第8章 图

8.1 图的基本概念

8.2 图的存储结构

8.3 图的遍历

8.4 最小生成树

8.5 一点到其他点最短路径问题

8.6 拓扑排序

8.1图的基本概念

- **图定义** 图是由顶点集合(vertex)及顶点间的关系集合组成的一种数据结构：

$$Graph = (V, E)$$

其中 $V = \{x \mid x \in$ 某个数据对象$\}$

是顶点的有穷非空集合；

$$E = \{(x, y) \mid x, y \in V\}$$

或 $$E = \{<x, y> \mid x, y \in V \&\& Path(x, y)\}$$

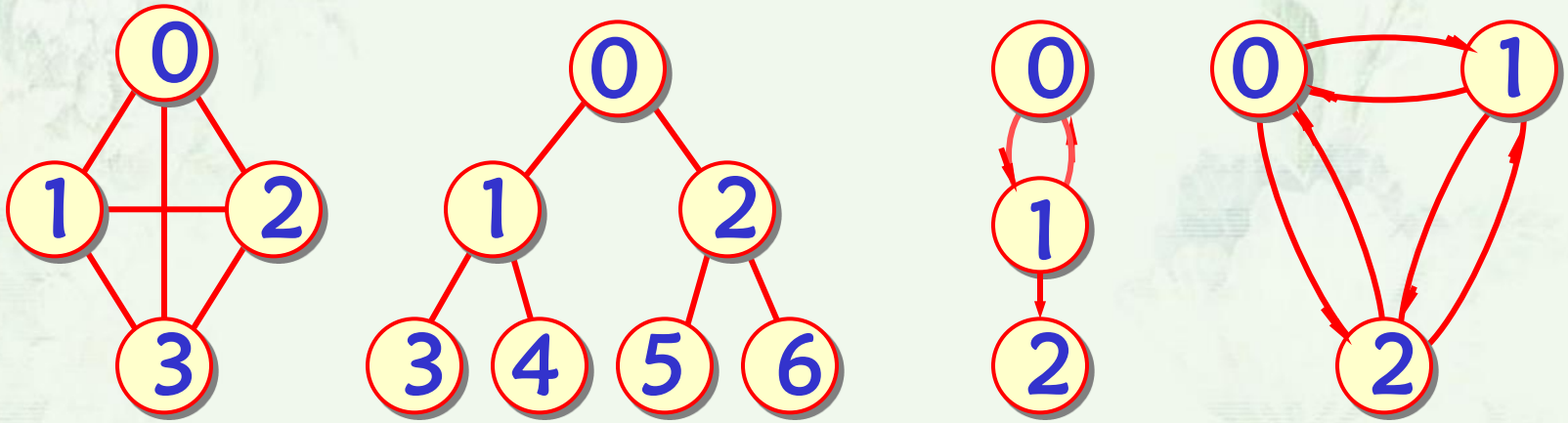
- **有向图与无向图**

  <x, y> 是有序的。

  边<x, y>就称为**弧**，x为**弧尾**，y为**弧头**。

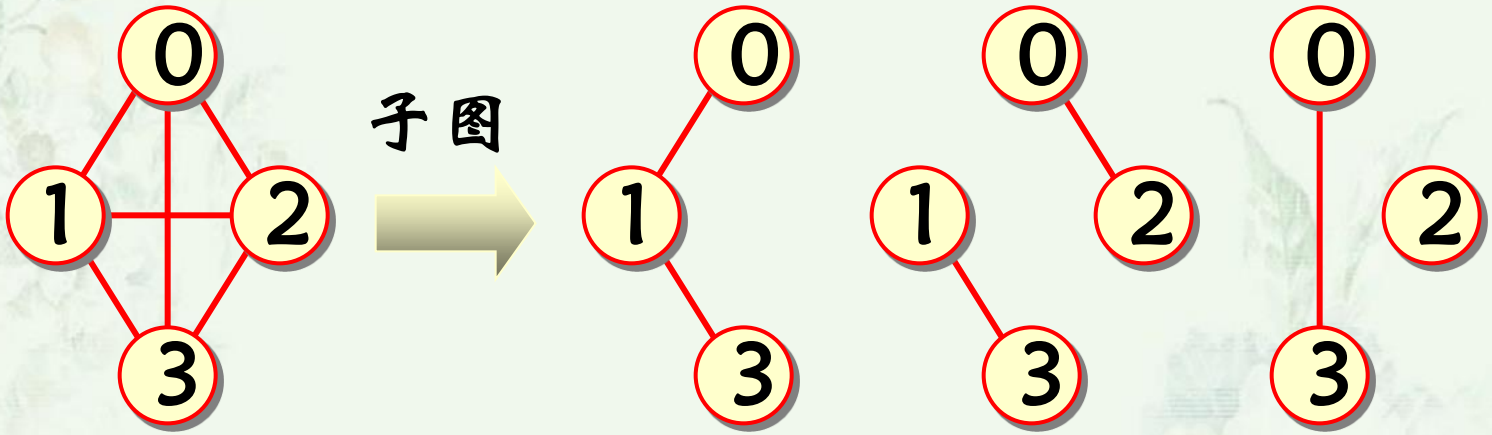
  (x, y)是无序的。

- **完全图**

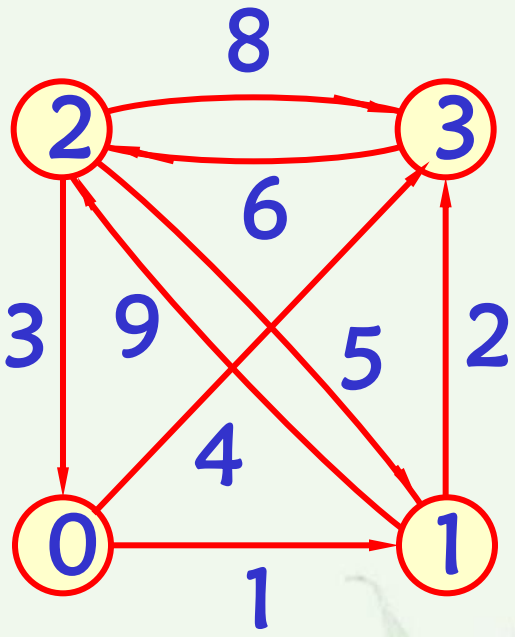


- **邻接顶点**

- **子图** 设有两个图 G = (V, E) 和 G’ = (V’, E’)。若 V’ ⊆ V 且 E’ ⊆ E, 则称 图G’ 是 图G 的子图。



- 权
- 带权图也叫做网络。



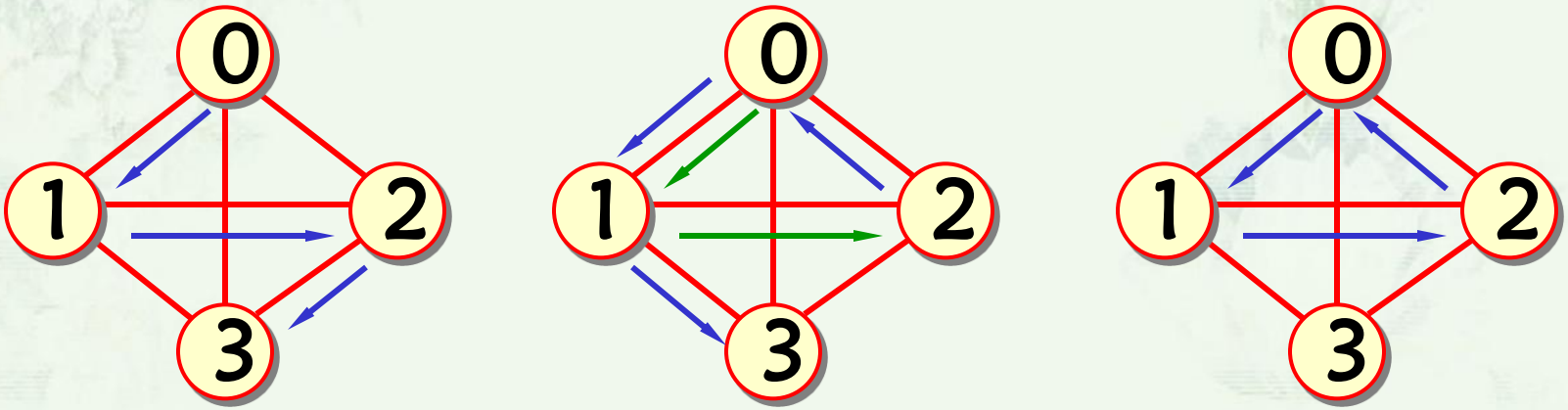
- 稠密图和稀疏图　　e<nlogn

- 顶点的度
- 入度
- 出度
- 路径

- **路径长度**

- **简单路径** 若路径上各顶点 $v_1, v_2, \ldots, v_m$ 均不互相重复，则称这样的路径为简单路径。

- **回路** 若路径上第一个顶点 v_1 与最后一个顶点 v_m 重合，则称这样的路径为回路或环。



- 连通图与连通分量

- 强连通图与强连通分量

- 生成树　一个连通图的生成树是其极小连通子图.

操作的归纳

图的抽象数据类型

```
class Graph {
//对象: 由一个顶点的非空集合和一个边集合构成
//每条边由一个顶点对来表示。
public:
   Graph();                          //建立一个空的图
   void insertVertex (const T& vertex);
     //插入一个顶点vertex, 该顶点暂时没有入边
   void insertEdge (int v1, int v2, int weight);
     //在图中插入一条边(v1, v2, w)
   void removeVertex (int v);
     //在图中删除顶点v和所有关联到它的边
```

```
    void removeEdge (int v1, int v2);
      //在图中删去边(v1,v2)
    bool IsEmpty();
      //若图中没有顶点, 则返回true, 否则返回false
    T getWeight (int v1, int v2);
      //函数返回边 (v1,v2) 的权值
    int getFirstNeighbor (int v);
      //给出顶点 v 第一个邻接顶点的位置
    int getNextNeighbor (int v, int w);
      //给出顶点 v 的某邻接顶点 w 的下一个邻接顶点
};
```

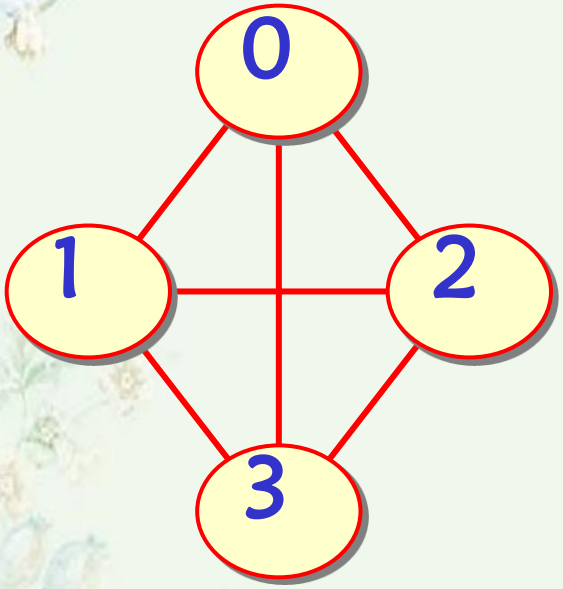
图的模板基类

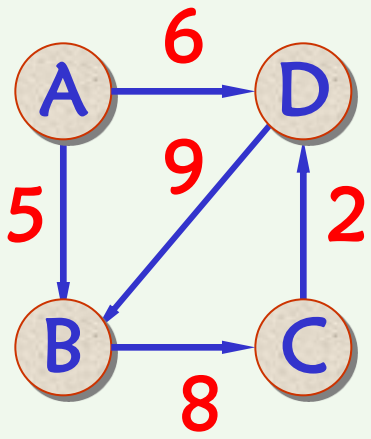
```
const int maxWeight = ……;          //无穷大的值
   (=∞)
const int DefaultVertices = 30;     //最大顶点数
   (=n)
template <class T, class E>
class Graph {                       //图的类定义
protected:
    int maxVertices;                //图中最大顶点数
    int numEdges;                   //当前边数
    int numVertices;                //当前顶点数
    int getVertexPos (T vertex);
      //给出顶点vertex在图中位置
public:
```

```
Graph (int sz = DefaultVertices);        //构造函数
~Graph();                                //析构函数
bool GraphEmpty () const                 //判图空否
   { return numEdges == 0; }
int NumberOfVertices () { return numVertices; }
  //返回当前顶点数
int NumberOfEdges () { return numEdges; }
  //返回当前边数
virtual T getValue (int i);              //取顶点 i 的值
virtual E getWeight (int v1, int v2);  //取边上权值
virtual int getFirstNeighbor (int v);
  //取顶点 v 的第一个邻接顶点
```

```
    virtual int getNextNeighbor (int v, int w);
      //取邻接顶点 w 的下一邻接顶点
    virtual bool insertVertex (const T vertex);
      //插入一个顶点vertex
    virtual bool insertEdge (int v1, int v2, E cost);
      //插入边(v1,v2), 权为cost
    virtual bool removeVertex (int v);
      //删去顶点 v 和所有与相关联边
    virtual bool removeEdge (int v1, int v2);
      //在图中删去边(v1,v2)
};
```

图的存储结构



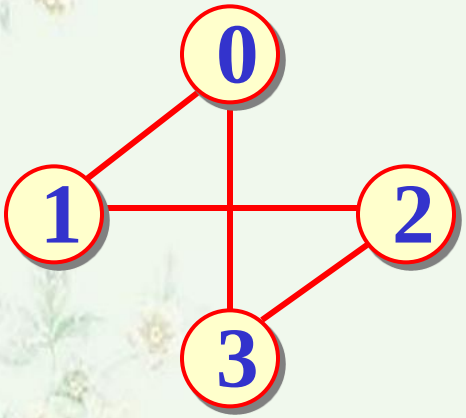


8.2 图的存储表示

一、邻接矩阵（相邻矩阵）

- 图(*n* 个顶点)的邻接矩阵是一个**二维数组** edge[*n*][*n*]，
- **定义：**

$$A.Edge[i][j]=\begin{cases}1, & \text{如果} <i,j> \in E \text{ 或者 } (i,j)\in E \\ 0, & \text{否则}\end{cases}$$

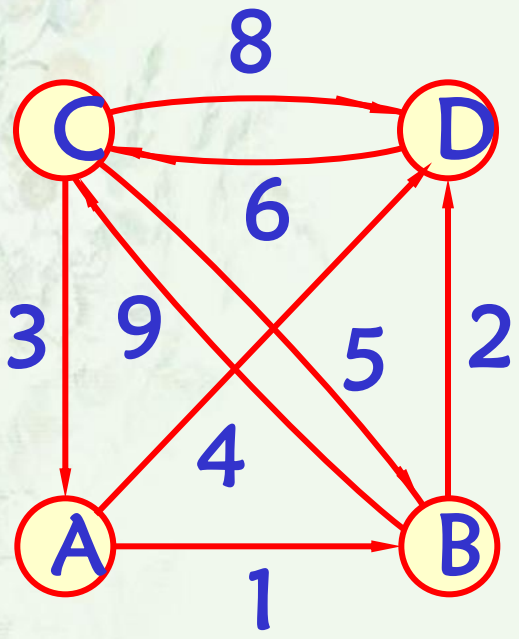


$$A.edge = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

$$A.edge = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

网络(带权图)的邻接矩阵

$$A.edge[i][j]=\begin{cases} W(i,j), & \text{若} i\neq j \text{且} <i,j>\in E \text{或} (i,j)\in E \\ \infty, & \text{若} i\neq j \text{且} <i,j>\notin E \text{或} (i,j)\notin E \\ 0, & \text{若} i = j \end{cases}$$



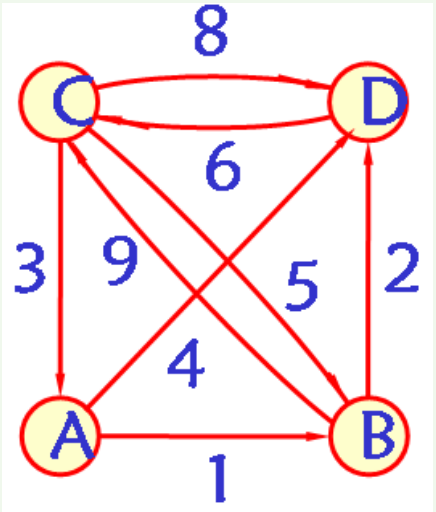
$$A.edge=\begin{bmatrix} 0 & 1 & \infty & 4 \\ \infty & 0 & 9 & 2 \\ 3 & 5 & 0 & 8 \\ \infty & \infty & 6 & 0 \end{bmatrix}$$

邻接矩阵存储图的类定义

```
template <class T, class E>
class Graphmtx : public Graph<T, E> {
friend istream& operator >> ( istream& in,
   Graphmtx<T, E>& G);                    //输入
friend ostream& operator << (ostream&
   out, Graphmtx<T, E>& G);               //输出
```

```
private:
    T *VerticesList;                    //顶点表
    E **Edge;                           //邻接矩阵
    int getVertexPos (T vertex) {
      //给出顶点vertex在图中的位置
        for (int i = 0; i < numVertices; i++)
            if (VerticesList[i] == Vertex) return i;
        return -1;
    };
public:
```

A	B	C	D
0	1	2	3



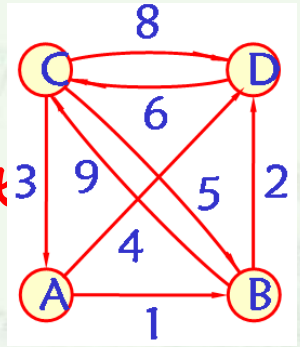
$$\mathbf{A.edge} = \begin{bmatrix} 0 & 1 & \infty & 4 \\ \infty & 0 & 9 & 2 \\ 3 & 5 & 0 & 8 \\ \infty & \infty & 6 & 0 \end{bmatrix}$$

```
Graphmtx (int sz = DefaultVertices);    //构造函数
~Graphmtx ()                            //析构函数
   { delete [ ]VerticesList;  delete [ ]Edge; }
T getValue (int i) {
  //取顶点 i 的值, i 不合理返回0
   return i >= 0 && i <= numVertices ?
                VerticesList[i] : NULL;
}
E getWeight (int v1, int v2) {  //取边(v1,v2)上权值
   return v1 != –1 && v2 != –1 ? Edge[v1][v2] : 0;
}
int getFirstNeighbor (int v);
  //取顶点 v 的第一个邻接顶点
```

```
    int getNextNeighbor (int v, int w);
      //取 v 的邻接顶点 w 的下一邻接顶点
    bool insertVertex (const T vertex);
      //插入顶点vertex
    bool insertEdge (int v1, int v2, E cost);
      //插入边(v1, v2),权值为cost
    bool removeVertex (int v);
      //删去顶点 v 和所有与它相关联的边
    bool removeEdge (int v1, int v2);
      //在图中删去边(v1,v2)
};
```

构造函数

```
template <class T, class E>
Graphmtx<T, E>::Graphmtx (int sz) {      //构造函数
    maxVertices = sz;
    numVertices = 0;  numEdges = 0;
    int i, j;
    VerticesList = new T[maxVertices];  //创建顶点表
    Edge = (E **) new E *[maxVertices];
    for (i = 0; i < maxVertices; i++)
        Edge[i] = new int[maxVertices];   //邻接矩阵
    for (i = 0; i < maxVertices; i++)        //矩阵初始化
        for (j = 0; j < maxVertices; j++)
            Edge[i][j] = (i == j)  ?  0 : maxWeight;
};
```



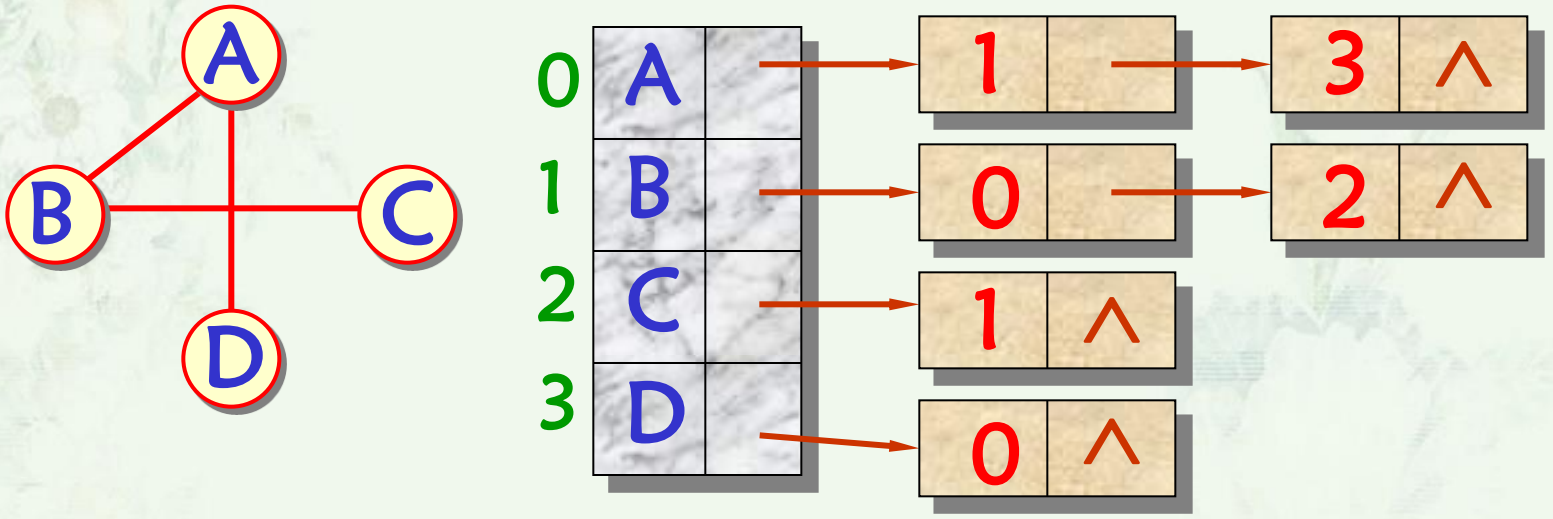
A	B	C	D
0	1	2	3

$$A.edge = \begin{bmatrix} 0 & 1 & \infty & 4 \\ \infty & 0 & 9 & 2 \\ 3 & 5 & 0 & 8 \\ \infty & \infty & 6 & 0 \end{bmatrix}$$

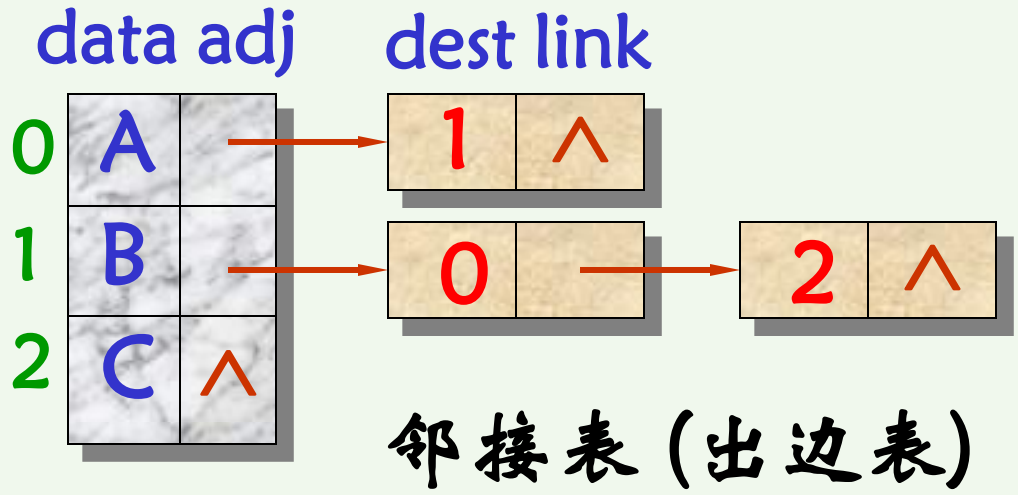
二、邻接表

1.无向图的邻接表

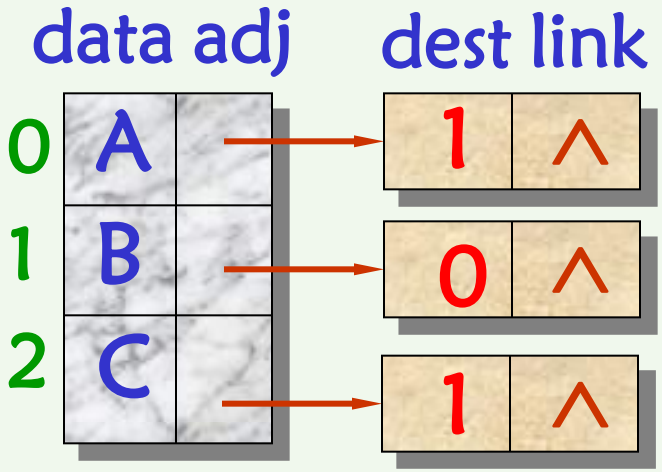
- 将与同一顶点相邻接的顶点链接成一个单链表.



2.有向图的邻接表和逆邻接表

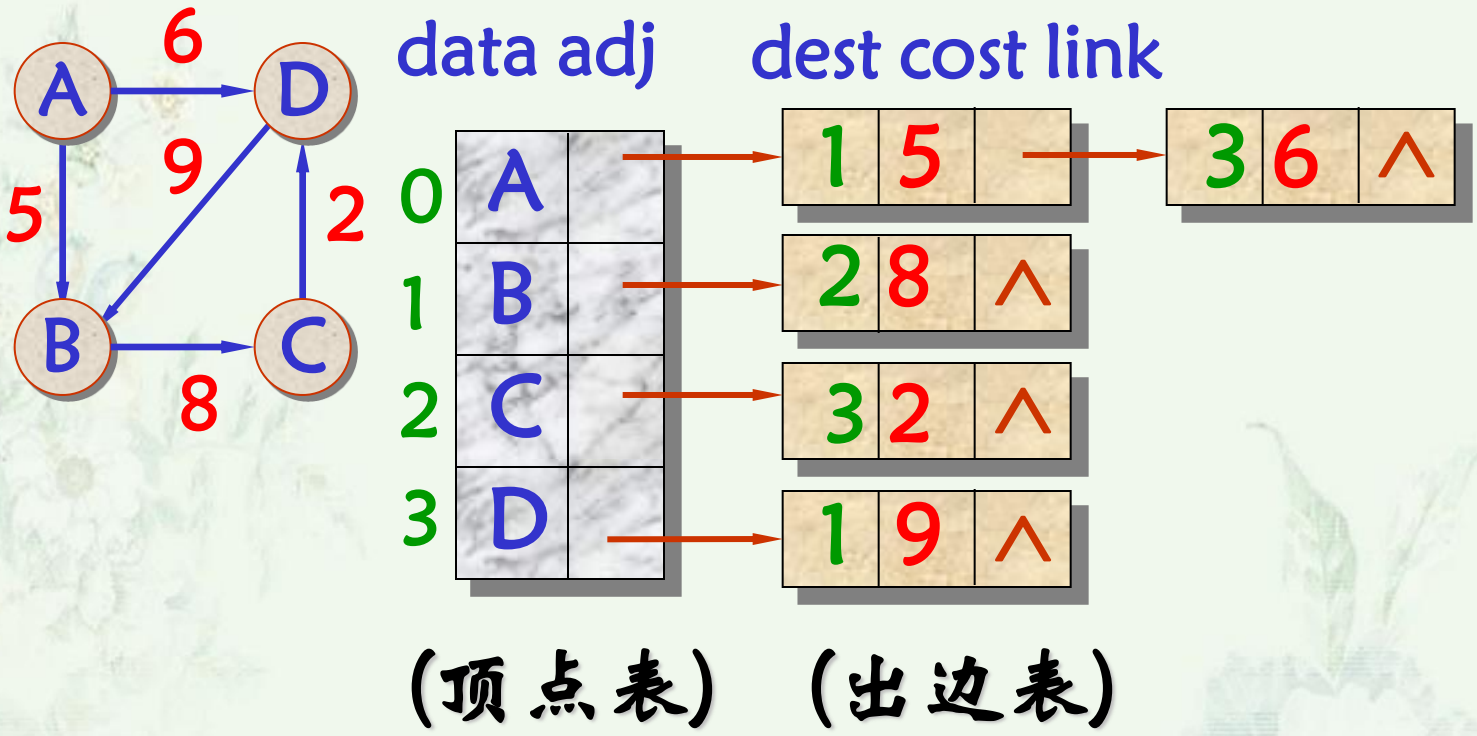


邻接表(出边表)



逆邻接表(入边表)

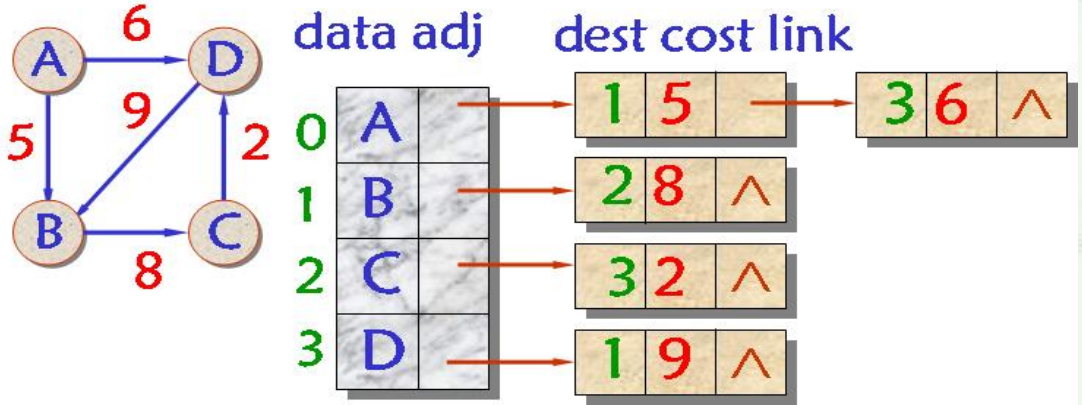
3.网络(带权图)的邻接表


A
D
B
C
6
9
5
2
8
data adj
dest cost link
0
A
1
B
2
C
3
D
1 5
3 6 ∧
2 8 ∧
3 2 ∧
1 9 ∧
(顶点表)
(出边表)

邻接表存储图的类定义

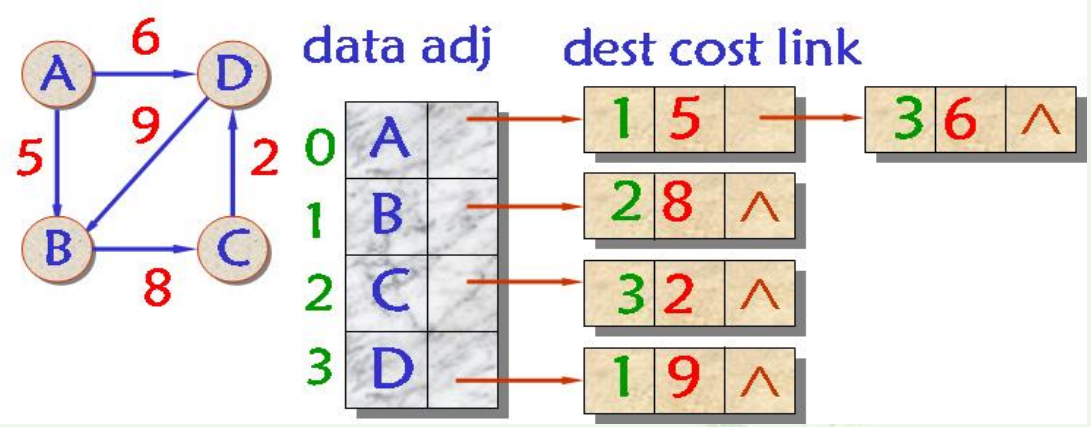
链表的结点结构类

```
template <class T, class E>
struct Edge {                                   //边结点的定义
    int dest;                                   //边的另一顶点位置
    E cost;                                     //边上的权值
    Edge<T, E> *link;                           //下一条边链指针
    Edge () {}                                  //构造函数
    Edge (int num, E weight)                    //构造函数
        : dest (num), cost(weight), link (NULL) { }
    bool operator != (Edge<T, E>& R) const
        { return dest != R.dest; }              //判边等否
};
```



顺序表结点结构类

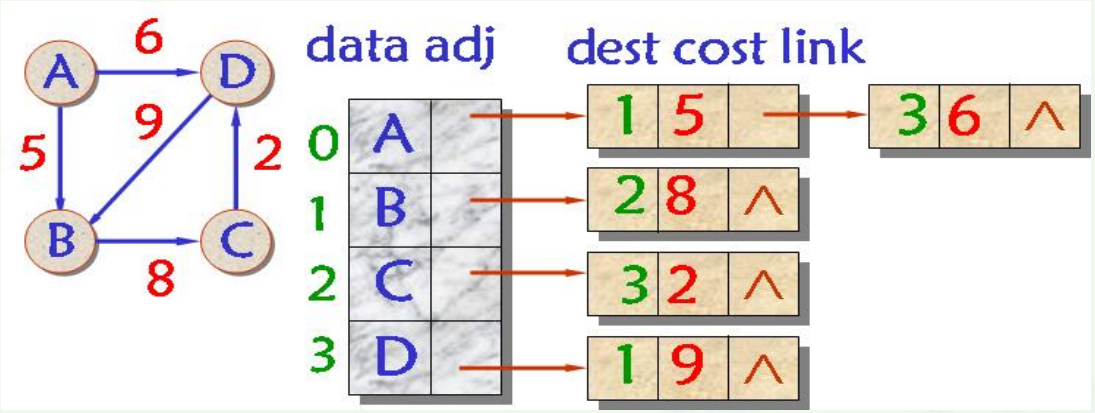
```
template <class T, class E>
struct Vertex {                          //顶点的定义
    T data;                              //顶点的名字
   Edge<T, E> *adj;                      //边链表的头指针
};
```



邻接表类

```
template <class T, class E>
class GraphInk : public Graph<T, E> {    //图的类定义
friend istream& operator >> (istream& in,
        GraphInk<T, E>& G);              //输入
friend ostream& operator << (ostream& out,
        GraphInk<T, E>& G);              //输出
```

```
private:
   Vertex<T, E> *NodeTable;
      //顶点表 (各边链表的头结点)
   int getVertexPos (const T vertx) {
      //给出顶点vertex在图中的位置
      for (int i = 0; i < numVertices; i++)
         if (NodeTable[i].data == vertx) return i;
      return –1;
   }
```

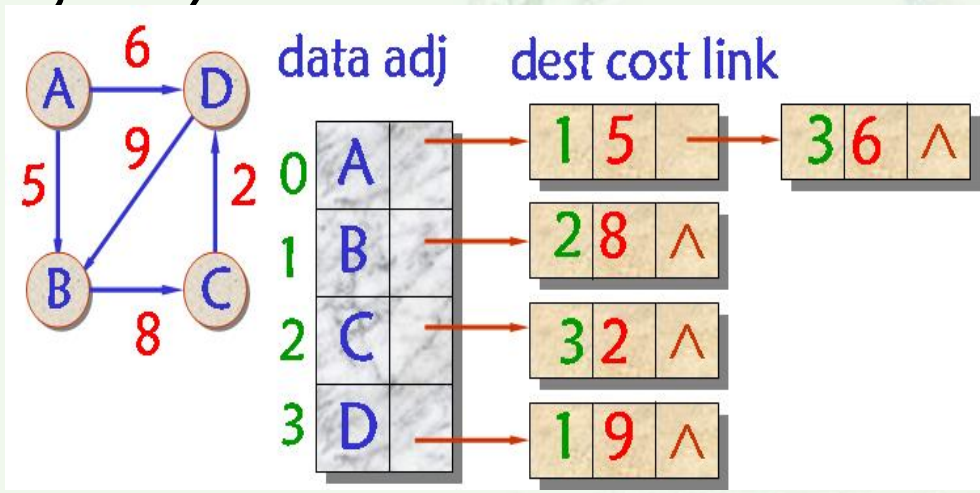



```
public:
   GraphInk (int sz = DefaultVertices);  //构造函数
   ~GraphInk();                          //析构函数
```

```
T getValue (int i) {                    //取顶点 i 的值
    return (i >= 0 && i < NumVertices) ?
          NodeTable[i].data : 0;
}
E getWeight (int v1, int v2);           //取边(v1,v2)权值
bool insertVertex (const T& vertex);  // 插入顶点
bool removeVertex (int v);   // 删除顶点
bool insertEdge (int v1, int v2, E cost); //插入边
bool removeEdge (int v1, int v2); // 删除边
int getFirstNeighbor (int v);   // 获取v的第一个邻接点
int getNextNeighbor (int v, int w);      // 获取w后的邻接点
void CreateNodeTable(void);  //  建立邻接表结构
};
```

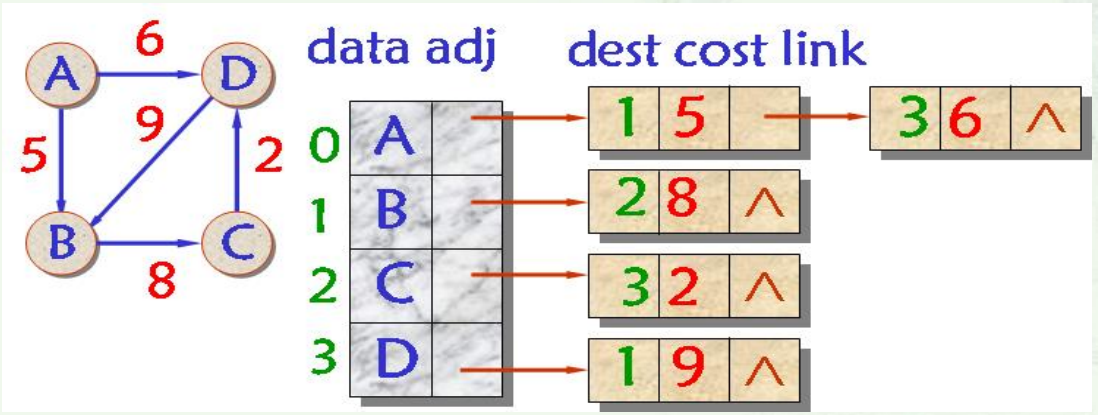
构造函数

```
template <class T, class E>
GraphInk<T, E>::GraphInk (int sz) {
//构造函数：建立一个空的邻接表
    maxVertices = sz;
    numVertices = 0;  numEdges = 0;
    NodeTable=new Vertex<T,E>[maxVertices];//创建顶点表数组
    if (NodeTable == NULL)
      { cerr << "存储分配错！ " << endl;  exit(1); }
    for (int i = 0; i < maxVertices; i++)
      NodeTable[i].adj = NULL;
};
```



析构函数

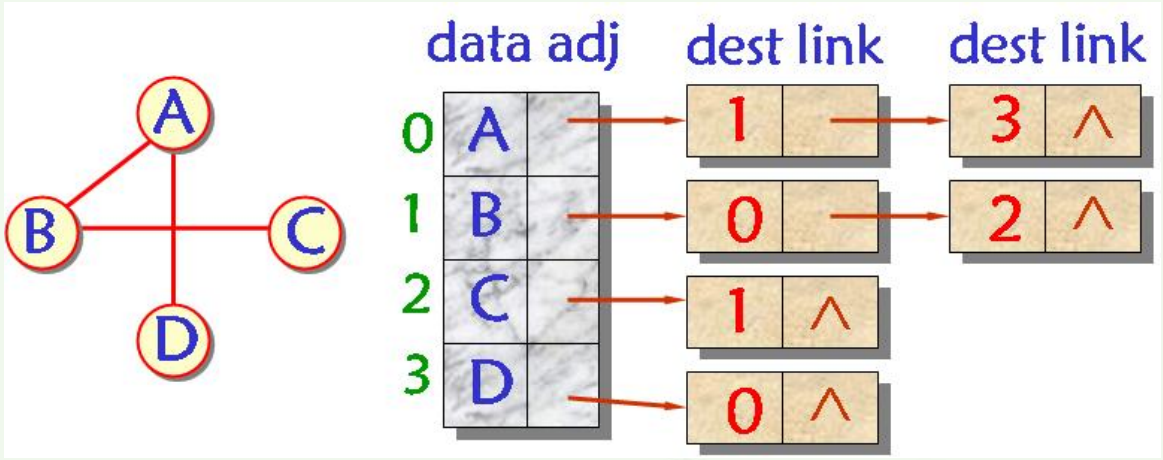
```
template <class T, class E>
GraphInk<T, E>::~GraphInk() {
//析构函数：删除一个邻接表
    for (int i = 0; i < numVertices; i++ ) {
        Edge<T, E> *p = NodeTable[i].adj;
        while (p != NULL)
        {  NodeTable[i].adj = p->link;
            delete p;  p = NodeTable[i].adj;
        }
    }
    delete [ ]NodeTable;                    //删除顶点表数组
};
```

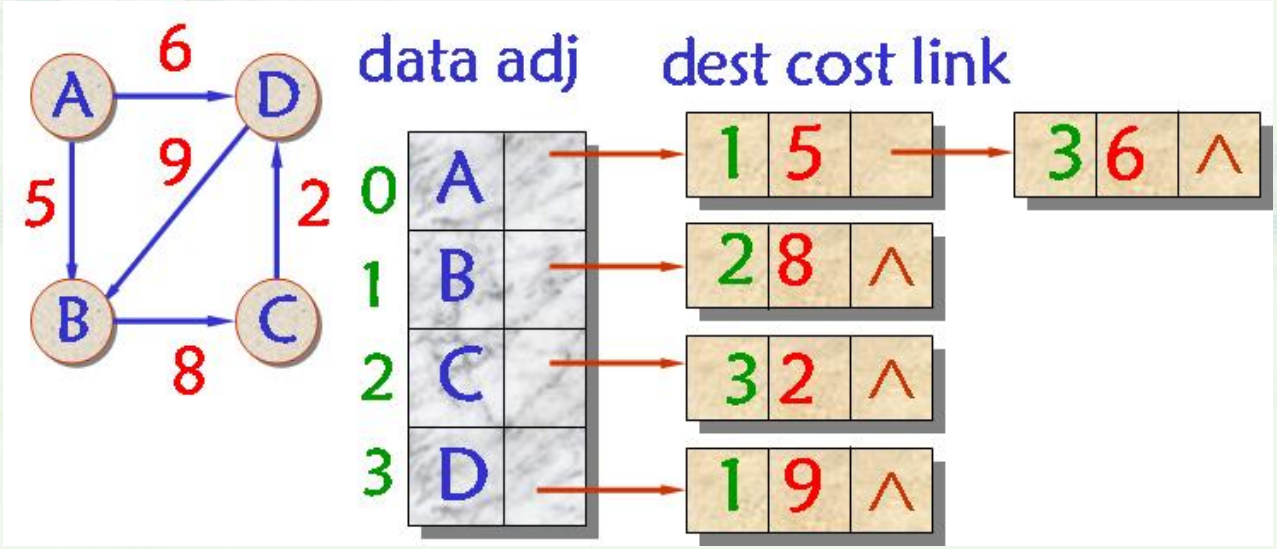


邻接表建立方法

邻接表建立算法

- 邻接表结构的分析
- 输入的组织





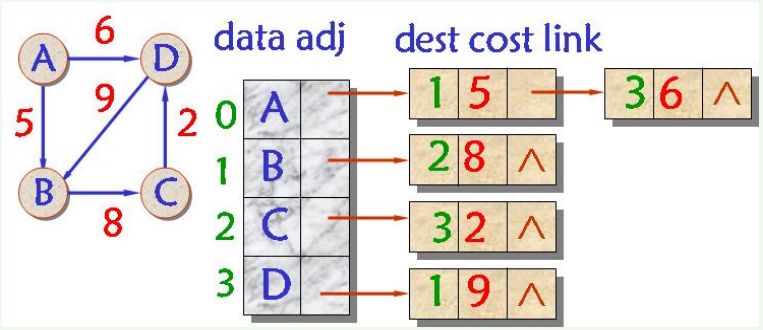
邻接表存储结构的实现算法

- 在输入数据前，顶点表NodeTable[ ]全部初始化，即将第i个结点的数据存入NodeTable[i].data中；
- 接着，把所有第i个结点的邻接点链接成一个单链表.
- 方法：在输入数据时，每输入一条边<i, k>，就需要建立一个边结点，并将它链入相应边链表中。

```
Edge<T, E> * p = new  Edge<T, E>;
p->dest=k;        //建立边结点，dest域赋为 k
p->cost=?
p->link = NodeTable[i].adj;
NodeTable[i].adj = p;      //头插入建链
```

邻接表的建立算法

```
template <class T, class E>
void GraphInk<T, E>::CreateNodeTable(void); // 建立邻接表结构
{   int n,i,j,m;     Edge<T, E> *p;
    cin>>n; //结点个数
    for(i=1;i<=n;i++)
    {
        NodeTable[i].adj=0; // 预设为空链
        cin>>NodeTable[i].data; //输入结点值
        cin>>m;  //每个结点的邻接点个数
        for(j=0;j<m;j++)
        {   p = new Edge<T, E>; // 生成一个新结点
            cin>>p->dest;      //建立边结点，输入结点值到dest域
            //带权图则多加一个权值的输入 cin>>p->cost;
            p->link = NodeTable[i].adj;
            NodeTable[i].adj = p;     //头插入建链
        }
    }
}
```



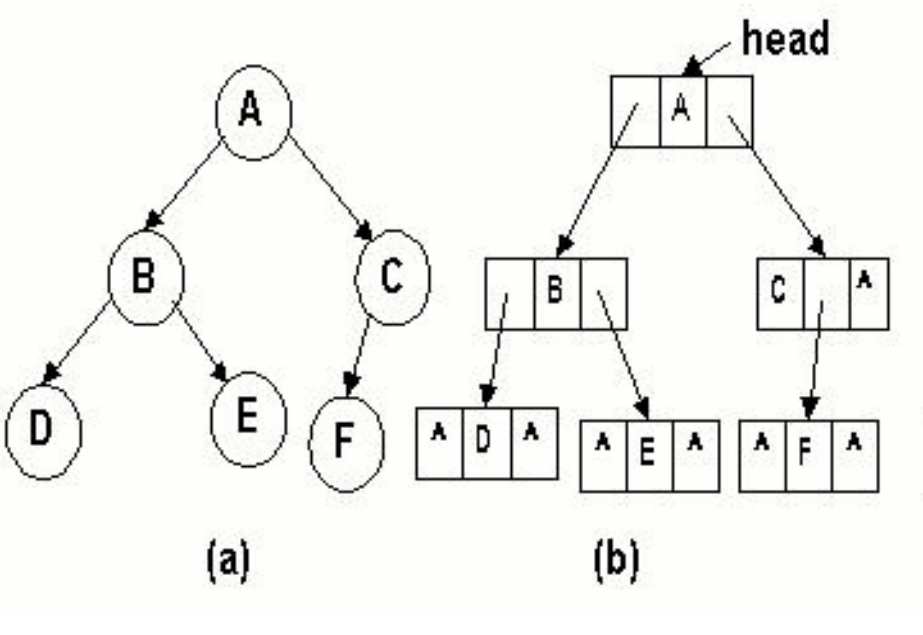
8.3 图的遍历

- 定义

- 图的遍历方法：

  - 深度优先 DFS
  - 广度优先 BFS

二叉树前序遍历算法的回顾

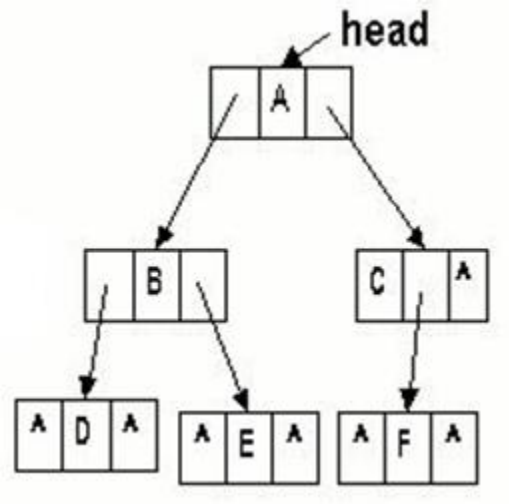


存储结构

```
struct treenode
{
    int data;
    struct treenode *child[2];
};
```

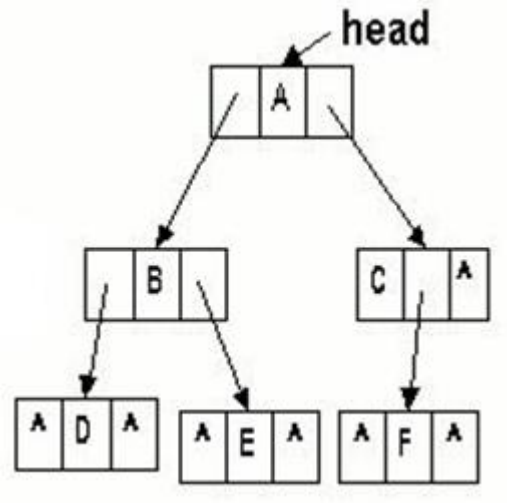
前序遍历算法

```
void DFS(struct treenode *root)
{
  int i;

  if (root!=0 )
  {
     cout<<root->data<<"   ";
      for (i=0;i<2;i++)
              DFS(root->child[i]);

      //DFS(root->child[0]);
      //DFS(root->child[1]);

  }

}
```



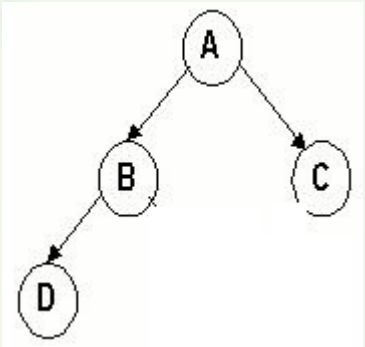
深度优先遍历算法

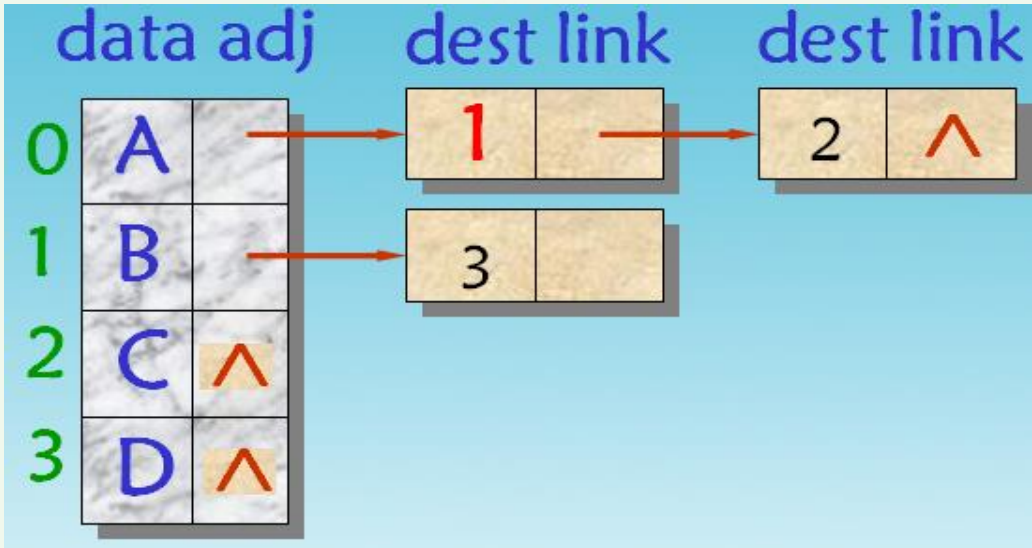
```
void DFS(struct node *root)
{  int i;

   if (root!=0 )
   {   cout<<root->data<<"   ";
       i=0;
      while ( i<2)
      {
           DFS( root->child[i] );
           i++;  // 下一个孩子
       }
     //DFS(root->child[0]);
     //DFS(root->child[1]);
  }
}
```



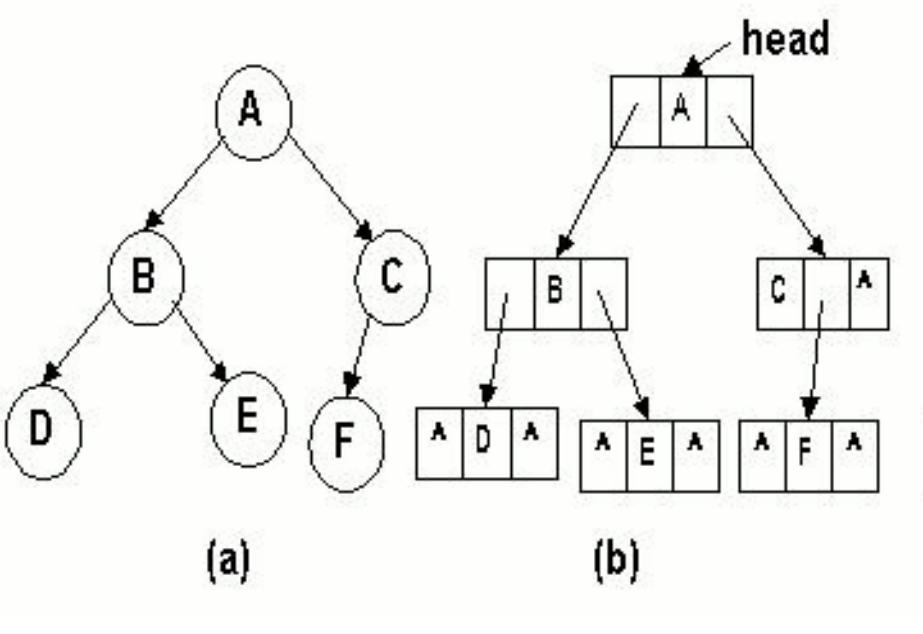
在邻接表存储结构下的实现算法：

```
void  DFS( int v)
{
   cout<<NodeTable[v].data;//访问v结点
    p=NodeTable[v].adj;
   while(p!=NULL)   // 找出孩子逐个进行递归调用
   {
     DFS( p->dest);
     p=p->link;//下一个孩子
   }
} // DFS
```



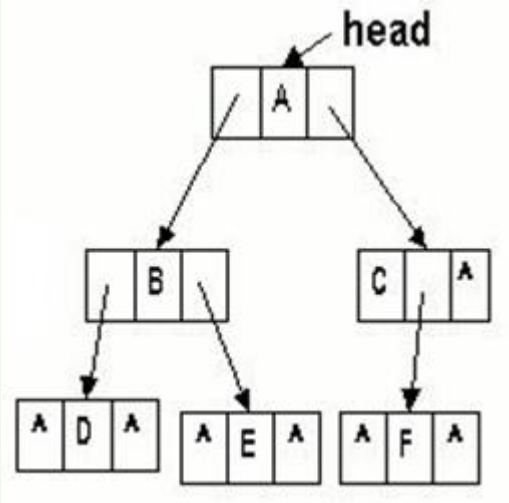


层次遍历算法



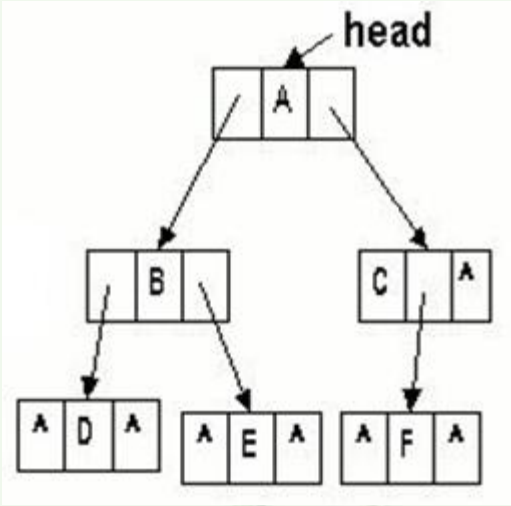
层次遍历算法

```
void BFS(struct treenode *root)
{
     queue<struct treenode *> qu;
     struct treenode *temp;

    qu.push(root);
     while(!qu.empty())
     {
          temp=qu.front();
          qu.pop();
          cout<<temp->data<<"  ";

          if (temp->child[0]!=0)
                    qu.push(temp->child[0]);

          if (temp->child[1]!=0)
                    qu.push(temp->child[1]);

        // 可以改写为 for 循环
     }
}
```



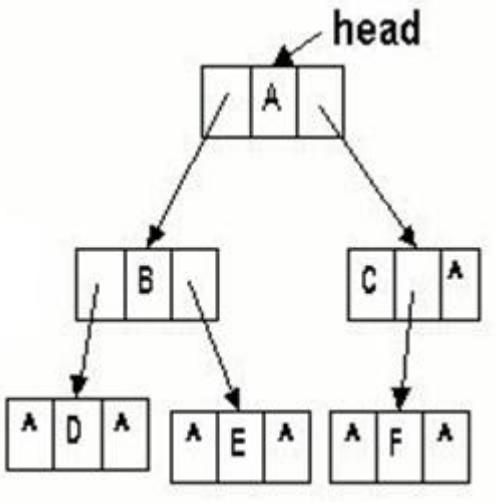
层次遍历算法

```
void BFS(struct treenode *root)
{
    queue<struct treenode *> qu;
    struct treenode *temp;

    qu.push(root);
    while(!qu.empty())
    {
        temp=qu.front();
        qu.pop();
        cout<<temp->data<<"   ";

         for(i=0;i<2;i++)
            if (temp->child[i]!=0)
                qu.push(temp->child[i]);
    }
}
```



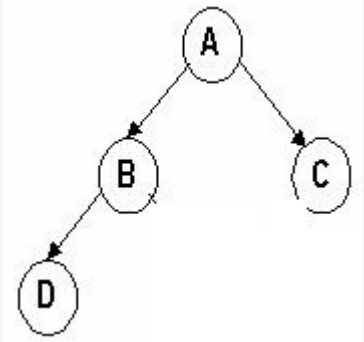
广度优先遍历算法

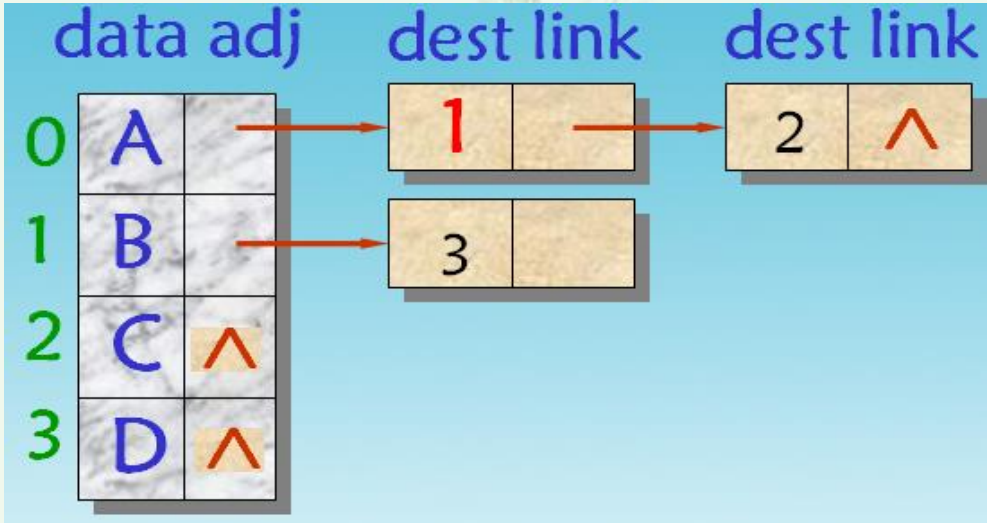
```
void BFS(struct node *root)
{   queue<struct node *> qu;
    struct node *temp;

    qu.push(root);
    while(!qu.empty())
    {     temp=qu.front();
          qu.pop();
          cout<<temp->data<<"  ";
          i=0;
          while ( i<2 )
          {    if (temp->child[i]!=0)
                     qu.push(temp->child[ i ] );
               i++; // 下一个孩子
          }
    }
}
```



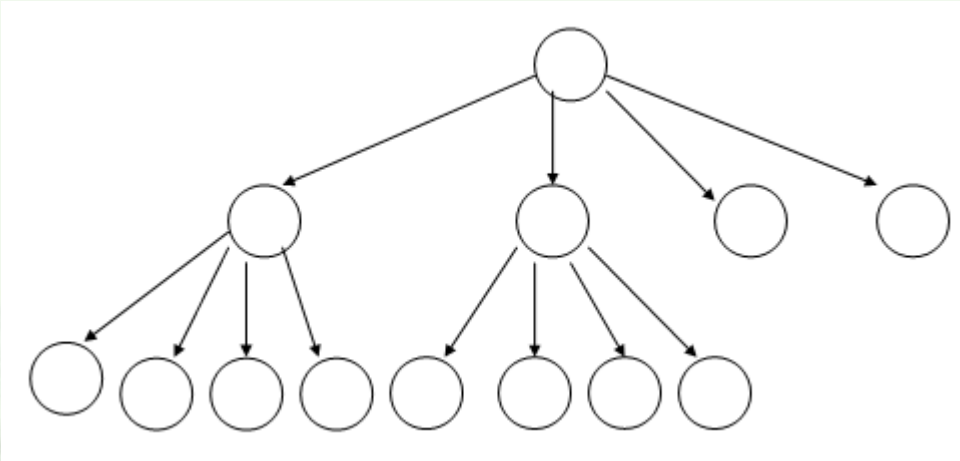
在邻接表存储结构下的实现算法：

```
void  BFS()
{  int v;
   queue<int> qu; //定义一个队列qu
   v=0;   // 根结点的下标为0
   qu.push(v); //v入列
   while (!qu.empty())
   {   v=qu.front(); qu.pop(); //出列
       cout<<NodeTable[v].data; //访问v结点
       p=NodeTable[v].adj; //p指向v结点对应邻接单链表的链头
       while (p!=NULL)
       {  //依次将v结点的孩子入列,以依次实施层次遍历
          qu.push(p->dest);
          p=p->link;//下一个孩子
       }//while
    }//if
  }//while
}// BFS
```





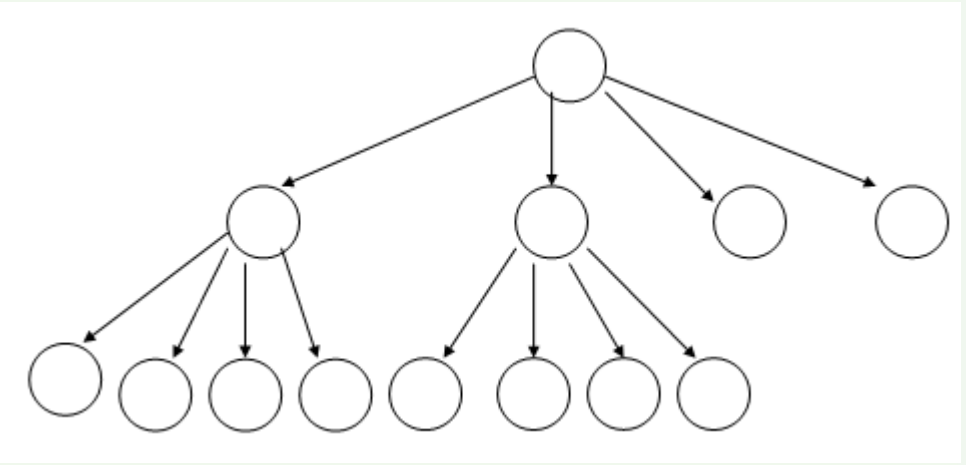
四叉特征树



存储结构

```
struct treenode
{
    int data;
    struct treenode *child[4];
};
```

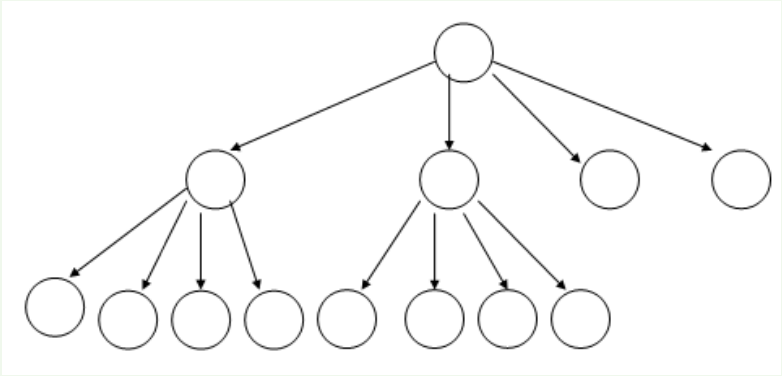
前序遍历算法



```
void DFS(struct treenode *root)
{
  int i;

  if (root!=0 )
  {
     cout<<root->data<<"   ";

      for (i=0;i<4;i++)
              DFS(root->child[i]);   ////////////

  }

}
```

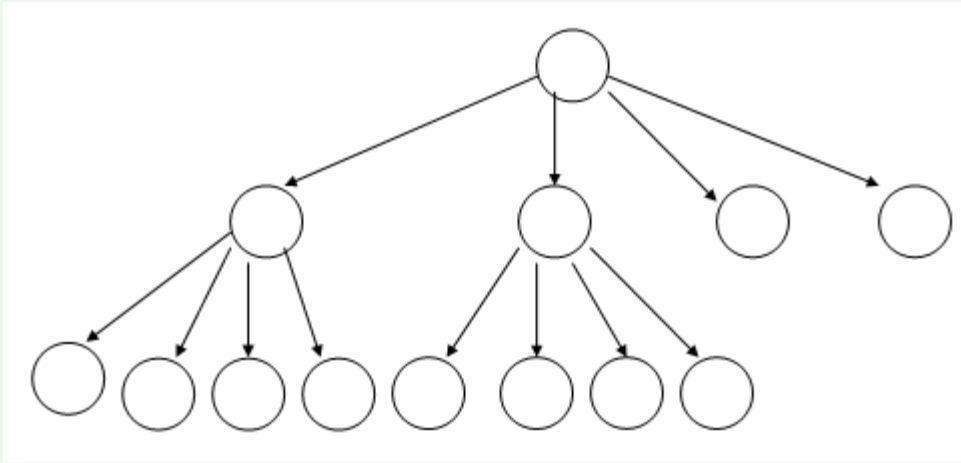
深度优先遍历算法

```
void DFS(struct node *root)
{  int i;

   if (root!=0 )
   {   cout<<root->data<<"   ";
       i=0;
      while ( i<4)
      {
           DFS( root->child[ i ] );
          i++;
       }

   }
}
```



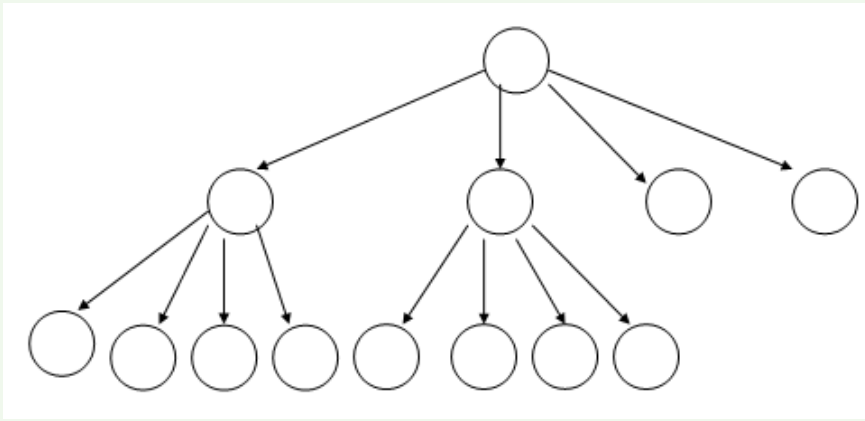
层次遍历算法

```
void BFS(struct treenode *root)
{
      int i;
      queue<struct treenode *> qu;
      struct treenode *temp;

     qu.push(root);
      while(!qu.empty())
      {
            temp=qu.front();
            qu.pop();
            cout<<temp->data<<"   ";

            for (i=0;i<4;i++)
              if (temp->child[i]!=0)
                        qu.push(temp->child[i]);
      }
}
```



广度优先遍历算法

```
void BFS(struct node *root)
{   queue<struct node *> qu;
    struct node *temp;

    qu.push(root);
    while(!qu.empty())
    {    temp=qu.front();
         qu.pop();
         cout<<temp->data<<"  ";
          i=0;
          while ( i<4 )
          {    if (temp->child[i]!=0)
                  qu.push(temp->child[ i ] );
               i++;
          }
    }
}
```



八叉特征树

存储结构

```
struct treenode
{
    int data;
    struct treenode *child[8];
};
```

八叉特征树的遍历方法

- 前序遍历方法
- 层次遍历方法

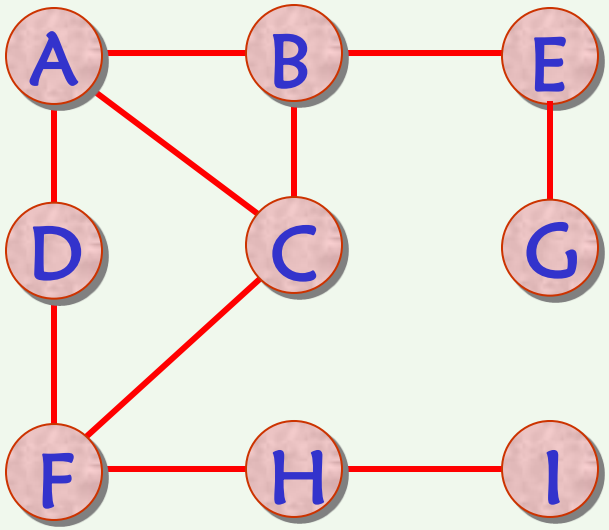
图的遍历

- 深度优先
- 广度优先

深度优先 DFS

- 在访问图中某一起始顶点 v 后，由 v 出发，访问它的任一邻接顶点 w_1；再从 w_1 出发，访问与 w_1 邻接但还没有访问过的顶点 w_2；然后再从 w_2 出发，进行类似的访问，… 如此进行下去，直至到达所有的邻接顶点都被访问过的顶点 u 为止。
- 接着，退回一步，退到前一次刚访问过的顶点，看是否还有其它没有被访问的邻接顶点。如果有，则访问此顶点，之后再从此顶点出发，进行与前述类似的访问；如果没有，就再退回一步进行搜索。
- 重复上述过程，直到连通图中所有顶点都被访问过为止。

深度优先遍历的示例



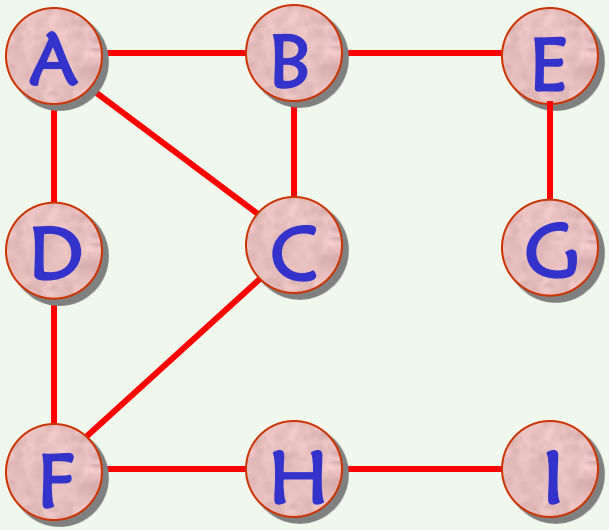
深度优先遍历过程

广度优先 BFS

- 在访问了起始顶点 v 之后，由 v 出发，依次访问 v 的各个未被访问过的邻接顶点 $w_1, w_2, \cdots, w_t$，然后再顺序访问 $w_1, w_2, \cdots, w_t$ 的所有还未被访问过的邻接顶点。再从这些访问过的顶点出发，再访问它们的所有还未被访问过的邻接顶点，… 如此做下去，直到图中所有顶点都被访问到为止。

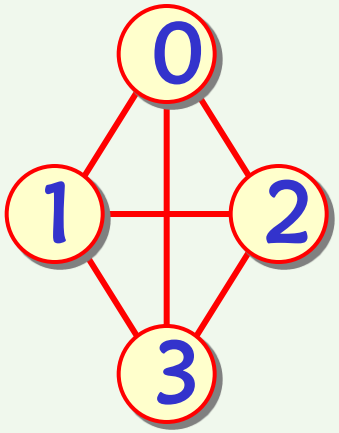
- 广度优先遍历是一种分层的搜索过程，每向前走一步可能访问一批顶点，不像深度优先遍历那样有往回退的情况。
- 因此，广度优先遍历不是一个递归的过程。

- 广度优先遍历的示例



广度优先遍历过程

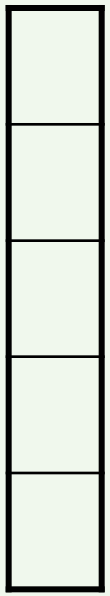
图的遍历方法的问题发现



1.如何解决重复访问?

图遍历的问题分析

- 如何解决重复访问?
- 可设置一个标志顶点是否被访问过的辅助数组 visited[ ]。



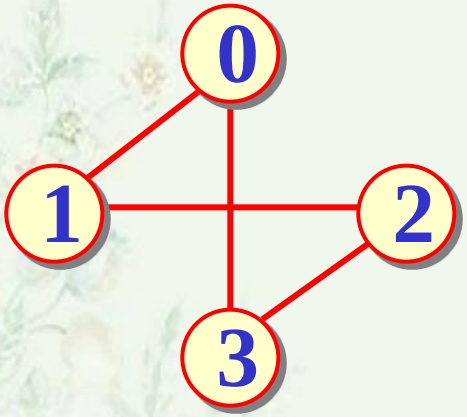
图的遍历方法的问题发现

2.各结点的叉数不统一

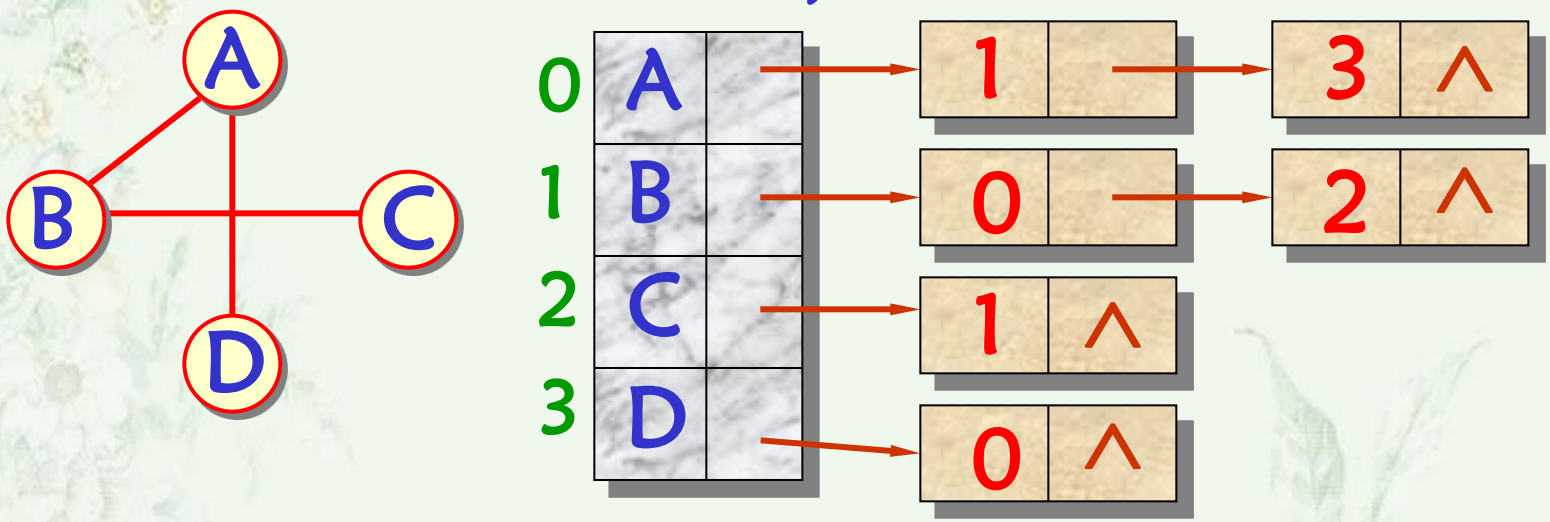
- 存储结构的问题

```
struct treenode
{
    int data;
    struct treenode *child[8]; ///问题所在
};
```

- 找孩子——→改变为找邻接点



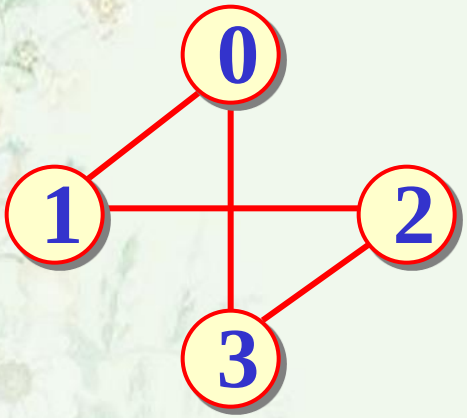
$$A.edge = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$


A
B
C
D
data adj
dest link
dest link
0
A
1
3
^
1
B
0
2
^
2
C
1
^
3
D
0
^

如何找结点i的邻接点

- 存储结构

邻接矩阵存储结构

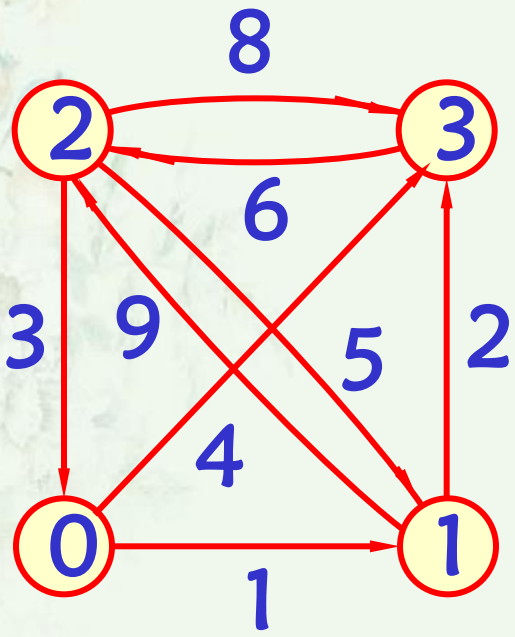


$$A.edge = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

找出结点i的所有邻接点

```
for (int col = 0; col < numVertices; col++)
{
    if ( Edge[i][col]==1 )
    {
        ……
    }
}
```

带权图

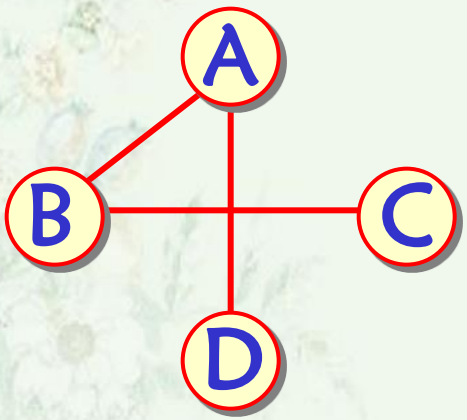


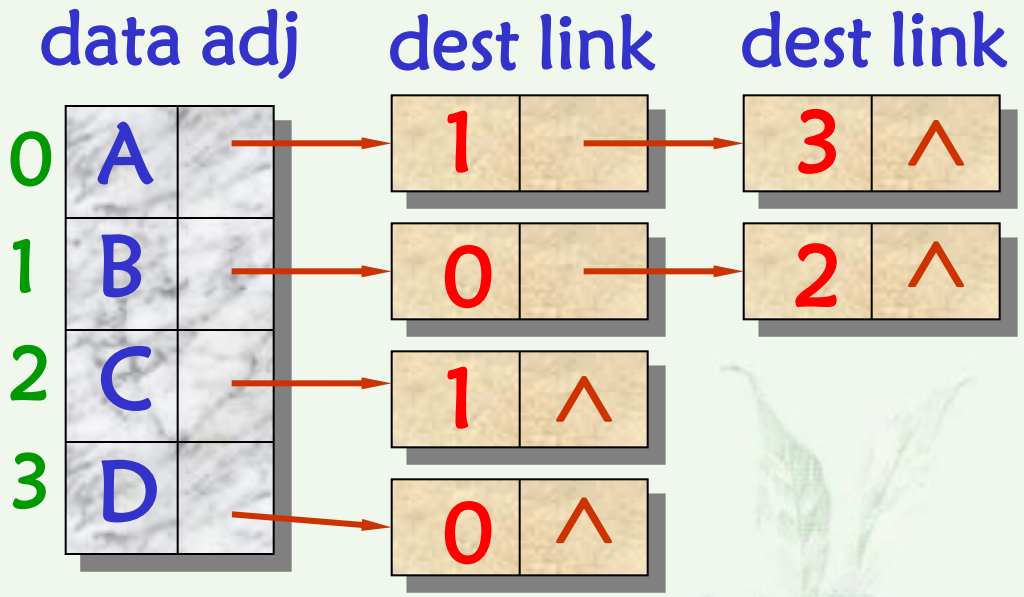
$$A.edge = \begin{bmatrix} 0 & 1 & \infty & 4 \\ \infty & 0 & 9 & 2 \\ 3 & 5 & 0 & 8 \\ \infty & \infty & 6 & 0 \end{bmatrix}$$

找出结点i的所有邻接点

```
for (int col = 0; col < numVertices; col++)
{
    if (( Edge[i][col]）&& Edge[i][col]<maxweight )
    {
        ……
    }
}
```

邻接表结构



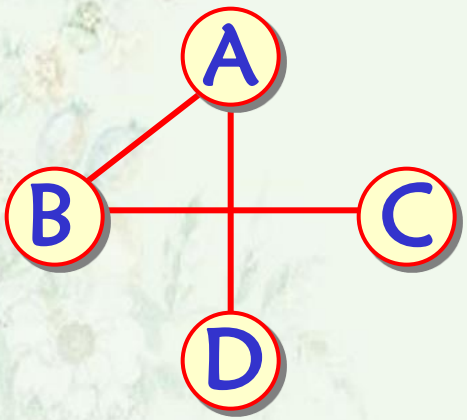


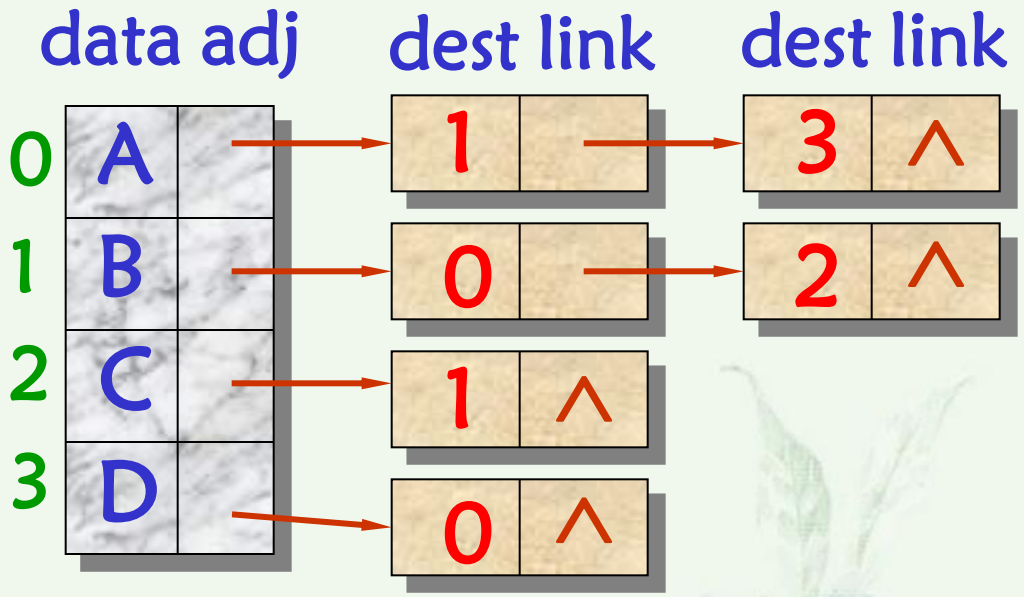
找出结点i的所有邻接点

```
Edge<T, E> *p = NodeTable[i].adj;
 while (p != NULL)
 {
        .....
        p = p->link;
 }
}
```

深度优先 DFS 算法的描述

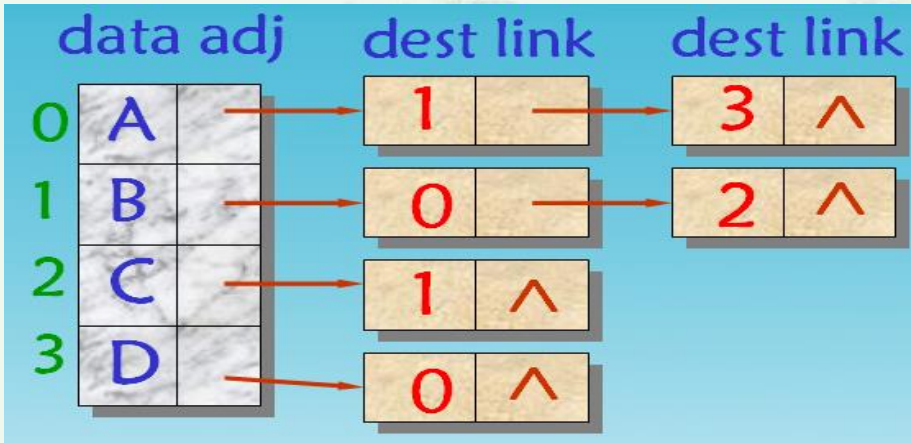
邻接表结构





在邻接表存储结构下的实现算法：

```
void GraphInk<T, E>:: DFS( int v)    // 私有函数
{   // 从顶点v出发，深度优先遍历遍历连通图 G
   visited[v] = true;
   cout<<NodeTable[v].data;//访问v结点
   p=NodeTable[v].adj;
   while(p!=NULL)   // 找出邻接点逐个进行递归调用
   {  // 对v的尚未访问的邻接顶点递归调用DFS
     if (!visited[p->dest]) DFS( p->dest);
     p=p->link; // 下一个邻接点
   }
}// DFS
```



深度优先遍历 公有函数

```
bool *visited;   // 成员属性      辅助数组
int numVertices ;    // 成员属性   表示结点个数

visited = new bool[n];     // 加入到GraphInk构造函数中
cin>> numVertices ;     //加入到GraphInk构造函数中

void GraphInk<T, E>:: dfs( )
{
   int i ,v0;

  for (i=0;i<numVertices;i++ ) visited[i]=false;

  cin>>v0; //输入深度优先遍历的出发点

   dfs(v0);  //调用深度优先的递归函数
}
```

程序的组织

```
void main()
{
    邻接表的建立;
    调用深度优先遍历公有成员函数
}
```

邻接矩阵存储的深度优先遍历

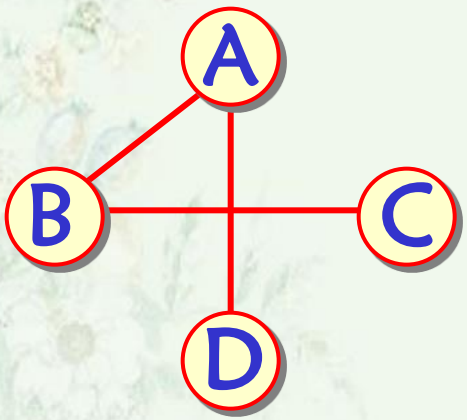
- int Edge[maxm][maxm];
- 邻接矩阵的输入
- 调用深度优先算法

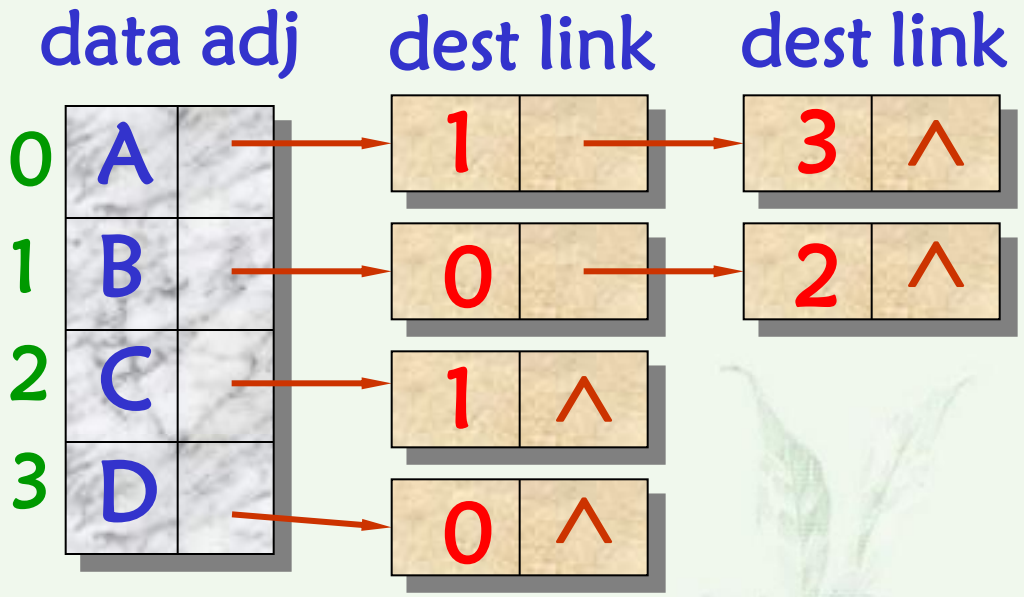
- 广度优先 BFS 算法的描述

四叉树的广度优先遍历算法

```
void BFS(struct node *root)
{   queue<struct node *> qu;
    struct node *temp;
    qu.push(root);
    while(!qu.empty())
    {   temp=qu.front();
        qu.pop();
        cout<<temp->data<<"  ";
        i=0;
        while ( i<4 )
        {   if (temp->child[i]!=0)
                qu.push(temp->child[ i ] );
            i++; // 下一个孩子
        }
    }
}
```

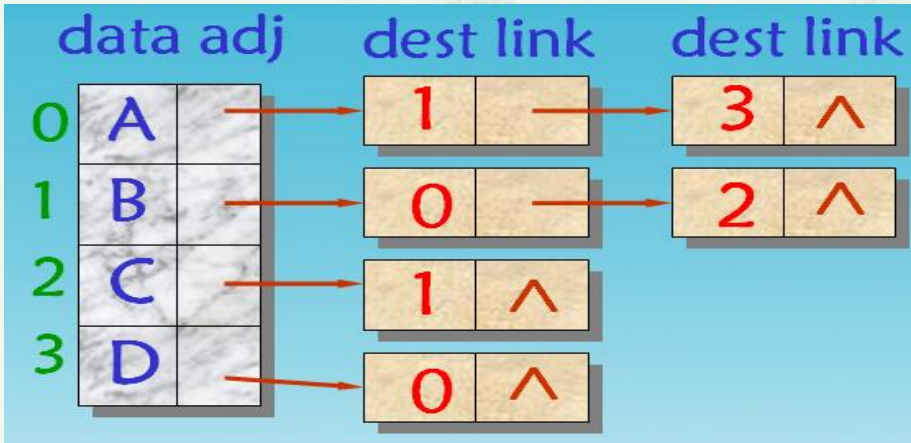
邻接表结构





在邻接表存储结构下的实现算法： // STL queue

```
void GraphInk<T, E>:: BFS( ) // 从顶点v出发，广度优先遍历遍历连通图 G
{  int  v;
   queue<int> qu; //定义一个队列qu
  for (int i=0;i<n;i++ ) visited[i]=false;  //设置未入列标志
   cin>>v;  //输入广度优先遍历的出发点
   qu.push(v);  // v入列
   visited[v] = true;   // 设置入列标志
   while (!qu.empty( ))
   {    v=qu.front( ); qu.pop( ); //出列
        cout<<NodeTable[v].data;  //访问v结点
        p=NodeTable[v].adj;  //p指向v结点对应邻接单链表的链头
        while (p!=NULL)
            {  //依次将v结点的邻接点入列,以依次实施层次遍历
               if (!visited[p->dest])   qu.push(p->dest);  // 没有进过队列的
               p=p->link;//下一个邻接点
            } //while
      } //if
  } //while
} // BFS
```



程序的组织

```
void main()
{
  邻接表的建立;
  调用广度优先遍历公有成员函数
}
```

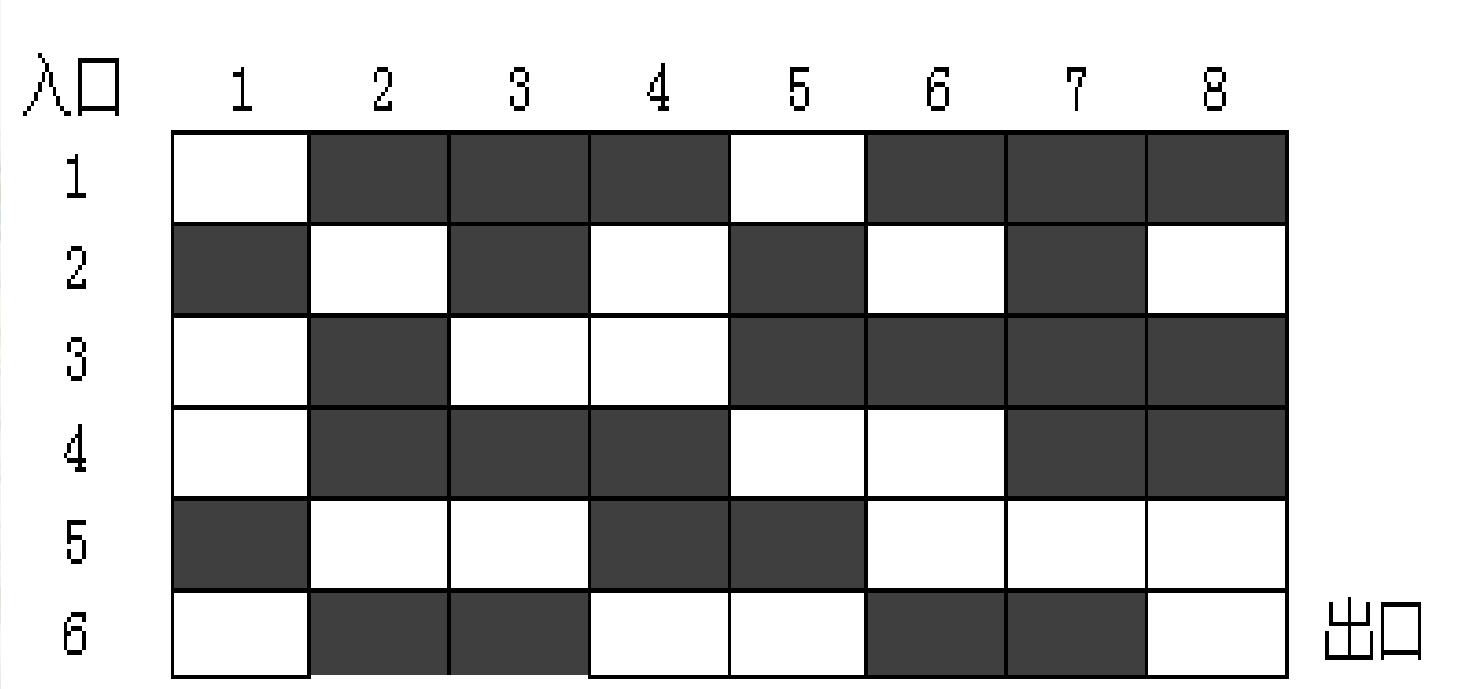
邻接矩阵存储的广度优先遍历

- int Edge[maxm][maxm];
- 邻接矩阵的输入
- 调用广度优先算法

图遍历算法的应用

- 符合状态转变的问题

迷宫问题



迷宫图中阴影部分是不通的路径，处于迷宫中的每个位置都可以向8个方向探索着按可行路径前进。假设出口位置在最右下角（6，8），入口在最左上角（1，1），要求设计寻找从入口到出口的算法。

问题一：迷宫的存储结构

- 用一个二维数组来存储迷宫，数组中的每个元素的值只取0或1，其中0表示此路可通，1表示此路不通。对于上例的迷宫可存储如图。

→Y
↓X

	1	2	3	4	5	6	7	8
1	0	1	1	1	0	1	1	1
2	1	0	1	0	1	0	1	0
3	0	1	0	0	1	1	1	1
4	0	1	1	1	0	0	1	1
5	1	0	0	1	1	0	0	0
6	0	1	1	0	0	1	1	0

图 3-15

搜索方向问题

但这样会导致迷宫中的每个位置可探索的情况就不一致，可分为：

（1）只有三个探索方向的位置：

如（1，1），探索的方向只有（1，2），（2，2），（2，1）；

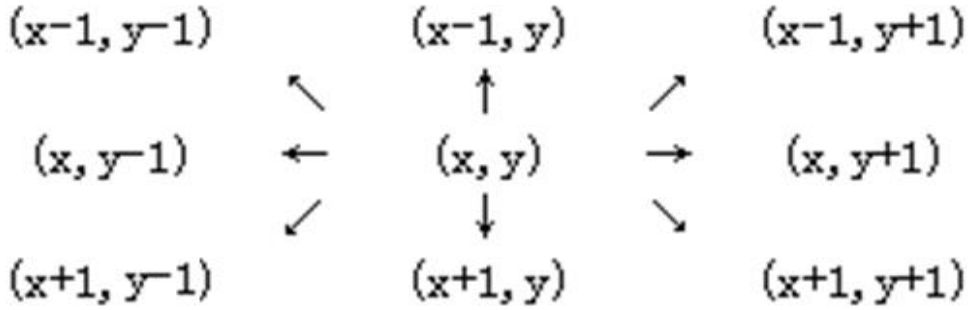
（2）有五个探索方向的位置：

如（3，1），探索的方向有（3，2），（4，2），（4，1），（2，1），（2，2）；

（3）有八个探索方向的位置：

如（3，2），探索的方向有（3，3），（4，3），（4，2），（4，1），（3，1），（2，1），（2，2），（2，3）；

探索路径的选择可描述为：



由于从当前位置（x,y）向上述八个方向探索，则可得到八个新位置，这八个新位置与当前位置的变化关系可用数组来存储：

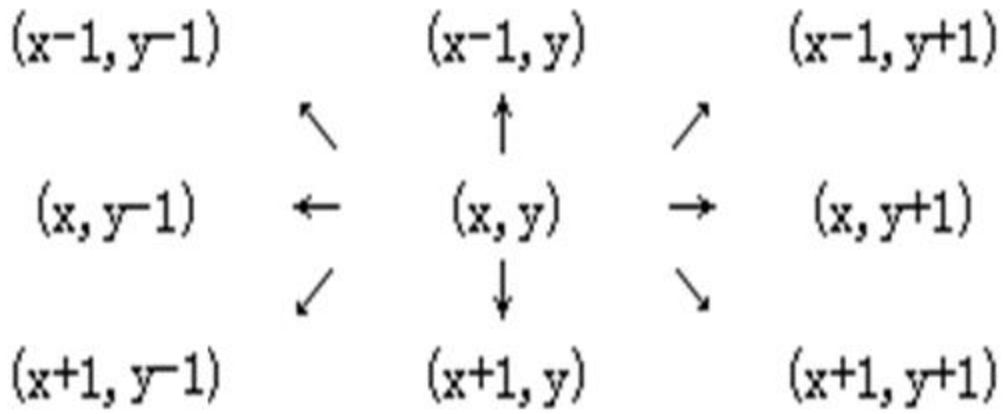
	Δx	Δy	说明
0	0	1	向→方向探索
1	1	1	向↘方向探索
2	1	0	向↓方向探索
3	1	-1	向↙方向探索
4	0	-1	向←方向探索
5	-1	-1	向↖方向探索
6	-1	0	向↑方向探索
7	-1	1	向↗方向探索

```
for ( loop=0 ; loop<8 ; loop++ )
    // 探索当前位置的8个相邻位置
     {
        x=x+move[loop].x;
        y=y+move[loop].y;
         ……
      }
```

深度优先遍历迷宫

- 找邻接点的问题

探索路径的选择可描述为：



	Δx	Δy	说明
0	0	1	向→方向探索
1	1	1	向↘方向探索
2	1	0	向↓方向探索
3	1	-1	向↙方向探索
4	0	-1	向←方向探索
5	-1	-1	向↖方向探索
6	-1	0	向↑方向探索
7	-1	1	向↗方向探索

- 重复访问的问题

问题：重复访问问题

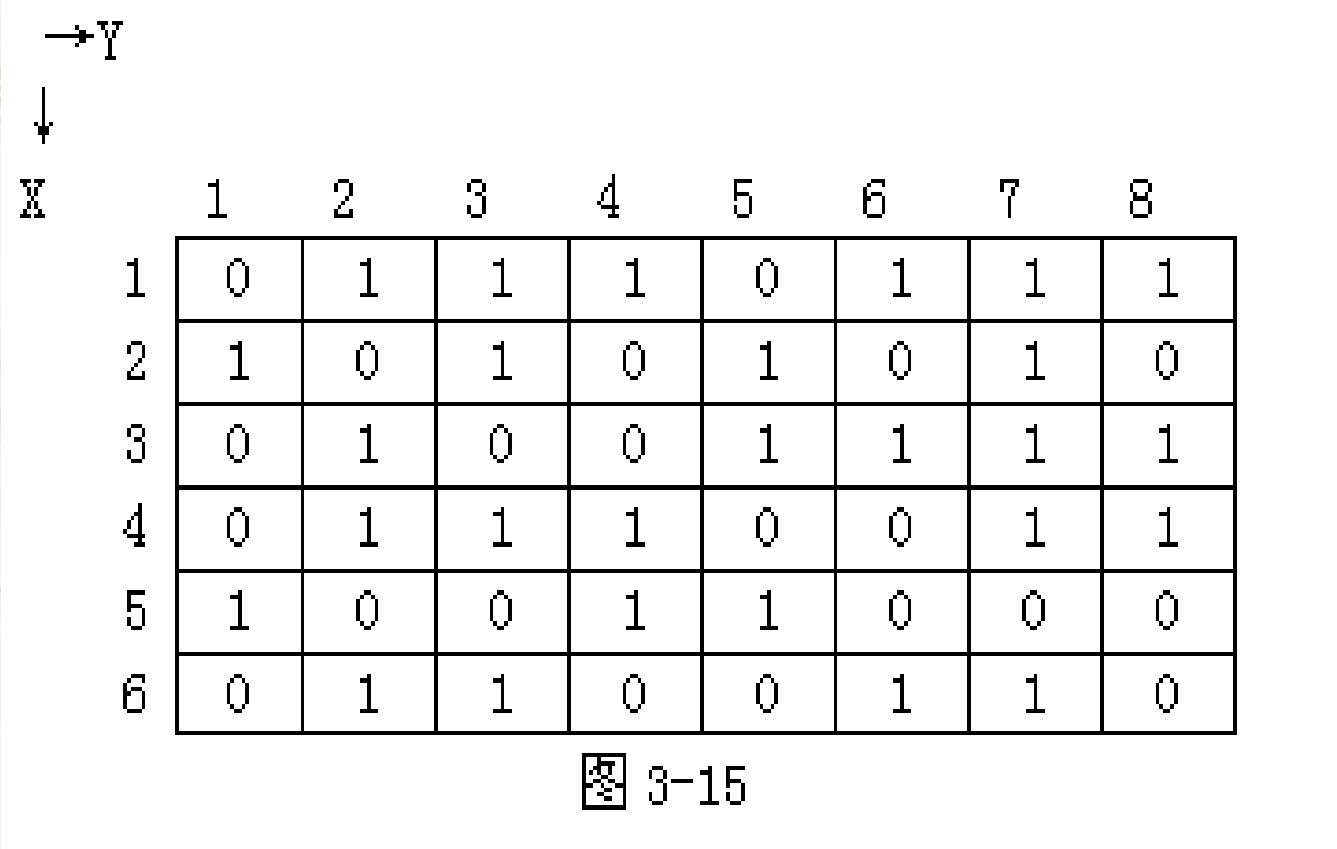


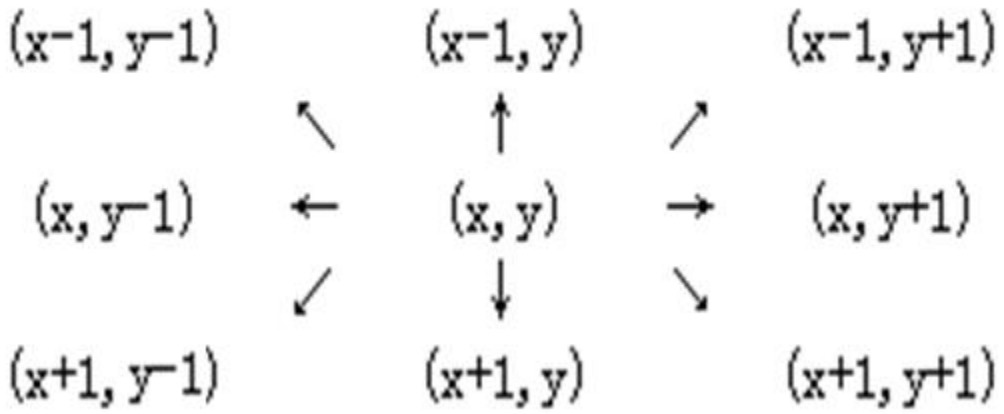
图 3-15

```
int maze[m+2][n+2];
int   MazePath( int x, int y )
{     int loop;
      Maze[x][y]=-1; // 标志入口位置已到达过
      for ( loop=0 ; loop<8 ; loop++ ) // 探索当前位置的8个相邻位置
       {
          x=x+move[loop].x; // 计算出新位置x位置值
          y=y+move[loop].y; // 计算出新位置x位置值
          if ((x==m)&&(y==n))  // 成功到达出口
              {     PrintPath( ); // 输出路径  该函数请自行完成
                    Restore(Maze); // 恢复迷宫  该函数请自行完成
                    return ( 1 ) ; // 表示成功找到路径
               }
          if ( Maze[x][y]= = 0 ) // 新位置是否可到达
              {    .....   // 保存该点坐标,以便以后输出路径
                   MazePath(x,y );
              }
        }
        }
       return(0); // 表示查找失败，即迷宫无路径
     } // MazePath
```

广度优先遍历迷宫

- 找邻接点的问题

探索路径的选择可描述为：



	Δx	Δy	说明
0	0	1	向→方向探索
1	1	1	向↘方向探索
2	1	0	向↓方向探索
3	1	-1	向↙方向探索
4	0	-1	向←方向探索
5	-1	-1	向↖方向探索
6	-1	0	向↑方向探索
7	-1	1	向↗方向探索

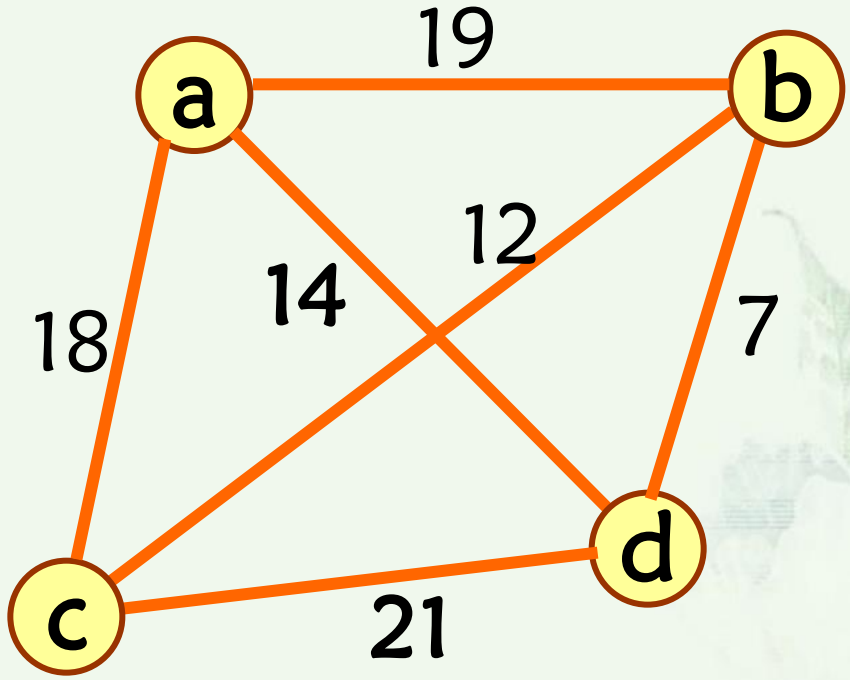
- 重复访问的问题

```
int maze[m+2][n+2];
int   MazePath( )
{   queue<DataType> q;  // 定义一个容量为m*n的队列    DataType { int x, int y , int pre; };
    DataType  Temp1,Temp2;       int x, y,loop;
     Temp1.x=1; Temp1.y=1; Temp1.pre=-1; Maze[1][1]=-1;  // 标志入口位置已到达过
     q.push( Temp1 );  // 将入口位置入列
      while ( !q.empty( ) )  // 队列非空，则反复探索
      {   Temp2=q.front( ); q.pop( );  // 队头元素出列
          for ( loop=0 ; loop<8 ; loop++ )  // 探索当前位置的8个相邻位置
          {   x=Temp2.x+move[loop].x;  // 计算出新位置x位置值
              y=Temp2.y+move[loop].y; // 计算出新位置x位置值
              if ((x==m)&&(y==n))   // 成功到达出口
               {      PrintPath( q ); // 输出路径   该函数请自行完成
                      Restore(Maze);  // 恢复迷宫   该函数请自行完成
                      return ( 1 ) ; // 表示成功找到路径
                }
              if ( Maze[x][y]= = 0 )  // 新位置是否可到达
               {     Temp1.x=x;  Temp1.y=y;
                     //Temp1.pre=q.front;//设置到达新位置的前趋位置
                     Maze[x][y]=-1;  //标志该位置已到达过
                     q.push(Temp1);// 新位置入列
               }
           }
      }
     return(0);  // 表示查找失败，即迷宫无路径
} // MazePath
```

8.4 最小生成树

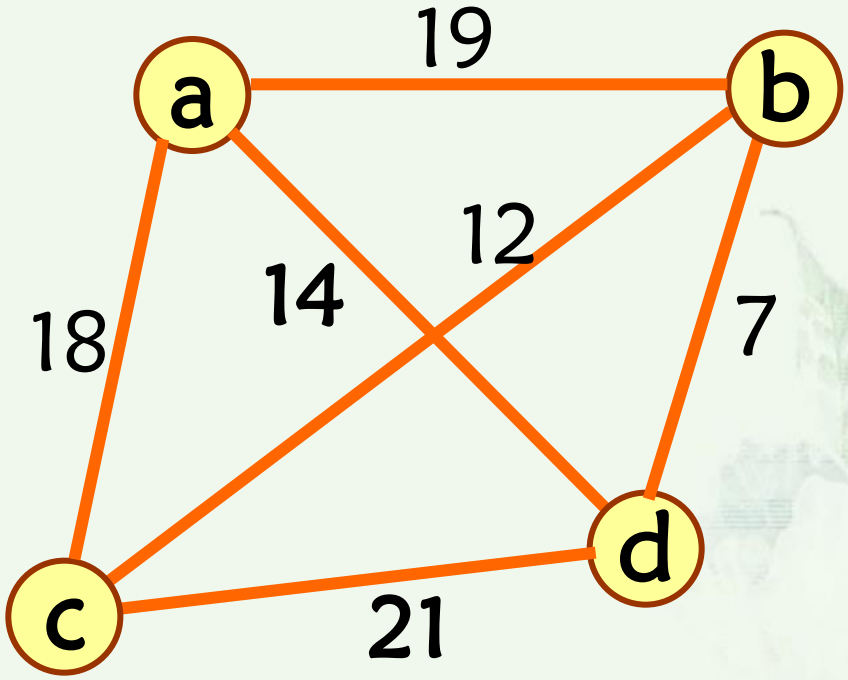
问题描述：

假设要在 n 个城市之间建立通讯联络网，则连通 n 个城市只需要修建 n-1条线路，如何在最节省经费的前提下建立这个通讯网？

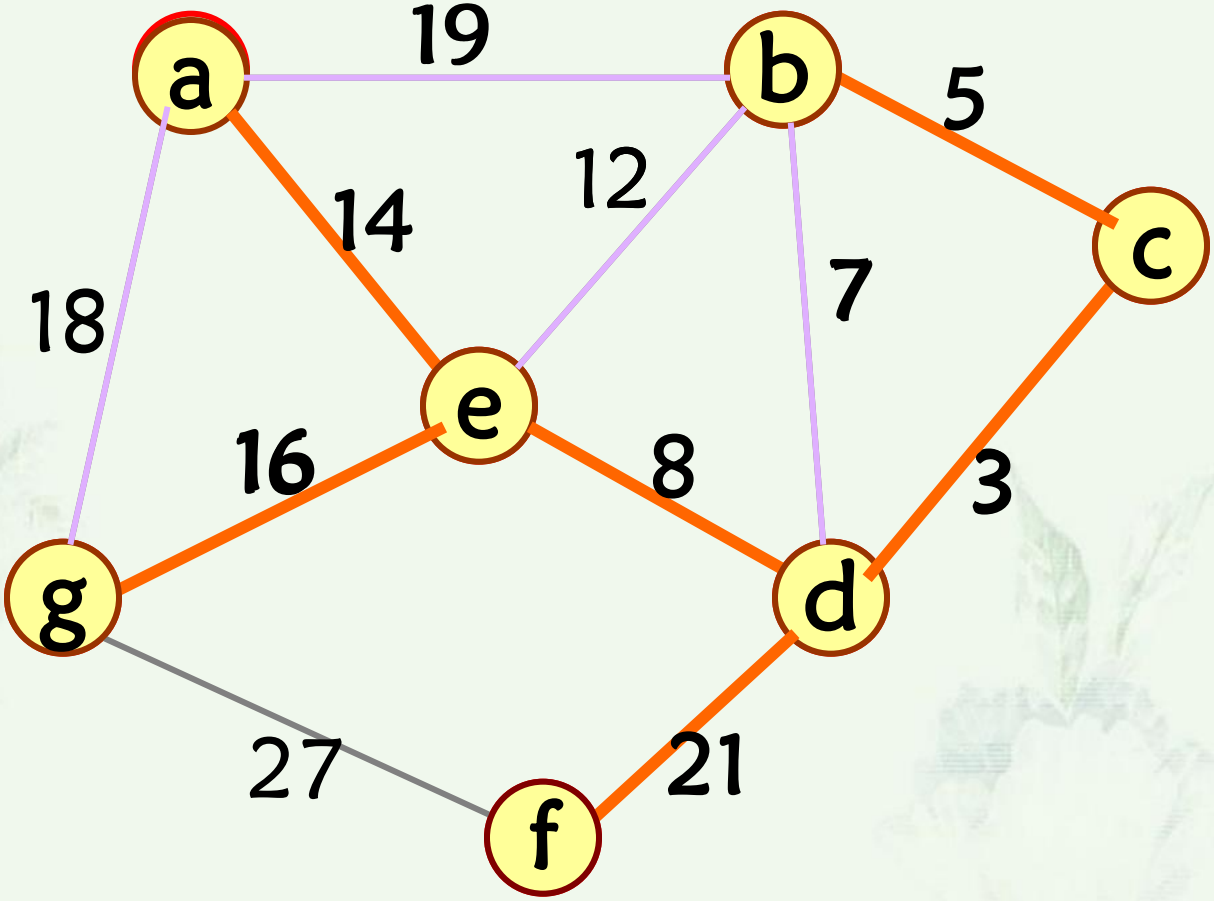


该问题等价于：

构造网的一棵最小生成树，即：　在 e 条带权的边中选取 n-1 条边（不构成回路），使“权值之和”为最小，即最小生成树。



例如：



算法描述:

构造非连通图 ST=(V,{});

k = i = 0; //k 统计已选中的边数

while (k<n-1)

{

  ++i;

  检查边集 E 中第 i 条权值最小的边(u,v);

  若(u,v)加入ST后不使ST中产生回路,

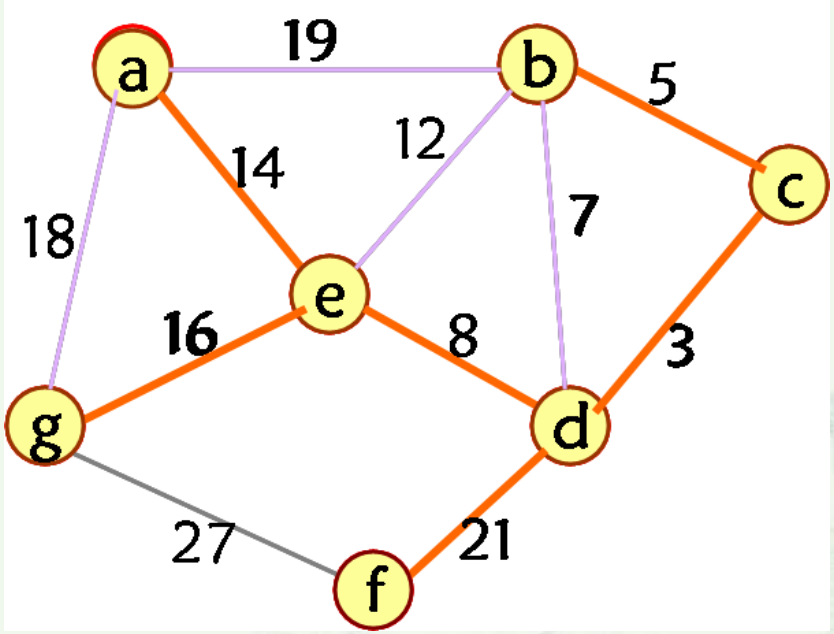
  则 输出边(u,v); 且 k++;

}

算法设计

- 如何找最小权值的边。

堆　（结构如何？）



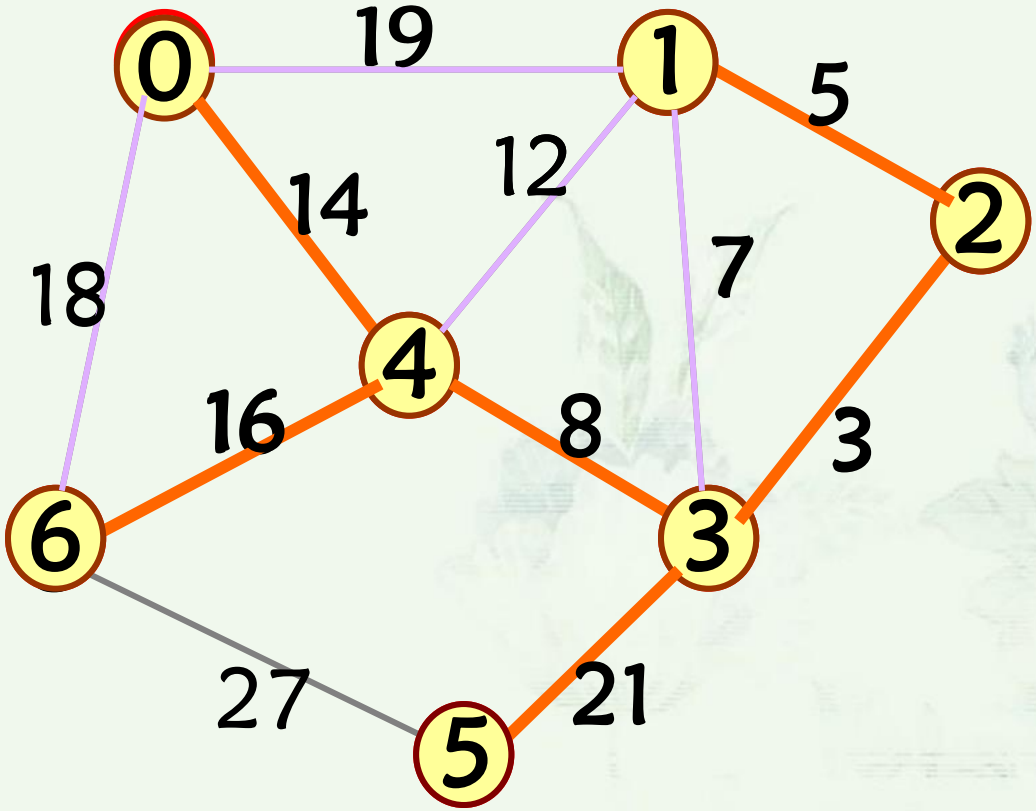
```
template <class T, class E>
struct MSTEdgeNode    //树边结点的类定义
{
    int tail, head;                  //两顶点位置
    E cost;     // 关系比较<的重载
};

MinHeap <MSTEdgeNode<T, E>> H(m); //最小堆
```

算法设计

- 如何判断是否存在回路。
- 现象：所选边的两个顶点在同一集合中

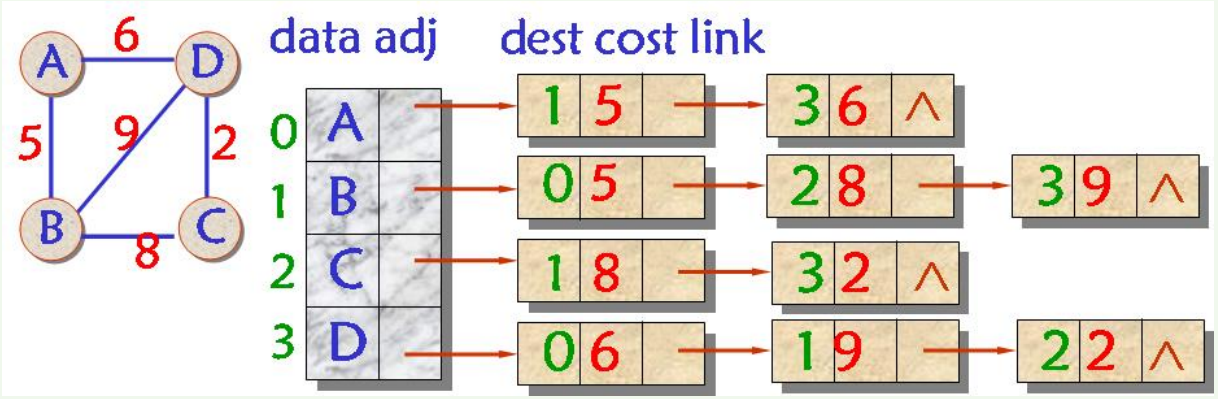
并查集

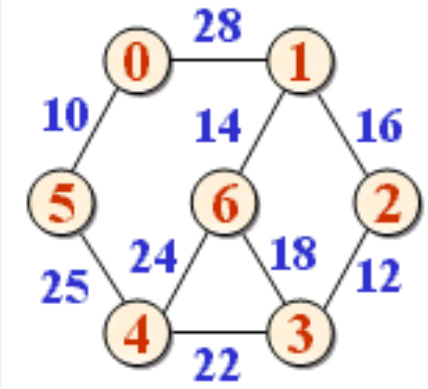
UFSets F(n); //并查集



图的存储结构

- 邻接表
- 邻接矩阵
- 边集数组



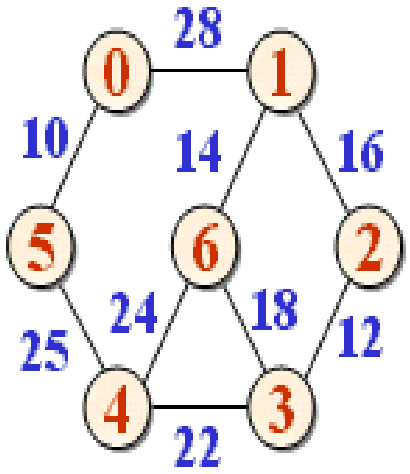


$$\begin{pmatrix} 0 & 28 & \infty & \infty & \infty & 10 & \infty \\ 28 & 0 & 16 & \infty & \infty & \infty & 14 \\ \infty & 16 & 0 & 12 & \infty & \infty & \infty \\ \infty & \infty & 12 & 0 & 22 & \infty & 18 \\ \infty & \infty & \infty & 22 & 0 & 25 & 24 \\ 10 & \infty & \infty & \infty & 25 & 0 & \infty \\ \infty & 14 & \infty & 18 & 24 & \infty & 0 \end{pmatrix}$$

初始化操作

图中所有边的数据（结点，结点，权）插入堆中。

```
for (u = 0; u < n; u++)  // 图采用邻接矩阵
  for (v = u+1; v < n; v++)
    if ( Edge[u][v]<maxWeight)
    {  ed.tail = u;  ed.head = v;
       ed.cost = Edge[u][v];
       //插入堆
       H.Insert(ed);
    }
```



$$\begin{pmatrix} 0 & 28 & \infty & \infty & \infty & 10 & \infty \\ 28 & 0 & 16 & \infty & \infty & \infty & 14 \\ \infty & 16 & 0 & 12 & \infty & \infty & \infty \\ \infty & \infty & 12 & 0 & 22 & \infty & 18 \\ \infty & \infty & \infty & 22 & 0 & 25 & 24 \\ 10 & \infty & \infty & \infty & 25 & 0 & \infty \\ \infty & 14 & \infty & 18 & 24 & \infty & 0 \end{pmatrix}$$

```
    count = 1;                    //最小生成树边数计数
    while (count < n) {           //反复执行，取n-1条边
      H.Remove(ed);               //从堆中取权值最小的边
      //检查该边的两个顶点是否在同一集合中
      //查找出该两顶点所在集合的根u与v—并查集
      u = F.Find(ed.tail);     v = F.Find(ed.head);
      if (u != v)
       {              //不是同一集合,不连通
        F.Union(u, v);  //集合合并,连通它们—并查集
        MST.Insert(ed); //将该边放入生成树MST中
        // cout <<ed.tail<<ed.head<<ed.cost 输出边
        count++;
       }
    }
};
```

总结：克鲁斯卡尔(Kruskal)算法

从边入手找顶点

具体做法：[贪心法]

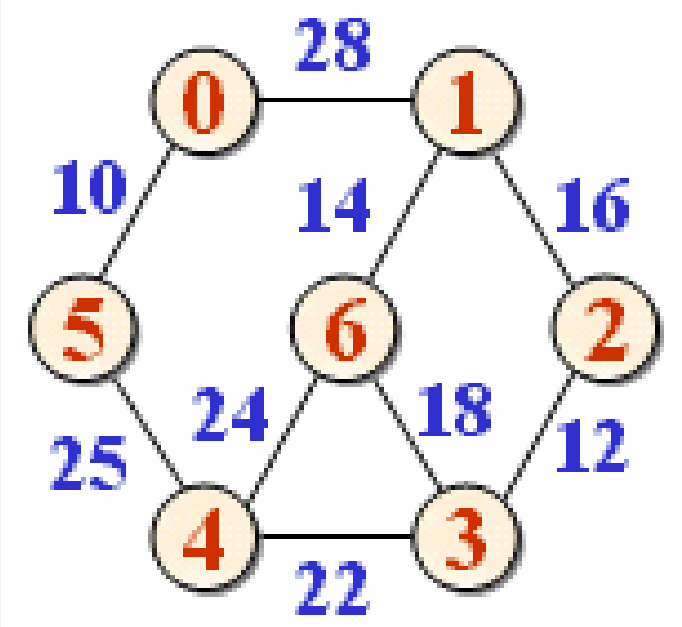
1.先构造一个只含 n 个顶点的子图 SG;

2.然后从权值最小的边开始，若它的添加不使SG中产生回路，则在 SG 上加上这条边，

3.反复执行第2步，直至加上 n-1 条边为止。

新的解决方案

从顶点入手找边



普里姆(prim)算法

算法步骤：

1.设定其中一个结点为出发点；

2.分组：出发点为第一组，其余结点为第二组。

3.在一端属于第一组和另一端属于第二组的边中选择一条权值最小的一条。

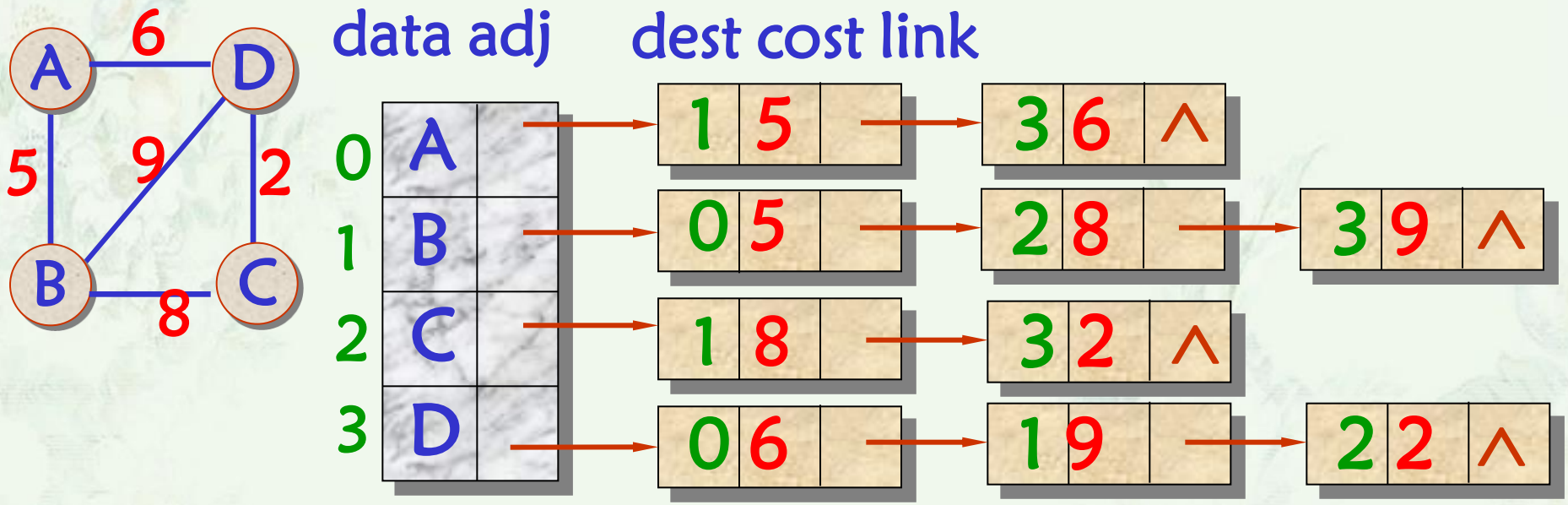
4.把原属于第二组的结点放入第一组中。

5.反复2，3两步，直到第二组为空为止。

算法设计

（1）图的存储结构

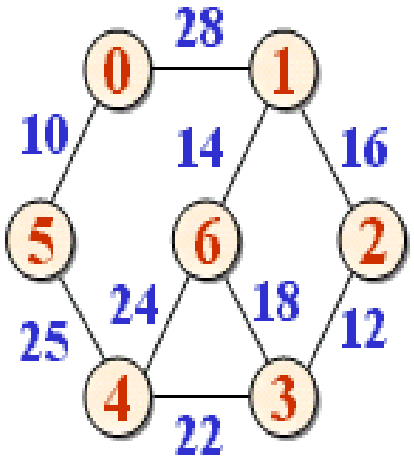
邻接表



算法设计

（1）图的存储结构

邻接矩阵



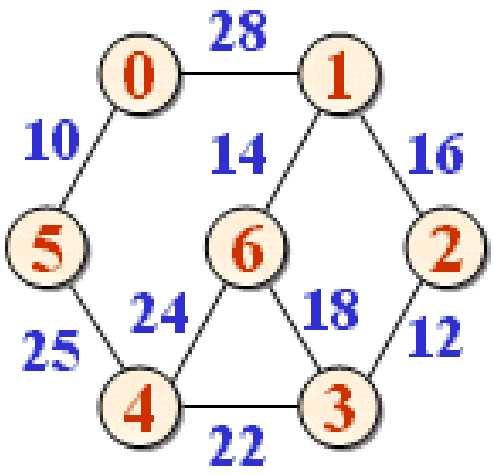
$$\begin{pmatrix} 0 & 28 & \infty & \infty & \infty & 10 & \infty \\ 28 & 0 & 16 & \infty & \infty & \infty & 14 \\ \infty & 16 & 0 & 12 & \infty & \infty & \infty \\ \infty & \infty & 12 & 0 & 22 & \infty & 18 \\ \infty & \infty & \infty & 22 & 0 & 25 & 24 \\ 10 & \infty & \infty & \infty & 25 & 0 & \infty \\ \infty & 14 & \infty & 18 & 24 & \infty & 0 \end{pmatrix}$$

算法设计

- （2）出发点问题

  结点u

- （3）如何进行结点的分组

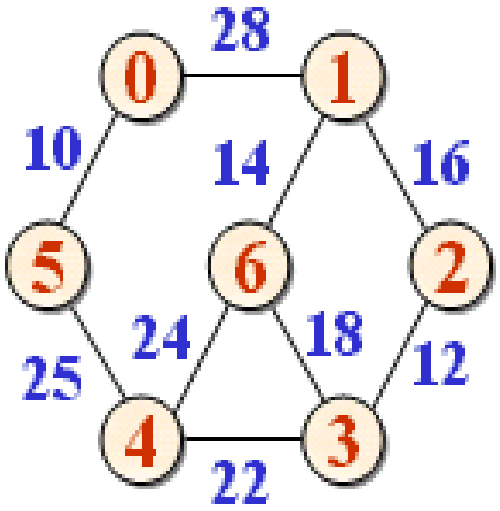


$$\begin{pmatrix} 0 & 28 & \infty & \infty & \infty & 10 & \infty \\ 28 & 0 & 16 & \infty & \infty & \infty & 14 \\ \infty & 16 & 0 & 12 & \infty & \infty & \infty \\ \infty & \infty & 12 & 0 & 22 & \infty & 18 \\ \infty & \infty & \infty & 22 & 0 & 25 & 24 \\ 10 & \infty & \infty & \infty & 25 & 0 & \infty \\ \infty & 14 & \infty & 18 & 24 & \infty & 0 \end{pmatrix}$$

算法设计

- （4）如何找出权值最小的边

  堆 （结构？）



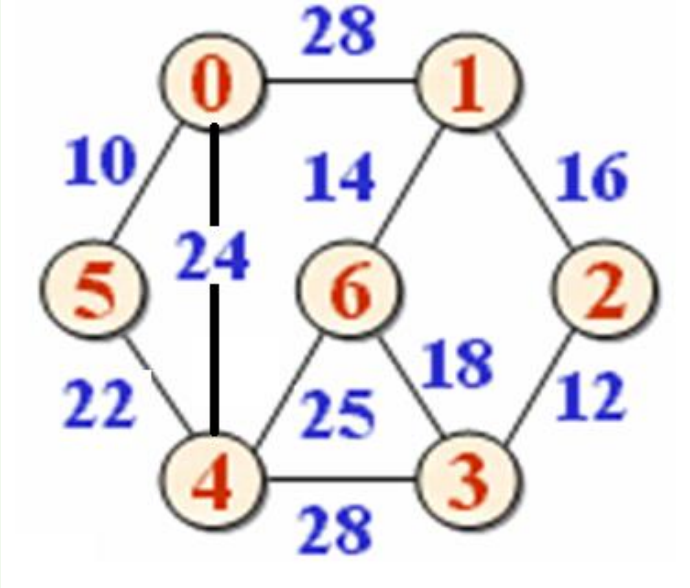
$$\begin{pmatrix} 0 & 28 & \infty & \infty & \infty & 10 & \infty \\ 28 & 0 & 16 & \infty & \infty & \infty & 14 \\ \infty & 16 & 0 & 12 & \infty & \infty & \infty \\ \infty & \infty & 12 & 0 & 22 & \infty & 18 \\ \infty & \infty & \infty & 22 & 0 & 25 & 24 \\ 10 & \infty & \infty & \infty & 25 & 0 & \infty \\ \infty & 14 & \infty & 18 & 24 & \infty & 0 \end{pmatrix}$$

算法设计

```
template <class T, class E>
struct MSTEdgeNode    //树边结点的类定义
{
    int  head;   //未加入生成树的结点编号
    int tail; //已加入生成树的结点编号
    E cost;      // 关系比较<的重载
};

MinHeap <MSTEdgeNode<T, E>> H(m); //最小堆
```

算法设计

- （5）从堆中删除的边，其两个顶点已经被加入生成树？

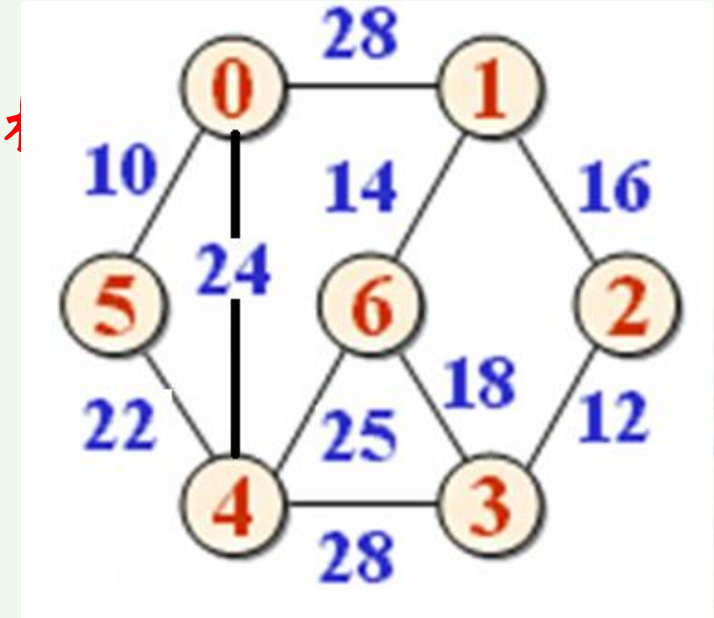


算法的描述

```
Edge[u][u] = true;      //u 加入生成树
count = 1;  // 记录已加入生成树的边数
while (count<n)
{   for (v=0;v<n;i++) // 逐个查看v是否在生成树中
   { if ((Edge[v][v]==0) &&Edge[u][v]<maxWeight)
     {  ed.tail=u;    ed.head =v ;
        ed.cost = Edge[u][v];
        //(u,v,w)加入堆
        H.Insert(ed);
      }
 }
```

$$\begin{pmatrix} 0 & 28 & \infty & \infty & \infty & 10 & \infty \\ 28 & 0 & 16 & \infty & \infty & \infty & 14 \\ \infty & 16 & 0 & 12 & \infty & \infty & \infty \\ \infty & \infty & 12 & 0 & 22 & \infty & 18 \\ \infty & \infty & \infty & 22 & 0 & 25 & 24 \\ 10 & \infty & \infty & \infty & 25 & 0 & \infty \\ \infty & 14 & \infty & 18 & 24 & \infty & 0 \end{pmatrix}$$

```
while (!H.IsEmpty() && count < n) {
  H.Remove(ed);          //从堆中删除最小权的边
  if (Edge[ed.head] [ed.head]==0)//是否符合选择要求
   {
      MST.Insert(ed); //将该边加入最小生成树
      // cout <<ed.tail<<ed.head<<ed.cost 输出边
      u = ed.head;
      Edge[u][u] = true; //u加入生成
      count++;
      break;
   }
  }
}
```



总结：普里姆算法

从顶点入手找边

实施步骤：[贪心法]

1.分组，出发点为第一组，其余结点为第二组。

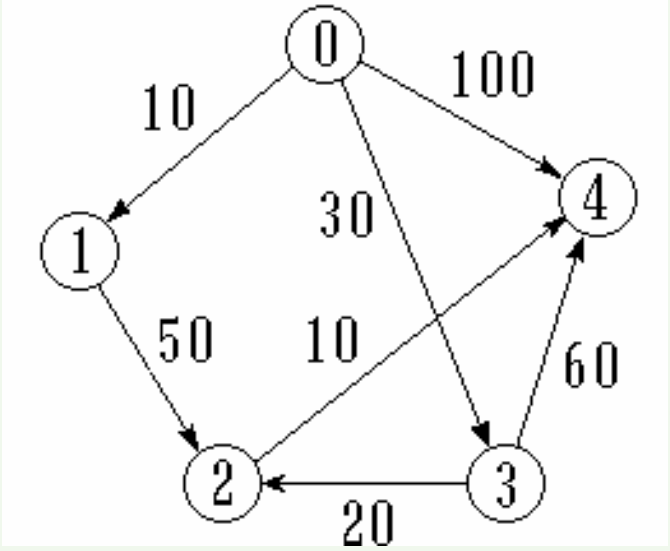
2.在一端属于第一组和另一端属于第二组的边中选择一条权值最小的一条。

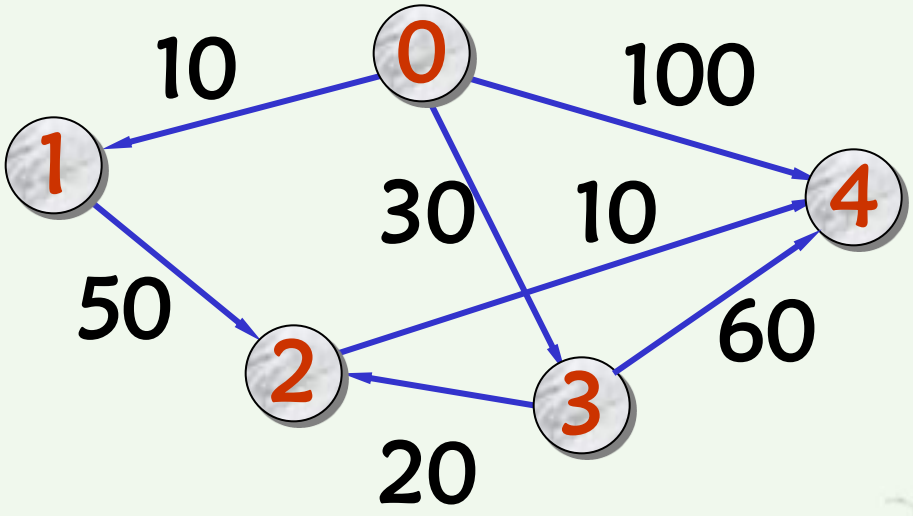
3.把原属于第二组的结点放入第一组中。

4.反复2，3两步，直到第二组为空为止。

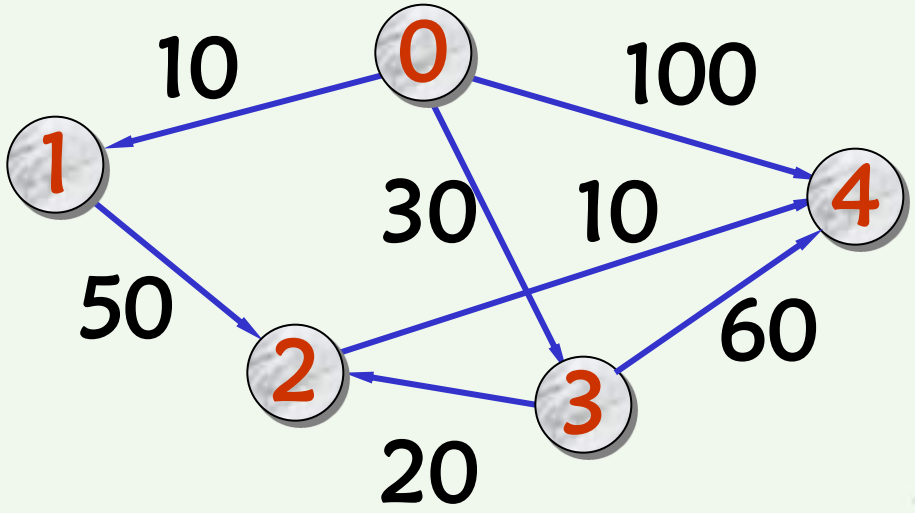
8.5 求从源点到其余各点的最短路径

- **问题的提出**：给定一个带权有向图D与源点v，求从v到D中其它顶点的最短路径。限定各边上的权值大于或等于0。




0
1
2
3
4
10
100
30
10
50
60
20

Dijkstra 算法实现需要考虑的问题

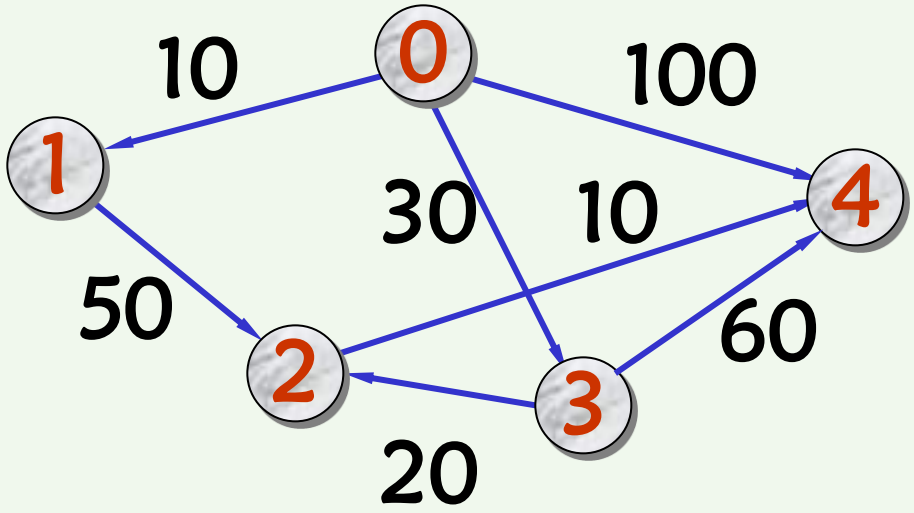


- 1.为了方便比较距离值，我们需要有空间保存这些以前得到的距离值。

  设置一个距离值表

- 2.标识哪些结点已经求出最短路径？

  设置一个访问数组



Dijkstra逐步求解过程中距离值表dist的变化过程：

dist　　visited

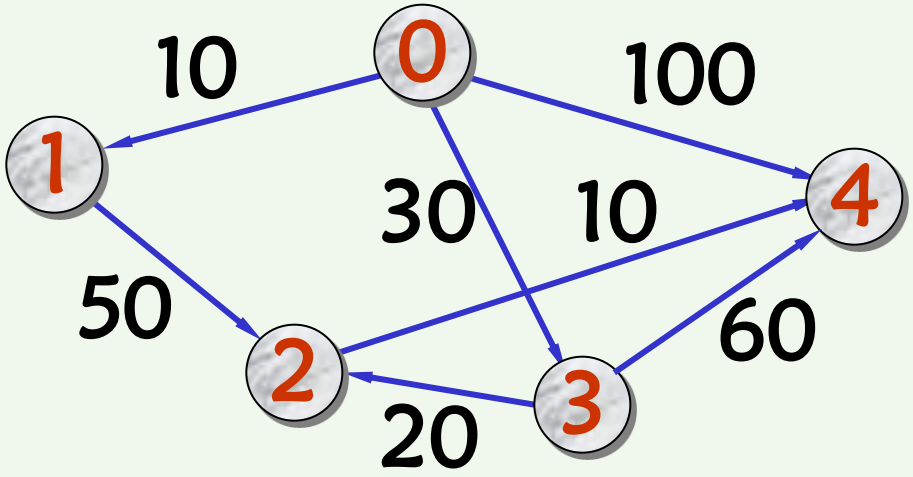
算法的实现

- 图的存储结构
- 针对具体存储结构的算法描述

图的存储结构

- 邻接表
- 邻接矩阵

$$\begin{pmatrix} 0 & 10 & \infty & 30 & 100 \\ \infty & 0 & 50 & \infty & \infty \\ \infty & \infty & 0 & \infty & 10 \\ \infty & \infty & 20 & 0 & 60 \\ \infty & \infty & \infty & \infty & 0 \end{pmatrix}$$



Dijkstra逐步求解过程中距离值表dist的变化过程：

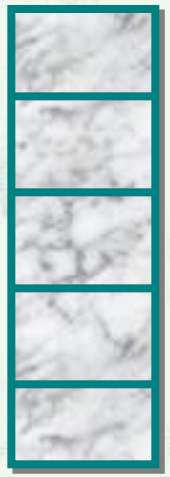
dist　　visited

发现问题

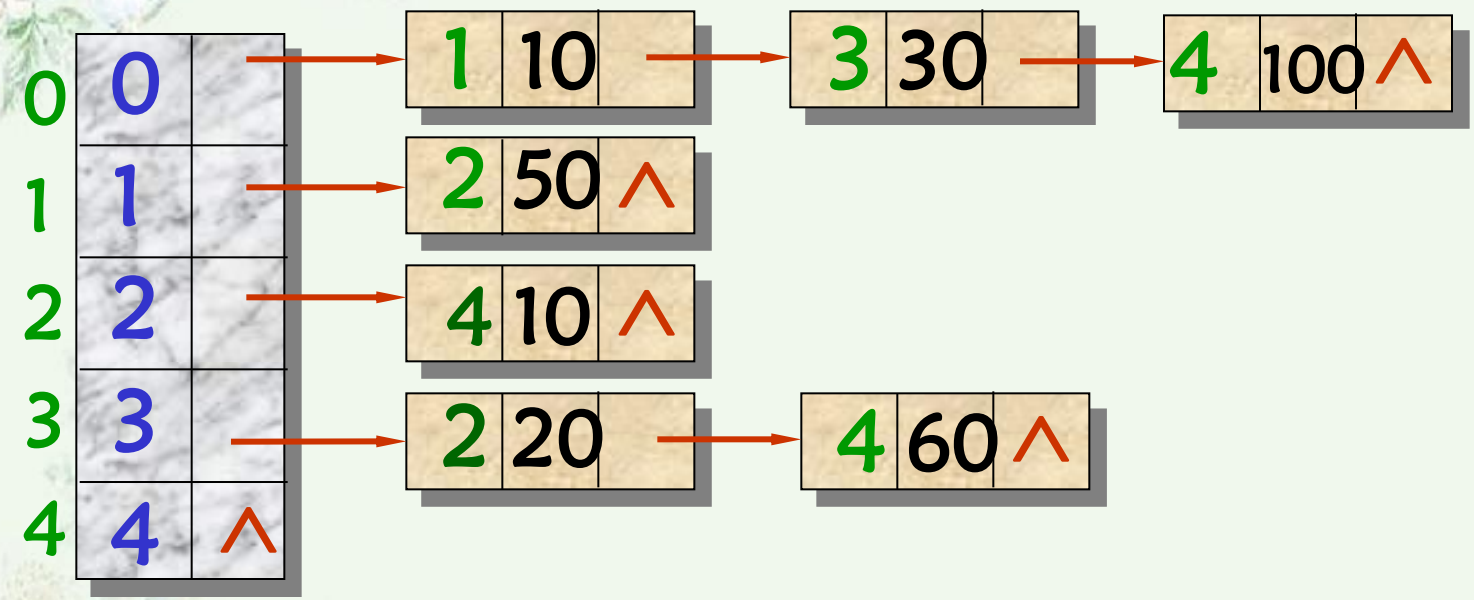
实现算法

```
for (i=0;i<n;i++)  // n 表示结点个数
      dist[i]=G[v0][i];    //初始化dist数组
 G[v0][v0]=1;  // 加入v0，置出发点访问标志
  for (i=0;i<n-1;i++) //共完成n-1次
  {    // 找最小值
     min=32767;
      for (k=0;k<n;k++)
      {  if ( (G[k][k]==0) &&(dist[k]<min))
                {    pos=k; min=dist[k];  }
       }
    G[pos][pos]=1;
     for (j=0;j<n;j++)
     {  if ((G[j][j]==0)&&(G[pos][j]+min<dist[j]))
            dist[j]=G[pos][j]+min;
      }
 }
```

0	10	∞	30	100
∞	0	50	∞	∞
∞	∞	0	∞	10
∞	∞	20	0	60
∞	∞	∞	∞	0

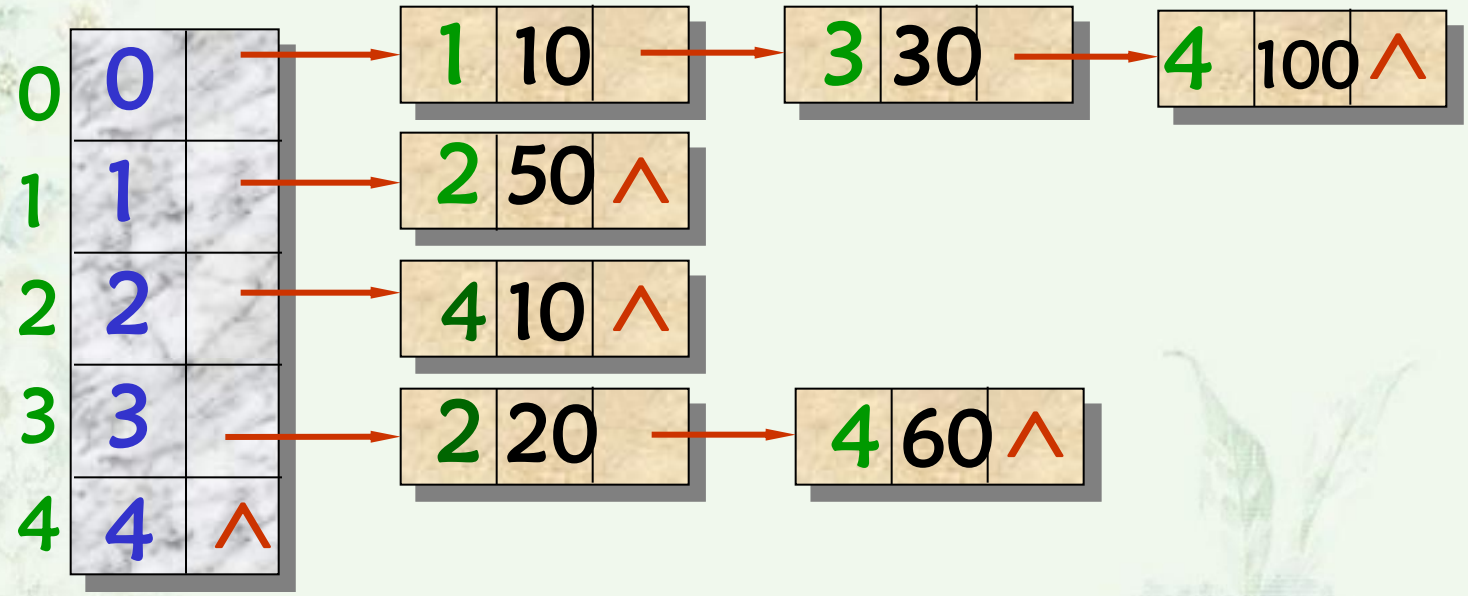


dist



Dijkstra逐步求解过程中距离值表dist的变化过程：

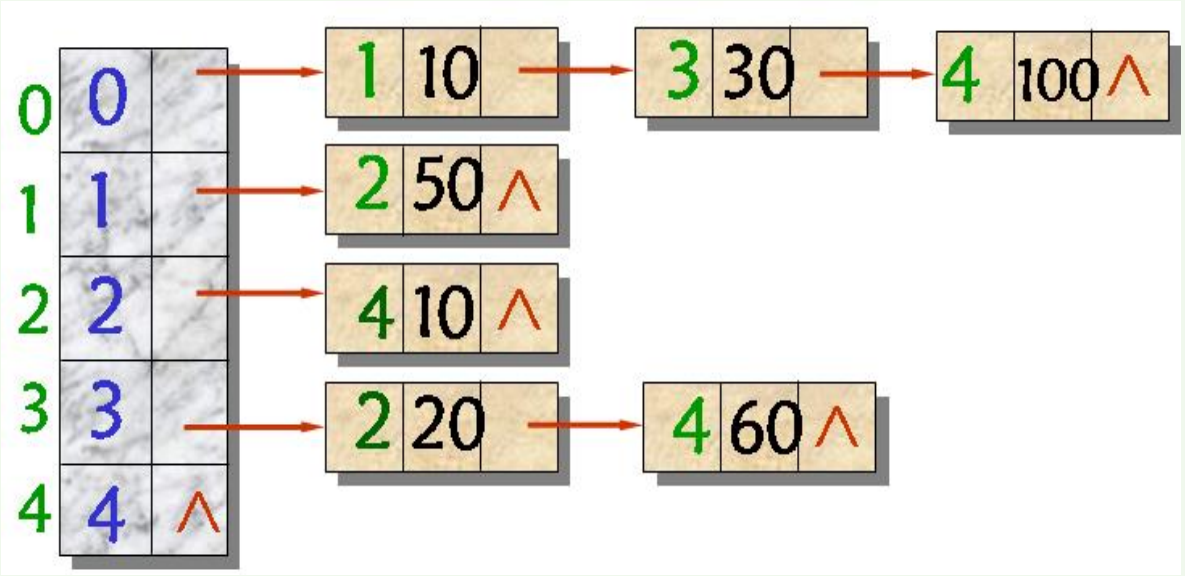
邻接表结构



找出结点i的所有邻接点

```
    Edge<T, E> *p = NodeTable[i].adj;
     while (p != NULL)
     {
          ……
          p = p->link;
     }
}
```

实现算法

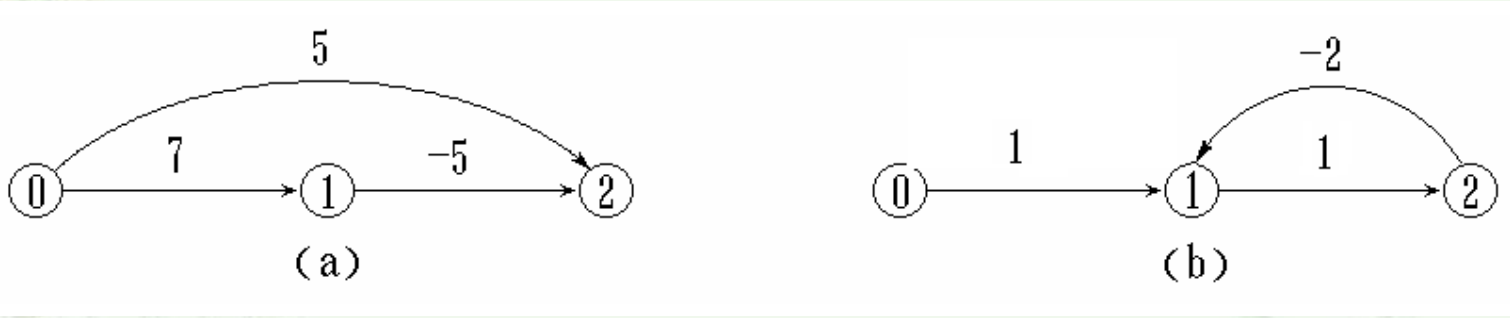


请自行完成算法的设计

Dijkstra算法的改进

分析算法中最耗费时间的部分。

Dijkstra算法的困境



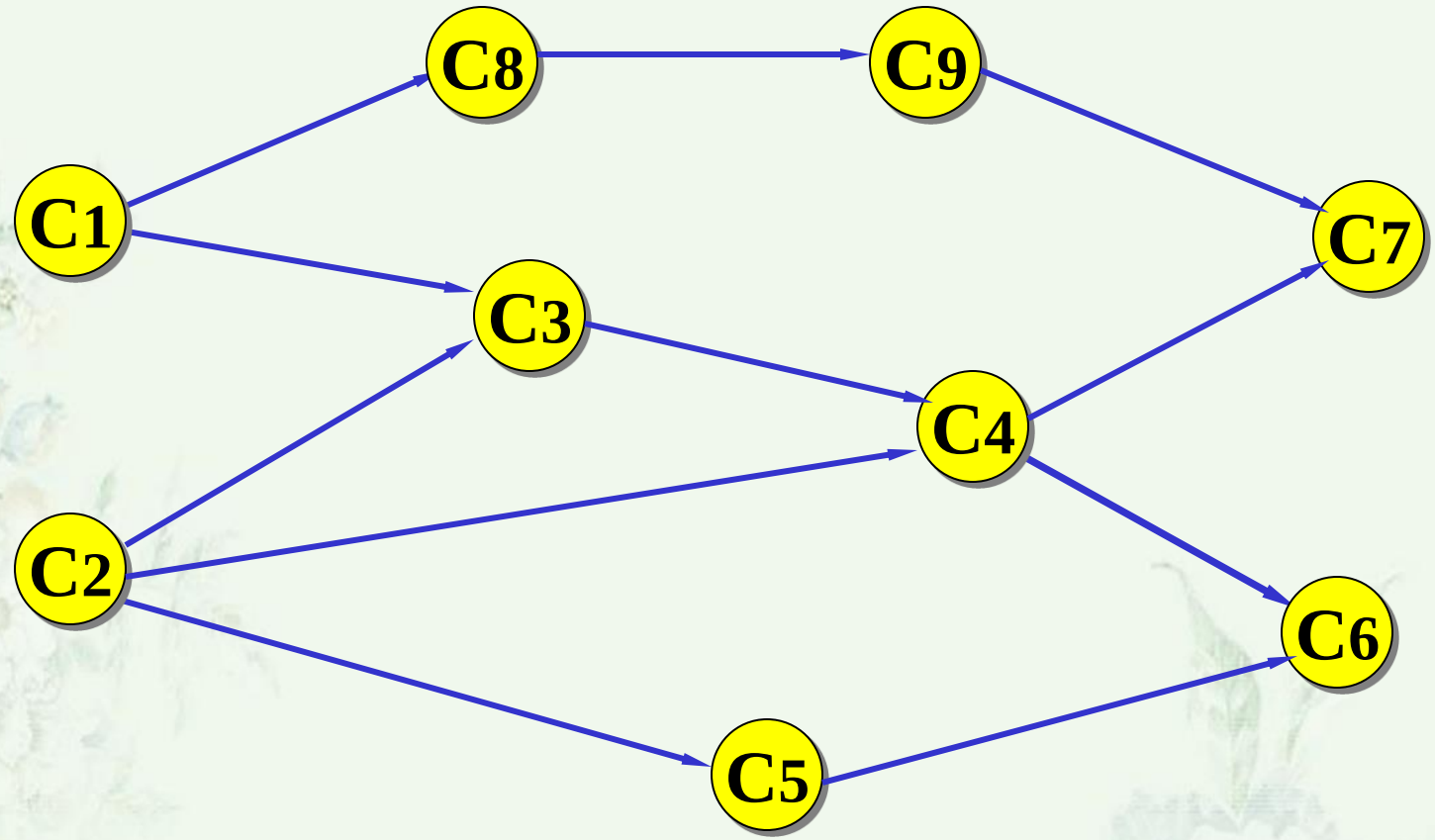
8.6 拓扑排序

用顶点表示活动的网络 (AOV网络)

计划、施工过程、生产流程、程序流程等都是“工程”。除了很小的工程外，一般都把工程分为若干个叫做“活动”的子工程。完成了这些活动，这个工程就可以完成了。

例如，计算机专业学生的学习就是一个工程，每一门课程的学习就是整个工程的一些活动。其中有些课程要求先修课程，有些则不要求。这样在有的课程之间有领先关系，有的课程可以并行地学习。

课程编号	课程名称	先决条件
C_1	高等数学	
C_2	程序设计基础	
C_3	离散数学	C_1, C_2
C_4	数据结构	C_3, C_2
C_5	高级语言程序设计	C_2
C_6	编译方法	C_5, C_4
C_7	操作系统	C_4, C_9
C_8	普通物理	C_1
C_9	计算机原理	C_8



学生课程学习工程图

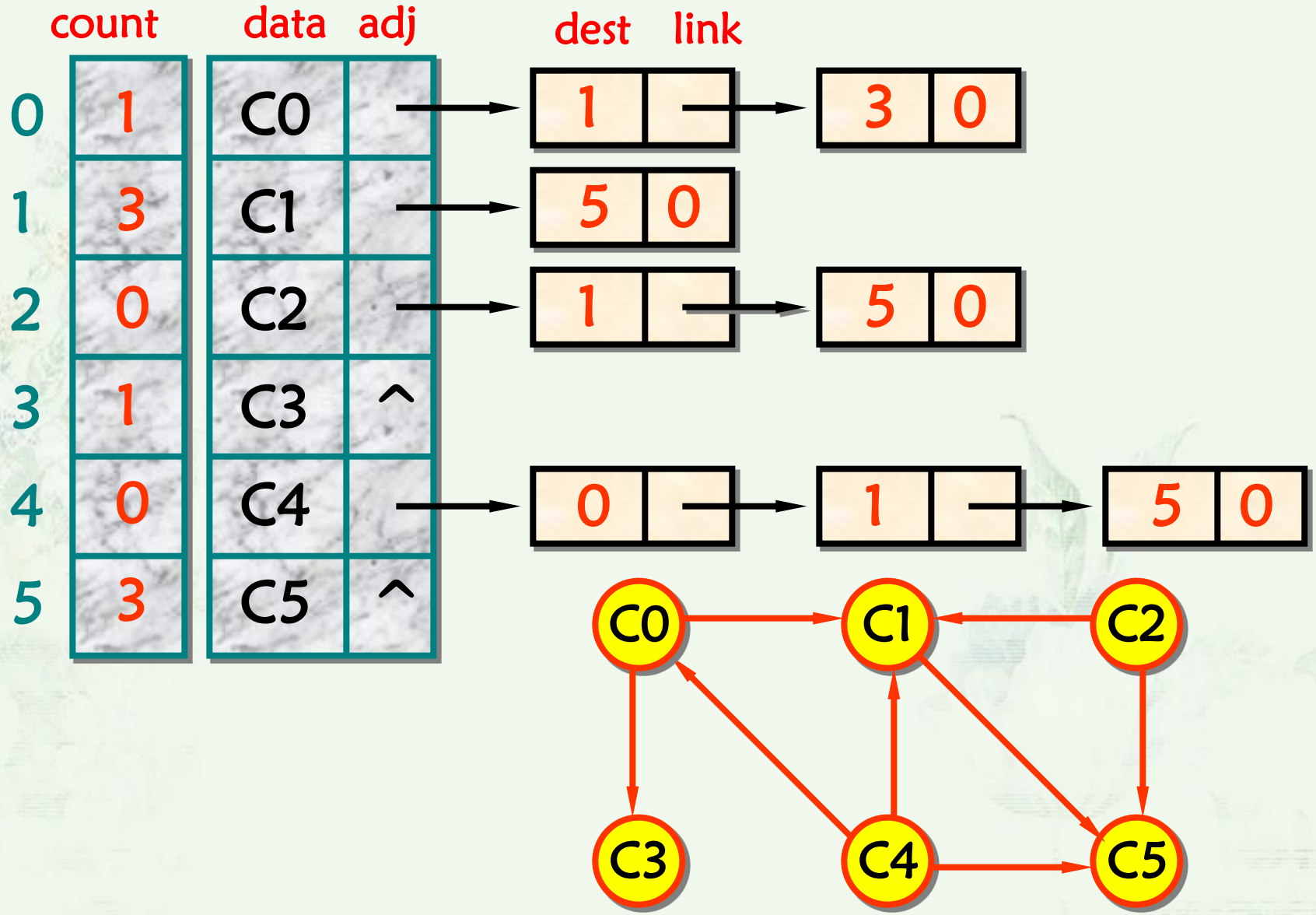
进行拓扑排序的步骤：

① 输入AOV网络。令 n 为顶点个数。
② 在AOV网络中选一个入度为0的结点，并输出之；
③ 从图中删去该顶点，同时删去所有它发出的有向边；
④ 重复以上②、③步，直到下面的情况之一出现：
(1)全部顶点均已输出，拓扑有序序列形成，拓扑排序完成；
(2)图中还有未输出的顶点，但已没有入度为0的结点(说明网络中必存在有向环)。

图的存储结构

- 邻接矩阵
- 邻接表

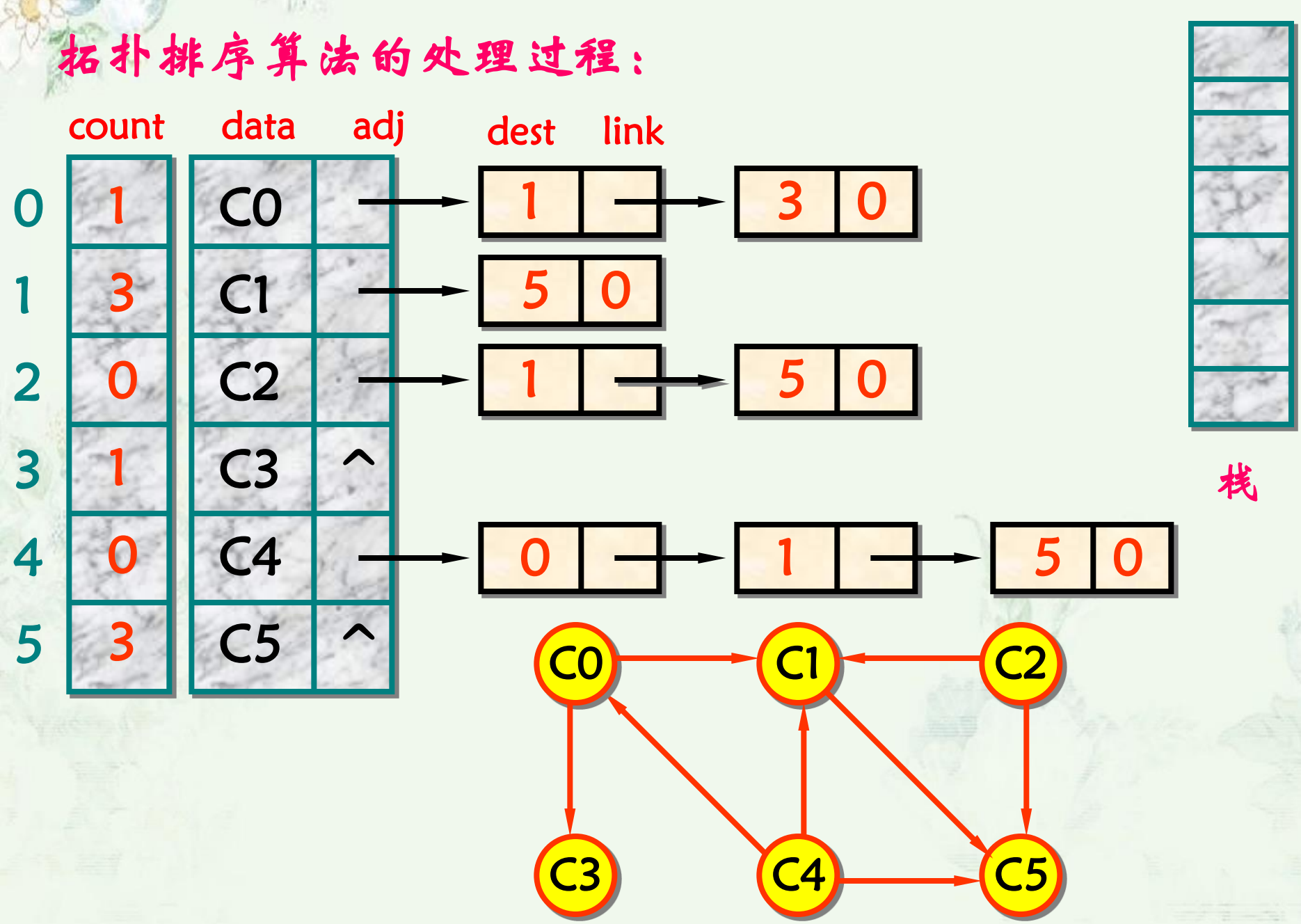
算法的设计


count
data
adj
dest
link
0
1
C0
1
3
0
1
3
C1
5
0
2
0
C2
1
5
0
3
1
C3
^
4
0
C4
0
1
5
0
5
3
C5
^
C0
C1
C2
C3
C4
C5

拓扑排序算法的描述：

```
取入度为零的顶点v;
while (顶点v存在) {
    cout<<v;   ++m;
    p=NoeTable[v].adj; //取第一个邻接点
    while (P!=NULL)
    {
      count[p->dest]--;
      p=p->link; //取下一个邻接点
    }
    取下一个入度为零的顶点v;
}
if (m<n)  cout<<“图中有回路”;
```

拓扑排序算法的处理过程：



于是，修改后的拓扑排序算法描述如下：

1.在count数组中找出入度为零的顶点，并分别入栈；

2.while (栈非空)

{ 出栈到v；

cout<<NodeTable[v].data;

++m;

p=NodeTable [v].adj;

while (P!=NULL)

{

count[p->dest]--;

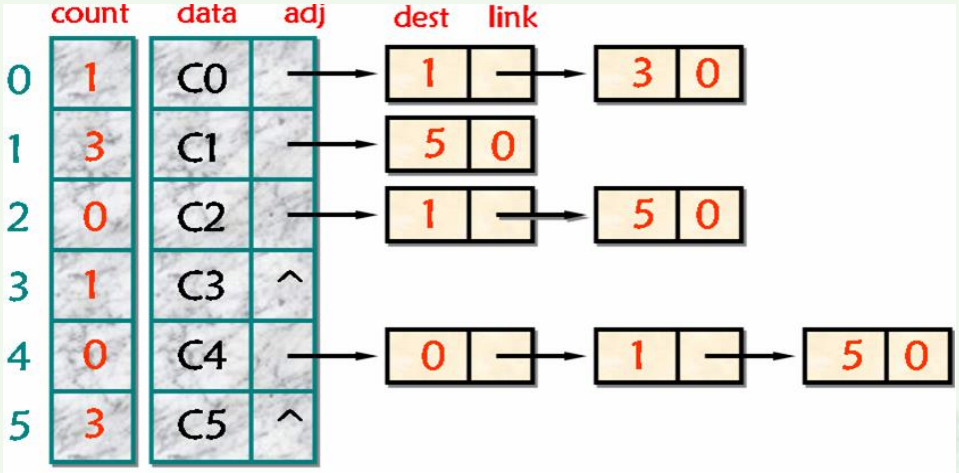
如果count[p->dest]==0则结点p->dest入栈

p=p->link;

}

}

3.if(m<n) cout<<“图中有回路”；



拓扑排序算法的处理过程：