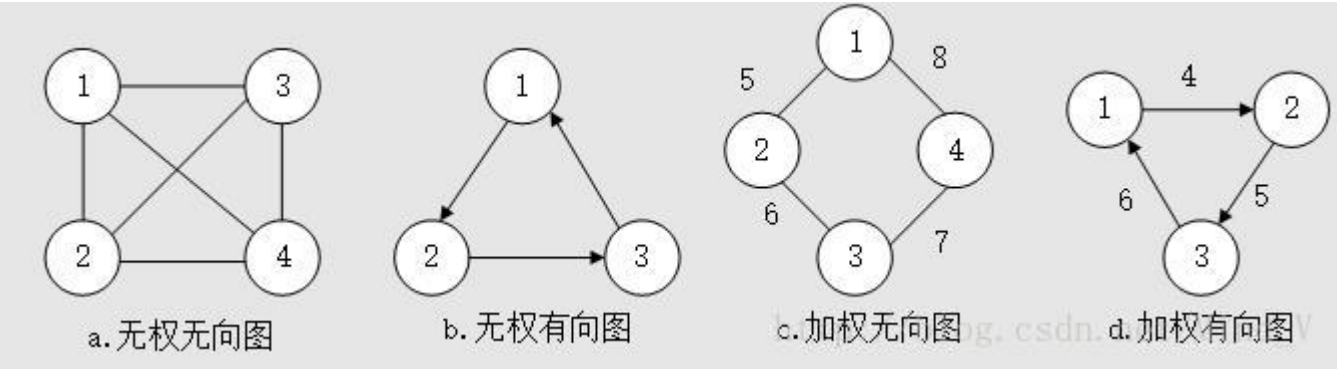


基本概念

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

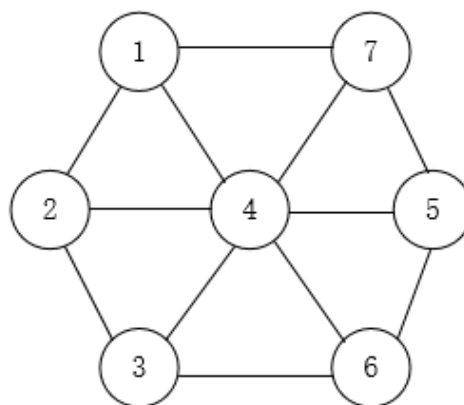
对无向图的描述方法都是基于邻接的方式：邻接矩阵、邻接链表和邻接数组。

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

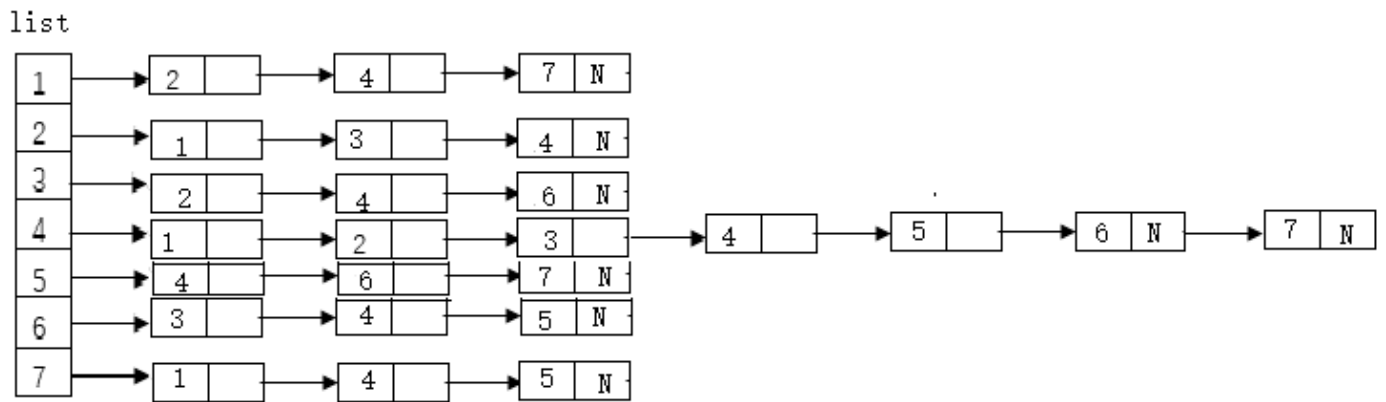
e. 图f对应的邻接矩阵



f. 无权图

邻接链表

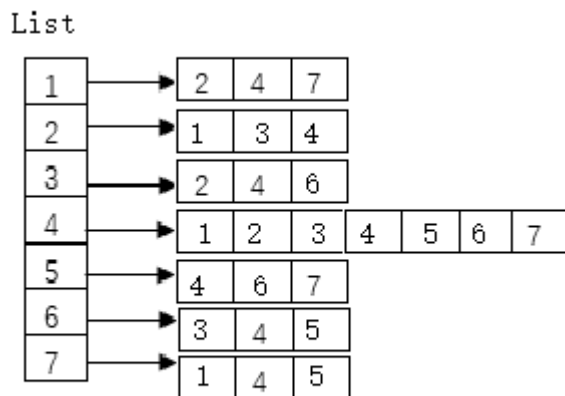
图g是图f对应的邻接链表，每一个和顶点 i 关联的顶点 j，依次构成一条链表，节点的元素是顶点，节点的next指针指向下一个 j 构成的顶点，最后一个节点指向nullptr，7条链表用一个表List来维护。



g. 图f对应的邻接链表 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。



h. 图f对应的邻接数组 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(theW) {}
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     //迭代器
29     class iterator {
30     public:
31         iterator(T *theRow, T theNoEdge, int numberOfVertices) {
32             row = theRow;
33             noEdge = theNoEdge;
34             n = numberOfVertices;
35             currentVertex = 1;
36         }
37
38         ~iterator() {}
39
40         int next() {
41             for (int j = currentVertex; j <= n; ++j)
42                 if (row[j] != noEdge) {
43                     currentVertex = j + 1;
44                     return j;
45                 }
46
47             currentVertex = n + 1;
48             return 0;
49         }
50
51         int next(T &theWeight) {
52             for (int j = currentVertex; j <= n; ++j)
53                 if (row[j] != noEdge) {
54                     currentVertex = j + 1;
55                     theWeight = row[j];
56                     return j;
57                 }
58
59             currentVertex = n + 1;
60             return 0;
61         }
62
63     private:
64         T *row;           //邻接矩阵的行
65         T noEdge;
66         int n;
67         int currentVertex;
68     };

```

```

69
70     //生成迭代器
71     iterator* makeIterator(int theVertex) {
72         checkVertex(theVertex);
73         return new iterator(a[theVertex], noEdge, n);
74     }
75
76 private:
77     int n;        //顶点数
78     int e;        //边数
79     int **a;      //邻接矩阵（二维数组）
80     T noEdge;     //表示不存在的边
81
82     //检查顶点是否存在
83     void checkVertex(int theVertex) const;
84 };
85
86 template <typename T>
87 adjacencyWDigraph<T>::adjacencyWDigraph(int numberOfVertices = 0, T theNoEdge) {
88     if (numberOfVertices < 0) {
89         cout << "number of vertices must be >= 0";
90         exit(1);
91     }
92
93     n = numberOfVertices;
94     e = 0;
95     noEdge = theNoEdge;
96     a = new T*[n + 1];
97     for (int i = 0; i <= n; ++i)
98         a[i] = new int[n + 1];
99
100     //初始化邻接矩阵
101     for (int i = 1; i <= n; ++i)
102         fill(a[i], a[i] + n + 1, noEdge);
103 }
104
105 template <typename T>
106 adjacencyWDigraph<T>::~~adjacencyWDigraph<T>() {
107     for (int i = 0; i <= n; ++i)
108         delete[] a[i];
109     delete[] a;
110     a = nullptr;
111 }
112
113 template <typename T>
114 bool adjacencyWDigraph<T>::existEdge(int i, int j) const {
115     if (i < 1 || i > n || j < 1 || j > n || a[i][j] == noEdge)

```

```

116         return false;
117     else
118         return true;
119 }
120
121 template <typename T>
122 void adjacencyWDigraph<T>::insertEdge(weightedEdge<T> *theEdge) {
123     int v1 = theEdge->vertex1();
124     int v2 = theEdge->vertex2();
125     if (v1 < 1 || v1 > n || v2 < 1 || v2 > n || v1 == v2) {
126         cout << "(" << v1 << "," << v2 << ") is not a permissble edge";
127         exit(1);
128     }
129
130     if (a[v1][v2] == noEdge)
131         ++e;
132     a[v1][v2] = theEdge->weight();
133 }
134
135 template <typename T>
136 void adjacencyWDigraph<T>::eraseEdge(int i, int j) {
137     if (i >= 1 && i <= n && j >= 1 && j <= n && a[i][j] != noEdge) {
138         a[i][j] = noEdge;
139         --e;
140     }
141 }
142
143 template <typename T>
144 void adjacencyWDigraph<T>::checkVertex(int theVertex) const {
145     if (theVertex < 1 || theVertex > n) {
146         cout << "no vertex " << theVertex;
147         exit(1);
148     }
149 }
150
151 template <typename T>
152 int adjacencyWDigraph<T>::inDegree(int theVertex) const {
153     checkVertex(theVertex);
154
155     int sum = 0;
156     for (int i = 1; i <= n; ++i)
157         if (a[i][theVertex] != noEdge)
158             ++sum;
159
160     return sum;
161 }
162

```

```

163 template <typename T>
164 int adjacencyWDigraph<T>::outDegree(int theVertex) const {
165     checkVertex(theVertex);
166
167     int sum = 0;
168     for (int j = 1; j <= n; ++j)
169         if (a[theVertex][j] != noEdge)
170             ++sum;
171
172     return sum;
173 }

```

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可到达的，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;
}
```