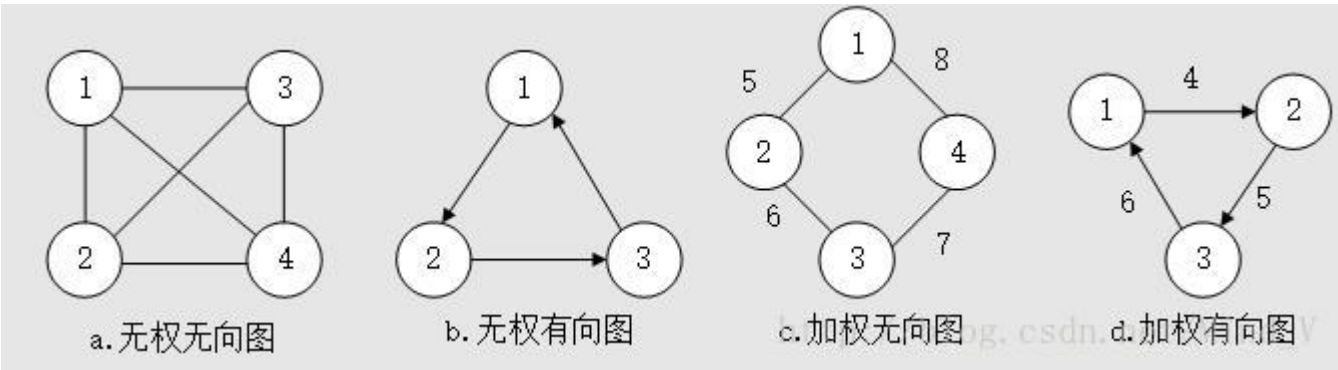


# 基本概念

图 (graph) 是用线连接在一起的顶点或节点的集合，即两个要素：**边**和**顶点**。每一条边连接个两个顶点，用 (i, j) 表示顶点为 i 和 j 的边。

如果用图示来表示一个图，一般用圆圈表示顶点，线段表示边。有方向的边称为**有向边**，对应的图成为**有向图**，没有方向的边称为**无向边**，对应的图叫**无向图**。对于**无向图**，边 (i, j) 和 (j, i) 是一样的，称顶点 i 和 j 是**邻接的**，边 (i, j) **关联于**顶点 i 和 j；对于**有向图**，边 (i, j) 表示由顶点 i 指向顶点 j 的边，即称顶点 i **邻接至**顶点 j，顶点 i **邻接于**顶点 j，边 (i, j) **关联至**顶点 j 而**关联于**顶点 i。

对于很多的实际问题，不同顶点之间的边的权值（长度、重量、成本、价值等实际意义）是不一样的，所以这样的图被称为**加权图**，反之边没有权值的图称为**无权图**。所以，图分为四种：加权有向图，加权无向图，无权有向图，无权无向图。



在一个无向图中，与一个顶点相关联的边数成为该顶点的**度**。而对于有向图，则用**入度**来表示关联至该顶点的边数，**出度**来表示关联于该顶点的边数。

一个具有n个顶点和 $n(n-1)/2$ 条边的无向图称为一个**完全图**，即每个顶点的度等于总顶点数减1。

## 图的描述

### 抽象数据类型

定义抽象数据类型graph，有向图、无向图、加权图和无权图都可以根据此ADT实现。

```
numberOfVertices(): 返回图的顶点数

numberOfEdges: 返回图的边数

existsEdge(i, j): 如果边 (i, j) 存在，则返回true，否则返回false

insertEdge(theEdge): 插入边theEdge

eraseEdge(i, j): 删除边 (i, j)
```

`degree(i)`: 返回顶点 *i* 的度（无向图）

`inDegree(i)`: 返回顶点 *i* 的入度

`outDegree(i)`: 返回顶点 *i* 的出度

`directed()`: 当且仅当有向图，返回`true`

`weighted()`: 当且仅当加权图，返回`true`

## 无权图的描述

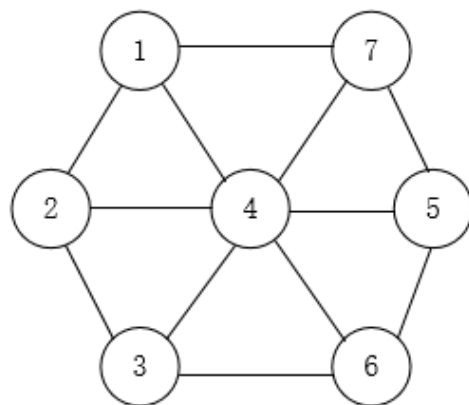
对无向图的描述方法都是基于邻接的方式：邻接矩阵、邻接 **链表** 和邻接数组。

### 邻接矩阵

用图e的邻接矩阵表示图f的无权图， $A(i, j)$ 等于1表示边  $(i, j)$  存在，等于0则表示不存在。有向图无向图都可以用矩阵这样表示，但是对于无向图，矩阵具有对称性，即 $A(i, j)$ 和 $A(j, i)$ 一样，所以为了节省空间，可以用下三角（上三角）矩阵表示。

	1	2	3	4	5	6	7
1	0	1	0	1	0	0	1
2	1	0	1	1	0	0	0
3	0	1	0	1	0	1	0
4	1	1	1	1	1	1	1
5	0	0	0	1	0	1	1
6	0	0	1	1	1	0	0
7	1	0	0	1	1	0	0

e. 图f对应的邻接矩阵

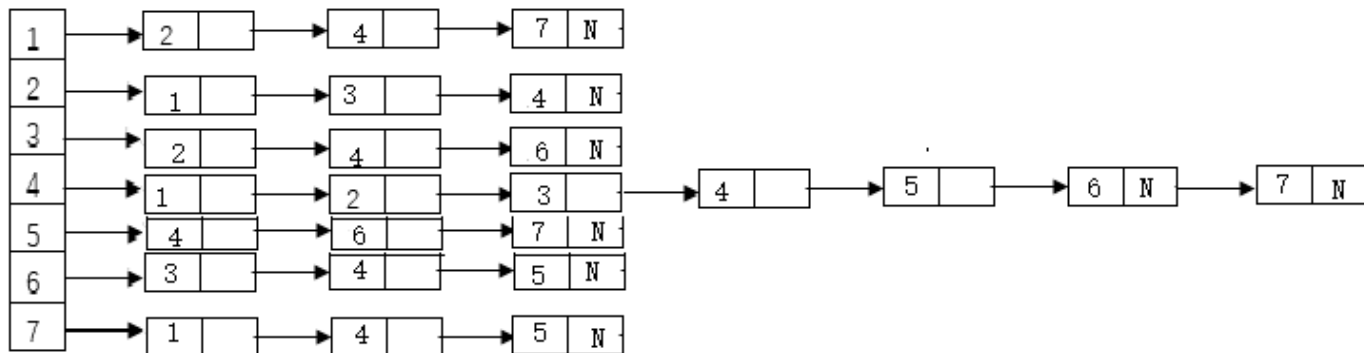


f. 无权图 [http://img.csdn.net/Mind\\_V](http://img.csdn.net/Mind_V)

### 邻接链表

图g是图f对应的邻接链表，每一个和顶点 *i* 关联的顶点 *j*，依次构成一条链表，节点的元素是顶点，节点的`next`指针指向下一个 *j* 构成的顶点，最后一个节点指向`nullptr`，7条链表用一个表`List`来维护。

list

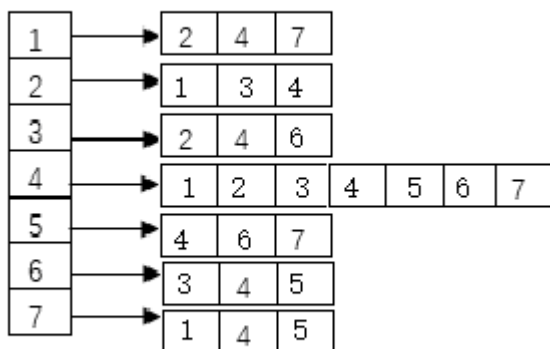


g. 图f对应的邻接链表 [http://blog.csdn.net/Mind\\_V](http://blog.csdn.net/Mind_V)

## 邻接数组

图g是图f对应的邻接数组，可以看成是一个列数不等的二维数组，也可以看成由一个元素为一维数组的表List。

List



[http://blog.csdn.net/Mind\\_V](http://blog.csdn.net/Mind_V)

## 加权图的描述

对于加权图，一般使用邻接矩阵或邻接链表描述。

### 邻接矩阵

和无权图的邻接矩阵描述相同道理，只不过 $A(i, j)$ 的值不再是0或1，而是对应的加权图中的权值，而对于不存在的边，可以用一个固定的值表示，如-1。

### 邻接链表

和无权图的邻接链表描述也类似，每一个和顶点  $i$  关联的顶点  $j$  依次构成一条链表，不过需要改变的是节点元素包含两个域{顶点  $j$ , 权值}，节点next指向下一个节点，也用一个List维护这些链表。

## 类实现(C++)

使用邻接矩阵描述实现加权有向图，权值类型为T，无向图和无权图都可以由它派生出。

```

1 //加权图的边
2 template <typename T>
3 class weightedEdge {
4 public:
5     weightedEdge() {}
6     weightedEdge(int theV1, int theV2, T theW) : v1(theV1), v2(theV2), w(t
7     ~weightedEdge() {}
8
9     int vertex1() const { return v1; }
10    int vertex2() const { return v2; }
11    T weight() const { return w; }
12    operator T() const { return w; }
13 private:
14    int v1,
15        v2;
16    T w;
17 };

```

```

1 //邻接矩阵加权有向图
2 template <typename T>
3 class adjacencyWDigraph {
4 public:
5     //构造函数和析构函数
6     adjacencyWDigraph(int numberOfVertices = 0, T theNoEdge = 0);
7     ~adjacencyWDigraph();
8
9     int numberOfVertices() const { return n; }
10    int numberOfEdges() const { return e; }
11    bool directed() const { return true; }
12    bool weighted() const { return true; }
13
14    //判断边(i, j)是否存在
15    bool existEdge(int i, int j) const;
16    //插入、删除边
17    void insertEdge(weightedEdge<T> *theEdge);
18    void eraseEdge(int i, int j);
19
20    //顶点的入度和出度
21    int inDegree(int theVertex) const;
22    int outDegree(int theVertex) const;
23
24    //广度优先遍历和深度优先遍历
25    void bfs(int v, int reach[], int lable);
26    void dfs(int v, int reach[], int lable);
27

```

```

28
29 //迭代器
30 class iterator {
31 public:
32     iterator(T *theRow, T theNoEdge, int numberOfVertices) {
33         row = theRow;
34         noEdge = theNoEdge;
35         n = numberOfVertices;
36         currentVertex = 1;
37     }
38
39     ~iterator() {}
40
41     int next() {
42         for (int j = currentVertex; j <= n; ++j)
43             if (row[j] != noEdge) {
44                 currentVertex = j + 1;
45                 return j;
46             }
47
48         currentVertex = n + 1;
49         return 0;
50     }
51
52     int next(T &theWeight) {
53         for (int j = currentVertex; j <= n; ++j)
54             if (row[j] != noEdge) {
55                 currentVertex = j + 1;
56                 theWeight = row[j];
57                 return j;
58             }
59
60         currentVertex = n + 1;
61         return 0;
62     }
63
64 private:
65     T *row;           //邻接矩阵的行
66     T noEdge;
67     int n;
68     int currentVertex;
69 };
70
71 //生成迭代器
72 iterator* makeIterator(int theVertex) {
73     checkVertex(theVertex);
74     return new iterator(a[theVertex], noEdge, n);
75

```

```

76     }
77
78 private:
79     int n;        //顶点数
80     int e;        //边数
81     int **a;      //邻接矩阵（二维数组）
82     T noEdge;     //表示不存在的边
83
84                 //检查顶点是否存在
85     void checkVertex(int theVertex) const;
86 };
87
88 template <typename T>
89 adjacencyWDigraph<T>::adjacencyWDigraph(int numberOfVertices = 0, T theNoE
90     if (numberOfVertices < 0) {
91         cout << "number of vertices must be >= 0";
92         exit(1);
93     }
94
95     n = numberOfVertices;
96     e = 0;
97     noEdge = theNoEdge;
98     a = new T*[n + 1];
99     for (int i = 0; i <= n; ++i)
100         a[i] = new int[n + 1];
101
102     //初始化邻接矩阵
103     for (int i = 1; i <= n; ++i)
104         fill(a[i], a[i] + n + 1, noEdge);
105 }
106
107 template <typename T>
108 adjacencyWDigraph<T>::~~adjacencyWDigraph<T>() {
109     for (int i = 0; i <= n; ++i)
110         delete[] a[i];
111     delete[] a;
112     a = nullptr;
113 }
114
115 template <typename T>
116 bool adjacencyWDigraph<T>::existEdge(int i, int j) const {
117     if (i < 1 || i > n || j < 1 || j > n || a[i][j] == noEdge)
118         return false;
119     else
120         return true;
121 }
122
123

```

```

123 template <typename T>
124 void adjacencyWDigraph<T>::insertEdge(weightedEdge<T> *theEdge) {
125     int v1 = theEdge->vertex1();
126     int v2 = theEdge->vertex2();
127     if (v1 < 1 || v1 > n || v2 < 1 || v2 > n || v1 == v2) {
128         cout << "(" << v1 << "," << v2 << ") is not a permissble edge";
129         exit(1);
130     }
131
132     if (a[v1][v2] == noEdge)
133         ++e;
134     a[v1][v2] = theEdge->weight();
135 }
136
137 template <typename T>
138 void adjacencyWDigraph<T>::eraseEdge(int i, int j) {
139     if (i >= 1 && i <= n && j >= 1 && j <= n && a[i][j] != noEdge) {
140         a[i][j] = noEdge;
141         --e;
142     }
143 }
144
145 template <typename T>
146 void adjacencyWDigraph<T>::checkVertex(int theVertex) const {
147     if (theVertex < 1 || theVertex > n) {
148         cout << "no vertex " << theVertex;
149         exit(1);
150     }
151 }
152
153 template <typename T>
154 int adjacencyWDigraph<T>::inDegree(int theVertex) const {
155     checkVertex(theVertex);
156
157     int sum = 0;
158     for (int i = 1; i <= n; ++i)
159         if (a[i][theVertex] != noEdge)
160             ++sum;
161
162     return sum;
163 }
164
165 template <typename T>
166 int adjacencyWDigraph<T>::outDegree(int theVertex) const {
167     checkVertex(theVertex);
168
169     int sum = 0;
170

```

```

171     for (int j = 1; j <= n; ++j)
172         if (a[theVertex][j] != noEdge)
173             ++sum;

    return sum;
}

```

## 图的遍历

很多算法需要从一个已知的顶点开始，搜索所有可以到达的顶点。所谓顶点 $u$ 是从顶点 $v$ 可达的，是指有一条顶点 $v$ 到顶点 $u$ 的路径。这种搜索有两种常见的方法：广度优先搜索（breadth first search, BFS）和深度优先搜索（depth first search, DFS）。一般来说，深度优先搜索方法效率更高，使用的也更多。

### 广度优先搜索

```

1  template <typename T>
2  void adjacencyWDigraph<T>::bfs(int v, int reach[], int lable) {
3      queue<int> q;
4      reach[v] = lable;
5
6      q.push(v);
7      while (!q.empty()) {
8          //从队列中删除一个标记过的顶点
9          int w = q.front();
10         q.pop();
11
12         iterator *iw = makeIterator(w);
13         int u;
14         while((u = iw->next()) != 0)
15             //u是w的相邻顶点
16             if (reach[u] == 0) {
17                 q.push(u);
18                 reach[u] = lable;
19             }
20
21         delete iw;
22     }
23 }

```

### 深度优先搜索



```
template <typename T>
void adjacencyWDigraph<T>::dfs(int v, int reach[], int lable) {
    reach[v] = lable;
    iterator *iv = makeIterator(v);

    int u;
    while ((u = iv->next()) != 0)
        //u是v的相邻顶点
        if (reach[u] == 0)
            dfs(u, reach, lable);

    delete iv;
}
```