第6章 图

#数据结构与算法

6.1 图的定义与术语

> **定义**：图G是由表示数据元素的集合V和表示数据元素之间关系的集合E组成。
>
> $$G = (V, E)$$

基本概念

- **顶点**：图中的数据元素，也称节点。
- **边**：顶点之间的关系。无向图中用 `(v, w)` 表示，有向图中用 `<v, w>` 表示弧，v为弧尾或起点，w为弧头或终点。
- **权**：与图的边或弧相关的数值，可以表示距离、成本等。
- **顶点的度**：
  - 无向图：与顶点v相关联的边的数目，记为 `TD(v)`。
  - 有向图：分为**入度(In-degree)** `ID(v)`（以v为终点的弧的数目）和**出度(Out-degree)** `OD(v)`（以v为起点的弧的数目）。

$$TD(v) = ID(v) + OD(v)$$

边数与顶点，度之间的关系

设图G=(V, E)有n个顶点和e条边。

- **无向图**：所有顶点的度之和等于边数的2倍。

$$\sum_{i=1}^{n} TD(v_i) = 2e$$

- **有向图**：所有顶点的入度之和等于出度之和，等于边数。

$$\sum_{i=1}^{n} ID(v_i) = \sum_{i=1}^{n} OD(v_i) = e$$

图的分类

- **无向图**：图中任意两个顶点之间的边都是无向边。
- **有向图**：图中任意两个顶点之间的边都是有向边，也称有向弧。
- **网**：带权的图。

完全图

- **无向完全图**：在无向图中，任意两个顶点之间都存在边。含有n个顶点的无向完全图有 $\frac{n(n-1)}{2}$ 条边。
- **有向完全图**：在有向图中，任意两个顶点之间都存在方向相反的两条弧。含有n个顶点的有向完全图有 $n(n-1)$ 条弧。

路径与连通性

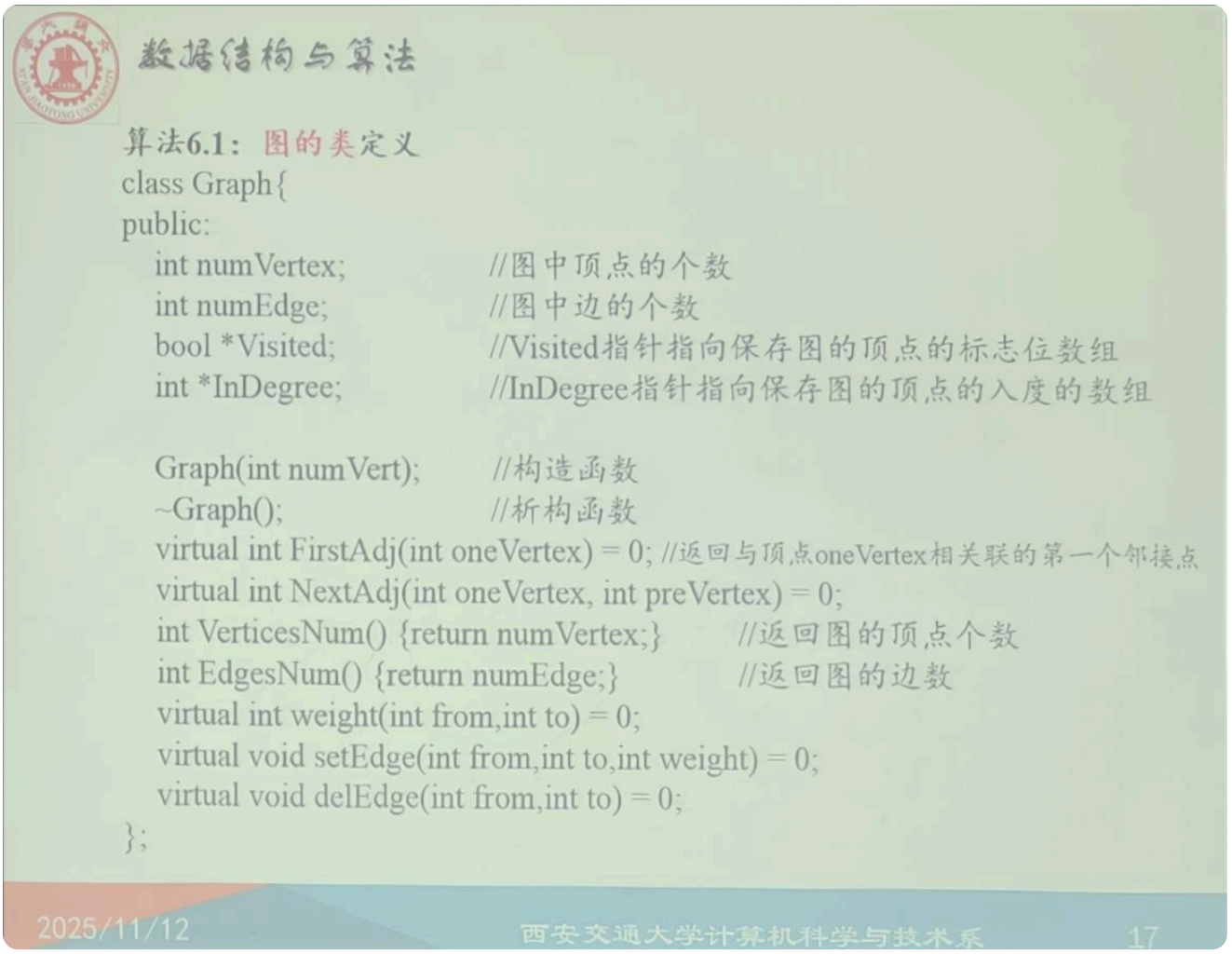
路径：

- **路径**：从顶点v到顶点w的一个顶点序列。
- **路径长度**：路径上边或弧的数目。
- **回路** 或 **环**：第一个顶点和最后一个顶点相同的路径。
  连通性：
- **连通图**：在**无向图**中，如果任意两个顶点之间都存在路径，则称该图是连通图。
- **强连通图**：在**有向图**中，如果对于每一对顶点v和w，都存在从v到w和从w到v的路径。
- **生成树**：一个连通图的极小连通子图，它含有图中全部n个顶点，但只有足以构成一棵树的$n-1$条边。

树与图的关系

树是一种特殊的图，可以看作是**无环的连通图**。

树是图的一种特殊情况，树是一种连通的无环图。当图称为树时，它的根不明确，所以将其称为自由树



图的主要操作：

1. 图的创建
2. 求邻接点
3. 顶点数与边数
4. Insert()与delete()
5. weight

6.2 图的存储结构

6.2.1 邻接矩阵存储方法

邻接矩阵是用一个一维数组存储顶点信息，一个二维数组（称为邻接矩阵）存储图中边或弧的信息的存储方式。

定义

假设图G=(V, E)有n个顶点，则邻接矩阵 `A` 是一个 `n*FDn` 的方阵，定义为：
对于无权图：

$$A[i][j] = \begin{cases} 1, & \text{若} < v_i, v_j > \text{是E中的边} \\ 0, & \text{反之} \end{cases}$$

对于带权图（网）：

$$A[i][j]=\begin{cases} w_{ij}, & \text{若} < v_i, v_j > \text{是E中的边} \\ \infty, & \text{若两顶点间没有边} \\ 0, & \text{若 } i=j \end{cases}$$

其中w_{ij}是边或弧的权值。∞通常用一个极大的值来表示。

存储结构

通常用一个结构体来封装顶点数组、邻接矩阵以及顶点数和边数。这里换成C++的 `std::vector` 来实现，更安全也更方便喵~

```
#include <vector>
struct MGraph {
    std::vector<char> Vex;                  // 顶点表
    std::vector<std::vector<int>> Edge;     // 邻接矩阵
    int vexnum, arcnum;                     // 当前顶点数和弧数
};
```

数据结构与算法

根据上述讨论，图的邻接矩阵的*数据结构*形式描述如下：

```
#define  MAX_VEX_NUM       <顶点个数>       // 最大顶点个数
typedef enum {DG, AG, WDG, WAG} GraphKind;     //有向, 无向, 带权有向, 带权无向
typedef struct AdjType {
    ArcValType    ArcVal;          // 若为无权图，用1或0表示是否邻接；
                                   // 若为带权图，则为权值类型。
    ArcInfoType   ArcInfo;         // 与弧（或边）相关的其它信息。
} AdjType;
typedef struct {
    GraphKind     kind;                          // 图的种类标志
    int           vexnum;                        // 图的当前顶点数
    Vextype       vexs[MAX_VEX_NUM];             // 顶点向量
    AdjType       adj[MAX_VEX_NUM][ MAX_VEX_NUM]; //邻接矩阵
} AdjGraph;
```

特点

优点：直观、简单，易于理解和实现。访问效率高

缺点：空间复杂度高：对于稀疏图，会浪费大量存储空间，空间复杂度为O(n^2)。

增删顶点不便：增加或删除顶点，效率低。

> 对于顶点集合相对稳定，而边的关系变动频繁的图，邻接矩阵是很不错的选择哦。

6.2.2 邻接表存储方法

邻接表是为图中每个顶点建立一个单链表，存储所有邻接于该顶点的顶点。我们用一个顶点数组来存放每个链表的头指针。

存储结构

这种方法需要两种数据结构：

1. **顶点表节点 (VNode)**：通常是一个数组，每个元素包含顶点的数据和指向第一条邻接边的指针。
2. **边表节点 (ArcNode)**：链表中的节点，代表一条边，包含邻接点在顶点表中的下标和指向下一条边的指针。

```
#include <vector>

// 边表节点
struct ArcNode {
    int adjvex;           // 该弧所指向的顶点的位置
    ArcNode *nextarc;     // 指向下一条弧的指针
    // int weight;        // 网的边权值可以放在这里
};

// 顶点表节点
struct VNode {
    char data;            // 顶点信息
    ArcNode *firstarc;    // 指向第一条依附该顶点的弧的指针
};

// 用邻接表表示的图
struct ALGraph {
    std::vector<VNode> vertices; // 顶点表，使用vector代替C风格数组
    int vexnum, arcnum;          // 图的当前顶点数和弧数
};
```

数据结构与算法

根据上述讨论，图的邻接表的*数据结构*形式描述如下：

```
#define   MAX_VEX_NUM     <顶点个数>          // 最大顶点个数
typedef enum {DG, AG, WDG, WAG} GraphKind;     //有向, 无向, 带权有向, 带权无向

typedef struct LinkNode {
    int                 Adjvex;          // 邻接点在头结点数组中的位置(下标)
    InfoType           Info;             // 与弧(或边)相关的信息
    struct LinkNode   *nextArc;          // 指向下一个表结点
} LinkNode;

typedef struct VexNode {
      VexType     Vertex;                  // 顶点数据信息
      LinkNode     *FirstArc;              // 指向第一个表结点
} VexNode;

typedef struct {
    GraphKind     kind;                   // 图的种类标志
    int           vexnum;                 // 图的当前顶点数
    VexNode       Adjlist[MAX_VEX_NUM];
} ALGraph;
```

6.2.3 图的边表存储结构

6.3 图的遍历

图的遍历可能出现两个问题：

1. 回路问题：遍历过程中可能会反复经过同一个顶点。
2. 连通问题：如果图不是连通的，从一个顶点出发无法访问到所有顶点。

> 为了解决这些问题，我们需要设置一个辅助数组 visited[] 来标记已访问过的顶点。图的遍历实质就是**查找每个顶点的邻接点**的过程。通常有**DFS**和**BFS**两种方法，对于有向图和无向图都适用。

6.3.1 深度优先搜索 (DFS)

基本思想：
深度优先搜索（DFS）类似于树的先序遍历。从图中某个顶点v出发，访问该顶点，然后选择一个与v邻接且未被访问过的顶点w，再从w出发进行深度优先遍历，直到一个路径走到底，再回溯到上一个顶点，继续寻找其他未被访问的邻接点。

```
#include <vector>

// 假设 ALGraph 结构体已定义
// visit() 函数用于访问顶点
```

```cpp
void DFS(const ALGraph& G, int v, std::vector<bool>& visited) {
    visited[v] = true;
    // visit(G.vertices[v]); // 访问顶点v

    // 遍历v的所有邻接点
    for (ArcNode* p = G.vertices[v].firstarc; p != nullptr; p = p->nextarc) {
        int w = p->adjvex;
        if (!visited[w]) {
            DFS(G, w, visited); // 对w进行递归调用
        }
    }
}

// 对整个图进行深度优先遍历
void DFSTraverse(const ALGraph& G) {
    // 初始化访问标记数组
    std::vector<bool> visited(G.vexnum, false);
    // 遍历所有顶点，防止图不连通
    for (int v = 0; v < G.vexnum; ++v) {
        if (!visited[v]) {
            DFS(G, v, visited);
        }
    }
}
```

思考：

1. 如何改为一重循环？
2. 时间和空间的最好情况和最坏情况？

时间复杂度：
- **邻接矩阵**：每个顶点都需要入队出队一次，并且查找其邻接点，时间复杂度为 $O(n^2)$。
- **邻接表**：每个顶点入队一次，每条边访问一次，时间复杂度为 O(n+e)。

空间复杂度：需要一个辅助队列，最坏情况下所有顶点同时在队列中，空间复杂度为 O(n)。

6.3.2 广度优先搜索 (BFS)

基本思想：
广度优先搜索（BFS）类似于树的层序遍历。从图中某个顶点v出发，访问v后，依次访问v的所有未被访问过的邻接点，然后再按照这些邻接点被访问的顺序，逐一访问它们各自的邻接点。

```cpp
#include <vector>
#include <queue>

// 假设 ALGraph 结构体已定义
// visit() 函数用于访问顶点，这里省略其具体实现

// 从顶点v开始广度优先遍历单个连通分量
void BFS(const ALGraph& G, int v, std::vector<bool>& visited) {
    std::queue<int> q;
    // visit(G.vertices[v]);      // 访问v
    visited[v] = true;
    q.push(v);                 // v入队
    while (!q.empty()) {
        int u = q.front();         // 队头元素出队
        q.pop();
        // 遍历u的所有邻接点
        for (ArcNode* p = G.vertices[u].firstarc; p != nullptr; p = p->nextarc) {
            int w = p->adjvex;
            if (!visited[w]) {
                // visit(G.vertices[w]); // 访问w
                visited[w] = true;
                q.push(w);         // w入队
            }
        }
    }
}

// 对整个图进行广度优先遍历
void BFSTraverse(const ALGraph& G) {
    // 初始化访问标记数组
    std::vector<bool> visited(G.vexnum, false);
    // 遍历所有顶点，防止图不连通
    for (int v = 0; v < G.vexnum; ++v) {
        if (!visited[v]) {
            BFS(G, v, visited);
        }
    }
}
```

6.4 图的应用：拓扑排序

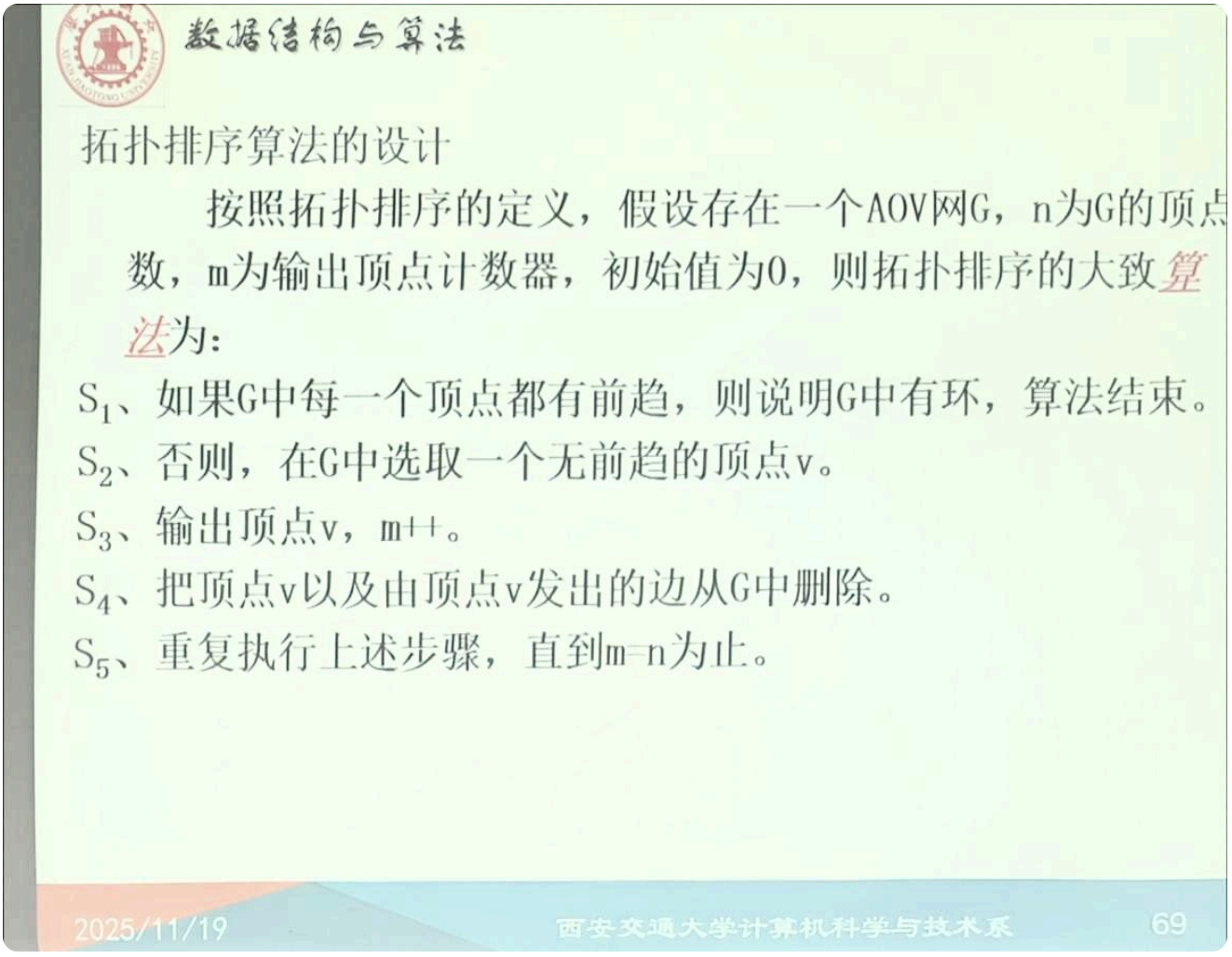
> 把子项目，工序或课程看成是图中的一个顶点，称之为**活动**。
> 用图中的**有向边表示各种活动的先后关系**。若存在<u,v>则表示活动u一定在v之前进

> 行。这样的有向图称为**顶点表示活动的网**，简称 **AOV网**。AOV网中不应存在任何回路，即它必须是一个**有向无环图 DAG**。

拓扑排序的定义

拓扑排序是对一个有向无环图（DAG）的顶点进行排序，使得对图中每一条有向边 `(u, v)`，顶点 `u` 都排在顶点 `v` 的前面。最终得到的是一个顶点的线性序列。这个序列通常不是唯一的。

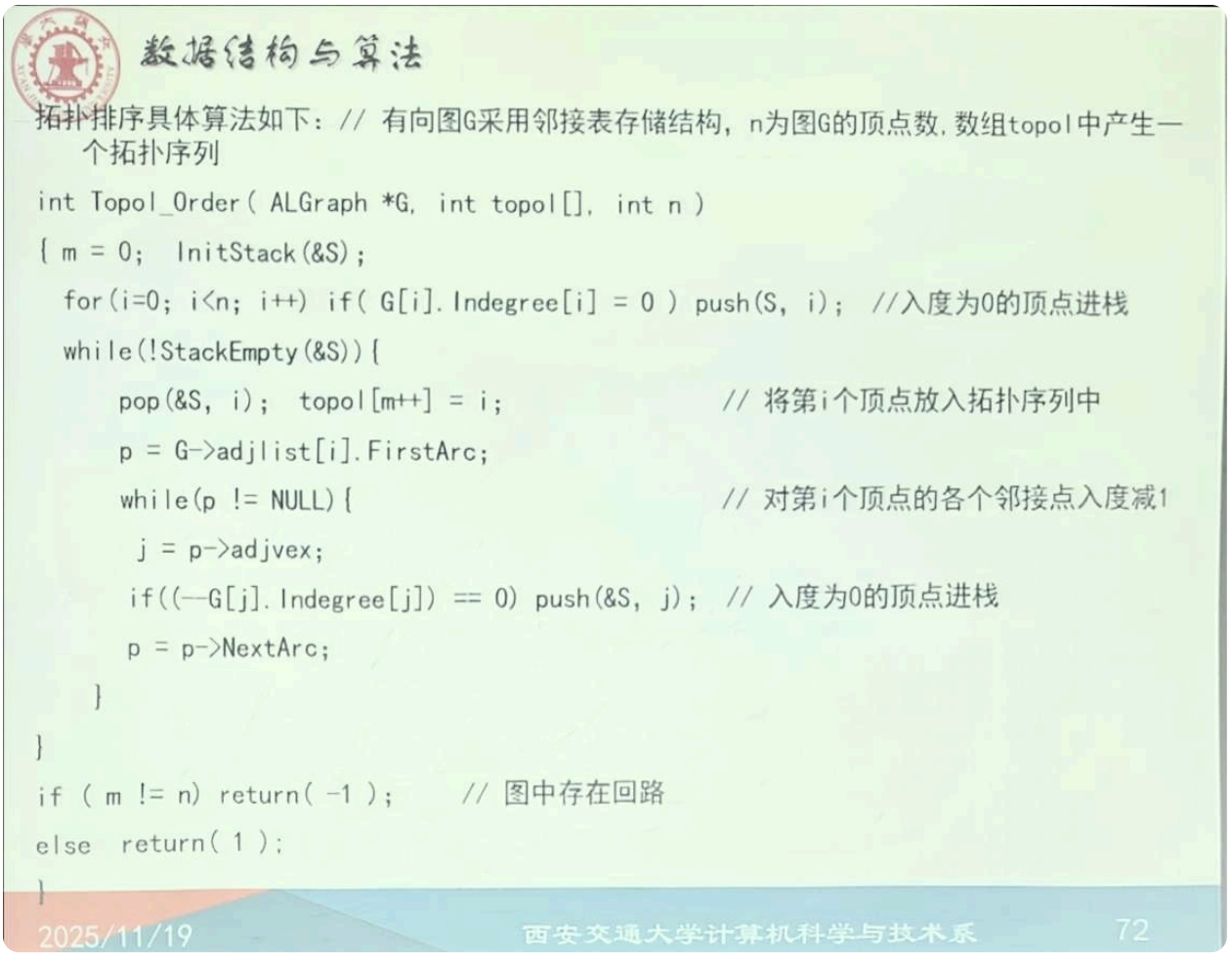
拓扑排序算法设计



优化：

1. 使用临时缓冲区存储入度为0的结点
2. 采用邻接表存储有向图

3. 数据结构添加入度



6.5 图的应用：关键路径（不作要求）

核心思想是并行操作的完成时间.

6.6 图的应用：图的最小生成树

对于一个带权的连通无向图G=(V, E)，**最小生成树**是其一个子图，如果这个子图包含G中所有的顶点，并且是一棵树，同时所有边的权值之和最小。

一个连通无向图的生成树是不唯一的，但最小生成树的权值之和是唯一的。

Prim算法

基本思想：从一个初始顶点开始，逐步将新的顶点加入到正在生长的树中。每一步都选择一个与当前树相连且权值最小的边，直到所有顶点都被包含进来。这是一种"加点"的策略。

Kruskal算法

基本思想：将图中所有的边按权值从小到大排序。依次考察每条边，如果这条边连接的两个顶点当前不属于同一个连通分量（即加入这条边不会形成环），就将它加入到生成树中。这是一种"加边"的策略。

上一章 第5章 树与二叉树
下一章 第7章 排序算法