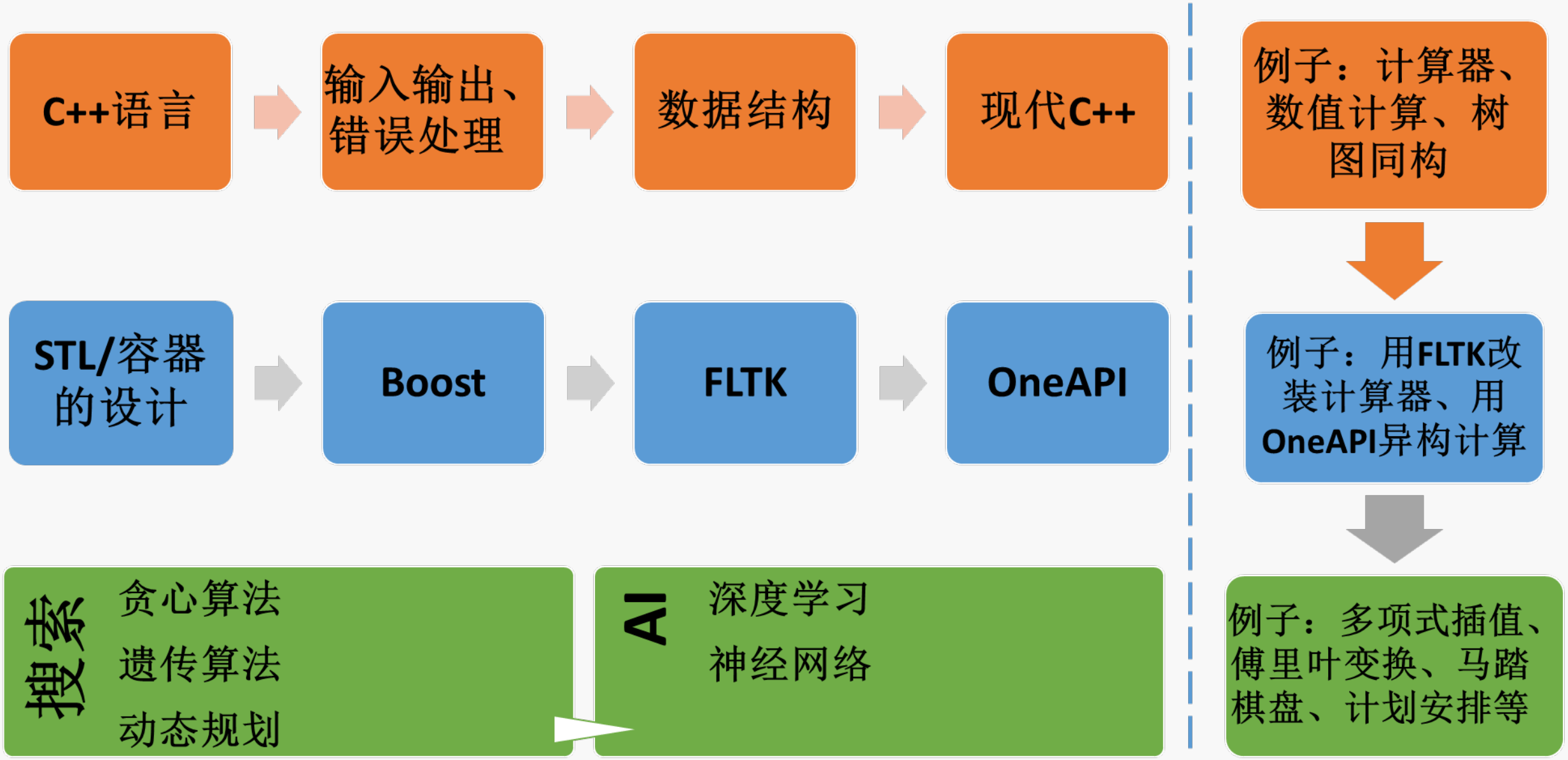

C++语言
输入输出、错误处理
数据结构
现代C++
STL/容器的设计
Boost
FLTK
OneAPI
贪心算法
遗传算法
动态规划
AI
深度学习
神经网络
例子：计算器、数值计算、树图同构
例子：用FLTK改装计算器、用OneAPI异构计算
例子：多项式插值、傅里叶变换、马踏棋盘、计划安排等

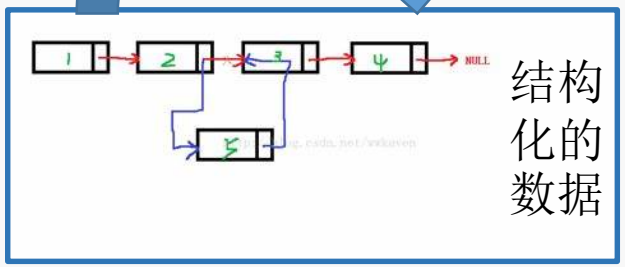
问题求解与实践
——数据结构的基本概念

主讲教师：陈雨亭、沈艳艳

数据结构的角色

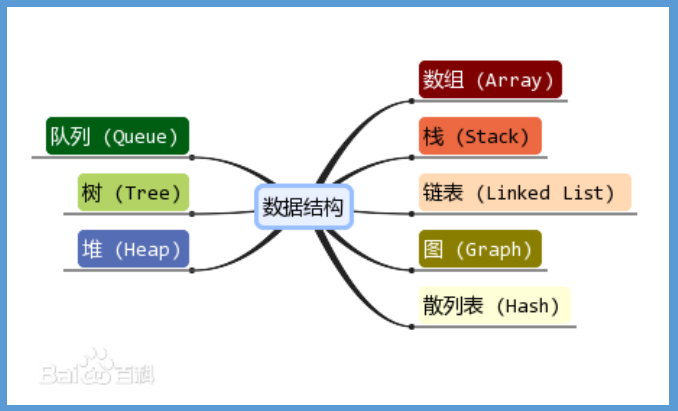
凌乱的数据

数据结构上的操作



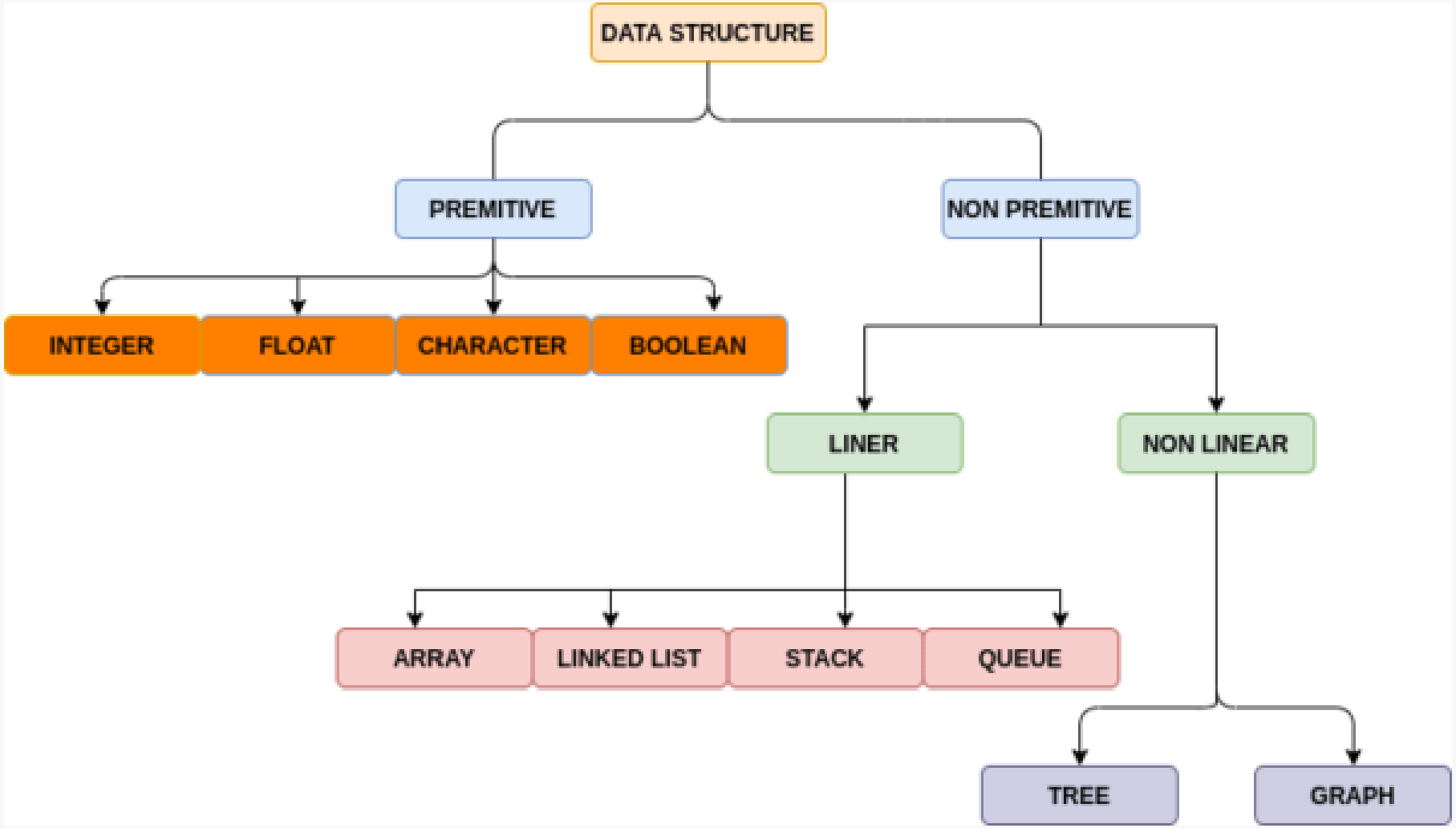
结构化的数据

数据结构

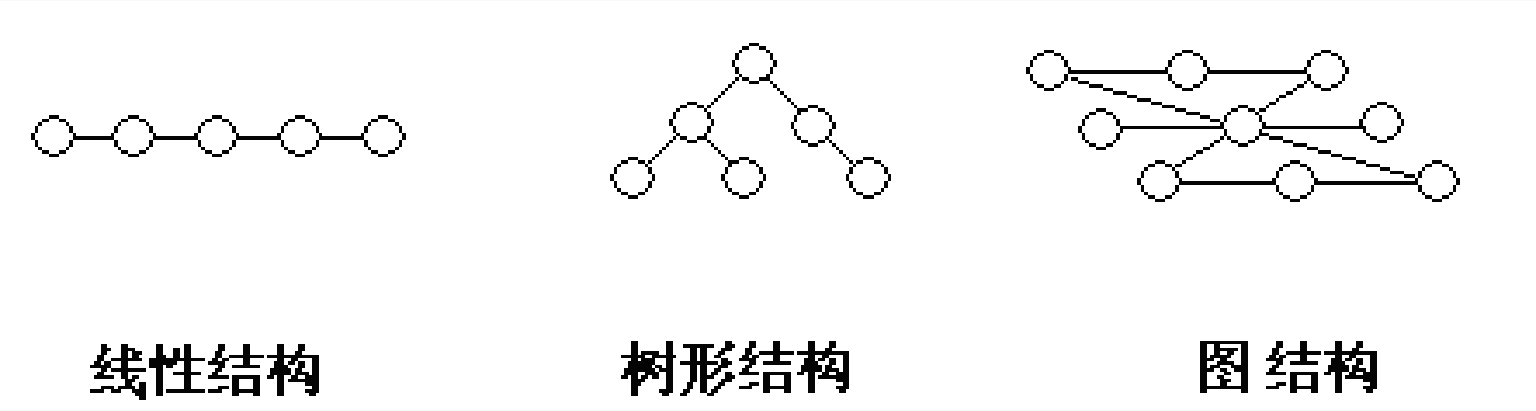


保存数据

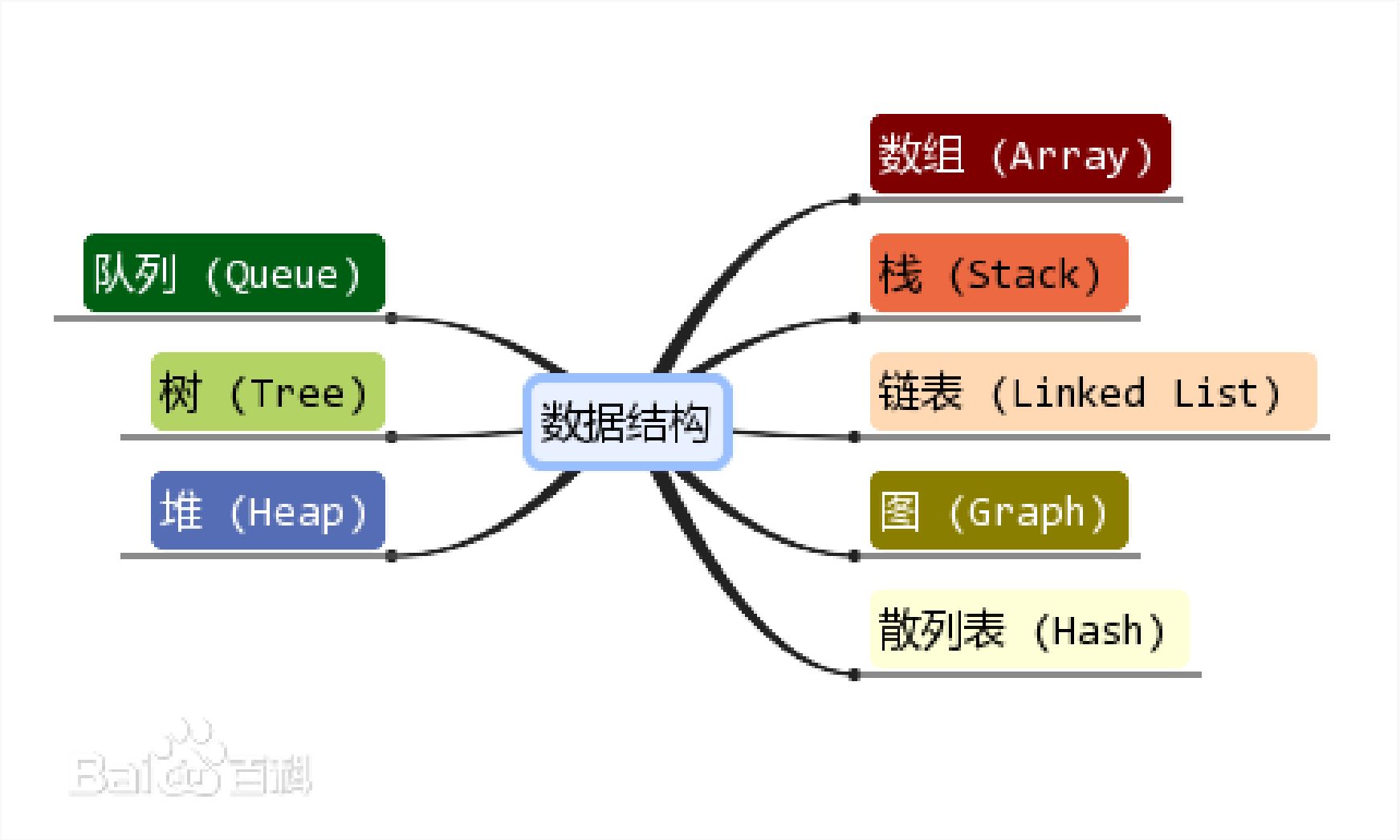
计算结果


DATA STRUCTURE
PREMITIVE
NON PREMITIVE
INTEGER
FLOAT
CHARACTER
BOOLEAN
LINER
NON LINEAR
ARRAY
LINKED LIST
STACK
QUEUE
TREE
GRAPH

三种基本的逻辑结构



基本数据结构


数组（Array）
栈（Stack）
链表（Linked List）
图（Graph）
散列表（Hash）
数据结构
队列（Queue）
树（Tree）
堆（Heap）

数据的运算

- **数据的运算**是对数据元素进行的某种操作，例如
  - 改变元素的个数（增加或删除）
  - 改变元素的顺序
  - 改变元素之间的关系
  - 浏览每个元素（遍历）
  - 检索符合某个条件的数据元素

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

问题求解与实践
——线性表

主讲教师：陈雨亭、沈艳艳

线性表

- **线性表**（Linear List）
  - 是由有限个相同类型的数据元素组成的有序序列，一般记作（$a_1,a_2,...,a_n$）
- 特点
  - 除了a_1和a_n之外，任意元素a_i都有一个直接前趋a_{i-1}和一个直接后继a_{i+1}
  - a_1无前趋
  - a_n无后继
- 表的长度：线性表中数据元素的个数
- 空表：元素个数为0的表

线性表的运算

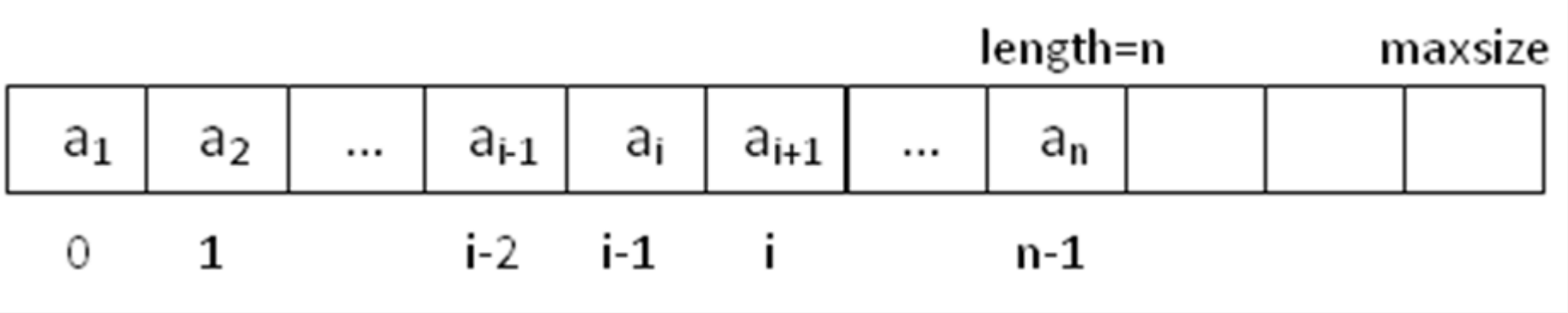
- 在线性表中，经常执行下列操作
  - 确定线性表是否为空
  - 确定线性表的长度
  - 查找某个元素
  - 删除第i个元素
  - 在第i个位置插入一个新元素

线性表的存储结构

- 采用顺序存储结构的称为**顺序表**
- 采用链式存储结构的称为**线性链表**

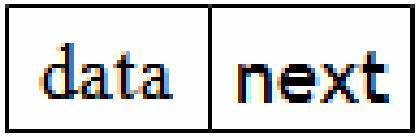
顺序表

- 顺序表中的数据元素按照逻辑顺序依次存放在一组连续的存储单元中
- 每个元素a_i的存储地址是该元素在表中位置i的线性函数



线性链表

- **线性链表**
  - 采用链式存储的线性表
- 存储特点：每个结点都分两部分
  - **数据域**：存储元素的值
  - **指针域**：存放直接前趋或直接后继元素的地址信息

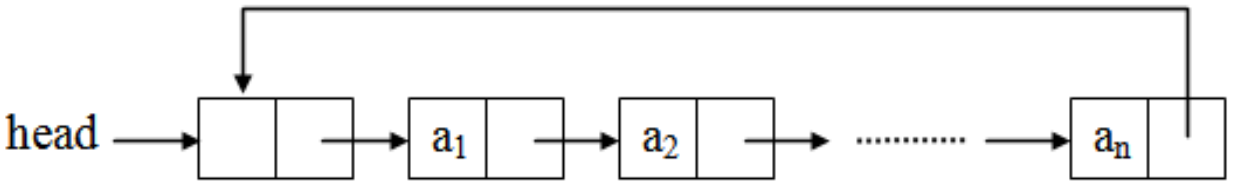


链表的存储结构

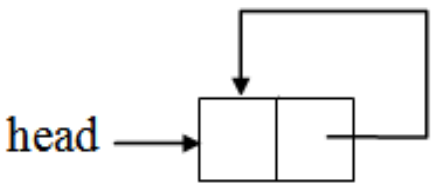
	存储地址	数据域	指针域
	20H	a_2	80H
head		…	…
40H	40H		90H
		…	…
	80H	a_3	NULL
	90H	a_1	20H

其它形式的链表

- **单向循环链表**
  - 将单链表尾结点的指针由NULL改为指向头结点，首尾连接形成一个环形，简称为**循环链表**



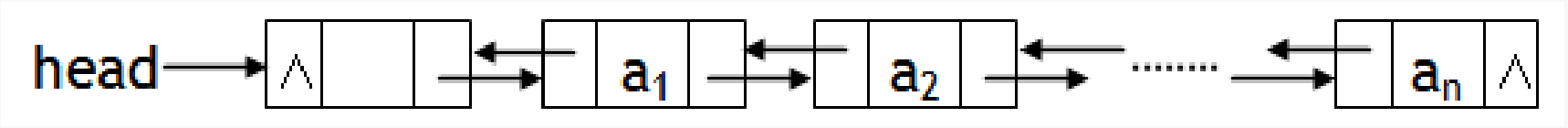
(a) 带有头节点的循环链表



(b) 空循环链表

其它形式的链表

- **双向链表**
  - 每个结点的指针域中再增加一个指针，使其指向该结点的直接前趋结点
  - 这样构成的链表中有两个不同方向的链



其它形式的链表

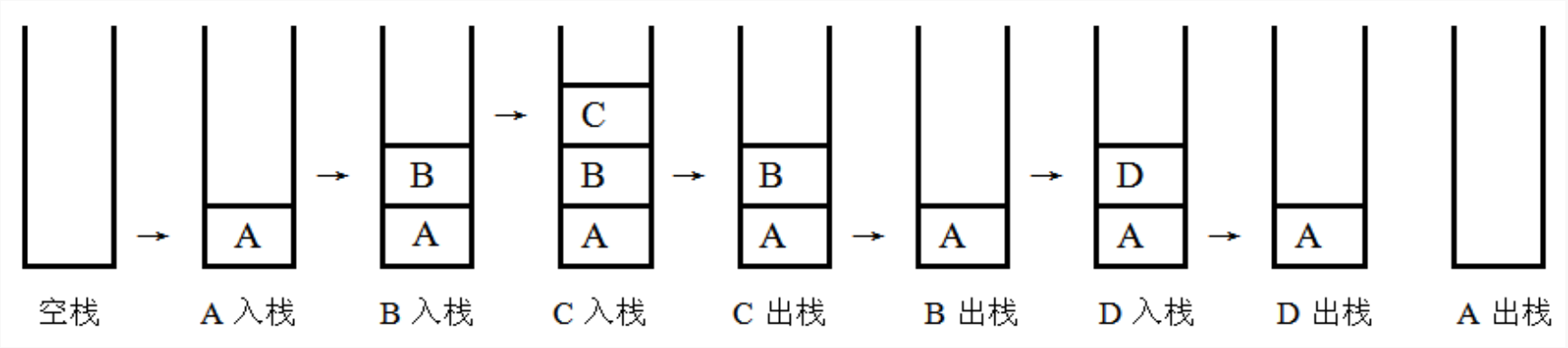
- **双向循环链表**
  - 将双向链表的头结点的前趋指针指向尾结点
  - 将尾结点的后继指针指向头结点

栈

- **栈**(Stack)
  - 只能在一端进行插入和删除操作的特殊线性表
  - 允许进行插入和删除操作的一端称为**栈顶**，另一端称为**栈底**
- 你举出一些栈的生活实例吗？

栈

- 栈的特点：
  - **后进先出**（LIFO, last in, first out ）或**先进后出**（FILO）

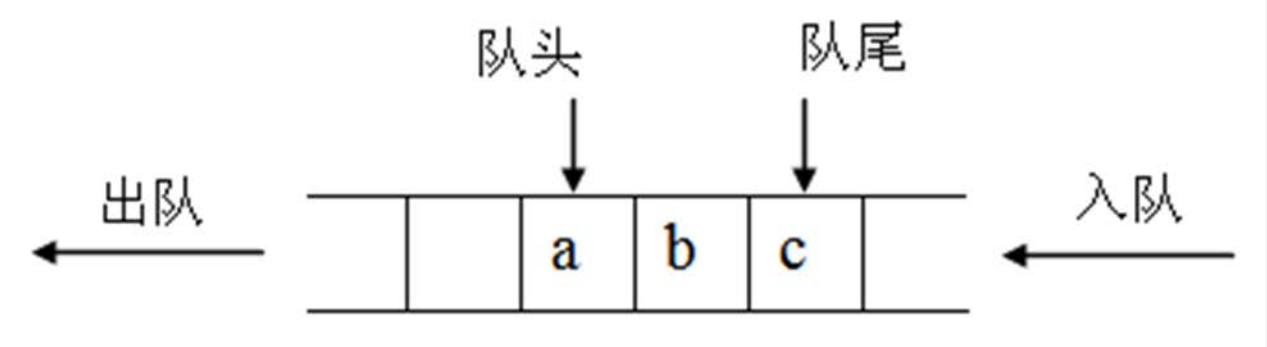


栈的主要操作

- **创建空栈**
- **入栈**（push）：也称为进栈、压栈，在栈顶加入新的元素
- **出栈**（pop）：也称退栈或弹栈，将栈顶元素删除
- **读栈顶元素**：只读出栈顶元素，但不出栈

队列

- **队列**（Queue）
  - 只能在表的一端进行插入操作、在另一端进行删除操作的特殊线性表
  - 允许删除元素的一端称为**队头**，允许插入元素的一端称为**队尾**
- 特点
  - **先进先出**（FIFO）或**后进后出**（LILO）

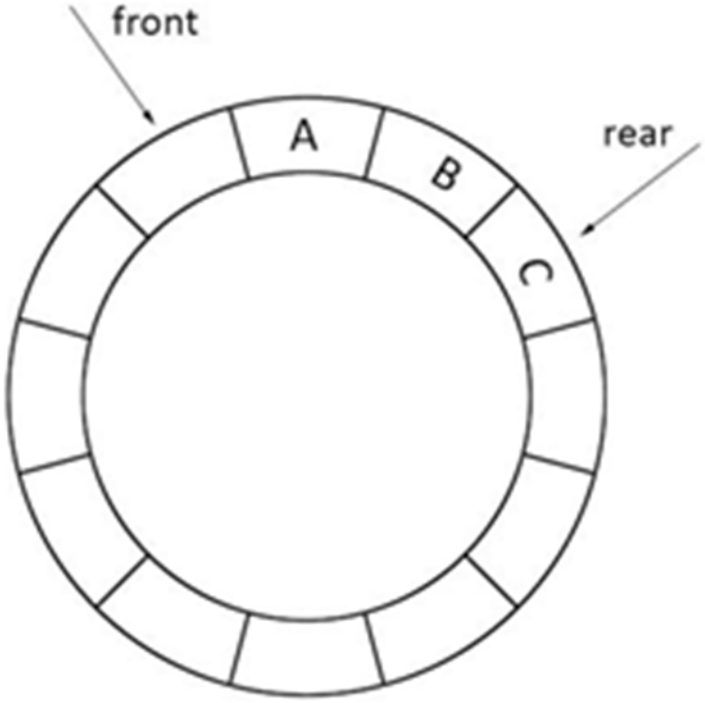


队列的主要操作

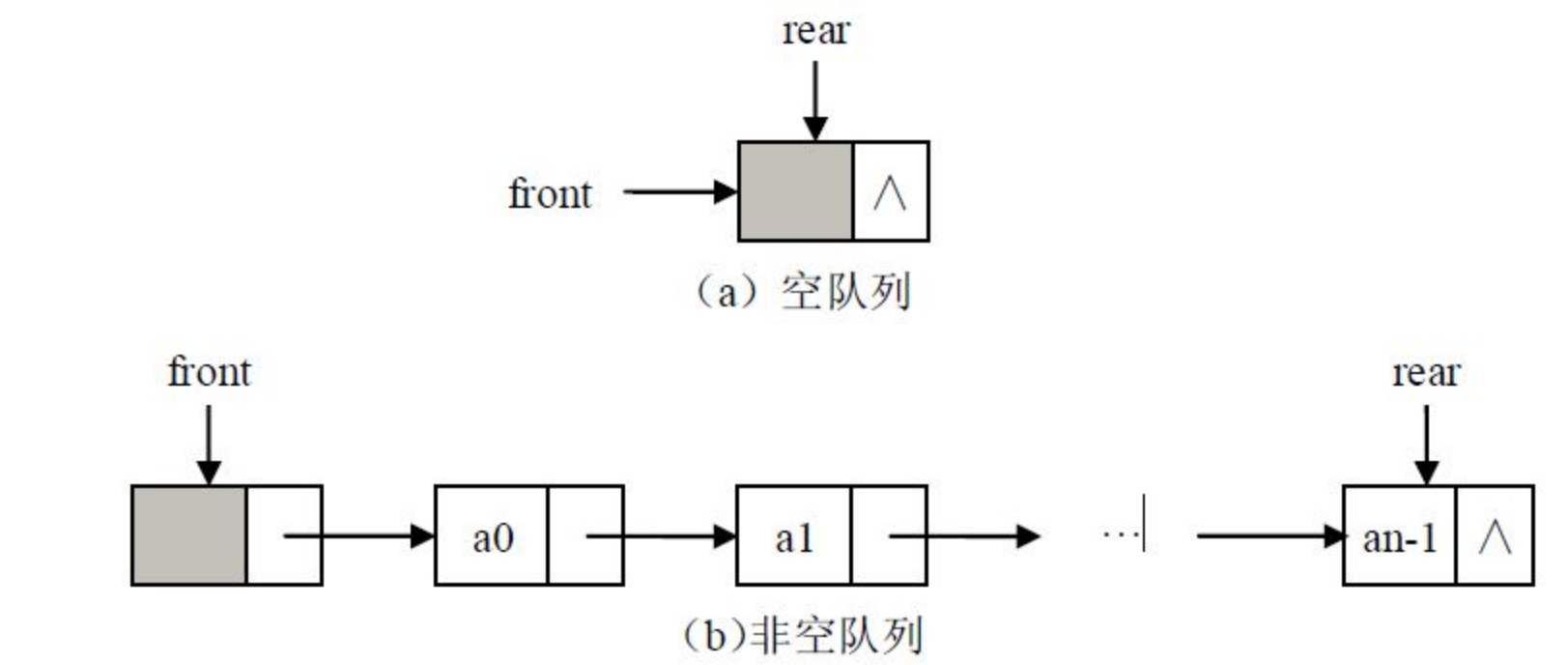
- 队列的主要操作
  - 判断队列是否满
  - 判断队列是否为空
  - **入队**：在队尾插入元素
  - **出队**：在队头删除元素
  - 取队头元素

循环队列

- 将队列的头尾相连形成一个圆圈
- 当队尾和队头重叠时，队列为空还是满呢?
- 约定：当队头和队尾相等时，队空。当队尾加1后等于队头时，队满



链式队列



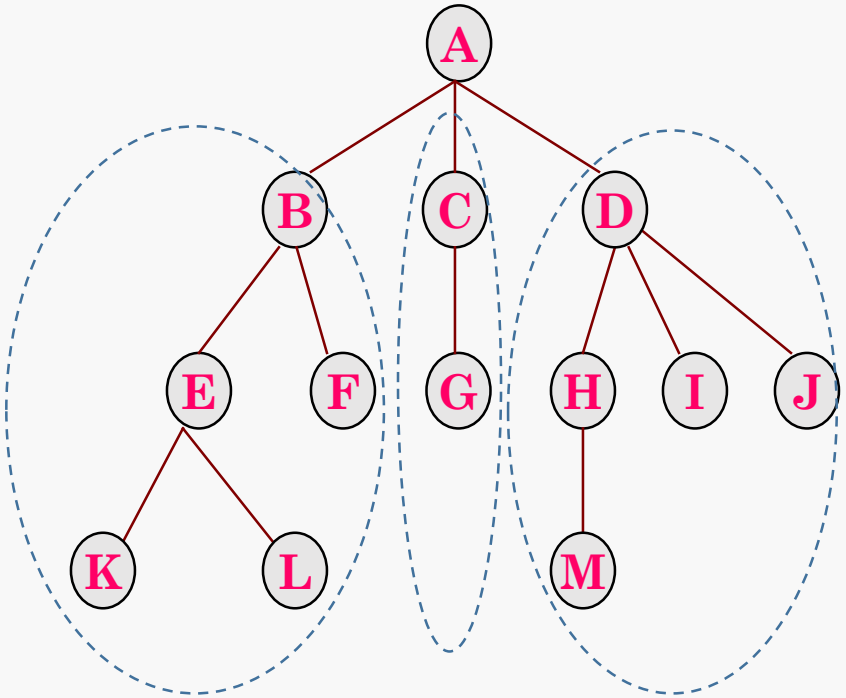
问题求解与实践
——树和二叉树

主讲教师： 陈雨亭、沈艳艳

树的定义

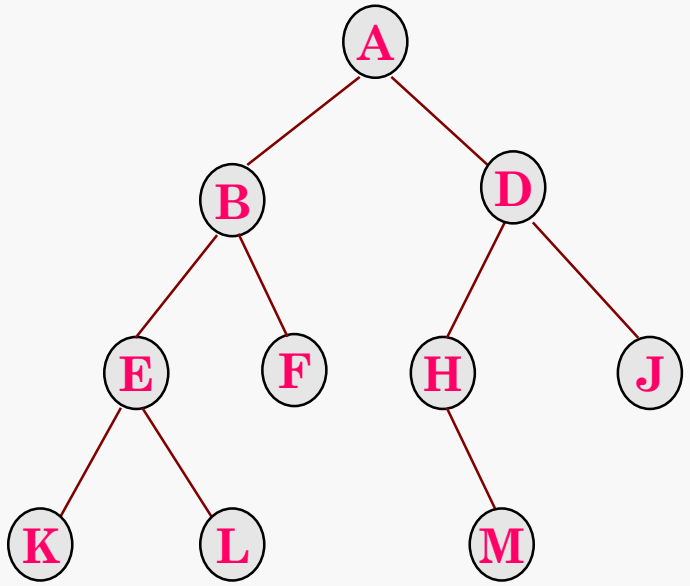
树是一个或多个结点组成的有限集合T，有一个特定结点称为根，其余结点分为m（m≥0）个互不相交的集合T1，T2，…,Tm。每个集合又是一棵树，被称为这个根的子树。

一般的树



二叉树

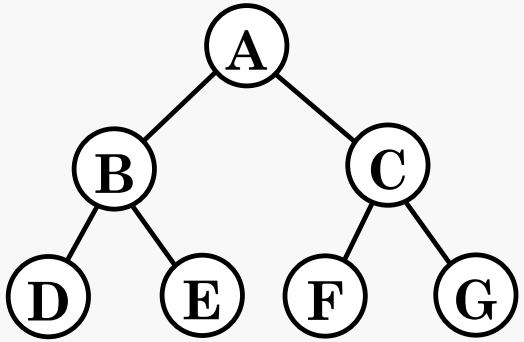
- 二叉树是每个节点最多有两个子树的树结构



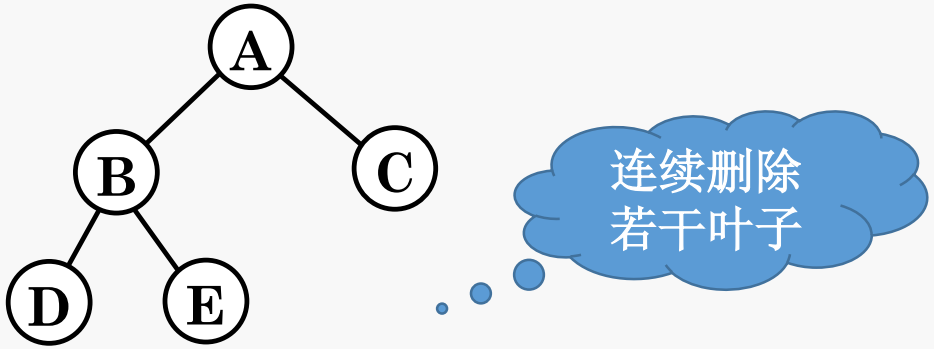
- 二叉树是有序树，结点的子树分别称为根的左子树和右子树

特殊形式的二叉树

- 满二叉树：当二叉树每个分支结点的度都是2，且所有叶子结点都在同一层上，则称其为满二叉树。
- 完全二叉树：从满二叉树叶子所在的层次中，自右向左连续删除若干叶子所得到的二叉树被称为完全二叉树。满二叉树可看作是完全二叉树的一个特例。



满二叉树



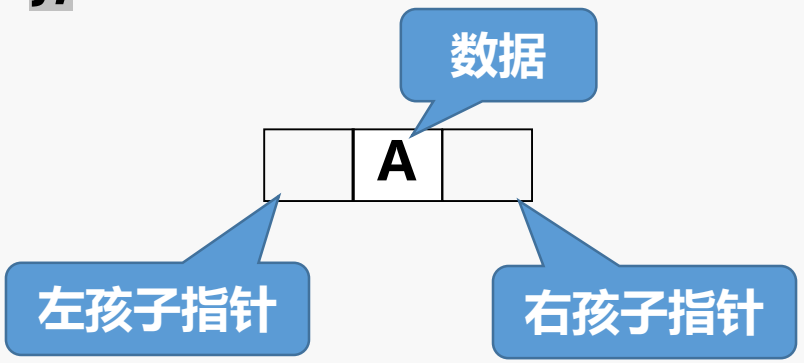
完全二叉树

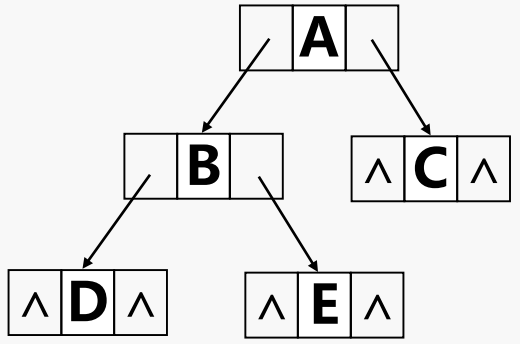
二叉树的实现——链式存储

- 二叉树是一种非线性数据结构，描述的是元素间一对多的关系，这种结构最常用、最适合的描述方法是用链表的形式
- 首先定义结点

  每个结点都包含一个数据域和两个指针域。一般可采用下面的形式定义结点：

```
struct BinTreeNode {
    char data;                    // 这里以字符型的数据为例
    struct BinTreeNode  *leftChild,  *rightChild;
};
```





二叉树的实现——链式存储

◆ 定义一颗二叉树就是定义一个空树，也就是定义一个空指针，可描述如下：

```
BinTreeNode *root;//定义根结点指针
root=NULL;          //定义空树
```

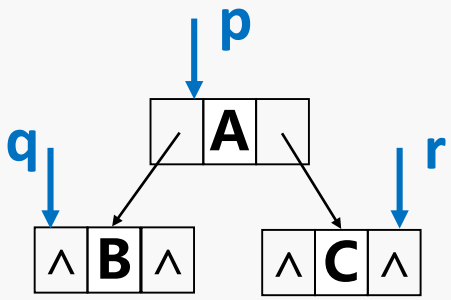
◆ 新建一个结点

```
BinTreeNode  *p = new BinTreeNode;
p->data = ‘A’ ;     //给数据域赋值
p->leftChild=NULL;          //左子树为空
p->rightChild=NULL;         //右子树为空
```

必须通过结点指针操作

◆ 建立二叉树

```
…… 建立结点A、B、C（方法如上面所示）
…… 指针p、q、r 分别指向结点A、B、C
p->leftChild=q;     //左孩子为B
p->rightChild=r;    //右孩子为C
```



完全二叉树的一个特性

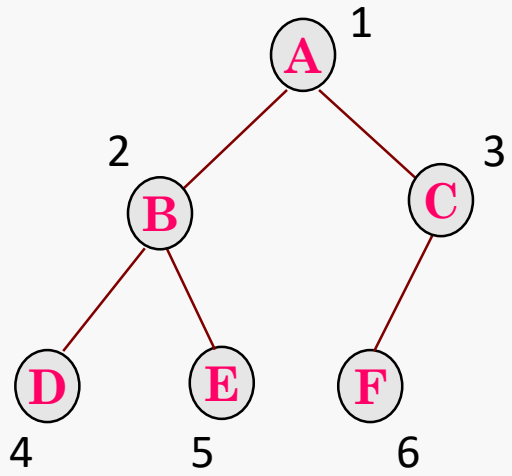
将完全二叉树的每个结点从上到下、每一层从左至右进行1至n的编号

- 性质：

① 若i=1，则该结点是二叉树的根，否则，编号为[i/2]的结点为结点 i 的父结点；

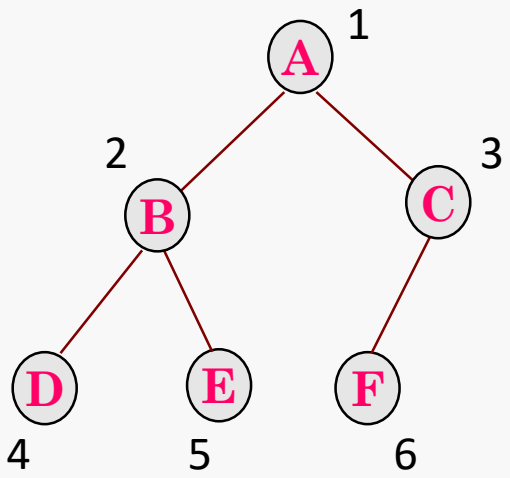
② 若2*i>n，则该结点无左孩子。否则，编号为2*i的结点为结点 i 的左孩子；

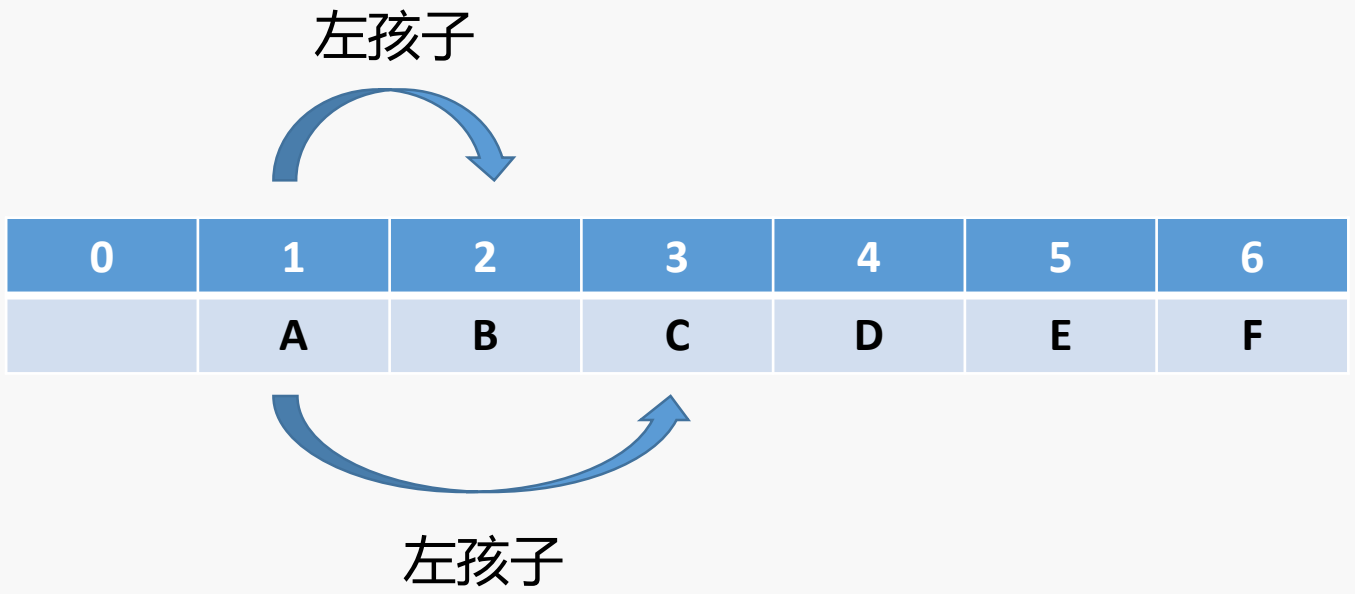
③ 若2*i+1>n，则该结点无右孩子。否则，编号为2*i+1的结点为结点 i 的右孩子。



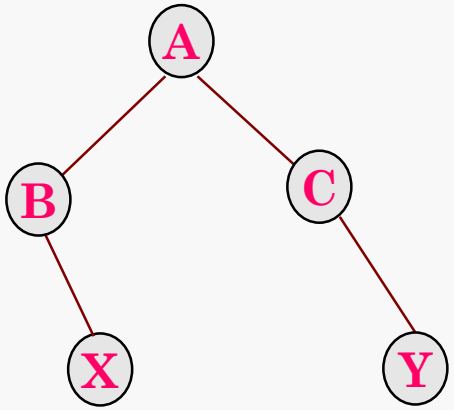
完全二叉树的顺序存储

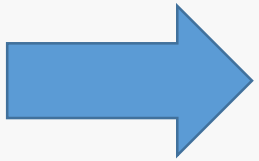
可以用一维数组存储：空出数组下标为0的位置，将结点存储在**下标为其编号的位置**。

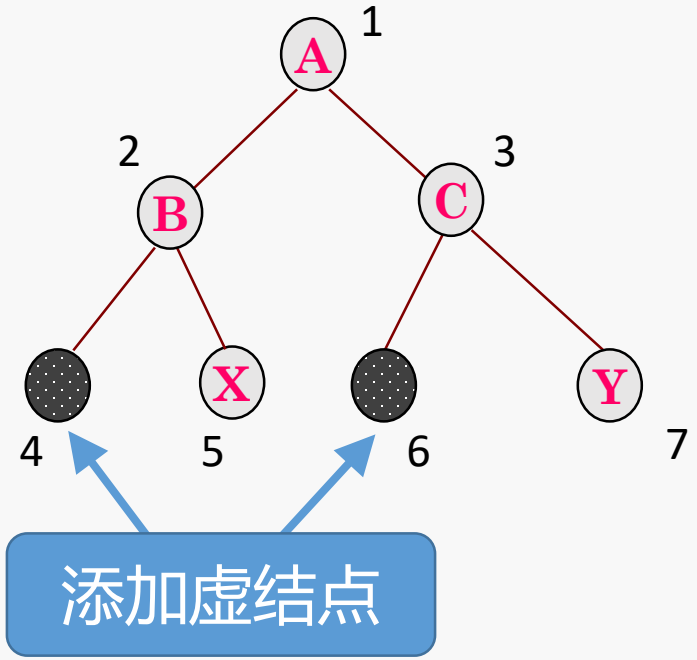




二叉树转换为完全二叉树后存储







存储情况

0	1	2	3	4	5	6	7
	A	B	C	#	X	#	Y

生成一个二叉树

可以有很多方式生成一个二叉树，这里我们讨论的问题为：

根据数组中存储的完全二叉树，生成一个链式结构的二叉树

比如将下面数组转换为链式结构的二叉树

0	1	2	3	4	5	6	7
	A	B	C	#	X	#	Y

其中#代表虚结点

生成一个二叉树

- **算法：**

  下标为1的元素作为根结点生成；

  依次处理数组中的其他元素：下标为 i 的元素，若 i 为偶数就作为 [i/2] 结点的左孩子，若 i 为奇数就作为 [i/2] 结点的右孩子生成；

  遇到虚元素就跳过，继续处理下一个元素。

- **问题：**

  1. 依次处理数组中元素，循环结束条件是什么？

     方法一：预先得到结点总数n（包括虚结点），循环n次

     方法二：在数组末尾加一个特殊符号（比如$），循环遇到该符号则停止

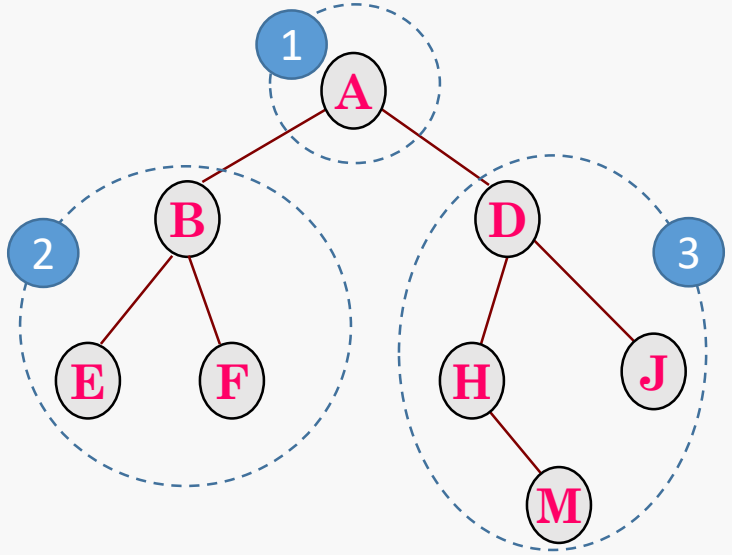
  2. 如何让结点 i 和父结点 [i/2] 连接起来？

     在链式结构中，由于对结点的操作要通过指针进行，所以先要得到结点 [i/2]和结点 i 的指针p和q，而后可利用p->leftChild=q或p->rightChild=q连接即可

二叉树的三种遍历方式

1. 先序遍历

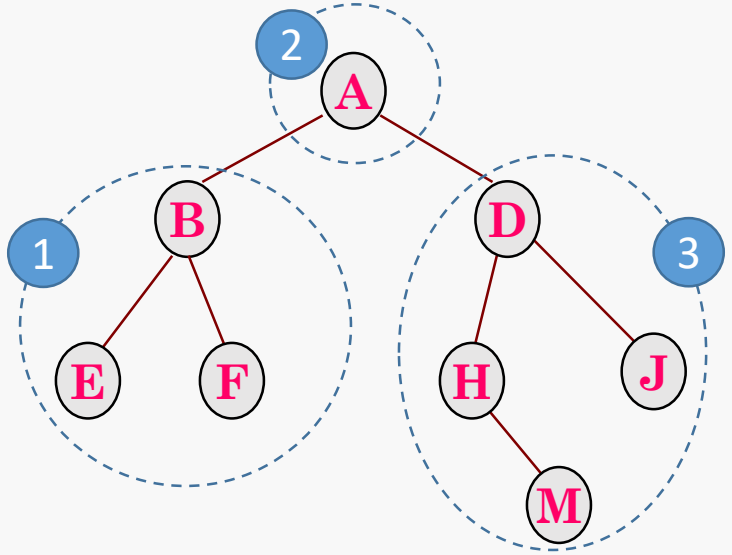
首先访问根，然后按先序遍历方式访问左子树，再按先序遍历方式访问右子树



二叉树的三种遍历方式

2. 中序遍历

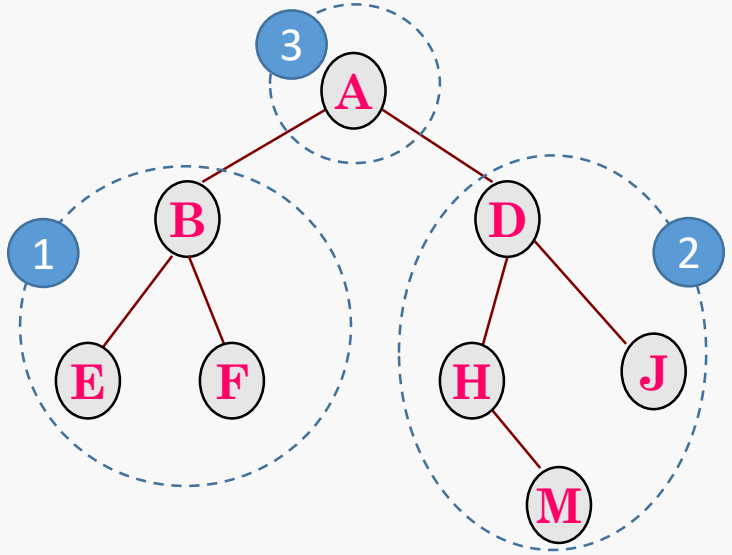
首先按中序遍历访问左子树，再访问根，最后按中序遍历方式访问右子树



二叉树的三种遍历方式

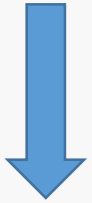
3. 后序遍历

首先按后序遍历访问左子树，再按后序遍历方式访问右子树，最后访问根



二叉树先序遍历的实现

1. 首先访问根
2. 然后按先序遍历方式访问左子树
3. 再按先序遍历方式访问右子树

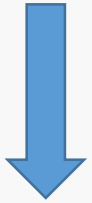


将以上步骤写成一个函数

```
先序遍历（根指针p）                    //只能通过根的指针进入二叉树
{
    visit（p结点）；                   // p结点——指针p指向的结点
    先序遍历（p->leftChild）；          // 先序遍历左子树
    先序遍历（p->rightChild）;// 先序遍历右子树
}
```

二叉树先序遍历的实现

1. 首先访问根
2. 然后按先序遍历方式访问左子树
3. 再按先序遍历方式访问右子树



将以上步骤写成一个函数

这个递归函数能终止吗?

```
PreOrder（指针p）                    //访问以p为根的二叉树
{
    visit（p结点）；                 // 指针p指向的结点
    PreOrder （p->leftChild）；      // 先序遍历左子树
    PreOrder（p->rightChild）；      // 先序遍历右子树
}
```

二叉树先序遍历的实现

先序遍历算法如下：

```
void PreOrder(BinTreeNode *p)  {
    if (p) {
        Visit(p );                          //访问根结点
        PreOrder(p->leftChild );      //先序遍历左子树
        PreOrder(p->rightChild );    //先序遍历右子树
    }
}
```

若遇到空指针则返回上一层函数

哈夫曼树

- **什么是路径、路径长度?**

  在一棵树中，从一个结点往下到另一个结点之间的通路，称为路径。通路中分支的数目称为路径长度。

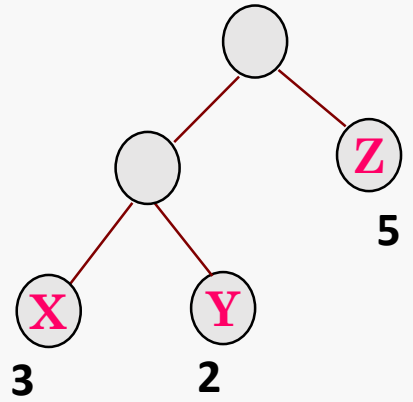
- **二叉树带权路径长度**

  设二叉树有n个带有权值的叶子结点，每个叶子到根的路径长度乘以其权值之和称为二叉树带权路径长度。记作：

$$WPL = \sum_{i=1}^{n} w_i * l_i$$

w_i —第 i 个叶子的权重

l_i —第 i 个叶子到根的路径长度



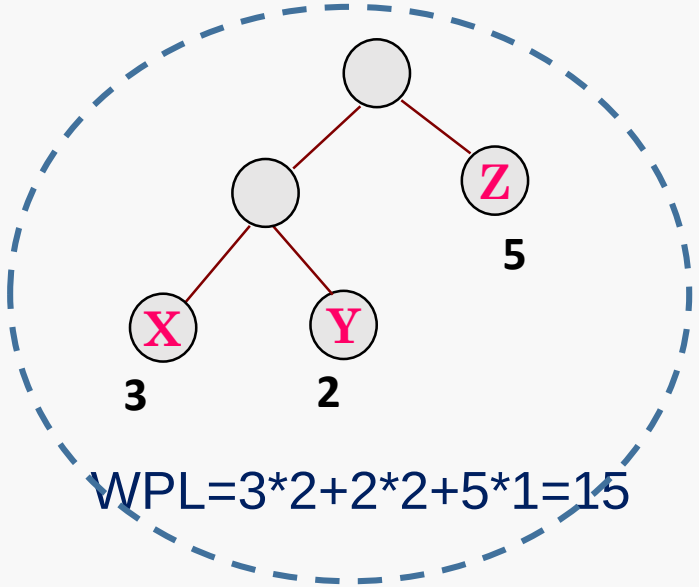
WPL=3*2+2*2+5*1=15

哈夫曼树

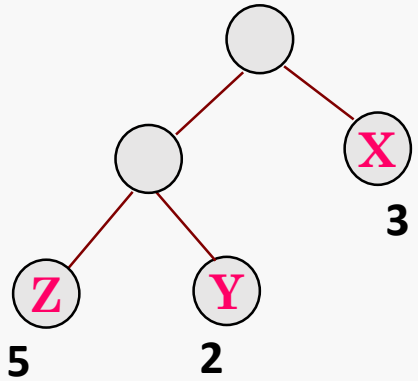
- **什么哈夫曼树?**

  以一些带有固定权值的结点作为叶子所构造的，具有最小带权路径长度的二叉树。

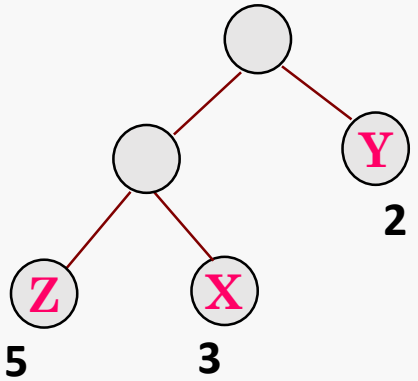
  设X、Y、Z权值为3、2、5，可以构造多种叶子含权的二叉树，例如



WPL=3*2+2*2+5*1=15



WPL=5*2+2*2+3*1=17



WPL=5*2+3*2+2*1=18

哈夫曼树的构造过程

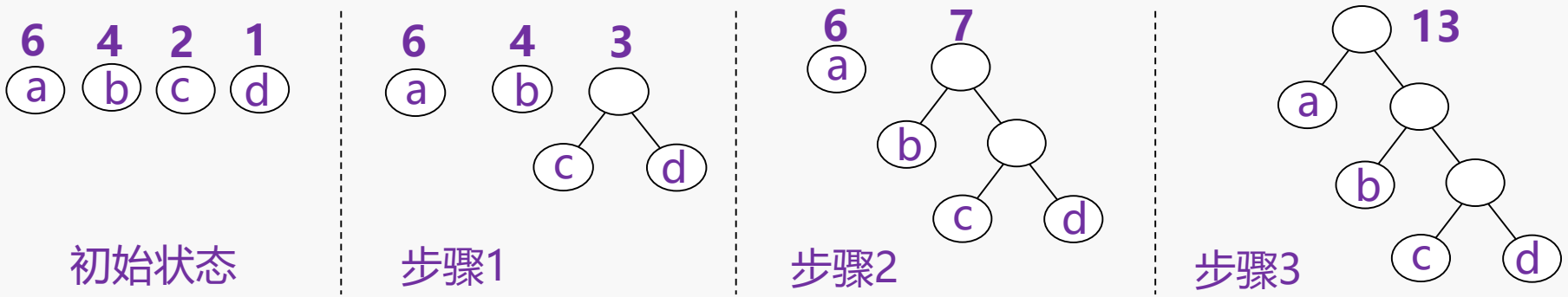
假定有n个具有权值的结点，则哈夫曼树的构造算法如下：

① 根据n个权值，构造n棵二叉树，其中每棵二叉树中只含一个权值为w_i的根结点；

② 在所有二叉树中选取根结点权值最小的两棵树，分别作为左、右子树构造一棵新的二叉树，这棵新的二叉树根结点权值为其左、右子树根结点的权值之和；删去原来的两棵树，留下刚生成的新树；

③ 重复执行②，直至最终合并为一棵树为止。

假定有a、b、c、d四个字符，它们的使用权重比为6：4：2：1



哈夫曼树与哈夫曼编码

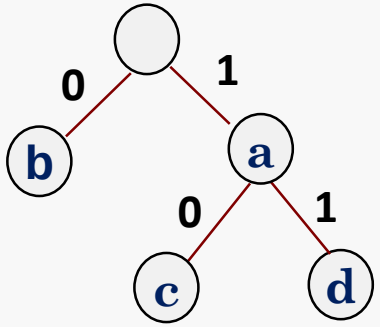
问题

假定有一段报文由a、b、c、d四个字符构成，它们的使用频率比为6：4：2：1，请构造一套二进制编码系统，使得报文翻译成二进制编码后**无二义性**且**长度最短**

任一编码方案

a -1 b-0 c-10 d-11

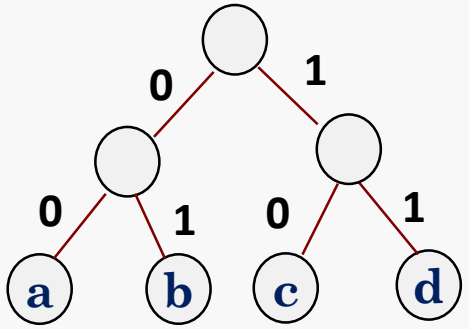
一一对应



无二义性要求任一编码不能是另一个编码的前缀；

若a为1, d为11,则a为d的前缀。这时111可以理解为 ad, da, aaa

无二义性编码对应



- 字符为叶子
- 其他结点不包含字符

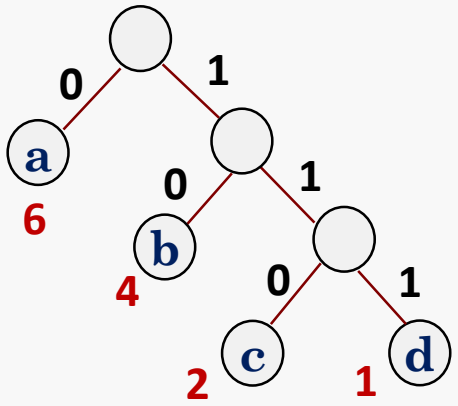
哈夫曼树与哈夫曼编码

问题

假定有一段报文由a、b、c、d四个字符构成，它们的使用频率比为6：4：2：1，
请构造一套二进制编码系统，使得报文翻译成二进制编码后**无二义性**且**长度最短**

最优编码方案

对应



哈夫曼树

哈夫曼编码
a-0　　b-10
c-110　d-111

长度最短 要求
下面式子的值最小

加权平均
长度的4倍

$$\sum_{i=1}^{4} w_i * l_i$$

$\sum_{i=1}^{4} w_i * l_i$ **最小**

w_i：第 i 个字符权重
l_i：第 i 个字符编码长度

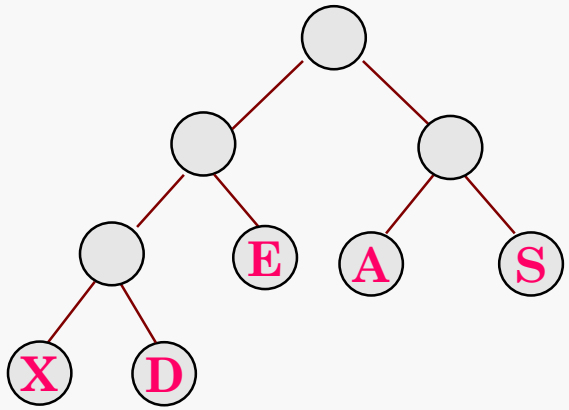
w_i：第 i 个叶子权重
l_i：第 i 个叶子到根的路径长度

哈夫曼树编程分析

- **以下面的问题为例进行分析**

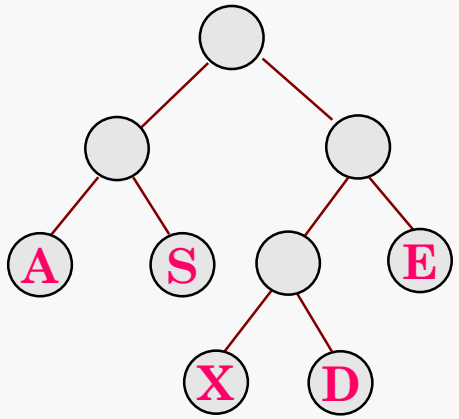
  设一段文本由字符 X, S, D, E, A构成，它们的使用权重为2：9：5：7：8， 请以这些字符构造哈夫曼树，并求出它们的哈夫曼编码

- **注意：**哈夫曼树及哈夫曼编码不是唯一的



X 000
D 001
E 01
A 10
S 11

WPL=69



X 100
D 101
E 11
A 00
S 01

WPL=69

哈夫曼树结点定义

◆结点要存储字符、权重

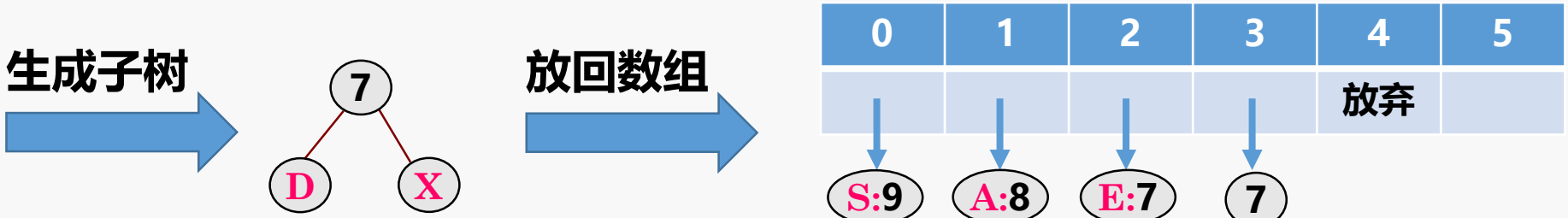
```
struct HNode
{
    char        data;                   //字符数据
    int         weight;                 //权重
    struct      HNode *lchild;          //左孩子指针
    struct      HNode *rchild;          //右孩子指针
};
```

初始 n 棵单根树构造

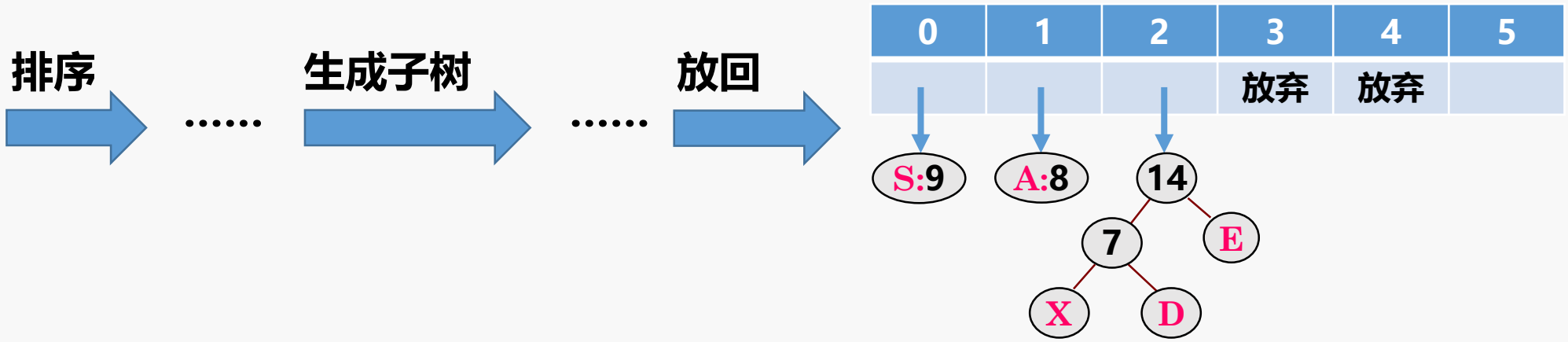
```
//假定结点数不超30（只用前 5 棵树）
HNode *h[30],  *root;
//让h[i]指向第i棵单根树
for (int i = 0; i<30; i++)   h[i] = new HNode;
//为每个单根树赋权值、字符
char ch[] = { 'X','S','D','E','A' };
int    weight[] = { 2,9,5,7,8 };
for (int i = 0; i<5; i++) {
      h[i]->data = ch[i];
      h[i]->weight = weight[i];
      h[i]->lchild = NULL;
      h[i]->rchild = NULL;
}
```

哈夫曼树合并生成过程

第一周期



第二周期



按权值排序函数Sort

将数组 h[] 前 n 项按权值排序 （冒泡）

```
void Sort(HNode* h[], int n)
{
    for(int i=1; i<n; i++)
        for (int j = 0; j < n - i; j++)
        {
            if (h[j]->weight < h[j + 1]->weight)
            {
                HNode *t = h[j];
                h[j] = h[j + 1];
                h[j + 1] = t;
            }
        }
}
```

合并生成哈夫曼树

- **关键代码分析**

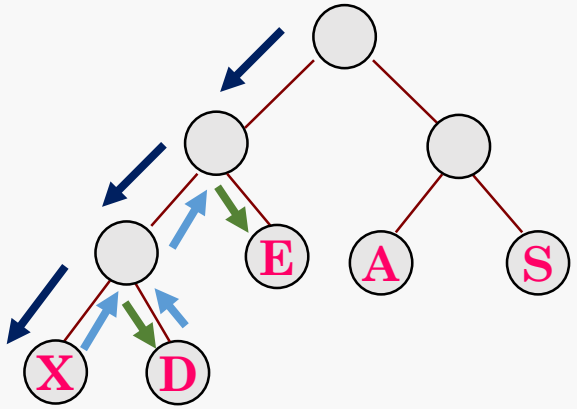
```
……
while(n>1)          // 合并 n-1 次
{
    Sort(h, n);    //将数组 h[] 前 n 项按权值排序
    HNode* s = new HNode;              //生成子树的根
    s->data = ' ';
    s->lchild = h[n-1];                //插入左子树
    s->rchild = h[n-2];                          //插入右子树
    s->weight = h[n-1]->weight + h[n-2]->weight;          //子树权值
    h[n - 2] = s;                  //放回数组 h[] 中
    n = n - 1;
}
```

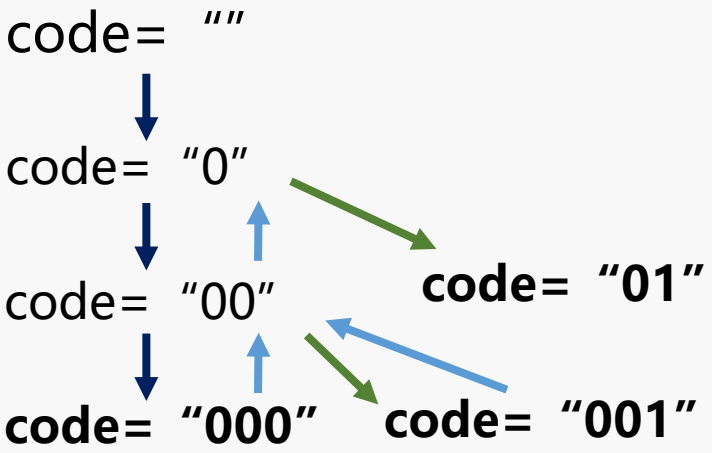
//哈夫曼树的根指针就是 h[0]

生成哈夫曼编码

- **利用二叉树先序遍历构造编码**

X 000
D 001
E 01
A 10
S 11





每次进入左孩子 code 尾部添加 0
每次从左孩子退回上一级 code 尾部截掉1位数

每次进入右孩子 code 尾部添加 1
每次从右孩子退回上一级 code 尾部截掉1位数

生成哈夫曼编码

- **关键代码分析**

```
char code[100] = "";
void PreOrder( HNode *t )                    // 对二叉树t进行先序遍历
{
        if (t) {
                PreOrder(t->lchild);                    //先序遍历左子树
                PreOrder(t->rchild);                    //先序遍历右子树
        }
}
```

生成哈夫曼编码

- **关键代码分析**

```
char code[100] = "";
void PreOrder( HNode *t )                    // 对二叉树t进行先序遍历
{
        if (t) {
                进入左孩子  code 尾部添加 0
                PreOrder(t->lchild);                    //先序遍历左子树
                退回上一级  code 尾部截掉1位数
                进入右孩子  code 尾部添加 1
                PreOrder(t->rchild);                    //先序遍历右子树
                退回上一级  code 尾部截掉1位数

                如果t指向叶子，则
                    输出 字符 和 code
        }
}
```

生成哈夫曼编码

- **关键代码分析**

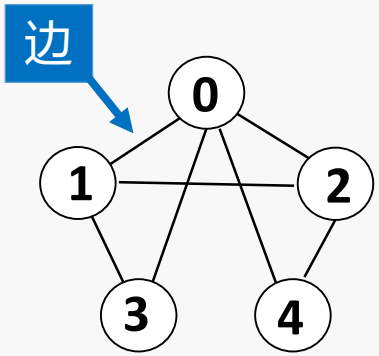
```
char code[100] = "";
void PreOrder( HNode *t )                    // 对二叉树t进行先序遍历
{
        if (t) {
                进入左孩子  code 尾部添加 0
                PreOrder(t->lchild);                    //先序遍历左子树
                退回上一级  code 尾部截掉1位数

                进入右孩子  code 尾部添加 1
                PreOrder(t->rchild);                    //先序遍历右子树
                退回上一级  code 尾部截掉1位数

                如果t指向叶子，则
                        输出 字符 和 code
        }
}
```

问题求解与实践
——图结构

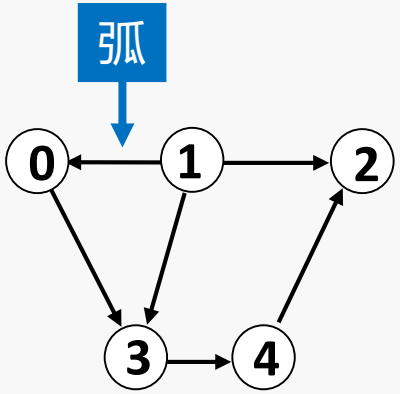
主讲教师： 陈雨亭、沈艳艳

图的基本概念

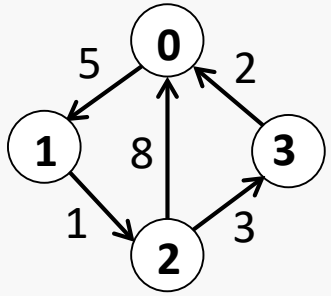
- 图结构来源于生活中诸如通信网、交通网之类的事物，它表现了数据对象间多对多的联系
- 在该结构中，数据元素一般称为顶点
- **图**是由顶点集合及顶点间的关系集合组成的一种数据结构。一般记作（V, E）。其中V是顶点的有限集合；E是顶点之间关系的有限集合



无向图



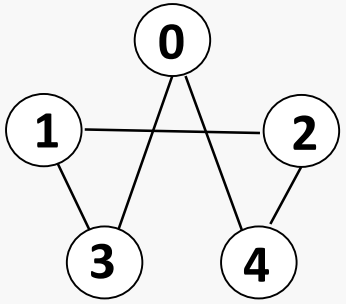
有向图



有权图(网络)

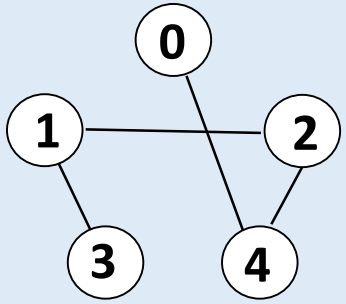
图的基本概念

- 路径：若从顶点v_i出发，沿一些边或弧，经过顶点 v_{p1}，v_{p2}，…，v_{pm} 到达顶点 v_j。则顶点序列(v_i，v_{p1}，…，v_{pm}，v_j)为从顶点 v_i 到顶点 v_j 的路径
- 路径长度：非带权图的路径长度是指此路径上边或弧的条数，带权图的路径长度是指路径上各边或弧的权之和
- 连通图：在无向图中，若从顶点 v_i 到 v_j 有路径，则顶点 v_i 与 v_j 是连通的。如果图中任意一对顶点都是连通的，则称此图是连通图
- 生成树：在无向图中，一个连通图的生成树是它的极小连通子图

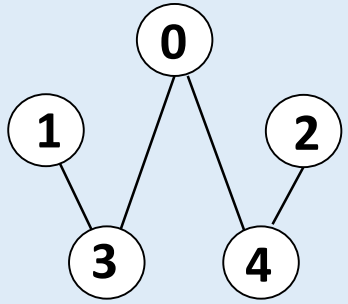


连通图

(1，3，0，4) 或
(1，2，4)
都是 1 到 4 的路径



生成树1



生成树2

图的存储方式

- 图的存储形式有多种，无论哪种形式都要存储两方面的信息：
  1. 顶点信息
  2. 顶点间的关系信息

- 图的存储形式最常见的有：
  - 邻接矩阵
  - 邻接表

图的存储方式——邻接矩阵

◆ 要点：

1. 利用一维数组存储顶点信息
2. 利用二维数组存储顶点间边或弧的信息。此二维数组称**邻接矩阵**

➢ 对于无向图G=(V,E)，邻接矩阵 A 的元素：

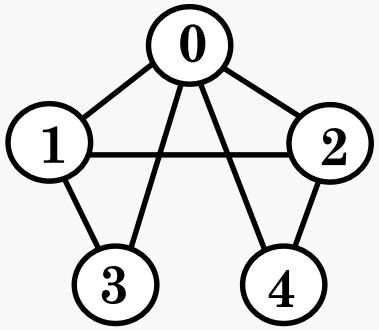
$$A[i][j]=\begin{cases}1 & \text{当}v_i\text{到}v_j\text{有边或弧直连}\\ 0 & \text{其它}\end{cases}$$

➢ 对于带权的图，邻接矩阵 A 的元素：

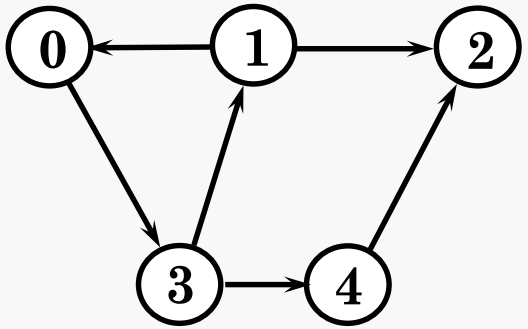
$$A[i][j]=\begin{cases}W(i,j) & \text{当}v_i\text{到}v_j\text{有边或弧直连}\\ \infty & \text{其它}\end{cases}$$

其中 $W(i,j)$ 是与边或弧相关的权

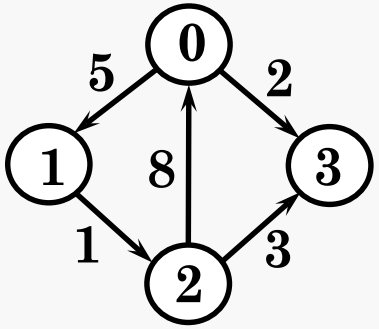
图的存储方式——邻接矩阵示例



(a)无向图



(b)有向图



(c)网络

$$A=\begin{array}{c}\begin{matrix}0&1&2&3&4\end{matrix}\\ \begin{bmatrix}0&1&1&1&1\\1&0&1&1&0\\1&1&0&0&1\\1&1&0&0&0\\1&0&1&0&0\end{bmatrix}\end{array}\begin{matrix}0\\1\\2\\3\\4\end{matrix}$$

(a)无向图邻接矩阵

$$A=\begin{array}{c}\begin{matrix}0&1&2&3&4\end{matrix}\\ \begin{bmatrix}0&0&0&1&0\\1&0&1&0&0\\0&0&0&0&0\\0&1&0&0&1\\0&0&1&0&0\end{bmatrix}\end{array}\begin{matrix}0\\1\\2\\3\\4\end{matrix}$$

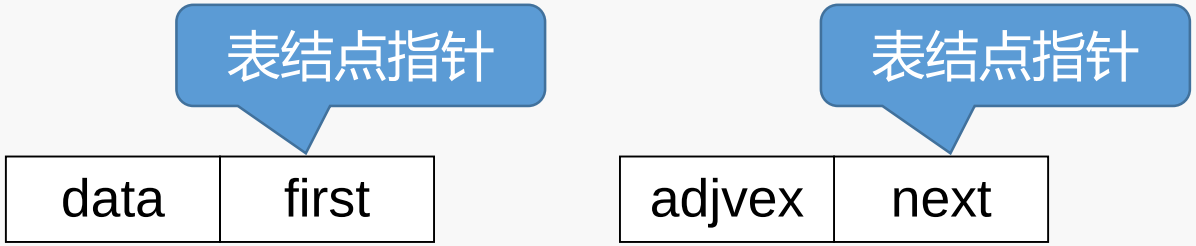
(b)有向图邻接矩阵

$$A=\begin{array}{c}\begin{matrix}0&1&2&3\end{matrix}\\ \begin{bmatrix}\infty&5&\infty&2\\\infty&\infty&1&\infty\\8&\infty&\infty&3\\\infty&\infty&\infty&\infty\end{bmatrix}\end{array}\begin{matrix}0\\1\\2\\3\end{matrix}$$

(c)网络邻接矩阵

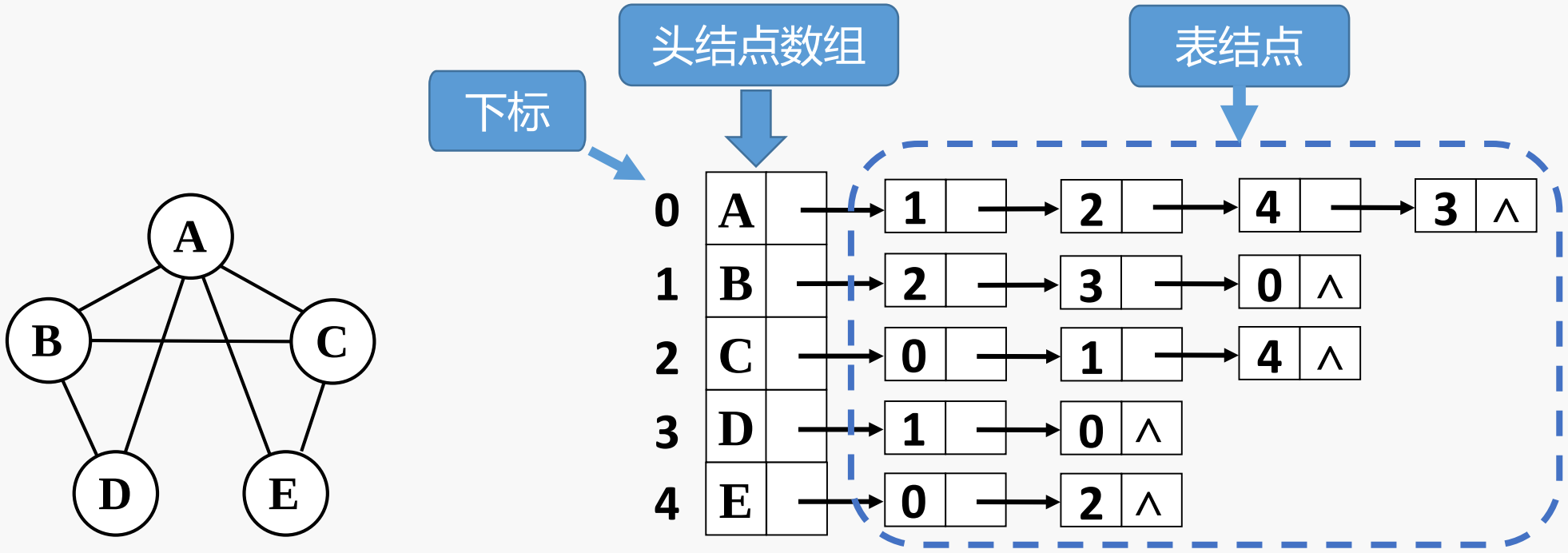
图的存储方式——邻接表

- 邻接表是数组与链表结合的存储形式
  - 邻接表中有两种结点，**头结点**和**表结点**
  - 每个头结点存储一个顶点的信息，一般头结点都存放在一个数组中
  - 对于某个顶点而言，需要将它的邻接点存储为表结点形式，并将它们链接成单链表，这个单链表就称为该顶点的邻接表
  - 每个头结点后面连接其对应的邻接表



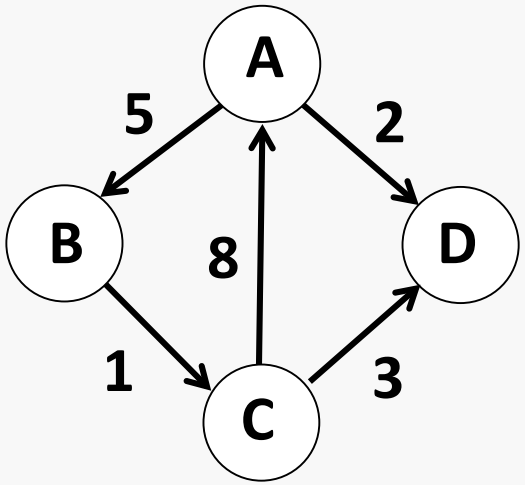
(a)头结点　　(b)无权图的表结点

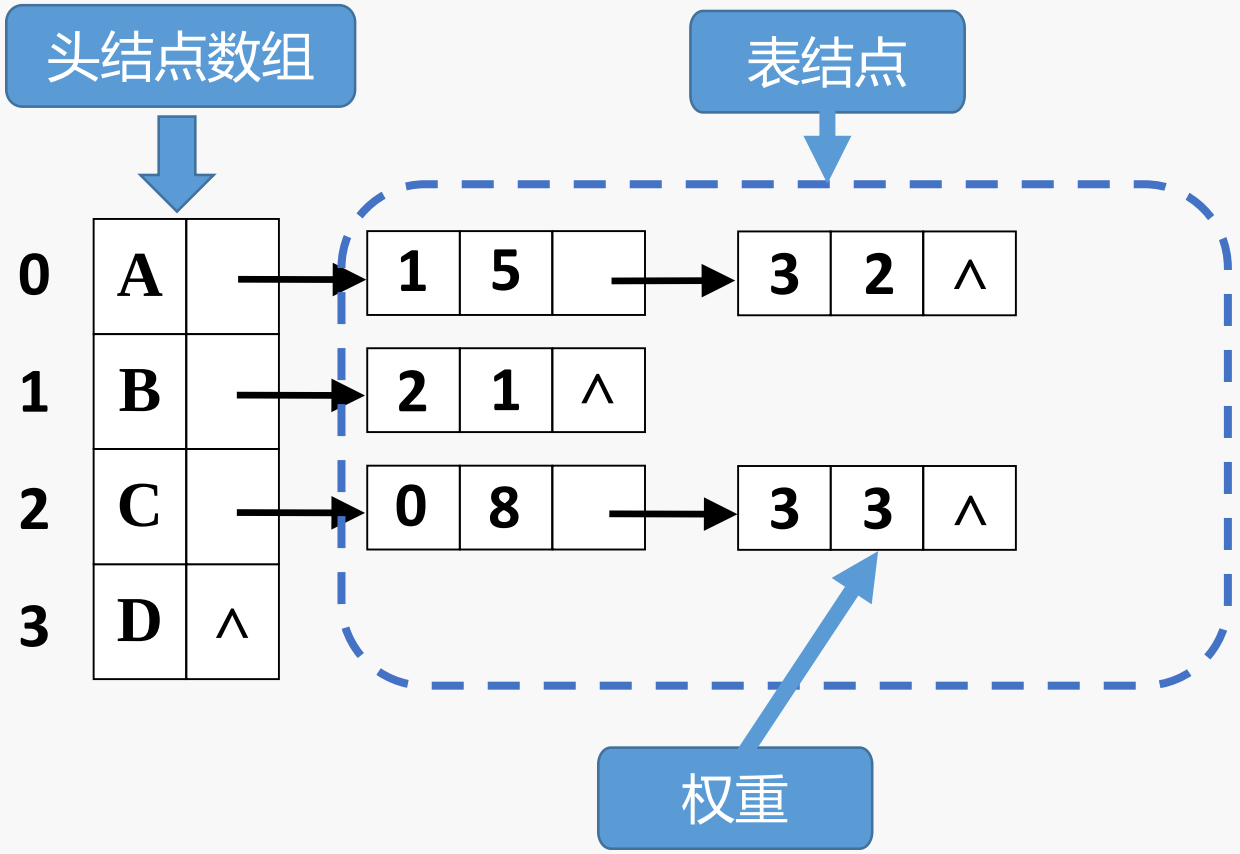
图的存储方式——邻接表示例



无向图的邻接表

图的存储方式——邻接表示例





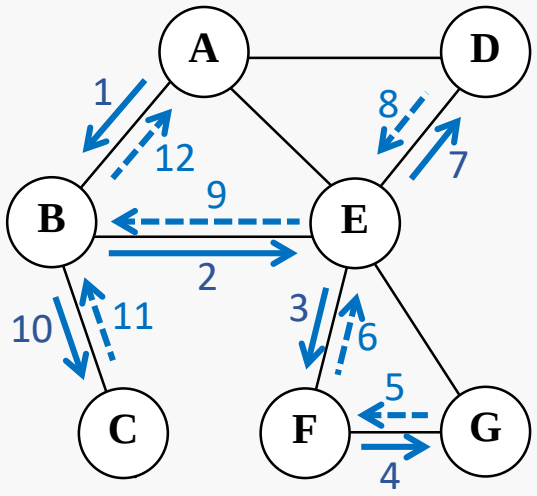
网络的邻接表

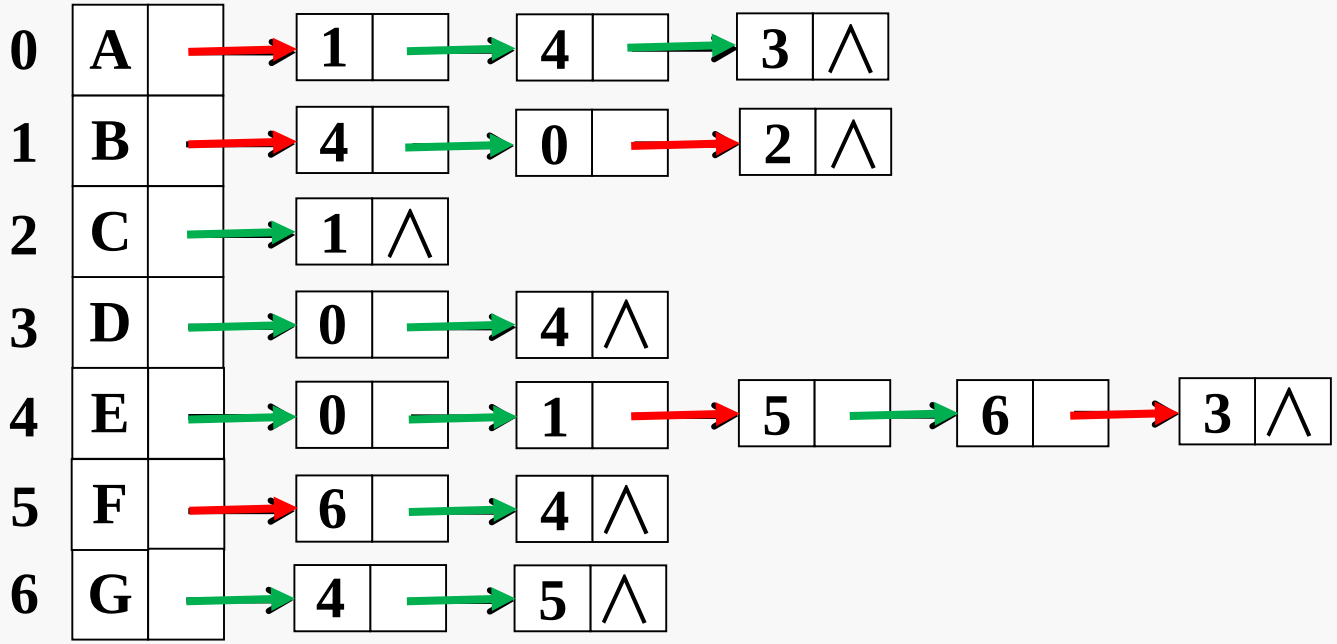
图的遍历

- 图的遍历是指从图的某个顶点出发访问图中所有顶点，并且使图中的每个顶点仅被访问一次的过程
- 图的遍历算法主要有深度优先搜索和广度优先搜索两种

深度优先搜索遍历

◆ **基于邻接表的深度优先搜索过程：**





A B E F G D C

深度优先搜索遍历

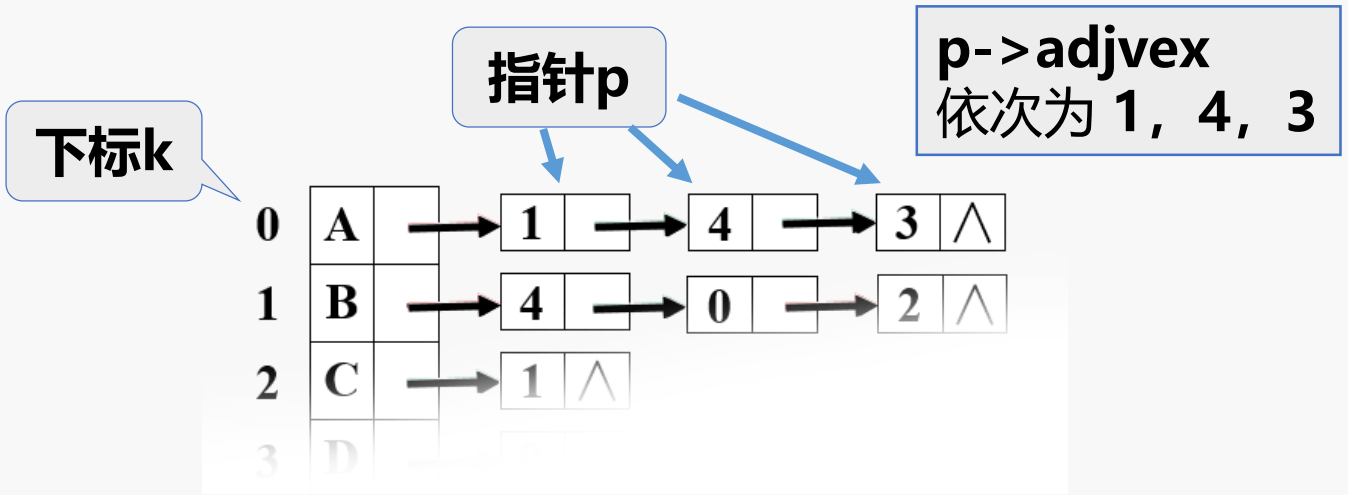
◆ **基于邻接表的深度优先搜索算法实现要点：**

```
DeepSearch(  下标k  )                          //从 k 号顶点开始搜索
{
    visit（k 号下标顶点）；
    沿着 k 号顶点邻接表搜索未被访问的结点p, 若p存在，则
        DeepSearch( p->adjvex  )；   //这里 p 为表结点指针
}
```

是一个循环过程

未被访问的结点p ：

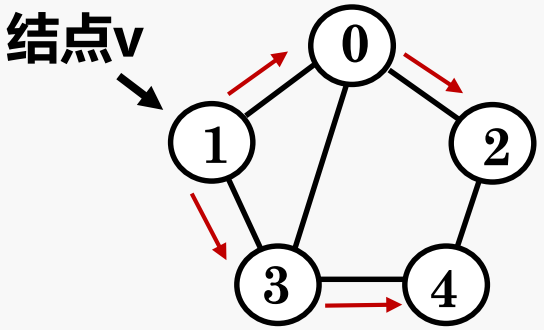
是指该表结点中数字对应下标的顶点未被访问

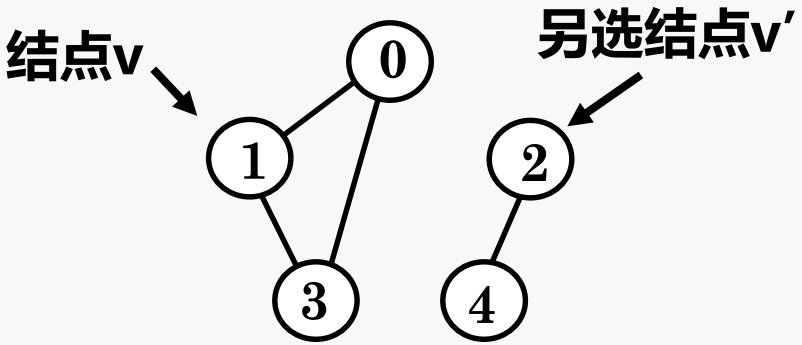


广度优先搜索遍历

◆ 图的广度优先搜索遍历过程：

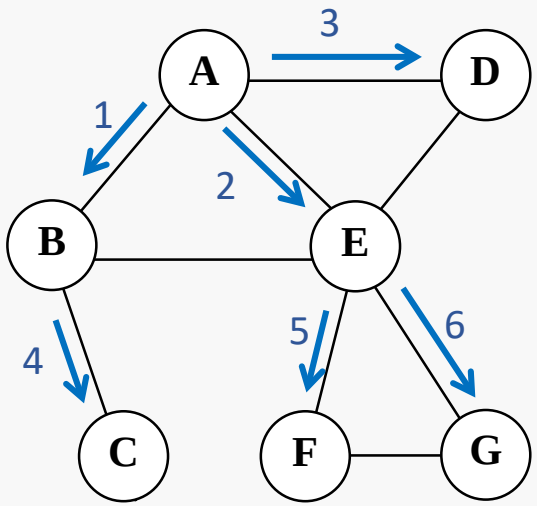
- 假定从图中某个顶点v 出发进行遍历，则首先访问此顶点；
- 再依次访问v的**所有未被访问过的邻接点**，然后按这些顶点被访问的先后次序**再依次访问它们的邻接点**，直至图中所有和v有路径相通的顶点都被访问到；
- 若此时图中尚有顶点未被访问，则另选图中一个未曾被访问的顶点作起始点，重复上述过程，直至图中所有顶点都被访问到为止。

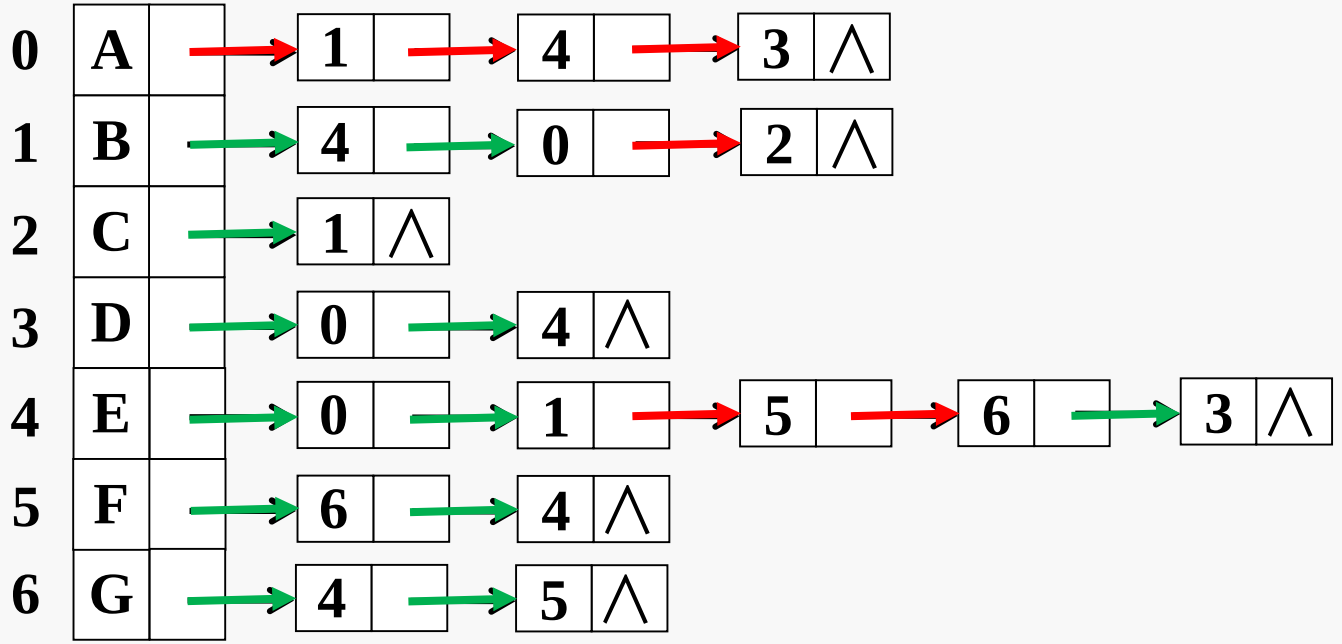




广度优先搜索遍历

◆ **基于邻接表的广度优先搜索过程：**





访问：

广度优先搜索遍历

◆ **广度优先搜索算法实现要点：**

```
定义整数队列 queue;                    // 用于存放表示顶点G[k]的下标k
定义V[]，初始全为0;                    // V[k]表示顶点k是否被访问过
将 0 放入queue;                        // 假设从 0 号结点开始遍历
while( queue 不空 )      {             // 队空则搜索结束
    元素从 queue 中出队放入k;          //出队列
    访问顶点 G[k];     V[k]=1;         //0-未访问，1-已访问
    p = G[k].first;                    // k 号顶点的第一个邻接点的指针
    while( p != NULL )
    {   若V[p->adjvex]=0 则将 p->adjvex 入队列;
        p = p->next;
    }
}
```