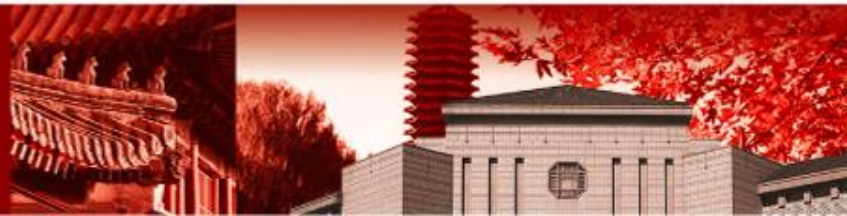
数据结构与算法 B

总复习

主讲教师：孙艳春

Email: sunyc@pku.edu.cn



声明

本文档用于帮助梳理课程所学习的内容，进一步强化所学习的知识点，但

绝非考试范围和考察重点！

教学大纲

- Python基础
- 数据结构与算法概论
  - 数据的逻辑结构和存储结构
  - 算法的复杂度分析
- 线性结构
  - 顺序表：实现与增删操作
  - 链表：单链表、双链表、循环链表
  - 字符串：
    - 字符串的基本操作
    - 模式匹配算法
      - 朴素模式匹配算法
      - KMP匹配算法
  - 栈：表达式求值、合法出栈序列判断
  - 队列：定义，实现及其应用
- 排序算法
  - 基本排序算法：
    - 插入排序（直接插入、二分插入、shell排序）
    - 选择排序（直接选择、堆排序）
    - 交换排序（冒泡排序、快速排序）
    - 归并排序（二路归并）
    - 分配排序（基数排序）
  - 算法的复杂度分析与比较

教学大纲

- 树结构
  - 二叉树：遍历算法、堆结构
  - 二叉排序树（不考AVL）
  - Huffman树：Huffman编码算法
  - 树与森林：表示方法及转换方法
  - 堆的定义和堆排序
- 字典与检索
  - 线性检索：应用索引
  - 基于散列表的检索：散列表的定义、碰撞的处理等
- 图结构
  - 图的存储表示与遍历
  - 最小生成树：Prim算法，Kruskal算法
  - 拓扑排序算法
  - 最短路径：Dijkstra算法、Floyd算法
- 课程总复习

教学内容（一）

逻辑结构	存储结构	数据结构	算法
线性结构（线性表）：至多具有一个前驱与一个后继	顺序存储	顺序表	增删改查操作 排序算法：选择、插入、交换、归并、分配
	链式存储	链表	单链表、双链表、循环链表的增删改查操作
	顺序存储	字符串	模式匹配：KMP算法
	顺序存储	栈	增删改查操作 栈的应用：递归程序、括号匹配、表达式求值
	链式存储		
	顺序存储	队列	队列、双端队列的增删改查操作
	链式存储		
集合结构（字典）：无前驱与后继关系	**顺序结构**	**字典**	**顺序检索、二分检索**
	散列结构		散列检索
	索引结构		二叉搜索树

教学内容（二）

逻辑结构	存储结构	数据结构	算法
树形结构：具有至多一个前驱与多个后继	顺序结构	二叉树	存储表示：顺序与链式存储； 二叉树的深度和广度优先遍历； 应用：堆的建立与增删操作、Huffman算法、二叉搜索树的增删操作
	链式结构		
	顺序结构	树和森林	存储表示：父指针表示法、子表表示法、长子兄弟表示法； 深度和广度优先遍历； 树和森林与二叉树的转换； 并查集算法
	链式结构		
图结构：具有多个前驱与多个后继	顺序存储	图	存储表示：邻接表与邻接矩阵； 图的搜索与遍历； 最小生成树算法； 最短路径算法； 拓扑排序与关键路径算法
	链式存储		

数据的存储结构

- **顺序存储结构**：把逻辑上相邻的结点存储在物理位置相邻的存储单元里，结点间的逻辑关系由存储单元的邻接关系来体现。
- **链式存储结构**：不要求逻辑上相邻的结点在物理位置上亦相邻，结点间的逻辑关系是由附加的指针字段表示的。
- **索引存储结构**：除建立存储结点信息外，还建立附加的索引表来标识结点的地址。
- **散列存储结构**：就是根据结点的关键字直接计算出该结点的存储地址。

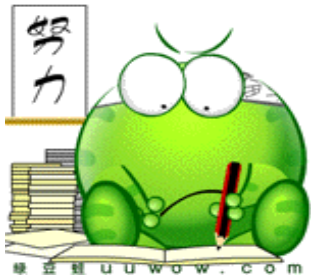
四种存储结构既可单独使用，又可组合使用

课程学习后应达到的能力

① 从逻辑结构、存储结构和相应的一组基本运算三个方面去掌握常用的数据结构。

② 对算法的时间和空间复杂性有一定的分析能力。

③ 针对简单的应用问题，应能选择合适的数据结构及设计有效的算法解决之。



第一讲 概论

算法、数据结构及数据结构三要素

- 算法+数据结构=程序
- **数据结构**是计算机科学中用于组织、存储和管理数据的方式，它定义了数据元素之间的关系、操作数据的规则以及数据在计算机内存中的存储方式。
- **数据结构的三要素**：逻辑结构、存储结构、运算

- 逻辑结构：
  线性结构 ⊆ 树形结构 ⊆ 图结构

- 存储结构：
  顺序方法、链接方法
  散列方法、索引方法

- 运算：
  增、删、改、查等基本运算
  遍历、排序等复杂运算

什么是算法分析

- 本课程中，我们主要感兴趣的是算法本身的特性
  - 有穷性：一个算法必须可以用有穷条指令描述，且必须在执行有穷次操作后终止。
  - 确定性：对于相同的输入，无论运行多少次，都得到相同的输出。
  - 可行性：算法中的指令含义明确无歧义，且可以被机械地自动执行。
  - 输入输出：可以无输入，但必须有输出
  - 效率：算法应在合理的时间内完成任务，通常通过**时间复杂度和空间复杂度**来衡量。

- 算法分析主要就是从计算资源消耗的角度来评判和比较算法
  - 更高效利用计算资源，或者更少占用资源的算法，就是好算法
  - 从这个角度，前述两段程序实际上是基本相同的，它们都采用了一样的算法来解决累计求和问题

算法复杂性度量

- 重要的不是算法运行时花费的具体时间，而是算法复杂性与输入数据规模(n)之间的关系
  - 例如，对于排序算法，输入数据规模一般就是待排序元素的个数
- 用输入规模与基本操作量级的函数关系来描述时间复杂性
  - 算法所实施的基本操作数量可作为独立于程序/机器的度量指标
  - 哪种基本操作跟算法的具体实现无关？
    - **执行次数最多的某种时间固定的基本操作的执行次数和n的关系**
- 赋值语句是一个合适的选择
- 分析SumOfN的基本操作数量
  - 对于“问题规模”n
  - 赋值语句数量T(n)=1+n

```
31    def sumOfN(n):
32  1     theSum = 0
33  n     for i in range(1, n + 1):
34            theSum = theSum + i
35  1     return theSum
```

渐进性分析方法--大O表示法

- 算法分析：估计当数据规模n逐步增大时，时间/空间资源开销f(n)的增长趋势
- 数量级函数（Order of Magnitude function）
  - 基本操作数量函数T(n)的精确值并不是特别重要，重要的是T(n)中起决定性因素的主导部分
  - **数量级函数**描述了T(n)中随着n增加而增加速度最快的部分
  - 称作**“大O”表示法，记作O(f(n))，其中f(n)表示T(n)中的主导部分**

- **算法的空间复杂度**是对算法在运行过程中占用额外存储空间大小的量度
- 空间复杂度算的是变量的个数，计算规则基本和时间复杂度类似，也使用大O表示法

简单练习（一）

一、选择题：

1、在数据结构中，从逻辑上可以把数据结构分成（ ）。
A．动态结构和静态结构 B．紧凑结构和非紧凑结构
C．线性结构和非线性结构 D．内部结构和外部结构

2、与数据元素本身的形式、内容、相对位置、个数无关的是数据的（ ）。
A．存储结构 B．存储实现
C．逻辑结构 D．运算实现

3、以下说法正确的是（ ）。
A．数据元素是数据的最小单位
B．数据项是数据的基本单位
C．数据结构是带有结构的各数据项的集合
D．一些表面上很不相同的数据可以有相同的逻辑结构

简单练习（一）

二、填空题：

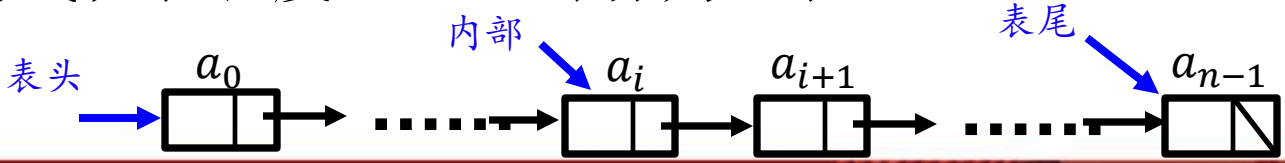
数据结构是研究数据的__逻辑结构__和__存储结构__，以及这个结构具有的行为特征，定义相应的运算，设计出相应的__算法__。

第二讲 线性表

1、线性表（线性结构）

线性表

- 线性表的逻辑结构：线性结构
- 线性结构：$\{a_0, a_1, \ldots, a_{n-1}\}$
  - **有序**的有限元素序列
  - 每个元素存在**唯一的前驱与后继**（除第一个与最后一个外）
- 线性结构的特点
  - **均匀性**：虽然不同线性表的数据元素可以是各种各样的，但对于同一线性表的各数据元素必定具有相同的数据类型和长度
  - **有序性**：各数据元素在线性表中都有自己的位置，且数据元素之间的相对位置关系是线性的
- 用下标表示元素的位置，由0开始计数
- n为线性表长度，n = 0时称为空表



线性表的运算

- 线性表的各种实现通常都需要提供一些“标准”操作
  - 创建和销毁线性表
  - 判断一个线性表是否为空，如果线性表的容量有限制，还需判断它是否满
  - 向线性表中加入元素或从中删除元素（增删）
  - 读取和写入线性表里的元素（改查）

顺序表

- 顺序表也称向量，是元素在内存中连续存放的线性表。
- 特点

  - 元素顺序地存储在连续存储空间中
  - 每一个元素有唯一的序号（下标）
  - 顺序表最大长度为定值
  - 读写其元素很方便，通过下标即可指定位置，其时间复杂度是O（1）
    - 只要确定了首地址，线性表中任意数据元素都可以随机存取
- 其他编程语言，如C/C++、Jave中的数组，是顺序表
- Python中的列表是顺序表

顺序表支持的操作

序号	操作	含义	时间复杂度
1	init(n)	生成一个n个元素的顺序表	O(1)
2	init(a_0,a_1,....a_n)	生成元素为a_0,a_1,.... a_n的顺序表	O(n)
3	len()	求表中元素个数	O(1)
4	append(x)	在表的尾部添加一个元素x	O(1)
5	pop()	删除表尾元素	O(1)
6	get(i)	返回下标为i的元素	O(1)
7	set(i,x)	将下标为i的元素设置为x	O(1)
8	index(x)	查找元素x在表中的位置	O(n)
9	insert(i,x)	在下标i处插入元素x	O(n)
10	delete(i)	删除下标为i的元素	O(n)

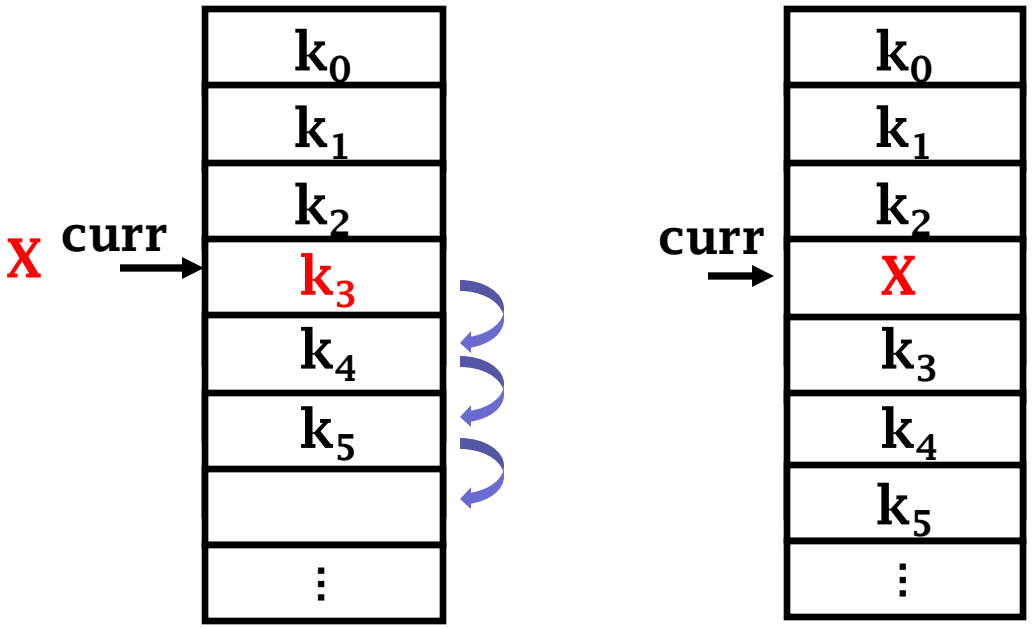
注：**黑色的操作**指，python的列表作为一种顺序表的实现，已经实现的顺序表操作。
红色的操作指需要自己实现的操作。

顺序表中增加/删除元素（表尾操作）

- 尾端加入和删除操作的实现很简单
- 尾端加入元素（O(1) 操作）：append (x)
  - 检查表是否满，表满时操作失败
  - 把新数据存入元素存储区的第 length 个单元（length表示当前长度）
  - 将元素计数变量 length 加一
- 尾端删除元素（O(1) 操作）：pop( )
  - 简单地把元素计数变量 length 减一
- 定位的加入和删除都比较麻烦
  - 因为要保证元素在存储区前段连续存储
  - 可能需要维持原有元素的顺序

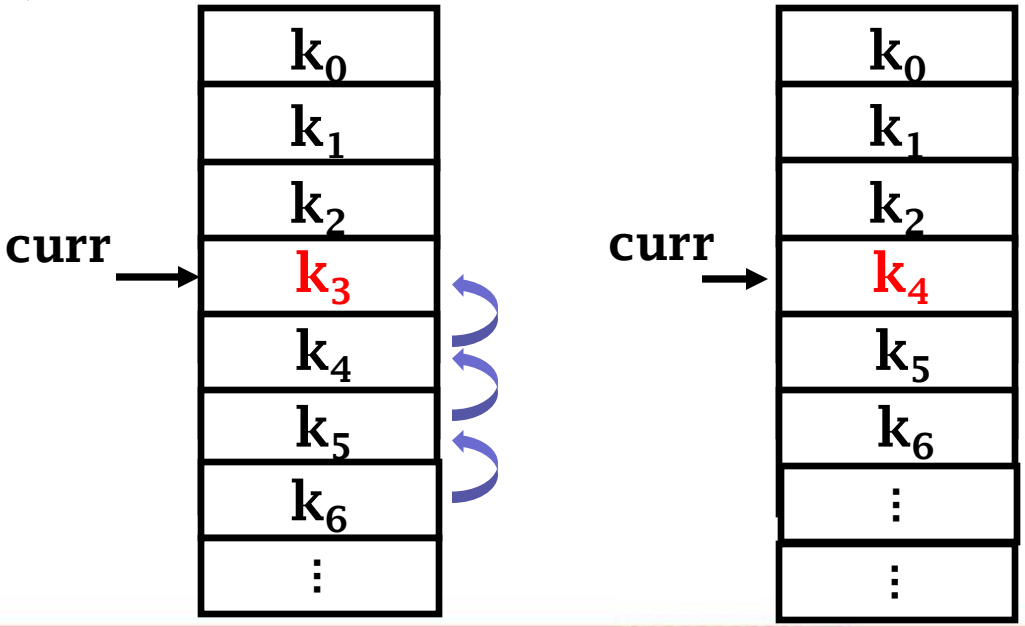
顺序表中增加元素（随机操作）

- 在任意位置加入元素（随机操作）时，需要移动已有元素，腾出要求存入元素的位置：insert(i, x)
  - 逆序地逐个后移后面元素，直至腾出指定位置后将元素放入



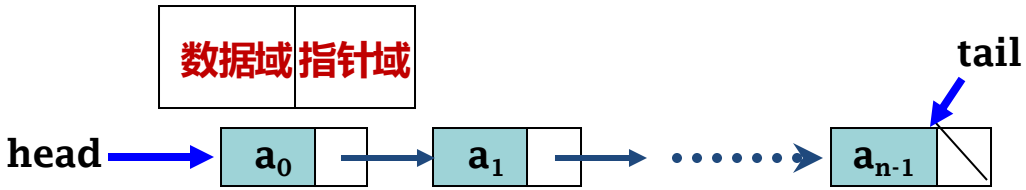
顺序表中删除元素（随机操作）

- 在任意位置删除元素（随机操作）时，同样需要移动已有元素，保证所有元素连续存储: delete(i)，相当于python中list的pop(i)操作
  - 顺序地逐个前移后面元素，直到所有元素连续存储
  - 待删除元素被自然“覆盖”了



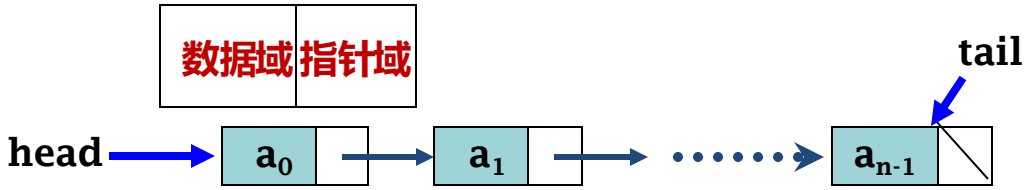
2、链表

- 采用链式存储结构的线性表
- 通过后继引用把一串存储结点链接成一个链
  - 每个结点存储指向后继结点的引用来表示数据元素之间的逻辑关系
  - 逻辑上相邻的元素在存储位置上不要求也相邻
  - 按照需要为表中新的元素动态地分配存储空间，动态改变长度
- 存储结点由两部分组成：
  - 数据项本身 + 指向后继结点的指针（引用）



单链表

- 要访问一个单链表，仅需要掌握表的首结点。从它：
  - 可以找到表的首元素
  - 还可以找到表中下一结点的位置
- 按同样方式继续下去，就可以找到表里的所有数据元素
  - 表头变量：保存着链表第一个结点的引用的变量
  - 对于空表，表头变量为None，表示第一个结点不存在
  - 根据需要还可以设置表尾变量，保存最后一个结点的引用



单链表的创建/删除

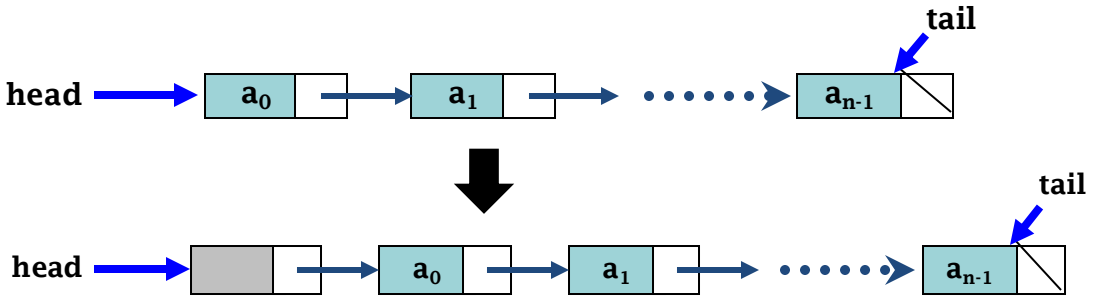
- 创建空链表：只需将表头变量设置为空
  - 在 Python 里将其设置为None
- 删除链表：丢弃表的所有结点，与具体环境有关
  - 在一些语言（如C 语言）里需要做许多事情，手动释放所有存储
  - 在 Python 里，只需简单将表头设为None，就丢掉了整个链表的所有结点。Python 的垃圾回收机制会自动回收不用的空间。

```
class LinkNode():
    def __init__(self, node_data):
        self.data = node_data
        self.next: LinkNode = None
```

```
class LinkList():
    def __init__(self):
        self.head: LinkNode = None
        self.length = 0
```

向单链表中插入元素

- 为了将指定元素插入到下标为 i 的位置上：
  - 定位到待插入位置的前驱结点
  - 依次更改待插入结点、前驱结点的引用
- 当然，存在一些稍微复杂的边界情况：
  - 在尾端插入时，不存在后继，将待插入结点的链接域设为None
  - 在首端插入时，不存在前驱，但需要另外设置表头变量
    - 引入一个不存放元素的空的“头结点”作为前驱，以方便应对这一情况：



从单链表中删除元素

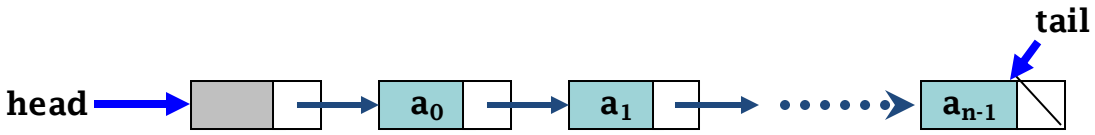
- 与插入操作类似，为了删除插入下标为 i 的位置的元素：
  - 定位到待插入位置的前驱结点
  - 直接修改前驱结点的指针（引用），使之指向下一个结点
  - 被删除的结点将被自动回收
- 练习：
  - 分析删除时的边界情况
  - 给出带头结点的单链表的删除操作实现

单链表操作的时间复杂性分析

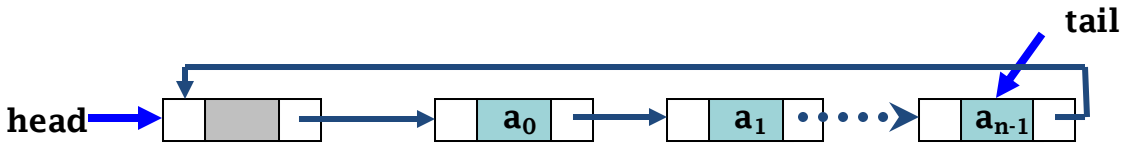
- 在任意位置增加与删除元素
  - 主要代价为定位过程，平均与最坏情况都是O(n)
  - 如果指定在头部操作，则可以以O(1)代价完成
- 按下标读取或写入元素值
  - 与顺序表不同，链式存储结构下必须顺序遍历索引
  - 平均与最坏情况都是O(n)
- 将两个单链表合并
  - 容易想到：为单链表设置表尾变量，就可以快速得到表尾
  - 这种实现下合并代价为O(1)

多种链式存储实现

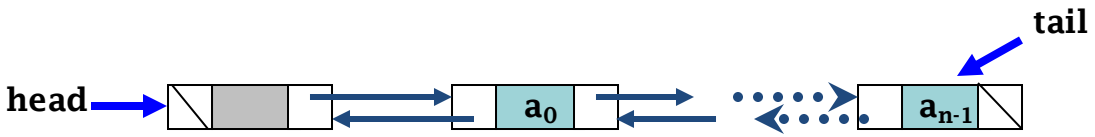
- 单链表



- 循环链表

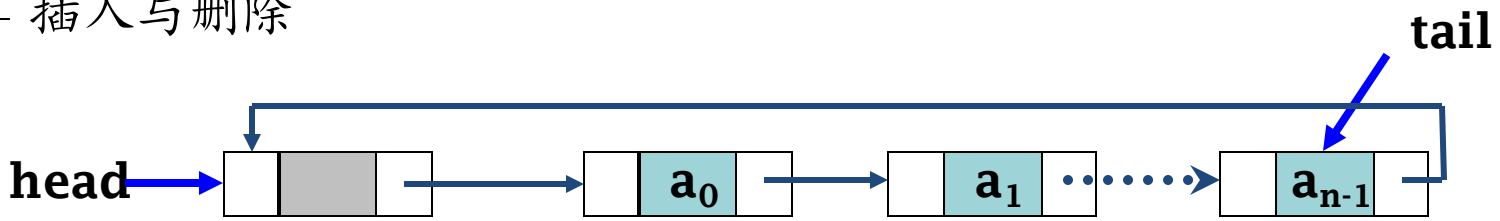


- 双链表



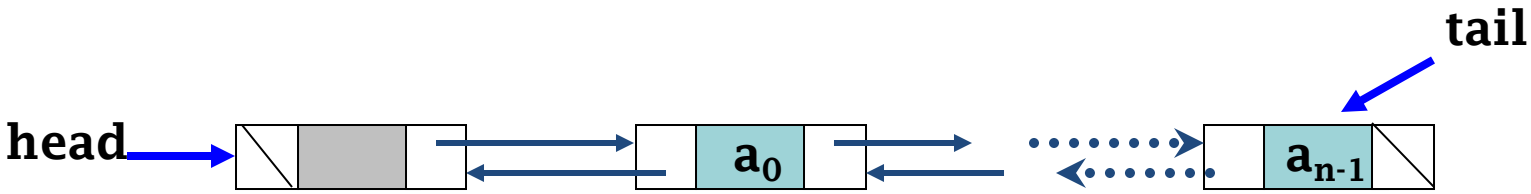
单循环链表

- 将最后一个结点的指针（引用）指向第一个结点（或头结点），构成一个循环链。
  - 从任一个结点出发，可访问表中任何一个结点元素。
  - 能够表示循环的数据关系，逻辑上是对线性结构的扩展
  - 插入与删除



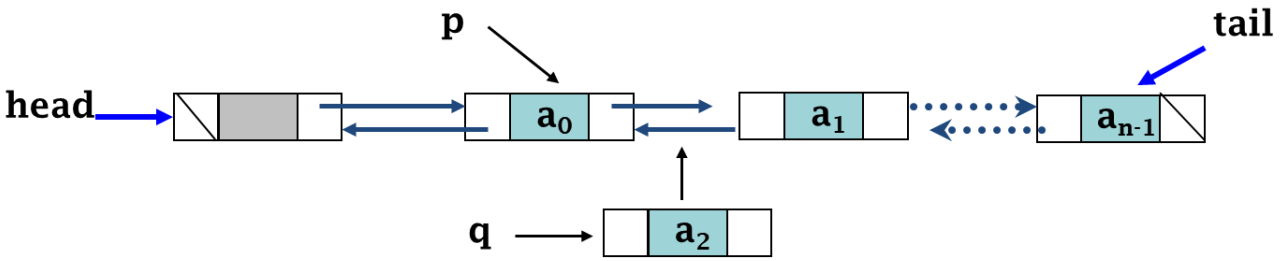
双链表

- 单链表的 next 字段仅仅指向后继结点，而无法访问前驱。
  - 为弥补单链表的不足，增加一个指向前驱的指针
  - 为链表上的遍历提供了更高的灵活性
- 思考：
  - 双链表上的插入、删除操作应该如何进行？



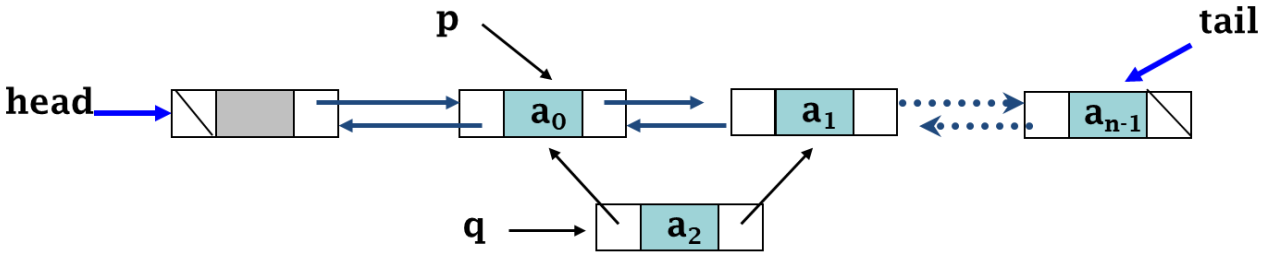
双链表的插入操作

- 插入元素q：



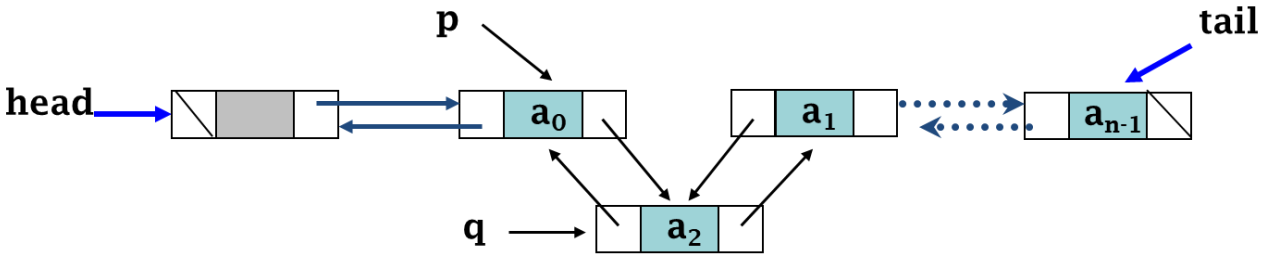
(1) q.next=p.next

(2) q.prev=p



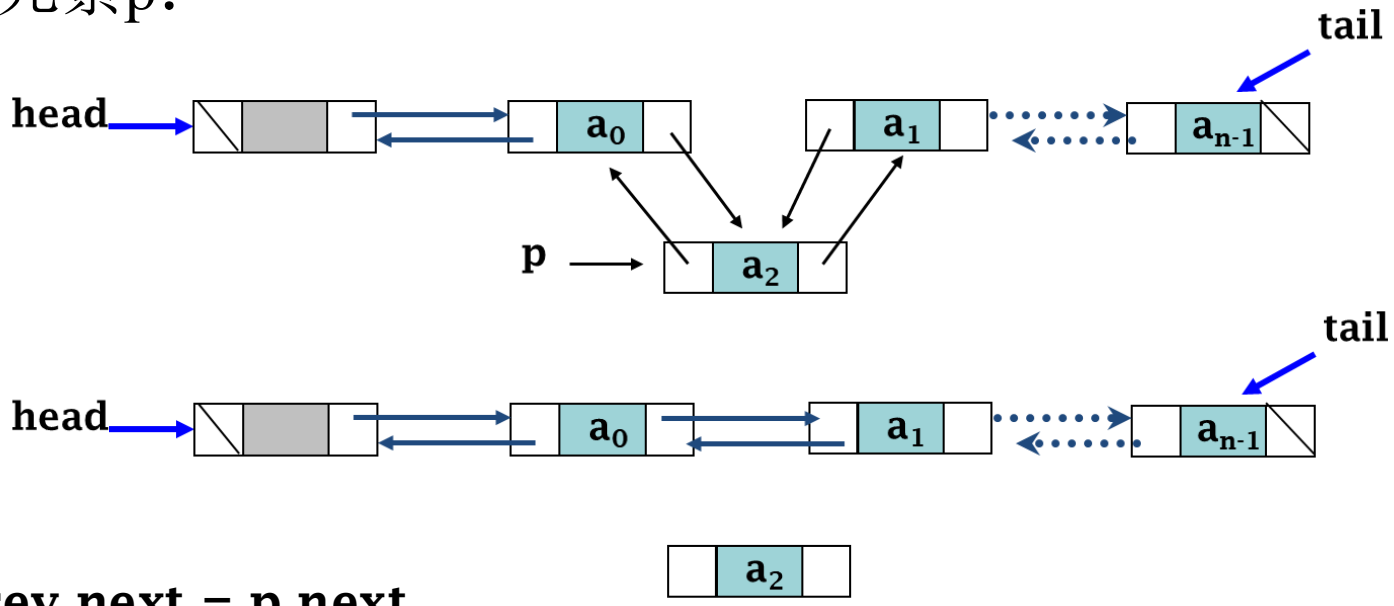
(3) p.next=q

(4) q.next.prev=q



双链表的删除操作

- 删除元素p：



(1) p.prev.next = p.next

(2) p.next.prev = p.prev

(3) p.prev = None

(4) p.next = None

顺序表和链表的比较

- 顺序表
  - 插入、删除运算时间代价 O(n)
  - 按下标的访问则O(1)完成
  - 预先申请固定长度的连续空间
  - 如果整个顺序表元素很满，则没有**结构性存储开销**

指在存储数据本身之外的额外存储开销

- 链表
  - 在给定位置上的插入、删除运算时间代价为O(1)
  - 但找第 i 个元素运算时间代价 O(n)
  - 动态地按照需要为表中新的元素分配存储空间
  - 每个元素都有结构性存储开销

顺序表和链表的比较

- 顺序表的主要优点
  - 无需使用变量引用（指针）来维护线性结构，不用花费额外开销
  - 元素的读访问非常简洁便利
- 链表的主要优点
  - 线性表的长度不受限，允许线性表的长度动态变化
  - 能够适应经常插入删除内部元素的情况
- 总结
  - 顺序表是存储静态数据的不二选择
  - 链表是存储动态变化数据的良方

应用场合的选择

- 顺序表不适用的场合
  - 经常插入删除时，不宜使用顺序表
    - 文档的编辑、任务队列的频繁插入删除，等等
  - 线性表的最大长度也是一个重要因素
- 链表不适用的场合
  - 当读操作比插入删除操作频率大时，不应选择链表
    - 数据库记录的查询，图像的像素存储，等等
  - 当指针的存储开销，和整个结点内容所占空间相比其比例较大时，应该慎重选择

•简单练习（二）

判断题

1. 在线性表的链式存储结构中，逻辑上相邻的元素在物理位置上不一定相邻;
2. 在线性表的顺序存储结构中，插入和删除时，移动元素的个数与该元素的位置有关;
3. 线性表的链式存储结构的特点是用一组任意的存储单元存储线性表的数据元素;
4. 在顺序表中取出第i个元素所花费的时间与i成正比。

Y Y Y N

选择题

- 链表不具有的特点是（　　）。

  A. 可随机访问任意元素

  B. 插入和删除不需要移动元素

  C. 不必事先估计存储空间

  D. 所需空间与线性表长度成正比

A

选择题

- 在双向链表中，删除p所指的结点时需(　　)。

A. (p—>llink)->rlink = p —> rlink;
(p—>rlink)->llink = p —>llink ;

B. p—> link=(p —> llink) —> link;
((p —> llink) —> llink) —> rlink = p ;

C. ((p —> llink) —> llink) —> rlink = p;
p —> llink=(p —> llink) —> llink ;

A

选择题

- 设环形队列中数组长度为 n，头尾指针分别为f和r，则队列中现有元素个数为：（ ）

  A．r-f

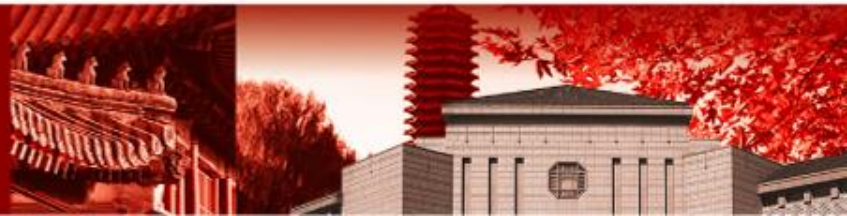
  B．r-f+1

  C．(r-f+1) mod n

  D．(r-f+n) mod n

D

第三讲 字符串



术语表

- **串的长度**：字符串中字符个数
- **空串**：长度为零的字符串，记为s = “”
  - 注意与空格串 “ ”的区别
- **子序列**：字符串s1中，任意个字符按顺序组成的序列s2称为s1的子序列
  - s1=“Hello world”, s2=“Helo word”
- **子串与主串**：字符串s1中任意个连续字符组成的序列s2称为s1的子串，称s1为s2的主串。
  - s1=“Hello world”, s2=“Hello”
  - 空串是任意串的子串；除s本身之外，s的其他子串称为s的真子串。

术语表

- **字符在串中的位置**：该字符在串中第一次出现时的位置。
- **子串在主串中的位置**：该子串在串中第一次出现时，第一个字符的位置。
- **两个字符串相等的充分必要条件**：长度相等，且对应位置上字符相同。

字符串是一种特殊的线性结构

- 字符串与一般线性表的区别与联系
  - 串的数据对象约束为字符集，其每个结点包含一个字符；
  - 线性表的基本操作大多以“单个元素”为操作对象，而串的基本操作通常以“串的整体”作为操作对象；
  - 线性表的存储方法同样适用于字符串，在选择存储结构时，应根据不同情况选择合适的存储表示。

模式匹配算法

- 模式匹配的定义
  - 设有两个串 $t = t_0t_1\ldots t_{n-1}$ (**target**) 和 $p = p_0p_1\ldots p_{m-1}$ (**pattern**)，其中 $1<m<=n$
  - 在t中找出和p相同的子串。此时，t称为“目标”，而p称为“模式”

def match(target: string, pattern: string);

  - 匹配成功：t中存在等于p的子串，返回子串在t中的位置
  - 匹配失败：返回一个特定的标志（如-1）。

两种方法

- 模式匹配是一个比较复杂的字符串操作，
  - 用于在大文本（诸如，句子、段落，或书本）中定位（查找）特定的模式
  - 对于大多数的算法而言，匹配的主要考虑在于其速度和效率

- 下面的讨论是基于字符串的顺序存储结构的算法
  - 朴素的模式匹配(Brute Force)方法
  - 无回溯的模式匹配Knuth-Morris-Pratt (KMP)方法

朴素的模式匹配思想

- 用P中的字符依次与T中的字符比较：
  - 如果P[0:m] == T[0:m]，则匹配成功;
  - 否则，将p右移一个字符，用p中字符从头开始与t中字符依次比较。
  - 如此反复执行，直到下面两种情况之一：
    - 到达某步时，P[0:m] == T[i: i+m]，匹配成功；
    - 将P移到无法与T继续比较为止，则匹配失败

算法时间效率分析

target串长度为n，pattern串长度为m

- 匹配失败
  - **最坏情况**：每趟匹配皆在最后一个字符不等，且有n-m+1趟匹配(每趟比较m个字符），共比较m*(n-m+1)次。
  - 通常m<=n，因此最坏时间复杂度O(n*m)
  - **最好情况**：比较n-m+1次 [每趟只比较第一个字符]
- 匹配成功
  - **最好情况**：m次比较
  - **最坏情况**：与匹配失败的最坏情况相同，即比较m*(n-m+1)次。
- 因此，朴素模式匹配算法的时间复杂度为O(m*n)

KMP算法

- 前例中，一旦P[i]与T[j]比较不等，有：
  - P[0:i] == T[j-i: j], P[i] ≠ T[j]
- 依照朴素方式右移P，实际上做的比较：
  - 尝试将P[0:i]的前缀与后缀进行比较！
  - 右移k位时，就是在比较长为 i-k 的前缀与后缀
- 如何利用这一点来减少冗余比较？
  - 寻找P的最大长度公共前后缀（记为k）
  - 右移小于i-k位的尝试，都一定失败
    - 因为它在尝试比较长度大于k的前后缀
  - 因此，可以跳过上述尝试，**右移i-k位，并直接从P[k], T[j]位置开始比较**
  - 达成的效果是，**在T的视角来看，完全没有回溯发生！**

前后缀指是字符串的真前缀，也是字符串的后缀，则称为字符串的**前后缀**，如abab的前后缀是ab

KMP算法：Next数组

- Next数组：Next[i] 表示 P[0: i] 的最长公共前后缀
  - 公共前后缀（或前后缀）需要同时是真前缀和后缀，排除了字符串本身
  - 如何计算?
- 方法：假定已经得到Next[i] = k, 如何计算Next[i+1]?
  - 如果P[k] = P[i]：Next[i+1] = k+1
  - 如果P[k] ≠ P[i]?

P[0]	……	P[k-1]	P[k]	……	P[i-k-1]	P[i-k]	……	P[i-1]	P[i]

KMP算法：Next数组

- 如果P[k] ≠ P[i]，如何计算Next[i+1]？
  - 首先，Next[i+1] < k+1；否则容易说明与Next[i]=k矛盾
  - 性质：若字符串S是P[0: i+1]的公共前后缀（下图第2行表示S），则S去掉最后一位后，构成P[0: i]的公共前后缀（下图第2行标红部分，记S'）
    - 进一步，还构成P[0, k]的公共前后缀
  - 利用P[0: k]的最长公共前后缀（假定已经求出Next[k]=k_1），并比较P[i]与P[k_1]
    - 如果不相等，继续考虑Next[k_1]，依此类推，直到发现公共前后缀

P[0]	……	P[k-1]	P[k]	……	P[i-k-1]	P[i-k]	……	P[i-1]	P[i]

P[0: k_1]	P[k_1]

P[0: k_1]	P[i]

P[0: k_2]	P[k_2]

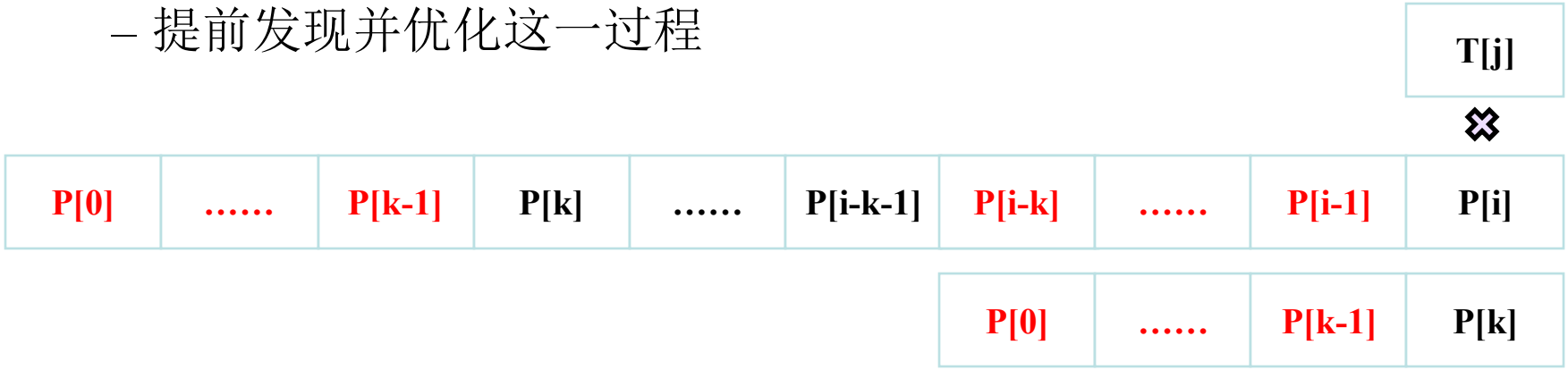
P[0: k_2]	P[i]

KMP算法的时间复杂性分析

- 总结：
  - 假设pattern的长度为m，target的长度为n（通常m<n）
  - 计算next数组的时间复杂度为O(m)
  - 匹配过程的时间复杂度为O(n)
  - KMP算法总体的时间复杂度为O(m+n)
    - 对比：朴素模式匹配算法的时间复杂度为O(m*n)，这是一个显著的改进！

KMP算法：改进Next数组

- 考虑P[i]与T[j]的失配：
  - 此时要将P右移(i-k)位，继续比较T[j]与P[k]（k=Next[i]）
- 如果P[i] = P[k]？
  - 也是必定失配的，因而要继续右移，
  - 继续比较T[j]与P[k’]（k’=Next[k]）
  - 提前发现并优化这一过程



•简单练习（三）

判断题

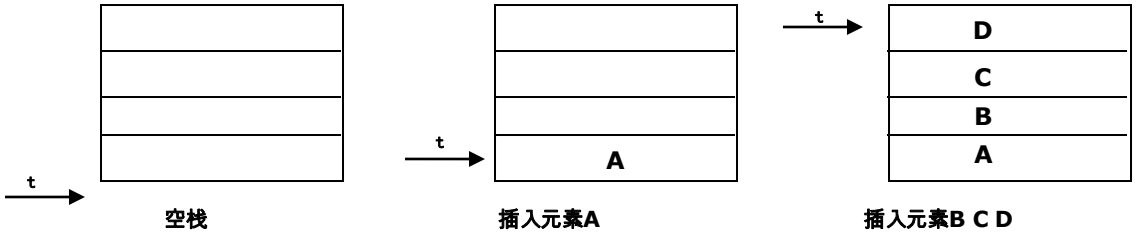
1. 若两个串含有相同的字符，则说它们相等;
2. KMP算法的最大特点是指示主串的指针不需回溯;

N Y

第四讲 栈

栈的定义

- 栈(Stack)是一种有次序的数据项集合，在栈中，数据项的加入和移除都仅发生在同一端
  - 这一端通常叫做栈“顶top”，另一端就叫做栈“底base”
- 这种次序通常称为“后进先出LIFO”：Last in First out



栈的抽象数据类型

- 抽象数据类型“栈”是一个有次序的数据集，每个数据项仅从“栈顶”一端加入到数据集中、从数据集中移除，栈具有后进先出LIFO的特性
- 栈是一种受限的线性结构，即仅允许在一端操作
- 抽象数据类型“栈”定义为如下的操作
  - Stack()：创建一个空栈，其中不包含任何数据项
  - push(item)：将item数据项加入栈顶，无返回值
  - pop()：将栈顶数据项移除，返回栈顶的数据项，栈被修改
  - peek()：“窥视”栈顶数据项，返回栈顶的数据项但不移除，栈不被修改
  - isEmpty()：返回栈是否为空栈
  - size()：返回栈中有多少个数据项

什么是递归(Recursion)

- 递归是一种解决问题的方法，其精髓是将问题分解为规模更小的相同问题，持续分解，直到问题规模小到可以用非常简单直接的方式来解决。
- 递归的问题分解方式非常独特，其算法方面的明显特征就是：在算法流程中调用自身。
- 递归为我们提供了一种对复杂问题的优雅解决方案，精妙的递归算法常会出奇简单，令人赞叹。

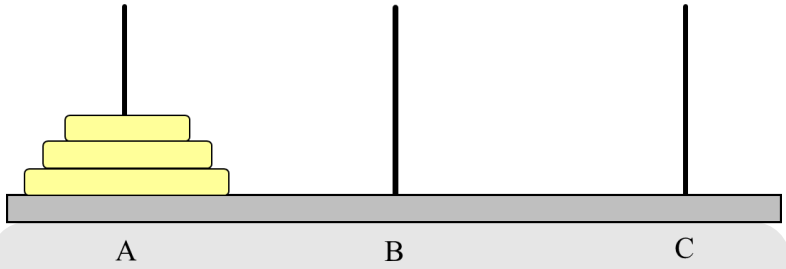
递归“三定律”

- 1. 递归算法必须有一个基本结束条件
  - 递归出口：最小规模问题的直接解决
- 2. 递归算法必须向基本结束条件演进
  - 减小问题规模
- 3. 递归算法必须调用自身
  - 关键：将问题分解成了规模更小的相同问题，通过解决更小规模的问题来解决原问题

递归程序是如何执行的

- 递归函数调用和返回过程的链条就是用栈来管理的。
- 例：Hanoi(3, ‘A’, ‘B’, ‘C’)

```
def hanoi(n, start, by, to):
    if n == 1:
        print(“%s -- %s” % (start, to))
        return
    else:
        hanoi(n-1, start, to, by)
        print(“%s -- %s” % (start, to))
        hanoi(n-1, by, start, to)
```



运行时内存（栈）

函数栈帧（Frame）

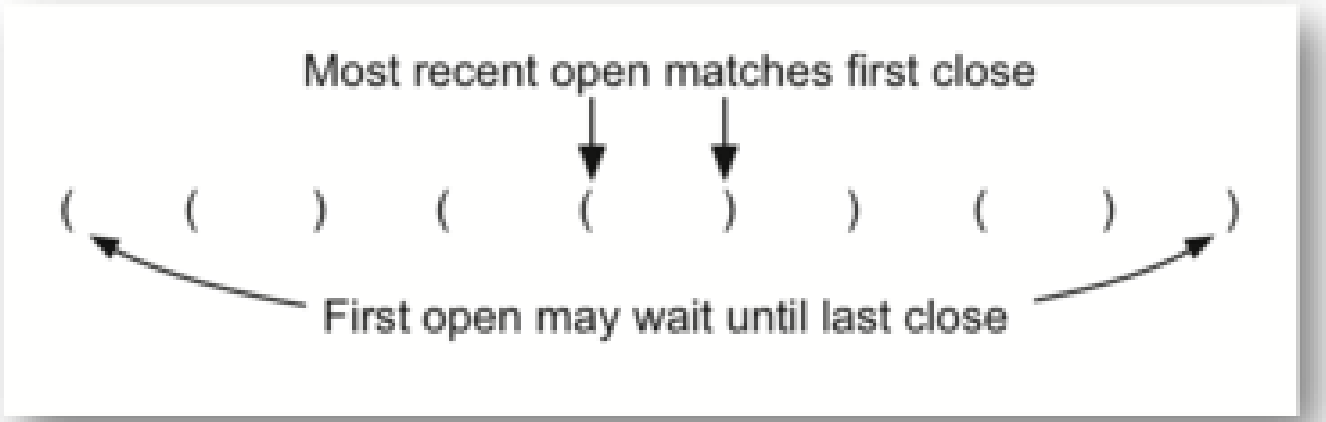
hanoi(1, ‘A’, ‘B’, ‘C’)
存储局部变量、返回值、内存基址等

hanoi(2, ‘A’, ‘C’, ‘B’)
存储局部变量、返回值、内存基址等

hanoi(3, ‘A’, ‘B’, ‘C’)
存储局部变量、返回值、内存基址等

栈的应用：括号匹配

- 对括号是否正确匹配的识别，是很多语言编译器的基础算法
- 下面分析如何构造这个括号匹配识别的算法
  - 从左到右扫描，最近打开的左括号，应该匹配最先遇到的右括号
  - 这样，第一个左括号（最早打开），就应该匹配最后一个右括号（最后遇到）
  - 这种次序反转的识别，正好符合栈的特性！



栈的应用：进制之间的转换

- 将整数在二进制和十进制之间转换
  - 如：$(233)_{10}$的对应二进制数为$(11101001)_2$，具体是这样：
  - $(233)_{10}=2\times10^2+3\times10^1+3\times10^0$
  - $(11101001)_2=1\times2^7+1\times2^6+1\times2^5+0\times2^4+1\times2^3+0\times2^2+0\times2^1+1\times2^0$
- 十进制转换为二进制，采用的是“除以2”的算法
  - 将整数不断除以2，每次得到的余数就是由低到高的二进制位
- 十进制转换为N进制，采用的是“除以N”的算法
  - 将整数不断除以N，每次得到的余数就是由低到高的N进制位
- 如何表示八进制及十六进制
  - 八进制可用8个不同数字0、1、2、3、4、5、6、7
  - 十六进制的16个数字则是0、1、2、3、4、5、6、7、8、9、A、B、C、D、E、F

栈的应用：前缀、中缀和后缀表达式

- 这种操作符（operator）介于操作数（operand）中间的表示法，称为中缀表示法，如： B*C
- 将操作符移到操作数的前面的表示法，称为前缀表达式（波兰表达式）
- 将操作符移到操作数的后面的表示法，称为后缀表达式（逆波兰表达式）
- 在**前缀和后缀表达式**中，**操作符的次序完全决定了运算的次序**，不再有混淆

栈的应用：通用的中缀转后缀算法

- 在从左到右扫描逐个字符扫描中缀表达式的过程中，**采用一个栈来暂存未处理的操作符**
- 栈顶的操作符就是最近暂存进去的，当遇到一个新的操作符，就需要跟栈顶的操作符比较下优先级，再行处理。

- 创建空栈opstack用于暂存操作符，空表用于保存后缀表达式
- 用split方法，将中缀表达式转换为单词（token）的列表
- 从左到右扫描中缀表达式单词列表
  - 如果单词是一个操作数，则直接添加到后缀表达式列表的末尾
  - 如果单词是一个左括号“(”，则压入opstack栈顶
  - 如果单词是一个右括号“)”，则反复弹出opstack栈顶的操作符，加入到输出列表末尾，直到碰到左括号
  - 如果单词是一个操作符“*/+-”，则压入opstack栈顶。但在压入之前，要比较其与栈顶操作符的优先级，如果栈顶的高于或等于它，就要反复弹出栈顶操作符，加入到输出列表末尾，直到栈顶的操作符优先级低于它
  - 中缀表达式单词列表扫描结束后，把opstack栈中的所有剩余操作符依次弹出，添加到输出列表末尾
- 把输出列表再用join方法合并成后缀表达式字符串，算法结束。

栈的应用：后缀表达式求值

- **在对后缀表达式从左到右扫描的过程中，由于操作符在操作数的后面，所以要暂存操作数，在碰到操作符的时候，再将暂存的两个操作数进行实际的计算**
  - 仍然是栈的特性：操作符只作用于离它最近的两个操作数

- 创建空栈operandStack用于暂存操作数
- 将后缀表达式用split方法解析为单词（stoken）的列表
- 从左到右扫描单词列表
  - 如果单词是一个操作数，将单词转换为整数int，压入operandStack栈顶
  - 如果单词是一个操作符（*/+-），就开始求值，从栈顶弹出2个操作数，先弹出的是右操作数，后弹出的是左操作数，计算后将值重新压入栈顶
- 单词列表扫描结束后，表达式的值就在栈顶
- 弹出栈顶的值，返回。

•简单练习（四）

填空题

1. 若已知一个栈的入栈序列是1，2，3，… n，其输出序列为$p_1, p_2, p_3, \ldots, p_n$，若$p_1=n$，则$p_i$为 **n-i+1** 。

选择题

2. 设有三个元素X，Y，Z 顺序进栈（进的过程中允许出栈），下列得不到的出栈排列是（ C ）。

- A：XYZ B：YZX C：ZXY D：ZYX

第五讲 队列

队列Queue：什么是队列？

- 队列是一种有次序的数据集合，其特征是，新数据项的添加总发生在一端（通常称为“尾rear”端），而现存数据项的移除总发生在另一端（通常称为“首front”端）
  - 当数据项加入队列，首先出现在队尾，随着队首数据项的移除，它逐渐接近队首。
- 新加入的数据项必须在数据集末尾等待，而等待时间最长的数据项则是队首。这种次序安排的原则称为（FIFO:First-in first-out）先进先出，或者叫“先到先服务first-come first-served”
- 队列的例子出现在我们日常生活的方方面面：排队
- 队列仅有一个入口和一个出口，不允许数据项直接插入队中，也不允许从中间移除数据项

队列

- 允许进行删除的这一端叫队列的头。
- 允许进行插入的这一端叫队列的尾。
- 当队列中没有任何元素时，称为空队列。
- 队列的插入操作通常称为进队列或入队列。
- 队列的删除操作通常称为退队列或出队列。

抽象数据类型Queue

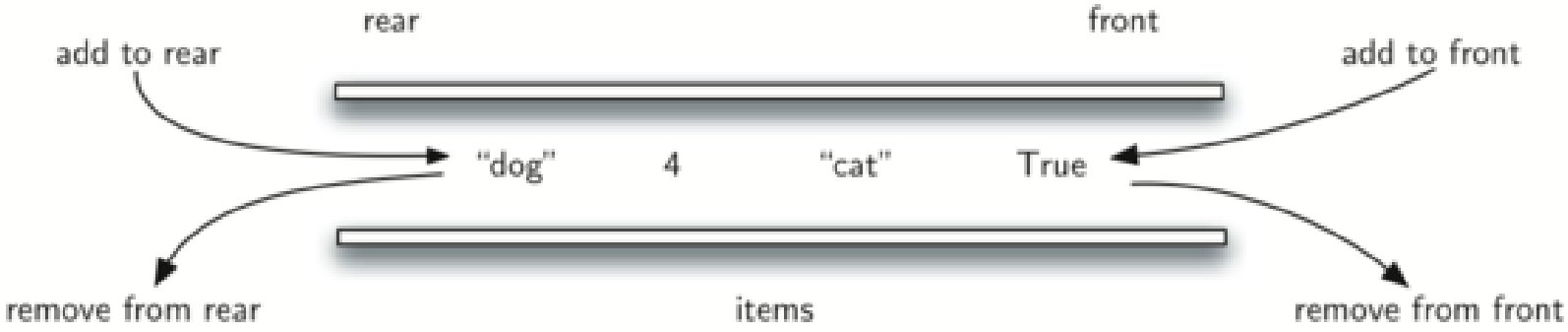
- 抽象数据类型Queue是一个有次序的数据集合，数据项仅添加到“尾rear”端，而且仅从“首front”端移除，Queue具有FIFO的操作次序
- 抽象数据类型Queue由如下操作定义：
  - Queue()：创建一个空队列对象，返回值为Queue对象；
  - enqueue(item)：将数据项item添加到队尾，无返回值；
  - dequeue()：从队首移除数据项，返回值为队首数据项，队列被修改；
  - is_empty()：测试是否空队列，返回值为布尔值；
  - size()：返回队列中数据项的个数。

队列的应用

- 热土豆问题（约瑟夫问题）
- 打印任务模拟算法

双端队列Deque：什么是Deque？

- 双端队列Deque是一种有次序的数据集，跟队列相似，其两端可以称作“首”“尾”端，但deque中数据项既可以从队首加入，也可以从队尾加入；数据项也可以从两端移除。
  - 某种意义上说，双端队列集成了栈和队列的能力。
- 但双端队列并不具有内在的LIFO或者FIFO特性，如果用双端队列来模拟栈或队列，需要由使用者自行维护操作的一致性



抽象数据类型Deque

- 抽象数据类型Deque是一个有次序的数据集，数据项可以从两端加入或者移除
- deque定义的操作如下：
  - Deque()：创建一个空双端队列
  - add_front(item)：将item加入队首
  - add_rear(item)：将item加入队尾
  - remove_front()：从队首移除数据项，返回值为移除的数据项
  - remove_rear()：从队尾移除数据项，返回值为移除的数据项
  - is_empty()：返回deque是否为空
  - size()：返回deque中包含数据项的个数

双端队列的应用

- “回文词”判定
- 滑动窗口案例：一个大小为k的滑动窗口从长度为n的序列最左端移动到最右端，窗口每次向右滑动一个数。请输出窗口在每个位置时窗口内的最大值和最小值。

•简单练习（五）

填空题

1. 假设以数组A[m]存放循环队列的元素，其头指针是front，当前队列有k个元素，则队列的尾指针为_____(front+k)%m_____。

选择题

- 设环形队列中数组长度为 n，头尾指针分别为f和r，则队列中现有元素个数为：（ ）

  A．r-f

  B．r-f+1

  C．(r-f+1) mod n

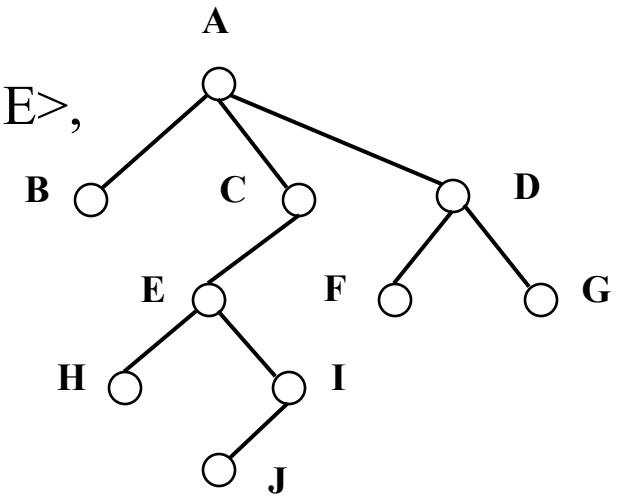
  D．(r-f+n) mod n

D

第六讲 二叉树

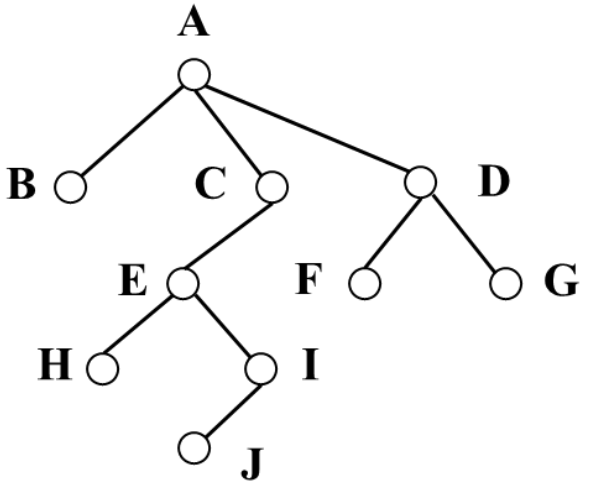
树的逻辑结构

- 包含n个节点的有穷集合K (n>0)，且在K上定义了一个关系 r，关系 r 满足以下条件：
  - 有且仅有一个节点$k_0 \in K$，它对于关系 r 来说没有前驱。节点k_0称作树的根；
  - 除节点k_0外，K中的每个节点对于关系 r 来说都有且仅有一个前驱；
- T = (N, R)
  - 结点集合 N = { A, B, C, D, E, F, G, H, I, J}
  - N上的关系 R = { <A, B>, <A, C>, <A, D>, <C, E>, <D, F>, <D, G>, <E, H>, <E, I>, <I, J> }



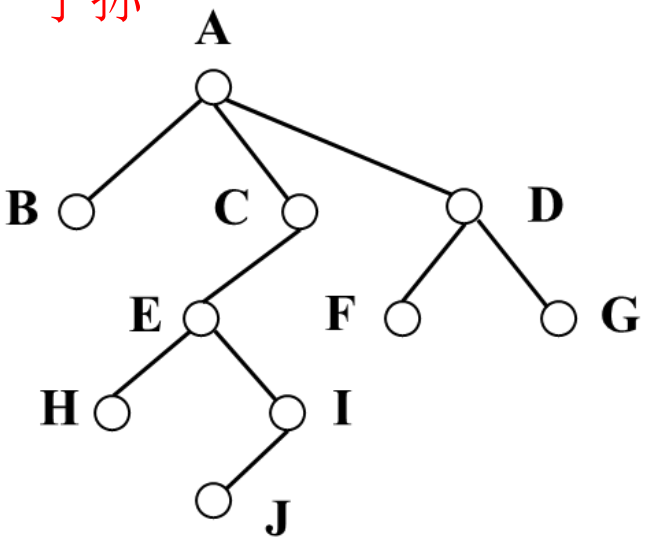
树的基本术语

- 父节点、子节点、边
  - 节点y是节点x的一棵子树的根，则x称作y的“父节点”（或父母）；
  - y称作x的“子节点”（或子女）；
  - 有序对<x，y>称作从x到y的“边”
- 兄弟节点
  - 具有同一父母的节点彼此称作“兄弟节点”
  - 树t中B，C，D互为兄弟节点，F，G互为兄弟节点，等等。
  - 注意，E和F并不是兄弟节点



树的基本术语

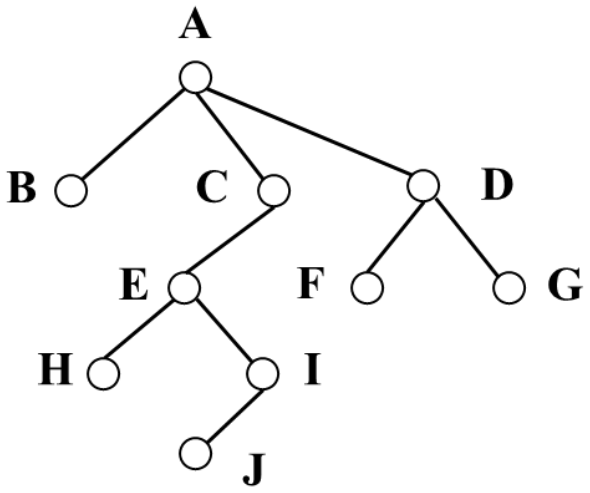
- 祖先、子孙
  - 若节点y在以节点x为根的一个子树（或树）中，且y≠x，称x是y的“祖先”，y是x的“子孙”



- 路径、路径长度
  - 对树上任意两个节点V_i，V_j，必然存在唯一的不重复的节点序列$\{V_i, P_1, P_2, \ldots, P_m, V_j\}$，使得$V_i$和$P_1$之间有边，$P_m$和$V_j$之间有边，$P_k$和$P_{k+1}$之间有边，这个节点序列就被称作为$V_i$到$V_j$的**路径**。
  - 路径上边的数量被称作是**路径长度**。

树的基本术语

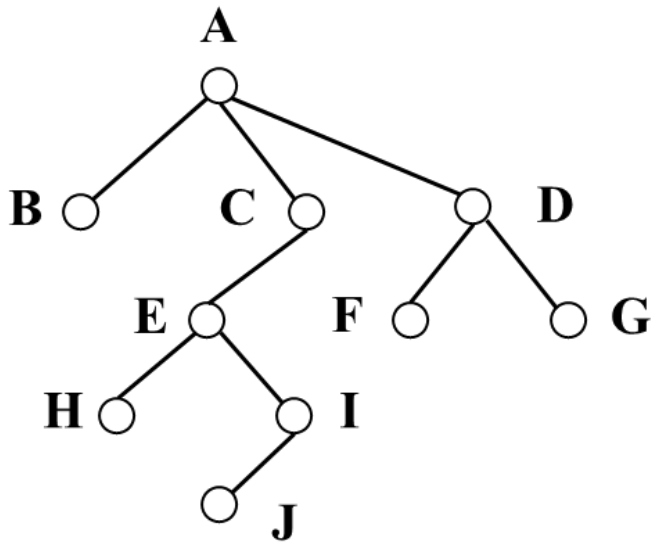
- 节点的层数、树的高度
  - **层数**：根为第 0 层
    - 其他节点的层数等于其父节点的层数加 1
  - **深度/高度：层数最大的叶节点的层数**
    - 也有些定义是上面的数值+1（取决于是否把根节点算上）



树的基本术语

节点的度数、树的度数

– 节点的子女个数叫作节点的“度数”。树中度数最大的节点的度数叫作“树的度数”



树叶、分支节点

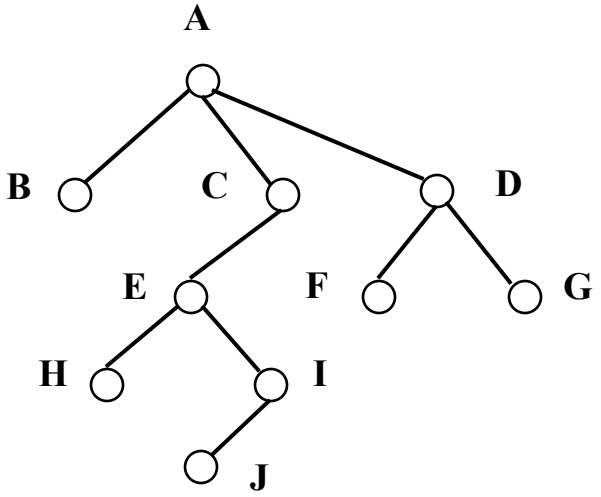
– 度数为0的节点称作“树叶”（又叫终端节点）

– 度数大于0的节点称作“分支节点”或非终端节点

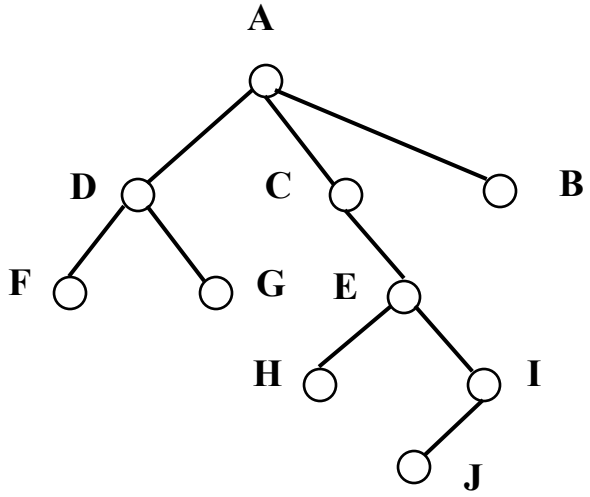
❖注意，节点的度数为1时，虽然只有一个子女，也叫分支节点。这两个术语对于根节点也不例外

树的基本术语

- 无序树、有序树
  - 对子树的次序不加区别的树叫作“无序树”。对子树之间的次序加以区别的树叫作“有序树”
  - 例如在下图中，按无序树的概念t和t’是同一棵树，按有序树的概念则是不同的树，**本课程讨论的树一般是有序树**



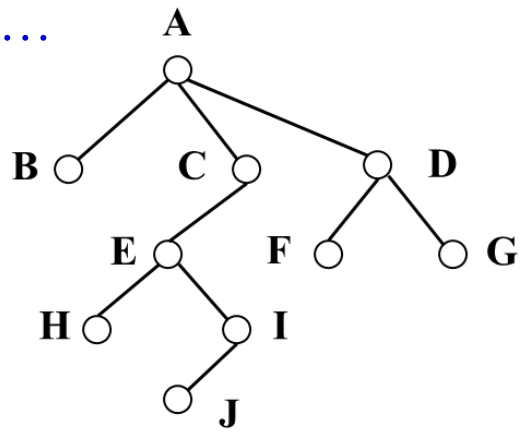
树t



树t’

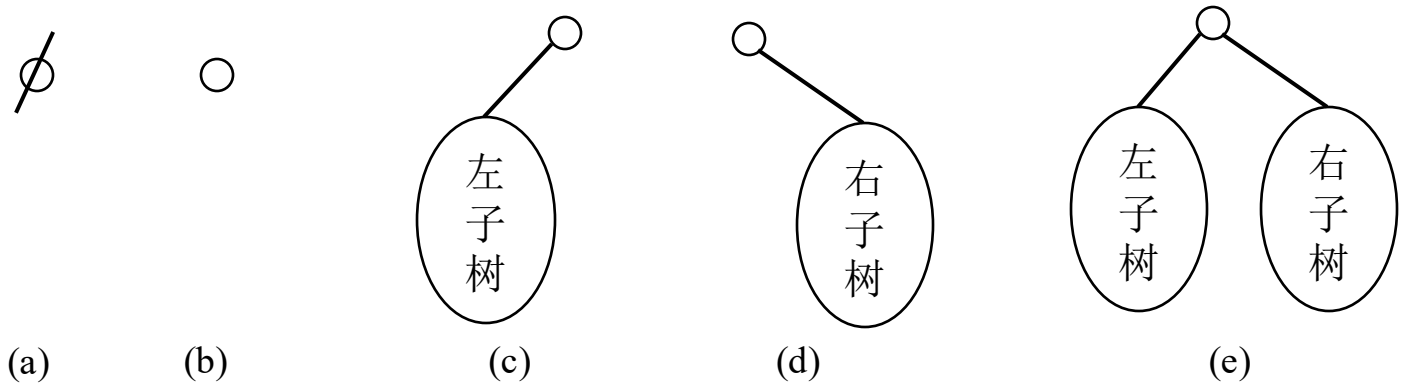
树的基本术语

- 节点的次序
  - 在有序树中可以从左到右地规定节点的次序
  - 例如下图中，节点B，C，D是从左到右排序的；可以说节点C是节点B右边的节点，是节点D左边的节点
  - 节点C的所有子女都在节点B及其子女的右边，而在节点D及其子女的左边
  - 按从左到右的顺序，可以把一个节点的子节点中最左边的节点简称“长子”，长子右边的节点称为“次子”，…



二叉树

- 递归定义：
  - 节点的有限集合，这个集合或者为**空集**，或者由一个**根**及两棵不相交的分别称作这个根的“**左子树**”和“**右子树**” 的二叉树组成。
- 二叉树的五种基本形态

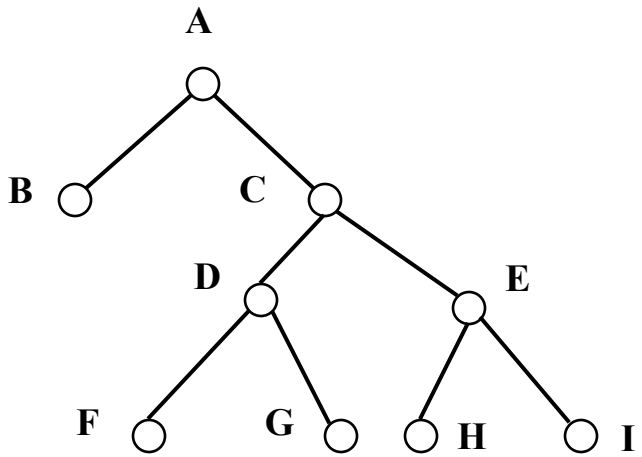


二叉树 vs 树

- 二叉树的术语与树中的相应部分类似
- 二叉树不是树的特殊情形，它们是两种数据结构
- 树和二叉树之间最主要的差别：
  - 二叉树中节点的子树要区分为左子树和右子树，即使在节点只有一棵子树的情况下也要明确指出该子树是左子树还是右子树
  - 譬如上例中，（c）和（d）是两棵不同的二叉树，但作为树，它们是相同的
- 度为 2 的有序树并不是二叉树
  - 第一子节点被删除，第二子节点就成为第一子节点

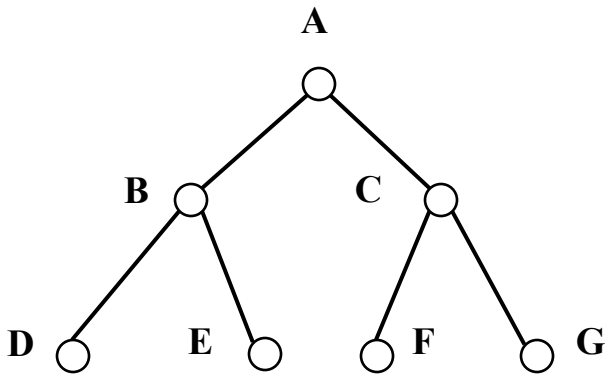
满二叉树（Full Binary Tree）

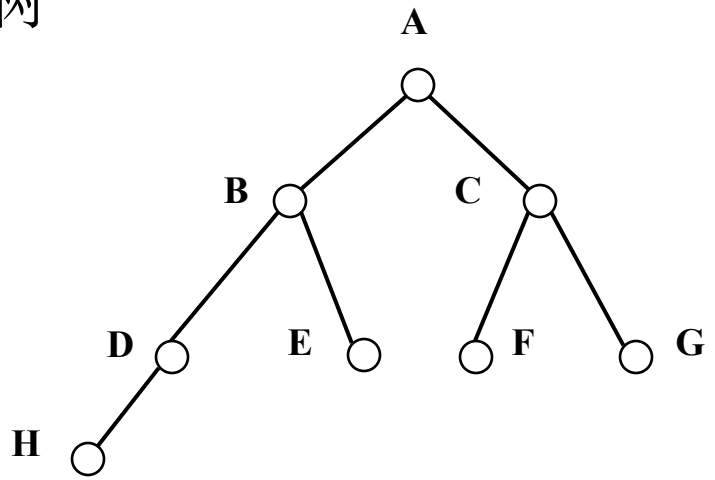
- 如果一棵二叉树的任何节点，或者是树叶，或者恰有两棵非空子树，则此二叉树称作满二叉树



完全二叉树（Complete Binary Tree）

- 若一棵二叉树，最多只有最下面的两层节点度数可以小于2，且最下面一层的节点都集中在该层最左边的若干位置上，则称此二叉树为完全二叉树

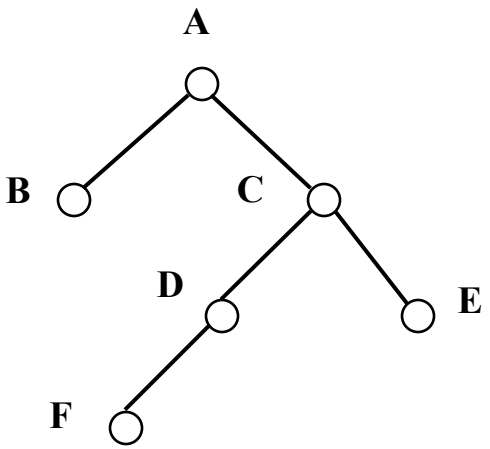


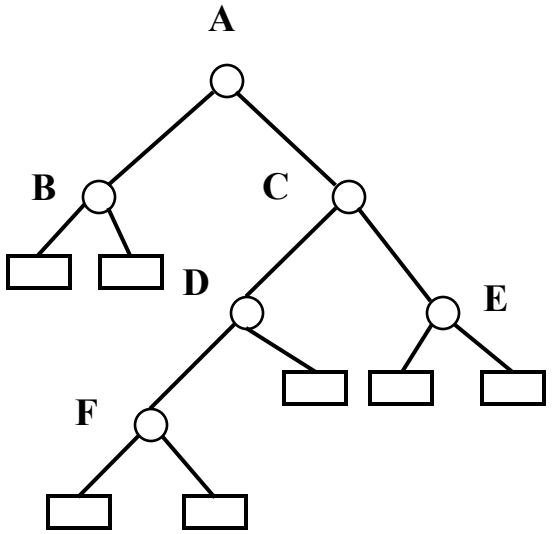


- 在许多算法和算法分析中都明显地或隐含地用到完全二叉树的概念

扩充二叉树（Extended Binary Tree）

- 扩充二叉树：把原二叉树的节点都变为度数为2的分支节点
  - 如果原节点的度数为2，则不变
  - 度数为1，则增加一个分支
  - 度数为0（树叶）增加两个分支





=>满二叉树

二叉树的基本性质

- 性质1：在非空二叉树的第i层上至多有2^i个节点($i \geq 0$)
- 性质2：深度为k的二叉树中最多有$2^{k+1} - 1$个节点($k \geq 0$)
- 性质3：对于任何一棵非空的二叉树，如果叶节点个数为n_0，度为2的节点个数为n_2，则有$n_0 = n_2 + 1$
- 性质4：具有n个节点的完全二叉树的深度k为 $\lfloor \log_2 n \rfloor$

二叉树的基本性质

- 性质5：对于具有n个节点的完全二叉树，如果按照从上到下和从左到右的顺序对树中的所有节点从1开始进行编号，则对于任意的序号为 i 的节点，有：
  - 如果i＞ 1 ，则其父节点的序号为 i // 2；

    如果i＝1 ，则其是根节点，它没有父节点。
  - 如果2i≤n，则其左子节点的序号为2i；

    如果2i＞n，则其没有左子节点。
  - 如果2i+ 1 ≤n，则其右子节点的序号为2i+1；

    如果2i+ 1 ＞n，则其没有右子节点。
- 性质6：在满二叉树中，叶节点个数比分支节点个数多1。
- 性质7：在扩充的二叉树里，新增加的外部节点的个数比原来的内部节点个数多1。
- 性质8：对任意扩充二叉树，E 和 I 之间满足：E＝I＋2n，其中n是内部节点个数。

二叉树的实现

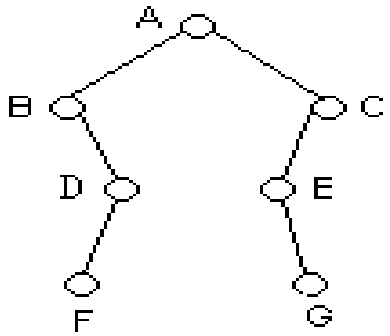
- 根据应用的不同以及二叉树本身的不同特点，二叉树可采用不同的存储结构
  - 顺序表示
  - 链式表示

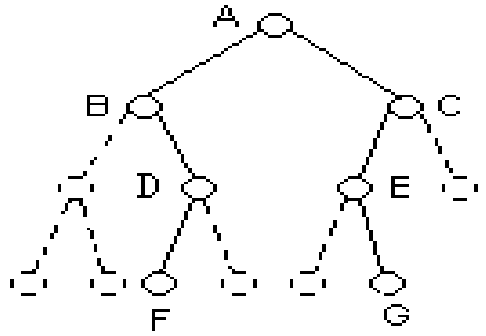
完全二叉树的顺序表示

- 完全二叉树节点的顺序存储
  - 所有节点按层次顺序依次存储在连续的存储单元中，
  - 根据一个节点的存储地址就可算出它的左右子女，父母的存储地址，如同存储了相应的指针一样。
- 顺序表示是存储完全二叉树的最简、最节省空间的存储方式
  - 完全二叉树的顺序存储，在存储结构上是线性的，但在逻辑结构上它仍然是二叉树型结构

一般二叉树的顺序表示

- 增加空节点来构造一棵完全二叉树，再以二叉树的方式存储
- 接近于完全二叉树的形态，需要增加的空节点数目不多，则也可采用顺序表示；

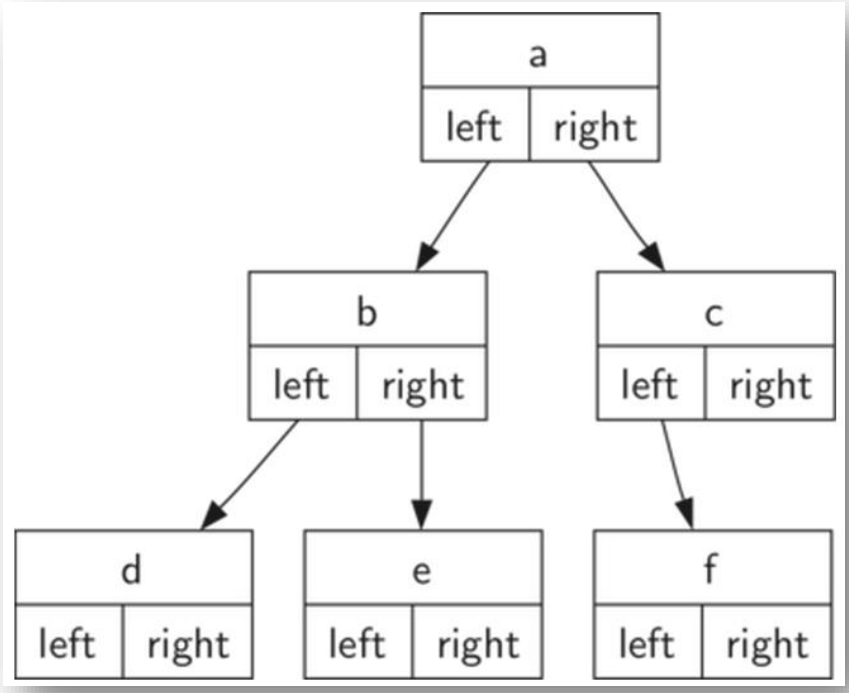




数组下标	0	1	2	3	4	5	6	7	8	9	10	11	12
	A	B	C	∧	D	E	∧	∧	∧	F	∧	∧	G

二叉树的链式存储结构

- 同样可以用类似链表的节点链接法来实现树
  - –每个节点保存根节点的数据项，以及指向左右子树的链接
- 定义一个BinaryTree类
  - –成员key保存根节点数据项
  - –成员left_child/right_child则保存指向左/右子树的引用（同样是BinaryTree对象）



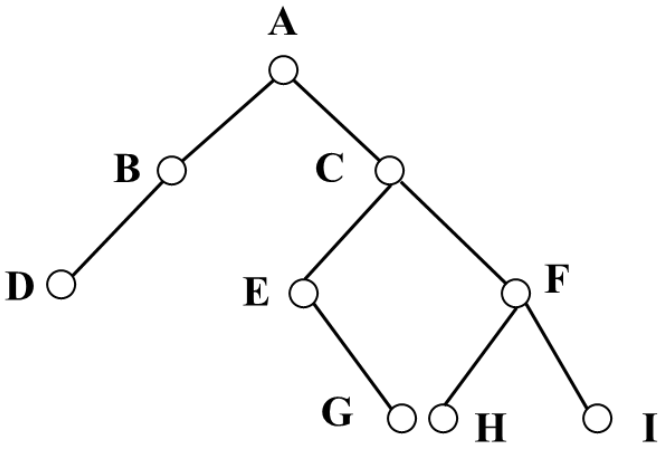

```
class BinaryTree:
    def __init__(self, rootobj):
        self.key = rootobj
        self.left_child = None
        self.right_child = None
```

二叉树的遍历概念

- 二叉树的遍历是指按某种方式访问二叉树中的所有节点，使每个节点被访问一次且只被访问一次
  - 遍历二叉树的过程实际上就是把二叉树的节点放入一个线性序列的过程，或者说是把二叉树进行线性化
- 按照遍历的顺序，可以将遍历方式分成：
  - 深度优先遍历（Depth-first traversal）
    - 按深度方向，尽可能深地访问节点，然后回溯
    - 按照访问根节点的时机，进一步分成：前序、中序、后序
  - 广度优先遍历（Breadth-first traversal）
    - 按层次顺序，逐层访问所有节点

前序遍历

- 若以符号D、L、R分别表示访问根节点、遍历根节点的左子树、遍历根节点的右子树。



- 前序次序：DLR
  (1) 访问根；
  (2) 按前序次序遍历左子树；
  (3) 按前序次序遍历右子树。

- 将按前序次序对一棵二叉树遍历得到的线性表称为这棵二叉树的前序序列。
  – 此二叉树的前序序列为：A B D C E G F H I

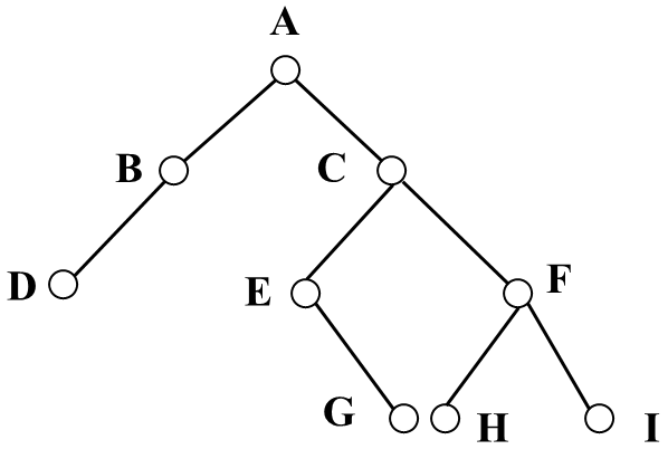
中序遍历

- 中序次序(对称序法)：LDR

  (1) 按对称序次序遍历左子树；

  (2) 访问根；

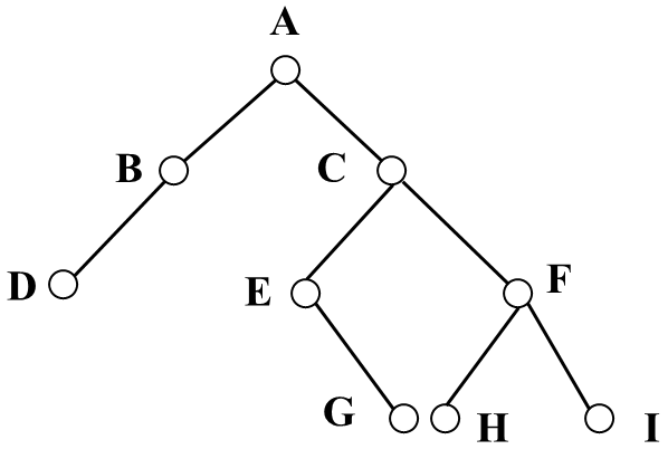
  (3) 按对称序次序遍历右子树。



- 将按中序次序对一棵二叉树遍历得到的线性表称为这棵二叉树的中序序列。
  - 中序(对称序)序列是： D B A E G C H F I

后序遍历

- 后序次序：LRD
  (1) 按后序次序遍历左子树；
  (2) 按后序次序遍历右子树；
  (3) 访问根。



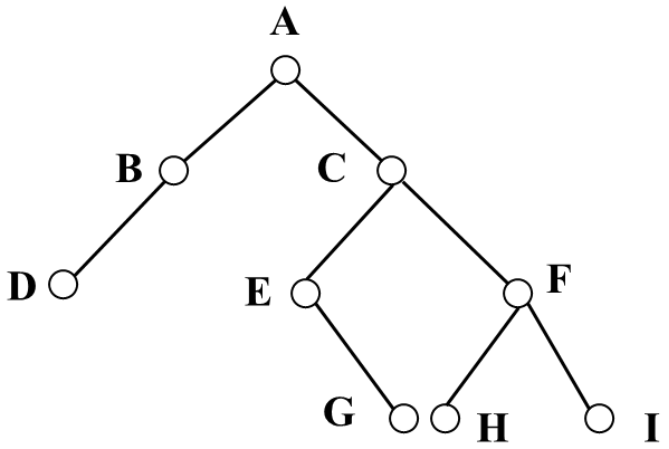
- 将按后序次序对一棵二叉树遍历得到的线性表称为这棵二叉树的后序序列。
  – 此二叉树的后序序列是： D B G E H I F C A

深度优先遍历算法的复杂度分析

- 时间复杂度为O(n)，n为树中节点的数量
  - 遍历每个节点本身只需要O(1)的操作
- 空间复杂度为递归调用栈的深度，与树的高度有关
  - 最好情况下，树为完全二叉树，高为 O(log n)
  - 最坏情况下，树退化为线性结构，高为 O(n)

广度（宽度）优先遍历二叉树

- 从二叉树的第 0 层（根节点）开始，自上而下 逐层遍历；在同一层中，按照 从左到右 的顺序对节点逐一访问。
- 例如：A B C D E F G H I



优先队列Priority Queue

- 优先队列的出队dequeue操作跟队列一样，都是从队首出队；
- 但在优先队列内部，数据项的次序却是由“优先级”来确定：高优先级的数据项排在队首，而低优先级的数据项则排在后面。
  - 这样，优先队列的入队操作就比较复杂，需要将数据项根据其优先级尽量挤到队列前方。
- 思考：有什么方案可以用来实现优先队列？
  - 出队和入队的复杂度大概是多少？

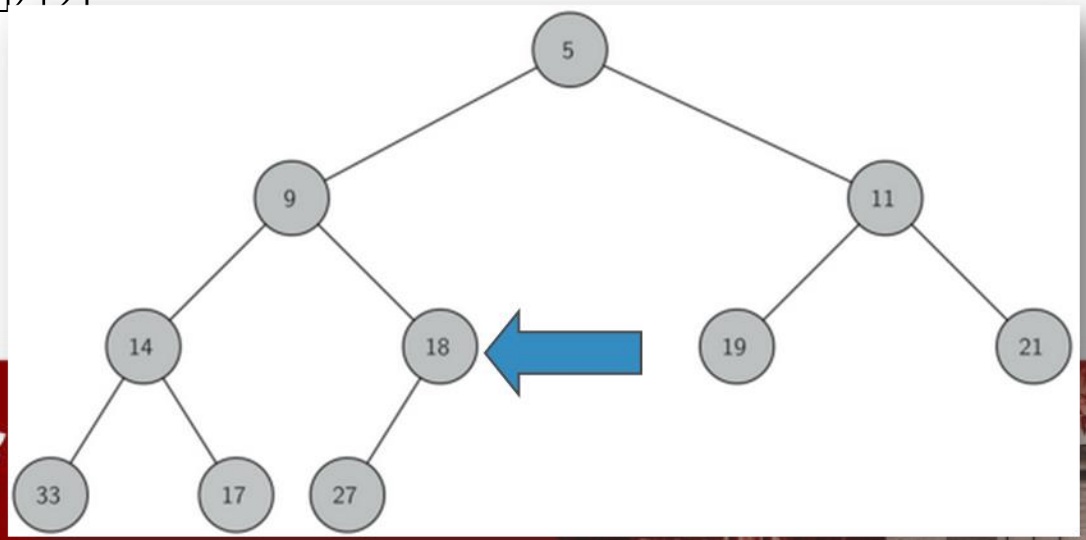
二叉堆Binary Heap实现优先队列

- 实现优先队列的经典方案是采用二叉堆数据结构
  - 二叉堆能够将优先队列的入队和出队复杂度都保持在O(log n)
- 二叉堆的有趣之处在于，其逻辑结构上象二叉树，却是用非嵌套的列表来实现的！
- 最小key排在队首的称为“最小堆min heap”
  - 反之，最大key排在队首的是“最大堆max heap”

- ADT BinaryHeap的操作定义如下：
  - BinaryHeap()：创建一个空二叉堆对象；
  - insert(k)：将新key加入到堆中；
  - findMin()：返回堆中的最小项，最小项仍保留在堆中；
  - delMin()：返回堆中的最小项，同时从堆中删除；
  - isEmpty()：返回堆是否为空；
  - size()：返回堆中key的个数；
  - buildHeap(list)：从一个key列表创建新堆

用完全二叉树实现堆

- 为了使堆操作能保持在对数水平上，必须采用二叉树结构；
- 同样，如果要使操作始终保持在对数数量级上，就必须始终保持二叉树的“平衡”——树根左右子树拥有相同数量的节点
- 我们采用“完全二叉树”的结构来近似实现“平衡”
  - 完全二叉树，叶节点最多只出现在最底层和次底层，而且最底层的叶节点都连续集中在最左边，每个内部节点都有两个子节点，最多可有1个节点例外



第七讲 Huffman树

问题的提出

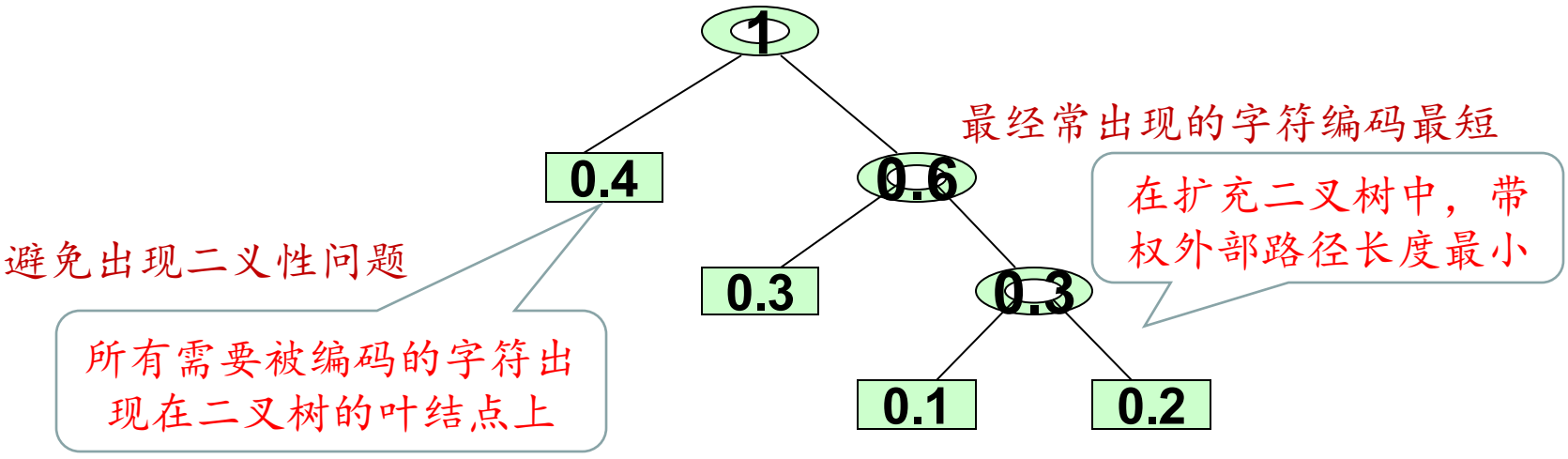
- 数据通信的二进制编码问题描述：
  - 设需要编码的字符集合$d=\{d_1, d_2, \ldots, d_n\}$，d中各种字符出现的频率$w=\{w_1, w_2, \ldots, w_n\}$，要对d里的字符进行二进制编码，使得

① 通信编码总长最短

② 若$d_i \neq d_j$，则d_i的编码不可能是d_j的编码的前缀。（这样就使得译码可以一个字符一个字符地进行，不需要在字符与字符之间添加分隔符）

概述

- 构造哈夫曼树是不定长编码的一种解决方案
  - 所构造哈夫曼树是一棵二叉树
  - 利用根节点到叶结点的路径进行编码
- 例对字符集 {A(0.1), B(0.2), C(0.3), D(0.4)}



哈夫曼树

- 哈夫曼（ Huffman ）树定义：
- 设有一组实数｛$w_1, w_2, w_3, \ldots, w_m$｝，现构造一棵以$w_i$（i = 1, 2, …，m）为权的m个外部结点的扩充的二叉树，使得该树带权外部路径长度最小。

带权外部路径长度

- 设扩充二叉树具有m个带权值的外部结点，那么从根结点到各个外部结点的路径长度与相应结点权值的乘积的和，叫做扩充二叉树的带权的外部路径长度。

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

  - w_i是第i个外部结点的权值
  - L_i为从根到第i个外部结点的路径长度
  - m为外部结点的个数

规律：权越大的叶子离根越近，则二叉树的带权外部路径长度就越小

构造哈夫曼树—哈夫曼算法

- 如何构造一棵哈夫曼树呢？
  - 最早给出了带有一般规律的算法，称哈夫曼算法

① 根据给定的n个权值$\{w_1,w_2,\ldots,w_n\}$，构成n棵二叉树的集合$F=\{T_1,T_2,\ldots,T_n\}$，其中每一棵二叉树T_i中只有一个带权为w_i的根结点，其左右子树为空。

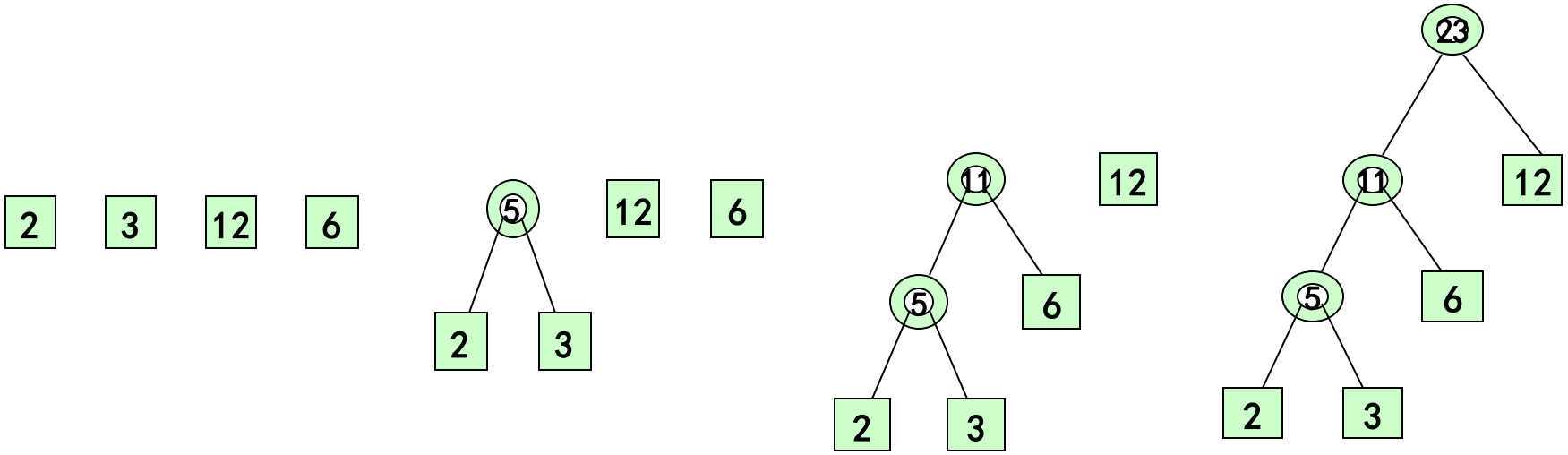
② 在F中选取两棵权值最小的树作为左右子树以构造一棵新的二叉树，且新二叉树的根结点的权值为其左右子树根结点权值之和。

③ 在F中删除这两棵树，同时将新得到的二叉树加入F。

④ 重复(2)和(3)，直到F中只含一棵树为止。

哈夫曼树的构造过程

- 设字符集 {A, B, C, D}出现的频率（对应的权值）分别为 { 2, 3, 6, 12}

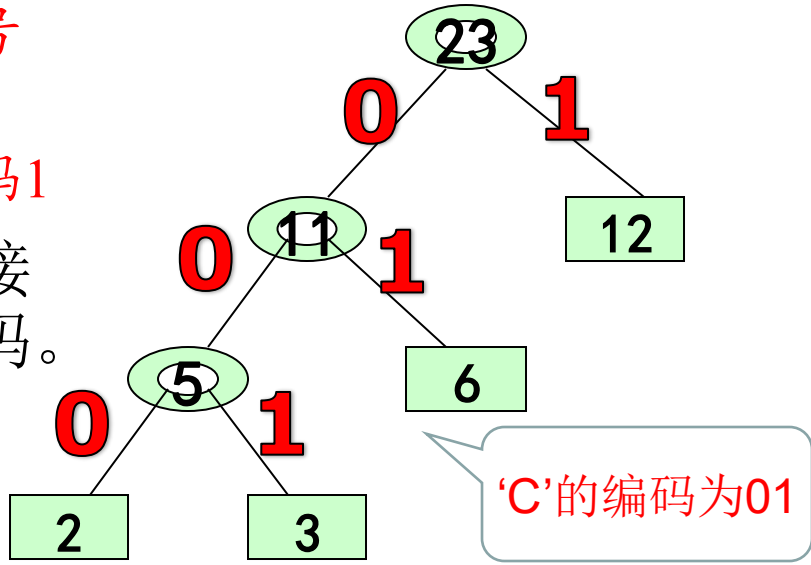


4棵只有根的二叉树　　2、3合并得到3棵二叉树　　5、6合并得到2棵二叉树　　11、12合并得到1棵二叉树

左右选择不同，得到形态不同的Huffman树，
但它们的WPL相同。

哈夫曼编码

- 根据字符出现的概率，建立Huffman树以后，
  - 把从每个结点引向其左子女的边标上号码0
  - 把从每个结点引向右子女的边标上号码1
- 从根到每个叶子的路径上的号码连接起来就是这个叶子代表的字符的编码。



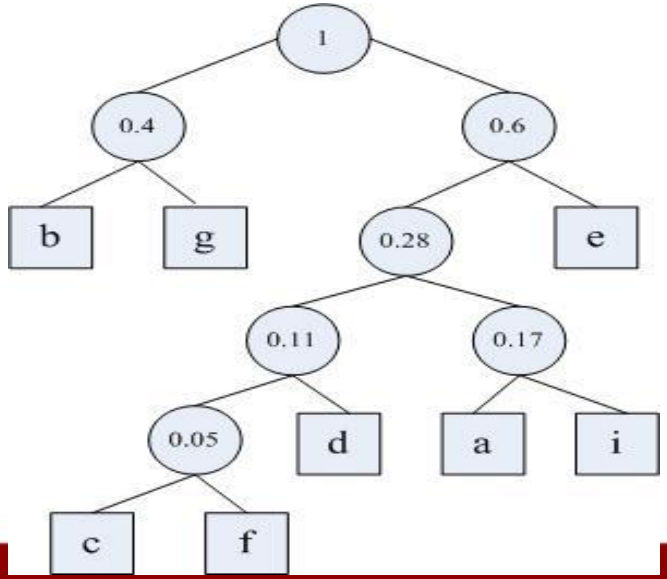
哈夫曼编码的译码

- 由Huffman编码得到的二进制序列如何译码？
  - 从二叉树的根结点开始，对二进制序列的第一个字符开始匹配，如果为0，沿左分支走，如果为1，沿右分支走，直到找到一个叶结点为止，则确定一条到达树叶的路径，译出一个字符。
  - 回到树根，从二进制位串中的下一位开始继续译码
  - 如：5个字符(A, B, C, D, E)的出现频率为(20, 5, 30, 30, 15),
    - 可能得到的编码为：　00, 010, 10, 11, 011
    - 二进制序列：　0001001011011011110 10
    - 译码：A　B　B　D　E　E　D　C　C

简单练习（六）

解答题

- 假设通讯电文由8个字符：a、b、c、d、e、f、g、i组成，
  - 出现频率依次为0.07、0.19、0.02、0.06、0.32、0.03、0.21、0.10
  - 为这8个字母设计哈夫曼编码
  - 请画出哈夫曼树，并给出字符串“edfa”的编码。



“11 1001 10001 1010”
或者
“00 0110 01110 0101”

第八讲 二叉搜索树

二叉搜索树

- 二叉树的一个主要用途是提供对数据（包括索引）的快速检索，而一般的二叉树对此并不具有性能优势
- 常用名称（同义词）
  - 二叉搜索树（Binary Search Tree,简称BST）
  - 二叉查找树
  - 二叉检索树
  - 二叉排序树

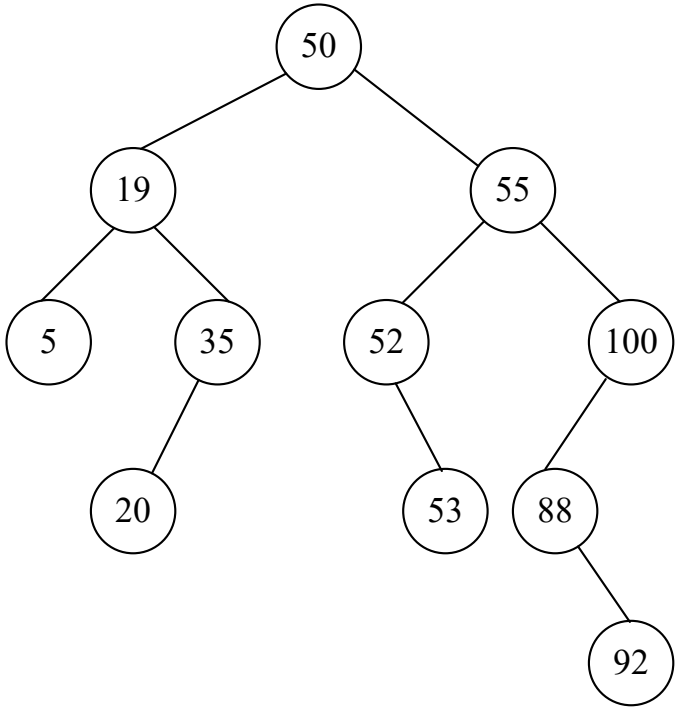
二叉搜索树：定义

- 假定存储在二叉树中的数据元素包含若干个域（field），其中一个称为码（key）K的域作为检索的依据，则二叉搜索树如下定义：
- 或者是一棵空树；
- 或者是具有下列性质的二叉树：
  - 对于任何一个结点，设其值为K
  - 则该结点的左子树(若不空)的任意一个结点的值都小于 K；
  - 该结点的右子树(若不空)的任意一个结点的值都大于 K；
  - 而且它的左右子树也分别为BST

二叉搜索树上的操作：检索、插入（生成）、删除

二叉搜索树的性质

- 按照中序遍历一棵二叉搜索树得到的序列将是按照码值由小到大的排列



二叉搜索树上的检索

- 从根结点开始，在二叉搜索树中检索值K。
  - 如果根结点储存的值为K，则检索结束
  - 如果K小于根结点的值，则只需检索左子树
  - 如果K大于根结点的值，就只检索右子树

- 一直持续到K被找到或者遇上了一个树叶
  - 如果遇上树叶仍没有发现K，那么K就不在该二叉搜索树中

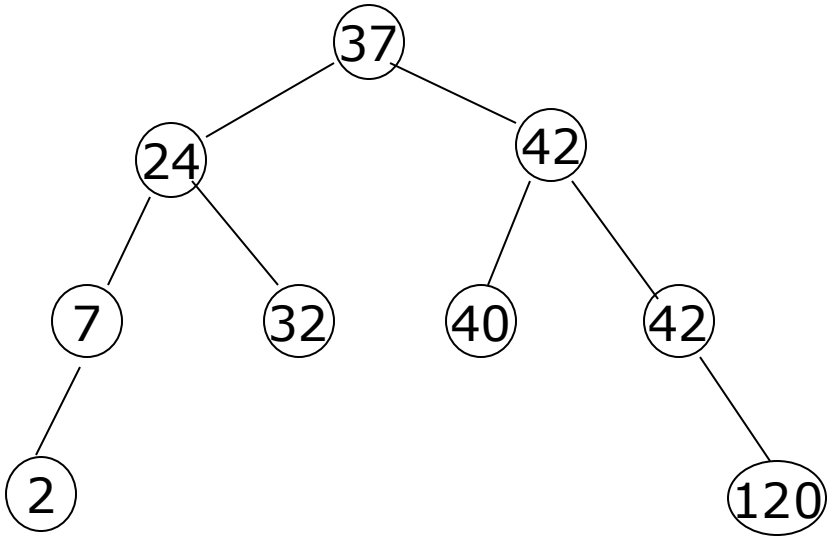
二叉搜索树上的插入

- 设待插入的结点码为 K:
  - 从根结点开始，若根结点为空，则将码K的结点作为根插入，操作结束
  - 如果K小于根结点的值，则将其按照二叉搜索树的要求插入左子树
  - 如果K大于根结点的值，就将其按照二叉搜索树的要求插入右子树
- 保证结点插入后仍符合二叉搜索树的定义

二叉检索树上的删除

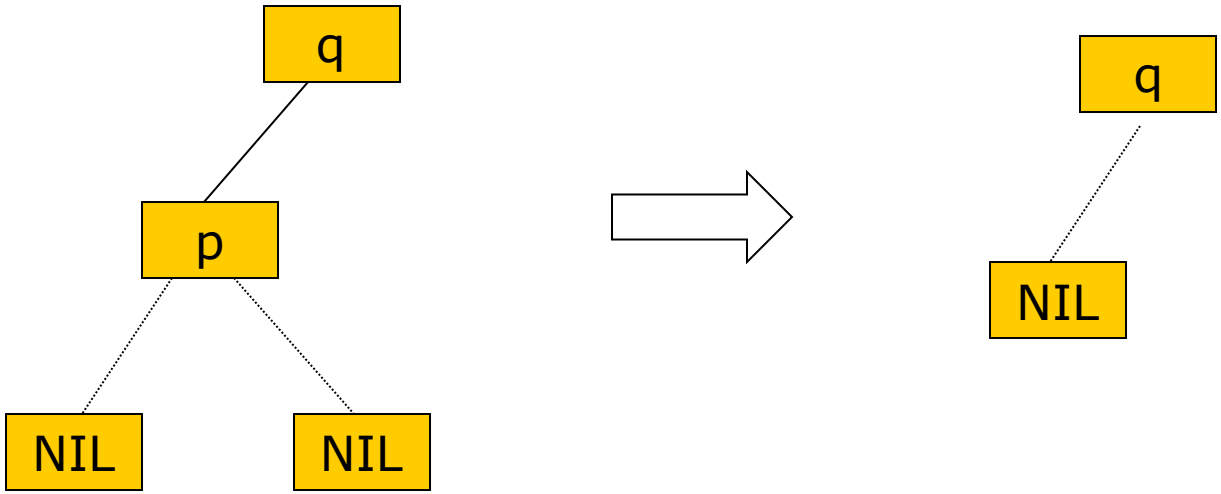
- 在二叉检索树删除一个结点，相当于删除有序序列中的一个记录，要求删除后仍能保持二叉检索树的排序特性，并且树高变化较小

删除2



删除示例1

情况1. 叶结点可以直接删除，其父结点的相应指针置为空

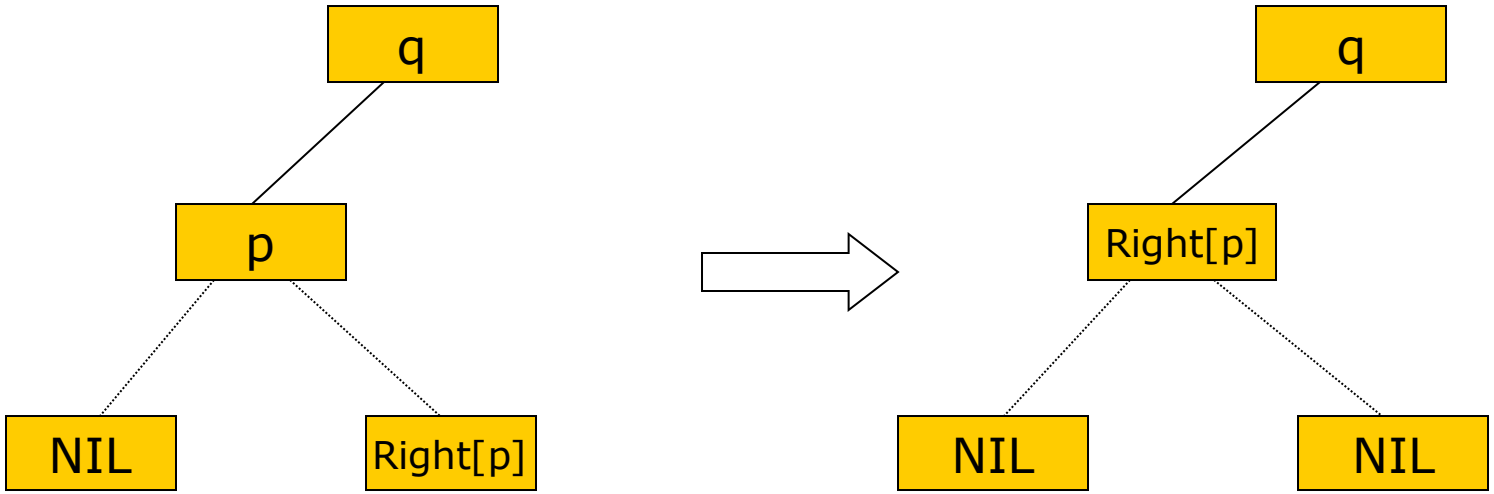


删除示例2

情况2. 只有一个子女的结点p被删除时，分以下情况：

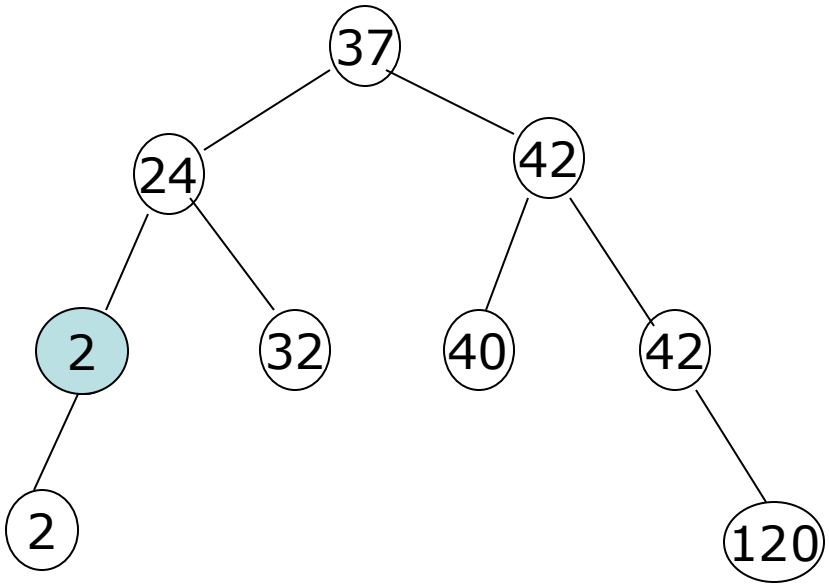
1. p是q的左子结点， p只有左子结点或右子结点
2. p是q的右子结点， p只有左子结点或右子结点

可让此子女直接代替即可



删除示例2

删除7



删除示例3

情况3. 被删除结点 p 的左右子女皆不空。

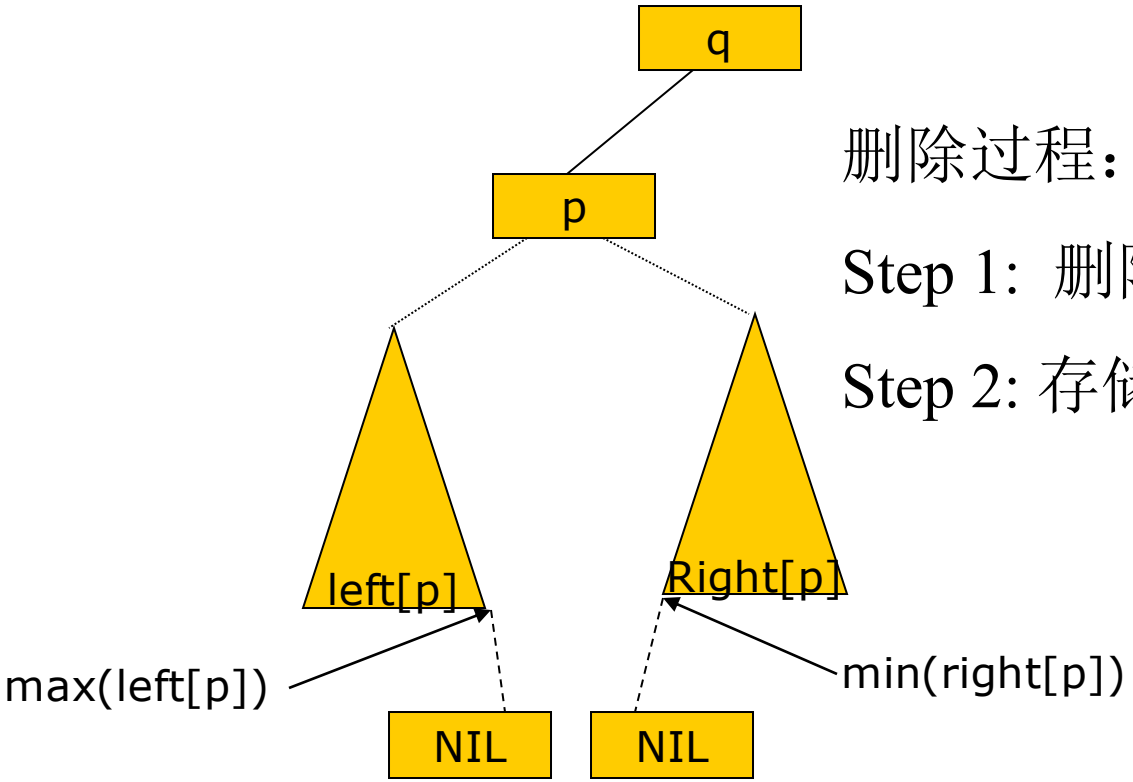
根据二叉检索树的性质，此时要寻找能代替此结点的结点必须是：比 p 左子树中的所有结点都大，比p的右子树的所有结点都小（或不大于）。

两个选择：

- 左子树中最大者
- **右子树中最小者**

而这二者都至多只有一个子结点

删除示例3



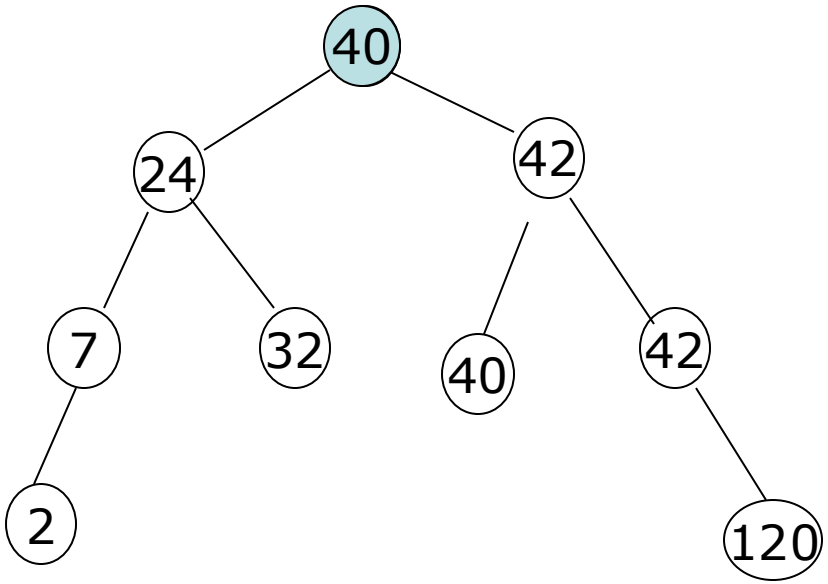
删除过程：

Step 1: 删除min(right[p])

Step 2: 存储min(right[p])到p中

删除示例3

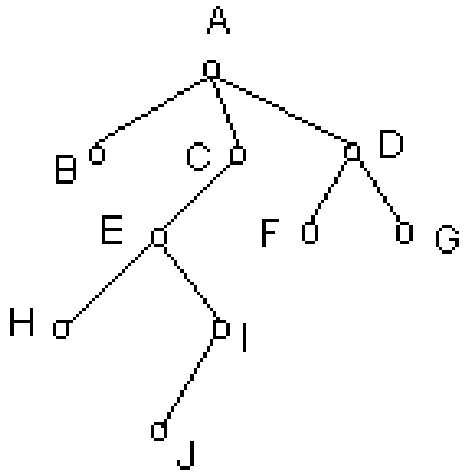
- 删除37



第九讲 树和森林

树的定义

- 树的递归定义如下：
  - (1) 树是包含有限结点的非空集合，其中有且仅有一个根结点
  - (2) 仅包含一个结点的集合是一棵树，根就是该结点
  - (3) 在树中，根以外的其他结点，被分成若干个不相交的子集，每个子集都是一棵树，称为子树 (subtree)
- 特别地，允许不包括任何结点的树，把它称作空树。
  - 右图所示树的三棵子树分别为
  - T_1={B}，T_2={C，E，H，I，J}，T_3={D，F，G}

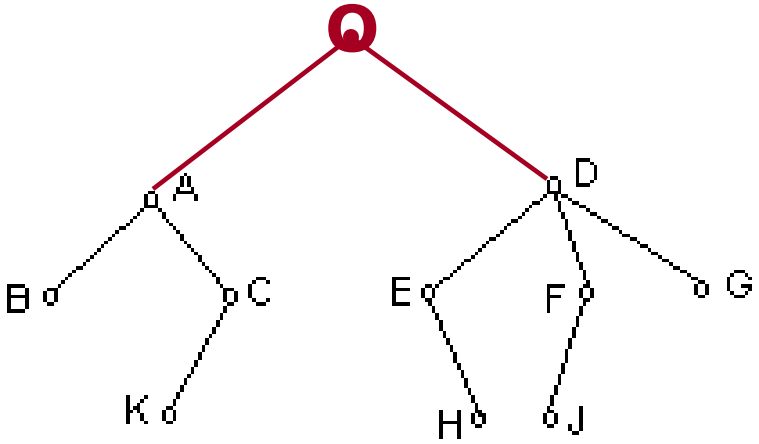


树 & 二叉树

- 树中的边、高度、深度、度数、父子结点、层次等概念，和二叉树中的类似。
- 有序树与无序树
  - 树通常是有序的，即有第1子树、第2子树、……、第n子树之分
  - 如果删除第1子树，第2子树就顶替成为第1子树
  - 如果规定树的各个子树之间不存在顺序，则称该树是无序树
- 注意：二叉树并不是树
  - 不能将二叉树看作度数为2的树
  - 二叉树区分左子树和右子树。
  - 只有一棵子树的树，无法区分这棵子树是左子树或者右子树

森林的定义

- 森林是空集或者不相交的树的集合
- 森林也通常是有序的，即有第1棵树、第2棵树、第3棵树之分。
  - 森林中每棵树的根彼此称为“兄弟”；
  - 如果将两棵树中将根结点连接到一个父节点，便得到一个树。



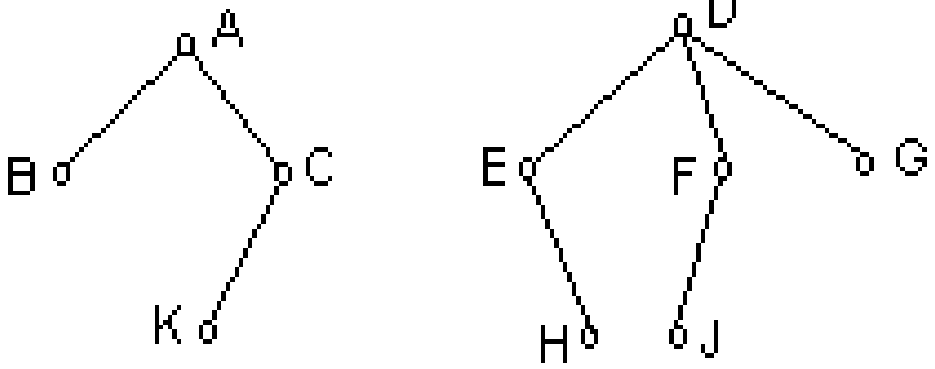
树和森林的存储表示

- 选择存储表示方法原则：结点本身+结点之间的关系

- 树和森林的存储表示（三种常见的结构）
  - 父指针表示法
  - 子表表示法
  - 长子-兄弟表示法（常用）

父指针表示法

- 树或森林组织成一个结点顺序表，其中每一结点包含父节点的下标。
  - 根节点不具有父节点，parent值设为-1
- 用这种方法表示的树是无序树，无法区分不同子树之间的顺序



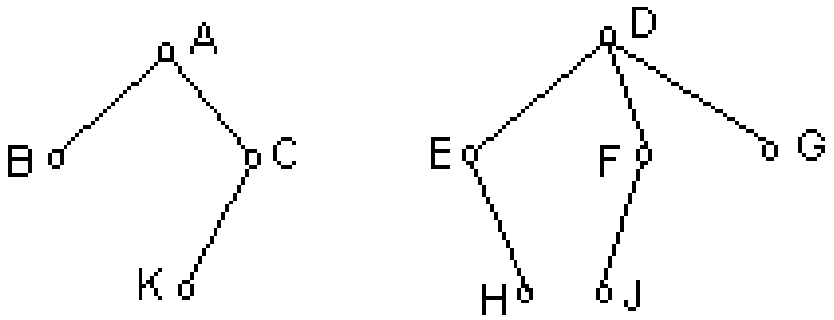
	info	parent
0	A	-1
1	B	0
2	C	0
3	K	2
4	D	-1
5	E	4
6	H	5
7	F	4
8	J	7
9	G	4

父指针表示法

- 如果两个结点到达同一根结点，它们一定在同一棵树中。
- 如果找到的根结点不同，那么两个结点就不在同一棵树中
- 优点
  - 容易找到父结点及其所有的祖先
  - 比较节省存储空间
- 缺点
  - 没有表示出结点之间的次序
  - 找结点的子女和兄弟比较费事（遍查整个数组）

子表表示法

- 整棵树或森林组织成一个结点顺序表，其中每一结点使用数组或链表记录该结点的所有子节点



nodelist

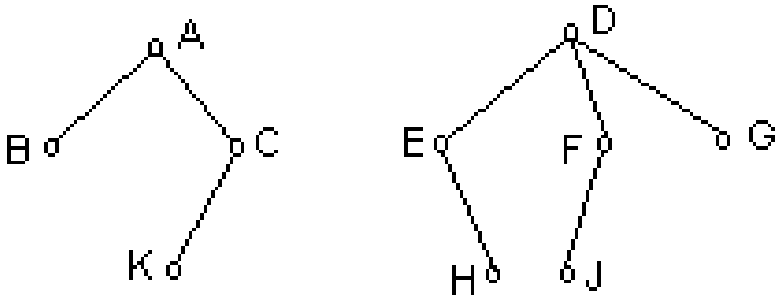
	info	children
0	A	1 → 2
1	B	∧
2	C	3
3	K	∧
4	D	5 → 7 → 9
5	E	6
6	H	∧
7	F	8
8	J	∧
9	G	∧

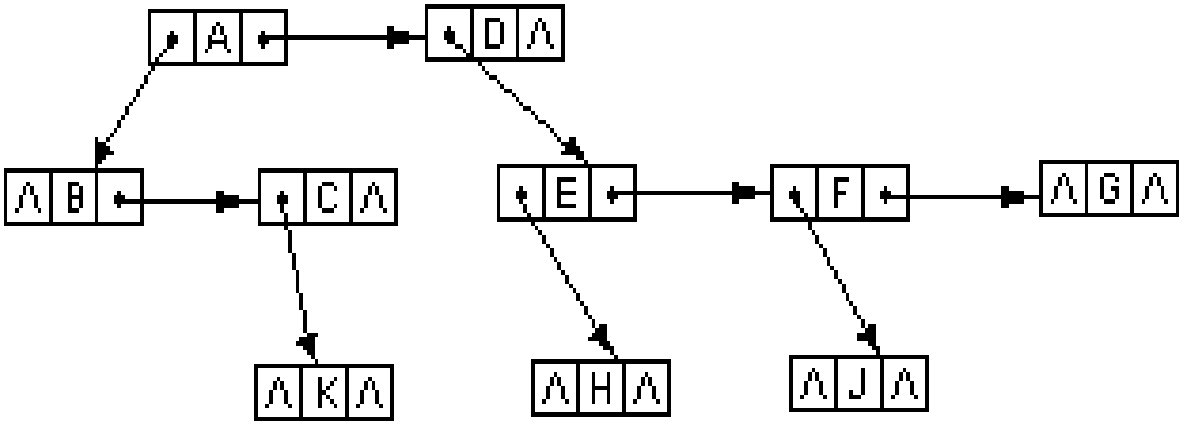
子表表示法

- 子表表示法的优点：
  - 方便找到父节点的所有子节点
- 缺点：
  - 由子节点找到父节点更困难
- 如果要将若干个子树合并成一个新树：
  - 子表表示法下，要考虑调整合并后树的子表
  - 父指针表示法下，只需调整子树的父指针即可

长子-兄弟表示法

- 每个结点存储：结点的值、最左子结点（长子）指针、右侧兄弟结点指针





长子-兄弟表示法

- 该表示法下，每个结点既包括了子结点（兄弟）信息，也包括了同层次结点（兄弟）的信息
- 比子表表示法空间效率更高
- 长子-兄弟表示法也称左子右兄表示法

树的遍历

- 定义：按某一规律系统地访问树的所有结点，并使每个结点恰好被访问一次。（又称为遍历）
- 遍历的结果：
  - 在遍历树的过程中，如果将各个结点按其被访问的先后顺序排列起来，则可得到一个包括所有结点的线性表

本质：将非线性结构转换为线性结构。

遍历方法

- 与二叉树的遍历类似，树的遍历方法同样包括：
  - 深度优先遍历[纵向遍历]
    - 前序遍历
    - 中序遍历
    - 后序遍历
  - 广度优先遍历[横向遍历]
    - 按层次顺序，逐层访问所有节点

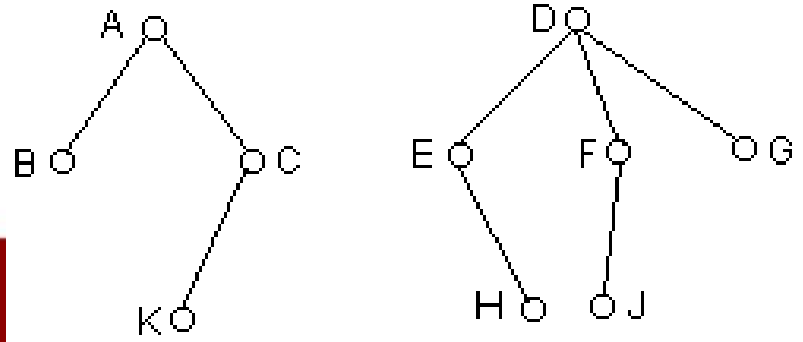
广度优先遍历

- 广度优先遍历：先访问层数为0的结点，然后从左到右逐个访问层数为1的结点，依此类推，直到访问完树中的全部结点。
- 按广度优先遍历所得到的线性表叫作树的层次序列
- 特点：
  - 在层次序列中，层数较低的结点总是排在层数较高的结点之前
  - 同层结点的左右次序还保持着，非同层结点的左右次序已被破坏

森林的遍历

- 森林的遍历方法有两种：前序遍历和后序遍历
- 前序遍历
  - 访问森林中第一棵树的根结点；
  - 前序遍历第一棵树的所有子树；
  - 前序遍历除去第一棵树之后的子森林。

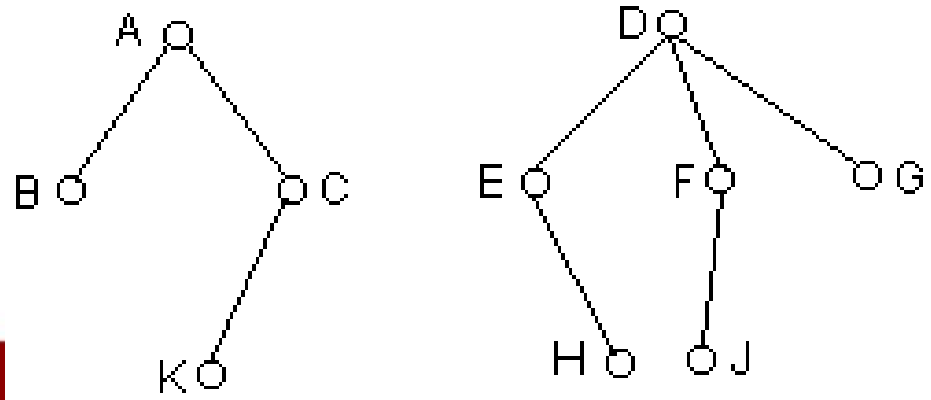
- 前序遍历序列：（A, B, C, K, D, E, H, F, J, G)



森林的遍历

- 后序遍历
  - 后序遍历第一棵树的所有子树；
  - 访问森林中第一棵树的根结点；
  - 后序遍历除去第一棵树之后的子森林。

- 后序遍历序列：(B, K, C, A, H, E, J, F, G, D)



树、森林与二叉树的转换

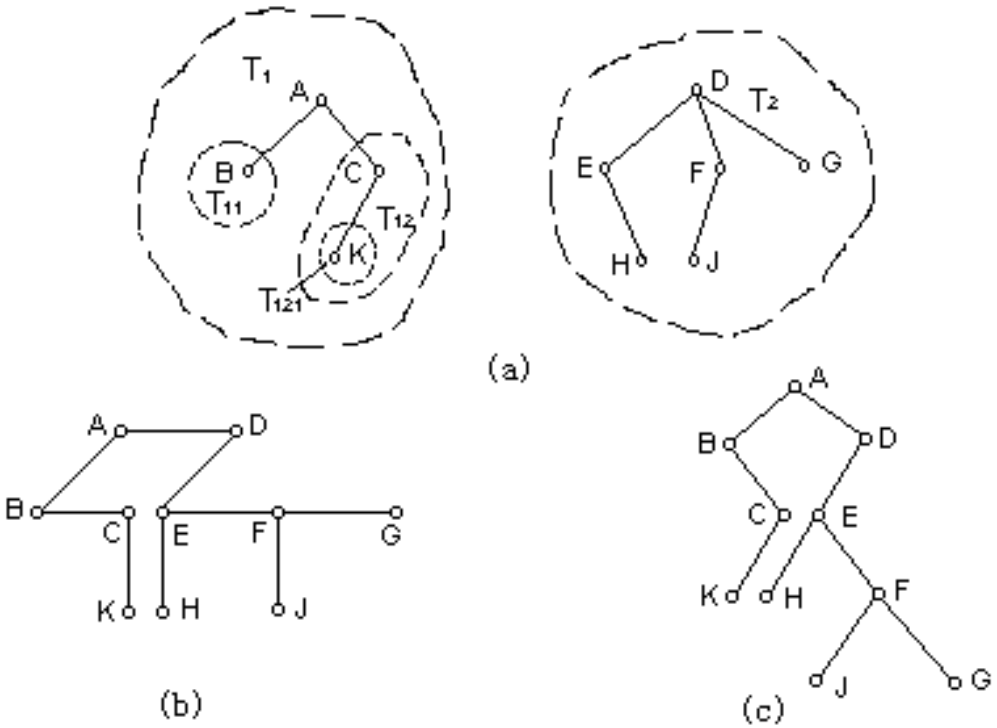
- 森林与二叉树一一对应
  - 任何森林都唯一地对应到一棵二叉树；
  - 反过来，任何二叉树也都唯一地对应到一个森林
- 森林对应的二叉树中：
  - 一个结点的左子节点，是原来森林中的长子
  - 右子节点是原来森林中的下一个兄弟
  - 即：左孩子、右兄弟

树、森林与二叉树的转换

- 森林($F=T_1, T_2, \ldots\ldots, T_n$)对应的二叉树B(F)：
- 若$n=0$，则B(F)为空；
- 若$n>0$，则：
  - B(F)的根是T_1的根
  - B(F)的左子树是$B(T_{11}, T_{12}, \ldots\ldots, T_{1m})$，其中$T_{11}, T_{12}, \ldots, T_{1m}$是$T_1$的子树；
  - B(F)的右子树是$B(T_2, \ldots\ldots, T_n)$
- 树对应的二叉树中，根节点的右子节点总是空的。
- 森林对应的二叉树中，根节点的右子节点是第2棵树的根

树、森林与二叉树的转换

- 加线：在树中所有相邻的兄弟之间加一连线
- 抹线：对树中每个结点，除其最左孩子外，抹去该结点与其余孩子间的连线
- 整理：以树的根结点为轴心，将整树顺时针转45°

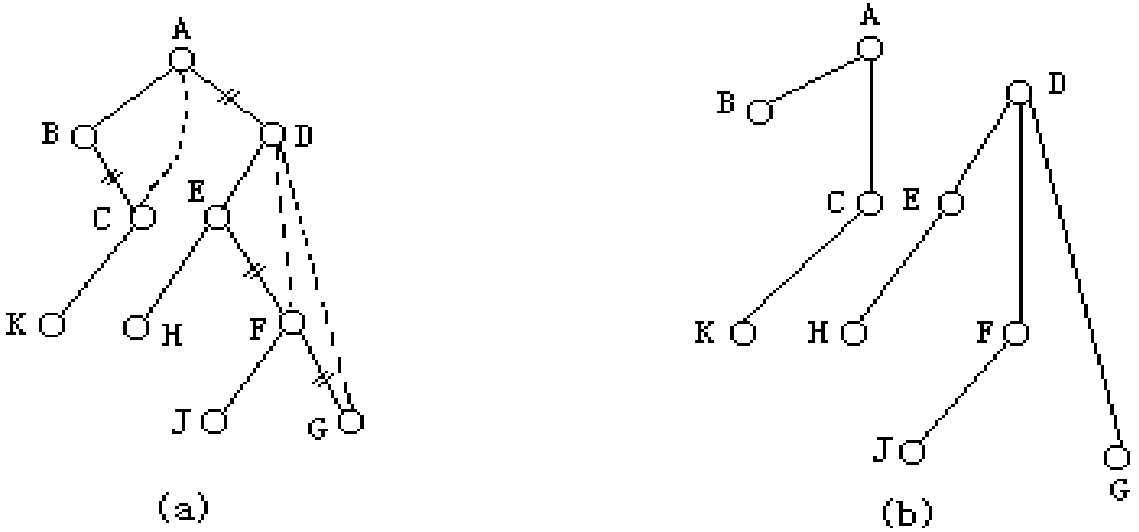


二叉树转换为树、森林

- 设B是一棵二叉树，r为根，L为左子树，R为右子树，则对应于B的森林F(B)的定义是：
  - 若B为空，则F(B)是空的森林；
  - 若B不为空，则F(B)是一棵树T_1加上森林F(R)，其中树T_1的根为r，r的子树为F(L)

二叉树转换为树、森林

- 方法
  - **若某结点是其父节点的左子节点**，则把该结点的右子节点，右子节点的右子节点……，都与该结点的父节点连起来，最后删除所有的父节点到右子节点的连线



并查集

- 并查集是用父结点表示法表示的树构成的森林
- 主要用于解决的问题：一些元素分布在若干个互不相交的集合(等价类)中，需要多次进行以下操作：
  - (1) 合并(Union) a, b两个元素所在的集合
  - (2) 查询(Find) 一个元素在哪个集合
  - (3) is_connected (a,b):判断两个元素a和b是否属于同一个集合。
- 例如：
  - 在社交网络中，判断两个人是否属于同一个社交圈。
  - 在图论中，判断两个节点是否连通。
  - 在一些算法竞赛中，处理动态连通性问题。

并查集

- 集合的表示
  - 并查集通过森林来表示多个集合，每个集合是一个树结构。每个节点存储其父节点的指针，根节点的父节点指向自身。
  - 如果两个节点的根节点相同，则它们属于同一个集合。
- 关键操作
  - 查找操作（Find）：通过不断向上查找父节点，直到找到根节点。
  - 合并操作（Union）：将一个集合的根节点连接到另一个集合的根节点上，作为另一个集合的根节点的子节点。

•简单练习（七）

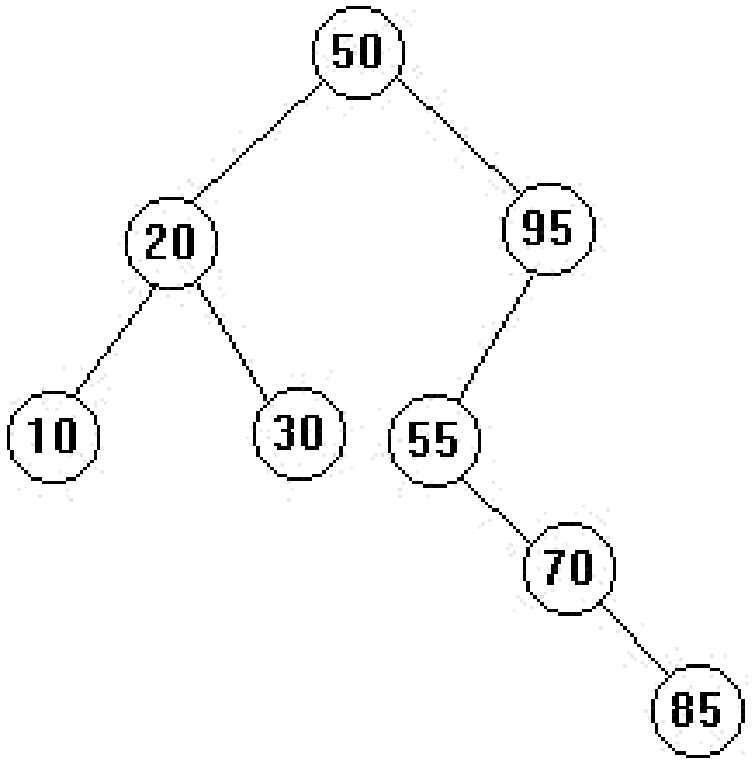
选择题

- 有一棵二叉树，如图所示该二叉树是(　)．

A. 二叉平衡树

B. 二叉排序树

C. 堆的形状



B

选择题

- 对线性表进行二分查找时,要求线性表必须(　　)

  A.以顺序方式存储

  B.以链接方式存储

  C.以顺序方式存储,且结点按关键字有序排序

  D.以链接方式存储,且结点按关键字有序排序

C

选择题

- 当有序表为

  {1，3，9，12，32，41，45，62，75，77，82，95，100}，

  则二分查找值为82的结点时，(　　)次比较后成功。

  A. 1

  B. 2

  C. 4

  D. 8

C

判断题

- 二叉树的先根遍历序列中，任意一个结点均处在其子女结点的前面。
- 由树转换成二叉树，其根结点的右子树总是空的。
- 哈夫曼树是带权路径长度最短的树，路径上权值较大的结点离根较近。

Y Y Y

选择题

- 已知某二叉树的后根周游序列是dabec，中根周游是debac，它的先根周游序列是（　）。

  A acbed

  B decab

  C deabc

  D cedba

D

选择题

- 任何一棵二叉树的叶结点在先根、中根和后根周游序列中的相对次序（）

  A 不发生改变

  B 发生改变

  C 不能确定

  D 以上都不对

A

选择题

- 某二叉树的先根周游序列和后根周游序列正好相反，则该二叉树一定是（　　）二叉树。

A. 空或只有一结点

B. 树的高度等于其结点数减1

C. 任一结点都只有右子结点

D. 任一结点都只有左子结点

B

选择题

- 设高度为h的二叉树上只有度为0和度为2的结点，则此类二叉树中所包含的结点数至少为(　　)。

  A. 2h

  B. 2h-1

  C. 2h +1

  D. h+1

C

解答题

- 对于表达式(a-b)*d/(e+f)，
  - 请画出它的中序二叉树，
  - 给出该表达式的前缀表达式形式和后缀表达式形式

第十讲 字典和检索

检索问题

- 检索：在一个数据结构中查找关键码值等于给定值的元素。
  - 数据结构中的元素可能包含不止一个属性，检索过程只需要针对其中的个别属性，称为检索的关键码
  - 检索也称为查找，在这一章二者是同义的
- 检索的结果：
  - 如果找到，则检索成功；
  - 否则应该报告检索失败，即数据结构中不存在符合要求的元素

检索问题

- 不同种类的检索
  - **精确匹配查询**（exact-matching query）：在数据结构中查找关键码值与查询值相等的所有元素。
  - **范围查询**（range query）：在数据结构中查找关键码值属于某个指定范围内的所有元素。
- 在本章中，假定关键码都为正整数，各数据记录类型相同，因此各元素可以按照关键码排序。

检索与字典

- 字典(Dictionary)是元素的有穷集合，其主要的操作为对元素的检索。
- 字典中的每个元素由两部分组成，分别称为元素的关键码(key)和属性 (attribute)。
  - 关键码本质上是一个特殊的属性
  - 必须保证字典中的每个元素具有唯一的关键码
    - 比如：学号就是每个同学的关键码，身份证号就是每个公民的关键码。
    - 可以通过关键码来查询元素的其他属性
    - 如果有两个元素关键码相同，我们就无法区分这两个元素

检索与字典

- 静态(static)字典：字典一经建立就基本固定不变，主要的操作就是字典元素的检索。
  - 为静态字典选择存储方法主要考虑检索效率、检索运算的简单性
- 在实际应用中，有时也要考虑字典的插入和删除操作
- 动态(dynamic)字典：经常需要改动的字典。
  - 对于动态字典，存储方法的选择不仅要考虑检索效率，还要考虑字典元素的插入、删除运算是否简便。

顺序检索

- 基本思想：
  - 顺序检索是基于线性表的检索方法
  - 从线性表的一端开始顺序扫描，将元素的关键码和给定值比较，如果相等，则检索成功；
  - 当扫描结束时，还未找到关键码等于给定值的元素，则检索失败。

顺序检索

- 平均检索长度ASL：
  - 若找到的是第i个元素，则比较次数为$c_i=i$。因此

$$ASL = 1 \times P_1 + 2 \times P_2 + \ldots + n \times P_n$$

  - 假设每个元素的检索概率相等，即$P_i=1/n$，则平均检索长度为：

$$ASL = \sum_{i=1}^{n} p_i c_i = \sum_{i=1}^{n} i/n = (n+1)/2$$

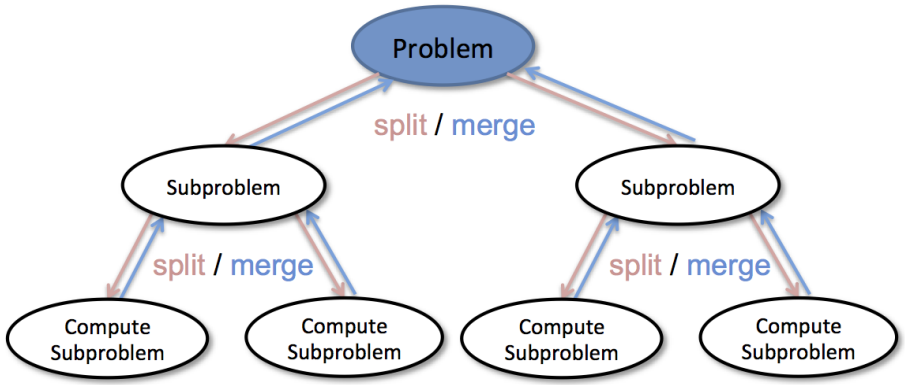
  - 因此，成功检索的平均比较次数约为字典长度的一半；若字典中不存在关键码为key的元素，则需进行n次比较。
    - 总之，顺序检索的平均检索长度为ASL=O(n)

二分查找（二分检索）

- 基本思想：
  - 要求线性表已经按照关键码顺序排序。
  - 将字典中间位置上元素的关键码 key’ 和给定值 key 比较，若
    - key’==key，则检索成功；
    - key’ > key，在字典前半部分中继续进行二分法检索；
    - key’ < key，在字典后半部分中继续进行二分法检索。
  - 二分检索的实质是逐步缩小查找区间。

二分查找（二分检索）

- 二分查找算法实际上体现了解决问题的一个典型策略：分而治之（Divide and Conquer）
  - 将问题分为若干更小规模的部分；通过解决每一个小规模部分问题，并将结果汇总得到原问题的解
  - 二分查找法也适合用递归方式实现




```
def binarySearch(alist, item):
    if len(alist) == 0:
        return False
    else:
        midpoint = len(alist)//2
        if alist[midpoint]==item:
          return True
        else:
          if item<alist[midpoint]:
            return binarySearch(alist[:midpoint],item)
          else:
            return binarySearch(alist[midpoint+1:],item)
```

分块查找

- 基本思想：“按块有序”，即在块的级别上有序
  - 设线性表中共有 n 个数据元素，将表分成 b 块
  - 额外设置一个索引表，记录每一块的最小或最大的关键码
  - 前一块最大关键码必须小于后一块最小关键码
  - 每一块中的关键码不一定有序
- 顺序与二分法的折衷
  - 先在索引表中二分查找所在的块，然后在块中顺序查找目标元素
  - 既有较快的检索，又有较灵活的更改

散列检索

- 散列法(hashing)的基本思想：
  - 直接根据记录的内容得到其存储位置
  - 插入关键码为key的字典元素时，按一个确定的散列函数 h 计算哈希值 h(key)，并把 h(key) 作为该元素的存储地址，即散列地址。
  - 检索时，同样根据 h(key) 得到关键码所在元素的存储地址。
- 散列法也称哈希法、杂凑法。
- 散列表/哈希表：用散列法表示的字典。
  - 散列表中的每一个存储位置，称为槽（slot），可以用来保存数据项。

散列检索

- 散列函数示例：
  - 有如下关键码的数据项： 54，26，93，17，77，31
  - 最常见的散列函数之一是“求余数”，将数据项除以散列表的大小，得到的余数作为槽号。
    - h(item)= item % 11
  - 例如，如果需要查找关键码26，计算出其哈希值4，直接访问下标为4的元素即可。

Item	Hash Value
54	10
26	4
93	5
17	6
77	0
31	9

散列地址	0	1	2	3	4	5	6	7	8	9	10	11
关键码	77				26	93	17			31	54	

散列检索

- 散列表的构造和检索中，需要解决的主要问题：

  1. 定义散列表和散列函数
     - 散列函数：散列表如何检索？
     - 检索效率如何？检索效率与哪些因素有关？
  2. 解决碰撞
     - 对于任意的散列函数，都可能出现“碰撞”现象
     - 碰撞发生时如何处理？

散列函数

- 散列函数的选择标准
  - 散列函数应该将关键字均匀映射到到整个地址空间中，从而尽可能减少碰撞。即落在任意一个槽中的概率应该均等。
  - 散列函数本身的计算应该尽可能简单。
  - 散列函数应该使得任意关键字的哈希值都落在表长范围内。

散列函数

- 完美散列函数
  - 给定一组数据项，如果一个散列函数能把每个数据项映射到不同的槽中，那么这个散列函数就可以称为“完美散列函数”
  - 对于固定的一组数据，总是能想办法设计出完美散列函数
- 但如果数据项经常性的变动，很难有一个系统性的方法来设计对应的完美散列函数。

碰撞处理

- 无论选择何种散列函数，碰撞都可能发生。换句话讲，合适的散列函数可以减少碰撞发生的几率，但不能保证不发生碰撞。 碰撞发生时如何处理？
- 碰撞处理
  - 开地址法（探查法）：为冲突的数据项再找一个开放的空槽
  - 拉链法：将容纳单个数据项的槽扩展为容纳数据项集合（或者对数据项链表的引用）

碰撞处理：开地址法

- 开地址法基本思想：
  - d_0 = h(hey) 称为关键码key的基地址
  - 当碰撞发生时，用某种方法在基本区域内确定一个探查序列。

$$d_i = d_0 + p(key, i)$$

  - p称为探查函数，d_i称为后继散列地址
- 按照确定探查序列的方式分为：
  - 线性探查法
  - 双散列函数法

碰撞处理：开地址法

- 插入关键码key时，若发生了碰撞：
  - 则按照探查函数生成的探查序列一次查找
  - 将找到的第一个空闲位置di作为key的存储位置
  - 若后续散列地址都不空闲，说明散列表已满，报告溢出
- 检索关键码key时：
  - 依然首先计算基地址，以及相应的探查序列
  - 在探查序列中依次遍历查找关键码key
  - 如果遇到了空闲位置，表示探查序列结束，检索失败
- 插入和检索时都需要考虑表满的情况
  - 探查序列可能会进入一个无限循环中
  - 可以限制探查序列的长度

线性探查法

- 线性探查法（Linear Probing）
  - 若在地址为d (d=h(key)) 的单元发生碰撞：
  - 则探查序列为：d+1,d+2,…,m-1,0,1,…,d-1 (m为基本存储区的长度)
    - 相当于将基本存贮区看作一个循环表，进行顺序遍历。
  - 如果从单元d开始探查，查找一遍后又回到地址d，则表示基本存贮区已经溢出。

双散列函数法

- 双散列函数法选用两个散列函数h_1和h_2
  - 假设m为散列表长
  - h_1产生一个0到m-1之间的数作为地址。
  - h_2产生一个1到m-1之间的数作为探查序列的间隔
- 例如，$h_1(key)=key\%m$，$h_2(key) = key \% (m-1) + 1$。
  - 如果$d=h_1(key)$发生碰撞，则再计算$h_2(key)$，
  - 产生的探查序列为：$(d+h_2(key))\%m$，$(d+2*h_2(key))\%m$，…
  - 结果是，探查序列既依赖于哈希值，也依赖于关键码值
- h_2产生的间隔必须与m互素
  - 为了保证，如果表中存在空闲位置，探查序列总能遍历到它
  - 实际中，将 m 设置为素数就可以保证这一点

简单练习（八）

填空题

1、有一个散列表如下图所示，其散列函数为 h(key)=key mod 13，该散列表使用再散列函数

- H2(Key)=Key MOD 3 解决碰撞，问从表中检索出关键码 38 需进行几次比较（ B ）。

0	1	2	3	4	5	6	7	8	9	10	11	12
26	38			17			33		48			25

A：1　　B：2　　C: 3　　D：4

判断题

1. 二叉排序的查找和折半查找时间的性能相同
2. 散列法存储的基本思想是由关键码的值决定数据的存储地址
3. 任一二叉排序树的平均查找时间都小于用顺序查找同样结点的线形表的平均查找时间

N Y N

第十一讲 排序和基本排序算法

排序的基本概念

- 记录(Record)：进行排序的基本单位
- 关键码(Key)：唯一确定记录的一个或多个域
- 排序码(Sort Key)：记录中作为排序运算依据的一个或多个域
- 序列(Sequence)：线性表，由记录组成的集合
- 排序(Sorting) — 将序列中的记录按照排序码特定的顺序排列起来，即排序码域的值具有不减(或不增)的顺序

排序的基本概念

- 给定一个序列R =$\{r_1, r_2, \ldots, r_n\}$，其排序码分别为k =$\{k_1, k_2, \ldots, k_n\}$，排序的目的就是将R中的记录按照特定的顺序重新排列，形成一个新的有序序列R'= $\{r'_1, r'_2, \ldots, r'_n\}$
  - 相应排序码为k' =$\{k'_1, k'_2, \ldots, k'_n\}$
  - 其中$k'_1 \le k'_2 \le \ldots \le k'_n$或$k'_1 \ge k'_2 \ge \ldots \ge k'_n$，前者称不减序（也称非递减序），后者称不增序（也称非递增序）

排序的基本概念

- 正序与逆序
  - “正序”序列：待排序序列**正好符合排序要求**
  - “逆序”序列：把待排序序列逆转过来，正好符合排序要求
  - 譬如，需要得到一个不减序序列？

- **正序：**

11	19	23	55	80	97

- **逆序：**

97	80	55	23	19	11

排序的基本概念

- 排序的稳定性：

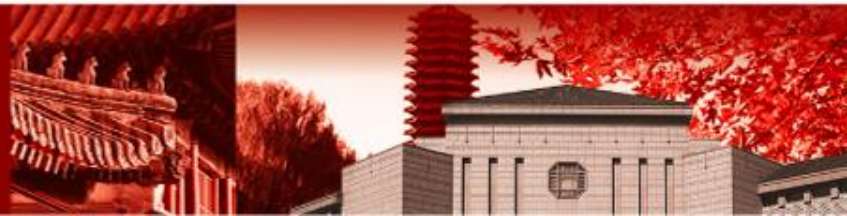
  在待排序的文件中，若存在多个排序码相同的记录，经过排序后记录的相对次序保持不变，则这种排序方法称为是“稳定的”；否则，是“不稳定的”

譬如，对下列数据进行不减序排序

23	19	55	97	19’	80

正确但不稳定的排序结果

19’	19	23	55	80	97

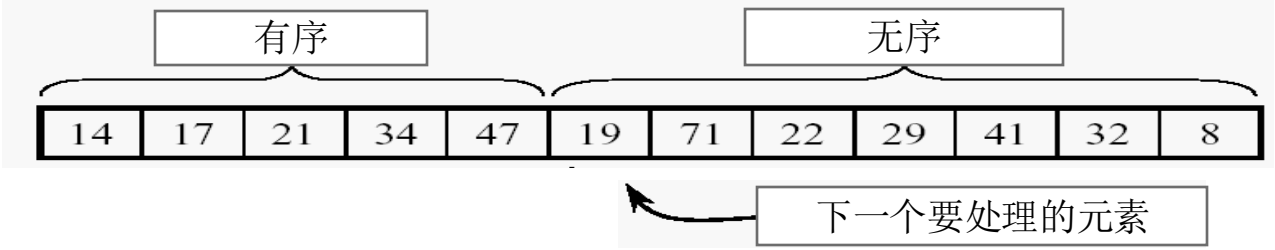


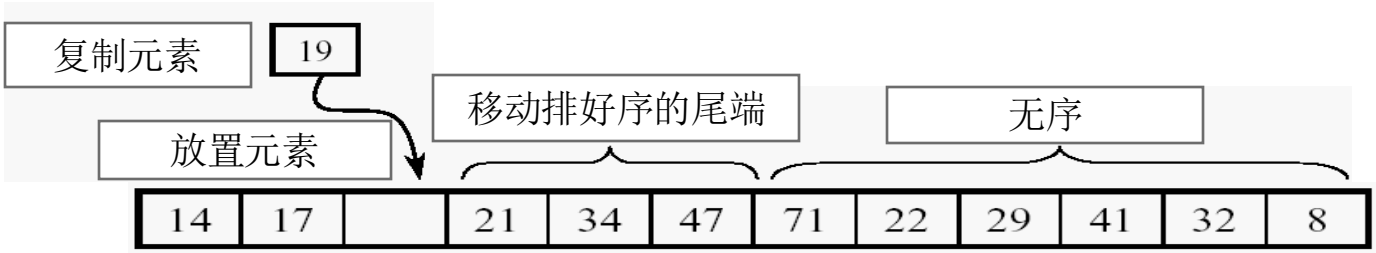
排序算法介绍

插入排序	1 直接插入排序 2 二分法插入排序 3 Shell排序
选择排序	4 直接选择排序 5 堆排序
交换排序	6 冒泡排序 7 快速排序
分配排序	8 基数排序
归并排序	9 二路归并排序

直接插入排序

- 基本思想：
  - 将待排序序列分为**已排序**和**未排序**两部分。初始时，已排序部分仅包含第一个元素，未排序部分包含剩余元素。
  - 逐个处理待排序的记录。每步将一个待排序的元素R_i按其排序码K_i大小插入到前面已排序表中的适当位置，直到全部插入完为止。





直接插入排序算法评价

- 算法是稳定的
- 空间代价：O(1)，交换操作需要一个辅助空间
- 时间代价
  - 最佳情况（正序）：n-1次比较，0次交换，O(n)复杂度
  - 最差情况（逆序）：比较和交换次数为$O(n^2)$

$$\sum_{i=1}^{n-1} i = n(n-1)/2$$

  - 平均情况：$O(n^2)$

实验表明：当记录数量n较小时，直接插入排序是一种高效的排序算法！

二分法插入排序

- 基本思想：

  在直接插入排序的基础上减少比较的次数，即在插入R_i时改用二分法找插入位置。

  – 插入记录R_i时，记录集合中子区间$\{R_0,R_1,\ldots,R_{i-1}\}$已经有序

  – low =0; high = i-1; mid= (low+high)/2 带入二分检索

  – 找出应该插入的位置；

  – 将原位置的记录向后顺移，将记录R_i插入。

- 二分插入排序采用顺序存储结构

二分插入排序分析

- 算法是稳定的
- 空间代价：O(1)，算法中有一个辅助空间
- 时间代价
  - 比较次数降为nlogn量级：插入每个记录需O(log *i*)次比较
  - 移动次数仍为n^2量级：每插入一个记录最多移动i+1次
  - 因此，最佳情况下总时间代价为O(nlog *n*) ，最差和平均情况下仍为$O(n^2)$

Shell排序

如何利用这两个性质？

1. 在初始无序时，进行等间隔的小序列分割

   - 先将待排序序列转化为若干小序列，在这些小序列内进行直接插入排序

2. 在整个序列趋向有序后，逐步扩大序列规模

   - 逐渐扩大小序列的规模，而减少小序列个数，使得待排序序列逐渐处于更有序的状态

3. 最后，对整个序列进行一次完整的插入排序

Shell排序实现方法

1) 选定一个间隔增量序列（$n>d_1>d_2>\ldots>d_t=1$）

   - n：文件长度，d_i：间隔增量，t：排序趟数

2) 将文件按 d_1 分组（彼此相距 d_1 的记录划为一组），在各组内采用直接插入法进行排序。

3) 分别按 $d_2, \ldots d_t$ 重复上述分组和排序工作。

- **Shell 最初提出的增量序列是**

$$d_1 = \lfloor n/2 \rfloor,\ d_{i+1} = \lfloor d_i/2 \rfloor$$

Shell排序算法性能分析

- Shell排序算法的速度比直接插入排序快，其时间复杂度分析比较复杂，Shell排序的平均比较次数和平均移动次数都为$n^{1.3}$左右

- Shell排序算法中增加了一个辅助空间，因此算法的辅助空间为S(n)=O(1)

- Shell排序是不稳定的

插入排序小结

- 直接插入排序思想最简单，容易实现，但时间效率低。
- 其它插入排序方法都是从减少比较次数、移动次数出发对直接插入排序进行改进。
  - 直接：顺序检索确定位置，记录移动实现插入；
  - 折半：减少比较次数（折半检索确定位置）；
  - Shell排序：改变增量（来自：当待排序序列基本有序，并且n较小时，提高了直接插入排序效率）
- 除Shell外，其它插入排序算法的时间复杂度为$O(n^2)$，并且是稳定的。

选择排序

每趟从待排序的记录序列中选择关键字最小/大的记录放置到已排序表的最前位置，直到全部排完。

1. 直接选择排序
2. 堆排序

直接选择排序

- 基本思想
  - 每一趟在后面n-i个待排记录中选取最小记录和第i个记录互换。
- 具体过程
  - 首先，在n个记录中选择最小者与r[0]互换；
  - 然后，从剩余的n-1个记录中选择最小者与r[1]互换；
  - …如此下去，直到全部有序为止。
- 优点：实现简单
- 缺点：每趟只能确定一个元素，表长为n时需要n-1趟

直接选择排序性能分析

- 直接选择排序的比较次数与记录的初始状态无关。
  - 第i趟排序：从第i个记录开始，顺序比较选择最小关键码记录需要n-i次比较。
  - 总的比较次数：$\sum_{i=1}^{n-1}(n-i)=\frac{n(n-1)}{2}$

- 时间复杂度：$T(n)=O(n^2)$，
- 辅助空间1个记录单位： $S(n)=O(1)$
- 稳定性：不稳定的排序。

堆的定义

1、堆的定义

当n个排序码序列K=$\{k_0,k_1,k_2,\ldots,k_{n-1}\}$，当且仅当满足如下条件时，称之为堆

$$k_i \le k_{2i+1} \quad k_i \ge k_{2i+1}$$

或

$$k_i \le k_{2i+2} \quad k_i \ge k_{2i+2} \quad (i=0,1,2,\ldots,\lfloor n/2 \rfloor -1)$$

如果堆中根结点的排序码最小，则称为最小堆

如果堆中根结点的排序码最大，则称为最大堆

2、堆的完全二叉树表示

堆可以用一棵完全二叉树表示，则 K_{2i+1} 是 K_i 的左孩子；K_{2i+2} 是 K_i 的右孩子。

堆排序的基本思想

1. 对所有记录建立最大堆（O(n)）
2. 取出堆顶的最大记录移到数组末端，放在下标n-1的位置；重新将剩下的n-1个记录建堆（O(logn)），再取新堆顶最大的记录，放到数组第n-2位；…；不断重复这一操作，直到堆为空。
3. 这时数组正好是按从小到大排序

时间效率评价

- 分析
  - 建初始堆比较次数C_1：O(n)
  - 重新建堆比较次数C_2：O(nlog_2n)
  - 总比较次数=C_1+C_2
  - 移动次数小于比较次数，因此，

- 堆排序算法的时间复杂性为：O（n$\log_2$n）
- 堆排序算法的空间复杂性为：O（1）
- 堆排序算法是不稳定的。
- 适用于n值较大的情况。

交换排序

两两比较待排序记录的排序码，交换不满足顺序要求的偶对，直到全部满足为止。

1. 冒泡排序
2. 快速排序

冒泡排序

- 基本思想
  - 不停地比较相邻的记录，如果不满足排序要求，就交换相邻记录，直到所有的记录都已经排好序
  - 原理
    - 若序列中有 n 个元素，通常进行 n - 1 趟。
      - 第1趟，针对第r[0]至r[n-1]个元素进行。
      - 第2趟，针对第 r[0]至r[n-2] 个元素进行。……
      - 第n-1趟，针对第r[0]至r[1]个元素进行。
    - 第i趟进行的过程：
      - 针对r[0]至r[n-i] 元素进行，比较两个相邻的元素。若相邻的元素的相对位置不正确，则进行交换；否则继续比较下面两个相邻的元素。

冒泡排序算法分析

- 算法是稳定的
- 空间代价：O(1)的临时空间
- 时间代价

表1 冒泡排序中每一轮的比较次数

轮次	比较次数
1	n-1
2	n-2
。。。	。。。
n-1	1

– 总的比较次数就是前n-1个整数之和，即$1/2*n^2+1/2*n-n$
– 交换次数最多为$O(n^2)$，最少为0，平均为$O(n^2)$。
– 最大，最小，平均时间代价均为$O(n^2)$。

快速排序是对冒泡排序的改进

- **冒泡排序**

  在相邻两个记录间比较和交换，每次交换只能上移或下移一个位置，导致总的比较与移动次数增多

- **快速排序**又称**分区交换排序**

  设待排序的n个记录{$R_0, R_1, \ldots R_{n-1}$}，选取第一个记录R_0为划分基准，寻找R_0的最终位置（一趟快速排序）：

  - **{R[0], R[1], …,R[i-1]} 存放的为小于R_0的记录**
  - **{R[i+1], R[i+2],…,R[n-1]}存放的为大于R_0的记录**
  - **R[i] 为R_0的最终位置**

一趟快速排序

- 设置变量i = 0，变量j = n-1 ；
- 保存记录temp=R_0， R_0为空出的位置（空位在前一区）；
- 令j向前扫描，寻找小于R_0的记录，找到小于R_0的记录R[j]，将记录R[j]移到当前空位中。这时R[j]为新空位（空位在后一区）；
- i自i+1起向后扫描，寻找大于R_0的记录，找到大于R_0的记录R[i]，将记录R[i]移到当前空位中，空位又到了前一区；
- 如此交替改变扫描方向，从两端向中间靠拢，直到i=j，这时i所指的位置为R_0的最终位置

快速排序算法性能分析

- 快速排序的平均时间复杂度是$T(n)=O(n\log_2 n)$
- 算法需要一个栈空间实现递归
  - 栈的大小取决于递归调用的深度，最多不超过n，若每次都选较大的部分进栈，处理较短的部分，则递归深度最多不超过$\log_2 n$，所以快速排序的辅助空间为$S(n)=O(\log_2 n)$

- 快速排序算法是不稳定的

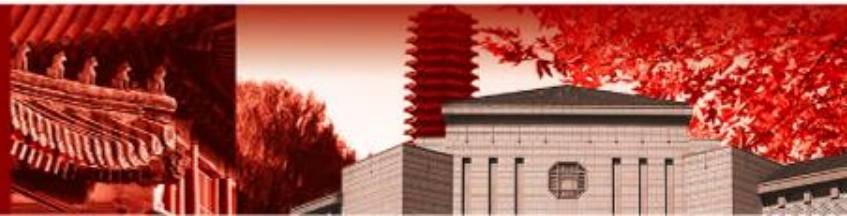
分配排序（基数排序）

- 把排序码分解成若干部分，然后通过对各部分排序码的分别排序，最终达到整个排序码的排序。

分配排序

- 分配排序：实现多关键码排序的方法
  - 高位优先法
    - 将文件逐层分割成若干子文件，各子文件独立排序；
  - 低位优先法
    - 对每个关键码都是所有记录参加排序，可通过若干次“分配”和“收集”实现排序。

基数排序



基数排序

- 把每个排序码看成是一个d元组：$K_i=(K_i^0, K_i^1,\ldots, K_i^{d-1})$

  其中每个K_i都是集合$\{C_0，C_1，\ldots，C_{r-1}\}$中的值

  即$C_0\le K_i^j\le C_{r-1}(0\le i\le n-1,0\le j\le d-1)$， 其中r称为基数。

- 基数排序的基本思想
  - 排序时先按K_i^{d-1}从小到大将记录分配到r个堆中；
  - 然后依次收集，再按K_i^{d-2}分配到r个堆中…
  - 如此反复，直到对K_i^0分配、收集，便得到最终排序序列

基数排序算法性能分析

- 基数排序的时间复杂度T(n)=O(d*(r+n))
  - 每趟排序中，将n个记录分配到队列的时间为O(n)，收集的时间为O(r)，因此一趟排序的时间为O(r+n)；
  - 共要进行d趟排序

- 当n较大、d较小，特别是记录的信息量较大时，基数排序非常有效。

- 基数排序的空间复杂度S(n)=O(n+r)
- 基数排序是稳定的

归并排序

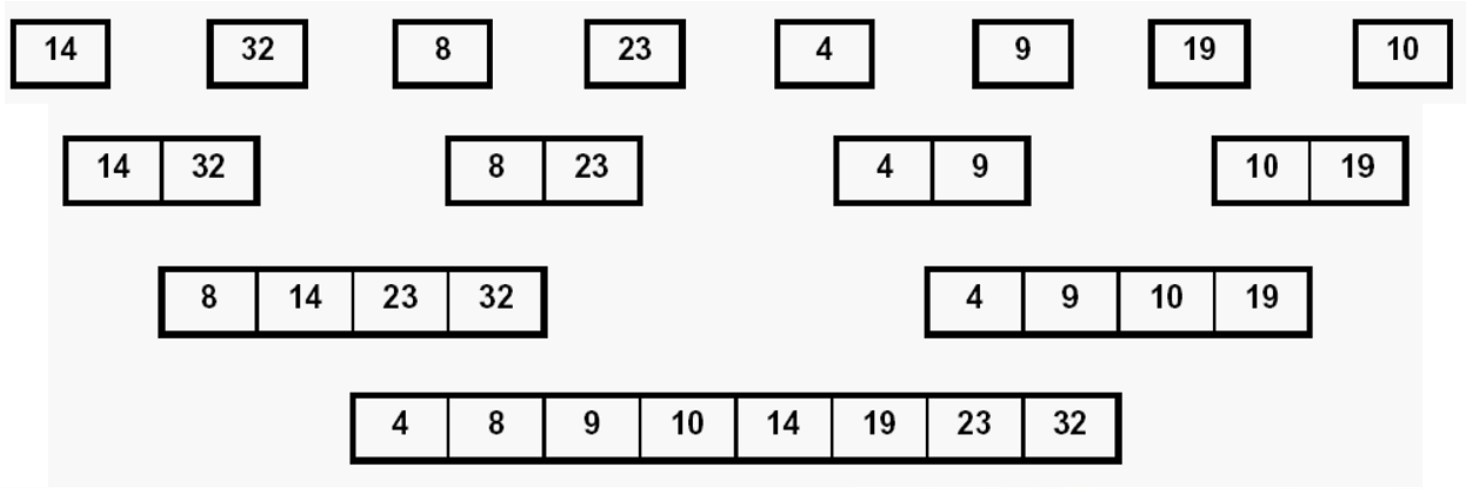
- 把待排序的文件分成若干子文件，每个子文件内排序；再将已排序的子文件合并，得到完全排序的文件。

二路归并排序

① 设文件中有n个记录可以看成n个子文件，每个文件只有一个记录；

② 将每两个子文件归并，得到n/2个、每个包含2个记录的子文件；

③ 将n/2个子文件归并，如此反复，直到得到一个文件。



算法评价

- 二路归并排序算法的时间复杂度为$T(n)=O(n\log_2 n)$
  - 第i次归并以后，有序记录的长度为2^i。因此n个记录的文件排序，必须做$\lceil \log_2 n \rceil$趟归并
  - 每一趟归并所花费的时间是O(n)

- 二路归并排序算法的辅助空间为$S(n)=O(n)$
  - 算法中增加了一个数组**result[ ]**

- 二路归并排序是稳定的

各种排序算法的比较

分类	算法	最大时间	平均时间	最小时间	空间代价	稳定性
插入排序	直接插入	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
	二分插入	$O(n^2)$	$O(n^2)$	$O(n\log n)$	$O(1)$	稳定
	Shell排序	$O(n^{3/2})$	$O(n^{3/2})$	$O(n^{3/2})$	$O(1)$	不稳定
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$	不稳定

各种排序算法的比较

分类	算法	最大时间	平均时间	最小时间	空间代价	稳定性
交换排序	冒泡排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	稳定
	快速排序	$O(n^2)$	O(nlog n)	O(nlog n)	O(log n)	不稳定
分配排序	桶式排序	O(n+m)	O(n+m)	O(n+m)	O(m)	稳定
	基数排序	O(d·(n+r))	O(d·(n+r))	O(d·(n+r))	O(n+r)	稳定
归并排序	二路归并	O(nlog n)	O(nlog n)	O(nlog n)	O(n)	稳定

算法评价

- 各种排序方法各有优缺点，可适用于不同的场合，应当根据具体情况，选择合适的排序算法。
- 从数据的规模n来看，
  - 当数据规模n较小时，n^2和$n\log_2 n$的差别不大，则采用简单的排序方法比较合适
    - 如直接插入排序或直接选择排序等
  - 当数据规模n较大时，应选用速度快的排序算法
    - Shell排序、堆排序、快速排序及归并排序的排序速度较快

算法评价(续)

- 从算法结构的简单性看，
  - 速度慢的排序算法比较简单、直接
  - Shell排序法，快速排序法、堆排序法及归并排序法可以看作是对某一种排序方法的改进，算法结构一般都比较复杂

算法评价(续)

- 从文件的初态来看，
  - 当文件的初态已基本有序时，可选择简单的排序方法，如直接插入排序或冒泡排序等
  - 快速排序有可能出现最坏情况，时间复杂度为$O(n^2)$，且此时递归深度为n，即所需栈空间为O(n)
  - 堆排序不会出现象快速排序那样的最坏情况，且堆排序所需的辅助空间比快速排序少
    - 但这两种算法都是不稳定的，如果要求排序是稳定的，则可以选择归并排序方法

算法评价(续)

- 基数排序法所需的辅助空间较大，
  - 其时间复杂度可简化成O(dn)；当排序码的位数d较少时，可进一步简化成O(n)，能达到较快的速度。
  - 但是基数排序只适用于像字符串和整数这类有明显结构特征的排序码，当排序码的取值范围为某个无穷集合时，则无法使用
  - 因此，当n较大，记录的排序码位数较少且可以分解时，采用基数排序方法较好

- 归并排序法可以用于内排序，也可用于外排序。

•简单练习（九）

判断题

1. 如果某种算法是不稳定的,则该方法没有实际应用价值。
2. 对于n个记录的集合进行冒泡排序，所需要的平均时间是O(n）。
3. 对于n个记录的集合进行快速排序，在最坏情况下所需要的平均时间是$O(n^2)$。
4. 对n个元素的序列进行起泡排序时最小的比较次数是n-1。

N N Y Y

选择题

- 对n个不同的排序码的元素进行不减序冒泡排序，在（A）情况下比较的次数最多，其比较次数为（ B ）。

A: 1. 从大到小排列好的； 2. 从小到大排列好的
3. 元素无序； 4. 元素基本有序

B: 1. n+1 ； 2. n
3. n-1 ； 4. n(n-1)/2

A: 1 B: 4

选择题

- 在内排序的过程中，通常需要对待排序的关键码集合进行多便扫描。采取不同排序方法会产生不同的排序中间结果。设要将序列{Q，H，C，Y，P，A，M，S，R，D，F，X}中的关键码按字母序的升序重新排列，则（　）是冒泡排序的一次扫描结果。

  （1）F，H，C，D，P，A，M，Q，R，S，Y，X

  （2）P，A，C，S，Q，D，F，X，R，H，M，Y

  （3）A，D，C，R，F，Q，M，S，Y，P，H，X

  （4）H，C，Q，P，A，M，S，R，D，F，X，Y

(4)

选择题

- 下述几种排序方法中，要求内存量（空间复杂性）最大的是(　　)。

  A.插入排序

  B.选择排序

  C.快速排序

  D.归并排序

D

选择题

- 以下排序方法中，关键字比较的次数与记录的初始排列次序无关的是（　　）。

A shell排序

B 起泡排序

C 直接插入排序

D 直接选择排序

D

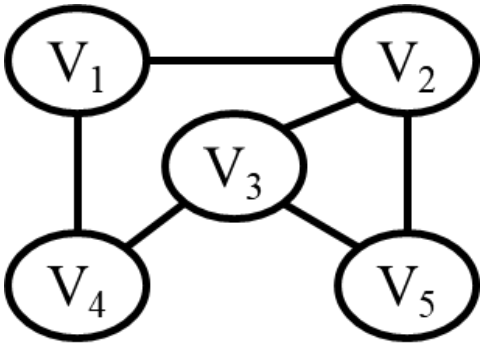
第十二讲 图

图的基本概念

- 图的逻辑结构：G = (V, E)
  - 图由顶点集合与边集合组成
  - V为有穷的顶点集合
  - E为边集合，是顶点的偶对（边的始点，边的终点）集合
  - 用 |V| 表示顶点的总数，|E| 表示边的总数
- 不同特点的图：
  - 稀疏图(sparse graph) / 稠密图(dense graph)：边数相对较少 / 较多
  - 完全图(complete graph)：包含所有可能的边
  - 有向图(directed graph) / 无向图(undirected graph)：顶点对有序 / 无序
  - 带权图(weighted graph)：边上标有权的图
  - 标号图(labeled graph)：各顶点均带有标号的图

无向图

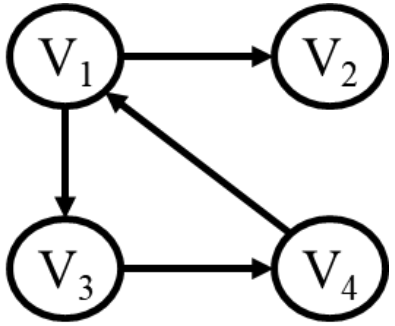
- 无向图：若图中每条边都是无方向的，则称为无向图。
  - 无向图中的边是由两个顶点组成的无序对
  - 无序对用圆括号表示，如(v_i, v_j)；(v_i, v_j)和(v_j, v_i)代表同一条边。
  - v_i和v_j是相邻结点，(v_i, v_j)是与顶点v_i和v_j相关联的边。



无向图

有向图

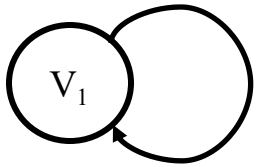
- 有向图：若图中每条边都是有方向的，则称为有向图。
  - 有向图中的边是由两个顶点组成的有序对。
  - 有序对用尖括号表示，如< v_i ,v_j >。v_i是边的始点，v_j是边的终点。
  - < v_j ,v_i >和< v_i ,v_j >表示不同的边。
  - 边< v_i ,v_j >与顶点 v_i , v_j 相关联

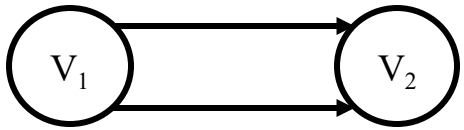


有向图

简单图

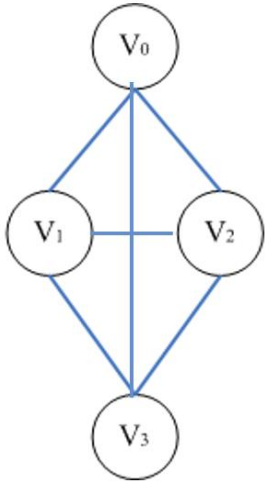
- 自环：一条边的两个端点是同一个顶点
- 重边：无向图中两个顶点之间有不止一条边，或有向图中两个顶点之间有不止一条同方向的边
- 简单图：不包含自环或重边的图
- 约定：除非特别说明，后续讨论的图都是简单图





完全图

- 完全图：图中的任意两个顶点之间都存在边
  - 对于有向图：任意两个顶点之间存在任意方向的边
- 记 n = |V|, e = |E|
- 若G是有向图，则$0 \le e \le n(n-1)$
  - 有向完全图具有n(n-1)条边
- 若G是无向图，则$0 \le e \le n(n-1)/2$
  - 无向完全图具有n(n-1)/2条边
- 在顶点个数相同的图中，完全图具有最多的边
  - 右图就是一个具有4个顶点的无向完全图，边数为:4*(4-1)/2=6



顶点的度数

- 在无向图中：
  - 与顶点v相关联的边数称为顶点v的度数，记为D(v)
- 在有向图中：
  - 入度：以v为终点的边的数目称为v的入度，记为ID(v)
  - 出度：以v为始点的边的数目称为v的出度，记为OD(v)
  - v的度数为其入度和出度之和，即D(v)=ID(v)+OD(v)
- 回忆对比：树结构中的度数
  - 结点的度数定义为子女的个数
  - 树的度数定义为所有结点度数的最大值
    - 如果将树视为无向图，度数还应该考虑连接父节点的边

顶点的度数

- 若图G有n个顶点，e条边，则有

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

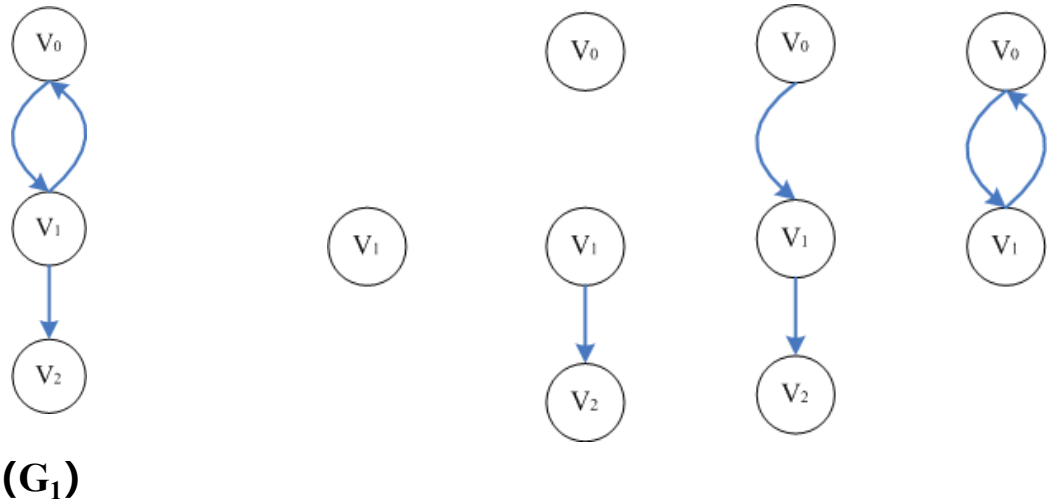
- 每一条边都参与计算了其两个顶点的度数
- 有向图与无向图均满足这一关系

相邻结点

- 在无向图中：
  - 若存在边(x, y)，则顶点x, y互为相邻结点，或者说x, y相邻。
- 在有向图中：
  - 若存在边<x, y>，即存在边由x指向y，则顶点y是顶点x的相邻结点。
- 相邻结点，也称为邻居、邻居结点、邻点等。

子图

- 定义：设有图G=(V，E)和G'=(V'，E')，如果V'是V的子集，E'是E的子集，则称G'是G的子图。
  - 下图给出了有向图G_1的若干子图。



(G_1)

路径

- 定义：对于无向图G=(V, E)，若存在顶点序列$v_{i0}, v_{i1}, \ldots, v_{in}$，使得$(v_{i0}, v_{i1}), (v_{i1}, v_{i2}), \ldots, (v_{in-1}, v_{in})$都在E中，则称从顶点$v_{i0}$到$v_{in}$存在一条路径。
  - 对于有向图G：只需要将条件修改为：$<v_{i0}, v_{i1}>, <v_{i1}, v_{i2}>, \ldots, <v_{in-1}, v_{in}>$都在E中
- 路径长度：路径上的边数。
- 简单路径：路径上的顶点除v_{i0}和v_{in}可以相同外，任意两个顶点都不相同。
- 回路或环：起点和终点相同的简单路径。

无向图的连通性

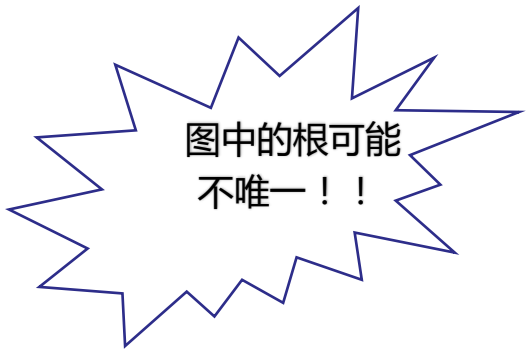
- 无向图的连通性
  - 连通：无向图G=(V，E)中，若从v_i到v_j有一条路径(从v_j到v_i也一定有一条路径)，则称v_i和v_j是连通的。
  - 连通图：若V(G)中任意两个不同的顶点v_i和v_j都是连通的(即有路径)，则称G为连通图。
  - 连通分量：无向图G的极大连通子图G’（极大：即任意增加G中结点和边到G’所得到的子图都不再连通），称为G的连通分量。
- 连通图只有一个连通分量，就是其自身；
- 非连通的无向图有多个连通分量。

有向图的连通性

- 有向图的连通性
  - 强连通图：有向图G=(V，E)中，若V中任意两个不同的顶点v_i和v_j都存在从v_i到v_j以及从v_j和v_i的路径，则称图G是强连通图。
  - 强连通分量：有向图G的极大强连通子图称为图G的强连通分量。
  - 弱连通图：将有向图G=(V, E)中的所有有向边替换为无向边，得到的无向图如果是连通图，则称有向图G是弱连通图。
  - 弱连通分量：有向图的G的极大弱连通子图称为图G的弱连通分量
- 强连通图只有一个强连通分量，就是其自身；
- 非强连通的有向图有多个强连通分量。

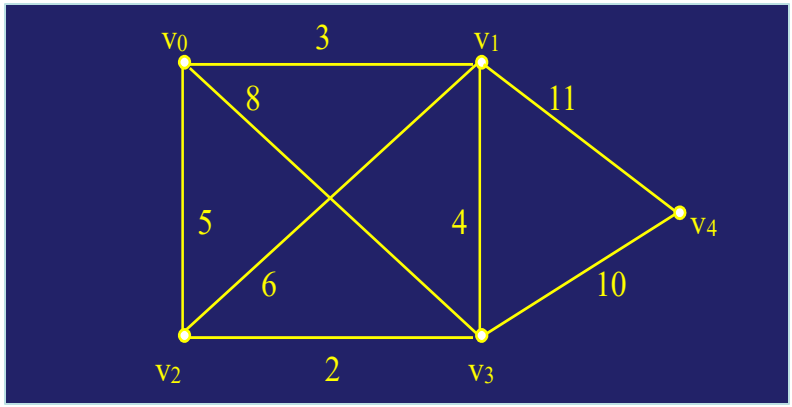
图的根

- 定义：有向图中，若存在一顶点v，从该顶点到图中其它所有顶点都存在路径，则称此有向图为有根图，v称为图的根
  - 连通的无向图中这一概念是平凡的



带权图

- 若给图的每条边都赋上一个权值，则称该图为带权图
  - 可以是无向图或者有向图
  - 通常这一权重具有实际意义，例如顶点间的距离、通信的花费、或者边所表示的关系的强弱等
- 连通的带权图也称为网络
- 下图展示了一个带权图（网络）



邻接矩阵表示法

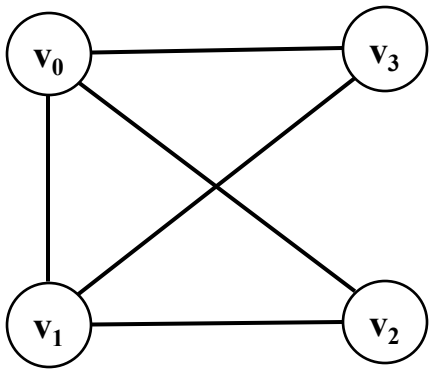
- 设G=(V, E)为具有n个顶点的图，其邻接矩阵(Adjacency matrix) A 为如下定义的 n 阶方阵

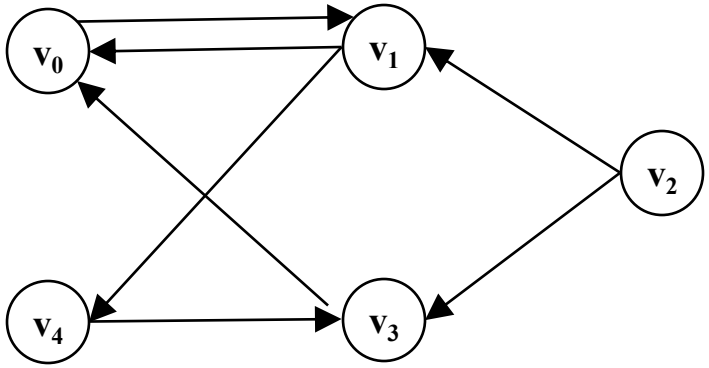
$$A[i,j]=\begin{cases}1, & \text{如果}(v_i,v_j)\text{或}<v_i,v_j>\text{是图}G\text{的边}\\ 0, & \text{如果}(v_i,v_j)\text{或}<v_i,v_j>\text{不是图}G\text{的边}\end{cases}$$

- 存储图的邻接矩阵，就维护了所有顶点以及顶点之间的相邻关系
  - 顶点信息：依次标号为0 ~ n-1
  - 关系信息：每条边都对应到邻接矩阵中的一个非零元素
- 对于不带权的无向图，邻接矩阵A是对称的；对于带权图或者有向图，A一般非对称

邻接矩阵表示法：示例

- 无向图(左)和有向图(右)的邻接矩阵分别为A_1和A_2
- A_1是对称矩阵，A_2是非对称矩阵





$$A_1 = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

$$A_2 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

邻接矩阵表示法：带权图

- 如果G是带权的图，w_{ij}是边(v_i,v_j)或$<v_i,v_j>$的权，则其邻接矩阵定义为：

$$A[i,j]=\begin{cases} w_{ij}, & \text{如果}(v_i,v_j)\text{或}<v_i,v_j>\text{是图}G\text{的边} \\ \infty \text{ 或 } 0, & \text{如果}(v_i,v_j)\text{或}<v_i,v_j>\text{不是图}G\text{的边} \end{cases}$$

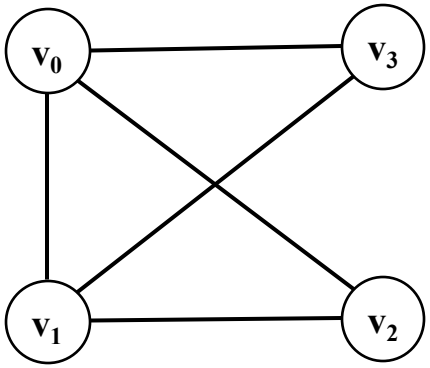
- 同样地，无向图的带权邻接矩阵是对称矩阵，有向图的带权邻接矩阵通常是非对称逆矩阵。
- 如果边(v_i,v_j)或$<v_i,v_j>$不存在，权重通常设置为无穷或0：
  - 如果权重表示结点间的距离等含义，则应该设置为无穷
  - 如果权重表示结点之间的流量、结点之间关系的强弱等含义，则应该设置为0。
  - 实际应用中也可以设定为其他约定好的特殊值，如 -1

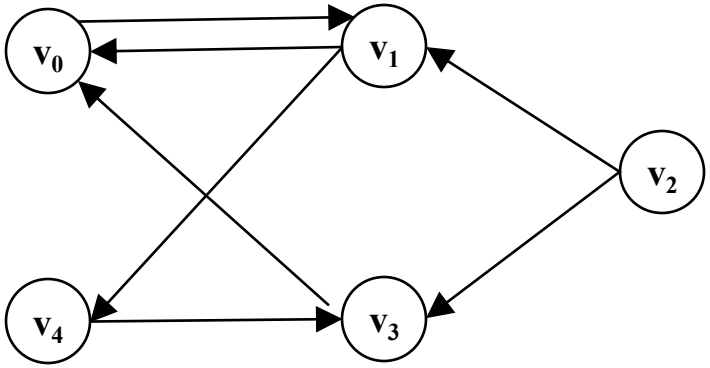
邻接表表示法

- 为了记录图中所有的边，可以为每个顶点 x 设置一个线性表，记录 x 的所有相邻结点。这种图的表示方式称为邻接表。
  - 该线性表是不定长的，x 有几个相邻结点，对应线性表中就有几个元素
  - 线性表中的一个元素记录了一个相邻结点 y，表示存在边 (x, y) 或 <x, y>
  - 对于有权图，或者边附带有其他信息，也可以存储在该线性表中

邻接表表示法：示例

- 无向图(左)和有向图(右)的邻接表分别为 G_1和 G_2





G_1=[[1, 2, 3],
[0, 2, 3],
[0, 1],
[0, 1]]

G_2=[[1],
[0, 4],
[1, 3],
[0],
[3]]

邻接矩阵表示法 vs 邻接表表示法

- 下面分别从不同的角度，比较这两种表示法的优劣
  - 用 n 表示结点数量，e 表示边数量
- 1. 构建一个图的复杂度
  - 邻接矩阵表示法：时间、空间复杂度都是 $O(n^2)$ 的。这意味着对于任何问题，如果采用这种存储方式，时间复杂度都至少是 $O(n^2)$
  - 邻接表表示法：时间、空间复杂度都是 O(n+e) 的。尽管 e 理论上能够达到 n^2 的量级，但实际大多数情况下远小于 n^2
  - 思考：O(n+e) 能够写成 O(e) 吗？
- 2. 对任意的两个结点 V_i, V_j，查询它们之间边的情况
  - 邻接矩阵表示法：O(1) 的复杂度就可以访问 A[i][j] 元素
  - 邻接表表示法：需要遍历 G[i] 中的元素，复杂度为 O(n)

邻接矩阵表示法 vs 邻接表表示法

- 3. 对于有向图，求顶点 V_i 的入度
  - 邻接矩阵表示法：查询邻接矩阵A的第 i 列即可，复杂度为O(n)
  - 邻接表表示法：遍历每一行查找 V_i，复杂度为O(n+e)
- 4. 查找顶点 V_i 的相邻结点，以及边的权值
  - 邻接矩阵表示法：查询邻接矩阵的第 i 行即可
  - 邻接表表示法：查询邻接表的第 i 行即可
  - 理论上二者的复杂度都是 O(n)。但是，邻接表只需要存储图中存在边，而邻接矩阵中还包含若干个 0 或无穷。相比之下，前者的信息更加紧凑，查询也会更快

邻接矩阵表示法 vs 邻接表表示法

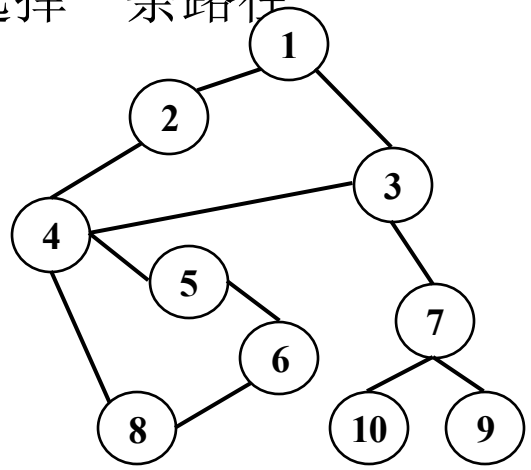
- 5. 存储稀疏图的效率
  - 邻接矩阵表示法：时间空间复杂度与图是否稀疏无关，$O(n^2)$
  - 邻接表表示法：每条边只需要存储一次（有向图）或两次（无向图），复杂度为 O(n+e)。如果 e 远小于 n^2，效率能够大大提高
- 现实中的图，绝大多数是稀疏图，而且找出相邻结点以及相连的边，基本上是最常用的操作。因此，邻接表表示法的应用比邻接矩阵表示法更加普遍。
- 如果要表示稠密图，或对图的操作主要是查询结点间是否存在边关系，使用邻接矩阵表示法就比较合适。

图的搜索与遍历

- 定义：
  - 图的搜索：从图中给定的一个起始顶点出发，寻找一条到目标顶点的路径，称为图的搜索。
  - 图的遍历：从图中某一顶点出发，按照某种方式系统地访问图中所有顶点，使得每一个顶点被访问且仅被访问一次，称为图的遍历，也称为图的周游。
- 搜索有两种基本策略：深度优先搜索（Depth First Search, DFS）以及广度优先搜索（Breadth First Search, BFS）
- 这两种策略同样可以用于图的遍历，即深度优先遍历与广度优先遍历。因此，图的搜索与遍历是联系紧密的两种操作

深度优先搜索

- 下面给出了一个图的搜索问题示例：在无向图中，寻找由结点1到结点8的路径
- 最简单的策略之一（DFS）：能向前走就向前走，无法前进时就回溯到之前的位置，尝试其他的路径
  - 在有多条路径可供选择时，可以任意/随机选择一条路径
  - 如果运气较好：1－2－4 －8



深度优先搜索：剪枝

- 我们已经看到，实际问题的状态空间可能非常大，因此即使是O(n+e)的复杂度也是不可接受的（魔方问题）
- 搜索过程中，存在一些冗余过程。例如：
  - 根据具体问题性质可以判断出，从某个结点出发无法再到达终点
  - 如果追求路径（或带权路径的权重和）尽可能短，一些代价高昂的路径也可以直接被舍弃掉
- 去除这些冗余的搜索，就是去除了搜索树中的若干分枝。因此，就将这一过程称为剪枝（Pruning）
  - 可行性剪枝：如果可以判断再走下去已经无法到达终点，就可以立即回溯
  - 最优性剪枝：求最小代价/最短路径时，如果当前的代价已经超过当前最优解的代价，就可以立即回溯

深度优先搜索：启发式搜索

- 在图的搜索过程中，不同路径的选择导致不同的搜索树，直接影响搜索的效率。
- 为了尽快地找到可行路径或者最短路径，在路径选择时，可以针对不同的具体问题，使用经验法则或代价估计来指导搜索的方向，从而减小搜索的范围，更快达到目标。
  - 采取的经验法则，可以是经过数学证明的严谨性质，也可以是只基于经验的规则。它们都统称为启发式规则（Heuristics）
  - 在一些问题上，严格证明过的启发式规则可以保证得到最优解。
  - 未经严格分析的启发式规则，也能对搜索效率提供不同程度的提升
  - 启发式搜索也可以视为广义的剪枝策略

深度优先遍历

- 采用深度优先搜索的策略，遍历图中的所有顶点，称为深度优先遍历。具体步骤如下
  - (1) 在图中任意选择没有走过的顶点，作为遍历的起点
  - (2) 当前顶点如果有未走过的相邻顶点，则任意选择一个走过去，并重复步骤 (2)；如果没有未走过的相邻顶点，就应该沿着搜索树向上回溯，并重复步骤 (2)。
  - (3) 当回溯到起点且无法继续走时，如果所有的顶点都已经走过，则遍历结束；否则转到步骤 (1)

深度优先遍历

- 深度优先遍历的时间复杂度（假定采用邻接表实现）：
  - 初始化所有结点的时间复杂度为 O(n)
  - 主函数中，每个结点都会被检查一次
  - 子函数中遍历了结点的所有邻居。最坏情况下，每条边都可能被遍历一次
  - 时间复杂度为 O(n+e)

深度优先遍历：应用

- 思考：如何判断一个无向图是否连通，以及是否存在回路？
- 连通性判断：
  - 任选一个顶点进行一次深度优先遍历的尝试（dfs_visit），如果能够访问到所有的顶点，就说明图是连通的。
- 回路判断：
  - 若图中不存在回路，则在深度优先遍历的过程中，对于当前顶点 v 的每个邻居 u，u 或者没有被访问过，或者是 v 在搜索树中的父节点。每前进一步都检查所有的邻居，若不满足上述性质，就说明图中存在回路。
  - 另一种思路：使用并查集。将每个顶点初始化为一个等价类；遍历所有的边，对每条边关联的两个顶点执行 Union 操作。如果某两个顶点在 Union 操作之前就已经属于同一等价类了，则说明存在回路。

广度优先搜索

- 给定图中的起点和终点，深度优先搜索（DFS）得到的路径不保证是最短的。
- 广度优先搜索能保证得到长度最短的路径。
  - 广度优先搜索，也称为宽度优先搜索
- 广度优先搜索的关键在于将顶点“分层”
  - 对应搜索树中结点的层次
  - 起点位于第 0 层，距离为 1 的顶点位于第一层，依此类推
  - 按层次从低到高扩展顶点，并用队列存放；先搜索层数较低的结点，后搜索层数较高的结点，从而保证能够找到最短路径
  - 为了记录最短路径，顶点入队时需要记录其在搜索树中的父结点

广度优先搜索

- 广度优先搜索的时间复杂度分析（假定采用邻接表实现）：
  - 初始化结点的代价为 O(n)
  - 迭代搜索过程中，每个结点至多入队 / 出队一次。因此，入队 / 出队的代价为O(n)
  - 对于队首元素，要对其所有邻居结点进行判断。该步骤中总的判断次数取决于图中边的总数，即代价为 O(e)
  - 重构路径的代价为 O(n)
- 广度优先搜索的时间复杂度为 O(n+e)

广度优先遍历

- 采用广度优先搜索的策略，遍历图中的所有顶点，称为广度优先遍历。具体步骤如下
  - (1) 在图中任意选择没有走过的顶点，作为遍历的起点，并访问之
  - (2) 依次访问图中与起点距离为 1 的顶点、距离为 2 的顶点等等，直到由起点可达的所有顶点都已经访问过。对于距离相同的顶点之间的先后次序则无要求。
    - 对图中的两个顶点 a, b，将 a 到 b 的距离定义为由 a 到 b 的所有路径中，边数最少的路径上的边数。
  - (3) 如果所有的顶点都已经走过，则遍历结束；否则转到步骤 (1)

广度优先遍历

- 广度优先遍历的时间复杂度：（假定采用邻接表实现）
  - 分析过程与广度优先搜索相同
  - 初始化结点的代价为 O(n)
  - 迭代搜索过程中，每个结点至多入队 / 出队一次。因此，入队 / 出队的代价为O(n)
  - 对于队首元素，要对其所有邻居结点进行判断。该步骤中总的判断次数取决于图中边的总数，即代价为 O(e)
- 总的时间复杂度同样为 O(n+e)

总结：图的搜索与遍历

- 图上的搜索操作与遍历操作是关系紧密的两种操作，有深度优先与广度优先两种基本策略。
  - 邻接表实现下，其时间复杂度都为 O(n+e)
- 许多问题可以建模为图上的搜索问题，使用 DFS/BFS解决。
- 深度优先搜索/遍历：
  - 特点是具有回溯过程；可以使用剪枝策略、启发式策略进行优化
  - 所需要的存储空间较少，主要是递归所需的栈空间
- 广度优先搜索/遍历：
  - 特点是优先遍历距离较近的顶点，能够搜索最短路径
  - 所需要的存储空间较大
- 无论采用哪一种策略，搜索 / 遍历时都需要进行判重操作。

第十三讲 最小生成树

最小生成树

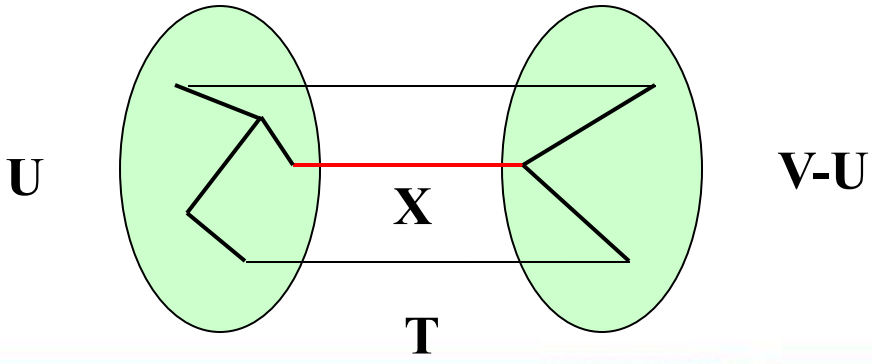
- 定义：生成树（Spanning Tree）
  - 对于带权无向连通图 G=(V, E)，以及 G 的一个子图 G'=(V', E')，若 V=V'，且 G' 是不含回路的连通图，则称 G' 是 G 的生成树
  - 连通图的生成树是连通图的一个极小连通子图，它含有图中的全部 n 个顶点，以及足以构成一棵树的 n-1 条边。
    - 增加一条边，则必定构成环；
    - 去掉一条边，则连通图变为不连通的。
  - 基于 DFS 以及 BFS 得到的搜索树，就是图的生成树，因此也称为 DFS 生成树、BFS 生成树
- 对于非连通图，从任一顶点出发无法访问到所有的顶点，只能得到各连通分量的生成树所组成的生成森林

最小生成树

- 定义：最小生成树（Minimum Spanning Tree, MST）
  - 定义生成树的权值为其中所有边的权值之和，则 G 的所有生成树中，权值最小的生成树称为图 G 的最小生成树
- 根据定义，最小生成树并不唯一
- 类似地，可以定义图的最大生成树
  - 求解最小生成树与最大生成树的本质是相同的，只需要将图中的所有边权取相反数即可转换二者
- 为了研究如何求解一个连通无向图的最小生成树，我们先来介绍最小生成树的一条重要性质

最小生成树的性质

- G=(V, E) 是带权连通无向图，U 是 V 的任意非空子集，考虑所有一端在 U 中而另一端不在 U 中的边，其中权值最小的边 X 一定属于 G 的一棵最小生成树 T。
- 直观理解：
  - 将图 G 任意地划分成两部分
  - 由于 G 是连通的，这两部分之间必然存在边作为“桥梁”
  - 任意一个权值最小的“桥梁”，一定属于某棵最小生成树 T



Prim算法

- Prim 算法是求解最小生成树问题的贪心算法，其基本思路如下：
  - 假设G = (V, E)是有n个节点的带权连通图，为构建 G = (V, E) 的一棵最小生成树 T = (V', E')，将 V' 初始化为包含V中一个顶点（任意一个顶点）的集合，将 E' 初始化为空集
  - 每次从 V – V' （即在T外的顶点）中，寻找一个“距离 V' 最近”的结点，加入 V' 中；对应关联的边加入 E' 中
    - 距离 V' 最近的含义为，与 V' 中的结点相邻，且边权最小
    - 可以理解为，每次从跨越了 V' 与 V – V' 的边中，寻找权值最小的，将其关联的顶点加入 V' 中，将这条边本身加入 E' 中
  - 重复该步骤 n – 1次，就得到了最小生成树 T

Prim算法的时间复杂度

- 记 n = |V|, e = |E|
- 算法的主要代价在于，不断从堆中取出最小元素，以及调整堆的操作
  - 要进行 O(n) 次取出最小元素操作，代价为 O(nlog n)
  - 每个结点加入 V' 后都需要更新其全部邻居结点（DECREASE-KEY）
  - 最坏情况下，要进行 O(e) 次更新，代价为 O(elog n)
- 基于堆实现的 Prim 算法复杂度为 O(nlog n + elog n)
  - 由于通常要求输入的图必须是连通图，即 e ≥ n -1
  - 因此，可以认为基于堆实现的 Prim 算法的时间复杂度为 O(elog n)

Prim算法的时间复杂度

- 如果每次寻找最小值都使用复杂度为 O(n) 的简单遍历，则可以避免维护堆所产生的代价。
  - 对于 n – 1个结点，需要进行寻找最小的操作
- 总的复杂度将变为 $O(n^2)$
  - 对于稠密图，即 e 达到 n^2 水平，n^2 的复杂度低于 elog n，此时适合采用基于遍历的方法
  - 对于稀疏图，则 elog n 远小于 n^2，此时适合采用基于堆的方法

Kruskal算法

- Kruskal 算法同样是贪心算法，其基本思路如下
  - 为构建 G = (V, E) 的一棵最小生成树 T = (V', E')，将 V' 初始化为 V，将 E' 初始化为空集，即 G' 初始时具有 n 个连通分量
  - 每次从边集 E 中取出权值最小的边 e 。如果 e 关联的两个顶点在 G' 中属于不同的连通分量，就将 e 加入到 E' 中
  - 不断重复上述步骤，直到图 G' 成为连通图

Kruskal算法的时间复杂度

- 记 n = |V|, e = |E|
- 初始时可以应用堆排序、快速排序和二路归并排序对图中所有的边进行排序，代价为 O(elog e)
- 初始化并查集的代价为 O(n)
- 并查集上的 Union 操作代价为 O(1)，Find 操作的均摊代价也可以认为是 O(1)
  - 执行 O(e) 次 Find 操作
  - 执行 O(n) 次 Union 操作
  - 代价为 O(e + n)
- 对于连通图，有 n – 1 ≤ e ≤ n(n-1)/2，则总代价即为 O(elog e)
  - 也可以写为等价的 O(elog n)，与基于堆实现的 Prim 算法相同

Kruskal算法的时间复杂度

- Kruskal 算法的优化方法：
  - Kruskal 算法不断取出权重最短的边，若先对所有边进行排序，然后在依次遍历，复杂度为 O(elog e)
- 针对这一操作，也可以采用堆数据结构进行优化
  - 对所有边建堆，时间复杂度为 O(e)
  - 每次取出权值最小的边，复杂度为 O(log e)
  - 如果执行每条边都被取出一次，最坏情况下的复杂度同样为 O(elog e)
- 但实际构建过程中，只需要取权值较小的一部分边进行判断
  - 尤其是对于稠密图，实际执行次数将远小于 e

总结：Prim 算法 & Kruskal 算法

- 是求解最小生成树问题的两种贪心算法，通过不断选取边来构建生成树
  - Prim 算法：选取跨越 V’ 与 V – V’ 的权重最小的边
  - Kruskal 算法：选取跨域两个连通分量的权重最小的边
- 要求输入为连通的带权无向图
- 时间复杂度
  - Prim 算法（堆实现）：O(elog n)，适用于稀疏图
  - Prim 算法（遍历实现）：$O(n^2)$，适用于稠密图
  - Kruskal 算法（并查集实现）：O(elog n)，适用于稀疏图

简单练习（十）

判断题

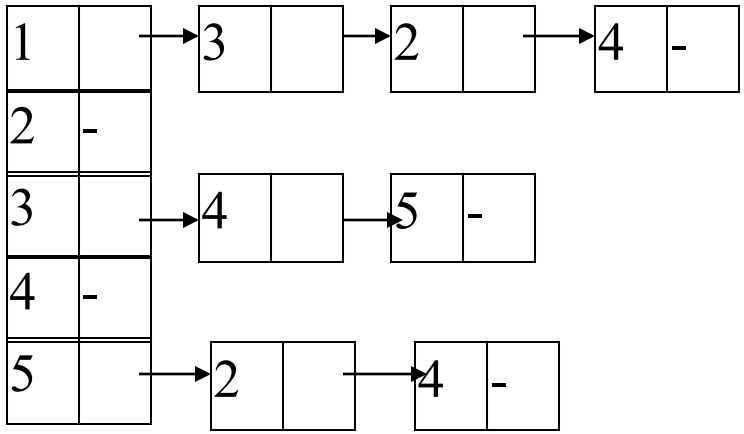
1. 用相邻接矩阵法存储一个图时，在不考虑压缩存储的情况下，所占用的存储空间大小只与图中结点个数有关，而与图的边数无关。
2. 任何有向图网络（AOV-网络）拓扑排序的结果是唯一的。
3. 强连通分量是有向图中的极大强连通子图。

Y N Y

选择题

- 已知一个有向图的邻接表存储结构如图所示，根据有向图的深度优先遍历算法,从顶点v1 出发，所得的顶点序列是(　　)，按广度优先遍历算法得到序列(　　)。

A. v1,v2,v3,v5,v4
B. v1,v2,v3,v4,v5
C. v1,v3,v4,v5,v2
D. v1,v4,v3,v5,v2
E. v1,v3,v2, v4,v5



C　　E

选择题

- 设无向图G中顶点数为n，则图最多有（　　）条边 。

  A、n（n+1）/2

  B、n（n-1）

  C、n（n-1）/2

  D、n（n+1）

C

选择题

- 图的生成树（　　），一个连通图的生成树是该图的一个（　　）连通子图，n个顶点的生成树有（　　）条边。

  A. 是唯一的、最大、n

  B. 不是唯一的、最小、n+1

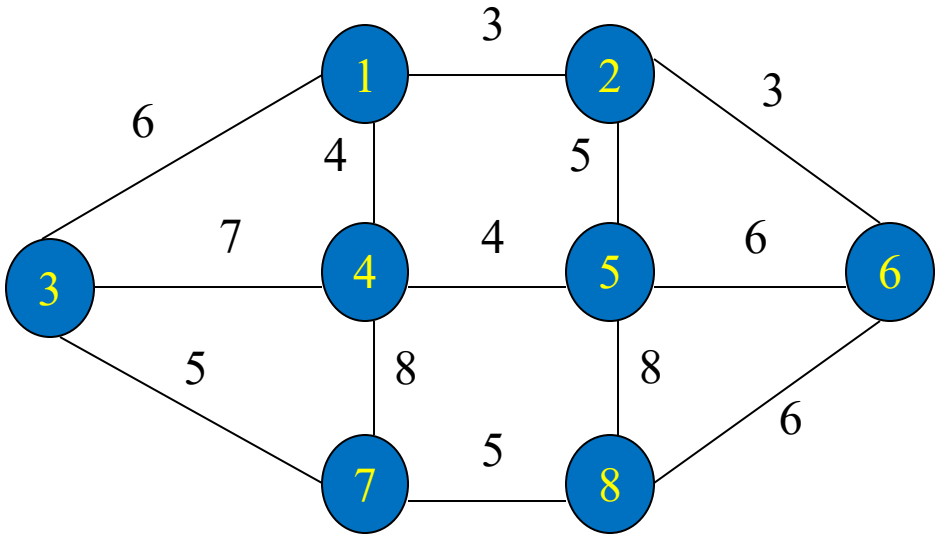
  C. 唯一性不能确定、最大、n-1

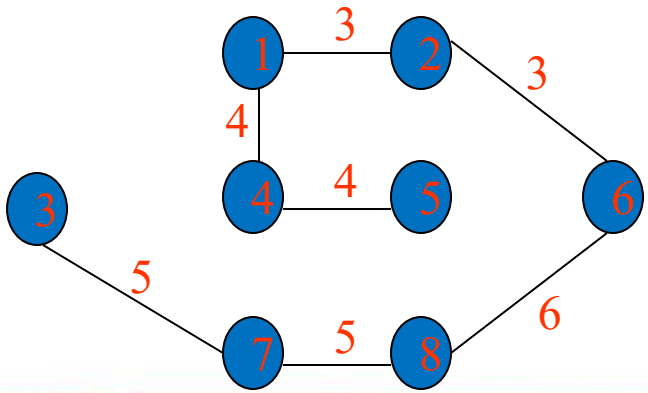
  D. 唯一性不能确定、最小、n-1

C

习题解答

- 由Kruskal算法，从V1点开始，求下图的一棵最小生成树。





第十四讲 最短路径算法

最短路问题

- 在带权无向或有向图上，一个顶点到另一个顶点的边权值之和最小的路径，称为最短路，边权值之和称为最短路的长度。
  - 现实问题中，顶点可能代表地址、阶段性目标、任务等
  - 边可能代表某种操作，例如从一个地方走到另一个地方，或者完成某个工作步骤
  - 边权值可能代表距离、时间、花费等
- 我们先前学过的广度优先搜索算法同样用于求最短路
  - 不过，现在需要考虑路径的总权重，而不是路径包含的边数
  - 如果所有边的权重相等，就同样可以使用广度优先搜索
- 求最短路也可以使用深度优先搜索算法
  - 但是，需要遍历的解空间往往过大，效率无法保证

最短路问题

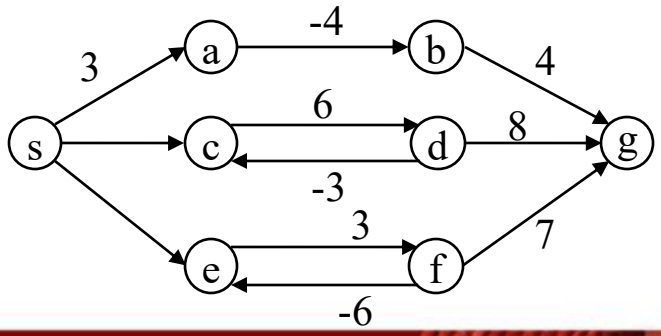
- 最短路径问题的各种变体：
- 单源最短路径问题：给定图 G=(V, E)，我们希望找到从给定源结点 s∈V 到每个结点 v∈V 的最短路径
  - Dijkstra 算法就是解决单源最短路径问题的经典算法
- 单目的地最短路径问题：找到从每个结点 v 到给定目的地结点 t 的最短路径。
  - 对于无向图，与单源最短路径问题是相同的
  - 对于有向图，将图中每条边的方向翻转过来，同样可以转换为单源最短路径问题

最短路问题

- 单结点对最短路径问题：找到从给定结点 u 到给定结点 v 的最短路径
  - 如果解决了结点 u 的单源最短路径问题，那么也就解决了这个问题
- 所有结点对的最短路径问题（多源最短路径问题）：对于每对结点 u 和 v，找到从结点 u 到结点 v 的最短路径
  - 虽然可以对每个结点都运行一遍单源最短路径算法，但通常可以更快地解决这个问题
  - Floyd 算法用于求解所有结点对的最短路径问题
- 接下来的讨论都是基于有向图。对于无向图，可以将每条边改为双向边，从而转化为有向图。

最短路中的负权边

- 某些单源最短路径问题可能包括权重为负值的边。
- 但如果图中存在着权重为负值的回路，且该回路对于结点 v 可达，则从 v 出发到部分顶点的最短路径可能不存在
  - 从源点 s 到回路上的任意结点的路径都不可能是最短路径
  - 因为我们只要沿着任何“最短”路径，再沿着权重为负的回路走一遍，就可以找到一条权重更小的路径
  - 下图中，结点 e, f 形成了一个权重为负，且从源点 s 可达的回路，这使得 s 到 e, f, g 的最短路径不存在



最短路中的负权边

- 对于单源最短路径问题，Dijkstra 算法假设输入图中所有的边权非负，不能在含有负权边的图上运行
  - 其他单源最短路径算法，例如 Bellman-Ford 算法，允许图中含有负权边，且可以识别图中可能存在的负回路，但效率低于 Dijkstra 算法
- 对于多源最短路径问题，Floyd 算法允许图中含有负权边，也可以检测图中可能存在的负回路

Dijkstra 算法的基本思路

- Dijkstra 算法运行时维护一组结点集合 U
  - 若结点 t 被加入集合 U，表明已经找到了由源点 s 到结点 t 最短路径
  - 初始时 U 仅包括源点
- 同时，维护每个顶点到源点 s 的最短路径长度估计列表
  - Dist[s] 初始化为 0，其余顶点i到s的Dist[i] 初始化为 ∞
- 进行重复迭代：
  - 选取 V – U 集合中最短路径长度估计最小的顶点 u，加入到集合 U 中
  - 根据顶点 u 关联的边，更新 V – U 中所有顶点的最短路径长度估计
  - 直到全部顶点都加入 U，求解完成

Dijkstra 算法的时间复杂度

- 记 n = |V|, e = |E|
- 算法的主要代价在于，不断从堆中取出最小元素，以及调整堆的操作
  - 要进行 O(n) 次取出最小元素操作，代价为 O(nlog n)
  - 每个结点加入 V' 后都需要更新其全部邻居结点（DECREASE-KEY）
  - 最坏情况下，要进行 O(e) 次更新，代价为 O(elog n)
- 基于堆实现的 Dijkstra 算法复杂度为 O((n+e)log n)
- 每次使用遍历方法寻找距离最小的顶点，实现的 Dijkstra 算法复杂度为 $O(n^2)$

Floyd 算法

- 为了计算图中任意源点到任意终点的最短路径：
  - 可以执行 n 次单源最短路径算法，复杂度为 O(n*(n+e)*log n) 或 $O(n^3)$
  - 还可以使用动态规划的方法，即 Floyd 算法
- Floyd 算法也称 Floyd-Warshall 算法
  - 该算法用于求解多源最短路径问题
  - 算法允许图中存在权值为负的边
  - 算法假定图中不存在负回路，但算法本身也可以用于检测负回路是否存在

Floyd 算法

- 动态规划的原问题：对于任意顶点 v_i, v_j，求出从 v_i 到 v_j 的最短路径长度
- 定义如下的子问题：对于任意顶点 v_i, v_j，求出从 v_i 到 v_j 且中途只经过 $v_1, v_2, \cdots, v_k$ 的最短路径长度（$0 \leq k \leq n$）
  - 这样，原问题就是取 $k = n$ 时的子问题，即“最大”的子问题
- 用最短路径权重矩阵 D 来记录子问题的答案
  - $D^k(i,j)$ 表示规模为 k 的子问题中，从 v_i 到 v_j 的最短路径长度
- 动态规划的边界条件
  - $k = 0$ 时的子问题要求路径不能途径任何顶点
  - 因此， D^0 实际上就是原图的带权邻接矩阵，不相邻的顶点之间权值为无穷

Floyd 算法

- 状态转移方程：如何由 D^k 计算 D^{k+1} ？
- 与规模为 k 的子问题相比，规模为 $k+1$ 的子问题允许路径中经过一个新的顶点 v_{k+1}
- 考虑新的子问题中，最短路径是否经过了 v_{k+1}
  - 若 $D^{k+1}(i,j)$ 对应的最短路径中没有经过 v_{k+1}：

$$D^{k+1}(i,j) = D^k(i,j)$$

  - 若 $D^{k+1}(i,j)$ 对应的最短路径中经过了 v_{k+1}，可以将该路径拆分成，只途经 $v_1, v_2, \cdots, v_k$，由 v_i 先到达 v_{k+1}，再到 v_j

$$D^{k+1}(i,j) = D^k(i,k+1) + D^k(k+1,j)$$

  - 两种情况取最小值：

$$D^{k+1}(i,j) = \min\{D^k(i,j), D^k(i,k+1) + D^k(k+1,j)\}$$

Floyd 算法

- 最短路径的构建
  - 为了得到最短路径本身，可以设置 Prev 矩阵记录最短路的前驱顶点
- 每次根据状态转移方程计算子问题时，
  - 若 $D^{k+1}(i,j)$ 对应的最短路径中没有经过 v_{k+1}：

$$Prev^{k+1}(i,j) = Prev^{k}(i,j)$$

  - 若 $D^{k+1}(i,j)$ 对应的最短路径中经过了 v_{k+1}：

$$Prev^{k+1}(i,j) = Prev^{k}(k+1,j)$$

- 算法的时间复杂度：
  - 算法顺序计算 O(n) 个子问题，每个子问题进行 $O(n^2)$ 次迭代，每次迭代的代价为 O(1)
  - 总的时间复杂度为 $O(n^3)$

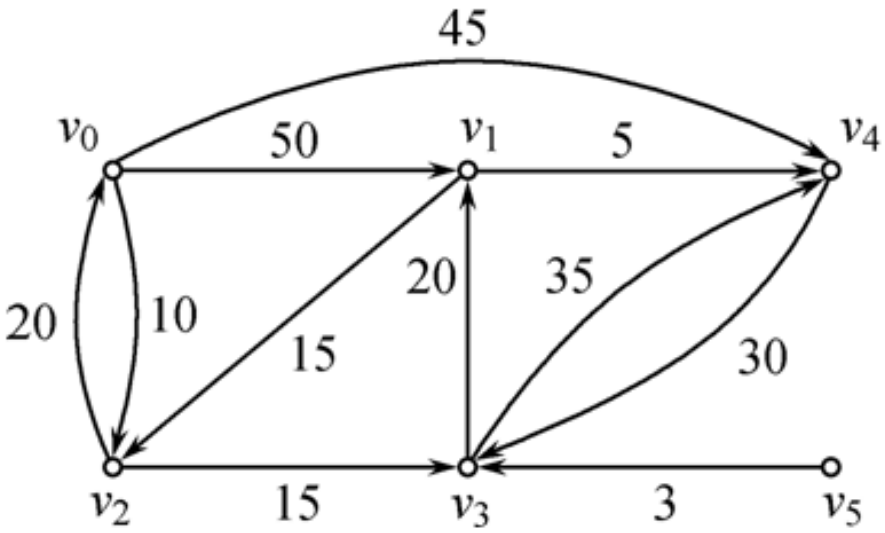
思考

- 给定无向图 G = (V, E)，如何计算任意两个顶点之间是否连通？如果 G 为有向图，又应该如何计算连通性？
- 对于无向图：
  - 可以从给定的源点开始进行 DFS/BFS 遍历，遍历到终点则说明连通
  - 可以利用并查集，依据边建立等价类，表示连通这一等价关系
  - 将所有边赋予权值为 1，运行 Floyd 算法，得到的结果中若两个顶点之间最短路长度为无穷，表示不连通；否则表示连通
- 对于有向图：
  - 仍可以使用 Floyd 算法，以及基于 DFS/BFS 的遍历方法
  - 并查集将不再适用，此时结点的连通性也有方向，不构成等价关系

简单练习（十一）

解答题

- 下面是一个带权图，试给出该图的邻接矩阵/邻接表存储表示，试给出该图中由V_0到V_4的最短路径。



第十五讲 拓扑排序和关键路径（可选）

拓扑排序问题

- 在有向图中，用顶点代表活动，有向边代表活动间的先后关系，则该有向图称为顶点活动图（AOV网，Activity on Vertex Network）
- 对于一个AOV网 G = (V, E)，其拓扑排序是 G 中所有结点的一种线性次序，满足如下条件
  - 如果图 G 包含边 <u, v>，则顶点 u 在拓扑排序中处于顶点 v 的前面
- 对于有向无环图（Directed Acyclic Graph, DAG），一定存在拓扑排序；如果图中存在环，则不存在拓扑排序，算法应该识别环的存在。

拓扑排序算法

- 直观的思路：不断查看图中是否存在可以立即完成的活动，如果存在就进行该活动
- 对应到图中：
  - 从 AOV 网中选择一个入度为 0 的顶点，将其输出到拓扑排序序列中
  - 从 AOV 网中删除此顶点，以及该顶点的所有出边
- 反复执行以上两步，直到：
  - 所有顶点都已经输出，此时拓扑排序已经完成
  - 或者，剩下的顶点入度都不为 0，此时说明 AOV 网中存在回路，拓扑排序无法进行

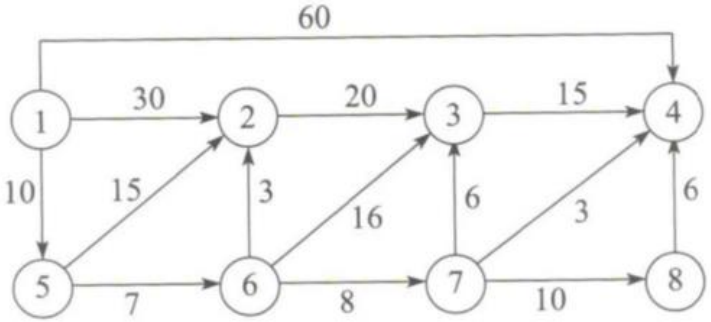
拓扑排序算法的时间复杂度

- 记 n = |V|，e = |E|，拓扑排序算法的主要代价包括：
  - 初始时维护所有结点的度数，代价为 O(n + e)
  - 每条边会被处理一次，代价为 O(e)
  - 每个顶点会被添加到队列中，并输出到结果序列中，代价为 O(n)
- 因此，拓扑排序算法的时间复杂度为 O(n+e)

- 补充：简单练习（十一）

简答题

- 运行 Dijkstra 算法求出顶点 1 到各顶点的最短路径，将算法运行各步的状态填入下表



所选顶点	U(已确定最短路径的集合)	T(未确定最短路径的集合)	2	3	4	5	6	7	8
初态	{1}	{2, 3, 4, 5, 6, 7, 8}	30	∞	60	10	∞	∞	∞
5	{1, 5}	{2, 3, 4, 6, 7, 8}	25	∞	60	10	17	∞	∞
6	{1, 5, 6}	{2, 3, 4, 7, 8}	20	33	60	10	17	25	∞
2	{1, 5, 6, 2}	{3, 4, 7, 8}	20	33	60	10	17	25	∞
7	{1, 5, 6, 2, 7}	{3, 4, 8}	20	31	28	10	17	25	35
4	{1, 5, 6, 2, 7, 4}	{3, 8}	20	31	28	10	17	25	35
3	{1, 5, 6, 2, 7, 4, 3}	{8}	20	31	28	10	17	25	35
8	{1, 5, 6, 2, 7, 4, 3, 8}	{}	20	31	28	10	17	25	35

算法填空

- 完成下列算法，计算无向图中所有连通分量的结点个数。
  - 输入：图的结点数，边数和边的列表
  - 输出：每个连通分量的结点个数

输入样例：

5

[(0, 1), (1, 2), (3, 4)]

输出样例：

[3, 2]

算法填空

```
def dfs(node, visited, graph, n):
    count = 1
    visited[node] = True
    for i in range(n):
        if i in graph[node] and ______(1)______:
            count += _________(2)_________
    return count

def count_components(n, edges):
    graph = {i: [] for i in range(n)}
    for u, v in edges:
        _________(3)_________
        _________(4)_________

    visited = [False] * n
    components = []
    for i in range(n):
        if __________(5)__________:
            components.append(___________(6)___________)
            return components

n = int(input())
edges = eval(input())
print(count_components(n, edges))
```

算法填空

```
def dfs(node, visited, graph, n):
    count = 1
    visited[node] = True
    for i in range(n):
        if i in graph[node] and not visited[i]:
            count += dfs(i, visited, graph, n)
    return count

def count_components(n, edges):
    graph = {i: [] for i in range(n)}
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)

    visited = [False] * n
    components = []
    for i in range(n):
        if not visited[i]:
            components.append(dfs(i, visited, graph, n))
            return components

n = int(input())
edges = eval(input())
print(count_components(n, edges))
```

算法填空

- 堆排序：输入若干个整数，下面的程序使用堆排序算法堆这些整数从小到大排序，请填空。程序中建立的堆是大顶堆，最大元素在堆顶。
- 输入样例：1, 3, 43, 8, 7
- 输出样例：1, 3, 7, 8, 43

算法填空

```
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2
    if left < n and arr[i] < arr[left]:
        largest = left
    if right < n and arr[largest] < arr[right]:
        largest = right
    if largest != i:
        ____________(1)____________
        ____________(2)____________

def buildMaxHeap(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        ____________(3)____________

def heapSort(arr):
    n = len(arr)
    buildMaxHeap(arr)
    for i in range(n - 1, 0, -1):
        ____________(4)____________
        ____________(5)____________
    return arr
```

算法填空

```
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2
    if left < n and arr[i] < arr[left]:
        largest = left
    if right < n and arr[largest] < arr[right]:
        largest = right
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def buildMaxHeap(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

def heapSort(arr):
    n = len(arr)
    buildMaxHeap(arr)
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
    return arr
```

期末考试

- 课程名称：数据结构与算法(B)
- **考试时间：2025.06.17（周二）下午2点-4点**
- **考试地点：理教203**
- 考试形式：闭卷
- 特别要求：
  - 带好校园卡！带好校园卡！
  - 考前及时清理座位上的物品
  - 开始30分钟以后才可以交卷，试卷和答题纸要同时交上来
  - 缓考由院系审批，考前把缓考回执单交给老师

- 祝大家考试顺利！