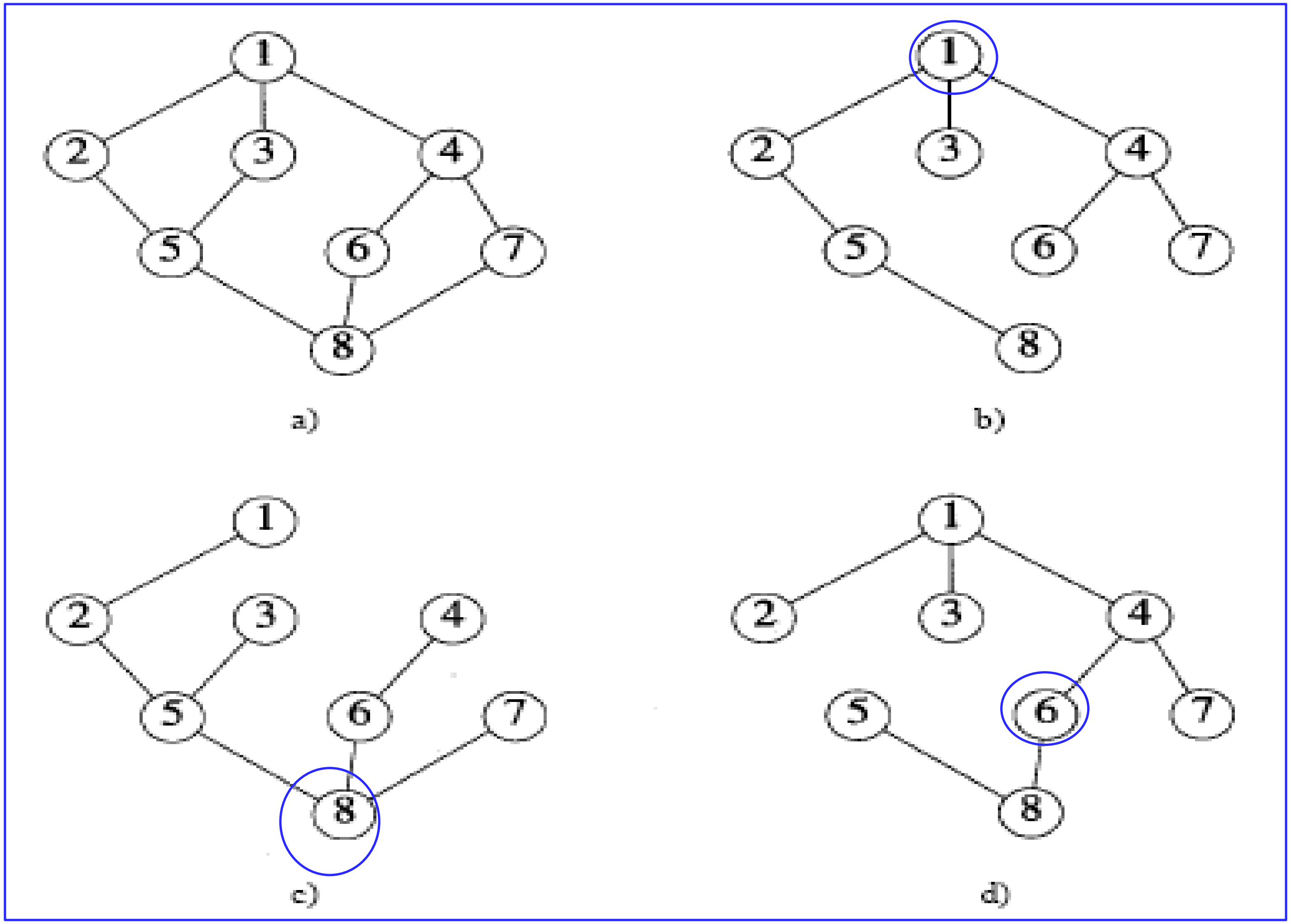
```
int labelcomponents(int c[])
{ // 构件标识, 返回构件的数目,并用c[1:n]表示构件编号
    if (directed()) throw  ……//如果图不是无向图， 抛出异常;
  int n = numberofVertices();//图中顶点数
  // 初始时， 所有顶点都不属于任何构件
  for (int i = 1; i <= n; i++)
        c[i] = 0;
  int label = 0; // 最后一个构件的编号
     // 识别构件
  for (int i = 1; i <= n; i++)
     if (c[i]==0) //顶点i未到达
      // 顶点i 属于一个新的构件
     {label++ ;
      bfs(i, c, label);} // 标记新构件
 return label;
}
```

复杂度?

16.9.3 生成树

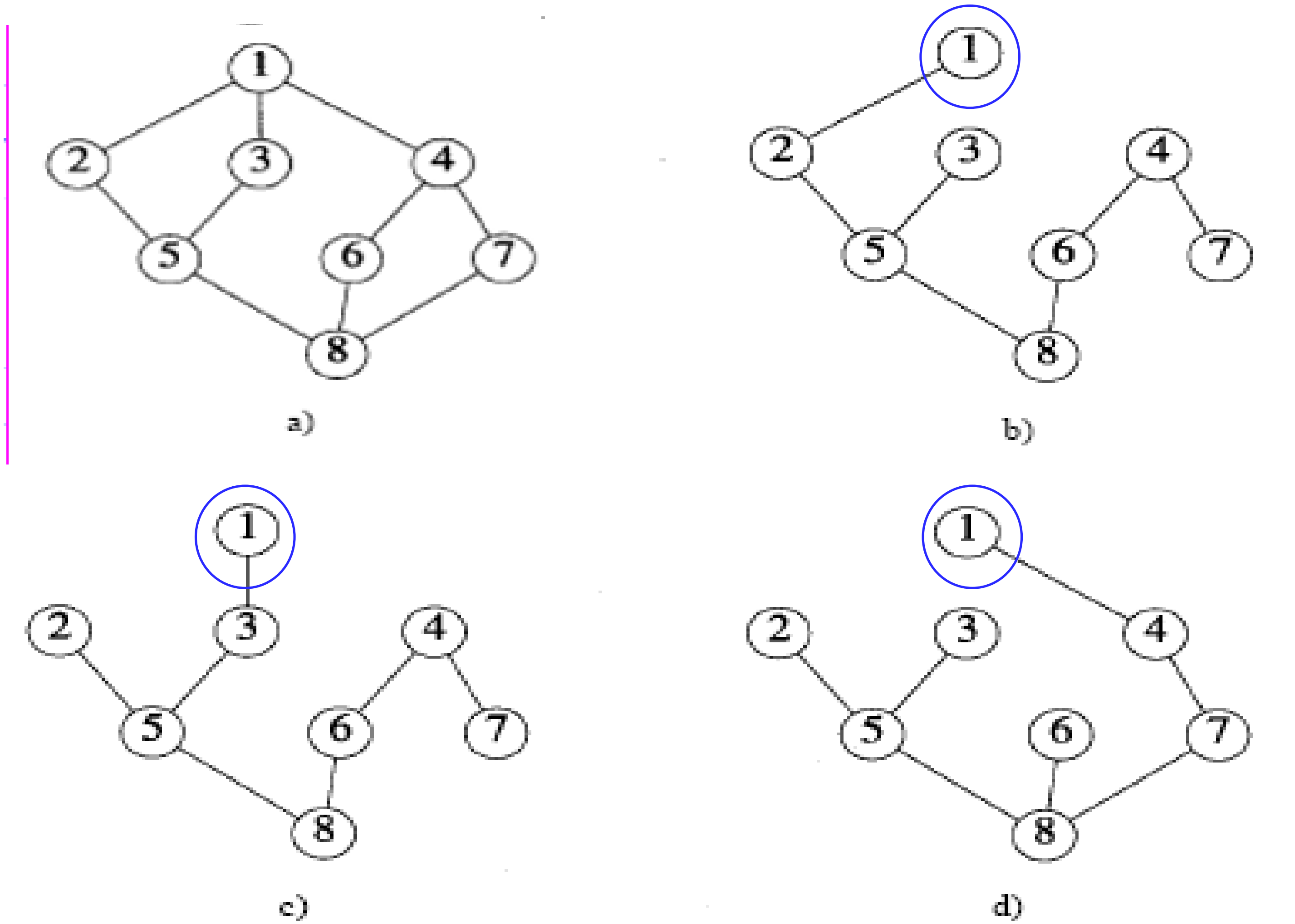
- 在一个n 顶点的**连通无向图**中，如果从任一个顶点开始进行BFS (DFS)，有n - 1个顶点是可到达的。
- 通过一条边到达一个新顶点u
  - ➔用来到达n - 1个顶点的边的数目正好是n - 1。
- 用来到达n - 1个顶点的边的集合中包含从v 到图中其他每个顶点的路径，因此它构成了一个连通子图，该子图即为G的生成树。➔<u>宽度优先生成树(深度优先生成树)</u>。

宽度优先生成树示例



{(1,2,), (1,3), (1,4), (2,5), (4,6), (4,7), (5,8)} ?

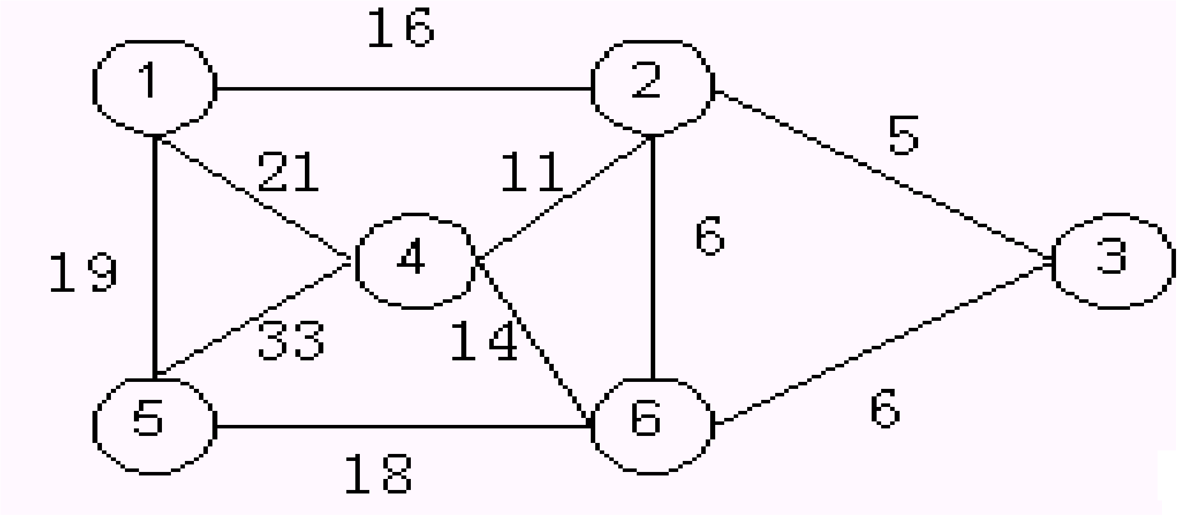
深度优先生成树示例



{(1,3),(3,5),(5,2),(5,8),(8,7),(7,4),(4,6)}　　?

练习

- 分别用深度优先搜索和宽度优先搜索遍历下图所示的无向图，给出以1为起点的顶点访问序列（同一个顶点的多个邻接点，按数字顺序访问），给出一棵深度优先生成树和宽度优先生成树。



作业

9. 对于图 16-8 的每一个图，确定下列的值：

1）每个顶点的入度。

2）每个顶点的出度。

3）邻接于顶点 2 的顶点集合。

4）邻接至顶点 1 的顶点集合。

5）关联于顶点 3 的边的集合。

6）关联至顶点 4 的边的集合。

7）所有的有向环路和它们的长度。

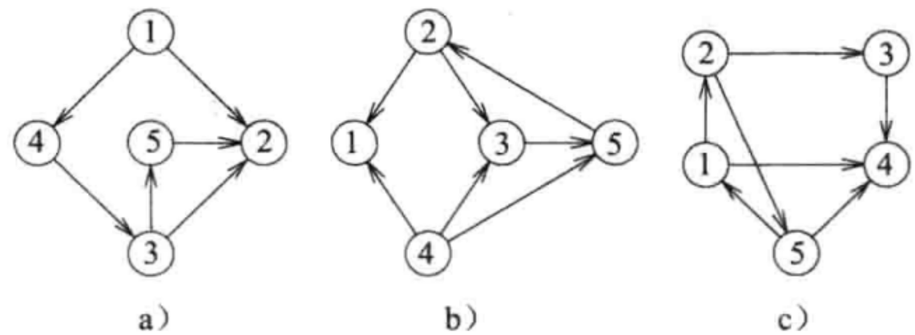


图 16-8　有向图

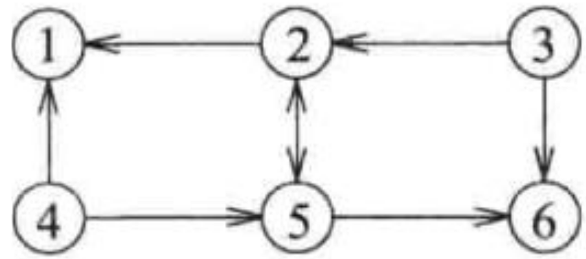
作业

16. 为图 16-2b 画出下列描述图；

1）邻接矩阵。

2）邻接链表。

3）邻接数组。



b）有向图

作业

18 为图 16- 5 画出下列描述图；

1）邻接矩阵。

2）邻接链表。

3）邻接数组。

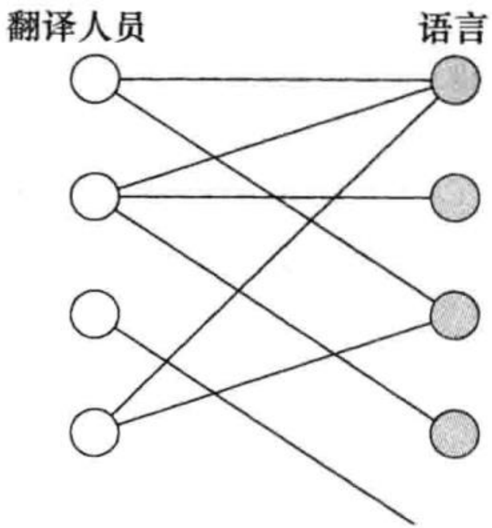


图 16-5 翻译人员与语言

作业

19 为图 16- 8 画出下列描述图；

1）邻接矩阵。

2）邻接链表。

3）邻接数组。

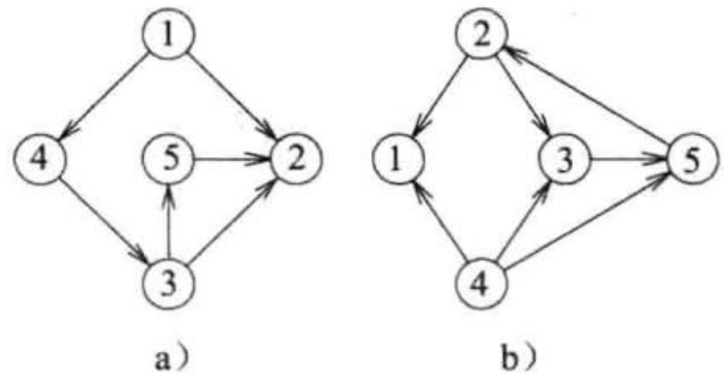


图 16-8 有向图

作业

31. 对图 16-13b 和 16-13c 的成本邻接矩阵，画出该加权图的邻接链表。

$$
\begin{array}{cc}
 & \begin{matrix}1 & 2 & 3 & 4\end{matrix} \\
\begin{matrix}1\\2\\3\\4\end{matrix} & \begin{bmatrix} - & 4 & 7 & 8 \\ 4 & - & 2 & - \\ 7 & 2 & - & 6 \\ 8 & - & 6 & - \end{bmatrix}
\end{array}
\quad
\begin{array}{cc}
 & \begin{matrix}1 & 2 & 3 & 4 & 5 & 6 & 7\end{matrix} \\
\begin{matrix}1\\2\\3\\4\\5\\6\\7\end{matrix} & \begin{bmatrix} - & 9 & 5 & - & - & - & - \\ 9 & - & - & - & - & - & - \\ 5 & - & - & - & - & - & - \\ - & - & - & - & 3 & - & - \\ - & - & - & 3 & - & 6 & 4 \\ - & - & - & - & 6 & - & 1 \\ - & - & - & - & 4 & 1 & - \end{bmatrix}
\end{array}
\quad
\begin{array}{cc}
 & \begin{matrix}1 & 2 & 3 & 4 & 5\end{matrix} \\
\begin{matrix}1\\2\\3\\4\\5\end{matrix} & \begin{bmatrix} - & 8 & - & - & - \\ - & - & 3 & - & - \\ - & - & - & 2 & 7 \\ - & - & 6 & - & - \\ - & - & - & 5 & - \end{bmatrix}
\end{array}
$$

a)　　　　b)　　　　c)

破折号表示不存在的边

图 16-13　图 16-1 对应的可能的成本邻接矩阵

作业

31. 对图 16-13b 和 16-13c 的成本邻接矩阵，画出该加权图的邻接链表。

$$
\begin{array}{cc}
 & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\
\begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} &
\begin{bmatrix}
- & 4 & 7 & 8 \\
4 & - & 2 & - \\
7 & 2 & - & 6 \\
8 & - & 6 & -
\end{bmatrix}
\end{array}
$$

a)

$$
\begin{array}{cc}
 & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{matrix} \\
\begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{matrix} &
\begin{bmatrix}
- & 9 & 5 & - & - & - & - \\
9 & - & - & - & - & - & - \\
5 & - & - & - & - & - & - \\
- & - & - & - & 3 & - & - \\
- & - & - & 3 & - & 6 & 4 \\
- & - & - & - & 6 & - & 1 \\
- & - & - & - & 4 & 1 & -
\end{bmatrix}
\end{array}
$$

b)

$$
\begin{array}{cc}
 & \begin{matrix} 1 & 2 & 3 & 4 & 5 \end{matrix} \\
\begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} &
\begin{bmatrix}
- & 8 & - & - & - \\
- & - & 3 & - & - \\
- & - & - & 2 & 7 \\
- & - & 6 & - & - \\
- & - & - & 5 & -
\end{bmatrix}
\end{array}
$$

c)

破折号表示不存在的边

图 16-13　图 16-1 对应的可能的成本邻接矩阵

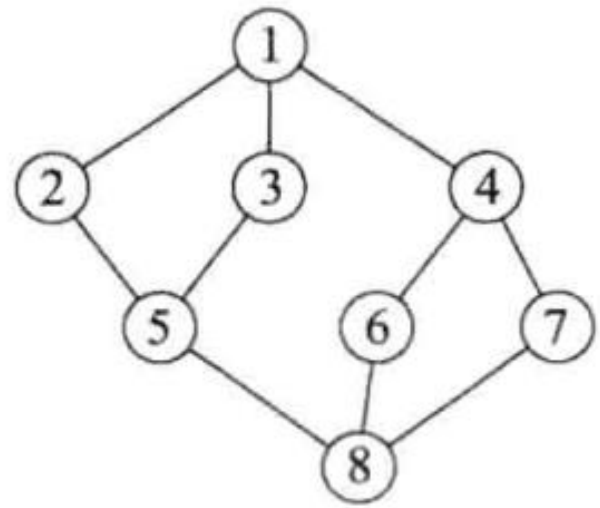
46. 根据图 16-20a，完成以下练习：

1）画出从顶点 3 开始的一个广度优先生成树。

2）画出从顶点 7 开始的一个广度优先生成树。

3）画出从顶点 3 开始的一个深度优先生成树。

4）画出从顶点 7 开始的一个深度优先生成树。



a）

50. 编写公有方法 graph::cycle()，用于确定一个无向图是否有一个环路。即可用 DFS 也可用 BFS 来实现。

1）证明代码的正确性。

2）确定程序的时间和空间复杂性。

52. 设 G 是一个无向图。它的**传递闭包**（tansitive closure）是一个 0/1 数组 tc，当且仅当 G 存在一条边数大于 1 的从 i 到 j 的路径时，tc[i][j]=1。编写一个方法 graph::undirectedTC()，计算且返回 G 的传递闭包。方法的复杂性应为 $O(n^2)$，其中 n 是 G 的顶点数。（提示：采用构件标记策略。）