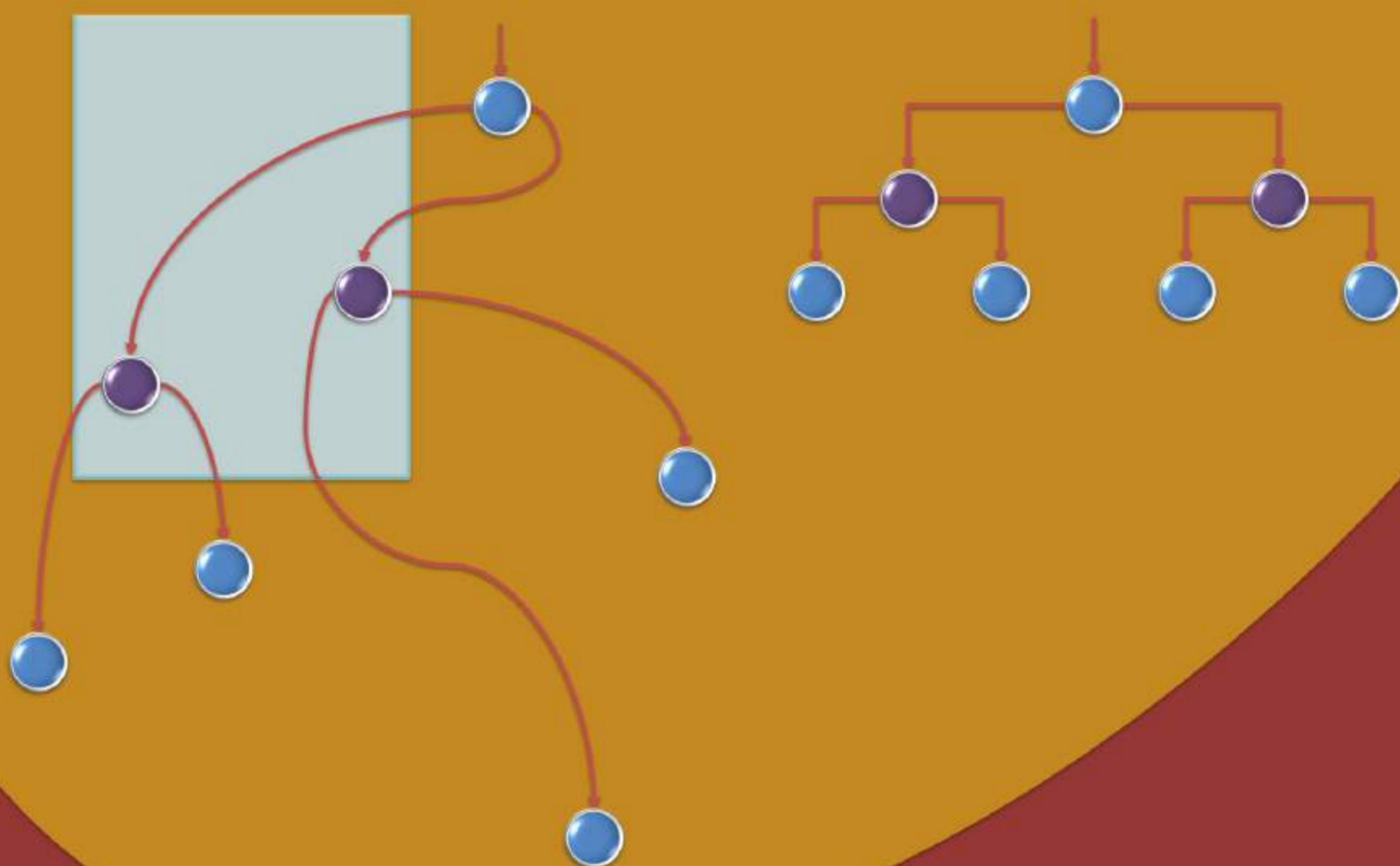
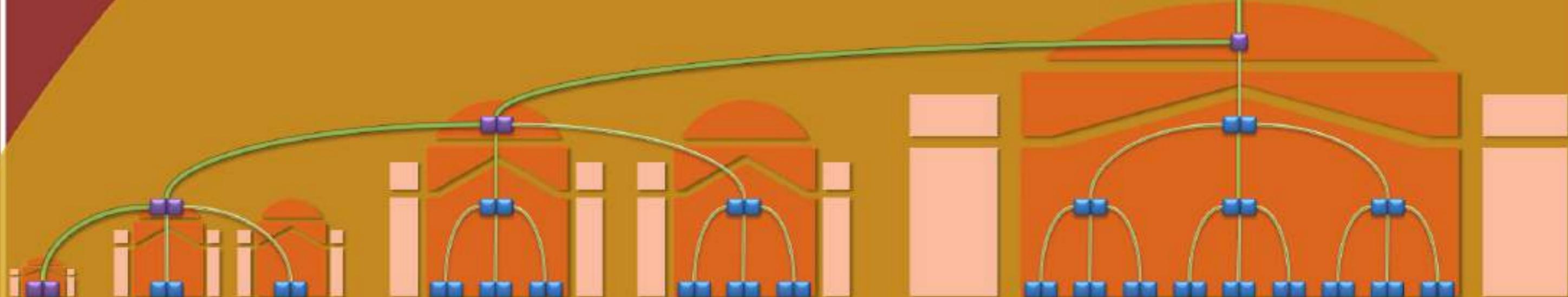


# 数据结构



清华大学 · 2016秋





**课程大纲**

**2016秋季学期**

考评环节	编程作业 ( PA )	期中考试	期末考试	参与	加分
30240184	15 x3次	20	25	10	5
00240074	15 x4次		30	10	5

郑伟輝  
deng@tsinghua.edu.cn

◆ 概念体情况，期末总评时可能有咨询  
Data Structures & Algorithms, Tsinghua University

**PA要求**

- ◆ **须独立完成** 否则题分可能折扣，直至记-100  
双方同意，不作区分；无论是否同班、同系、同届
- ◆ **如何判定？** 代码雷同度——<http://MOSS.stanford.edu>  
标准算法除外；二分查找、KMP、Dijkstra、快速排序、DFS，等等
- ◆ **什么不可以？** 通代码，或足以导致雷同的伪代码  
故意理解及解题思路
- ◆ **什么可以交流？** 算法及数据结构的设计与选用方案  
测试用例及对应的输出
- ◆ **凡犯雷同不准？** 涉及的人员、文献、资源等，须在代码声明中标注  
只要注明完整、准确，至少不会扣分

Data Structures & Algorithms, Tsinghua University

**教学团队**

 梁湘 助教	 丁伟 助教
 何俊 助教	 陈贤 助教
 李静 助教	 王帅 助教

Data Structures & Algorithms, Tsinghua University

## 教材 + 教辅

- 

**数据结构与习题解析**  
2013年9月第三版，刘俊辉  
7-302-33064-6  
7-302-33065-3
- 

**The Design & Analysis of Computer Algorithms**  
J. E. Hopcroft, et al  
7-111-17775-4
- 

**数据结构与算法分析**  
M. A. Weiss原著，陈越改编  
7-115-13984-9
- 

**数据结构基础（C语言版）**  
E. Horowitz等著，朱仲涛译  
7-302-18696-0
- 

**数据结构（C++语言版）**  
殷人昆等  
7-302-14811-1

Data Structures & Algorithms, Tsinghua University

⑤ Data Structures & Algorithms, Tsinghua University

## 基础知识

- ◆ DS涉及多个学科，但并不意味着必须首先逐一精通，常用部分只是其中不大的子集
- ◆ C/C++语言程序设计：类，继承，重载，重写，虚方法，模板
- ◆ 离散数学：集合、偏序集、同序、数学归纳法；  
级数、递归、递推、排列、组合；Stirling逼近
- ◆ 概率论：随机分布、概率、数学期望、期望值的性质等，几何分布
- ◆ ...
- ◆ 关于数学，我们会尽可能使用初等方法

Data Structures & Algorithms, Tsinghua University

⑥ Data Structures & Algorithms, Tsinghua University

## 选修，还是不选修

- ◆ 目标定位：是否需要进阶数据结构？
  - 程序设计语言：编写出合法的程序
  - 数据结构与算法：实现高效处理大规模数据的算法
- ◆ 软件工程：
  - 参与团队编写大规模、复杂、鲁棒和高效的软件
  - 已修C/C++语言程序设计，有一定的编程基础
  - 试做社团项目0次和最后一次PA，进一步自我测试
- ◆ 其他条件：
  - 可否进阶数据结构？
  - 对计算机科学与应用的兴趣
  - 多思考、多动手、多讨论的习惯

⑦ Data Structures & Algorithms, Tsinghua University

## 平台 + 资源

- MOOC：<http://xuetangX.com> + <https://EDX.org>
- 网络学堂：<http://LEARN.tsinghua.edu.cn>
- 讨论交流：<https://piazza.com>
- 观察阅读：<http://DSA.cs.tsinghua.edu.cn/~deng/DS>
- 实践训练：<http://DSA.cs.tsinghua.edu.cn/QJ>

⑧ Data Structures & Algorithms, Tsinghua University

⑨ Data Structures & Algorithms, Tsinghua University

# 目 录

01. 绪论		02. 向量	
A. 计算			
1	01. 工具	99	02. 最长公共子序列
4	02. 算法	105	XB. 局限
8	03. 优劣	111	01. 缓存
B. 计算模型			
12	01. 统一尺度	113	02. 字宽
16	02. 图灵机	122	03. 随机数
20	03. RAM	XC. 下界	
C. 渐进分析			
26	01. 大 O 记号	113	01. 代数判定树
32	02. 多项式	122	02. 归约
37	03. 指数	02. 向量	
43	04. 复杂度层次	122	A. 抽象数据类型
D. 算法分析			
48	01. 级数	127	01. 接口与实现
52	02. 迭代	131	02. 从数组到向量
56	03. 正确性	135	03. 模板类
62	04. 封底估算	135	B. 可扩充向量
E. 迭代与递归			
66	01. 减而治之	139	01. 算法
72	02. 分而治之	144	02. 分摊
77	03. Max2	149	C. 无序向量
82	04. 尾递归	154	01. 基本操作
XA. 动态规划			
86	01. 记忆化	158	02. 查找
		163	03. 去重
		166	04. 遍历
		173	D. 有序向量
		183	01. 唯一化
			02. 二分查找 ( A )
			03. Fib 查找

189	04. 二分查找 (B)		<b>04. 栈与队列</b>
195	05. 二分查找 (C)	303	A. 栈 ADT 及实现
200	06. 插值查找		B. 调用栈
206	E. 起泡排序	307	01. 原理与算法
	F. 归并排序	312	02. 实例
211	01. 分而治之	316	03. 消除递归
214	02. 二路归并	319	C. 进制转换
220	03. 复杂度	325	D. 括号匹配
		333	E. 栈混淆
			F. 中缀表达式求值
223	A. 循秩访问	341	01. 问题
228	B. 接口与实现	344	02. 构思
	C. 无序列表	348	03. 算法
234	01. 循秩访问	353	04. 实例
236	02. 查找		G. 逆波兰表达式
239	03. 插入	357	01. 定义
241	04. 基于复制的构造	359	02. 求值
243	05. 删除	364	03. 转换
245	06. 析构	368	04. PostScript
247	07. 去重	371	H. 队列 ADT 及实现
249	08. 遍历	375	I. 队列应用
	D. 有序列表	379	XA. Steap + Queap
251	01. 唯一化	384	XB. 试探回溯法：八皇后
253	02. 查找	395	XC. 试探回溯法：迷宫寻径
256	E. 选择排序		<b>05. 二叉树</b>
264	F. 循环节		A. 树
270	G. 插入排序	400	B. 树的表示
278	H. 归并排序	408	C. 有根有序树 = 二叉树
281	I. 逆序对	415	D. 二叉树的实现
285	XA. 游标实现	421	E. 先序遍历
291	XB. Java 序列	430	01. 遍历
298	XC. Python 列表	432	02. 迭代算法 A

439	03. 观察	542	05. 性能分析
442	04. 迭代算法 B	545	C. 邻接表
	F. 中序遍历		D. 广度优先搜索
446	01. 观察	552	01. 算法
451	02. 迭代算法	556	02. 实例
454	03. 实例	559	03. 推广
456	04. 分析	563	04. 性质
459	05. 后继与前驱		E. 深度优先搜索
	G. 后序遍历	569	01. 算法
462	01. 观察	573	02. 实例 (无向图)
467	02. 迭代算法	578	03. 推广
470	03. 实例	581	04. 实例 (有向图)
473	04. 分析	587	05. 性质
476	05. 表达式树		F. 拓扑排序
	H. 层次遍历	592	01. 零入度算法
479	01. 算法	598	02. 零出度算法
483	02. 分析	602	G. 优先级搜索
486	03. 完全二叉树		H. Prim 算法
490	I. 重构	607	01. 最小支撑树
	J. Huffman 树	613	02. 极短跨边
493	01. PFC 编码	617	03. 实例
500	02. 算法	621	04. 正确性
502	03. 正确性	624	05. 实现
512	04. 实现		I. Dijkstra 算法
517	05. 改进	628	01. 最短路径
		632	02. 最短路径树
	<b>06. 图</b>	636	03. 算法
520	A. 概述	643	04. 实例
	B. 邻接矩阵	647	05. 实现
525	01. 构思		X. 双连通分量
529	02. 实现	651	01. 关节点
533	03. 简单接口	653	02. 判定准则
537	04. 复杂接口	658	03. 算法

662	04. 实例	B. B-树
667	05. 复杂度	767      01. 大数据
	XB. Kruskal 算法	772      02. 结构
669	01. 算法	781      03. 查找
673	02. 实现	789      04. 插入
677	03. Union-Find	798      05. 删除
685	XC. Floyd-Warshall 算法	C. 红黑树
<b>07. 二叉搜索树</b>		
	A. 概述	810      01. 一致性
692	01. 循关键码访问	815      02. 结构
696	02. 中序	821      03. 插入
700	03. 接口	833      04. 删除
	B. 算法及实现	D. 范围查询
703	01. 查找	848      01. 一维范围查询
707	02. 插入	851      02. 蛮力算法
710	03. 删除	854      03. 二分查找
	C. 平衡	857      04. 输出敏感
716	01. 期望树高	859      05. 平面范围查询
721	02. 理想平衡与适度平衡	E. 一维 kd-树
724	03. 等价变换	866      01. 结构
	D. AVL 树	868      02. 查询
728	01. 适度平衡	871      03. 复杂度
731	02. 重平衡	F. 二维 kd-树
734	03. 插入	875      01. 结构
738	04. 删除	879      02. 构造
742	05. (3+4)-重构	882      03. 正则子集
<b>08. 高级搜索树</b>		
	A. 伸展树	885      04. 查询
747	01. 逐层伸展	891      05. 优化
753	02. 双层伸展	893      06. 复杂度
759	03. 算法实现	898      07. 高维
		901      08. 四叉树
		XA. 多层搜索树
		903      01. x-查询 + y-查询
		906      02. 最坏情况

908	03. x-查询 * y-查询		C. 排解冲突
911	04. 查询	1017	01. 开放散列
915	05. 复杂度	1021	02. 封闭散列
	XB. 范围树	1026	03. 懒惰删除
921	01. Y-列表	1029	04. 平方试探
924	02. 相关性	1034	05. 双向平方试探
926	03. 构思	1039	06. 再散列
929	04. 分散层叠	1041	07. 重散列
933	05. 复杂度		D. 桶排序
	XC. 区间树	1043	01. 算法
937	01. 穿刺查询	1047	02. 最大缝隙
940	02. 构造		E. 基数排序
946	03. 复杂度 (1)	1052	01. 算法
950	04. 查询	1055	02. 分析
953	05. 复杂度 (2)	1059	03. 整数排序
	XD. 线段树	1062	F. 计数排序
956	01. 离散化		XA. 跳转表
961	02. 二叉搜索树	1071	01. 结构
964	03. 最坏情况	1078	02. 查找
967	04. 公共祖先	1083	03. 插入
970	05. 正则子集	1087	04. 删除
972	06. $O(n \log n)$ 空间		XB. 位图
974	07. 构造	1090	01. 结构
977	08. 查询	1094	02. 应用
		1102	03. 快速初始化
		1105	XC. MD5

## 09. 词典

	A. 散列
981	01. 循值访问
991	02. 原理
999	03. 冲突
	B. 散列函数
1003	01. 基本
1011	02. 更多

## 10. 优先级队列

	A. 概述
1112	01. 需求与动机
1118	02. 基本实现
	B. 完全二叉堆
1125	01. 结构

1130	02. 插入	1267	02. 构造 GS[]表
1135	03. 删除	1271	03. 性能
1140	04. 批量建堆		F. KR 算法
1147	C. 堆排序	1274	01. 串即是数
	D. 锦标赛排序	1281	02. 散列
1155	01. 锦标赛树	1286	G. 键树
1161	02. 败者树		
1164	XA. 多叉堆		<b>12. 排序</b>
	XB. 左式堆		A. 快速排序
1172	01. 结构	1291	01. 算法 A
1178	02. 合并	1301	02. 性能分析 (1)
1186	03. 插入 + 删除	1304	03. 性能分析 (2)
1191	XC. 优先级搜索树	1309	04. 性能分析 (3)
		1314	05. 重复元素
		1320	06. 变种
	<b>11. 串</b>		B. 选取
1195	A. ADT		01. 众数
	B. 模式匹配	1325	02. 中位数
1200	01. 问题描述	1331	03. QuickSelect
1204	02. 蛮力算法	1337	04. LinearSelect
	C. KMP 算法	1345	C. 希尔排序
1210	01. 记忆法		01. 框架+实例
1215	02. 查询表	1350	02. 输入敏感性
1224	03. 理解 next[]表	1357	03. Shell 序列
1228	04. 构造 next[]表	1361	04. 逆序对
1234	05. 分摊分析	1363	05. PS 序列
1238	06. 再改进	1371	06. Pratt 序列
	D. BM 算法 : BC 策略	1376	07. Sedgewick 序列
1244	01. 以终为始	1379	
1250	02. 坏字符		
1256	03. 构造 BC[]表		
1258	04. 性能		
	E. BM 算法 : GS 策略		
1261	01. 好后缀		

· 对象：规律、技巧

## 1. 绪论

目标：高效、低耗

计算  
工具

邓俊辉

灭规矩而不用兮，背绳墨之正方

deng@tsinghua.edu.cn

## 绳索计算机及其算法

❖ 输入：任给直线 $\ell$ 及其上一点A

输出：经过A做 $\ell$ 的一条垂线

❖ 算法 ( 2000 B.C. , 古埃及人 )

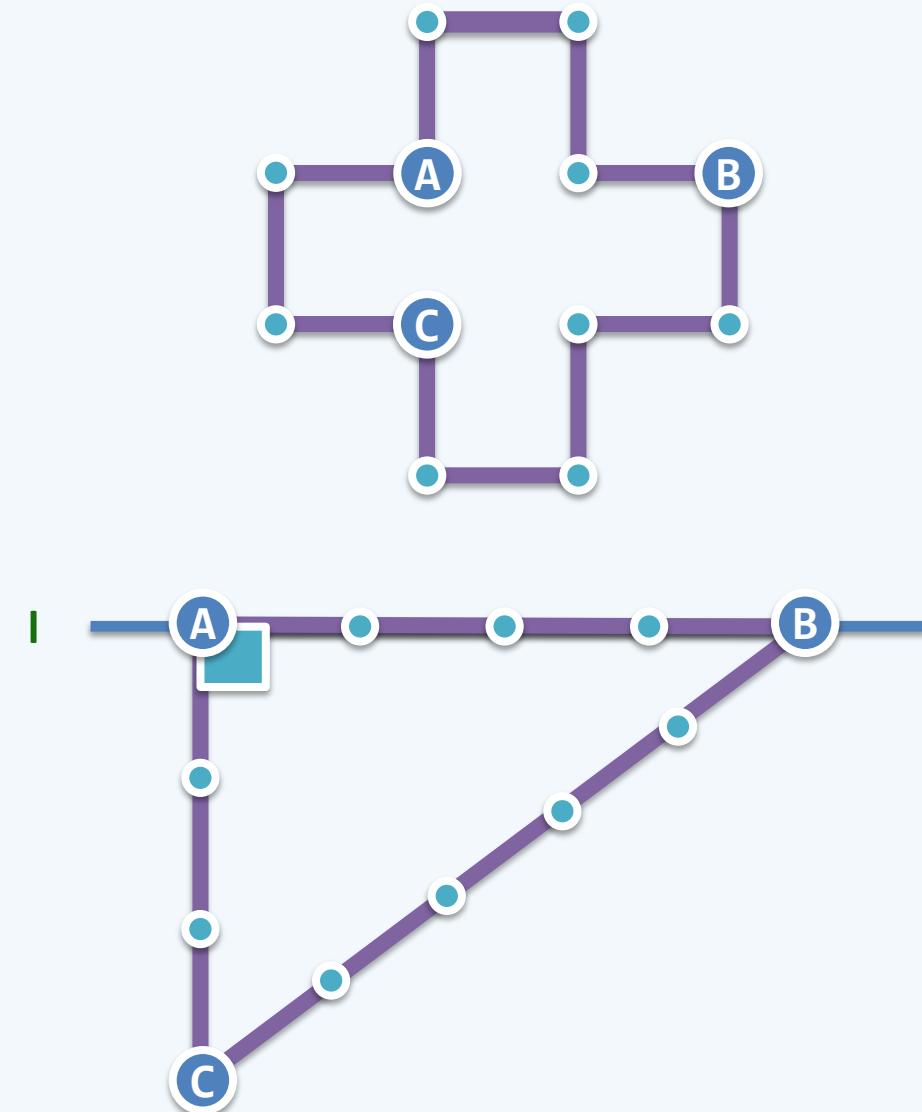
取12段等长的绳索，首尾联接成环

从A点起，将4段绳索沿 $\ell$ 抻直并固定于B

沿另一方向找到第3段绳索的终点C

移动点C，将剩余的 $3 + 5$ 段绳索抻直

❖ 这里的计算机是什么？



## 尺规计算机及其算法

❖ 任给平面上线段AB（输入），将其三等分（输出）

❖ 算法：从A发出一条与AB不重合的射线 $\rho$

在 $\rho$ 上取  $|AC'| = |C'D'| = |D'B'|$

联接 $B'B$

经 $D'$ 做 $B'B$ 的平行线，交 $AB$ 于 $D$

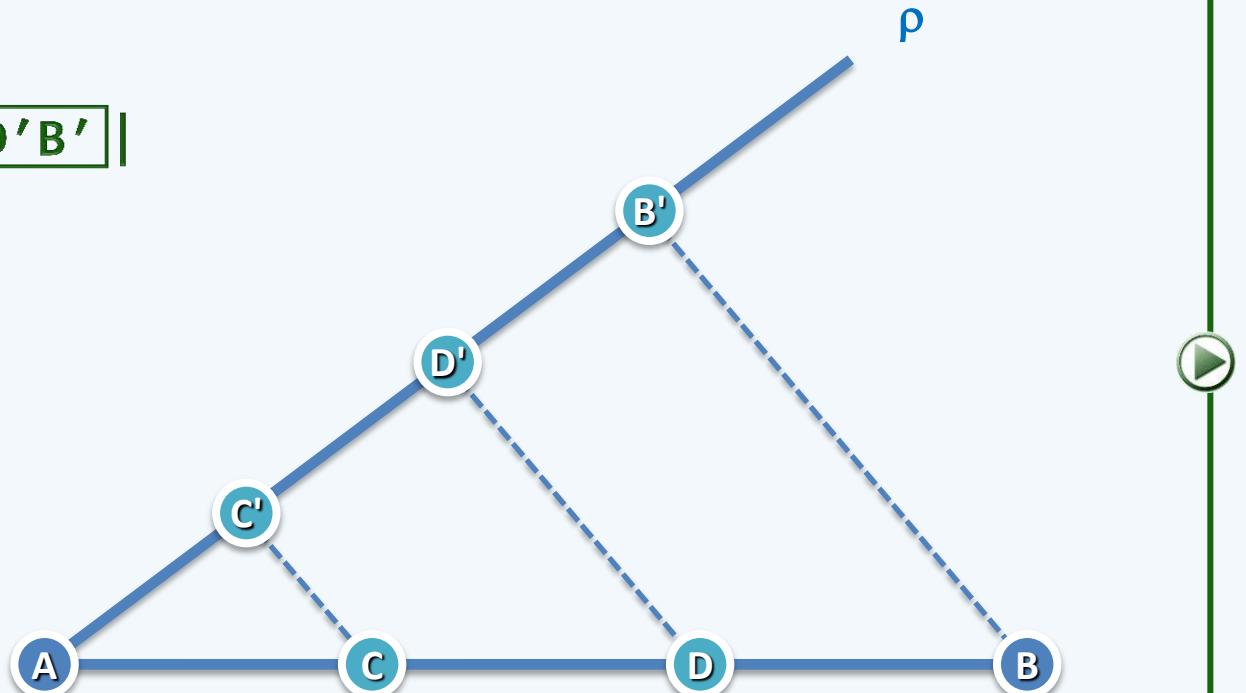
经 $C'$ 做 $B'B$ 的平行线，交 $AB$ 于 $C$

❖ 这里的计算机是什么？

❖ 它能够解决什么问题？

不能解决什么问题？

❖ 子程序：过直线外一点，做平行线



## 1. 绪论

计算  
算法

Computer science should be called  
computing science, for the same reason why  
surgery is not called knife science.

邓俊辉

- E. Dijkstra

deng@tsinghua.edu.cn

# 算法

❖ 计算 = 信息处理

    借助某种工具，遵照一定规则，以明确而机械的形式进行

❖ 计算模型 = 计算机 = 信息处理工具

❖ 所谓算法，即特定计算模型下，旨在解决特定问题的指令序列

    输入 待处理的信息（问题）

    输出 经处理的信息（答案）

    正确性 的确可以解决指定的问题

    确定性 任一算法都可以描述为一个由基本操作组成的序列

    可行性 每一基本操作都可实现，且在常数时间内完成

**有穷性** 对于任何输入，经有穷次基本操作，都可以得到输出

... ...

**有穷性**

❖ 序列  $Hailstone(n) = \begin{cases} \{1\} & n \leq 1 \\ \{n\} \cup Hailstone\left(\frac{n}{2}\right) & n \text{偶} \\ \{n\} \cup Hailstone(3n + 1) & n \text{奇} \end{cases}$

❖  $Hailstone(42) = \{ 42, 21, 64, 32, \dots, 1 \}$

$Hailstone(7) = \{ 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, \dots, 1 \}$

$Hailstone(27) = \{ 27, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, \dots \}$

## 有穷性

❖ int hailstone( int n ) { //计算序列Hailstone(n)的长度

    int length = 1; //从1开始，以下按定义逐步递推，并累计步数，直至n = 1

    while ( 1 < n ) {  $n \% 2$  ?  $n = 3 * n + 1$  :  $n /= 2$ ; length++; }

    return length; //返回|Hailstone(n)|

} //对于任意的n，总有|Hailstone(n)| <  $\infty$  ?

## 1. 绪论

计算  
优劣

邓俊辉

deng@tsinghua.edu.cn

好 = ?

❖ 正确：符合语法，能够编译、链接

能够正确处理简单的输入

能够正确处理大规模的输入

能够正确处理一般性的输入

能够正确处理退化的输入

能够正确处理任意合法的输入

❖ 健壮：能辨别不合法的输入并做适当处理，而不致非正常退出

❖ 可读：结构化 + 准确命名 + 注释 + ...

❖ 效率：速度尽可能快；存储空间尽可能少

Algorithms + Data Structures = Programs

//N. Wirth, 1976

(Algorithms + Data Structures) x Efficiency = Computation

## 为什么要学？学什么？学习目标？

- ❖ **数据结构** 在计算机相关专业课程体系中，一直处于核心位置
  - 是计算机科学的重要组成部分
  - 是设计与实现高效算法的基石
- ❖ **讲授范围** 各类数据结构设计和实现的基本原理与方法
  - 算法设计和分析的主要技巧与工具
- ❖ **学习数据结构**，就是要学会
  - 高效地利用计算机，有效地存储、组织、传递和转换数据
    - 掌握各类数据结构功能、表示、实现和基本操作接口
    - 理解各类（基本）算法与不同数据结构之间的内在联系
    - 了解各类数据结构适用的应用环境
    - 灵活地选用各类（基本）算法及对应的数据结构，解决实际问题

## 内容纵览

### ❖ 各数据结构的ADT接口及其不同实现

序列（向量、列表、栈、队列），树及搜索树（AVL树、伸展树、红黑树、B-树、kd-树）

优先队列（堆），字典（散列表、跳转表），图的算法与应用

### ❖ 构造有效算法模块的常用技巧

顺序和二分查找，选取与排序，遍历

模式匹配，散列，几何查找

### ❖ 算法设计的典型策略与模式

迭代、贪心、递归、分治、减治、试探-剪枝-回溯、动态规划

### ❖ 复杂度分析的基本方法

渐进分析与相关记号

递推关系、递归跟踪

分摊分析、后向分析

## 1. 绪论

计算模型  
统一尺度

测量

To measure is to know.

If you can not measure it,

you can not improve it.

- Lord Kelvin

邓俊辉

deng@tsinghua.edu.cn

## 算法分析

- ❖ 两个主要方面 **正确** : 算法功能与问题要求一致?  
                          数学证明? 可不那么简单...
- 成本** : 运行时间 + 所需存储空间  
                          如何度量? 如何比较?
- ❖ 考察:  $T_A(P)$  = 算法A求解问题实例P的计算成本  
                          意义不大, 毕竟... 可能出现的问题实例太多  
                          如何归纳概括?
- ❖ 观察: 问题实例的**规模**, 往往是决定计算成本的主要因素
- ❖ 通常: 规模接近, 计算成本也接近  
                          规模扩大, 计算成本亦上升

## 特定算法 + 不同实例

❖ 令： $T_A(n)$  = 用算法A求解某一问题规模为n的实例，所需的计算成本  
讨论特定算法A（及其对应的问题）时，可简记作 $T(n)$

❖ 然而：这一定义仍有问题...

❖ 观察：同一问题等规模的不同实例，计算成本不尽相同，甚至有实质差别...

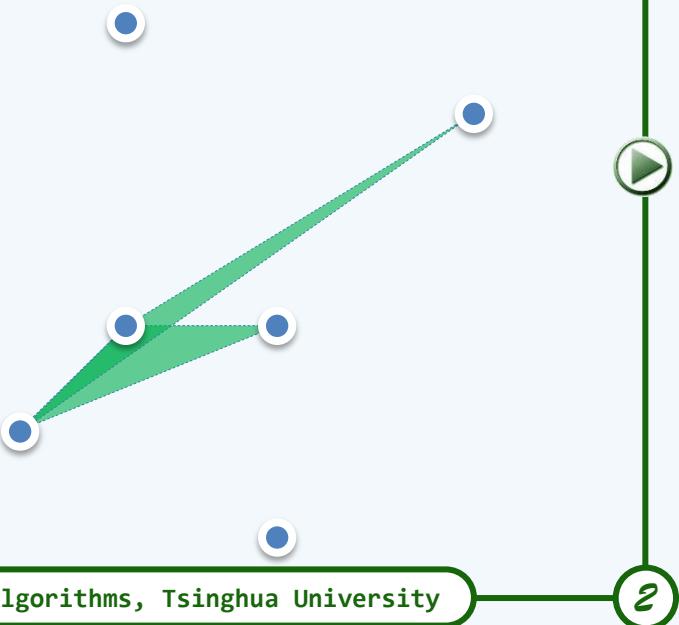
❖ 例如：任给平面上n个点，在它们定义的 $\binom{n}{3}$ 个三角形中，是否有某个的面积不超过5.0？

❖ 蛮力：最坏情况下需枚举所有三角形；但运气好的话...

❖ 既然如此，又该如何定义 $T(n)$ 呢？

❖ 稳妥起见，取 $T(n) = \max \{ T(P) \mid |P| = n \}$

亦即，在规模同为n的所有实例中，只关注**最坏**（成本最高）者



## 特定问题 + 不同算法

- ❖ 同一问题通常有多种算法，如何评判其优劣？
- ❖ 实验统计是最直接的方法，但足以准确反映算法的真正效率？不足够！

不同的算法，可能更适应于不同规模的输入

不同的算法，可能更适应于不同类型的输入

同一算法，可能由不同程序员、用不同程序语言、经不同编译器实现

同一算法，可能实现并运行于不同的体系结构、操作系统...

- ❖ 为给出客观的评判，需要抽象出一个理想的平台或模型

不再依赖于上述种种具体的因素

从而直接而准确地描述、测量并评价算法

## 1. 绪论

计算模型

图灵机

Sometimes it is the people  
no one can imagine anything of who  
do the things no one can imagine.

- A. Turing

邓俊辉

deng@tsinghua.edu.cn

## Turing Machine

❖ Tape 依次均匀地划分为单元格

各注有某一字符，默认为'#'



❖ Alphabet 字符的种类有限

❖ Head 总是对准某一单元格，并可读取和改写其中的字符

每经过一个节拍，可转向左侧或右侧的邻格

❖ State TM总是处于有限种状态中的某一种

每经过一个节拍，可（按照规则）转向另一种状态

## 转换函数

### ❖ Transition Function



❖ 若当前状态为  $q$ ，且当前字符为  $c$ ，则

将当前字符  $c$  改写为  $d$

转向左/右侧邻格

转入 ' $p$ ' 状态

❖ 特别地，一旦转入约定的状态 ' $h$ '，则停机

## Increase

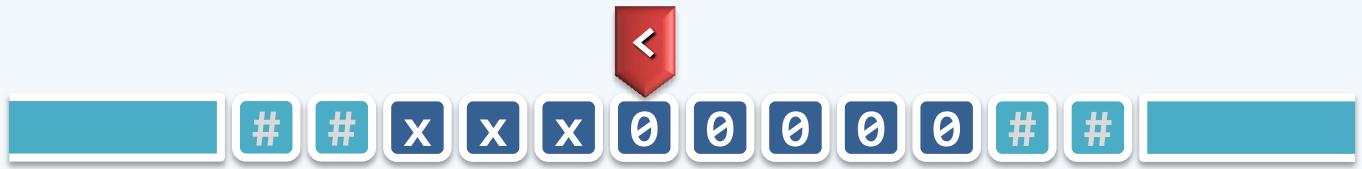
❖ 功能：

将二进制非负整数加一



❖ 原理：

全'1'的后缀，翻转为全'0'



原最低位的'0'或'#'翻转为'1'

❖ (<, 1, 0, L, <) //左行，1->0



(<, 0, 1, R, >) //掉头，0->1

(<, #, 1, R, >) // ?



(>, 0, 0, R, >) //右行

(>, #, #, L, h) //复位



❖ 规范 ~ 接口

## 1. 绪论

计算模型

RAM

There is an infinite set A  
that is not too big.  
- J. von Neumann

邓俊辉

deng@tsinghua.edu.cn

## Random Access Machine

- ❖ 寄存器顺序编号，总数没有限制：  $R[0], R[1], R[2], R[3], \dots$  //但愿如此
- ❖ 通过编号，可以直接访问任意寄存器 //call-by-rank
- ❖ 每一基本操作仅需常数时间 //循环及子程序本身非基本操作

$R[i] \leftarrow c$

$R[i] \leftarrow R[R[j]]$

$R[i] \leftarrow R[j] + R[k]$

$R[i] \leftarrow R[j]$

$R[R[i]] \leftarrow R[j]$

$R[i] \leftarrow R[j] - R[k]$

IF  $R[i] = 0$  GOTO 1

IF  $R[i] > 0$  GOTO 1

GOTO 1

STOP

## Random Access Machine

- ❖ 与TM模型一样，RAM模型也是一般计算工具的简化与抽象  
使我们可以独立于具体的平台，对算法的效率做出可信的比较与评判
- ❖ 在这些模型中
  - 算法的运行时间  $\propto$  算法需要执行的基本操作次数
  - $T(n) =$  算法为求解规模为 $n$ 的问题，所需执行的基本操作次数

# Floor Division

◆ 对任意  $\theta \leq c$  和  $\theta < d$ , 计算

$$\lfloor \frac{c}{d} \rfloor = \max \{ x \mid d \cdot x \leq c \}$$

$$= \max \{ x \mid d \cdot x < 1 + c \}$$

❖ 例如： $\lfloor 5/2 \rfloor = 2$        $\lfloor 2015/56 \rfloor = 35$

$\lfloor 6/3 \rfloor = 2$        $\lfloor 2016/36 \rfloor = 56$

$\lfloor 12/5 \rfloor = 2$

◆ 思路：反复地从 $R[0] = 1 + c$ 中，减去 $R[1] = d$   
统计在下溢之前，所做减法的次数x

## Floor Division

### ❖ RAM 算法

```

0   R[3] <- 1           //increment

1   R[0] <- R[0] + R[3] //c++

2   R[0] <- R[0] - R[1] //c -= d

3   R[2] <- R[2] + R[3] //x++

4   IF R[0] > 0 GOTO 2  //if c > 0 goto 2

5   R[0] <- R[2] - R[3] //else x-- and

6   STOP    //return R[0] = x = ⌊c/d⌋
  
```

Step	IR	R[0]	R[1]	R[2]	R[3]
0	0	12	5	0	0
1	1	^	^	^	1
2	2	13	^	^	^
3	3	8	^	^	^
4	4	^	^	1	^
5	2	^	^	^	^
6	3	3	^	^	^
7	4	^	^	2	^
8	2	^	^	^	^
9	3	0	^	^	^
10	4	^	^	3	^
11	5	^	^	^	^
12	6	2	^	^	^

## 课后

- ❖ 举例说明：随着问题实例规模增大，同一算法的求解时间可能波动甚至下降
- ❖ 在哪些方面，现代电子计算机仍未达到RAM模型的要求？
- ❖ 在TM、RAM等模型中衡量算法效率，为何通常只需考察运行时间？
- ❖ 图灵机算法Increase中，以下这条指令可否省略：  
 $\langle 0, \#, 1, R, 1 \rangle$
- ❖ 设计一个图灵机，实现对正整数的减一（Decrease）功能

好读书不求甚解  
每有会意，便欣然忘食  
— 陶渊明

## 1. 绪论

渐进分析  
大O记号

Mathematics is more in need  
of good notations than  
of new theorems.

- A. Turing

邓俊辉

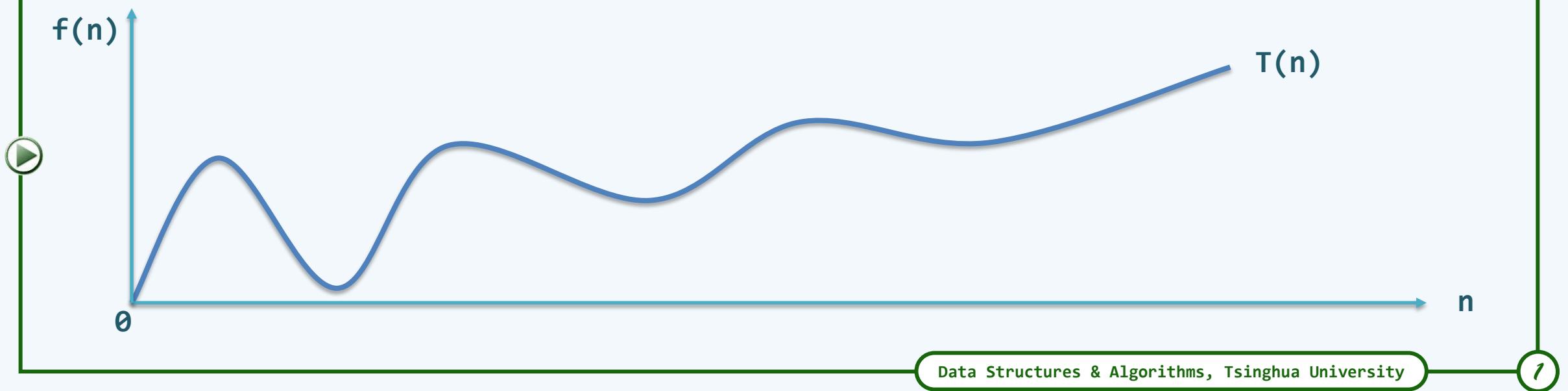
deng@tsinghua.edu.cn

## 渐进分析

❖ 回到原先的问题：随着问题规模的增长，计算成本如何增长？

注意：这里更关心**足够大**的问题，注重考察成本的增长趋势

❖ 在问题规模足够大后，计算成本如何增长？



## 漸進分析

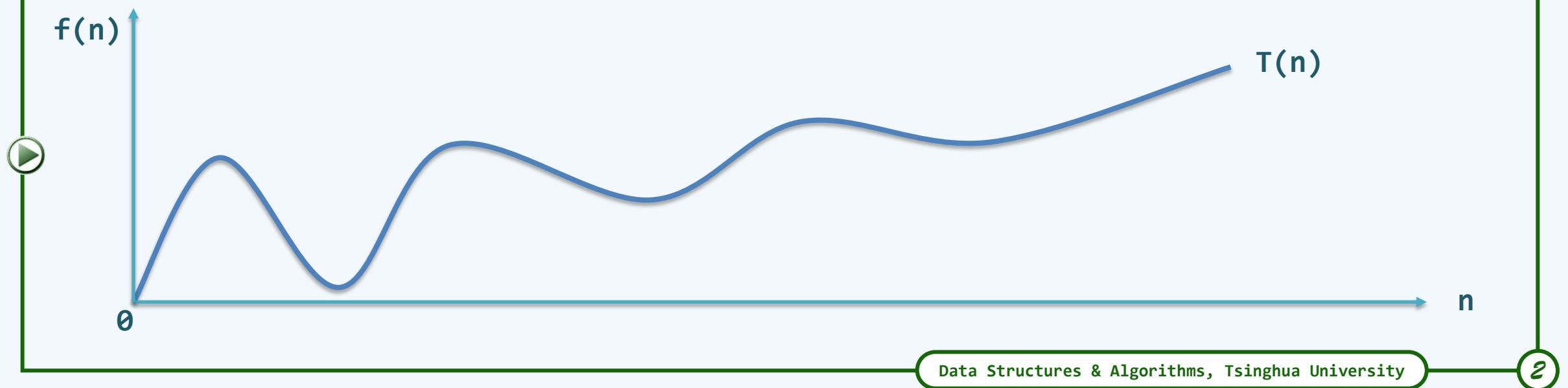
### ❖ 【Asymptotic analysis】

当  $n \gg 2$  后，对于规模为  $n$  输入，算法

需执行的基本操作次数： $T(n) = ?$

需占用的存储单元数： $S(n) = ?$

//通常可不考虑，为什么？



# 大O记号

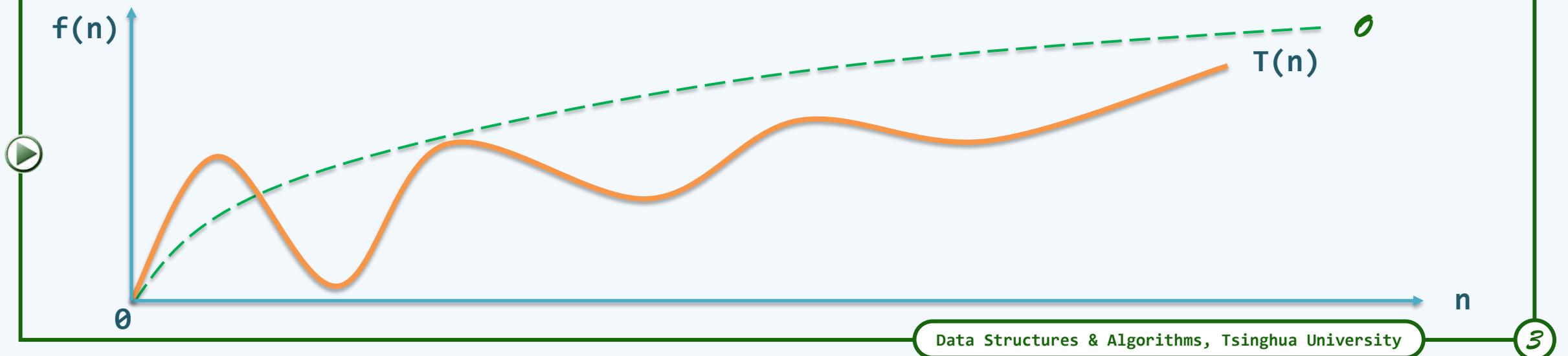
## ❖ big- $\Theta$ notation

//Paul Bachmann, 1894

$$T(n) = \mathcal{O}(f(n)) \quad \text{iff} \quad \exists c > 0 \quad s.t. \quad T(n) < c \cdot f(n) \quad \forall n \gg 2$$

$$Ex: \sqrt{5n \cdot [3n \cdot (n+2) + 4] + 6} < \sqrt{5n \cdot [\boxed{6n^2} + 4] + 6} < \sqrt{\boxed{35n^3} + 6} < 6 \cdot n^{1.5} = \mathcal{O}(n^{1.5})$$

$\cancel{n}$        $\cancel{n^2}$        $\cancel{n^3}$

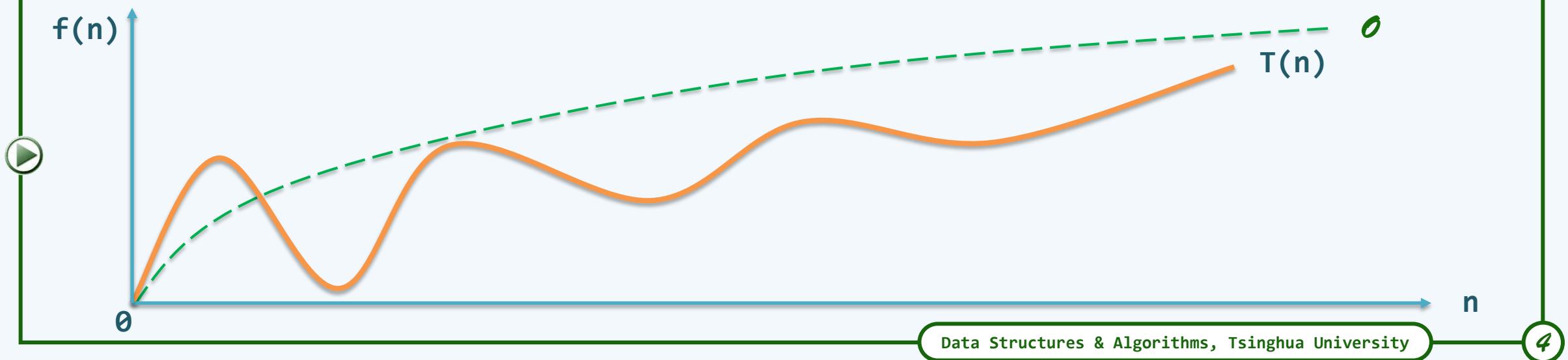


## 大O记号

❖ 与  $T(n)$  相比， $f(n)$  在形式上更为简洁，但依然反映前者的增长趋势

常系数可忽略： $\mathcal{O}(f(n)) = \mathcal{O}(c \cdot f(n))$

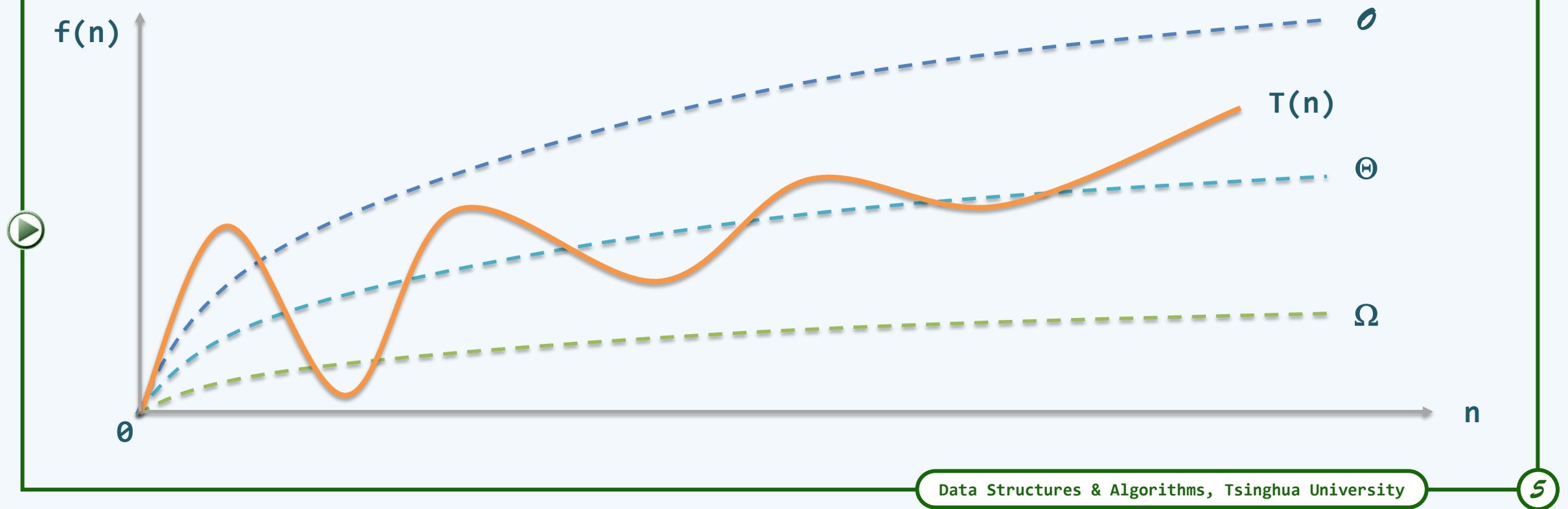
低次项可忽略： $\mathcal{O}(n^a + n^b) = \mathcal{O}(n^a), a \geq b > 0$



## 其它记号

$$T(n) = \Omega(f(n)) \quad \text{iff} \quad \exists c > 0 \quad \text{s.t.} \quad T(n) > c \cdot f(n) \quad \forall n \gg 2$$

$$T(n) = \Theta(f(n)) \quad \text{iff} \quad \exists c_1 > c_2 > 0 \quad \text{s.t.} \quad c_1 \cdot f(n) > T(n) > c_2 \cdot f(n) \quad \forall n \gg 2$$



## 1. 绪论

渐进分析

多项式

Computational problems can be feasibly  
computed on some computational device  
only if they can be computed in  
polynomial time.

邓俊辉

- A. Cobham & J. Edmonds

deng@tsinghua.edu.cn

$\mathcal{O}(1)$ 

## ❖ 常数 ( constant function )

//含RAM各基本操作

- $2 = 2015 = 2015 \times 2015 = \mathcal{O}(1)$ , 甚至
- $2015^{2015} = \mathcal{O}(1)$

## ❖ 渐进而言，再大的常数，也要小于递增的变数

## ❖ [General twin prime conjecture, de Polignac 1849]

For every natural number  $k$ , there are infinitely many prime pairs  $p$  and  $q$  such that  $p - q = 2k$

❖ [Yitang Zhang, April 2013]  $k \leq 35,000,000$

❖ [Terence Tao, May 2013]  $k \leq 6,500,000$

❖ [Polymath Project, April 2014]  $k \leq 123$

$\Theta(1)$ 

❖ 这类算法的效率最高

//总不能奢望不劳而获吧

❖ 什么样的代码段对应于常数执行时间？

//应具体分析

一定不含循环？

```
for ( i = 0; i < n; i += n/2015 + 1 );
```

```
for ( i = 1; i < n; i = 1 << i );
```

// $\log^* n$ , 几乎常数

一定不含分支转向？

```
if ( (n + m) * (n + m) < 4 * n * m ) goto UNREACHABLE; //不考虑溢出
```

一定不能有（递归）调用？

```
if ( 2 == (n * n) % 5 ) 01(n);
```

...

$\Theta(\log^c n)$ ❖ 对数  $\Theta(\log n)$ 

// 为何不注明底数？

$$\ln n \mid \lg n \mid \log_{100} n \mid \log_{2015} n$$

❖ 常底数无所谓

$$\forall a, b > 0, \log_a n = \boxed{\log_a b} \cdot \log_b n = \Theta(\log_b n)$$

❖ 常数次幂无所谓

$$\forall c > 0, \log n^c = c \cdot \log n = \Theta(\log n)$$

❖ 对数多项式 ( poly-log function )

$$123 * \log^{321} n + \log^{105}(n^2 - n + 1) = \Theta(\log^{321} n)$$

❖ 这类算法非常有效，复杂度无限接近于常数

$$\forall c > 0, \log n = \Theta(n^c)$$

$\mathcal{O}(n^c)$ 

## ❖ 多项式 ( polynomial function )

$$100n + 200 = \mathcal{O}(n)$$

$$(100n - 500)(20n^2 - 300n + 2015) = \mathcal{O}(n \times n^2) = \mathcal{O}(n^3)$$

$$(2015n^2 - 20)/(1999n - 1) = \mathcal{O}(n^2/n) = \mathcal{O}(n)$$

一般地：  $a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0 = \mathcal{O}(n^k), a_k > 0$

❖ 线性 ( linear function ) : 所有  $\mathcal{O}(n)$  类函数❖ 从  $\mathcal{O}(n)$  到  $\mathcal{O}(n^2)$  : 编程习题主要覆盖的范围

❖ 幂 :  $[ (n^{2015} - 24n^{2009})^{1/3} + 512n^{567} - 1978n^{123} ]^{1/11} = \mathcal{O}(n^{61})$

❖ 这类算法的效率通常认为已可令人满意，然而...

这个标准是否太低了？

// P难度！

## 1. 绪论

渐进分析

指数

If  $P = NP$  is proved . . . , mathematics  
would be transformed, because computers  
could find a formal proof of any theorem  
which has a proof of reasonable length.

邓俊辉

- S. Cook

deng@tsinghua.edu.cn

$\mathcal{O}(2^n)$ 

❖ 指数 ( exponential function ) :  $T(n) = a^n$

❖  $\forall c > 1, n^c = \mathcal{O}(2^n)$

//  $e^n = 1 + n + n^2/2! + n^3/3! + n^4/4! + \dots$

$n^{1000} = \mathcal{O}(1.0000001^n) = \mathcal{O}(2^n)$

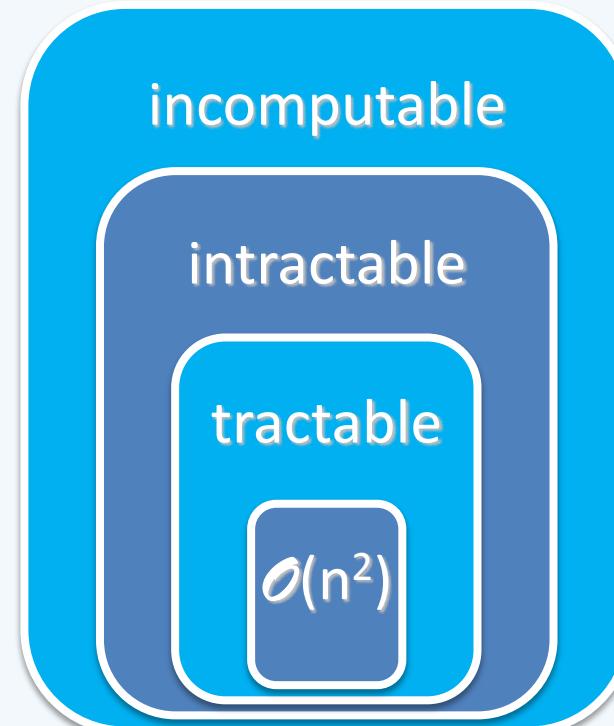
$1.0000001^n = \Omega(n^{1000})$

❖ 这类算法的计算成本增长极快

通常被认为不可忍受

❖ 从  $\mathcal{O}(n^c)$  到  $\mathcal{O}(2^n)$

是从 **有效算法** 到 **无效算法** 的分水岭



$\mathcal{O}(2^n)$ 

❖ 很多问题的 $\mathcal{O}(2^n)$ 算法往往显而易见

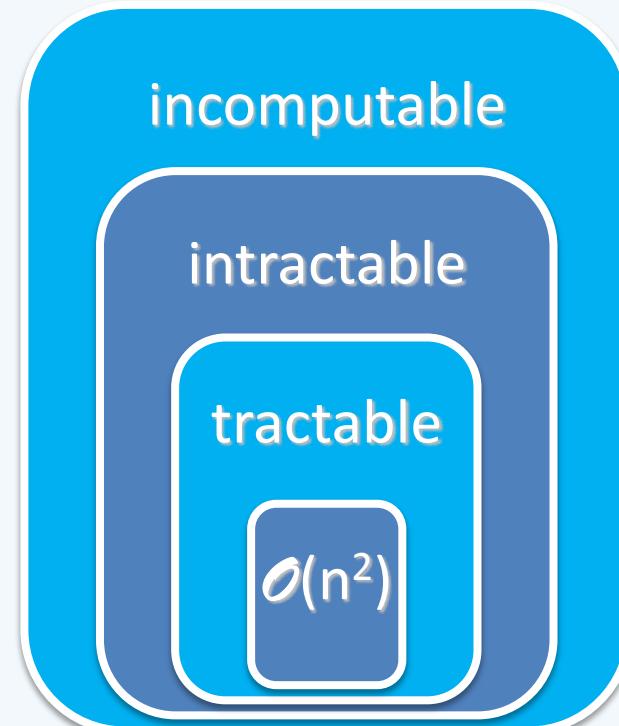
然而，设计出 $\mathcal{O}(n^c)$ 算法却极其不易

甚至，有时注定地只能是徒劳无功

❖ 更糟糕的是

这类问题要远比我们想象的多得多

...



## 2-Subset

## ❖ 【问题描述】

s包含n个正整数， $\sum s = 2m$

**S是否有子集T，满足 $\sum T = m$ ？**

## ❖ 【选举人制】

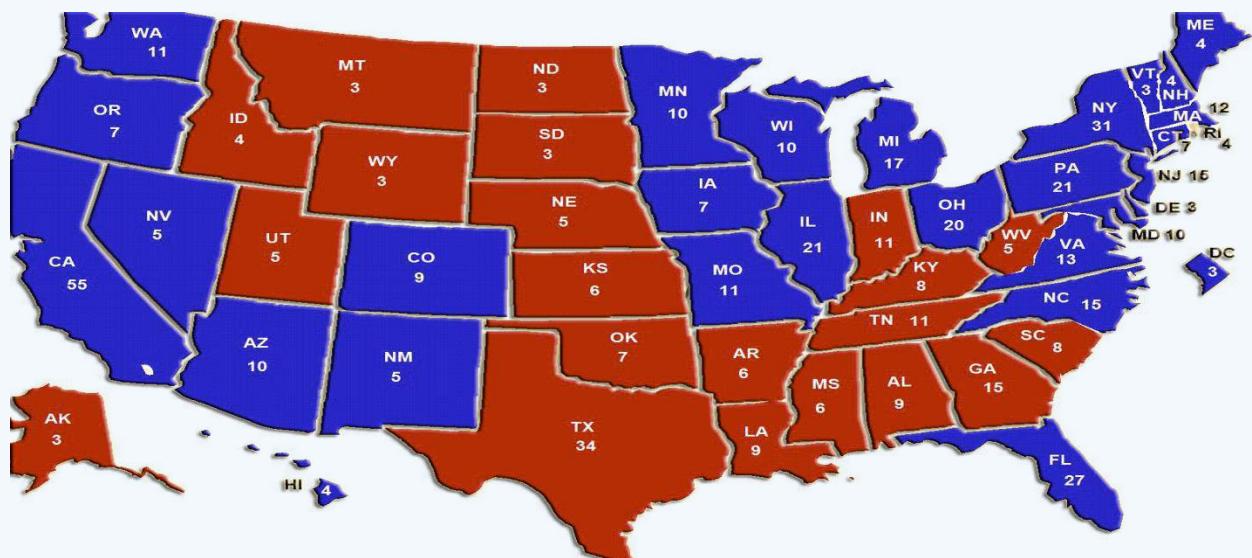
## 各州议会选出的选举人团投票

## 而不是由选民直接投票

**50个州加1个特区，共538票**

获 270 张选举人票，即可当选

55	California	11	Indiana	7	Connecticut	4	Idaho
34	Texas	11	Missouri	7	Iowa	4	Maine
31	New York	11	Tennessee	7	Oklahoma	4	New Hampshire
27	Florida	11	Washington	7	Oregon	4	Rhode Island
21	Illinois	10	Arizona	6	Arkansas	3	Alaska
21	Pennsylvania	10	Maryland	6	Kansas	3	Delaware
20	Ohio	10	Minnesota	6	Mississippi	3	D. C.
17	Michigan	10	Wisconsin	5	Nebraska	3	Montana
15	Georgia	9	Alabama	5	Nevada	3	North Dakota
15	New Jersey	9	Colorado	5	New Mexico	3	South Dakota
15	North Carolina	9	Louisiana	5	Utah	3	Vermont
13	Virginia	8	Kentucky	5	West Virginia	3	Wyoming
12	Massachusetts	8	South Carolina	4	Hawaii		538 = Σ



# 2-Subset

## ❖ 但是...

## ❖ 若共有两位候选人

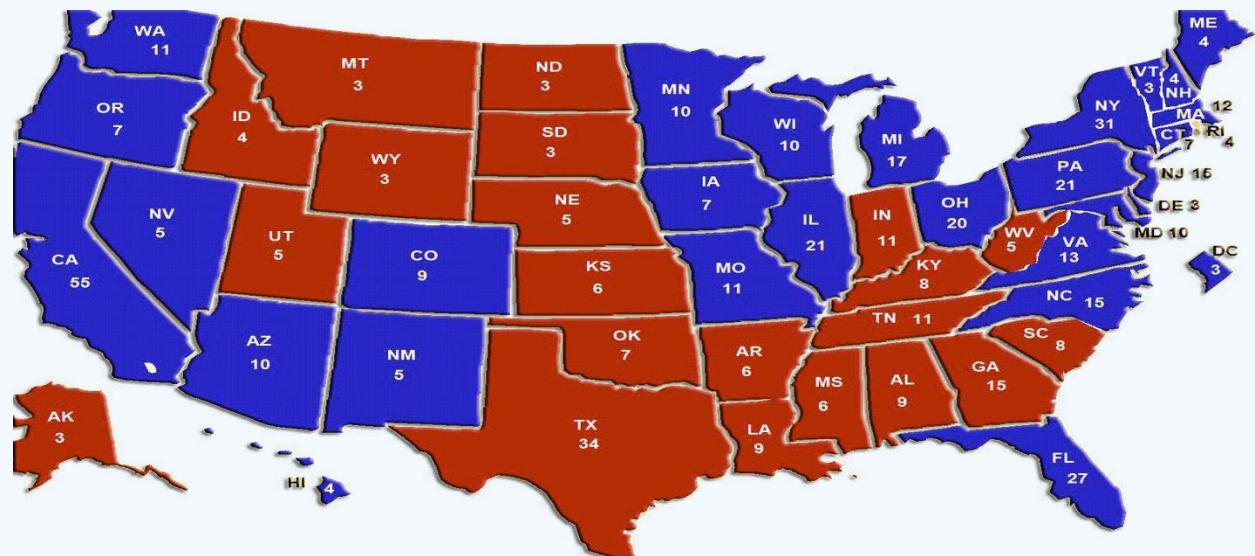
## 是否可能恰好各得 269 票？

## ❖ 【直觉算法】

## 逐一枚举S的每一子集

并统计其中元素的总和

55	California	11	Indiana	7	Connecticut	4	Idaho
34	Texas	11	Missouri	7	Iowa	4	Maine
31	New York	11	Tennessee	7	Oklahoma	4	New Hampshire
27	Florida	11	Washington	7	Oregon	4	Rhode Island
21	Illinois	10	Arizona	6	Arkansas	3	Alaska
21	Pennsylvania	10	Maryland	6	Kansas	3	Delaware
20	Ohio	10	Minnesota	6	Mississippi	3	D. C.
17	Michigan	10	Wisconsin	5	Nebraska	3	Montana
15	Georgia	9	Alabama	5	Nevada	3	North Dakota
15	New Jersey	9	Colorado	5	New Mexico	3	South Dakota
15	North Carolina	9	Louisiana	5	Utah	3	Vermont
13	Virginia	8	Kentucky	5	West Virginia	3	Wyoming
12	Massachusetts	8	South Carolina	4	Hawaii		538 = $\Sigma$



## 2-Subset

❖ 定理： $|2^S| = 2^{|S|} = 2^n$

❖ 亦即：直觉算法需要迭代 $2^n$ 轮，并（在最坏情况下）至少需要花费这么多的时间

— 不甚理想！

//严格讲，这只是程序，而不是算法

❖ 还是直觉：应该有更好的办法吧？

❖ 定理：2-Subset is NP-complete

— 什么意思？

❖ 意即：就目前的计算模型而言，不存在可在多项式时间内回答此问题的算法

— 就此意义而言，上述的直觉算法已属最优

## 1. 绪论

渐进分析

复杂度层级

好读书，不求甚解

邓俊辉

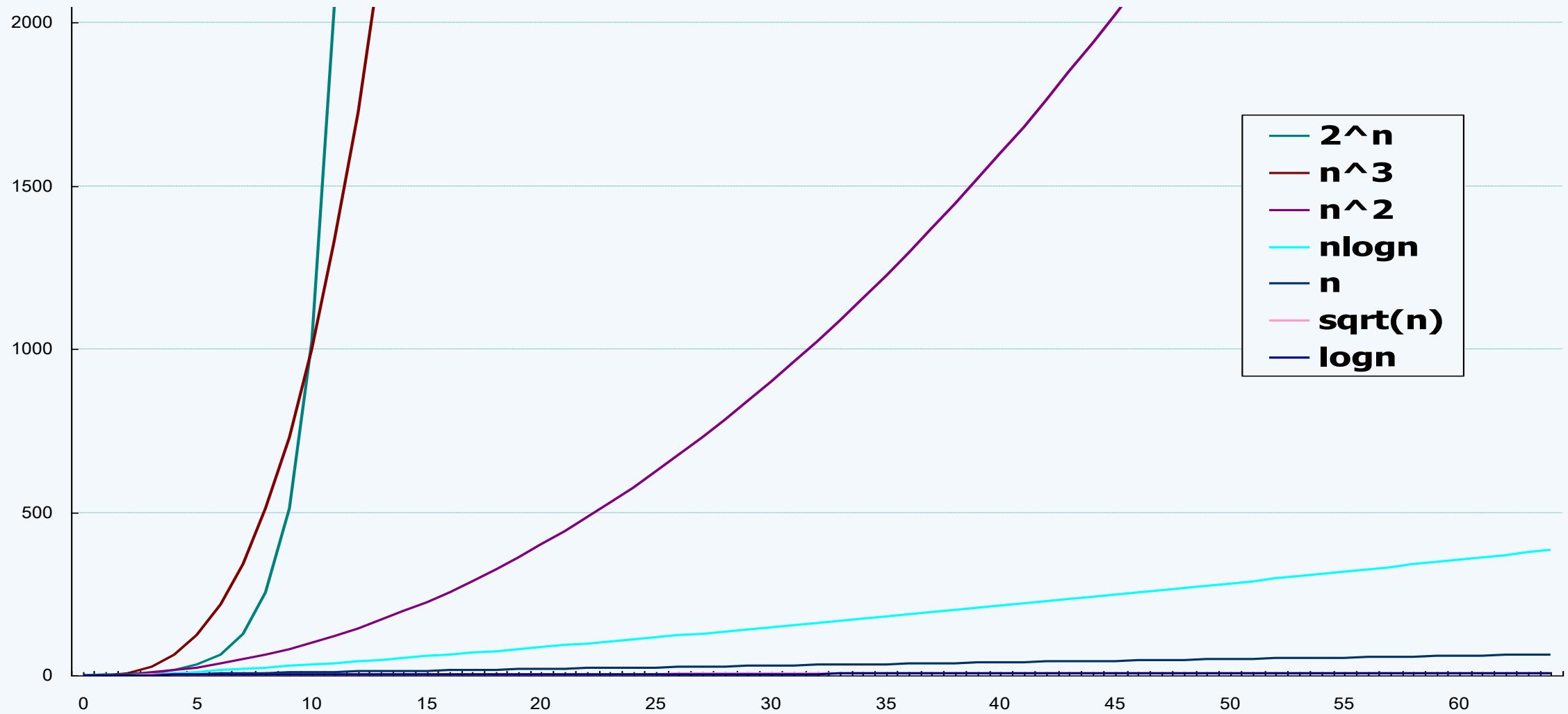
每有会意，便欣然忘食

deng@tsinghua.edu.cn

## 增长速度



## 增长速度



## 层次级别

$\Theta(1)$	常数复杂度	再好不过，但难得如此幸运	对数据结构的基本操作
$\Theta(\log^* n)$		在这个宇宙中，几乎就是常数	
$\Theta(\log n)$	对数复杂度	与常数无限接近，且不难遇到	有序向量的二分查找 堆、词典的查询、插入与删除
$\Theta(n)$	线性复杂度	努力目标，经常遇到	树、图的遍历
$\Theta(n \log^* n)$		几乎几乎几乎接近线性	某些MST算法
$\Theta(n \log \log n)$		几乎接近线性	某些三角剖分算法
$\Theta(n \log n)$		最常出现，但不见得最优	排序、EU、Huffman编码
$\Theta(n^2)$	平方复杂度	所有输入对象两两组合	Dijkstra算法
$\Theta(n^3)$	立方复杂度	不常见	矩阵乘法
$\Theta(n^c)$ , c常数	多项式复杂度	P问题 = 存在多项式算法的问题	
$\Theta(2^n)$	指数复杂度	很多问题的平凡算法，再尽可能优化	
...		绝大多数问题，并不存在算法	

## 课后

❖ 证明、证否或计算： Fibonacci数  $\text{fib}(n) = \Theta(2^n)$

$$12n + 5 = \Theta(n \log n)$$

$$\log^2(n^{1024} - 2*n^6 + 101) = \Theta(?)$$

$$\log^d n = \Theta(n^c), \forall c > 0, d > 1$$

$$\log^{1.001} n = \Theta(\log(n^{1001}))$$

$$(n^2 + 1) / (2n + 3) = \Theta(n)$$

$$n^{2013} = \Theta(n!)$$

$$n! = \Theta(n^{2013})$$

$$2^n = \Theta(n!)$$

❖ k-Subset：任给整数集  $S$ ，判定  $S$  可否划分为  $k$  个不交子集，其和均为  $(\sum S)/k$

证明或证否：  $(k+1)$ -Subset 的难度，不低于  $k$ -Subset

❖ Google: small-o notation

## 1. 绪论

算法分析

级数

邓俊辉

谁校对时间，谁就会突然老去。

deng@tsinghua.edu.cn

## 算法分析

- ❖ 两个主要任务 = 正确性(不变性  $\times$  单调性) + 复杂度
- ❖ 为确定后者，真地需要将算法描述为RAM的基本指令，再统计累计的执行次数？不必！
- ❖ C++等高级语言的**基本指令**，均等效于常数条RAM的**基本指令**；在渐进意义下，二者大体相当
  - 分支转向：`goto` //算法的灵魂；出于结构化考虑，被隐藏了
  - 迭代循环：`for()`、`while()`、... //本质上就是“`if + goto`”
  - 调用 + 递归(自我调用) //本质上也是`goto`
- ❖ 复杂度分析的主要方法
  - 迭代：级数求和
  - 递归：递归跟踪 + 递推方程
  - 猜测 + 验证

## 级数

❖ 算数级数：与末项平方同阶  $T(n) = 1 + 2 + \dots + n = \binom{n+1}{2} = \frac{n(n+1)}{2} = \mathcal{O}(n^2)$

❖ 幂方级数：比幂次高出一阶： $\sum_{k=0}^n k^d \approx \int_0^n x^d dx = \frac{x^{d+1}}{d+1} \Big|_0^n = \frac{n^{d+1}}{d+1} = \mathcal{O}(n^{d+1})$

$$T_2(n) = \sum_{k=1}^n k^2 = 1^2 + 2^2 + 3^2 + \dots + n^2 = n(n+1)(2n+1)/6 = \mathcal{O}(n^3)$$

$$T_3(n) = \sum_{k=1}^n k^3 = 1^3 + 2^3 + 3^3 + \dots + n^3 = n^2(n+1)^2/4 = \mathcal{O}(n^4)$$

$$T_4(n) = \sum_{k=1}^n k^4 = 1^4 + 2^4 + 3^4 + \dots + n^4 = n(n+1)(2n+1)(3n^2+3n-1)/30 = \mathcal{O}(n^5)$$

❖ 几何级数：与末项同阶

$$T_a(n) = \sum_{k=0}^n a^k = a^0 + a^1 + a^2 + a^3 + \dots + a^n = \frac{a^{n+1} - 1}{a - 1} = \mathcal{O}(a^n), \quad 1 < a$$

$$T_2(n) = \sum_{k=0}^n 2^k = 1 + 2 + 4 + 8 + \dots + 2^n = 2^{n+1} - 1 = \mathcal{O}(2^{n+1}) = \mathcal{O}(2^n)$$

## 级数

### ◆ 收敛级数

$$\sum_{k=2}^n \frac{1}{(k-1) \cdot k} = \frac{1}{1 \cdot 2} + \frac{1}{2 \cdot 3} + \frac{1}{3 \cdot 4} + \dots + \frac{1}{(n-1) \cdot n} = 1 - \frac{1}{n} = \mathcal{O}(1)$$

$$\sum_{k=1}^n \frac{1}{k^2} = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \dots + \frac{1}{n^2} < 1 + \frac{1}{2^2} + \frac{1}{3^2} + \dots = \frac{\pi^2}{6} = \mathcal{O}(1)$$

$$\sum_{k \text{ is a perfect power}} \frac{1}{k-1} = \frac{1}{3} + \frac{1}{7} + \frac{1}{8} + \frac{1}{15} + \frac{1}{24} + \frac{1}{26} + \frac{1}{31} + \frac{1}{35} + \dots = 1 = \mathcal{O}(1)$$

### ◆ 有必要讨论这类级数吗？难道，基本操作次数、存储单元数可能是分数？是的，某种意义上！

$$(1 - \lambda) \cdot [1 + 2\lambda + 3\lambda^2 + 4\lambda^3 + \dots] = 1/(1 - \lambda) = \mathcal{O}(1), \quad 0 < \lambda < 1 \quad // \text{几何分布}$$

### ◆ 可能未必收敛，然而长度有限

$$h(n) = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n} = \Theta(\log n) \quad // \text{调和级数}$$

$$\log 1 + \log 2 + \log 3 + \log 4 + \dots + \log n = \log(n!) = \Theta(n \log n) \quad // \text{对数级数}$$

### ◆ 如有兴趣，不妨读读：Concrete Mathematics

// ex-2.35, Goldbach Theorem

Go To Statement Considered Harmful.

- E. Dijkstra, 1968

## 1. 绪论

### 算法分析

'GOTO Considered Harmful' Considered Harmful.

- F. Rubin , 1985

### 迭代

'"GOTO Considered Harmful" Considered Harmful'

Considered Harmful?

邓俊辉

- Communications of the ACM , 1987

deng@tsinghua.edu.cn

## 迭代 vs. 级数

```
❖ for (int i = 0; i < n; i++)
    for (int j = 0; j < [n]; j++)
        O10peration(i, j);
```

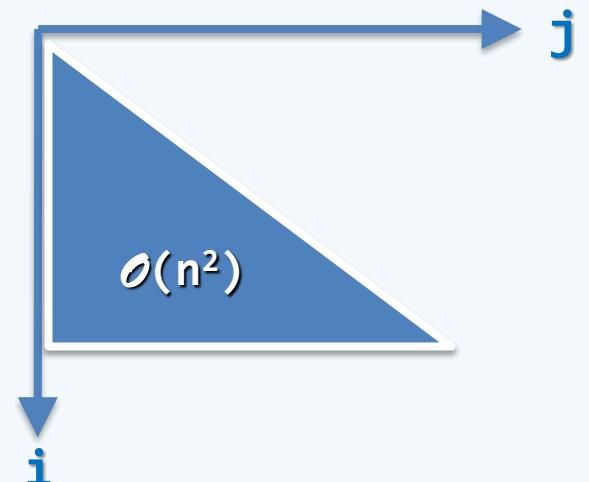
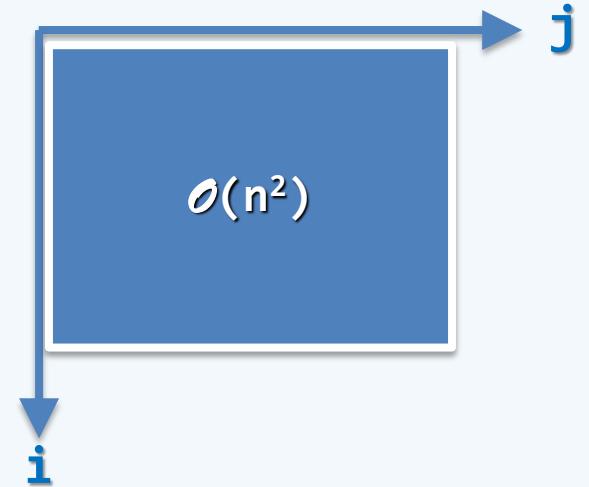
算术级数：

$$\sum_{i=0}^{n-1} n = n + n + \dots + n = n * n = O(n^2)$$

```
❖ for (int i = 0; i < n; i++)
    for (int j = 0; j < [i]; j++)
        O10peration(i, j);
```

算术级数：

$$\sum_{i=0}^{n-1} i = 0 + 1 + \dots + (n - 1) = \frac{n(n-1)}{2} = O(n^2)$$



## 迭代 vs. 级数

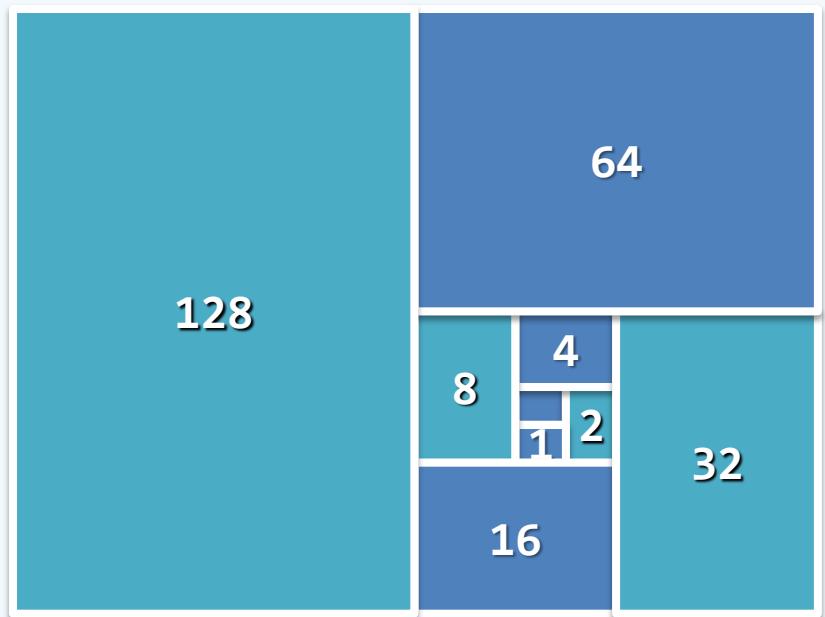
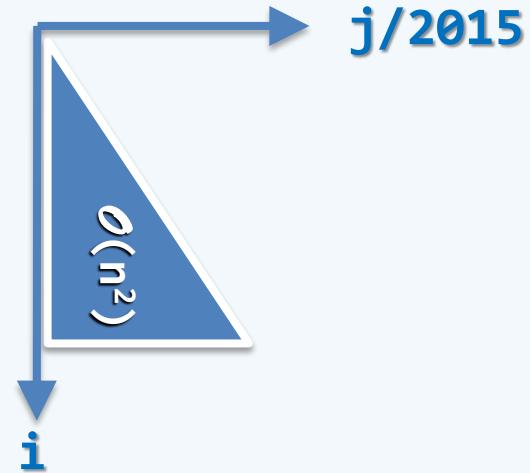
```
❖ for (int i = 0; i < n; i++)
    for (int j = 0; j < [i]; [j += 2015])
        O1Operation(i, j);
```

算术级数 : ...

```
❖ for (int i = 1; i < n; [i <<= 1])
    for (int j = 0; j < [i]; j++)
        O1Operation(i, j);
```

几何级数 :  $1 + 2 + 4 + \dots + 2^{\lfloor \log_2(n-1) \rfloor}$

$$\begin{aligned}
 &= \sum_{k=0}^{\lfloor \log_2(n-1) \rfloor} 2^k \quad (\text{let } k = \log_2 i) \\
 &= 2^{\lceil \log_2 n \rceil} - 1 = \mathcal{O}(n)
 \end{aligned}$$



## 迭代 vs. 级数

```

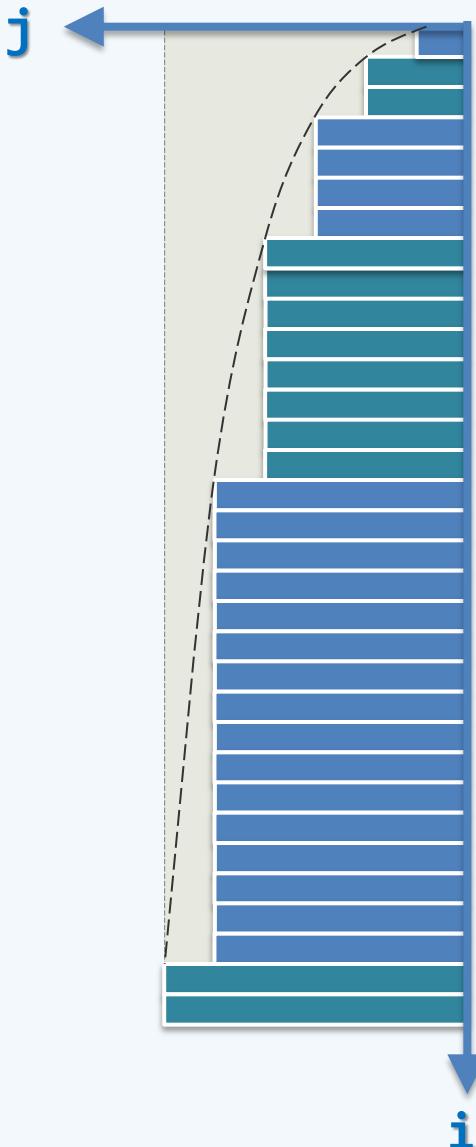
❖ for (int i = 0; i <= n; i++)
    for (int j = 1; j < i; [j += j])
        operation(i, j);

几何级数 :  $\sum_{k=0}^n \lceil \log_2 i \rceil = O(n \log n)$ 

(i = 0, 1, 2, 3~4, 5~8, 9~16, ...)

= 0 + 0 + 1 + 2*2 + 3*4 + 4*8 + ...
=  $\sum_{k=0.. \log n} (k * 2^{k-1})$ 
=  $O(\log n * 2^{\log n})$  (CM page#33)

```



## 1. 绪论

算法分析

正确性

Beware of bugs in the above code;

I have only proved it correct, not tried it.

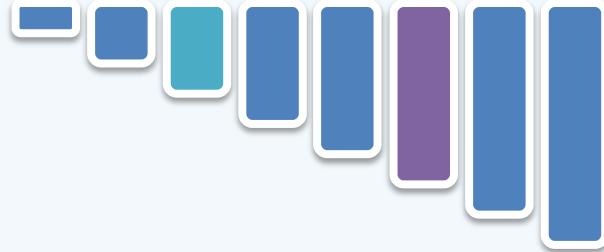
- D. Knuth

邓俊辉

deng@tsinghua.edu.cn

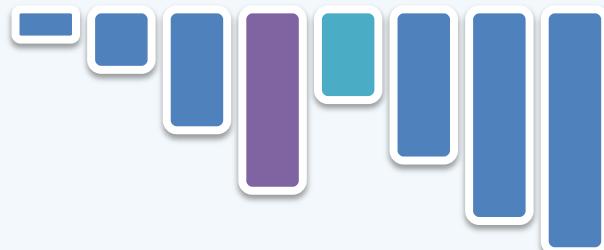
## 起泡排序

❖ 问题： 给定 $n$ 个整数，  
将它们按（非降）序排列



❖ 观察： 有序 / 无序 序列中，

任意 / 总有一对相邻元素 顺序 / 逆序



❖ 扫描交换：依次比较每一对相邻元素；如有必要，交换之

❖ 若整趟扫描都没有进行交换，则排序完成；否则，再做一趟扫描交换

## 实例



(a)



(b)



(c)



(d)



(e)



(f)



(g)

## 算法

```
❖ void bubblesort(int A[], int n) { //第二章将进一步改进  
    for ( bool sorted = false; [sorted = !sorted]; n-- ) //反复扫描交换，每一趟...  
        for ( int i = 1; i < n; i++ ) //自左向右，逐对检查A[0, n)内各相邻元素  
            if ( A[ i - 1 ] > A[ i ] ) { //若逆序，则  
                swap( A[ i - 1 ], A[ i ] ); //令其互换，同时  
                sorted = false; //清除（全局）有序标志  
            }  
    } //该算法...的确实现了所需的功能？必然会结束？至多需迭代多少趟？
```



## 证明

- ❖ 不变性：经  $k$  趟扫描交换后，最大的  $k$  个元素必然就位
- ❖ 单调性：经  $k$  趟扫描交换后，问题规模缩减至  $n - k$
- ❖ 正确性：经至多  $n$  趟扫描后，算法必然终止，且能给出正确解答



## 性能

◆ 最坏情况：输入数据反序排列

共  $n - 1$  趟扫描交换

每趟的效果，都等同于当前有效区间循环左移一位

第  $k$  趟中，需做  $n - k$  次比较和  $3*(n - k)$  次移动， $0 < k < n$

累计： $\#KMP = (n-1) + (n-2) + \dots + 1 = n(n-1)/2$

$\#MOV = 3 \times n(n-1)/2$

$T(n) = 4 \times n(n-1)/2 = O(n^2)$

◆ 最好情况：所有输入元素已经完全（或接近）有序

外循环仅 1 次，做  $n - 1$  次比较和 0 次元素交换

累计： $T(n) = n - 1 = \Omega(n)$

## 1. 绪论

算法分析

封底估算

He calculated just as men breathe,  
as eagles sustain themselves in the air.

- Francois Arago

邓俊辉

deng@tsinghua.edu.cn

## Back-Of-The-Envelope Calculation

❖ 地球(赤道)周长  $\approx 787 \times 360/7.2$   
 $= 787 \times 50 = 39,350 \text{ km}$

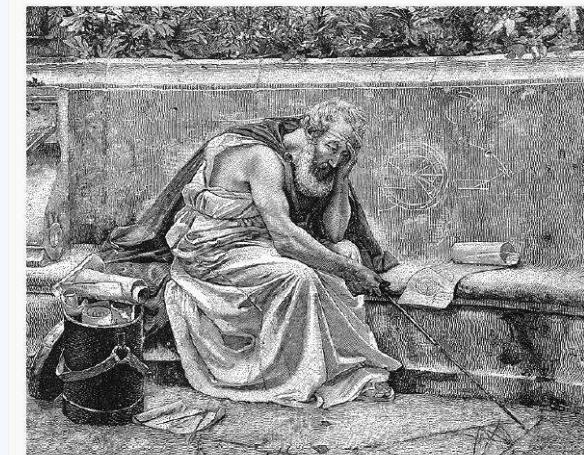
❖ 1天  $= 24\text{hr} \times 60\text{min} \times 60\text{sec}$   
 $\approx 25 \times 4000 = 10^5 \text{ sec}$

❖ 1生  $\approx 1\text{世纪} = 100\text{yr} \times 365 = 3 \times 10^4 \text{ day} = 3 \times 10^9 \text{ sec}$

❖ “为祖国健康工作五十年”  $\approx 1.6 \times 10^9 \text{ sec}$

❖ “三生三世”  $\approx 300 \text{ yr} = 10^{10} = (1 \text{ googol})^{(1/10)} \text{ sec}$

❖ 宇宙大爆炸至今  $= 10^{21} = 10 \times (10^{10})^2 \text{ sec}$



Eratosthenes  
(276 ~ 194 B.C.)



Bill Gates  
(1955 - )

## Back-Of-The-Envelope Calculation

❖ 考察对全国人口普查数据的排序

$$n = 10^9 \dots$$

普通PC

1GHz

$10^9$  flops

天河1A

千万亿次 = 1P

$10^{15}$  flops

硬件

Bubblesort  
 $(10^9)^2$   
 $10^{18}$

$10^9$  sec  
30 yr

$10^3$  sec  
20 min

Mergesort  
 $(10^9) \times \log(10^9)$   
 $30 \times 10^9$

30 sec

0.03 ms

算法

## 课后

- ❖ 试按照“不变性+单调性”的模式，归纳证明本章各算法的正确性
- ❖ 试举例说明，`operation()`对循环体的复杂度也可能有实质影响
- ❖ 学习不同开发环境提供的Profiler工具，并藉此优化你的程序性能
- ❖ 习题[1-32]

## 1. 绪论

迭代与递归

减而治之

迭代乃人工，递归方神通

To iterate is human,  
to recurse, divine.

邓俊辉

deng@tsinghua.edu.cn

**Sum**

❖ 问题：计算任意n个整数之和

❖ 实现：逐一取出每个元素，累加之

```
int SumI( int A[], int n ) {  
    int sum = 0; //O(1)  
    for ( int i = 0; i < n; i++ ) //O(n)  
        sum += A[i]; //O(1)  
    return sum; //O(1)  
}
```

❖ 无论A[]内容如何，都有：

$$T(n) = 1 + n*1 + 1 = n + 2 = O(n) = \Omega(n) = \Theta(n)$$

❖ 空间呢？

## Decrease-and-conquer

❖ 为求解一个大规模的问题，可以

将其划分为两个子问题：其一**平凡**，另一规模**缩减**

//单调性

分别求解子问题

由子问题的解，得到原问题的解



## Linear Recursion: Trace

```
❖ sum( int A[], int n ) {  

    return  

    n < 1 ?  

        0 : sum(A, n - 1) + A[n - 1];  

}
```

❖ 递归跟踪分析

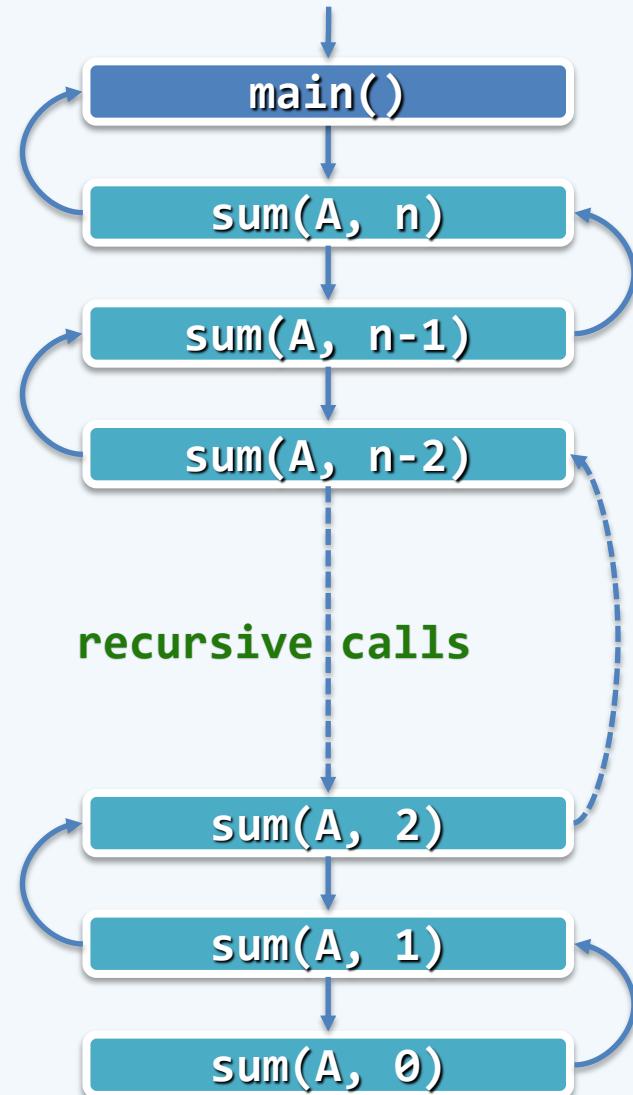
检查每个**递归实例**

累计所需时间（调用语句本身，计入对应的子实例）

其总和即算法执行时间

❖ 本例中，单个递归实例自身只需 $\mathcal{O}(1)$ 时间

$$T(n) = \mathcal{O}(1) * (n + 1) = \mathcal{O}(n)$$



## Linear Recursion: Recurrence

❖ 从递推的角度看，为求解 $\text{sum}(A, n)$ ，需

- 递归求解规模为 $n-1$ 的问题 $\text{sum}(A, n - 1)$ ，再  $//T(n-1)$
- 再累加上 $A[n - 1]$   $//O(1)$

❖ 递推方程  $T(n) = T(n - 1) + O(1)$   $//recurrence$

$$T(0) = O(1) \quad //base: \text{sum}(A, 0)$$

$$\begin{aligned} T(n) - n &= T(n - 1) - (n - 1) = \dots \\ &= T(2) - 2 \\ &= T(1) - 1 \\ &= T(0) \end{aligned}$$

$$T(n) = O(1) + n = O(n)$$

## Reverse

❖ 任给数组A[0, n)，将其中的子区间A[lo, hi]前后颠倒

统一接口 : void reverse( int \* A, int lo, int hi );

❖ if (lo < hi) //问题规模的**奇偶性**不变，需要两个递归基 //递归版

```
{ swap( A[lo], A[hi] ); reverse( A, lo + 1, hi - 1 ); }
```

❖ next:



```
{ swap( A[lo], A[hi] ); lo++; hi--; goto next; }
```

❖ while (lo < hi) swap( A[lo++], A[hi--] ); //迭代精简版

## 1. 绪论

凡治众如治寡，分数是也

迭代与递归

The control of a large force is  
the same principle as

分而治之

the control of a few men:

it is merely a question of  
dividing up their numbers.

邓俊辉

deng@tsinghua.edu.cn

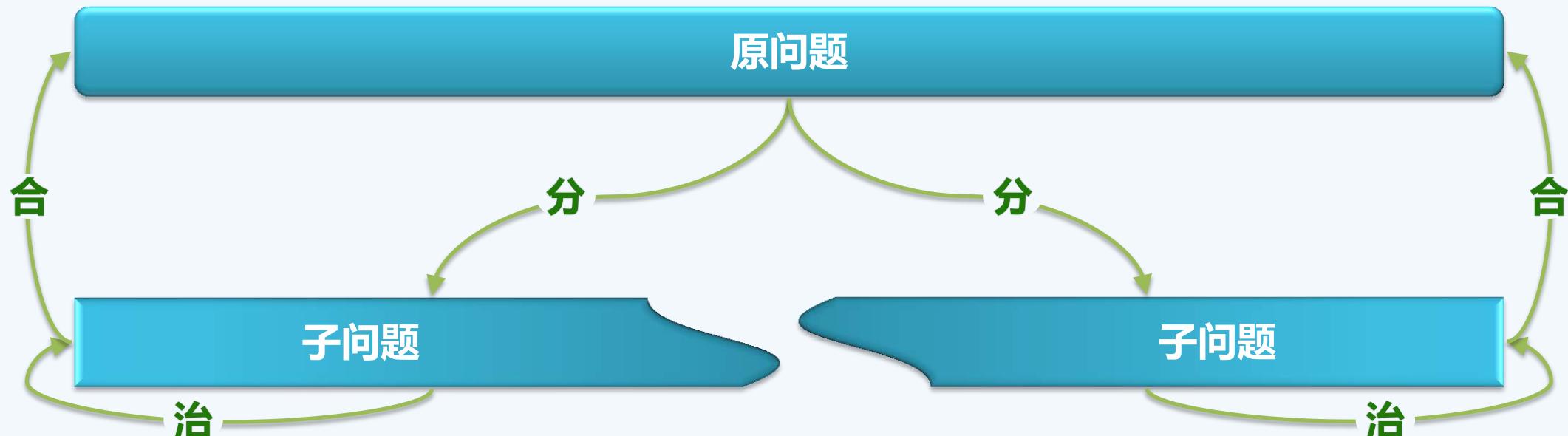
## Divide-and-conquer

❖ 为求解一个大规模的问题，可以

将其划分为若干（通常两个）子问题，规模大体相当

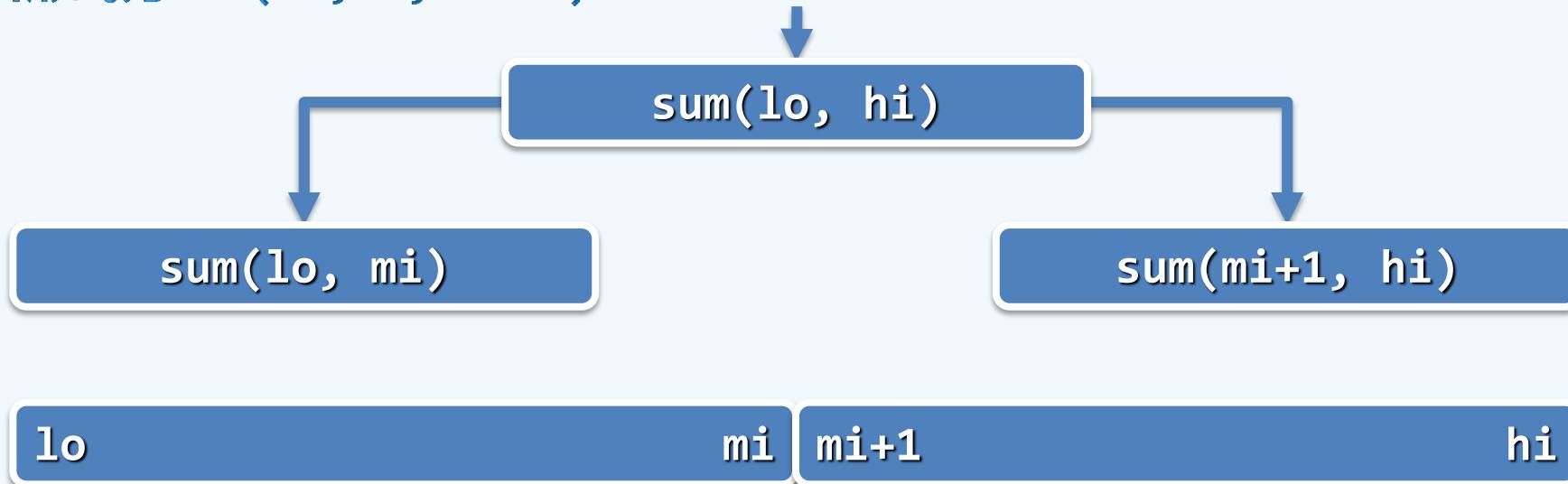
分别求解子问题

由子问题的解，得到原问题的解



## Binary Recursion

```
❖ sum( int A[], int lo, int hi ) { //区间范围A[lo, hi]  
    if ( lo == hi ) return A[lo];  
  
    int mi = (lo + hi) >> 1;  
  
    return sum( A, lo, mi ) + sum( A, mi + 1, hi );  
} //入口形式为sum( A, 0, n-1 )
```



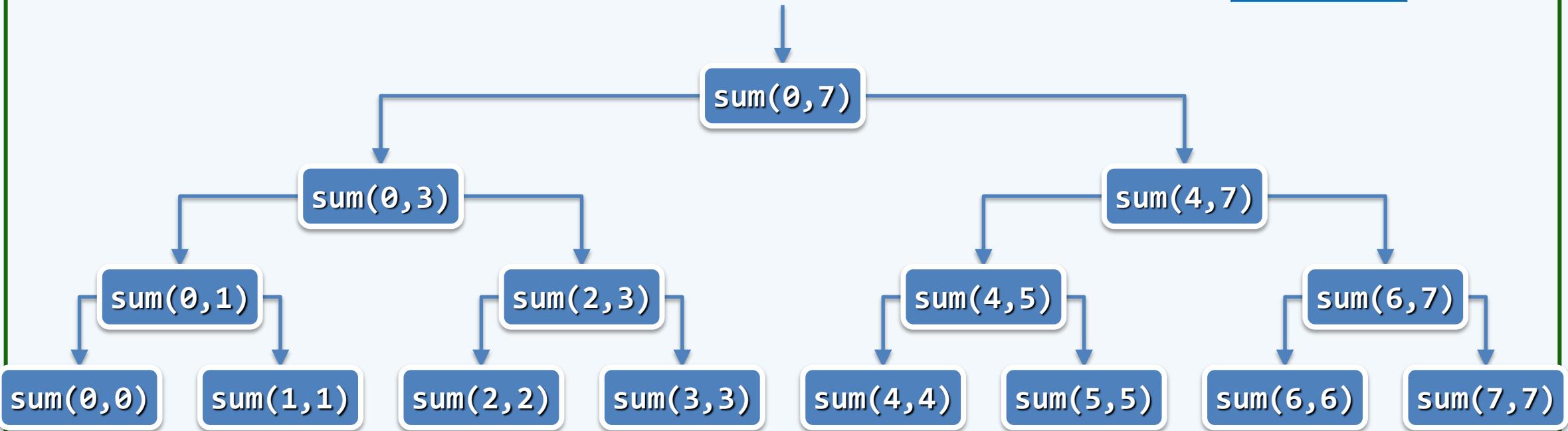
## Binary Recursion: Trace

❖  $T(n) = \text{各层递归实例所需时间之和}$

// 递归跟踪

$$= O(1) \times (2^0 + 2^1 + 2^2 + \dots + 2^{\log n})$$

$$= O(1) \times (2^1 + \log n - 1) = O(n) \quad // \text{更快捷地，几何级数与最大/末项等阶}$$



## Binary Recursion: Recurrence

❖ 从递推的角度看，为求解 $\text{sum}(A, \text{lo}, \text{hi})$ ，需

- 递归求解 $\text{sum}(A, \text{lo}, \text{mi})$ 和 $\text{sum}(A, \text{mi}+1, \text{hi})$ ，进而  $//2*T(n/2)$
- 将子问题的解累加  $//O(1)$

❖ 递推关系

$$T(n) = 2*T(n/2) + O(1)$$

$$T(1) = O(1) \quad //\text{base: } \text{sum}(A, k, k)$$

❖ 求解  $T(n) = 2*T(n/2) + c_1$

$$\begin{aligned} T(n) + c_1 &= 2*(T(n/2) + c_1) = 2^2 * (T(n/4) + c_1) = \dots \\ &= 2^{\log n} (T(1) + c_1) = n * (c_2 + c_1) \end{aligned}$$

$$T(n) = (c_1 + c_2)n - c_1 = O(n)$$

# 1. 绪论

迭代与递归

Max2

邓俊辉

deng@tsinghua.edu.cn

## 迭代1

❖ 从数组区间 $A[lo, hi]$ 中找出最大的两个整数 $A[x_1]$ 和 $A[x_2]$   $// A[x_1] \geq A[x_2]$   
 元素比较的次数，要求尽可能地少



```

❖ void max2(int A[], int lo, int hi, int & x1, int & x2) { // 1 < n = hi - lo
    for ( x1 = lo, int i = lo + 1; i < hi; i++ ) //扫描A[lo, hi], 找出A[x1]
        if ( A[x1] < A[i] ) x1 = i; // hi - lo - 1 = n - 1
    for ( x2 = lo, int i = lo + 1; i < x1; i++ ) //扫描A[lo, x1]
        if ( A[x2] < A[i] ) x2 = i; // x1 - lo - 1
    for ( int i = x1 + 1; i < hi; i++ ) //再扫描A(x1, hi), 找出A[x2]
        if ( A[x2] < A[i] ) x2 = i; // hi - x1 - 1
} //无论如何，比较次数总是  $\Theta(2n - 3)$ 
    
```

## 迭代2

```

❖ void max2( int A[], int lo, int hi, int & x1, int & x2 ) { // 1 < n = hi-lo

    if ( A[x1 = lo] < A[x2 = lo + 1] ) swap(x1, x2);

    for ( int i = lo + 2; i < hi; i++ )

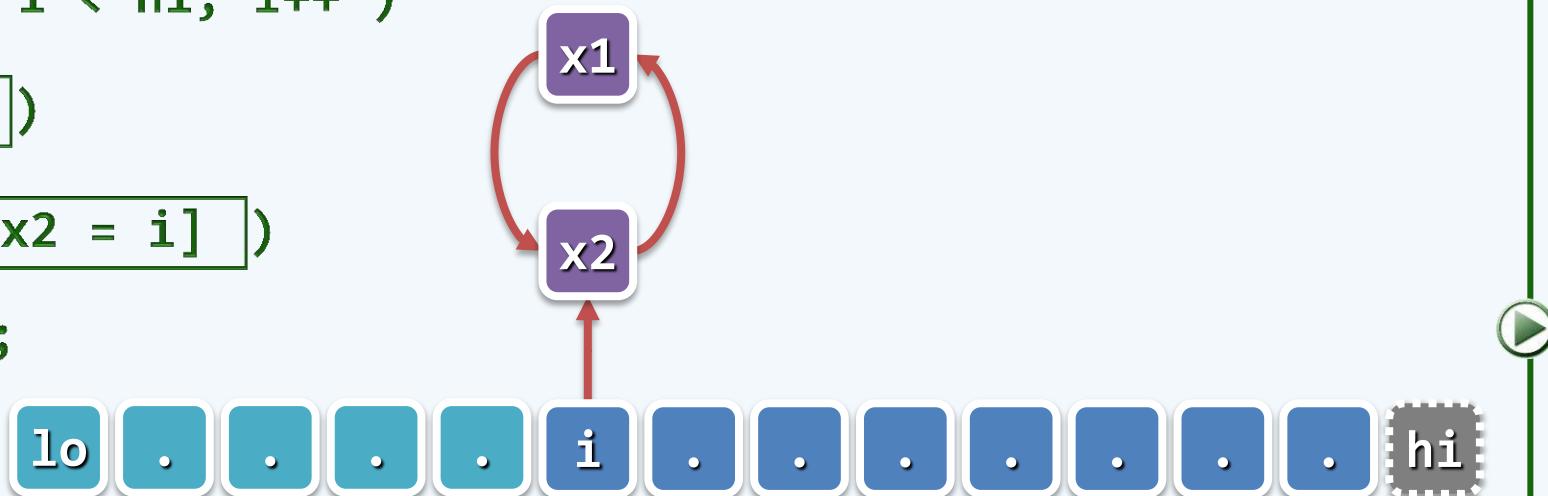
        if ( A[x2] < A[i] )

            if ( A[x1] < A[x2 = i] )

                swap(x1, x2);

}

```



❖ 最好情况 ,  $1 + (n - 2) * 1 = n - 1$

❖ 最坏情况 ,  $1 + (n - 2) * 2 = 2n - 3$

❖ 比较次数可否进一步减少呢 ? 分而治之 !

## 递归 + 分治

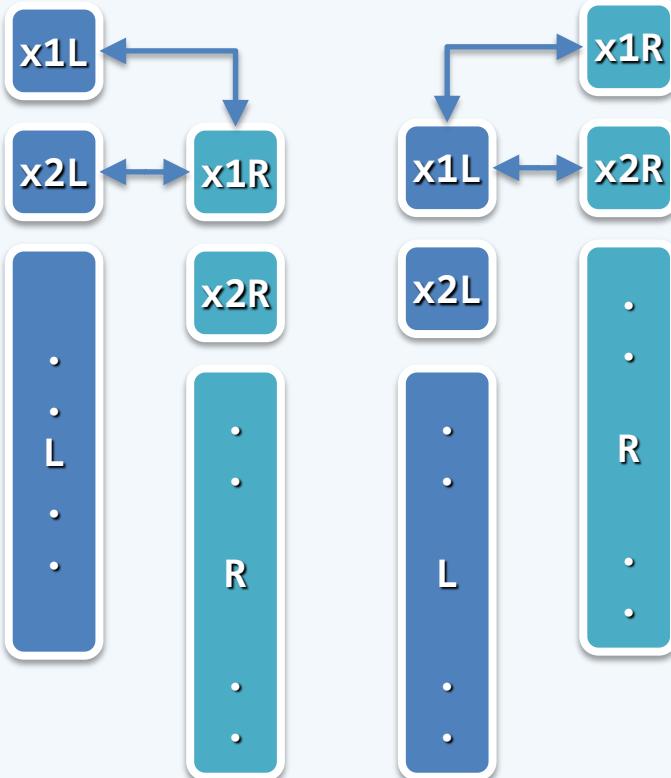
```

❖ void max2( int A[], int lo, int hi, int & x1, int & x2 ) {
    if ( [ hi <= lo + 3 ] ) { trivial( A, lo, hi, x1, x2 ); return; } //T(3) <= 3

    int mi = (lo + hi)/2; //divide
    int x1L, x2L; max2( A, lo, mi, x1L, x2L );
    int x1R, x2R; max2( A, mi, hi, x1R, x2R );

    if ( [ A[x1L] > A[x1R] ] ) {
        x1 = x1L; x2 = [ A[x2L] > A[x1R] ] ? x2L : x1R;
    } else {
        x1 = x1R; x2 = [ A[x1L] > A[x2R] ] ? x1L : x2R;
    } //1 + 1 = 2
} //T(n) = 2*T(n/2) + 2 <= 5n/3 - 2 ; 借助数据结构，还可进一步优化（第10章）

```



## Master Theorem

❖ [AHU-74], p64, Theorem 2.1

Recurrence	Solution	Examples
$T(n) = T(n-1) + 1$	$\Theta(n)$	向量求和之线性递归版
$T(n) = T(n-1) + n$	$\Theta(n^2)$	列表起泡排序之线性递归版
$T(n) = 2*T(n-1) + 1$	$\Theta(2^n)$	Hanoi塔、Fibonacci数
$T(n) = 2*T(n-1) + n$	$\Theta(2^n)$	
$T(n) = T(n/2) + 1$	$\Theta(\log n)$	向量的二分查找
$T(n) = T(n/2) + n$	$\Theta(n)$	列表的二分查找
$T(n) = 2*T(n/2) + 1$	$\Theta(n)$	向量求和之二分递归版
$T(n) = 2*T(n/2) + n$	$\Theta(n \log n)$	归并排序

# 1. 绪论

迭代与递归

尾递归

邓俊辉

deng@tsinghua.edu.cn

## Tail Recursion

❖ 递归算法易于理解和实现，但空间（甚至时间）效率低

在讲求效率时，应将递归改写为等价的迭代形式

❖ fac(n) { return ( $1 > n$ ) ? 1 :  $n * \text{fac}(n-1)$ ; }

❖ fac(n) {  
    if ( $1 > n$ ) return 1;  
    else return  $n * \text{fac}(n - 1)$ ; //tail recursion

}

❖ 尾递归：最后一步是递归调用

最简单的递归模式，可便捷地改写

多分支递归

二分递归

线性递归

尾递归

## Tail Recursion

<code>fac(n) { /* 递归 */</code>	<code>fac(n) { /* 统一转换为迭代 */</code>	<code>fac(n) { /* 简捷 */</code>
<code>     </code>	<code>     </code>	<code>     </code>
<code>      int f = 1; //记录子问题的解</code>	<code>      next: //转向标志，模拟递归调用</code>	<code>      int f = 1;</code>
<code>     </code>	<code>     </code>	<code>     </code>
<code>if (1 &gt; n) return 1;</code>	<code>if (1 &gt; n) return f;</code>	<code>while (1 &lt; n)</code>
<code>     </code>	<code>     </code>	<code>     </code>
<code>else return n*fac(n-1);</code>	<code>    f *= n--;</code>	<code>    f *= n--;</code>
<code>     </code>	<code>      goto next; //模拟递归返回</code>	<code>      return f;</code>
<code>}</code>	<code>}</code>	<code>}</code>
<code>//O(n)时间 + O(n)空间</code>	<code>//O(n)时间 + O(1)空间</code>	<code>//O(n)时间 + O(1)空间</code>

## 课后

❖ 做递归跟踪分析时，为什么递归调用语句本身可不统计？

❖ 试用递归跟踪法，分析fib()二分递归版的复杂度

通过递归跟踪，解释该版本复杂度过高的原因

❖ 递归算法的空间复杂度，主要取决于什么因素？

❖ 本节数组求和问题的两个（线性和二分）递归算法

时间复杂度相同，空间呢？

## 1. 绪论

动态规划

记忆法

请告诉我谁是中国人  
启示我，如何把记忆抱紧  
请告诉我这个民族的伟大  
轻轻的告诉我，不要喧哗

邓俊辉

deng@tsinghua.edu.cn

## fib() : 递归

❖ `fib(n) = fib(n-1) + fib(n-2)` // $\{0, 1, 1, 2, 3, 5, 8, \dots\}$

❖ `int fib(n)` //为何这么慢？

```
{ return (2 > n) ? n : fib(n - 1) + fib(n - 2); }
```

❖ 复杂度：  $T(0) = T(1) = 1$ ;  $T(n) = T(\lfloor n - 1 \rfloor) + T(\lfloor n - 2 \rfloor) + 1, \forall n > 1$

令  $S(n) = [T(n) + 1]/2$

则  $S(0) = 1 = fib(1), S(1) = 1 = fib(2)$

故  $S(n) = S(n - 1) + S(n - 2) = fib(n + 1)$

$T(n) = 2 \cdot S(n) - 1 = 2 \cdot fib(n + 1) - 1 = O(fib(n + 1)) = O(\phi^n) = O(2^n)$

其中  $\phi = (1 + \sqrt{5})/2 \approx 1.618$

**封底估算**

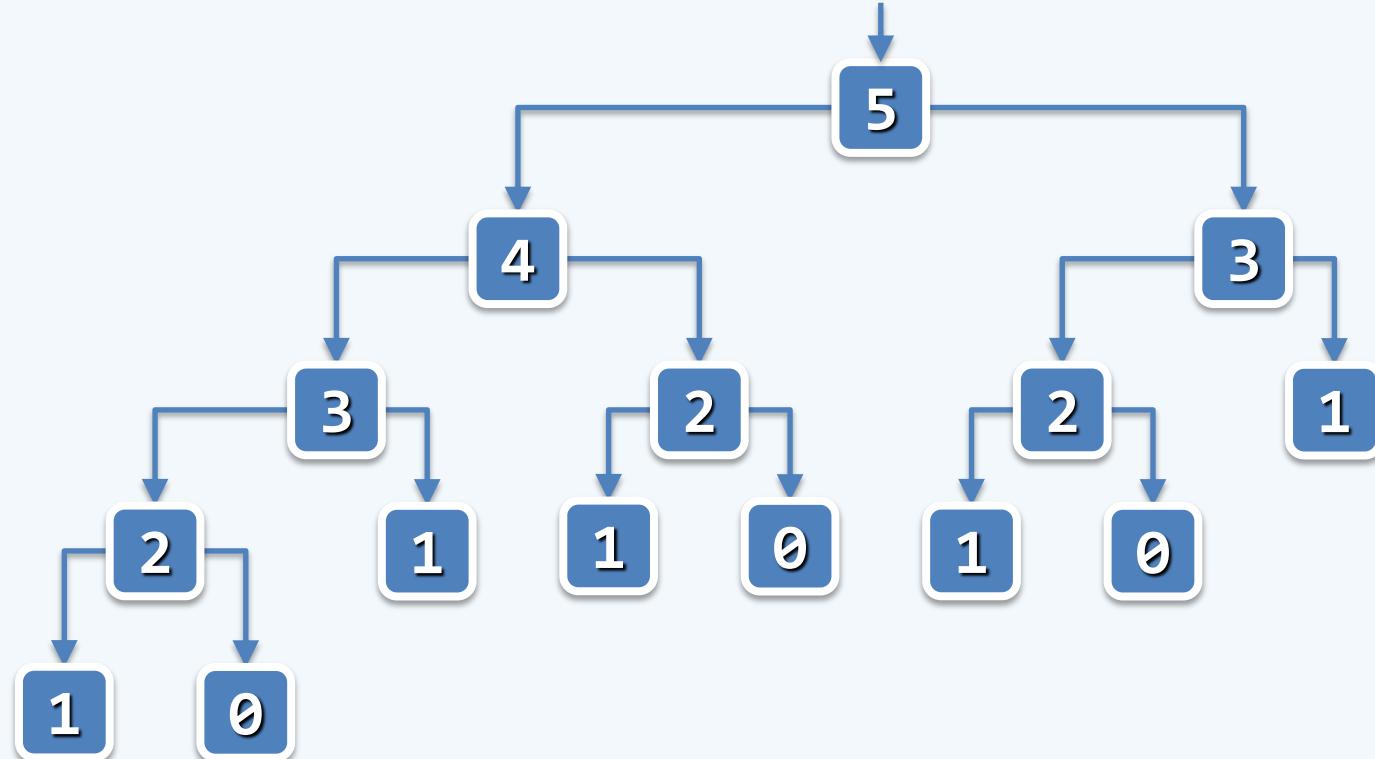
$$\phi^{36} \approx 2^{25} \quad \phi^{43} \approx 2^{30} \approx 10^9 \text{ flo} = 1 \text{ sec}$$

$$\phi^5 \approx 10 \quad \phi^{67} \approx 10^{14} \text{ flo} = 10^5 \text{ sec} \approx 1 \text{ day}$$

$$\phi^{92} \approx 10^{19} \text{ flo} = 10^{10} \text{ sec} \approx 10^5 \text{ day} \approx 3 \text{ century}$$

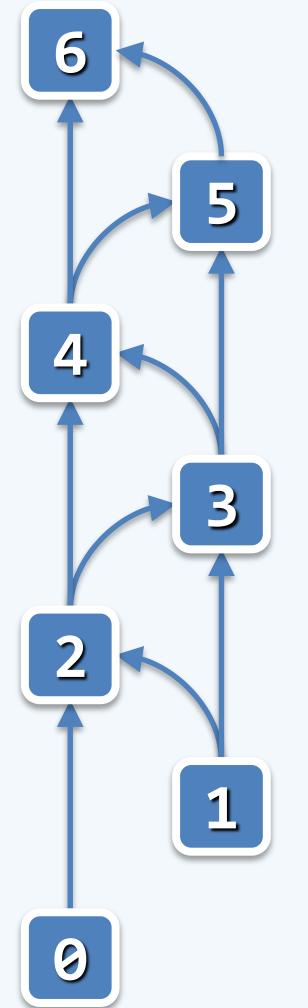
## fib() : 递归

❖ 递归版fib()低效的根源在于，各递归实例均被大量重复地调用



❖ 先后出现的递归实例，共计  $\mathcal{O}(\phi^n)$  个

而去除重复之后，总共不过  $\mathcal{O}(n)$  种



## fib() : 迭代

❖ 解决方法A ( 记忆 : memoization )

将已计算过实例的结果制表备查，避免重复调用

❖ 解决方法B ( 动态规划 : dynamic programming )

颠倒计算方向：由自顶而下递归，为自底而上迭代

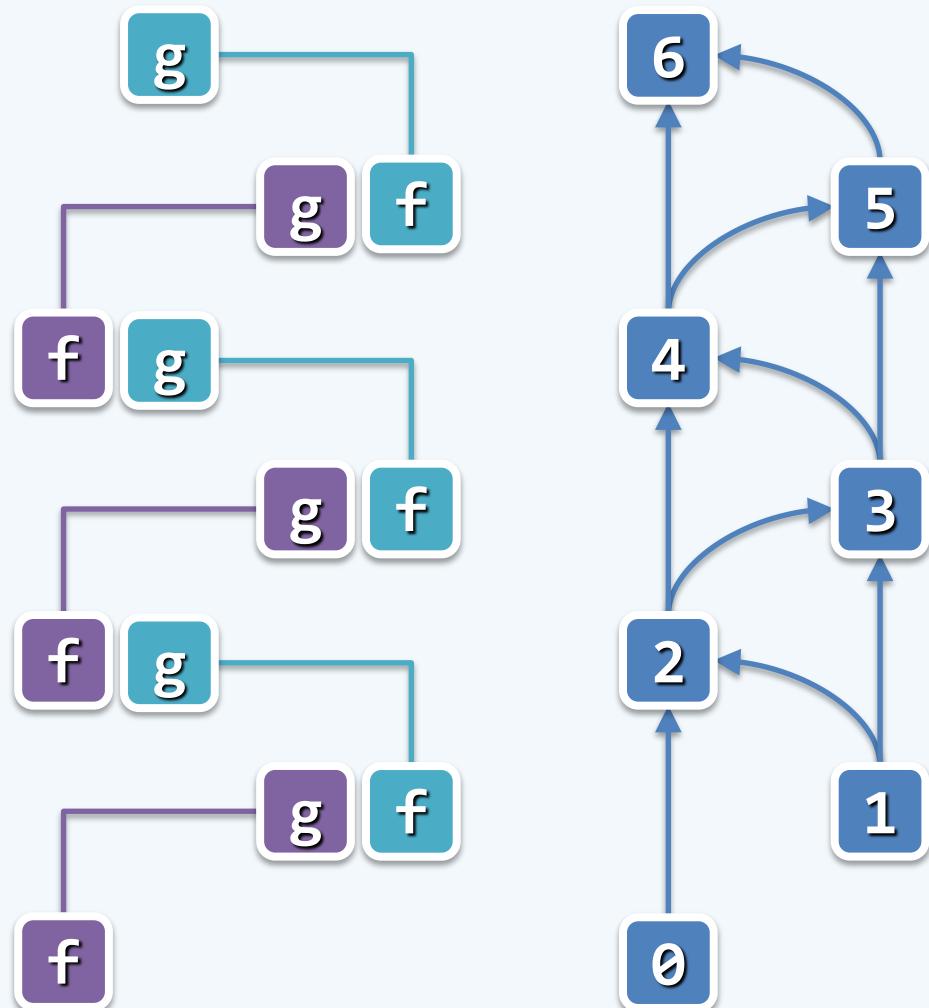
❖ 

```
f = 1; g = 0; //fib(-1), fib(0)
```

```
while ( 0 < n-- ) {
    g = g + f;
    f = g - f;
}
```

```
return g;
```

❖  $T(n) = \mathcal{O}(n)$  , 而且仅需 $\mathcal{O}(1)$ 空间 !



## 1. 绪论

动态规划

最长公共子序列

Make it work,  
make it right,  
make it fast.

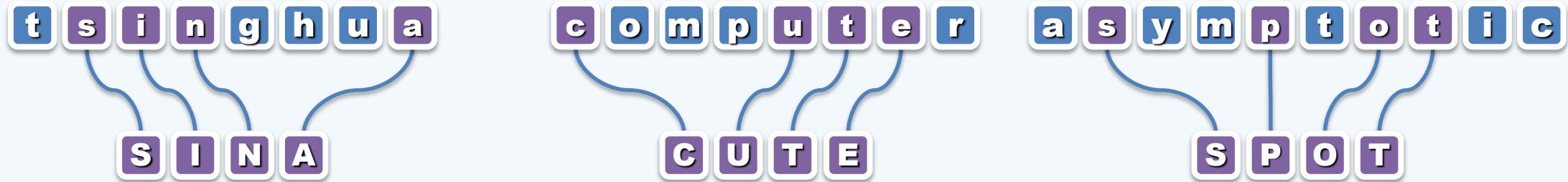
- Kent Beck

邓俊辉

deng@tsinghua.edu.cn

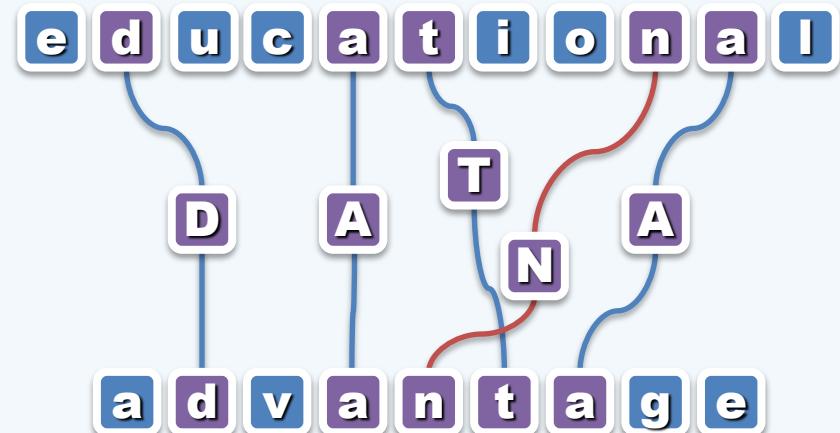
## LCS : 递归

❖ 子序列 ( Subsequence ) : 由序列中若干字符 , 按原相对次序构成

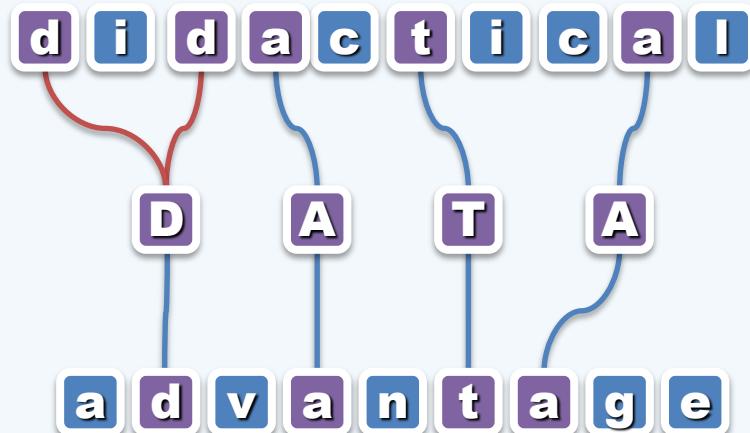


❖ 最长公共子序列 ( Longest Common Subsequence ) : 两个序列公共子序列中的最长者

可能有多个



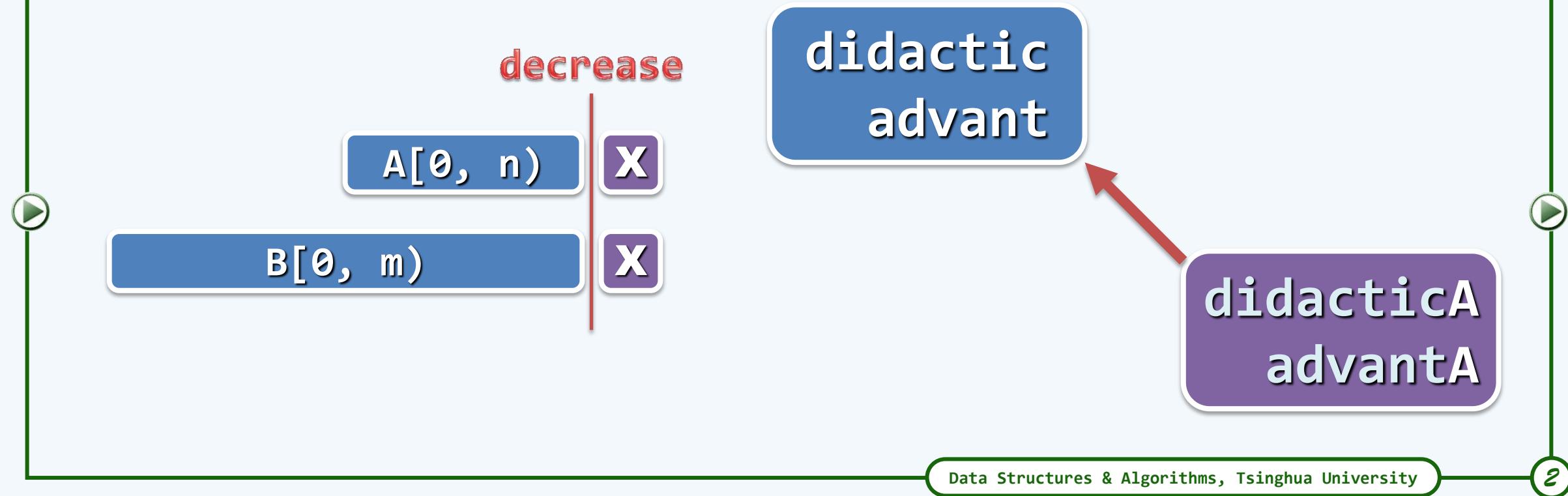
可能有歧义



## LCS : 递归

❖ 对于序列 $A[0, n]$ 和 $B[0, m]$ ， $\text{LCS}(A, B)$ 无非三种情况

- 0) 若 $n = -1$ 或 $m = -1$ ，则取作空序列 ("") //递归基
- 1) 若 $A[n] = 'X' = B[m]$ ，则取作： $\text{LCS}(A[0, n], B[0, m]) + 'X'$  //减而治之



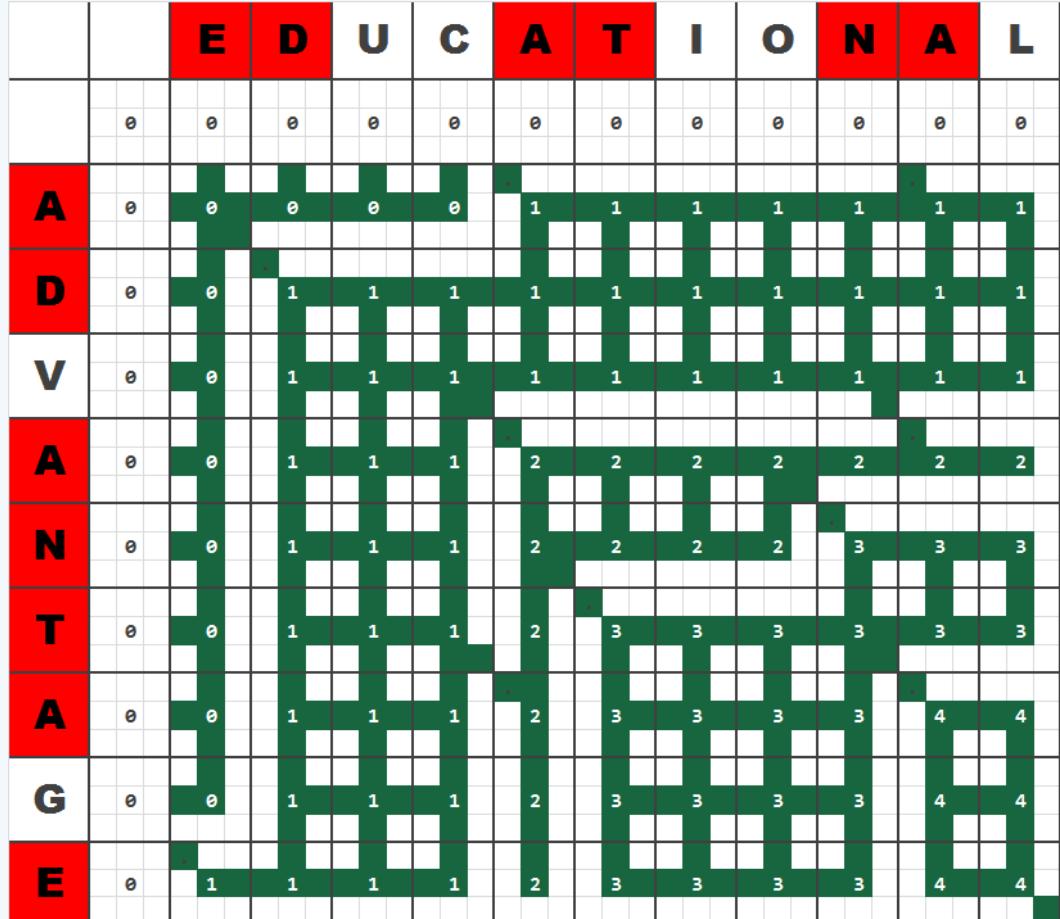
## LCS : 递归

2)  $A[n] \neq B[m]$ , 则在  $\text{LCS}(A[0, n], B[0, m])$  与  $\text{LCS}(A[0, n), B[0, m])$  中取更长者 //分而治之

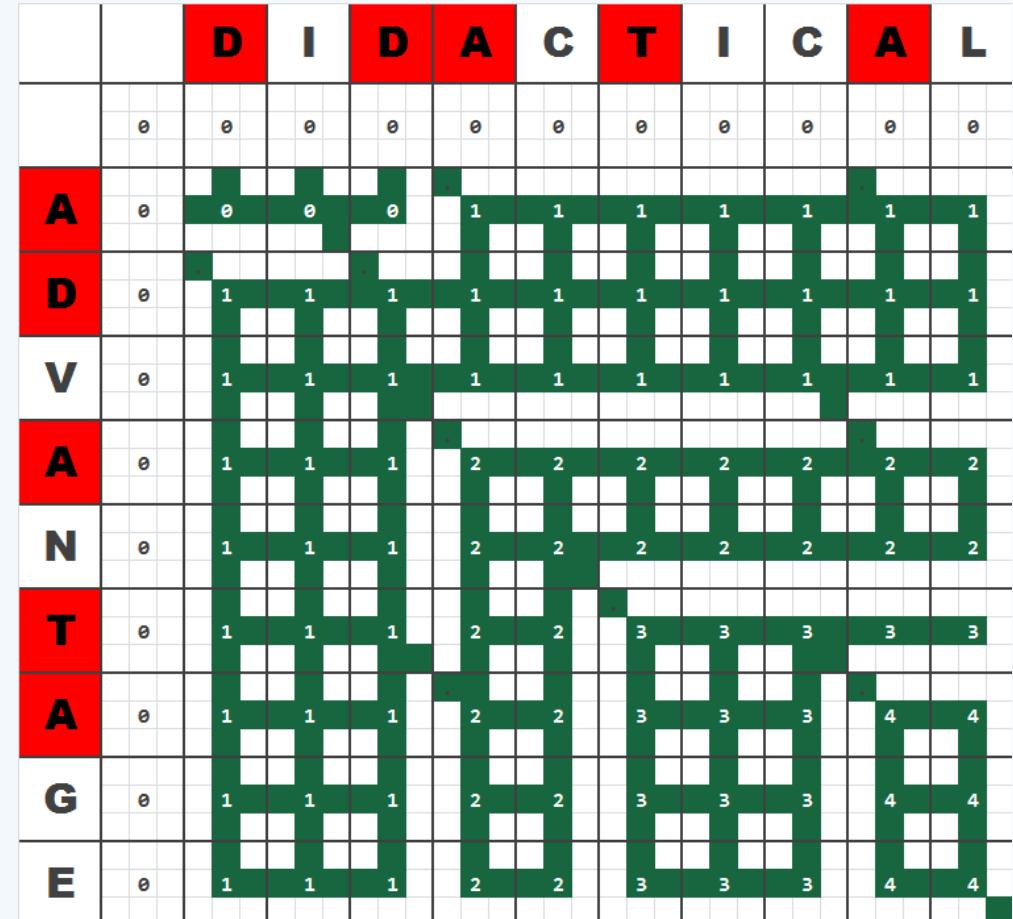


## LCS : 理解

❖ LCS的每一个解，对应于(0, 0)与(n, m)之间的一条**单调通路**；反之亦然



多解



歧义

## LCS : 递归

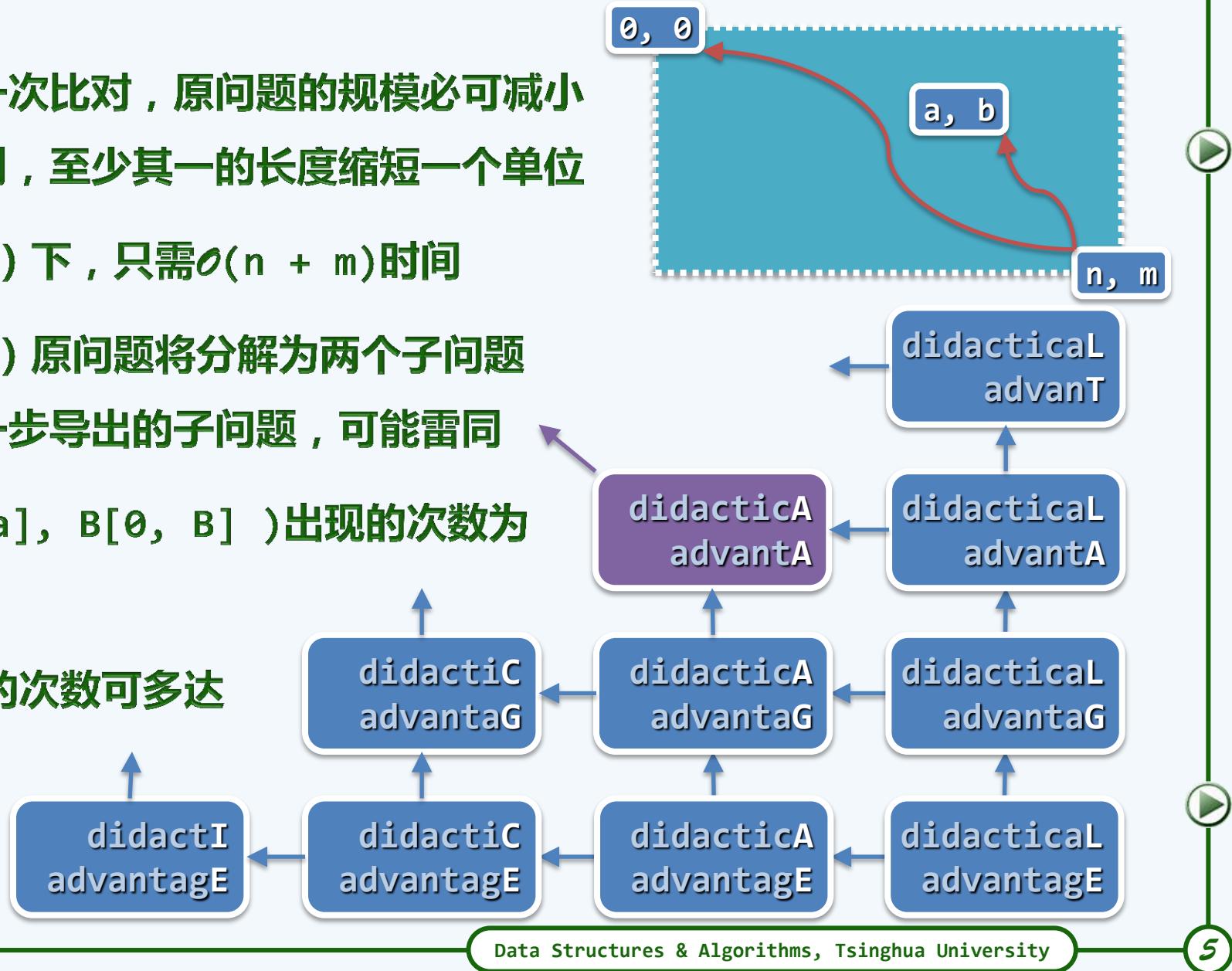
- ◆ 单调性：无论如何，每经过一次比对，原问题的规模必可减小  
具体地，作为输入的两个序列，至少其一的长度缩短一个单位
- ◆ 最好情况（不出现第2种情况）下，只需 $O(n + m)$ 时间
- ◆ 但问题在于，（在第2种情况）原问题将分解为两个子问题  
更糟糕的是，它们在随后进一步导出的子问题，可能雷同
- ◆ 在最坏情况下， $\text{LCS}(A[0, a], B[0, B])$ 出现的次数为

$$\binom{n+m-a-b}{n-a} = \binom{n+m-a-b}{m-b}$$

特别地， $\text{LCS}(A[0], B[0])$ 的次数可多达

$$\binom{n+m}{n} = \binom{n+m}{m}$$

当 $n = m$ 时，为 $\Omega(2^n)$



## LCS : 迭代

❖ 与fib()类似

这里也有大量重复的递归实例（子问题）

（最坏情况下）先后共计出现 $\Theta(2^n)$ 个

❖ 各子问题，分别对应于A和B的某个前缀组合  
因此总共不过 $\Theta(n * m)$ 种

❖ 采用动态规划的策略

只需 $\Theta(n * m)$ 时间即可计算出所有子问题

❖ 为此，只需

0) 将所有子问题（假想地）列成一张表

1) 颠倒计算方向，从LCS( $A[0], B[0]$ )出发

**依次计算出所有项**

		d	i	d	a	c	t	i	c	a	1
	0	0	0	0	0	0	0	0	0	0	0
a	0	0	0	0	1	1	1	1	1	1	1
d	0	1	1	1	1	1	1	1	1	1	1
v	0	1	1	1	1	1	1	1	1	1	1
a	0	1	1	1	2	2	2	2	2	2	2
n	0	1	1	1	2	2	2	2	2	2	2
t	0	1	1	1	2	2	3	3	3	3	3
a	0	1	1	1	2	2	3	3	3	4	4
g	0	1	1	1	2	2	3	3	3	4	4
e	0	1	1	1	2	2	3	3	3	4	4



## 课后

- ❖ 温习：程序设计基础（第3版）之第11章（动态规划）
- ❖ 自学：Introduction to Algorithms, §15.1, §15.3, §15.4
- ❖ 本节所介绍的迭代式LCS算法，似乎需要记录每个子问题的局部解，从而导致空间复杂度激增  
实际上，这既不现实，亦无必要  
试改进该算法，使得每个子问题只需常数空间，即可保证最终得到LCS的组成（而非仅仅长度）
- ❖ 考查序列 A = "immaculate" 和 B = "computer"
  - 1) 它们的LCS是什么？
  - 2) 这里的解是否唯一？是否有歧义性？
  - 3) 按照本节所给的算法，找出的是其中哪一个解？
- ❖ 实现LCS算法的递归版和迭代版，并通过实测比较其运行时间
- ❖ 采用memoization策略，改进fib()与LCS算法的递归版

## 1. 绪论

局限

缓存

不学诗，何以言

邓俊辉

不学礼，何以立

deng@tsinghua.edu.cn

## 循环位移

❖ //将数组A[0, n)中元素向左循环移动k个单元

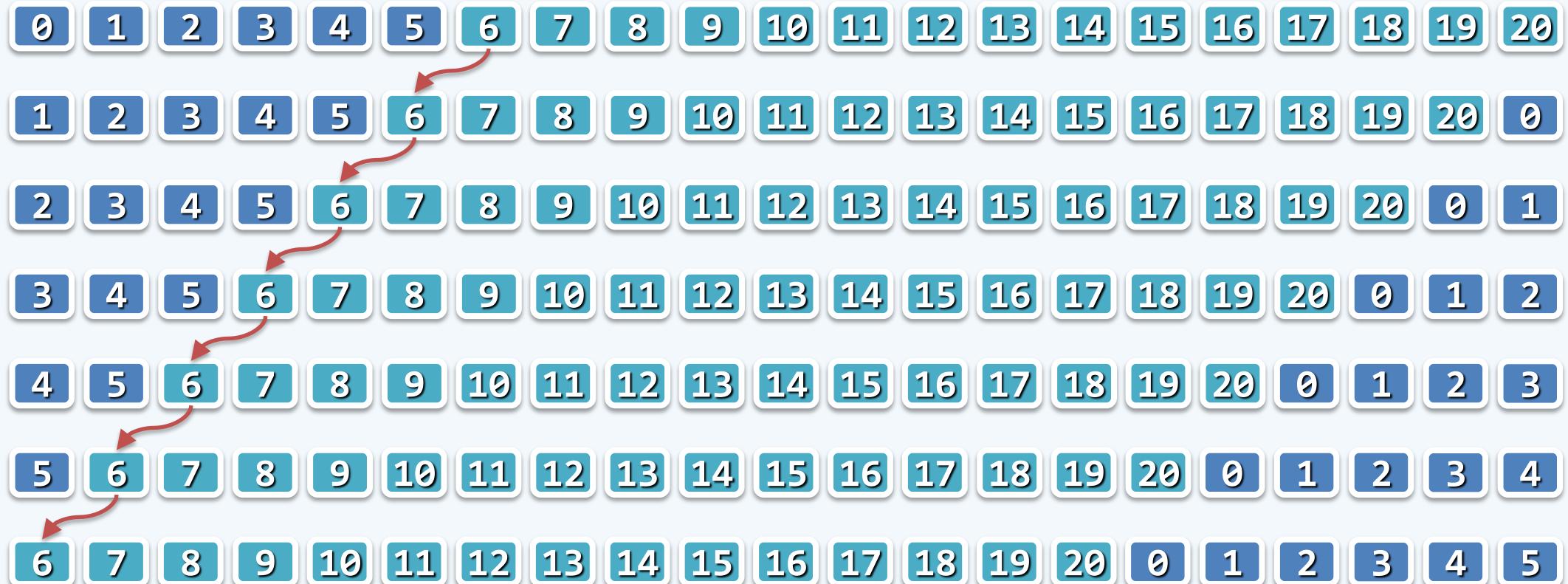
```
void shift( int * A, int n, int k )
```

❖ 比如：shift( A, 21, 6 );



## 蛮力版

```
❖ void shift0( int * A, int n, int k ) //反复以1为间距循环左移
{ while ( k-- ) shift( A, n, 0, 1 ); } //共迭代k次, O(nk)
```

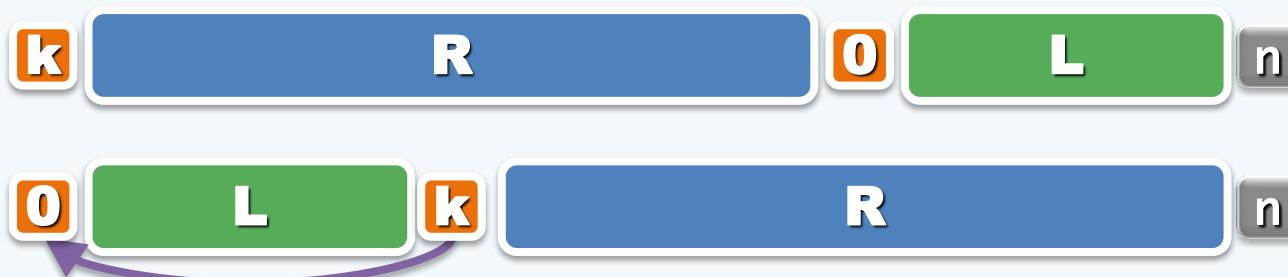


## 迭代版

```

❖ int shift( int * A, int n, int s, int k ) { // O( n / GCD(n, k)
    int b = A[s]; int i = s, j = (s + k) % n; int mov = 0; //mov记录移动次数
    while ( s != j ) //从A[s]出发，以k为间隔，依次左移k位
        { A[i] = A[j]; i = j; j = (j + k) % n; mov++; }
    A[i] = b; return mov + 1; //最后，起始元素转入对应位置
}

```



- ❖  $[0, n]$ 由关于 $k$ 的 $g = \text{GCD}(n, k)$ 个同余类组成 //各含 $n/g$ 个元素
- ❖  $\text{shift}(s, k)$ 能够且只能使其中之一就位 //即 $s$ 所属的同余类
- ❖ 其它的同余类呢...

## 迭代版

```
void shift1(int* A, int n, int k) { //经多轮迭代，实现数组循环左移k位，累计 $\theta(n)$ 
    for (int s = 0, mov = 0; mov < n; s++) // $\theta(\text{GCD}(n, k)) = \theta(n*k/\text{LCM}(n, k))$ 
        mov += shift(A, n, s, k);
}
```



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
6	1	2	9	4	5	12	7	8	15	10	11	18	13	14	0	16	17	3	19	20
6	7	2	9	10	5	12	13	8	15	16	11	18	19	14	0	1	17	3	4	20
6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	0	1	2	3	4	5

## 倒置版

```
void shift2( int * A, int n, int k ) { //借助倒置算法，将数组循环左移k位，O(3n)
    reverse( A, k ); //O(3k/2)
    reverse( A + k, n - k ); //O(3(n-k)/2)
    reverse( A, n ); //O(3n/2)
}
```



## 1. 绪论

局限

字宽

God kisses the finite in his love and  
man the infinite.

邓俊辉

deng@tsinghua.edu.cn

幂函数： $\Theta(2^r)$

- ❖ 观察：同一问题的不同算法，复杂度不尽相同
- ❖ 问题：对任何给定的整数  $n > 0$ ，计算  $a^n$ （ $a$ 为常数）
- ❖ 平凡实现： `pow = 1; //O(1)`  
`while ( 0 < n ) { //O(n)`  
`{ pow *= a; n--; } //O(1) + O(1)`
- ❖ 复杂度与  $n$  成正比， $T(n) = 1 + n*2 = \Theta(n)$
- ❖ 线性？**伪线性！**
- ❖ 所谓输入规模，准确地应定义为**用以描述输入所需的空间的规模**  
 对于此类数值计算，即是  $n$  的二进制位数  $r = \log_2(n + 1)$
- ❖ 复杂度与  $r$  成指数关系， $T(r) = \Theta(2^r)$  //太高了

a<sup>98765</sup>

$$\begin{aligned}
 &= a^{\wedge [9 \times 10^4 + 8 \times 10^3 + 7 \times 10^2 + 6 \times 10^1 + 5 \times 10^0]} \\
 &= (a^{10^4})^9 \times (a^{10^3})^8 \times (a^{10^2})^7 \times (a^{10^1})^6 \times (a^{10^0})^5 \\
 &= \text{pow}(\text{pow}(a, 10^4), 9) = \text{pow}(\text{pow}(\text{pow}(a, 10^3), 10), 9) \\
 &\quad \times \text{pow}(\text{pow}(a, 10^3), 8) \quad \times \text{pow}(\text{pow}(\text{pow}(a, 10^2), 10), 8) \\
 &\quad \times \text{pow}(\text{pow}(a, 10^2), 7) \quad \times \text{pow}(\text{pow}(\text{pow}(a, 10^1), 10), 7) \\
 &\quad \times \text{pow}(\text{pow}(a, 10^1), 6) \quad \times \text{pow}(\text{pow}(\text{pow}(a, 10^0), 10), 6) \\
 &\quad \times \text{pow}(\text{pow}(a, 10^0), 5) \quad \times \text{pow}(\text{pow}(a, 10^0), 5)
 \end{aligned}$$

❖ 若能在  $O(1)$  时间内得到  $\text{pow}(x, n)$ ,  $0 \leq n \leq 10$   
 则自右 (低) 向左 (高), 每个数位只需  $O(1)$  时间

a<sup>10110b</sup>

$$\begin{aligned}
 &= a^{\wedge [1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0]} \\
 &= (a^{2^4})^1 \times (a^{2^3})^0 \times (a^{2^2})^1 \times (a^{2^1})^1 \times (a^{2^0})^0 \\
 &= \text{pow}(\text{pow}(a, 2^4), 1) \quad = \quad \text{pow}(\text{sqr}(\text{pow}(a, 2^3)), 1) \\
 &\quad \times \text{pow}(\text{pow}(a, 2^3), 0) \quad \times \text{pow}(\text{sqr}(\text{pow}(a, 2^2)), 0) \\
 &\quad \times \text{pow}(\text{pow}(a, 2^2), 1) \quad \times \text{pow}(\text{sqr}(\text{pow}(a, 2^1)), 1) \\
 &\quad \times \text{pow}(\text{pow}(a, 2^1), 1) \quad \times \text{pow}(\text{sqr}(\text{pow}(a, 2^0)), 1) \\
 &\quad \times \text{pow}(\text{pow}(a, 2^0), 0) \quad \times \text{pow}(\text{pow}(a, 2^0), 0)
 \end{aligned}$$

$$\begin{aligned}
 \diamond \text{pow}(x, 0) &= 1 \quad // O(1) \\
 \text{pow}(x, 1) &= x \quad // O(1) \\
 \text{pow}(x, 2) &= \text{sqr}(x) \quad // O(1)
 \end{aligned}$$

❖ 故对应于每个数位，只需  $O(1)$  时间！

## 幂函数： $\mathcal{O}(r)$

```

❖ int power( int a, int n ) {
    int pow = 1; int p = a; //O(1 + 1)
    while (0 < n) { //O(logn)
        if (n & 1) pow *= p; //O(1 + 1)
        n >>= 1; p *= p; //O(1 + 1)
    }
    return pow; //O(1)
}

```

❖ 输入规模 = n的二进制位数 = r =  $\lceil \log_2(n+1) \rceil$

❖ 复杂度主要取决于**循环次数**， $T(r) = 1 + 1 + 4 \times r + 1 = \mathcal{O}(r)$

❖ 从 $\mathcal{O}(2^r)$ 到 $\mathcal{O}(r)$ 的改进，实际效果如何？

## 悖论？

- ❖ 观察： $\text{power}(n) = a^n$  的二进制展开宽度，可以度量为  $\Theta(n)$
- ❖ 根据以上算法，可在  $O(r = \log n)$  时间内计算出  $\text{power}(n)$
- ❖ 然而，即便是直接打印  $\text{power}(n)$ ，也至少需要  $\Omega(n)$  时间
- ...哪里错了？
- ❖ RAM 模型

常数代价准则 (uniform cost criterion)

对数代价准则 (logarithmic cost criterion)

## 1. 绪论

局限

随机数

As I have said so many times,  
God doesn't play dice with the world.

- A. Einstein

邓俊辉

deng@tsinghua.edu.cn

## 随机置乱

- ◆ 任给一个数组  $A[0, n)$ ，理想地将其中元素的次序 **随机打乱**
- ◆ // [R. Fisher & F. Yates, 1938], [R. Durstenfeld, 1964], [D. E. Knuth, 1969]

```
void shuffle( int A[], int n )
{ while ( 1 < n ) swap( A[ rand() % n ], A[ --n ] ); }
```



- ◆ 策略：自后向前，依次将各元素与随机选取的某一前驱（含自身）交换
- ◆ 的确可以**等概率**地生成**所有** $n!$ **种**排列？

## 1. 绪论

下界

代数判定树

两个吃罢饭，又走了四五十里，却来到一市镇上，  
地名唤做瑞龙镇，却是个三岔路口。宋江借问那里  
人道：“小人们欲投二龙山、清风镇上，不知从那  
条路去？”

邓俊辉

deng@tsinghua.edu.cn

## 难度与下界

- ❖ 由前述实例可见，同一问题的不同算法，复杂度可能相差悬殊
- ❖ 在可解的前提下，可否谈论问题的**难度**？如何比较不同问题的难度？
- ❖ 问题P若存在算法，则所有算法中**最低**的复杂度称为P的**难度**
- ❖ 为什么要确定问题的难度？给定问题P，如何确定其**难度**？
- ❖ 两个方面着手：设计复杂度更低的算法 + 证明更高的问题**难度下界**
- ❖ 一旦算法的复杂度达到**难度下界**，则说明就大O记号的意义而言，算法已经最优
- ❖ 例如，排序问题下界为 $\Omega(n \log n)$ ，而且是紧的...

## 排序

❖ 任给  $n$  个元素  $\{ R_1, \dots, R_n \}$ , 对应关键码  $\{ K_1, \dots, K_n \}$

需按某种 **次序** 排列

// $\leq$  : 偏序/全序

❖ 亦即, 找出  $\langle 1, \dots, n \rangle$  的一个 **排列**

$\langle i_1, \dots, i_n \rangle$ , 使得

$$K_{i1} \leq \dots \leq K_{in}$$

❖ 例如:  $\{ 3, 1, 4, 1, 5, 9, 2, 6 \}$

$$\rightarrow \{ 1, 1, 2, 3, 4, 5, 6, 9 \}$$

❖ 注意: 此处的关键码集是复集 **multiset**

//可能存在重复关键码

❖ 应用: 25~50%的计算都可归于排序

## 算法分类

- ❖ 直接算法      直接移动元素本身 //元素结构简单时适宜采用
- ❖ 间接算法      下标 + 关键码 + 指针 //元素结构复杂时适宜采用
- ❖ 内部 / 外部    internal / external
- ❖ 脱机 / 在线    offline / online
- ❖ 串行 / 并行    sequential / parallel
- ❖ 确定性 / 随机   deterministic / randomized
- ❖ 基于比较式 / 散列式   comparison-based / hash-based

## 时空性能、稳定性

❖ 多种角度估算的时间、空间复杂度

最好 / best-case

最坏 / worst-case

平均 / average-case

分摊 / amortized

❖ 其中，对最坏情况的估计最保守、最稳妥

因此，首先应考虑**最坏情况最优**的算法

//worst-case optimal

❖ 排序所需的时间，主要取决于

关键码比较的次数 / # {key comparison}

元素交换的次数 / # {data swap}

❖ 就地 **in-place**：除输入数据本身外，只需 $\mathcal{O}(1)$ 附加空间

❖ 稳定 **stability**：关键码相同的元素，在排序后相对位置保持

## 最坏情况最优 + 基于比较

❖ 排序算法，最快能够有多快？

语境1：就最坏情况最优而言

语境2：就某一大类主流算法而言...

❖ 基于比较的算法 ( comparison-based algorithm )

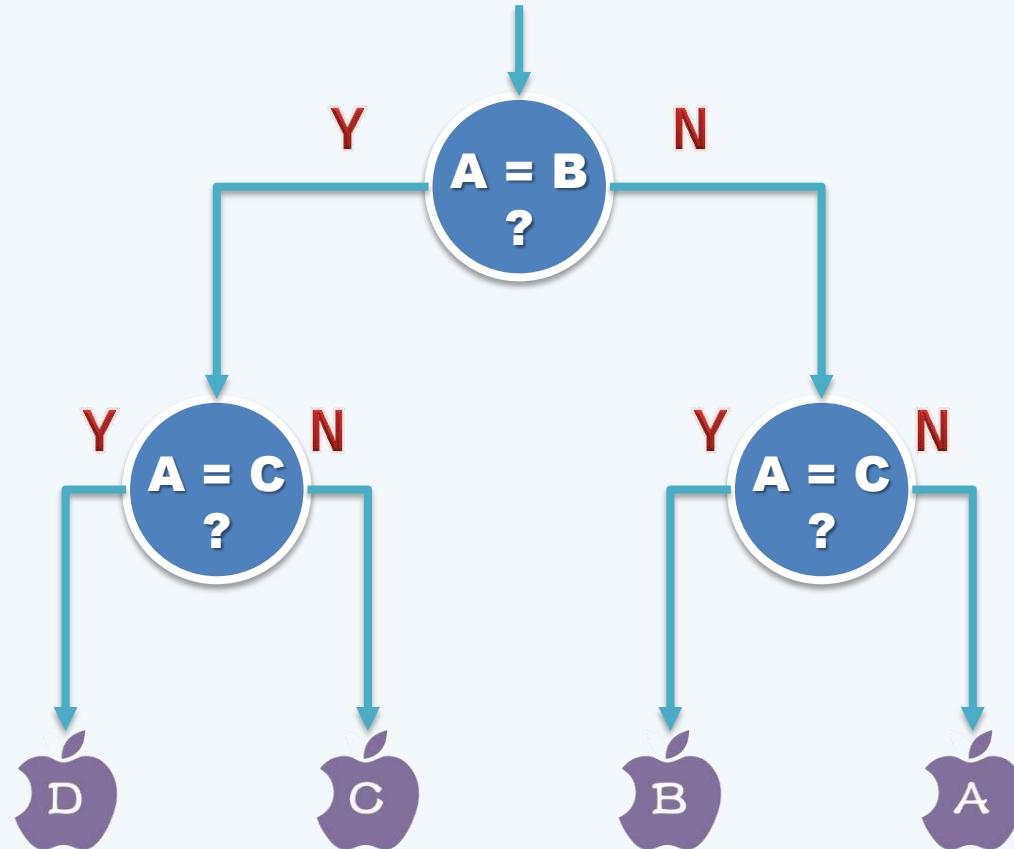
算法执行的进程，取决于一系列的数值（这里即关键码）比对结果

比如，`max()` 和 `bubbleSort()`

❖ 任何CBA在最坏情况下，都需 $\Omega(n \log n)$ 时间才能完成排序

## 判定树

- ❖ 每个CBA算法，都对应于一棵判定树
  - 从根节点通往任一叶子的路径，都对应于算法的某次运行过程
  - 每一可能的输出，都对应于至少一匹叶子（一条通往叶子的路径）



- ❖ 实例： 经过2/4次称量  
必可从4/16只苹果中  
找出唯一的重量不同者

- ❖ 问题： 称量次数可否更少？

## 代数判定树

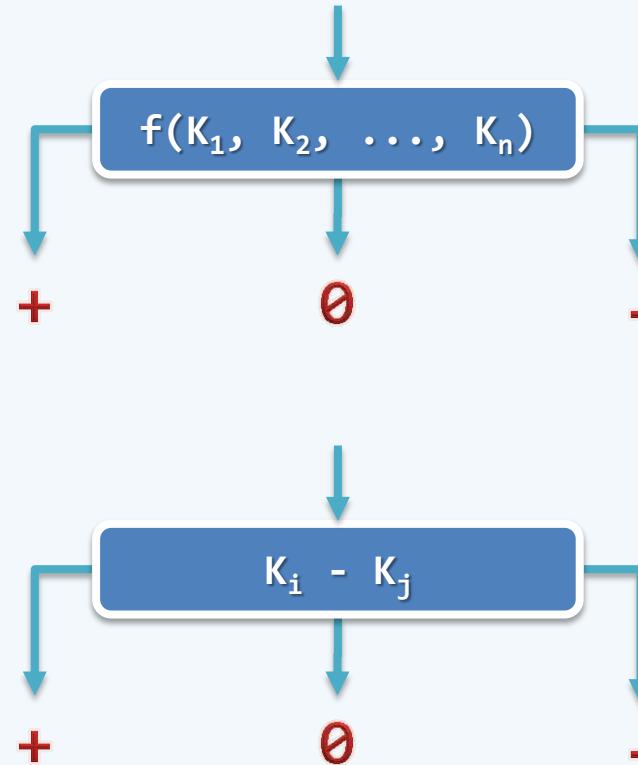
### ❖ Algebraic Decision Tree

- 针对“比较-判定”式算法的计算模型
- 给定输入的规模，将所有可能的输入所对应的一系列判断表示出来

### ❖ 代数判定：

- 使用某一常次数代数多项式  
将任意一组关键码做为变量，对多项式求值
- 根据结果的符号，确定算法推进方向

### ❖ Comparison Tree：最简单的ADT，二元一次多项式，形如： $K_i - K_j$



下界 :  $\Omega(n \log n)$

❖ 比较树是三叉树 (ternary tree)

每个内节点至多三个分支 //+, 0, -

从根节点到叶子的每一条路径，对应于算法的某次运行过程

每匹叶子对应于一个输出 //在此，即排序后的序列

树高 = 最坏情况下所需的比较次数 //最坏情况最优

树高的下界 = 所有CBA的时间复杂度下界

❖ 对n个元素进行排序的任何一棵ADT，高度至少为 $\Omega(n \log n)$

#叶子  $\geq$  #可能的输出 = n个元素可能的排列 =  $n!$

树高  $\geq \log_3 n! = \log_3 e \cdot \ln(n!)$

$= \log_3 e \cdot [n \ln n - n + O(\ln n)] = \Omega(n \log n)$  //Stirling approximation

❖ 上述结论，可进一步推广至理想平均情况、随机情况（概率 $\geq 25\%$ ）...

## 1. 绪论

下界  
归约

邓俊辉

不怕不识货，就怕货比货

deng@tsinghua.edu.cn

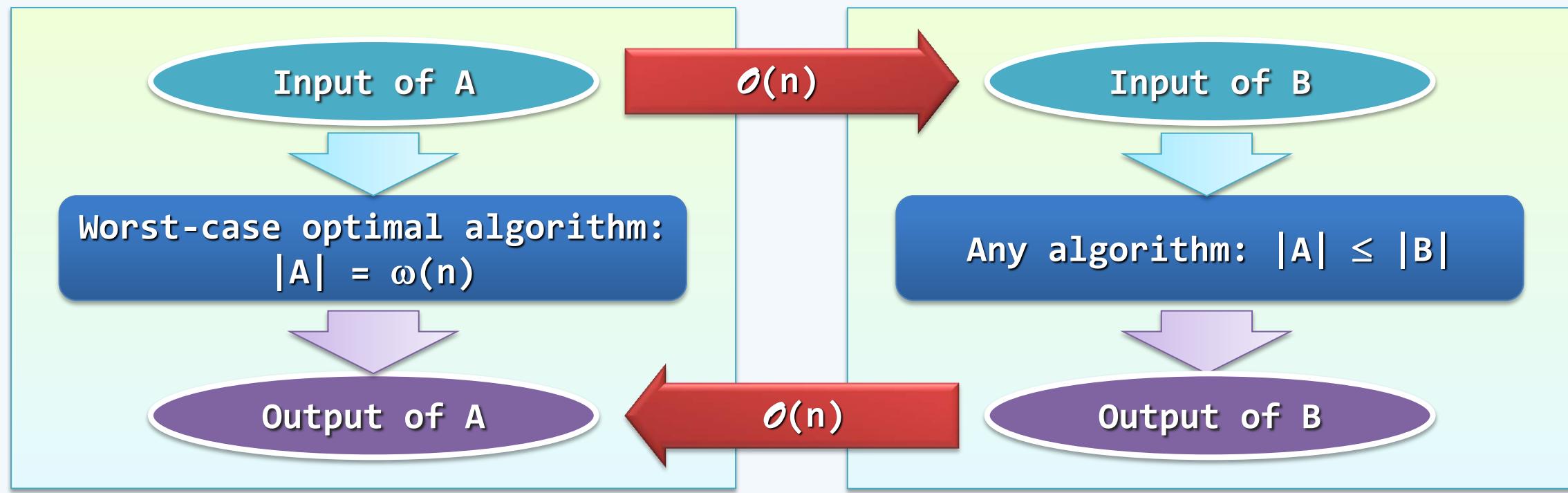
## 线性归约

❖ 除了(代数)判定树, 归约 reduction 也是确定下界的有力工具

$\mathcal{O}(n \log n)$  linear-time reduction

NP-complete/P polynomial-time reduction

P-SPACE complete polynomial-time many-one reduction



## 实例

- ❖ 【Red-Blue Matching】平面上任给n个红色点和n个蓝色点，如何以互不相交的线段配对联接

$\text{Sorting} \leq_N \text{Red-Blue Matching}$

- ❖ 【Element Uniqueness】任意n个实数中，是否包含雷同？ //EU的下界为 $\Omega(n \log n)$

$\text{EU} \leq_N \text{Closest Pair}$

- ❖ 【Integer Element Uniqueness】任意n个整数中，是否包含雷同？ //下界亦是 $\Omega(n \log n)$

$\text{IEU} \leq_N \text{Segment Intersection Detection}$

- ❖ 【Set Disjointness】任意一对集合A和B，是否存在公共元素？ //下界亦是 $\Omega(n \log n)$

$\text{SD} \leq_N \text{Diameter}$

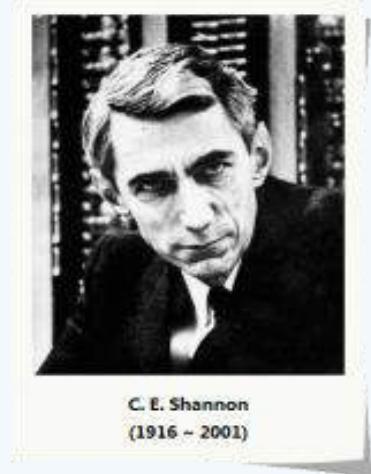
## 熵与下界

❖ 热力学第二定律：能量不会自动地从低温物体传向高温物体

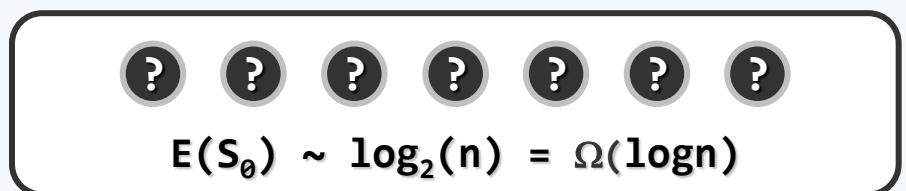
❖ Shannon : 数据系统S中蕴含的信息量，可由**信息熵度量**

$$\text{Entropy}(S) \sim \log_2 N \quad (N = S\text{可能的状态总数})$$

❖ 热系统的**熵减少**，都须付出一定的**能量**



数据系统的**信息熵减少**，也须付出一定的**计算量**



鉴别

7个已知数的集合

$E(S_1) \sim \log_2 1 = 0$

7个无序数的集合

$E(S_0) \sim \log_2(n!) = \Omega(n \log n)$

排序

7个有序数的集合

$E(S_1) \sim \log_2 1 = 0$

Data Structures & Algorithms, Tsinghua University

3

## 熵与下界

### ❖ Landauer's principle

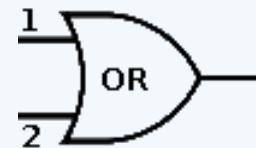
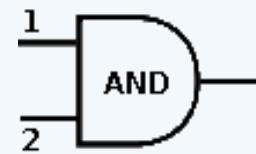
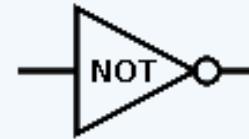
信息的减少或丢失，必伴随着熵的增加，并发出相等的热量

就最低功耗而言，AND和OR逻辑门必高于NOT门

### ❖ “幸福的人都是一样的，不幸福的人却各有各的不幸”

从计算的角度看，幸福的可能状态，要远少于不幸

前者的熵远小于后者，故从后者到前者，需要付出巨大的努力



## 2. 向量

抽象数据类型

接口与实现

邓俊辉

deng@tsinghua.edu.cn

## Abstract Data Type vs. Data Structure

❖ **抽象数据类型** = 数据模型 + 定义在该模型上的一组操作

抽象定义

外部的逻辑特性

操作&语义

一种定义

不考虑时间复杂度

不涉及数据的存储方式

**数据结构** = 基于某种特定语言，实现ADT的一整套算法

具体实现

内部的表示与实现

完整的算法

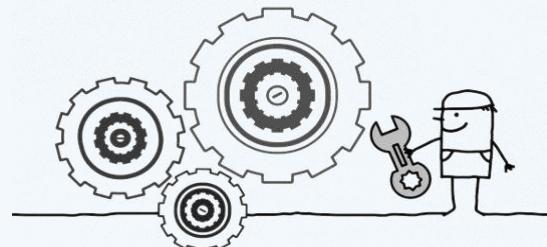
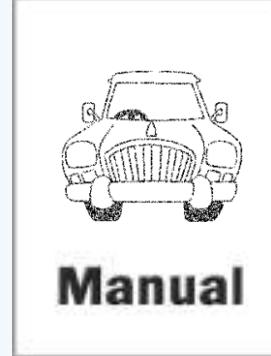
多种实现

与复杂度密切相关

要考虑数据的具体存储机制



Application



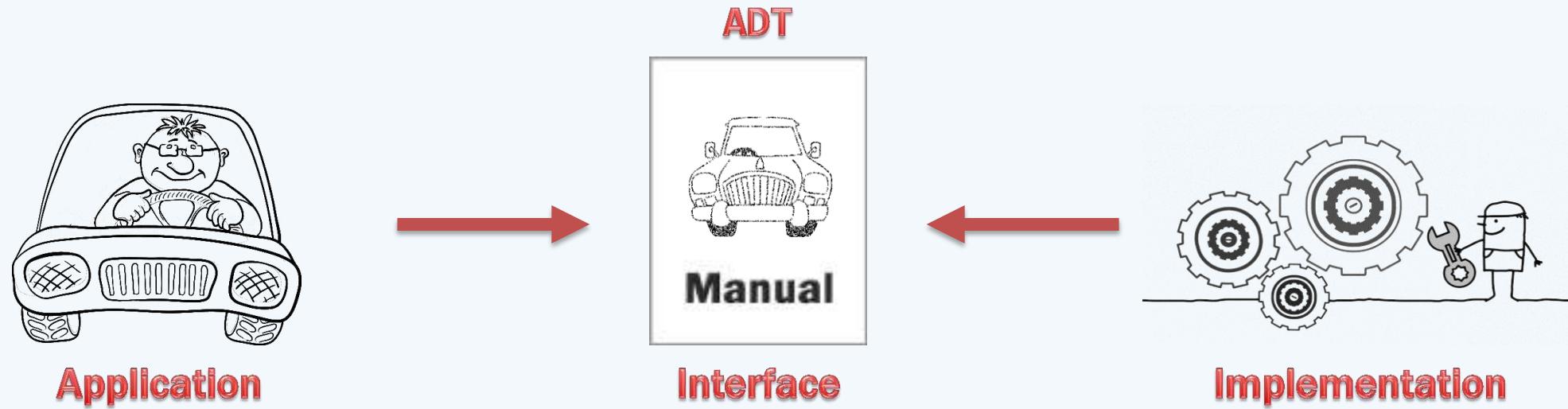
Implementation

## Application = Interface × Implementation

❖ 在数据结构的**具体实现**与**实际应用**之间，ADT就分工与接口制定了统一的规范

**实现**：高效率地兑现数据结构的ADT接口操作 //做冰箱、造汽车

**应用**：便捷地通过操作接口使用数据结构 //用冰箱、开汽车

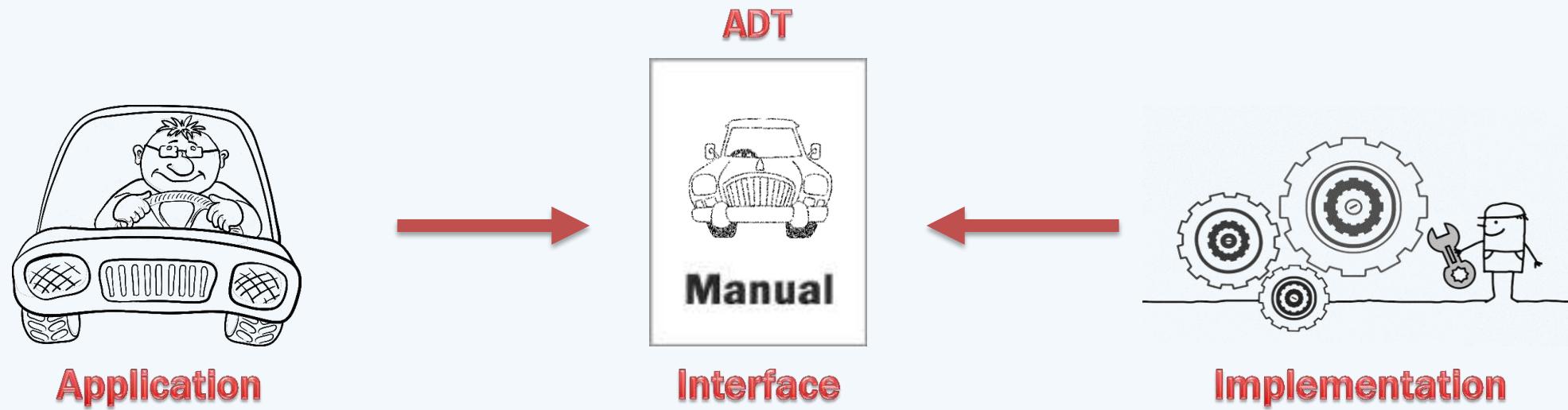


## Application = Interface × Implementation

❖ 按照ADT规范： 高层**算法设计**者与底层**数据结构实现**者可高效地分工协作

不同的算法与数据结构可以**任意组合**，便于确定最优配置

每种操作接口只需统一地实现一次，代码篇幅缩短，软件**复用**度提高



## 2. 向量

抽象数据类型

从数组到向量

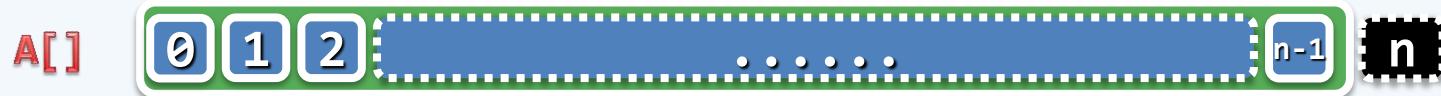
邓俊辉

deng@tsinghua.edu.cn

## 循秩访问

❖ C/C++语言中，数组A[ ]中的元素与[0, n)内的编号一一对应

$A[0], A[1], A[2], \dots, A[n-1]$



❖ 反之，每个元素均由（非负）**编号**唯一指代，并可**直接**访问

$A[i]$ 的物理地址 =  $A + i \times s$ ,  $s$ 为单个元素占用的空间量

故亦称作线性数组（linear array）

❖ 向量是数组的抽象与泛化，由一组元素按线性次序**封装**而成

各元素与[0, n)内的**秩**（rank）一一对应

typedef int Rank; //循秩访问 (call-by-rank)

操作、管理维护更加简化、统一与安全

元素类型可灵活**选取**，便于**定制**复杂数据结构

`//Vector< PFCTree* > pfcForest;`

## 向量ADT接口

操作	功能	适用对象
<code>size()</code>	报告向量当前的规模（元素总数）	向量
<code>get(r)</code>	获取秩为r的元素	向量
<code>put(r, e)</code>	用e替换秩为r元素的数值	向量
<code>insert(r, e)</code>	e作为秩为r元素插入，原后继依次后移	向量
<code>remove(r)</code>	删除秩为r的元素，返回该元素原值	向量
<code>disordered()</code>	判断所有元素是否已按非降序排列	向量
<code>sort()</code>	调整各元素的位置，使之按非降序排列	向量
<code>find(e)</code>	查找目标元素e	向量
<code>search(e)</code>	查找e，返回不大于e且秩最大的元素	有序向量
<code>deduplicate(), uniquify()</code>	剔除重复元素	向量/有序向量
<code>traverse()</code>	遍历向量并统一处理所有元素	向量

## ADT操作实例

操作	输出	向量组成(自左向右)	操作	输出	向量组成(自左向右)
初始化			disordered()	3	4 3 7 4 9 6
insert(0, 9)		9	find(9)	4	4 3 7 4 9 6
insert(0, 4)		4 9	find(5)	-1	4 3 7 4 9 6
insert(1, 5)		4 5 9	sort()		3 4 4 6 7 9
put(1, 2)		4 2 9	disordered()	0	3 4 4 6 7 9
get(2)	9	4 2 9	search(1)	-1	3 4 4 6 7 9
insert(3, 6)		4 2 9 6	search(4)	2	3 4 4 6 7 9
insert(1, 7)		4 7 2 9 6	search(8)	4	3 4 4 6 7 9
remove(2)	2	4 7 9 6	search(9)	5	3 4 4 6 7 9
insert(1, 3)		4 3 7 9 6	search(10)	5	3 4 4 6 7 9
insert(3, 4)		4 3 7 4 9 6	uniquify()		3 4 6 7 9
size()	6	4 3 7 4 9 6	search(9)	4	3 4 6 7 9

## 2. 向量

抽象数据类型

模板类

邓俊辉

deng@tsinghua.edu.cn

```

template <typename T> class Vector { //向量模板类

private: Rank _size; int _capacity; T* _elem; //规模、容量、数据区

protected:

/* ... 内部函数 */

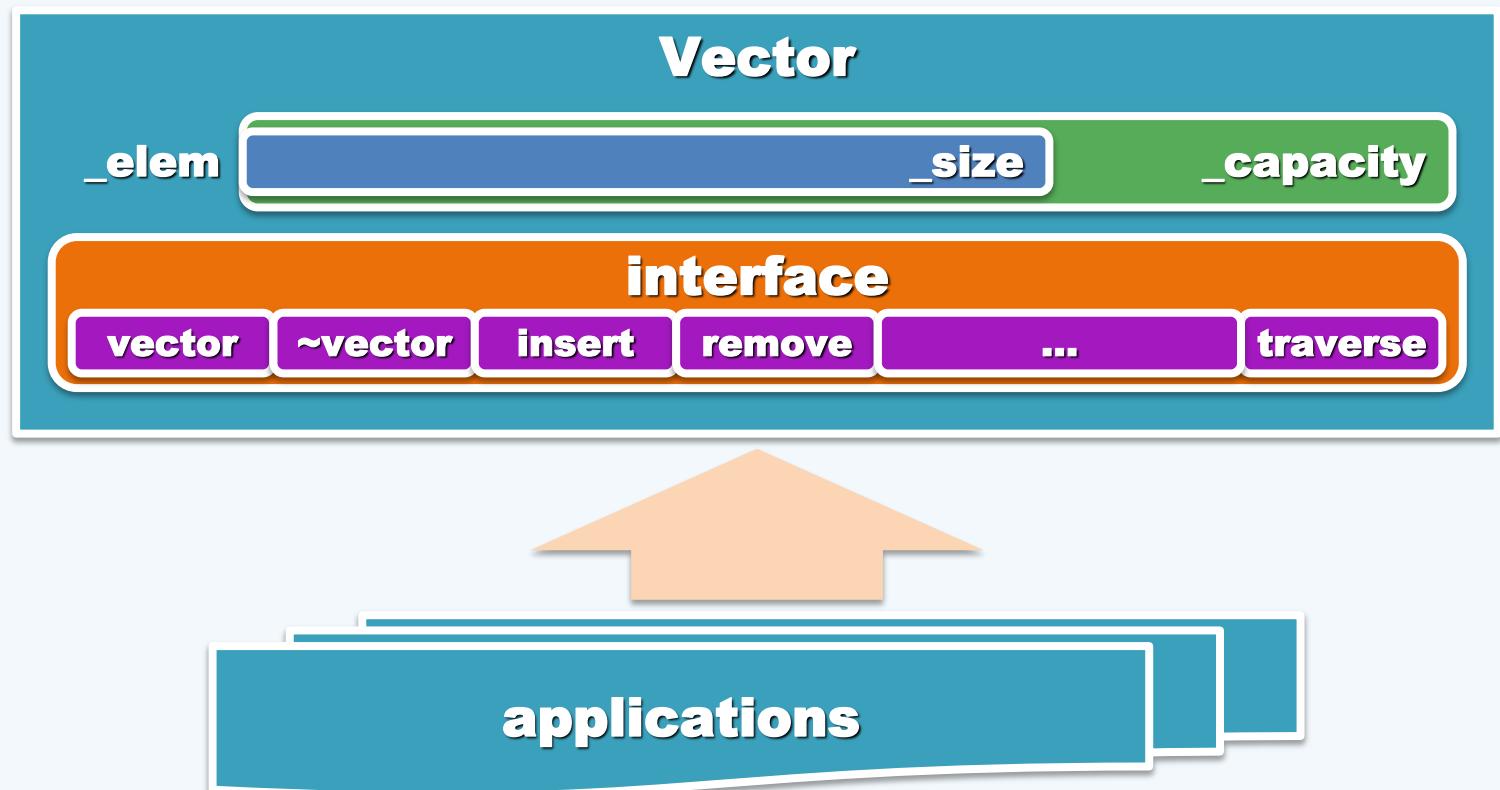
public:

/* ... 构造函数 */
/* ... 析构函数 */
/* ... 只读接口 */
/* ... 可写接口 */
/* ... 遍历接口 */

};

}

```



## 构造 + 析构

```
❖ #define DEFAULT_CAPACITY 3 //默认初始容量(实际应用中可设置为更大)

❖ Vector( int c = DEFAULT_CAPACITY )

{ _elem = new T[ _capacity = c ]; _size = 0; } //默认

❖ Vector( T const * A, Rank lo, Rank hi ) //数组区间复制

{ copyFrom( A, lo, hi ); }

Vector( Vector<T> const& V, Rank lo, Rank hi )

{ copyFrom( V._elem, lo, hi ); } //向量区间复制

Vector( Vector<T> const& V )

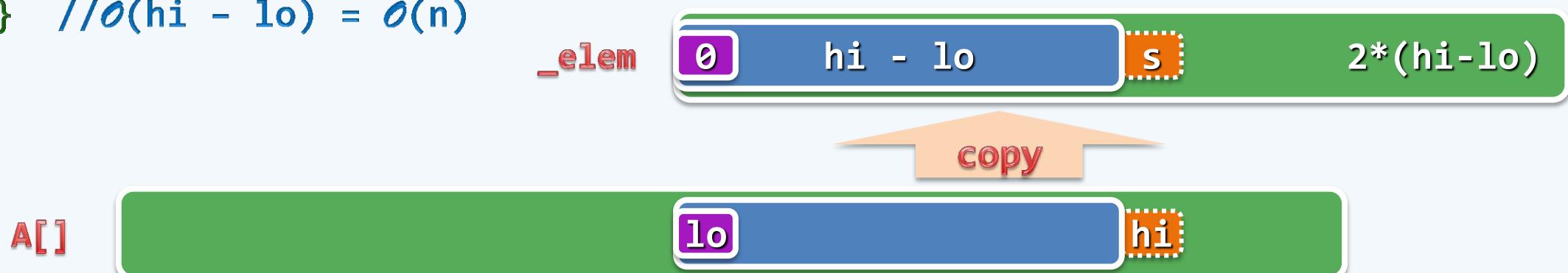
{ copyFrom( V._elem, 0, V._size ); } //向量整体复制

❖ ~Vector() { delete [] _elem; } //释放内部空间
```

## 基于复制的构造

❖ template <typename T> //T为基本类型，或已重载赋值操作符'='

```
void Vector<T>::copyFrom( T const * A, Rank lo, Rank hi ) {
    _elem = new T[ _capacity = 2*(hi - lo) ]; //分配空间
    _size = 0; //规模清零
    while ( lo < hi ) //A[lo, hi)内的元素逐一
        _elem[ _size++ ] = A[ lo++ ]; //复制至_elem[0, hi - lo)
} //O(hi - lo) = O(n)
```



## 2. 向量

### 可扩充向量 算法

爱我要加倍，不爱我没所谓  
早点来排队，否则你没座位

邓俊辉

deng@tsinghua.edu.cn

## 静态空间管理



❖ 开辟内部数组 `_elem[]` 并使用一段地址连续的物理空间

`_capacity` : 总容量

`_size` : 当前的实际规模n

❖ 若采用静态空间管理策略，容量 `_capacity` 固定，则有明显的不足...

## 空间效率



- ❖ - 上溢/**overflow** : `_elem[ ]`不足以存放所有元素，尽管此时系统往往仍有足够的空间
- 下溢/**underflow** ) : `_elem[ ]`中的元素寥寥无几
- 装填因子/**load factor** ) :  $\lambda = \frac{\text{_size}}{\text{_capacity}} << 50\%$
- ❖ 更糟糕的是，一般的应用环境中难以准确**预测**空间的需求量
- ❖ 可否使得向量可随实际需求动态**调整**容量，并同时保证高效率？

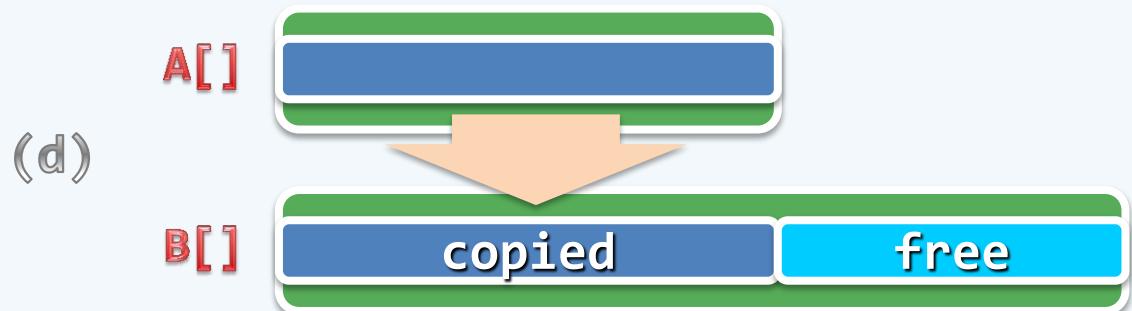
## 动态空间管理

### ❖ 蝉的哲学：

- 身体每经过一段时间的生长以致无法为外壳容纳
- 即蜕去原先的外壳，代之以...

### ❖ 在即将发生上溢时

适当扩大内部数组的容量



## 扩容算法

❖ `template <typename T> void Vector<T>::expand() { //向量空间不足时扩容`

```

if ( _size < _capacity ) return; //尚未满员时，不必扩容

_capacity = max( _capacity, DEFAULT_CAPACITY ); //不低于最小容量

T* oldElem = _elem; _elem = new T[ _capacity <<= 1 ]; //容量加倍

for ( int i = 0; i < _size; i++ ) //复制原向量内容

    _elem[i] = oldElem[i]; //T为基本类型，或已重载赋值操作符'='

delete [] oldElem; //释放原空间

} //得益于向量的封装，尽管扩容之后数据区的物理地址有所改变，却不易出现野指针

```

❖ 为何必须采用**容量加倍**策略呢？其它策略是否可行？

## 2. 向量

可扩充向量

分摊

...在他的心理上，他总以为北平是天底下最可靠的大城，不管有什么灾难，到三个月必定灾消难满，而后诸事大吉。北平的灾难恰似一个人免不了有些头疼脑热，过几天自然会好了的。

邓俊辉

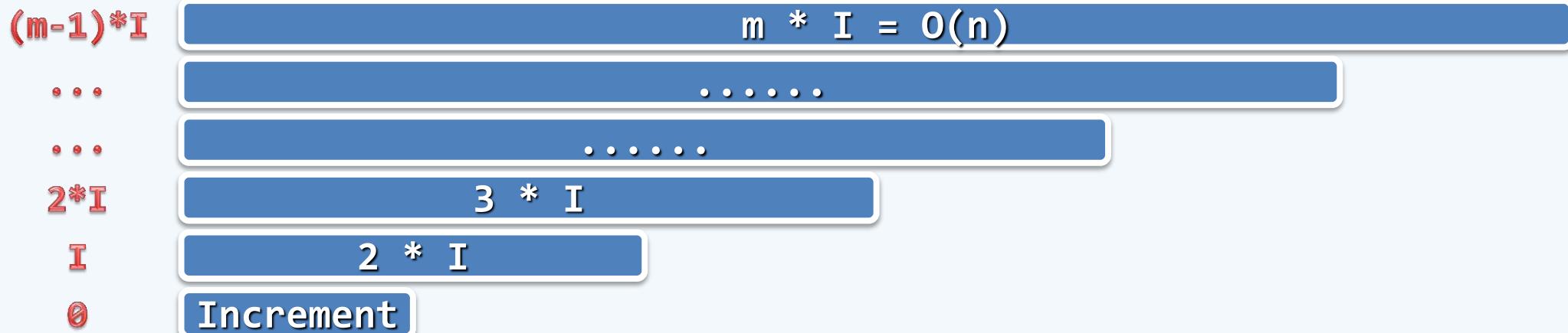
deng@tsinghua.edu.cn

## 容量递增策略

- ❖ `T* oldElem = _elem; _elem = new T[_capacity += INCREMENT]; //追加固定增量`
- ❖ 最坏情况：在初始容量 $\theta$ 的空向量中，连续插入 $n = m*I \gg 2$ 个元素...
- ❖ 于是，在第1、 $I + 1$ 、 $2I + 1$ 、 $3I + 1$ 、...次插入时，都需扩容
- ❖ 即便不计申请空间操作，各次扩容过程中复制原向量的时间成本依次为

$\theta, I, 2I, \dots, (m-1)*I$  //算术级数

总体耗时 =  $I * (m-1) * m/2 = O(n^2)$ ，每次扩容的分摊成本为 $O(n)$

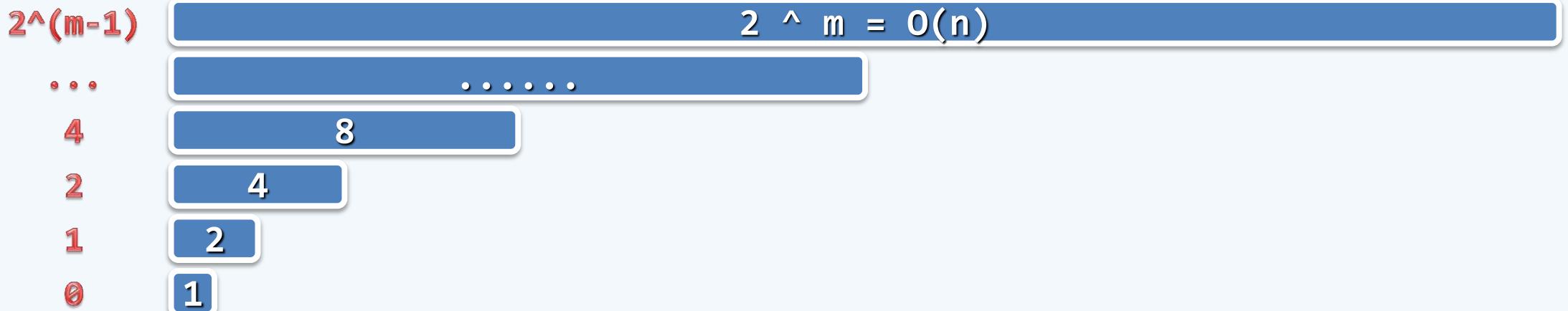


## 容量加倍策略

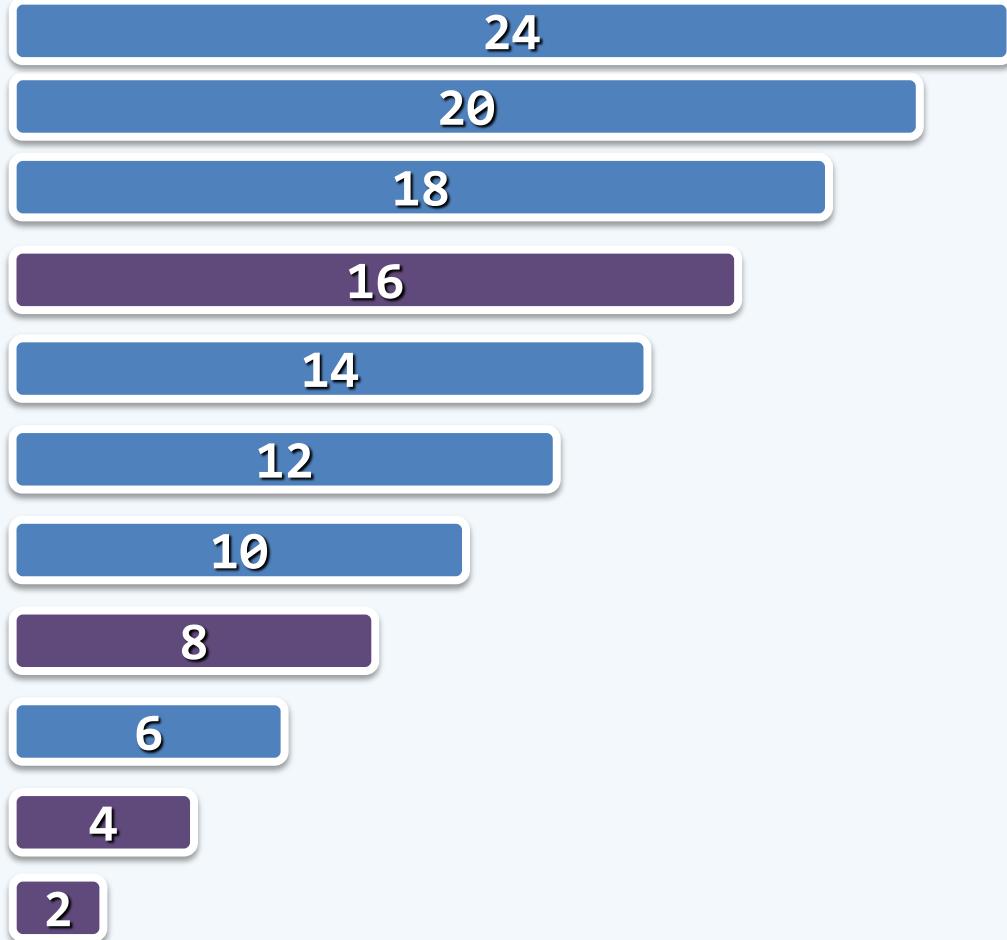
- ❖ `T* oldElem = _elem; _elem = new T[_capacity <<= 1];` //容量加倍
- ❖ 最坏情况：在初始容量1的满向量中，连续插入 $n = 2^m \gg 2$ 个元素...
- ❖ 于是，在第1、2、4、8、16、...次插入时都需扩容
- ❖ 各次扩容过程中复制原向量的时间成本依次为

$1, 2, 4, 8, \dots, 2^m = n$  //几何级数

总体耗时 =  $\mathcal{O}(n)$ ，每次扩容的分摊成本为 $\mathcal{O}(1)$



## 对比



	递增策略	倍增策略
累计增容时间	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
分摊增容时间	$\mathcal{O}(n)$	$\mathcal{O}(1)$
装填因子	$\approx 100\%$	$> 50\%$

## 平均分析 vs. 分摊分析

### ❖ 平均复杂度或期望复杂度 (average/expected complexity)

根据数据结构各种操作出现概率的分布，将对应的成本加权平均

各种可能的操作，作为独立事件分别考查

割裂了操作之间的相关性和连贯性

往往不能准确地评判数据结构和算法的真实性能

### ❖ 分摊复杂度 (amortized complexity)

对数据结构连续地实施足够多次操作，所需总体成本分摊至单次操作

从实际可行的角度，对一系列操作做整体的考量

更加忠实地刻画了可能出现的操作序列

更为精准地评判数据结构和算法的真实性能

### ❖ 后面将看到更多、更复杂的例子

## 2. 向量

无序向量

基本操作

它污浊，它美丽，它衰老，它活泼，它杂乱，  
它安闲，它可爱，它是伟大的夏初的北平

邓俊辉

deng@tsinghua.edu.cn

## 元素访问

- ❖ 似乎不是问题：通过 `v.get(r)` 和 `v.put(r, e)` 接口，已然可以读、写向量元素
- ❖ 但就 **便捷性** 而言，远不如数组元素的访问方式：`A[r]` // 可否沿用借助下标的访问方式？

- ❖ 可以！为此，需**重载**下标操作符“`[ ]`”

```
template <typename T> //0 <= r < _size
T & Vector<T>::operator[]( Rank r ) const { return _elem[ r ]; }
```

- ❖ 此后，对外的 `v[ r ]` 即对应于内部的 `v._elem[ r ]`

右值：`T x = v[r] + u[s] * w[t];`

左值：`v[r] = (T) (2*x + 3);`

- ❖ 为便于讲解，这里采用了简易的方式处理意外和错误（比如，入口参数越界等）

实际应用中，应采用更为严格的方式

## 插入

❖ template <typename T> //e作为秩为r元素插入， $0 \leq r \leq \text{size}$

```
Rank Vector<T>::insert( Rank r, T const & e ) { //O(n - r)
    expand(); //若有必要，扩容
    for ( int i = _size; i > r; i-- ) //自后向前
        _elem[i] = _elem[i - 1]; //后继元素顺次后移一个单元
    _elem[r] = e; _size++; return r; //置入新元素，更新容量，返回秩
}
```

(a) [0, n) : may be full

(b) [0, r) [r, n) expanded if necessary

right shift

(c)  (r, n]

(d)  e

## 区间删除

❖ template <typename T> //删除区间[lo, hi), 0 <= lo <= hi <= size

```
int Vector<T>::remove( Rank lo, Rank hi ) { //O(n - hi)
    if ( lo == hi ) return 0; //出于效率考虑，单独处理退化情况
    while ( hi < _size ) _elem[ lo ++ ] = _elem[ hi ++ ]; // [hi, _size)顺次前移
    _size = lo; shrink(); //更新规模，若有必要则缩容
    return hi - lo; //返回被删除元素的数目
}
```



## 单元素删除

- ❖ 可以视作区间删除操作的特例 :  $[r] = [r, r + 1)$
- ❖ template <typename T> // 删除向量中秩为r的元素,  $0 \leq r < \text{size}$

```

T Vector<T>::remove( Rank r ) { //  $\Theta(n - r)$ 
    T e = _elem[r]; // 备份被删除元素
    remove( r, r + 1 ); // 调用区间删除算法
    return e; // 返回被删除元素
}

```

- ❖ 反过来, 基于remove(r)接口, 通过反复的调用, 实现remove(lo, hi)呢?
- ❖ 每次循环耗时正比于删除区间的后缀长度 =  $n - hi = \Theta(n)$   
而循环次数等于区间宽度 =  $hi - lo = \Theta(n)$   
如此, 将导致总体  $\Theta(n^2)$  的复杂度

## 2. 向量

无序向量

查找

他便站将起来，背着手踱来踱去，侧眼把那些人逐个个觑将去，内中一个果然衣领上挂着一寸来长短彩线头。

邓俊辉

deng@tsinghua.edu.cn

## 无序向量：判等器

```
❖ template <typename K, typename V> struct Entry { //词条模板类  
    K key; V value; //关键码、数值  
  
    Entry ( K k = K(), V v = V() ) : key ( k ), value ( v ) {}; //默认构造函数  
  
    Entry ( Entry<K, V> const& e ) : key ( e.key ), value ( e.value ) {}; //克隆  
  
    bool operator== ( Entry<K, V> const& e ) { return key == e.key; } //等于  
    bool operator!= ( Entry<K, V> const& e ) { return key != e.key; } //不等  
  
    /* ... */  
  
    /* ... */  
  
};
```

## 有序向量：比较器

❖ template <typename K, typename V> struct Entry { //词条模板类

```
K key; V value; //关键码、数值
```

```
Entry ( K k = K(), V v = V() ) : key ( k ), value ( v ) {}; //默认构造函数
```

```
Entry ( Entry<K, V> const& e ) : key ( e.key ), value ( e.value ) {}; //克隆
```

```
bool operator== ( Entry<K, V> const& e ) { return key == e.key; } //等于
```

```
bool operator!= ( Entry<K, V> const& e ) { return key != e.key; } //不等于
```

```
bool operator< ( Entry<K, V> const& e ) { return key < e.key; } //小于
```

```
bool operator> ( Entry<K, V> const& e ) { return key > e.key; } //大于
```

}; //得益于比较器和判等器，从此往后，不必严格区分词条及其对应的关键码

## 顺序查找

❖ template <typename T> //在命中多个元素时可返回秩最大者

Rank Vector<T>:: // $\theta(hi - lo) = \theta(n)$

```
find( T const & e, Rank lo, Rank hi ) const { //0 <= lo < hi <= _size
    while ( (lo < hi--) && (e != _elem[hi]) ); //逆向查找
    return hi; //hi < lo意味着失败；否则hi即命中元素的秩
} //Excel::match(e, range, type)
```



❖ 输入敏感 ( input-sensitive ) : 最好  $\theta(1)$  , 最差  $\theta(n)$

## 2. 向量

### (c3) 无序向量 - 去重

邓俊辉

世雷同而炫曜兮，何毁譽之昧昧！

deng@tsinghua.edu.cn

❖ 应用实例：网络搜索的局部结果经过去重操作，汇总为最终报告

❖ template <typename T> //删除重复元素，返回被删除元素数目

```
int Vector<T>::deduplicate() { //繁琐版 + 错误版  
    int oldSize = _size; //记录原规模  
    Rank i = 1; //从_elem[1]开始  
    while ( i < _size ) //自前向后逐一考查各元素_elem[i]  
        find( _elem[i], 0, i ) < 0 ? //在前缀中寻找雷同者  
            i++ //若无雷同则继续考查其后继  
        : remove(i); //否则删除雷同者(至多一个？！)  
    return oldSize - _size; //向量规模变化量，即删除元素总数  
}
```

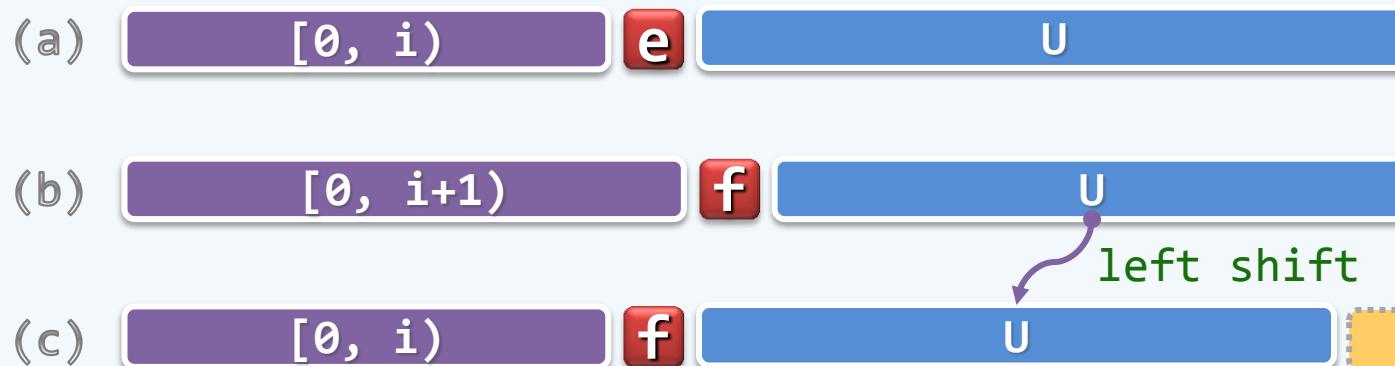
## 正确性

❖ 易见，凡被剔除者均为重复元素（不多）

故只需证明，算法不致遗漏重复元素（不少）

❖ 不变性：在当前元素 $v[i]$ 的前缀 $v[0, i)$ 中，各元素彼此互异

❖ 初始 $i = 1$ 时自然成立；后续的一般情况...



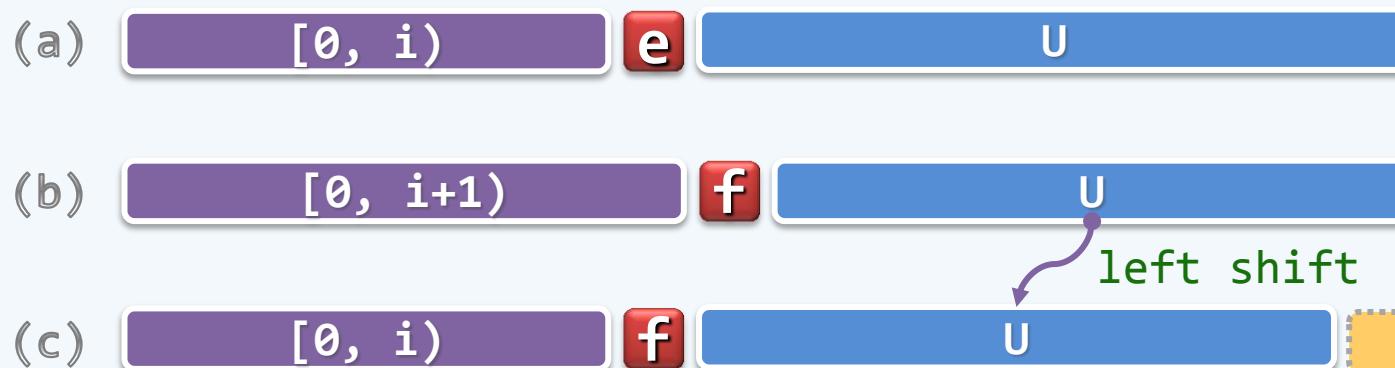
## 正确性

❖ **单调性**：随着反复的while迭代

1) 当前元素**前缀**的长度单调非降，且迟早增至`_size //1`和2)对应

2) 当前元素**后缀**的长度单调下降，且迟早减至0 //2)更易把握

故算法**必然终止**，且至多迭代 $O(n)$ 轮



## 复杂度

❖ 每轮迭代中 `find()` 和 `remove()` 累计耗费 **线性** 时间，总体  $\mathcal{O}(n^2)$  // 可进一步优化，比如...

- 仿照 `uniqueify()` 高效版的思路

(a) [0, i) e U

元素移动的次数可降至  $\mathcal{O}(n)$

(b) [0, i+1) f U

但比较次数依然是  $\mathcal{O}(n^2)$

(c) [0, i) f U

U

U

left shift



- 先对需删除的重复元素做标记，然后再统一删除

稳定性保持，但因查找长度更长，从而导致更多的比对操作

- V. `sort().uniqueify()`：简明实现最优的  $\mathcal{O}(n \log n)$

// 下节

## 2. 向量

### (c4) 无序向量

- 遍历

可是我们惊愕地发现，“发动起来的群众”，就像通了电的机器人，都随着按钮统一行动，都不是个人了

邓俊辉

deng@tsinghua.edu.cn

## 遍历

❖ 对向量中的每一元素，统一实施 `visit` 操作

如何指定 `visit` 操作？如何将其传递到向量内部？

❖ `template <typename T> //利用函数指针机制，只读或局部性修改`

```
void Vector<T>::traverse( void ( * visit )( T & ) )
{ for ( int i = 0; i < _size; i++ ) visit( _elem[i] ); }
```

❖ `template <typename T> template <typename VST> //利用函数对象机制，可全局性修改`

```
void Vector<T>::traverse( VST & visit )
{ for ( int i = 0; i < _size; i++ ) visit( _elem[i] ); }
```

❖ 体会两种方法的优劣

## 实例

❖ 比如，为统一将向量中所有元素分别加一，只需...

❖ 首先，实现一个可使单个T类型元素加一的类

```
template <typename T> //假设T可直接递增或已重载操作符“++”

struct Increase //函数对象：通过重载操作符“()”实现
{
    virtual void operator()( T & e ) { e++; } //加一
```

❖ 此后...

```
template <typename T> void increase( Vector<T> & v )
{
    v.traverse( Increase<T>() ); } //即可以之为基本操作遍历向量
```

❖ 作为练习，可模仿此例，实现统一**减一**、**加倍**，甚至**求和**等遍历功能

## 2. 向量

### (d1) 有序向量

#### - 唯一化

贾政道：“我要你另换个主意，不许雷同了前人，只做个破题也使得。”宝玉只得答应着，低头搜索枯肠

邓俊辉

deng@tsinghua.edu.cn

## 有序性及其甄别

❖ 与起泡排序算法的理解相同：有序 / 无序 序列中，任意 / 总有一对相邻元素顺序 / 逆序

❖ 因此，相邻逆序对的数目，可用以度量向量的逆序程度

❖ `template <typename T>`

```
int Vector<T>::disordered() const { //统计向量中的逆序相邻元素对
    int n = 0; //计数器
    for ( int i = 1; i < _size; i++ ) //逐一检查各对相邻元素
        n += (_elem[i - 1] > _elem[i]); //逆序则计数
    return n; //向量有序当且仅当n = 0
} //若只需判断是否有序，则首次遇到逆序对之后，即可立即终止
```

❖ 无序向量经预处理转换为有序向量之后，相关算法多可优化

## 低效算法

❖ 观察：在有序向量中，重复的元素必然相互紧邻构成一个区间

因此，每一区间只需保留单个元素即可



❖ `template <typename T> int Vector<T>::uniquify() {`

`int oldSize = _size; int i = 1; //从首元素开始`

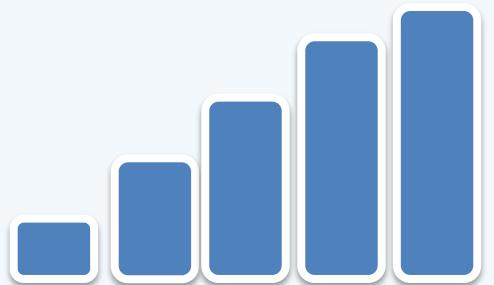
`while ( i < _size ) //从前向后，逐一比对各对相邻元素`

`//若雷同，则删除后者；否则，转至下一元素`

`_elem[i - 1] == _elem[i] ? remove( i ) : i++;`

`return oldSize - _size; //向量规模变化量，即删除元素总数`

`} //注意：其中_size的减小，由remove()隐式地完成`



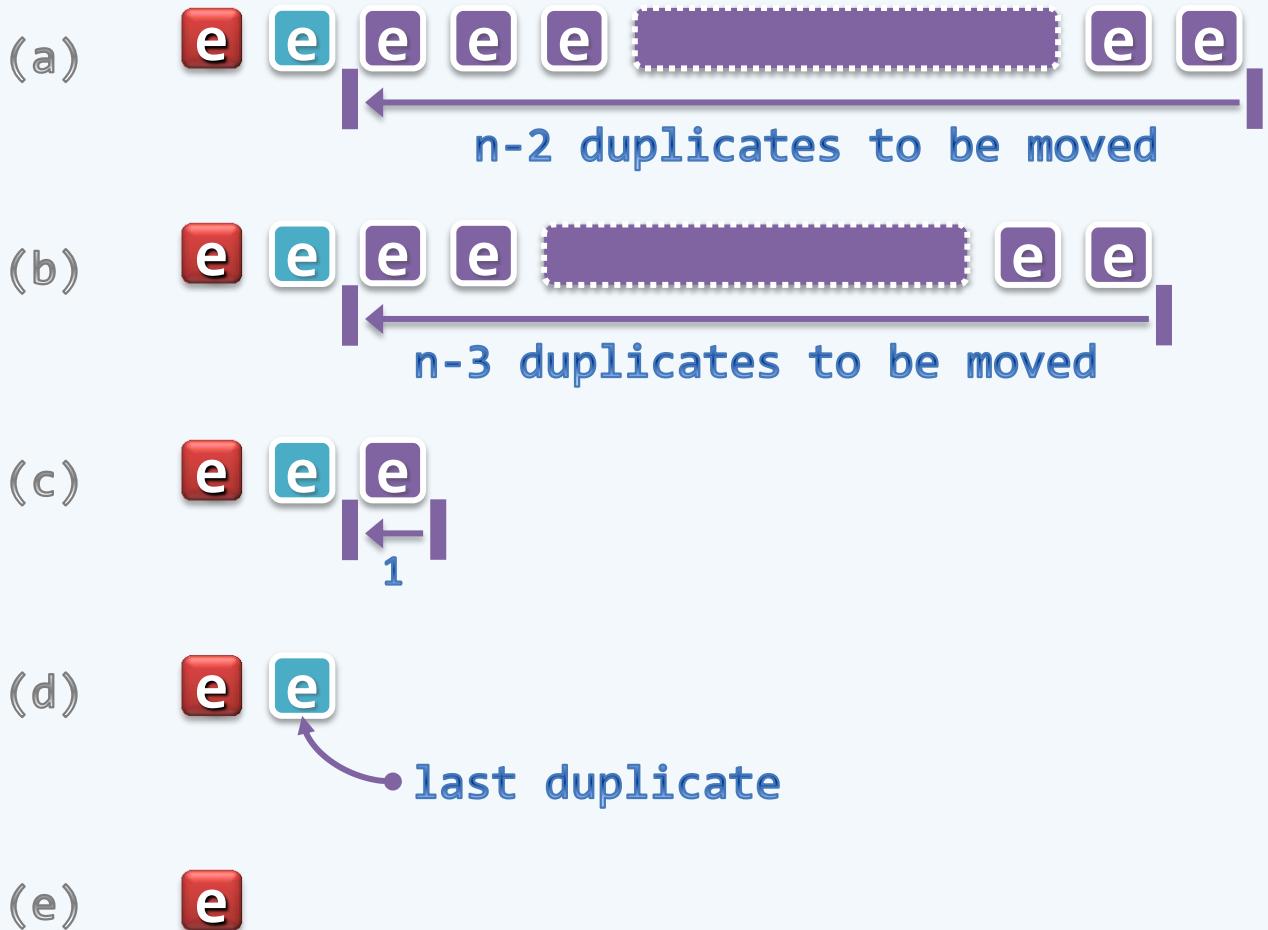
## 复杂度

❖ 运行时间主要取决于while循环次数

共计：`_size - 1 = n - 1`

❖ 最坏情况下，每次都需要调用`remove()`

- 各耗时  $\mathcal{O}(n-1) \sim \mathcal{O}(1)$
- 累计  $\mathcal{O}(n^2)$
- 尽管省去`find()`，总体竟与  
无序向量的`deduplicate()`相同



❖ 反思：低效的根源在于，同一元素可作为被删除元素的后继多次前移

启示：若能以重复区间为单位，**成批**删除雷同元素，性能必将改进...

## 高效算法

```

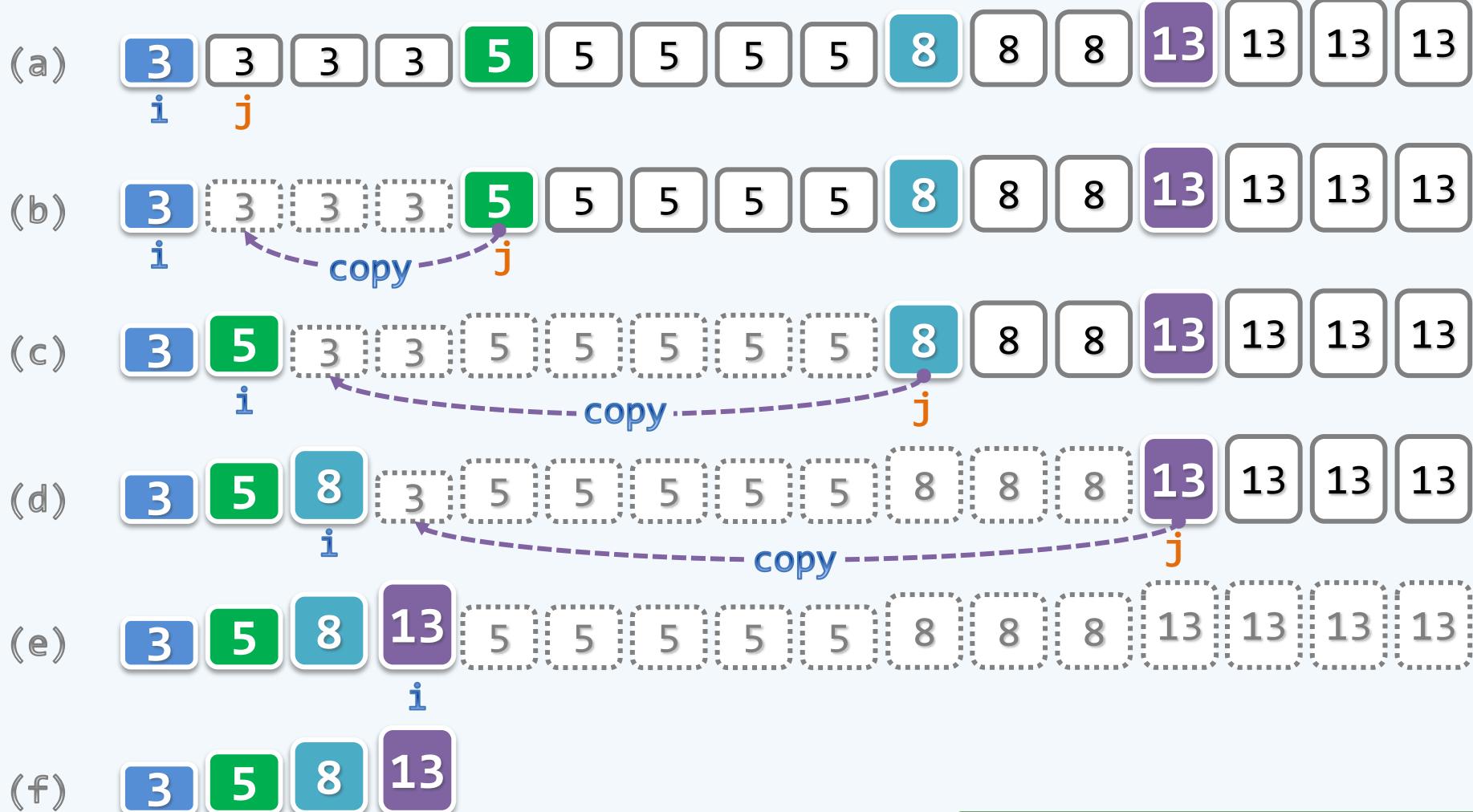
❖ template <typename T> int Vector<T>::uniquify() {
    Rank i = 0, j = 0; //各对互异“相邻”元素的秩
    while ( ++ j < _size ) //逐一扫描，直至末元素
        //跳过雷同者；发现不同元素时，向前移至紧邻于前者右侧
        if ( _elem[ i ] != _elem[ j ] ) _elem[ ++ i ] = _elem[ j ];
    _size = ++i; shrink(); //直接截除尾部多余元素
    return j - i; //向量规模变化量，即被删除元素总数
} //注意：通过remove(lo, hi)批量删除，依然不能达到高效率

```



## 实例与复杂度

◆ 共计 $n - 1$ 次迭代，每次常数时间，累计 $\Theta(n)$ 时间



## 课后

- ❖ 较之无序向量，有序向量的唯一化可以更快地完成其中的原因，如何理解和解释？

## 2. 向量

### 有序向量

### 二分查找（版本A）

自从爷爷去后，这山被二郎菩萨点上火，烧杀了大半。我们蹲在井里，钻在洞内，藏于铁板桥下，得了性命。及至火灭烟消，出来时，又没花果养赡，难以存活，别处又去了一半。我们这一半，捱苦的住在山中，这两年，又被些打猎的抢了一半去也。

邓俊辉

deng@tsinghua.edu.cn

## 统一接口

❖ template <typename T> //查找算法统一接口， $0 \leq lo < hi \leq _size$

```
Rank Vector<T>::search( T const & e, Rank lo, Rank hi ) const {
    return ( rand() % 2 ) ? //按各50%的概率随机选用
        binSearch( _elem, e, lo, hi ) //二分查找算法，或者
        : fibSearch( _elem, e, lo, hi ); //Fibonacci查找算法
}
```



## 减而治之

- 以任一元素  $x = s[mi]$  为界，都可将待查找区间分为三部分 //  $s[mi]$  称作轴点
- 既然向量整体有序，则必有：

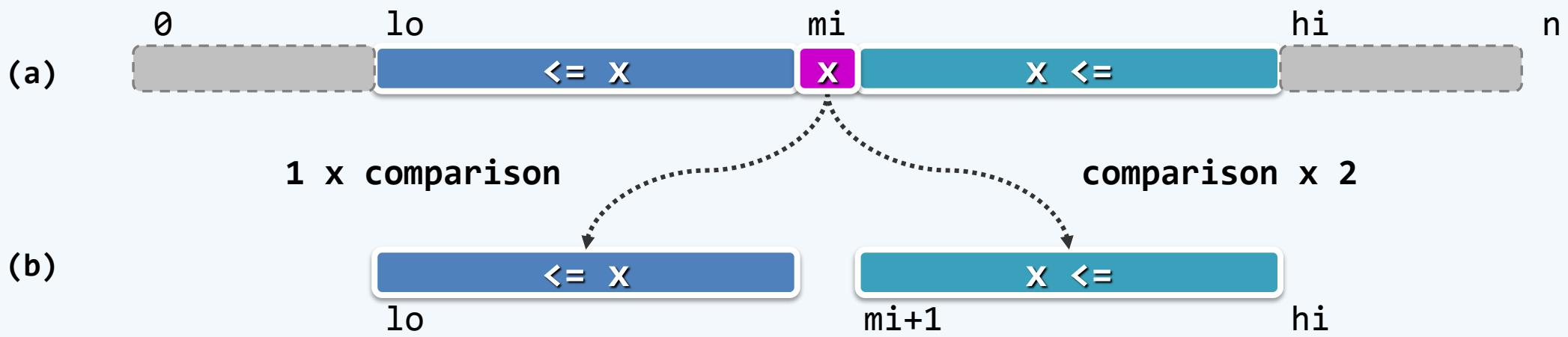
$$s[lo, mi] \leq s[mi] \leq s(mi, hi)$$



## 减而治之

❖ 只需将目标元素 $e$ 与 $x$ 做一比较，即可分三种情况进一步处理：

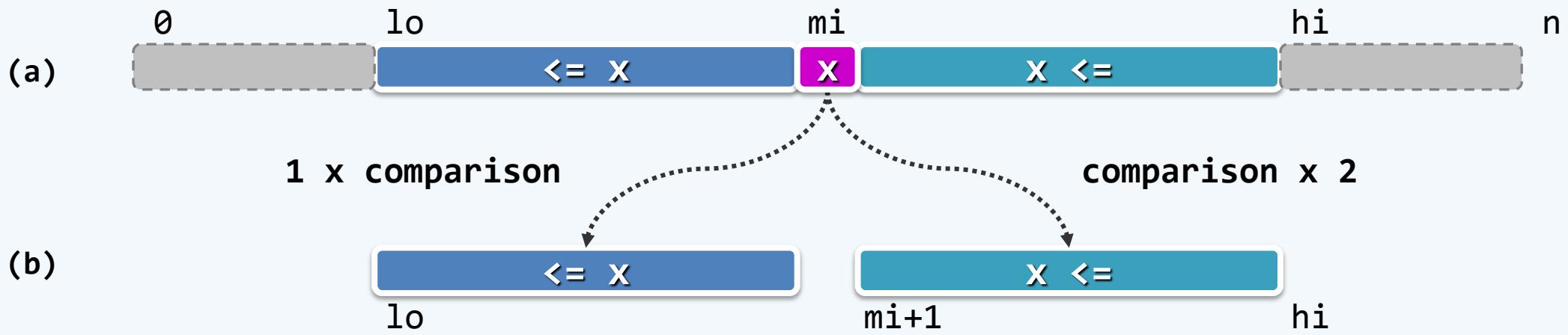
- $e < x$ ：则 $e$ 若存在必属于**左侧子区间** $S[lo, mi]$ ，故可递归深入
- $x < e$ ：则 $e$ 若存在必属于**右侧子区间** $S(mi, hi)$ ，亦可递归深入
- $e = x$ ：已在此处**命中**，可随即返回 *//若多个，返回何者？*



## 二分折半

❖ 若轴点 $mi$ 取作中点，则每经过**至多两次**比较

- 或者能够命中
- 或者将问题规模缩减一半



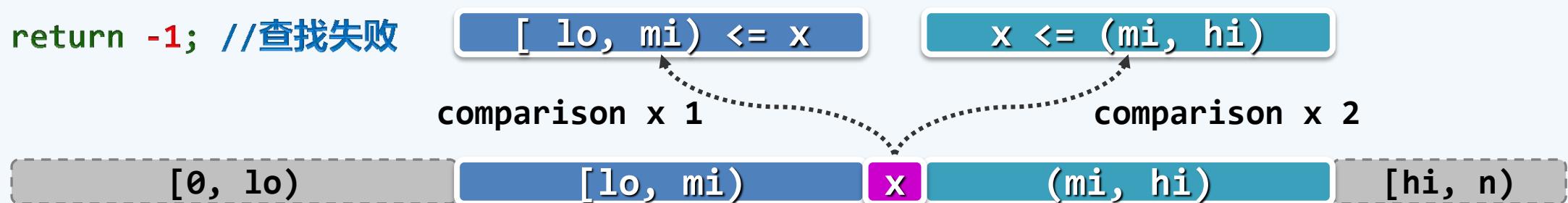
## 实现

```

❖ template <typename T> //在有序向量区间[lo, hi)内查找元素e

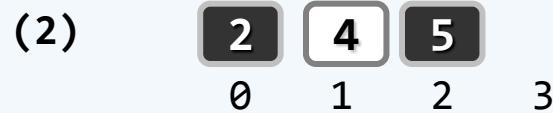
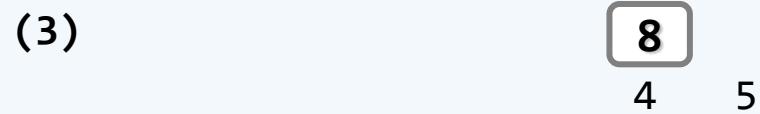
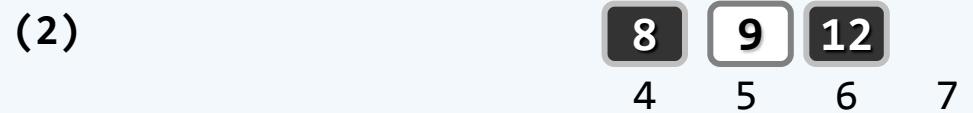
static Rank binSearch( T * A, T const & e, Rank lo, Rank hi ) {
    while ( lo < hi ) { //每步迭代可能要做两次比较判断，有三个分支
        Rank mi = ( lo + hi ) >> 1; //以中点为轴点
        if ( e < A[mi] ) hi = mi; //深入前半段[lo, mi)继续查找
        else if ( A[mi] < e ) lo = mi + 1; //深入后半段(mi, hi)
        else return mi; //在mi处命中
    }
    return -1; //查找失败
}

```



## 实例 + 复杂度

- ❖ `S.search(8, 0, 7)` : 共经  $2 + 1 + 2 = 5$  次比较，在`S[4]`处命中
- ❖ `S.search(3, 0, 7)` : 共经  $1 + 1 + 2 = 4$  次比较，在`S[1]`处失败



- ❖ 线性递归： $T(n) = T(n/2) + O(1) = O(\log n)$ ，大大优于顺序查找  
 递归跟踪：轴点总取中点，递归深度  $O(\log n)$ ；各递归实例均耗时  $O(1)$

## 查找长度

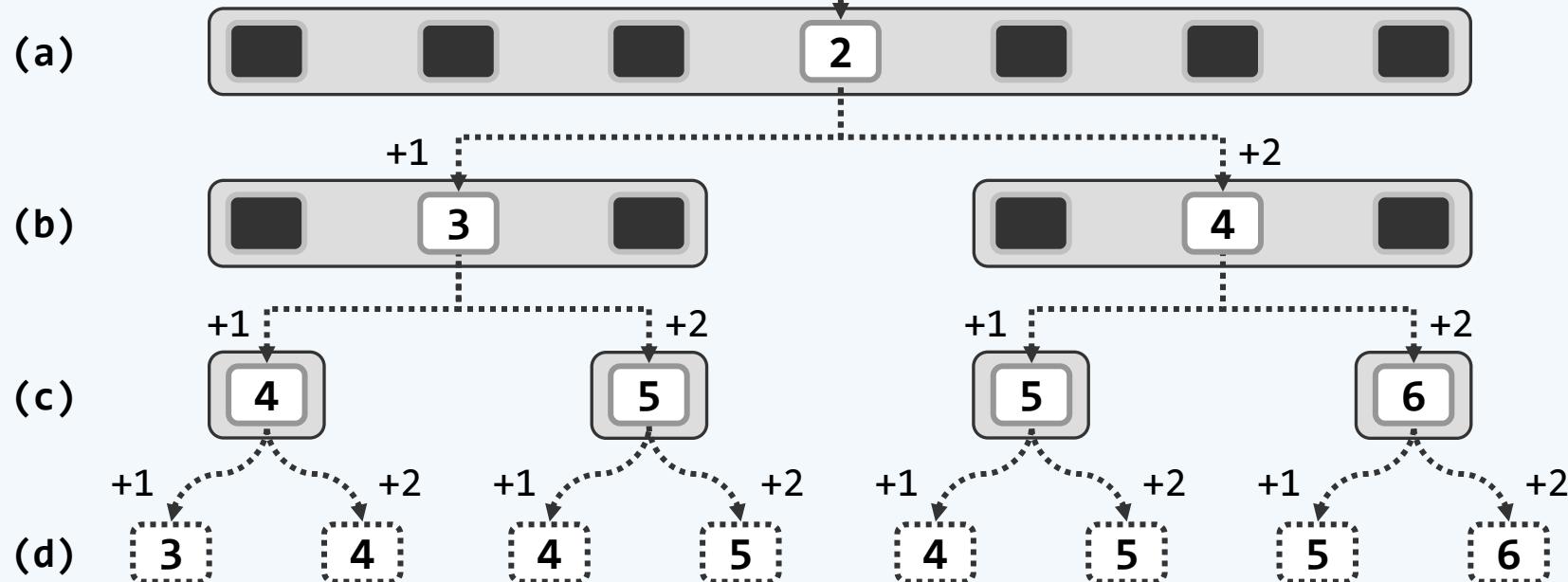
❖ 如何更为精细地评估查找算法的性能？

考查关键码的比较次数，即查找长度（search length）

❖ 通常，需分别针对成功与失败查找，从最好、最坏、平均等角度评估

❖ 比如，成功、失败时的平均查找长度均大致为  $\mathcal{O}(1.50 \cdot \log n)$

//详见教材、习题解析



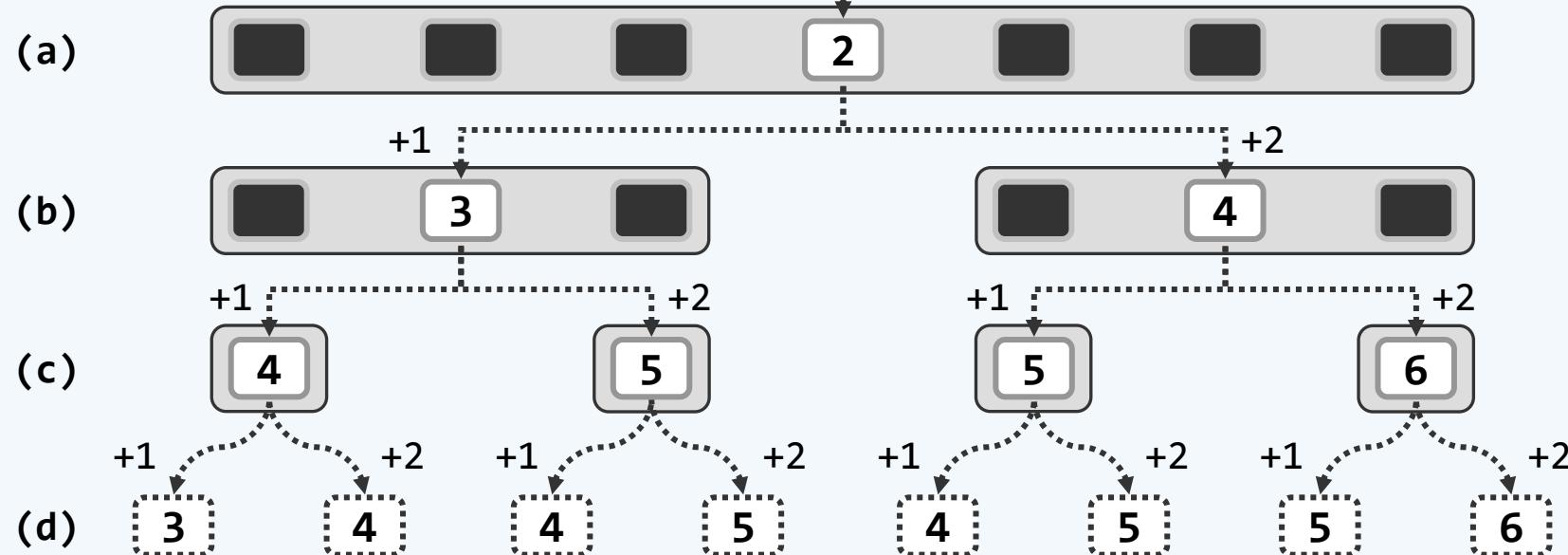
## 查找长度

❖  $n = 7$  时，各元素对应的成功查找长度为 { 4, 3, 5, 2, 5, 4, 6 }

在等概率情况下，平均成功查找长度 =  $29 / 7 = 4.14$

❖ 共8种失败情况，查找长度分别为 {3, 4, 4, 5, 4, 5, 5, 6}

在等概率情况下，平均失败查找长度 =  $36 / 8 = 4.50$



❖ 各种查找结果出现的概率不均等时，查找长度应该如何定义和计算？

## 2. 向量

有序向量

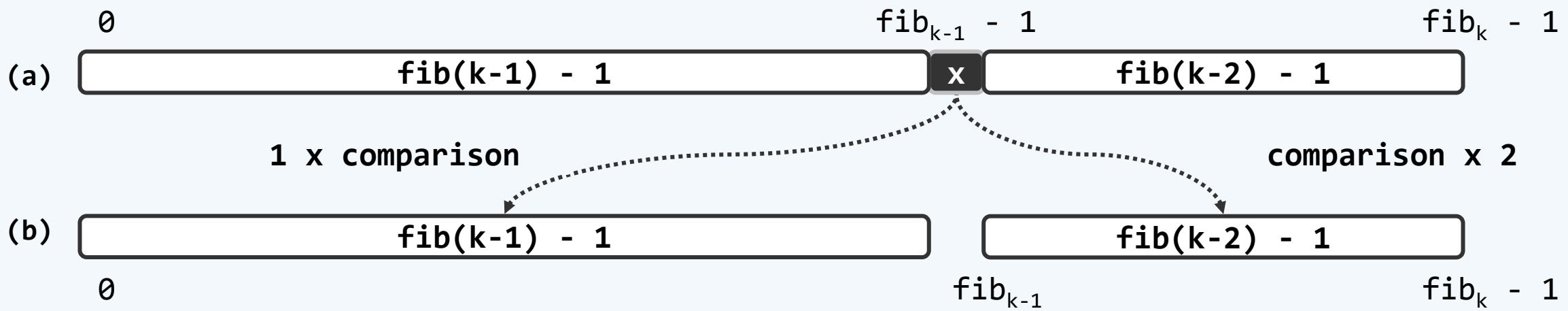
Fibonacci查找

邓俊辉

deng@tsinghua.edu.cn

## 思路及原理

- ❖ 二分查找版本A的效率仍有改进余地，因为不难发现转向左、右分支前的关键码**比较次数**不等，而**递归深度**却相同
  - ❖ 若能通过**递归深度**的不均衡，对**转向成本**的不均衡进行**补偿**平均查找长度应能进一步缩短...
  - ❖ 比如，若设 $n = \text{fib}(k) - 1$ ，则可取 $mi = \text{fib}(k - 1) - 1$   
于是，前、后子向量的长度分别为 $\text{fib}(k - 1) - 1$ 、 $\text{fib}(k - 2)$



## 实现

```

❖ template <typename T> //0 <= lo <= hi <= _size
    static Rank fibSearch( T * A, T const & e, Rank lo, Rank hi ) {
        Fib fib(hi - lo); //用 $O(\log_{\Phi}n) = O(\log_{\Phi}(hi - lo))$ 时间创建Fib数列
        while ( lo < hi ) {
            while ( hi - lo < fib.get() ) fib.prev(); //至多迭代几次？整体累计几次？
                //通过向前顺序查找，确定形如Fib(k) - 1的轴点（分摊 $O(1)$ ）
            Rank mi = lo + fib.get() - 1; //按黄金比例切分
            if ( e < A[mi] ) hi = mi; //深入前半段[lo, mi)继续查找
            else if ( A[mi] < e ) lo = mi + 1; //深入后半段(mi, hi)
            else return mi; //在mi处命中
        }
        return -1; //查找失败
    }
}

```

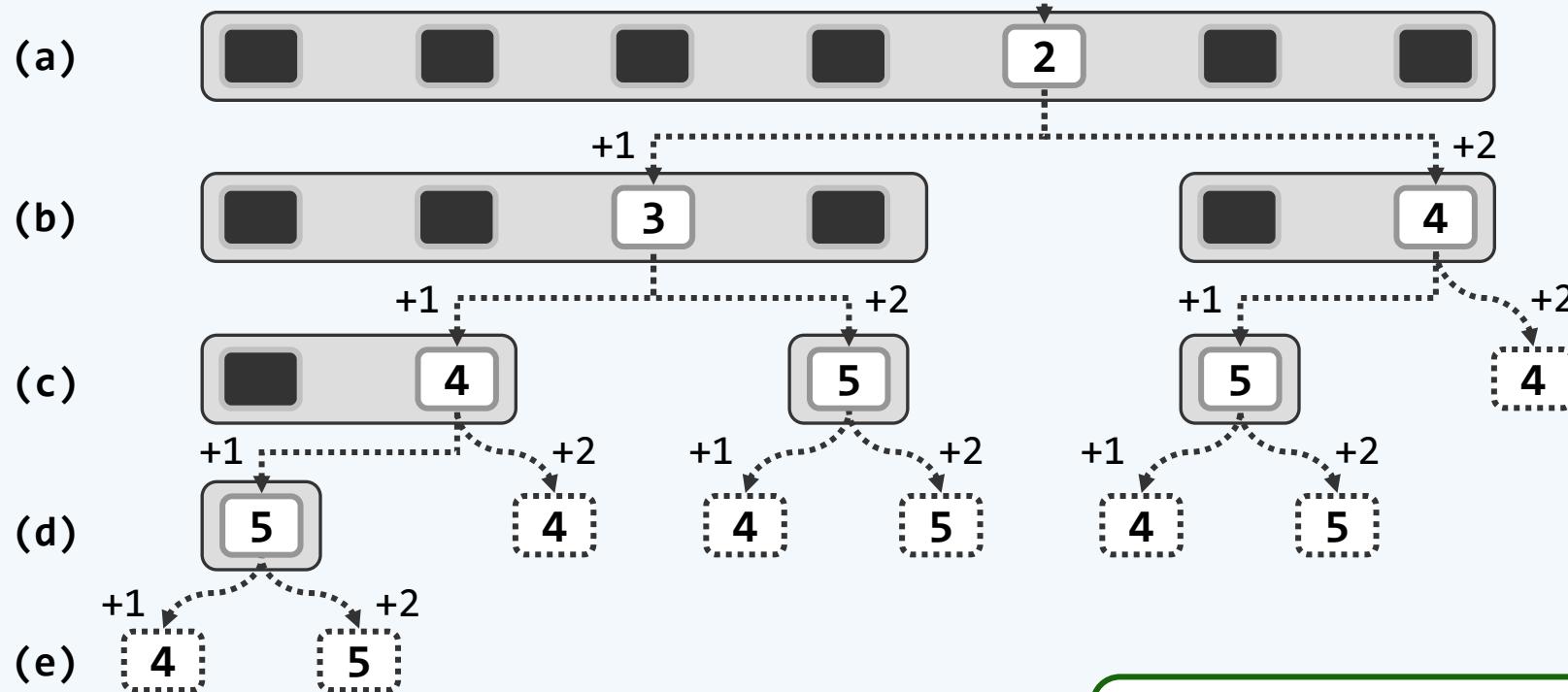
## 查找长度

❖ Fibonacci查找的ASL，（在常系数的意义上）优于二分查找 //详见教材、习题解析

❖ 仍以 $n = \text{fib}(6) - 1 = 7$ 为例，在等概率情况下

$$\text{平均成功查找长度} = (5 + 4 + 3 + 5 + 2 + 5 + 4) / 7 = 28/7 = 4.00$$

$$\text{平均失败查找长度} = (4 + 5 + 4 + 4 + 5 + 4 + 5 + 4) / 8 = 35 / 8 = 4.38$$

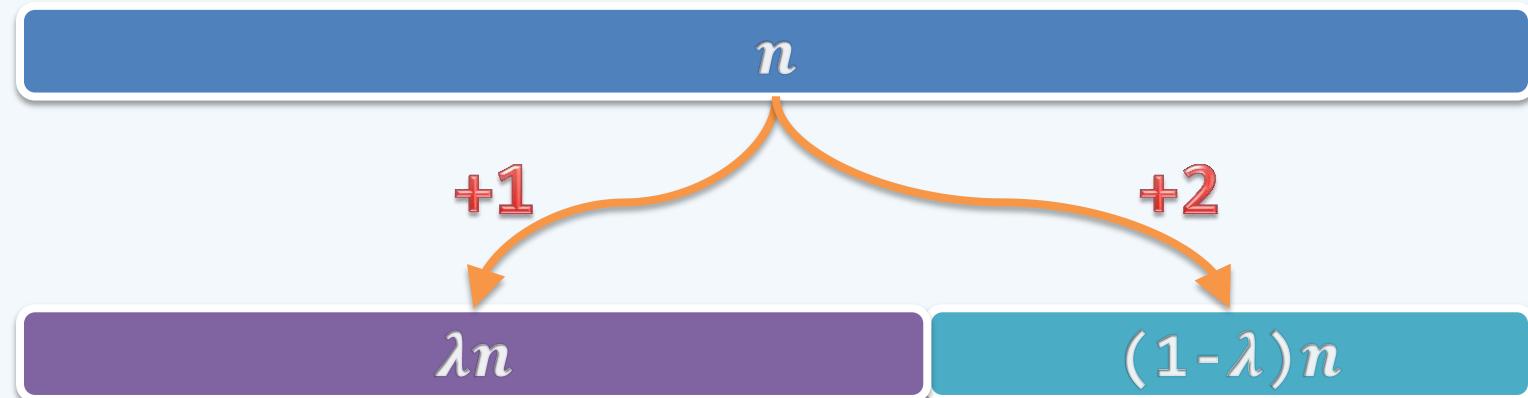


## 通用策略

❖ 对于任何的  $A[\theta, n]$ ，总是选取  $A[\lambda n]$  作为轴点， $\theta \leq \lambda < 1$

比如：二分查找对应于  $\lambda = 0.5$ ，Fibonacci 查找对应于  $\lambda = \phi = 0.6180339\dots$

❖ 在  $[\theta, 1)$  内， $\lambda$  如何取值才能达到最优？设平均查找长度为  $\alpha(\lambda) \cdot \log_2 n$ ，何时  $\alpha(\lambda)$  最小？

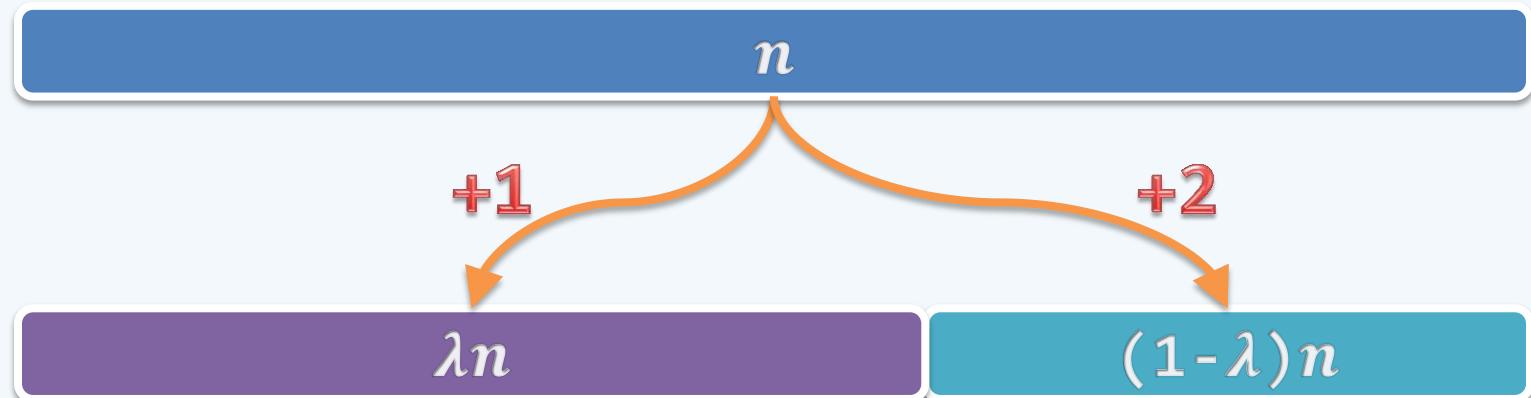


$$\phi = 0.6180339\dots$$

❖ 递推式： $\alpha(\lambda) \cdot \log_2 n = \lambda \cdot [1 + \alpha(\lambda) \cdot \log_2(\lambda n)] + (1 - \lambda) \cdot [2 + \alpha(\lambda) \cdot \log_2((1 - \lambda)n)]$

❖ 整理后： $\frac{-\ln 2}{\alpha(\lambda)} = \frac{\lambda \cdot \ln \lambda + (1 - \lambda) \cdot \ln(1 - \lambda)}{2 - \lambda}$

❖ 当 $\lambda = \phi$ 时， $\alpha(\lambda) = 1.440420\dots$ 达到最小



## 2. 向量

有序向量

二分查找（版本B）

和微风匀到一起的光，象冰凉的刀刃儿似的，把  
宽静的大街切成两半，一半儿黑，一半儿亮。那  
黑的一半，使人感到阴森，亮的一半使人感到凄  
凉。

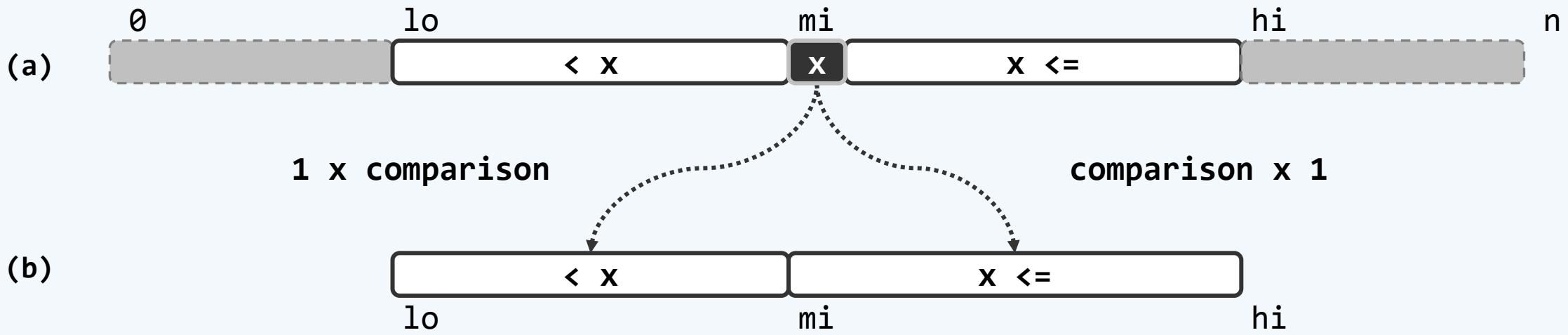
邓俊辉

deng@tsinghua.edu.cn

## 改进思路

- ❖ 二分查找中左、右分支转向代价不平衡的问题，也可直接解决
- ❖ 比如，每次迭代（或每个递归实例）仅做1次关键码比较

如此，所有分支只有2个方向，而不再是3个

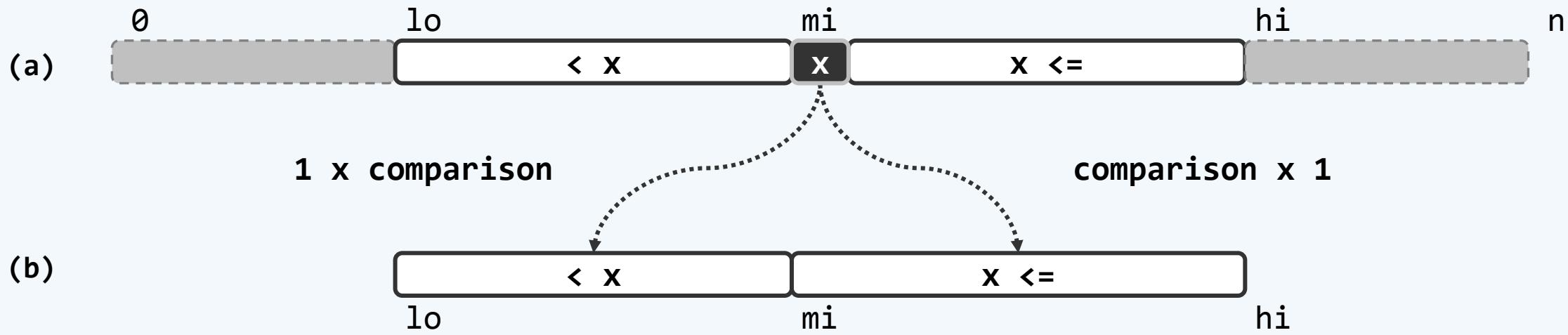


## 改进思路

同样地，轴点 $mi$ 取作中点，则查找每深入一层，问题规模也缩减一半

- 1 )  $e < x$  : 则 $e$ 若存在必属于左侧子区间 $s[lo, mi)$ ，故可递归深入
- 2 )  $x \leq e$  : 则 $e$ 若存在必属于右侧子区间 $s[mi, hi)$ ，亦可递归深入

只有当元素数目 $hi - lo = 1$ 时，才判断该元素是否命中



## 实现

```

❖ template <typename T> static Rank binSearch( T * A, T const & e, Rank lo, Rank hi ) {

    while ( 1 < hi - lo ) { //有效查找区间的宽度缩短至1时，算法才会终止

        Rank mi = (lo + hi) >> 1; //以中点为轴点，经比较后确定深入

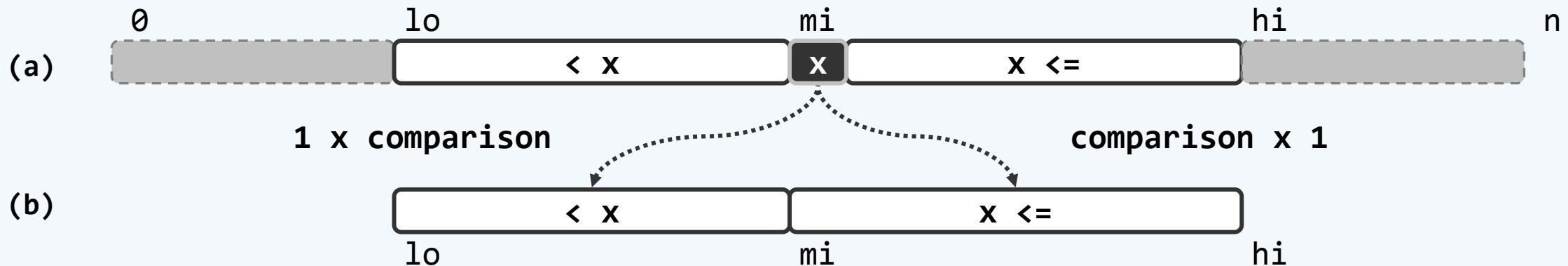
        e < A[mi] ? hi = mi : lo = mi; // [lo, mi) 或 [mi, hi)

    } //出口时hi = lo + 1，查找区间仅含一个元素A[lo]

    return e == A[lo] ? lo : -1 ; //返回命中元素的秩或者-1

} //相对于版本A，最好(坏)情况下更坏(好)；各种情况下的SL更加接近，整体性能更趋稳定

```



## 语义约定

❖ 各种特殊情况，如何统一地处置？比如

- 目标元素不存在；或反过来
- 目标元素同时存在多个

❖ 有序向量自身，如何便捷地维护？

比如：`V.insert( 1 + V.search(e), e )`

- 即便失败，也应给出新元素适当的插入位置 //有序性
- 若允许重复元素，则每一组也需按其插入的次序排列 //稳定性

❖ 为此，需要更为精细、明确、简捷地定义`search()`的返回值

## 语义约定

- ❖ 约定：search()总是返回**不大于e的最后一个元素** //其后继，即大于e的第一个元素
- 若  $-\infty < e < v[lo]$ ，则返回**lo - 1** //左侧哨兵 = 首元素的前驱
- 若  $v[hi - 1] < e < +\infty$ ，则返回**hi - 1** //末元素 = 右侧哨兵的前驱
- 若  $v[k] < e < v[k + 1]$ ，则返回**k** //e可作为v[k]的后继插入
- 若  $v[k] \leq e < v[k + 1]$ ，亦返回**k** //e可作为v[k]的后继插入，且稳定
- ❖ 二分查找版本A、版本B及fibSearch()，均未严格兑现这一语义约定
- ❖ 课后：对版本B略作调整，使之符合约定；对fibSearch()略作调整，使之符合约定
- ❖ 有没有更为简明的实现方式？

## 2. 向量

### 有序向量

### 二分查找（版本C）

Teach me half the gladness  
That thy brain must know,  
Such harmonious madness  
From my lips would flow  
The world should listen then,  
as I am listening now.

邓俊辉

deng@tsinghua.edu.cn

## 实现

```

❖ template <typename T> static Rank binSearch( T * A, T const & e, Rank lo, Rank hi ) {

    while ( lo < hi ) { //不变性 : A[0, lo) <= e < A[hi, n)

        Rank mi = (lo + hi) >> 1; //以中点为轴点，经比较后确定深入

        e < A[mi] ? hi = mi : lo = mi + 1; // [lo, mi) 或 (mi, hi)

    } //出口时，A[lo = hi] 为大于e的最小元素

    return --lo; //故lo - 1即不大于e的元素的最大秩

}

```

### ❖ 与版本B的差异

- 待查找区间宽度缩短至0而非1时，算法才结束 //lo == hi
- 转入右侧子向量时，左边界取作mi + 1而非mi //A[mi]会被遗漏？
- 无论成功与否，返回的秩严格符合接口的语义约定... //如何证明其正确性？

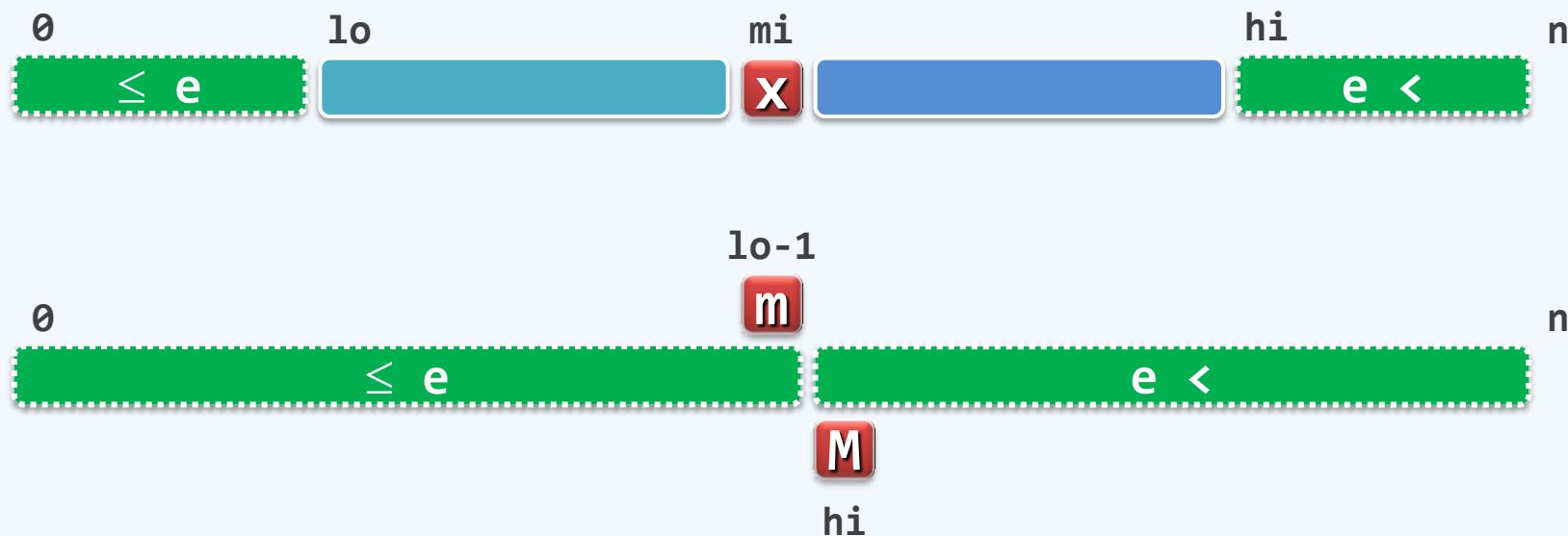
不变性： $A[0, lo) \leq e < A[hi, n)$

❖ 在算法执行过程中的任意时刻

$A[lo - 1] / A[hi]$  总是（截至当时已确认的）**不大于e的最大者** / **大于e的最小者**

❖ 当算法终止时 ( $lo == hi$ )

$A[lo - 1] / A[hi]$  即是（全局）**不大于e的最大者** / **大于e的最小者**

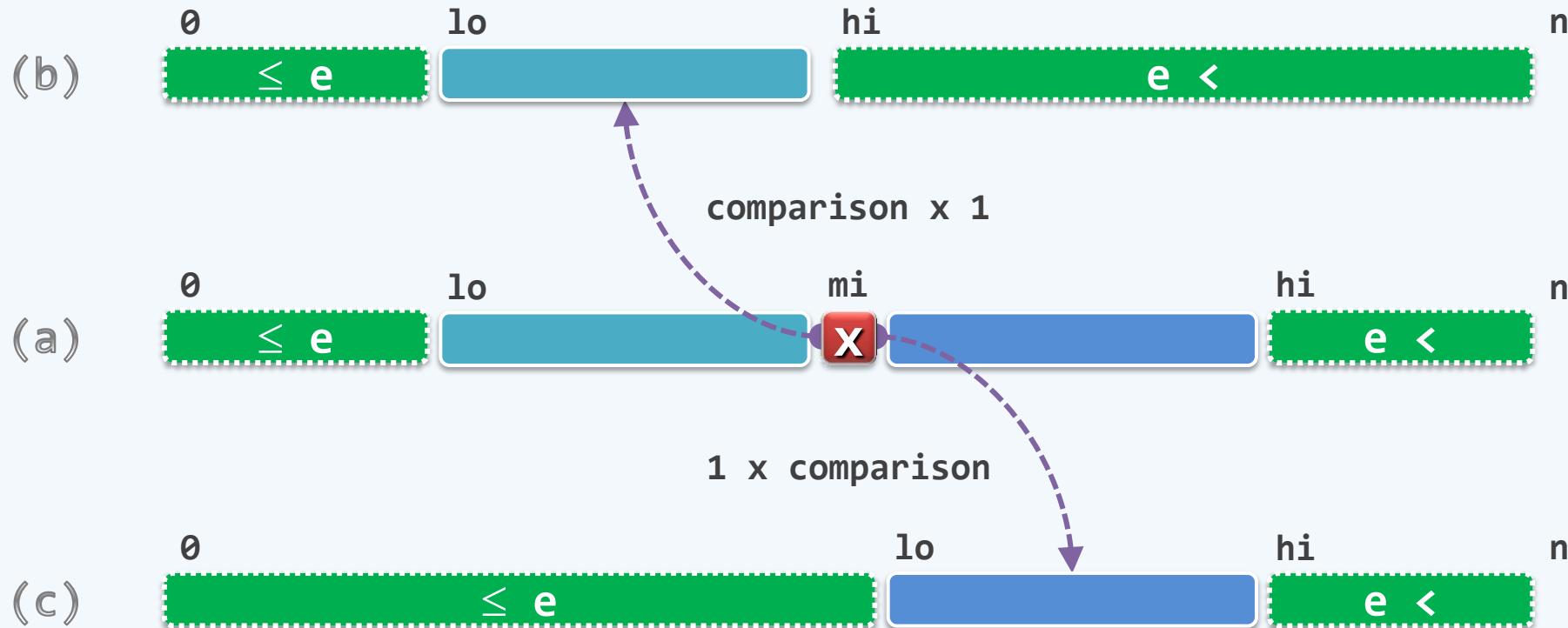


不变性： $A[0, lo] \leq e < A[hi, n]$

❖ 初始时， $lo = 0$ 且 $hi = n$ ， $A[0, lo] = A[hi, n] = \emptyset$ ，自然成立

❖ 数学归纳：假设不变性一直保持至(a)

//以下无非两种情况...



## 课后

❖ 针对二分查找，Knuth ( ACP-v3-s6.2.1-ex\_23 ) 曾指出：

将三分支变为两分支后的改进效果

需要到n非常大 ( $2^{2*(17-(-16))} = 2^{66}$ ) 后方能体现

试阅读相关段落；这一结论，对当下的实际应用有何意义？

## 2. 向量

有序向量

插值查找

邓俊辉

deng@tsinghua.edu.cn

## 原理与算法

❖ 假设：已知有序向量中各元素随机分布的规律

比如：均匀且独立的随机分布

❖ 于是：[lo, hi)内各元素应大致按照线性趋势增长

$$\frac{mi - lo}{hi - lo} \approx \frac{e - A[lo]}{A[hi] - A[lo]}$$

❖ 因此：通过猜测轴点mi，可以极大地提高收敛速度

$$mi \approx lo + (hi - lo) \cdot \frac{e - A[lo]}{A[hi] - A[lo]}$$

❖ 以英文词典为例：**binary**大致位于2/26处

**search**大致位于19/26处

[lo]	0	A	1	[1, 53)
	1	B	74	[53, 104)
	2	C	158	[104, 156)
	3	D	292	[156, 208)
	4	E	368	[208, 259)
	5	F	409	[259, 311)
	6	G	473	[311, 363)
	7	H	516	[363, 414)
	8	I	562	[414, 466)
	9	J	607	[466, 518)
	10	K	617	[518, 569)
	11	L	628	[569, 621)
	12	M	681	[621, 673)
	13	N	748	[673, 724)
	14	O	771	[724, 776)
	15	P	806	[776, 827)
	16	Q	915	[827, 879)
	17	R	922	[879, 931)
	18	S	1002	[931, 982)
	19	T	1176	[982, 1034)
	20	U	1253	[1034, 1086)
	21	V	1271	[1086, 1137)
	22	W	1289	[1137, 1189)
	23	X	1337	[1189, 1241)
	24	Y	1338	[1241, 1292)
	25	Z	1341	[1292, 1344)
[hi]	26		1344	

## 实例

❖  $e = 50$

$lo = 0, hi = 18$



插值:  $mi = 0 + (18 - 0)*(50 - 5)/(92 - 5) \approx 9.3$

取:  $mi = 9$

比较:  $A[9] = 46 < e$

❖  $lo = 10, hi = 18$

插值:  $mi = 10 + (18 - 10)*(50 - 49)/(92 - 49) \approx 10.2$

取:  $mi = 10$

比较:  $A[10] = 49 < e$

❖  $lo = 11, hi = 18$

插值:  $mi = 11 + (18 - 11)*(50 - 51)/(92 - 51) \approx 10.8$

取:  $mi = 10 < lo$

查找完成 (NOT\_FOUND)

## 性能

- ❖ 最坏： $\Theta(hi - lo) = \Theta(n)$  //具体实例？
- ❖ 平均：每经一次比较，待查找区间宽度由  $n$  缩至  $\sqrt{n}$  // [Yao76, PIA78]，习题解析[2-24]

$$n, \sqrt{n}, \sqrt{\sqrt{n}}, \sqrt{\sqrt{\sqrt{n}}}, \dots, 2$$

$$n, n^{1/2}, n^{1/2^2}, \dots, n^{1/2^k}, \dots, 2$$

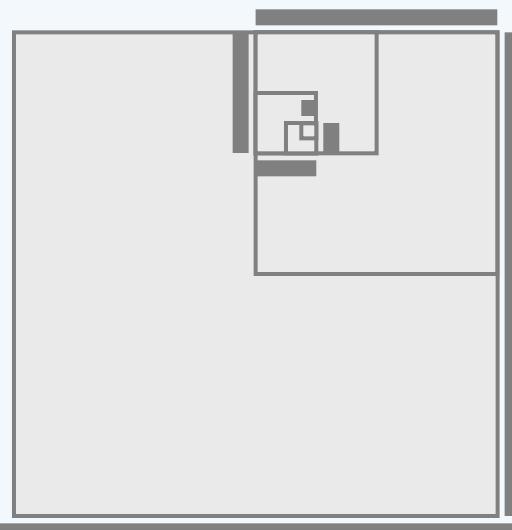
❖ 经多少次比较之后，有  $n^{1/2^k} < 2$ ，或等价地， $1/2^k \cdot \log n < 1$ ？

不难解得： $k = \Theta(\log \log n)$  //如何理解？

❖ 每经一次比较，待查找区间宽度的  $[$  数值  $n$  开方  $]$ ，有效字长  $\log n$  减半

**插值查找** = 在字长意义上的 **折半查找**

**二分查找** = 在字长意义上的 **顺序查找**



## 综合

- ❖ 从 $\Theta(\log n)$ 到 $\Theta(\log \log n)$ ，是否值得？
- ❖ 通常优势不明显——除非查找区间宽度极大，或者比较操作成本极高

比如， $n = 2^{(2^5)} = 2^{32} = 4G$ 时， $\log_2(n) = 32$ ， $\log_2(\log_2(n)) = 5$

- ❖ 易受小扰动的干扰和“蒙骗”
- ❖ 须引入乘法、除法运算
- ❖ 实际可行的方法
  - 首先通过插值查找，将查找范围缩小到一定的尺度
  - 然后再进行二分查找

## 课后

❖ 关于插值查找算法的平均性能分析，试阅读：

A. C. Yao & F. F. Yao

"The Complexity of Searching an Ordered Random Table"

Proc. of 17<sup>th</sup> FOCS (1976), 222-227

## 2. 向量

起泡排序

邓俊辉

deng@tsinghua.edu.cn

## 排序器：统一入口

❖ `template <typename T>`

```
void Vector<T>::sort( Rank lo, Rank hi ) { //区间[lo, hi)  
    switch ( rand() % 5 ) { //可视具体问题的特点灵活选取或扩充  
        case 1 : bubbleSort( lo, hi ); break; //起泡排序  
        case 2 : selectionSort( lo, hi ); break; //选择排序(习题)  
        case 3 : mergeSort( lo, hi ); break; //归并排序  
        case 4 : heapSort( lo, hi ); break; //堆排序(第10章)  
        default: quickSort( lo, hi ); break; //快速排序(第12章)  
    }  
} //在此统一接口下，具体算法的不同实现，将在后续各章节陆续讲解
```

## 起泡排序

```

❖ template <typename T> void Vector<T>::bubbleSort( Rank lo, Rank hi )
{ while ( ! bubble( lo, hi -- ) ); } //逐趟做扫描交换，直至全序

❖ template <typename T> bool Vector<T>::bubble( Rank lo, Rank hi ) {
    bool sorted = true; //整体有序标志
    while ( ++lo < hi ) //自左向右，逐一检查各对相邻元素
        if ( _elem[lo - 1] > _elem[lo] ) { //若逆序，则
            sorted = false; //意味着尚未整体有序，并需要
            swap( _elem[lo - 1], _elem[lo] ); //交换
        }
    return sorted; //返回有序标志
} //乱序限于[0,  $\sqrt{n}$ )时，仍需 $\Theta(n^{3/2})$ 时间——按理， $\Theta(n)$ 应已足矣

```

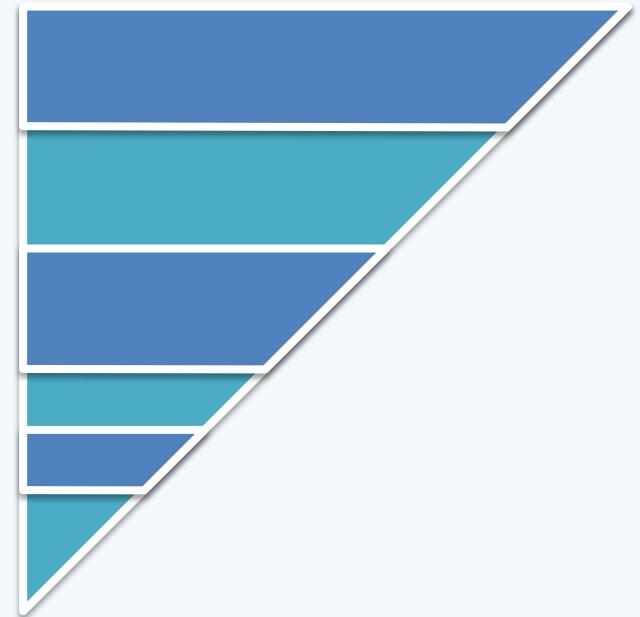
## 再改进

```

❖ template <typename T> void Vector<T>::bubbleSort( Rank lo, Rank hi )
{ while ( lo < ( hi = bubble( lo, hi ) ) ); } //逐趟扫描交换，直至全序

❖ template <typename T> Rank Vector<T>::bubble(Rank lo, Rank hi) {
    Rank last = lo; //最右侧的逆序对初始化为[lo - 1, lo]
    while (++lo < hi) //自左向右，逐一检查各对相邻元素
        if ( _elem[lo - 1] > _elem[lo] ) { //若逆序，则
            last = lo; //更新最右侧逆序对位置记录，并
            swap( _elem[lo - 1], _elem[lo] ); //交换
        }
    return last; //返回最右侧的逆序对位置
} //前一版本中的逻辑型标志sorted，改为秩last

```



## 综合评价

- ❖ 效率与第一章针对整数数组的版本相同，最好  $\mathcal{O}(n)$ ，最坏  $\mathcal{O}(n^2)$
- ❖ 输入含重复元素时，算法的 **稳定性** ( stability ) 是更为细致的要求  
重复元素在输入、输出序列中的相对次序，是否保持不变？
  - 输入： 6, **7a**, 3, 2, **7b**, 1, 5, 8, **7c**, 4
  - 输出： 1, 2, 3, 4, 5, 6, **7a**, **7b**, **7c**, 8 //stable
  - 1, 2, 3, 4, 5, 6, **7a**, **7c**, **7b**, 8 //unstable
- ❖ 以上起泡排序算法是稳定的吗？是的！为什么？
- ❖ 在起泡排序中，元素a和b的相对位置发生变化，只有一种可能：  
经分别与其它元素的交换，二者相互**接近**直至**相邻**  
在接下来一轮扫描交换中，二者因**逆序**而交换位置
- ❖ 在if一句的判断条件中，若把 “**>**”换成 “ **$\geq$** ”，将有何变化？

## 2. 向量

归并排序  
分而治之

白玉堂前春解舞，东风卷得均匀。  
蜂团蝶阵乱纷纷。

几曾随逝水，岂必委芳尘。  
万缕千丝终不改，任他随聚随分。

韶华休笑本无根。  
好风频借力，送我上青云。

邓俊辉

deng@tsinghua.edu.cn

## 原理

❖ //分治策略

//向量与列表通用

//J. von Neumann, 1945

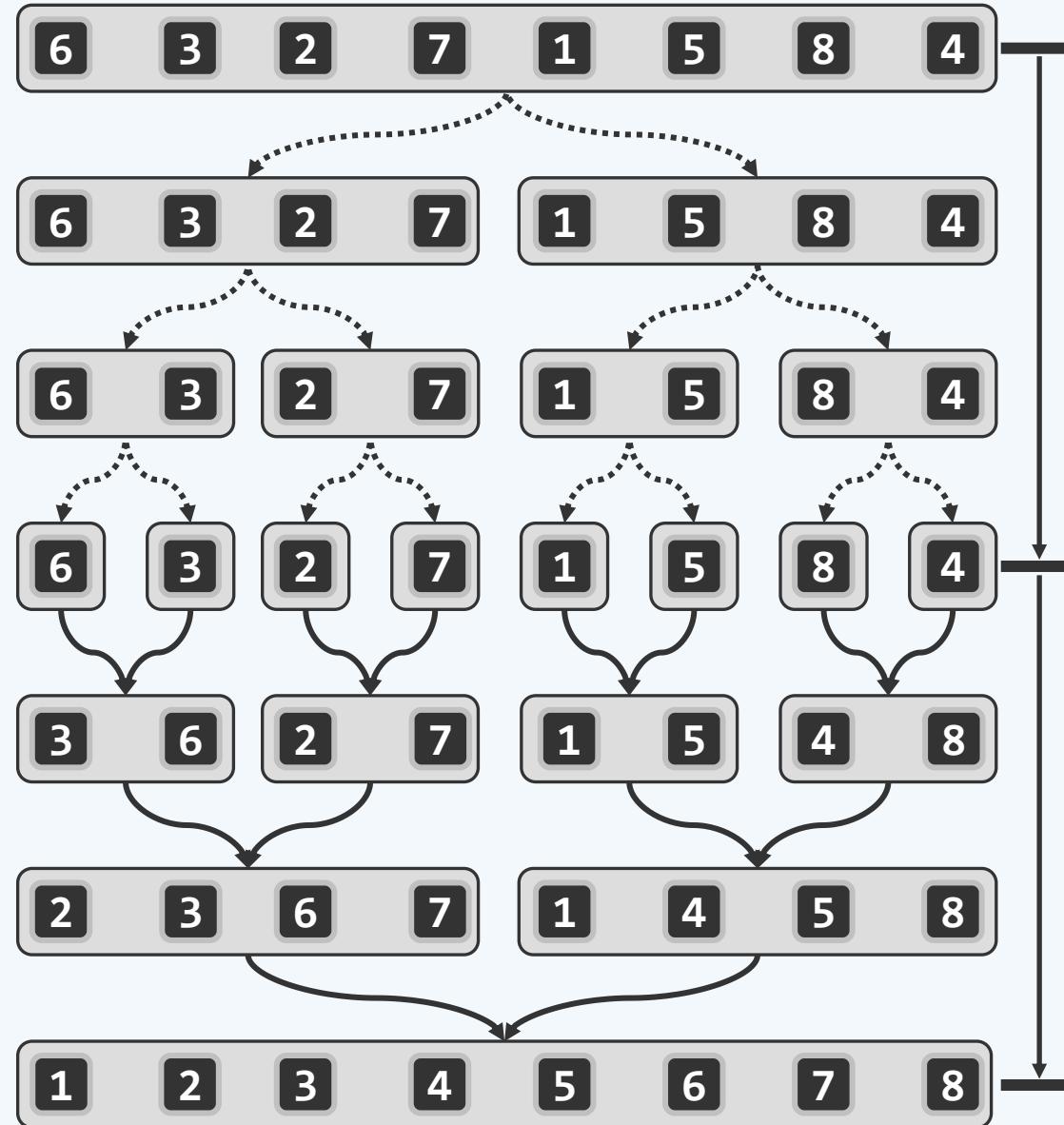
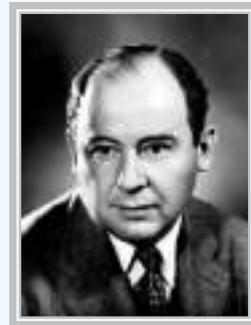
序列一分为二 // $\Theta(1)$

子序列递归排序 // $2 \times T(\lfloor n/2 \rfloor)$

合并有序子序列 // $\Theta(n)$

❖ 若真能如此，整体的运行成本应该是

$\Theta(n \log n)$



无序向量的递归分解

有序向量的逐层归并

1

## 分而治之

❖ template <typename T>

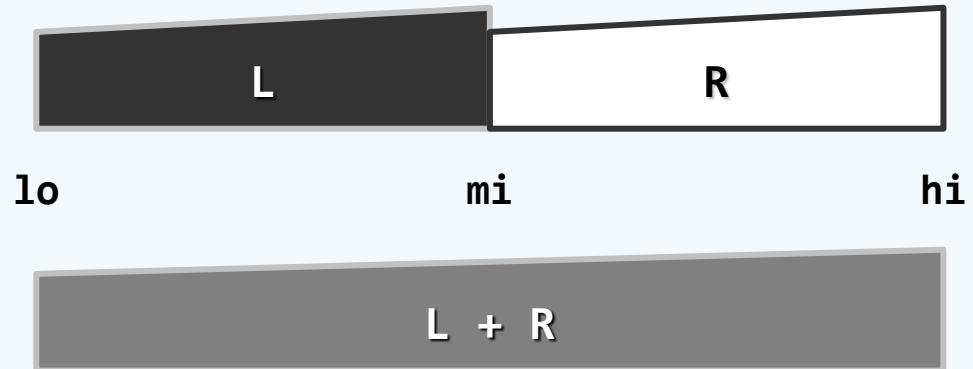
```
void Vector<T>::mergeSort( Rank lo, Rank hi ) { // [lo, hi)

    if ( hi - lo < 2 ) return; // 单元素区间自然有序，否则...

    int mi = (lo + hi) >> 1; // 以中点为界

    mergeSort( lo, mi ); // 对前半段排序
    mergeSort( mi, hi ); // 对后半段排序
    merge( lo, mi, hi ); // 归并

}
```



## 2. 向量

归并排序

二路归并

邓俊辉

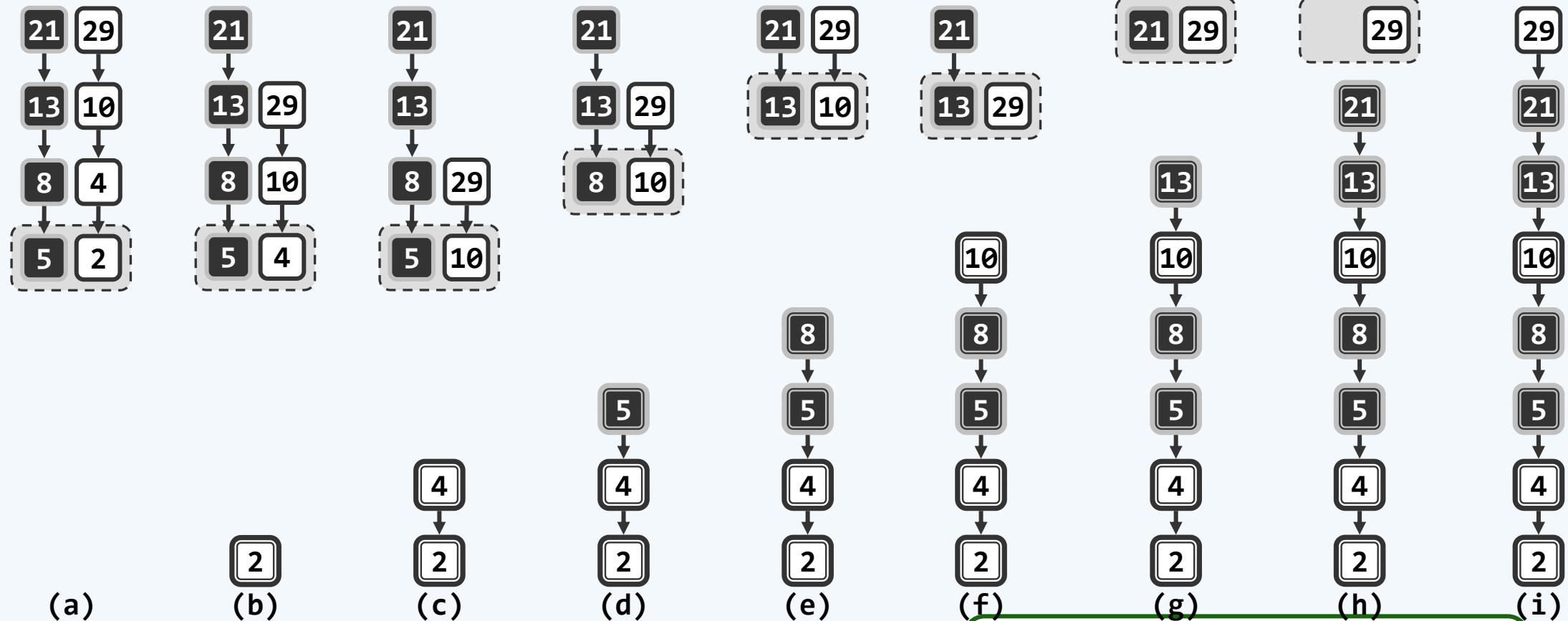
天下大势，分久必合，合久必分

deng@tsinghua.edu.cn

## 二路归并

❖ 2-way merge : 两个有序序列，合并为一个有序序列：

$$S[lo, hi] = S[lo, mi] + S[mi, hi]$$



## 基本实现

```

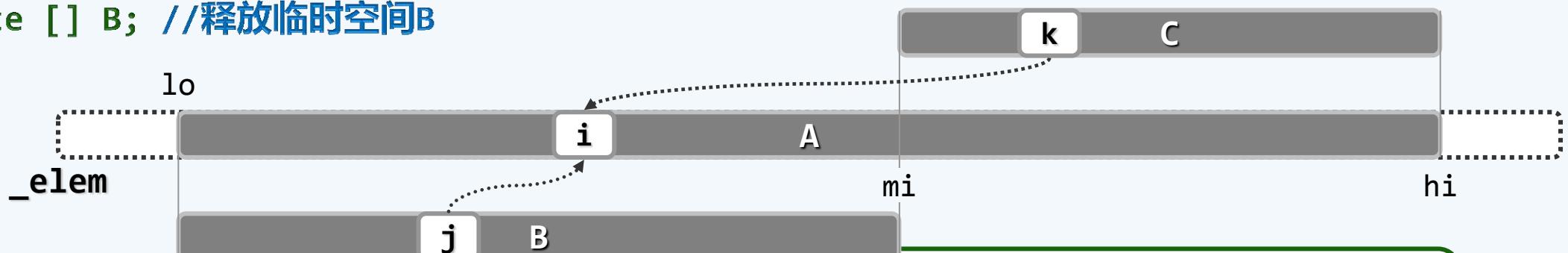
template <typename T> void Vector<T>::merge( Rank lo, Rank mi, Rank hi ) {
    T* A = _elem + lo; int lb = mi - lo; T* B = new T[lb]; //A[0, hi - lo) = _elem[lo, hi)
    for ( Rank i = 0; i < lb; B[i] = A[i++] ); //复制前子向量B[0, lb) = _elem[lo, mi)

    int lc = hi - mi; T* C = _elem + mi; //后子向量C[0, lc) = _elem[mi, hi)

    for ( Rank i = 0, j = 0, k = 0; j < lb || k < lc; ) { //B[j]和C[k]中小者转至A的末尾
        if ( j < lb && ( lc <= k || B[j] <= C[k] ) ) A[i++] = B[j++]; //C[k]已无或不大
        if ( k < lc && ( lb <= j || C[k] < B[j] ) ) A[i++] = C[k++]; //B[j]已无或更大
    } //该循环实现紧凑；但就效率而言，不如拆分处理

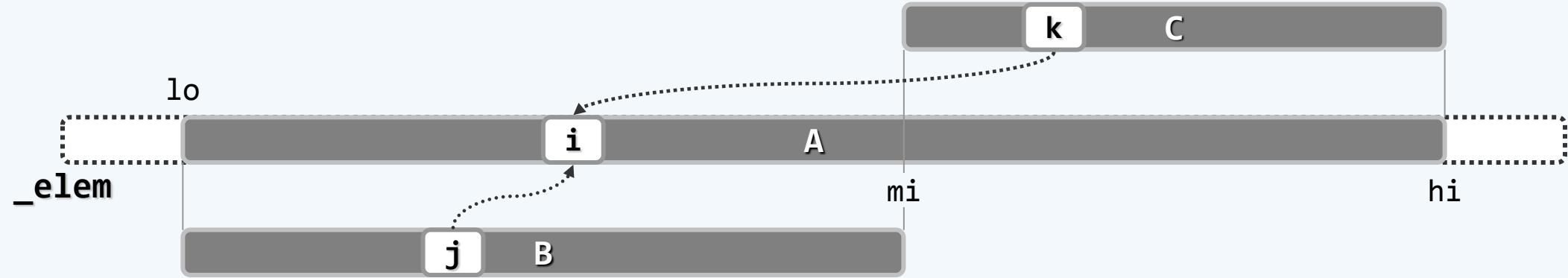
    delete [] B; //释放临时空间B
}

```

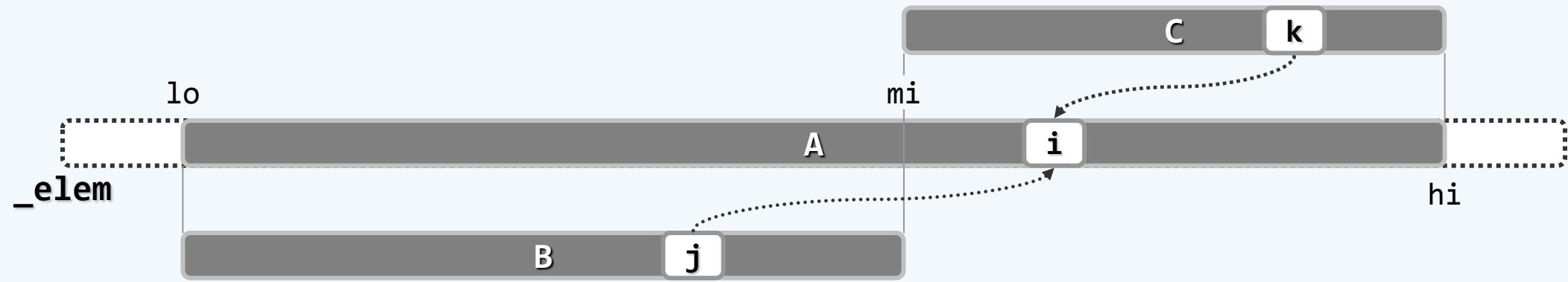


## 正确性

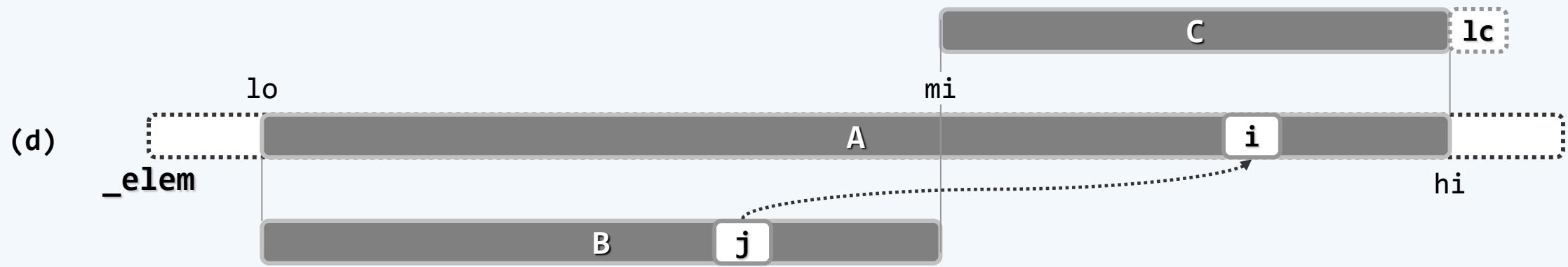
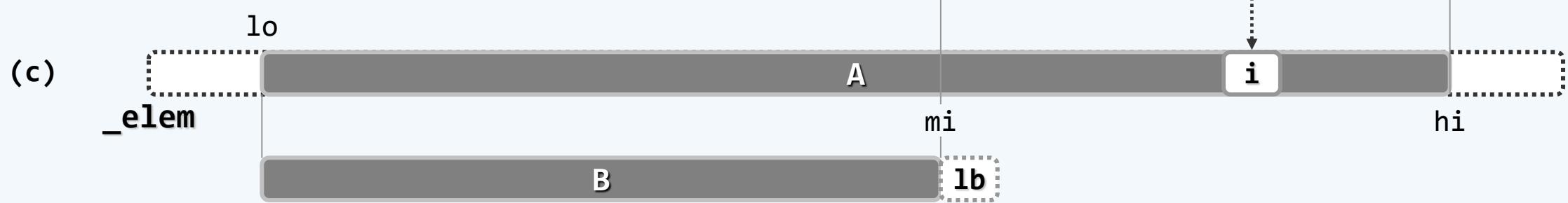
(a)



(b)



## 正确性



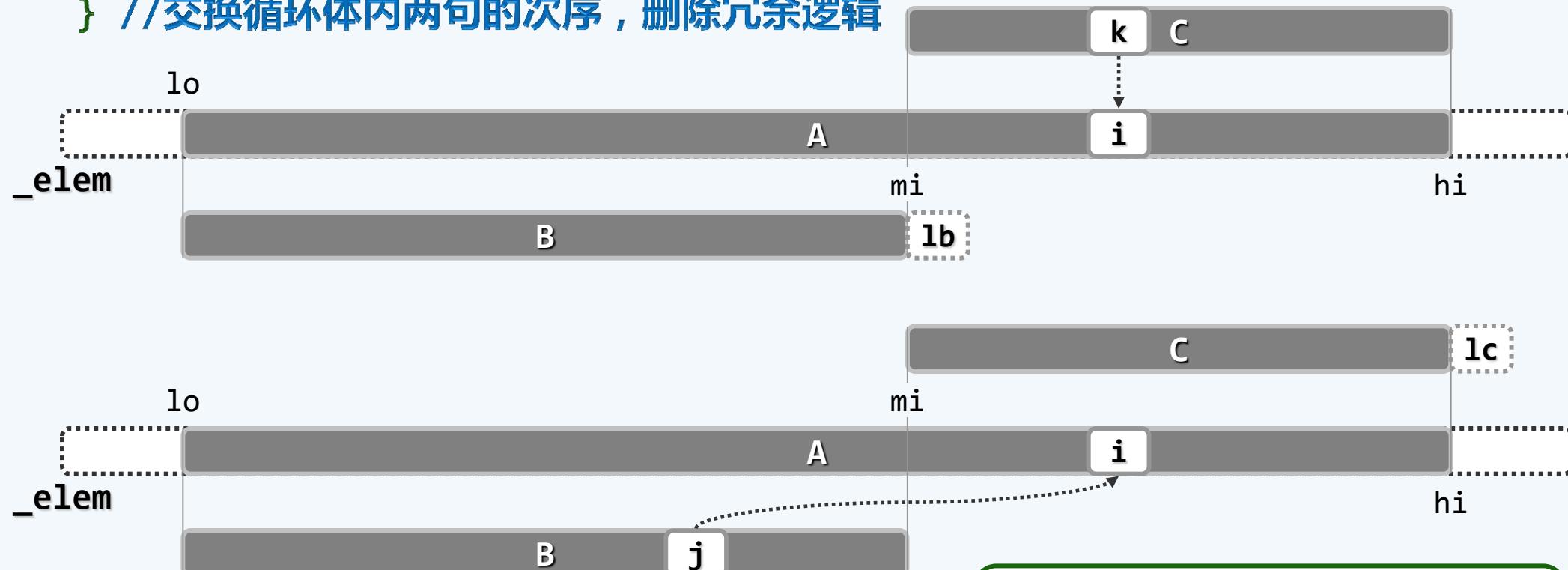
## 精简实现

```

❖ for ( Rank i = 0, j = 0, k = 0; !(j < 1b) || !(k < 1c); ) {
    if ( k < 1c && !(1b <= j || C[k] < B[j]) ) A[i++] = C[k++];
    if ( j < 1b && !(1c <= k || B[j] <= C[k]) ) A[i++] = B[j++];
}

```

//交换循环体内两句的次序，删除冗余逻辑



## 2. 向量

归并排序

复杂度

I think there is a world market  
for about five computers.

- T. J. Watson, 1943

邓俊辉

deng@tsinghua.edu.cn

❖ 算法的运行时间主要消耗于for循环，共有两个控制变量

初始： $j = 0, k = 0$

最终： $j = 1b, k = 1c$

亦即： $j + k = 1b + 1c = hi - lo = n$

❖ 观察：每经过一次迭代， $j$ 和 $k$ 中至少有一个会加一（ $j + k$ 也必至少加一）

❖ 故知：merge()总体迭代不过 $\mathcal{O}(n)$ 次，累计只需线性时间！

❖ 这一结论与排序算法的 $\Omega(n \log n)$ 下界并不矛盾——毕竟这里的B和C均已各自有序

❖ 注意：待归并子序列不必等长

亦即：允许 $1b \neq 1c, mi \neq (lo + hi)/2$

❖ 实际上，这一算法及结论也适用于另一类序列——列表（下一章）

## 综合评价

### ❖ 优点

实现最坏情况下最优 $\Theta(n \log n)$ 性能的第一个排序算法

不需随机读写，完全顺序访问——尤其适用于

列表之类的序列

磁带之类的设备

只要实现恰当，可保证稳定——出现雷同元素时，左侧子向量优先

可扩展性极佳，十分适宜于外部排序——海量网页搜索结果的归并

易于并行化

### ❖ 缺点

非就地，需要对等规模的辅助空间——可否更加节省？

即便输入完全（或接近）有序，仍需 $\Theta(n \log n)$ 时间——改进...

### 3. 列表

循位置访问

邓俊辉

deng@tsinghua.edu.cn

## 从静态到动态

❖ 根据是否修改数据结构，所有操作大致分为两类方式

- 1 ) 静态：仅读取，数据结构的内容及组成一般不变：get、search
- 2 ) 动态：需写入，数据结构的局部或整体将改变：insert、remove

❖ 与操作方式相对应地，数据元素的存储与组织方式也分为两种

- 1 ) 静态：数据空间整体创建或销毁

数据元素的物理存储次序与其逻辑次序严格一致；可支持高效的静态操作

比如向量，元素的物理地址与其逻辑次序线性对应

- 2 ) 动态：为各数据元素动态地分配和回收的物理空间

相邻元素记录彼此的物理地址，在逻辑上形成一个整体；可支持高效的动态操作

## 从向量到列表

❖ 列表 ( list ) 是采用动态储存策略的典型结构

其中的元素称作节点 ( node )

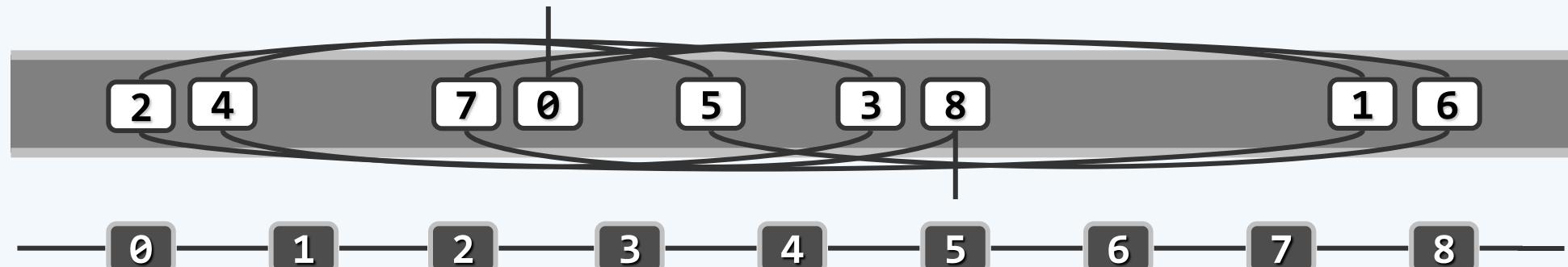
各节点通过指针或引用彼此联接，在逻辑上构成一个线性序列

$$L = \{ a_0, a_1, \dots, a_{n-1} \}$$

❖ 相邻节点彼此互称前驱 ( predecessor ) 或后继 ( successor )

前驱或后继若存在，则必然唯一

没有前驱/后继的唯一节点称作首 ( first/front ) / 末 ( last/rear ) 节点



## 从秩到位置

❖ 向量支持循秩访问 (call-by-rank) 的方式

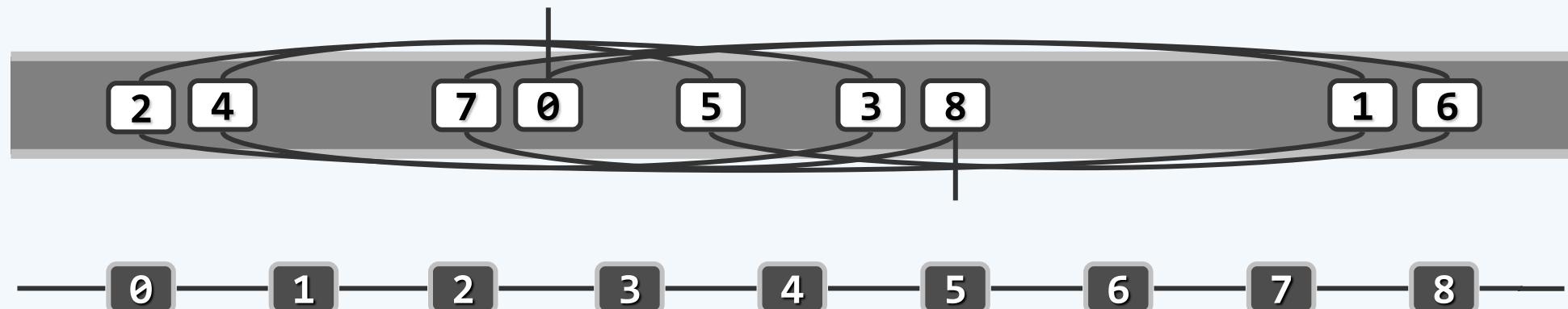
根据数据元素的秩，可在 $\mathcal{O}(1)$ 时间内直接确定其物理地址

$v[i]$ 的物理地址 =  $v + i \times s$ ,  $s$ 为单个单元占用的空间量

❖ 比喻：假设沿北京市海淀区的街道 $v$ ，各住户的地理间距均为 $s$

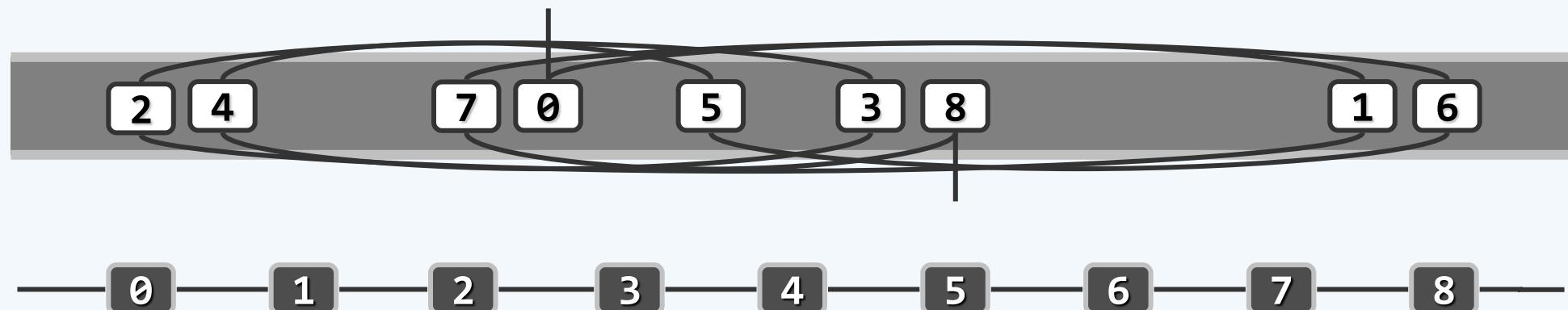
则对于门牌号为 $i$ 的住户，地理位置 =  $v + i \times s$

❖ 这种高效的方式，可否被列表沿用？



## 从秩到位置

- ❖ 既然同属线性序列，列表固然也可通过秩来定位节点：从头/尾端出发，沿后继/前驱引用...
- ❖ 然而，此时的循秩访问成本过高，已不合时宜    `//List::operator[](Rank r)`，下节详解  
`//兼顾两种访问方式的skiplist`，第九章
- ❖ 因此，应改用**循位置访问**（call-by-position）的方式  
 亦即，转而利用节点之间的相互**引用**，找到特定的节点
- ❖ 比喻：找到 我的朋友A的亲戚B的同事C的战友D的...的同学Z



### 3. 列表

## 接口与实现

百只骆驼绕山走，九十八只在山后

尾驼露尾不见头，头驼露头出山沟

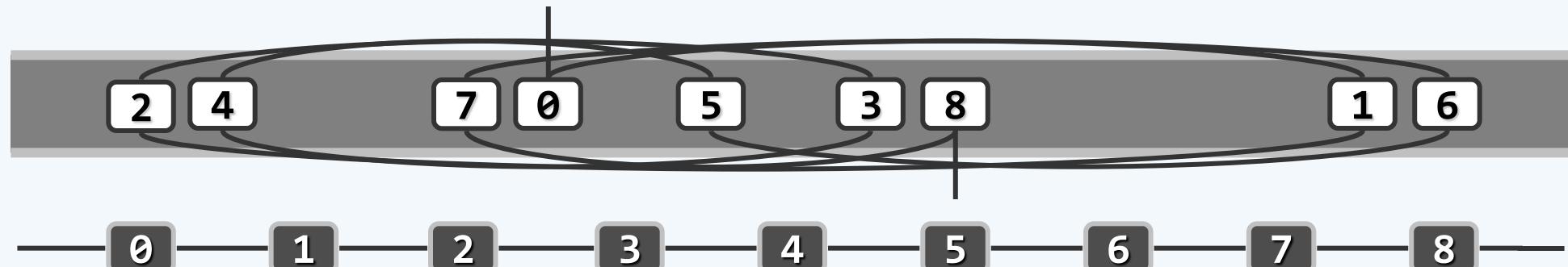
邓俊辉

deng@tsinghua.edu.cn

## 列表节点：ADT接口

作为列表的基本元素，列表节点首先需要独立地“封装”实现  
为此，可设置并约定若干基本的操作接口

操作	功能
<code>pred()</code>	当前节点前驱节点的位置
<code>succ()</code>	当前节点后继节点的位置
<code>data()</code>	当前节点所存数据对象
<u><code>insertAsPred(e)</code></u>	插入前驱节点，存入被引用对象e，返回新节点位置
<u><code>insertAsSucc(e)</code></u>	插入后继节点，存入被引用对象e，返回新节点位置



## 列表节点：模板类

- ❖ `#define Posi(T) ListNode<T>* //列表节点位置 ( ISO C++0x, template alias )`
- ❖ `template <typename T> //简洁起见，完全开放而不再过度封装`
- `struct ListNode { //列表节点模板类（以双向链表形式实现）`
- `T data; //数值`
- `Posi(T) pred; //前驱`
- `Posi(T) succ; //后继`
- `ListNode() {} //针对header和trailer的构造`
- `ListNode(T e, Posi(T) p = NULL, Posi(T) s = NULL)`
- `: data(e), pred(p), succ(s) {} //默认构造器`
- `Posi(T) insertAsPred(T const& e); //前插入`
- `Posi(T) insertAsSucc(T const& e); //后插入`
- `};`

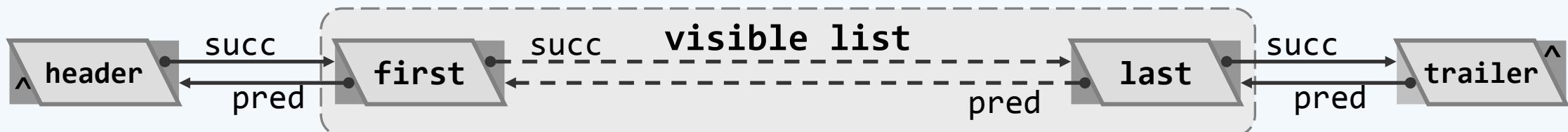


## 列表 : ADT 接口

操作接口	功能	适用对象
<code>size()</code>	报告列表当前的规模 ( 节点总数 )	列表
<code>first(), last()</code>	返回首、末节点的位置	列表
<code>insertAsFirst(e), insertAsLast(e)</code>	将e当作首、末节点插入	列表
<code>insertA(p, e), insertB(p, e)</code>	将e当作节点p的直接后继、前驱插入	列表
<code>remove(p)</code>	删除位置p处的节点，返回其引用	列表
<code>disordered()</code>	判断所有节点是否已按非降序排列	列表
<code>sort()</code>	调整各节点的位置，使之按非降序排列	列表
<code>find(e)</code>	查找目标元素e，失败时返回NULL	列表
<code>search(e)</code>	查找e，返回不大于e且秩最大的节点	有序列表
<code>deduplicate(), uniquify()</code>	剔除重复节点	列表/有序列表
<code>traverse()</code>	遍历列表	列表

## 列表：模板类

- ❖ `#include "listNode.h" //引入列表节点类`
- ❖ `template <typename T> class List { //列表模板类`
- `private: int _size; //规模`
- `Posi(T) header; Posi(T) trailer; //头、尾哨兵`
- `protected: /* ... 内部函数 */`
- `public: /* ... 构造函数、析构函数、只读接口、可写接口、遍历接口 */`
- `};`



- ❖ 等效地，头、首、末、尾节点的秩可分别理解为 -1、0、n-1、n

## 构造

❖ template <typename T> void List<T>::init() { //初始化，创建列表对象时统一调用

```

header = new ListNode<T>; //创建头哨兵节点
trailer = new ListNode<T>; //创建尾哨兵节点
header->succ = trailer; header->pred = NULL; //互联
trailer->pred = header; trailer->succ = NULL; //互联
_size = 0; //记录规模
}

```



### 3. 列表

#### 无序列表：循秩访问

Don't lose the link.

- Robin Milner

邓俊辉

deng@tsinghua.edu.cn

❖ 可否模仿向量的循秩访问方式？

❖ 可以，比如，通过重载下标操作符

❖ `template <typename T> //assert: 0 <= r < size`

`T List<T>::operator[]( Rank r ) const { // $\theta(r)$ 效率低下，可偶尔为之，却不宜常用`

`Posi(T) p = first(); //从首节点出发`

`while ( 0 < r-- ) p = p->succ; //顺数第r个节点即是`

`return p->data; //目标节点`

`} //任一节点的秩，亦即其前驱的总数`

❖ 时间复杂度为  $\theta(r)$ ，线性正比于待访问节点的秩

以均匀分布为例，单次访问的期望复杂度为

$$(1 + 2 + 3 + \dots + n) / n = (n + 1) / 2 = \theta(n)$$

### 3. 列表

无序列表：查找

邓俊辉

deng@tsinghua.edu.cn

❖ 在节点p（可能是trailer）的n个（真）前驱中，找到等于e的最后者

❖ template <typename T> //从外部调用时， $0 \leq n \leq \text{rank}(p) < \_size$

```
Posi(T) List<T>::find( T const & e, int n, Posi(T) p ) const { //O(n)
```

while (  $0 < n--$  ) //顺序查找：从右向左，逐个将p的前驱与e比对

```
if ( e == ( p = p->pred )->data ) //这里假定类型T已重载操作符 “==”
```

return p; //直至命中或范围越界

return NULL; //若越出左边界，意味着查找失败

} //header的存在使得处理更为简洁

❖ 典型的调用模式：通过返回值判定

```
x = L.find( e, n, p ) ?  
  
    cout << x->data :  
  
    cout << "not found";
```

❖ Posi(T) find( T const & e ) const { //重载全局查找接口

```
return find( e, _size, trailer ); //从内部调用时，rank(trailer) == _size  
}
```

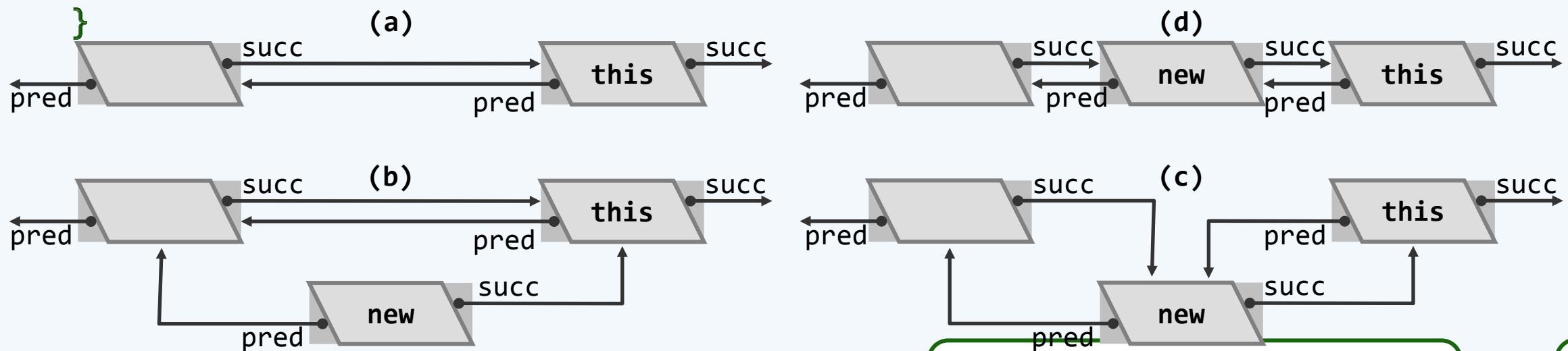
### 3. 列表

无序列表：插入

邓俊辉

deng@tsinghua.edu.cn

- ❖ `template <typename T> Posi(T) List<T>::insertB( Posi(T) p, T const & e )`  
`{ _size++; return p->insertAsPred( e ); } //e当作p的前驱插入 ( Before )`
- ❖ `template <typename T> //前插入算法 ( 后插入算法完全对称 )`  
`Posi(T) ListNode<T>::insertAsPred( T const & e ) { //O(1)`  
`Posi(T) x = new ListNode( e, pred, this ); //创建 ( 耗时100倍 )`  
`pred->succ = x; pred = x; return x; //建立链接，返回新节点的位置`



### 3. 列表

无序列表：基于复制的构造

邓俊辉

deng@tsinghua.edu.cn

❖ template <typename T> //基本接口

```
void List<T>::copyNodes( Posi(T) p, int n ) { //O(n)
    init(); //创建头、尾哨兵节点并做初始化
    while (n--) //将起自p的n项依次作为末节点插入
        { insertAsLast( p->data ); p = p->succ; }
}
```

❖ 重载的接口

```
List<T>::List( List<T> const & L ) //O(_size)
{ copyNodes( L.first(), L._size ); }

List<T>::List( List<T> const & L, int r, int n ) //O(r + n)
{ copyNodes( L[r], n ); }
```

### 3. 列表

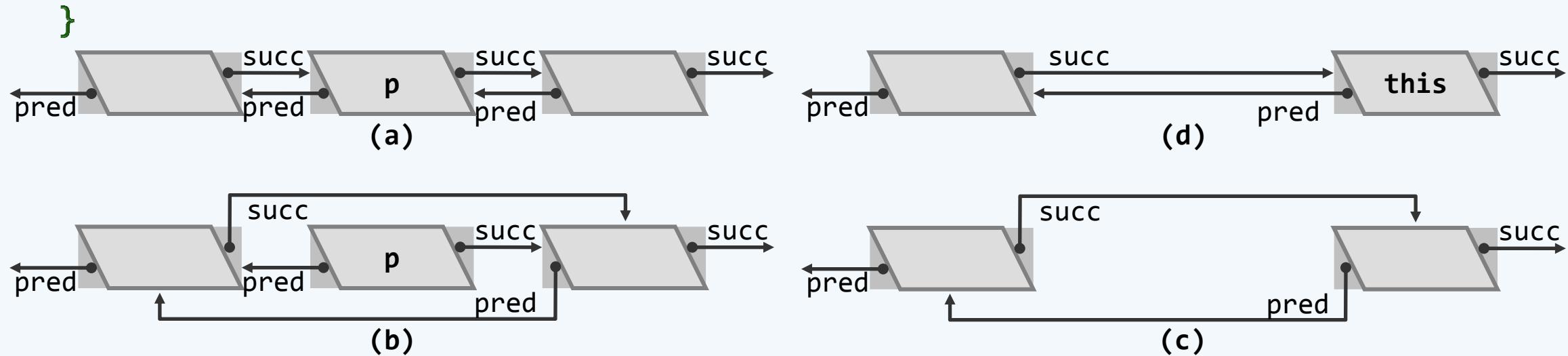
无序列表：删除

邓俊辉

deng@tsinghua.edu.cn

❖ template <typename T> //删除合法位置p处节点，返回其数值

```
T List<T>::remove( Posi(T) p ) { //O(1)
    T e = p->data; //备份待删除节点数值（设类型T可直接赋值）
    p->pred->succ = p->succ;
    p->succ->pred = p->pred;
    delete p; _size--; return e; //返回备份数值
}
```



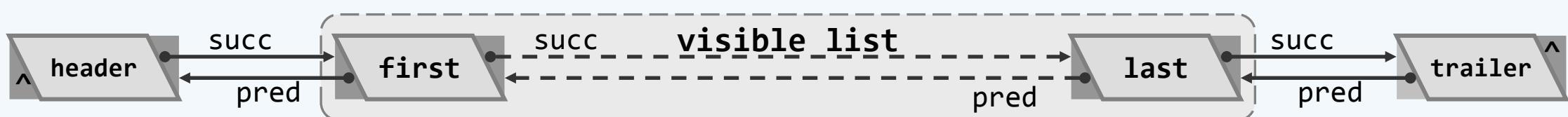
### 3. 列表

无序列表：析构

邓俊辉

deng@tsinghua.edu.cn

- ❖ `template <typename T> List<T>::~List() //列表析构`  
`{ clear(); delete header; delete trailer; } //清空列表，释放头、尾哨兵节点`
- ❖ `template <typename T> int List<T>::clear() { //清空列表`  
`int oldSize = _size;`  
`while ( 0 < _size ) //反复删除首节点，直至列表变空`  
`remove( header->succ );`  
`return oldSize;`  
`} //O(n)，线性正比于列表规模`



- ❖ 若`remove( header->succ )`改作`remove( trailer->pred )`呢？

### 3. 列表

无序列表：去重

邓俊辉

deng@tsinghua.edu.cn

```

❖ template <typename T> int List<T>::deduplicate() { //剔除无序列表中的重复节点
    if (_size < 2) return 0; //平凡列表自然无重复
    int oldSize = _size; //记录原规模
    Posi(T) p = first(); Rank r = 1; //p从首节点起
    while (trailer != (p = p->succ)) { //依次直到末节点
        Posi(T) q = find( p->data, r, p ); //在p的r个(真)前驱中，查找与之雷同者
        q ? remove(q) : r++; //若的确存在，则删除之；否则秩递增——可否remove(p)？
    } //assert: 循环过程中的任意时刻，p的所有前驱互不相同
    return oldSize - _size; //列表规模变化量，即被删除元素总数
} //正确性及效率分析的方法与结论，与Vector::deduplicate()相同

```

### 3. 列表

无序列表：遍历

邓俊辉

deng@tsinghua.edu.cn

❖ template <typename T>

```
void List<T>::traverse( void ( * visit )( T & ) ) { //函数指针  
    Posi(T) p = header;  
    while ( ( p = p->succ ) != trailer ) visit( p->data );  
}
```

❖ template <typename T>

template <typename VST>

```
void List<T>::traverse( VST & visit ) { //函数对象
```

```
    Posi(T) p = header;
```

```
    while ( ( p = p->succ ) != trailer ) visit( p->data );
```

```
}
```

### 3. 列表

有序列表：唯一化

邓俊辉

deng@tsinghua.edu.cn

```

❖ template <typename T> int List<T>::uniquify() { //剔除重复元素
    if ( _size < 2 ) return 0; //平凡列表自然无重复
    int oldSize = _size; //记录原规模

    ListNodePosi(T) p = first();; //p为各区段起点
    ListNodePosi(T) q; //q为其后继

    while ( trailer != ( q = p->succ ) ) //反复考查紧邻的节点对(p, q)
        if ( p->data != q->data ) p = q; //若互异，则转向下一区段
        else remove(q); //否则(雷同)，删除后者

    return oldSize - _size; //规模变化量，即被删除元素总数
} //只需遍历整个列表一趟， $\Theta(n)$ 

```

### 3. 列表

有序列表：查找

邓俊辉

deng@tsinghua.edu.cn

- ❖ 在有序列表内节点p的n个(真)前驱中，找到不大于e的**最后**者

- ❖ `template <typename T>`

```

Posi(T) List<T>::search(T const & e, int n, Posi(T) p) const {
    while ( 0 <= n-- ) //对于p的最近的n个前驱，从右向左
        if ( ( ( p = p->pred ) -> data ) <= e ) break; //逐个比较

    return p; //直至命中、数值越界或范围越界后，返回查找终止的位置
} //最好O(1)，最坏O(n)；等概率时平均O(n)，正比于区间宽度

```

- ❖ `template <typename T>`
- `Posi(T) List<T>::search(T const & e, int n, Posi(T) p) const ;`
- ❖ 语义与向量相似，便于插入排序等后续操作：`insertA( search(e, r, p), e )`
- ❖ 为何未能借助有序性提高查找效率？实现不当，还是根本不可能？
- ❖ 按照循位置访问的方式，物理存储地址与其逻辑次序无关  
依据秩的随机访问无法高效实现，而只能依据元素间的引用顺序访问

### 3. 列表

#### 选择排序

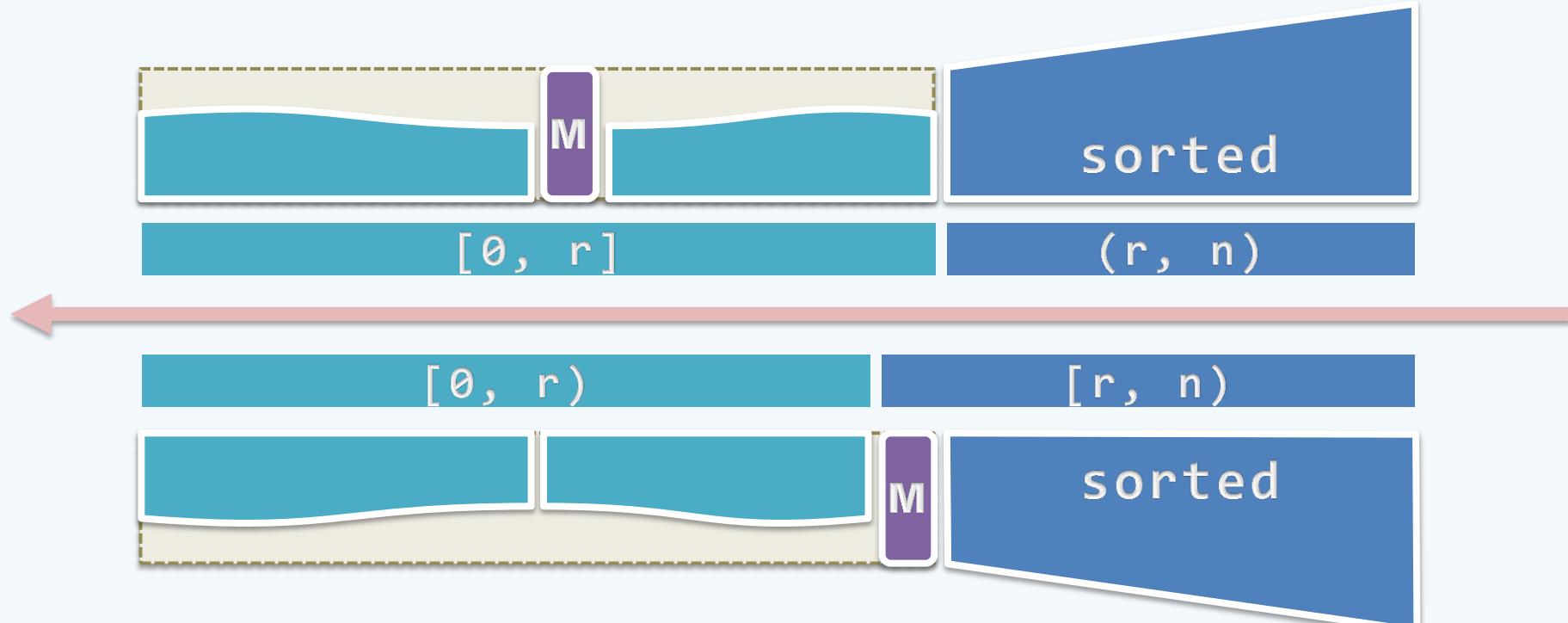
当下又选了几样果菜与凤姐送去，  
凤姐儿也送了几样来。

邓俊辉

deng@tsinghua.edu.cn

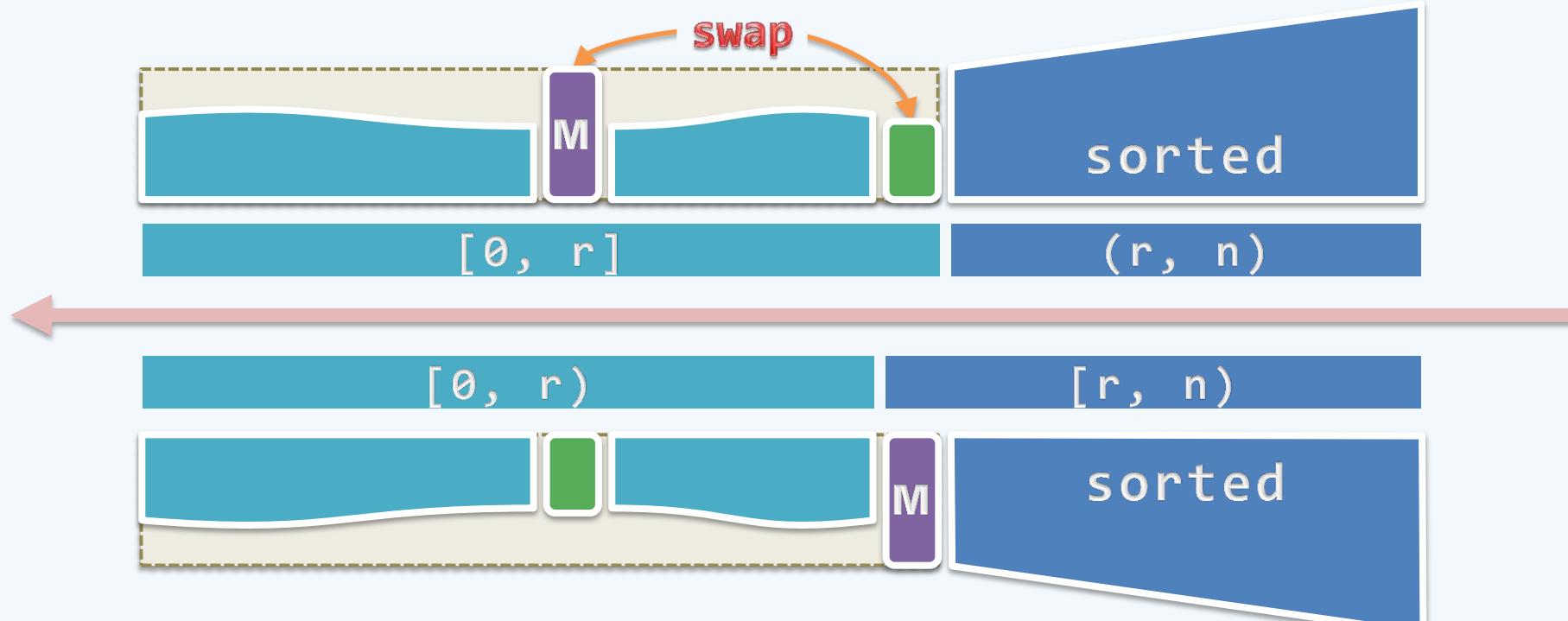
## 回忆起泡排序...

- ❖ 累计需要 $\Theta(n^2)$ 时间，是因为每趟扫描交换都需要 $\Theta(n)$ 时间： $\Theta(n)$ 次比较 +  $\Theta(n)$ 次交换
- ❖  $\Theta(n)$ 次比较或许无可厚非，但 $\Theta(n)$ 次交换绝对没有必要 //大量无谓的逆向移动



## 回忆起泡排序...

- 每趟扫描交换的实质效果，无非就是通过比较找到当前的最大元素M，并通过交换使之就位
- 如此看来，在经 $O(n)$ 次比较确定M之后，仅需一次交换即足矣

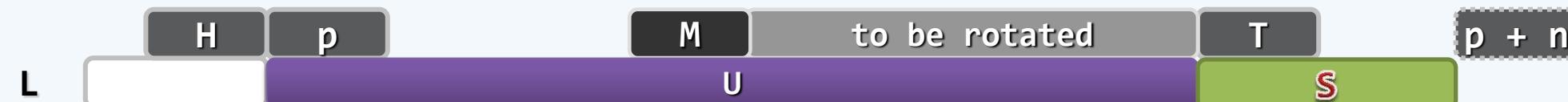


## 实例

迭代轮次	前缀无序子序列							后缀有序子序列						
0	5	2	7	4	6	3	1				^			
1	5	2	4	6	3	1					7			
2	5	2	4	3	1					6	7			
3	2	4	3	1					5	6	7			
4	2	3	1					4	5	6	7			
5	2	1						3	4	5	6	7		
6	1							2	3	4	5	6	7	
7	^							1	2	3	4	5	6	7

## selectionSort()

```
//对列表中起始于位置p的连续n个元素做选择排序，valid(p) && rank(p) + n <= size
template <typename T> void List<T>::selectionSort( Posi(T) p, int n ) {
    Posi(T) head = p->pred; Posi(T) tail = p; //待排序区间(head, tail)
    for ( int i = 0; i < n; i++ ) tail = tail->succ; //head/tail可能是头/尾哨兵
    while ( 1 < n ) { //反复从(非平凡)待排序区间内找出最大者，并移至有序区间前端
        insertB( tail, remove( selectMax( head->succ, n ) ) ); //改进...
        tail = tail->pred; n--; //待排序区间、有序区间的范围，均同步更新
    }
}
```



## selectMax()



❖ template <typename T> //从起始于位置p的n个元素中选出最大者， $1 < n$

```

Posi(T) List<T>::selectMax( Posi(T) p, int n ) { //Θ(n)

    Posi(T) max = p; //最大者暂定为p

    for ( Posi(T) cur = p; 1 < n; n-- ) //后续节点逐一与max比较

        if ( ! lt( ( cur = cur->succ )->data , max->data ) ) //若 >= max

            max = cur; //则更新最大元素位置记录

    return max; //返回最大节点位置
}

```

## 稳定性



- ❖ 有多个重复元素同时命中时，往往需要按照某种附加的约定，返回其中特定的某一个
- ❖ 比如，通常都约定“靠后者优先返回”
- ❖ 为此，必须采用比较器 `!lt()` 或 `ge()`，即等效于后者优先

2	<b>6a</b>	4	<b>6b</b>	3	0	<b>6c</b>	1	5	7	8	9
2	<b>6a</b>	4	<b>6b</b>	3	0	1	5	<b>6c</b>	7	8	9
2	<b>6a</b>	4	3	0	1	5	<b>6b</b>	<b>6c</b>	7	8	9
2	4	3	0	1	5	<b>6a</b>	<b>6b</b>	<b>6c</b>	7	8	9

- ❖ 如此即可保证，重复元素在列表中的次序与其插入次序一致

## 性能分析

❖ 共迭代 $n$ 次，在第 $k$ 次迭代中

selectMax() 为  $\Theta(n - k)$  //算术级数

swap() 为  $O(1)$  //或 remove() + insertB()

故总体复杂度应为  $\Theta(n^2)$

❖ 尽管如此，元素移动操作远远少于起泡排序 //实际更为费时

也就是说， $\Theta(n^2)$ 主要来自于元素比较操作 //成本相对更低

❖ 可否...每轮只做  $O(n)$  次比较，即找出当前的最大元素？

❖ 可以！...利用高级数据结构，selectMax()可改进至  $O(\log n)$  //稍后分解

当然，如此立即可以得到  $O(n \log n)$  的排序算法 //保持兴趣

### 3. 列表

循环节

邓俊辉

deng@tsinghua.edu.cn

## Cycle

- ❖ 任何一个序列  $A[0, n)$ ，都可以分解为若干个循环节 //设元素之间可定义次序
- ❖ 任何一个序列  $A[0, n)$ ，都对应于一个有序序列  $S[0, n)$  //经排序之后
- ❖ 元素  $A[k]$  在  $S[]$  中对应的秩，记作  $r(A[k]) = r(k)$
- ❖ 元素  $A[k]$  所属的循环节是：  
$$A[k], A[r(k)], A[r(r(k))], A[r(r(r(k)))], \dots, A[r(\dots(r(k))\dots)] = A[k]$$
- ❖ 每个循环节，长度均不超过  $n$
- ❖ 循环节之间，没有重复元素

## 实例

❖ rank: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15  
A[]: j n p m a i g o d c h b k l f e  
S[]: a b c d e f g h i j k l m n o p  
r[]: 9 13 4 12 0 8 6 14 3 2 7 1 10 11 5 4

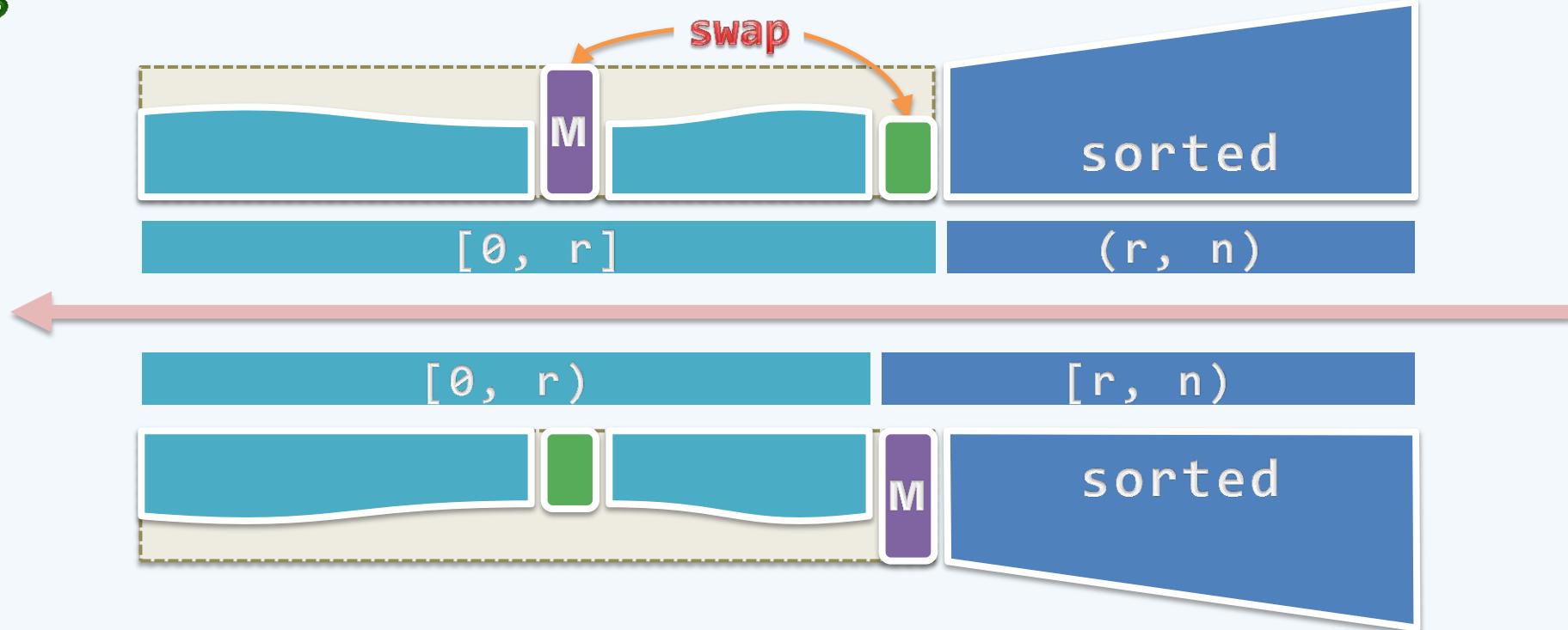
j	.	p	.	a	.	.	.	.	c	.	.	.	.	.	e
.	n	.	.	.	.	.	.	.	b	.	l	.	.	.	.
.	.	.	m	.	i	.	o	d	.	h	.	k	.	f	.
.	.	.	.	.	.	g	.	.	.	.	.	.	.	.	.

## 单调性

❖ 每迭代一步， $M$ 所属循环节的长度都恰好减少一个单位

//直到长度为1

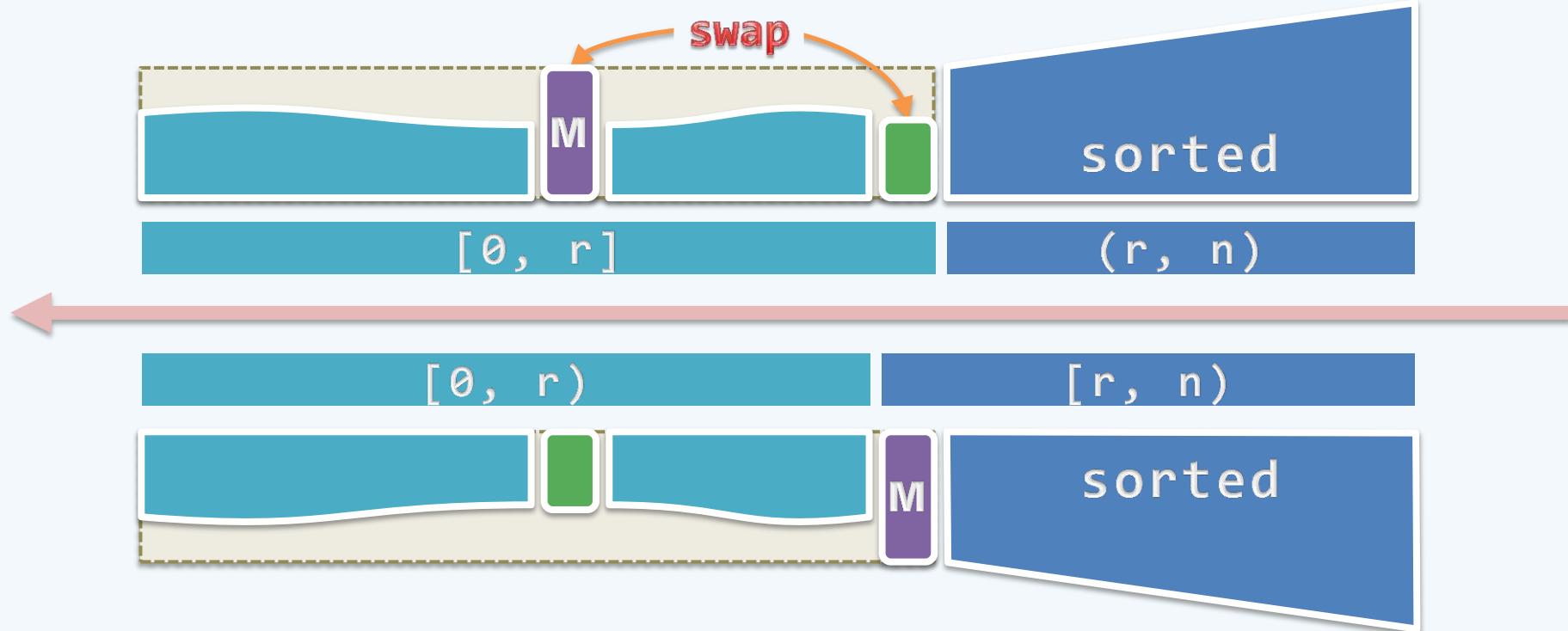
为什么？



❖ 观察： $r[M] = r$

**M与A[r]交换之后**

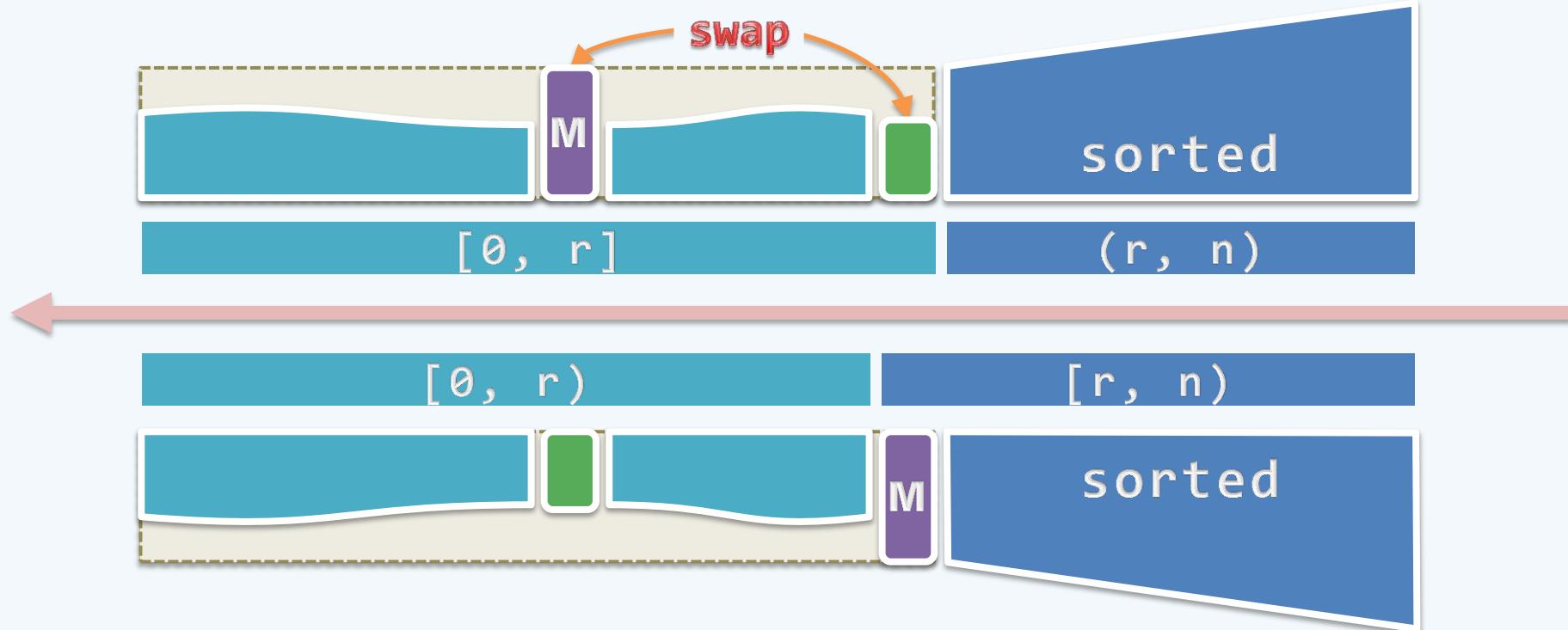
- ❖ M脱离原属的循环节，自成为一个长度为1的循环节



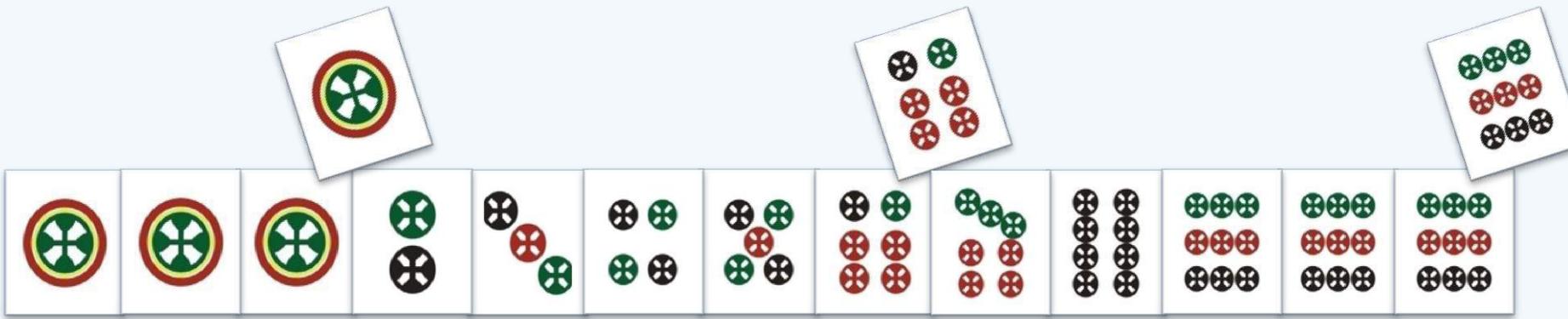
- ❖ 其余循环节保持不变

## 无效的交换

- ❖ **M**已经就位，无需交换 —— 这种情况会出现几次？



- ❖ 有多少个循环节，就出现几次 —— 最大值为  $n$ ，期望值为  $\Theta(\log n)$



### 3. 列表 插入排序

一语未了，只见宝玉笑嘻嘻的掮了一枝红梅进来，  
众丫鬟忙已接过，插入瓶内。

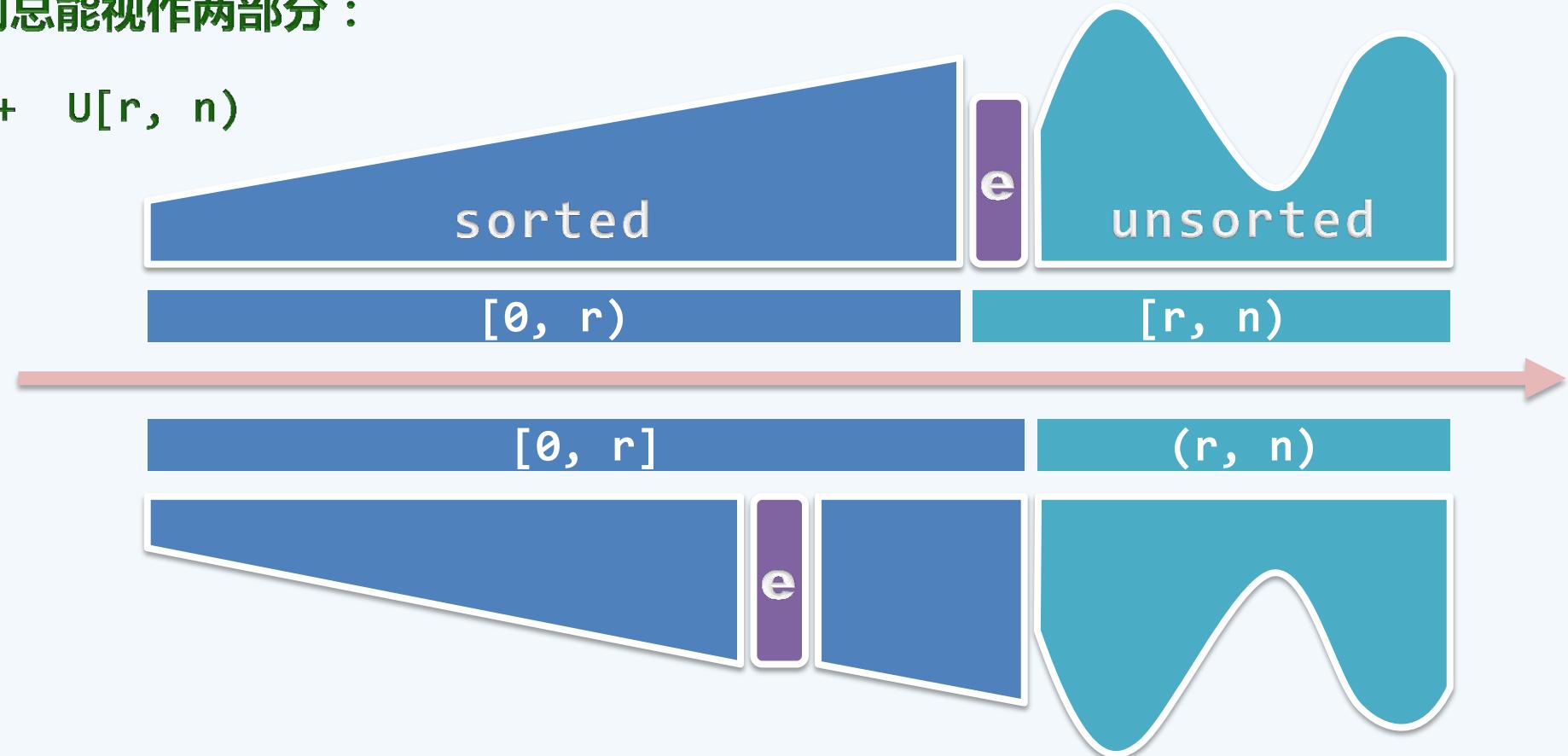
邓俊辉

deng@tsinghua.edu.cn

## 构思

- ❖ 【不变性】序列总能视作两部分：

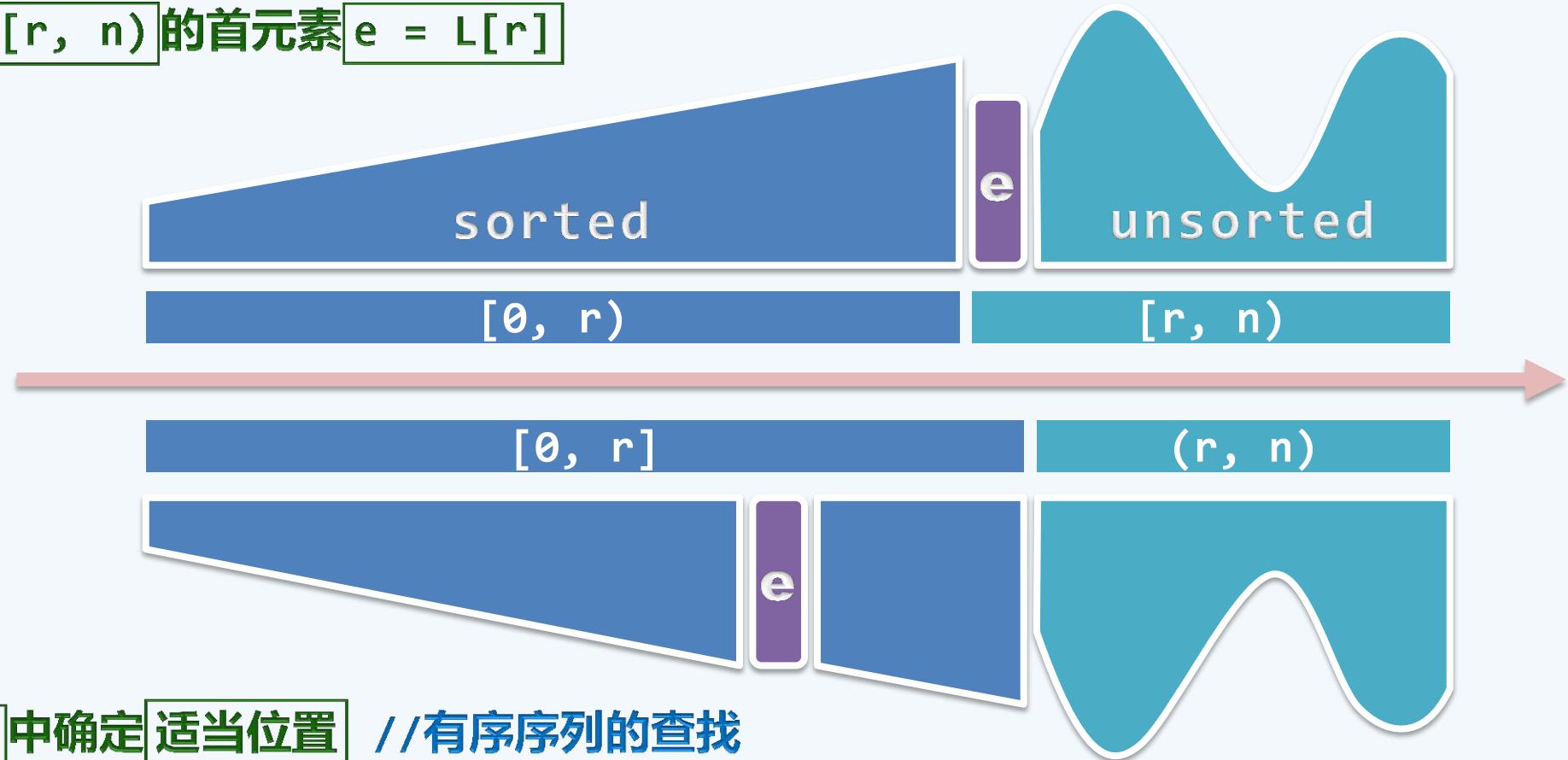
$s[0, r) + u[r, n)$



- ❖ 【初始化】： $|s| = r = 0$  //空序列无所谓有序或无序

## 减而治之

❖ 反复地，针对  $U[r, n]$  的首元素  $e = L[r]$



- 在  $S[0, r)$  中确定适当位置 // 有序序列的查找
- 插入  $e$ ，得到  $S[0, r]$  // 有序序列的插入

## 实例

迭代轮次	前缀有序子序列	当前元素	后缀无序子序列
-1	^	^	5 2 7 4 6 3 1
0	^	5	2 7 4 6 3 1
1	(5)	2	7 4 6 3 1
2	(2) 5	7	4 6 3 1
3	2 5 (7)	4	6 3 1
4	2 (4) 5 7	6	3 1
5	2 4 5 (6) 7	3	1
6	2 (3) 4 5 6 7	1	^
7	(1) 2 3 4 5 6 7	^	^

## 实现

```
//对列表中起始于位置p的连续n个元素做插入排序，valid(p) && rank(p) + n <= size
template <typename T> void List<T>::insertionSort( Posi(T) p, int n ) {
    for ( int r = 0; r < n; r++ ) { //逐一引入各节点，由sr得到sr+1
        insertA( search( p->data, r, p ), p->data ); //查找 + 插入
        p = p->succ; remove( p->pred ); //转向下一节点
    } //n次迭代，每次O(r + 1)
} //仅使用O(1)辅助空间，属于就地算法
```

- ❖ 紧邻于search()接口返回的位置之后插入当前节点，总是保持有序
- ❖ 验证各种情况下的正确性，体会哨兵节点的作用：

$s_r$ 中含有/不含与p相等的元素； $s_r$ 中的元素均严格小于/大于p

## 性能分析

### ❖ 最好情况：完全（或几乎）有序

- 每次迭代，只需 $1$ 次比较， $0$ 次交换
- 累计 $\Theta(n)$ 时间！

### ❖ 最坏情况：完全（或几乎）逆序

- 第 $k$ 次迭代，需 $\Theta(k)$ 次比较， $1$ 次交换  $//$ 改用向量呢？稍后分析
- 累计 $\Theta(n^2)$ 时间！

### ❖ 一般情况：包含 $I$ 个逆序对

- 第 $k$ 次迭代，只需 $\Theta(I_k)$ 次比较， $1$ 次交换  $//$ 为什么？
- 累计 $\Theta(n + I)$ ！  $//$ input-sensitive，对Shellsort至关重要

### ❖ 平均而言呢？

$//$ 当然，首先需要假定具体的随机分布...

## 平均性能：后向分析

❖ 假定：各元素的取值遵守均匀、独立分布

$L[\theta, r)$

$L[r, n)$

于是：平均要做多少次元素比较？

❖ 考查： $L[r]$ 刚插入完成的那一时刻（穿越？）

$L[\theta, r]$

$L(r, n)$

试问：此时的有序前缀 $L[\theta, r]$ 中，哪个元素是此前的 $L[r]$ ？

❖ 观察：其中的 $r + 1$ 个元素均有可能，且概率均等于 $1/(r + 1)$

❖ 因此，在刚完成的这次迭代中，为引入 $s[r]$ 所花费时间的数学期望为

$$[r + (r - 1) + \dots + 3 + 2 + 1 + 0] / (r + 1) + 1 = r/2 + 1$$

❖ 于是，总体时间的数学期望 =  $[0 + 1 + \dots + (n - 1)] / 2 + n = O(n^2)$

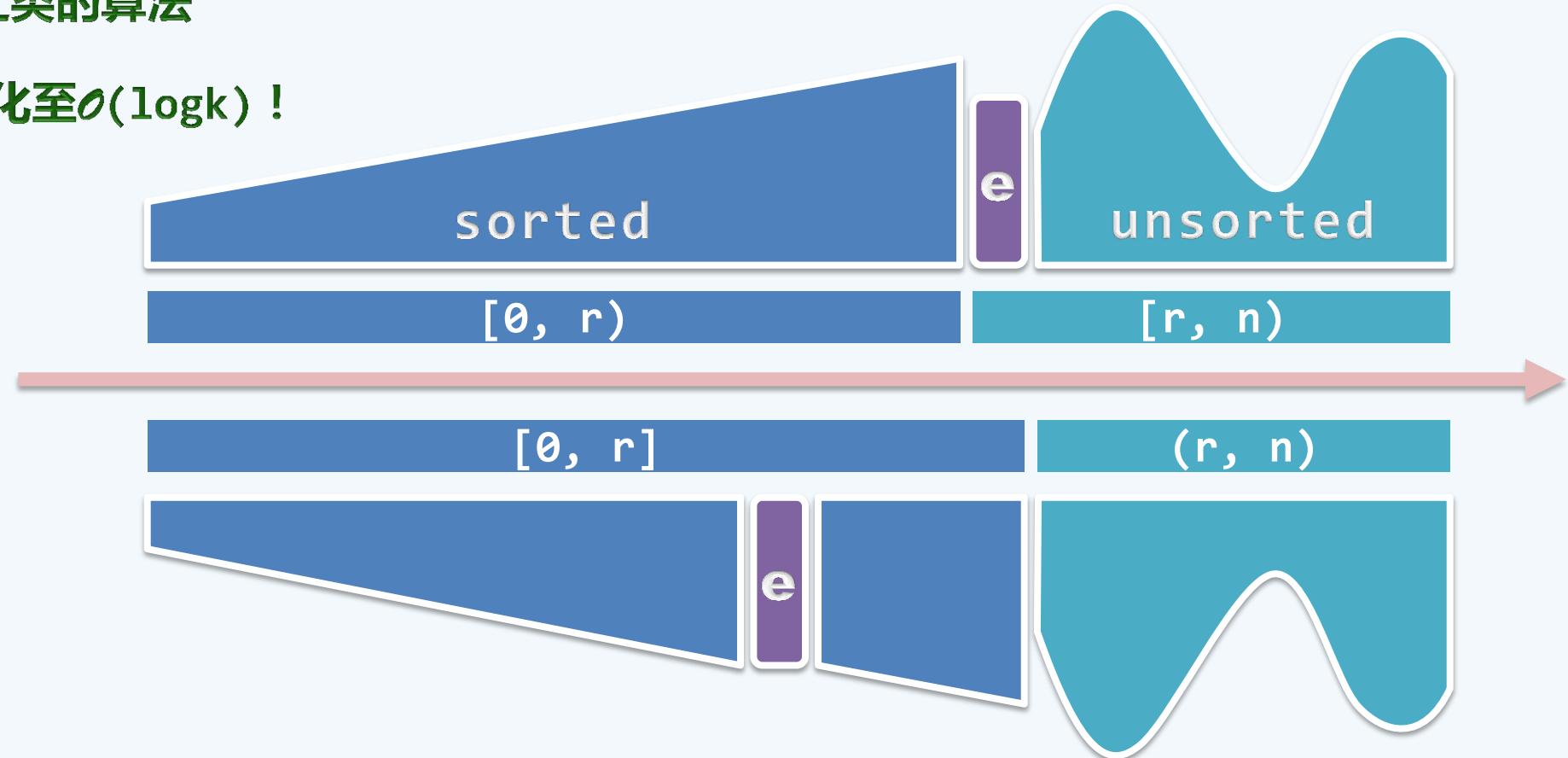
❖ 再问：在 $n$ 次迭代中，平均有多少次无需交换呢？

//习题[3-10]

若改用向量...

❖ 借助二分查找之类的算法

查找效率可优化至 $\mathcal{O}(\log k)$ !



❖ 然而，**总体性能**会否因此相应地提高？

### 3. 列表

#### 归并排序

四牡孔阜，六辔在手

骐骥是中，駔骊是骖

龙盾之合，鋈以觶軺

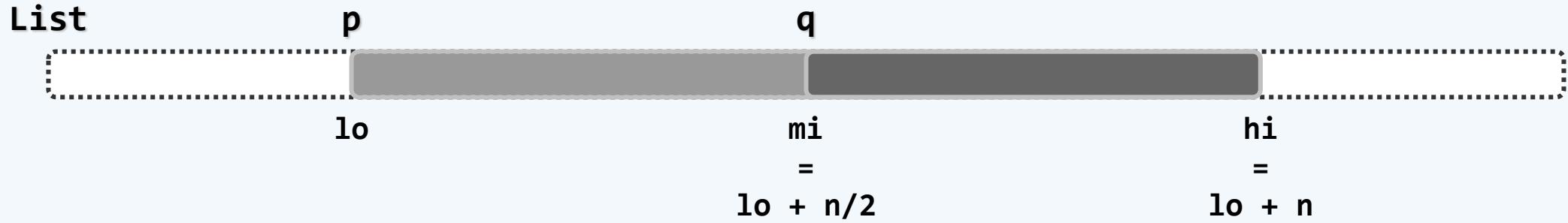
日两美其必合兮，孰信修而慕之？

思九州岛之博大兮，岂惟是其有女？

邓俊辉

deng@tsinghua.edu.cn

## 归并排序



```

template <typename T> //valid(p) && rank(p) + n <= size

void List<T>::mergeSort( Posi(T) & p, int n ) { //对起始于位置p的n个元素排序
  if ( n < 2 ) return; //待排序范围足够小时直接返回，否则...
  Posi(T) q = p; int m = n >> 1; //以中点为界
  for ( int i = 0; i < m; i++ ) q = q->succ; //均分列表
  mergeSort( p, m ); mergeSort( q, n - m ); //子序列分别排序
  merge( p, m, *this, q, n - m ); //归并
} //若归并可在线性时间内完成，则总体运行时间亦为O(nlogn)
  
```

## 二路归并

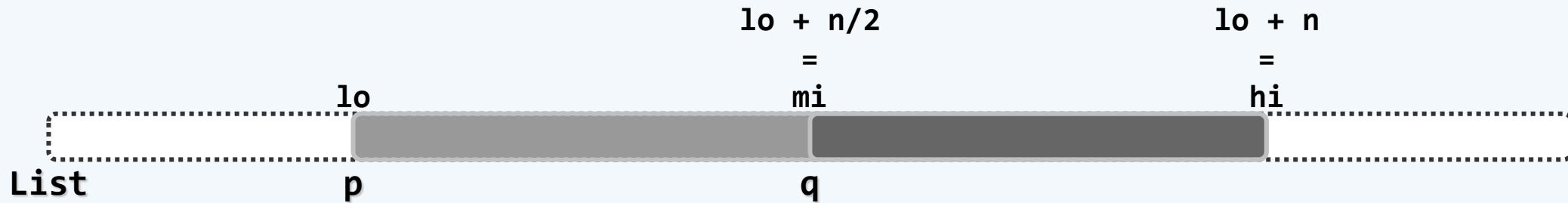
//当前列表中自p起的n个元素，与列表L中自q起的m个元素归并

```
template <typename T> // ( 归并排序时为同一列表 , this == L )
void List<T>::merge( Posi(T) & p, int n, List<T> & L, Posi(T) q, int m ) {
    while ( 0 < m ) //在q尚未移出区间之前
        if ( ( 0 < n ) && ( p->data <= q->data ) ) //若p仍在区间内且v(p) <= v(q)
            if ( q == ( p = p->succ ) ) break; n--;
        else //若p已超出右界或v(q) < v(p) , 则将q插至p之前

```

{ if ( q == ( p = p->succ ) ) break; n--; } //则将p直接后移

else //若p已超出右界或v(q) < v(p) , 则将q插至p之前



```
{ insertB( p, L.remove( ( q = q->succ )->pred ) ); m--; }
```

} //每经过一次迭代n + m必减少1，故总体运行时间为O(n + m)，线性正比于节点总数

### 3. 列表

逆序对

邓俊辉

deng@tsinghua.edu.cn

## Inversion

❖ 考查序列  $A[0, n)$ ，设元素之间可比较大小

❖  $[i, j]$  称作一个逆序对，如果  $0 \leq i < j < n$  且  $A[i] > A[j]$

❖ 为便于统计，可将逆序对统一记到 **后者** 的账上

❖ 例  $A[] = \{5, 3, 1, 4, 2\}$  中，共有  $0 + 1 + 2 + 1 + 3 = 7$  个逆序对

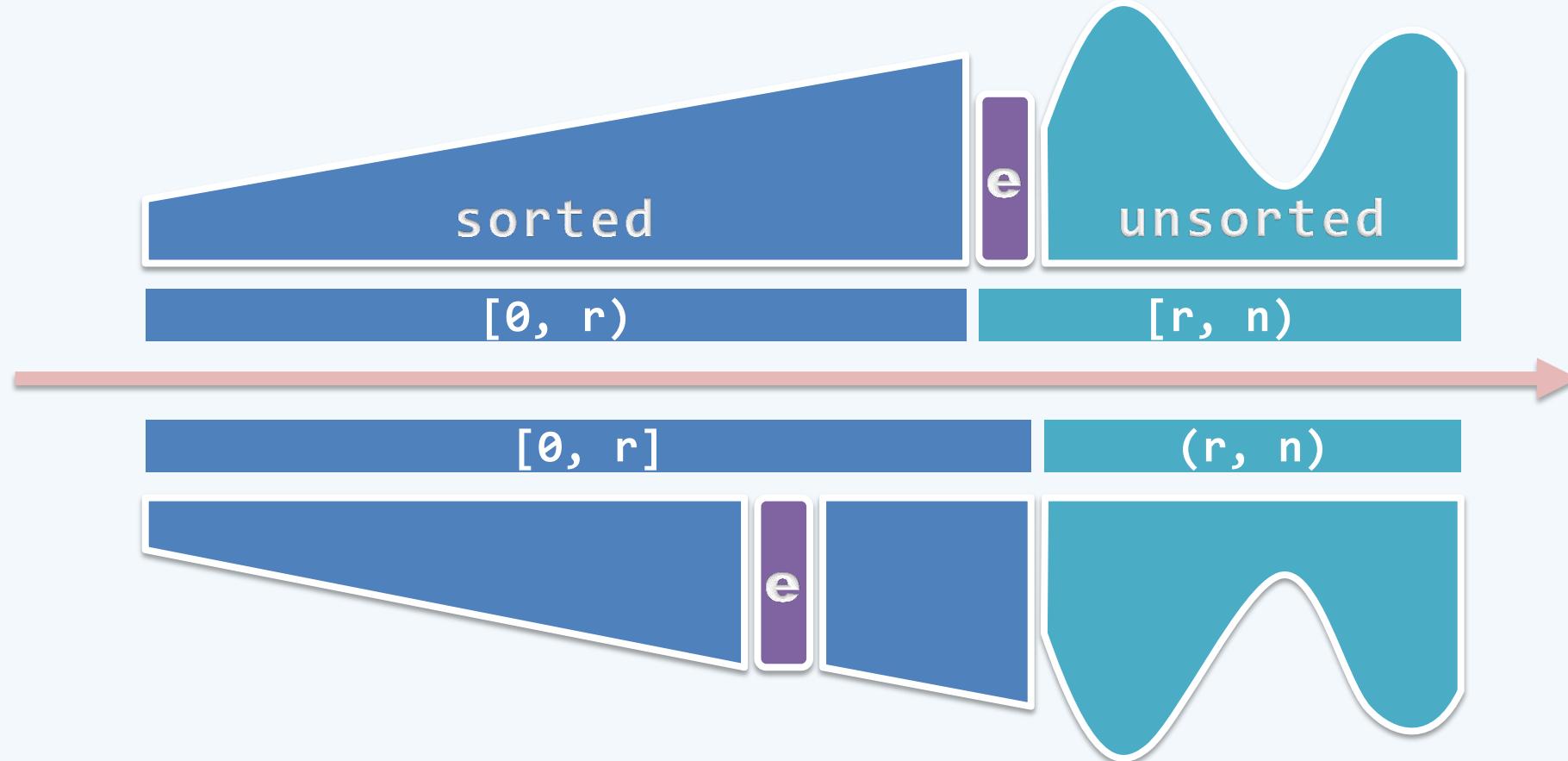
$A[] = \{1, 2, 3, 4, 5\}$  中，共有  $0 + 0 + 0 + 0 + 0 = 0$  个逆序对

$A[] = \{5, 4, 3, 2, 1\}$  中，共有  $0 + 1 + 2 + 3 + 4 = 10$  个逆序对

❖ 一般地，逆序对总数  $I \leq \binom{n}{2} = O(n^2)$

## 在insertionsort中

❖ 若  $e = A[r]$  账上的逆序对共有  $I(r)$  个，则在接下来的一步迭代中，恰好需要做  $I(r)$  次比较

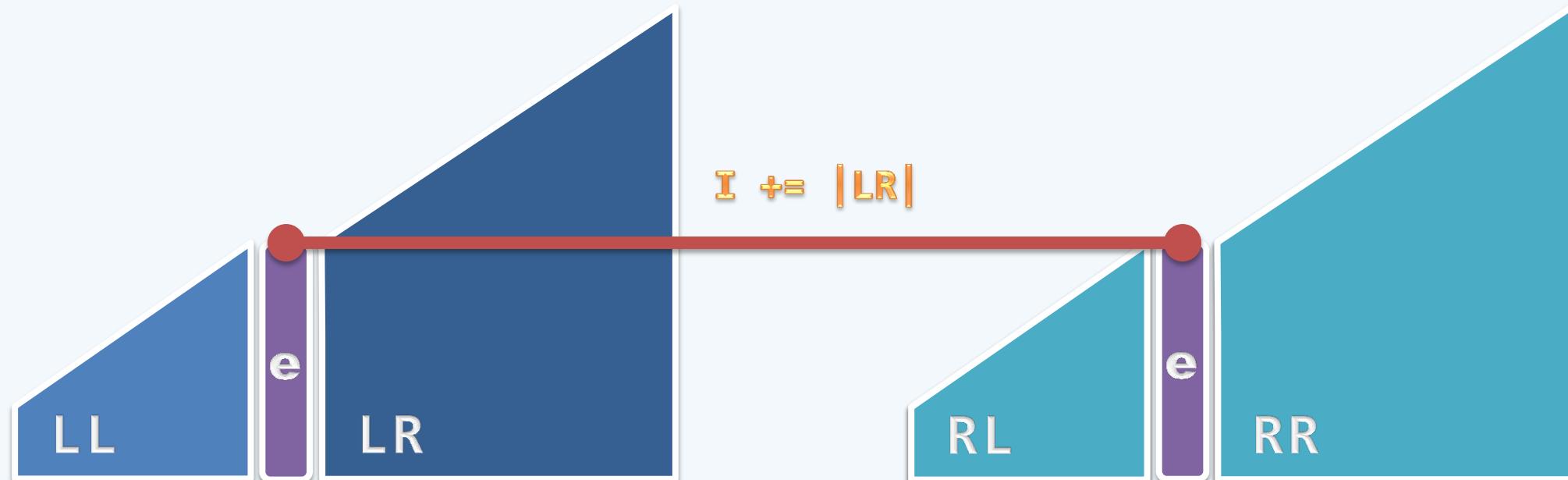


❖ 若共含  $I$  个逆序对，则关键码比较次数为  $\mathcal{O}(I)$ ，运行时间为  $\mathcal{O}(n + I)$

//习题[3-11]

## 计数

- ❖ 任意给定一个序列，如何统计其中逆序对的总数？
- ❖ 蛮力算法在最坏情况下，需要 $\Omega(n^2)$ 时间 //  $I = \binom{n}{2}$ 时
- ❖ 参照归并排序的框架，仅需 $O(n \log n)$ 时间 // 怎么做到的



### 3. 列表

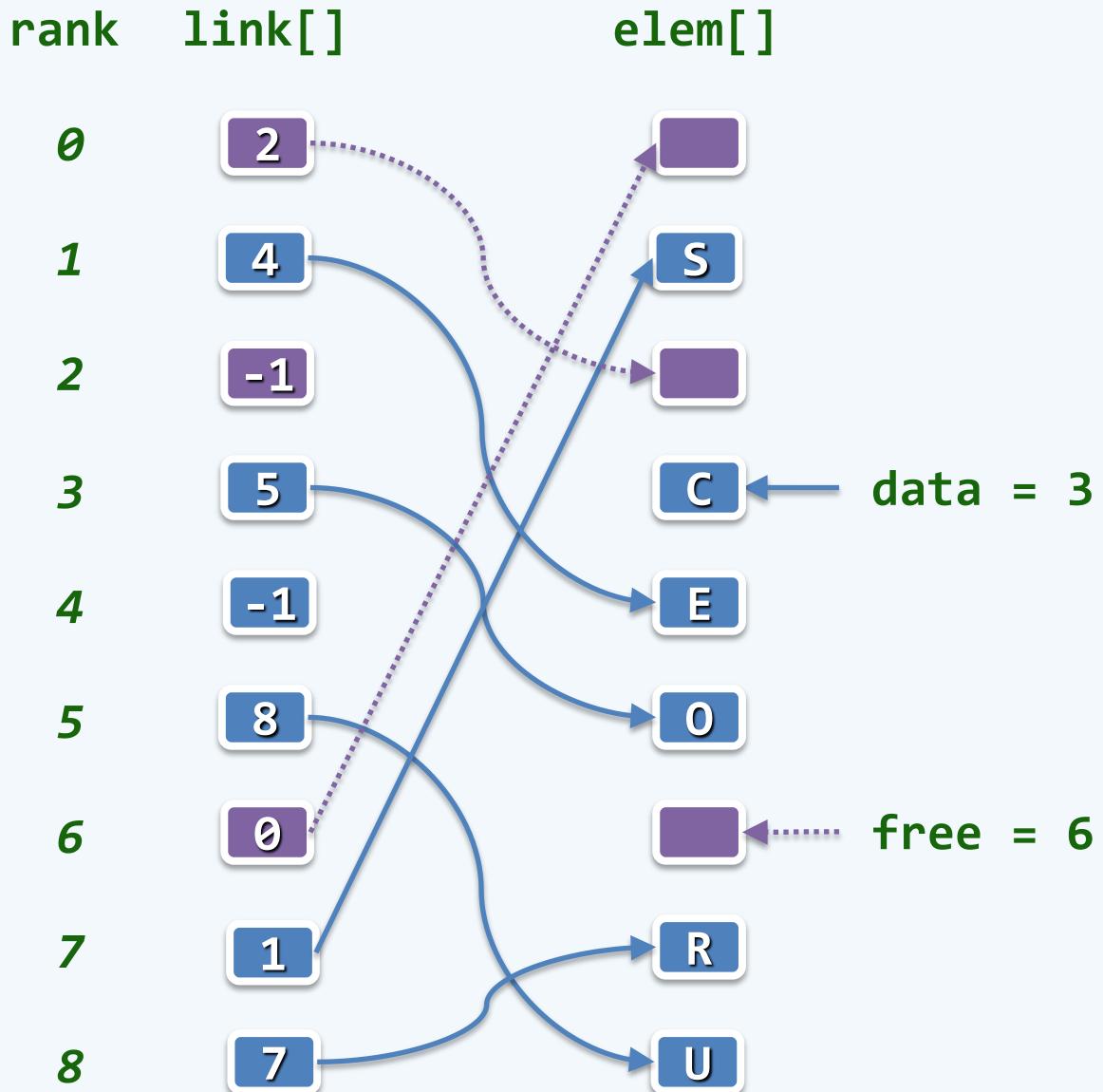
(xa) 游标实现

邓俊辉

deng@tsinghua.edu.cn

## 动机与构思

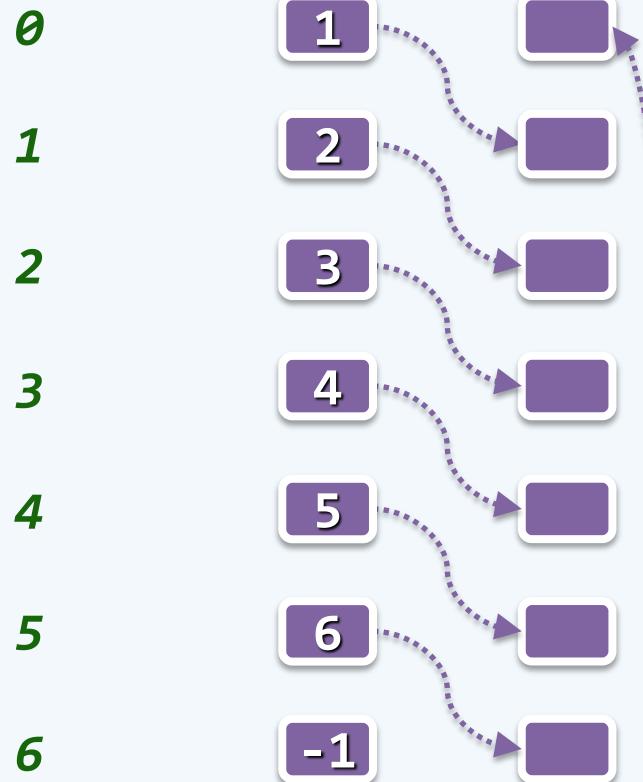
- ❖ 某些特定语言或环境中  
或者不（直接）支持指针  
或者不支持动态空间分配  
此时，如何实现列表结构呢？
- ❖ 利用线性数组，以游标方式模拟列表
  - elem[ ] : 对外可见的数据项**
  - link[ ] : 数据项之间的引用**
- ❖ 维护逻辑上互补的列表data和free
- ❖ 在插入或删除元素时，应如何调整？



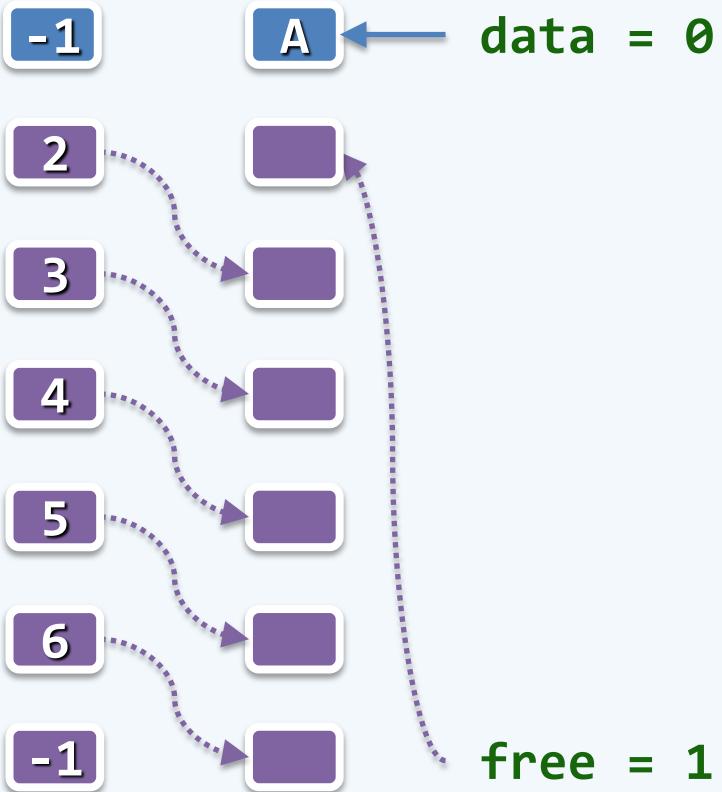
## 实例

`init(7)`

rank	link[]	elem[]
------	--------	--------

`data = -1``free = 0``insert('A')`

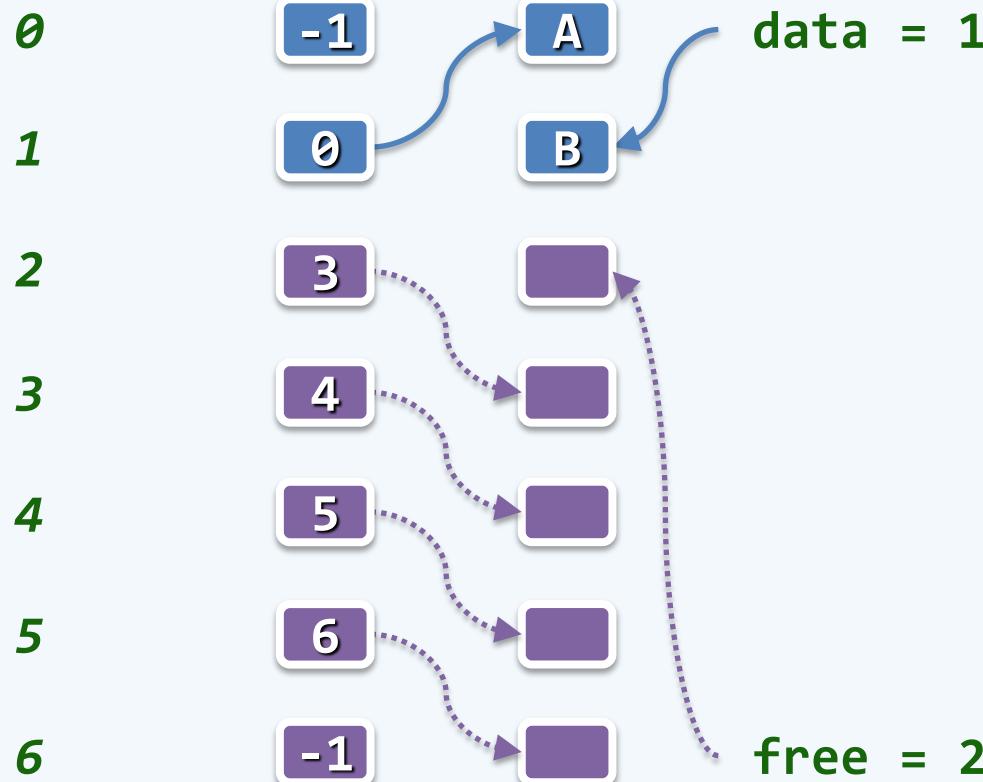
link[]	elem[]
--------	--------

`data = 0``free = 1`

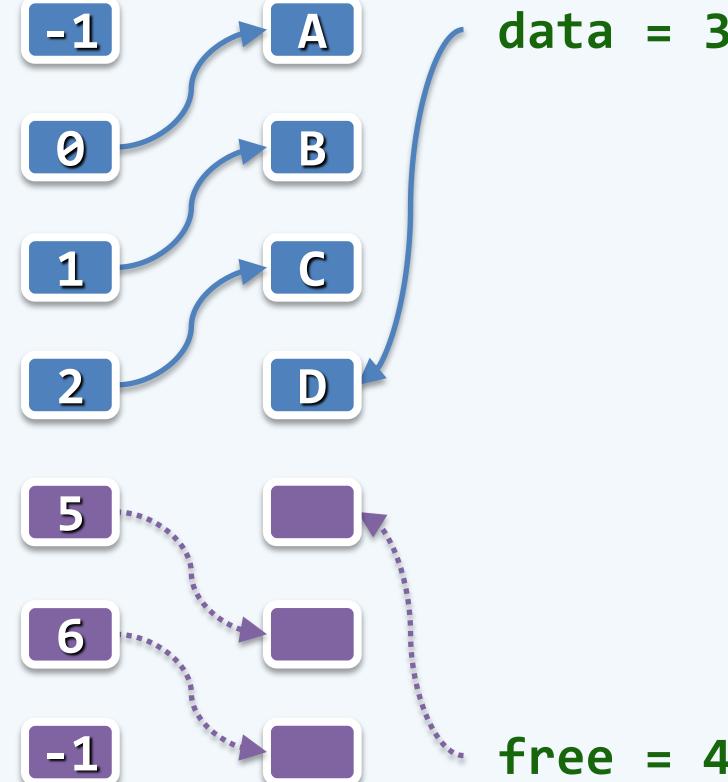
## 实例

insert('B')

rank link[] elem[]

insert('C')  
insert('D')

link[] elem[]



## 实例

`remove( 'A' )`

rank	link[]	elem[]
0	4	[ ]
1	-1	B
2	1	C
3	2	D
4	5	[ ]
5	6	[ ]
6	-1	[ ]

`data = 3``free = 0``remove( 'C' )`

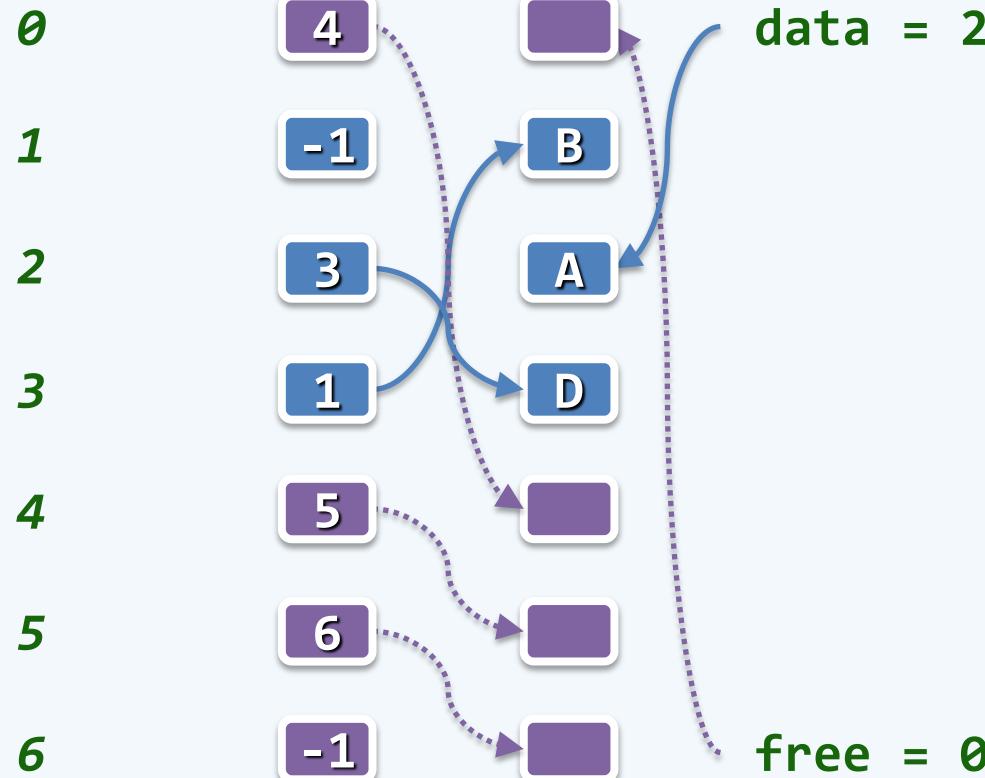
rank	link[]	elem[]
0	4	[ ]
1	-1	B
2	0	[ ]
3	1	D
4	5	[ ]
5	6	[ ]
6	-1	[ ]

`data = 3``free = 2`

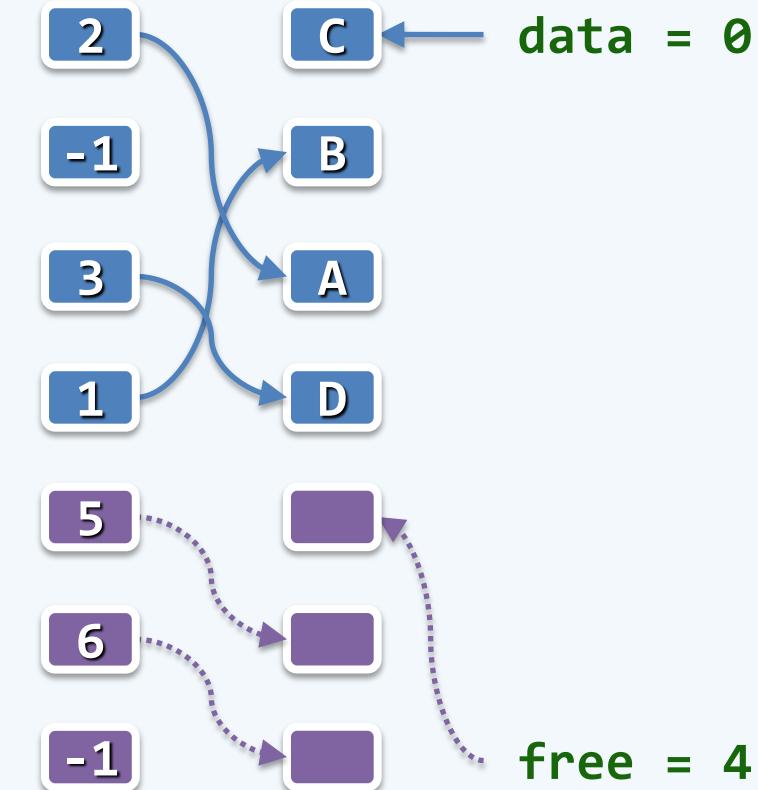
## 实例

`insert('A')`

rank	link[]	elem[]
------	--------	--------

`insert('C')`

rank	link[]	elem[]
------	--------	--------



### 3. 列表

(xb) Java序列

邓俊辉

deng@tsinghua.edu.cn

## Interface : 定义

- ❖ Java支持ADT的一种机制  
在同一接口规范下，允许不同的实现

- ❖ 实例

```
interface Geometry { //几何物体  
    final double PI = 3.1415926; //常量定义，类定义可直接使用  
    double area(); //无参数的接口方法  
    boolean inside(Point p); //带参数的接口方法  
}
```

- ❖ interface不能直接实例化为对象  
符合interface定义的任何类，都需要具体地**实现**其中的接口方法

## Interface : 实现

```
class Disk implements Geometry { //符合Geometry接口的Disk类  
    Point c;  double r;  
  
    public Disk(Point center, double radius) //构造方法  
    {c = center; r = radius;}  
  
    public double perimeter() { return 2 * PI * r; } //类方法  
    public double area() { return PI * r * r; } //接口方法的实现  
    public boolean inside(Point p) { //接口方法的实现  
        double dx = p.x - c.x, dy = p.y - c.y;  
        return dx*dx + dy*dy < r*r;  
    }  
}
```

## 向量接口 : Vector.java

```
public interface Vector {  
  
    public int getSize();  
  
    public boolean isEmpty();  
  
    public Object getAtRank(int r) throws ExceptionBoundaryViolation;  
  
    public Object replaceAtRank(int r, Object obj)  
        throws ExceptionBoundaryViolation;  
  
    public Object insertAtRank(int r, Object obj)  
        throws ExceptionBoundaryViolation;  
  
    public Object removeAtRank(int r) throws ExceptionBoundaryViolation;  
}  
}
```

## 向量实现1 : Vector\_Array.java

```
public class Vector_Array implements Vector {  
    private final int N = 1024; //数组容量固定  
    private Object[] A; private int n = 0;  
    public Vector_Array() { A = new Object[N]; n = 0; }  
    public int getSize() { return n; }  
    public boolean isEmpty() { return 0 == n; }  
    public Object insertAtRank(int r, Object obj) throws ExceptionBoundaryViolation {  
        if (0 > r || r > n) throw new ExceptionBoundaryViolation("out of range");  
        if (n >= N) throw new ExceptionBoundaryViolation("overflow");  
        for (int i = n; i > r; i--) A[i] = A[i - 1];  
        A[r] = obj; n++; return obj;  
    }  
    /* ..... */  
}
```

## 向量实现2 : Vector\_ExtArray.java

```
public class Vector_ExtArray implements Vector {  
    private int N = 8; //数组的初始容量，可不断增加  
    /* ..... */  
  
    public Object insertAtRank(int r, Object obj) throws ExceptionBoundaryViolation {  
        if (0 > r || r > n) throw new ExceptionBoundaryViolation("out of range");  
        if (N <= n) { //空间溢出的处理  
            N *= 2; Object B[] = new Object[N]; //容量加倍  
            for (int i = 0; i < n; i++) B[i] = A[i]; A = B; //用B[]替换A[]  
        }  
        for (int i = n; i > r; i--) A[i] = A[i - 1]; //后续元素顺次后移  
        A[r] = obj; n++; return obj;  
    }  
    /* ..... */  
}
```

## 序列接口及其实现

❖ //列表, List.java

```
interface List
{ /* ... */ }
```

//List\_DLNode.java

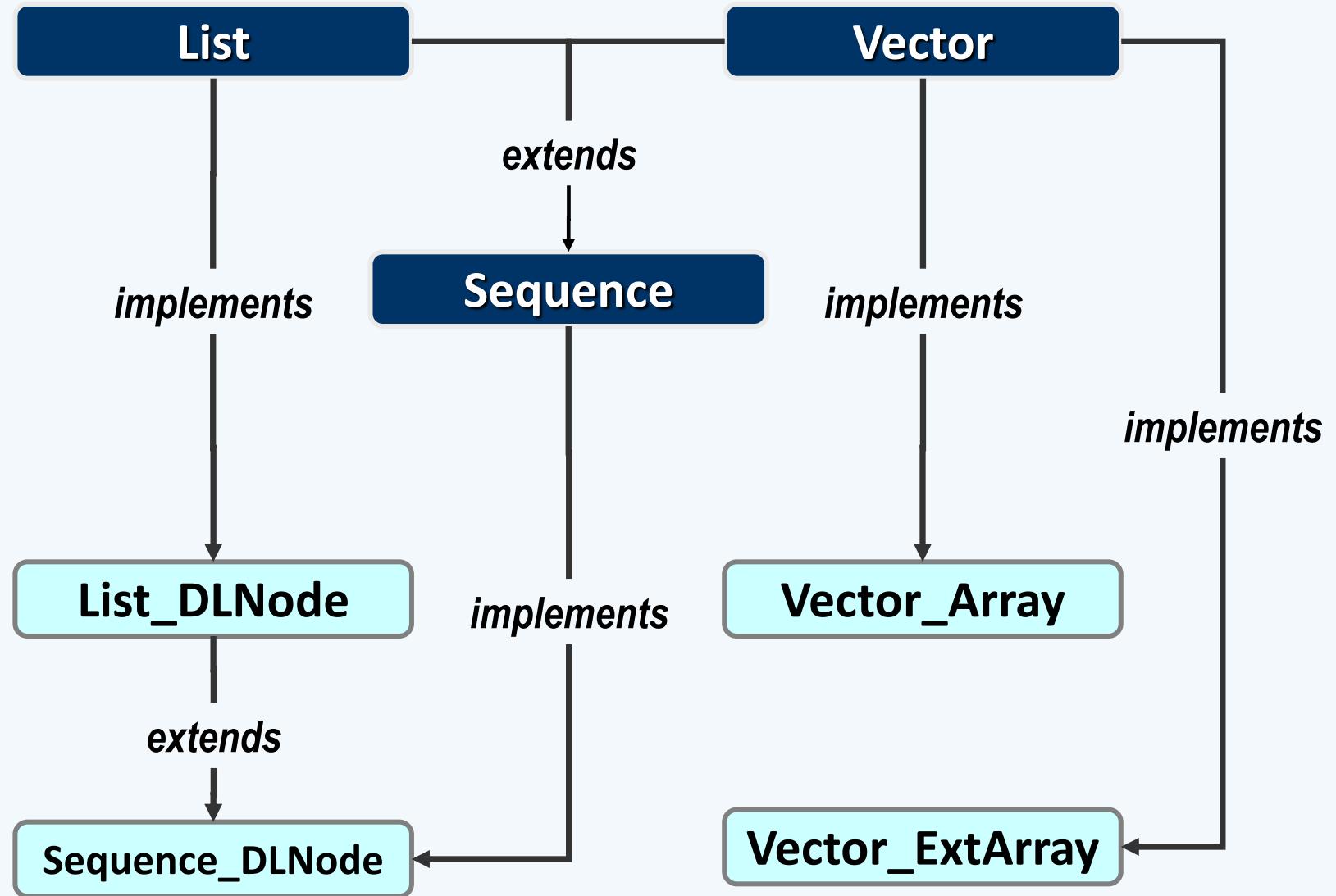
```
class List_DLNode
implements List
{ /* ... */ }
```

❖ //序列, Sequence.java

```
interface Sequence
extends Vector, List
{ /* ... */ }
```

//Sequence\_DLNode.java

```
class Sequence_DLNode
extends List_DLNode
implements Sequence
{ /* ... */ }
```



### 3. 列表

(xc) Python列表

邓俊辉

deng@tsinghua.edu.cn

## Python List

❖ 在Python中，List属于内置的标准数据类型

❖ `box = [ 'pencil', 'pen', 'ruler', 'rubber' ]; print box`

```
# ['pencil', 'pen', 'ruler', 'rubber']
```

❖ `for item in box: print item,`

```
# pencil pen ruler rubber
```

❖ `box.reverse()`

```
for item in box: print item,
```

```
# rubber ruler pen pencil
```

❖ `box.sort()`

```
for item in box: print item,
```

```
# pen pencil rubber ruler
```

## Python List

```
❖ for i in range(0, len(box)): # [0, n)
    print box[i],
    # pen pencil rubber ruler

❖ for i in range(len(box)-1, -1, -1): # [n-1, -1)
    print box[i],
    # ruler rubber pencil pen

❖ for i in range(-1, -len(box)-1, -1): # [-1, -n-1)
    print box[i],
    # ruler rubber pencil pen
```

## Python List

```
❖ bag = [ 'data structures', 'calculus', box, 2012012012 ]  
      print bag  
  
      # ['data structures', 'calculus',  
      # ['pen', 'pencil', 'rubber', 'ruler'], 2012012012]  
  
❖ for item in bag: print item,  
      # data structures calculus  
      # ['pen', 'pencil', 'rubber', 'ruler'] 2012012012  
  
❖ for item in bag[2]: print item,  
      # pen pencil rubber ruler  
  
❖ for item in bag[2][1:3]: print item,  
      # pencil rubber
```

## reverse()

❖ def reverse\_1(L): # 循位置访问？

```
for i in range(0, len(L)): # 对[0, n)内的每个i，依次  
    L.insert(i, L.pop()) # 将末元素转移至位置i  
  
return L # 最终即得倒置后的列表
```

❖ def reverse\_2(L): # 循秩访问？

```
i, j = 0, len(L) - 1 # 从首、末元素开始  
  
while i < j: # 依次令对称的L[i]及L[n-1-i]  
    L[i], L[j] = L[j], L[i] # 互换，然后  
    i, j = i + 1, j - 1 # 考查下一对元素  
  
return L # 最终即得倒置后的列表
```

❖ 哪个版本效率更高？实测结果如何解释？

## 4. 栈与队列

### 栈接口与实现

陛下用群臣，如积薪耳，后来者居上。

邓俊辉

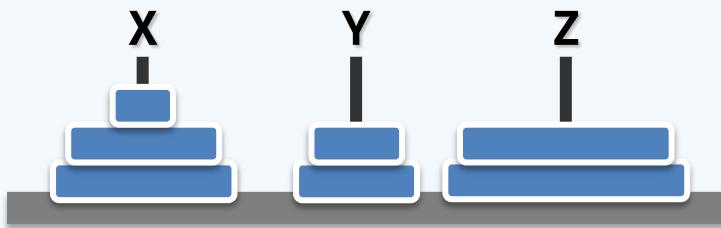
deng@tsinghua.edu.cn

## 操作与接口

❖ 栈 ( stack ) 是受限的序列

只能在栈顶 ( top ) 插入和删除

栈底 ( bottom ) 为盲端



❖ 基本接口

`size()` / `empty()`

`push()` 入栈

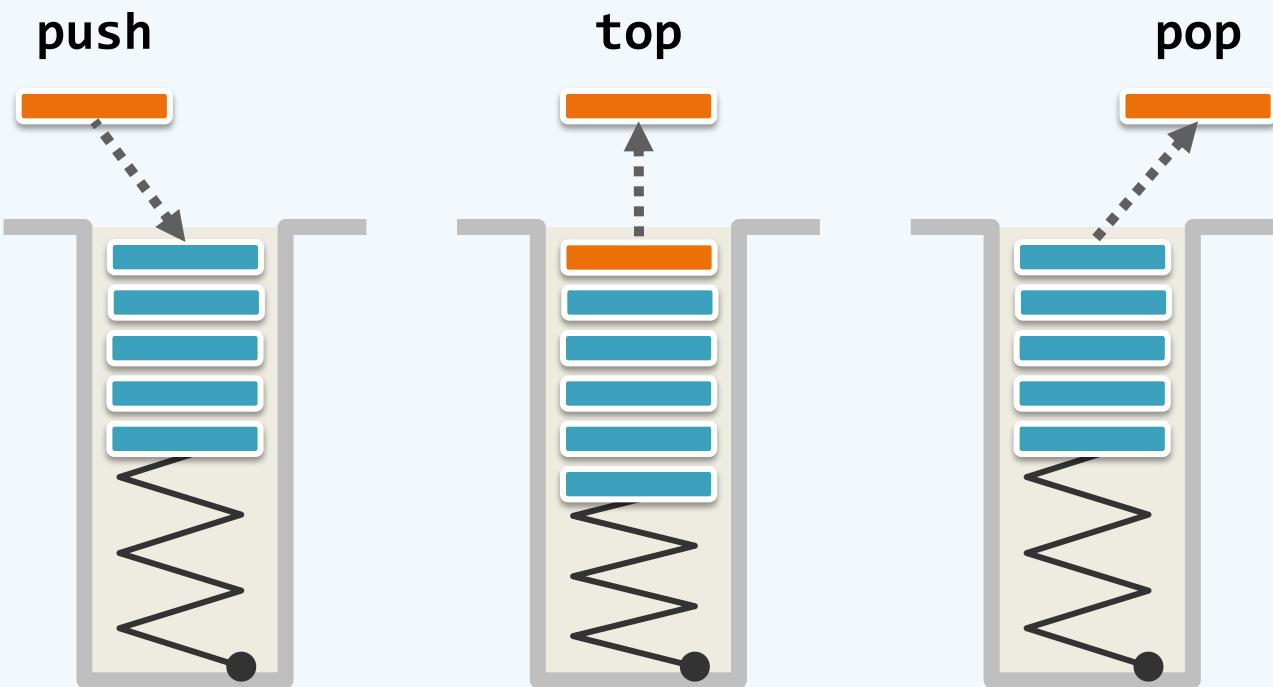
`pop()` 出栈

`top()` 查顶

❖ 后进先出 ( LIFO )

先进后出 ( FILO )

❖ 扩展接口 : `getMax()` ...



## 操作实例

操作	输出	栈 (左侧栈顶)
<code>Stack()</code>		
<code>empty()</code>	<code>true</code>	
<code>push(5)</code>		5
<code>push(3)</code>		3 5
<code>pop()</code>	3	5
<code>push(7)</code>		7 5
<code>push(3)</code>		3 7 5
<code>top()</code>	3	3 7 5
<code>empty()</code>	<code>false</code>	3 7 5

操作	输出	栈 (左侧栈顶)
<code>push(11)</code>		11 3 7 5
<code>size()</code>	4	11 3 7 5
<code>push(6)</code>		6 11 3 7 5
<code>empty()</code>	<code>false</code>	6 11 3 7 5
<code>push(7)</code>		7 6 11 3 7 5
<code>pop()</code>	7	6 11 3 7 5
<code>pop()</code>	6	11 3 7 5
<code>top()</code>	11	11 3 7 5
<code>size()</code>	4	11 3 7 5

## 实现

❖ 栈既然属于序列的特例，故可直接基于向量或列表派生

❖ `template <typename T> class Stack: public Vector<T> { //由向量派生的栈模板类  
public: //size()、empty()以及其它开放接口均可直接沿用`

`void push( T const & e ) { insert( size(), e ); } //入栈`

`T pop() { return remove( size() - 1 ); } //出栈`

`T & top() { return (*this)[ size() - 1 ]; } //取顶`

`}; //以向量首/末端为栈底/顶——颠倒过来呢？`

❖ 确认：如此实现的栈各接口，均只需 $\mathcal{O}(1)$ 时间

❖ 课后：基于列表，派生定义栈模板类

评测：你所实现的栈接口，效率如何？

## 4. 栈与队列

### 调用栈：原理与算法

Yessiree. We do not doubt his word,  
an stack ourselfs into the bus like flapjacks.

邓俊辉

deng@tsinghua.edu.cn

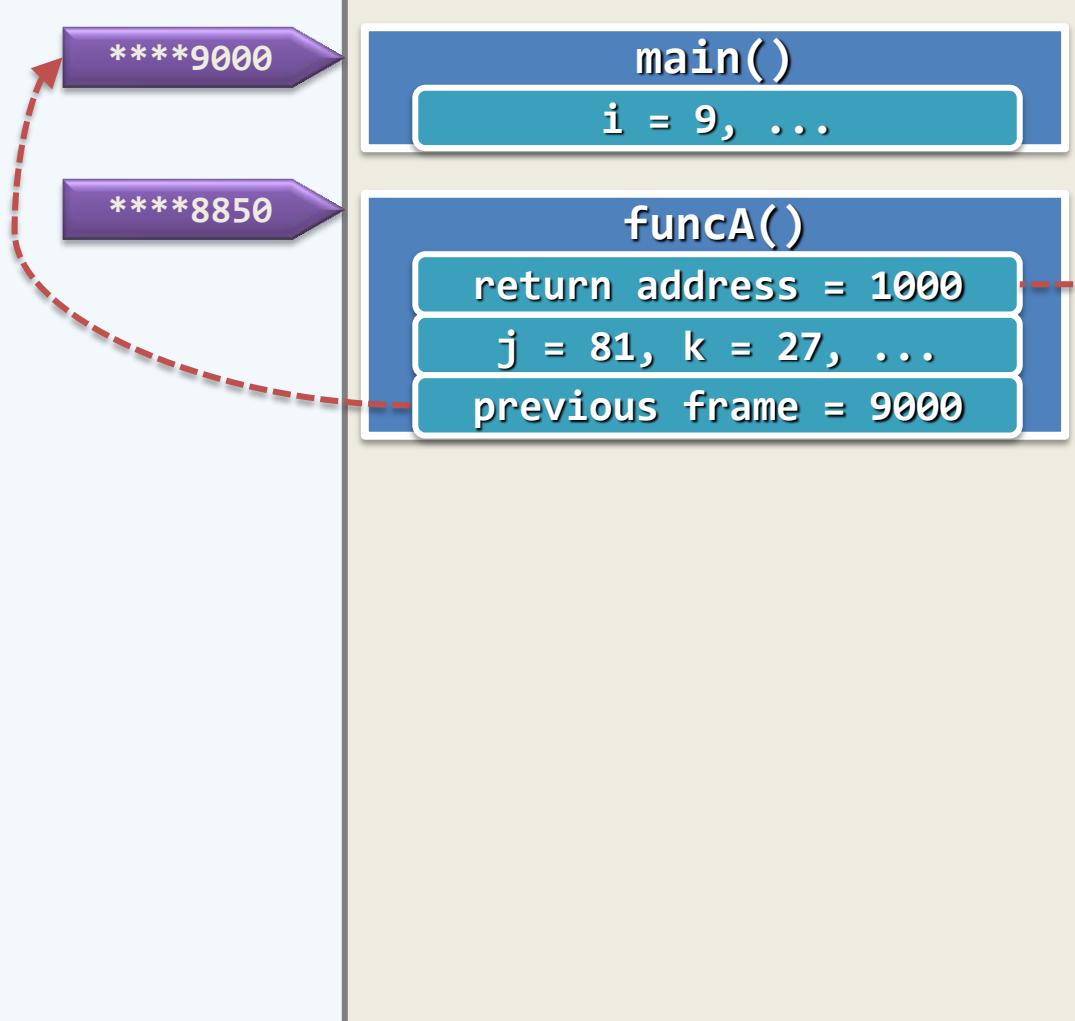
**call stack**

\*\*\*\*9000

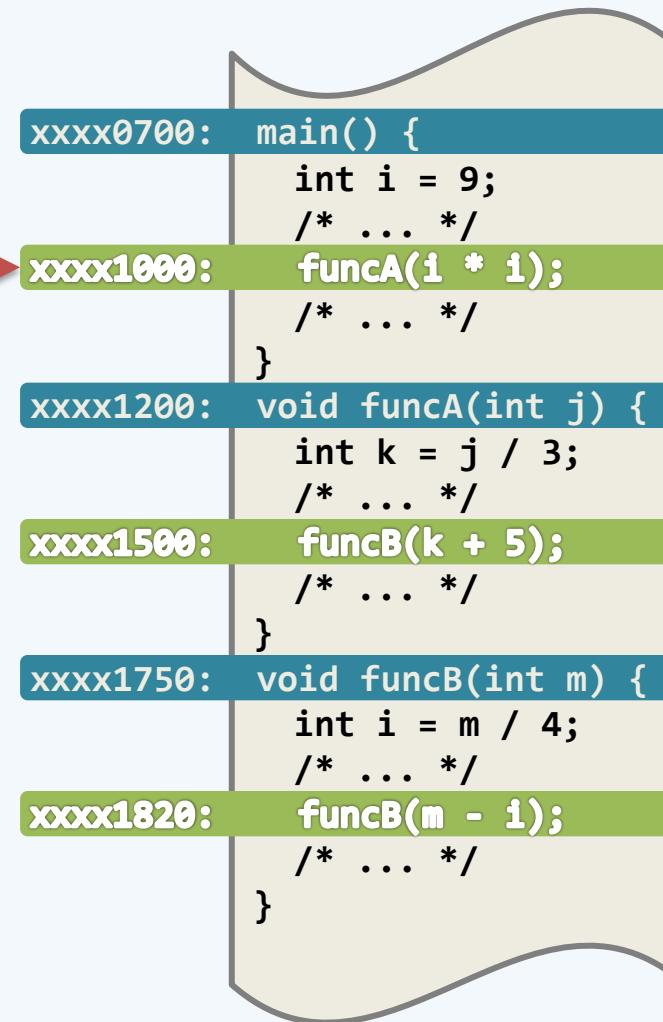
**main()****i = 9, ...****binary executable**

```
xxxx0700: main() {  
    int i = 9;  
    /* ... */  
    xxxx1000: funcA(i * i);  
    /* ... */  
}  
xxxx1200: void funcA(int j) {  
    int k = j / 3;  
    /* ... */  
    xxxx1500: funcB(k + 5);  
    /* ... */  
}  
xxxx1750: void funcB(int m) {  
    int i = m / 4;  
    /* ... */  
    xxxx1820: funcB(m - i);  
    /* ... */  
}
```

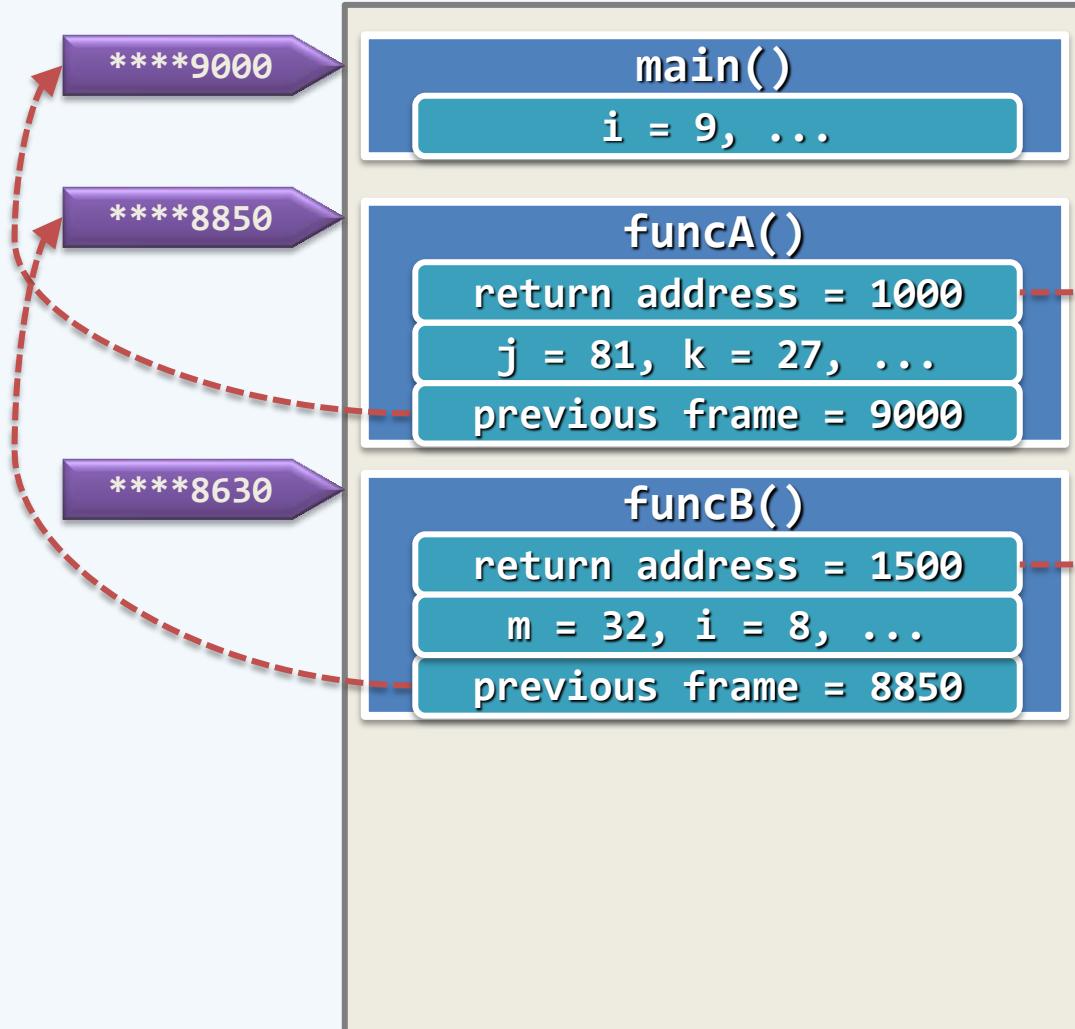
## call stack



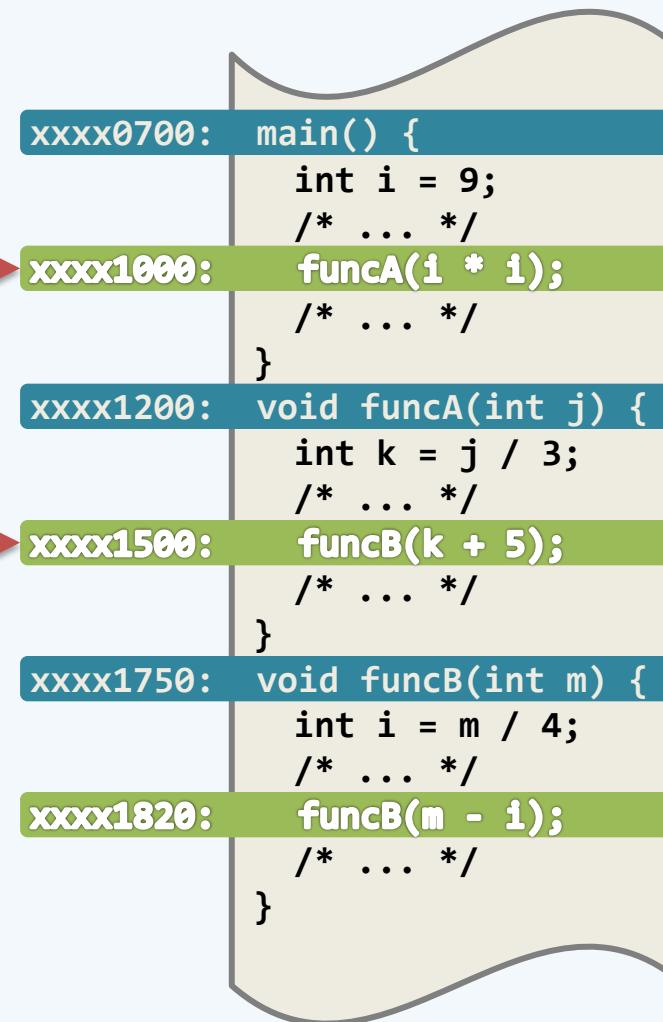
## binary executable



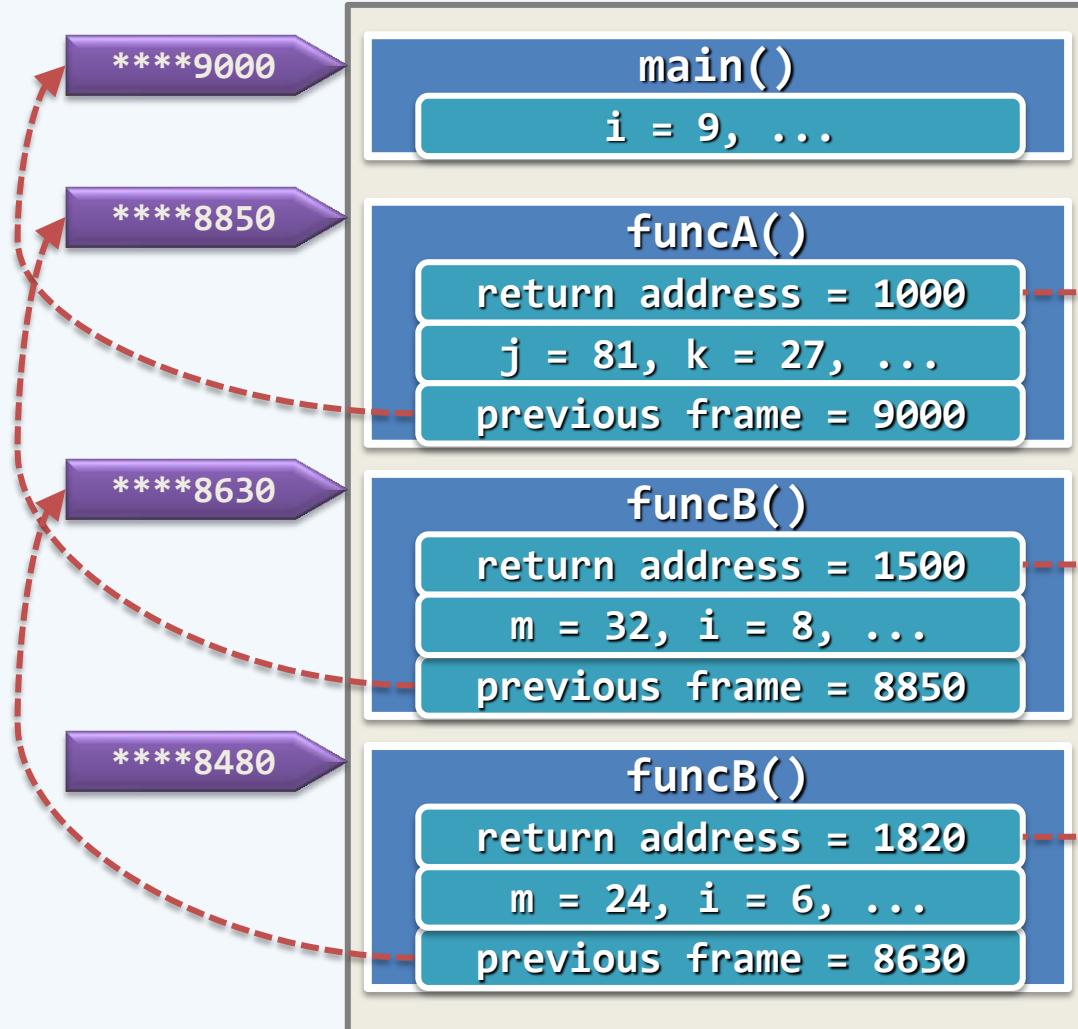
## call stack



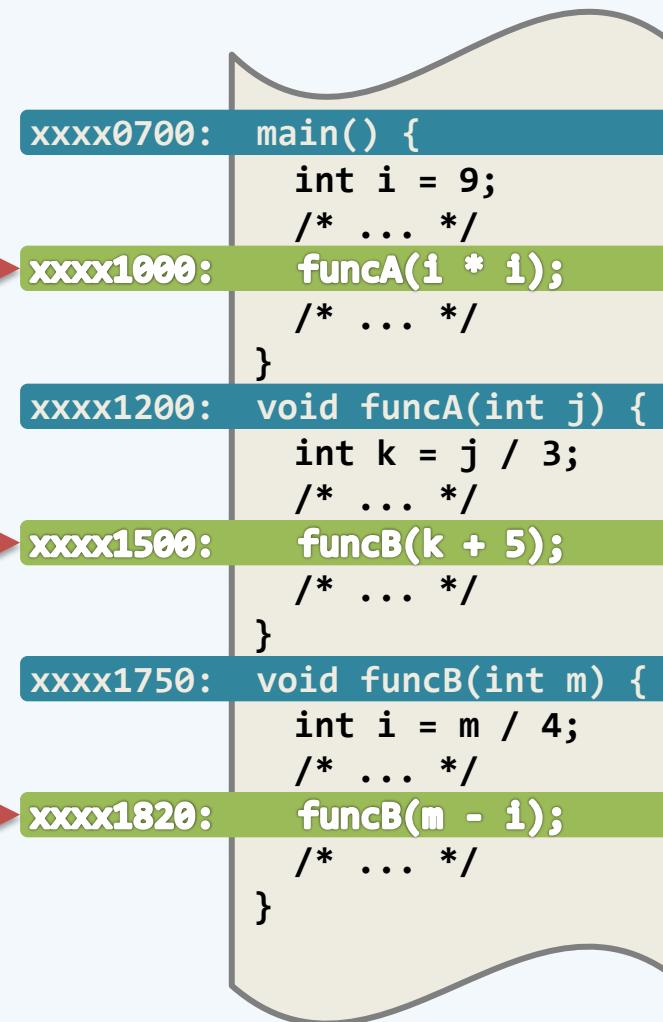
## binary executable



## call stack



## binary executable



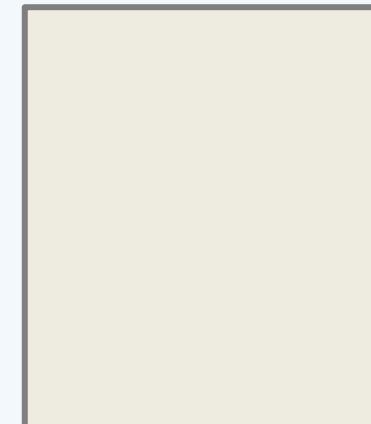
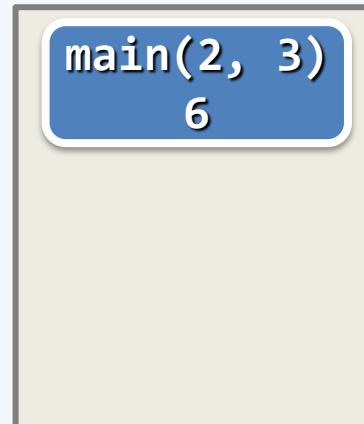
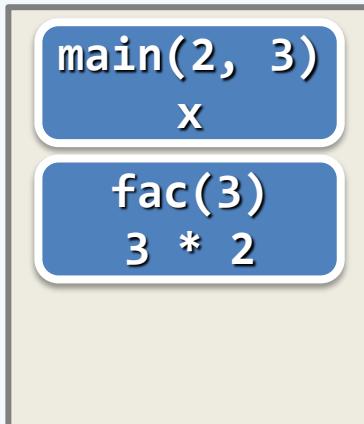
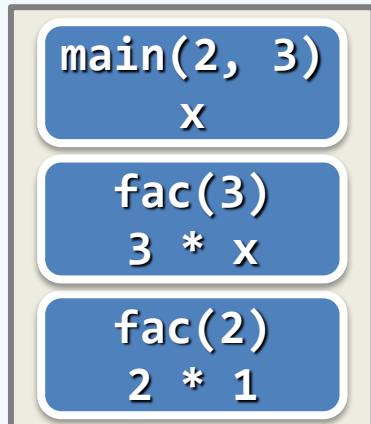
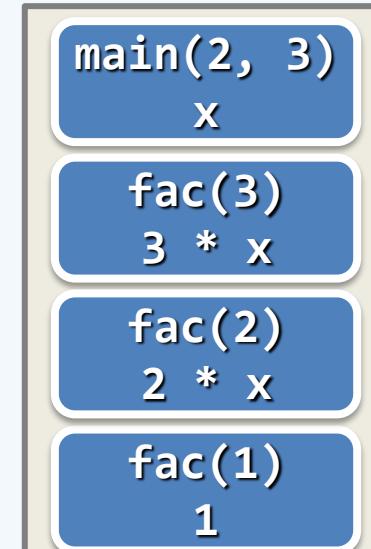
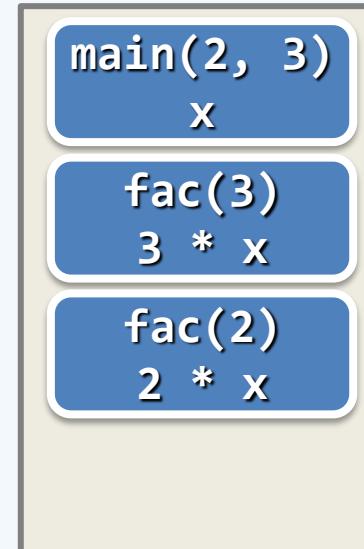
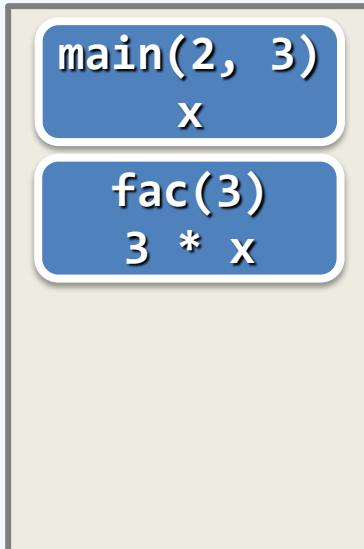
## 4. 栈与队列

调用栈：实例

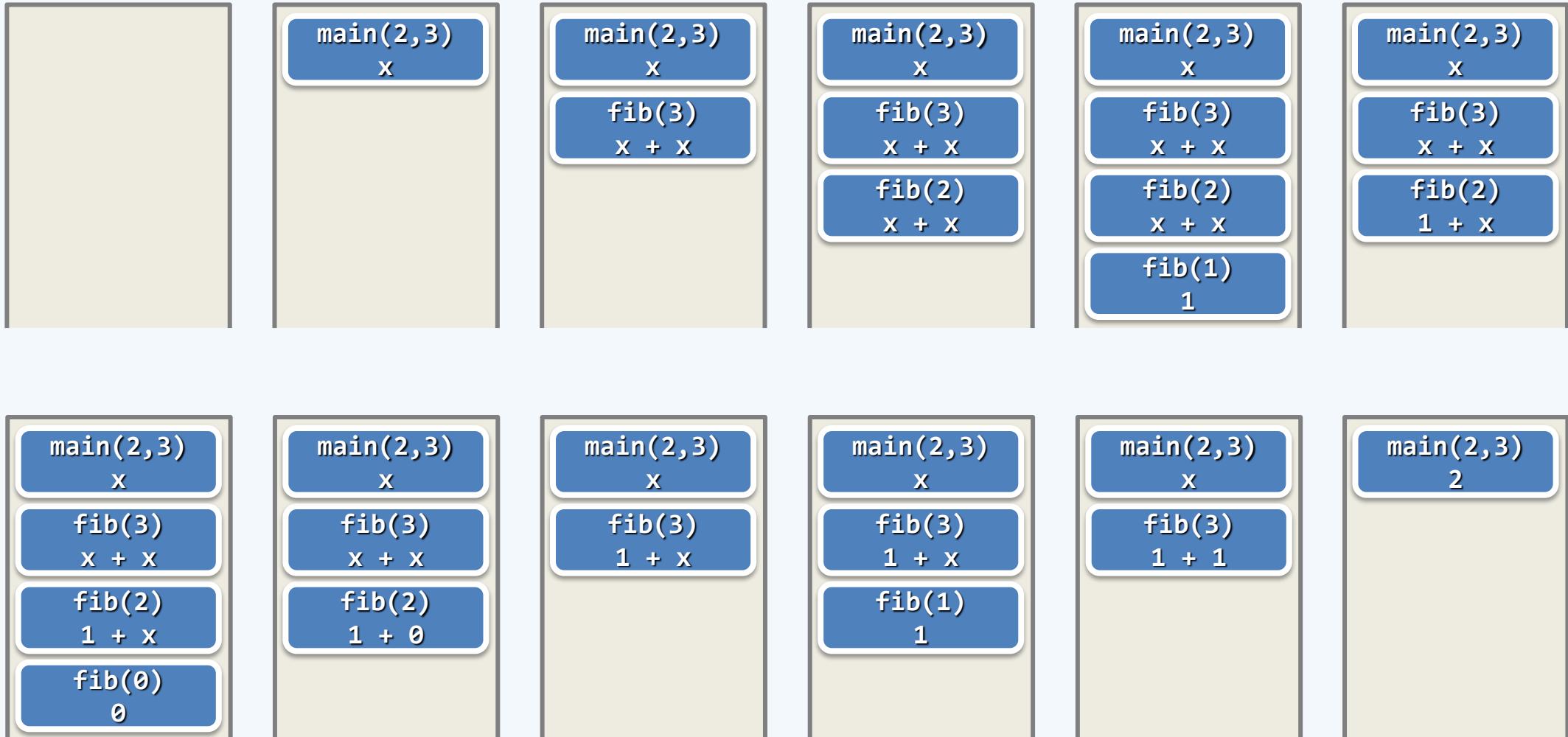
邓俊辉

deng@tsinghua.edu.cn

```
int fac(int n) { return (n < 1) ? 1 : n * fac(n - 1); }
```



```
int fib( int n ) { return (n < 2) ? n : fib(n - 1) + fib(n - 2); }
```



```

❖ hailstone(int n) {
    if ( 2 > n ) return;
    n % 2 ? odd( n ): even( n );
}

❖ even( int n ) { hailstone( n / 2 ); }

odd( int n ) { hailstone( 3*n + 1 ); }

❖ main(int argc, char* argv[])
{ hailstone( atoi( argv[1] ) ); }

```

**call stack**

main(2, 10)

hailstone(10)

even(10)

hailstone(5)

odd(5)

hailstone(16)

even(16)

hailstone(8)

even(8)

hailstone(4)

even(4)

hailstone(2)

even(2)

hailstone(1)

**call stack**

main(2, 27)

hailstone(27)

odd(27)

hailstone(82)

even(82)

hailstone(41)

odd(41)

hailstone(124)

even(124)

hailstone(62)

even(62)

hailstone(31)

odd(31)

hailstone(94)

... ...

## 4. 栈与队列

调用栈：消除递归

邓俊辉

deng@tsinghua.edu.cn

❖ 动机：递归函数的空间复杂度，主要取决于**最大递归深度**，而非**递归实例总数**

为**隐式地**维护调用栈，需花费额外的处理时间

❖ 方法：**显式地**维护调用栈，将递归算法改写为迭代版本...

```
❖ int fac( int n ) {  
    int f = 1; //  $\mathcal{O}(1)$ 空间  
    while ( n > 1 )  
        f *= n--;  
    return f;  
}
```

- ❖ `int fib( int n ) {`
- `int f = 0, g = 1; //O(1)空间`
- `while ( 0 < n-- )`
- `{ g += f; f = g - f; }`
- `return f;`
- `}`
- ❖ `void hailstone( int n ) { //O(1)空间`
- `while ( 1 < n )`
- `n = n % 2 ? 3*n + 1 : n/2;`
- `}`

## 4. 栈与队列

### 进制转换

Hickory, Dickory, Dock

The mouse ran up the clock

- Nursery Rhyme Medley

邓俊辉

deng@tsinghua.edu.cn

## 典型应用场景

### 逆序 输出

- conversion
- 输出次序与处理过程颠倒；递归深度和输出长度不易预知

### 延迟 缓冲

- evaluation
- 线性扫描算法模式中，在预读足够长之后，方能确定可处理的前缀

### 递归 嵌套

- stack permutation + parenthesis
- 具有自相似性的问题可递归描述，但分支位置和嵌套深度不固定

### 栈式 计算

- RPN
- 基于栈结构的特定计算模式

## 进制转换

❖ 描述：给定任一10进制非负整数，将其转换为λ进制表示形式

$$12345_{(10)} = 30071_{(8)}$$

```
printf("%d | %I64d | %b | %o | %x", n);
```

❖ 巴比伦楔形文字 ( Babylonian cuneiform ) 中的60进制...

❖ 正方形的对角线

$$1^{\circ}24'51''10'''$$

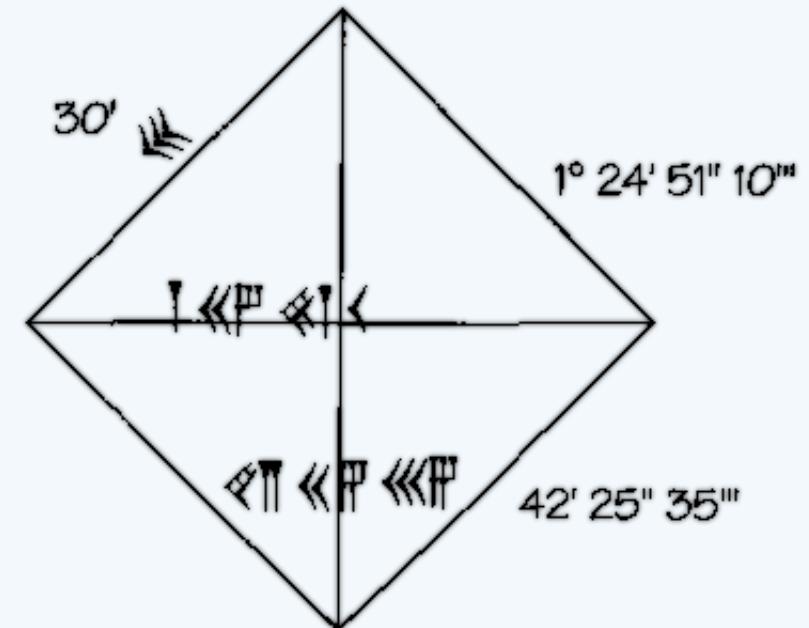
$$= 1 + 24/60 + 51/60^2 + 10/60^3$$

$$= 1.41421296\cancel{2}96296296\dots$$

❖ 误差

$$|1^{\circ}24'51''10''' - \sqrt{2}| < 0.000,000,6 = 0.6 \times 10^{-6}$$

即便边长为1km，误差亦不足1mm



## 思路

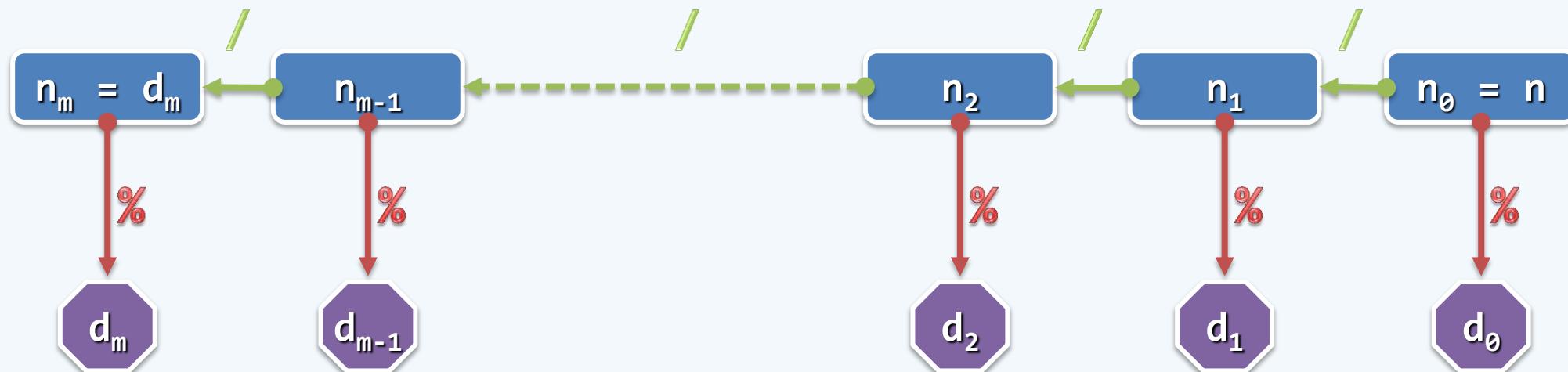
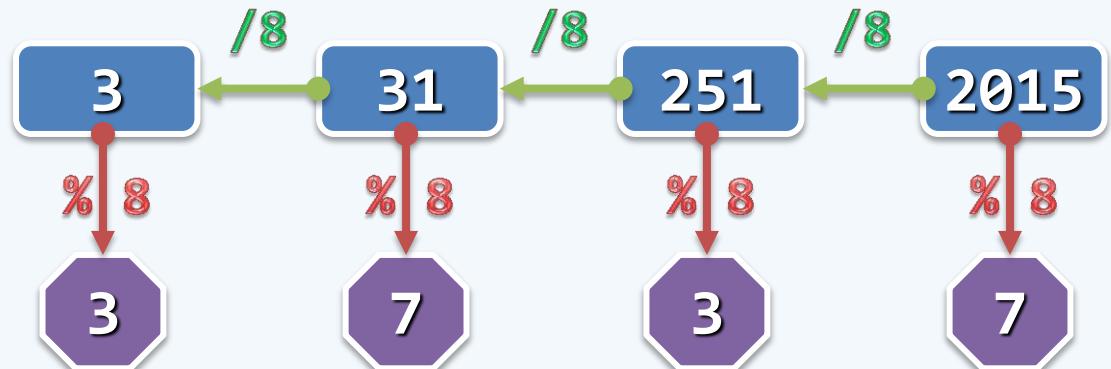
设： $n = (d_m \dots d_2 d_1 d_0)_{\lambda} = d_m \times \lambda^m + \dots + d_1 \times \lambda^1 + d_0 \times \lambda^0$

令： $n_i = (d_m \dots d_{i+1} d_i)_{\lambda}$

则有： $n_{i+1} = n_i / \lambda$  和  $d_i = n_i \% \lambda$

构思：n对 $\lambda$ 反复取模、整除

即可自低到高得出 $\lambda$ 进制的各位

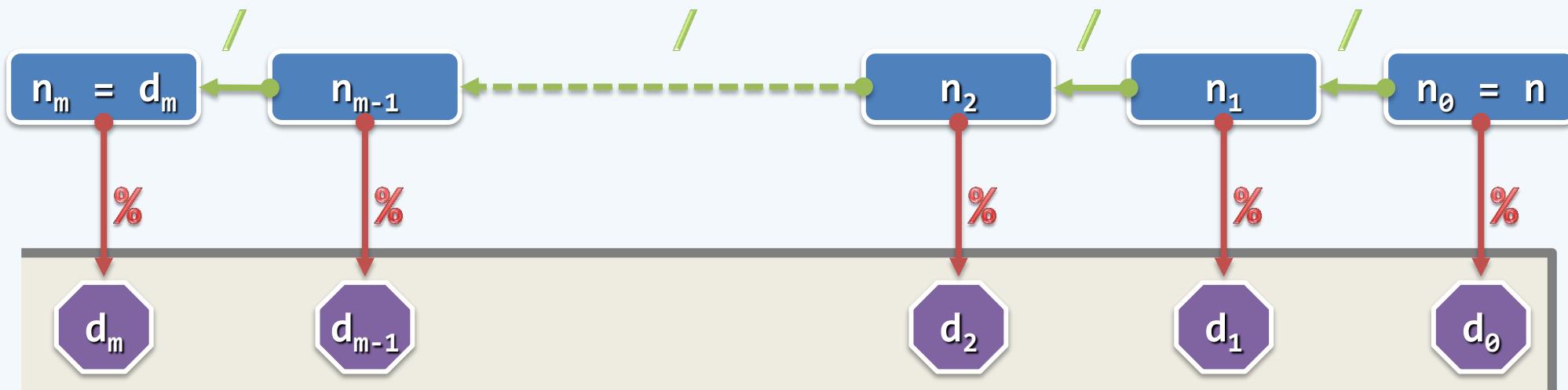


## 难点及解决方法

❖ 位数 $m$ 并不确定，如何正确记录并输出转换结果？具体地  
如何支持足够大的 $m$ ，同时空间也不浪费？

**自低到高**得到的数位，如何**自高到低**输出？

❖ 若使用**向量**，则扩容策略必须得当；若使用**列表**，则多数接口均被闲置  
❖ 使用栈，既可满足以上要求，亦可有效控制计算成本



## 算法实现

```
❖ void convert( Stack<char> & S, __int64 n, int base ) {  
    static char digit[] = //新进制下的数位符号，可视base取值范围适当扩充  
    { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };  
    while ( n > 0 ) { //由低到高，逐一计算出新进制下的各数位  
        S.push( digit[ n % base ] ); //余数（当前的数位）入栈  
        n /= base; //n更新为其对base的除商  
    }  
}  
  
❖ main() {  
    Stack<char> S; convert( S, n, base ); //用栈记录转换得到的各数位  
    while ( ! S.empty() ) printf( "%c", S.pop() ); //逆序输出  
}
```

## 4. 栈与队列

括号匹配

邓俊辉

deng@tsinghua.edu.cn

## 实例

❖ `( a [ i - 1 ] [ j + 1 ] ) + a [ i + 1 ] [ j - 1 ] ) * 2` //失配

`( a [ i - 1 ] [ j + 1 ] + a [ i + 1 ] [ j - 1 ] ) * 2` //匹配

❖ 观察：除了各种括号，其余符号均可暂时忽略

`( [ ] [ ] ) [ ] [ ] )` //失配

`( [ ] [ ] [ ] [ ] )` //匹配

❖ 从简单入手，先来考查只有一种括号的情况...

## 尝试

0) 平凡：无括号的表达式是匹配的

1) 减治？  $E$  匹配，仅当  $(E)$  匹配

2) 分治？  $E$  和  $F$  均匹配，仅当  $E F$  匹配

❖ 然而，根据以上性质，却不易直接应用已知的策略

❖ 究其根源在于，1) 和2) 均为必要性，比如反例：

$$( ( ) ( ) ) ( ) = ( ( ) ( ) ( ) ) ( )$$

$$( ( ) ( ) ) ( ) = ( ( ) ( ) ) ( ( ) )$$

❖ 而为使问题有效简化，必须发现并借助充分性

## 构思

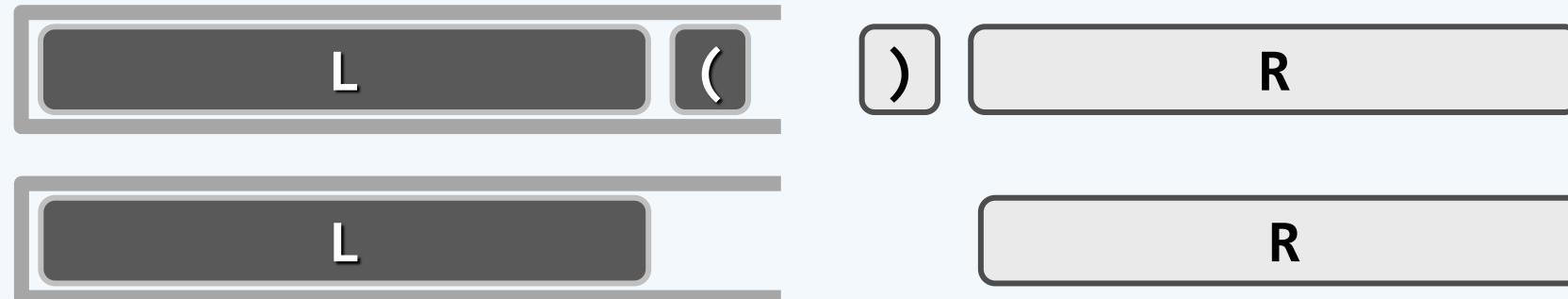
❖ 颠倒以上思路：消去一对**紧邻**的左右括号，不影响全局的匹配判断

亦即：**L ( R** 匹配，仅当 **L R** 匹配

❖ 那么，如何**找到**这对括号？再者，如何使问题的这种简化得以**持续**进行？

❖ 顺序扫描表达式，用栈记录已扫描的部分 //实际上只需记录左括号

反复迭代：凡遇 **(**，则进栈；凡遇 **)**，则出栈



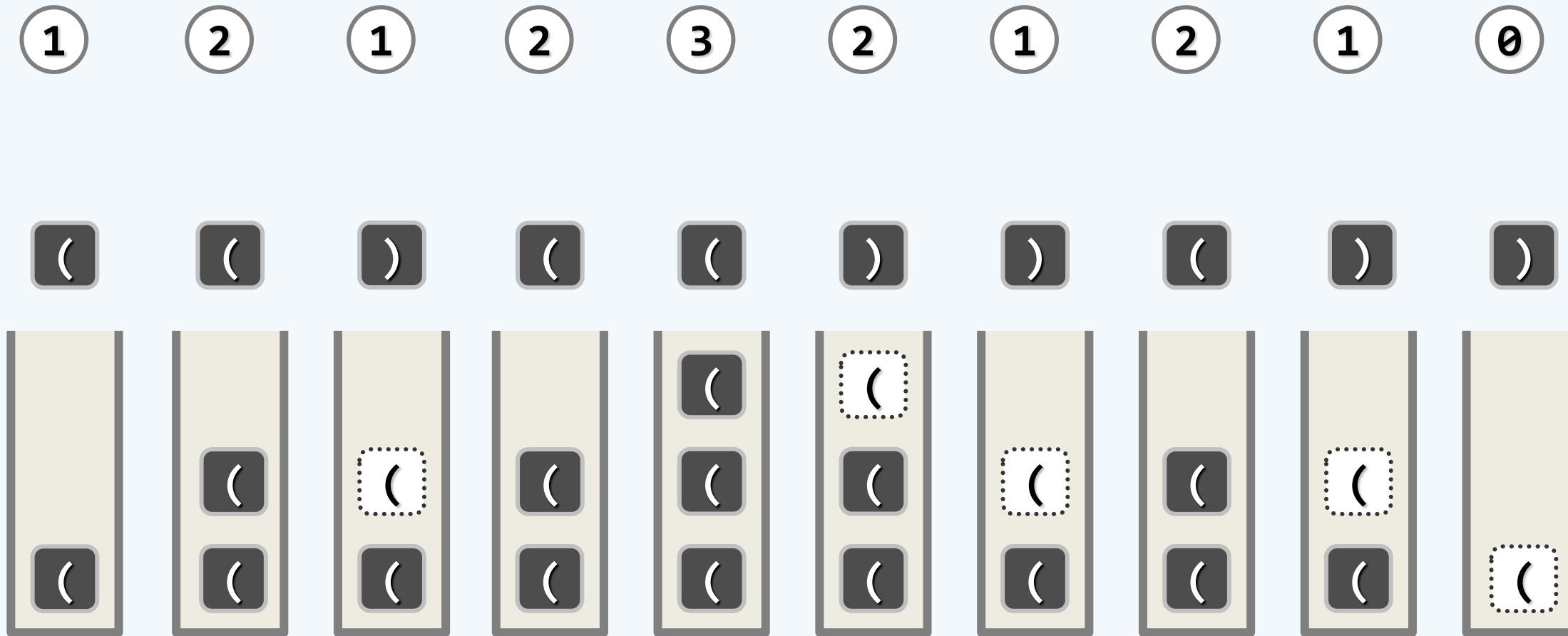
## 实现

```
❖ bool paren( const char exp[], int lo, int hi ) { //exp[lo, hi)  
    Stack<char> S; //使用栈记录已发现但尚未匹配的左括号  
  
    for ( int i = lo; i < hi; i++ ) //逐一检查当前字符  
  
        if ( '(' == exp[i] ) S.push( exp[i] ); //遇左括号：则进栈  
  
        else if ( ! S.empty() ) S.pop(); //遇右括号：若栈非空，则弹出左括号  
  
        else return false; //否则（遇右括号时栈已空），必不匹配  
  
    return S.empty(); //最终，栈空当且仅当匹配  
}
```

## 实例

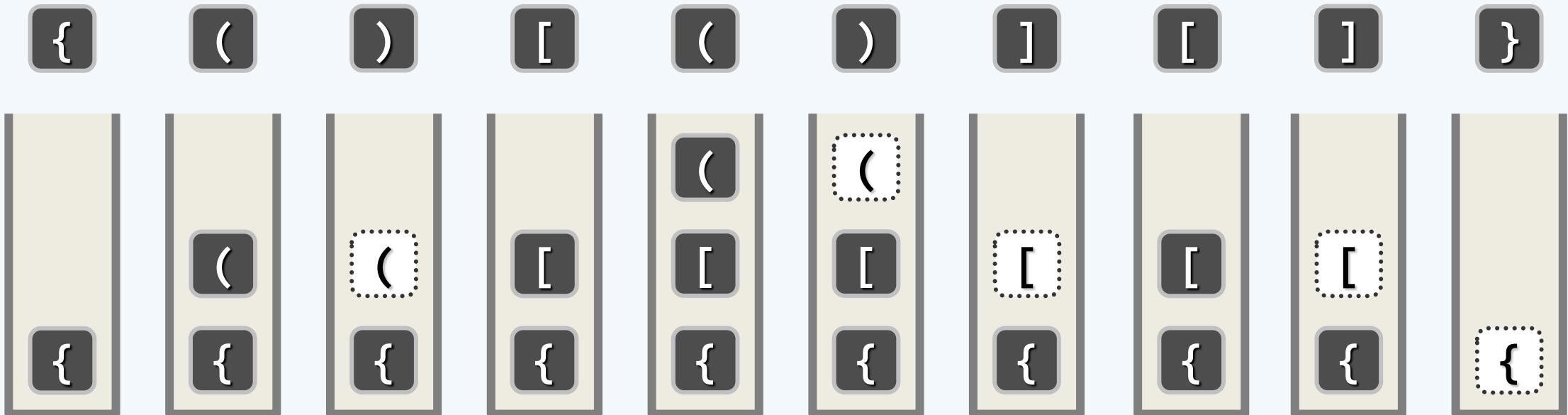
❖ 实际上，若仅考虑一种括号，只需一个计数器足矣：始终非负；最后归零

//`s.size()`



## 拓展

- ❖ 以上思路及算法，可便捷地推广至多种括号并存的情况
- ❖ 可否，使用多个计数器？不行，反例：`[ ( ] )`
- ❖ 甚至，只需约定“括号”的通用格式，而不必事先固定括号的类型与数目  
比如：`<body>|</body>`, `<h1>|</h1>`, `<font>|</font>`, `<p>|</p>`, `<ol>|</ol>`, ...



## 拓展

❖ 按字典序，枚举由 $n$ 对匹配括号组成的所有表达式

ACP-v4-f4-p5, Algorithm P, I. Sembra, 1981

❖ 在由 $n$ 对匹配括号组成的所有表达式中，按字典序取出第 $N$ 个

ACP-v4-f4-p14, Algorithm U, F. Ruskey, 1978

❖ 在由 $n$ 对匹配括号组成的所有表达式中，等概率地随机任选其一

ACP-v4-f4-p15, Algorithm W, D. B. Arnold & M. R. Sleep, 1980

❖ 由算法U，不是可以直接实现算法W的功能吗？后者的意义何在？

## 4. 栈与队列

栈混洗

邓俊辉

deng@tsinghua.edu.cn

## 栈混洗

❖ 考查栈  $A = \langle a_1, a_2, \dots, a_n \rangle$  、  $B = S = \emptyset$  //左端为栈顶

❖ 只允许 将  $A$  的顶元素弹出并压入  $S$ ，或  
将  $S$  的顶元素弹出并压入  $B$

//  $S.push(A.pop())$   
//  $B.push(S.pop())$

❖ 若经过一系列以上操作后， $A$  中元素全部转入  $B$  中

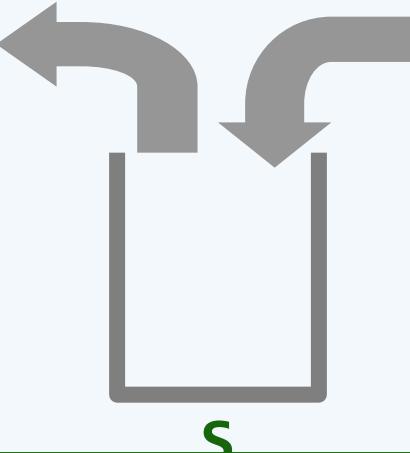
$B = [a_{k1}, \dots, a_{kn}]$  //右端为栈顶

则称之为  $A$  的一个栈混洗 (stack permutation)

$B = [a_{k1}, \dots, a_{kn}]$

1 2 3 4

$\langle a_1, a_2, \dots, a_n \rangle = A$



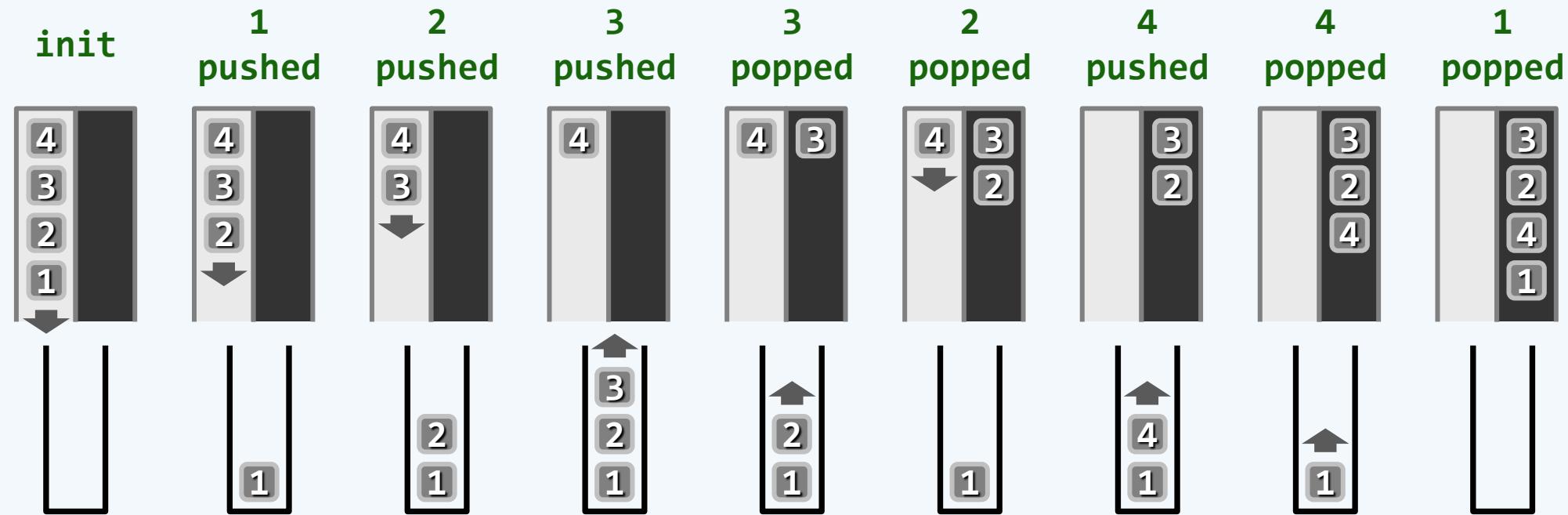
## 计数

❖ 同一输入序列，可有多种栈混洗

[ 1, 2, 3, 4 >, [ 4, 3, 2, 1 >, [ 3, 2, 4, 1 >, ...

❖ 长度为n的序列，可能的混洗总数  $SP(n) = ?$

//显然， $SP(n) \leq n!$



## 计数

❖  $SP(1) = 1$

❖ 设栈s在第 $k$ 次 $pop()$ 之后首次重新变空

则 $k$ 无非 $n$ 种情况：

$$SP(n) = \sum_{k=1}^n SP(k-1) \cdot SP(n-k)$$

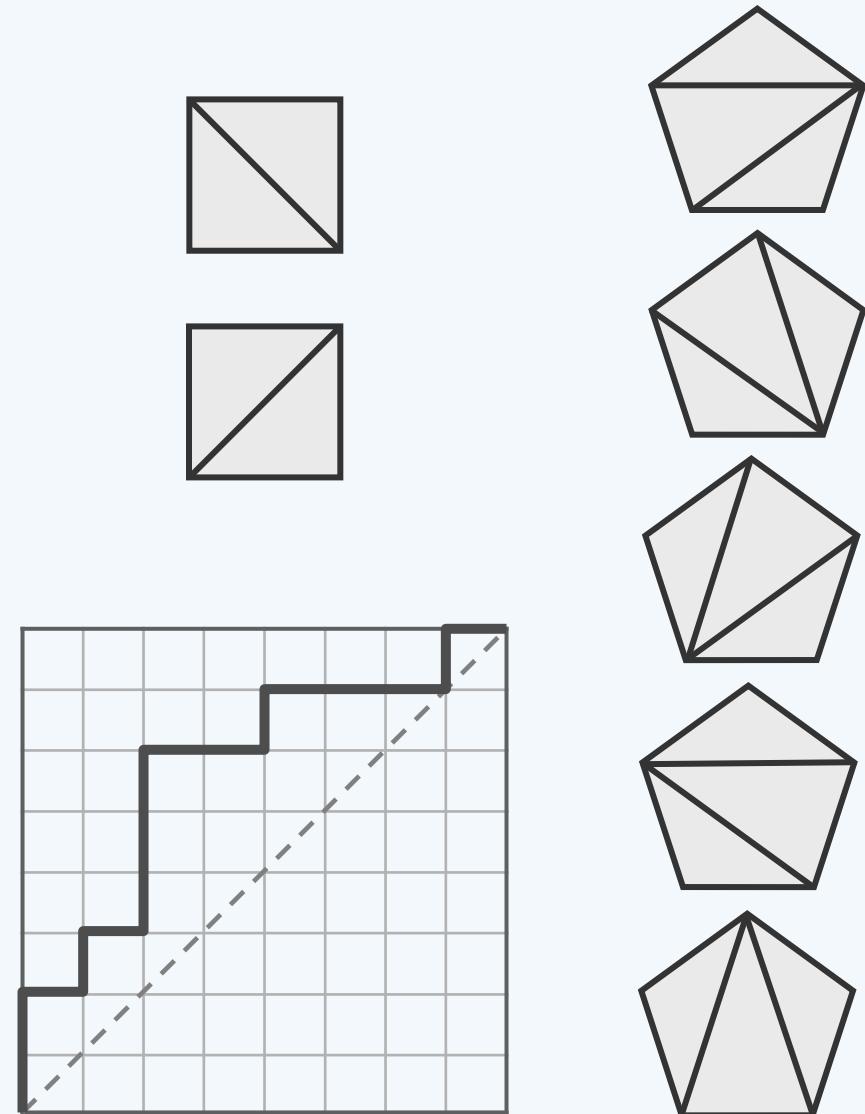
$$= \text{Catalan}(n) = (2n)! / (n+1)! / n!$$

❖  $SP(2) = 4! / 3! / 2! = 2$

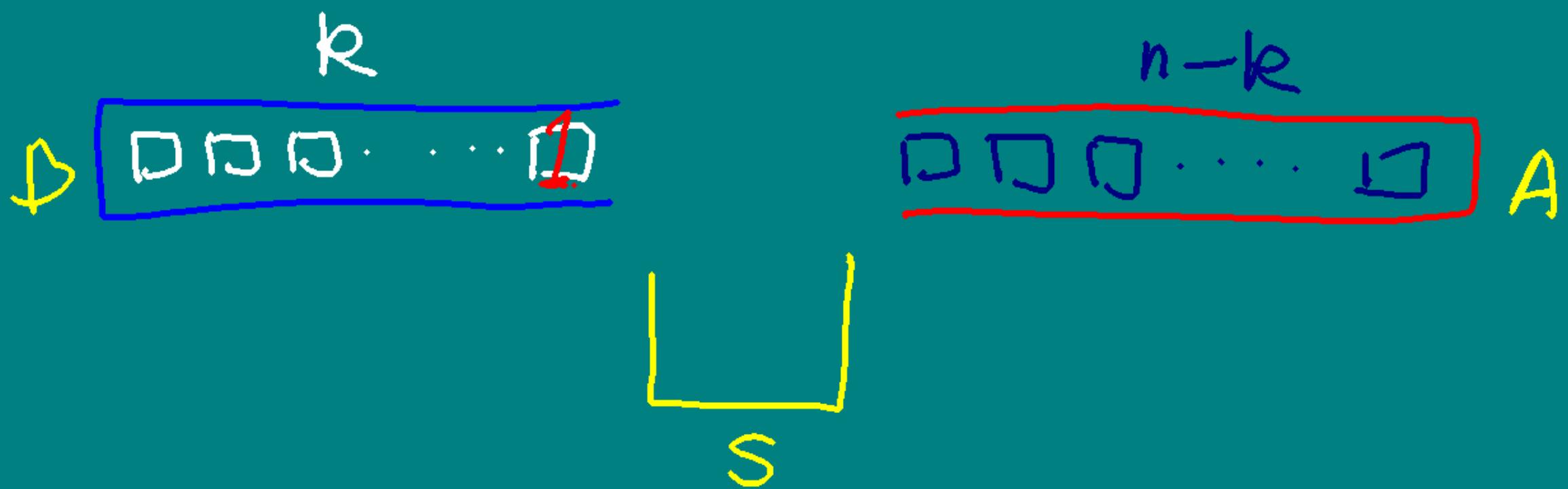
$SP(3) = 6! / 4! / 3! = 5$

... ... ...

$SP(6) = 12! / 7! / 6! = 132$



$$SP(n) = \sum_{k=1}^n SP(k-1) \cdot SP(n-k)$$



## 甄别

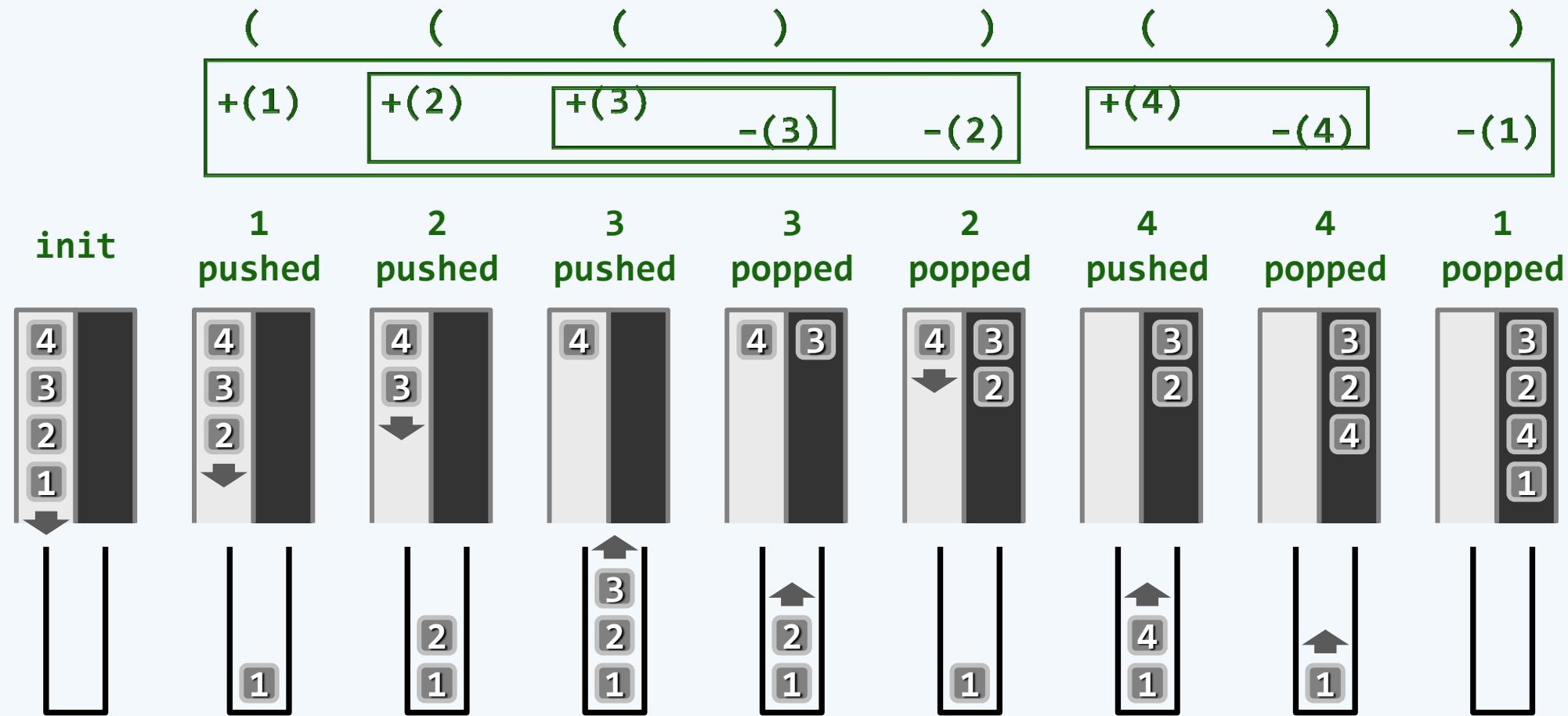
- ❖ 输入序列  $< 1, 2, 3, \dots, n >$  的任一排列  $[ p_1, p_2, p_3, \dots, p_n ]$  是否为栈混洗？
- ❖ 简单情况： $< 1, 2, 3 >$ ,  $n = 3$ 
  - 栈混洗共  $6! / 4! / 3! = 5$  种
  - 全排列共  $3! = 6$  种 //少了一种...
- ❖  $[ 3, 1, 2 ]$  //为什么是它？
- ❖ 观察：任意三个元素能否按某相对次序出现于混洗中，与其它元素无关 //故可推而广之...
- ❖ 对于任何  $1 \leq i < j < k \leq n$ 
 $[ \dots, \boxed{k}, \dots, \boxed{i}, \dots, \boxed{j}, \dots ]$  必非栈混洗
- ❖ 反过来，不存在 “ $\boxed{3}\boxed{1}\boxed{2}$ ” 模式的序列，一定是栈混洗吗？

## 甄别

- ❖ 充要性： A permutation is a stack permutation iff  
( Knuth, 1968 ) it does NOT involve the permutation 312 //习题[4-3]
- ❖ 如此，可得一个  $\mathcal{O}(n^3)$  的甄别算法 //进一步地...
- ❖  $[ p_1, p_2, p_3, \dots, p_n ]$  是  $[ 1, 2, 3, \dots, n ]$  的栈混洗， 当且仅当  
对于任意  $i < j$ ，不含模式  $[ \dots, [j + 1], \dots, [i], \dots, [j], \dots ]$
- ❖ 如此，可得一个  $\mathcal{O}(n^2)$  的甄别算法 //再进一步地...
- ❖  $\mathcal{O}(n)$  算法：直接借助栈A、B和S，模拟混洗过程 //为何可行？  
每次  $S.pop()$  之前，检测 S 是否已空；或需弹出的元素在 S 中，却非顶元素

## 括号匹配

◆ 观察：每一栈混洗，都对应于栈S的n次push与n次pop操作构成的某一序列



◆ n个元素的栈混洗，等价于n对括号的匹配

## 4. 栈与队列

中缀表达式求值

问题

邓俊辉

知实而不知名，知名而不知实，皆不知也

deng@tsinghua.edu.cn

## 表达式求值

❖ 给定语法正确的算术表达式 $S$ ，计算与之对应的数值

❖ \$ echo \$(( 0 + ( 1 + 23 ) / 4 \* 5 \* 67 - 8 + 9 ))

❖ \> set /a (!0 ^<^< ( 1 - 2 + 3 \* 4 ) ) - 5 \* ( 6 ^| 7 ) / ( 8 ^^ 9 )

❖ GS> 0 1 23 add 4 div 5 mul 67 mul add 8 sub 9 add = //PostScript

❖ Excel: = COS(0) + 1 - ( 2 - POWER( ( FACT(3) - 4 ), 5 ) \* 67 - 8 + 9

❖ Word: = NOT(0) + 12 + 34 \* 56 + 7 + 89

❖ calc: 0 ! + 12 + 34 \* 56 + 7 + 89 =

❖ calc: 0 ! + 1 - ( 2 - ( 3 ! - 4 ) <sup>5</sup> ) \* 67 - 8 + 9 =

y

## 表达式求值

❖ 表达式的求值，可视作 **字符串** 与 **对应数值** 交替转换的过程

❖ **str(v)**：数值v对应的（十进制）**字符串**（名）

**val(S)**：符号串S对应的（十进制）**数值**（实）

❖ 设表达式

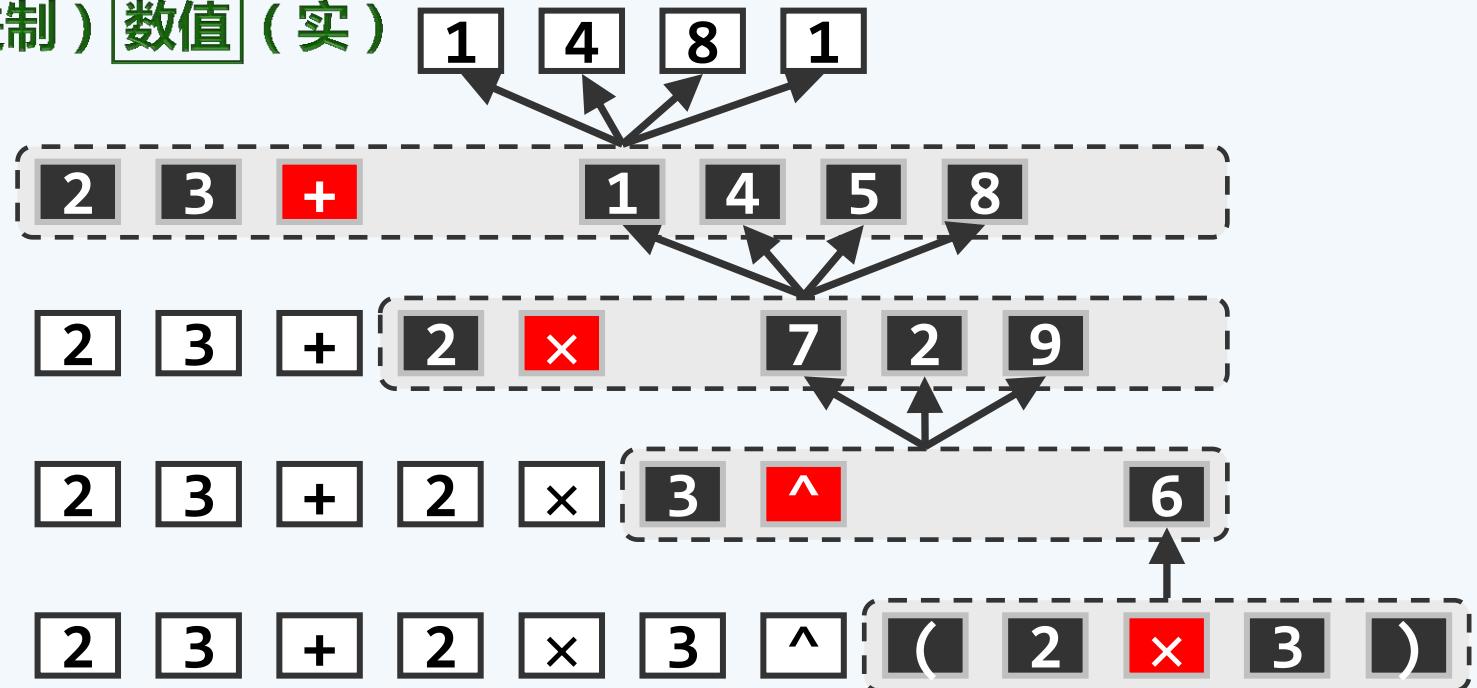
$$S = S_L + S_\theta + S_R$$

1)  **$S_\theta$**  可优先计算，且

$$2) \text{val}(S_\theta) = v_\theta$$

❖ 则有递推化简关系

$$\text{val}(S) = \text{val}(\boxed{S_L + \text{str}(v_\theta) + S_R})$$



## 4. 栈与队列

中缀表达式求值

构思

邓俊辉

deng@tsinghua.edu.cn

## 优先级

❖ 难点：如何高效地找到可优先计算的  $S_a$ （亦即，其对应的运算符）？

❖ 与括号匹配迭代版类似，但亦不尽相同

- 不能简单地按“左先右后”次序处理各运算符
- 此时，需要考虑更多因素...

❖ (约定俗成的) 优先级 :  $1 + 2 * 3 ^ 4 !$

可强行改变次序的括号 :  $((((1+2)*3)^4) !$

## 延迟缓冲

- ❖ 仅根据表达式的前缀，不足以确定各运算符的计算次序  
只有在获得足够的后续信息之后，才能确定其中哪些运算符可以执行
- ❖ 体现在求值算法的流程上  
为处理某一前缀，必须提前预读并分析更长的前缀
- ❖ 为此，需借助某种支持延迟缓冲的机制...



**求值算法 = 栈 + 线性扫描**

❖ 自左向右扫描表达式，用栈记录已扫描的部分（含已执行运算的结果）  
在每一字符处

while ( 栈的顶部存在可优先计算的子表达式 )

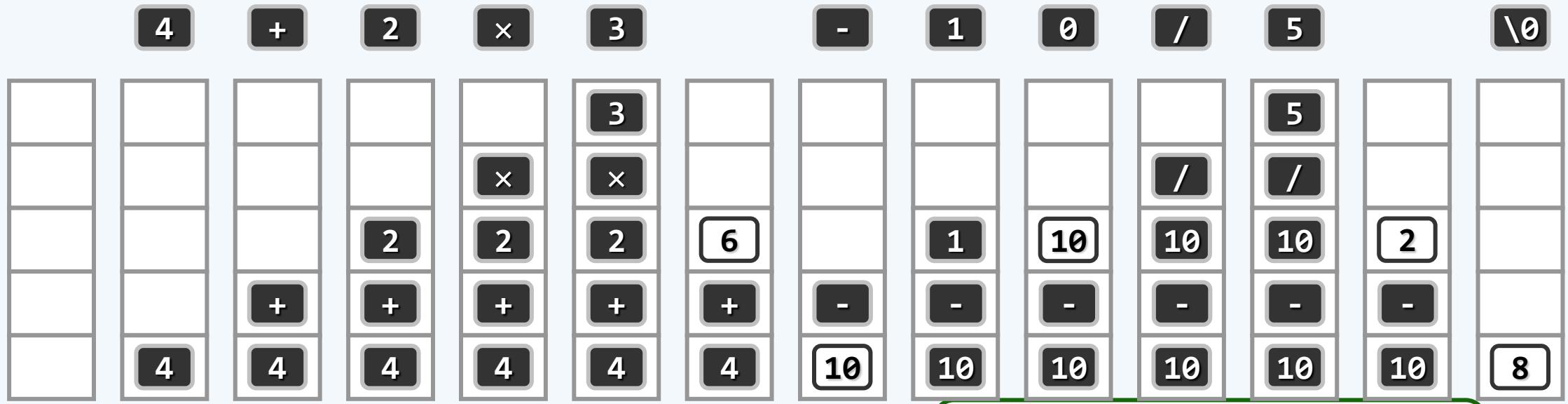
## //如何判断？

**该子表达式退栈；计算其数值；计算结果进栈**

**当前字符进栈，转入下一字符**

❖ 只要语法正确，则栈内最终应只剩一个元素

//即表达式对应的数值



## 4. 栈与队列

中缀表达式求值

算法

邓俊辉

deng@tsinghua.edu.cn

## 主算法

❖ float evaluate( char\* S, char\* & RPN ) { //中缀表达式求值 : S 语法正确

Stack<float> opnd; Stack<char> optr; //运算数栈、运算栈

optr.push( '\0' ); //尾哨兵 '\0' 也作为头哨兵首先入栈

while ( ! optr.empty() ) { //逐个处理各字符，直至运算符栈空

if ( isdigit( \*S ) ) //若当前字符为操作数，则

readNumber( S, opnd ); //读入（可能多位的）操作数

else //若当前字符为运算符，则视其与栈顶运算符之间优先级的高低

switch( orderBetween( optr.top(), \*S ) ) { /\* 分别处理 \*/ }

} //while

return opnd.pop(); //弹出并返回最后的计算结果

}

## 优先级表

```

const char pri[N_OPTR][N_OPTR] = { //运算符优先等级 [栈顶][当前]
    //-----|-----当前运算符-----|
    //      +     -     *     /     ^     !     (     )     \0
/* -- + */ '>', '>', '<', '<', '<', '<', '<', '>', '>',
/* | - */ '>', '>', '<', '<', '<', '<', '<', '>', '>',
/* 栈 */ '>', '>', '>', '>', '<', '<', '<', '>', '>',
/* 顶 */ '>', '>', '>', '>', '<', '<', '<', '>', '>',
/* 运 */ '>', '>', '>', '>', '>', '<', '<', '>', '>',
/* 算 */ '>', '>', '>', '>', '>', '>', '>', '>', '>',
/* 符 */ '<', '<', '<', '<', '<', '<', '<', '=', '>',
/* | ) */ '>', '>', '>', '>', '>', '>', '>', '>', '>',
/* -- \0 */ '<', '<', '<', '<', '<', '<', '<', '<', '='
};

}

```

## 不同优先级处理方法

```

❖ switch( orderBetween( optr.top(), *s ) ) {
    case '<': //栈顶运算符优先级更低
        optr.push( *s ); S++; break; //计算推迟，当前运算符进栈
    case '=': //优先级相等（当前运算符为右括号，或尾部哨兵'\0'）
        optr.pop(); S++; break; //脱括号并接收下一个字符
    case '>': { //栈顶运算符优先级更高，实施相应的计算，结果入栈
        char op = optr.pop(); //栈顶运算符出栈，执行对应的运算
        if ( '!' == op ) opnd.push( calcu( op, opnd.pop() ) ); //一元运算符
        else { float p0opnd2 = opnd.pop(), p0opnd1 = opnd.pop(); //二元运算符
            opnd.push( calcu( p0opnd1, op, p0opnd2 ) ); //实施计算，结果入栈
        } //为何不直接：opnd.push( calcu( opnd.pop(), op, opnd.pop() ) )?
        break;
    } //case '>'
} //switch

```

## 优先级表 (理解)

```
const char pri[N_OPTR][N_OPTR] = { //运算符优先等级 [栈顶][当前]
```

//	----- 当前运算符 -----							
//	+    -    *    /    ^    !    (    )    \0							
/* -- + */	'>', '>', '<', '<', '<', '<', '<', '<', '\0'							
/*   - */	'>', '>', '<', '<', '<', '<', '<', '<', '\0'							
/* 栈 */	'>', '>', '>', '>', '<', '<', '<', '<', '\0'							
/* 顶 */	'>', '>', '>', '>', '<', '<', '<', '<', '\0'							
/* 运 */	'>', '>', '>', '>', '>', '<', '<', '<', '\0'							
/* 算 */	'>', '>', '>', '>', '>', '>', '>', '>', '\0'							
/* 符 */	'<', '<', '<', '<', '<', '<', '<', '<', '\0'							
/*   */	' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '\0'							
/* -- \0 */	'<', '<', '<', '<', '<', '<', '<', '<', '\0'							

};

## 4. 栈与队列

中缀表达式求值

实例

邓俊辉

deng@tsinghua.edu.cn

1/3

表达式	运算符栈	操作数栈	注解
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$		表达式起始标识入栈
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ (		左括号入栈
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ (	0	操作数0入栈
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ ( !	0	运算符'!'入栈
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ (	1	运算符'!'出栈执行
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ ( +	1	运算符'+'入栈
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ ( +	1 1	操作数1入栈
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ (	2	运算符'+'出栈执行
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$	2	左括号出栈
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ *	2	运算符'*'入栈
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ *	2 2	操作数2入栈
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ * ^	2 2	运算符'^'入栈
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ * ^ (	2 2	左括号入栈
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ * ^ (	2 2 3	操作数3入栈

2/3

表达式	运算符栈	操作数栈	注解
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ * ^ ( !	2 2 3	运算符'!'入栈
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ * ^ (	2 2 6	运算符'!'出栈执行
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ * ^ ( +	2 2 6	运算符'+'入栈
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ * ^ ( +	2 2 6 4	操作数4入栈
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ * ^ (	2 2 10	运算符'+'出栈执行
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ * ^	2 2 10	左括号出栈
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ *	2 1024	运算符'^'出栈执行
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$	2048	运算符'*'出栈执行
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ -	2048	运算符'-'入栈
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ - (	2048	左括号入栈
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ - (	2048 5	操作数5入栈
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ - ( !	2048 5	运算符'!'入栈
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ - (	2048 120	运算符'!'出栈执行
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ - ( -	2048 120	运算符'-'入栈

表达式	运算符栈	操作数栈	注解
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ - ( -	2048 120 67	操作数67入栈
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ - (	2048 53	运算符'-'出栈执行
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ - ( -	2048 53	运算符'-'入栈
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ - ( - (	2048 53	左括号入栈
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ - ( - (	2048 53 8	操作数8入栈
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ - ( - ( +	2048 53 8	运算符'+'入栈
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ - ( - ( +	2048 53 8 9	操作数9入栈
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ - ( - (	2048 53 17	运算符'+'出栈执行
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ - ( -	2048 53 17	左括号出栈
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ - (	2048 36	运算符'-'出栈执行
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$ -	2048 36	左括号出栈
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$	\$	2012	运算符'-'出栈执行
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$		2012	表达式起始标识出栈
$(0!+1)*2^(3!+4)-(5!-67-(8+9))\$$			返回唯一的元素2012

## 4. 栈与队列

逆波兰表达式

- 定义

邓俊辉

deng@tsinghua.edu.cn

RPN = Reverse Polish Notation

❖ 逆波兰表达式 : J. Lukasiewicz ( 12/21/1878 - 02/13/1956 )

❖ 在由运算符 ( operator ) 和操作数 ( operand ) 组成的表达式中  
不使用括号 ( parenthesis-free ) , 即可表示带优先级的运算关系

❖ 例如 :    0 ! + 123 + 4 \* ( 5 \* 6 ! + 7 ! / 8 ) / 9

              0 ! 123 + 4 5 6 ! \* 7 ! 8 / + \* 9 / +

❖ 又如 :    0 ! - ( 1 + 23 - 4 - 56 ) \* 7 \* 8 - 9

              0 ! 1 23 + 4 - 56 - 7 \* 8 \* - 9 -

❖ 相对于日常使用的中缀式 ( infix ) , RPN 亦称作后缀式 ( postfix )

❖ 作为补偿 , 须额外引入一个起分隔作用的元字符 ( 比如空格 ) // 较之原表达式 , 未必更短

## 4. 栈与队列

### 逆波兰表达式

- 求值

邓俊辉

deng@tsinghua.edu.cn

## 算法

❖ rpnEvaluate( expr ) //假定RPN表达式expr的语法正确

引入栈s，用以存放操作数 //故亦称作栈式求值

while ( expr尚未扫描完毕 ) {

    读入expr的下一元素x

    if ( x是操作数 )

        将x压入s

    else { //x是运算符

        从栈s中弹出运算符x所需数目的操作数

        对弹出的操作数实施x运算，并将运算结果重新压入s

    } //else

} //while

返回栈顶 //也是栈底

## 实例

0 ! 123 + 4 5 6 ! \* 7 ! 8 / + \* 9 / +

630

4230

1880

2004

0 ! 1 + 2 3 ^ 4 ! - 5 ! 6 / - 7 \* 8 \* - 9 -

0 ! 1 + 2 3 ^ 4 ! - 5

0	!	1	+	2	3	^	4	!	-	5
0	1	1	2	2	2	2	4	24	8	5

! 6 / - 7 \* 8 \* - 9 -

!	6	/	-	7	*	8	*	-	9	-
120	6	20	2	7	-36	8	-252	-2016	9	2009
-16	120	-16	-36	-36	2	2	2	2018	2018	
2	2	2	2	2	2	2	2	2		

363

0 ! 1 + 2 3 ! 4 - 5 ^ - 67 \* - 8 - 9 +

0 ! 1 + 2 3 ! 4 - 5 ^ - 67 \* - 8 - 9 +

32

-2010

2004

## 4. 栈与队列

### 逆波兰表达式

- 转换

将欲去之，必固举之

将欲夺之，必固予之

将欲灭之，必先学之

邓俊辉

deng@tsinghua.edu.cn

## infix到postfix：手工转换

❖ 例如： $(0! + 1)^\wedge (2 * 3! + 4 - 5)$

假设：事先未就运算符之间的优先级关系做过任何约定

1) 用括号显式地表示优先级

$$\{ (([0!] + 1)^\wedge ([((2 * [3!]) + 4] - 5))) \}$$

2) 将运算符移到对应的右括号后

$$\{ (([0]! 1) + ([((2 [3]!) * 4] + 5) - ])^\wedge$$

3) 抹去所有括号

$$0 ! 1 + 2 3 ! * 4 + 5 - ^$$

4) 稍事整理，即得

$$0 ! 1 + 2 3 ! * 4 + 5 - ^$$

❖ 中缀式求值算法evaluate()略做扩展，亦可同时完成RPN转换...

## infix到postfix：转换算法

```

❖ float evaluate( char* S, char* & RPN ) { //RPN转换
    /* ..... */
    while ( ! optr.empty() ) { //逐个处理各字符，直至运算符栈空
        if ( isdigit( * S ) ) //若当前字符为操作数，则直接将其
            { readNumber( S, opnd ); append( RPN, opnd.top() ); } //接入RPN
        else //若当前字符为运算符
            switch( orderBetween( optr.top(), *S ) ) {
                /* ..... */
                case '>': { //且可立即执行，则在执行相应计算的同时将其
                    char op = optr.pop(); append( RPN, op ); //接入RPN
                    /* ..... */
                } //case '>'
            }
    }
}

```

## 思考

- ❖ 这里的多数实例中，为何操作数都是顺序排列？
- ❖ “3 + 4” 除了可以转换为 “3 4 +” ，是否也可转换为 “4 3 +” ？
- ❖ 既然evaluate()算法已经能够求值，同时完成RPN转换又有何意义？
- ❖ 相对于原表达式，存储RPN所需的空间是否一定更少？
- ❖ 在数学意义上完全对称的前缀表达式，为何在类似问题中很少应用？

## 4. 栈与队列

逆波兰表达式

- PostScript

邓俊辉

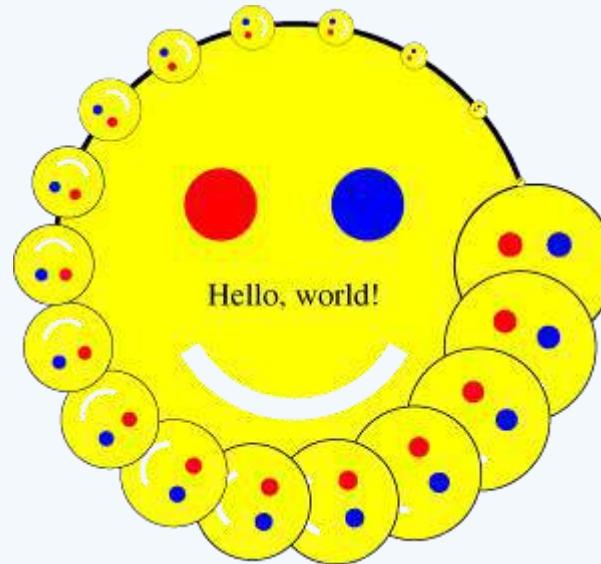
deng@tsinghua.edu.cn

## PostScript

- ❖ 诞生于1985，支持设备独立的图形描述
- ❖ ( 1个解释器 + 5个栈 ) × RPN语法
- ❖ operand stack：存放操作数及运算结果
- ❖ 一旦遇到操作符，则
  - 弹出相应数目的元素
  - 实施计算，并
  - 将（可能多个、一个或零个）结果入栈
- ❖ 实例：`4 4 mul 5 5 mul add 7 mul 7 mul`
- ❖ 提供基础且强大的图形功能，支持数据类型、变量、函数/宏 ...

## PostScript

```
❖ /smile {  
    newpath  
    gsave  
    rotate  
    0 translate  
    180 div dup scale  
    yellow 0 0 180 0 360 arc fill  
    red -55 45 27 0 360 arc fill  
    blue 55 45 27 0 360 arc fill  
    fatline white 0 -18 90 210 330 arc stroke  
    thinline black 0 0 180 0 360 arc stroke  
    grestore  
} def
```



## 4. 栈与队列

队列接口与实现

邓俊辉

deng@tsinghua.edu.cn

## 操作与接口

❖ 队列 ( queue ) 也是受限的序列

只能在队尾插入 ( 查询 ) : enqueue()

rear()

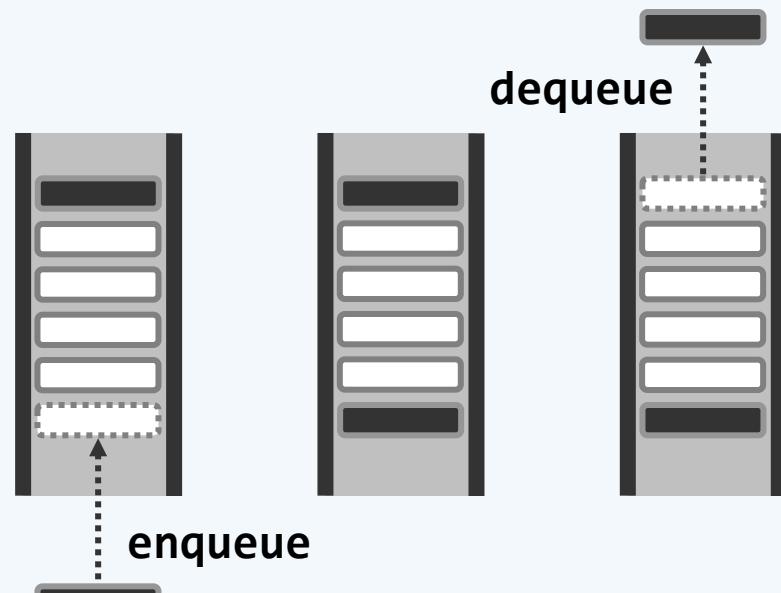
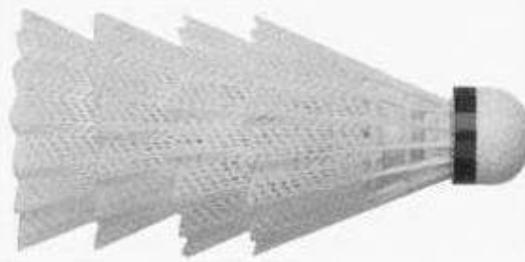
只能在队头删除 ( 查询 ) : dequeue()

front()

❖ 先进先出 ( FIFO )

后进后出 ( LIFO )

❖ 扩展接口 : getMax() ...



## 操作实例

操作	输出	队列 ( 右侧为队头 )			
Queue()					
empty()	true				
enqueue(5)		5			
enqueue(3)		3	5		
dequeue()	5	3			
enqueue(7)		7	3		
enqueue(3)		3	7	3	
front()	3	3	7	3	
empty()	false	3	7	3	

操作	输出	队列 ( 右侧为队头 )			
enqueue(11)		11	3	7	3
size()	4	11	3	7	3
enqueue(6)		6	11	3	7
empty()	false	6	11	3	7
enqueue(7)		7	6	11	3
dequeue()	3	7	6	11	3
dequeue()	7	7	6	11	3
front()	3	7	6	11	3
size()	4	7	6	11	3

## 模板类

- ❖ 队列既然属于序列的特列，故亦可直接基于向量或列表派生
- ❖ `template <typename T> class Queue: public List<T> { //由列表派生的队列模板类`
- `public: //size()与empty()直接沿用`
- `void enqueue( T const & e ) { insertAsLast( e ); } //入队`
- `T dequeue() { return remove( first() ); } //出队`
- `T & front() { return first()->data; } //队首`
- `}; //以列表首/末端为队列头/尾——颠倒过来呢？`
- ❖ 确认：如此实现的队列接口，均只需 $O(1)$ 时间
- ❖ 课后：基于向量，派生定义队列模板类  
评测：你所实现的队列接口，效率如何？

## 4. 栈与队列

队列应用

邓俊辉

deng@tsinghua.edu.cn

## 资源循环分配

❖ 一群客户 (client) 共享同一资源时，如何兼顾公平与效率？

比如，多个应用程序共享CPU，实验室成员共享打印机，...

❖ RoundRobin { //循环分配器

```
Queue Q( clients ); //参与资源分配的所有客户组成队列
```

```
while ( ! ServiceClosed() ) { //在服务关闭之前，反复地
```

```
    e = Q.dequeue(); //令队首的客户出队，并
```

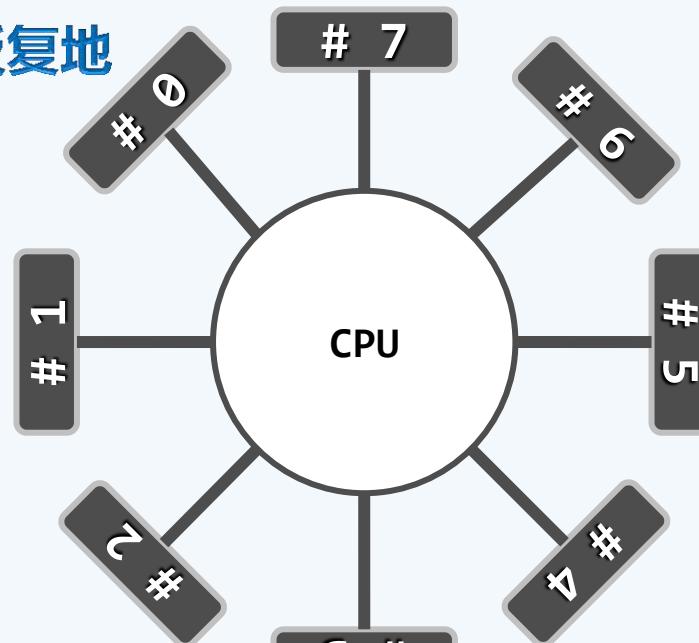
```
    serve( e ); //接受服务，然后
```

```
    Q.enqueue( e ); //重新入队
```

```
}
```

```
}
```

❖ 利用队列改进迷宫算法，找出最短的通路



## 银行服务模拟

❖ 模型： 提供n个服务窗口

任一时刻，每个窗口至多接待一位顾客，其他顾客排队等候

顾客到达后，自动地选择和加入最短队列（的末尾）

❖ 参数： nWin //窗口（队列）数目

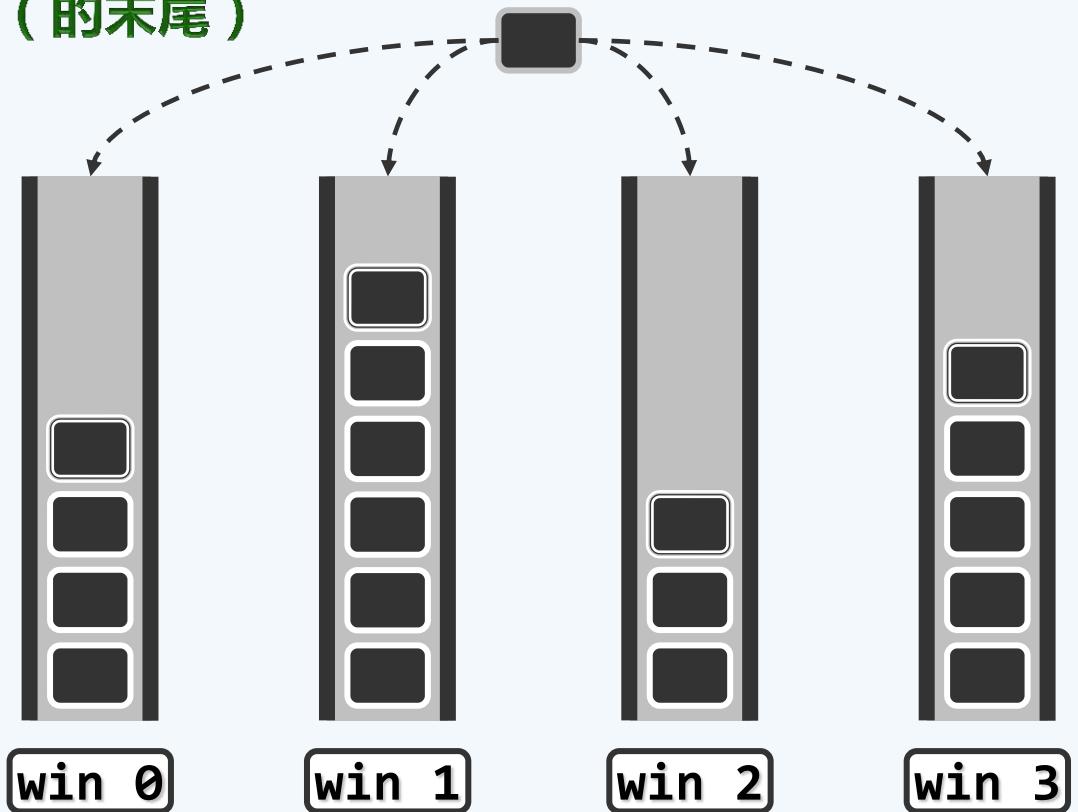
servTime //营业时长

❖ struct Customer { //顾客类

int window; //所属窗口（队列）

unsigned int time; //服务时长

};



## 银行服务模拟

```
❖ void simulate( int nWin, int servTime ) {  
    Queue<Customer> * windows = new Queue<Customer>[ nWin ];  
    for ( int now = 0; now < servTime; now++ ) { //在下班之前，每隔单位时间  
        Customer c ; c.time = 1 + rand() % 50; //一位新顾客到达，其服务时长随机指定  
        c.window = bestWindow( windows, nWin ); //找出最佳（最短）服务窗口  
        windows[ c.window ].enqueue( c ); //新顾客加入对应的队列  
        for ( int i = 0; i < nWin; i++ ) //分别检查  
            if ( ! windows[ i ].empty() ) //各非空队列  
                if ( -- windows[ i ].front().time <= 0 ) //队首顾客接受服务  
                    windows[ i ].dequeue(); //服务完毕则出列，由后继顾客接替  
    } //for  
    delete [] windows; //释放所有队列  
}
```

## 4. 栈与队列

Stead + Queap

邓俊辉

deng@tsinghua.edu.cn

**Steap = Stack + Heap = push + pop + getMax**

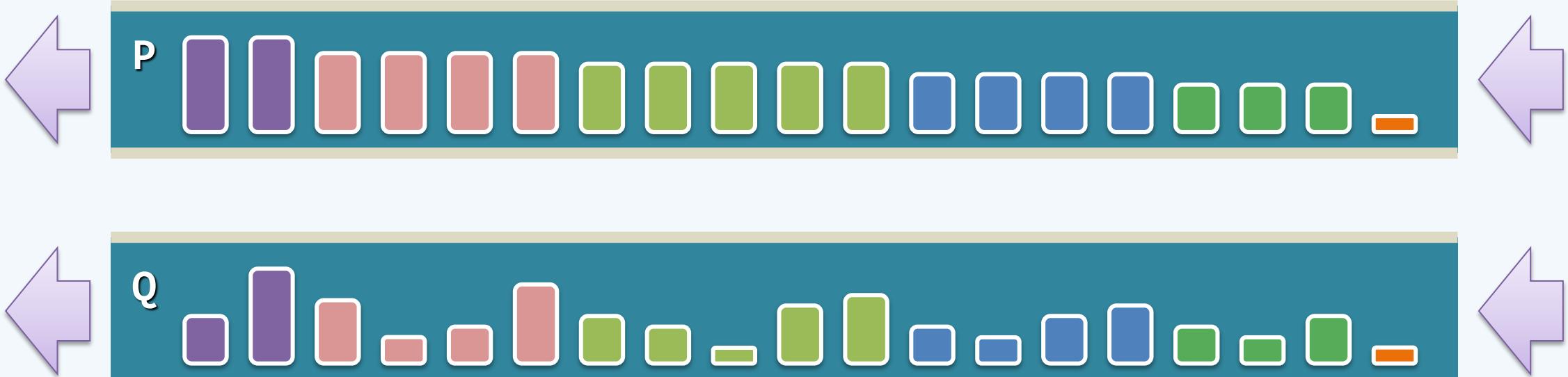


- ❖ `S.pop(); P.pop(); //O(1)`
- ❖ `S.push(e); P.push( max( e, P.top() ) ); //O(1)`

Steap = Stack + Heap = push + pop + getMax



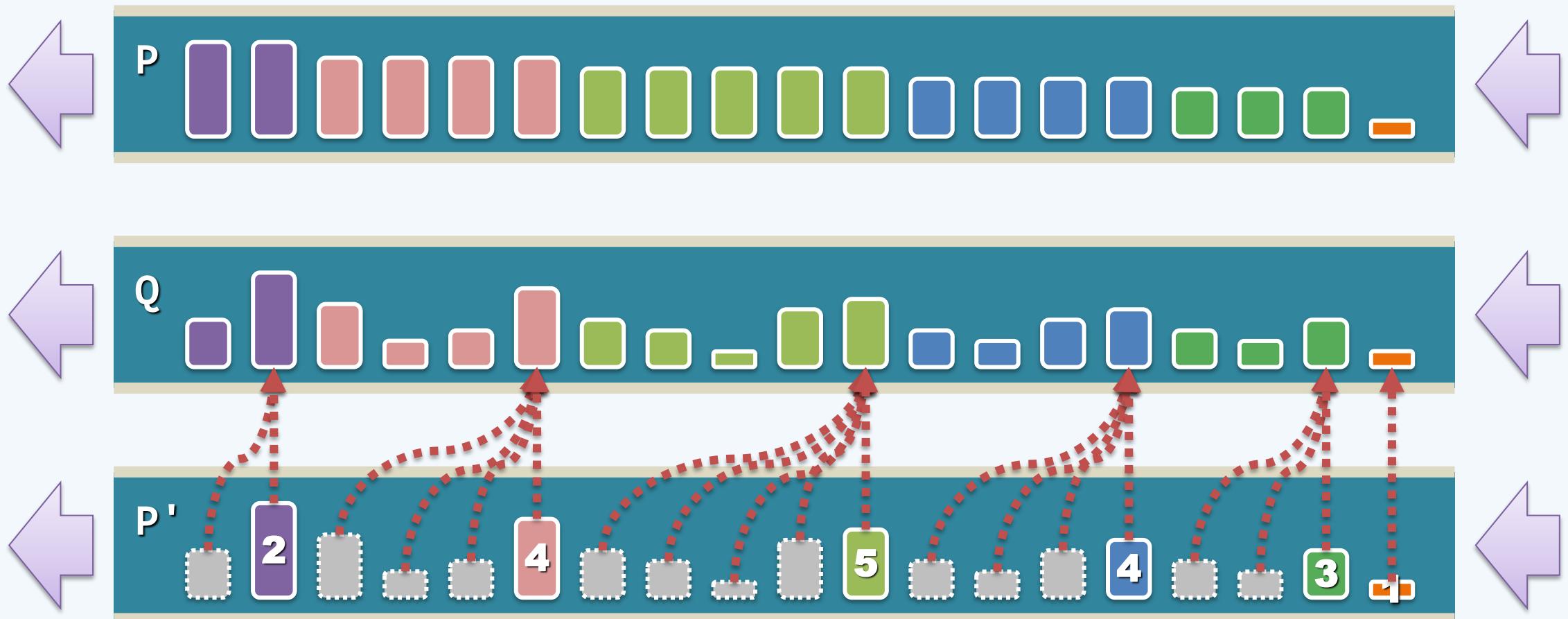
Queap = Queue + Heap = enqueue + dequeue + getMax



❖ Q.dequeue(); P.dequeue(); //O(1)

❖ Q.enqueue(e); P.enqueue(e);  
for ( x = P.rear(); x && (x->key <= e); x = x->pred ) //最坏情况O(n)  
x->key = e;

**Queap** = Queue + Heap = enqueue + dequeue + getMax



```
/* D. Osovianski & B. Nissenbaum, 1990 */
int v,i,j,k,l,s,a[99];void main(){for(scanf("%d",&s);*a-s;
v=a[j*=v]-a[i],k=i<s,j+=(v=j<s&&(!k&&!!printf(2+"\\n\\n%c"-(
!l<<!j)," .Q"[l^v?(l^j)&1:2])&&++l||a[i]<s&&v&&v-i+j&&v+i-
j))&&!(l%=s),v||(i==j?a[i+=k]=0:++a[i])>=s*k&&++a[--i]);}
```

```
/* Andor@net9.org, 2002 */
#define q(o)a[j]o[j+i+7]o[j-i+31]
a[39];
main(i,j)
{ for(j=9;--j;i>8?printf("%10d",a[j]):q(|a)||q(=a)=i,main(i+1),q(=a)=0)); }
```

## 4. 栈与队列

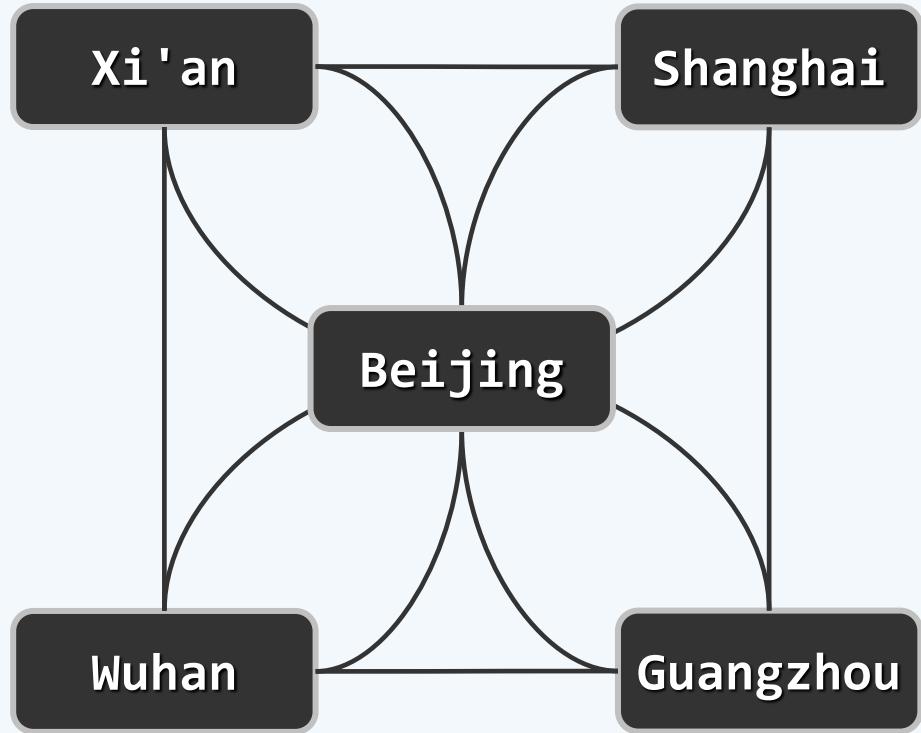
### 试探回溯法：八皇后

邓俊辉

deng@tsinghua.edu.cn

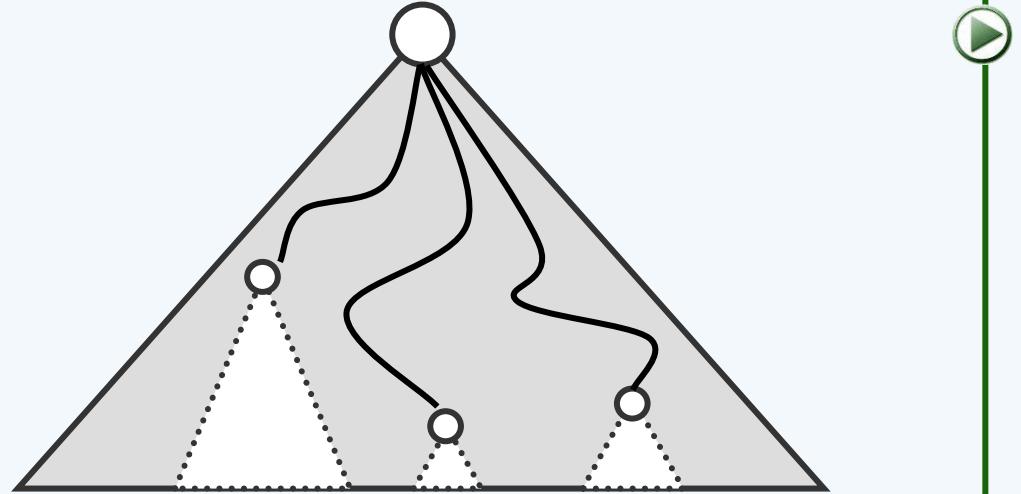
## 指数爆炸

- ❖ 很多问题的解，形式上都可看作若干元素按特定次序构成的序列
- ❖ 以TSP问题为例，即给定 $n$ 个城市之间总成本最低的环游路线
- ❖ 每一排列组合都是一个候选解往往构成一个极大的搜索空间
- ❖ 仍以TSP为例，共有： $n! / n = (n-1)! = O(n^n)$
- ❖ 若采用蛮力策略求解  
需逐一生成可能的候选解，并检查其是否合理  
如此，必然无法将时间复杂度控制在多项式以内



## 试探-回溯-剪枝

- ❖ 为尽可能多、尽可能早地排除候选解，须深刻理解应用问题，并利用其特有的规律
- ❖ 事实上，根据候选解的某种**局部特征**，即可判断其是否合理  
此时只要策略得当，便可成批地排除候选解  
此即所谓剪枝（pruning）
- ❖ 试探回溯（probe-backtrack）模式  
从0开始，逐渐增加候选解长度 // 试探  
一旦发现注定要失败，则  
收缩至前一长度，并 // 剪枝回溯  
继续试探
- ❖ 特修斯的法宝 = 线绳 + 粉笔  
如何以数据结构的形式兑现？



## 八皇后

❖ 在 $n \times n$ 的棋盘上放置 $n$ 个皇后，使得她们彼此互不攻击

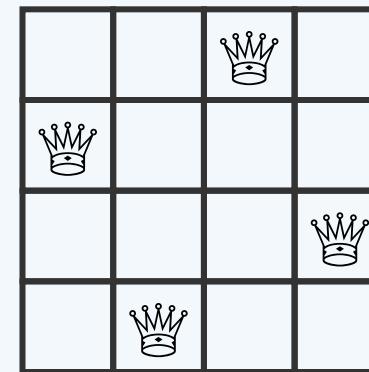
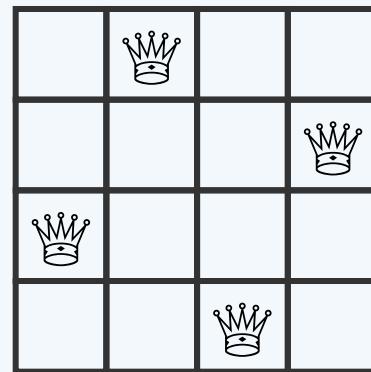
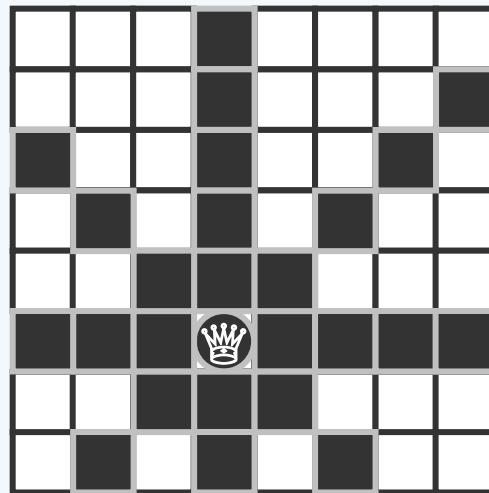
有多少种可行的布局？如何布局？

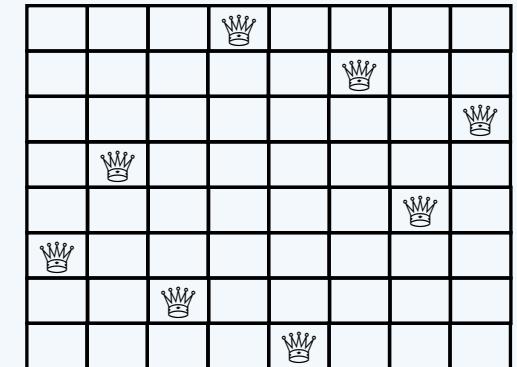
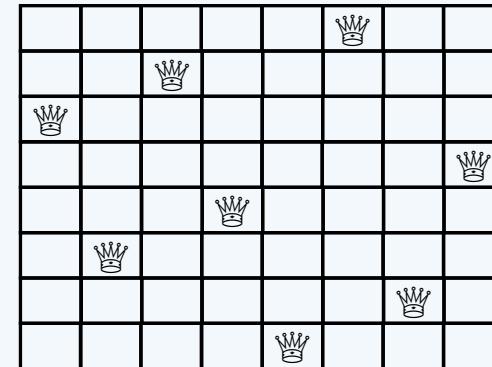
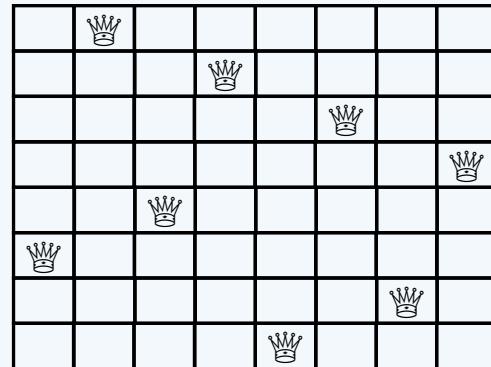
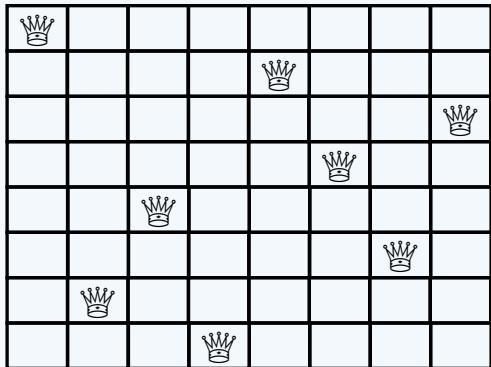
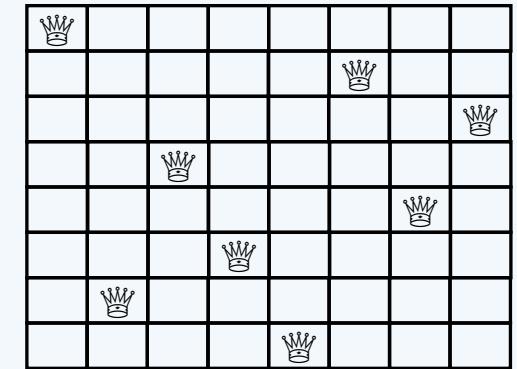
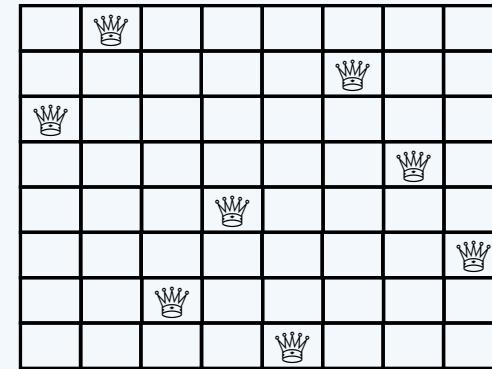
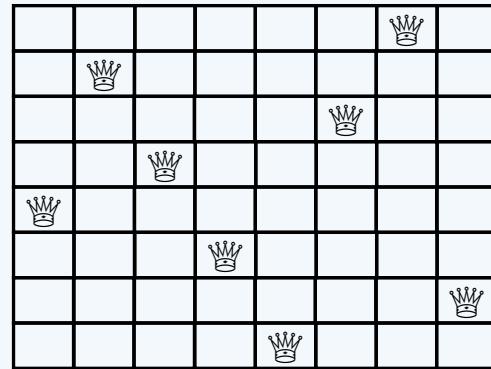
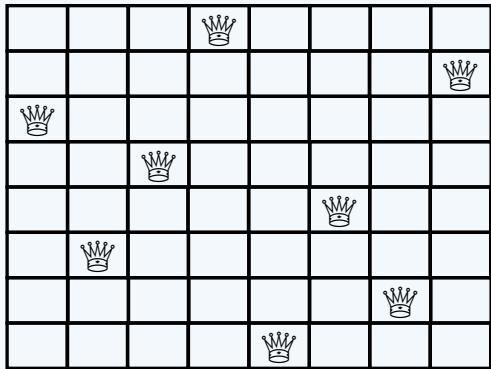
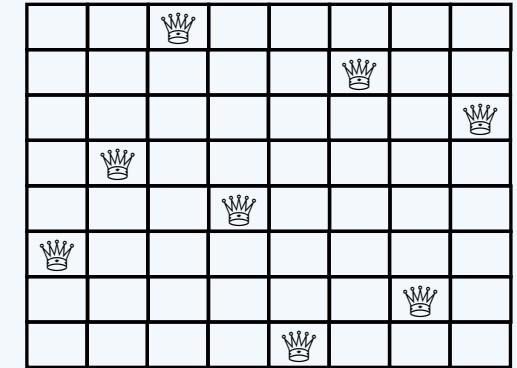
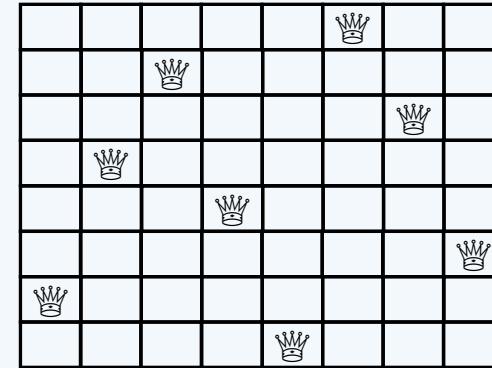
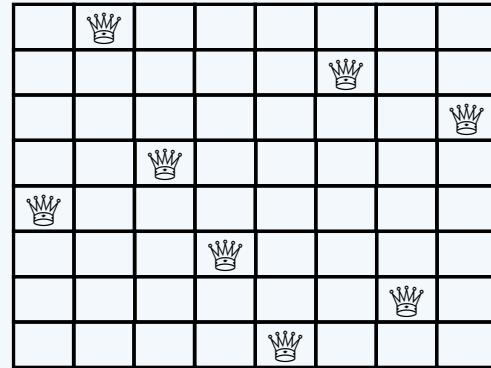
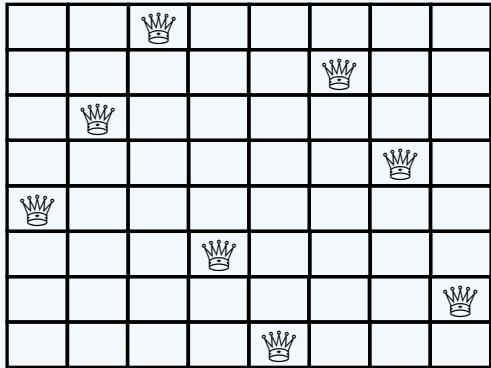
是否考虑**旋转、翻转之后的等价**？

❖  $n = 1, 2, 3, 4, \dots$

允许重复： 1, 0, 0, 2, 10, 4, 40, 92, ...

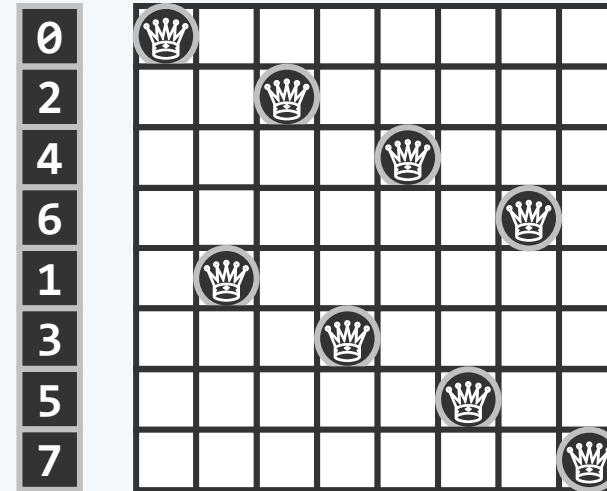
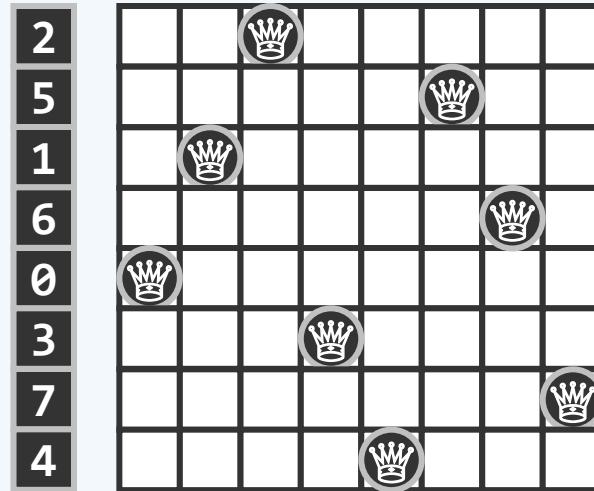
不许重复： 1, 0, 0, 1, 2, 1, 6, 12, 46, 92, ...





## 编码

- ❖ 观察：每行（列）有且仅有一个皇后
- ❖ 因此，每一布局（候选解）都可编码为整数{ 0, ..., n - 1 }的一个排列
- ❖ 反之，每一这样的排列，未必是一个可行布局（解）



## 蛮力搜索

```
❖ void place4Queens_BruteForce() { //4皇后蛮力算法  
    int solu[4]; //候选解编码向量  
  
    for ( solu[0] = 0; solu[0] < 4; solu[0]++ )  
        for ( solu[1] = 0; solu[1] < 4; solu[1]++ )  
        for ( solu[2] = 0; solu[2] < 4; solu[2]++ )  
        for ( solu[3] = 0; solu[3] < 4; solu[3]++ ) { //枚举所有候选解  
            if ( collide( solu, 0 ) ) continue;  
            if ( collide( solu, 1 ) ) continue;  
            if ( collide( solu, 2 ) ) continue;  
            if ( collide( solu, 3 ) ) continue;  
            nSolu++; displaySolution( solu, 4 );  
        }  
    } //复杂度高达O(4^4) = O(n^n)
```

## 剪枝

```
❖ void place4Queens() { //4皇后剪枝算法
    int solu[4]; //候选解编码向量
    for ( solu[0] = 0; solu[0] < 4; solu[0]++ )
        if ( ! collide( solu, 0 ) ) //剪枝
            for ( solu[1] = 0; solu[1] < 4; solu[1]++ )
                if ( ! collide( solu, 1 ) ) //剪枝
                    for ( solu[2] = 0; solu[2] < 4; solu[2]++ )
                        if ( ! collide( solu, 2 ) ) //剪枝
                            for ( solu[3] = 0; solu[3] < 4; solu[3]++ )
                                if ( ! collide( solu, 3 ) ) { //剪枝
                                    nSolu++; displaySolution( solu, 4 );
                                }
    } //复杂度大大降低，但算法的通用性欠佳
```

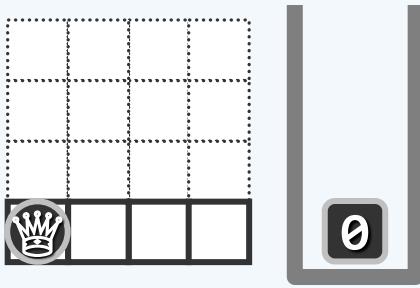
## 通用算法

```

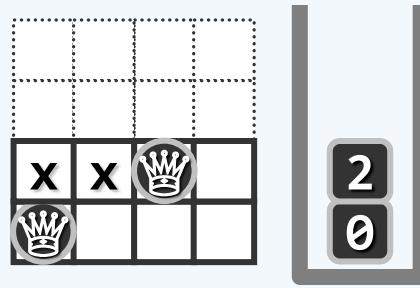
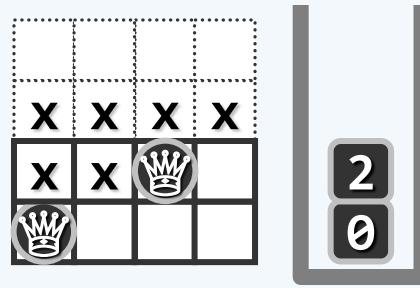
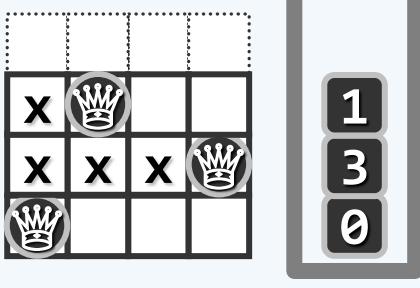
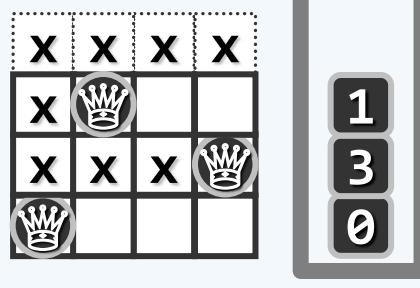
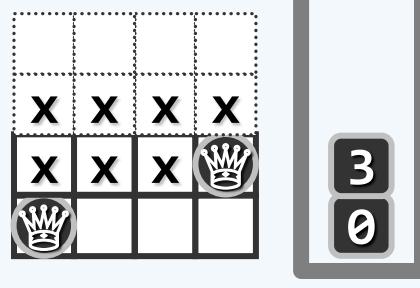
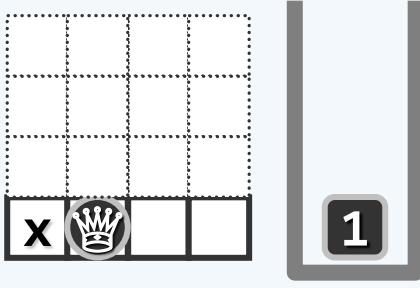
❖ void placeQueens(int N) { //N = 棋盘大小 = 皇后总数，问题的规模可任意
    Stack<Queen> solu; Queen q(0, 0); //存放（部分）解的栈，从原点位置出发
    do { //反复试探、回溯
        if (N <= solu.size() || N <= q.y) { //若已出界，则
            q = solu.pop(); q.y++; //回溯一行，并继续试探下一列
        } else { //否则，试探下一行
            while ( (q.y < N) && ( 0 <= solu.find(q) ) ) //通过与已有皇后的比对
                q.y++; //尝试找到可摆放下一皇后的列
            if (N > q.y) { //若存在可摆放的列，则摆上当前皇后
                solu.push(q); if (N <= solu.size()) nSolu++; //若局部解已成全局解，则计数
                q.x++; q.y = 0; //转入下一行，从第0列开始，试探下一皇后
            }
        }
    } while ((0 < q.x) || (q.y < N)); //直至所有分支均已被检查或剪枝
}

```

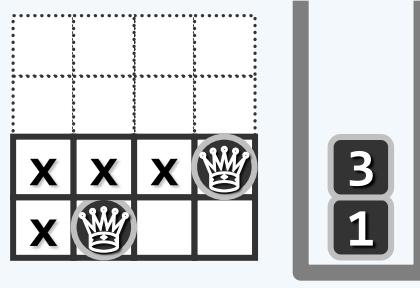
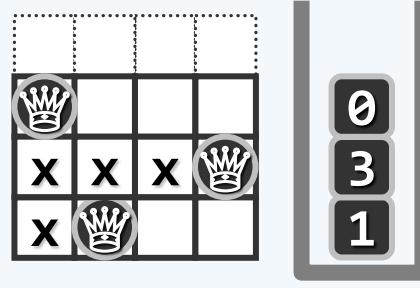
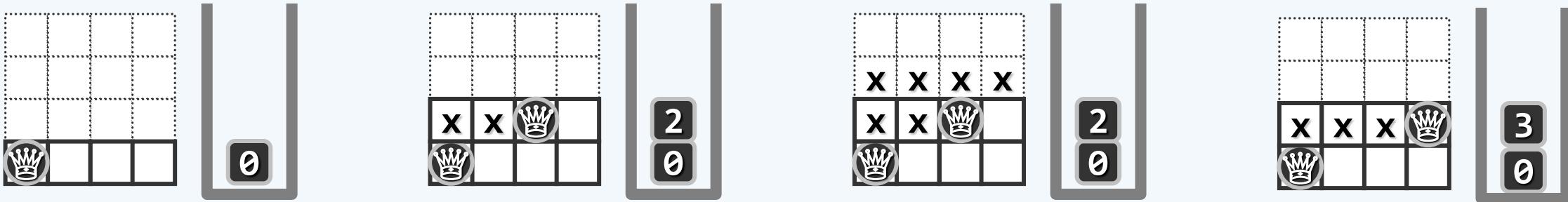
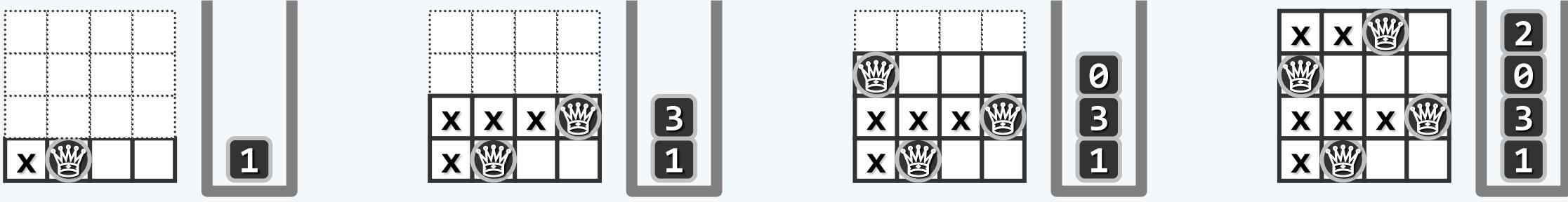
## 实例



0

2  
02  
01  
3  
01  
3  
03  
0

1

3  
10  
3  
13  
02  
0  
3  
1

❖ 以上算法中的 “`solu.find(q)`” , 如何利用栈 ( 向量 ) 的查找接口 ?

❖ 定义皇后类`Queen` , 重新定义判等器 , 使之在语义上与冲突等价

❖ `struct Queen { //皇后类`

`int x, y; Queen( int xx = 0, int yy = 0 ) : x(xx), y(yy) {};` //皇后的坐标

`bool operator==( Queen const & q ) { //重裁判等操作符`

`return ( x == q.x ) //行冲突 ( 不会发生 , 可省略 )`

`|| ( y == q.y ) //列冲突`

`|| ( x + y == q.x + q.y ) //沿正对角线冲突`

`|| ( x - y == q.x - q.y ); //沿反对角线冲突`

`}`

`bool operator!=( Queen const & q ) { return !( *this == q ); }`

`};`

## 4. 栈与队列

### 试探回溯法：迷宫寻径

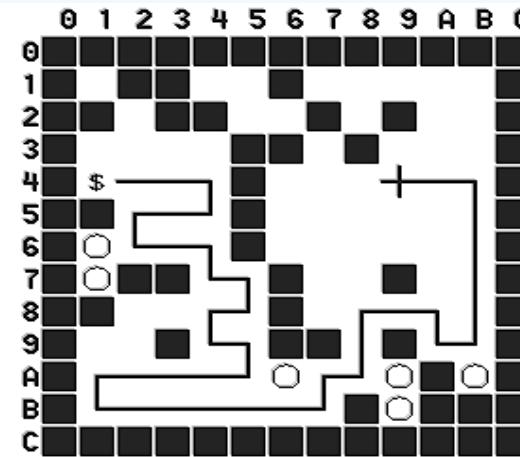
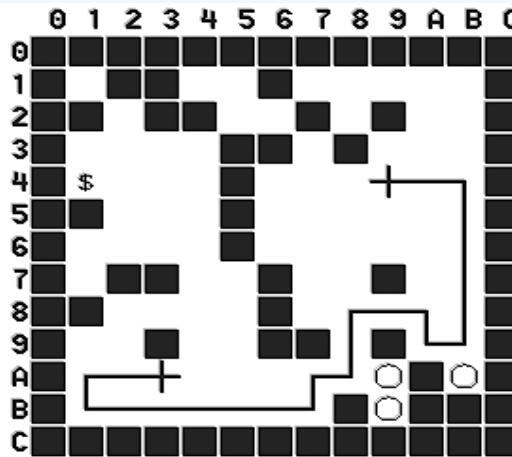
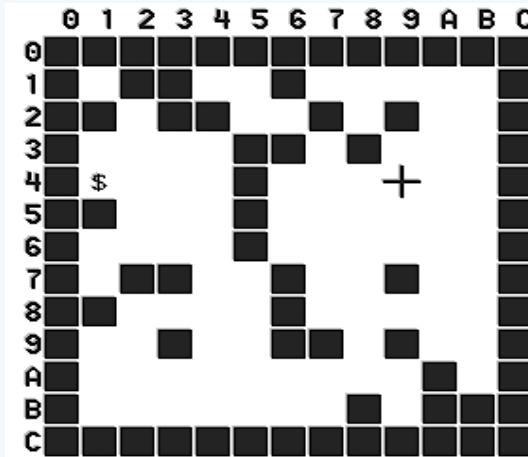
No matter where they take us,  
We'll find our own way back.

邓俊辉

deng@tsinghua.edu.cn

## 迷宫寻径

- ❖ 路径规划 ( path planning ) , 系人工智能等领域的基本问题
- ❖  $n \times n$  方格组成的迷宫，四周方格构成围墙，中间若干方格为障碍物  
机器人漫游其间，每步只能运动到东、南、西、北方向的某一邻格
- ❖ 指定的起点  $s$  和终点  $t$  , 在其间找出一条四连通的通路 ( 如果存在 )



## 算法

```

❖ bool labyrinth( Cell Laby[MAX][MAX], Cell* s, Cell* t ) {
    Stack<Cell*> path; //用栈记录通路 ( Theseus的线绳 )
    s->incoming = UNKNOWN; s->status = ROUTE; path.push(s); //从起点出发
    do { //不断试探、回溯，直到抵达终点，或者穷尽所有可能
        Cell* c = path.top(); if (c == t) return true; //找到通路；否则...
        while ( NO_WAY > ( c->outgoing = nextESWN(c->outgoing) ) ) //查找另一
            if ( AVAILABLE == neighbor(c)->status ) break; //尚未试探的方向
        if ( NO_WAY <= c->outgoing ) //若所有方向都已尝试过，则向后回溯一步
            { c->status = BACKTRACKED; c = path.pop(); } // ( Theseus的粉笔 )
        else //否则，向前试探一步
            { path.push( c = advance(c) ); c->outgoing = UNKNOWN; c->status = ROUTE; }
    } while ( !path.empty() );
    return false;
}

```

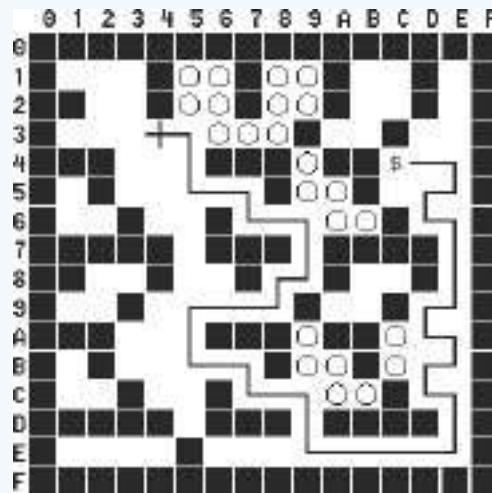
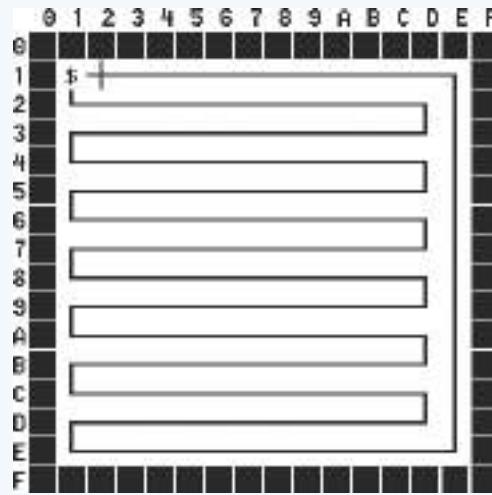
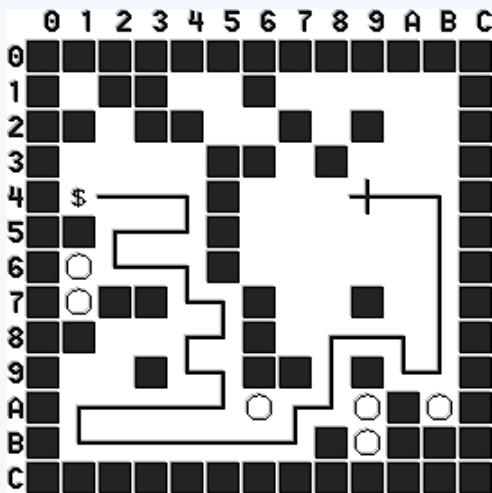
## 数据结构

```
❖ struct Cell { //迷宫单元
    int x, y; //坐标
    Status status; //类型
    ESWN incoming, outgoing; //进入、走出方向
};

❖ 相关类型
//状态：可用、在当前路径上、所有方向均尝试失败后回溯过、不可使用（墙）
typedef enum { AVAILABLE, ROUTE, BACKTRACKED, WALL } Status;
//单元的相对邻接方向：未定、东、南、西、北、无路可通
typedef enum { UNKNOWN, EAST, SOUTH, WEST, NORTH, NO_WAY } ESWN;
//依次转至下一邻接方向
inline ESWN nextESWN( ESWN eswn ) { return ESWN (eswn + 1); }
```

## 进一步的考虑

- ❖ 如何降低最坏情况的概率？  
采用随机策略，等概率试探各方向
- ❖ 如何支持八连通运动规则？  
改写neighbor()，扩充四个方向
- ❖ 如何找出更短的通路？  
环路：尽可能发现并消去  
弯路：尽可能发现并消去  
贪心：终点方向优先试探  
...
- ❖ 通用算法



## 5. 二叉树

树

Two roads diverged in a yellow wood

And sorry I could not travel both

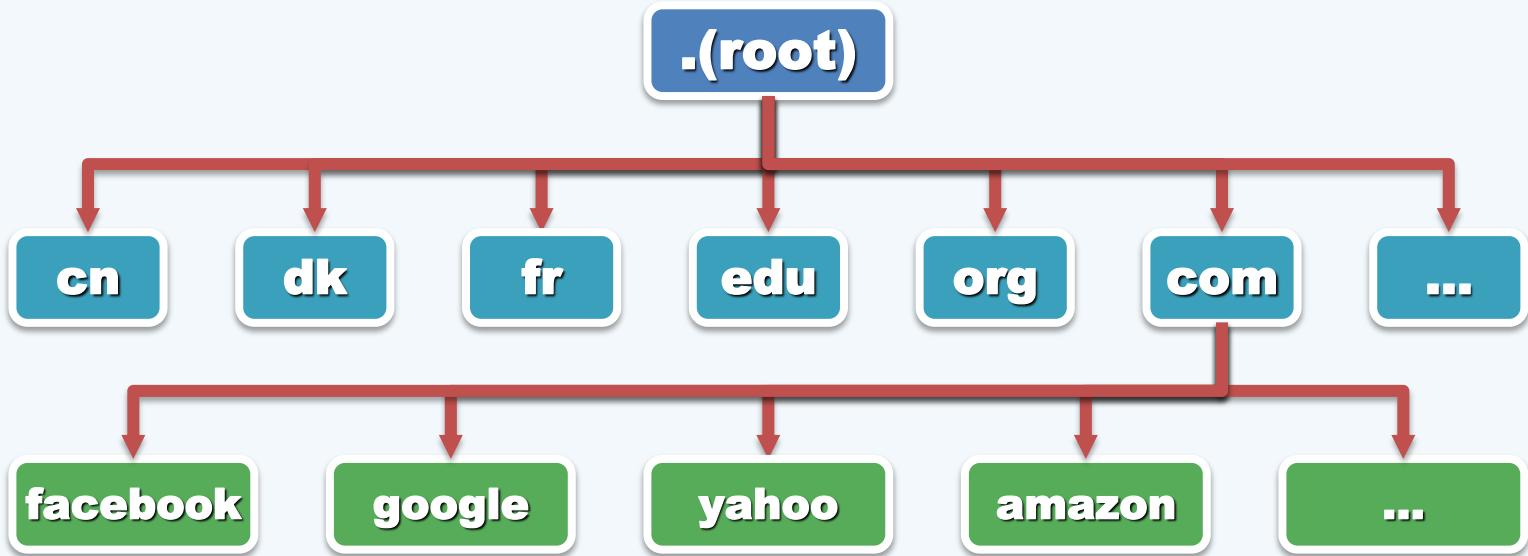
邓俊辉

deng@tsinghua.edu.cn

## 动机

### ❖ 【应用】层次结构的表示

- 表达式
- 文件系统
- URL ...

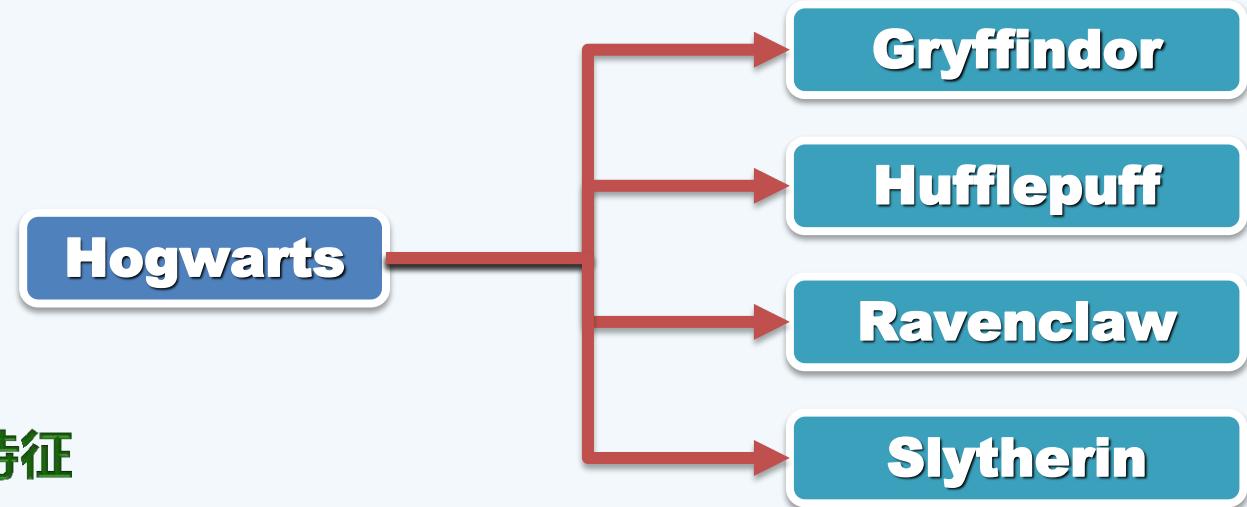


### ❖ 【数据结构】综合性

- 兼具Vector和List的优点
- 兼顾高效的查找、插入、删除

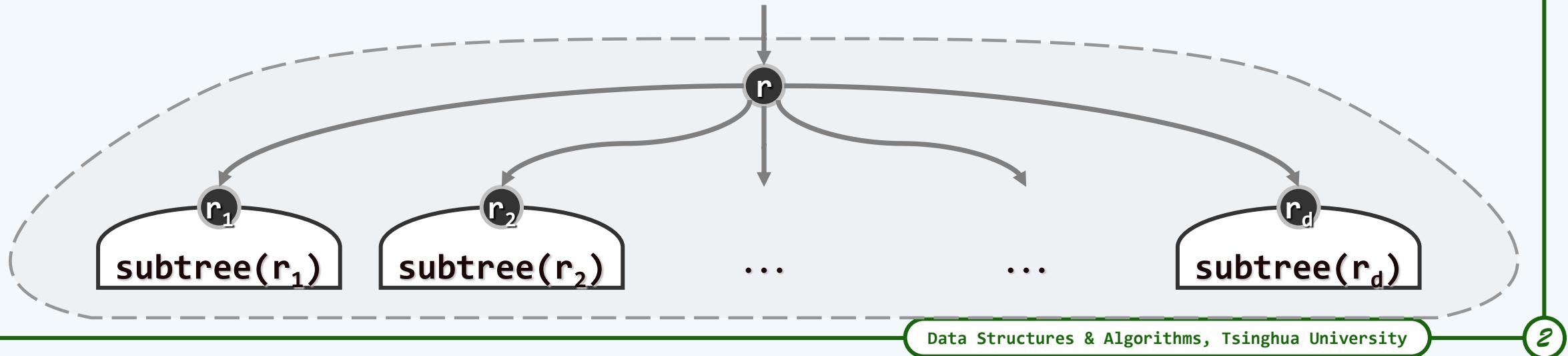
### ❖ 【半线性】

- 不再是简单的线性结构，但
- 在确定某种次序之后，具有线性特征



## 有根树

- ◆ 树是特殊的图  $T = (V, E)$ , 节点数  $|V| = n$ , 边数  $|E| = e$
- ◆ 指定任一节点  $r \in V$  作为 **根** 后,  $T$  即称作 **有根树** (rooted tree)
- ◆ 若:  $T_1, T_2, \dots, T_d$  为有根树  
则:  $T = ((\cup V_i) \cup \{r\}, (\cup E_i) \cup \{ \langle r, r_i \rangle \mid 1 \leq i \leq d \})$  也是
- ◆ 相对于  $T$ ,  $T_i$  称作以  $r_i$  为根的 **子树** (subtree rooted at  $r_i$ ), 记作  $T_i = \text{subtree}(r_i)$



## 有序树

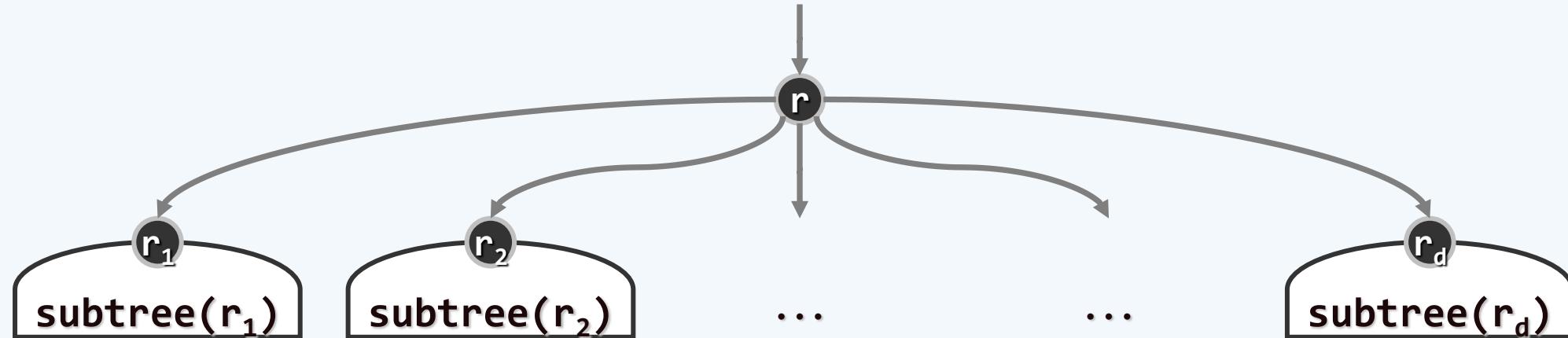
❖  $r_i$  称作  $r$  的 **孩子** ( child ) ,  $r_i$  之间互称 **兄弟** ( sibling )

$r$  为其 **父亲** ( parent ) ,  $d = \text{degree}(r)$  为  $r$  的 **(出)度** ( degree )

❖ 可归纳证明 :  $e = \sum_{r \in V} \text{degree}(r) = n - 1 = \Theta(n)$

故在衡量相关复杂度时 , 可以  $n$  作为参照

❖ 若指定  $T_i$  作为  $T$  的第  $i$  棵子树 ,  $r_i$  作为  $r$  的第  $i$  个孩子 , 则  $T$  称作 **有序树** ( ordered tree )



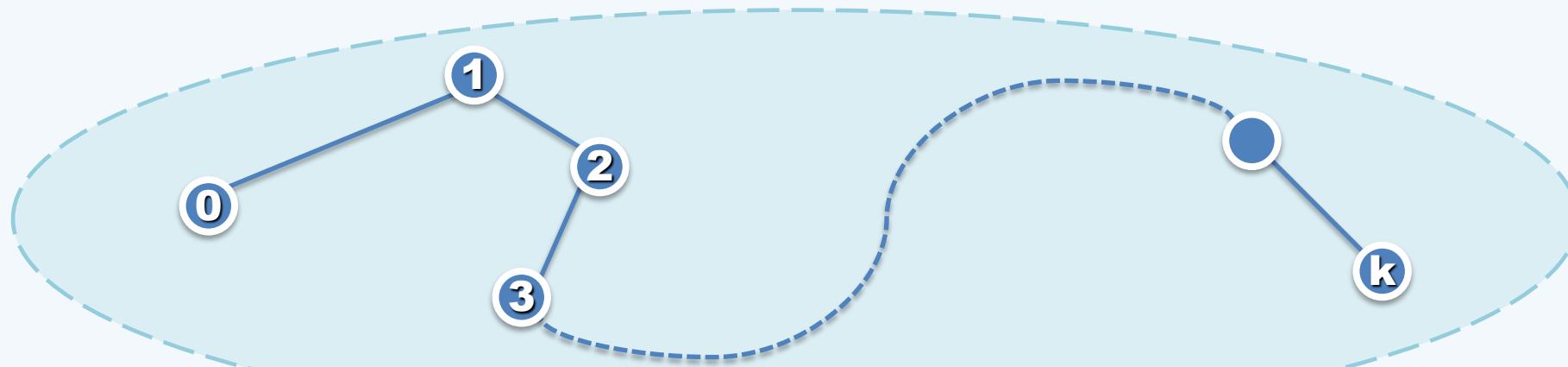
## 路径 + 环路

❖  $V$  中的  $k+1$  个节点，通过  $E$  中的  $k$  条边依次相联，构成一条 **路径** ( path ) // 亦称 **通路**

$$\pi = \{ (v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k) \}$$

❖ 路径 **长度** :  $|\pi| = \text{边数} = k$  // 早期文献，或以节点数为长度

❖ **环路** ( cycle/loop ) :  $v_k = v_0$



## 连通 + 无环

❖ 节点之间均有路径，称作 **连通图** ( connected )

不含环路，称作 **无环图** ( acyclic )

❖ 树：  
    **无环连通图**

**极小连通图**

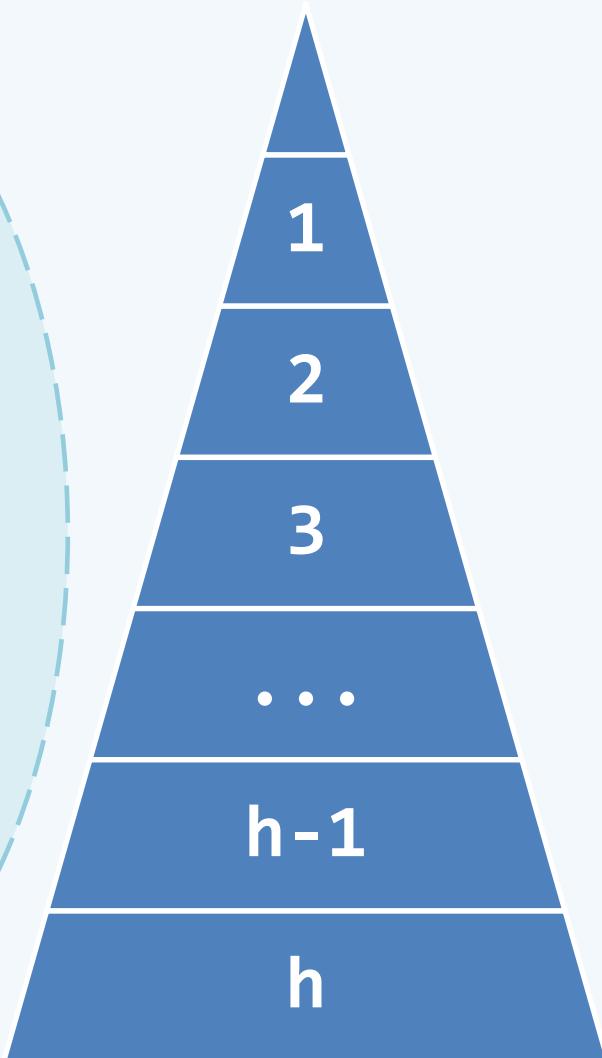
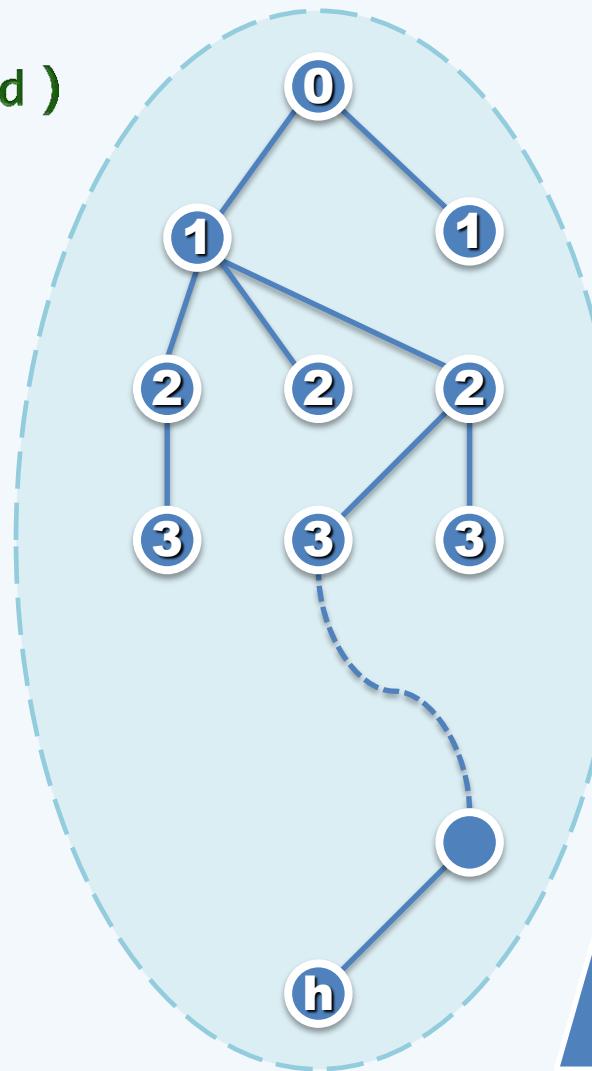
**极大无环图**

❖ 故：任一节点 $v$ 与根之间存在 **唯一** 路径

$$\text{path}(v, r) = \text{path}(v)$$

❖ 于是：以  $|\text{path}(v)|$  为指标

可对所有节点做 **等价类** 划分...



## 深度 + 层次

❖ 不致歧义时，路径、节点和子树可相互指代

$\text{path}(v) \sim v \sim \text{subtree}(v)$

❖  $v$  的 **深度** :  $\text{depth}(v) = |\text{path}(v)|$

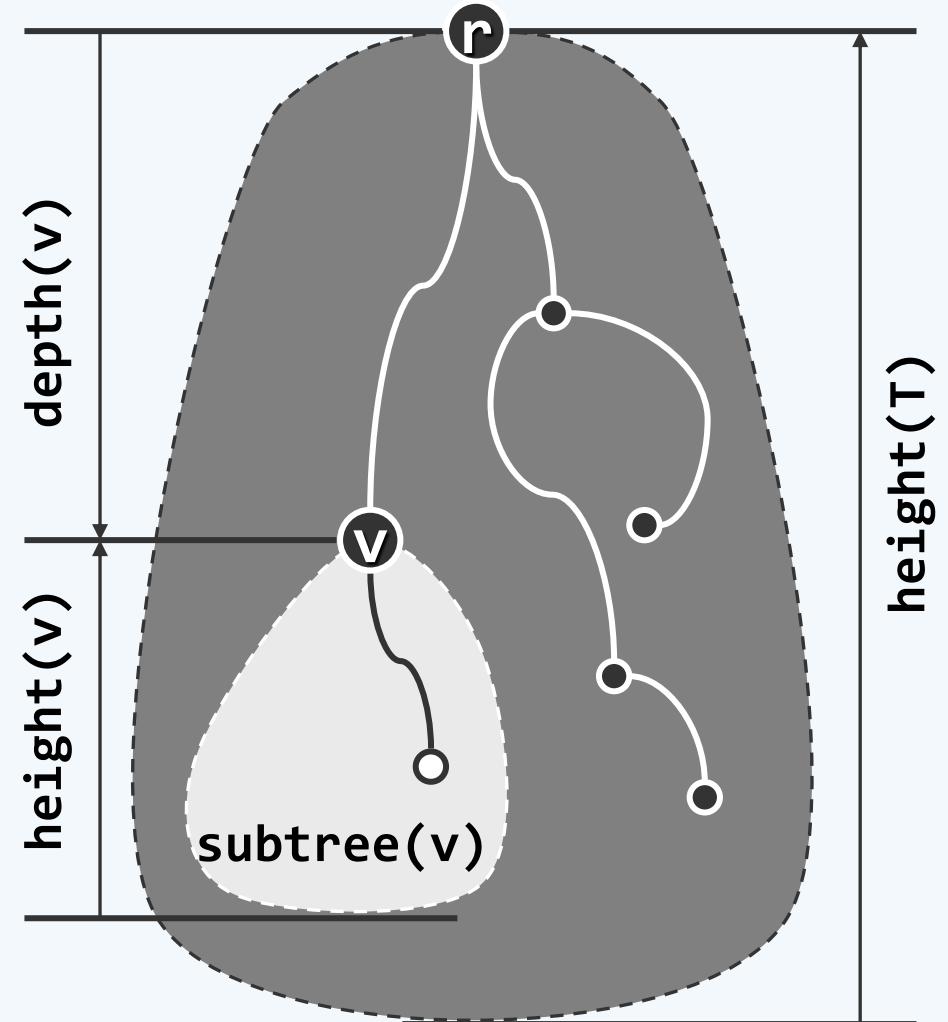
❖  $\text{path}(v)$  上节点，均为  $v$  的 **祖先** ( ancestor )

$v$  是它们的 **后代** ( descendant )

❖ 其中，除自身以外，是 **真** ( proper ) 祖先/后代

❖ **半线性**：在任一深度

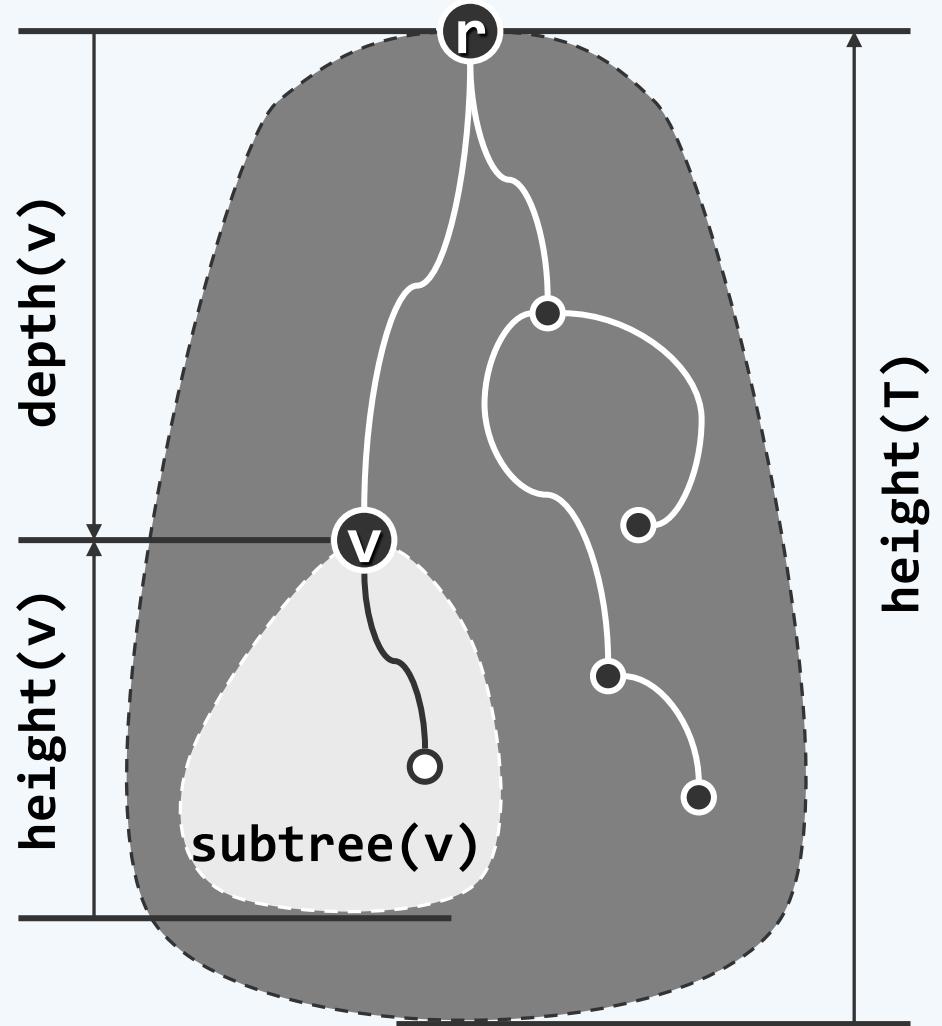
$v$  的祖先/后代若存在，则 **必然** / **未必** 唯一



## 深度 + 层次

- ❖ 根节点是所有节点的**公共祖先**，深度为0
- ❖ 没有后代的节点称作**叶子** (leaf)
- ❖ 所有叶子深度中的最大者称作(子)树(根)的**高度**  

$$\text{height}(v) = \text{height}(\text{subtree}(v))$$
- ❖ 特别地，空树的高度取作**-1**
- ❖  $\text{depth}(v) + \text{height}(v) \leq \text{height}(T)$   
 何时取等号？



## 5. 二叉树

树的表示

邓俊辉

deng@tsinghua.edu.cn

## 接口

节点	功能
<code>root()</code>	根节点
<code>parent()</code>	父节点
<code>firstChild()</code>	长子
<code>nextSibling()</code>	兄弟
<code>insert(i, e)</code>	将e作为第i个孩子插入
<code>remove(i)</code>	删除第i个孩子（及其后代）
<code>traverse()</code>	遍历

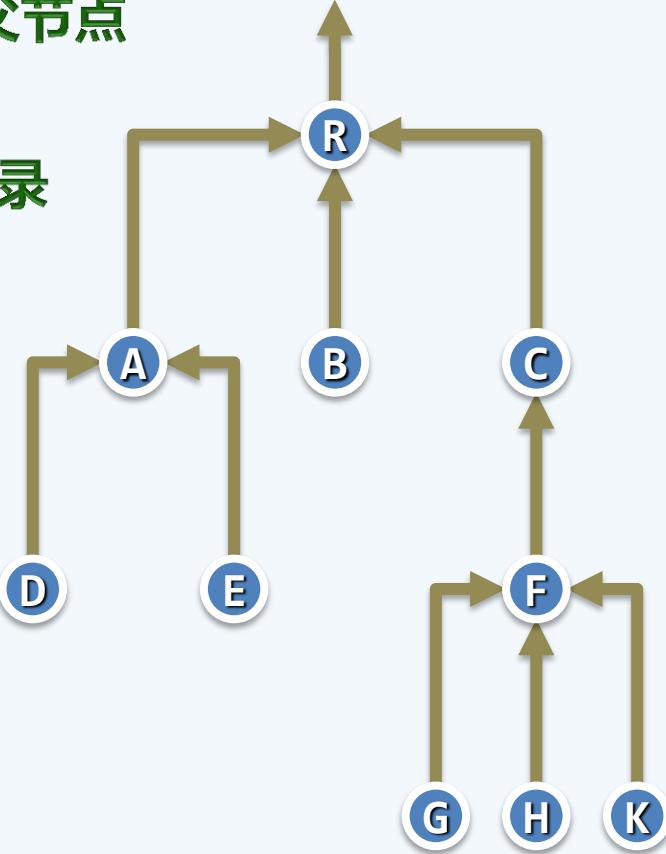
## 父节点

❖ 观察：除根外，任一节点**有且仅有**一个父节点

❖ 构思：将节点组织为序列，各节点分别记录

**data** 本身信息

**parent** 父节点的秩或位置



❖ 树根(0)也有“**虚构的**”父节点

$\text{parent}(0) = -1$  或

$\text{parent}(0) = \text{NULL}$

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

## 父节点

❖ 空间性能： $O(n)$

❖ 时间性能

☺ parent() :  $O(1)$

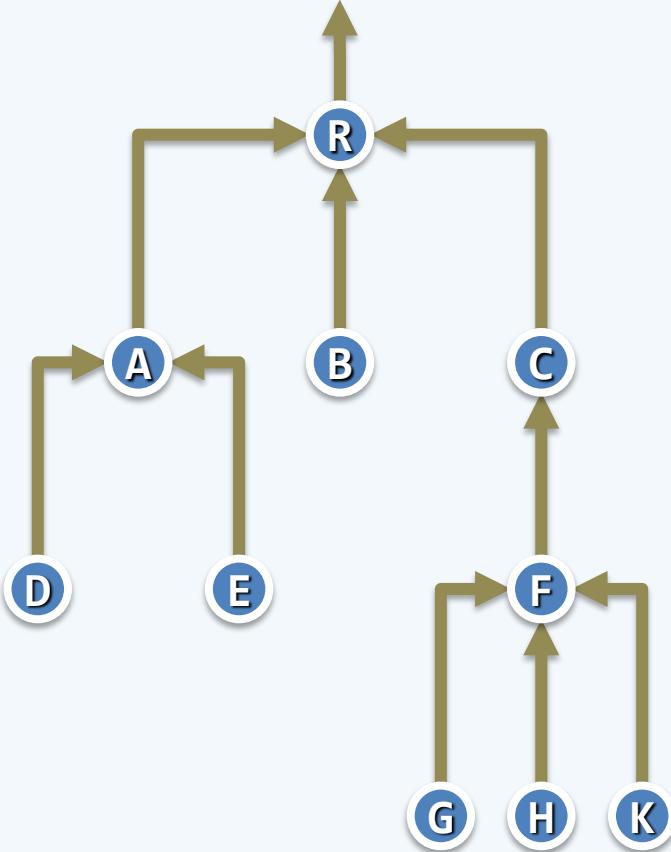
☺ root() :  $O(n)$  或  $O(1)$

☺ firstChild() :  $O(n)$

☺ nextSibling() :  $O(n)$

❖ 动态操作姑且不论，首先

如何加速对孩子、兄弟的查找？



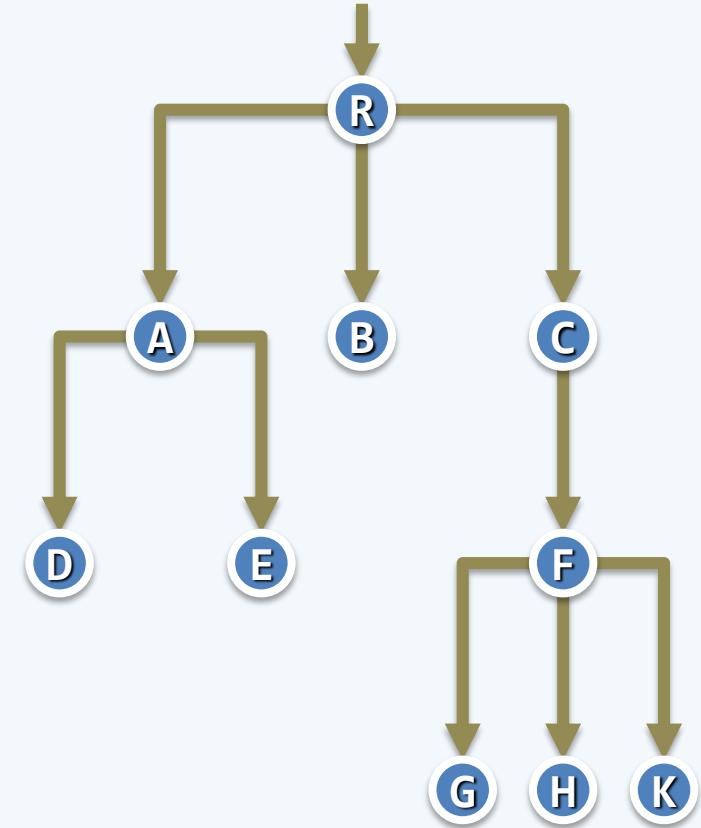
rank	data	parent
0	R	-1
1	A	0
2	B	0
3	C	0
4	D	1
5	E	1
6	F	3
7	G	6
8	H	6
9	K	6

## 孩子节点

❖ 同一节点的所有孩子，组织为一个序列

❖ 序列的长度，分别等于对应节点的度数

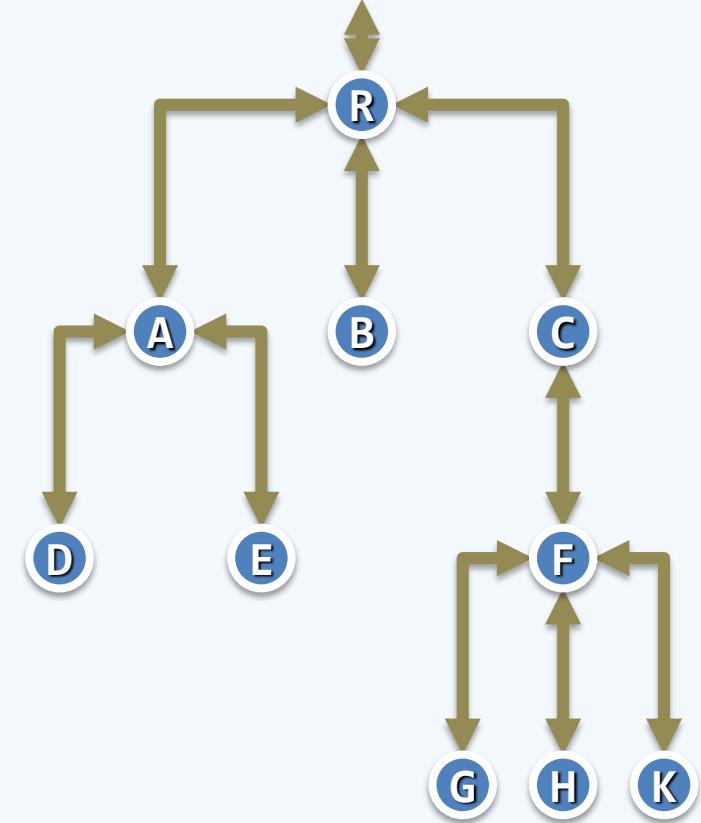
	data	children
0	A	→ [3] → [5] ^
1	B	^
2	C	→ [6] ^
3	D	^
4	R	→ [0] → [1] → [2] ^
5	E	^
6	F	→ [7] → [8] → [9] ^
7	G	^
8	H	^
9	K	^



❖ 现在，所有孩子都可很快找出，但parent()却很慢...

## 父节点 + 孩子节点

	data	parent	children
0	A	4	• → [3] [5] ^
1	B	4	^
2	C	4	• → [6] ^
3	D	0	^
4	R	-1	• → [0] [1] [2] ^
5	E	0	^
6	F	2	• → [7] [8] [9] ^
7	G	6	^
8	H	6	^
9	K	6	^



## 长子 + 兄弟

❖ 每个节点均设两个引用

纵 : `firstChild()`

横 : `nextSibling()`

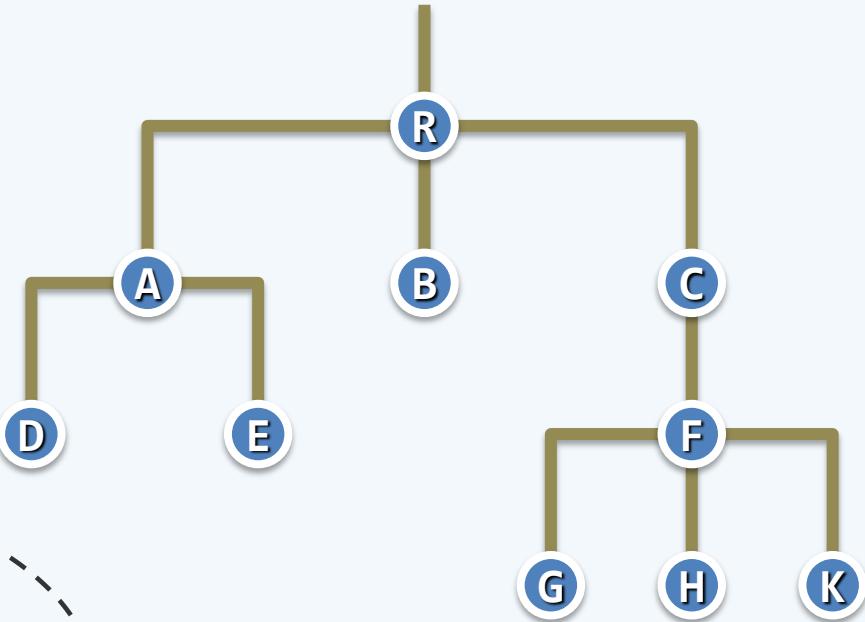
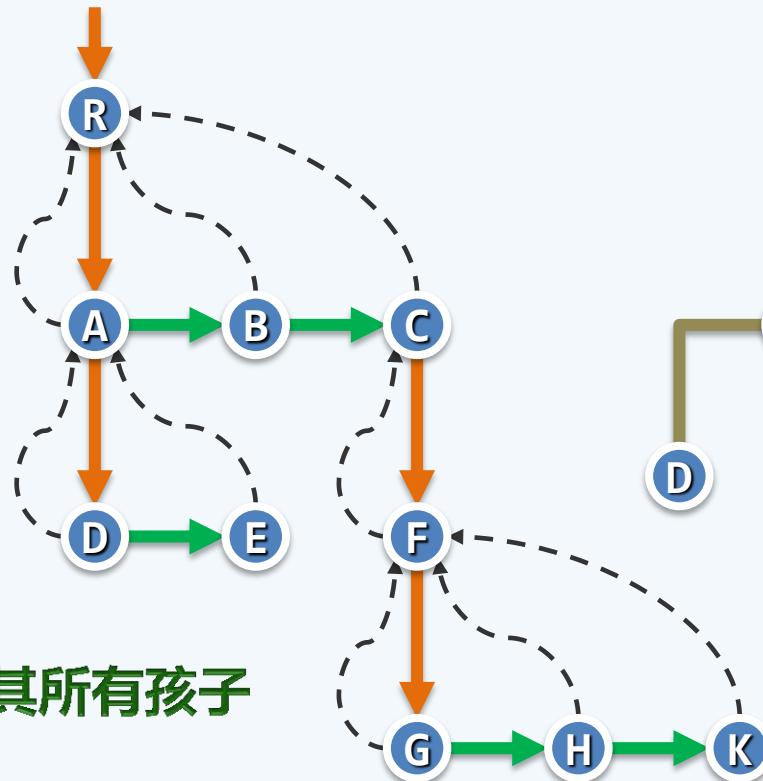
❖ 如此

对于度数为 $d$ 的节点

可在 $\mathcal{O}(d + 1)$ 时间内遍历其所有孩子

❖ 若再设置`parent`引用

则`parent()`接口也仅需 $\mathcal{O}(1)$ 时间



## 5. 二叉树

有根有序树 = 二叉树

宝玉终是不安本分之人，竟一味的随心所欲，因此又发了癖性，又特向秦钟悄说道：“咱们俩个人一样的年纪，况又是同窗，以后不必论叔侄，只论弟兄朋友就是了。”

邓俊辉

deng@tsinghua.edu.cn

## 二叉树

❖ 节点度数不超过2的树

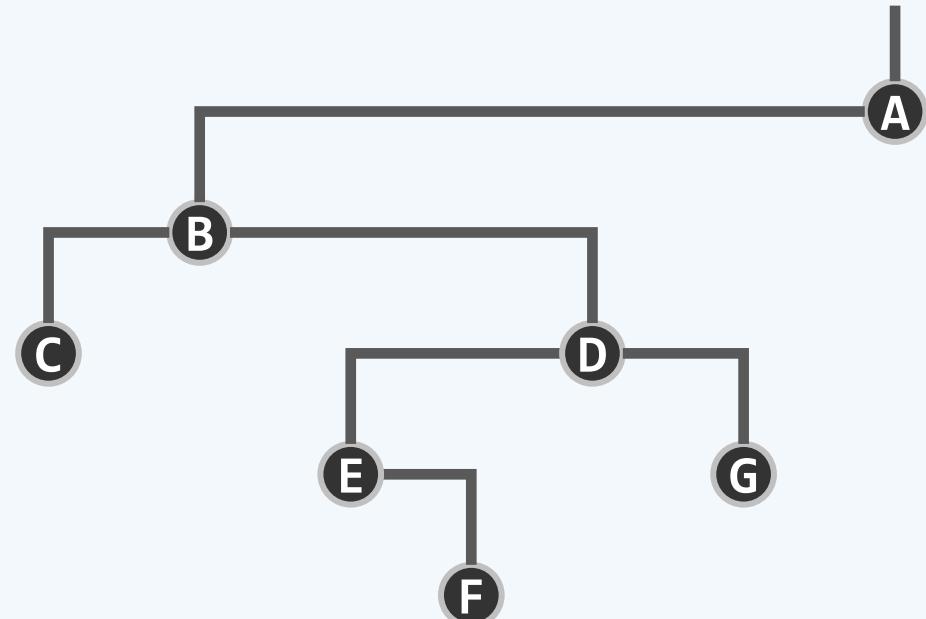
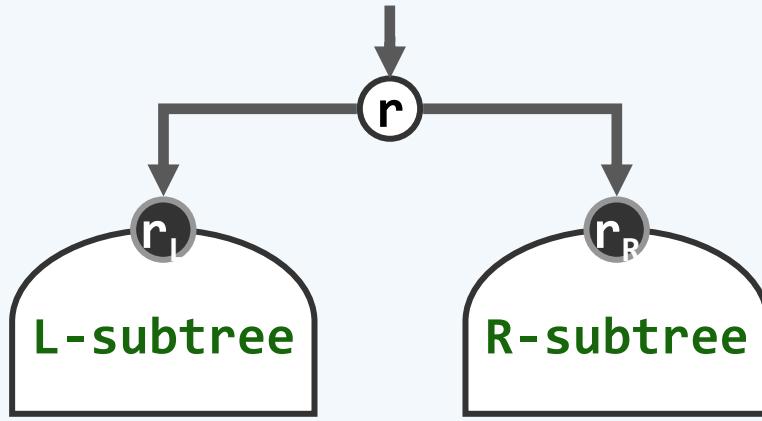
称作二叉树 (binary tree)

❖ 同一节点的孩子和子树，均以左、右区分

`lc() ~ lSubtree()`

`rc() ~ rSubtree()`

隐含有序



## 基数

◆ 深度为 $k$ 的节点，至多 $2^k$ 个

◆ 含 $n$ 个节点、高度为 $h$ 的二叉树中

$$h < n < 2^{h+1}$$

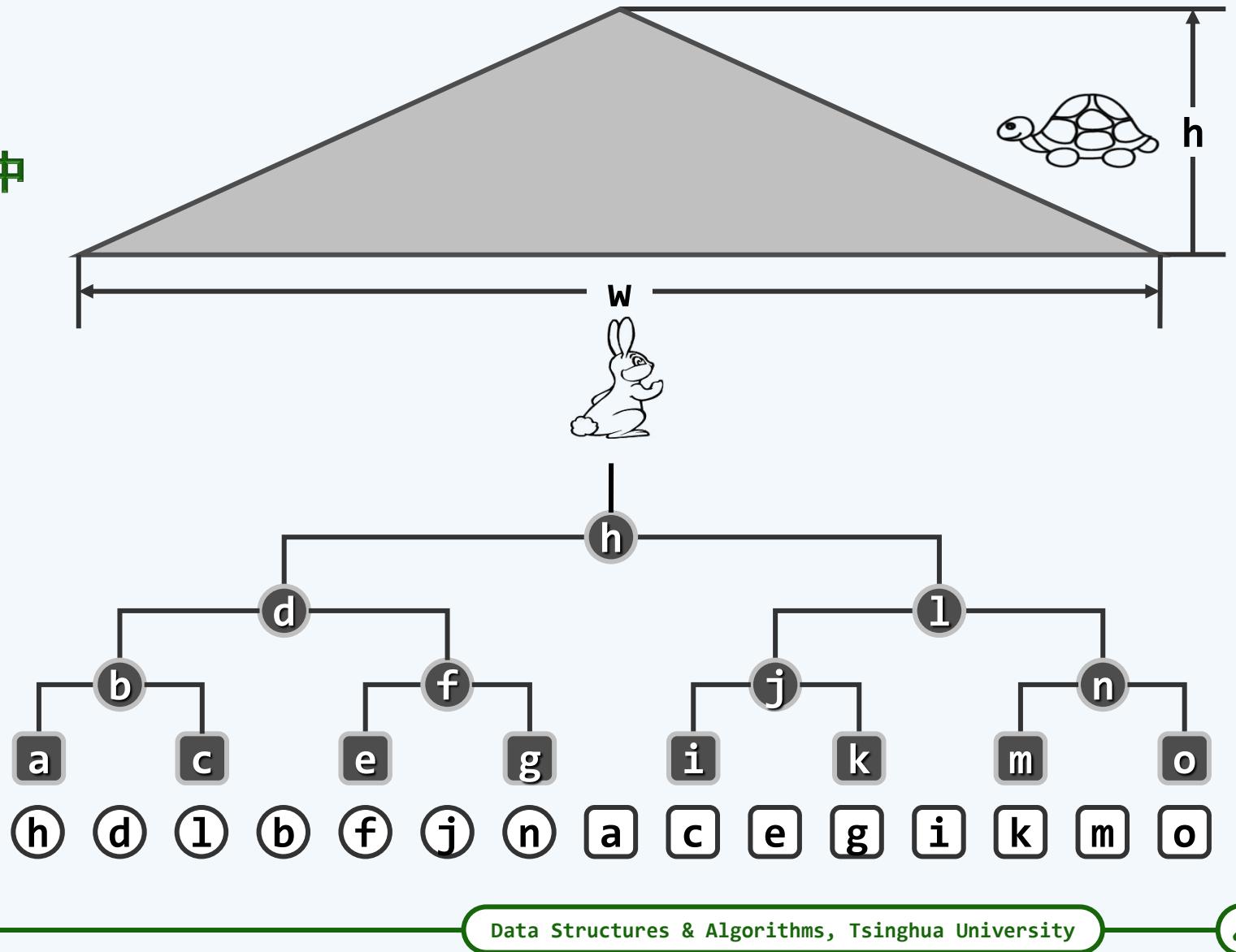
1)  $n = h + 1$ 时

退化为一条单链

2)  $n = 2^{h+1} - 1$ 时

即所谓满二叉树

( full binary tree )



## 基数

设度数为0、1和2的节点，各有 $n_0$ 、 $n_1$ 和 $n_2$ 个

边数  $e = n - 1 = n_1 + 2n_2$

1/2度节点各对应于1/2条入边

叶节点数  $n_0 = n_2 + 1$

$n_1$ 与 $n_0$ 无关

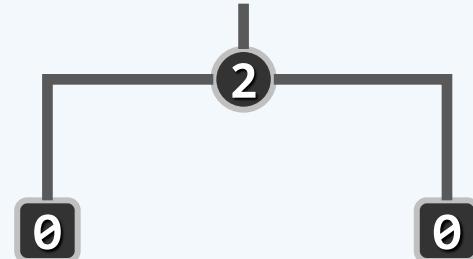
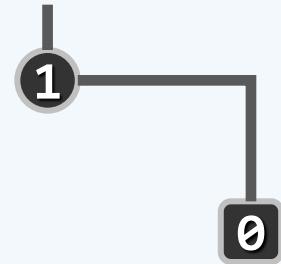
$h = 0$ 时， $1 = 0 + 1$ ；此后， $n_0$ 与 $n_2$ 同步递增

节点数  $n = n_0 + n_1 + n_2 = 1 + n_1 + 2n_2$

特别地，当 $n_1 = 0$ 时，有

$e = 2n_2$  和  $n_0 = n_2 + 1 = (n + 1)/2$

此时，节点度数均为偶数，不含单分支节点...



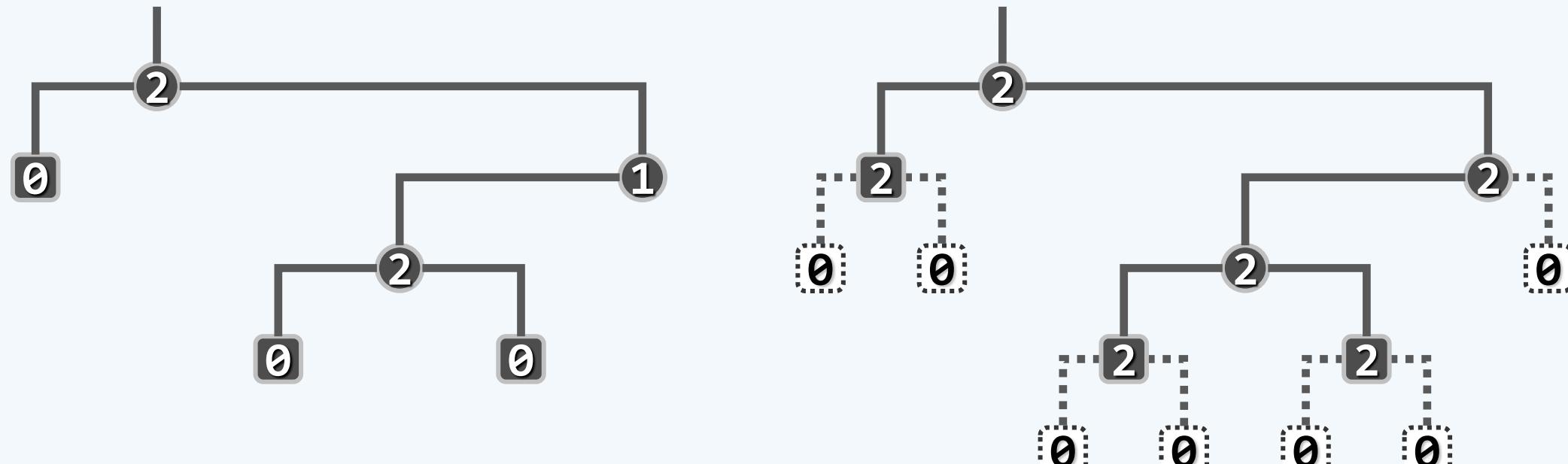
## 真二叉树

通过引入 $n_1 + 2n_0$ 个外部节点，可使原有节点度数统一为2

如此，即可将任一二叉树转化为真二叉树（proper binary tree）

验证：如此转换之后，全树自身的复杂度并未实质增加

对于红黑树之类的结构，真二叉树可以简化描述、理解、实现和分析

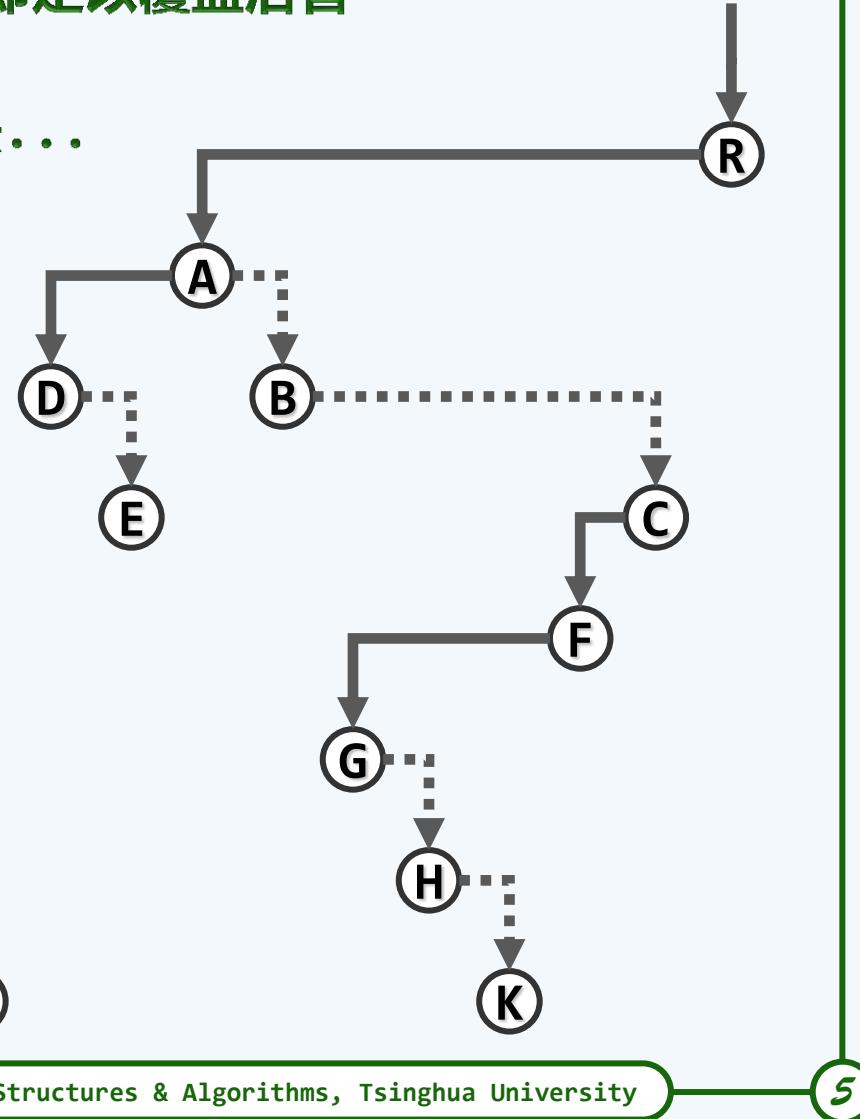
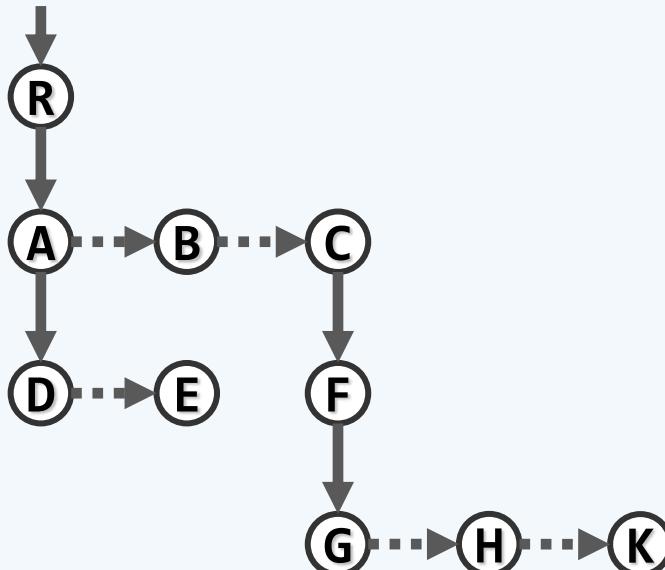
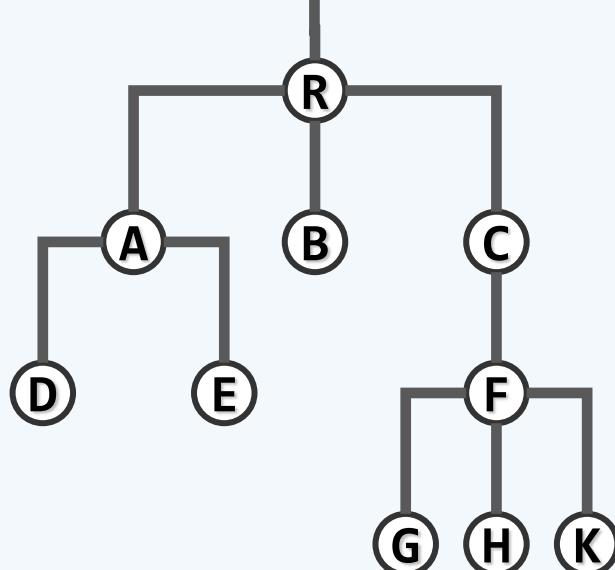


## 描述多叉树

❖ 二叉树是多叉树的特例，但在**有根且有序**时，其描述能力却足以覆盖后者

❖ 多叉树均可转化并表示为二叉树——回忆**长子-兄弟**表示法...

❖ **长子** ~ **左孩子**      `firstChild()` ~ `lc()`  
**兄弟** ~ **右孩子**      `nextSibling()` ~ `rc()`



## 5. 二叉树

二叉树实现

邓俊辉

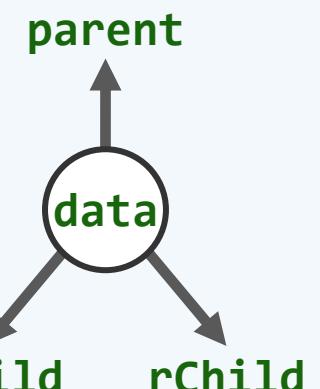
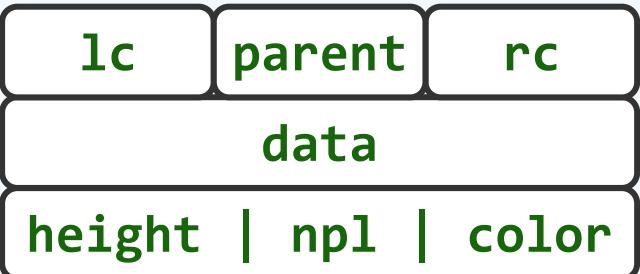
deng@tsinghua.edu.cn

## BinNode模板类

```

❖ #define BinNodePosi(T) BinNode<T>* //节点位置

❖ template <typename T> struct BinNode {
    BinNodePosi(T) parent, lc, rc; //父亲、孩子
    T data; int height; int size(); //高度、子树规模
    BinNodePosi(T) insertAsLC( T const & ); //作为左孩子插入新节点
    BinNodePosi(T) insertAsRC( T const & ); //作为右孩子插入新节点
    BinNodePosi(T) succ(); // (中序遍历意义下) 当前节点的直接后继
    template <typename VST> void travLevel( VST & ); //子树层次遍历
    template <typename VST> void travPre( VST & ); //子树先序遍历
    template <typename VST> void travIn( VST & ); //子树中序遍历
    template <typename VST> void travPost( VST & ); //子树后序遍历
};
```



## BinNode接口实现

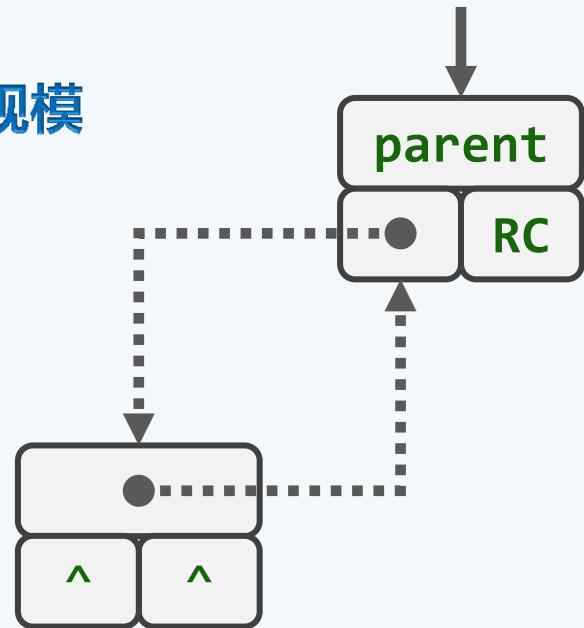
```

❖ template <typename T> BinNodePosi(T) BinNode<T>::insertAsLC( T const & e )
  { return lc = new BinNode( e, this ); }

❖ template <typename T> BinNodePosi(T) BinNode<T>::insertAsRC( T const & e )
  { return rc = new BinNode( e, this ); }

❖ template <typename T>
  int BinNode<T>::size() { //后代总数，亦即以其为根的子树的规模
    int s = 1; //计入本身
    if (lc) s += lc->size(); //递归计入左子树规模
    if (rc) s += rc->size(); //递归计入右子树规模
    return s;
} //O( n = |size| )

```



## BinTree模板类

```
❖ template <typename T> class BinTree {  
protected:  
    int _size; //规模  
    BinNodePosi(T) _root; //根节点  
    virtual int updateHeight( BinNodePosi(T) x ); //更新节点x的高度  
    void updateHeightAbove( BinNodePosi(T) x ); //更新x及祖先的高度  
public:  
    int size() const { return _size; } //规模  
    bool empty() const { return !_root; } //判空  
    BinNodePosi(T) root() const { return _root; } //树根  
    /* ... 子树接入、删除和分离接口 ... */  
    /* ... 遍历接口 ... */  
}
```

## 高度更新

- ❖ `#define stature(p) ( (p) ? (p)->height : -1 ) //节点高度——约定空树高度为-1`
- ❖ `template <typename T> //更新节点x高度，具体规则因树不同而异`  
`int BinTree<T>::updateHeight( BinNodePosi(T) x ) {`  
 `return x->height = 1 +`  
 `max( stature( x->lC ), stature( x->rC ) );`  
`} //此处采用常规二叉树规则，O(1)`
- ❖ `template <typename T> //更新v及其历代祖先的高度`  
`void BinTree<T>::updateHeightAbove( BinNodePosi(T) x ) {`  
 `while (x) //可优化：一旦高度未变，即可终止`  
 `{ updateHeight(x); x = x->parent; }`  
`} //O( n = depth(x) )`

## 节点插入

❖ template <typename T> BinNodePosi(T)

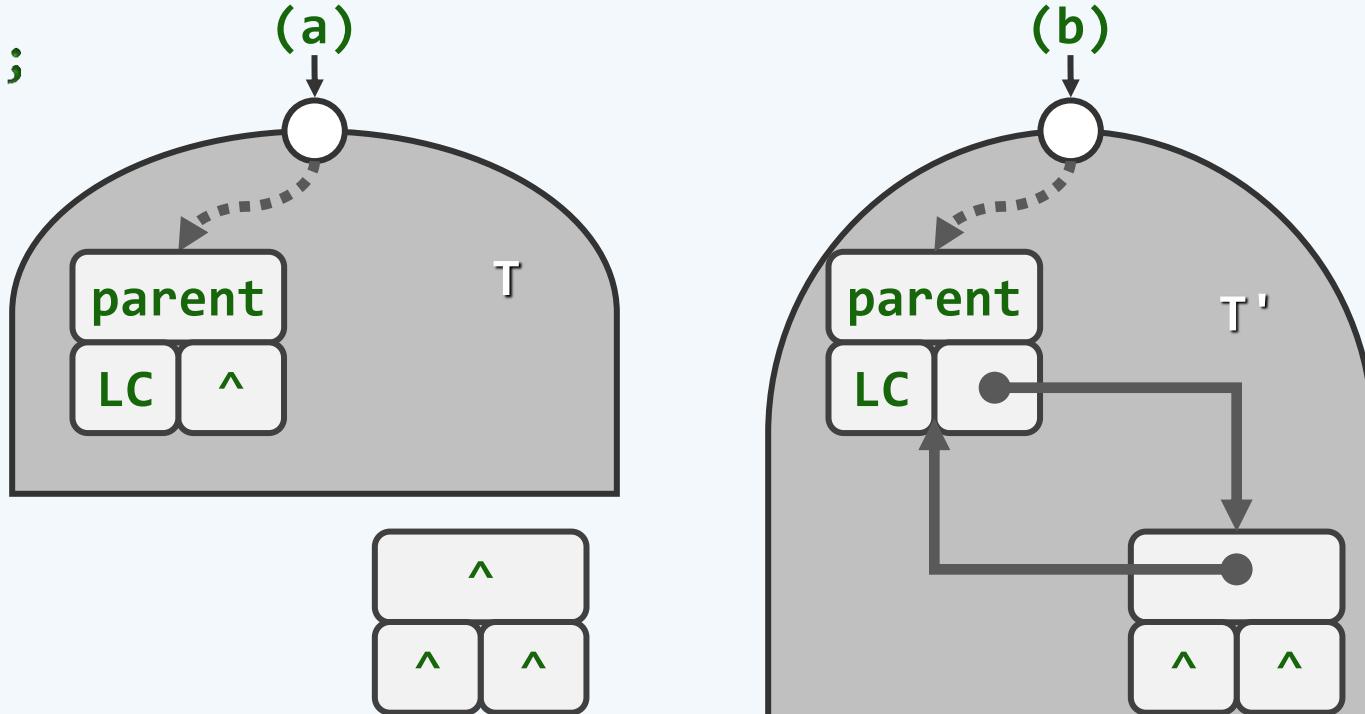
BinTree<T>::insertAsRC( BinNodePosi(T) x, T const & e ) { //insertAsLC()对称

\_size++; x->insertAsRC(e); //x祖先的高度可能增加，其余节点必然不变

updateHeightAbove(x);

return x->rc;

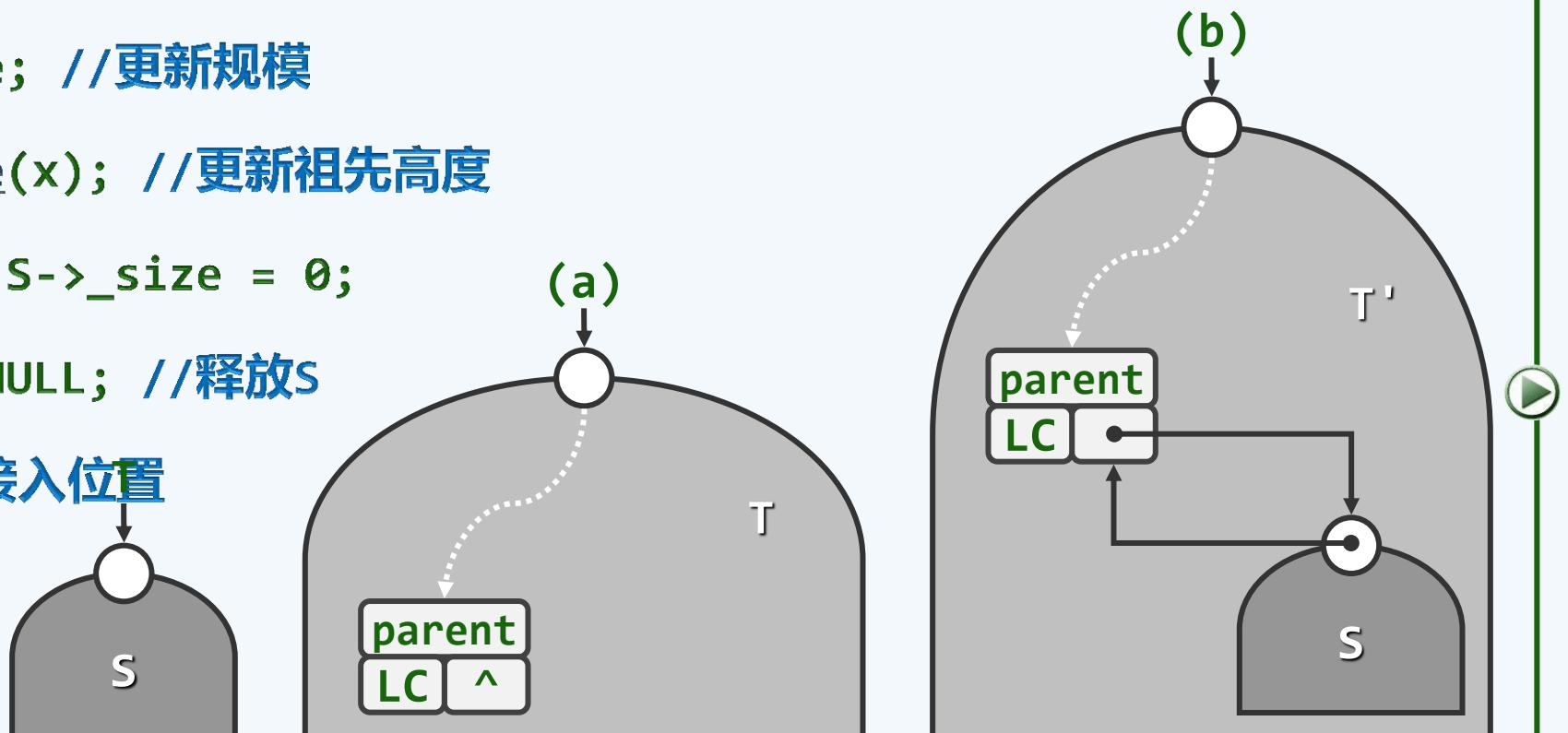
}



## 子树接入

❖ template <typename T>

```
BinNodePosi(T) BinTree<T>::attachAsRC( BinNodePosi(T) x, BinTree<T>* & S ) {
    if ( x->rc = S->_root ) x->rc->parent = x; //接入
    _size += S->_size; //更新规模
    updateHeightAbove(x); //更新祖先高度
    S->_root = NULL; S->_size = 0;
    release(S); S = NULL; //释放S
    return x; //返回接入位置
} //attachAsLC()对称
```



## 子树删除

❖ `template <typename T>`

```
int BinTree<T>::remove( BinNodePosi(T) x ) { //子树接入的逆过程  
    FromParentTo( * x ) = NULL; //切断来自父节点的指针  
    updateHeightAbove( x->parent ); //更新祖先高度 ( 其余节点亦不变 )  
    int n = removeAt(x); _size -= n; //递归删除x及其后代，更新规模  
    return n; //返回被删除节点总数  
}
```

❖ `template <typename T> static int removeAt( BinNodePosi(T) x ) {`

```
if ( ! x ) return 0; //终止于空子树，否则左、右递归  
int n = 1 + removeAt( x->lC ) + removeAt( x->rC );  
release(x->data); release(x); return n; //释放被摘除节点，并返回被删除节点总数  
}
```

## 子树分离

- ❖ 过程与以上的子树删除操作`BinTree<T>::remove()`基本一致
- ❖ 不同之处在于，需对分离出来的子树重新封装，并返回给上层调用者
- ❖ 

```
template <typename T> BinTree<T>* BinTree<T>::secede( BinNodePosi(T) x ) {
    FromParentTo( * x ) = NULL; //切断来自父节点的指针
    updateHeightAbove( x->parent ); //更新原树中所有祖先的高度
    // 以下对分离出的子树做封装
    BinTree<T> * S = new BinTree<T>; //创建空树
    S->_root = x; x->parent = NULL; //新树以x为根
    S->_size = x->size(); _size -= S->_size; //更新规模
    return S; //返回封装后的子树
}
```

## 5. 二叉树

先序遍历

遍历

真君曰：“昔吕洞宾居庐山而成仙，鬼谷子居云梦而得道，今或无此吉地么？”

璞曰：“有，但当遍历耳。”

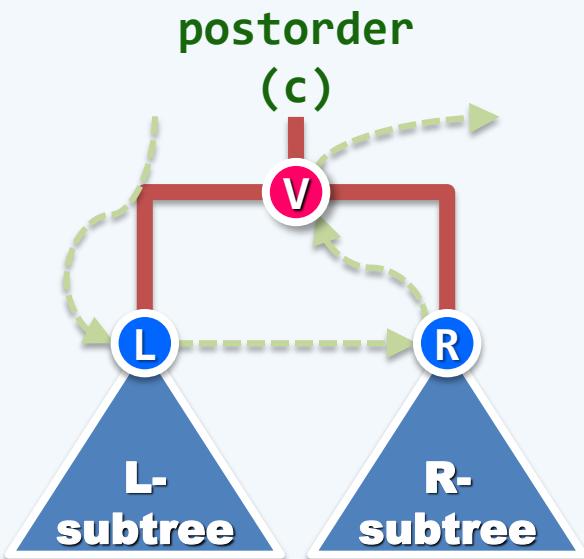
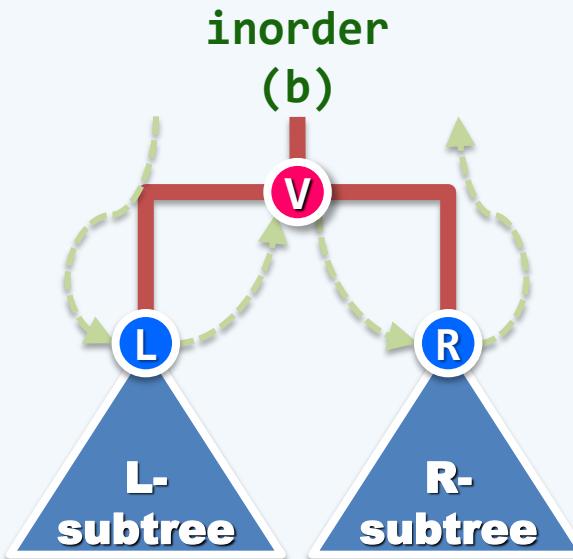
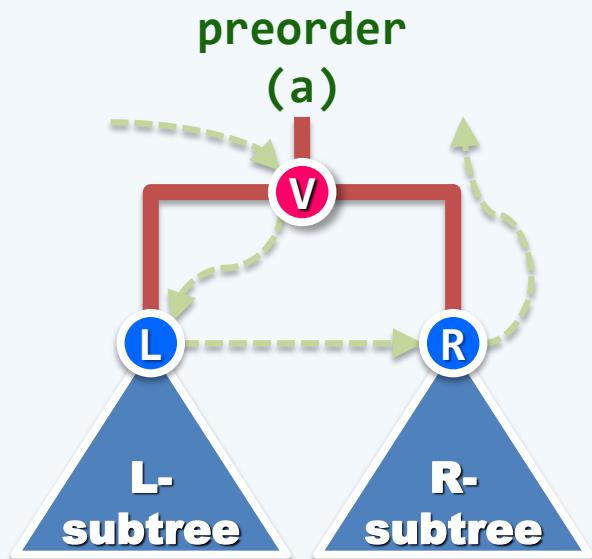
邓俊辉

deng@tsinghua.edu.cn

❖ 按照**某种次序**访问树中各节点，每个节点被访问**恰好一次**： $T = L \cup V \cup R$

❖ 先序                  中序                  后序                  层次(广度)

$V | L | R$              $L | V | R$              $L | R | V$             自上而下，先左后右



❖ 遍历结果 ~ 遍历过程 ~ 遍历次序 ~ 遍历策略

## 5. 二叉树

先序遍历

算法A

邓俊辉

deng@tsinghua.edu.cn

# 递归

```

❖ template <typename T, typename VST>

void traverse( BinNodePosi(T) x, VST & visit ) {

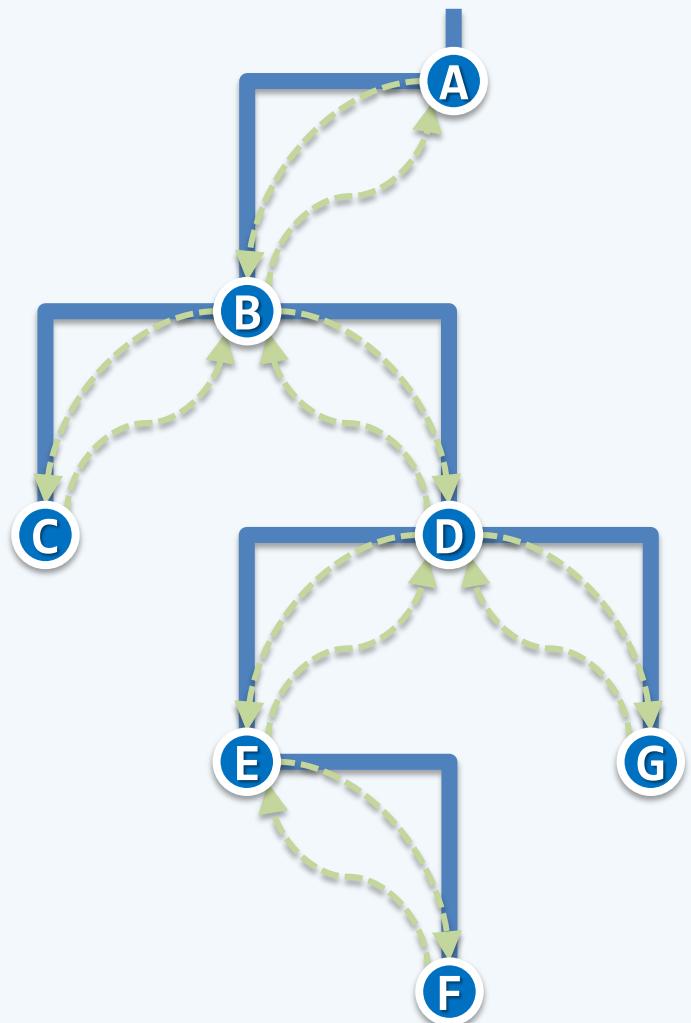
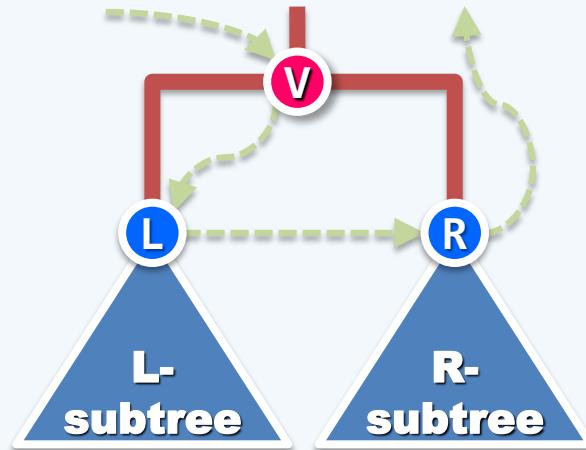
    if ( ! x ) return;

    visit( x->data );

    traverse( x->lc, visit );
    traverse( x->rc, visit );
}

} //T(n) = O(1) + T(a) + T(n - a - 1) = O(n)

```



❖ 先序输出文件树结构 : `c:\> tree.com c:\windows`

❖ 挑战：不依赖递归机制，能否实现先序遍历？如何实现？效率如何？

## 思路

❖ 先序遍历任一二叉树T的过程，无非是

先访问根节点  $r$

再先后递归地遍历  $TL$  和  $TR$

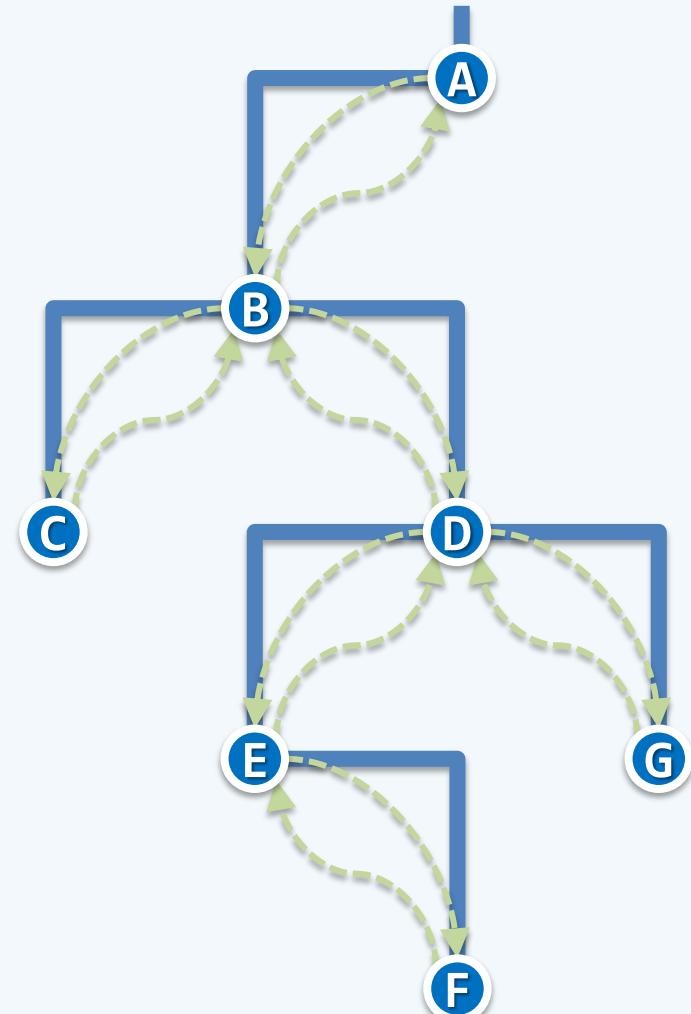
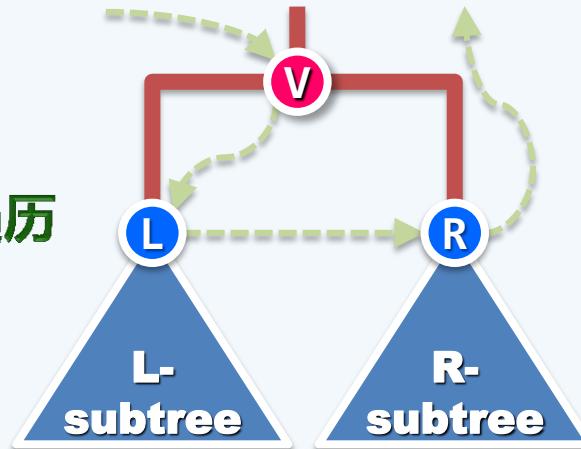
❖ 递归实现中，对左、右子树的递归遍历

都类似于尾递归

故不难直接消除

❖ 思路：

二分递归  $\rightarrow$  迭代 + 单递归  $\rightarrow$  迭代 + 栈



## 算法

```

❖ template <typename T, typename VST>

void travPre_I1( BinNodePosi(T) x, VST & visit ) {

Stack < BinNodePosi(T) > S; //辅助栈

if (x) S.push( x ); //根节点入栈

while ( ! S.empty() ) { //在栈变空之前反复循环

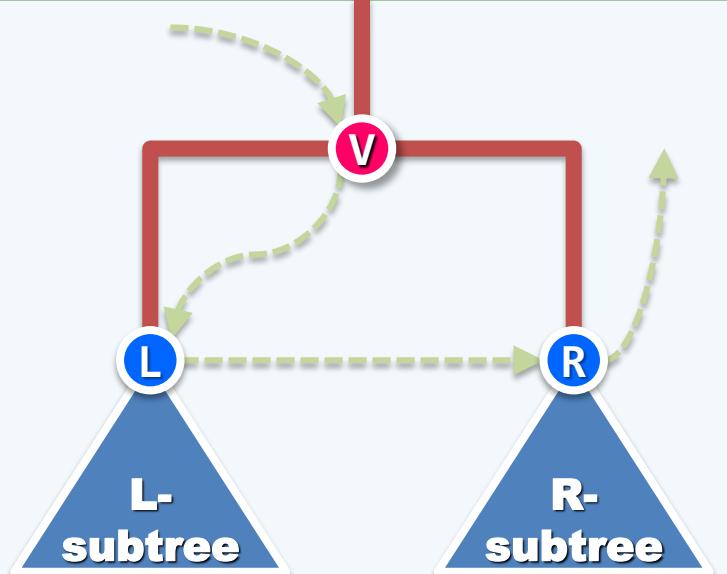
    x = S.pop(); visit( x->data ); //弹出并访问当前节点

    if ( HasRChild( *x ) ) S.push( x->rc ); //右孩子先入后出

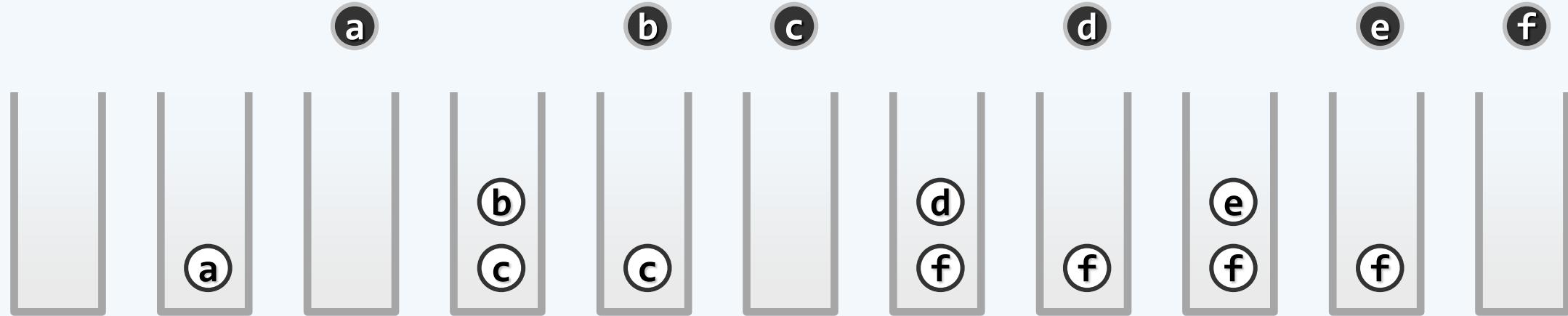
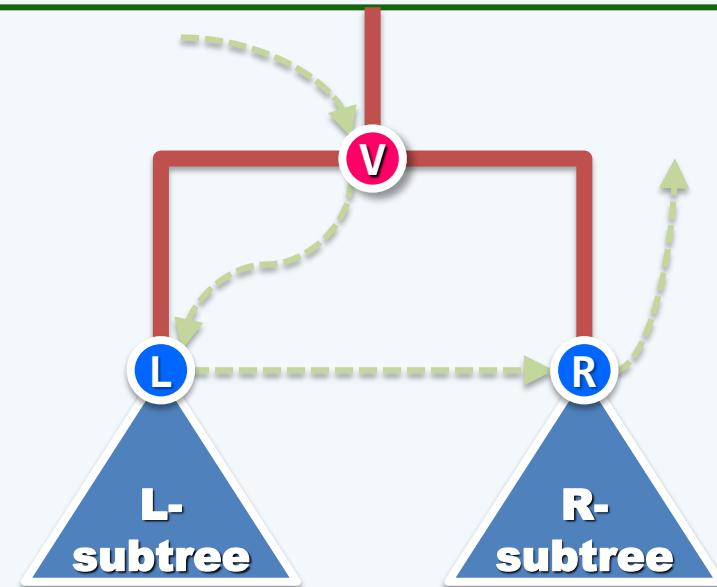
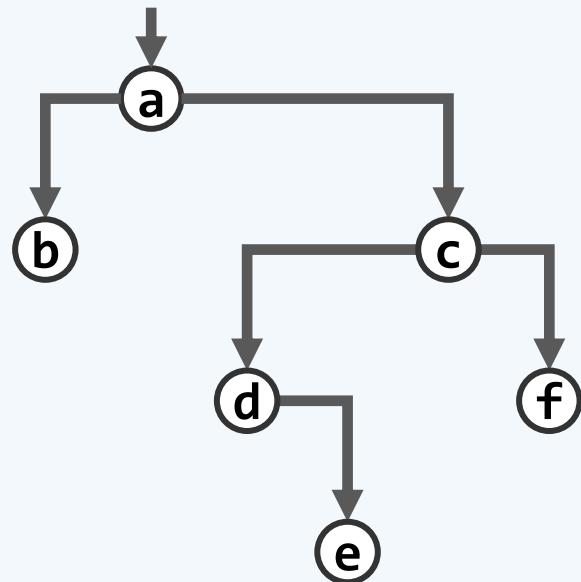
    if ( HasLChild( *x ) ) S.push( x->lc ); //左孩子后入先出

} //体会以上两句的次序

```



## 实例



## 正确性

### 无遗落 :

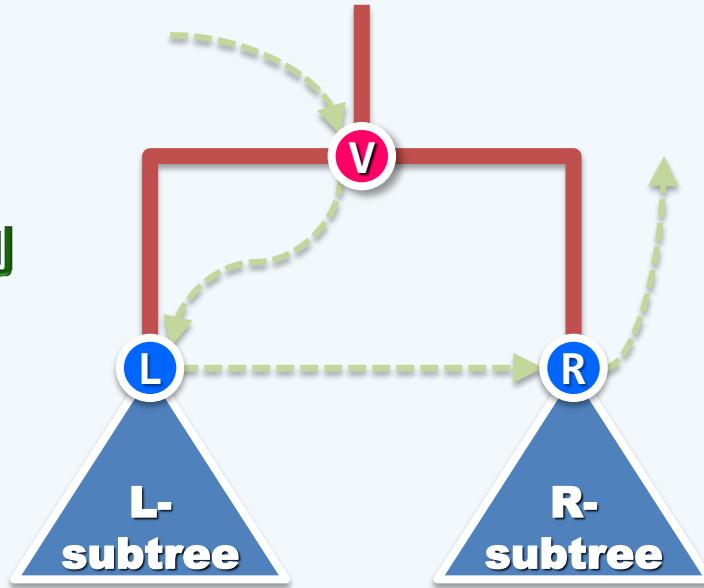
- 每个节点都会被访问到
- 归纳假设：若深度为 $d$ 的节点都能被正确访问到，则深度为 $d+1$ 的也是

### 根先 :

- 对于任一子树，根被访问后才会访问其它节点
- 只需注意到：若 $u$ 是 $v$ 的真祖先，则 $u$ 必先于 $v$ 被访问到

### 左先右后 :

- 同一节点的左子树，先于右子树被访问



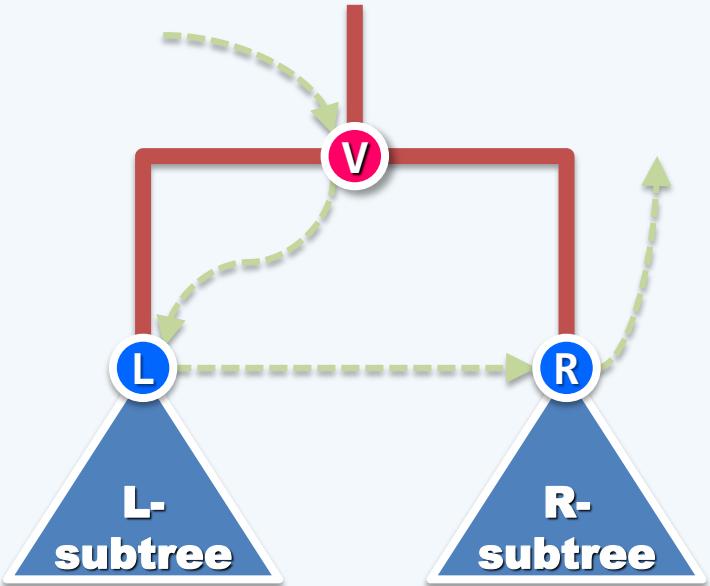
## 效率

❖  $\Theta(n)$

- 每步迭代，都有一个节点出栈并被访问
- 每个节点入/出栈一次且仅一次
- 每步迭代只需 $\Theta(1)$ 时间

❖ 以上消除尾递归的思路不易推广

需要另寻他法...



## 5. 二叉树

一桩事情的真相与奥妙，通常并不藏在最深的地方，有时就在表面。只不过，一般人视若无睹。要想成为一个好的算命先生，首先就必须学会观察...

先序遍历  
观察

警幻冷笑道：“贵省女子固多，不过择其紧要者录之。

下边二厨则又次之。余者庸常之辈，则无册可录矣。”

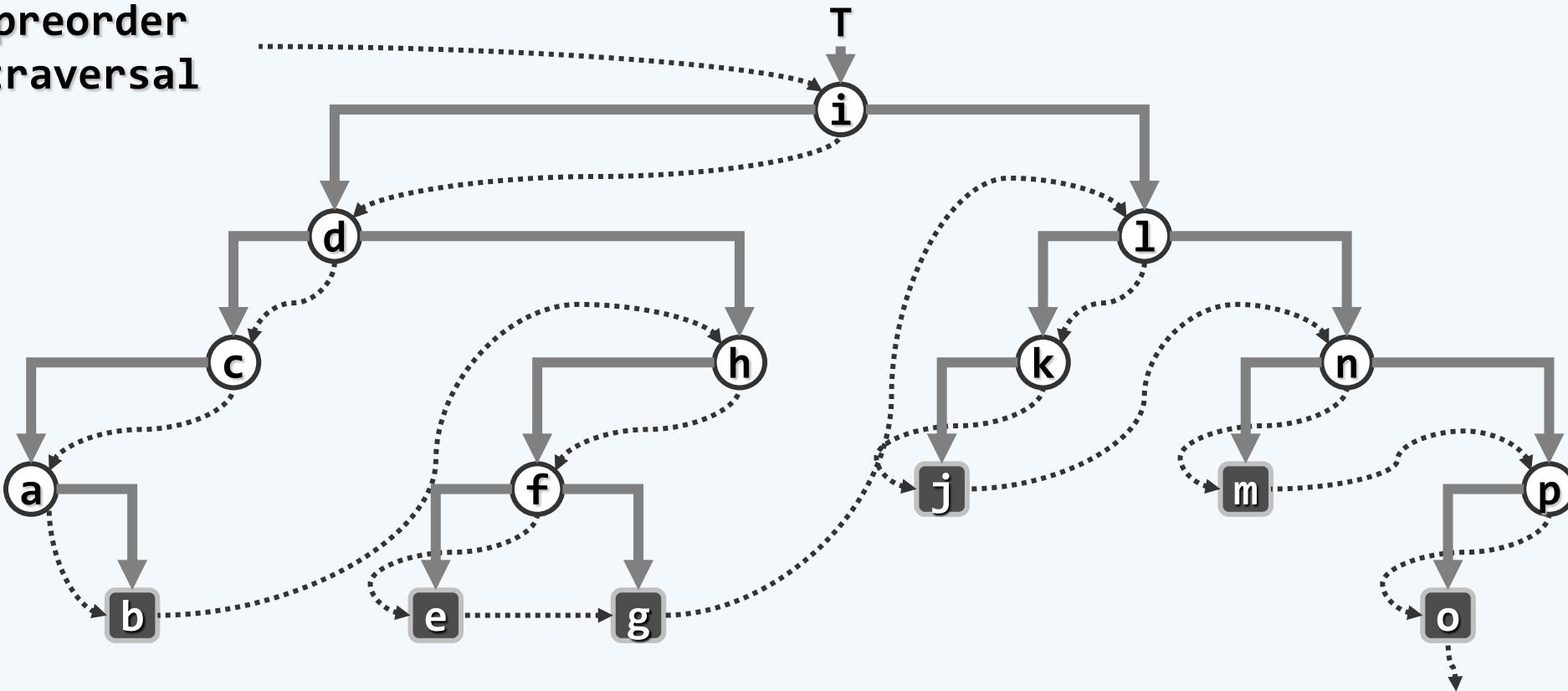
宝玉听说，再看下首二厨上，果然写着“金陵十二钗副

册”，又一个写着“金陵十二钗又副册”。

邓俊辉

deng@tsinghua.edu.cn

观察

preorder  
traversal

→ **i** → **d** → **c** → **a** → **b** → **h** → **f** → **e** → **g** → **l** → **k** → **j** → **n** → **m** → **p** → **o** →

## 藤缠树

沿着左侧分支

各节点与其右孩子（可能为空）一一对应

从宏观上，整个遍历过程可划分为

自上而下对左侧分支的访问，及随后

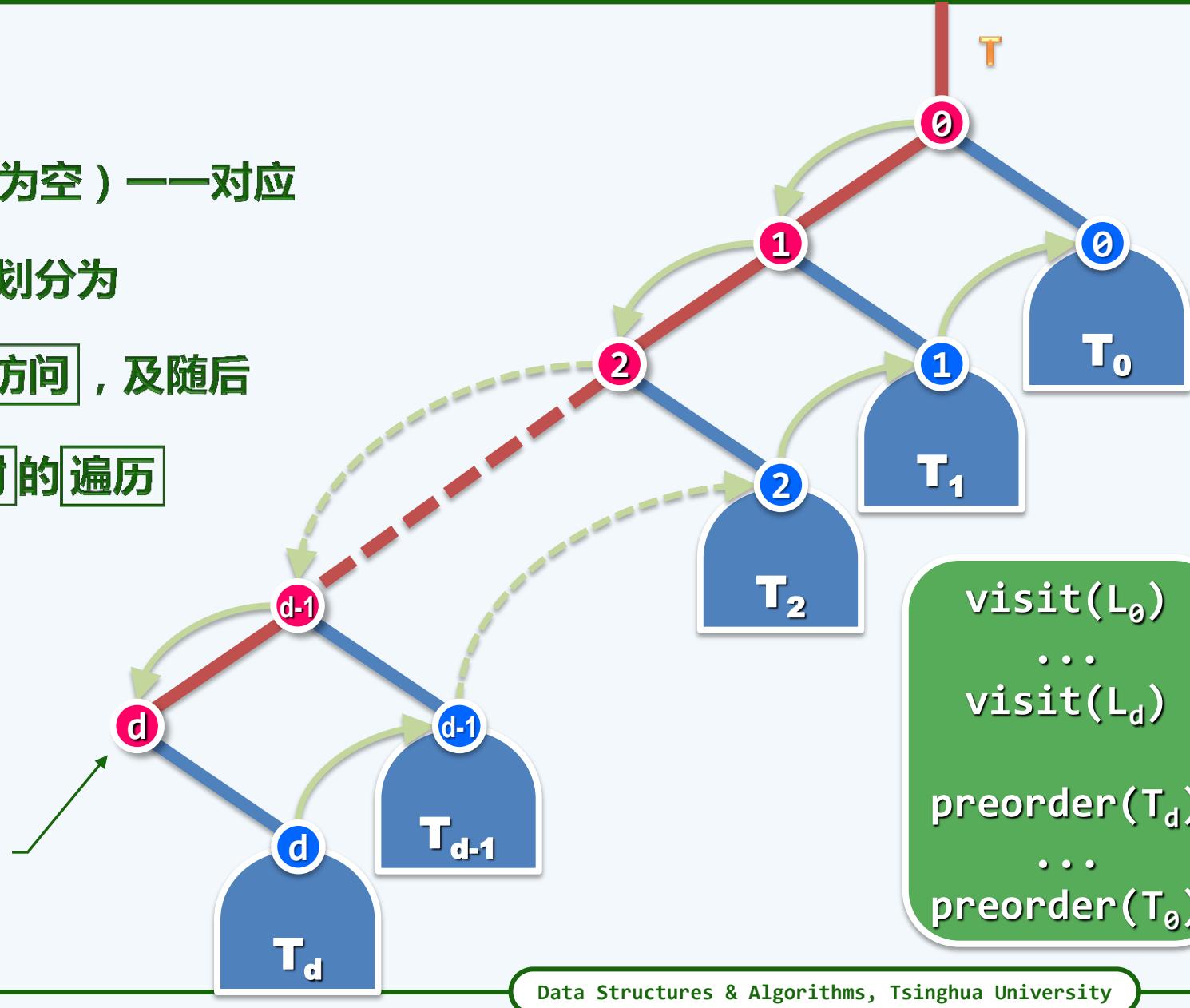
自下而上对一系列右子树的遍历

不同右子树的遍历

相互独立

自成一个子任务

endpoint of the vine



## 5. 二叉树

先序遍历

算法B

邓俊辉

deng@tsinghua.edu.cn

❖ template <typename T, typename VST>

static void visitAlongLeftBranch

( BinNodePosi(T) x, VST & visit, Stack < BinNodePosi(T) > & S ) { //分摊 $O(1)$

while ( x ) { //反复地

visit( x->data ); //访问当前节点

S.push( x->rc ); //右孩子（右子树）入栈（将来逆序出栈）

x = x->lc; //沿左侧链下行

} //只有右孩子、NULL可能入栈—增加判断以剔除后者，是否值得？

}

❖ template <typename T, typename VST>

```
void travPre_I2( BinNodePosi(T) x, VST & visit ) {
```

```
Stack < BinNodePosi(T) > S; //辅助栈
```

```
while ( true ) { //以 (右)子树为单位，逐批访问节点
```

```
visitAlongLeftBranch( x, visit, S ); //访问子树x的左侧链，右子树入栈缓冲
```

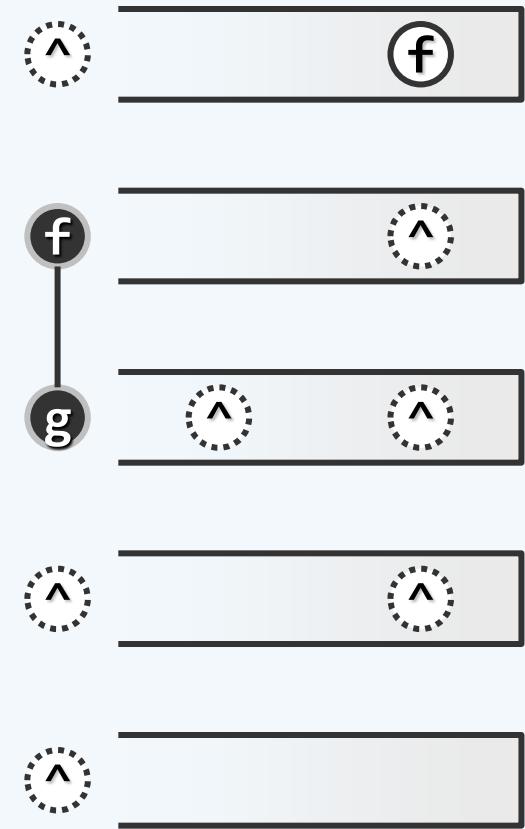
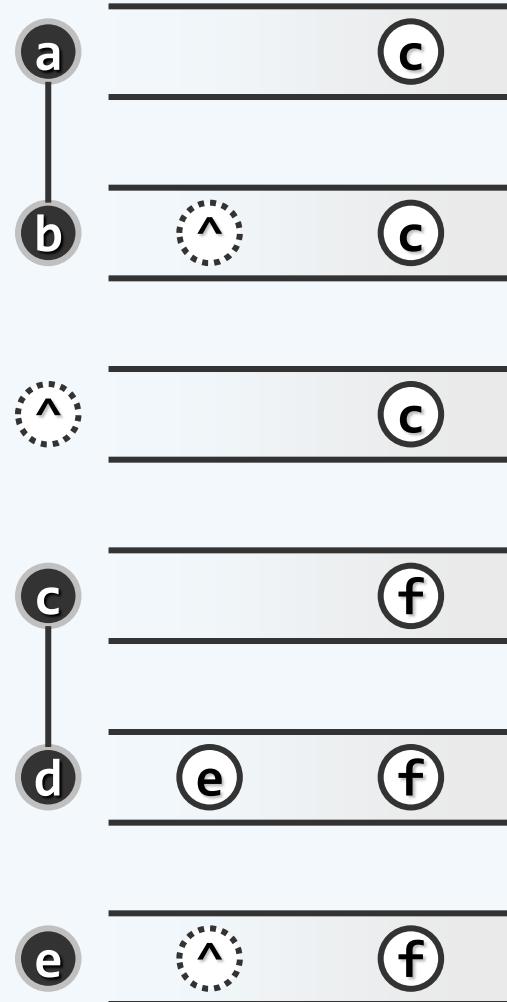
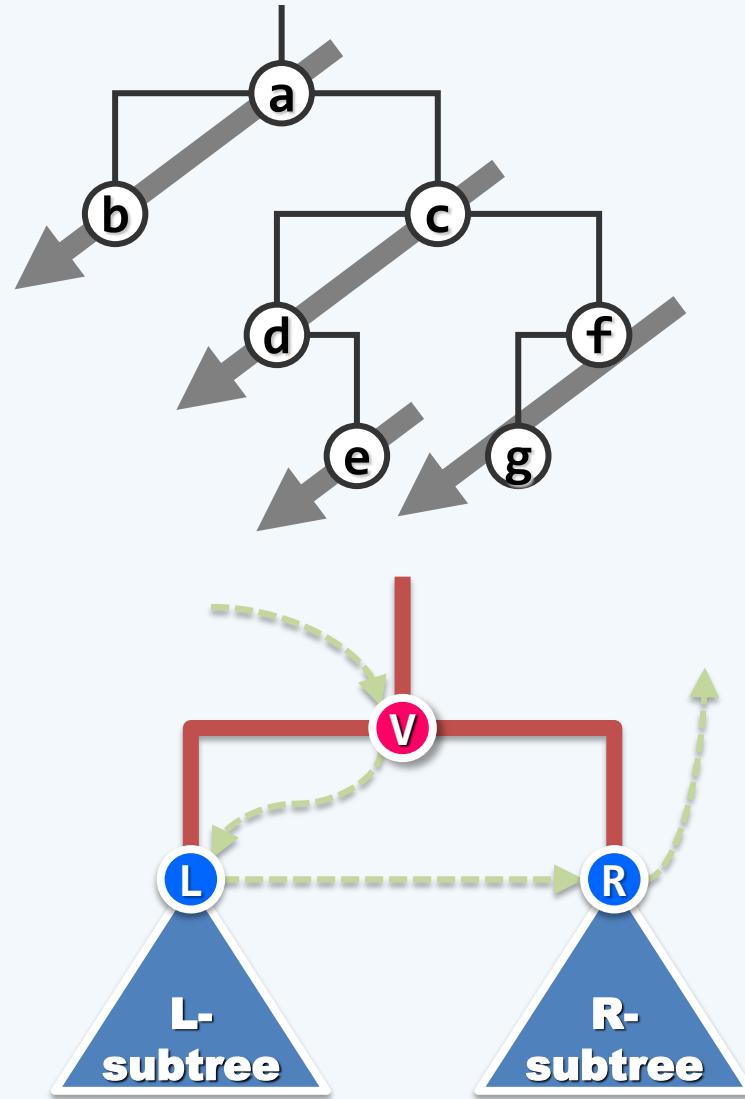
```
if ( S.empty() ) break; //栈空即退出
```

```
x = S.pop(); //弹出下一子树的根
```

```
} // #pop = #push = #visit = O(n) = 分摊 O(1)
```

```
}
```

## 实例



## 5. 二叉树

中序遍历

观察

山中只见藤缠树  
世上哪见树缠藤  
青藤若是不缠树  
枉过一春又一春

邓俊辉

deng@tsinghua.edu.cn

# 递归

❖ `template <typename T, typename VST>`

```
void traverse( BinNodePosi(T) x, VST & visit ) {
```

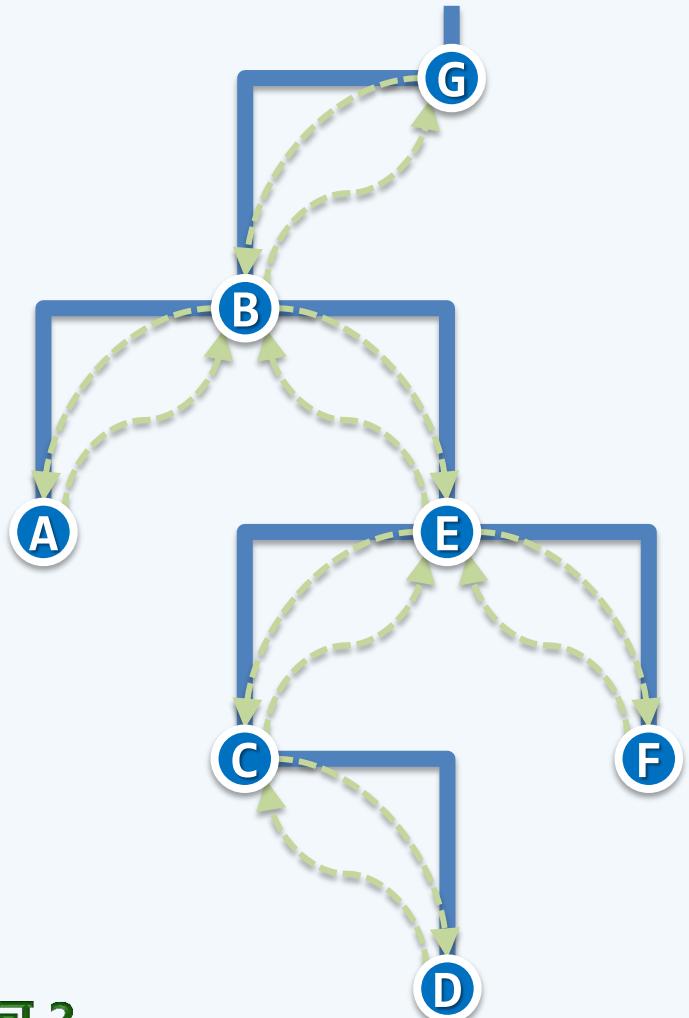
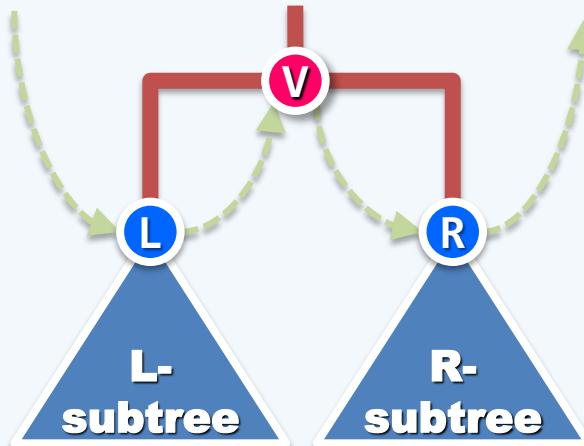
```
    if ( !x ) return;
```

```
    traverse( x->lc, visit );
```

```
    visit( x->data );
```

```
    traverse( x->rc, visit );
```

```
} //T(n) = T(a) + O(1) + T(n-a-1) = O(n)
```



❖ 中序输出文件树结构 : printBinTree()

❖ 挑战 : 不依赖递归机制 , 能否实现中序遍历 ? 如何实现 ? 效率如何 ?

## 难点

❖ 难度在于

尽管右子树的递归遍历是尾递归，但左子树却严格地不是

❖ 解决方法

找到第一个被访问的节点 //仿照迭代的先序遍历算法

将其祖先用栈保存 //按照被访问过程的逆序

❖ 这样，原问题就被分解为

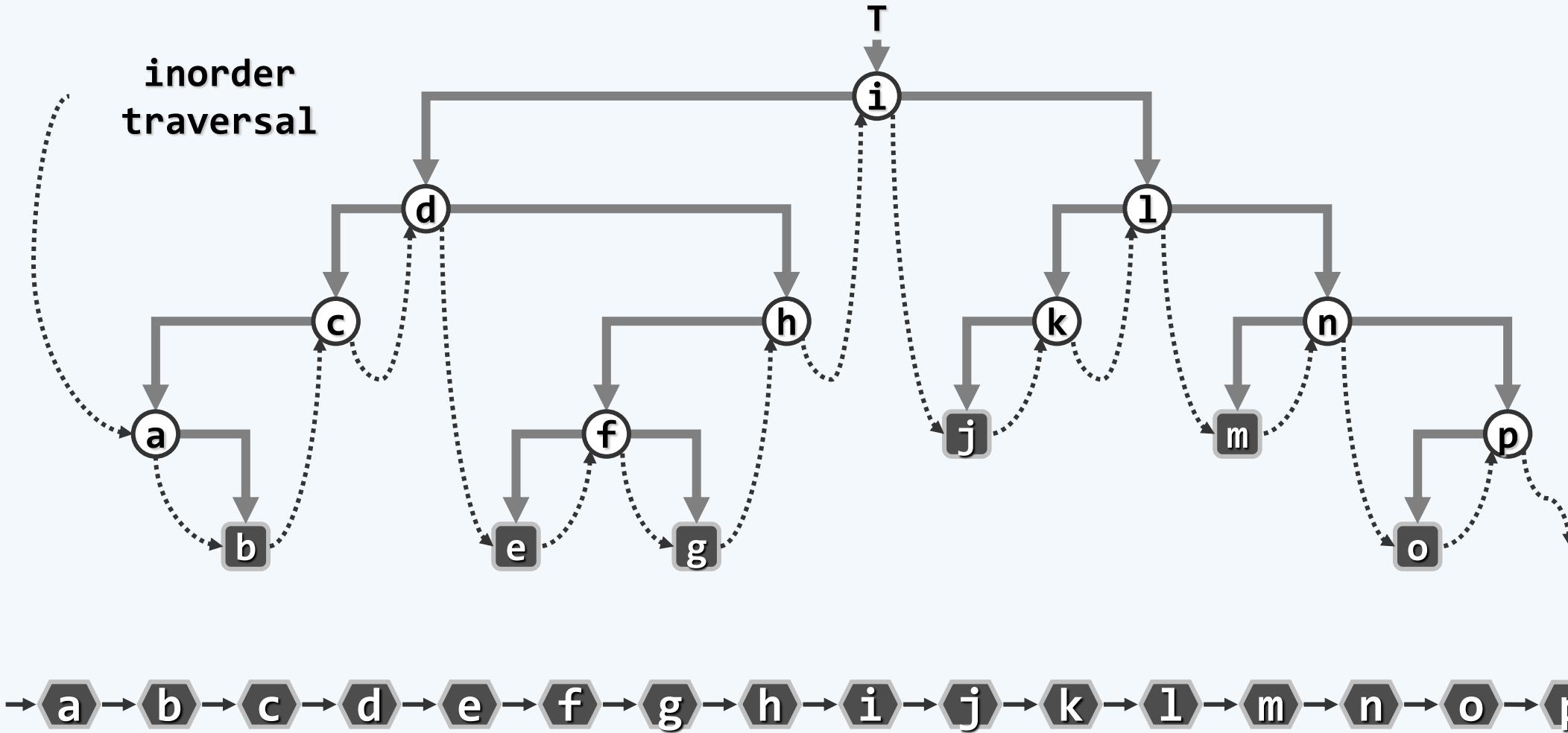
依次对若干棵右子树的遍历问题 //依什么“次”？

❖ 于是，首先要解决的问题就是：

中序遍历任一二叉树T时

首先被访问的是哪个节点？如何找到它？

观察

inorder  
traversal

```
→ a → b → c → d → e → f → g → h → i → j → k → l → m → n → o → p →
```

## 藤缠树

❖ 从根出发沿**左分支**下行，直到**最深**的节点

——它就是全局首先被访问者

❖ 从宏观上，整个遍历过程可划分为 $d+1$ 步迭代

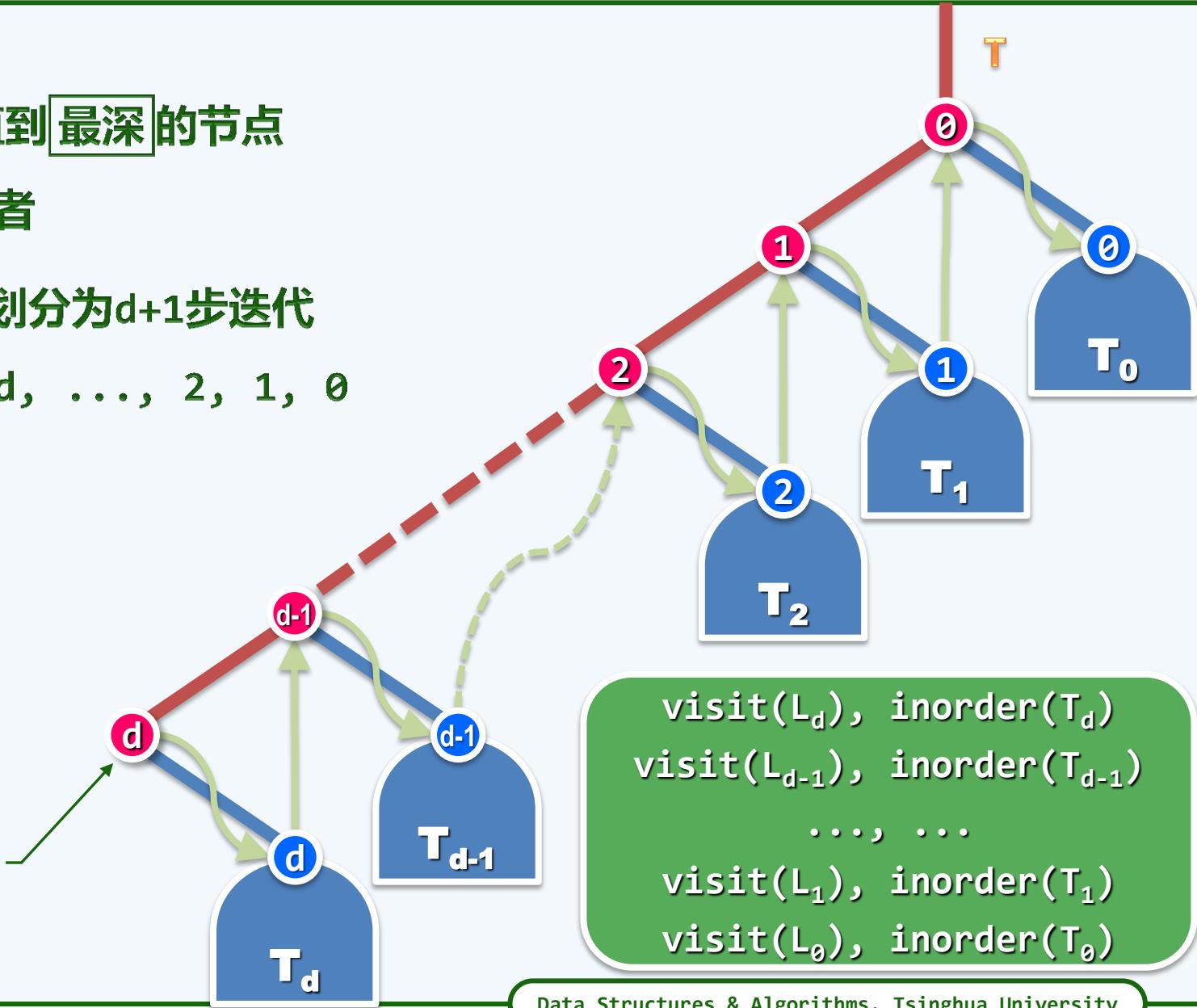
访问 $L_k$ ，再遍历 $T_k$ ， $k = d, \dots, 2, 1, 0$

❖ 不同右子树的遍历

相互独立

自成一个子任务

endpoint of the vine



## 5. 二叉树

中序遍历  
迭代算法

邓俊辉

deng@tsinghua.edu.cn

```
❖ template <typename T> static void  
goAlongLeftBranch( BinNodePosi(T) x, Stack <BinNodePosi(T)> & S ) {  
    while ( x ) {  
        S.push( x );  
        x = x->lC;  
    }  
} //反复地入栈，沿左分支深入
```

```
❖ template <typename T, typename V>

void travIn_I1( BinNodePosi(T) x, V& visit ) {

Stack < BinNodePosi(T) > S; //辅助栈

while ( true ) { //反复地

    goAlongLeftBranch( x, S ); //从当前节点出发，逐批入栈

    if ( S.empty() ) break; //直至所有节点处理完毕

    x = S.pop(); //x的左子树或为空，或已遍历（等效于空），故可以

    visit( x->data ); //立即访问之

    x = x->rc; //再转向其右子树（可能为空，留意处理手法）

}

}
```

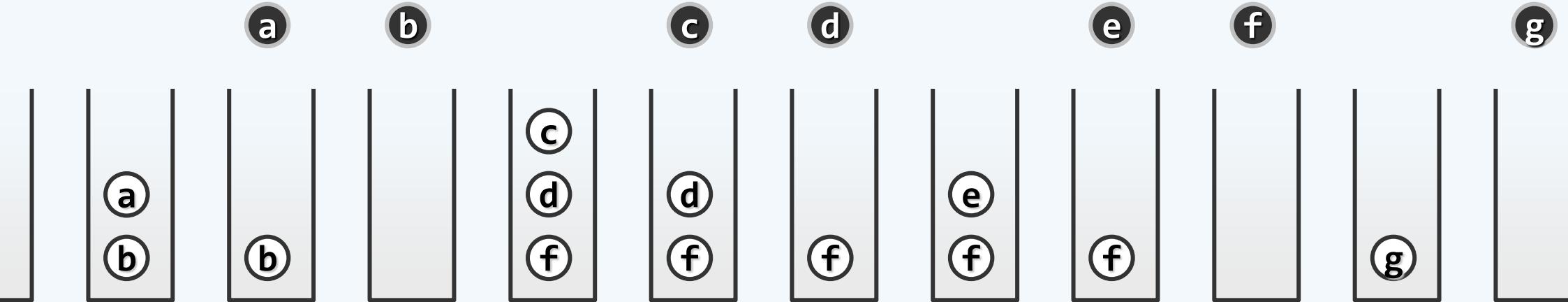
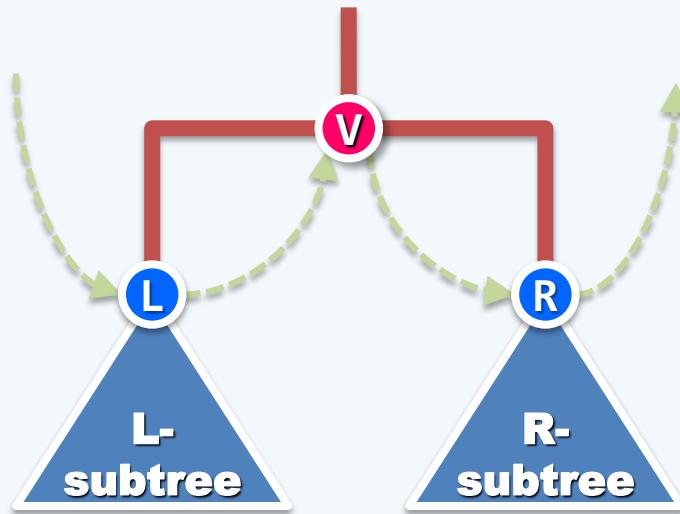
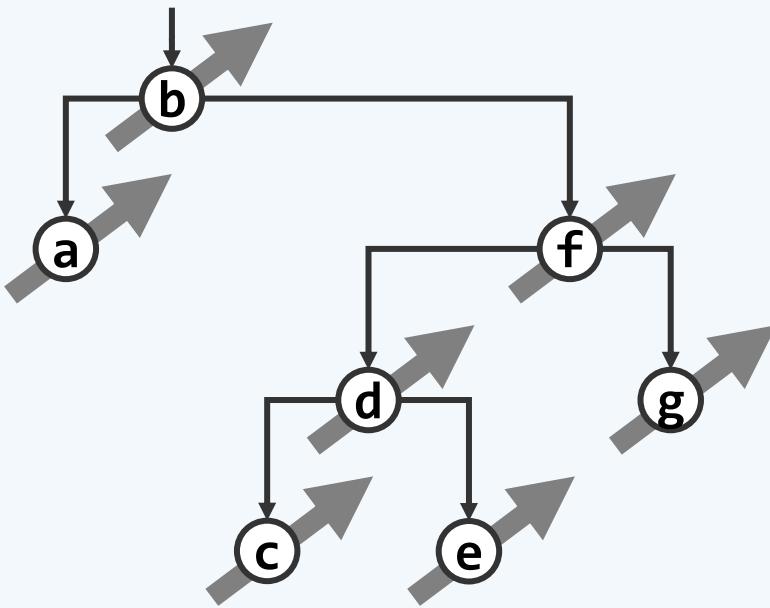
## 5. 二叉树

中序遍历

实例

邓俊辉

deng@tsinghua.edu.cn



## 5. 二叉树

中序遍历

分析

邓俊辉

deng@tsinghua.edu.cn

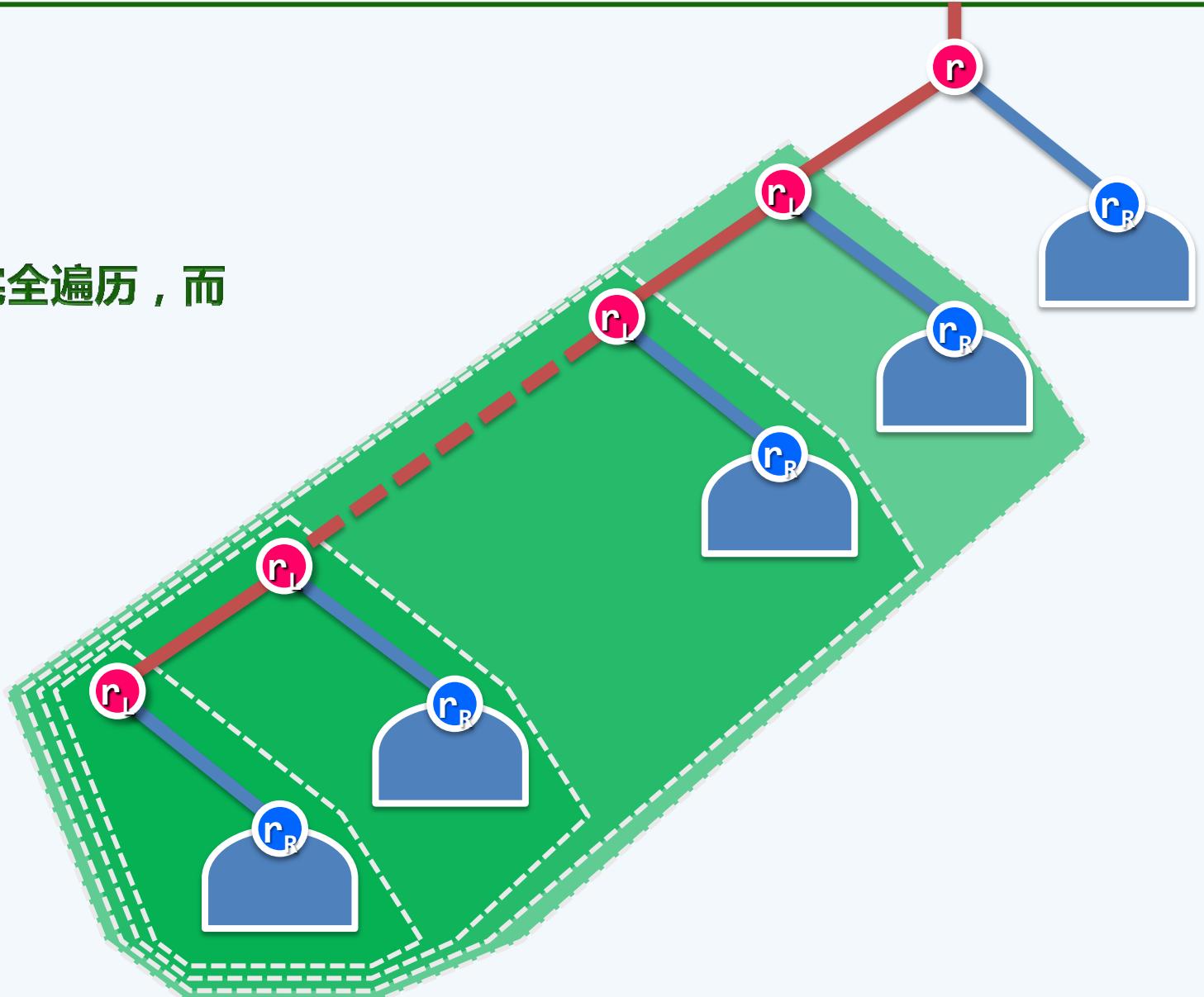
## 正确性

- ❖ 可归纳证明：
- 每个节点出栈时

其**左子树**（若存在）已经完全遍历，而

**右子树尚未入栈**

- ❖ 于是，每当有节点出栈，只需访问它，然后从其**右孩子**出发...



**效率**

❖ 是否 $\Theta(n)$ ，取决于以下条件

- 1) 每次迭代，都恰有一个节点出栈并被访问 //满足
- 2) 每个节点入栈一次且仅一次 //满足
- 3) 每次迭代只需 $O(1)$ 时间 //不再满足，因为...

❖ 单次调用goAlongLeftBranch()

就可能需做 $\Omega(n)$ 次入栈操作，共需 $\Omega(n)$ 时间

❖ 既然如此，难道总体将需要... $O(n^2)$ 时间？

❖ 事实上，这个界远远不紧...

请利用分摊原理，自行分析

❖ 更多的实现：travIn\_I2() + travIn\_I3() + travIn\_I4()

## 5. 二叉树

中序遍历

后继与前驱

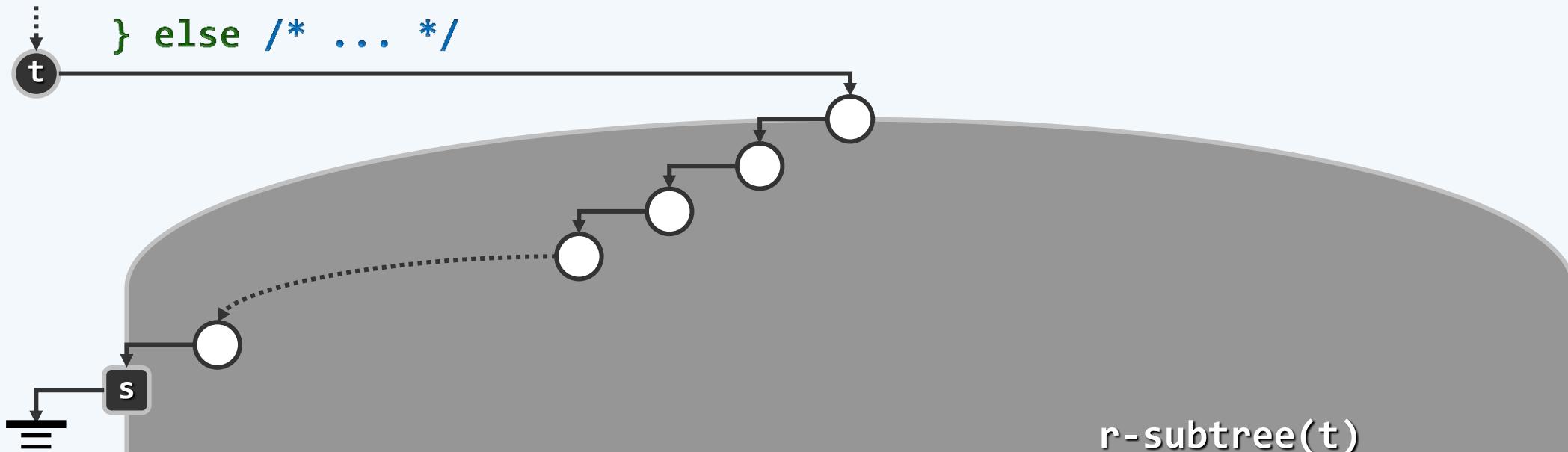
邓俊辉

deng@tsinghua.edu.cn

## 直接后继

❖ template <typename T> //稍后将被BST::remove中的removeAt()调用

```
BinNodePosi(T) BinNode<T>::succ() { //在中序遍历意义下的直接后继
    BinNodePosi(T) s = this; //记录后继的临时变量
    if ( rc ) { //若有右孩子，则直接后继必在右子树中，具体地就是
        s = rc; while ( HasLChild( * s ) ) s = s->lC; //右子树中最小节点
    } else /* ... */
}
```



## 直接后继

❖ } else { //否则，后继应是“将当前节点包含于其左子树中的最低祖先”

while ( IsRChild( \* s ) ) //根节点是左是右？

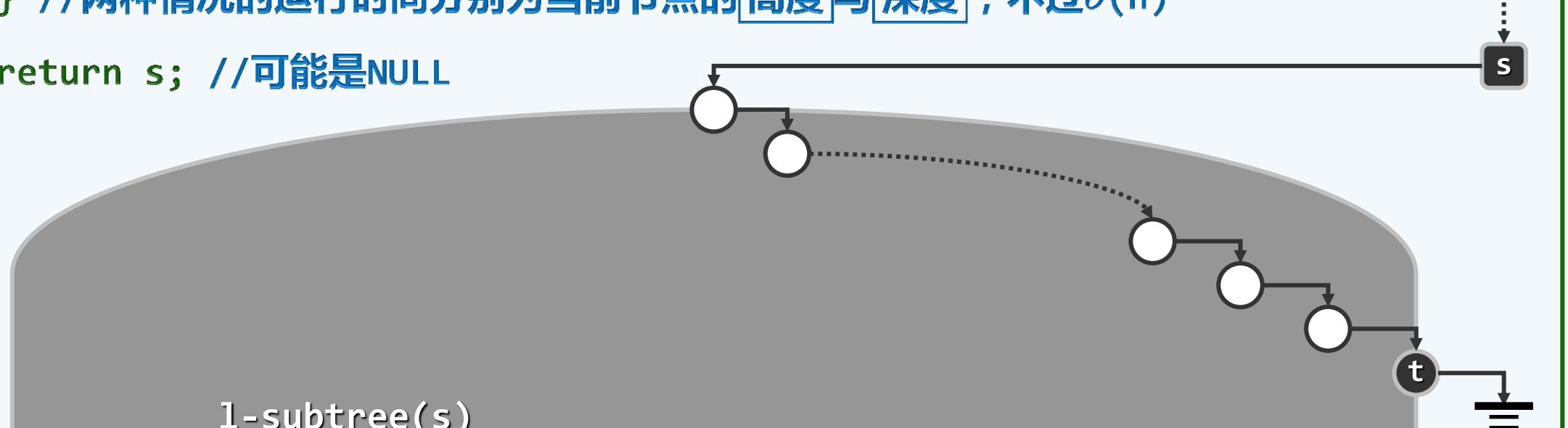
s = s->parent; //逆向地沿右向分支，不断朝**左上方**移动

s = s->parent; //最后再朝**右上方**移动一步，即抵达后继（若存在）

} //两种情况的运行时间分别为当前节点的**高度**与**深度**，不过 $\theta(h)$

return s; //可能是NULL

}



## 5. 二叉树

后序遍历

观察

当下随了仙姑进入二层门内，至两边配殿，皆有匾额对联，一时看不尽许多，惟见有几处写的是：“痴情司”、“结怨司”、“朝啼司”、“夜怨司”、“春感司”、“秋悲司”。看了，因向仙姑道：“敢烦仙姑引我到那各司中游玩游玩，不知可使得？”仙姑道：“此各司中皆贮的是普天之下所有的女子过去未来的簿册，尔凡眼尘躯，未便先知的。”

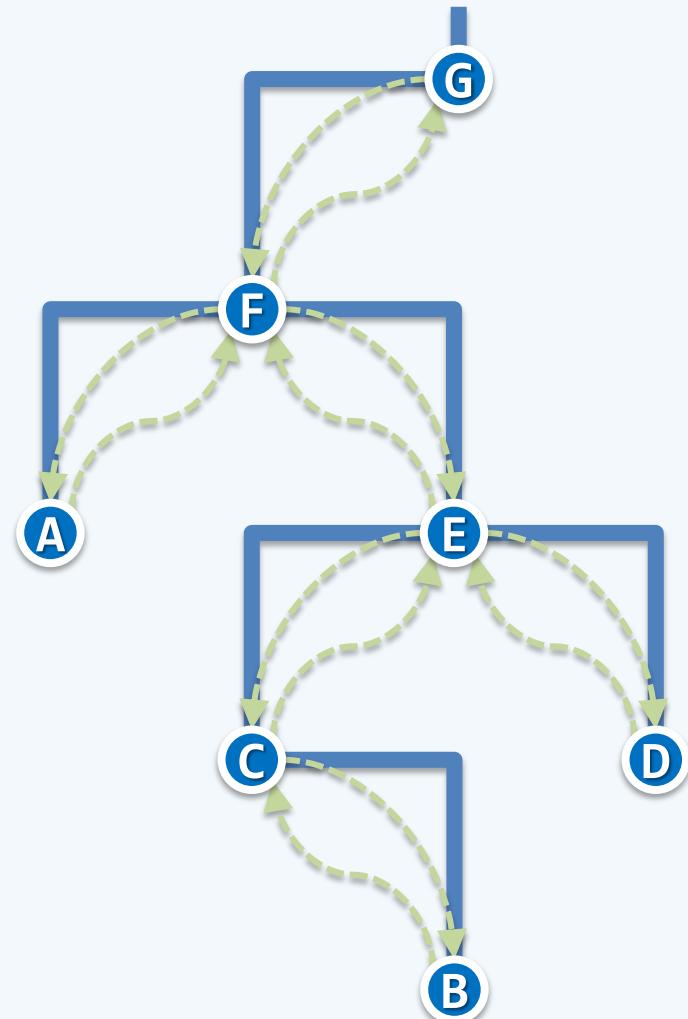
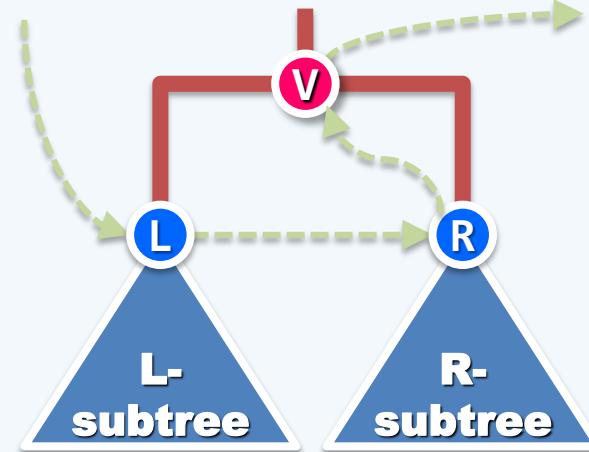
邓俊辉

deng@tsinghua.edu.cn

# 递归

❖ template <typename T, typename VST>

```
void traverse( BinNodePosi(T) x, VST & visit ) {
    if ( ! x ) return;
    traverse( x->lc, visit );
    traverse( x->rc, visit );
    visit( x->data );
} //T(n) = T(a) + T(n - a - 1) + O(1) = O(n)
```



❖ 应用：BinNode::size()

BinTree::updateHeight()

❖ 挑战：不依赖递归机制，能否实现后序遍历？如何实现？效率如何？

## 难点

❖ 难度在于

- 对左、右子树的递归遍历，都不是尾递归

❖ 解决方法

- 找到第一个被访问的节点
- 将其祖先及其右兄弟（如果存在）用栈保存

❖ 这样，原问题就被分解为

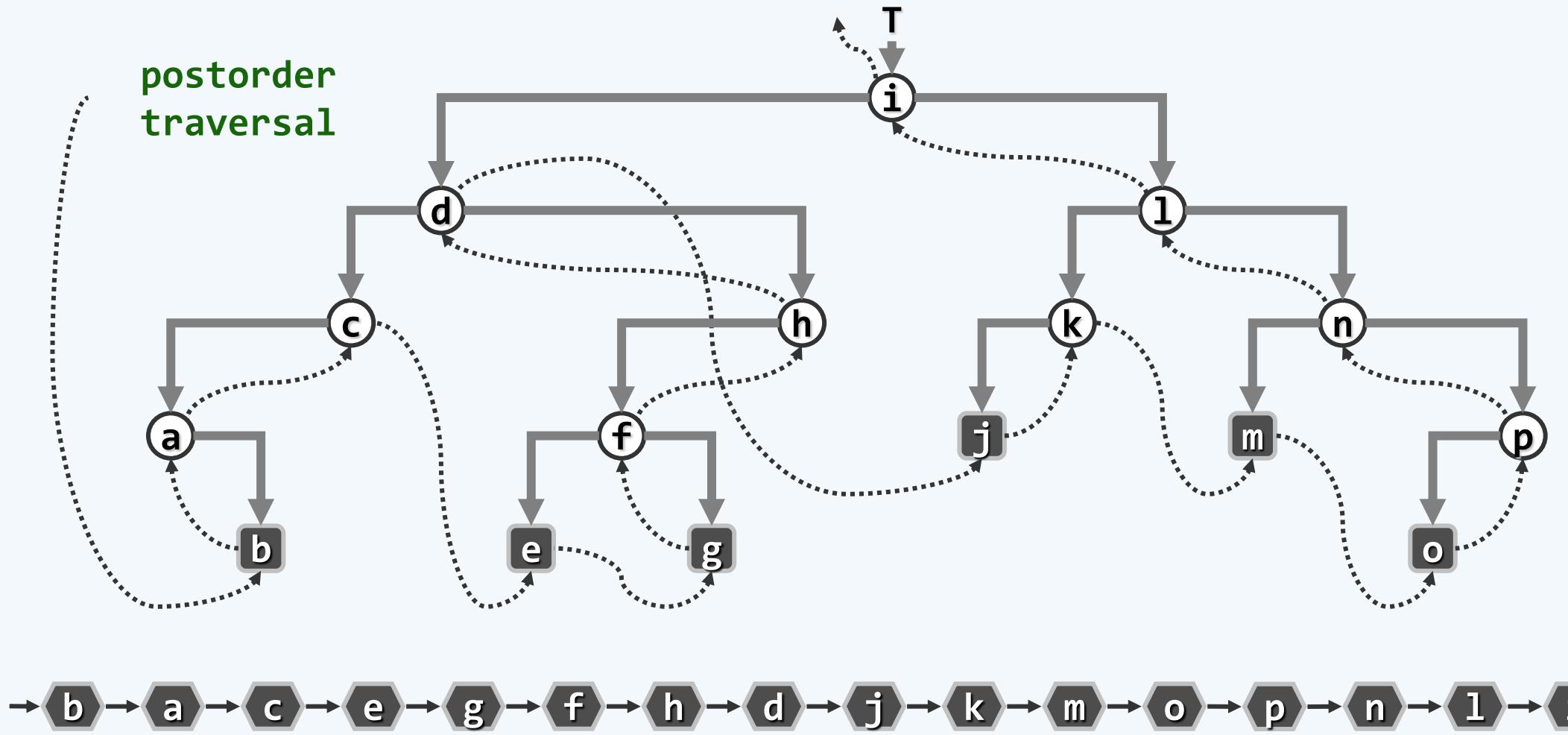
- 依次对若干棵右子树的遍历问题 //同样地，这里应依什么“次”？

❖ 于是，首先要解决的问题仍是

- 后序遍历任一二叉树T时，首先被访问的是哪个节点？如何找到它？

观察

postorder traversal



## 藤缠树

❖ 从根出发下行

尽可能沿左分支

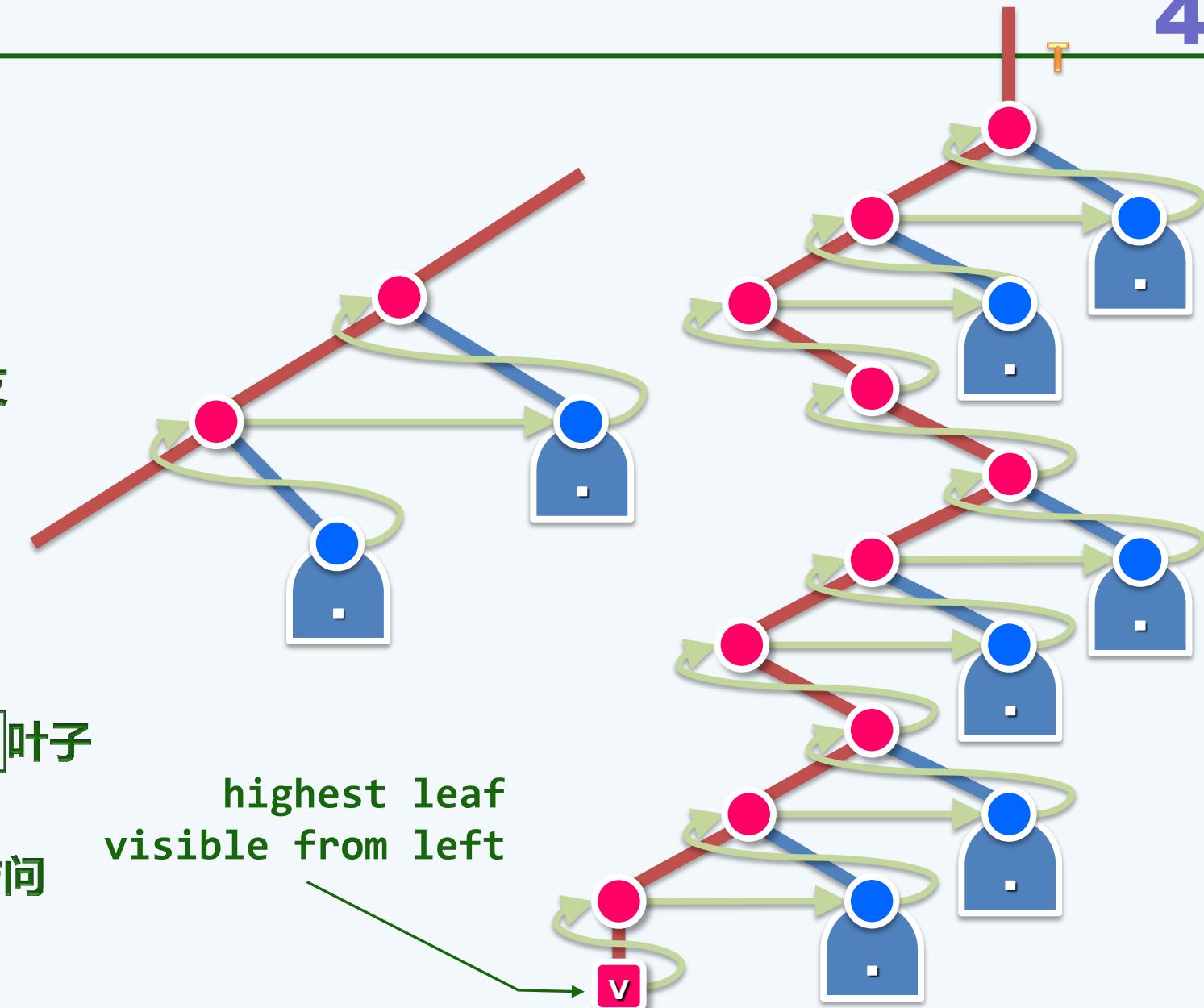
实不得已，才沿右分支

❖ 最后一个节点

必是叶子，而且

是从左侧**可见的最高**叶子

❖ 这匹叶子，将首先接受访问



## 5. 二叉树

后序遍历

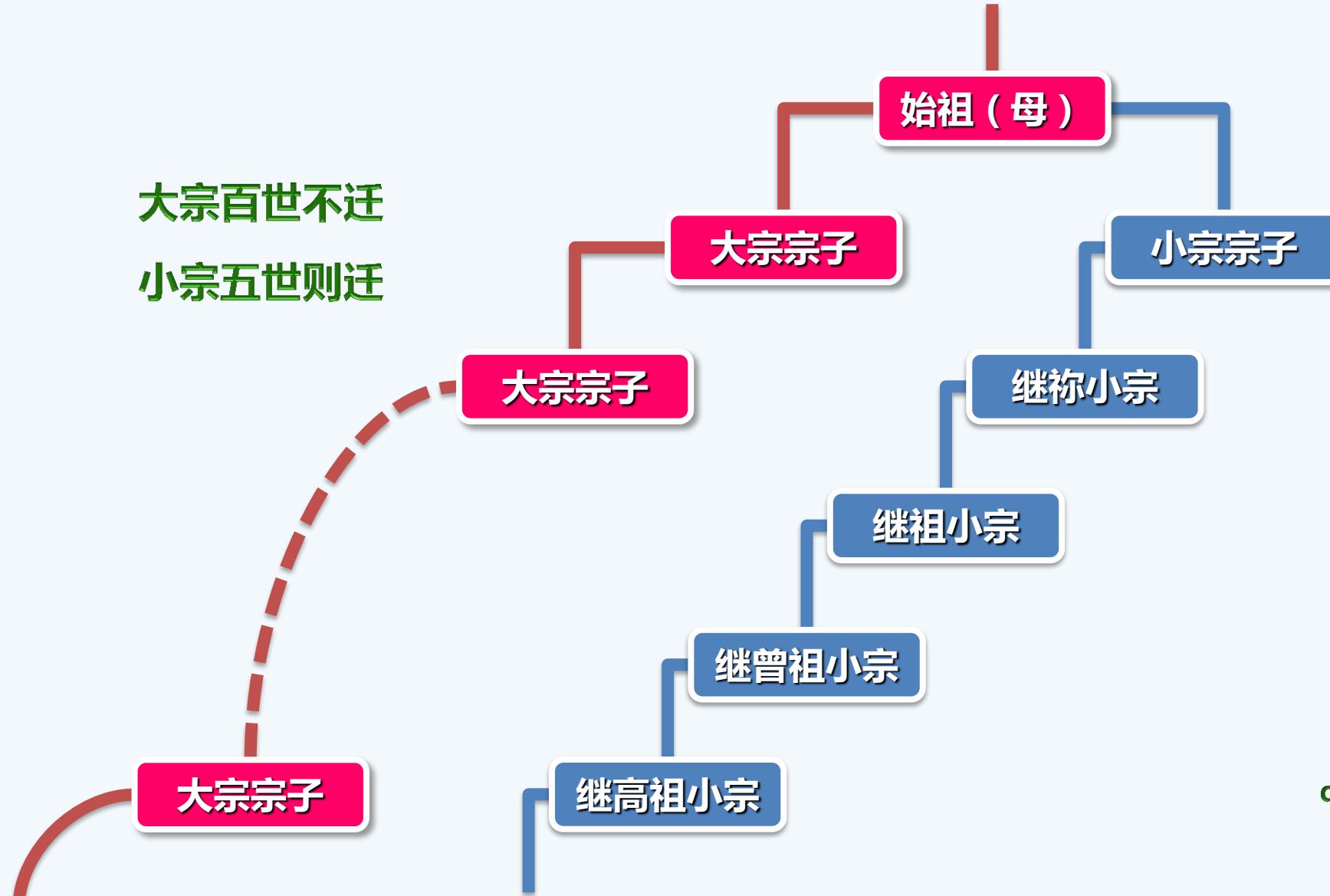
迭代算法

邓俊辉

deng@tsinghua.edu.cn

大宗百世不迁

小宗五世则迁



## gotoHLVFL()

```

❖ template <typename T> static void gotoHLVFL( Stack <BinNodePosi(T)> & S ) {

    while ( BinNodePosi(T) x = S.top() ) //自顶而下反复检查栈顶节点

        if ( HasLChild( * x ) ) { //尽可能向左。在此之前

            if ( HasRChild( * x ) ) //若有右孩子，则

                S.push( x->rc ); //优先入栈

                S.push( x->lc ); //然后转向左孩子

        } else //实不得已

            S.push( x->rc ); //才转向右孩子

    S.pop(); //返回之前，弹出栈顶的空节点
}

```

## travPost\_I()

```

❖ template <typename T, typename VST>

void travPost_I( BinNodePosi(T) x, VST & visit ) {

    Stack < BinNodePosi(T) > S; //辅助栈

    if ( x ) S.push( x ); //根节点非空则首先入栈

    while ( ! S.empty() ) { //x为当前节点

        if ( S.top() != x->parent ) //栈顶非x之父（则必为其右兄）
            gotoHVLFL( S ); //在x的右子树中，找到HVLFL

        x = S.pop(); //弹出栈顶（即前一节点之后继）以更新x，并随即

        visit( x->data ); //访问之

    }
}

```

## 5. 二叉树

后序遍历

实例

很久以来

我就渴望升起

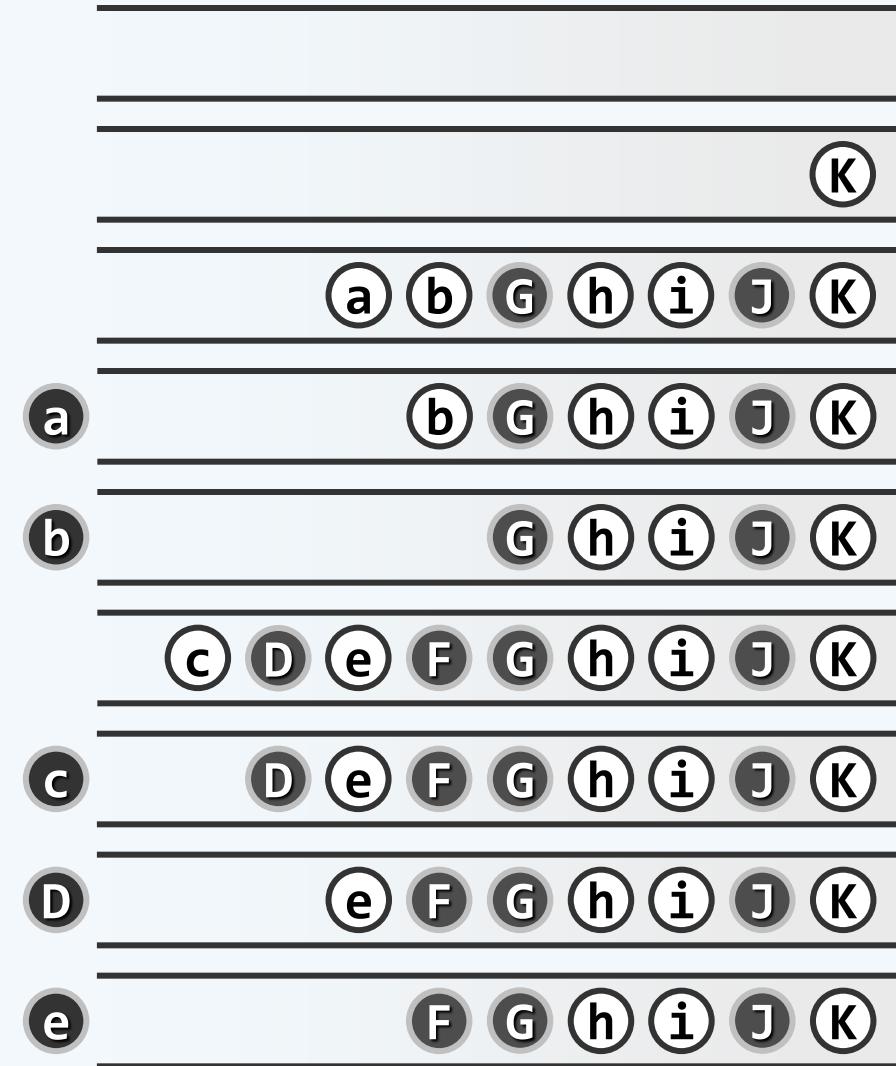
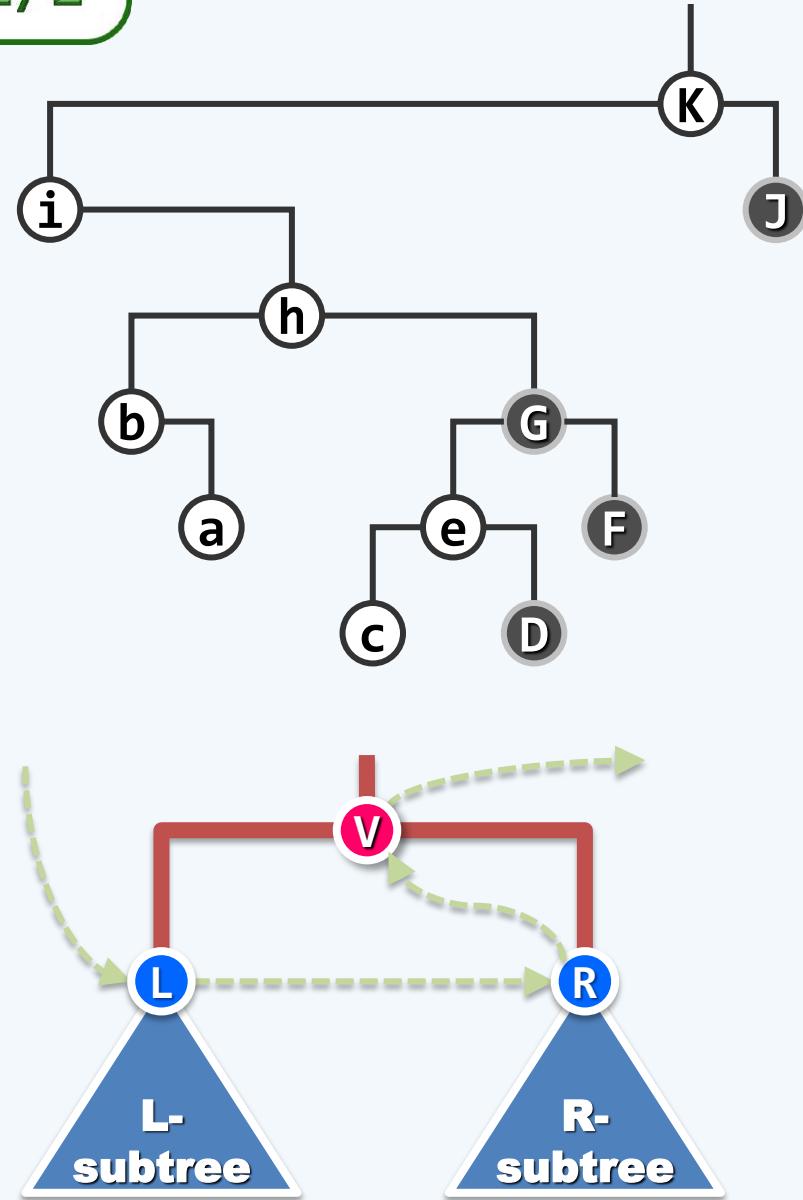
长长的，象绿色植物

去缠绕黄昏的光线

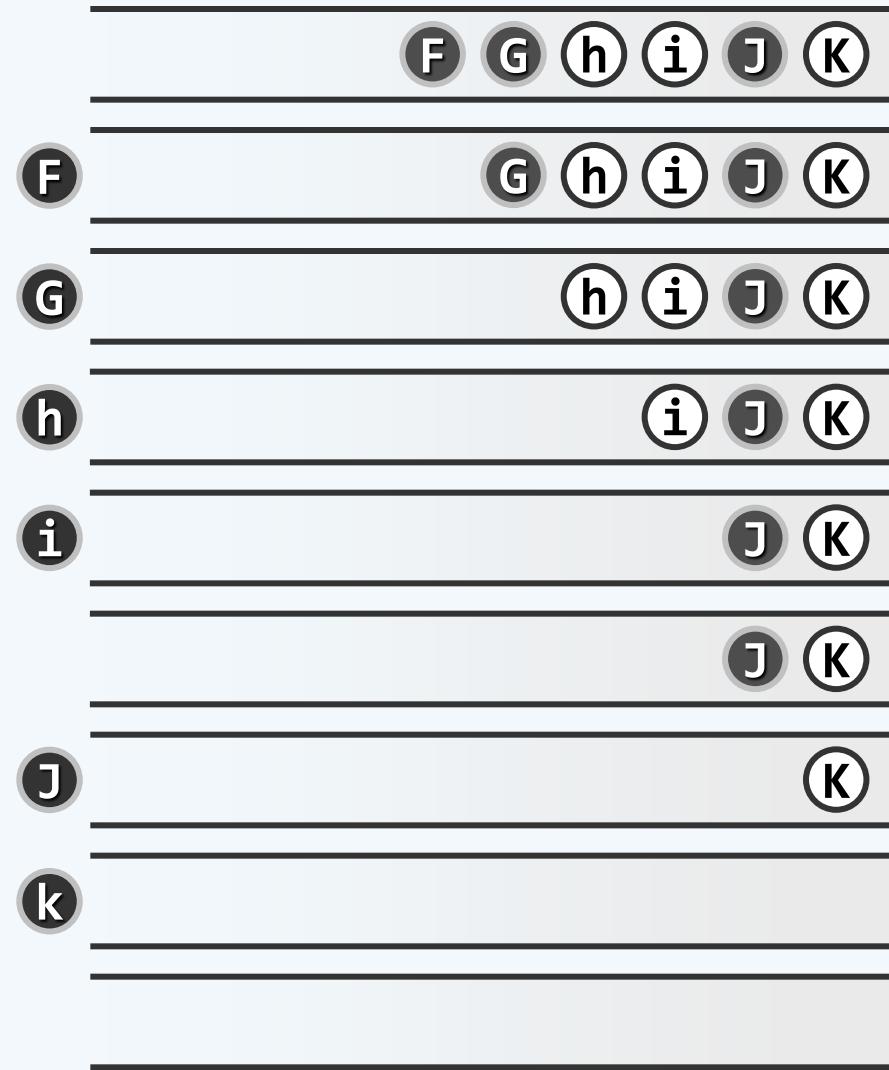
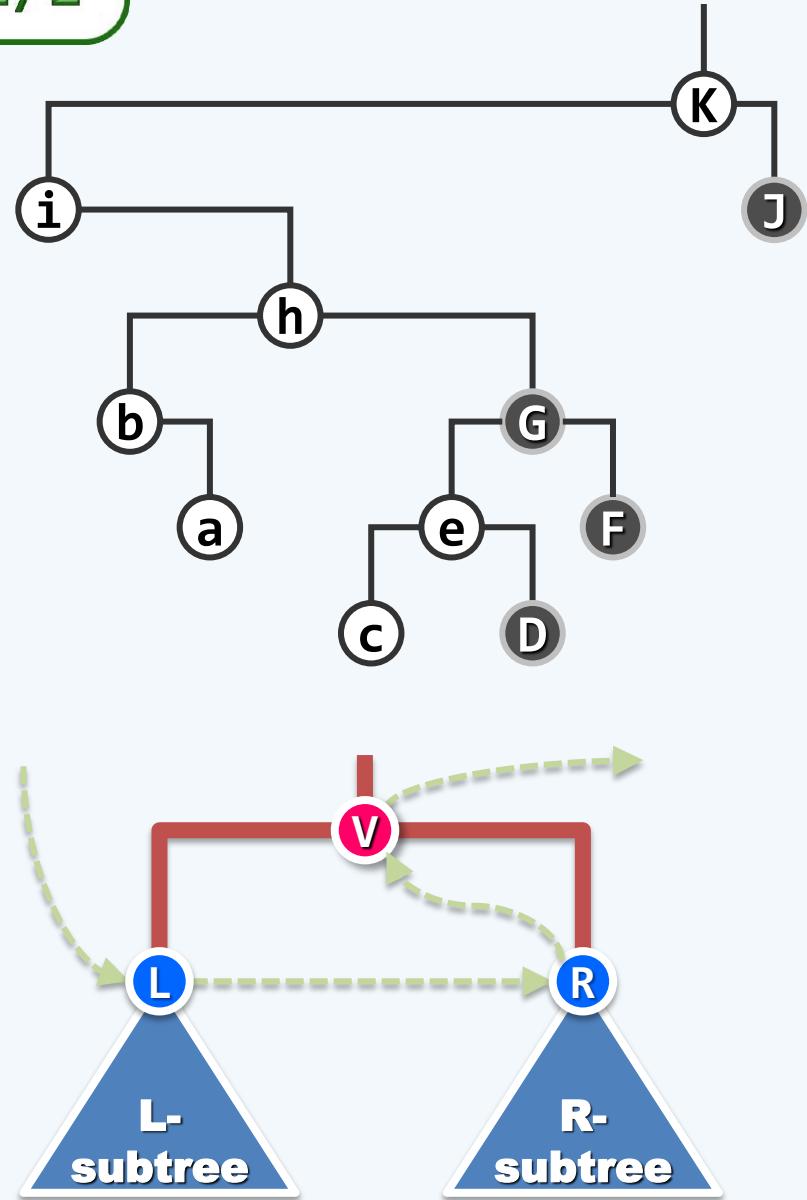
邓俊辉

deng@tsinghua.edu.cn

1/2



2/2



## 5. 二叉树

后序遍历

分析

邓俊辉

deng@tsinghua.edu.cn

## 正确性

❖ 可归纳证明：

每个节点出栈后

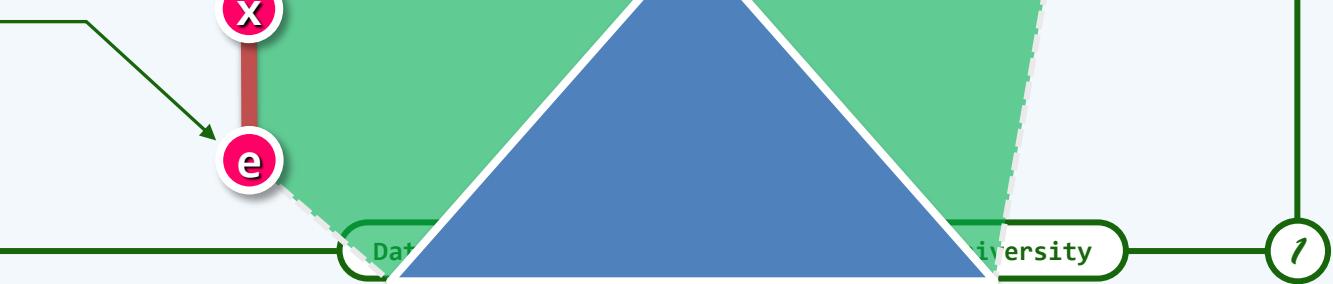
以之为根的子树已经完全遍历，而且

其右兄弟  $r$  (如果存在) 就在栈顶

❖ 于是，为“递归”遍历  $r$  对应的子树，只需

从  $r$  出发...

highest leaf  
visible from left



## 效率

- ❖ 是否 $\Theta(n)$ ，取决于以下条件
  - 每次迭代，都有一个节点出栈并被访问 //满足
  - 每个节点入栈一次且仅一次 //满足
  - 每次迭代只需 $\Theta(1)$ 时间 //不再满足，因为...
- ❖ 单次调用gotoHLVFL()  
就可能需要 $\Omega(n)$ 时间
- ❖ 既然如此，难道总体将需要... $\Theta(n^2)$ 时间？
- ❖ 同样地，事实上这个界远远不紧...
- ❖ 空间？

## 5. 二叉树

后序遍历  
表达式树

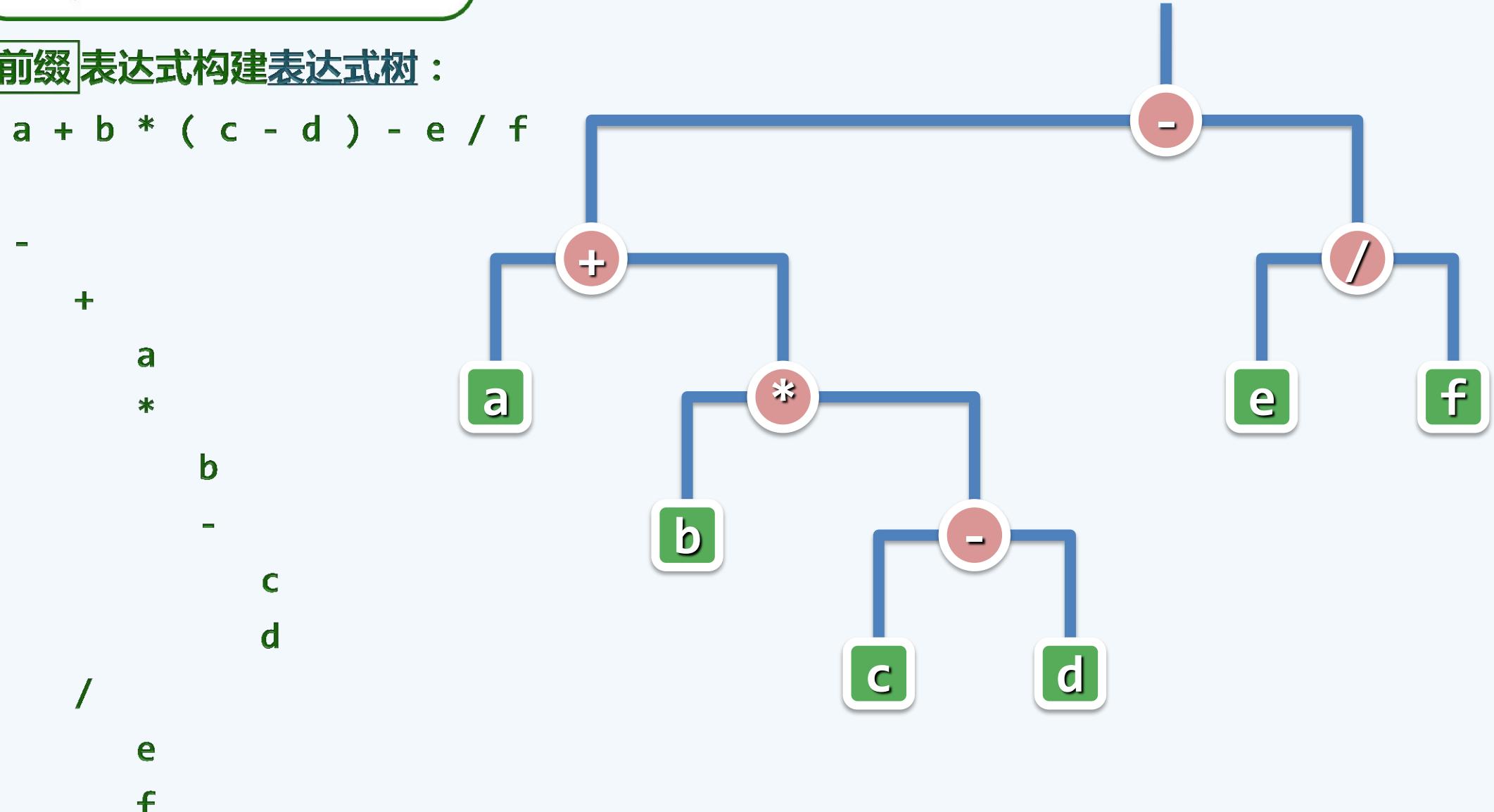
邓俊辉

deng@tsinghua.edu.cn

## Expression Tree

❖ 由前缀表达式构建表达式树：

$a + b * ( c - d ) - e / f$



❖ postorder

( postfix = RPN )

a b c d - \* + e f / -

❖ preorder

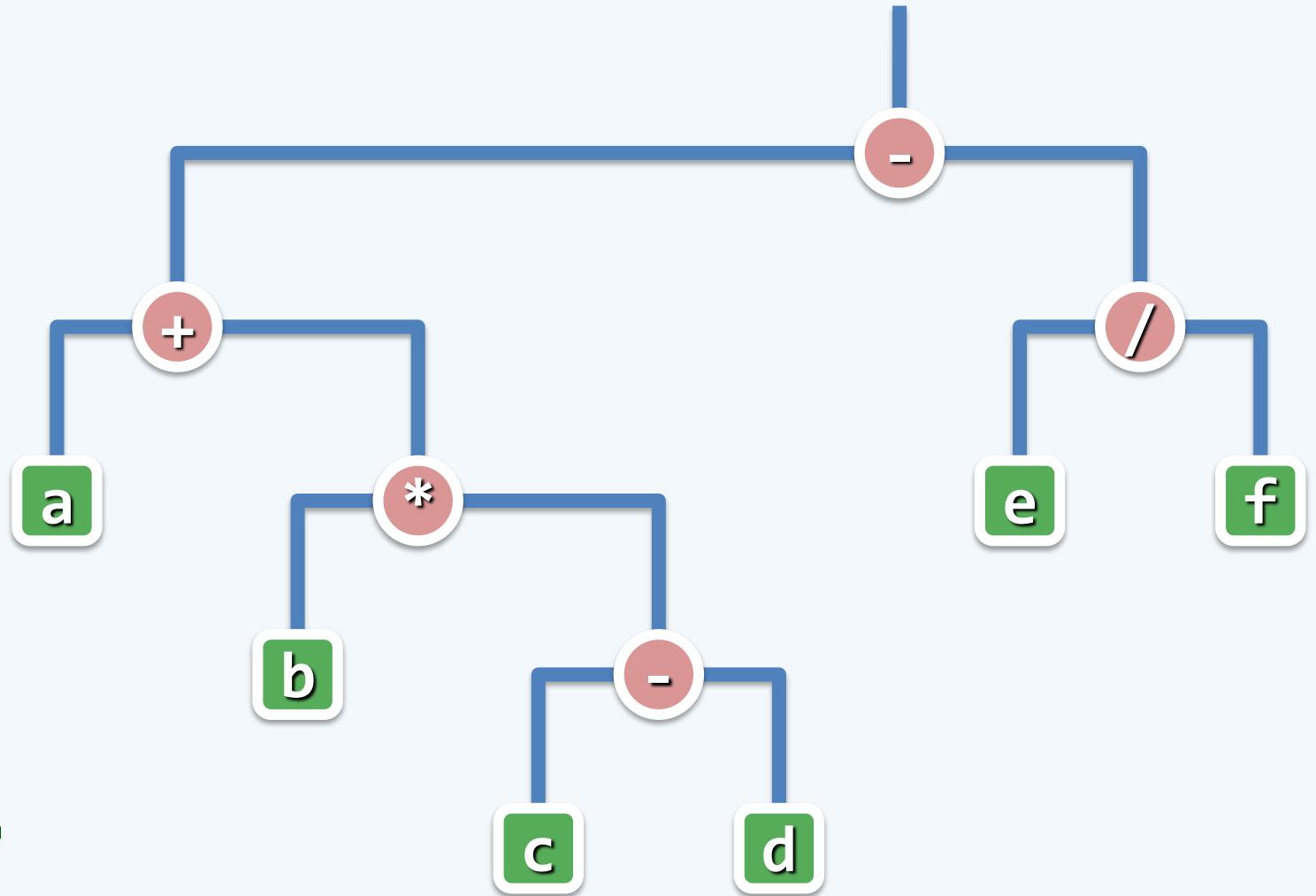
( prefix , 求值并不便捷 )

- + a \* b - c d / e f

❖ inorder

( infix , 无优先级 , 有歧义 )

a + b \* c - d - e / f



## 5. 二叉树

### 层次遍历 算法

邓俊辉

deng@tsinghua.edu.cn

```
❖ template <typename T> template <typename VST>

void BinNode<T>::travLevel( VST & visit ) { //二叉树层次遍历

Queue< BinNodePosi(T) > Q; //引入辅助队列

Q.enqueue( this ); //根节点入队

while ( ! Q.empty() ) { //在队列再次变空之前，反复迭代

BinNodePosi(T) x = Q.dequeue(); //取出队首节点，并随即

visit( x->data ); //访问之

if ( HasLChild( * x ) ) Q.enqueue( x->lc ); //左孩子入队

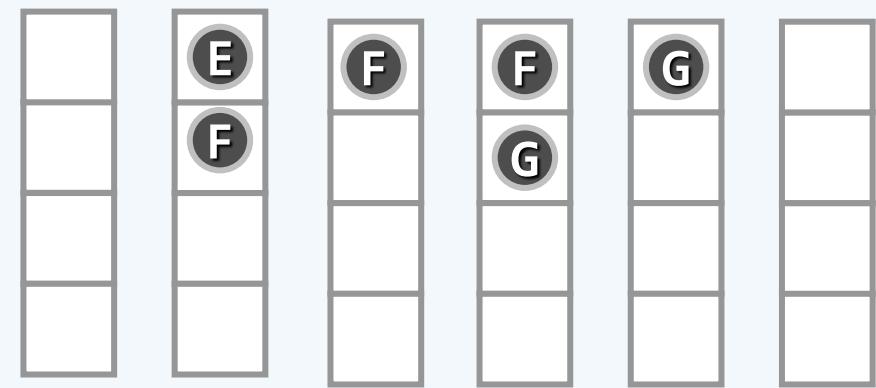
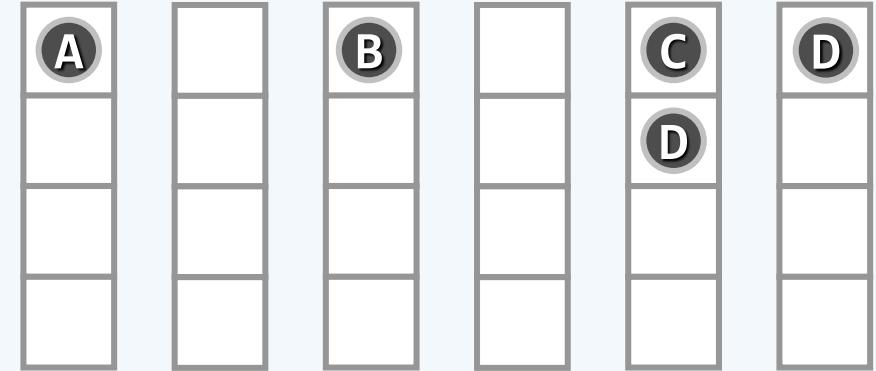
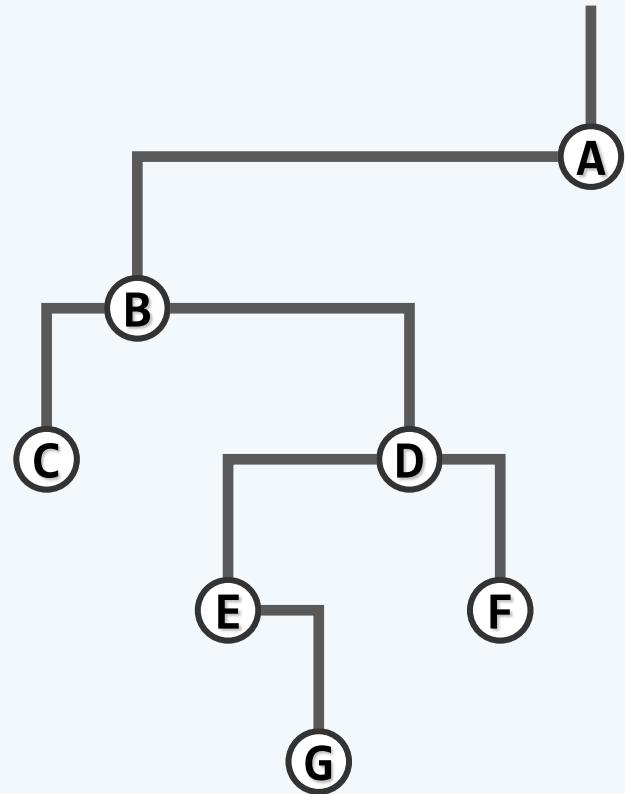
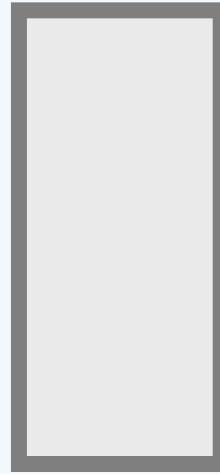
if ( HasRChild( * x ) ) Q.enqueue( x->rc ); //右孩子入队

}

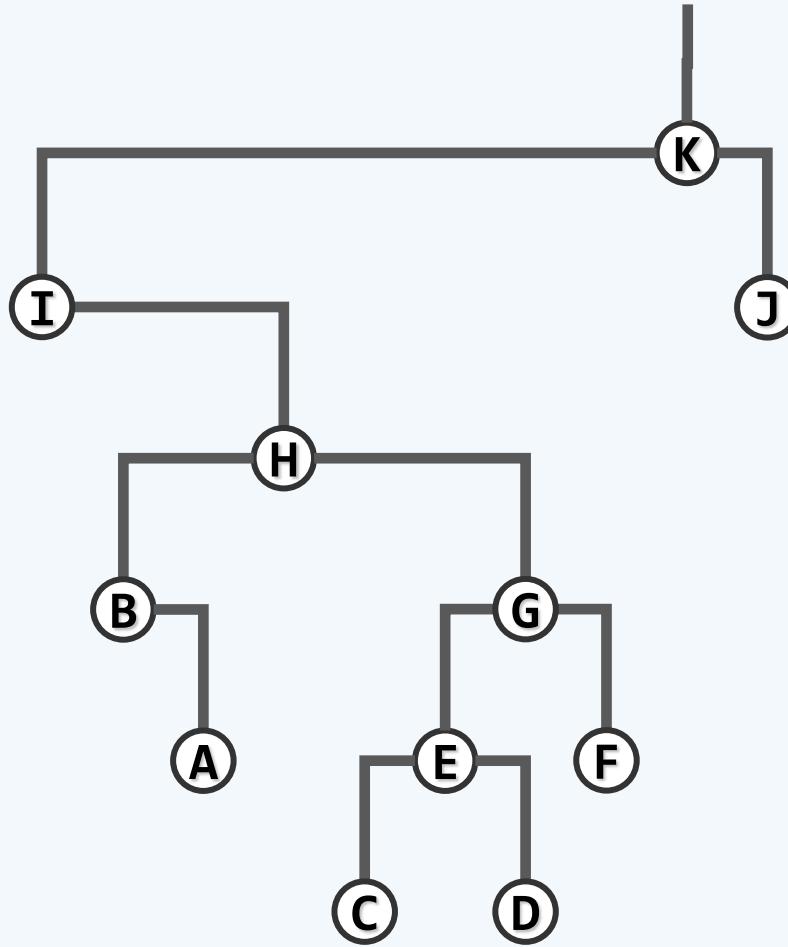
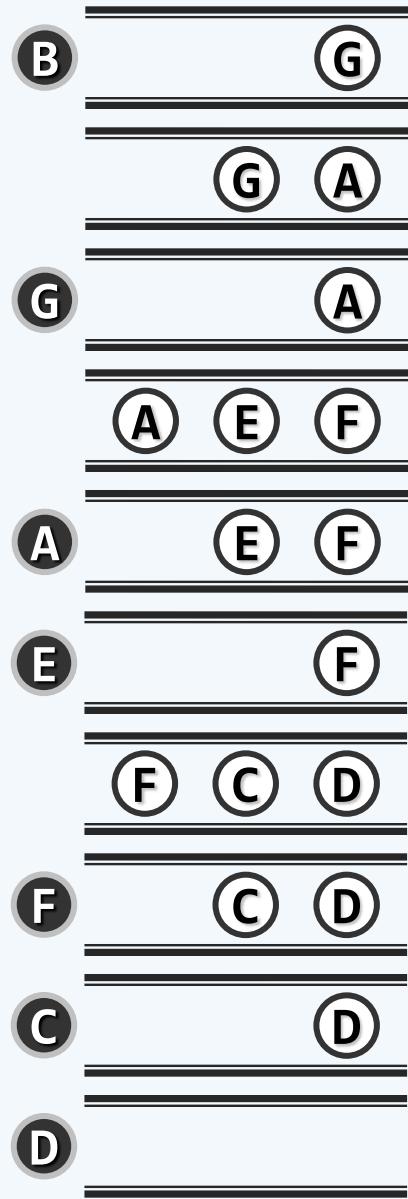
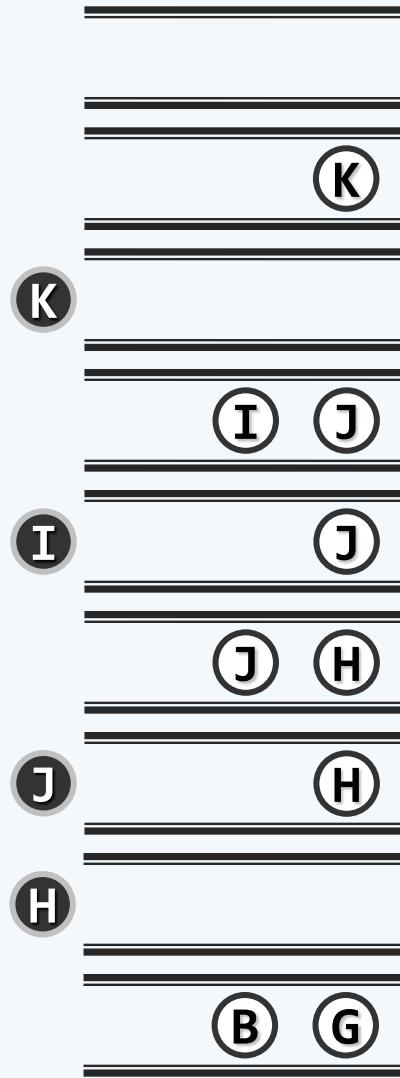
}
```

}

## 实例



## 实例



## 5. 二叉树

层次遍历

分析

邓俊辉

deng@tsinghua.edu.cn

## 正确性

- ❖ 何以见得？以上迭代算法符合广度优先遍历的规则...
- ❖ 每次迭代， 入队节点（若存在）都是出队节点的孩子，深度增加一层
- ❖ 任何时刻， 队列中各节点按深度**单调**排列，而且
  - (相邻) 节点的**深度**相差不超过**1**层
- ❖ 进一步地， 所有节点迟早都会入队，而且
  - 更**高**/**低**的节点，更**早**/**晚**入队
  - 更**左**/**右**的节点，更**早**/**晚**入队

## 复杂度

- ❖ 效率如何？
- ❖ 每次迭代
  - 都有一个节点出队并接受访问
  - 但可能有两个节点入队
- ❖ 更精确地

每个节点入、出队各恰好一次

整体效率 =  $\mathcal{O}(n)$

## 5. 二叉树

层次遍历

完全二叉树

邓俊辉

deng@tsinghua.edu.cn

## 完全二叉树

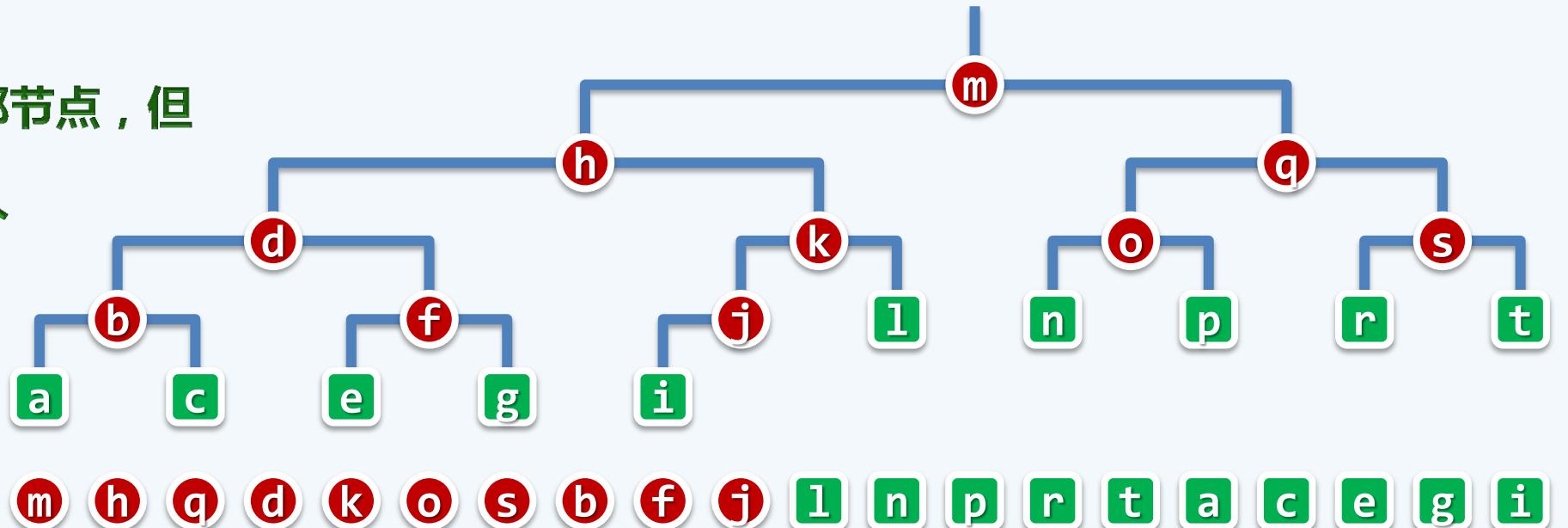
❖ 叶节点仅限于最低两层

底层叶子，均居于次底层叶子左侧

除末节点的父亲，内部节点均有双子

❖ 叶节点

- 不致少于内部节点，但
- 至多多出一个



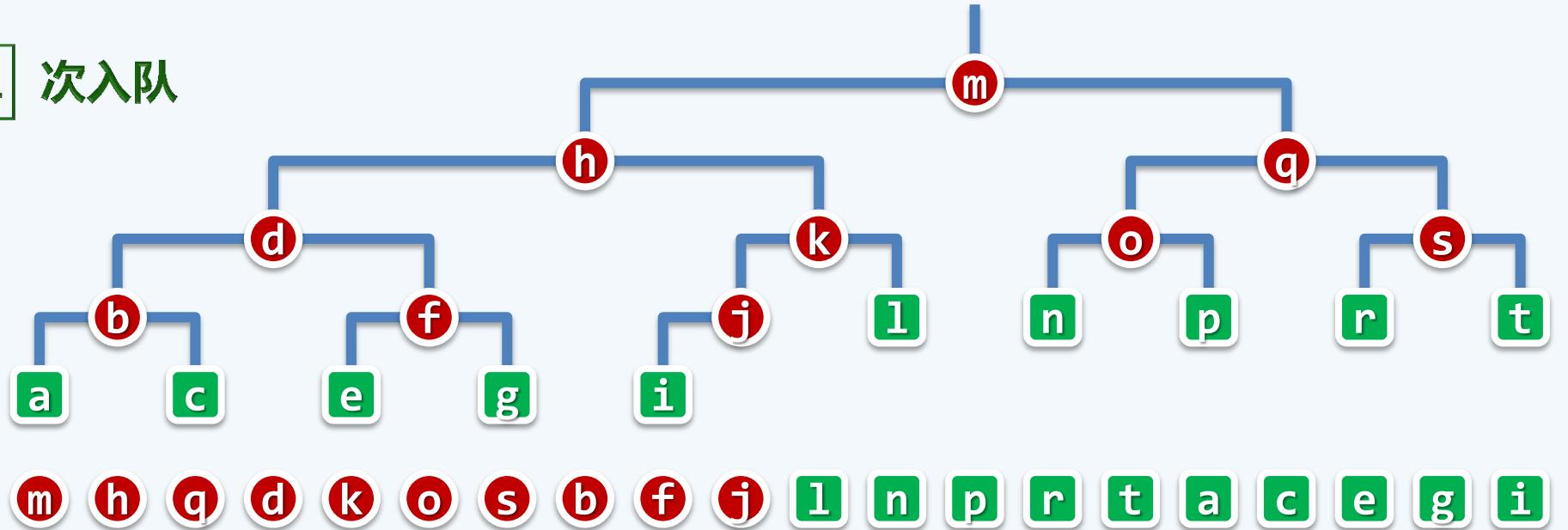
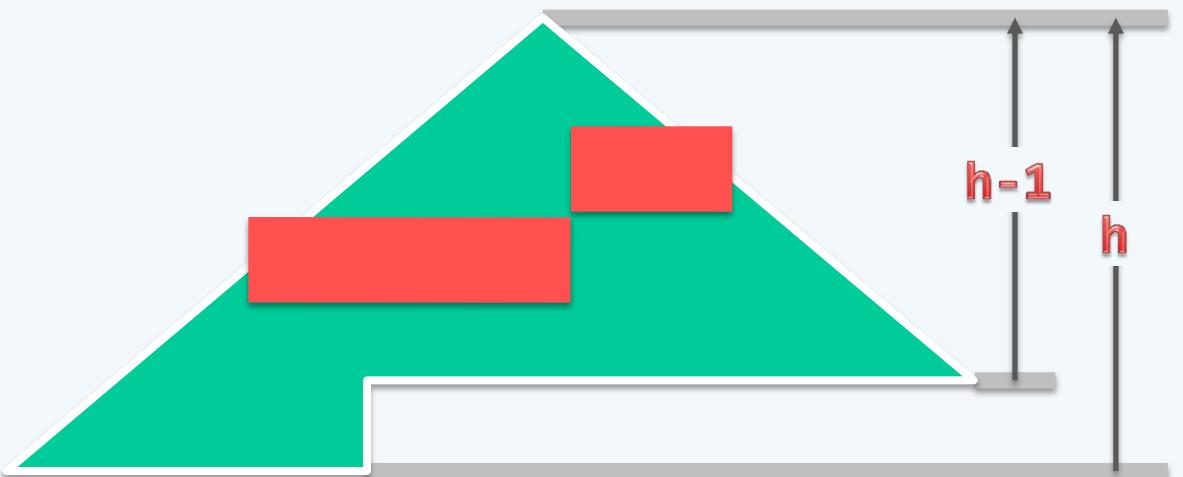
## 层次遍历

❖ 考察遍历过程中的n步迭代...

❖ 前  $\left\lceil \frac{n}{2} \right\rceil - 1$  步迭代中，都有右孩子入队

前  $\left\lceil \frac{n}{2} \right\rceil$  步迭代中，都有左孩子入队

累计至少  $n - 1$  次入队



## 层次遍历

❖ 考察遍历过程中的n步迭代...

❖ 辅助队列的规模

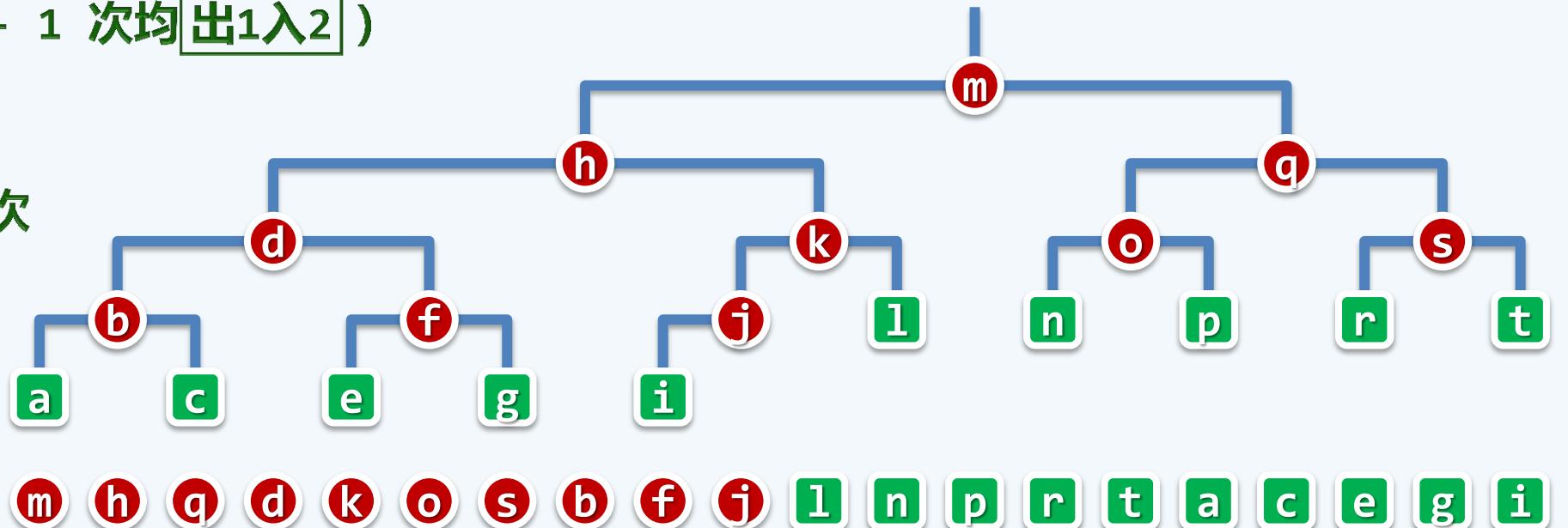
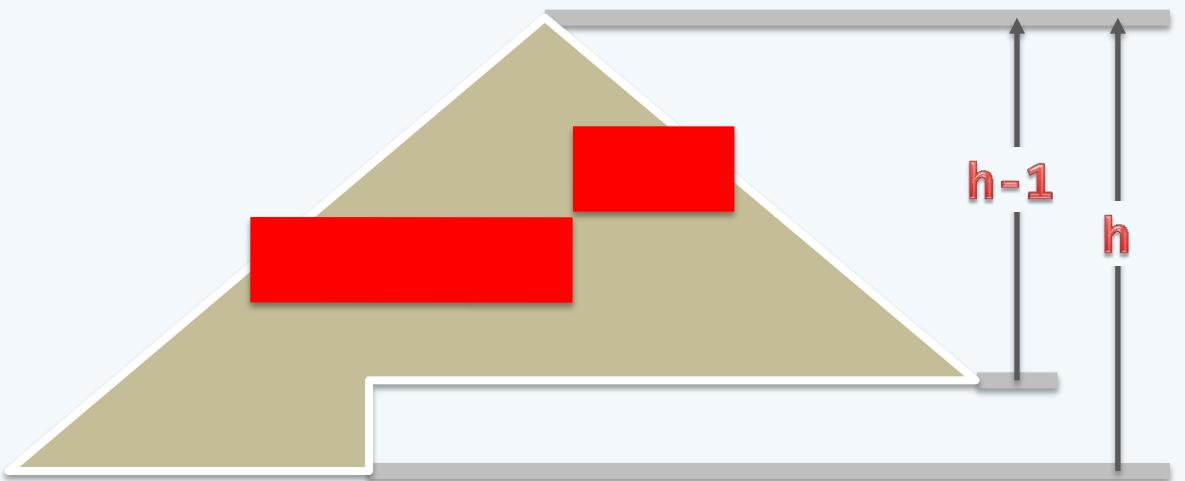
- 先增后减，**单峰对称**

- 最大规模 =  $\lceil n/2 \rceil$

(前  $\lceil n/2 \rceil - 1$  次均**出1入2**)

- 最大规模

**可能出现两次**



## 5. 二叉树

重构

邓俊辉

deng@tsinghua.edu.cn

[ 先序 | 后序 ] + 中序

❖ 详见视频

[ 先序 + 后序 ] × 真

❖ 详见视频

## 5. 二叉树

Huffman 编码树

PFC 编码

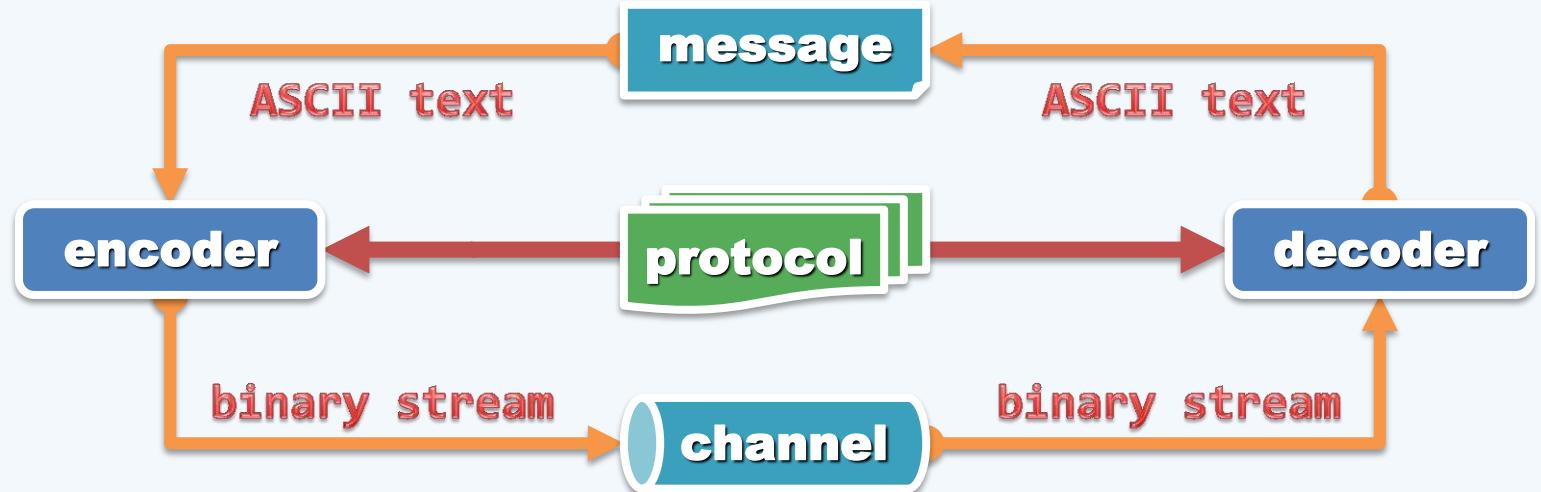
句读之不知，惑之不解，或师焉，或不焉，  
小学而大遗，吾未见其明也

邓俊辉

deng@tsinghua.edu.cn

## 应用

### ❖ 通讯 / 编码 / 译码



### ❖ 二进制编码

- 组成数据文件的字符来自字符集 $\Sigma$
- 字符被赋予互异的二进制串

"101001100" = "MAIN"

### ❖ 文件的大小取决于

- 字符的数量  $\times$  各字符编码的长短



### ❖ 通讯带宽有限时

- 如何对各字符编码，使文件最小？

## 二叉编码树

将 $\Sigma$ 中的字符组织成一棵二叉树

以0/1表示左/右孩子

各字符x分别存放于对应的叶子 $v(x)$ 中

字符x的编码串 $rps(v(x)) = rps(x)$

由根到 $v(x)$ 的通路 (root path) 确定

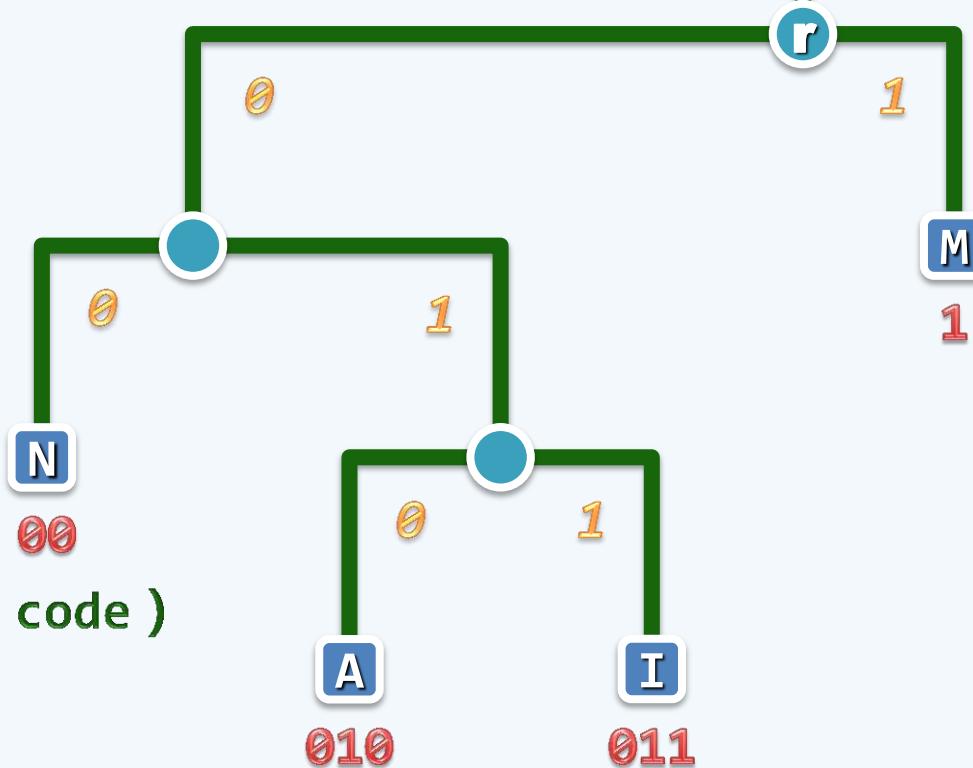
优点：字符编码不必等长，且  
不致出现解码歧义

这属于“前缀无歧义”编码 (prefix-free code)

不同字符的编码互不为前缀，故不致歧义

缺点：你能发现吗？

"101001100" = "MAIN"



## 编码长度 vs. 叶节点平均深度

❖  $|\text{rps}(x)| = \text{depth}(v(x))$

❖ 编码总长 =  $\sum_x \text{depth}(v(x))$

平均编码长度

$$= \sum_x \text{depth}(v(x)) / |\Sigma|$$

= 叶节点平均深度  $\text{ald}(T)$

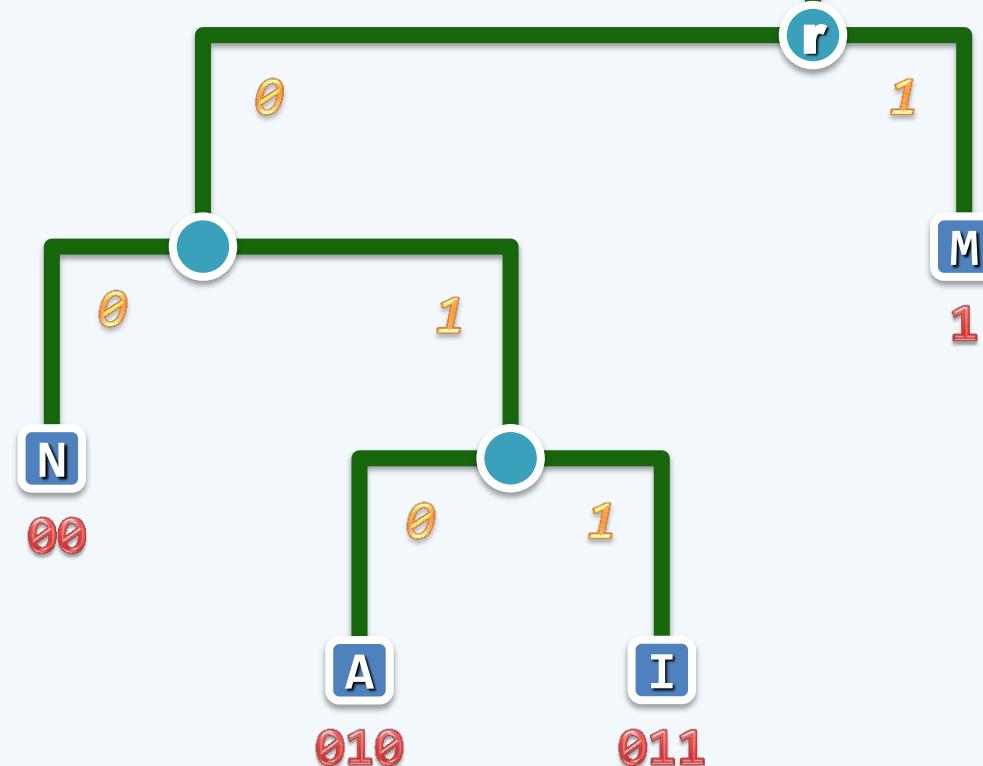
❖ 对于特定的  $\Sigma$

$\text{ald}()$  最小者即为最优编码树  $T_{\text{opt}}$

❖ 最优编码树必然存在，但不见得唯一  
它们具有哪些特征？

$$\text{ald}(T) * 4 = 2+3+3+1 = 9$$

$$"1010_01100" = "MAIN"$$



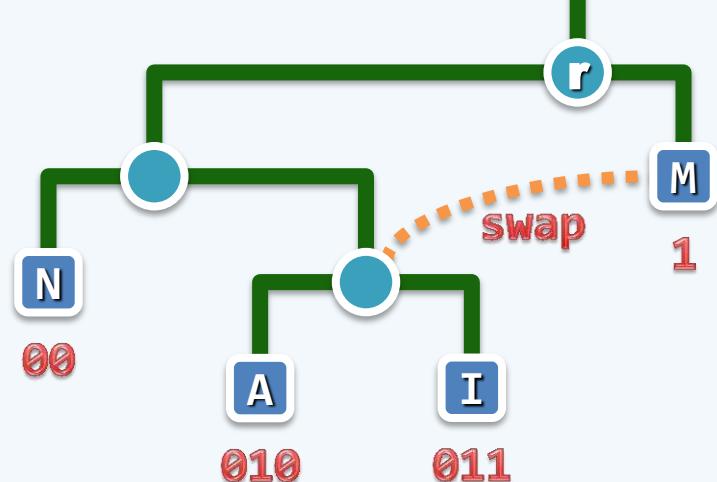
## 最优编码树

❖  $\forall v \in T_{\text{opt}}, \deg(v) = 0$  [only if]  $\text{depth}(v) \geq \text{depth}(T_{\text{opt}}) - 1$

亦即，叶子只能出现在倒数两层内——否则，通过节点交换即可...

$$\text{ald}(T) * 4 = 2+3+3+1 = 9$$

$$\text{"1}^010_011^00" = \text{"MAIN"}$$



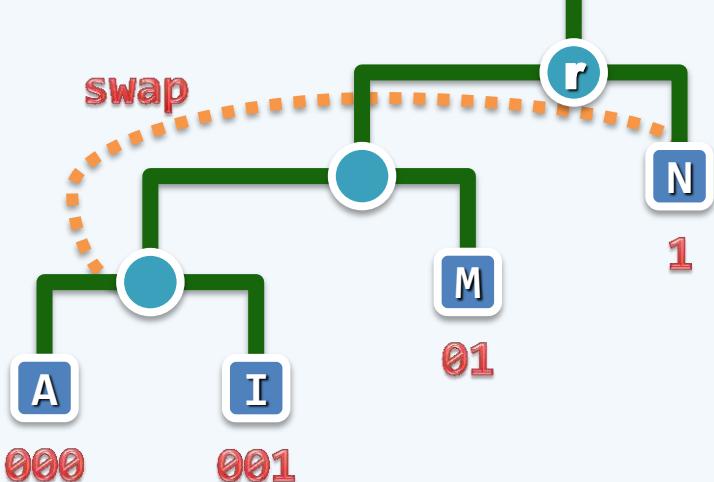
$$\text{ald}(T) * 4 = 2+2+2+2 = 8$$

$$\text{"01}^10_11^00" = \text{"MAIN"}$$



$$\text{ald}(T) * 4 = 2+3+3+1 = 9$$

$$\text{"01}^000_001^1" = \text{"MAIN"}$$



❖ 特别地，完全树即是最优编码树

❖ 实际上，字符的出现频率不尽相同，例如  $w('E') \gg w('Z')$

## 带权编码长度 vs. 叶节点平均带权深度

❖ 已知各字符的 **期望频率**，如何构造最优编码树？

❖ 文件长度  $\propto$  平均带权深度

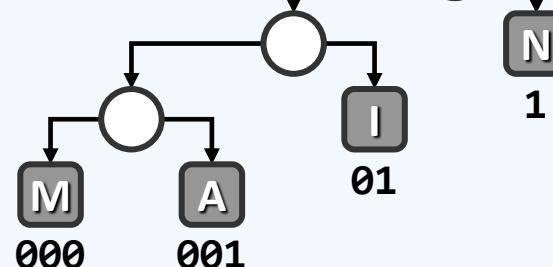
$$= \text{wald}(T) = \sum_x \text{rps}(x) \times w(x)$$

❖ 此时，完全树 **未必** 就是最优编码树

比如，考查 "mamani" 和 "mammamia" ...

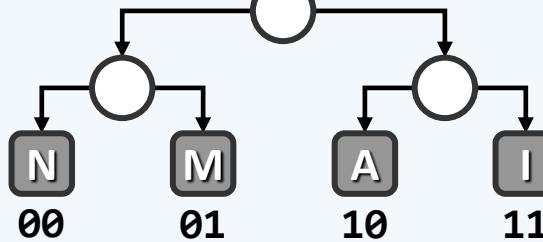
$$| "00000100000101" | = 15$$

$$| "00000100000000100001001" | = 23$$



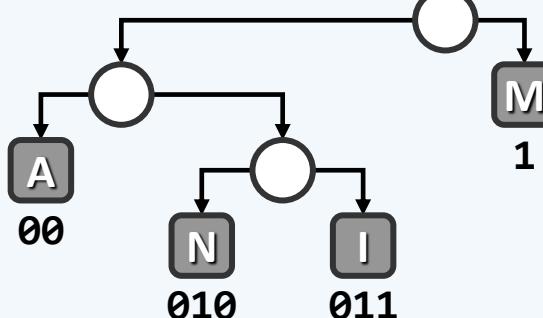
$$| "011001100011" | = 12$$

$$| "011001011001110" | = 16$$



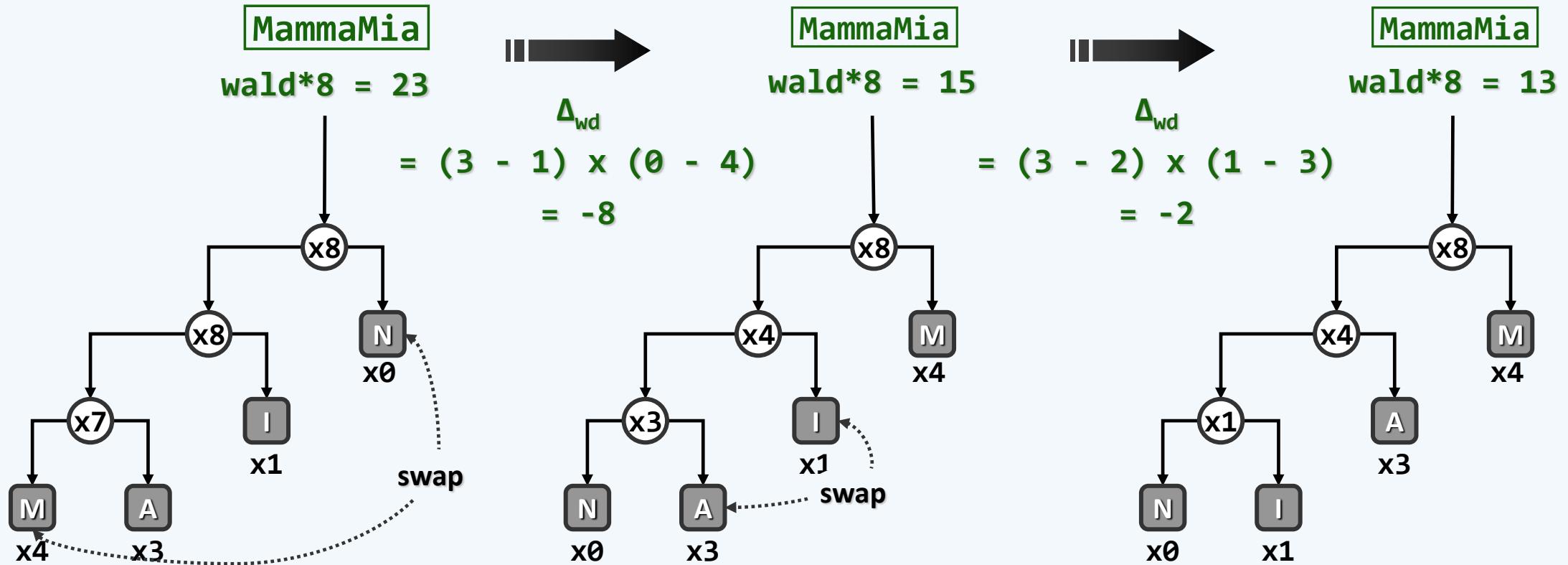
$$| "100100010011" | = 12$$

$$| "100100101100" | = 13$$



## 最优带权编码树

- 同样，频率高/低的（超）字符，应尽可能放在高/低处
- 故此，通过交换，同样可以缩短 $\text{wald}(T)$



## 5. 二叉树

Huffman 编码树  
算法

两年的时间，在你看来，也许就是一眨眼的功夫，对不对？可对我来说，它实在长得没边。我用不着为两年后的事情操心。

邓俊辉

deng@tsinghua.edu.cn

## 自底而上，逐层构造

// 贪婪策略：频率低的字符优先引入，从而使其位置更低

为每个字符创建一棵单节点的树，组成森林F

按照出现频率，对所有树排序

while ( F中的树不止一棵 )

取出频率最小的两棵树： $T_1$ 和 $T_2$

将它们合并成一棵新树T，并令：

$lchild(T) = T_1$  且  $rchild(T) = T_2$

$w(\ root(T) ) = w(\ root(T_1) ) + w(\ root(T_2) )$

// 非常幸运，如此得到的，的确是最优编码树之一

## 5. 二叉树

Huffman 编码树

正确性

邓俊辉

deng@tsinghua.edu.cn

## 正确性？

❖ 贪婪策略？

在多数场合并不适用

不见得能得到最优解

甚至反而得到**最差解**

//比如，最短路径

❖ Huffman树的构造采用了贪婪策略，它是最优编码树？总是？

❖ 易见：任一指定频率的字符集，都存在对应的最优编码树

❖ 然而，最优编码树可能**不止**一棵

❖ 断言：Huffman树必是其中**之一**

//为什么？

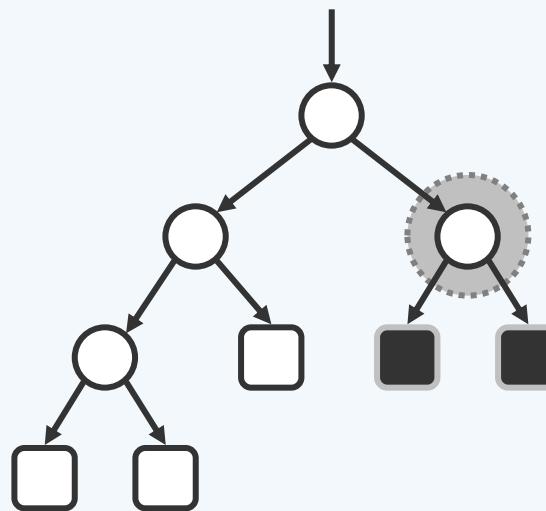
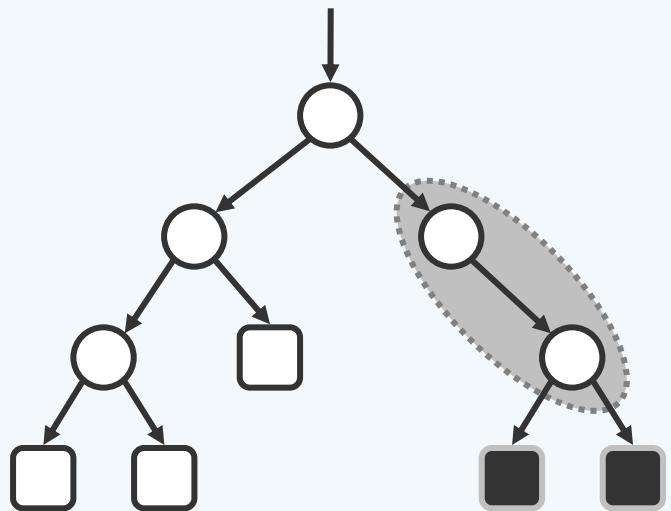
❖ 不妨，先来考察最优编码树的特性...

**双子性**

- 只要  $|\Sigma| > 1$ ，最优编码树中每一内部节点都有两个孩子，亦即  
节点度数均为偶数（0或2）

Huffman树必为**真二叉树**

- 否则，将1度节点替换为其唯一的孩子，则新树的wald将更小



## 不唯一性

- 任一内部节点的左、右子树相互交换之后，`wald`不变  
//上述算法中，左右子树的次序可以随机选取，故此...

- 为消除这种歧义，可以（比如）明确要求**左子树的频率更低**
- 不过，倘若它们（甚至更多节点）的频率恰好相等...

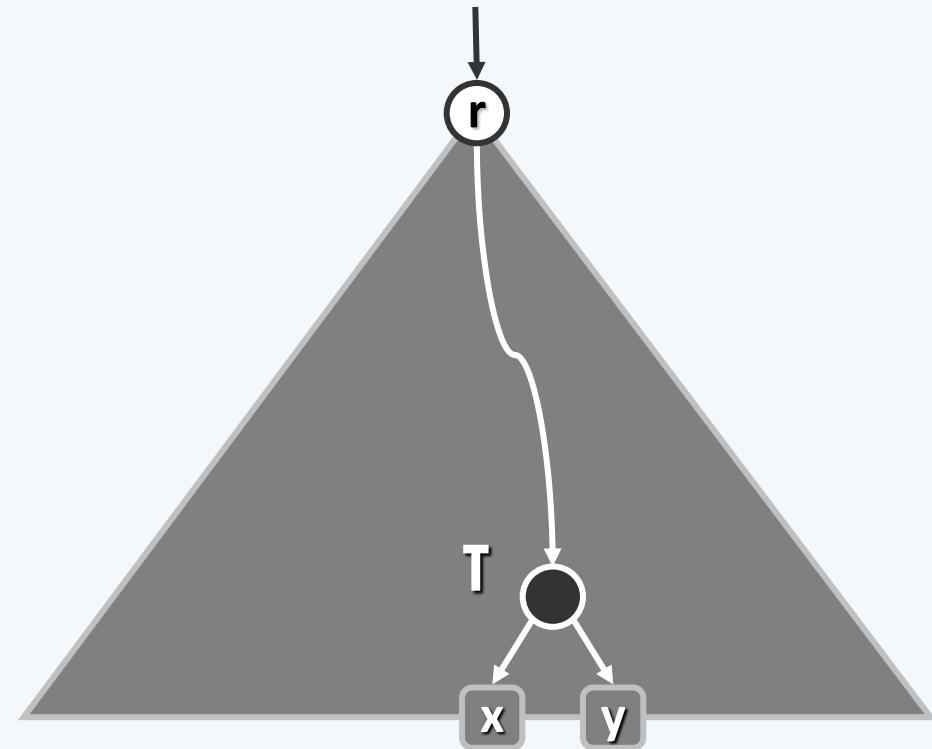


## 层次性

❖ 若：在字符表中， $x$ 和 $y$ 是出现频率最低的两个字符

则：存在某棵最优编码树， $x$ 和 $y$ 在其中处于最底层，且互为兄弟

❖ 为什么？



## 层次性

❖ 任取一棵最优编码树

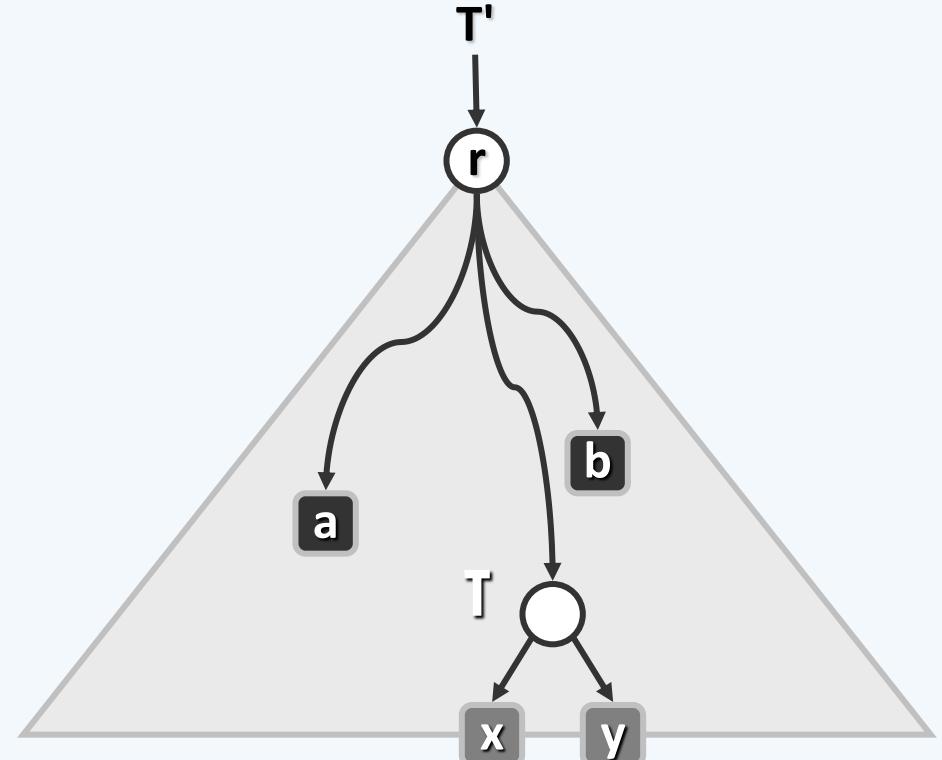
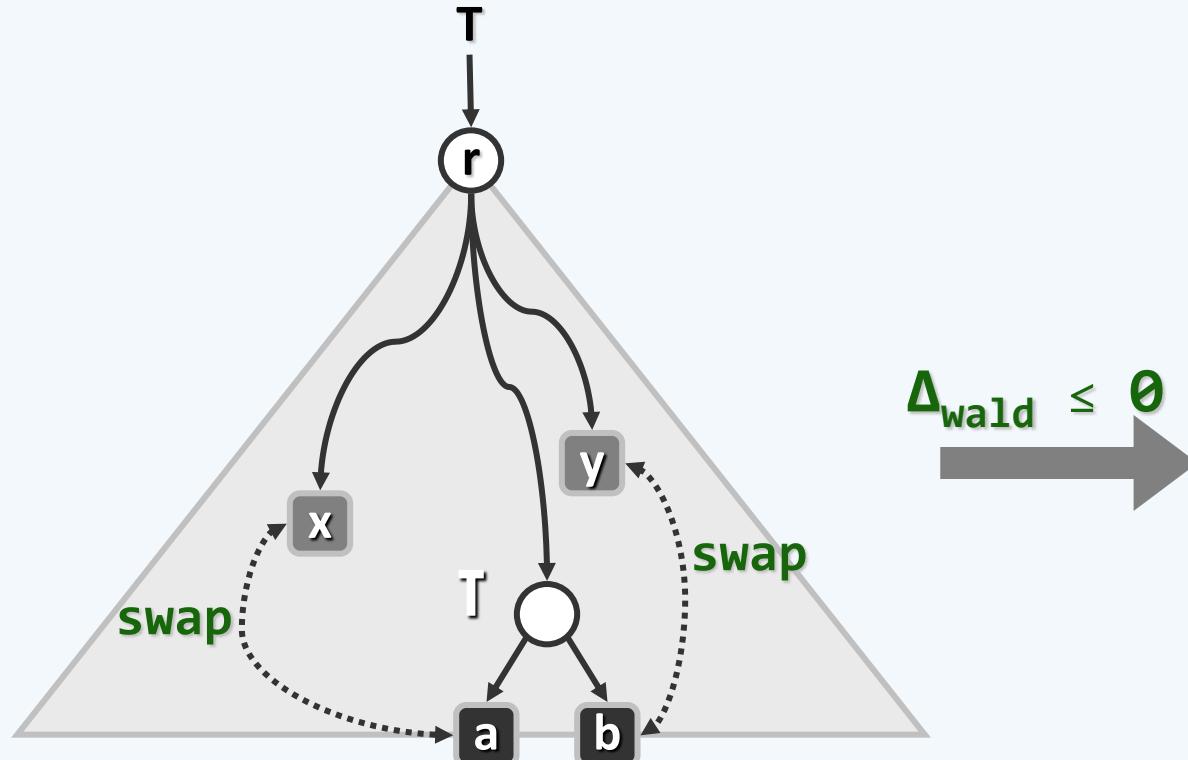
在其最底层，任取一对兄弟 **a** 和 **b**

交换 **a** 和 **x**，交换 **b** 和 **y** 之后，wald 绝不会增加

// 注意 T 的存在性

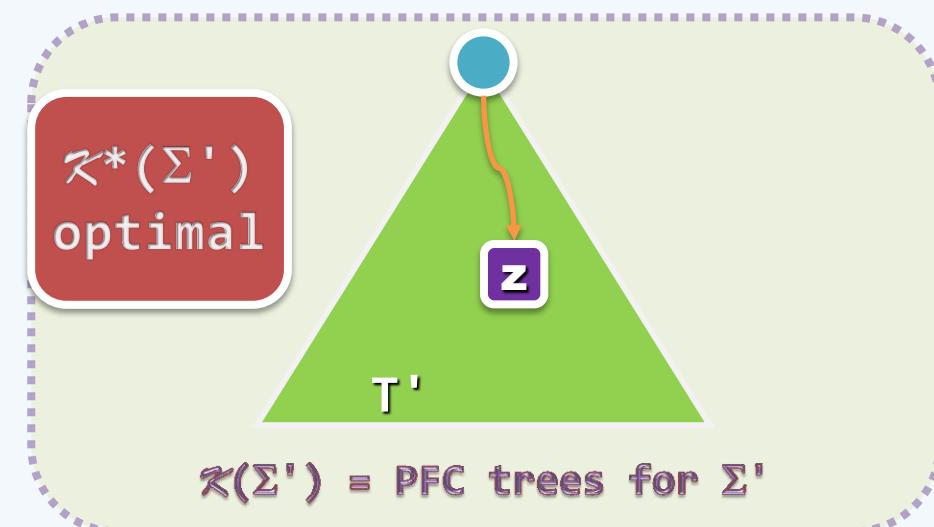
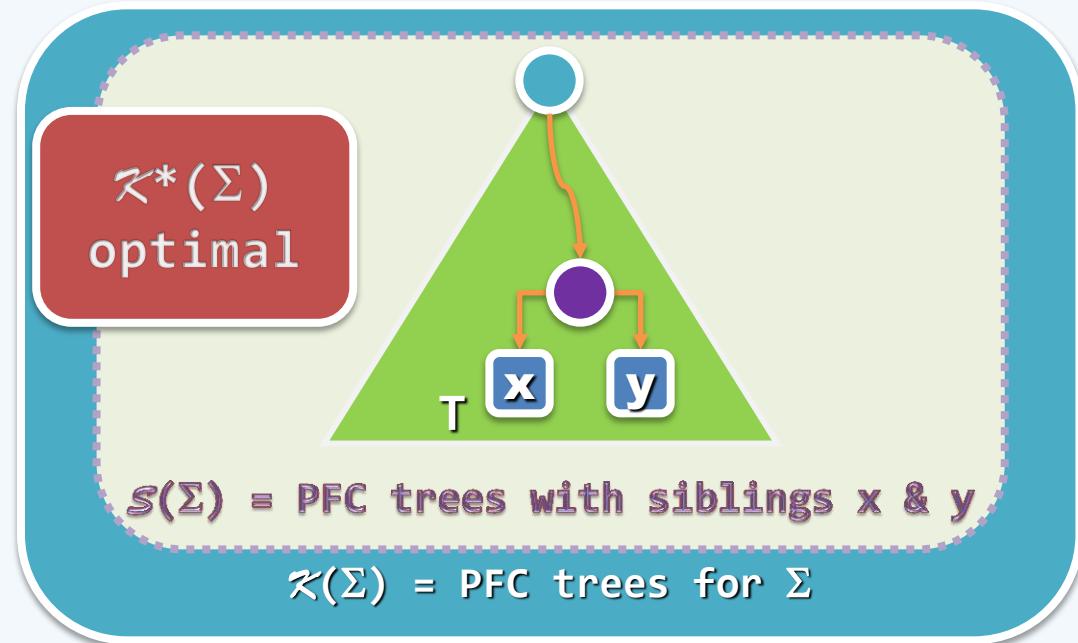
// 同样，注意其存在性

// 正如此前已看到的



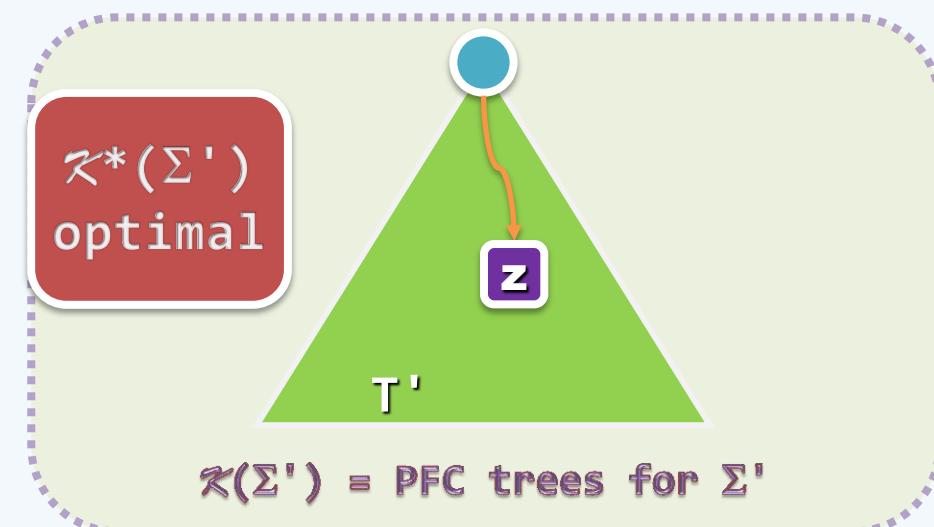
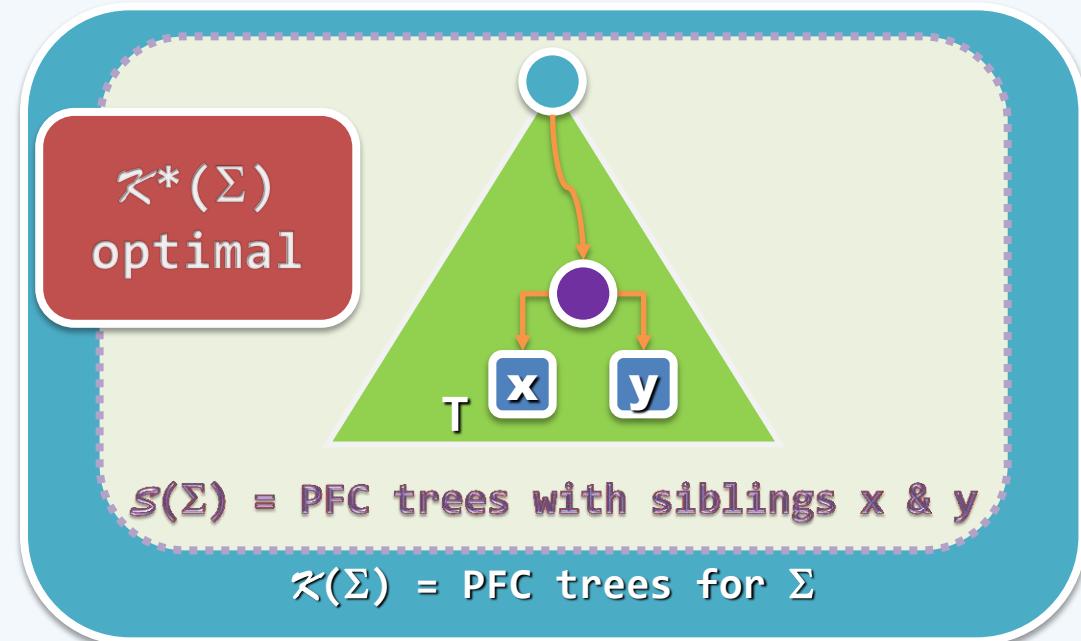
## 正确性

- ❖ Huffman ( 算法所生成的 ) 编码树 , 的确最优 !
- ❖ 对  $|\Sigma|$  做归纳 :  $|\Sigma| = 1$  时显然  
设  $|\Sigma| < n$  时 Huffman 算法都能最优编码 , 考虑  $|\Sigma| = n$  的情况 ...



## 正确性

- ❖ 取 $\Sigma$ 中频率最低的x和y //由层次性，仅考虑其互为兄弟的情形
- ❖ 令 $\Sigma' = (\Sigma \setminus \{x, y\}) \cup \{z\}$ ,  $w(z) = w(x) + w(y)$

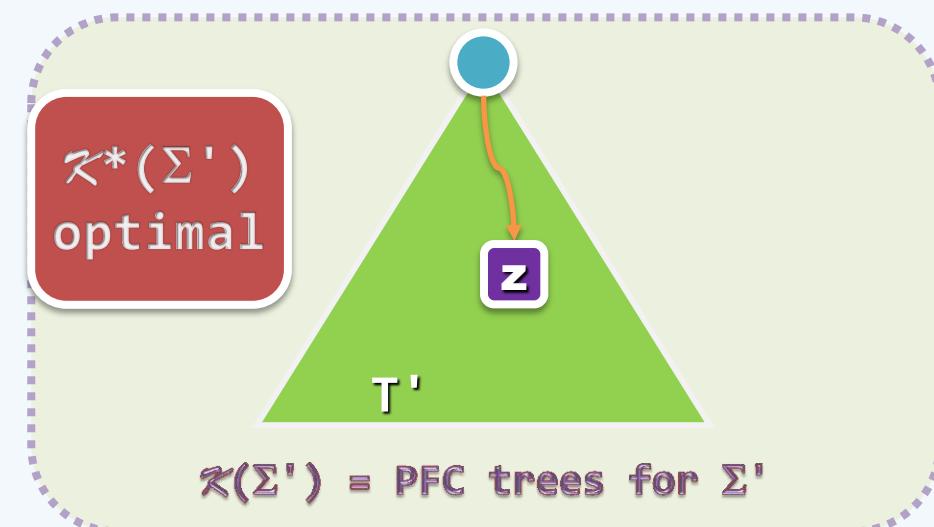
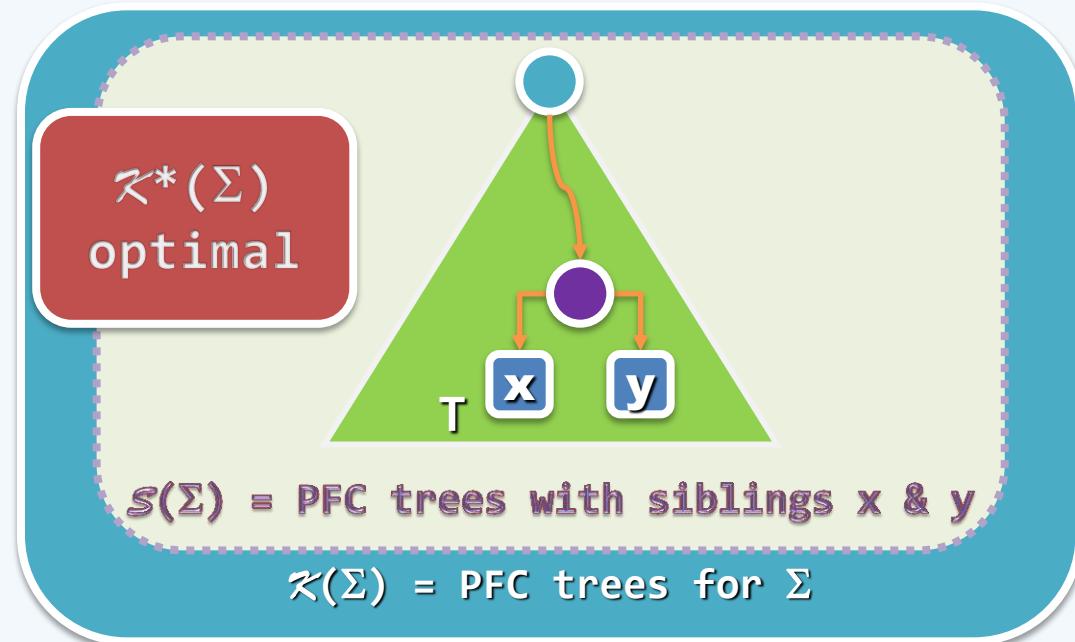


## 正确性

◆ 对于 $\Sigma'$ 的任一编码树 $T'$ ，只要为 $z$ 添加孩子 $x$ 和 $y$ ，即可

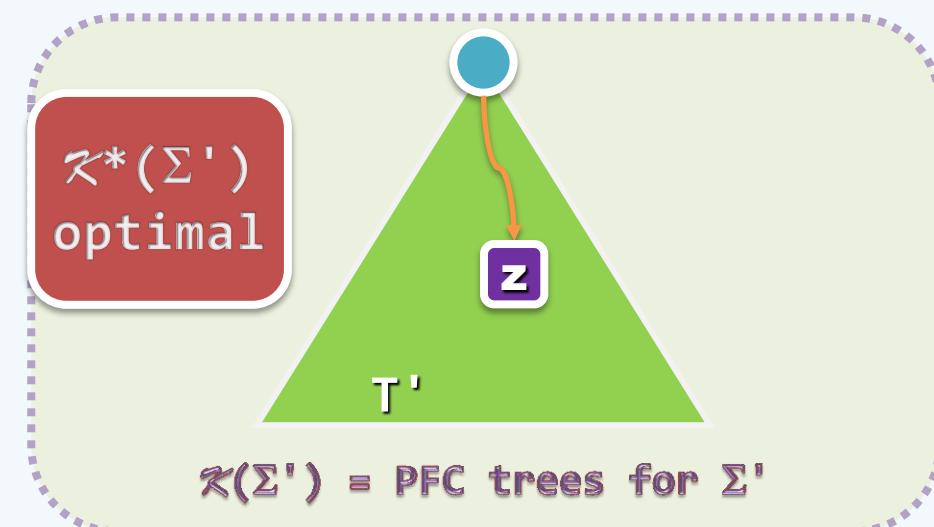
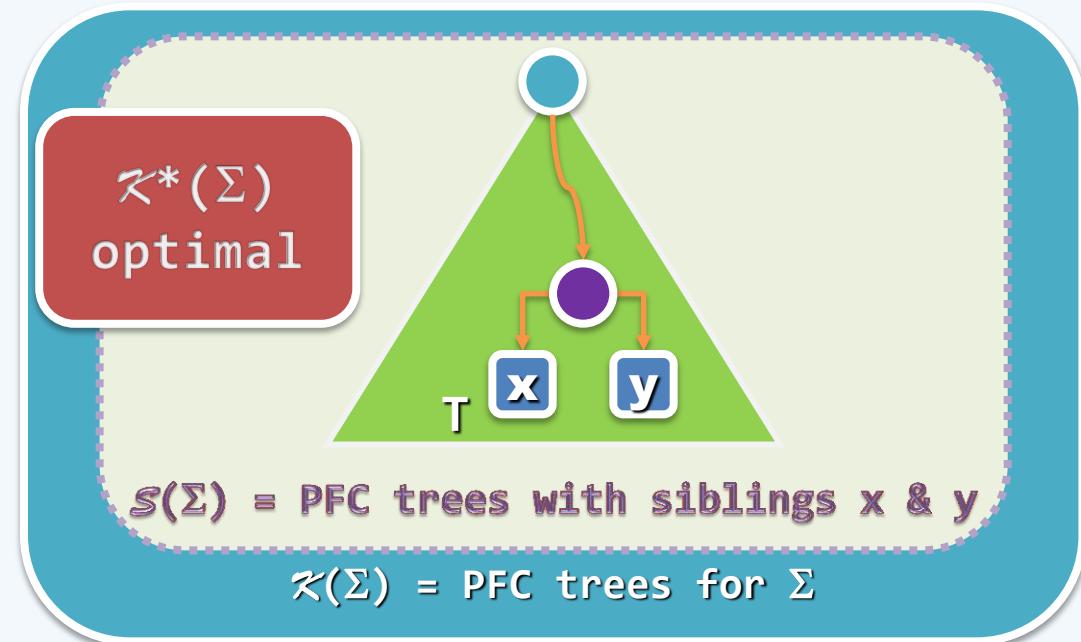
得到 $\Sigma$ 的一棵编码树 $T$ ，且  $wd(T) - wd(T') = w(x) + w(y) = w(z)$

◆ 亦即，如此对应的 $T$ 和 $T'$ ， $wd$ 之差与 $T$ 的具体形态无关



## 正确性

- ❖ 因此，只要  $T'$  是  $\Sigma'$  的最优编码树，则  $T$  也必是  $\Sigma$  的最优编码树（之一）
- ❖ 实际上，Huffman 算法的过程，与上述归纳过程完全一致



## 5. 二叉树

Huffman 编码树

算法实现

邓俊辉

deng@tsinghua.edu.cn

## Huffman树与森林

- ❖ `#define N_CHAR (0x80 - 0x20) //仅以可打印字符为例`
- ❖ `struct HuffChar { //Huffman(超)字符  
 char ch; int weight; //字符、频率  
 HuffChar ( char c = '^', int w = 0 ) : ch ( c ), weight ( w ) {};  
 bool operator< ( HuffChar const& hc ) //比较器  
 { return weight > hc.weight; } //此处故意大小颠倒  
 bool operator== ( HuffChar const& hc ) //判等器  
 { return weight == hc.weight; }  
};`
- ❖ `#define HuffTree BinTree< HuffChar > //Huffman树，节点类型HuffChar`
- ❖ `typedef List< HuffTree * > HuffForest; //Huffman森林`

## 构造编码树

```
❖ HuffTree* generateTree( HuffForest * forest ) { //Huffman编码算法  
    while ( 1 < forest->size() ) { //反复迭代，直至森林中仅含一棵树  
        HuffTree *T1 = minHChar( forest ), *T2 = minHChar( forest );  
        HuffTree *S = new HuffTree(); //创建新树，准备合并T1和T2  
        S->insertAsRoot( HuffChar( '^' ), //根节点权重，取作T1与T2之和  
                           T1->root()->data.weight + T2->root()->data.weight );  
        S->attachAsLC( S->root(), T1 ); S->attachAsRC( S->root(), T2 );  
        forest->insertAsLast( S ); //T1与T2合并后，重新插回森林  
    } //assert: 循环结束时，森林中唯一的那棵树即Huffman编码树  
    return forest->first()->data; //故直接返回之  
}
```

## 查找最小超字符

❖ Huffman编码的整体效率，直接决定于minHChar()的效率

以下版本仅达到 $\mathcal{O}(n)$ ，整体为 $\mathcal{O}(n^2)$

```
❖ HuffTree* minHChar( HuffForest * forest ) {  
    ListNodePosi( HuffTree* ) p = forest->first(); //从首节点出发  
    ListNodePosi( HuffTree* ) minChar = p; //记录最小树的位置及其  
    int minWeight = p->data->root()->data.weight; //对应的权重  
    while (forest->valid(p = p->succ)) //遍历所有节点  
        if( minWeight > p->data->root()->data.weight ) { //如必要  
            minWeight = p->data->root()->data.weight; minChar = p; //则更新记录  
        }  
    return forest->remove( minChar ); //从森林中摘除该树，并返回  
}
```

## 构造编码表

```

❖ #include "../Hashtable/Hashtable.h" //用HashTable(第9章)实现

typedef Hashtable< char, char* > HuffTable; //Huffman编码表

❖ static void generateCT //通过遍历获取各字符的编码

( Bitmap* code, int length, HuffTable* table, BinNodePosi(HuffChar) v ) {

if ( IsLeaf( * v ) ) //若是叶节点(还有多种方法可以判断)

{ table->put( v->data.ch, code->bits2string( length ) ); return; }

if ( HasLChild( * v ) ) //Left = 0, 深入遍历

{ code->clear(length); generateCT( code, length + 1, table, v->lc ); }

if ( HasRChild( * v ) ) //Right = 1

{ code->set(length); generateCT( code, length + 1, table, v->rc ); }

} //总体O(n)

```

## 5. 二叉树

Huffman 编码树

改进

邓俊辉

deng@tsinghua.edu.cn

## 优先级队列

### ❖ 方案1， $\mathcal{O}(n^2)$

- 初始化时，通过**排序**得到一个**非升序向量**  $\mathcal{O}(n \log n)$
- 每次（从**后端**）取出频率最低的两个节点  $\mathcal{O}(1)$
- 将合并得到的新树插入向量，并保持有序  $\mathcal{O}(n)$

### ❖ 方案2， $\mathcal{O}(n^2)$

- 初始化时，通过**排序**得到一个**非降序列表**  $\mathcal{O}(n \log n)$
- 每次（从**前端**）取出频率最低的两个节点  $\mathcal{O}(1)$
- 将合并得到的新树插入列表，并保持有序  $\mathcal{O}(n)$

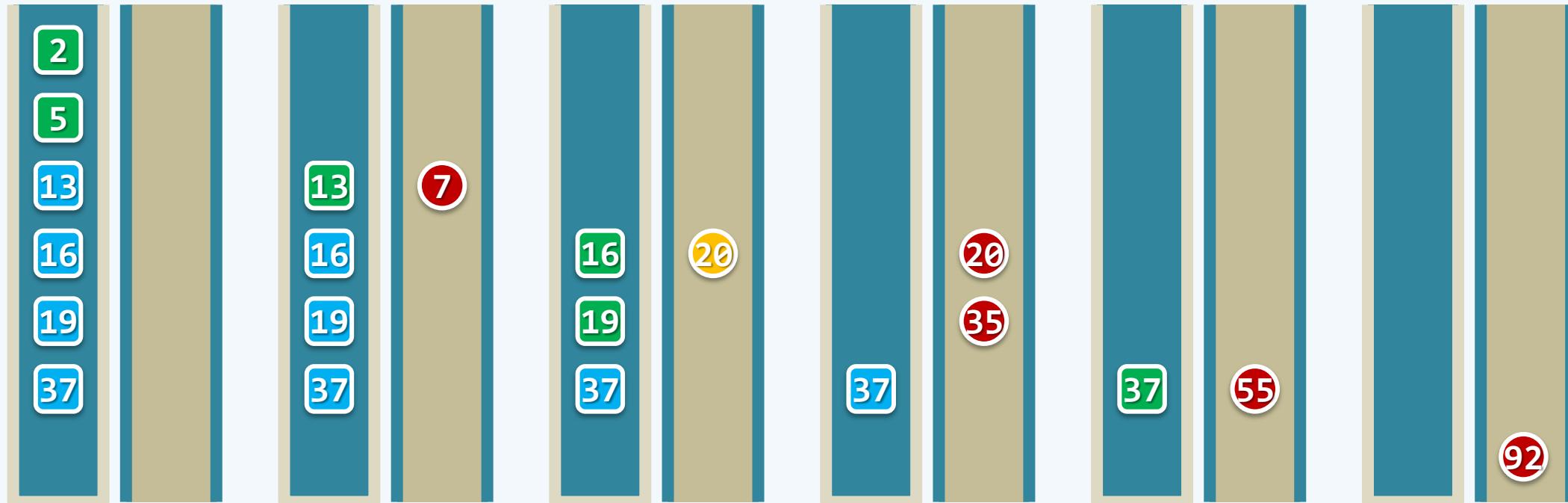
### ❖ 方案3， $\mathcal{O}(n \log n)$ //稍后第10章...保持兴趣

- 初始化时，将所有树组织为一个**优先级队列**  $\mathcal{O}(n)$
- 取出频率最低的两个节点，合并得到的新树插入队列  $\mathcal{O}(\log n) + \mathcal{O}(\log n)$

预排序  $\times$  (栈 + 队列)

## ◆ 方案4

- 所有字符按频率排序，构成一个栈  $\text{// } \theta(n \log n)$
- 维护另一个有序队列...  $\text{// } \theta(n)$



## 6. 图

概述

邓俊辉

deng@tsinghua.edu.cn

## 基本术语

❖  $G = (V; E)$

**vertex**:  $n = |V|$

**edge** | **arc**:  $e = |E|$

❖ 同一条边的两个顶点，彼此邻接 **adjacency**

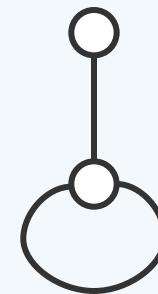
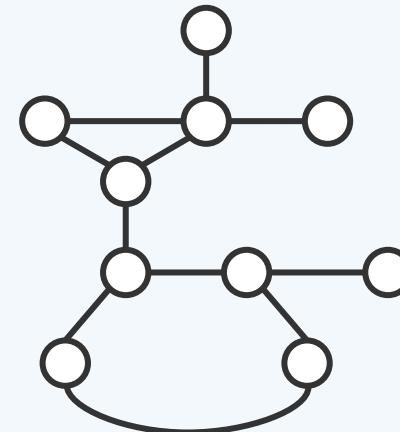
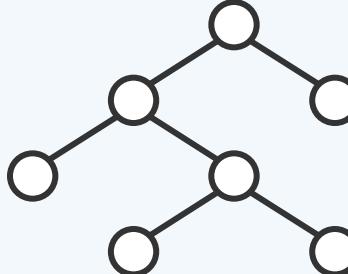
同一顶点自我邻接，构成自环 **self-loop**

不含自环，即为简单图 **simple graph**

非简单图 **non-simple**，暂不讨论

❖ 顶点与其所属的边，彼此关联 **incidence**

度 (**degree** / **valency**) : 与同一顶点关联的边数



## 无向图 + 有向图

❖ 若邻接顶点 $u$ 和 $v$ 的次序无所谓

则 $(u, v)$ 为无向边 undirected edge

❖ 所有边均无方向的图，即无向图 undigraph

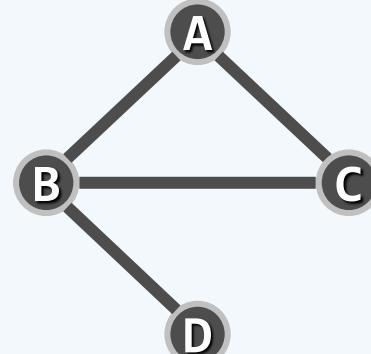
❖ 反之，有向图 digraph 中均为有向边 directed edge

$u$ 、 $v$ 分别称作边 $(u, v)$ 的尾 ( tail )、头 ( head )

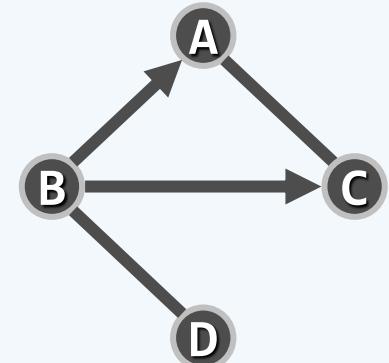
❖ 无向边、有向边并存的图，称作混合图 mixed graph

❖ 有向图通用性更强

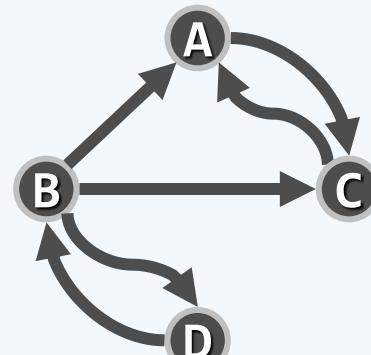
故本章主要针对有向图介绍相关结构及算法



(a) undigraph



(b) mixed graph



(c) digraph

## 路径 + 环路

❖ 路径  $\pi = \langle v_0, v_1, \dots, v_k \rangle$

长度  $|\pi| = k$

❖ 简单路径：

$v_i = v_j$  除非  $i = j$

❖ 环/环路： $v_0 = v_k$

❖ 有向无环图 (DAG)

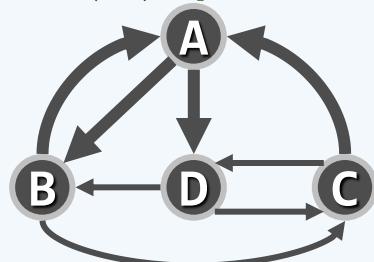
❖ 欧拉环路： $|\pi| = |E|$

各边恰好出现一次

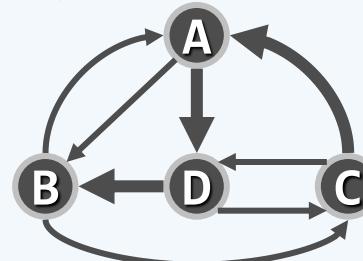
❖ 哈密尔顿环路： $|\pi| = |V|$

各顶点恰好出现一次

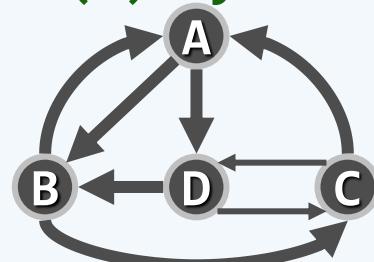
(i) path



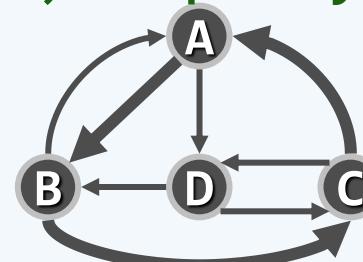
(ii) simple path



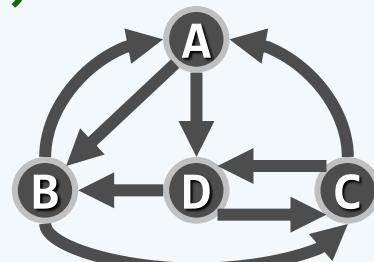
(i) cycle



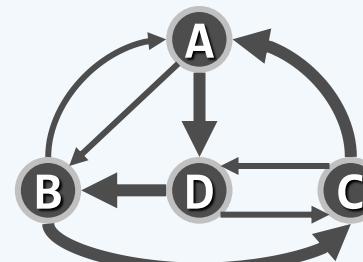
(ii) simple cycle



(i) Eulerian tour

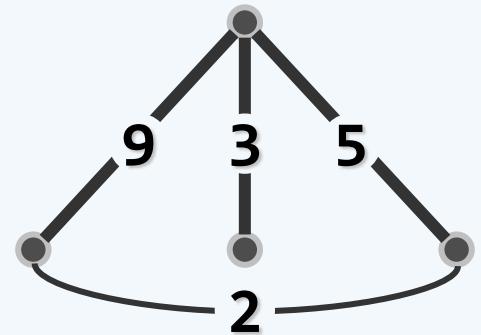


(ii) Hamiltonian tour

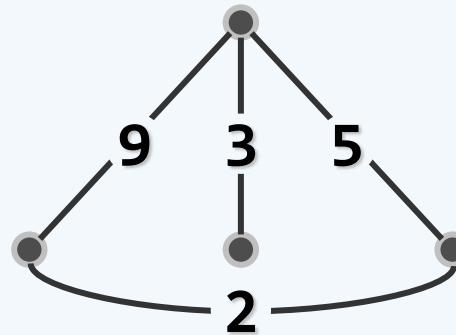


## 支撑树 + 带权网络 + 最小支撑树

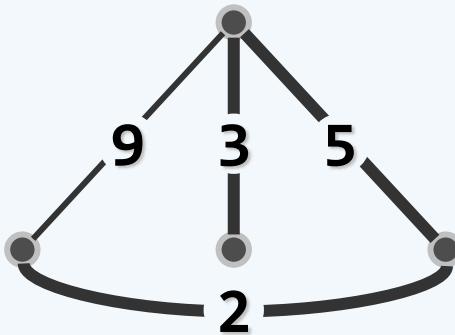
- ❖ 图 $G = (V; E)$ 的子图 $T = (V; F)$ 若是树，即为其支撑树spanning tree
- 同一图的支撑树，通常并不唯一
- ❖ 各边 $e$ 均有对应的权值 $\text{wt}(e)$ ，则为带权网络weighted network
- ❖ 同一网络的支撑树中，总权重最小者为最小支撑树MST



spanning tree



weighted network  
(triangle inequality?)



minimum spanning tree

## 6. 图

邻接矩阵  
构思

邓俊辉

deng@tsinghua.edu.cn

## Graph模板类

❖ template <typename Tv, typename Te> class Graph { //顶点类型、边类型  
private:

```
void reset() { //所有顶点、边的辅助信息复位
    for ( int i = 0; i < n; i++ ) { //顶点
        status(i) = UNDISCOVERED; dTime(i) = fTime(i) = -1;
        parent(i) = -1; priority(i) = INT_MAX;
    }
    for ( int j = 0; j < n; j++ ) //边
        if ( exists(i, j) ) type(i, j) = UNDETERMINED;
}
```

public: /\* ... 顶点操作、边操作、图算法：无论如何实现，接口必须统一 ... \*/  
} //Graph

## 邻接矩阵 + 关联矩阵

❖ **adjacency matrix** : 用二维矩阵记录顶点之间的邻接关系

一一对应：矩阵元素  $\leftrightarrow$  图中可能存在的边

$A(i, j) = 1$       若顶点*i*与*j*之间存在一条边

$= 0$       否则

既然只考察简单图，对角线统一设置为0

空间复杂度为 $\Theta(n^2)$ ，与图中实际的边数无关

❖ **incidence matrix** : 用二维矩阵记录顶点与边之间的关联关系

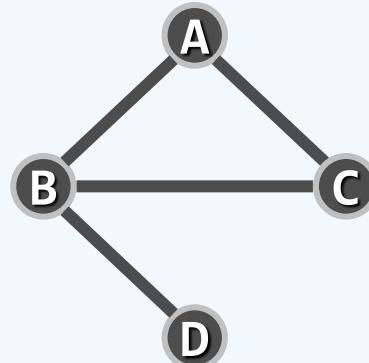
空间复杂度为 $\Theta(n * e) = \Theta(n^3)$

空间利用率  $< 2/n$

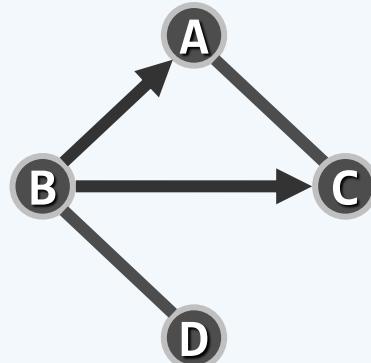
解决某些问题时十分有效

## 实例

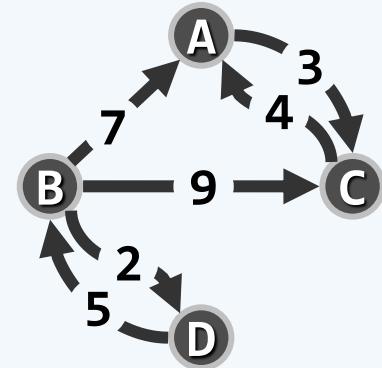
(a) undigraph



(b) digraph



(c) network



0	A	B	C	D
A		1	1	
B	1		1	1
C	1	1		
D		1		

redundancy

0	A	B	C	D
A			1	
B	1		1	1
C	1			
D		1		

∞	A	B	C	D
A			3	
B	7		9	2
C	4			
D		5		

## 6. 图

邻接矩阵  
模板实现

邓俊辉

deng@tsinghua.edu.cn

**Vertex**

```
❖ typedef enum { UNDISCOVERED, DISCOVERED, VISITED } VStatus;  
  
❖ template <typename Tv> struct Vertex { //顶点对象（并未严格封装）  
    Tv data; int inDegree, outDegree; //数据、出入度数  
    VStatus status; //（如上三种）状态  
    int dTime, fTime; //时间标签  
    int parent; //在遍历树中的父节点  
    int priority; //在遍历树中的优先级（最短通路、极短跨边等）  
    Vertex( Tv const & d ) : //构造新顶点  
        data( d ), inDegree( 0 ), outDegree( 0 ), status( UNDISCOVERED ),  
        dTime( -1 ), fTime( -1 ), parent( -1 ), priority( INT_MAX ) {}  
};
```

**Edge**❖ **typedef**

```
enum { UNDETERMINED, TREE, CROSS, FORWARD, BACKWARD }
```

EType;

❖ **template <typename Te> struct Edge { //边对象（并未严格封装）**

Te data; //数据

int weight; //权重

EType type; //在遍历树中所属的类型

Edge( Te const & d, int w ) : //构造新边

data(d), weight(w), type(UNDETERMINED) {}

};

## GraphMatrix

```
❖ template <typename Tv, typename Te> class GraphMatrix : public Graph<Tv, Te> {
    private:
```

```
    Vector< Vertex<Tv> > V; //顶点集
```

```
    Vector< Vector< Edge<Te>*> > E; //边集
```

```
    public:
```

```
    /* 操作接口：顶点相关、边相关、... */
```

```
GraphMatrix() { n = e = 0; } //构造
```

```
~GraphMatrix() { //析构
```

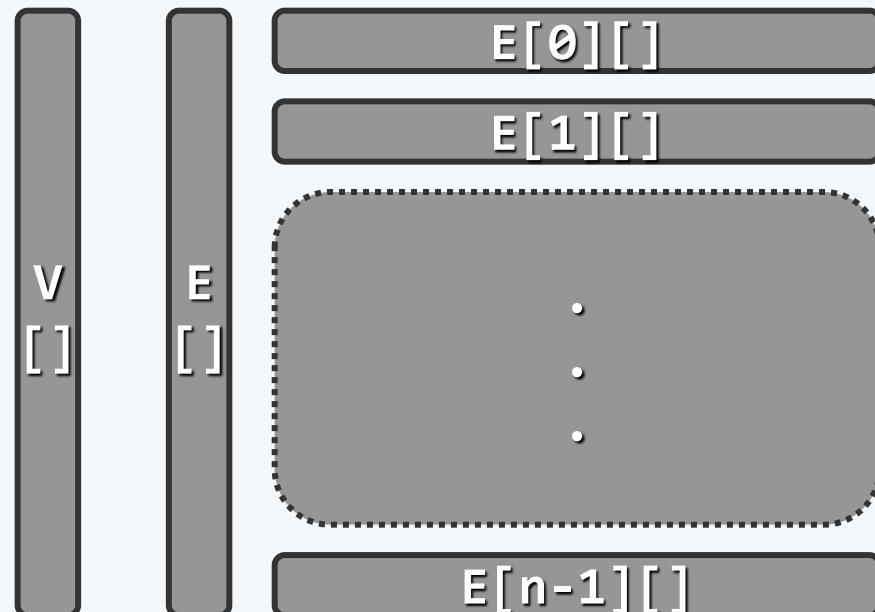
```
    for (int j = 0; j < n; j++)
```

```
        for (int k = 0; k < n; k++)
```

```
            delete E[j][k]; //清除所有动态申请的边记录
```

```
}
```

```
};
```



## 6. 图

邻接矩阵  
简单接口

邓俊辉

deng@tsinghua.edu.cn

## 顶点的读写

```
❖ Tv & vertex(int i) { return V[i].data; } //数据  
  
int inDegree(int i) { return V[i].inDegree; } //入度  
  
int outDegree(int i) { return V[i].outDegree; } //出度  
  
vStatus & status(int i) { return V[i].status; } //状态  
  
int & dTime(int i) { return V[i].dTime; } //时间标签dTime  
  
int & fTime(int i) { return V[i].fTime; } //时间标签fTime  
  
int & parent(int i) { return V[i].parent; } //在遍历树中的父亲  
  
int & priority(int i) { return V[i].priority; } //优先级数
```

## 邻点的枚举

❖ 对于任意顶点*i*，如何枚举其所有的邻接顶点neighbor？

❖ int nextNbr( int i, int j ) { //若已枚举至邻居j，则转向下一邻居

```
while ( ( -1 < j ) && ! exists( i, --j ) ); //逆向顺序查找，O(n)
```

```
return j;
```

} //改用邻接表可提高至O(1 + outDegree(i))

❖ int firstNbr( int i ) {

```
return nextNbr( i, n ); //假想哨兵
```

} //首个邻居

## 边的读写

- ❖ `bool exists( int i, int j ) { //判断边(i, j)是否存在 (短路求值)`  
`return (0 <= i) && (i < n) && (0 <= j) && (j < n) && E[i][j] != NULL;`  
`} //以下假定exists(i, j)...`
- ❖ `Te & edge( int i, int j ) //边(i, j)的数据`  
`{ return E[i][j]->data; } //O(1)`
- ❖ `EType & type( int i, int j ) //边(i, j)的类型`  
`{ return E[i][j]->type; } //O(1)`
- ❖ `int & weight( int i, int j ) //边(i, j)的权重`  
`{ return E[i][j]->weight; } //O(1)`

## 6. 图

邻接矩阵  
复杂接口

邓俊辉

deng@tsinghua.edu.cn

## 边的插入

```
❖ void insert( Te const & edge, int w, int i, int j ) { //插入(i, j, w)  
    if ( exists(i, j) ) return; //忽略已有的边  
    E[i][j] = new Edge<Te>( edge, w ); //创建新边  
    e++; //更新边计数  
  
    v[i].outDegree++; //更新关联顶点i的出度  
  
    v[j].inDegree++; //更新关联顶点j的入度  
}
```

## 边的删除

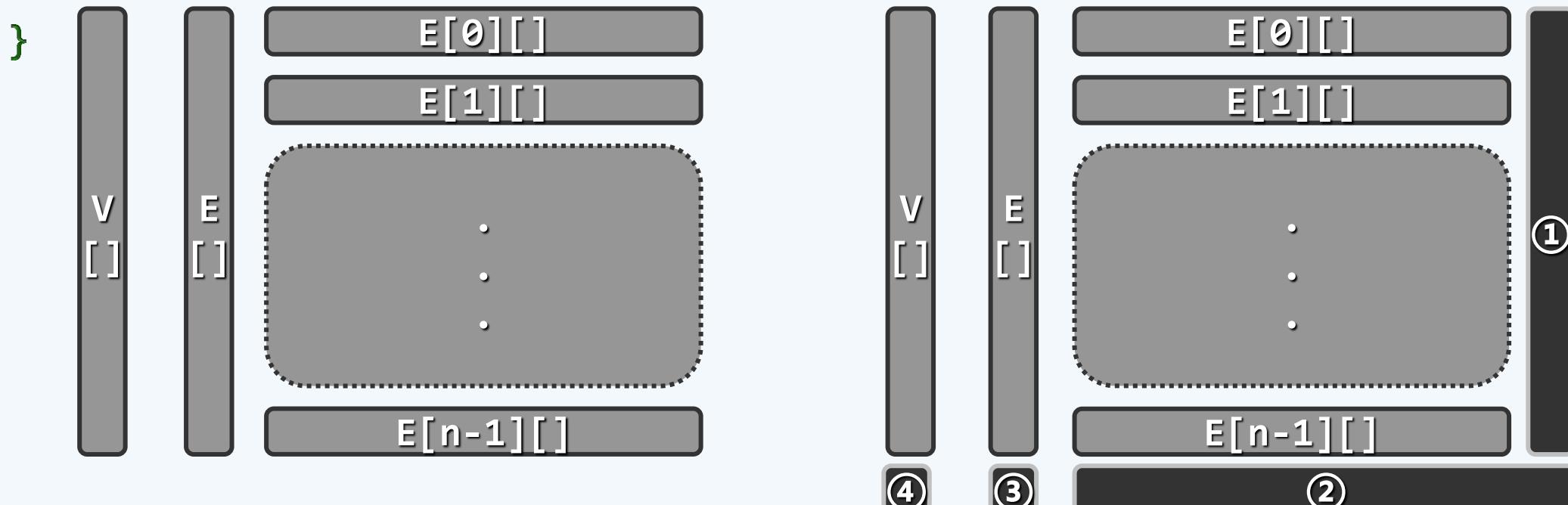
```
❖ Te remove( int i, int j ) { //删除顶点i和j之间的联边 ( exists(i, j) )  
    Te eBak = edge(i, j); //备份边(i, j)的信息  
  
    delete E[i][j]; E[i][j] = NULL; //删除边(i, j)  
  
    e--; //更新边计数  
  
    v[i].outDegree--; //更新关联顶点i的出度  
  
    v[j].inDegree--; //更新关联顶点j的入度  
  
    return eBak; //返回被删除边的信息  
}
```

## 顶点插入

```

❖ int insert( Tv const & vertex ) { //插入顶点，返回编号
    for ( int j = 0; j < n; j++ ) E[j].insert( NULL ); n++; //①
    E.insert( Vector< Edge<Te>*>( n, n, NULL ) ); //②③
    return V.insert( Vertex<Tv>( vertex ) ); //④
}

```



## 顶点删除

❖ Tv remove( int i ) { //删除顶点及其关联边，返回该顶点信息

    for ( int j = 0; j < n; j++ ) //删除所有出边

        if ( exists( i, j ) ) { delete E[i][j]; V[j].inDegree--; }

    E.remove(i); n--; //删除第i行

    Tv vBak = vertex( i ); V.remove( i ); //备份之后，删除顶点i

    for ( int j = 0; j < n; j++ ) //删除所有入边及第i列

        if ( Edge<Te> \* e = E[j].remove( i ) ) { delete e; V[j].outDegree--; }

    return vBak; //返回被删除顶点的信息

}

## 6. 图

邻接矩阵  
性能分析

邓俊辉

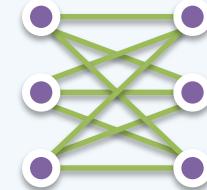
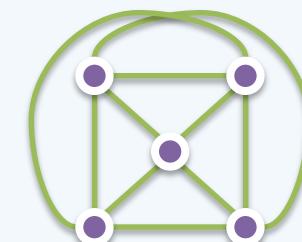
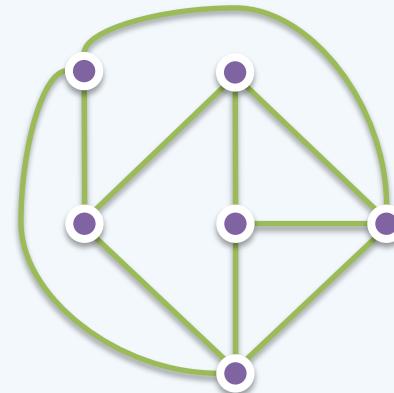
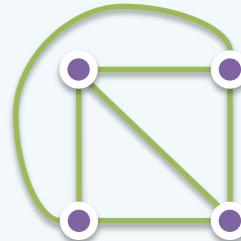
deng@tsinghua.edu.cn

## 优点

- ❖ 直观，易于理解和实现
- ❖ 适用范围广泛：digraph / network / cyclic / ...  
尤其适用于稠密图（dense graph）
- ❖ 判断两点之间是否存在联边： $O(1)$
- ❖ 获取顶点的（出/入）度数： $O(1)$   
添加、删除边后更新度数： $O(1)$
- ❖ 扩展性（scalability）：  
得益于Vector良好的空间控制策略  
空间溢出等情况可“透明地”予以处理

## 缺点

- ❖  $\Theta(n^2)$ 空间，与边数无关！
- ❖ 真会有这么多条边吗？不妨考察一类特定的图...
- ❖ 平面图（planar graph）：可嵌入于平面的图
- ❖ Euler's formula (1750)：
 
$$v - e + f - c = 1, \text{ for any PG}$$
- ❖ 平面图： $e \leq 3 \times n - 6 = O(n) \ll n^2$   
此时，空间利用率  $\approx 1/n$
- ❖ 稀疏图 sparse graph  
空间利用率同样很低，可采用压缩存储技术



6. 图

邻接表

邓俊辉

deng@tsinghua.edu.cn

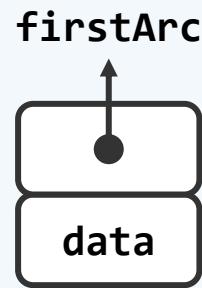
## 邻接表

❖ 如何避免关联矩阵的空间浪费？

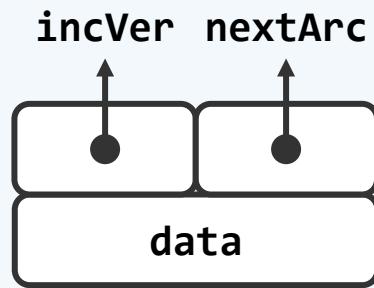
- 将关联矩阵的各行组织为列表
- 只记录存在的边

❖ 等效于，每一顶点v对应于列表

$$L_v = \{ u \mid \langle v, u \rangle \in E \}$$

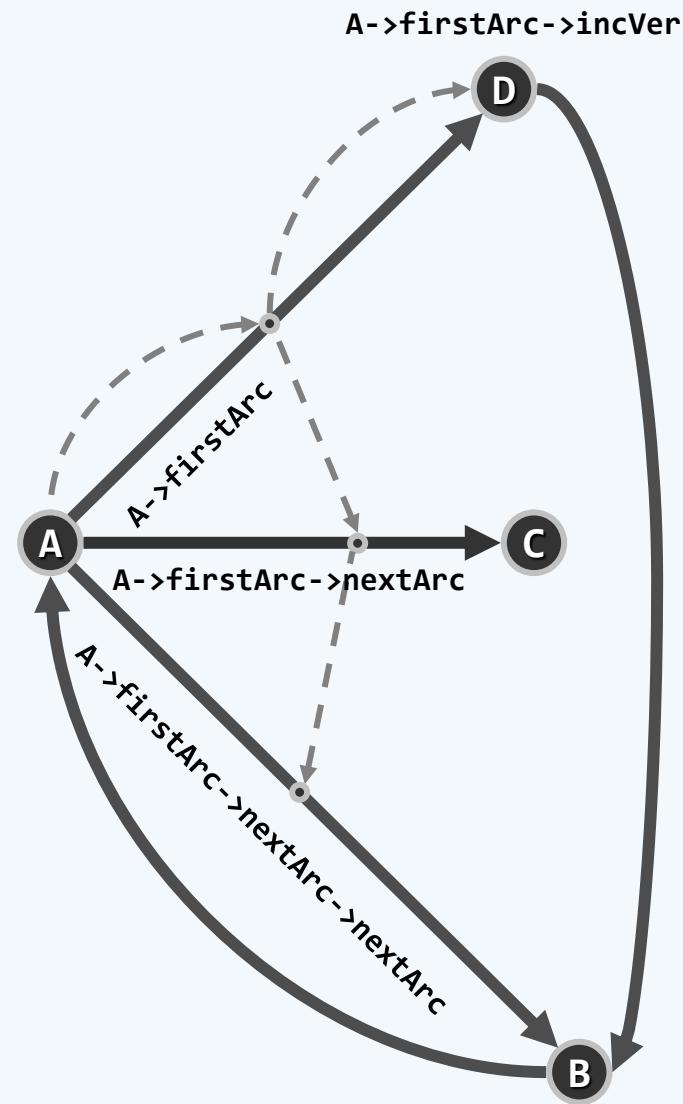


VNode



ArcNode

ALGraph



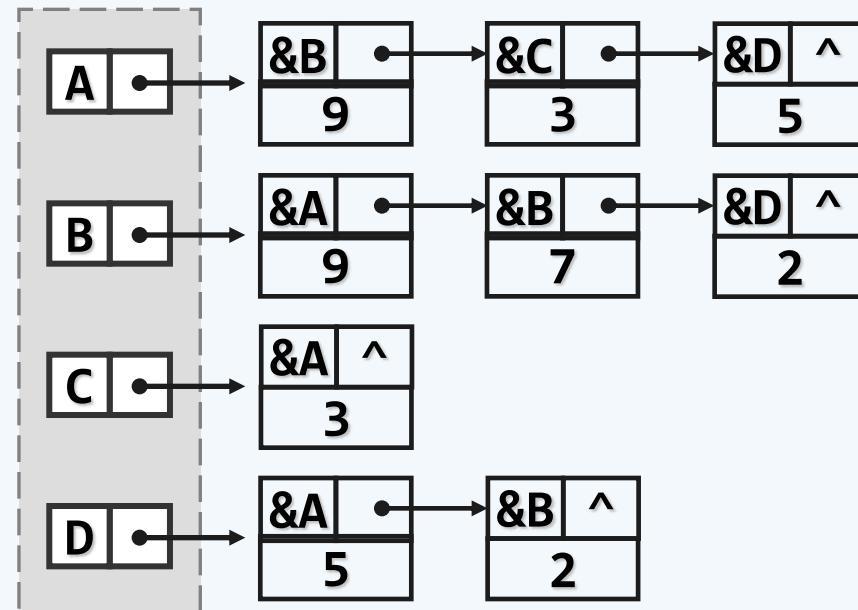
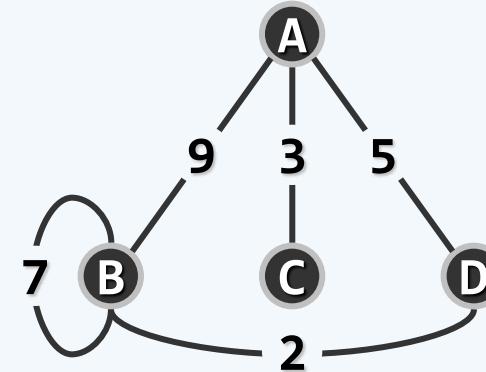
## 实例

❖ 4个顶点，5条弧

❖ 不必占用 $4 \times 4 = 16$ 个单元

但还是占用了9个单元，另加4个表头

$\infty$	A	B	C	D
A		9	3	5
B	9	7		2
C	3			
D	5	2		



## 空间复杂度

❖ 有向图 =  $\mathcal{O}(n + e)$

❖ 无向图 =  $\mathcal{O}(n + 2 \times e) = \mathcal{O}(n + e)$

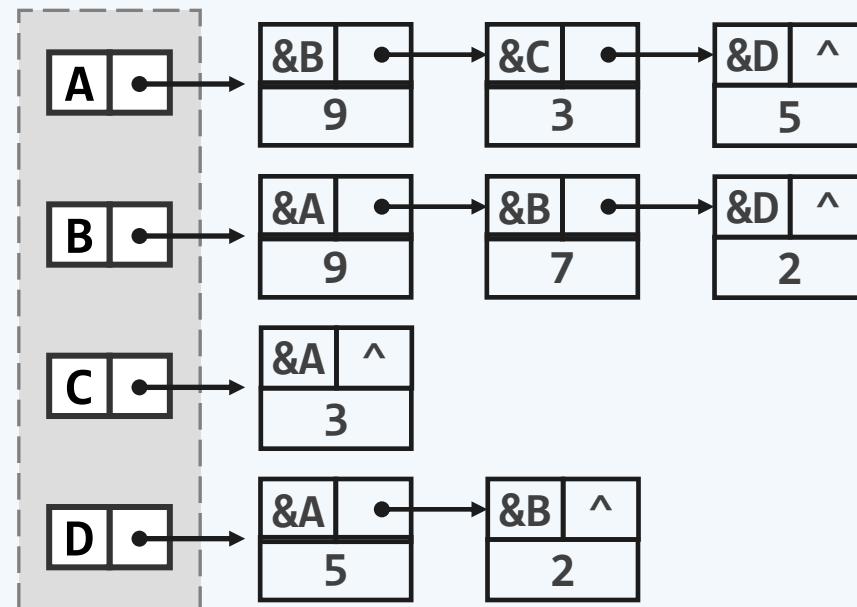
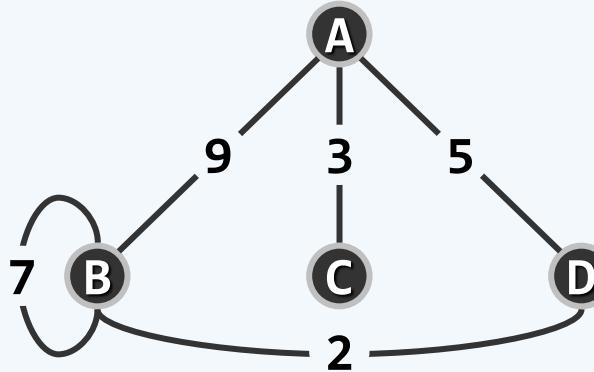
注意：无向弧被重复存储

问题：如何改进？

❖ 适用于稀疏图

❖ 平面图 =  $\mathcal{O}(n + 3 \times n) = \mathcal{O}(n)$

较之邻接矩阵，有极大改进



## 时间复杂度

- ❖ 建立邻接表（递增式构造）： $\mathcal{O}(n + e)$  //如何实现
- ❖ 枚举所有以顶点v为尾的弧： $\mathcal{O}(1 + \deg(v))$  //遍历v的邻接表
- ❖ 枚举（无向图中）顶点v的邻居： $\mathcal{O}(1 + \deg(v))$  //遍历v的邻接表
- ❖ 枚举所有以顶点v为头的弧： $\mathcal{O}(n + e)$  //遍历所有邻接表  
可改进至  $\mathcal{O}(1 + \deg(v))$  //建立逆邻接表  
为此，空间需增加多少？
- ❖ 计算顶点v的出度/入度  
增加度数记录域： $\mathcal{O}(n)$ 附加空间  
增加/删除弧时更新度数： $\mathcal{O}(1)$ 时间 //总体 $\mathcal{O}(e)$ 时间  
每次查询  $\mathcal{O}(1)$ 时间！

## 时间复杂度

❖ 给定顶点 $u$ 和 $v$ ，判断是否 $\langle u, v \rangle \in E$

有向图：搜索 $u$ 的邻接表， $O(\deg(u)) = O(e)$

无向图：搜索 $u$ 或 $v$ 的邻接表， $O(\max(\deg(u), \deg(v))) = O(e)$

“并行” 搜索： $O(2 \times \min(\deg(u), \deg(v))) = O(e)$

能够达到邻接矩阵的 $O(1)$ 吗？

❖ 散列！如果装填因子选取得当

//保持兴趣

弧的判定：expected- $O(1)$ ，与邻接矩阵“相同”

空间： $O(n + e)$ ，与邻接表相同

❖ 为何有时仍使用邻接矩阵？仅仅因为实现简单？不，有更多用处！

如：可处理Euclidean graph和intersection graph之类的

隐式图（implicitly-represented graphs）

## 取舍原则

❖ 空间/速度

❖ 顶点类型

- bit
- int
- float
- struct
- class
- ...

❖ 弧类型（方向 / 权值）

❖ 图类型（稀疏 / 稠密）

适用场合

邻接矩阵

邻接表

经常检测边的存在

经常计算顶点的度数

经常做边的插入/删除

顶点数目不确定

图的规模固定

经常做遍历

稠密图

稀疏图

## 6. 图

### 广度优先搜索 算法

邓俊辉

deng@tsinghua.edu.cn

## Breadth-First Search

❖ 始自顶点s的广度优先搜索

访问顶点s

依次访问s所有尚未访问的邻接顶点

依次访问它们尚未访问的邻接顶点

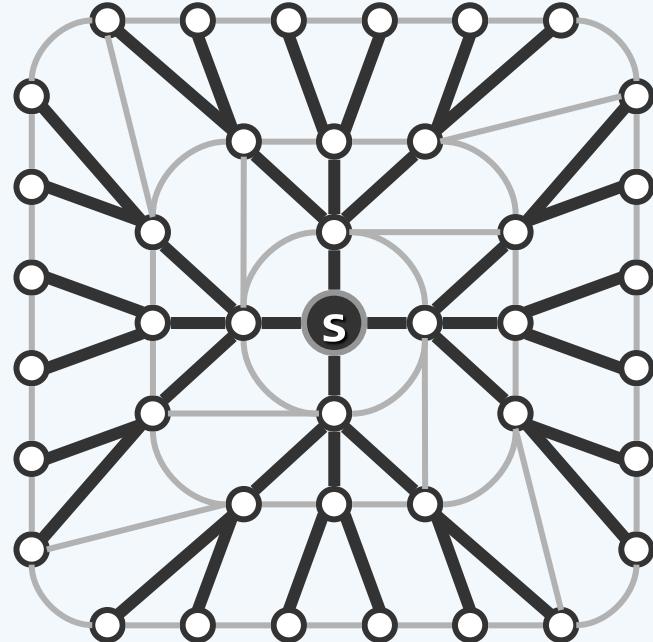
...

如此反复

直至没有尚未访问的邻接顶点

❖ 以上策略及过程完全等同于树的广度优先遍历

❖ 事实上，BFS也的确会构造出原图的一棵支撑树（BFS tree）



## Graph::BFS()

❖ template <typename Tv, typename Te>

```
void Graph<Tv, Te>::BFS( int v, int & clock ) {
```

```
    Queue<int> Q; status(v) = DISCOVERED; Q.enqueue(v); //初始化
```

```
    while ( ! Q.empty() ) { //反复地
```

v      int v = Q.dequeue(); dTime(v) = ++clock; //取出队首顶点v，并

```
for ( int u = firstNbr(v); -1 < u; u = nextNbr(v, u) ) //考察v的每一邻居u
```

```
/* ... 视u的状态，分别处理 ... */
```

v      status(v) = VISITED; //至此，当前顶点访问完毕

}

## Graph::BFS()

```

❖ while ( ! Q.empty() ) { //反复地

    v int v = Q.dequeue(); dTime(v) = ++clock; //取出队首顶点v，并

    for ( int u = firstNbr(v); -1 < u; u = nextNbr(v, u) ) //考察v的每一邻居u

        u if ( UNDISCOVERED == status(u) ) { //若u尚未被发现，则

            u status(u) = DISCOVERED; Q.enqueue(u); //发现该顶点

            type(v, u) = TREE; parent(u) = v; //引入树边

        u } else //若u已被发现(正在队列中)，或者甚至已访问完毕(已出队列)，则

            type(v, u) = CROSS; //将(v, u)归类于跨边

    v status(v) = VISITED; //至此，当前顶点访问完毕

}

```

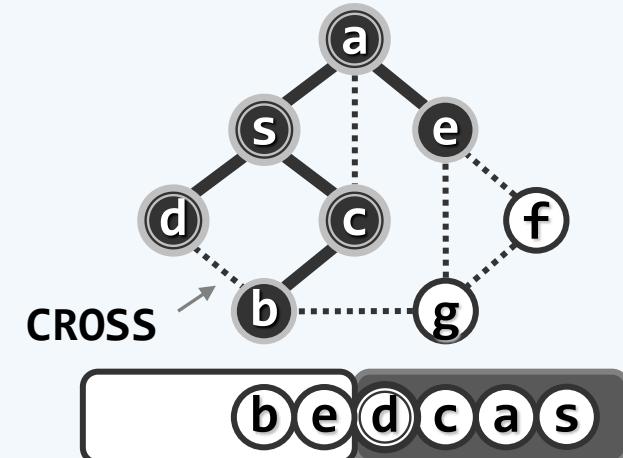
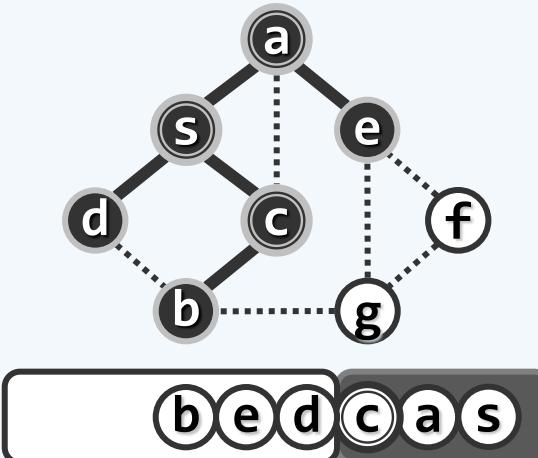
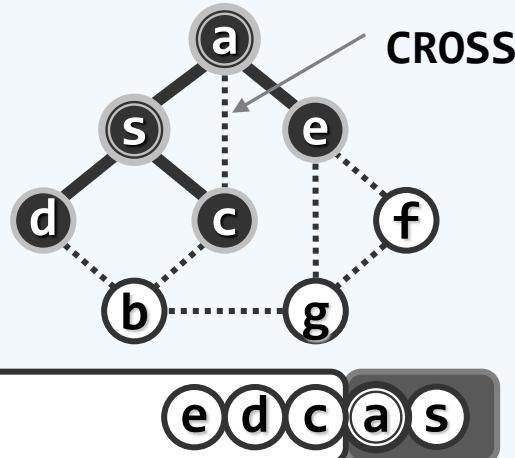
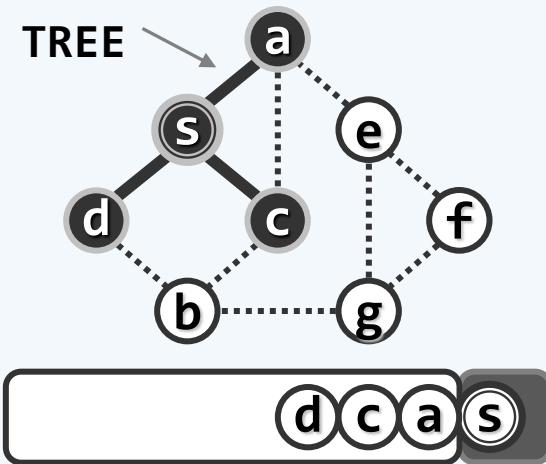
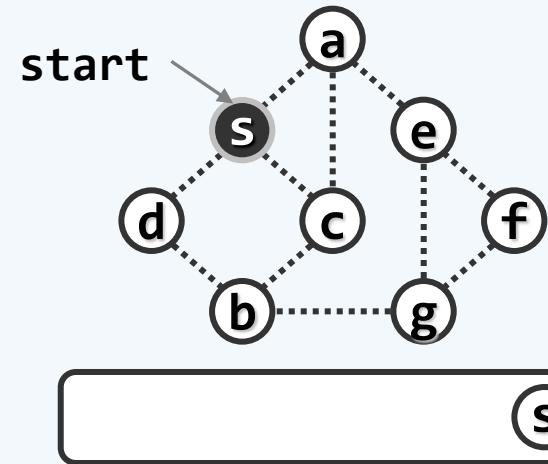
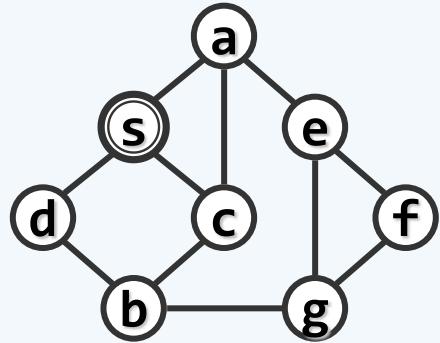
## 6. 图

### 广度优先搜索 实例

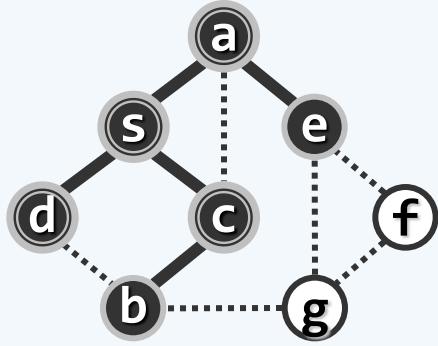
邓俊辉

deng@tsinghua.edu.cn

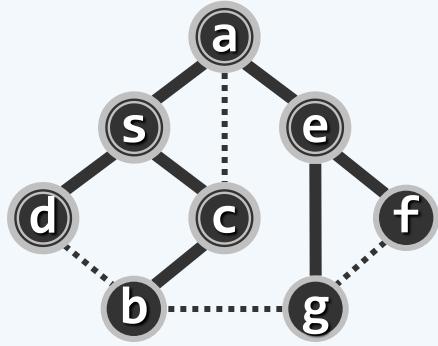
## 无向图



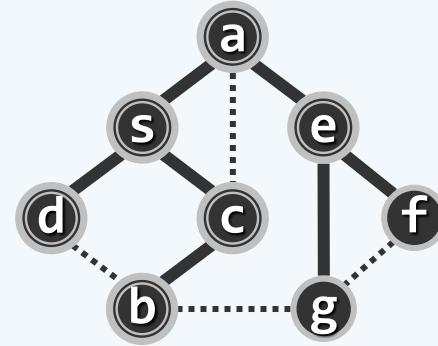
## 无向图



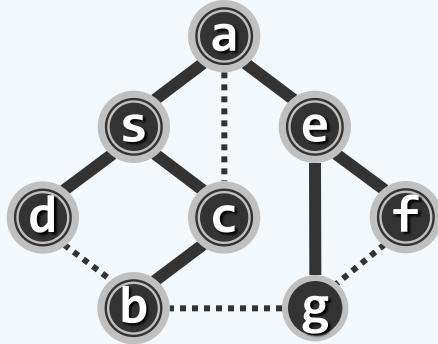
b e d c a s



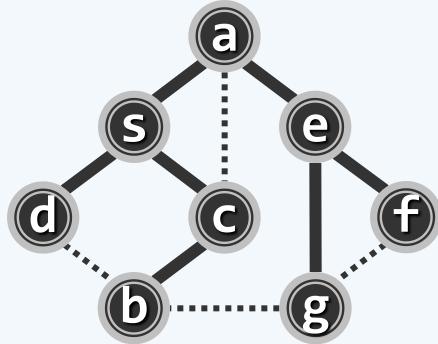
g f b e d c a s



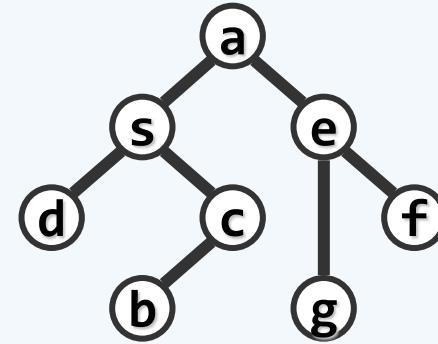
g f b e d c a s



g f b e d c a s



g f b e d c a s



## 6. 图

广度优先搜索  
推广

邓俊辉

deng@tsinghua.edu.cn

## 连通分量 + 可达分量

### ❖ 问题

给定**无向图**，找出其中任一顶点 $s$ 所在的**连通分量**

给定**有向图**，找出源自其中任一顶点 $s$ 的**可达分量**

### ❖ 算法

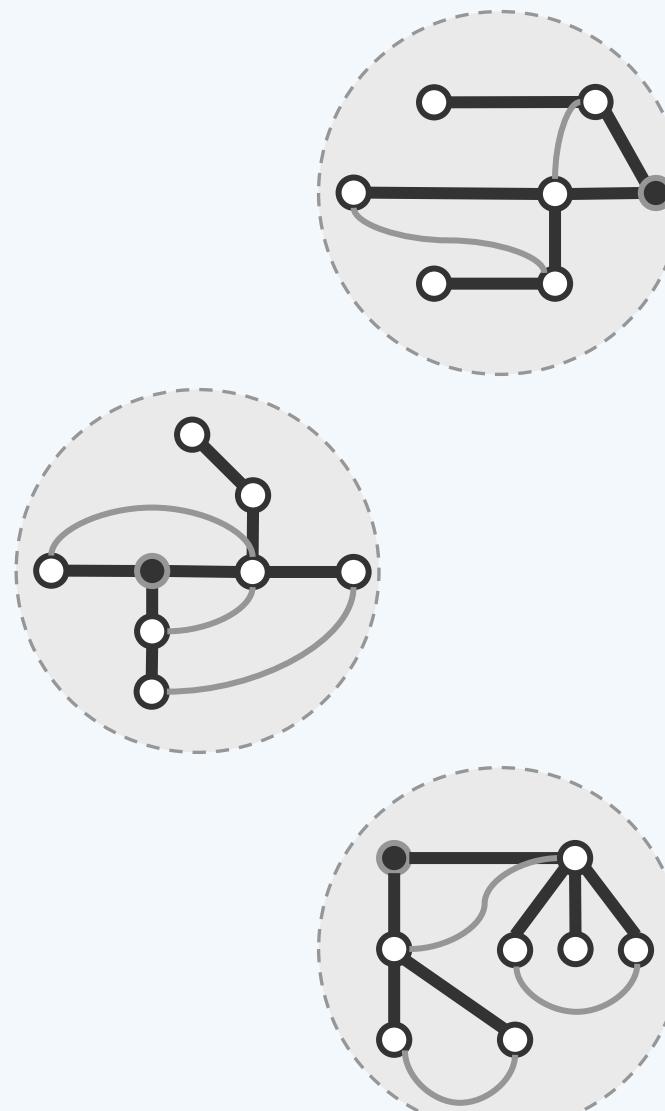
从 $s$ 出发做BFS

输出所有**被发现**的顶点

队列为空后立即终止，无需考虑其它顶点

### ❖ 若图中包含**多个**连通/可达分量

又该如何保证对**全图**的遍历呢？



## Graph::bfs()

❖ template <typename Tv, typename Te> //顶点类型、边类型

```
void Graph<Tv, Te>::bfs( int s ) { //s为起始顶点
```

```
    reset(); int clock = 0; int v = s; //初始化 $\Theta(n + e)$ 
```

do //逐一检查所有顶点，一旦遇到尚未发现的顶点

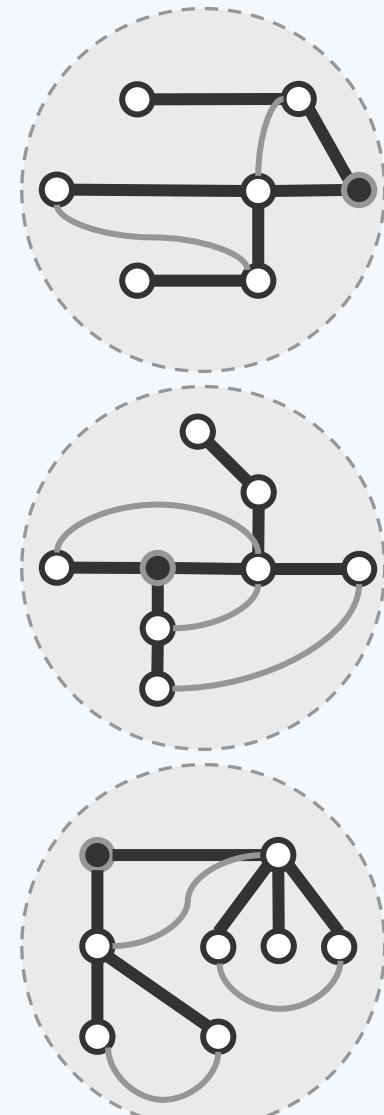
```
    if ( UNDISCOVERED == status(v) ) //累计 $\Theta(n)$ 
```

```
        BFS( v, clock ); //即从该顶点出发启动一次BFS
```

```
    while ( s != ( v = ( ++v % n ) ) ); //按序号访问，不漏不重
```

} //无论共有多少连通/可达分量...

❖ bfs()均可遍历它们，而且自身累计仅需线性时间...



## 复杂度

❖ 考查无向图...

❖ `bfs()`的初始化 (`reset()`)  $O(n + e)$

❖ `BFS()`的迭代  $O(n + 2e)$

    外循环 (`while (!Q.empty())`)，每个顶点只进入1次，累计 $n$ 次  $O(n)$

    内循环（枚举 $v$ 的每一邻居），每个邻居至多进入1次，累计 $\deg(v)$ 次

    采用邻接矩阵  $O(n)$

    采用邻接表  $O(1 + \deg(v))$

    总共 =  $O(\sum v (1 + \deg(v))) = O(n + 2e)$

❖ 整个算法： $O(n + e) + O(n + 2e) = \boxed{O(n + e)}$

❖ 有向图呢？亦是如此！

## 6. 图

广度优先搜索

性质

啊 五环 你比四环多一环

啊 五环 你比六环少一环

终于有一天 你会修到七环

修到七环怎么办

你比五环多两环

邓俊辉

deng@tsinghua.edu.cn

## 边分类

❖ 经BFS后，所有边将确定方向，且被分为两类

//有向图有何不同？

tree edges + cross edges

❖ 树边(v, u)



联边之前：v DISCOVERED & u UNDISCOVERED

联边之后：v == parent(u)

❖ 跨边(v, u)



无向图：只可能v和u均为DISCOVERED

有向图：还可能v是DISCOVERED、u是VISITED

## BFS树/森林

❖ 对于每一连通/可达分量，`bfs()`进入BFS( $v$ )恰好1次 ( $v$ 为该分量的起始顶点)

❖ 进入BFS( $v$ )时，队列为空

$v$ 所属分量内的每个顶点

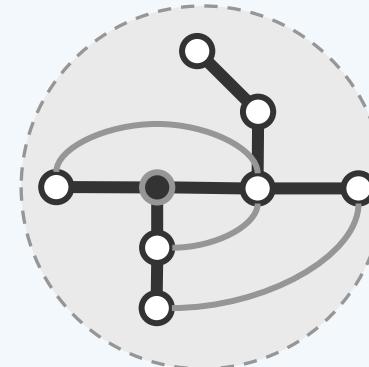
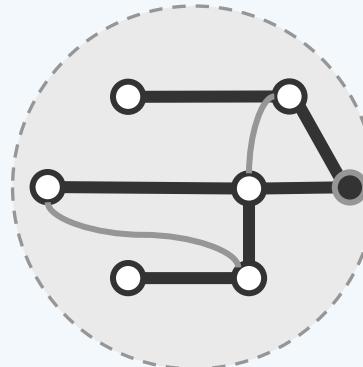
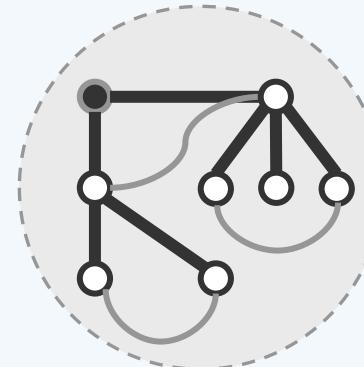
- 迟早会以UNDISCOVERED状态进队1次
- 进队后随即转为DISCOVERED状态，并生成一条树边
- 迟早会出队并转为VISITED状态

退出BFS( $v$ )时，队列为空

❖ BFS( $v$ )以 $v$ 为根，生成一棵BFS树

❖ `bfs()`生成一个BFS森林包含

$c$ 棵树、 $n - c$ 条树边和 $e - n + c$ 条跨边



## 最短路径

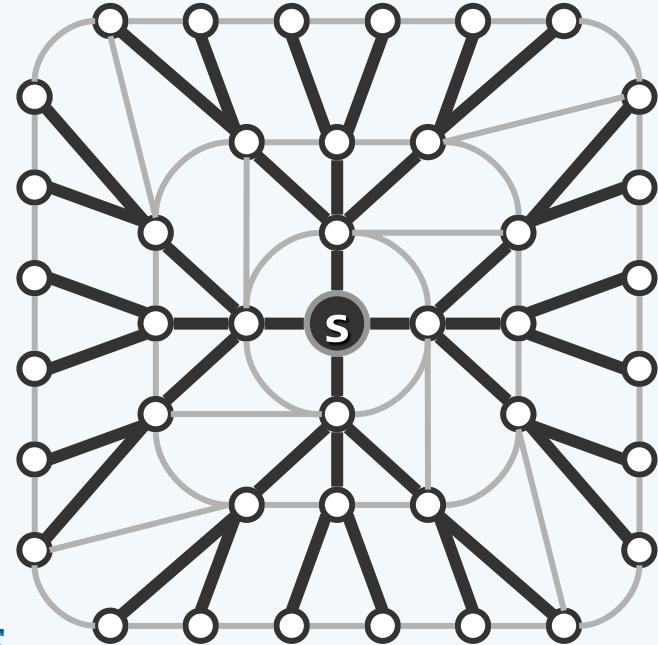
❖ 从无向图中顶点s出发，到任一顶点v的所有路径中

- 长度最短者 $\pi(s, v) = ?$
- (最短)距离 =  $dist(v) = |\pi(s, v)| = ?$
- 退化情况：可能有多条 //任取其一

❖ 在BFS过程中的任一时刻 //贪吃蛇

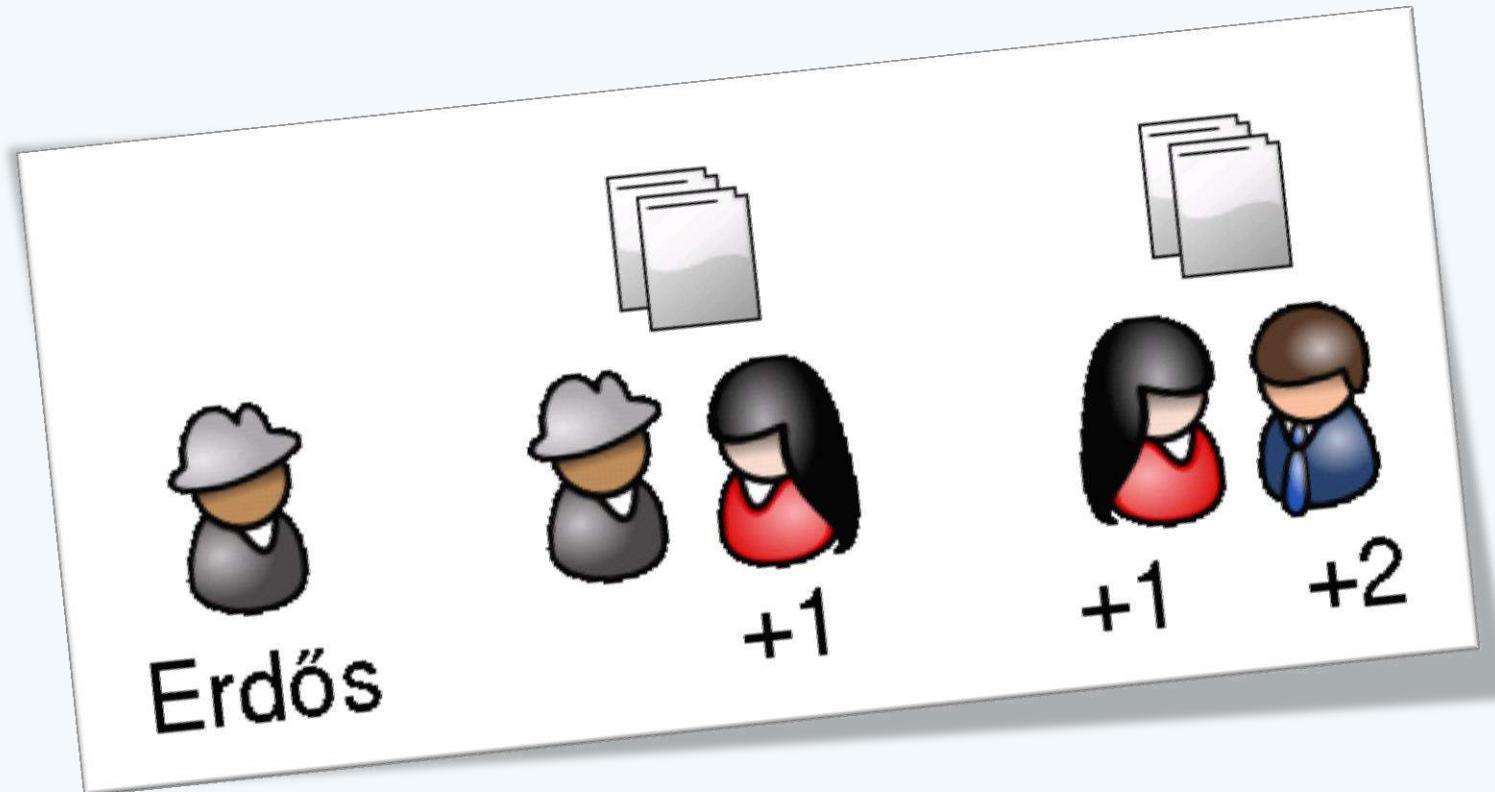
- 队列中顶点按 $dist()$ 单调排列 //自内而外，由近及远
- 队列中相邻顶点， $dist()$ 相差不超过1
- 队首、队末顶点， $dist()$ 相差不超过1 // (模拟) 环路遍历
- 由树边联接的顶点， $dist()$ 恰好相差1 //头紧追尾
- 由跨边联接的顶点， $dist()$ 至多相差1 //五至六环，通路可达；从五至七，无环际路

❖ BFS树中从s到v的路径，即是 $\pi(s, v)$



## Erdős Number

❖ Describes the "collaborative distance" between mathematician Paul Erdős and another person, as measured by authorship of mathematical papers



## Chow Number

❖ chow (吴孟达) = 1

[整蛊专家] (1) \*周星驰 +吴孟达 成奎安 刘德华 关之琳 邱淑贞

❖ chow (葛优) = 2

[没完没了] (2) +葛优 \*吴倩莲 傅彪

[97家有喜事] (1) \*周星驰 +吴倩莲 吴镇宇 钟丽缇 伍咏薇 黄百鸣

❖ chow (姜昆) = 3

[京都球侠] (3) \*张丰毅 孙敏 +姜昆 陈佩斯 于绍康 唐杰忠

[热线追击] (2) +张丰毅 任达华 王馨平 \*吴家丽

[审死官] (1) \*周星驰 吴孟达 +吴家丽 秦沛 朱咪咪 梅艳芳

❖ chow ("Julia Roberts") = Infinity

## 6. 图

### 深度优先搜索 算法

悔相道之不察兮，延伫乎吾将反  
回朕车以复路兮，及行迷之未远

邓俊辉

deng@tsinghua.edu.cn

## Depth-First Search

❖ DFS( $s$ ) //始自顶点 $s$ 的深度优先搜索

访问顶点 $s$

若 $s$ 尚有未被访问的邻居，则任取其一 $u$ ，递归执行DFS( $u$ )

否则，返回

❖ 若此时图中尚有顶点未被访问 //何时出现这一情况？

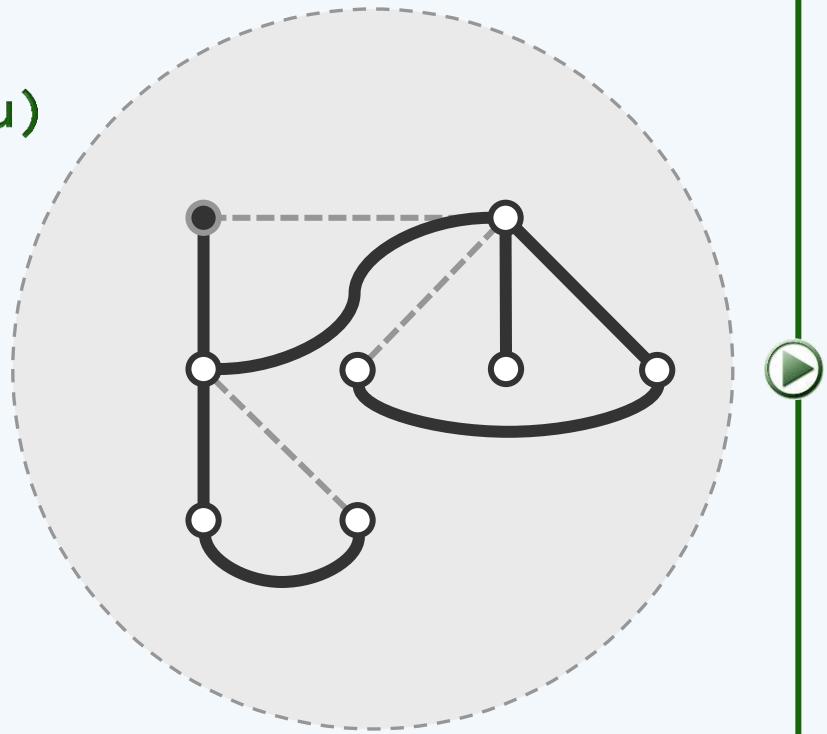
任取这样的一个顶点作起始点

重复上述过程

直至所有顶点都被访问到

❖ 等效于树的先序遍历

事实上，DFS也的确会构造出原图的一棵支撑树 (DFS tree)



## Graph::DFS()

❖ template <typename Tv, typename Te> //顶点类型、边类型

```
void Graph<Tv, Te>::DFS( int v, int & clock ) {
```

```
dTime(v) = ++clock; status(v) = DISCOVERED; //发现当前顶点v
```

```
for ( int u = firstNbr(v); -1 < u; u = nextNbr(v, u) ) //枚举v的每一邻居u
```

```
/* ... 视u的状态，分别处理 ... */
```

```
/* ... 与BFS不同，含有递归 ... */
```

```
status(v) = VISITED; fTime(v) = ++clock; //至此，当前顶点v方告访问完毕
```

```
}
```

## Graph::DFS()

```

❖ for ( int u = firstNbr(v); -1 < u; u = nextNbr(v, u) ) //枚举v所有邻居u

switch ( status(u) ) { //并视其状态分别处理

    case UNDISCOVERED: //u尚未发现，意味着支撑树可在此拓展

        type(v, u) = TREE; parent(u) = v; DFS( u, clock ); break; //递归

    case DISCOVERED: //u已被发现但尚未访问完毕，应属被后代指向的祖先

        type(v, u) = BACKWARD; break;

    default: //u已访问完毕 ( VISITED , 有向图 )，则视承袭关系分为前向边或跨边

        type(v, u) = dTime(v) < dTime(u) ? FORWARD : CROSS; break;

} //switch

```

## 6. 图

深度优先搜索

实例（无向图）

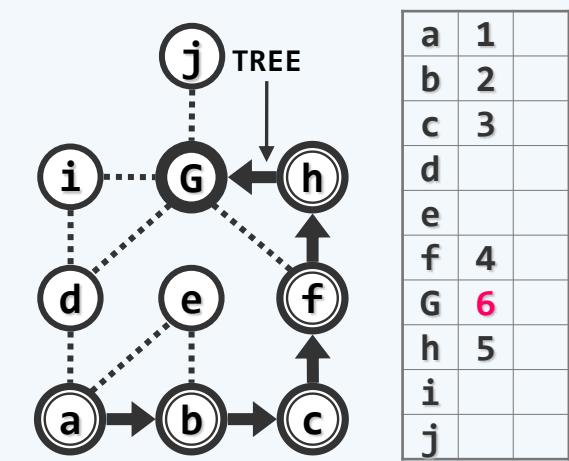
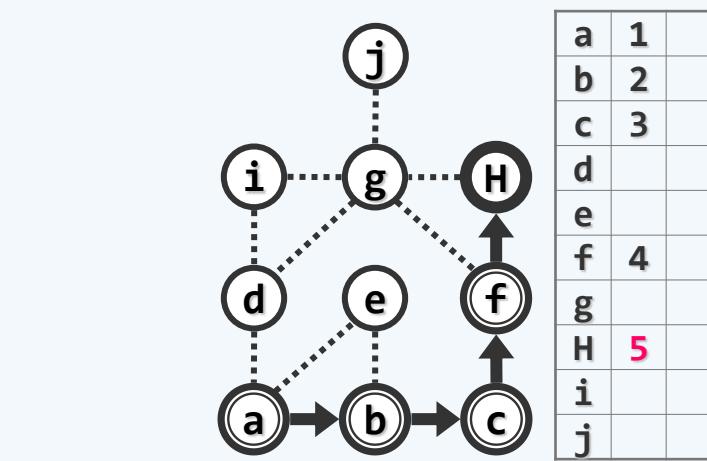
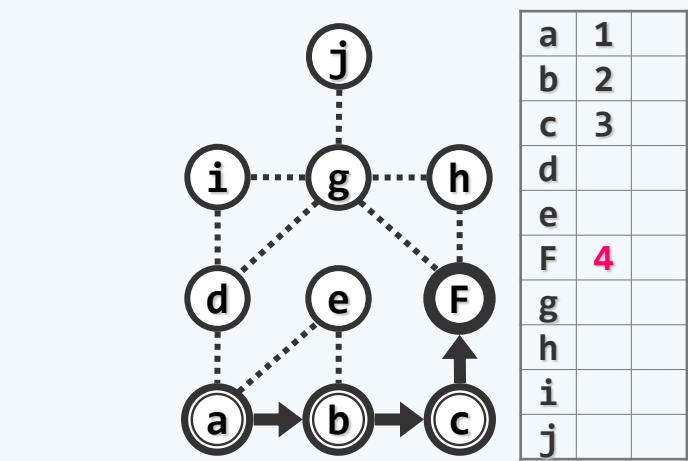
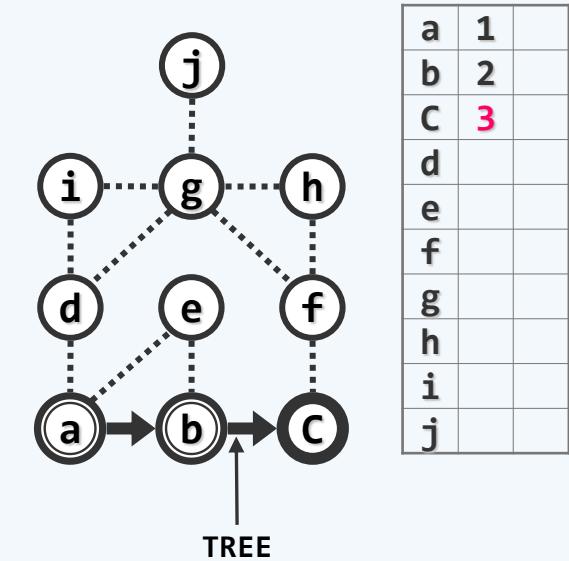
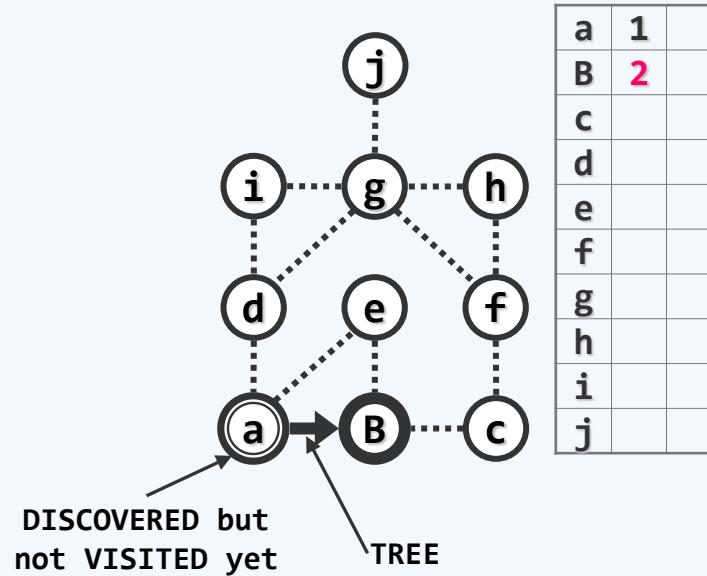
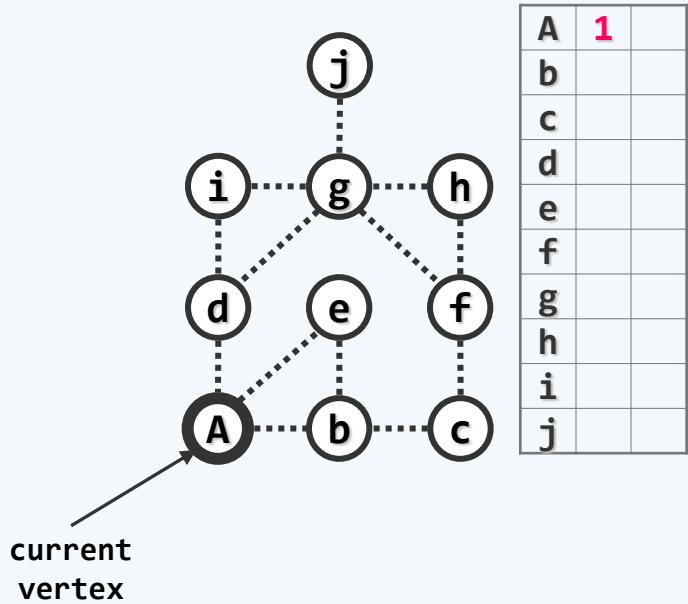
Keep it simple, stupid.

- K. Johnson

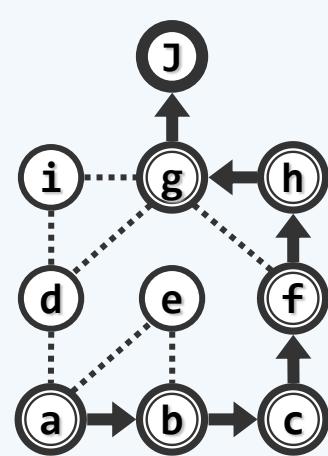
邓俊辉

deng@tsinghua.edu.cn

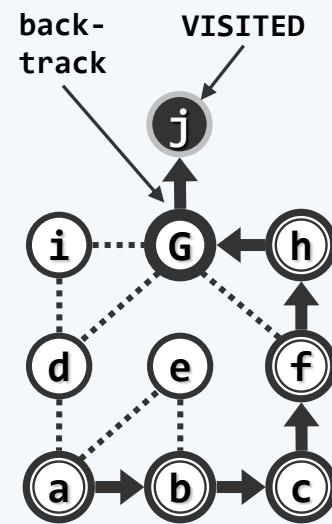
1/4



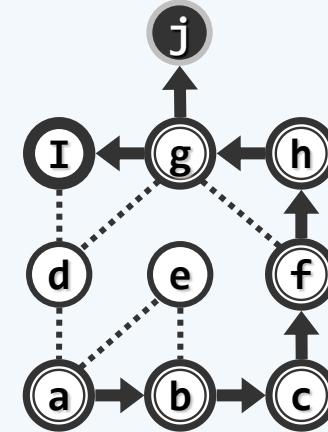
2/4



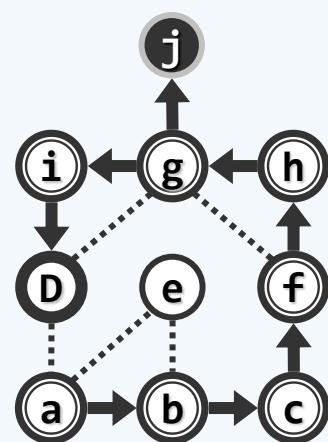
a	1	
b	2	
c	3	
d		
e		
f	4	
g	6	
h	5	
i		
j	7	



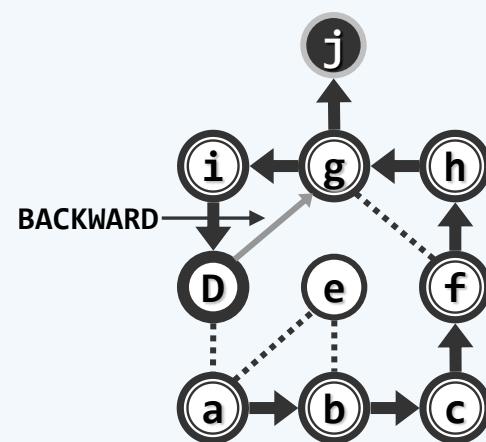
a	1	
b	2	
c	3	
d		
e		
f	4	
G	6	
h	5	
i		
j	7	8



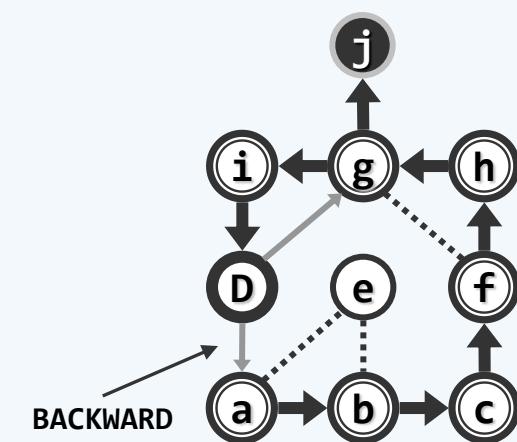
a	1	
b	2	
c	3	
d		
e		
f	4	
g	6	
h	5	
I	9	
j	7	8



a	1	
b	2	
c	3	
D	10	
e		
f	4	
g	6	
h	5	
i	9	
j	7	8

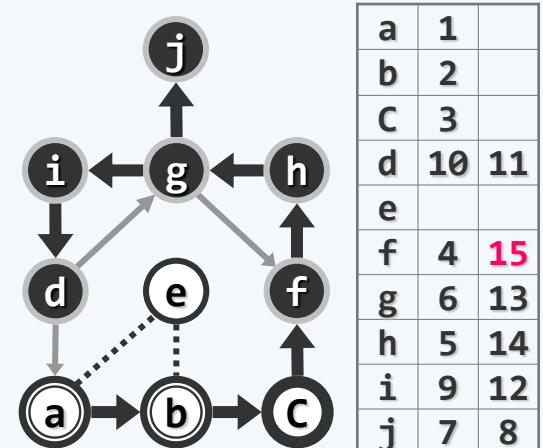
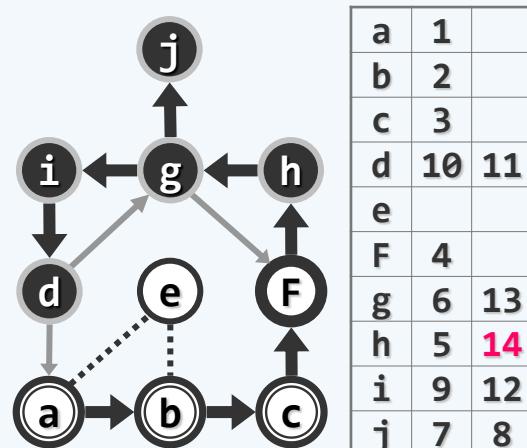
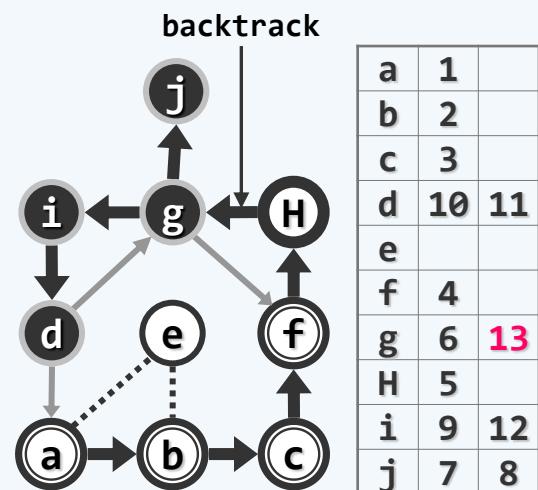
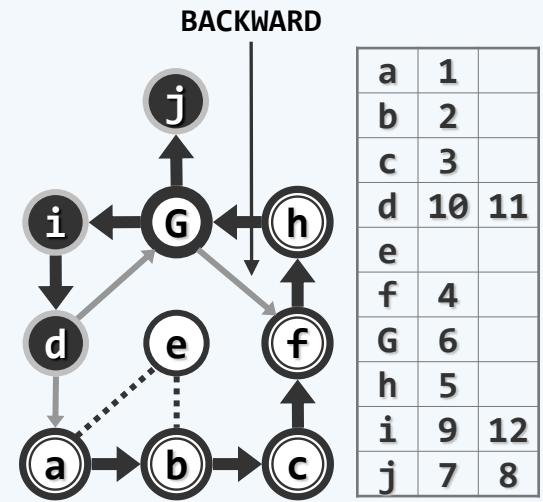
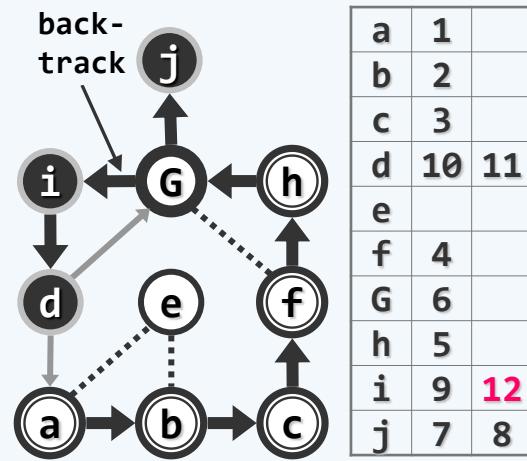
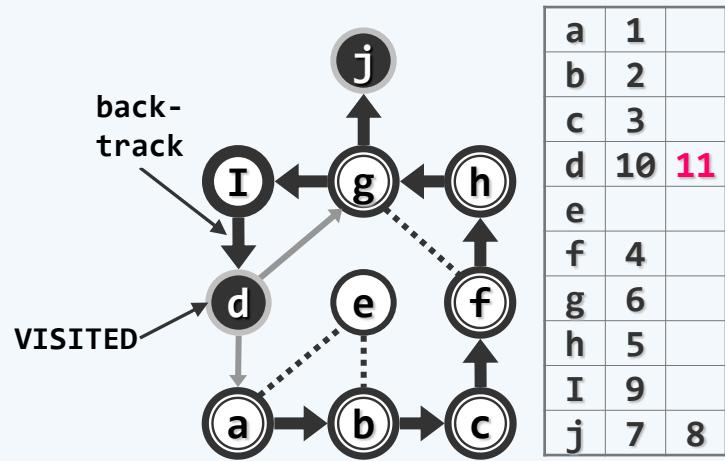


a	1	
b	2	
c	3	
D	10	
e		
f	4	
g	6	
h	5	
i	9	
j	7	8

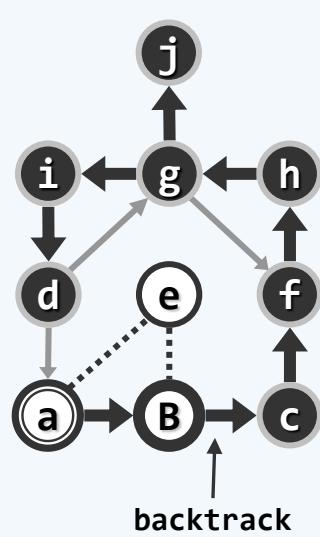


a	1	
b	2	
c	3	
D	10	
e		
f	4	
g	6	
h	5	
i	9	
j	7	8

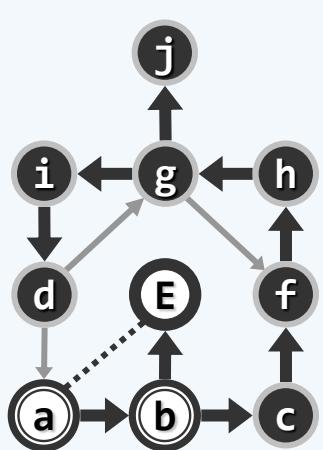
3/4



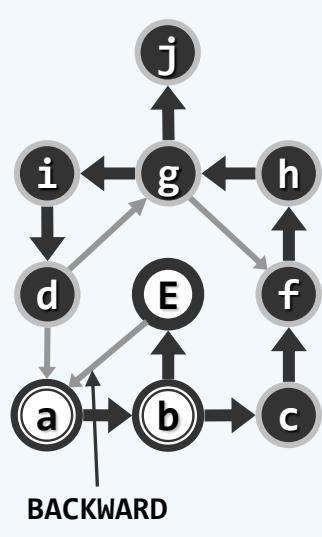
4/4



a	1	
B	2	
c	3	16
d	10	11
e		
f	4	15
g	6	13
h	5	14
i	9	12
j	7	8

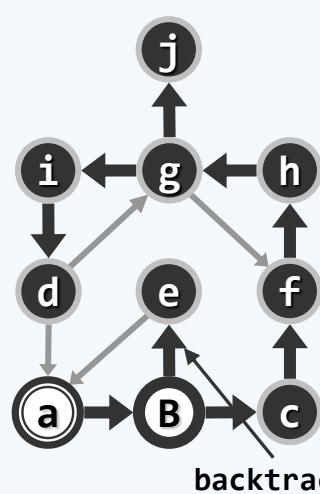


a	1	
b	2	
c	3	16
d	10	11
E	17	
f	4	15
g	6	13
h	5	14
i	9	12
j	7	8

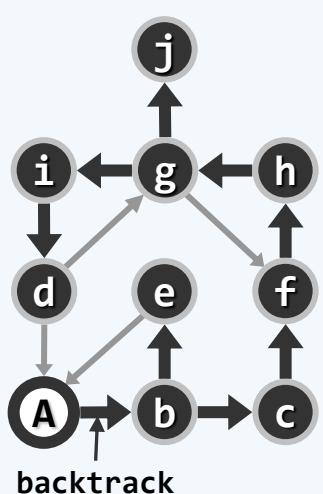


BACKWARD

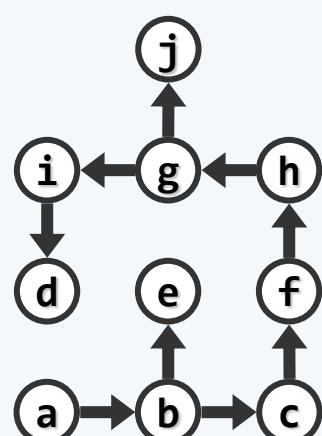
a	1	
b	2	
c	3	16
d	10	11
E	17	
f	4	15
g	6	13
h	5	14
i	9	12
j	7	8



a	1	
B	2	
c	3	16
d	10	11
e	17	18
f	4	15
g	6	13
h	5	14
i	9	12
j	7	8



A	1	
b	2	
c	3	16
d	10	11
e	17	18
f	4	15
g	6	13
h	5	14
i	9	12
j	7	8



a	1	20
b	2	19
c	3	16
d	10	11
e	17	18
f	4	15
g	6	13
h	5	14
i	9	12
j	7	8

## 6. 图

深度优先搜索

推广

邓俊辉

deng@tsinghua.edu.cn

## 非连通

❖ 与BFS( $v$ )类似，DFS( $v$ )也可遍历 $v$ 所属分量

——若含多个分量呢？

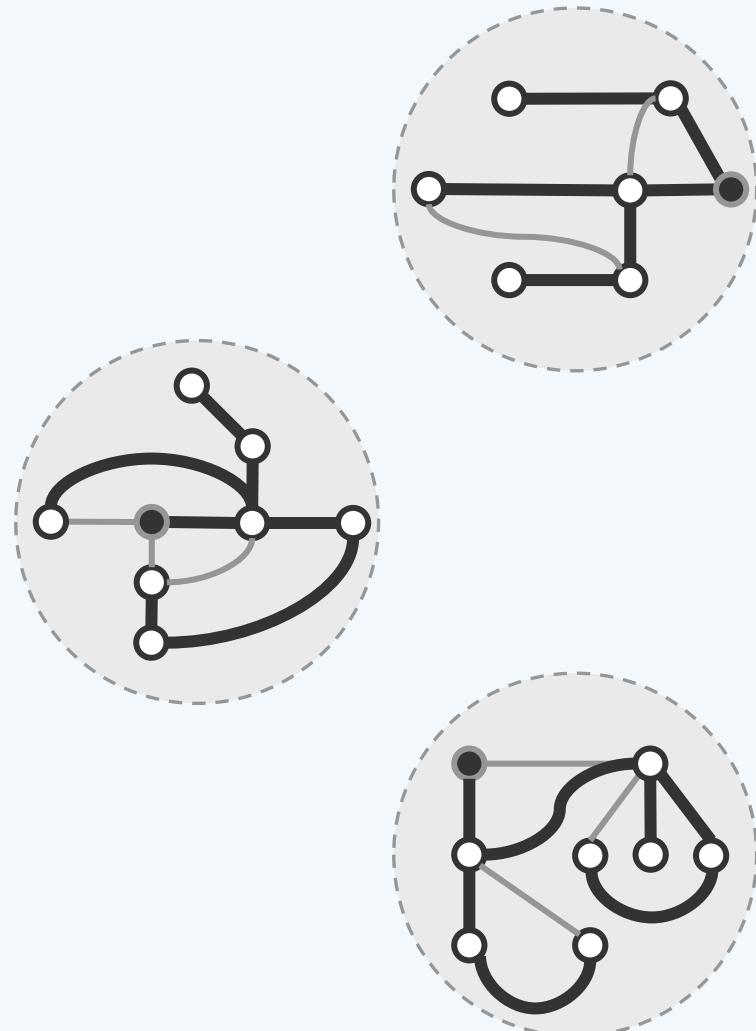
❖ 与bfs( $s$ )类似（采用邻接表）

dfs( $s$ )也可在累计 $\mathcal{O}(n + e)$ 时间内

- 对于每一连通/可达分量

从其起始顶点 $v$ 进入DFS( $v$ )恰好1次，并

- 最终生成一个DFS森林（包含 $c$ 棵树、 $n - c$ 条树边）



## Graph::dfs()

❖ template <typename Tv, typename Te> //顶点类型、边类型

```
void Graph<Tv, Te>::dfs( int s ) { //s为起始顶点
```

```
    reset(); int clock = 0; int v = s; //初始化
```

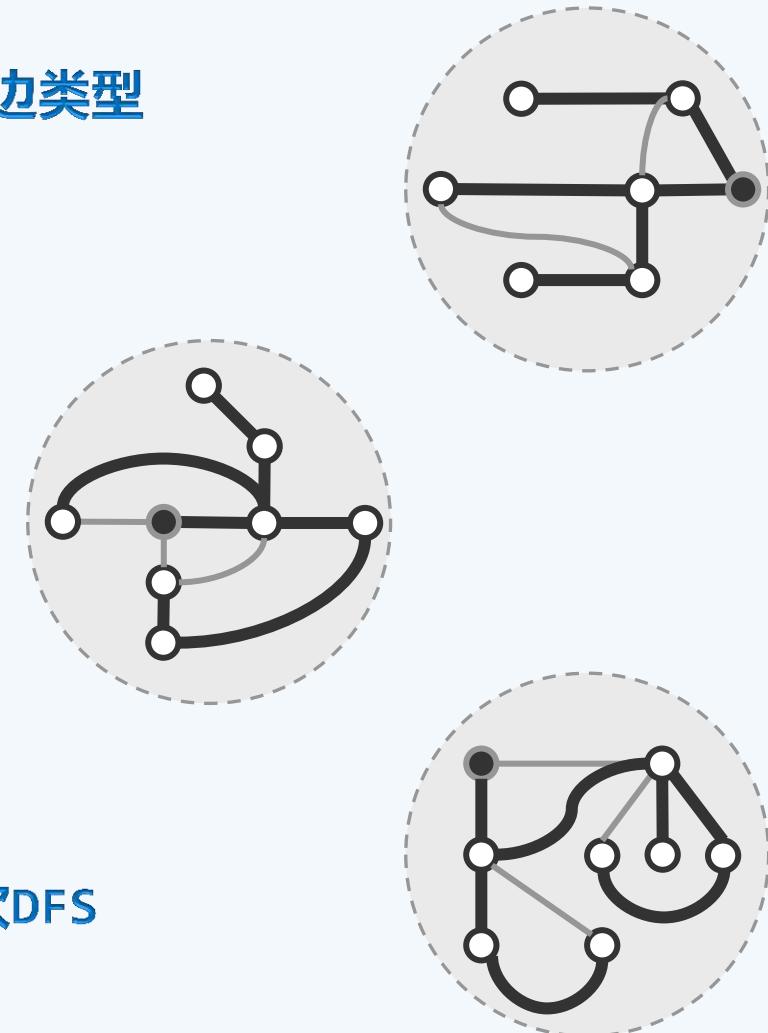
```
    do //逐一检查所有顶点，一旦遇到尚未发现的顶点
```

```
        if ( UNDISCOVERED == status(v) )
```

```
            DFS( v, clock ); //即从该顶点出发启动一次DFS
```

```
    while ( s != ( v = ( ++v % n ) ) ); //按序号访问，故不漏不重
```

```
}
```



## 6. 图

深度优先搜索

实例（有向图）

邓俊辉

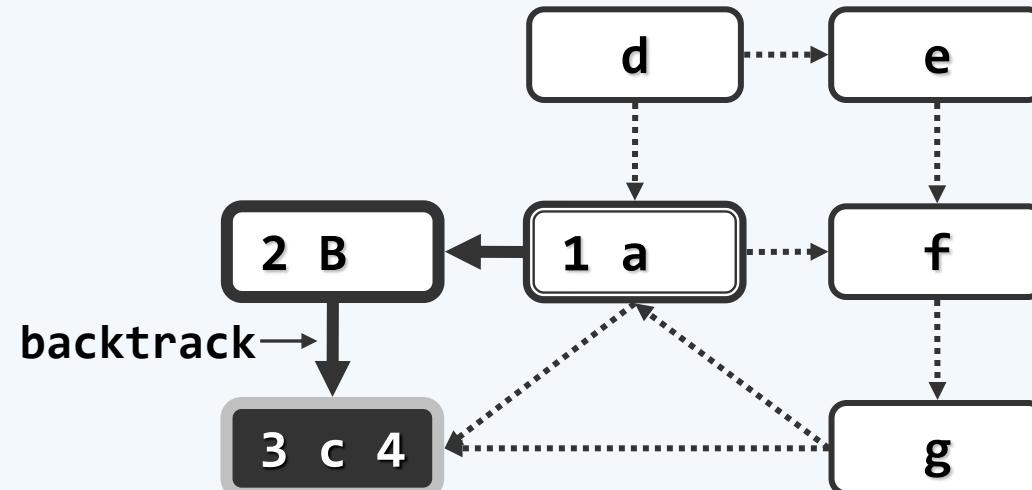
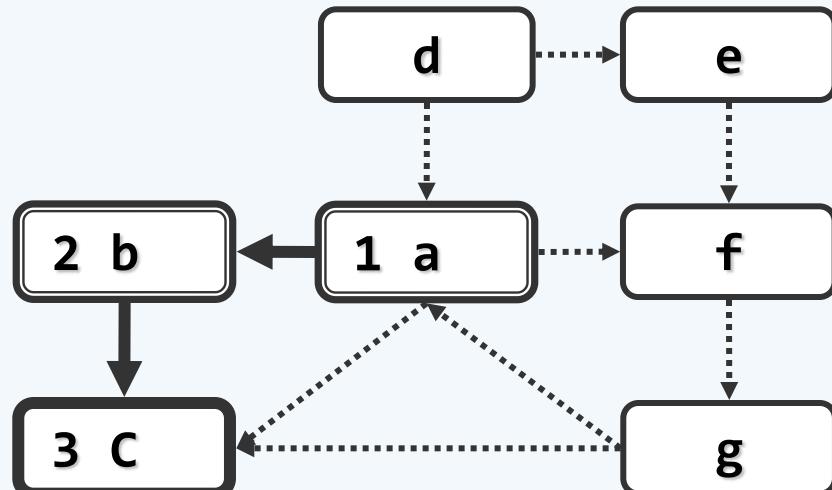
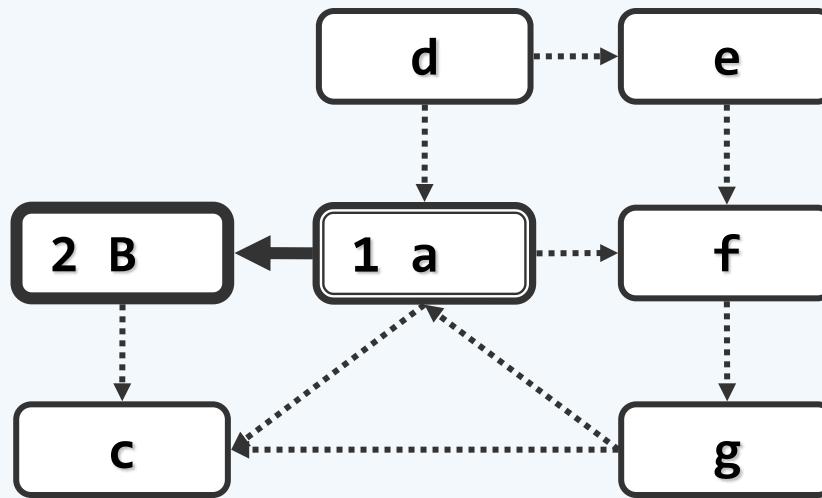
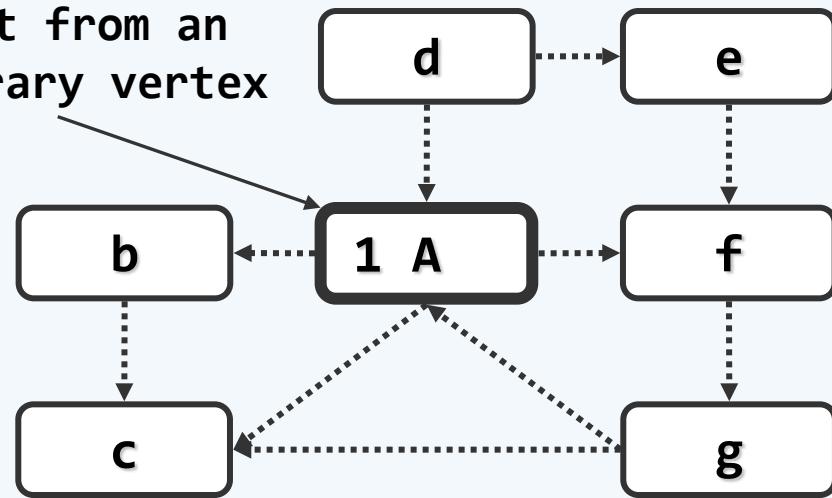
deng@tsinghua.edu.cn

1/5

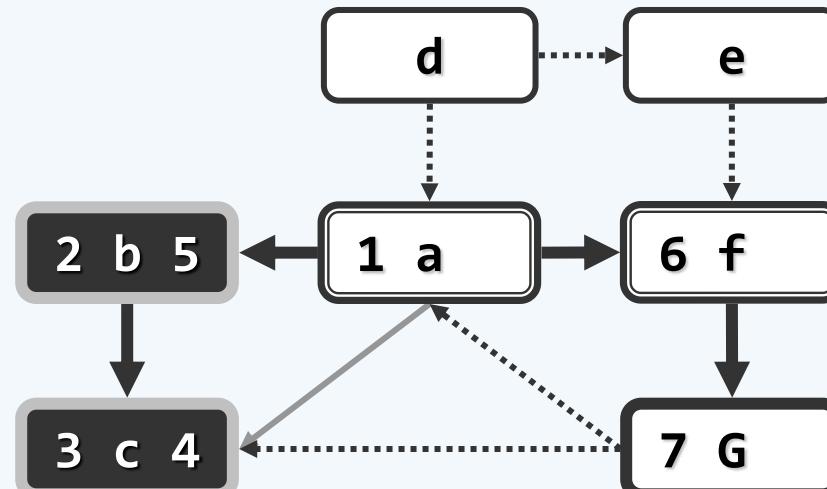
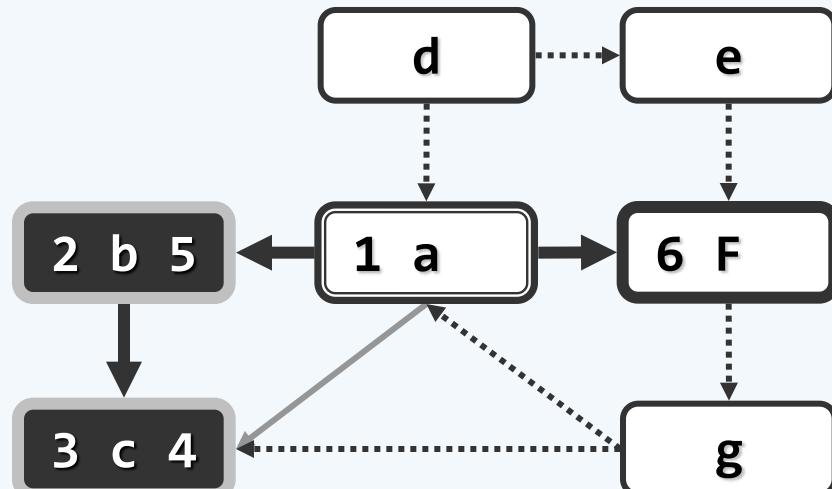
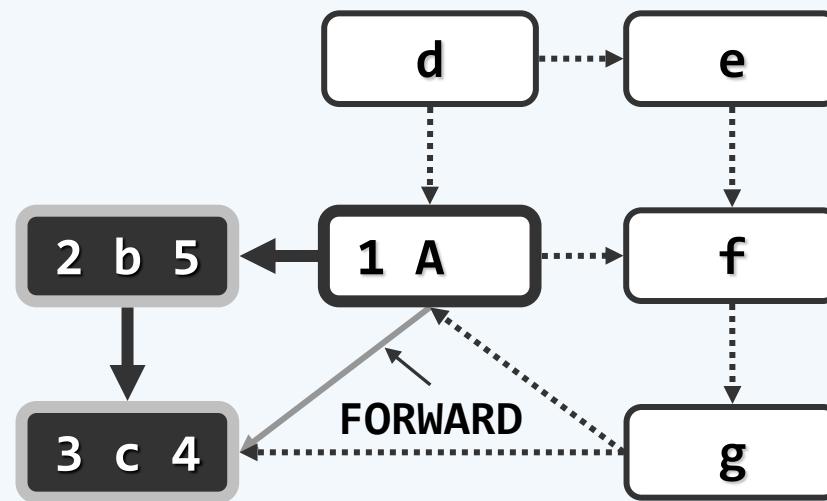
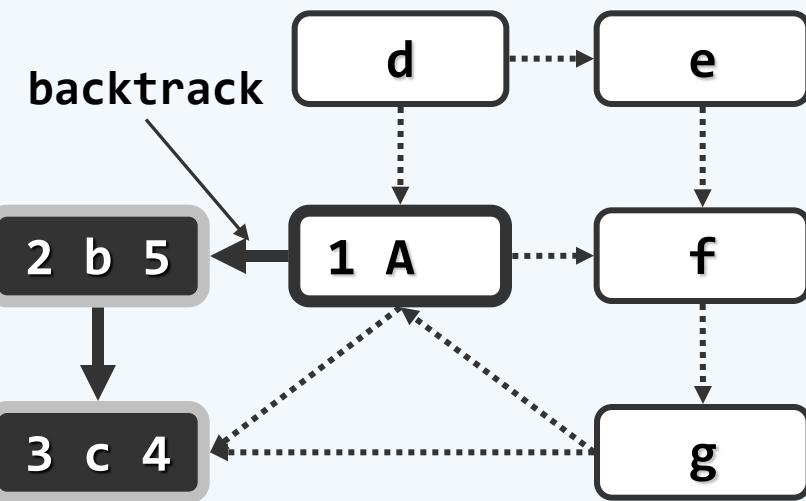
start from an arbitrary vertex

```

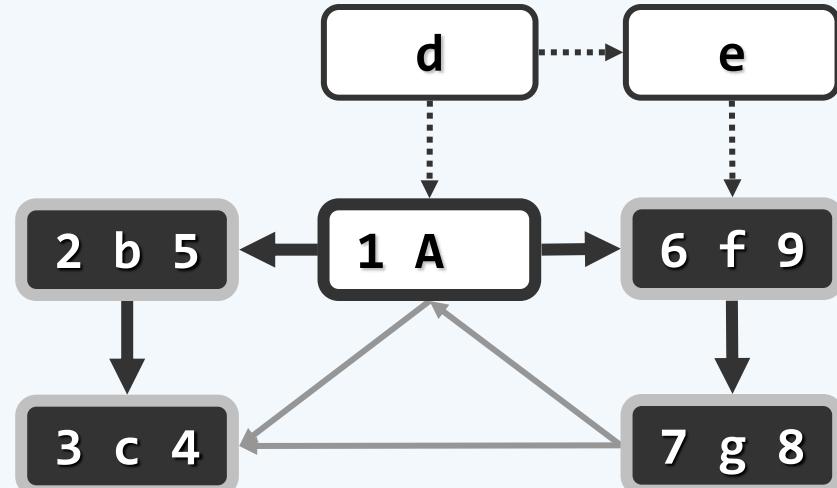
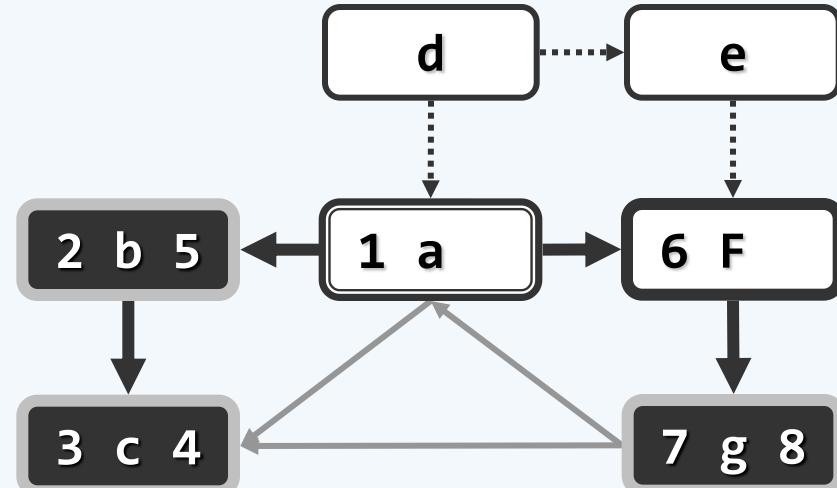
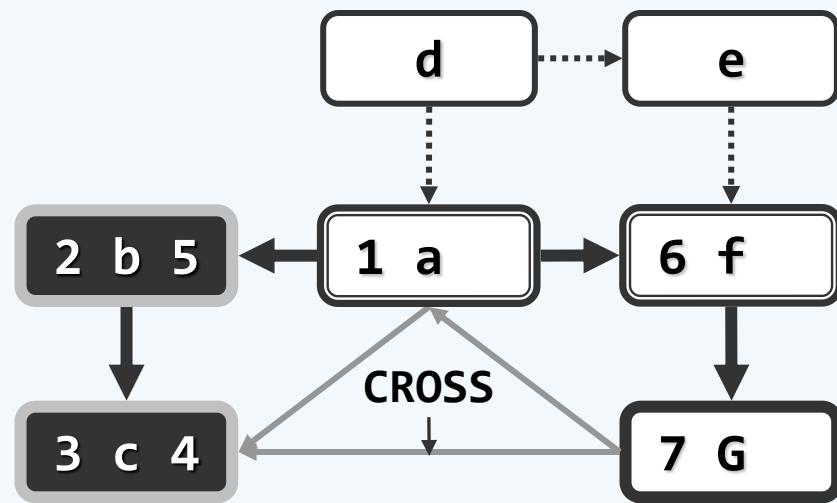
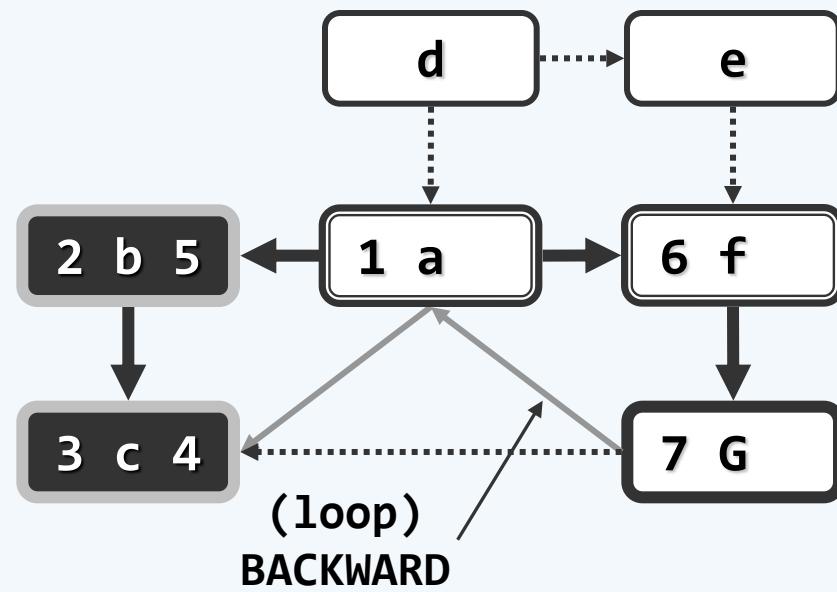
graph TD
    d[d] --> e[e]
    e --> f[f]
    b[b] --> A1[1 A]
    A1 --> f
    f --> g[g]
    g --> c[c]
    d -.-> b
    e -.-> f
    A1 -.-> c
    A1 -.-> g
  
```



2/5

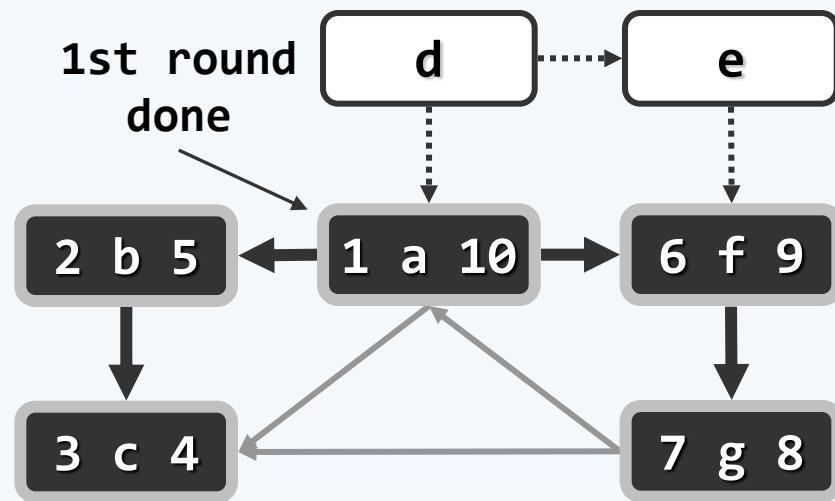


3/5

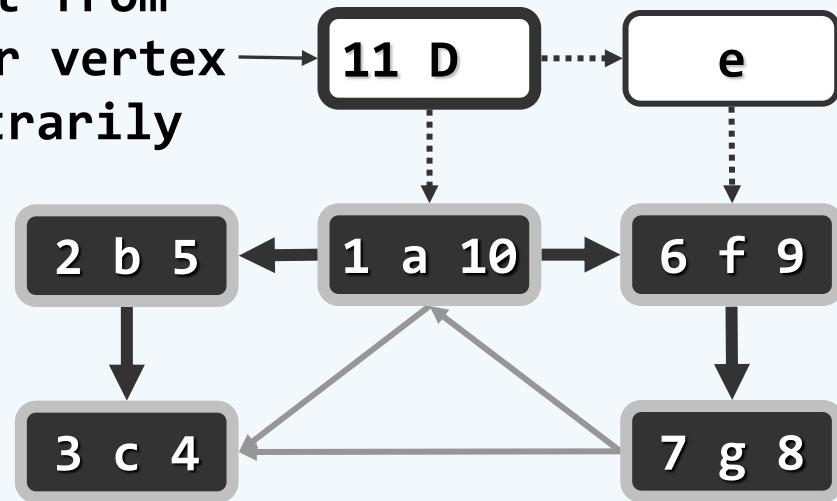


4/5

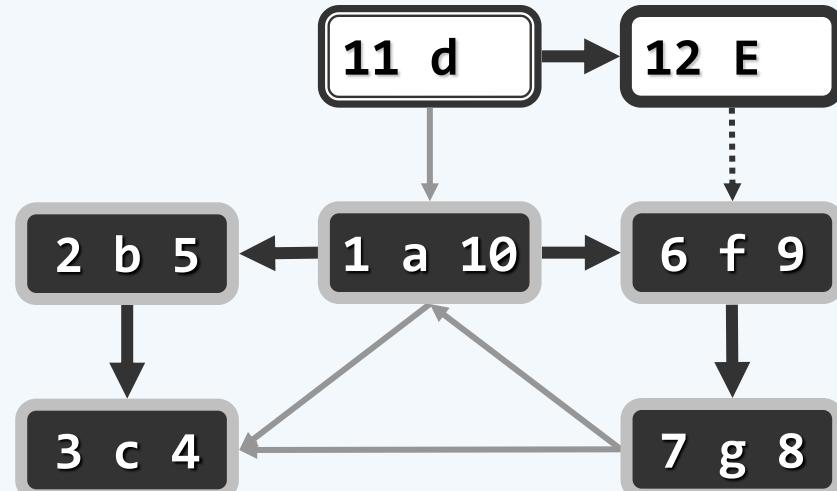
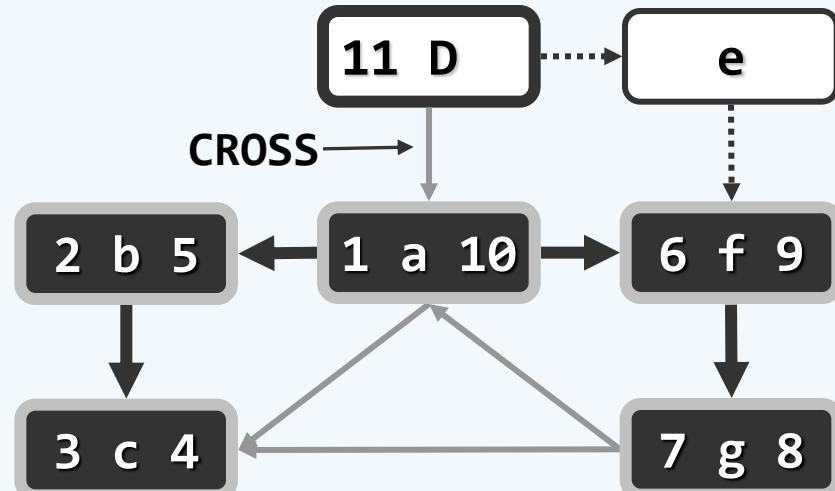
1st round  
done



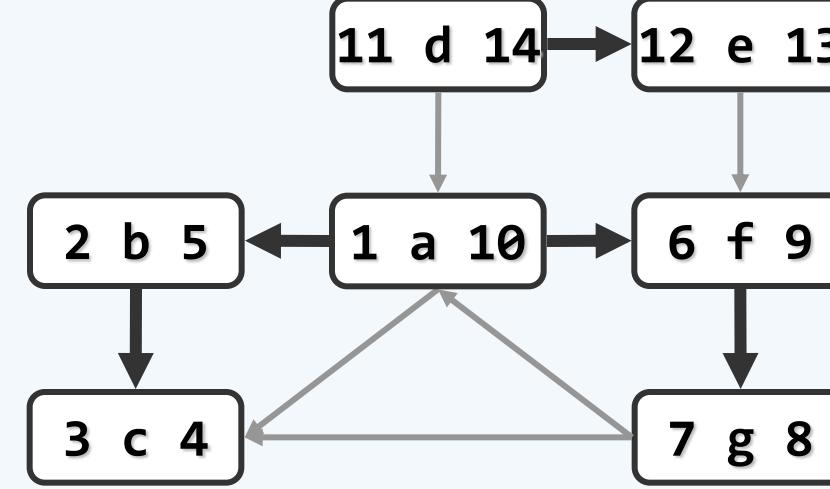
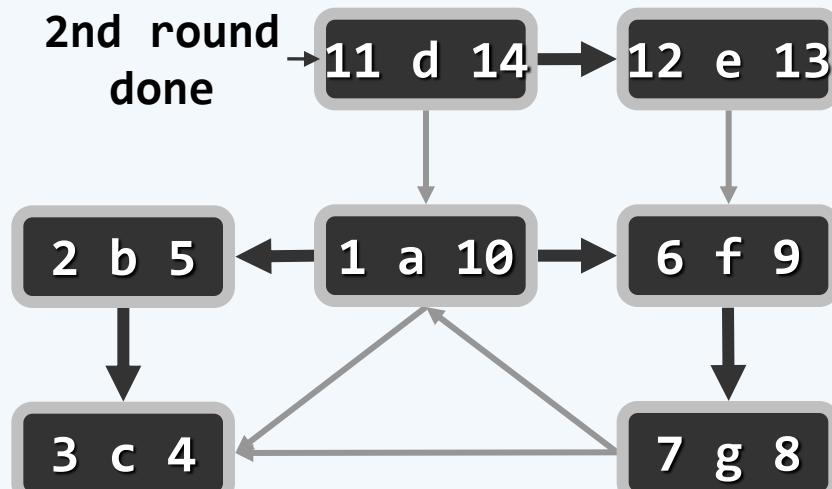
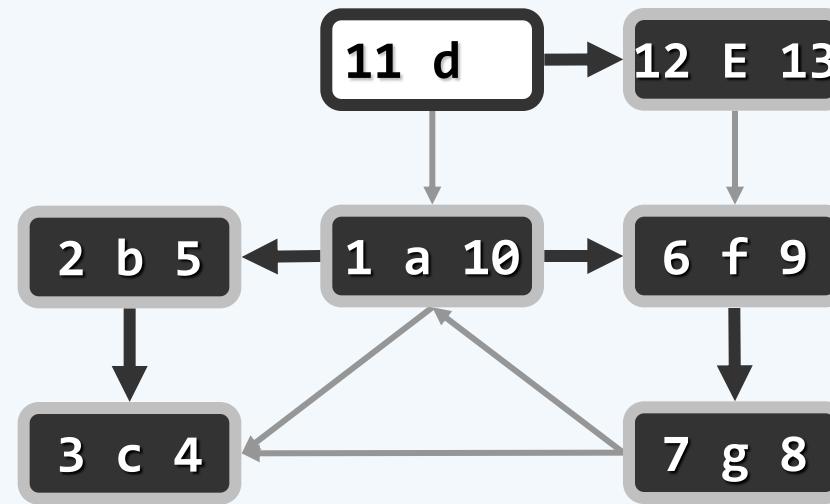
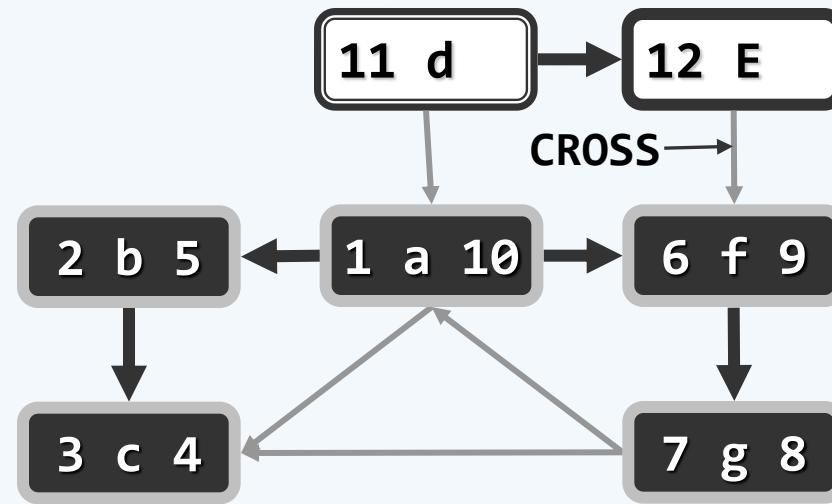
start from  
another vertex  
arbitrarily



CROSS



5/5



## 6. 图

深度优先搜索  
性质

身后有余忘缩手  
眼前无路想回头

邓俊辉

deng@tsinghua.edu.cn

## DFS树/森林

❖ 从顶点s出发的DFS

- 在无向图中将访问与s连通的所有顶点 connected component
- 在有向图中将访问由s可达的所有顶点 reachable component

❖ 经DFS确定的树边，不会构成回路

❖ 从s出发的DFS，将以s为根生成一棵DFS树；所有DFS树，进而构成DFS森林

❖ DFS树及森林由parent指针描述（只不过所有边取反向）

❖ DFS之后，我们已经知道森林乃至原图的全部信息了吗？

就某种意义而言，是的...

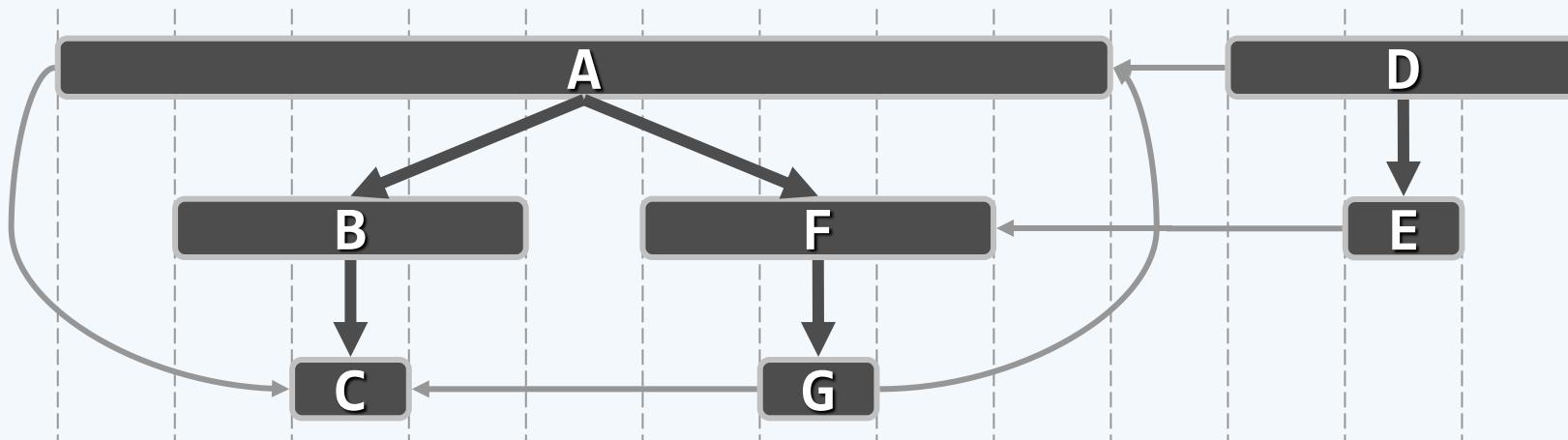
## 括号引理

❖ 活跃期 :  $\text{active}[u] = (\text{dTime}[u], \text{fTime}[u])$

❖ Parenthesis Lemma : 给定有向图  $G = (V, E)$  及其任一DFS森林，则

- $u$  是  $v$  的后代 iff  $\text{active}[u] \subseteq \text{active}[v]$
- $u$  是  $v$  的祖先 iff  $\text{active}[u] \supseteq \text{active}[v]$
- $u$  与  $v$  “无关” iff  $\text{active}[u] \cap \text{active}[v] = \emptyset$

❖ 仅凭  $\text{status}[]$ 、 $\text{dTime}[]$  和  $\text{fTime}[]$ ，即可对各边分类…



## 边分类

❖ TREE(v, u) :   可从当前v进入处于`UNDISCOVERED`状态的u

❖ BACKWARD(v, u) :   试图从当前v进入处于`DISCOVERED`状态的u

DFS发现后向边 iff 存在回路

//后向边数 == 回路数 ?

❖ FORWARD(v, u) :  

试图从当前顶点v进入处于`VISITED`状态的u , 且v更早被发现

❖ CROSS(v, u) :  

试图从当前顶点v进入处于`VISITED`状态的u , 且u更早被发现

❖ 无向图中，后向边与前向边不予区分，跨边没有 //为什么？

## 遍历算法的应用

连通图的支撑树 ( DFS/BFS Tree )	DFS/BFS
非连通图的支撑森林	DFS/BFS
连通性检测	DFS/BFS
无向图环路检测	DFS/BFS
有向图环路检测	DFS
顶点之间可达性检测/路径求解	DFS/BFS
顶点之间的最短距离	BFS
直径	BFS
Eulerian tour	DFS
拓扑排序	DFS
双连通分量、强连通分量分解	DFS
...	...

## 6. 图

拓扑排序

零入度算法

邓俊辉

deng@tsinghua.edu.cn

## 有向无环图

❖ **Directed Acyclic Graph**

❖ 应用

类派生和继承关系图中，是否存在循环定义

操作系统中，相互等待的一组线程可否调度，如何调度

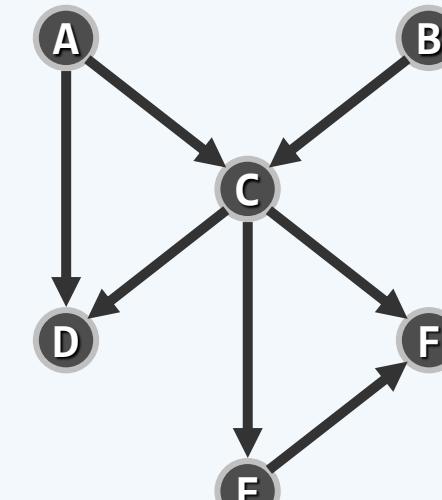
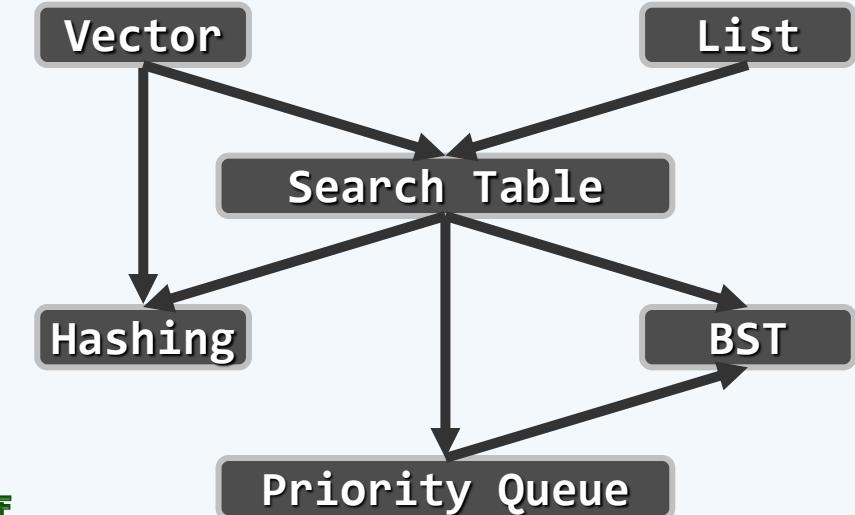
给定一组相互依赖的课程，是否存在可行的培养方案

给定一组相互依赖的知识点，是否存在可行的教学进度方案

项目工程图中，是否存在可串行施工的方案

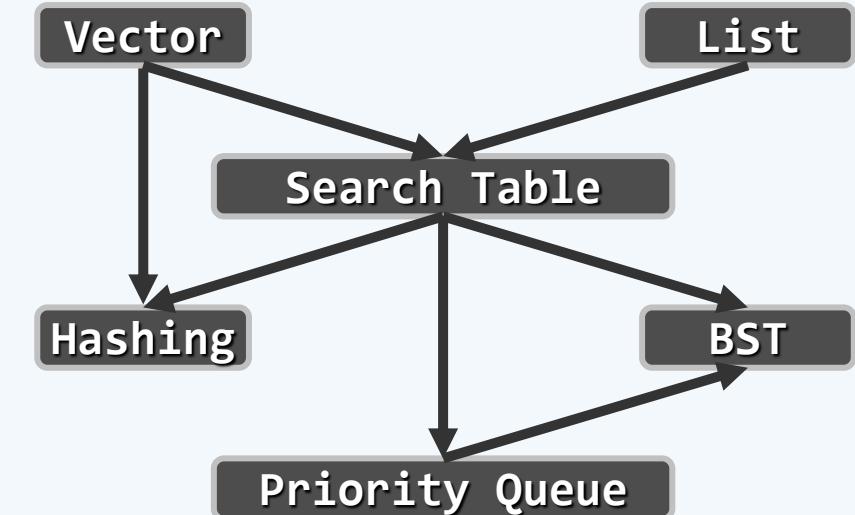
email系统中，是否存在自动转发或回复的回路

...

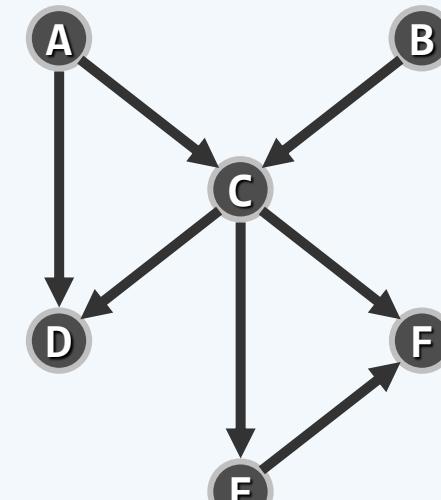
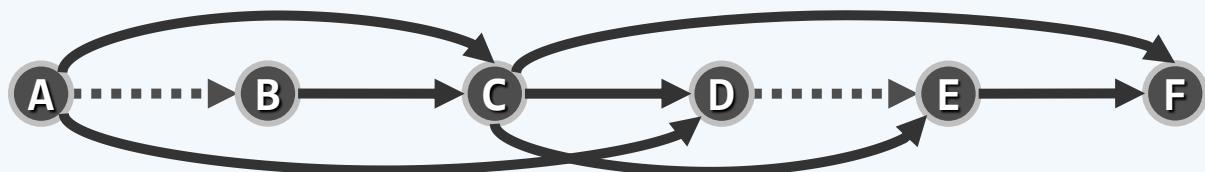


## 拓扑排序

- 任给有向图G，不一定是DAG
- 尝试将其中顶点排成一个线性序列
  - 其次序须与原图相容，亦即
  - 每一顶点都不会通过边指向前驱顶点



- 算法要求
  - 若原图存在回路（即并非DAG），检查并报告
  - 否则，给出一个相容的线性序列



## 存在性

❖ 每个DAG对应于一个偏序集；拓扑排序对应于一个全序集

所谓的拓扑排序，即构造一个与指定偏序集相容的全序集

❖ 可以拓扑排序的有向图，必定无环 //反之

任何DAG，都存在（至少）一种拓扑排序？是的！ //为什么…

❖ 有限偏序集必有极大/极大元素

任何DAG都存在（至少）一种拓扑排序

❖ 可归纳证明，并直接导出一个算法…

## 存在性

1. 任何DAG，必有（至少一个）顶点入度为零 //记作 $m$
  2. 若  $DAG \setminus \{m\}$  存在拓扑排序  $S = \langle u_{k_1}, \dots, u_{k(n-1)} \rangle$  //subtraction  
则  $S' = \langle m, u_{k_1}, \dots, u_{k(n-1)} \rangle$  即为DAG的拓扑排序 //DAG子图亦为DAG
- ❖ 只要 $m$ 不唯一，拓扑排序也应不唯一 //反之呢？

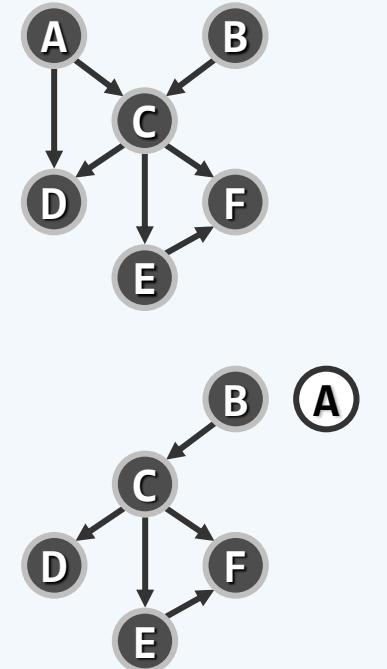
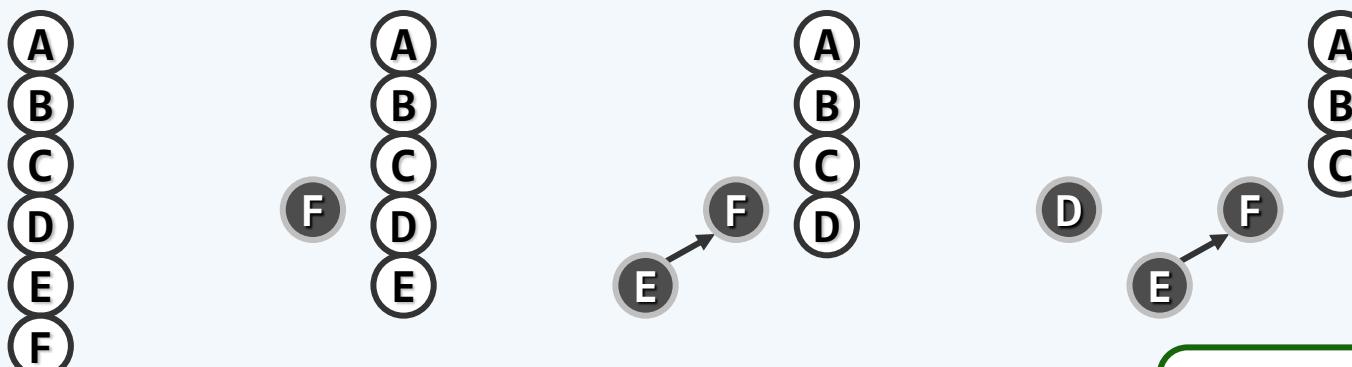
## 算法A：顺序输出零入度顶点

将所有入度为零的顶点存入栈S，取空队列Q // $O(n)$

```

while ( ! S.empty() ) { // $O(n)$ 
    Q.enqueue( v = S.pop() ); //栈顶v转入队列
    for each edge( v, u ) //v的邻接顶点u若入度仅为1
        if ( u.inDegree < 2 ) S.push( u ); //则入栈
        G = G \ { v }; //删除v及其关联边（邻接顶点入度减1）
} //总体 $O(n + e)$ 
return |G| ? Q : "NOT_A_DAG"; //残留的G空，当且仅当原图可拓扑排序

```



## 6. 图

拓扑排序

零出度算法

邓俊辉

deng@tsinghua.edu.cn

## 算法B：逆序输出零出度顶点

❖ /\* 基于DFS，借助栈S \*/

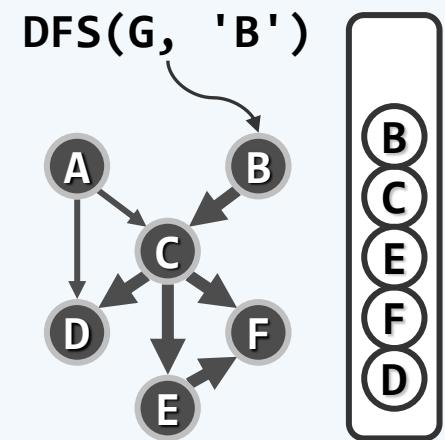
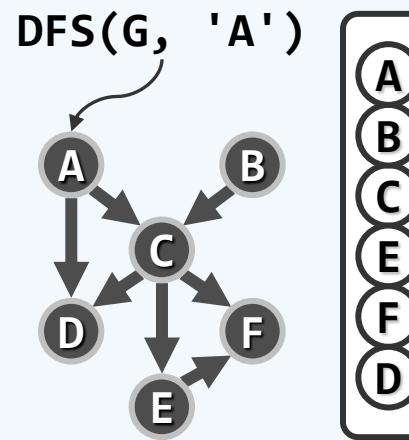
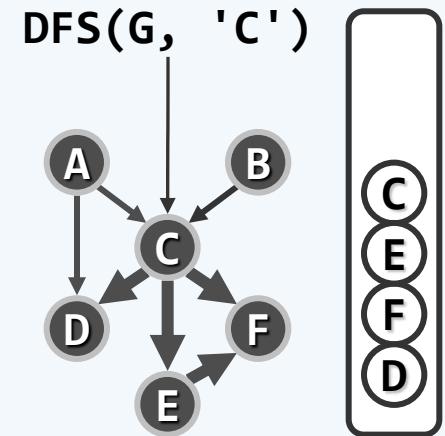
对图G做DFS，其间 //得到组成DFS森林的一系列DFS树

每当有顶点被标记为VISITED，则将其压入S

一旦发现有后向边，则报告非DAG并退出

DFS结束后，顺序弹出S中的各个顶点

❖ 复杂度与DFS相当，也是 $\theta(n + e)$



## 实现 ( 1/2 )

❖ template <typename Tv, typename Te> //顶点类型、边类型

```
bool Graph<Tv, Te>::TSort(int v, int & clock, Stack<Tv>* S) {  
    dTime(v) = ++clock; status(v) = DISCOVERED; //发现顶点v  
    for ( int u = firstNbr(v); -1 < u; u = nextNbr(v, u) ) //枚举v所有邻居u  
        /* ... 视u的状态，分别处理 ... */  
    status(v) = VISITED; S->push( vertex(v) ); //顶点被标记为VISITED时入栈  
    return true;  
}
```

## 实现 ( 2/2 )

```

❖ for ( int u = firstNbr(v); -1 < u; u = nextNbr(v, u) ) //枚举v所有邻居u

    switch ( status(u) ) { //并视u的状态分别处理

        case UNDISCOVERED:

            parent(u) = v; type(v, u) = TREE; //树边(v, u)

            if ( ! TSort(u, clock, S) ) return false; break; //从顶点u处深入

        case DISCOVERED: //一旦发现后向边 ( 非DAG )

            type(v, u) = BACKWARD; return false; //则退出而不再深入

        default: //VISITED (digraphs only)

            type(v, u) = dTime(v) < dTime(u) ? FORWARD : CROSS; break;

    }

```

## 6. 图

优先级搜索

邓俊辉

deng@tsinghua.edu.cn

## 通用算法

❖ 各种遍历算法的区别，仅在于选取顶点进行访问的次序

**广度** / **深度**：优先访问与**更早** / **更晚**被发现的顶点相邻接者

    ...

❖ 不同的遍历算法，取决于顶点的**选取策略**

❖ 不同的顶点选取策略，取决于存放和提供顶点的**数据结构——Bag**

❖ 此类结构，为每个顶点 $v$ 维护一个**优先级数**  $\text{priority}(v)$

    每个顶点都有**初始**优先级数；并可能随算法的推进而**调整**

❖ 通常的习惯是，优先级数越**大/小**，优先级越**低/高**

    特别地， $\text{priority}(v) == \text{INT\_MAX}$ ，意味着 $v$ 的优先级最低

## 统一框架

❖ template <typename Tv, typename Te> //顶点类型、边类型

template <typename PU> //优先级更新器（函数对象）

void Graph<Tv, Te>::pfs( int s, PU prioUpdater ) { //PU的策略，因算法而异

    priority(s) = 0; status(s) = VISITED; parent(s) = -1; //起点s加至PFS树中

    while (1) { //将下一顶点和边加至PFS树中

        /\* ... 依次引入n - 1个顶点（和n - 1条边） ... \*/

    } //while

} //如何推广至非连通图？

## 统一框架

```
❖ while (1) { //依次引入n - 1个顶点(和n - 1条边)  
    for ( int w = firstNbr(s); -1 < w; w = nextNbr(s, w) ) //对s各邻居w  
        prioUpdater( this, s, w ); //更新顶点w的优先级及其父顶点  
    for ( int shortest = INT_MAX, w = 0; w < n; w++ )  
        if ( UNDISCOVERED == status(w) ) //从尚未加入遍历树的顶点中  
            if ( shortest > priority(w) ) //选出下一个  
                { shortest = priority(w); s = w; } //优先级最高的顶点s  
        if ( VISITED == status(s) ) break; //直至所有顶点均已加入  
        status(s) = VISITED; type( parent(s), s ) = TREE; //将s加入遍历树  
    } //while
```

## 复杂度

- ❖ 执行时间主要消耗于内、外两重循环；其中两个内循环前、后并列
- ❖ 前一内循环的累计执行时间：若采用邻接矩阵，为 $\Theta(n^2)$ ；若采用邻接表，为 $\Theta(n + e)$
- 后一循环中，优先级更新的次数呈算术级数变化{ n, n - 1, …, 2, 1 }，累计为 $\Theta(n^2)$
- 两项合计，为 $\Theta(n^2)$
- ❖ 后面将会看到：若采用优先队列，以上两项将分别是 $\Theta(e \log n)$ 和 $\Theta(n \log n)$  //保持兴趣
- 两项合计，为 $\Theta((e + n) * \log n)$
- ❖ 这是很大的改进——尽管对于稠密图而言，反而是倒退 //已有接近于 $\Theta(e + n \log n)$ 的算法
- ❖ 基于这个统一框架，如何解决具体的应用问题…

## 6. 图

Prim算法

最小支撑树

“疯子么，怎能绳之以常理？还有更荒唐的事呢，  
他要在普济造一条风雨长廊，把村里的每一户人家  
都连接起来，哈哈，他以为，这样一来，普济人就  
可免除日晒雨淋之苦了。”

邓俊辉

deng@tsinghua.edu.cn

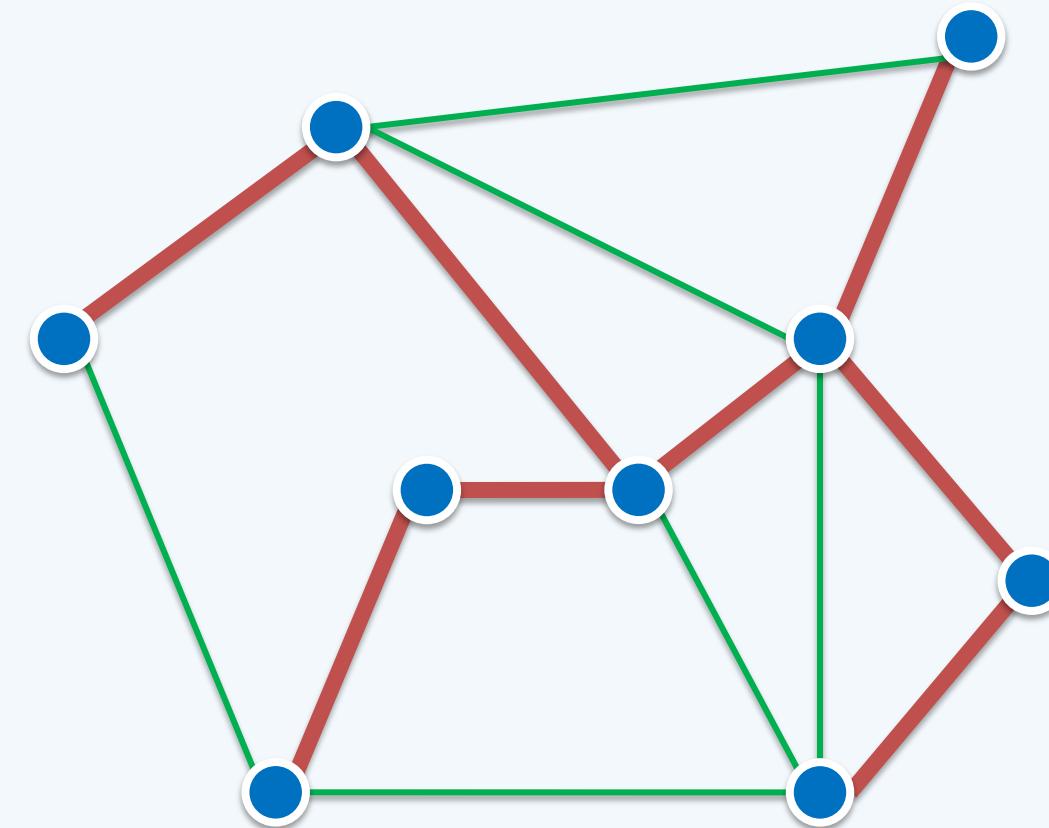
## 最小 + 支撑 + 树

❖ 连通网络  $N = (V; E)$  的子图  $T = (V; F)$

❖ 支撑/spanning = 覆盖  $N$  中所有顶点

❖ 树/tree =

- 连通且无环 ,  $|V| = |F| + 1$
- 加边出单环 , 再删同环边即恢复为树
- 删边不连通 , 再加联接边即恢复为树



❖ 不难验证 , 同一网络的支撑树不唯一

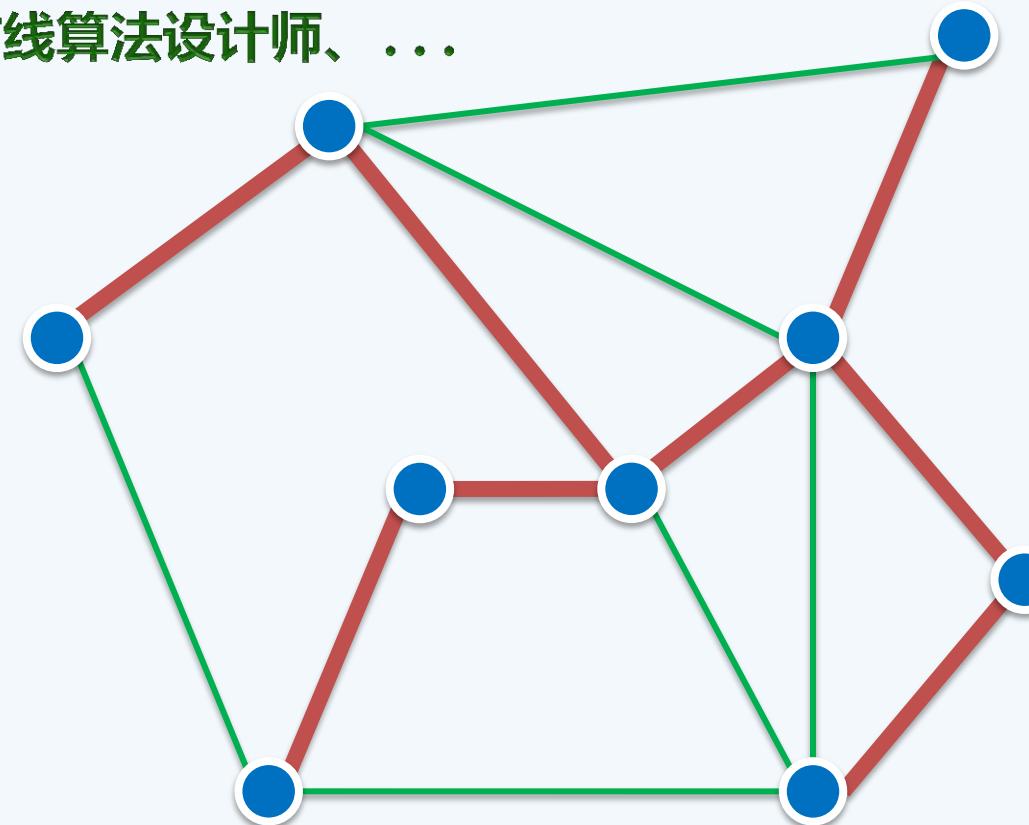
❖ 最小/minimum = 各边总权重  $\text{wt}(T) = \sum_{e \in F} \text{wt}(e)$  达到最小

❖ 谁感兴趣？电信公司、网络设计师、VLSI布线算法设计师、...

❖ 为何重要？

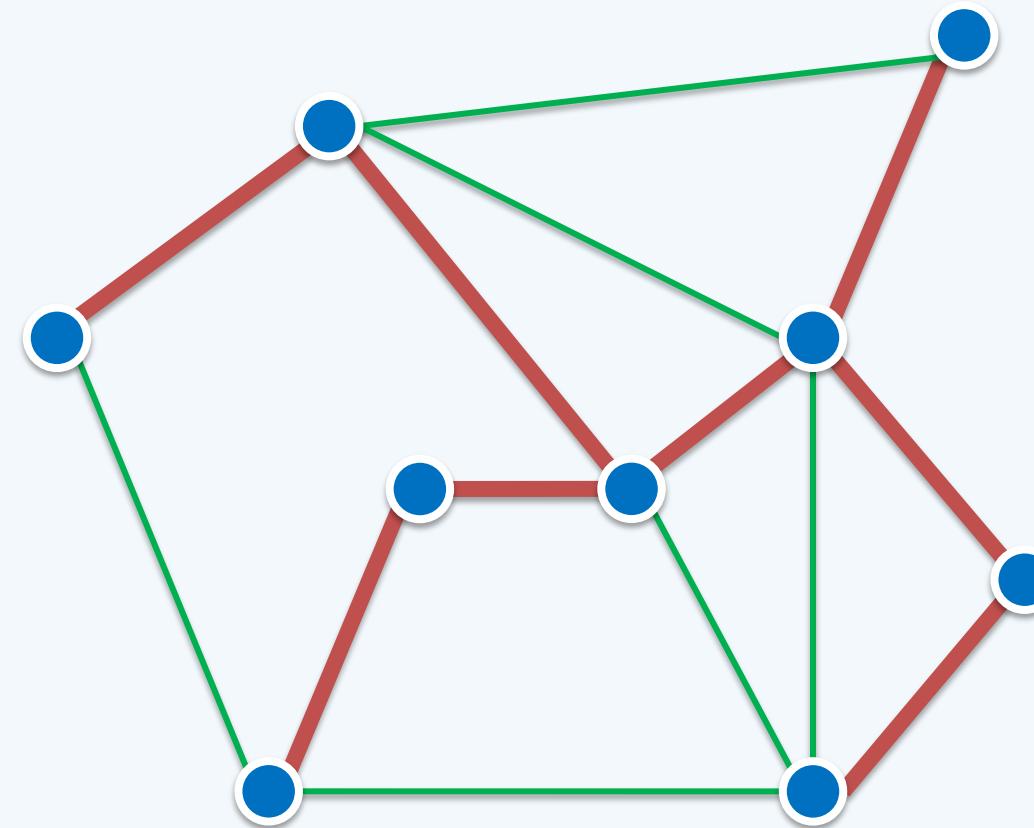
- 应用中常见的共性问题
- 也是很多优化问题的**基本模型**
- 自身可**有效计算**
- 为许多NP问题提供足够好的**近似解**

比如，Euclidean TSP



## 已有算法

- ❖ Boruvka-1926
- ❖ Jarnik-1930
- ❖ Prim-1956
- ❖ Kruskal-1956
- ❖ Karger-Klein-Tarjan-1995
- ❖ Chazelle-2000
- ❖ ...是否...存在 $\theta(n + e)$ 算法?

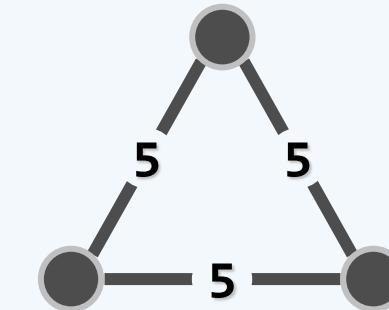
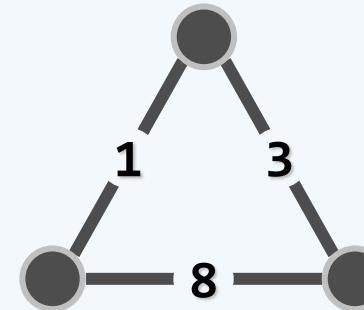
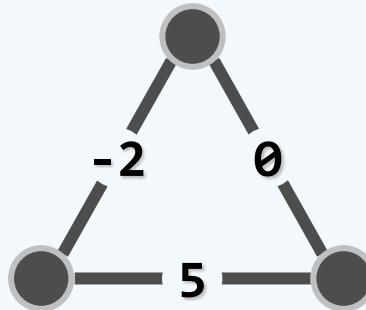


## 退化

❖ 权值必须是正数？

允许为零，会有什么影响？

允许为负数呢？



❖ 所有支撑树所含的边数，必然相等

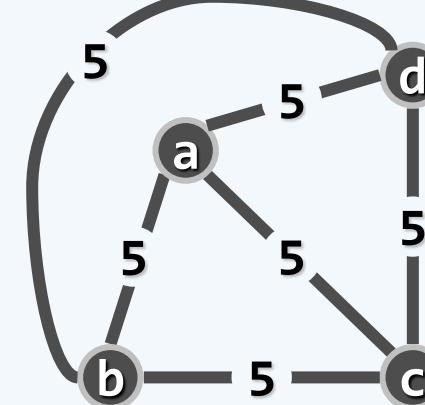
故可统一调整：`increase( 1 - findMin() )`

❖ The minimum？

- A minimal 同一网络N可能有多棵MST
- The minimum！ 可强制消除歧义...

❖ 合成数 ( composite number ) : (  $w(u, v)$ ,  $\min(u, v)$ ,  $\max(u, v)$  )

$$5ab < 5ac < 5ad < 5bc < 5bd < 5cd$$



## 蛮力算法

❖ 枚举出N的所有支撑树，从中找出代价最小者

❖ 这一策略是否可行，取决于…

❖ n个互异顶点组成的图，可能有多少棵支撑树？

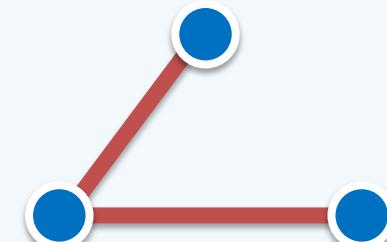
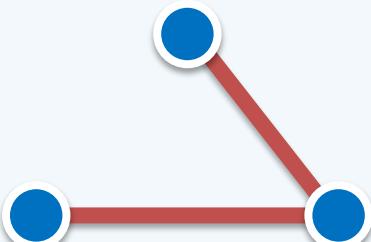
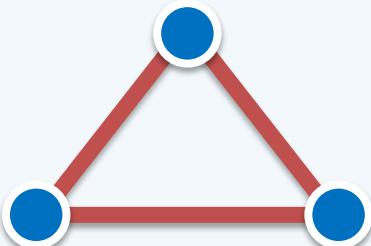
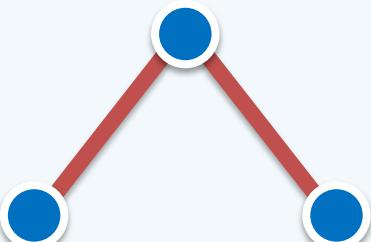
$n = 1 \quad 1$

$n = 2 \quad 1$

$n = 3 \quad 3$

$n = 4 \quad 16$

... ...



❖ Cayley公式：联接n个互异顶点的树共有 $n^{n-2}$ 棵；或等价地，完全图 $K_n$ 有 $n^{n-2}$ 棵支撑树

❖ 如何高效地构造MST呢？

## 6. 图

Prim算法

极短跨边

邓俊辉

deng@tsinghua.edu.cn

## 割 & 极短跨边

❖ 设 $(U; V \setminus U)$ 是 $N$ 的割 cut

❖ 【Cut Property-A】

若： $(u, v)$ 是该割的极短跨边 ( shortest crossing edge )

则：必存在一棵包含 $(u, v)$ 的MST

❖ 反证：假设 $(u, v)$ 未被任何MST采用...

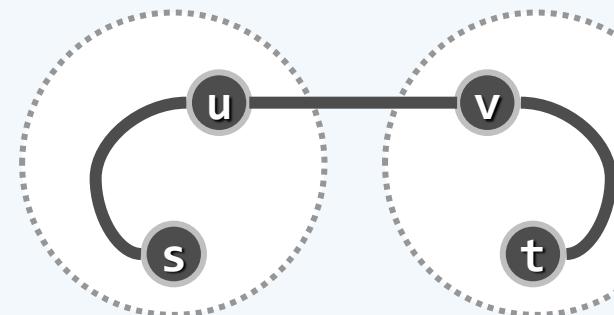
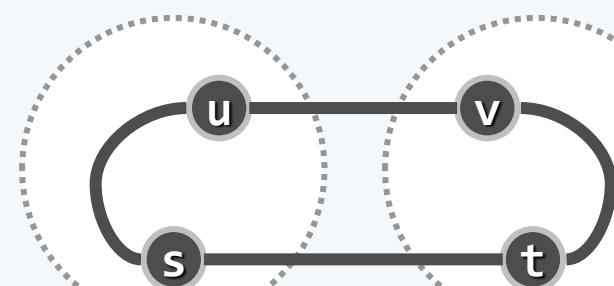
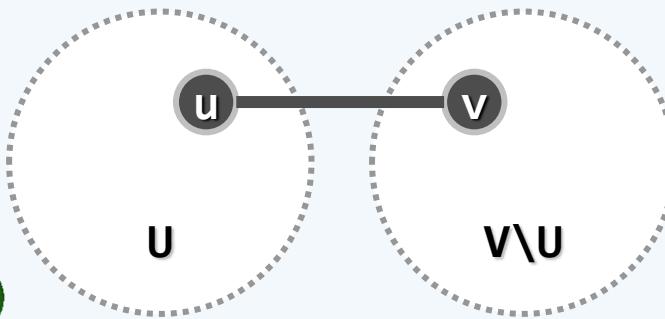
❖ 任取一棵MST，将 $(u, v)$ 加入其中，于是

将出现唯一的回路，且该回路必经过

$(u, v)$ 以及至少另一跨边 $(s, t)$

❖ 现在，将原MST中的 $(s, t)$ 替换为 $(u, v)$ ...

❖ 【Cut Property-B】反之， $N$ 的任一MST也必通过极短跨边联接每一割



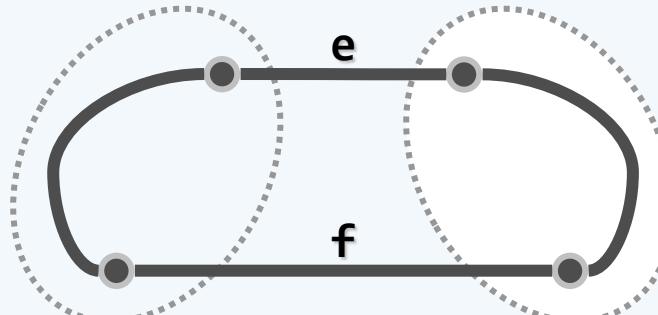
## 环 & 极长环边

❖ 设 $T$ 是 $N$ 的一棵MST，且在 $N$ 中添加边 $e$ 后得到 $N'$

❖ 【Cycle Property】

若：沿着 $e$ 在 $T$ 中对应的环路， $f$ 为一极长边

则： $T - \{f\} + \{e\}$ 即为 $N'$ 的一棵MST



❖ 1) 若 $e$ 为环路上的最长边，则与前同理， $e$ 不可能属于 $N'$ 的MST

此时， $f = e$ ， $T - \{f\} + \{e\} = T$ 依然是 $N'$ 的MST

❖ 2) 否则有： $|e| \leq |f|$ ；移除 $f$ 后 $T - \{f\}$ 一分为二，对应于 $N/N'$ 的割

在 $N/N'$ 中， $f/e$ 应是该割的极短跨边

此割在 $N$ 和 $N'$ 中导出的一对互补子图完全一致

故，这对子图各自的MST经 $e$ 联接后，即是 $N'$ 的一棵MST

## 算法

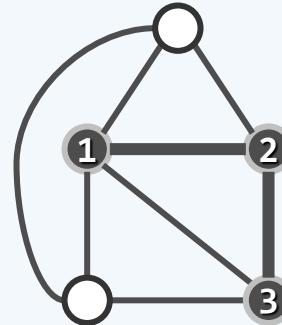
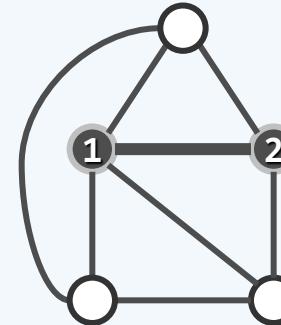
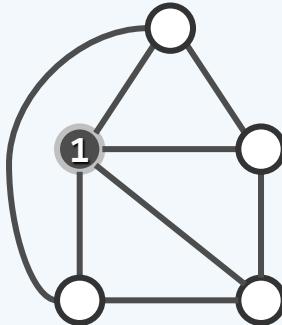
❖ 从  $T_1 = (\{v_1\}; \emptyset)$  开始，逐步构造  $T_2$ 、 $T_3$ 、 $\dots$ 、 $T_n$ ，其中

- $v_1$  可以任选

- $T_k = (V_k; E_k)$

$$|V_k| = k, |E_k| = k-1$$

$$V_k \subset V_{k+1}$$

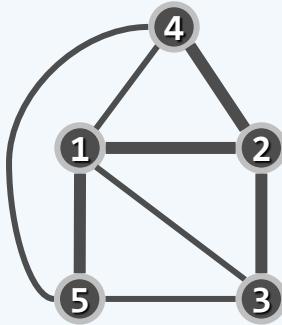
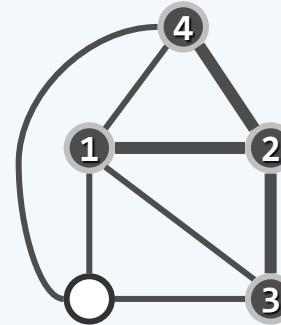


❖ 由以上分析，为由  $T_k$  构造  $T_{k+1}$ ，只需

- 将  $(V_k : V \setminus V_k)$  视作原图的一个割

- 在该割的所有跨边中，找出极短者  $e_k = (v_k, u_k)$

- 令  $T_{k+1} = (V_{k+1}; E_{k+1}) = (V_k \cup \{u_k\}; E_k \cup \{e_k\})$



## 6. 图

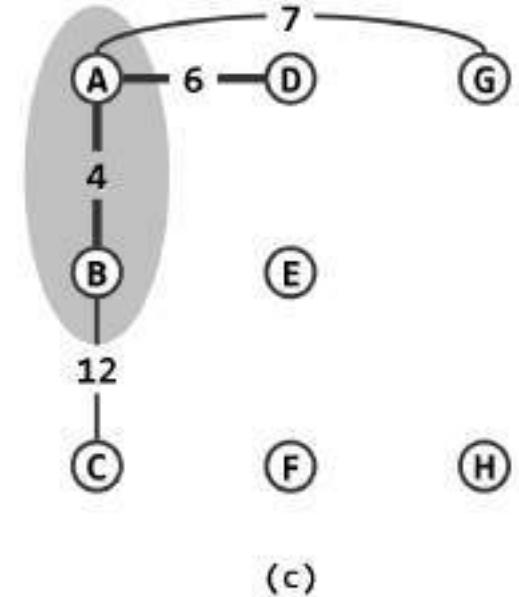
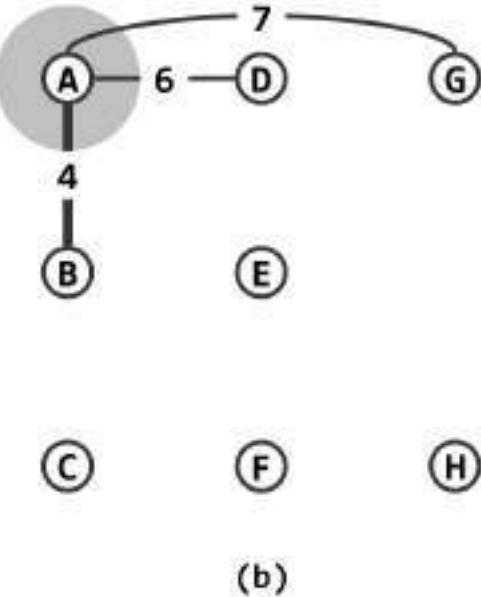
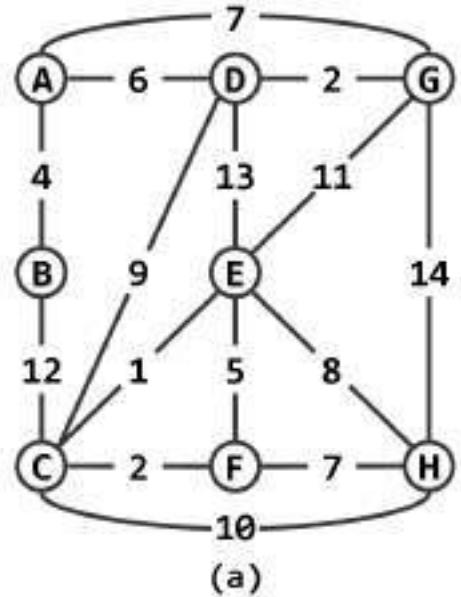
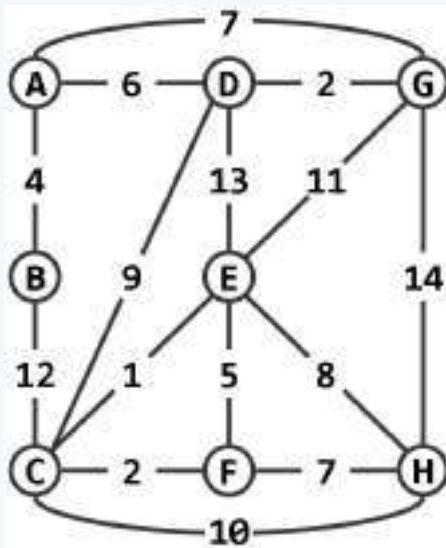
Prim算法

实例

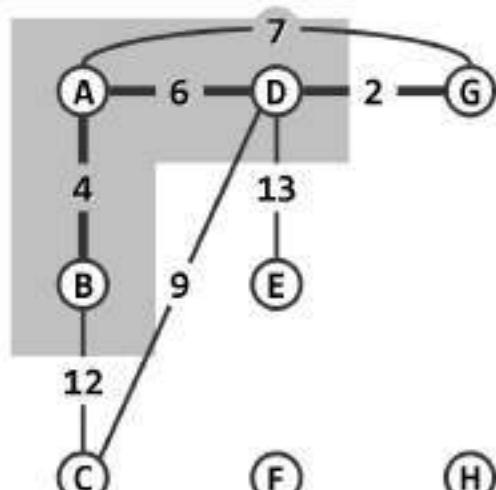
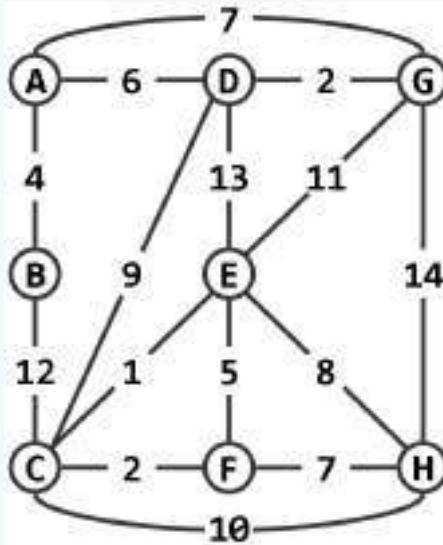
邓俊辉

deng@tsinghua.edu.cn

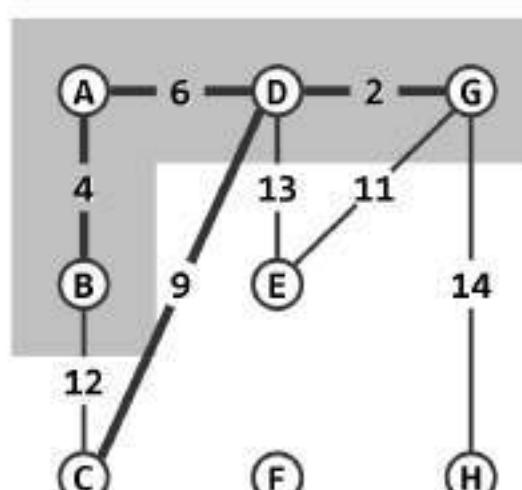
1/3



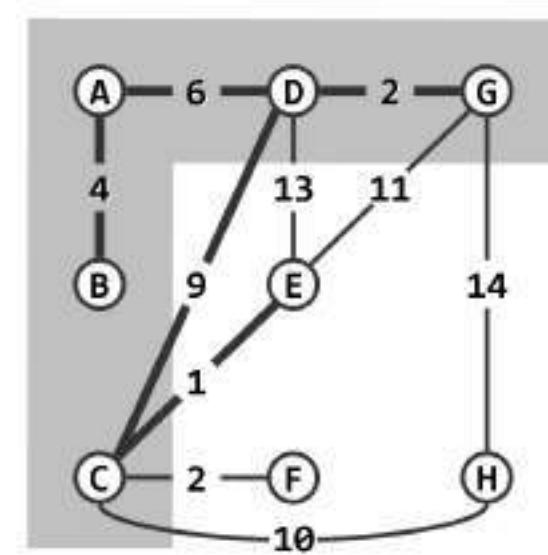
2/3



(d)

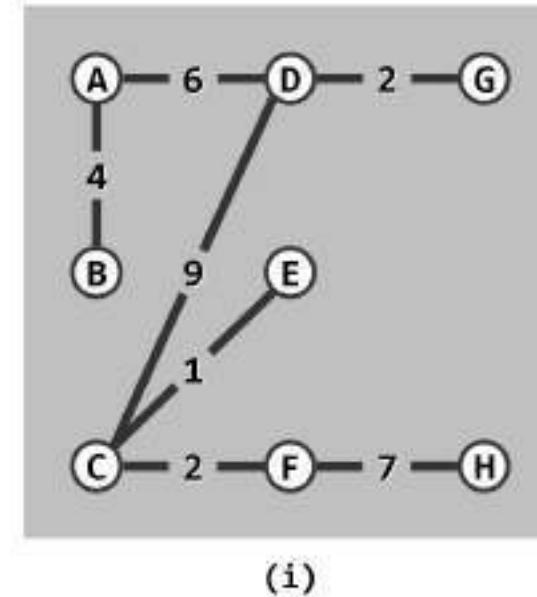
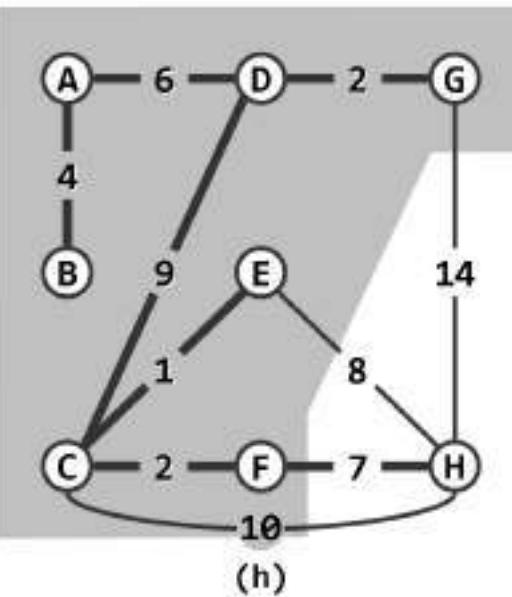
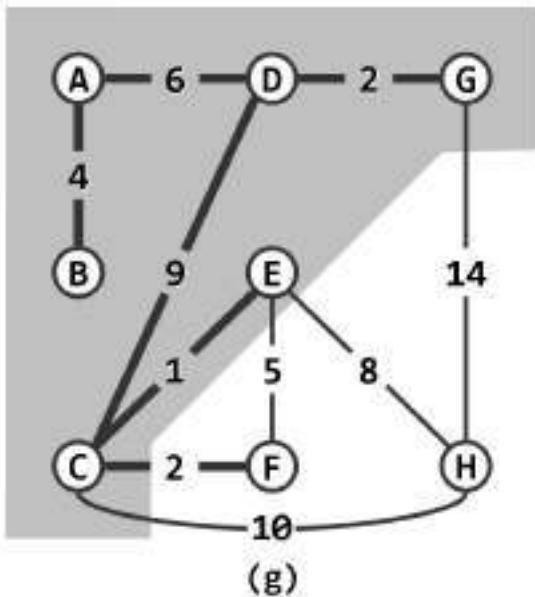
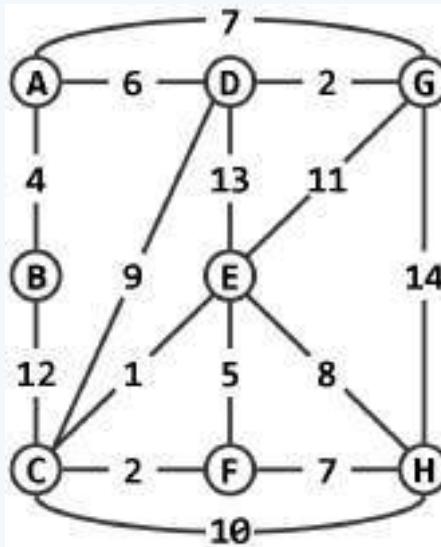


(e)



(f)

3/3



## 6. 图

Prim算法  
正确性

邓俊辉

deng@tsinghua.edu.cn

## 似是而非

❖ 设Prim依次选取边{  $e_2, e_3, \dots, e_n$  }，构造出树T

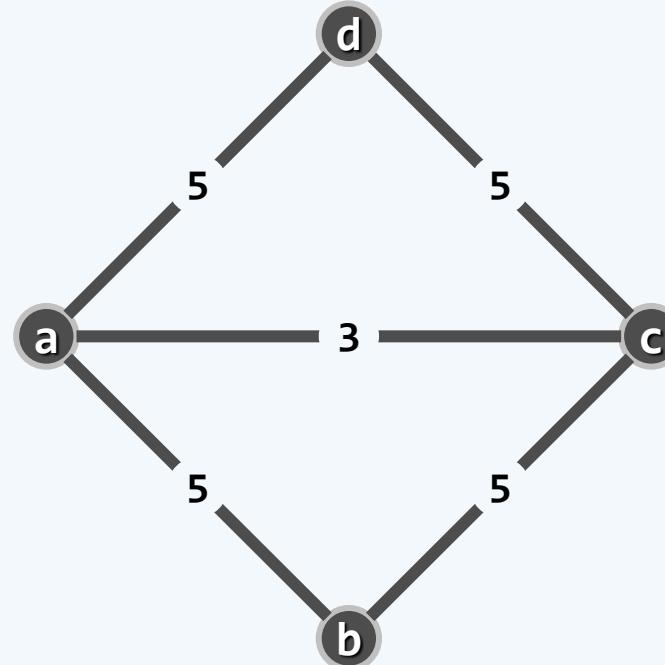
❖ 其中每一条边 $e_k$ ，的确都属于某棵MST

❖ 但在MST不唯一时...

由此并不能确认，最终的T必是MST（之一）

❖ 由极短跨边构成的支撑树，未必就是一棵MST

❖ 反例...



## 可行的证明方法

❖ 在不增加总权重的前提下

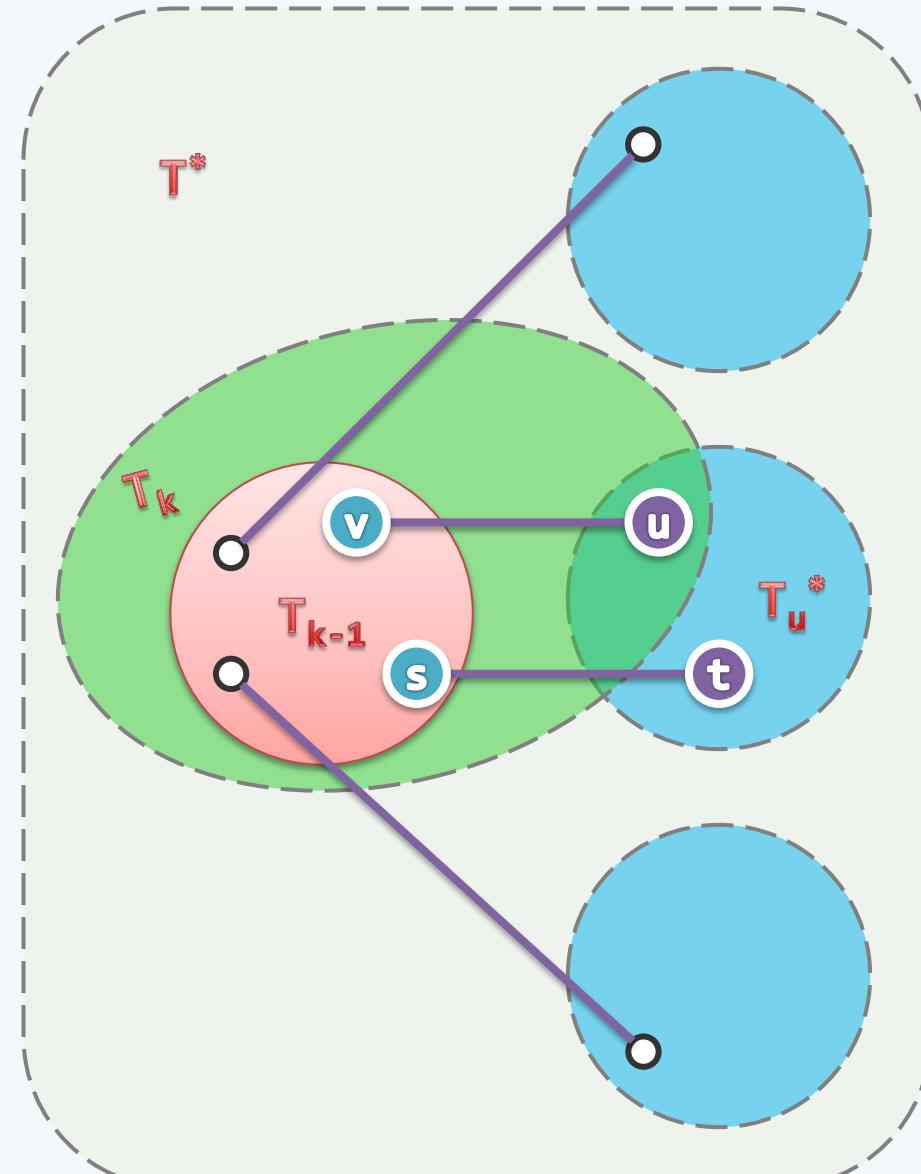
可以将任一MST [转换] 为  $T$

每一  $T_k$  都是某棵MST的 [子树] ,  $1 \leq k \leq n$

❖ 《习题解析》 , 6-28题

数学归纳

...



## 6. 图

Prim算法

实现

傍边一将，圆睁环眼，倒竖虎须，挺丈八蛇矛，飞马大叫：‘三姓家奴休走！燕人张飞在此！」吕布见了，弃了公孙瓒，便战张飞。

邓俊辉

deng@tsinghua.edu.cn

❖ 对于  $v_k$  之外的每一顶点  $v$ ，令：

$\text{priority}(v) = v \text{ 到 } v_k \text{ 的距离}$

于是套用优先级遍历算法框架...

❖ 为将  $T_k$  扩充至  $T_{k+1}$ ，可以

- 选出优先级最高的（极短）跨边  $e_k$ ，及其对应顶点  $u_k$ ，并将其加入  $T_k$
- 随后，更新  $v_{k+1}$  之外每一顶点的优先级（数）

❖ 注意：优先级数随后可能改变（降低）的顶点，必与 $u_k$ 邻接

❖ 因此，只需

- 枚举 $u_k$ 的每一邻接顶点 $v$ ，并取
- $\text{priority}(v) = \min(\text{priority}(v), |u_k, v|)$

❖ 以上完全符合PFS的框架，唯一要做的工作无非是

按照

编写一个优先级（数）更新器...

## PrioUpdater()

❖ `g->pfs( 0, PrimPU<char, int>() ); //从顶点0出发，启动Prim算法`

❖ `template <typename Tv, typename Te> //顶点类型、边类型`

```
struct PrimPU { //Prim算法的顶点优先级更新器
    virtual void operator()( Graph<Tv, Te>* g, int uk, int v ) { //对uk的每个
        if ( UNDISCOVERED != g->status(v) ) return; //尚未被发现的邻居v，按
        if ( g->priority(v) > g->weight(uk, v) ) { //Prim
            g->priority(v) = g->weight(uk, v); //策略
            g->parent(v) = uk; //做松弛 (v ~ lv = 吕)
        }
    }
};
```

## 6. 图

Dijkstra算法

最短路径

邓俊辉

deng@tsinghua.edu.cn

## 问题描述

◆ 给定：连通有向图G及其中的顶点u和v

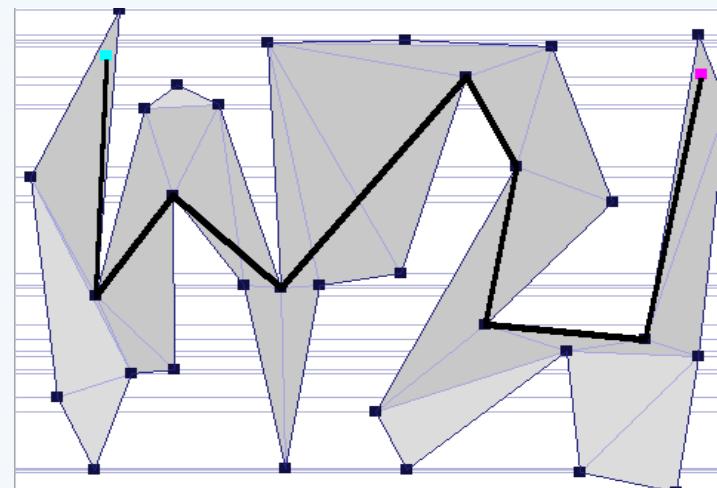
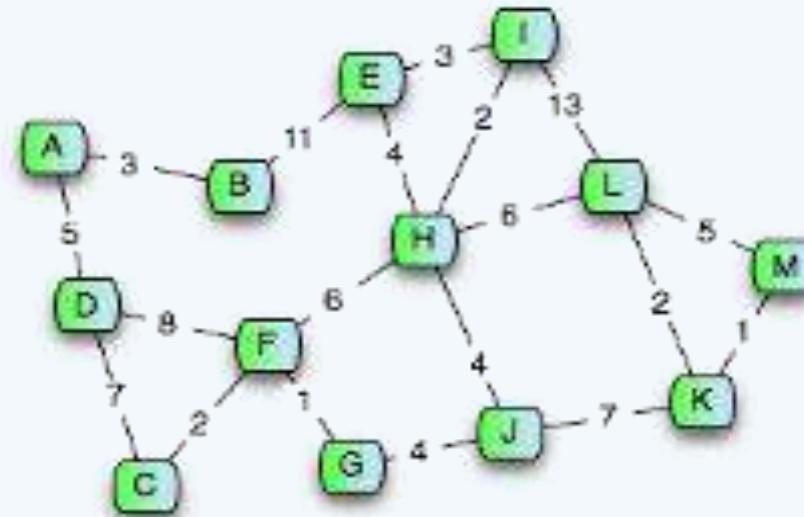
找到：从u到v的最短路径及其长度

◆ 旅游者：最经济的出行路线

路由器：最快地将数据包传送到目标位置

路径规划：多边形区域内的自主机器人

.....



## 问题分类

### ❖ 按照图的类型

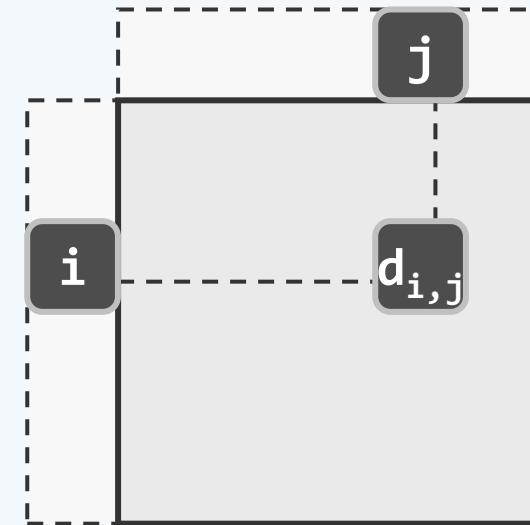
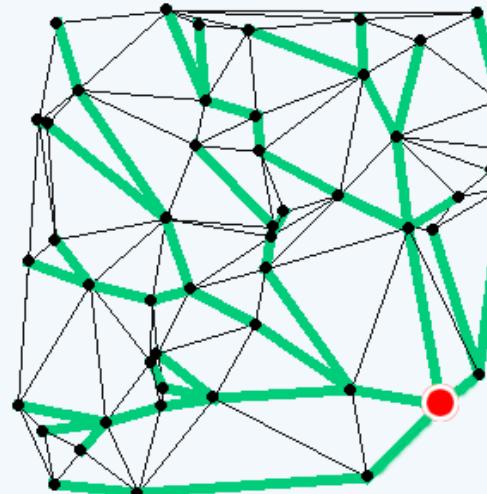
- 无权图/等权图 : BFS
- 带权有向图 //负权值呢 ?

### ❖ 单源点 (Single-source) 到各顶点的最短路径

- 给定顶点  $x$  , 计算  $x$  到其余各个顶点的最短路径及长度
- E. Dijkstra, 1959

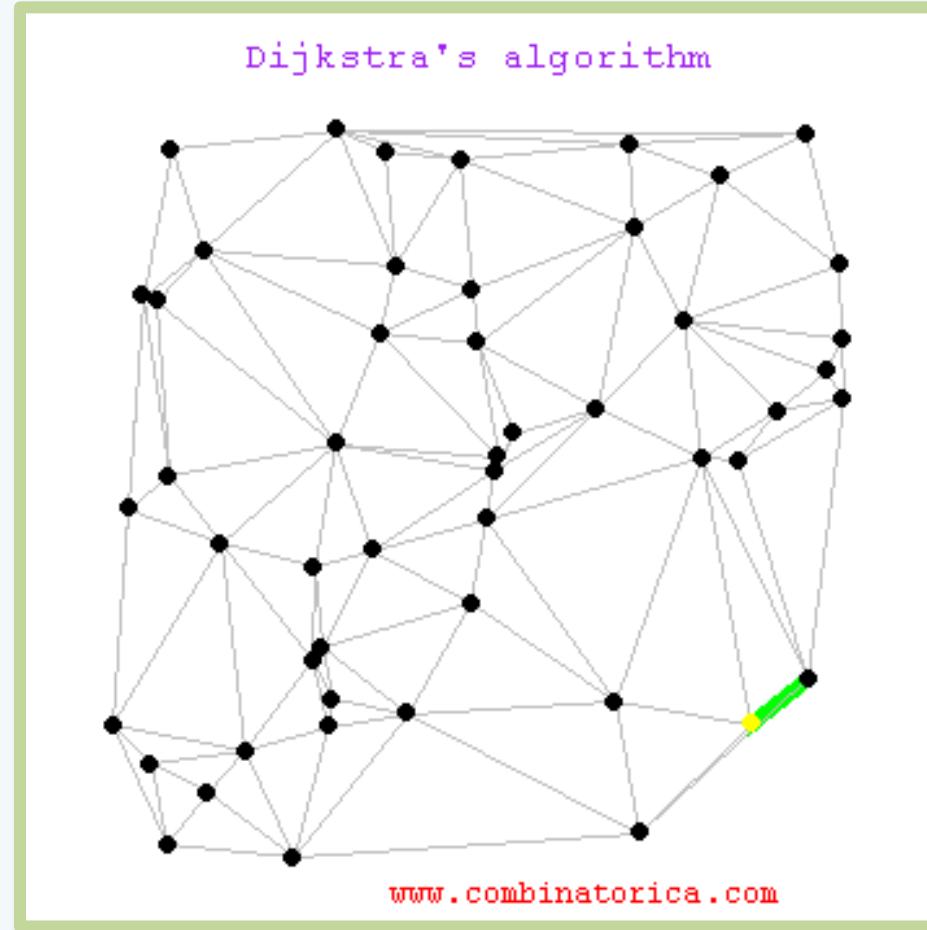
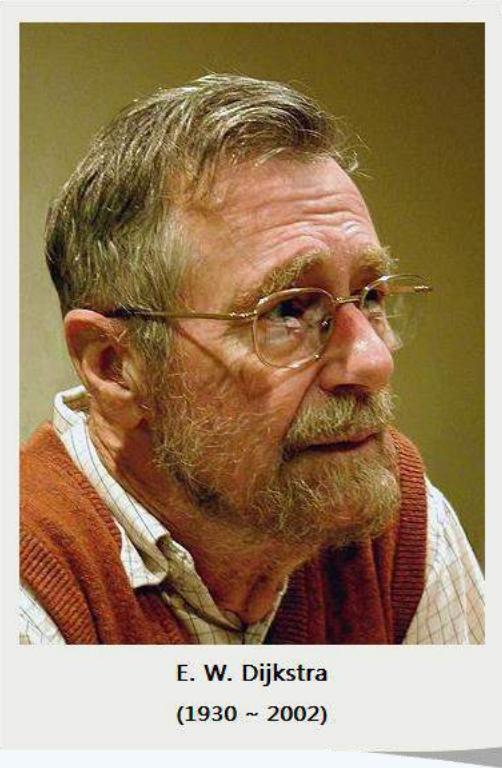
### ❖ 所有顶点对之间的最短路径 (All shortest paths)

- 找出每对顶点  $i$  和  $j$  之间的最短路径及长度
- Floyd-Warshall, 1962



## E. W. Dijkstra

❖ Turing Award, 1972



## 6. 图

Dijkstra算法

最短路径树

邓俊辉

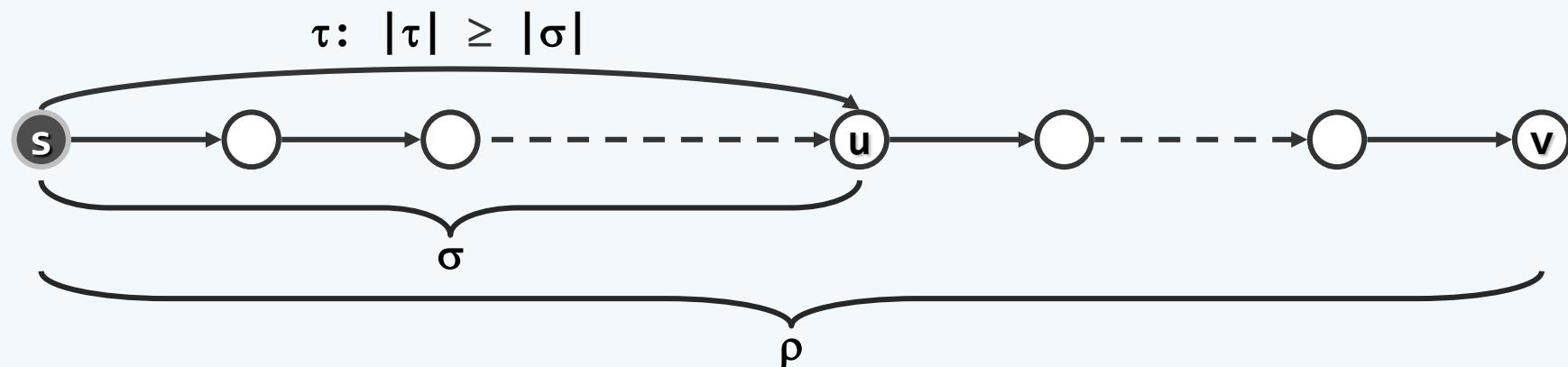
deng@tsinghua.edu.cn

## 最短路径

❖ 连通图中， $s$ 到每个顶点都有（至少）一条最短路径

//非退化假设：每个顶点对应的最短路径唯一

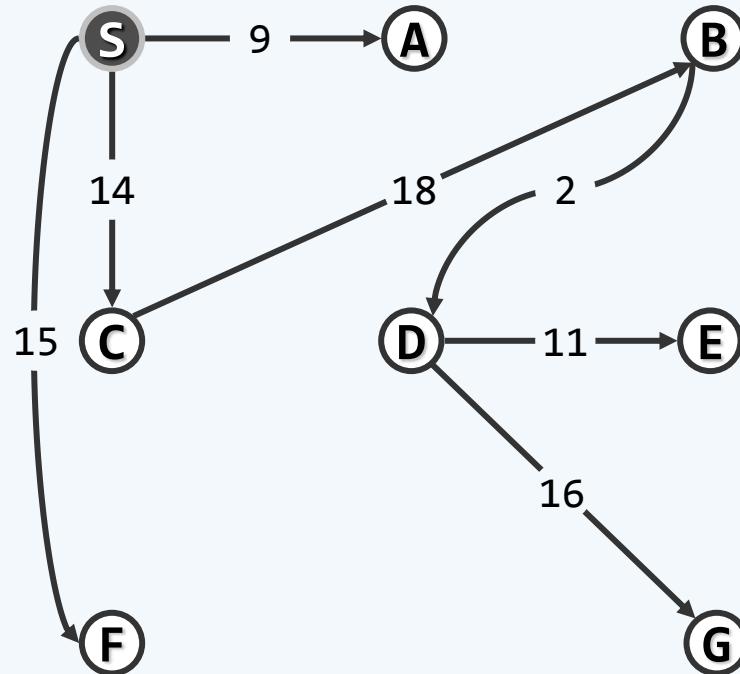
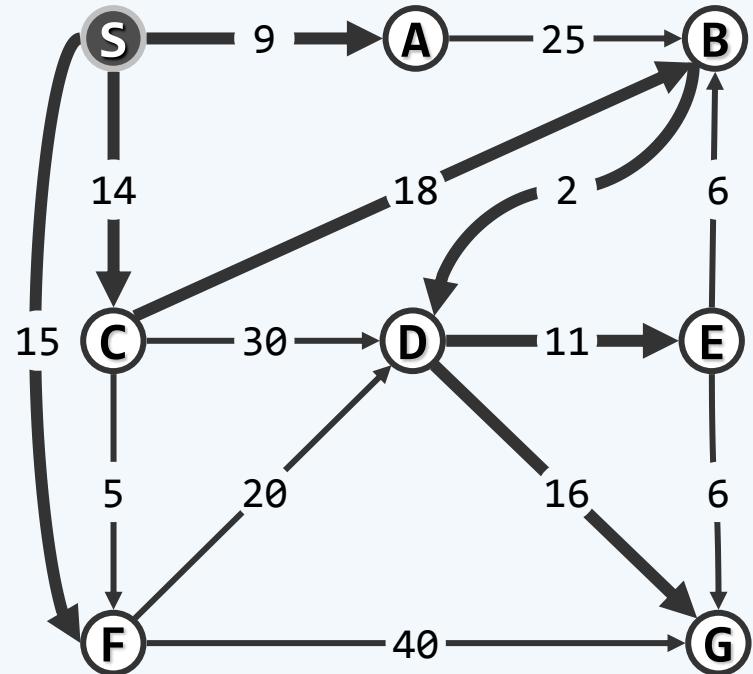
❖ 就同一起点 $s$ 而言，任何最短路径的**前缀**，**也是一条最短路径**



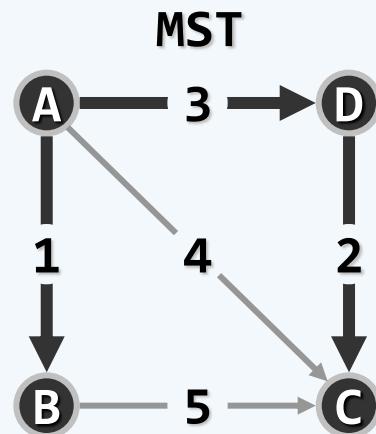
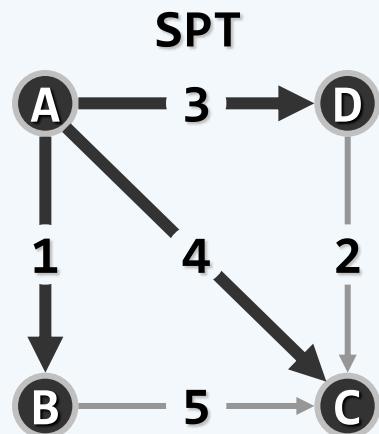
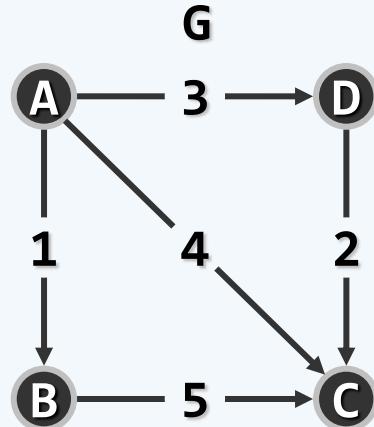
## Shortest Path Tree

❖ 就同一起点s而言，所有最短路径的并，

- 不含回路，因此
- 构成一棵树



SPT  $\neq$  MST



## 6. 图

Dijkstra 算法  
算法

邓俊辉

deng@tsinghua.edu.cn

## 绳索计算机



**u<sub>1</sub>**

❖ 按照到s的最短距离，对其余的顶点排序

$$\text{dist}(s, u_1) \leq \text{dist}(s, u_2) \leq \dots \leq \text{dist}(s, u_{n-1})$$

❖ 最短距离最短者u<sub>1</sub> = ?

❖ 沿任一最短路径，各顶点到s的最短距离单调变化

❖ u<sub>1</sub>必与s直接相联

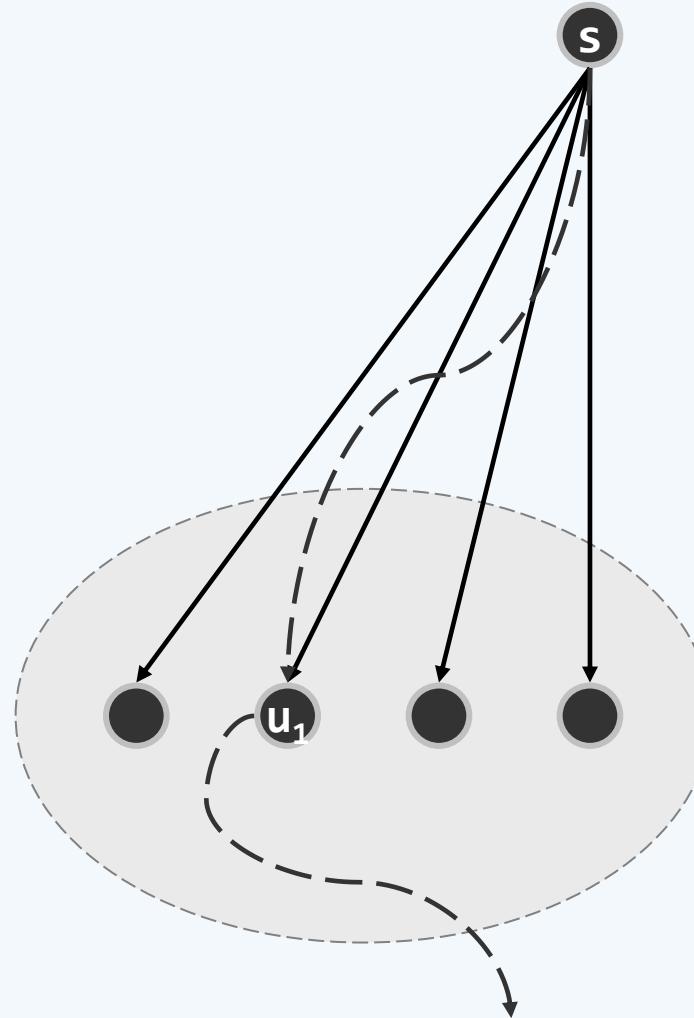
$$\text{dist}(s, s_1) = w(s, s_1) < \infty$$

❖  $\forall u \neq s,$

$$w(s, u) < \infty \text{ 仅当 } w(s, u_1) \leq w(s, u)$$

❖ 为找到u<sub>1</sub>，只需

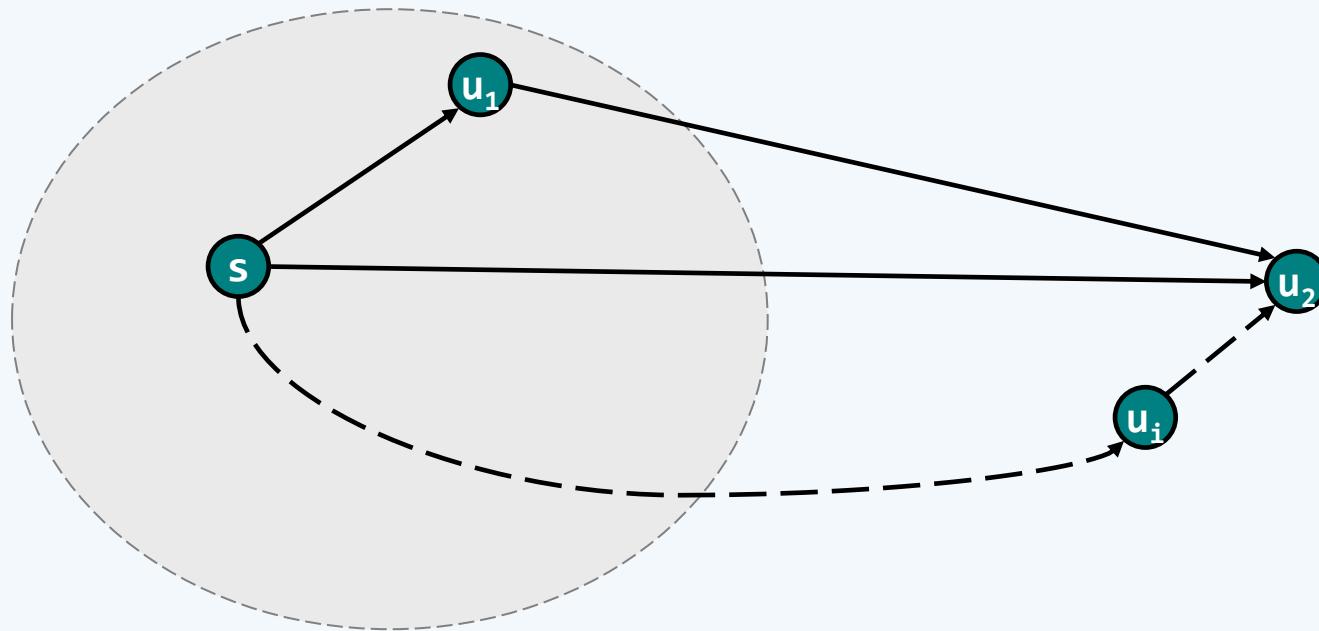
在与s关联的各顶点中，找到对应边权值最小者



$u_2$

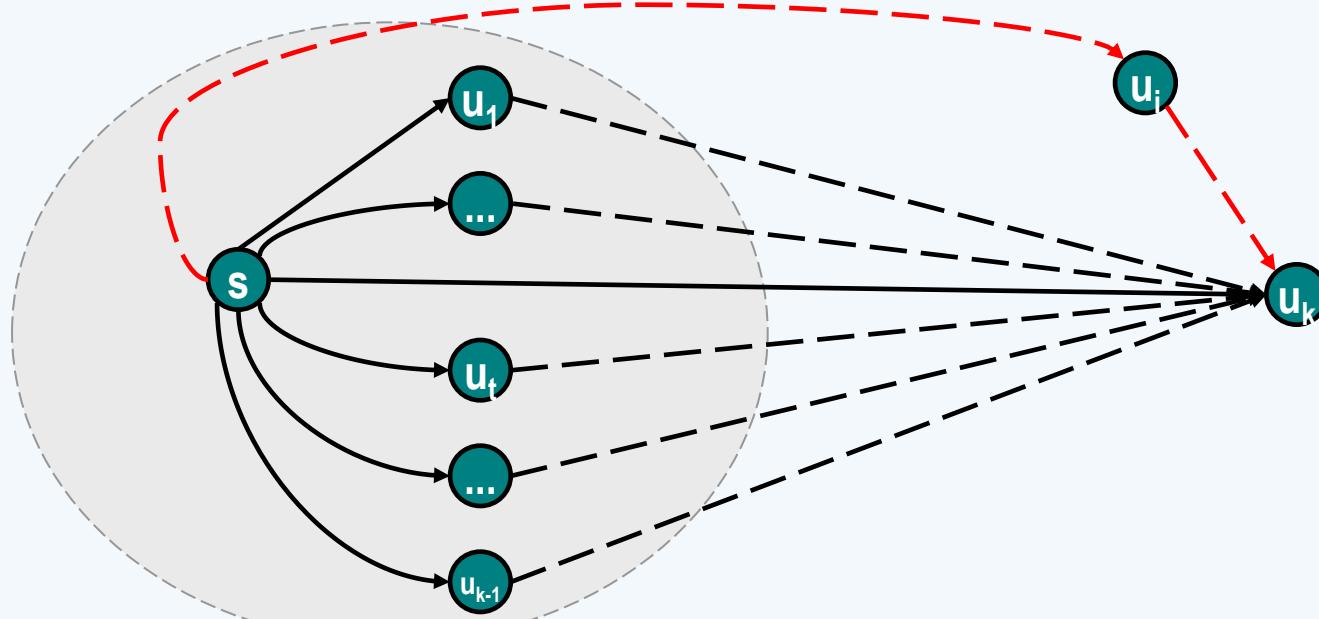
❖ 最短距离次小的顶点  $u_2 = ?$

❖  $\text{dist}(s, u_2) = \min\{ w(s, u_2), \text{dist}(s, u_1) + w(u_1, u_2) \}$



$u_k$

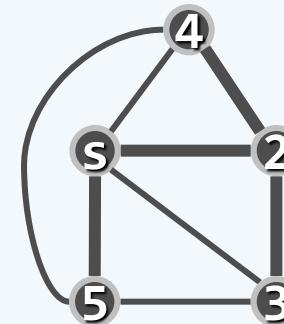
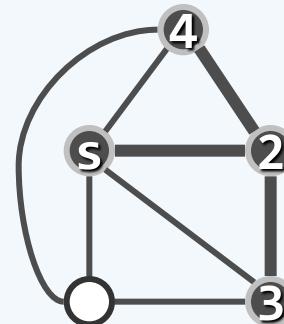
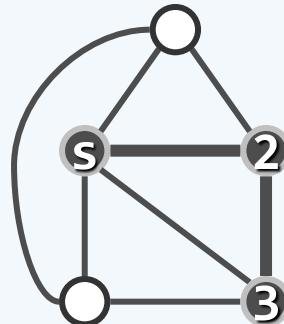
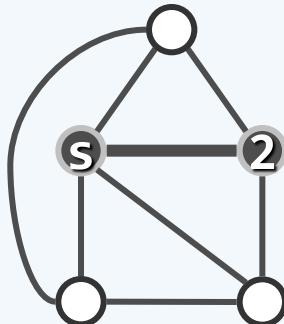
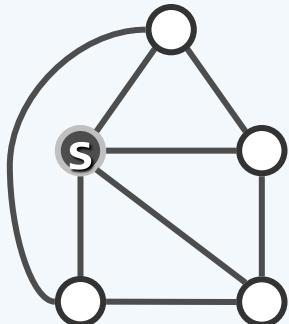
- ❖  $u_3 = ?, u_4 = ?, \dots, u_k = ?$
- ❖ 三角不等式 :  $\text{dist}(s, v) \leq \text{dist}(s, u) + w(u, v)$
- ❖ 若记  $u_0 = s$ , 则有 :  $\text{dist}(s, u_k) = \min\{ \text{dist}(s, u_i) + w(u_i, u_k) \mid 0 \leq i < k \}$
- ❖ 算法 ?



## 策略

❖ 从  $T_1 = (\{v_1\}; \emptyset)$  开始，逐步构造  $T_2, T_3, \dots, T_n$ ，其中

- $v_1 = s$
- $T_k = (V_k; E_k)$
- $|V_k| = k$
- $|E_k| = k-1, V_k \subset V_{k+1}$



## 算法

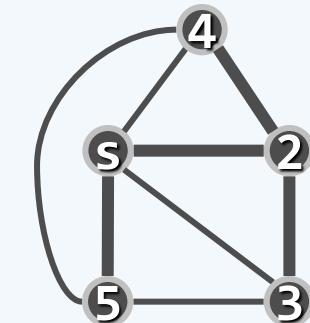
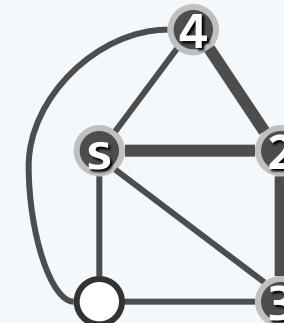
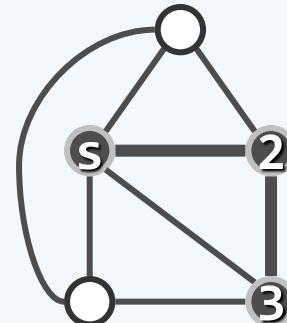
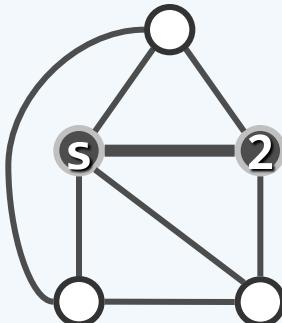
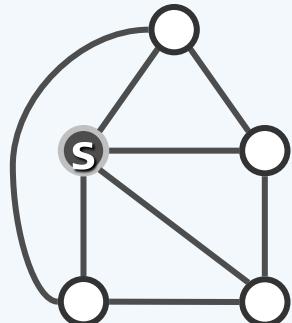
❖ 由以上分析，为由 $T_k$ 构造 $T_{k+1}$ ，只需

- 将 $(V_k : V \setminus V_k)$ 视作原图的一个割

- 在该割的所有**跨边**中

找出**极近者** $e_k = (v_k, u_k)$  ( $u_k$ 到 $s$ 距离**极近**)

- 令 $T_{k+1} = (V_{k+1}; E_{k+1}) = (V_k \cup \{u_k\}; E_k \cup \{e_k\})$



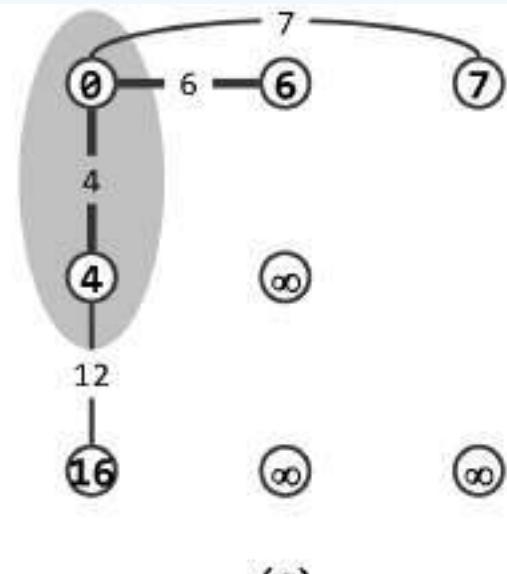
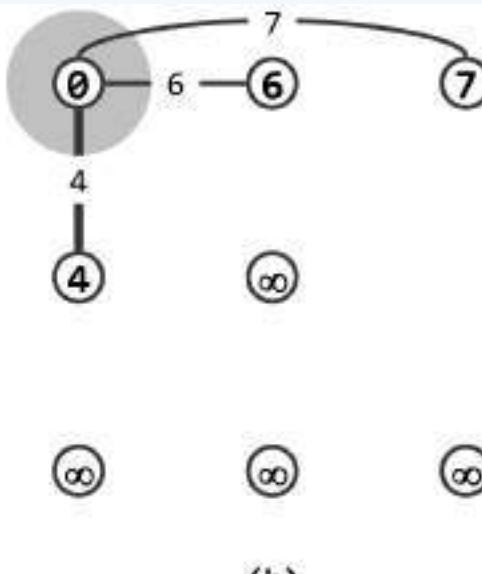
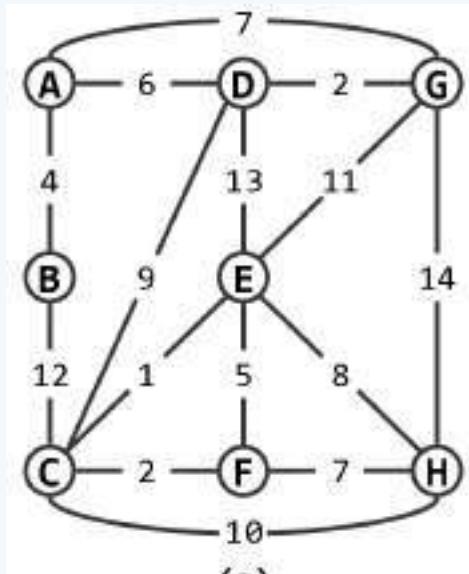
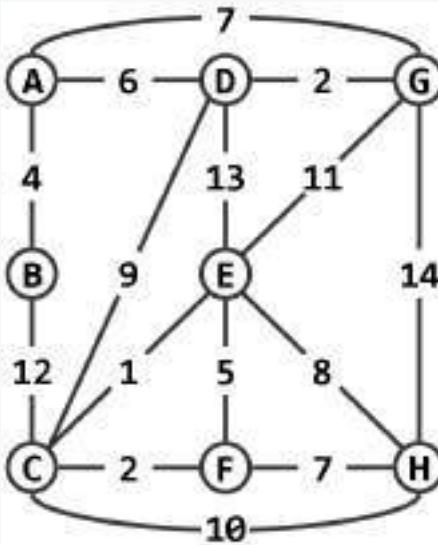
## 6. 图

Dijkstra 算法  
实例

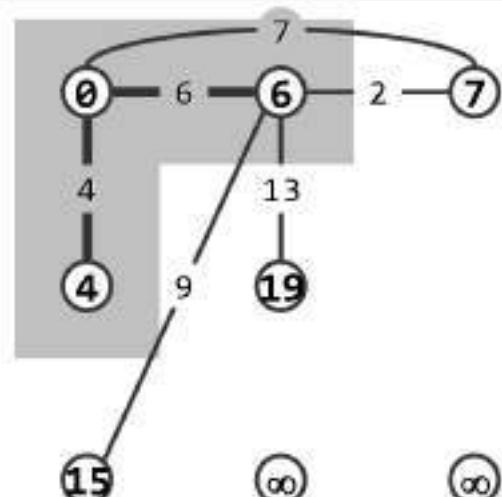
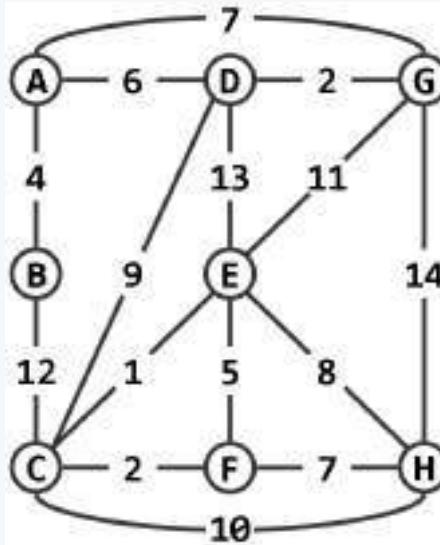
邓俊辉

deng@tsinghua.edu.cn

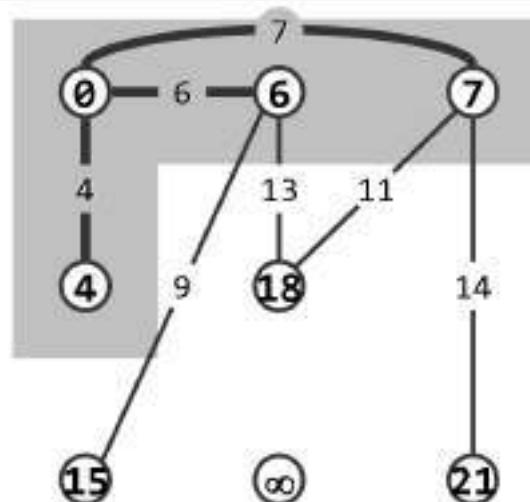
1/3



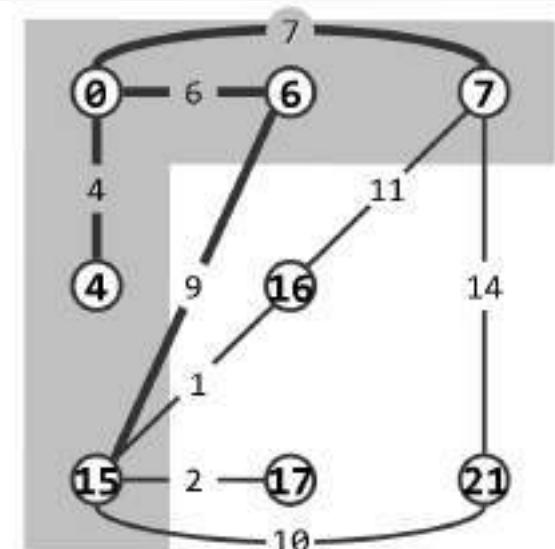
2/3



(d)

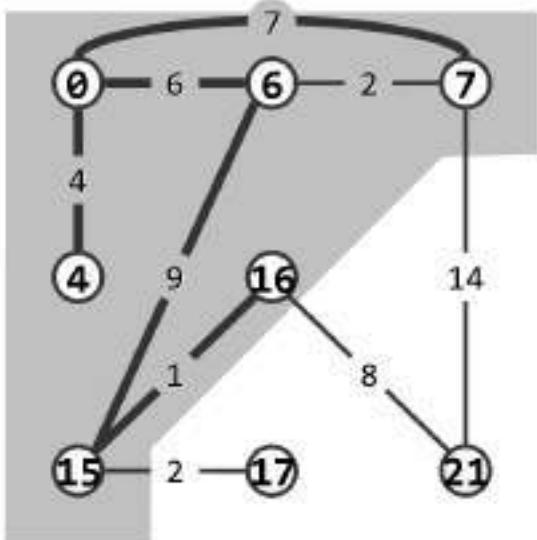
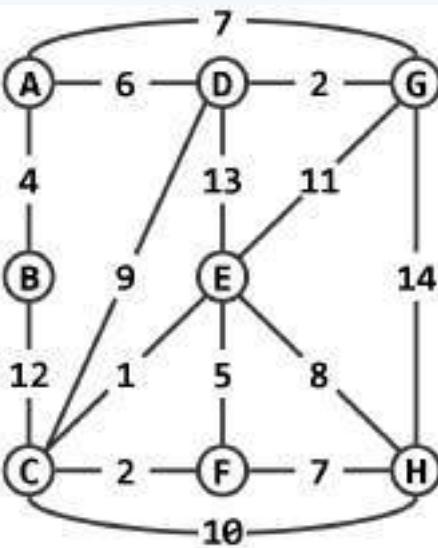


(e)

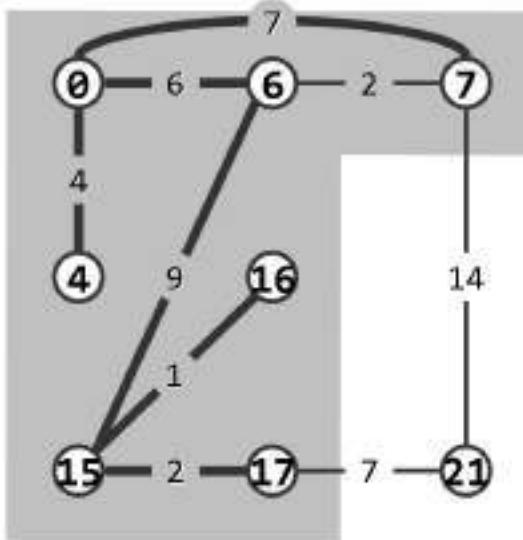


(f)

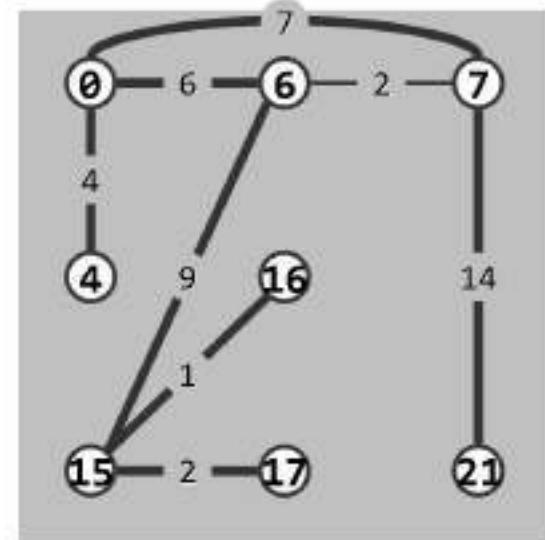
3/3



(g)



(h)



(i)

## 6. 图

Dijkstra算法  
实现

邓俊辉

deng@tsinghua.edu.cn

❖ 对于  $v_k$  外各顶点  $v$ ，令：

$\text{priority}(v) = v \rightarrow s$  的距离

于是套用优先级遍历算法框架...

❖ 为将  $T_k$  扩充至  $T_{k+1}$ ，可以

- 选出优先级最高的跨边  $e_k$  及其对应顶点  $u_k$ ，并将其加入  $T_k$
- 随后，更新  $v_{k+1}$  外所有顶点的优先级（数）

❖ 注意：优先级数随后可能改变（降低）的顶点，必与 $u_k$ 邻接

❖ 因此，只需

- 枚举 $u_k$ 的每一邻接顶点 $v$ ，并取
- $\text{priority}(v) = \min(\text{priority}(v), \text{priority}(u_k) + |u_k, v|)$

❖ 以上同样完全符合PFS的框架，唯一要做的工作无非是

按照模式

编写一个优先级（数）更新器...

## 实现

```

❖ g->pfs( 0, DijkstraPU<char, int>() ); //从顶点0出发，启动Dijkstra算法

❖ template <typename Tv, typename Te> //顶点类型、边类型

struct DijkstraPU { //Dijkstra算法的顶点优先级更新器

    virtual void operator()( Graph<Tv, Te>* g, int uk, int v ) { // 对uk的每个
        if ( UNDISCOVERED != g->status(v) ) return; //尚未被发现的邻居v，按

        if ( g->priority(v) > g->priority(uk) + g->weight(uk, v) ) { //Dijkstra
            g->priority(v) = g->priority(uk) + g->weight(uk, v); //策略
            g->parent(v) = uk; //做松弛
        }
    }
};

}

```

## 6. 图

双连通分量

关节点

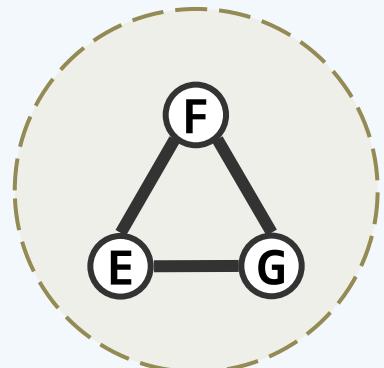
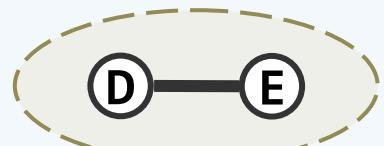
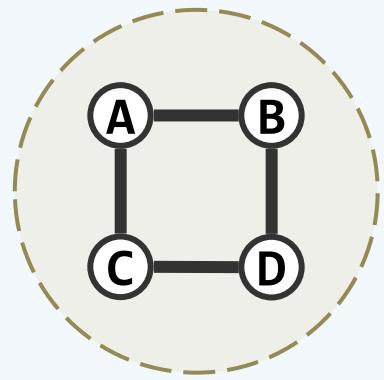
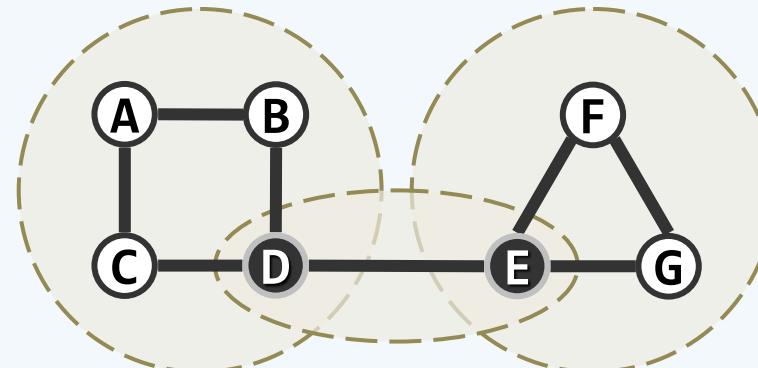
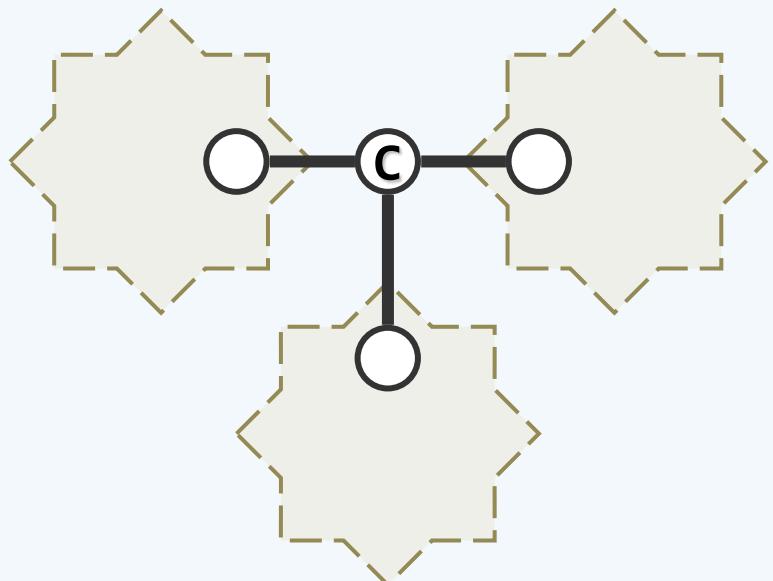
邓俊辉

伤其十指，不如断其一指

deng@tsinghua.edu.cn

## 关节点 + 双连通分量

- ❖ 无向图的关节点： //articulation point, cut-vertex  
其删除之后，原图的连通分量增多 //connected components
- ❖ 无关节点的图，称作双（重）连通图 //bi-connectivity
- ❖ 极大的双连通子图，称作双连通分量 //Bi-Connected Components



## 6. 图

双连通分量

判定准则

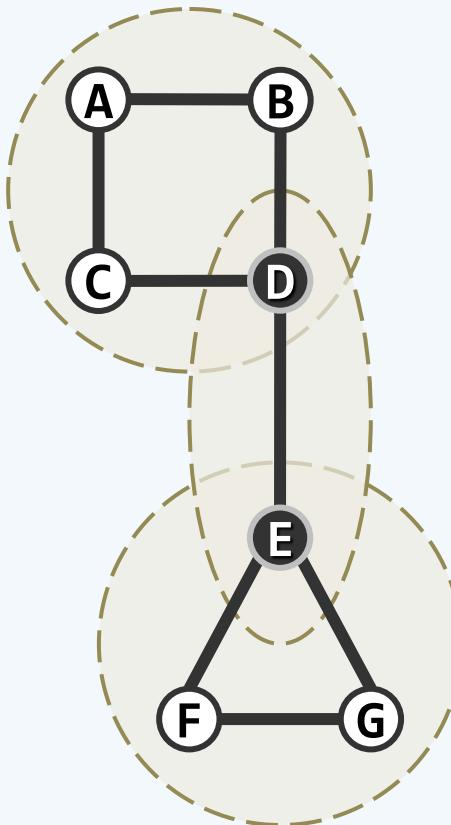
邓俊辉

帝高阳之苗裔兮，朕皇考曰伯庸

deng@tsinghua.edu.cn

## Brute-Force

- ❖ 给定无向图，如何确定各BCC？
- ❖ 先考察简单的版本：如何确定关节点？
- ❖ 蛮力： 对每一顶点  $v$ ，通过遍历检查  $G \setminus \{v\}$  是否连通
- ❖ 共需  $\mathcal{O}(n * (n + e))$  时间，太慢！  
而且，即便找出关节点，各BCC仍需确定
- ❖ 改进： 从任一顶点出发，构造DFS树  
根据DFS留下的标记，甄别是否关节点
- ❖ 比如，叶节点 绝不可能是关节点 //为什么？



## 根顶点

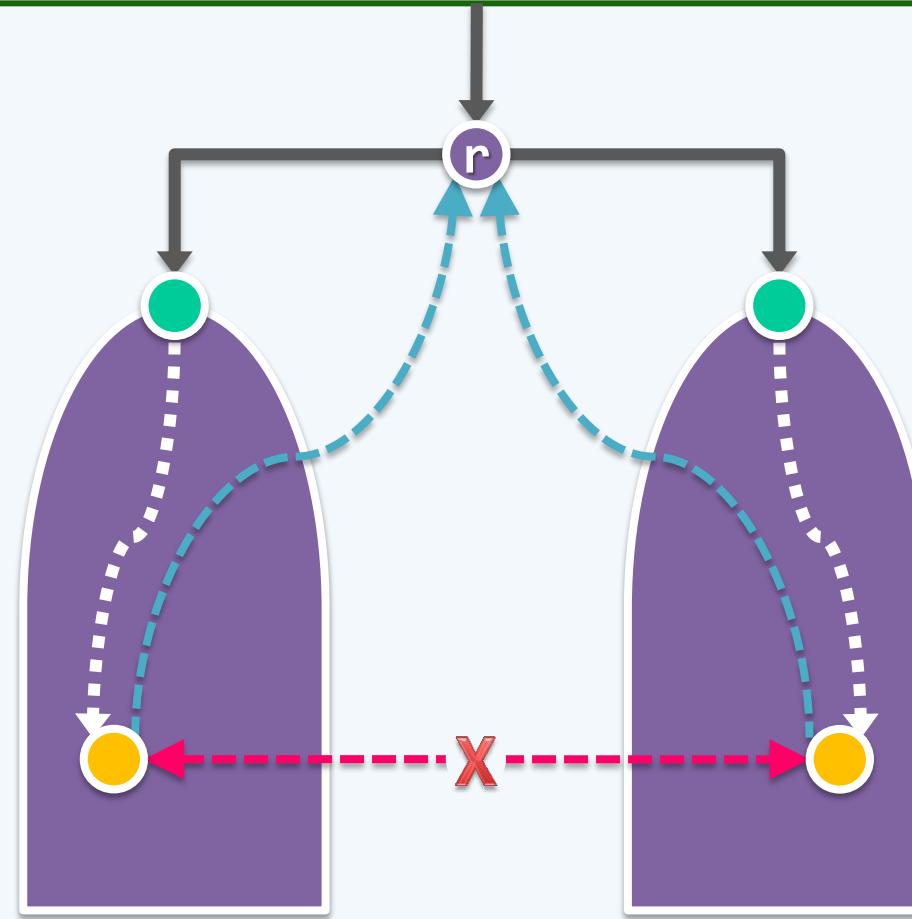
❖ 根顶点是关节点 iff

树根至少有2棵子树

❖ 在DFS树中，不难检查

从根顶点R发出的树边数目

❖ 那么，一般的内部顶点呢？



## 内顶点

❖ 内顶点  $v$  是关节点

iff

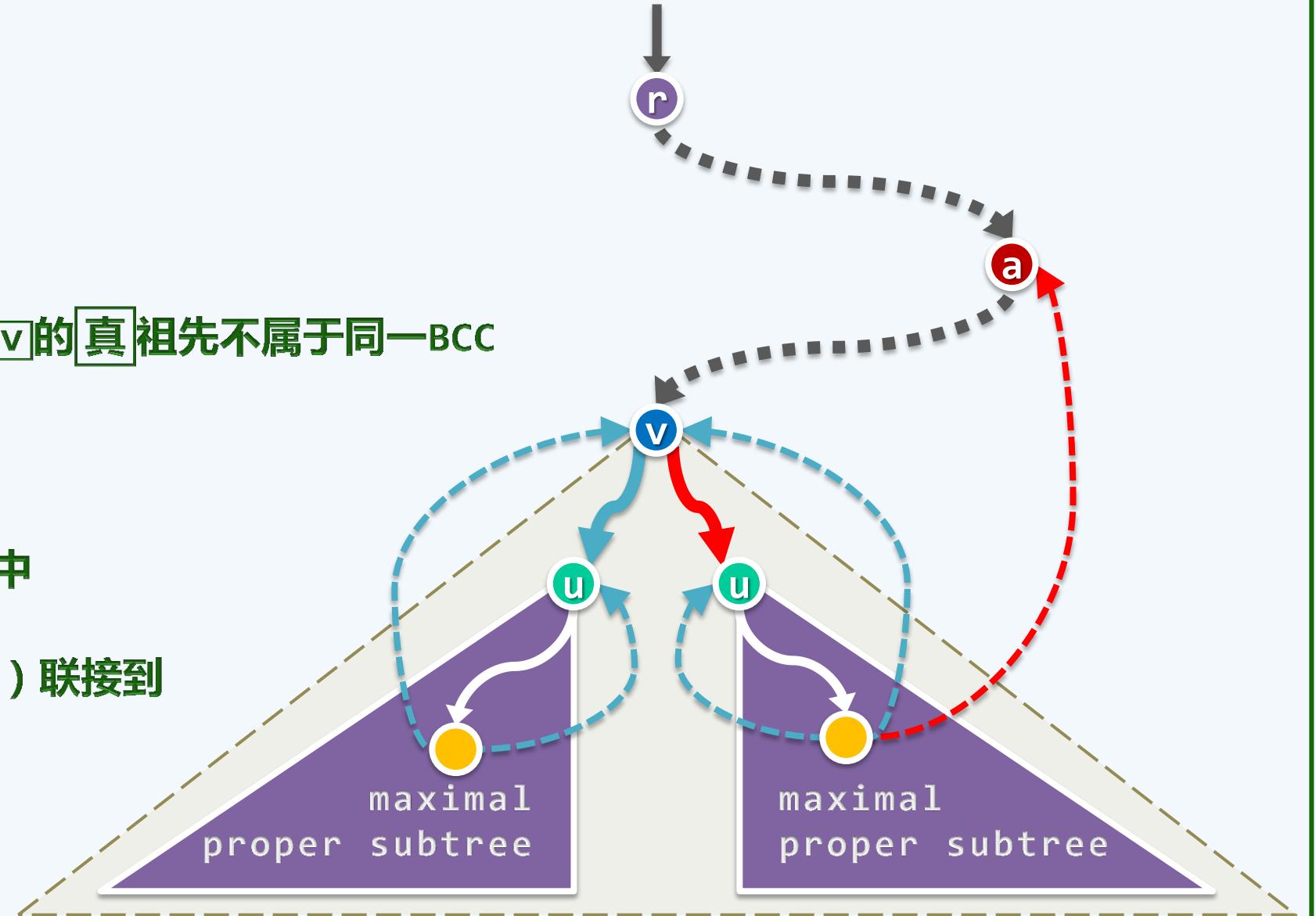
$v$  的某棵极大真子树与  $v$  的真祖先不属于同一BCC

iff

在  $v$  的某棵极大真子树中

没有顶点（经后向边）联接到

$v$  的真祖先



# 祖先

❖ 如何记录可联接的最高祖先？

比对 HCA ...

❖ Highest Connected Ancestor

❖  $hca(v) =$

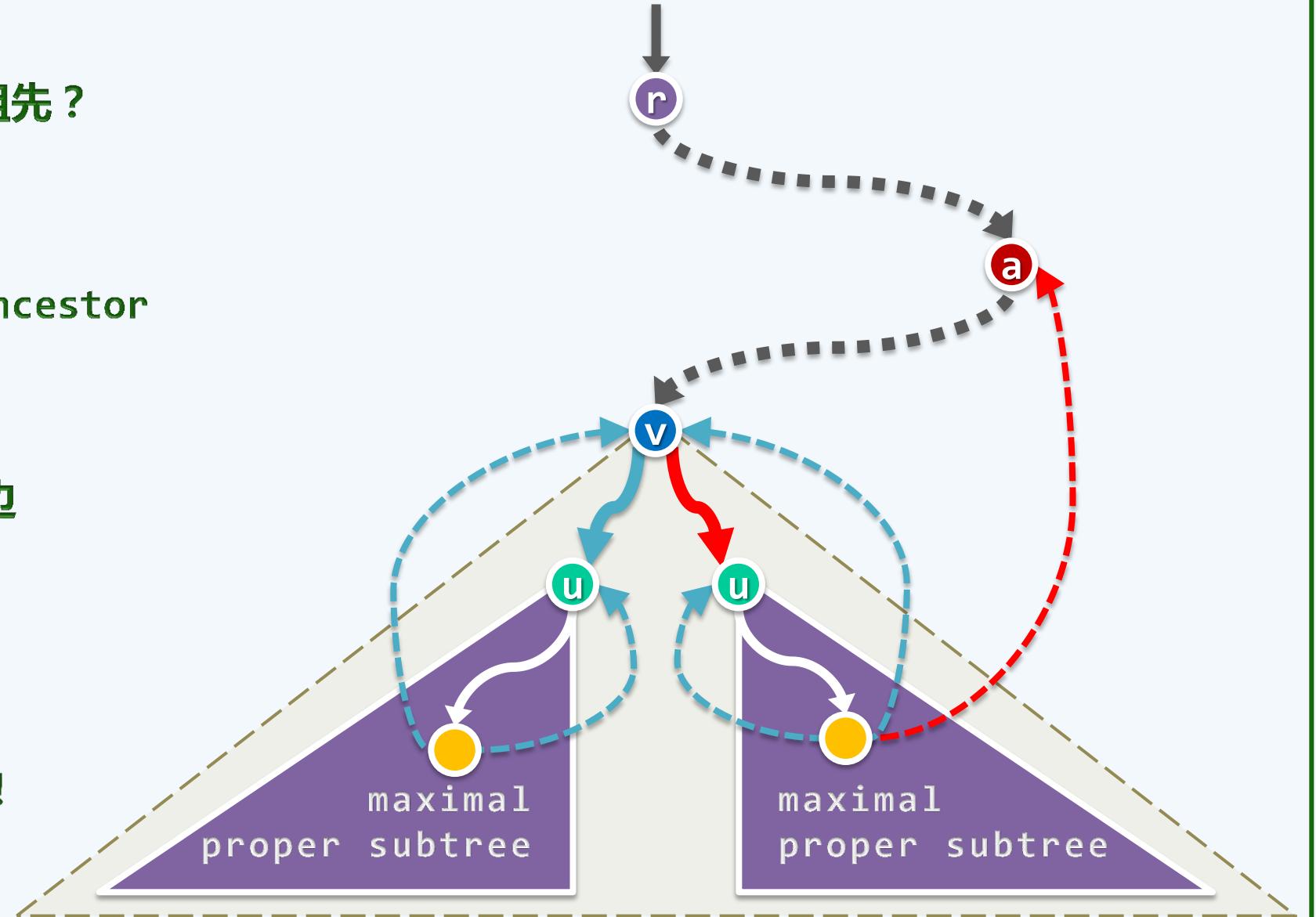
$\text{subtree}(v)$  经后向边

能抵达的最高祖先

❖ 如何判断哪个祖先更高？

比对 dTime，越小越高！

$O(1)!!!$



## 6. 图

### 双连通分量 算法

邓俊辉

deng@tsinghua.edu.cn

## Graph::BCC()

```

❖ #define hca(x) ( fTime(x) ) //利用此处闲置的fTime[ ]充当hca[ ]

template <typename Tv, typename Te> //顶点类型、边类型
void Graph<Tv, Te>::BCC(int v, int& clock, Stack<int>& S) {
    hca(v) = dTime(v) = ++clock; status(v) = DISCOVERED; //发现v
    S.push(v); //顶点v入栈，以下枚举v的所有邻居u
    for(int u = firstNbr(v); -1 < u; u = nextNbr(v, u))
        switch (status(u))
            { /* ... 视u的状态分别处理 ... */ }
    status(v) = VISITED; //对v的访问结束
}
#define hca

```

```
switch ( status(u) )
```

❖ case **UNDISCOVERED**:

**parent(u) = v; type(v, u) = TREE;** //拓展树边

BCC(u, **clock**, **S**); //从u开始遍历，返回后

**if ( hca(u) < dTime(v) ) {** //若u经后向边指向v的真祖先

**hca(v) = min( hca(v), hca(u) );** //则v亦必如此

**} else {** //否则，以v为关节点，u以下即为一个BCC，且其中顶点此时正集中于栈S的顶部

**while ( v != S.pop() );** //故可依次弹出这些顶点，或根据实际需求另做处理

**S.push(v);** //最后一个顶点（关节点）重新入栈

**}** //尽管同一顶点可作为（关节点）重复入栈，但**分摊**不足一次

**break;**

```
switch ( status(u) )
```

❖ case DISCOVERED:

```
    type(v, u) = BACKWARD;
```

```
    if ( u != parent(v) )
```

```
        hca(v) = min( hca(v), dTime(u) ); //更新hca[v]，越小越高
```

```
    break;
```

❖ default: //VISITED (digraphs only)

```
    type(v, u) = dTime(v) < dTime(u) ? FORWARD : CROSS;
```

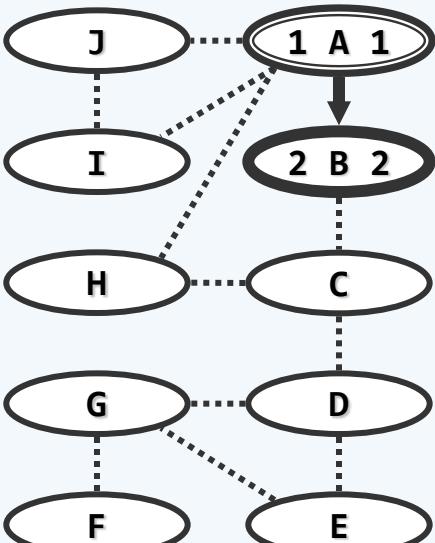
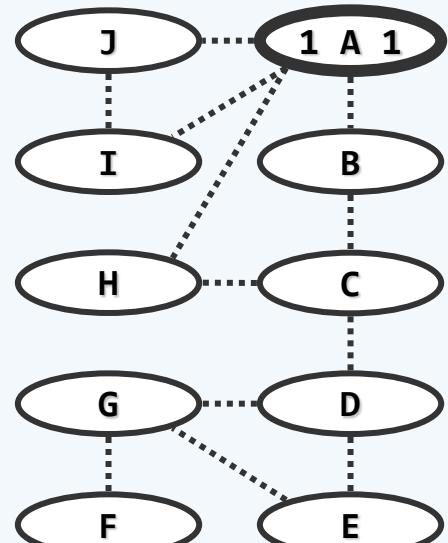
```
    break;
```

## 6. 图

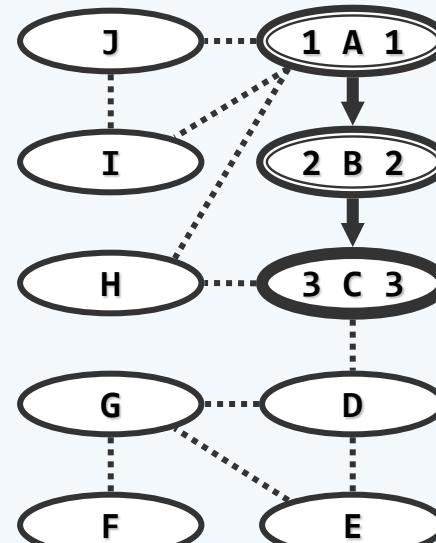
双连通分量  
实例

邓俊辉

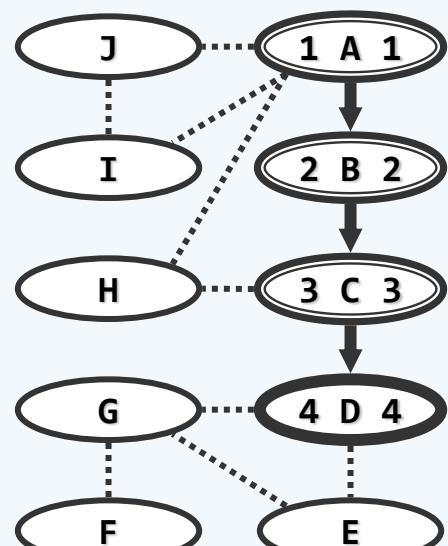
deng@tsinghua.edu.cn



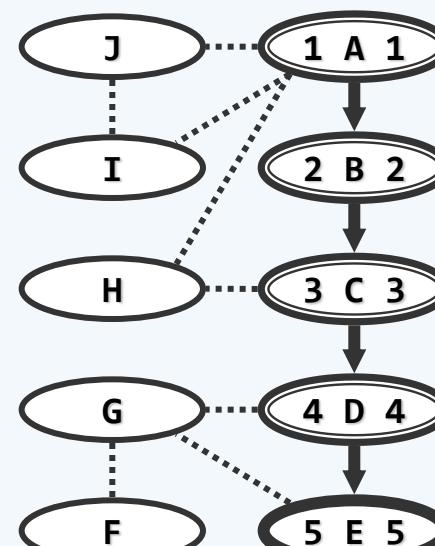
A-B



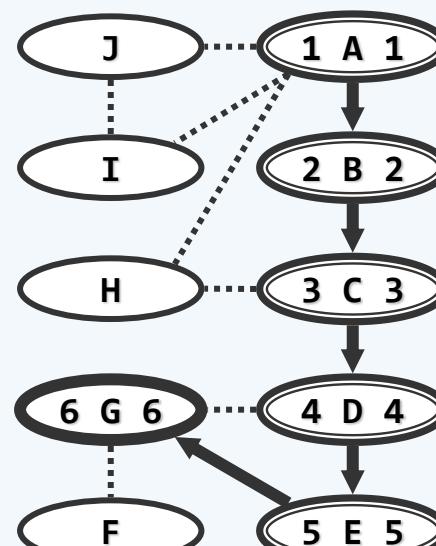
B-C  
A-B



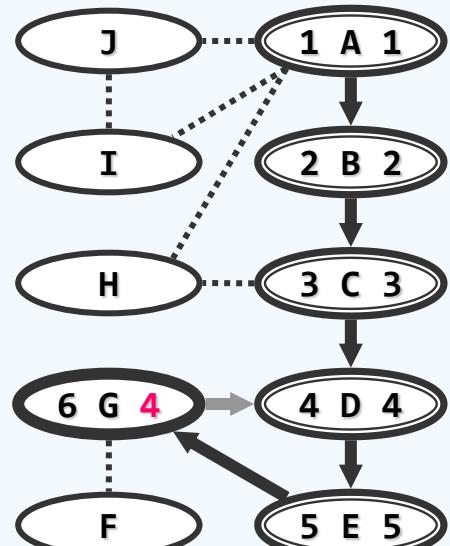
C-D  
B-C  
A-B



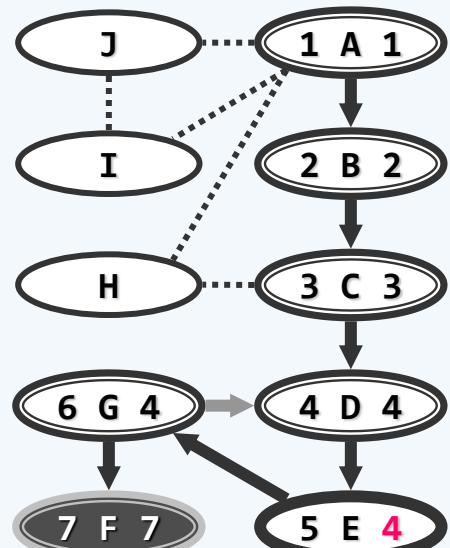
D-E  
C-D  
B-C  
A-B



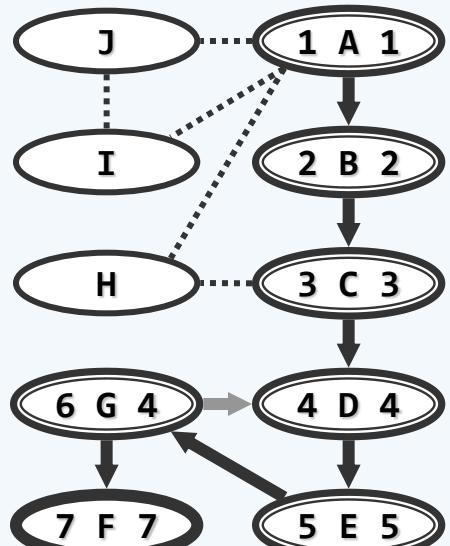
E-G  
D-E  
C-D  
B-C  
A-B



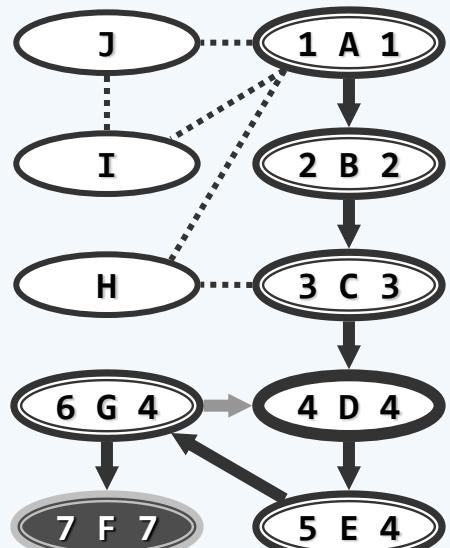
**G-D**  
E-G  
D-E  
C-D  
B-C  
A-B



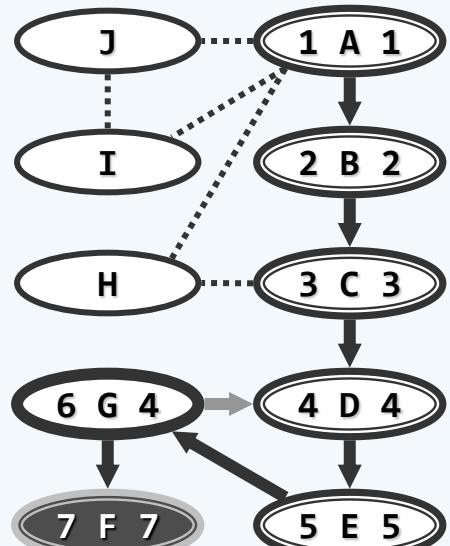
**G-F**  
**G-D**  
E-G  
D-E  
C-D  
B-C  
A-B



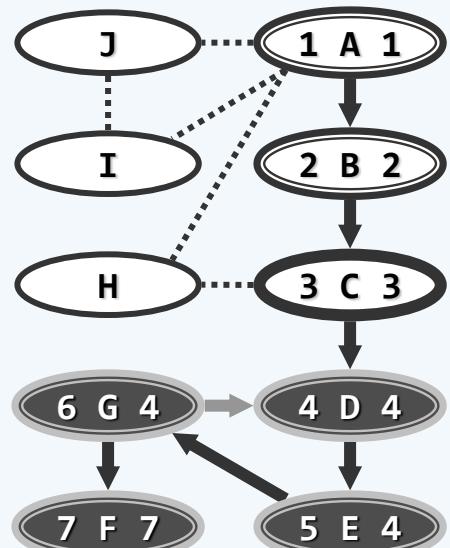
**G-F**  
**G-D**  
E-G  
D-E  
C-D  
B-C  
A-B



**G-F**  
**G-D**  
E-G  
D-E  
C-D  
B-C  
A-B

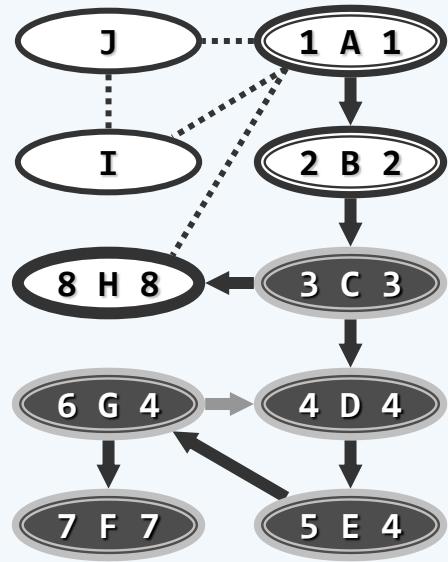


**G-F**  
**G-D**  
E-G  
D-E  
C-D  
B-C  
A-B

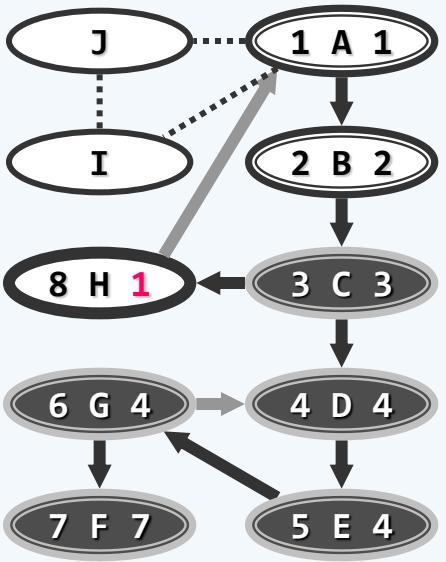


**G-F**  
**G-D**  
E-G  
D-E

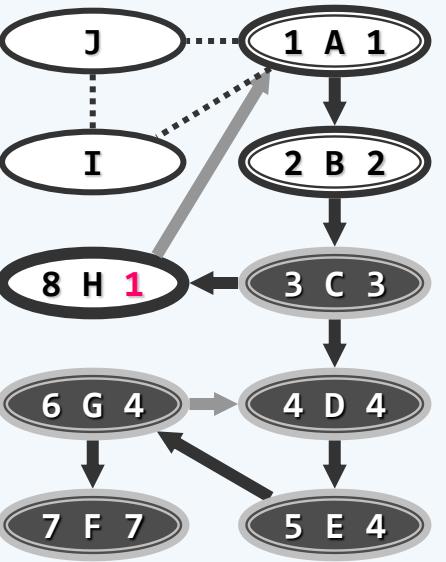
C-D  
B-C  
A-B



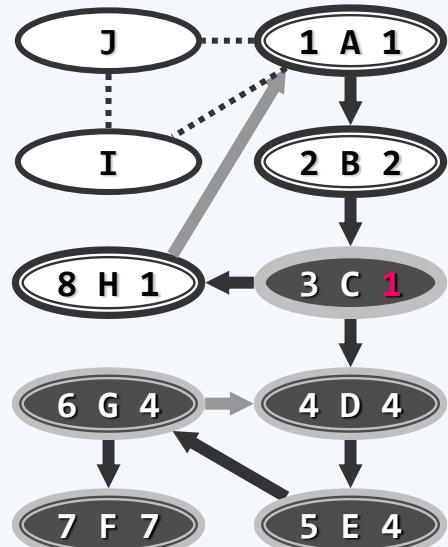
G-F  
G-D  
E-G  
D-E  
  
C-D  
  
C-H  
B-C  
A-B



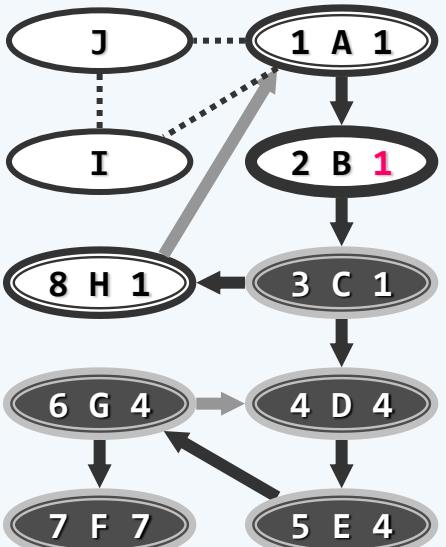
G-F  
G-D  
E-G  
D-E  
  
C-D  
  
H-A  
C-H  
B-C  
A-B



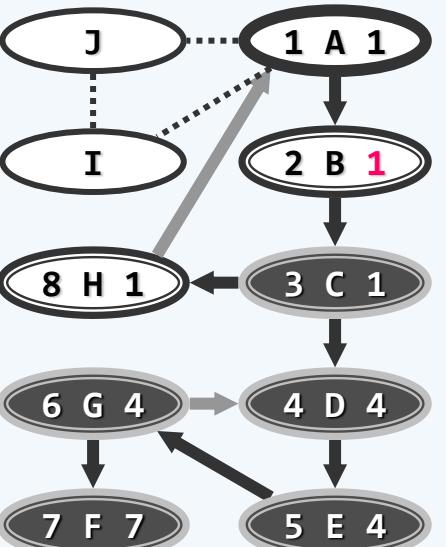
G-F  
G-D  
E-G  
D-E  
  
C-D  
  
H-A  
C-H  
B-C  
A-B



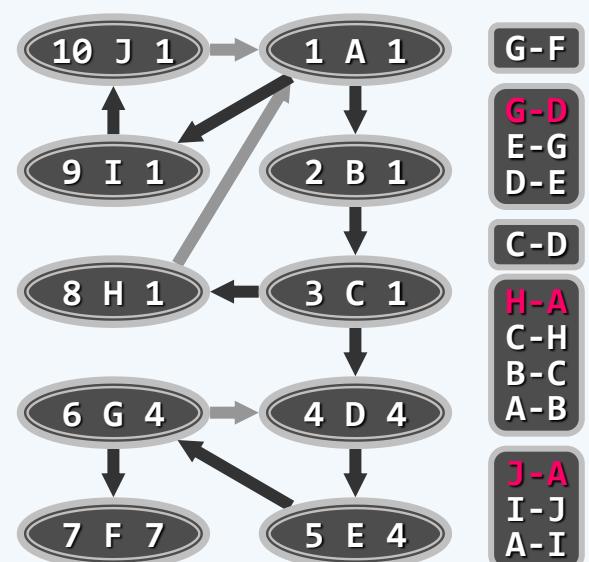
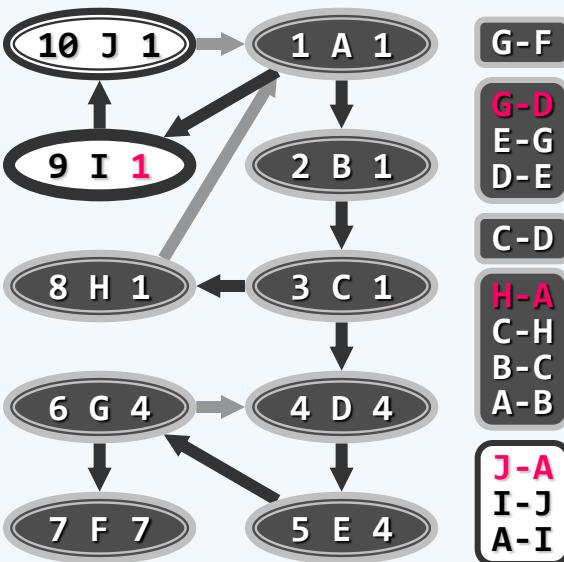
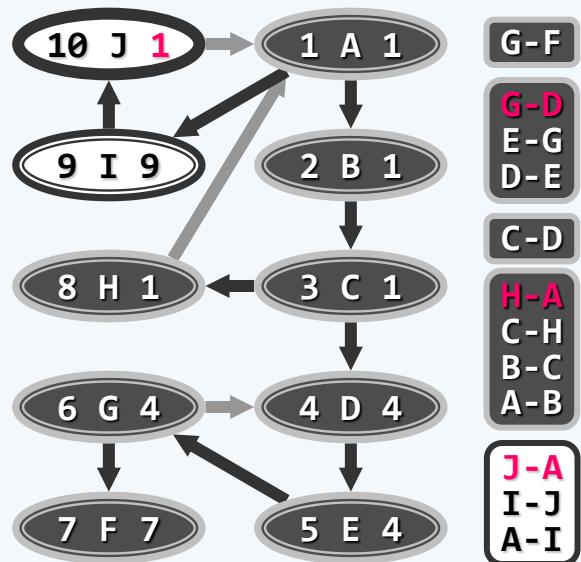
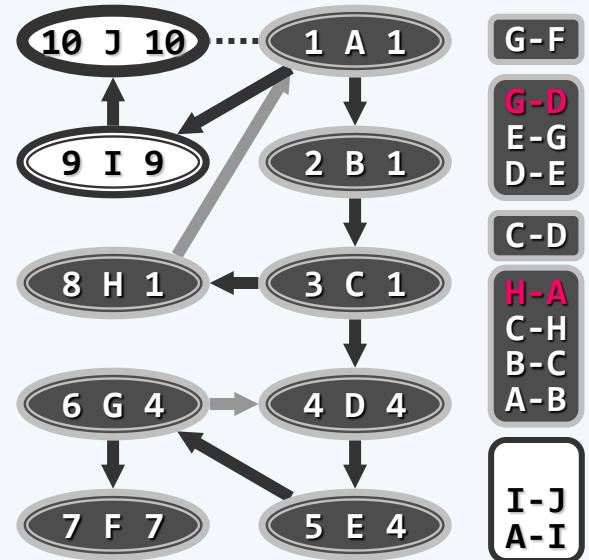
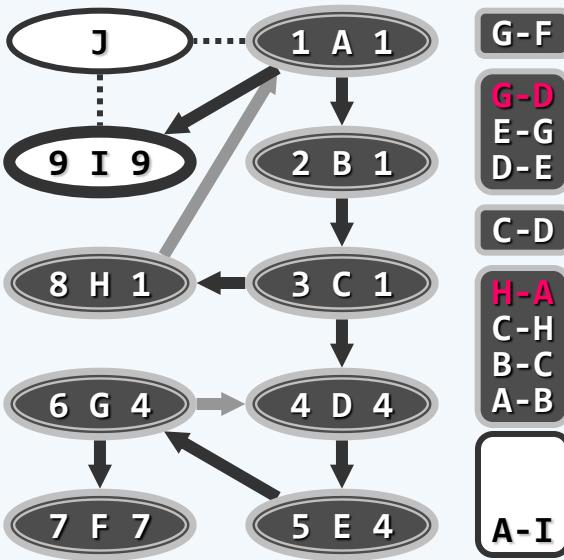
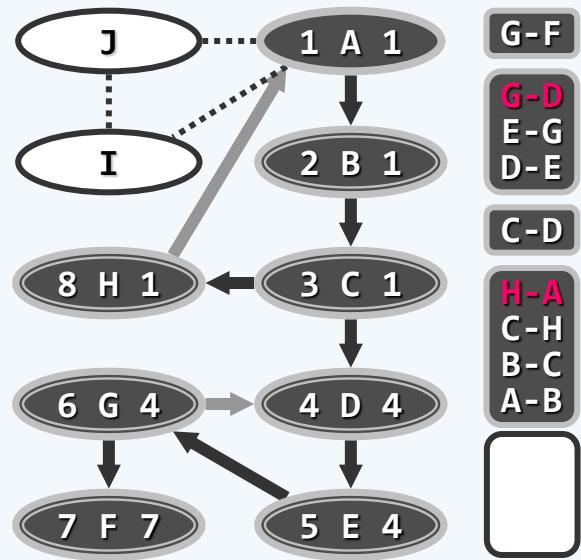
G-F  
G-D  
E-G  
D-E  
  
C-D  
  
H-A  
C-H  
B-C  
A-B



G-F  
G-D  
E-G  
D-E  
  
C-D  
  
H-A  
C-H  
B-C  
A-B



G-F  
G-D  
E-G  
D-E  
  
C-D  
  
H-A  
C-H  
B-C  
A-B



6. 图

双连通分量  
复杂度

邓俊辉

deng@tsinghua.edu.cn

❖ 运行时间与常规的DFS相同，也是 $\mathcal{O}(n + e)$

自行验证：栈操作的复杂度也不过如此

❖ 除原图本身，还需一个容量为 $\mathcal{O}(e)$ 的栈存放已访问的边

为支持递归，另需一个容量为 $\mathcal{O}(n)$ 的运行栈

❖ 课后：该算法是否也适用于非连通图？

在有向图中如何实现对应的计算？

❖ Strongly-connected component

Kosaraju's algorithm

Tarjan's algorithm

## 6. 图

### Kruskal算法 算法

煮豆持作羹，漉菽以为汁  
萁在釜下燃，豆在釜中泣  
本是同根生，相煎何太急

Junhui DENG

deng@tsinghua.edu.cn

## 贪心策略

❖ 回顾Prim算法：

代价**最小**的边，迟早会被采用

**次小**的边，亦是如此

**再次小**的，则未必 //回路！

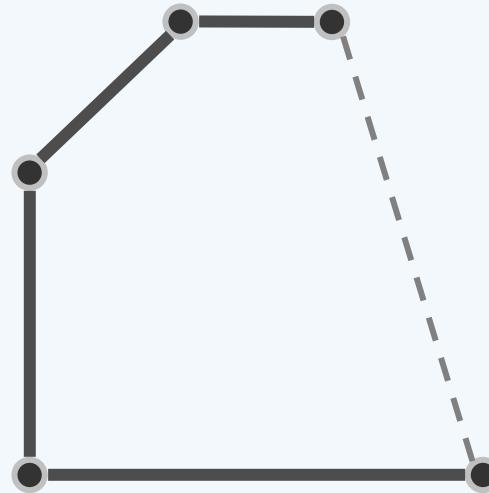
❖ Kruskal：贪心原则

根据代价，从小到大依次尝试各边

只要“安全”，就加入该边

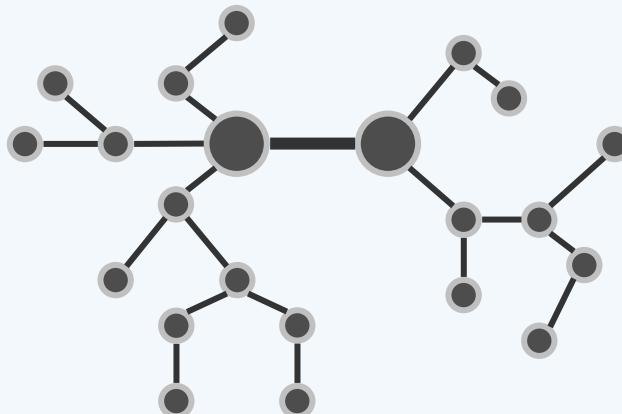
❖ 但是，每步局部最优 = 全局最优？

❖ 确实，Kruskal很幸运...



## 算法框架

- ❖ 维护 $N$ 的一个森林： $F = (V; E') \subseteq N = (V; E)$
- ❖ 初始化： $F = (V; \emptyset)$ 包含 $n$ 棵树（各含 1 个顶点）和0条边  
将所有边按照代价排序
- ❖ 迭代：找到当前最廉价的边 $e$   
若 $e$ 的顶点来自 $F$ 中不同的树，则  
令 $E' = E' \cup \{e\}$ ，然后  
将 $e$ 联接的2棵树合二为一  
*// 注意：引入 $e$ 不致造成回路*
- ❖ 重复上述过程，直到 $F$ 成为1棵树
- ❖ 整个过程共迭代 $n - 1$ 次，选出 $n - 1$ 条边



## 正确性

❖ 定理：Kruskal引入的每条边都属于某棵MST

❖ 设边 $e = (u, v)$ 的引入导致树T和S的合并

❖ 若：将 $(T; V \setminus T)$ 视作原网络N的割

则：e当属该割的一条跨边

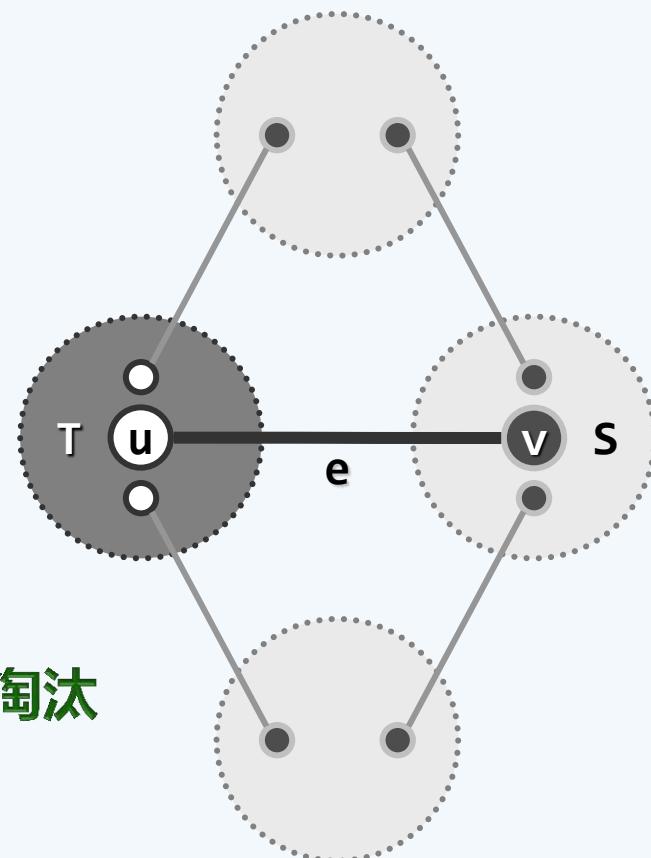
❖ 在确定应引入e之前

该割的所有跨边都经Kruskal考察，且只可能因不短于e而被淘汰

❖ 故：e当属该割的一条极短跨边

❖ 与Prim同理，以上论述也不充分，为严格起见，仍需归纳证明：

Kruskal 算法过程中不断生长的森林，总是某棵MST的子图



## 6. 图

Following the leader,  
the leader, the leader  
  
We're following the leader  
wherever he may go  
  
We won't be home till morning,  
till morning, till morning  
  
We won't be home till morning  
because he told us so

Kruskal算法

实现

Junhui DENG

deng@tsinghua.edu.cn

## 排序

❖ 需做全排序吗？

若做，将耗时  $\mathcal{O}(e \log e) = \mathcal{O}(n^2 \log n)$  // 稠密图可达上界

❖ 实际上，大多数情况下只需考虑前  $\Theta(n)$  条边

❖ 将所有边组织成优先队列 // 比如，以堆实现

建堆， $\mathcal{O}(e)$

删除并复原，

$\mathcal{O}(\log e)$

共迭代  $\mathcal{O}(e)$  次 // 实际中往往远小于  $e$ ，尤其是对于稠密图

总共 =  $\mathcal{O}(e) + \mathcal{O}(\log e) \times \mathcal{O}(e) = \mathcal{O}(elogn)$

## 回路检测

❖ 如何高效地检测回路，并且合并树？

❖ 首领节点 **leader node**

每棵树各选出一个首领

各顶点设指针**parent**

沿**parent**指针可找到对应的首领

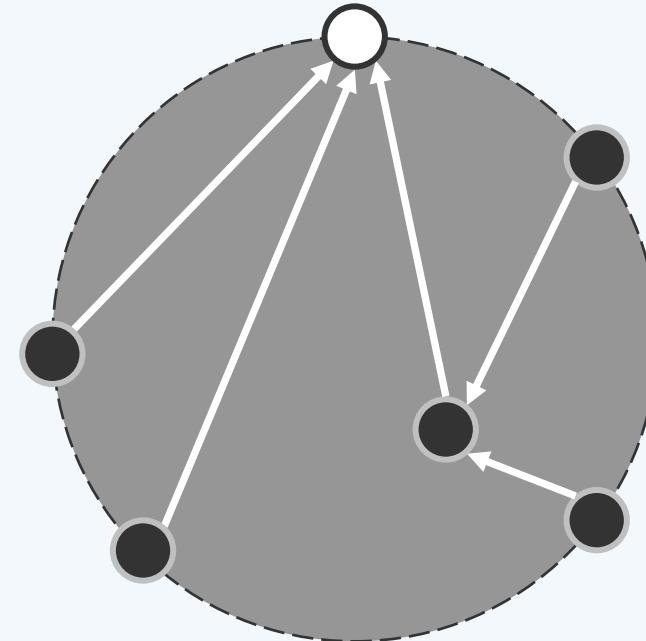
`leader.parent = NO_PARENT`

❖ 尝试引入新边 $e = (u, v)$ 时

由**parent**指针，找到并比对  $\text{leader}(u)$  和  $\text{leader}(v)$

$e$ 的引入造成回路      iff       $\text{leader}(u) = \text{leader}(v)$

❖ 如经检查确定可以合并，又该如何高效地合并 $u$ 和 $v$ 所属的树？



## 树合并

❖ 合并树 $T(v)$ 和 $T(u)$ 后，令 $\text{leader}(u).\text{parent} = \text{leader}(v)$

❖ 注意：合并后，树的深度 =  $O(n)$

总会不幸达到上界吗？很有可能！于是 ...

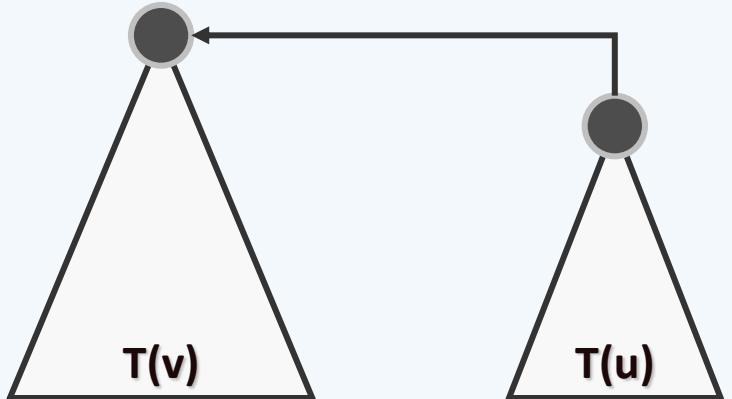
❖ 对 $\text{leader}$ 的 $O(n)$ 次查询，共需 $O(n^2)$ 时间

❖ 改进：  
 $\text{leader.parent} = -(\#\text{nodes})$

```
if  (leader(u).parent > leader(v).parent)
    leader(u).parent = leader(v);
else leader(v).parent = leader(u);
```

❖ 使用改进后算法，树合并后深度 =  $O(\log n)$  //YAN, p.142

❖ 因此，查询 $\text{leader}$ 的总复杂度 =  $O(n \log n)$



## 6. 图

Kruskal算法  
并查集

Junhui DENG

deng@tsinghua.edu.cn

## Union-Find

### ❖ Union-Find问题

给定一组互不相交的等价类  
由各自的一个成员作为代表

### ❖ Singleton

初始时各包含一个元素

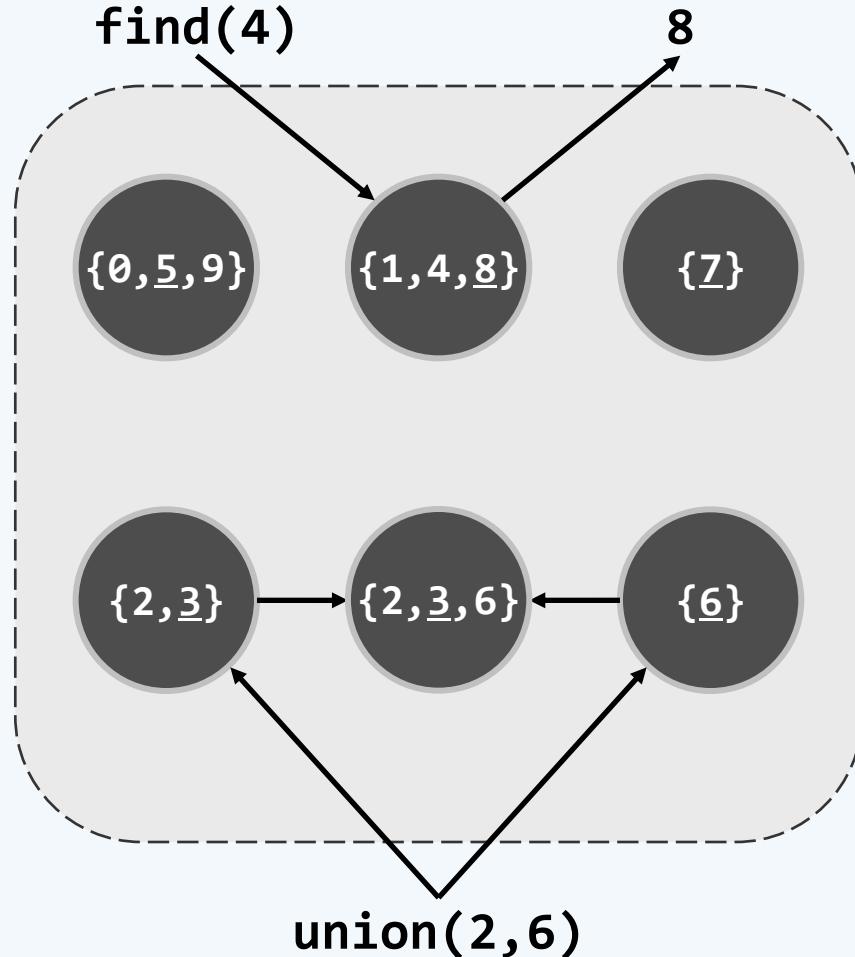
### ❖ Find(x)

找到元素x所属等价类

### ❖ Union(x, y)

合并x和y所属等价类

### ❖ Kruskal = Union-Find



## Union-Find

❖ [Tarjan-83] :  $\mathcal{O}(\alpha(n))$  amortized time per Union/Find

❖  $\alpha(n)$ : inverse Ackermann function

$$\log^* n = \# \text{logs s.t. } \log(\log(\dots(\log n)\dots)) < 2$$

$$\log^{**} n = \# \log^* s \text{ s.t. } \log^*(\log^*(\dots(\log^* n)\dots)) < 2$$

... ... ...

$$\log^{**\dots*n} = \dots$$

... ... ...

❖  $\alpha(n) = \# \text{stars s.t. } \log^{**\dots*n} < 3$

❖  $\alpha(\text{目前可观察到宇宙范围内的粒子总数}) = \alpha(10^{83}) < 4$

## 复杂度

- ❖ 初始化森林， $\mathcal{O}(n)$
- ❖ 建立PQ， $\mathcal{O}(e)$
- ❖ 迭代共 $\mathcal{O}(e)$ 次： 取出队首并调整PQ， $\mathcal{O}(\log e)$ 
  - 回路检测 + 边输出， $\mathcal{O}(\log n)$
  - 树合并， $\mathcal{O}(1)$
- ❖ 总共 =  $\mathcal{O}(e \log e) = \mathcal{O}(e \log n)$
- ❖ 对稀疏图，迭代次数 =  $e = \Theta(n)$ ，共 $\mathcal{O}(n \log n) \ll \mathcal{O}(n^2)$   
能更快吗？
- ❖ [Fredman & Tarjan-87]

采用Fibonaccian Heap，对稀疏图可做到  $\mathcal{O}(e \cdot \log \log n)$

**更多算法**

❖ Boruvka (1920s)

每个**点**与自己的**最近邻居**相连（构造出一个森林）

每棵**树**与自己的**最近邻居**相连

迭代上述过程，直到…

❖ Vyssotsky (1960s)

每次增加一条边

如果出现回路，将回路上**最长**的边删去

## 更新结果

❖ Gabow, Galil, Spencer, & Tarjan :  $O(m \cdot \log(\beta(m, n)))$

Efficient algorithms for finding minimum spanning trees  
in undirected and directed graphs

Combinatorica, vol. 6, 1986, pp. 109–122

$\beta(m, n) = \text{smallest } i \text{ s.t. } \log(\log(\log(\dots \log(n)\dots))) < m/n$

where the logs are nested  $i$  times

❖ Fredman & Willard : 权值均为不大的整数时，最坏情况仅需线性时间

Trans-dichotomous algorithms for  
minimum spanning trees and shortest paths

## 更新结果

- ❖ YAO (1995):  $\mathcal{O}(e \cdot \log\log n)$
- ❖ Karger, Klein, & Tarjan : 随机算法，期望的线性时间  
A randomized linear-time algorithm  
to find minimum spanning trees  
J. ACM, vol. 42, 1995, pp.321-328
- ❖ CHAZELLE (2000): MST与union-find问题的复杂度相同

## 相关话题

❖ Proximity Graphs // $\Omega(n \log n)$

Euclidean MST

Delaunay Triangulation

Gabriel Graph

Relative Neighborhood Graph

❖ Steiner MST //NP-hard

Approximation

## 6. 图

Floyd-Warshall算法

邓俊辉

deng@tsinghua.edu.cn

## 从Dijkstra到Floyd-Warshall

- ❖ 给定图G，计算其中所有点对之间的最短距离
- ❖ 应用：搜索图G的中心点 center vertex
  - s中心的半径  $\text{radius}(G, s) = \text{所有顶点到}s\text{的最大距离}$
  - 中心点 = 半径最小的顶点s
- ❖ 直觉：依次将各顶点作为源点，调用Dijkstra算法
  - 时间 =  $n \times \mathcal{O}(n^2) = \mathcal{O}(n^3)$  —— 可否更快？
- ❖ 思路：图矩阵 ~ 最短路径矩阵
- ❖ 效率： $\mathcal{O}(n^3)$ ，与执行n次Dijkstra相同 —— 既如此，为何还要用FW？
- ❖ 优点：形式简单、算法紧凑、便于实现
  - 允许负权边（尽管仍不能有负权环路）

## 问题特点

❖  $u$ 和 $v$ 之间的最短路径可能是

- 0) 不存在通路，或者
- 1) 直接连接，或者
- 2) 最短路径( $u$ ,  $\boxed{x}$ ) + 最短路径( $\boxed{x}$ ,  $v$ )

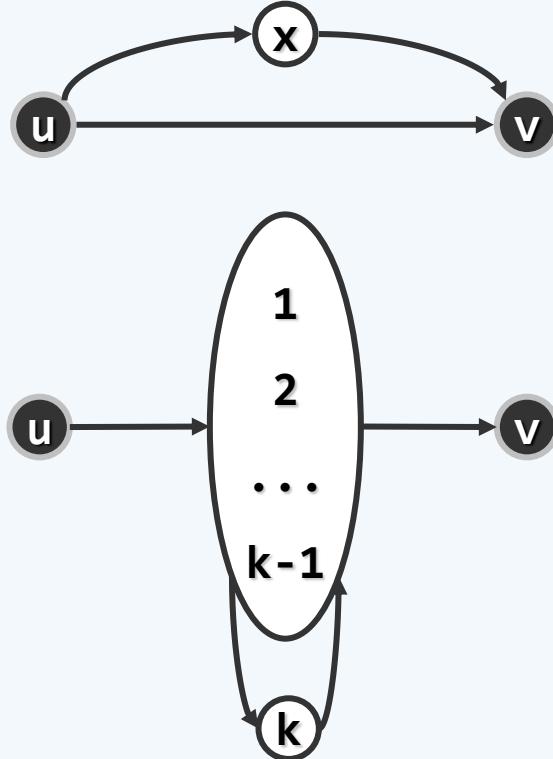
❖ 将所有顶点随意编号： $1, 2, \dots, n$

❖ 定义： $d^k(u, v)$

= 中途只经过前 $k$ 个顶点、联接 $u$ 和 $v$ 的最短路径长度

=  $w(u, v)$  (if  $k = 0$ )

=  $\min(d^{k-1}(u, v), d^{k-1}(u, \boxed{k}) + d^{k-1}(\boxed{k}, v))$  (if  $k \geq 1$ )



## 蛮力递归

```
weight dist( node * u, node * v, int k ) { //蛮力递归  
    if ( k < 1 ) return w( u, v ); //递归基：中途不经过任何点  
  
    minDist = dist( u, v, k - 1 ); //递归：中途可经过前 k-1 个点  
  
    for each node x ∈ { u, v } { //枚举其余各点，分别作为第 k 个点  
  
        u2x2vDist = dist( u, x, k - 1 ) + dist( x, v, k - 1 ); //递归  
  
        minDist = min( minDist, u2x2vDist ); //优化  
    }  
  
    return minDist;  
}
```

## 复杂度

❖  $T(n, 0) = 1$

$$T(n, k) = (n-2) \times 2 \times T(k - 1) = 2^k \times (n-2)^k$$

$$T(n, n-2) = 2^{n-2} \times (n-2)^{n-2}$$

$= O(n^n)$  //这还仅仅只是一对节点所需的时间

❖ 注意：蛮力算法中，存在大量的**重复**递归调用

❖ 能否**避免**这些重复计算？如何避免？你应该还记得...

❖ 动态规划 **dynamic programming**

维护一张表，记录需要**反复计算**的数值 //如此，只需计算一次

## 动态规划

```
//Initialization
for ( int u = 0; u < n; u++ )
for ( int v = 0; v < n; v++ )
{ dist[u][v] = w[u][v]; midV[u][v] = -1; }

//Iteration
for ( int k = 0; k < n; k++ )
for ( int u = 0; u < n; u++ )
for ( int v = 0; v < n; v++ )
if ( dist[u][v] > dist[u][k] + dist[k][v] )
{ dist[u][v] = dist[u][k] + dist[k][v]; midV[u][v] = k; }
```

