用邻接矩阵存储的 Graph_Matrix 类

采用邻接矩阵存储的 Graph_Matrix 类定义如下。其中，顶点的序号用整数表示，顶点的序号从 0 开始，顶点个数的上界用常数 MaxGraphSize 表示，权值的上界用常数 MaxWeight 表示。

```
const int MaxGraphSize = 256 ;    // 图的最大顶点个数
const int MaxWeight = 1000;       // 图中允许的最大权值
class Graph_Matrix
{
private:
      //邻接矩阵
      int edge[ MaxGraphSize][ MaxGraphSize] ;
      // 当前图中顶点个数
      int graphsize;
public:
      // I. 图的构造函数和析构函数
    1. Graph_Matrix () ;
    2. virtual ~ Graph_Matrix () ;

    // II.图的维护函数
      //3. 检测图是否为空
      int GraphEmpty(void) const ;
      //4. 检测图中顶点个数是否已超过规定的顶点个数上界
      int GraphFull(void) const ;
      //5. 返回图的顶点个数
      int NumberOfVertices( void ) const ;
      //6. 返回图的边个数
      int NumberOfEdges( void ) const ;
      //7. 返回指定边的权值
      int GetWeight( const int & v1, const int & v2 ) ;
      //8. 返回顶点 v 的邻接顶点表
      int* & GetNeighbors( const int & v ) ;
      //9. 返回序号为 v 的顶点的第一个邻接顶点的序号
      int GetFirstNeighbor( const int v ) ;
      //10. 返回序号为 v1 的顶点相对于序号为 v2 的顶点的下一个邻接顶点的序号
      int GetNextNeighbor( const int v1, const int v2 ) ;
     // 11. 向图中插入一个顶点
      void InsertVertex( const int & v) ;
     // 12. 向图中插入一条边
      void InsertEdge( const int & v1, const int & v2, int weight ) ;
     // 13. 从图中删除一个顶点
      void DeleteVertex( const int & v ) ;
    // 14. 从图中删除一条边
      void DeleteEdge( const int & v1, const int & v2 ) ;
      // III. 图的基本算法
     // 15. 采用递归的方法从顶点表的第一个顶点开始进行图的深度优先搜索
      void DepthFirstSearch( ) ;
    // 16. 采用迭代的方法从指定顶点 v 开始进行图的深度优先搜索
    void DFS(const int v) ;
    // 17. 从顶点 v 开始进行图的广度优先搜索
    void BFS ( const int v ) ;
    // 18. 输出图的拓扑排序
    void TopoOrder( ) ;
    // 19. 输出图的关键路径
    void CriticalPath ( ) ;
    // 20. 在无权图中求指定顶点到其他所有顶点的最短路径
    void ShortestPath(const int v ) ;
      // 21. 在正权图中求指定顶点到其他所有顶点的最短路径
     void DShortestPath(const int v ) ;
    // 22. 在正权图中求每对顶点间的最短路径
    void AllLength () ;
    // 23. 构造图的最小支撑树的普里姆算法
    void Prim ( ) ;
} ;
```

用邻接表存储的 Graph_List 类

存储图的另一种方式是用邻接表。如果存储结构使用邻接表，为了实现 Graph_List 类，首先定义该类使用的两个结构体 ——边结点的结构体和顶点表结点的结构体。

```
// 边结点的结构体
struct Edge
{
        friend class Graph_List ;
        int VerAdj ;        // 邻接顶点序号，从 0 开始编号
        int cost ;                  // 边的权值
        Edge *link ;                // 指向下一个边结点的指针
} ;

// 顶点表中结点的结构体
struct Vertex
{
        friend class Graph_List;
        int VerName;                    // 顶点的名称
        Edge *adjacent;          // 边链表的头指针
}

//采用邻接表存储的 Graph_List 类定义
class Graph_List
{
       private :
                Vertex *Head;                   // 顶点表头指针
                int graphsize;                    // 图中当前顶点的个数
       public :
       // I.  图的构造函数和析构函数
                Graph_List ( );
              virtual ~Graph_List ( );
     // II.  图的维护函数
          // 与 Graph_Matrix 类定义中的维护函数定义相同，这里省略。
        // III.  图的基本算法
          // 与 Graph_Matrix 类定义中的基本算法定义相同，这里省略。
} ;
```

采用邻接矩阵存储的 Graph_Matrix 类的部分实现

```
// I.1  构造函数, 创建一个图
Graph_Matrix :: Graph_Matrix ()
{
    cin >> graphsize;
    for( int i = 0 ; i < graphsize ; i ++ )
       for( int j = 0 ; j < graphsize ; j ++ )
             cin >> edge[i][j];
}

// II.9  取得序号为 v 的顶点的第一个邻接顶点的序号
int Graph_Matrix ::GetFirstNeighbor( const int v )
{
            if (v = = -1) return -1;
            for( int i = 0 ; i < graphsize ; i ++ )
                    if( edge[v][i] > 0 && edge[v][i] < MaxWeight)
                          return i ;

             return -1 ; //  若 v 没有邻接顶点，则返回-1
}

//II.10  取得顶点 v1 相对于 v2 的下一个邻接顶点的序号
int Graph_Matrix ::GetNextNeighbor( const int v1 , const int v2 )
{
      if (v1 = = -1 || v2= = -1) return -1;
      for( int i = v2 + 1 ; i < graphsize ; i ++ )
```

```
                if( edge[v1][i] > 0 && edge[v1][i] < MaxWeight)
                    return i ;
        return -1 ; //若在 v2 之后没有与 v1 邻接的顶点，则返回-1
    }
```

采用邻接链表存储的 Graph_List 类的部分实现

```
// I.1  构造函数
    Graph_List :: Graph_List ()
    {
       int e, from, to, weight ;
    //用数组实现顶点表，Head 指向数组的第一个元素
    Head = new Vertex [MaxGraphSize];
    cin >> graphsize ;                                          //输入顶点个数
    for ( int i = 0 ; i < graphsize ; i ++ )              //初始化 Head 数组
    {
     Head[i].VerName = i;
          Head[i].adjacent = NULL;
    }
    cin >> e;                                                  //输入边的个数
    for ( i = 0 ; i < e ; i ++ )                              //依次输入各边
    {
          cin >> from >> to >> weight ;                       //输入新边的始点、终点和权值
          Edge* p = new Edge;                                 //将新边插入图中
           p→VerAdj = to;
           p→cost = weight;
           p→link = NULL;
           Edge *q = Head[from].adjacent;
           if (q = = NULL)
                 Head[from].adjacent = p;
           else
           {
               while (q→link != NULL)
                  q = q→link;
               q→link = p;
           }
    }
}

// II.2  析构函数
      Graph_List :: ~Graph_List ( )
      {
            for( int i = 0 ; i < graphsize ; i + + )        //  删除每个顶点的边链表
            {
                  Edge *p = Head[i].adjacent ;
                  while ( p != NULL )                        //  循环删除
                  {
                        Head[i].adjacent = p→link ;
                        delete p ;                                 //  释放结点的空间
                        p = Head[i].adjacent;
                  }
            }
            delete [ ] Head ;                                  //  释放顶点数组
      }

  // II. 7 返回指定边的权值
      int Graph_List:: GetWeight( const int& v1 , const int & v2 )
      {
            if (v1 = = -1 || v2 = = -1) return 0;
             Edge *p = Head[v1].adjacent ;                 // v1 的边链表头指针
            while( p != NULL )                             //  循环搜索此边
            {
                 if( p→VerAdj = = v2 )
                       return p→cost ;                     //  找到，返回边的权值
```

```
            p = p→link ;
        }
      return 0 ;    // 此边不存在，返回 0
}

// II.9 求序号为 v 的顶点的第一个邻接顶点的序号
int Graph_List:: GetFirstNeighbor( const int v )
{
     if ( v = = -1 ) return -1;
     Edge *p = Head[v].adjacent ;
   if (p != NULL)
            return p→VerAdj ;
   else
            return -1;
}

// II.10 求序号为 v1 的顶点相对于序号为 v2 的顶点的下一个邻接顶点的序号
int Graph_List :: GetNextNeighbor( const int v1 , const int v2 )
{
       if ( v1 != -1 && v2 != -1 )
       {
            Edge *p = Head[v1].adjacent ;
            while( p→ VerAdj != v2 && p!=NULL )
                 p = p→ link ;
          if (p = = NULL) return -1;
            p = p→ link ;
            if (p = = NULL) return -1;
            return p→ VerAdj ;
       }
       return -1 ;
}
```

深度优先遍历图的 C++实现

```
void Graph_List:: DepthFirstSearch( )
{
          int *visited = new int[graphsize] ;              // 为辅助数组申请空间
          for( int k = 0 ; k < graphsize ; k ++ )          // 数组初始化
               visited[k] = 0 ;
       DepthFirstSearch   ( 0 , visited) ;  // 从序号为 0 的顶点出发，深度优先遍历图
          delete[ ] visited ;                  // 释放辅助数组空间
}

// 从序号为 v 的顶点出发，深度优先遍历图的递归算法
void Graph_List :: DepthFirstSearch ( const int v ,int* visited)
{
     cout << v << " " ;                    // 输出 v 的名称
     visited[v] = 1 ;                           // 说明 v 已被访问过
     Edge *p = Head[v].adjacent ;       // 取得 v 的第一个邻接顶点的序号
     while( p != NULL )                         // 若存在顶点 p
  {
     if( visited[p-> VerAdj]!=1 )                    // 若 p 未被访问过，从 p 递归访问
          DepthFirstSearch( p-> VerAdj , visited) ;
          p = p-> link   ;                          // p 为 v 关于 p 的下一个邻接顶点
     }
}
```