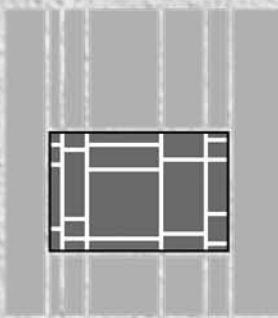
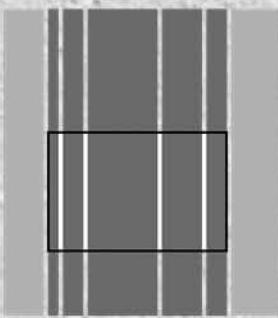
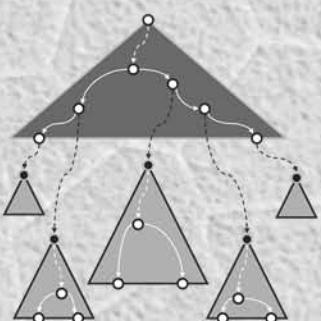


DATA STRUCTURES & ALGORITHMS



Tsinghua University
Fall 2012

简要目录

1	00.课程大纲	195	(x3) Python 列表
8	01.绪论	200	04.栈与队列
8	(a) 计算	200	(a) 栈接口与实现
15	(b) 大O记号	204	(b) 栈与递归
32	(c) 算法分析	210	(c1) 栈应用：进制转换
45	(d) 迭代与递归	216	(c2) 栈应用：括号匹配
65	(x1) 局限	228	(c3) 栈应用：中缀表达式求值
75	(x2) 排序与下界	240	(c4) 栈应用：逆波兰表达式
86	02.向量	248	(d1) 试探回溯法：八皇后
86	(a) 接口与实现	259	(d2) 试探回溯法：迷宫寻径
95	(b) 可扩充向量	264	(e) 队列接口与实现
102	(c) 无序向量	268	(f) 队列应用
112	(d) 有序向量	272	(x) Steap+Queap
137	(e) 起泡排序	275	05.树
141	(f) 归并排序	275	(a) 概述
148	03.列表	282	(b) 树的表示
148	(a) 接口与实现	288	(c) 二叉树概述
159	(b) 无序列表	294	(d) 二叉树实现
168	(c) 有序列表	303	(e1) 先序遍历
171	(d) 选择排序	314	(e2) 中序遍历
177	(e) 插入排序	324	(e3) 后序遍历
182	(f) 归并排序	335	(e4) 层次遍历
185	(x1) 游标实现	342	(f) PFC 编码树
188	(x2) Java 序列	349	(g) Huffman 树

362	06. 图		667	(c) 散列函数
362	(a) 概述		678	(d) 排解冲突
367	(b) 接口与实现		693	(e) 桶排序与基数排序
387	(c) 广度优先搜索		702	(x1) 跳转表
398	(d) 深度优先搜索		716	(x2) MD5
416	(e) 拓扑排序		723	10. 优先级队列
423	(f) 优先级搜索		723	(a) 基本实现
427	(g) Prim 算法		731	(b) 完全二叉堆
441	(h) Dijkstra 算法		748	(c) 锦标赛排序
457	(x1) 关节点与双连通分量		754	(d) 堆排序
470	(x2) Kruskal 算法		761	(x1) 左式堆
484	(x3) Floyd-Warshall 算法		773	(x2) 多叉堆
491	07. 二叉搜索树		780	11. 串
491	(a) 接口		780	(a) ADT
497	(b) 算法及实现		785	(b) 串匹配
507	(c) 平衡性与等价性		794	(c) KMP 算法
514	(d) AVL 树		813	(d) BM 算法
528	08. 高级搜索树		830	(e) Karp-Rabin 算法
528	(a) 伸展树		839	12. 排序
546	(b) B-树		839	(a) 快速排序
580	(x1) 红黑树		856	(b) 中位数
612	(x2) kd-树		865	(x1) 选取
633	(x3) 更多变种		879	(x2) 希尔排序
647	09. 词典			
647	(a) 循值访问			
657	(b) 散列			

0. 课程大纲

邓俊辉

deng@tsinghua.edu.cn



邓俊辉



尹霞



方宇剑



姜禹



李锐喆



刘雨辰

Data Structures & Algorithms (Fall 2012), Tsinghua University

2

选修，还是不选修

◆ 目标定位—是否需要选修数据结构

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

◆ 基本条件—可否选修数据结构

已修C/C++语言程序设计，有一定的编程基础

试做往届第0次和最后一次编程作业，进一步自我测试

◆ 更多条件—能否学好数据结构

对计算机科学与应用的兴趣 目标明确，心态平和
多思考、多动手、多讨论的习惯 投入落实，持之以恒

Data Structures & Algorithms (Fall 2012), Tsinghua University

3

考评及要求

◆ 考核环节（根据实际情况，期末可能做总体微调）

30240184：编程作业 × 3次 × 5题 + 笔试 × 2次 + 加分

00240074：编程作业 × 4次 × 4题 + 加分

◆ 编程作业 限时（多在周六晚）提交，建议同时贴出MDS码

三日内可在答疑区提交，成绩按每12小时15%累计折扣

白盒、黑盒测试结合，结果正确性兼顾代码编写质量

成绩公布之后，可于72小时内向答疑区提出申诉

◆ 笔试 基本知识点的充分掌握；基本方法和技巧的灵活运用

◆ 加分 参与交流 + 作业创意 + 独立思考 + 进步幅度

◆ 因材施教 前两周内提交个人简历和申请

Data Structures & Algorithms (Fall 2012), Tsinghua University

4

考评及要求

◆ 编程作业须独立完成，违者题分倒扣（记-100）

双方同论，不作区分；无论是否同班、同系、同届

◆ 如何判定（更准确地，判定什么）

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

◆ 什么不可以？ 源代码，或足以导致雷同的伪代码

◆ 什么可以交流？ 题意理解及解题思路

算法及数据结构的设计与选用方案

测试用例

◆ 尺度拿捏不准？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，至少不会倒扣

Data Structures & Algorithms (Fall 2012), Tsinghua University

5

教材与教辅



数据结构 (C++ 语言版)

2012年9月第二版

邓俊辉

7-302-29652-2



The Design & Analysis of Computer Algorithms

J. E. Hopcroft, et al

7-111-17775-4



数据结构 (C 语言版)

严蔚敏等

7-302-02368-5 (¥ 22)

7-302-14751-0 (¥ 30 附 CD)



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 (C++ 语言版)

殷人昆等

7-302-14811-1



数据结构基础 (C 语言版)

E. Horowitz 等著，朱仲涛译

7-302-18696-0

Data Structures & Algorithms (Fall 2012), Tsinghua University

6

学习资源

◆ 网络学堂

讲义：电子阅读版、打印装订版

代码：VS-2008解决方案DSACPP.sln，含50多个示例工程

可直接编译、运行

答疑：“课程讨论”（公开）或“答疑区”（私下）

◆ 编程作业在线评测

<http://thudsa.3322.org/oi/>

◆ 教学演示

<http://thudsa.3322.org/~deng/ds/demo>

同一结构和算法，具体实现不尽相同，须融会贯通

◆ 互联网资源...

7

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

- E. Dijkstra

邓俊辉

deng@tsinghua.edu.cn

1. 结论

(a) 计算

8

绳索计算机及其算法

◆问题 输入：任给直线l及其上一点A

输出：经过A做l的一条垂线

◆算法 (2000 B.C., 埃及人)

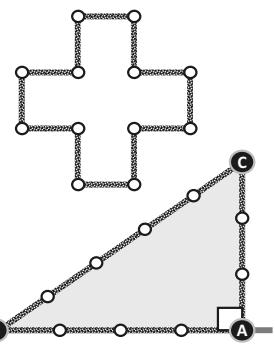
> 取12段等长的绳索，首尾联接成环

> 从A点起，将4段绳索沿l抻直并固定

> 沿另一方向找到第3段绳索的终点

> 移动该点，将剩余的8段绳索抻直

>



◆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？

9

算法

◆计算 = 信息处理

◆计算模型 = 计算机 = 信息处理工具 //尺规机、绳索机、图灵机...

◆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

... ...

Data Structures & Algorithms (Fall 2012), Tsinghua University

11

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

Data Structures & Algorithms (Fall 2012), Tsinghua University

13

1. 绪论

(b) 大O记号

To measure is to know.

If you can not measure it, you can not improve it.

- Lord Kelvin

好读书不求甚解
每有会意，便欣然忘食
——陶渊明

郑俊辉

deng@tsinghua.edu.cn

尺规计算机及其算法

◆输入：平面上任一线段AB

输出：将AB三等分

◆算法 从A发出一条与AB不重合的射线p

在p上取AC' = C'D' = D'B'

联接B'C

经C'做B'C的平行线，交AB于D

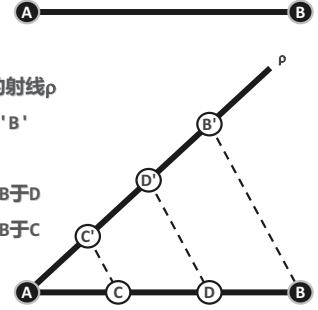
经C做B'C的平行线，交AB于C

◆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？ //三等分角 / 倍方 / ...

◆子程序：过直线外一点，做平行线



Data Structures & Algorithms (Fall 2012), Tsinghua University

10

好算法

◆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

Data Structures & Algorithms (Fall 2012), Tsinghua University

12

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），树（着重于平衡二分查找树）

优先队列（堆），字典（散列表），图、相关算法及应用

◆构造有效算法模块的常用技巧

顺序和二分查找，排序与选取，遍历

模式匹配，散列，几何查找

◆典型的算法设计策略与模式

迭代、贪心，递归、分治、减治

试探、剪枝、回溯，动态规划

◆基本的复杂度分析方法

渐进分析与相关记号递推关系、递归跟踪、分摊分析

Data Structures & Algorithms (Fall 2012), Tsinghua University

14

15

算法分析

◆两个主要方面

正确性：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： $T_A(P) = \text{算法 } A \text{求解问题实例 } P \text{的计算成本}$

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

规模接近，计算成本也接近

规模扩大，计算成本通常呈上升趋势

Data Structures & Algorithms (Fall 2012), Tsinghua University

16

问题规模 vs. 计算成本

- 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，简记作 $T(n)$
- 但是，这一定义似乎仍有问题 ...
- 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：在平面上的n个点中，找到所成三角形面积最小的三个点
通常，要枚举所有 $C(n, 3)$ 种组合
但运气好的话 ...
- 既然如此，又该如何定义 $T(n)$ 呢？
- 保险起见，还是取 $T(n) = \max(T(P) \mid |P| = n)$

Data Structures & Algorithms (Fall 2012), Tsinghua University

8

算法 vs. 计算效率

- 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？
- 不足以！
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，可能由不同人、用不同语言、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，需要一个理想的平台或模型
不依赖于人或具体的（高级）程序语言等因素
从而直接而准确地描述算法

Data Structures & Algorithms (Fall 2012), Tsinghua University

9

RAM

- Random Access Machine (J. Shepherdson & H. Sturgis, 1963)
//源自冯·诺依曼体系结构（1945），等价于通用图灵机
- 寄存器顺序编号，总数没有限制 //但愿如此
- 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
寄存器读写（赋值）、四则运算、比较、goto、call、return
- RAM模型是一般计算平台的简化与抽象
使我们可以独立于具体的平台，对算法的效率做可信的比较与评判
- 在RAM模型中
算法的运行时间 \propto 算法需要执行的基本操作次数
 $T(n)$ = 算法在RAM中求解规模为n问题所需的基本操作次数

Data Structures & Algorithms (Fall 2012), Tsinghua University

6

渐进分析

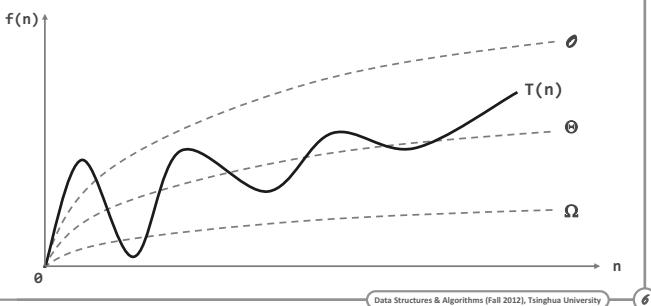
- 回到原先的问题：随着问题规模的增长，计算成本如何增长？
注意：这里更关心**足够大**的问题，注重考察成本的增长趋势
- 渐进分析：在问题规模足够大后，计算成本如何增长？
Asymptotic analysis：当 $n \gg 2$ 后，对于规模为n输入，算法
需执行的基本操作次数 $T(n) = ?$
需占用的存储单元数 $S(n) = ?$ //通常可不考虑，为什么？
- 大O记号 (big-O notation)
 $T(n) = O(f(n))$ iff $\exists c > 0$, 当 $n \gg 2$ 后, $T(n) < c \times f(n)$
- 与 $T(n)$ 相比， $f(n)$ 更为简洁，但依然反映前者的增长趋势
常系数可忽略： $O(f(n)) = O(c \times f(n))$
低次项可忽略： $O(n^a + n^b) = O(n^a)$, $a > b > 0$

Data Structures & Algorithms (Fall 2012), Tsinghua University

5

渐进分析

- 其它记号
 $T(n) = \Omega(f(n)) : \exists c > 0, n \gg 2$ 后, $T(n) > c \times f(n)$
 $T(n) = \Theta(f(n)) : \exists c_1 > c_2 > 0, n \gg 2$ 后, $c_1 \times f(n) > T(n) > c_2 \times f(n)$



Data Structures & Algorithms (Fall 2012), Tsinghua University

6

O(1)

- 常数 (constant function)
 $2 = 2012 = 2012 \times 2012 = 2012^{2012} = O(1)$ //以及所有RAM基本操作
- 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/2012 + 1); /* log*n */ for (i = n; i > 2012; i = log(i));`
 - 一定不含分支转向？ `if (1 + 1 != 2) goto UNREACHABLE;`
 - 一定不能有（递归）调用？
 - 全是顺序执行即可？
 - ...

Data Structures & Algorithms (Fall 2012), Tsinghua University

7

O(log^cn)

- 对数 $O(\log n)$ //为何不注明底数？
 $\ln n \mid \lg n \mid \log_{10} n \mid \log_{2012} n$
- 常底数无所谓
 $\forall a, b > 0, \log_a n = \log_b a \cdot \log_b n = \Theta(\log_b n)$
- 常数次幂无所谓
 $\forall c > 0, \log^c n = c \cdot \log n = \Theta(\log n)$
- 对数多项式 (ploy-log function)
 $123 * \log^{321} n + \log^{105}(n^2 - n + 1)$
- 这类算法非常有效，复杂度无限接近于常数
 $\forall c > 0, \log n = O(n^c)$

Data Structures & Algorithms (Fall 2012), Tsinghua University

8

O(n^c)

- 多项式 (polynomial function)
 - $100n + 200 = O(n)$
 - $(100n - 500)(20n^2 - 300n + 2012) = O(n \times n^2) = O(n^3)$
 - $(2012n^2 - 20)/(1999n - 1) = O(n^2/n) = O(n)$
 - 一般地： $a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0 = O(n^k)$, $a_k > 0$
- 线性 (linear function)：所有 $O(n)$ 类函数
- 从 $O(n)$ 到 $O(n^2)$
- 幂： $\sqrt{n} = O(n^{1/2})$
- 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！

Data Structures & Algorithms (Fall 2012), Tsinghua University

9

$\Theta(2^n)$

- ❖ 指数 (exponential function) //为何底数取2?
- ❖ $\forall c > 1, n^c = \Theta(2^n)$ //为什么?
 $n^{1000} = \Theta(1.0000001^n) = \Theta(2^n)$
 $1.0000001^n = \Omega(n^{1000})$
- ❖ 这类算法的计算成本增长极快，通常被认为不可忍受
- ❖ 从 $\Theta(n^c)$ 到 $\Theta(2^n)$ ，是从 **有效算法** 到 **无效算法** 的分水岭
- ❖ 很多问题的 $\Theta(2^n)$ 算法往往显而易见，但设计出 $\Theta(n^c)$ 算法却极其不易甚至，有时注定只能徒劳无功
- ❖ 更糟糕的是，这类问题要比我们想象的要多得多...

Data Structures & Algorithms (Fall 2012), Tsinghua University 10

2-Subset

- ❖ 【问题描述】
S为一组共n个正整数， $\sum S = 2m$
是否有S的子集T满足 $\sum T = m$?
- ❖ 【选举人制】
总统由各州议会选出的选举人团选举产生，而不是由选民直接选举产生
50个州加1个特区共538票
获270张选举人票即当选
//但是...
- ❖ 若共有两位候选人
是否可能恰好各得269票?

伊利诺伊	22	怀俄明	3	康涅狄格	8
弗吉尼亚	13	缅因	4	印第安纳	12
夏威夷	4	科罗拉多	8	西弗吉尼亚	5
犹他	5	佛罗里达	25	特拉华	3
新墨西哥	5	内布拉斯加	5	蒙大拿	3
田纳西	11	阿肯色	6	亚拉巴马	9
路易斯安那	9	俄勒冈	7	马里兰	10
肯塔基	8	亚利桑那	8	俄克拉何马	8
罗得岛	4	马萨诸塞	12	内华达	4
阿拉斯加	3	新罕布什尔	4	密苏里	11
明尼苏达	10	艾奥瓦	7	南达科他	3
南卡罗来纳	8	俄亥俄	21	得克萨斯	32
堪萨斯	6	北达科他	3	北卡罗来纳	14
密歇根	18	华盛顿	11	爱达荷	4
密西西比	7	加利福尼亚	54	新泽西	15
佐治亚	13	佛蒙特	3	威斯康星	11
宾夕法尼亚	23	纽约	33	华盛顿特区	3

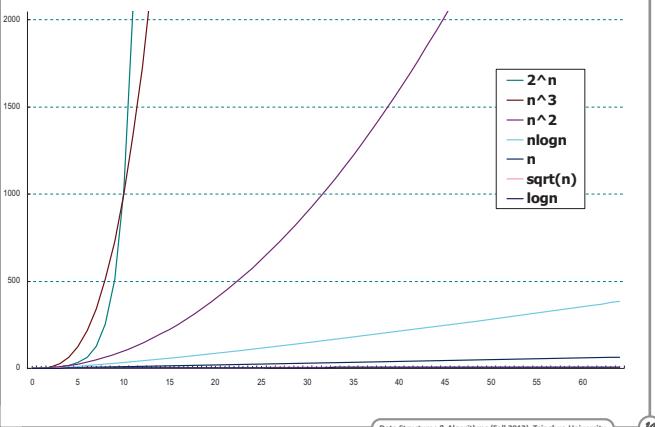
Data Structures & Algorithms (Fall 2012), Tsinghua University 11

2-Subset

- ❖ 直觉算法：逐一检查S的每一子集
- ❖ 定理： $|2^S| = 2^{|S|} = 2^n$
- ❖ 也就是说，直觉算法需要迭代 2^n 轮 //似乎不甚理想
- ❖ 还是直觉：应该有更好的办法吧？ //同意的举手
- ❖ 定理：2-SubSet是NP-complete的 //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，直觉算法已经是最优的了

Data Structures & Algorithms (Fall 2012), Tsinghua University 12

增长速度

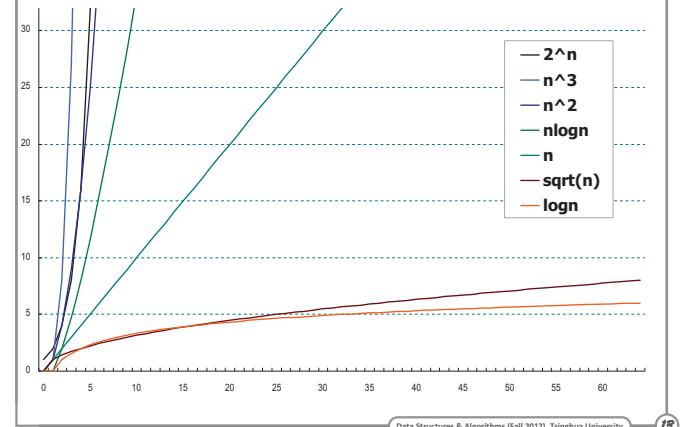


课后

- ❖ 举例：随问题实例规模增大，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RAM模型中衡量算法效率，为什么只需考察运行时间？
- ❖ 证明、证否或计算：Fibonacci数 $Fib(n) = \Theta(2^n)$
 $12n + 5 = \Theta(n\log n)$ $\log^2(n^{1024} - 2^*n^6 + 101) = \Theta(?)$
 $\log^dn = \Theta(n^c), \forall c > 0, d > 1$ $\log^{1.001}n = \Theta(\log(n^{1001}))$
 $(n^2 + 1) / (2n + 3) = \Theta(n)$ $n^{2012} = \Theta(n!)$
 $n! = \Theta(n^{2012})$ $2^n = \Theta(n!)$
- ❖ k-Subset：任给整数集S，判定S可否分为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

Data Structures & Algorithms (Fall 2012), Tsinghua University 14

增长速度



复杂度层次

$\Theta(1)$	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
$\Theta(n^c), c \text{ 常数}$	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	

Data Structures & Algorithms (Fall 2012), Tsinghua University 16

1. 结论

(c) 算法分析

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

- Francois Arago

郑俊辉

deng@tsinghua.edu.cn

算法分析

- ◆ 主要任务 = 正确性 + 复杂度
- ◆ 正确性 = 不变性 + 单调性
- ◆ 从代码执行的角度来看，时间复杂度等于基本指令的执行次数
亦即各行代码执行次数的总和
- 迭代循环：`for()`、`while()`、...
- 调用 & 递归（自我调用）
- 分支转向：`goto` //本质上就是迭代，幸好基本不用了
- ◆ 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

Data Structures & Algorithms (Fall 2012), Tsinghua University



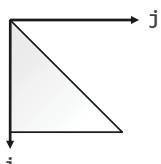
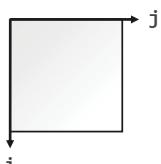
循环 vs. 级数和

- ```

◆ for (int i = 0; i < n; i++)
 for (int j = 0; j < n; j++)
 O10operation(i, j); //算术级数1
 // $\sum_{i=0..n-1} n = n + n + \dots + n$
 // = $n^2 = O(n^2)$

◆ for (int i = 0; i < n; i++)
 for (int j = 0; j < i; j++)
 O10operation(i, j); //算术级数2
 // $\sum_{i=0..n-1} i = 0 + 1 + \dots + (n-1) = n(n-1)/2 = O(n^2)$

```



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University



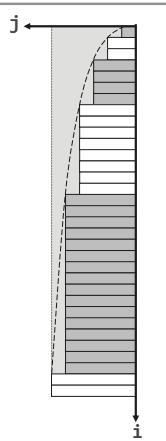
35

## 循环 vs. 级数和

- ```

◆ for (int i = 0; i <= n; i++)
    for (int j = 1; j < i; j += j)
        O10operation(i, j); //几何级数2
    //  $\sum_{i=0..n} \lceil \log_2 i \rceil = O(n \log n)$ 
    // (i = 0, 1, 2, 3~4, 5~8, 9~16, ...)
    // = 0 + 0 + 1 + 2*2 + 3*4 + 4*8 + ...
    // =  $\sum_{k=0.. \log_2 n} (k * 2^{k-1})$ 
    // =  $O(\log n * 2^{\log n})$  (CM page#33)

```



Data Structures & Algorithms (Fall 2012), Tsinghua University



37

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
- ◆ 版本0：[bubble.cpp](#)
- ◆ 版本1：[bubble1A.cpp](#) + [bubble1B.cpp](#)

```

for (bool sorted = false; sorted = !sorted; n--) //...
    for (int i = 1; i < n; i++) //检查A[0, n]内各相邻元素
        if (A[i-1] > A[i]) //若逆序
            { swap(A[i-1], A[i]); sorted = false; } //则交换
    
```
- ◆ 版本2：[bubble2.cpp](#)

Data Structures & Algorithms (Fall 2012), Tsinghua University



级数和

- ◆ 算数级数： $T(n) = 1 + 2 + \dots + n = n(n+1)/2 = O(n^2)$
- ◆ 平方级数： $T(n) = 1^2 + 2^2 + \dots + n^2 = n(n+1)(2n+1)/6 = O(n^3)$
- ◆ 幂级数： $T_a(n) = a^0 + a^1 + a^2 + \dots + a^n, a > 0$
最常见： $1+2+4+\dots+2^n = 2^{n+1}-1 = O(2^{n+1}) = O(2^n)$ //总和与末项同阶
- ◆ 收敛： $1/1/2 + 1/2/3 + 1/3/4 + \dots + 1/(n-1)/n = 1 - 1/n = O(1)$
 $1 + 1/2^2 + \dots + 1/n^2 < 1 + 1/2^2 + \dots = \pi^2/6 = O(1)$
- ◆ 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？
- ◆ 调和级数： $h(n) = 1 + 1/2 + 1/3 + \dots + 1/n = \Theta(\log n)$
// $h(n) < 1 + \int_{1..n} 1/x = 1 + \ln n = O(\log n)$
// $h(n) > \int_{1..n+1} 1/x = \ln(n+1) = \Omega(\log n)$
- ◆ 对数级数： $\log 1 + \log 2 + \log 3 + \dots + \log n = \log(n!) = \Theta(n \log n)$
- ◆ 更多级数：<http://mathworld.wolfram.com/Series.html>
- ◆ 如有兴趣，不妨读读：[Concrete Mathematics](#)

Data Structures & Algorithms (Fall 2012), Tsinghua University

36

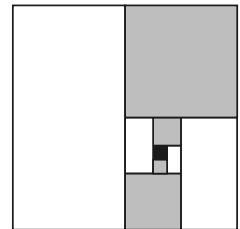
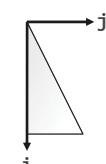
循环 vs. 级数和

- ```

◆ for (int i = 0; i < n; i++)
 for (int j = 0; j < i; j += 2012)
 O10operation(i, j); //算术级数3
 // ...

◆ for (int i = 1; i < n; i <= 1)
 for (int j = 0; j < i; j++)
 O10operation(i, j); //几何级数1
 // $1 + 2 + 4 + \dots + 2^{\lfloor \log_2(n-1) \rfloor}$
 // = $2^{\lceil \log_2 n \rceil} - 1$
 // = $O(n)$

```



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

38

## 取非极端元素

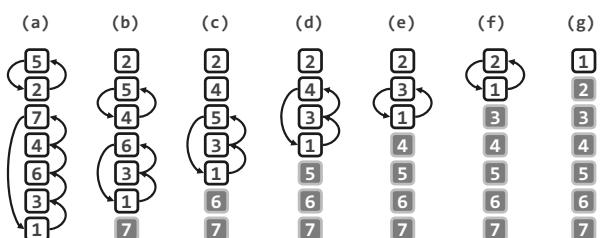
- ◆ 问题：给定整数子集  $S, |S| = n \geq 3$   
找出元素  $a \in S, a \neq \max(S)$  且  $a \neq \min(S)$
- ◆ 算法：任取的三个元素  $x, y, z \in S$ 
  - //若S以数组形式给出，不妨取前三个
  - //由于S是集合，这三个元素必互异
  - 确定并排除其中的最小、最大者
  - //不妨设  $x = \max\{x, y, z\}, y = \min\{x, y, z\}$
  - 输出剩下的元素  $z$
- ◆ 无论输入规模n多大，上述算法需要的执行时间都不变  
 $T(n) = \text{常数} = O(100) = \Omega(1) = \Theta(1)$

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

39

## 起泡排序

- ◆ 问题：该算法必然会结束？至多需迭代多少趟？
- ◆ 不变性：经k轮扫描交换后，最大的k个元素必然就位  
单调性：经k轮扫描交换后，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



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University



40

## 起泡排序

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

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

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

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

❖ 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： $T(n) = n-1 = \Omega(n)$

Data Structures & Algorithms (Fall 2012), Tsinghua University

9

Data Structures & Algorithms (Fall 2012), Tsinghua University

10

## Back-Of-The-Envelope Calculation

$\diamond 787 \times 360/7.2 = 787 \times 50 = 39,350 \text{ km}$



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

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

$3\text{生}3\text{世} = 300 \text{ yr} = 10^{10} = (1 \text{ googel})^{(1/10)} \text{ sec}$

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

Data Structures & Algorithms (Fall 2012), Tsinghua University

10

## Back-Of-The-Envelope Calculation

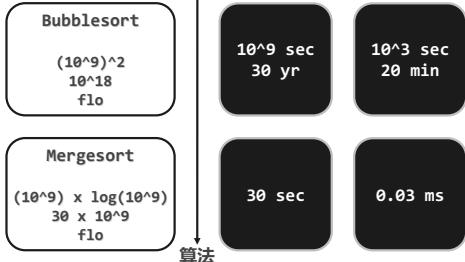
❖ 考察对全国人口普查数据的排序

普查数据的排序

$n = 10^9 \dots$



硬件



算法

Data Structures & Algorithms (Fall 2012), Tsinghua University

11

## 课后

❖ 通过指出不变性并做数学归纳，试证明本章各算法的正确性

❖ 试举例说明，非直接递归的函数调用，也会引起代码的多次执行

❖ 试举例说明，`O1operation()`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

Data Structures & Algorithms (Fall 2012), Tsinghua University

12

## 1. 绪论

## (d) 迭代与递归

迭代乃人工，递归方神通

To iterate is human, to recurse, divine.

凡治众如治寡，分步是也

The control of a large force is  
the same principle as  
the control of a few men:  
it is merely a question of  
dividing up their numbers.

邓俊辉

deng@tsinghua.edu.cn

## 数组求和：迭代

❖ 问题：计算给定的 $n$ 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： $T(n) = 1 + n*1 + 1 = n + 2$

❖  $T(n) = \Theta(n) = \Omega(n) = \Theta(n)$

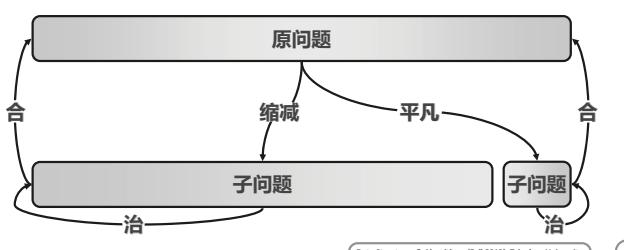
//空间呢？

Data Structures & Algorithms (Fall 2012), Tsinghua University

13

## 减而治之

❖ 【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  
分别求解子问题  
由子问题的解，得到原问题的解



Data Structures & Algorithms (Fall 2012), Tsinghua University

14

## 数组倒置

❖ 任给一个数组`A[0, n]`，将其中元素的次序前后颠倒

❖ `void reverse(int* A, int lo, int hi);`

❖ 递归版

```
if (lo < hi) //问题规模的奇偶性不变，需要两个递归基
 { swap(A[lo], A[hi]); reverse(A, lo+1, hi-1); }
```

❖ 迭代原始版

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

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

Data Structures & Algorithms (Fall 2012), Tsinghua University

15

## 数组求和：线性递归

```
* sum(int A[], int n) {
 return
 (1 > n) ?
 0 : sum(A, n-1) + A[n-1];
}
```

## ◆ 递归跟踪 (recursion trace) 分析

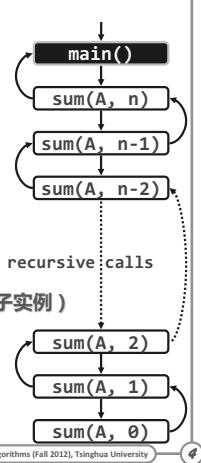
检查每个递归实例

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

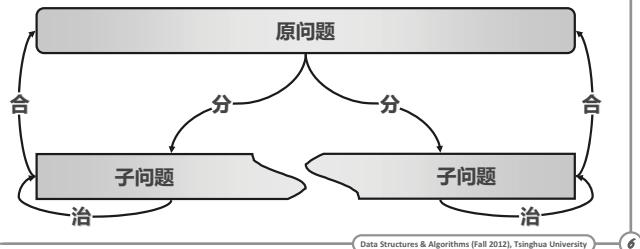
◆ 本例中, 单个递归实例自身只需 $O(1)$ 时间

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



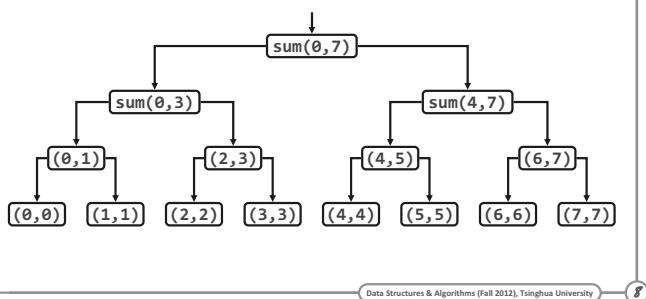
## 分而治之

◆ 【Divide-and-conquer】为求解一个大规模的问题, 可以将其划分为若干子问题, 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



## 数组求和：二分递归

◆  $T(n) = \text{各层递归实例所需时间之和}$  //递归跟踪  
 $= O(1) \times (2^0 + 2^1 + 2^2 + \dots + 2^{\log n})$   
 $= O(1) \times (2^{\log n+1} - 1) = O(n)$



◆ Fibonacci数列: {0, 1, 1, 2, 3, 5, 8, 13, ...}  
 $\text{fib}(0) = 0, \text{fib}(1) = 1, \text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$

◆ 二分递归版 //为何这么慢?  
 $\text{return } (2 > n) ? n : \text{fib}(n-1) + \text{fib}(n-2);$

◆ 复杂度:  $T(0) = T(1) = 1$   
 $T(n) = T(n-1) + T(n-2) + 1, \quad n > 1$

令  $S(n) = (T(n) + 1)/2$   
则  $S(0) = 1 = \text{fib}(1), S(1) = 1 = \text{fib}(2)$   
故  $S(n) = S(n-1) + S(n-2) = \text{fib}(n+1)$   
 $T(n) = 2S(n) - 1 = 2\text{fib}(n+1) - 1$   
 $= O(\text{fib}(n+1)) = O(2^n)$

◆ 迭代版:  $T(n) = O(n)$



## 数组求和：线性递归

◆ 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 $\text{sum}(A, n-1)$  // $T(n-1)$   
再累加上 $A[n-1]$  // $O(1)$   
递归基:  $\text{sum}(A, 0)$  // $O(1)$

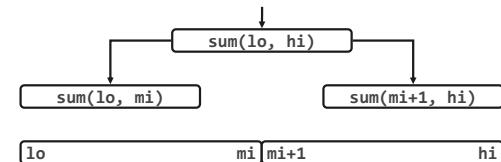
◆ 递推方程  $T(n) = T(n-1) + O(1)$  //recurrence  
 $T(0) = O(1)$  //base

◆ 求解  
 $T(n) - n = T(n-1) - (n-1)$   
 $= T(n-2) - (n-2)$   
 $= \dots$   
 $= T(0)$   
 $T(n) = O(1) + n = O(n)$

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 数组求和：二分递归

```
* 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)
```



## 数组求和：二分递归

◆ 从递推的角度看, 为求解 $\text{sum}(A, lo, hi)$   
需递归求解 $\text{sum}(A, lo, mi)$ 和 $\text{sum}(A, mi+1, hi)$  // $2*T(n/2)$   
进而将子问题的解累加 // $O(1)$   
递归基:  $\text{sum}(A, lo, lo)$  // $O(1)$

◆ 递推关系  $T(n) = 2*T(n/2) + O(1)$   
 $T(1) = O(1)$

◆ 求解  $T(n) = 2*T(n/2) + c_1$   
 $T(n) + c_1 = 2*(T(n/2) + c_1) = 2^{2n}(T(n/4) + c_1)$   
 $= \dots$   
 $= 2^{\log n}(T(1) + c_1) = n*(c_2 + c_1)$   
 $T(n) = (c_1 + c_2)n - c_1 = O(n)$

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## fib()

◆ Fibonacci数列: {0, 1, 1, 2, 3, 5, 8, 13, ...}  
 $\text{fib}(0) = 0, \text{fib}(1) = 1, \text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$

◆ 二分递归版 //为何这么慢?  
 $\text{return } (2 > n) ? n : \text{fib}(n-1) + \text{fib}(n-2);$

◆ 复杂度:  $T(0) = T(1) = 1$   
 $T(n) = T(n-1) + T(n-2) + 1, \quad n > 1$

令  $S(n) = (T(n) + 1)/2$   
则  $S(0) = 1 = \text{fib}(1), S(1) = 1 = \text{fib}(2)$   
故  $S(n) = S(n-1) + S(n-2) = \text{fib}(n+1)$   
 $T(n) = 2S(n) - 1 = 2\text{fib}(n+1) - 1$   
 $= O(\text{fib}(n+1)) = O(2^n)$

◆ 迭代版:  $T(n) = O(n)$

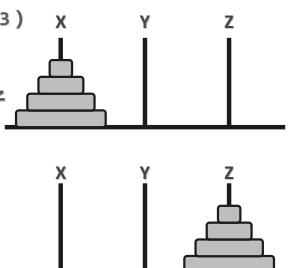
Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## Hanoi塔

◆ Tower of Hanoi (E. Lucas, 1883)  
给定三根木桩 X、Y 和 Z  
X自下而上串有半径递减的n个盘子  
将这n个盘子移到Z上  
问题 $\text{hanoi}(n, X, Y, Z)$

◆ 在中间过程中, 始终要求  
所有盘子只能串在木桩上  
自下而上, 每根木桩上盘子的半径必须依次递减 (保持塔状)  
每次只能移动一个盘子

◆ 如何求解 $\text{hanoi}(n)$ ?



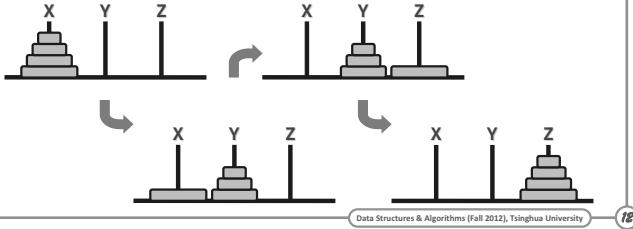
Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## Hanoi塔

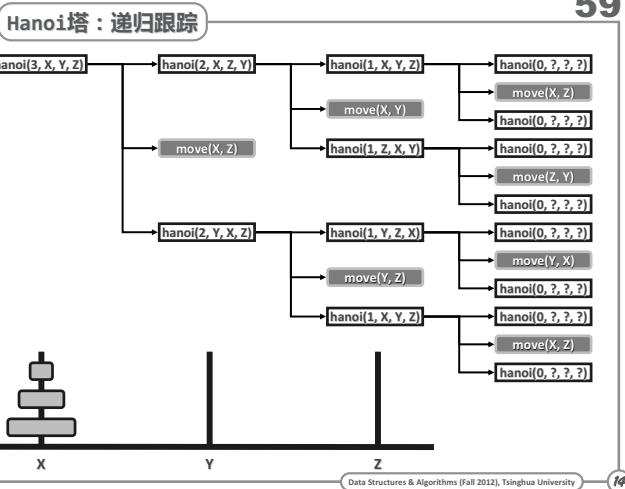
若已知Hanoi( $n-1$ )的解法，则只需

- 将前 $n-1$ 个盘子从X（借助Z）移至Y //hanoi( $n-1$ , X, Z, Y)
- 将X上最大的那只盘子（直接）移至Z //move(X, Z)
- 将前 $n-1$ 个盘子从Y（借助X）移至Z //hanoi( $n-1$ , Y, X, Z)

分而治之：hanoi(n), hanoi(n-1), ..., hanoi(1), hanoi(0)



Data Structures & Algorithms (Fall 2012), Tsinghua University 18



Data Structures & Algorithms (Fall 2012), Tsinghua University 19

## 典型的递推方程

| 递推式                   | 解                  | 实例             |
|-----------------------|--------------------|----------------|
| $T(n) = T(n-1) + 1$   | $\Theta(n)$        | 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   |
| $T(n) = 2*T(n-1) + 1$ | $\Theta(2^n)$      | 汉诺塔、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           |

Data Structures & Algorithms (Fall 2012), Tsinghua University 16

## 尾递归及递归消除

函数的最后一步若是递归调用自身，则可直接改写为迭代形式

```
Fac(n) {
 | Fac(n) {
 | | int f = 1; //记录子问题的解
 | | next: //引入转向标志，模拟递归调用
 if (1 > n) return 1;
 else return n*Fac(n-1);
 | | f *= n--;
 | | goto next; //跳转，模拟递归返回
}
```

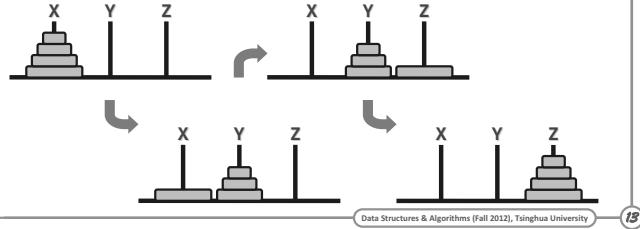
两种版本的空间复杂度有何差异？

Data Structures & Algorithms (Fall 2012), Tsinghua University 18

## Hanoi塔

hanoi(n, X, Y, Z) //借助Y，将X上的n个盘子移往Z

```
if (n > 0)
 hanoi(n-1, X, Z, Y); //借助Z，将X顶端的n-1个盘子移往Y
 move(X, Z); //将最后一个盘子从X移往Z
 hanoi(n-1, Y, X, Z); //借助X，将Y顶端的n-1个盘子移往Z
```



Data Structures & Algorithms (Fall 2012), Tsinghua University 19

## Hanoi塔：递推方程

记  $T(n) = \Theta(1) \times \# \text{ 盘子移动}$ ，则

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

令  $S(n) = T(n) + 1$

$$则 S(0) = 1, S(n) = 2 * (T(n-1) + 1) = 2 * S(n-1)$$

解得： $S(n) = 2^n, T(n) = 2^n - 1 = \Theta(2^n)$

若共有64个盘子， $T(64) = 2^{64} - 1 = 2^4 \cdot (2^{10})^6 = 16 \times 10^{18}$

即使你的电脑每秒能够移动1G =  $10^9$ 个盘子

$$T(64) = 16 \times 10^9 \text{ sec} = 500 \text{ yr}$$

即便是天河-1A

$$T(64) = 16 \times 10^3 \text{ sec} = 5 \text{ hr}$$

Data Structures & Algorithms (Fall 2012), Tsinghua University 19

## 递归，还是不递归？

递归 易于实现，易于理解

空间效率低

时间效率低 //比如fib()

在讲求效率时，要尽可能消除递归

最直接的办法，就是改写为迭代模式

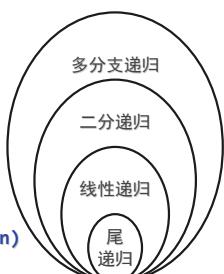
例：阶乘函数  $fac(n) = n! //\text{迭代版} : \Theta(n)$

递归版  $fac(n) = n * fac(n-1)$

递推分析  $T(n) = T(n-1) + 1$

$$T(0) = 1$$

$$\text{解得: } T(n) = n + 1 = \Theta(n) //\text{线性递归}$$



Data Structures & Algorithms (Fall 2012), Tsinghua University 19

## 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？

温习：程序设计基础（第3版），第11章

自学：Introduction to Algorithms, §15.1, §15.3, §15.4

Data Structures & Algorithms (Fall 2012), Tsinghua University 19

## 1. 绪论

### (x1) 局限

邓俊辉

不学诗，何以言；不学礼，何以立

deng@tsinghua.edu.cn

### 循环移位：蛮力版

❖ 将数组A[0, n]中元素向左循环移动k个单元

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

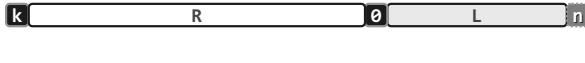
//从A[s]出发, 以k为间隔循环左移, O(n/GCD(n,k)) = LCM(n,k)/k
int shift(int* A, int n, int s, int k) {
 int b = A[s]; int i = s, j = (s+k) % n; int mov = 0;
 while (s != j) //从首元素出发, 以k为间隔, 依次左移k位
 { A[i] = A[j]; i = j; j = (j + k) % n; mov++; }
 A[i] = b; return mov + 1; //最后, 起始元素转入对应位置
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

### 循环移位：迭代版

//通过GCD(n, k)轮迭代, 将数组循环左移k位, O(n)

```
void shift1(int* A, int n, int k) {
 for (int s = 0, mov = 0; mov < n; s++)
 //O(GCD(n, k)) = O(n * k / LCM(n, k))
 mov += shift(A, n, s, k);
}
```



Data Structures & Algorithms (Fall 2012), Tsinghua University

### 循环移位：倒置版

//借助倒置算法, 将数组循环左移k位, O(3n)

```
void shift2(int* A, int n, int k) {
 reverse(A, k); //O(3k/2)
 reverse(A+k, n-k); //O(3(n-k)/2)
 reverse(A, n); //O(3n/2)
}
```



Data Structures & Algorithms (Fall 2012), Tsinghua University

a<sup>98765</sup>

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

❖ 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

Data Structures & Algorithms (Fall 2012), Tsinghua University

### a<sup>10110b</sup>

```

= a ^ [1 × 2^4 + 0 × 2^3 + 1 × 2^2 + 1 × 2^1 + 0 × 2^0]
= (a^2^4)^1 × (a^2^3)^0 × (a^2^2)^1 × (a^2^1)^1 × (a^2^0)^0
= pow(pow(a, 2^4), 1) = pow(sqr(pow(a, 2^3)), 1)
 × pow(pow(a, 2^3), 0) × pow(sqr(pow(a, 2^2)), 0)
 × pow(pow(a, 2^2), 1) × pow(sqr(pow(a, 2^1)), 1)
 × pow(pow(a, 2^1), 1) × pow(sqr(pow(a, 2^0)), 1)
 × pow(pow(a, 2^0), 0) × pow(pow(a, 2^0), 0)

```

❖ pow(x, 0) = 1, pow(x, 1) = x, pow(x, 2) = sqr(x)

都可在O(1)时间内得到

❖ 故对应于每个数位, 只需O(1)时间!

Data Structures & Algorithms (Fall 2012), Tsinghua University

### 幂函数 : O(r)

#### ❖ 算法

```

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

```

#### ❖ 复杂度

循环次数  
= n的二进制位数  
= r  
=  $\lceil \log_2(n+1) \rceil$   
 $T(r) = 1 + 1 + 4 \times r + 1 = O(r)$

❖ 从O(2^r)到O(r)的改进  
实际效果如何?

Data Structures & Algorithms (Fall 2012), Tsinghua University

## 幂函数：悖论

- 观察：随着 $n$ 的增加， $\text{power}(n) = a^n$ 的二进制宽度可度量为 $\Theta(n)$
- 根据以上算法，可在 $\Theta(\log n)$ 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？

## 1. 绪论

## (x2) 排序与下界

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

## 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 //广义的次序： $\leq$ ，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  
 $K_{i_1} \leq \dots \leq K_{i_n}$
- 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
  - 飞机调度系统：(航班号，起降时间)
  - 文件系统：(文件名，扩展名，大小，日期)
  - 加速查找：电话簿
  - 加速比对：在花名册A和B之间，找出重复出现者
- ...

## 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 (worst-case optimal) 的算法
- 排序所需的时间，主要取决于
  - 关键码比较的次数 / #key comparisons
  - 元素交换的次数 / #data swaps
- 就地 (in-place)：除输入数据本身外，只需 $\Theta(1)$ 附加空间
- 退化 (degeneracy)：若存在雷同关键码，则排序结果不唯一
- 稳定 (stable) 算法：关键码雷同的元素，在排序后相对位置保持

## 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 &amp; 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？

## 难度与下界

- 由上可见，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 //广义的次序： $\leq$ ，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  
 $K_{i_1} \leq \dots \leq K_{i_n}$
- 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
  - 飞机调度系统：(航班号，起降时间)
  - 文件系统：(文件名，扩展名，大小，日期)
  - 加速查找：电话簿
  - 加速比对：在花名册A和B之间，找出重复出现者
- ...

## 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)
- 在线 / 脱机 (online / offline)
- 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)

## 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()`

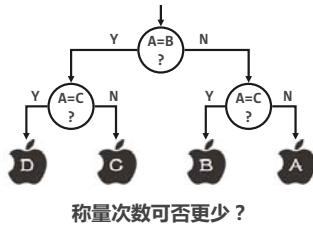
- 任何CBA在最坏情况下都需 $\Omega(n \log n)$ 时间才能完成排序 //为什么

## 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

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



称量次数可否更少？

Data Structures & Algorithms (Fall 2012), Tsinghua University

6

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  $\geq \log_2(n!) = \Omega(n \log n)$  //为什么

Stirling approximation :

$n! \sim \sqrt{2\pi n} \cdot (n/e)^n$  或  $\ln n! = n \ln n - n + O(\log n)$

Data Structures & Algorithms (Fall 2012), Tsinghua University

8

## 课后

起泡排序的平均性能如何？

起泡排序是稳定的吗？

如何通过预处理，使不稳定的排序算法成为（人为）稳定的？

在时间、空间方面，你设计的方法为此需额外付出多大代价？

【Element Uniqueness】

任意 $n$ 个实数中是否有两个相等？

EU问题的下界是什么？

【Integer Element Uniqueness】

任意 $n$ 个整数中是否有两个相等？

IEU问题的下界是什么？

Data Structures & Algorithms (Fall 2012), Tsinghua University

10

## 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

数据结构 = 基于某种特定语言实现ADT的一整套算法

区别与联系

| 抽象定义       | 具体实现         |
|------------|--------------|
| 外部的逻辑特性    | 内部的表示与实现     |
| 操作&语义      | 完整的算法        |
| 一种定义       | 多种实现         |
| 不考虑时间复杂度   | 与时间复杂度密切相关   |
| 不涉及数据的存储方式 | 要考虑数据的具体存储机制 |

Data Structures & Algorithms (Fall 2012), Tsinghua University

11

## 代数判定树

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

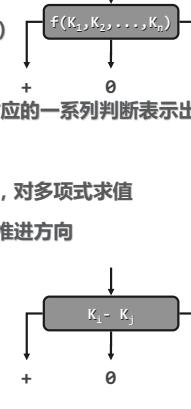
将任意一组关键码做为变量，对多项式求值

根据结果的符号，确定算法推进方向

比较树 (comparison tree)

最简单的ADT

二元一次多项式，形如： $K_i - K_j$



Data Structures & Algorithms (Fall 2012), Tsinghua University

7

## 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：

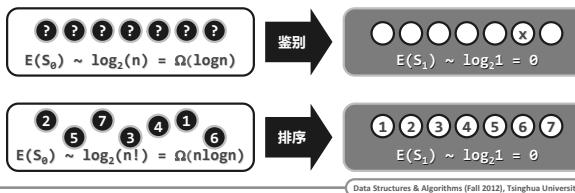
数据系统S中蕴含的信息量，可由信息熵度量

$E(S) \sim \log_2 N$  (N为S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



Data Structures & Algorithms (Fall 2012), Tsinghua University

## 2. 向量

(a) 接口与实现

邓俊辉

deng@tsinghua.edu.cn

## Application = Interface x 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

//造汽车

应用：便捷地通过操作接口使用数据结构

//开汽车

按照ADT规范

信息隐蔽和数据封装，使用与实现得以分离

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

前者可超脱于具体实现；后者只需将精力于具体环节的优化

不同的算法与数据结构可以任意组合，便于确定最优配置

每种操作接口只需实现一次，代码篇幅缩短，软件复用度提高

## 从数组到向量

◆ 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], ..., A[n-1]



◆ 反之，每个元素由编号唯一指代，可直接访问

A[i]的物理地址 = A + i \* 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) —— 对应

元素的类型不限于基本类型，甚至可以互不相同

操作、管理维护更加简化、统一与安全

可更为便捷地参与复杂数据结构的定制与实现

Data Structures & Algorithms (Fall 2012), Tsinghua University

3

Data Structures & Algorithms (Fall 2012), Tsinghua University

4

## 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(y)      | 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() | 8   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   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)    | 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                     |             | search(9)    | 4   3   4   6   7   9      |             |

Data Structures & Algorithms (Fall 2012), Tsinghua University

5

Data Structures & Algorithms (Fall 2012), Tsinghua University

6

## 向量ADT接口

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

Data Structures & Algorithms (Fall 2012), Tsinghua University

7

## 构造与析构

```

◆ Vector(int c = DEFAULT_CAPACITY)
{ _elem = new T[_capacity = c]; _size = 0; } //默认

◆ Vector(T* A, Rank lo, Rank hi){ copyFrom(A, lo, hi); }
Vector(T* A, Rank n) { copyFrom(A, 0, n); }
//数组区间或整体复制

◆ 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

```

Data Structures & Algorithms (Fall 2012), Tsinghua University

7

Data Structures & Algorithms (Fall 2012), Tsinghua University

8

## Vector模板类

typedef int Rank; //秩

#define DEFAULT\_CAPACITY 3 //默认初始容量 (实际应用中可设置为更大)

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 */
};
```

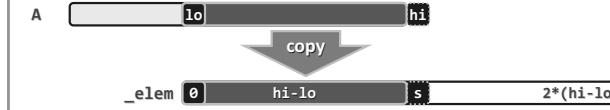
Data Structures & Algorithms (Fall 2012), Tsinghua University

6

## 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)
}
```



Data Structures & Algorithms (Fall 2012), Tsinghua University

8

## 2. 向量

## (b) 可扩充向量

郑俊辉

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 | \_capacity  
\_size : 当前的实际规模n    \_elem

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

1. 上溢 (overflow) : \_elem[] 不足以存放所有元素  
尽管此时系统仍有足够的空间

2. 下溢 (underflow) : \_elem[] 中的元素寥寥无几

装填因子 (load factor)  $\lambda = \frac{\text{_size}}{\text{_capacity}} \ll 50\%$

◆ 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？如何保证高效率？

Data Structures & Algorithms (Fall 2012), Tsinghua University

9

## 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

**适当地**扩大内部数组的容量



Data Structures & Algorithms (Fall 2012), Tsinghua University

## 扩容算法实现

```
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，尽管扩容之后物理地址改变，却不出现野指针

◆ 为何必须采用“容量加倍”的策略呢？

其它策略是否可行？

Data Structures & Algorithms (Fall 2012), Tsinghua University

## 容量递增策略

## ◆ 在即将上溢之前，追加固定大小的容量

```
T* oldElem = _elem; _elem = new T[_capacity += INCREMENT];
```

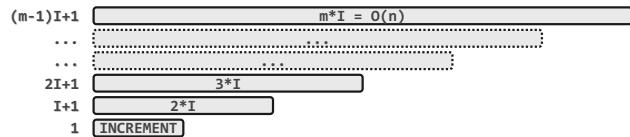
◆ 最坏情况：在初始容量0的空向量中，连续插入 $n = m \cdot I \gg 2$ 个元素...

◆ 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

0, I, 2I, ..., (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 $O(1)$



Data Structures & Algorithms (Fall 2012), Tsinghua University

## 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

可以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

Data Structures & Algorithms (Fall 2012), Tsinghua University

6

## 2. 向量

## (c) 无序向量

邓俊辉

deng@tsinghua.edu.cn

## 元素访问

◆ 通过V.get(r)和V.put(r, e)接口，固然可以读、写向量元素

但便捷性远不如数组元素的下标式访问方式A[r]

◆ 通过重载下标操作符“[]”，便可沿用数组的下标方式访问向量元素

◆ template <typename T>

```
T& Vector<T>::operator[](Rank r) const //0 <= r < _size
{ 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，暂且利用assertion机制，简易处理入口参数越界等意外  
实用中应采用更为严格的方式

Data Structures & Algorithms (Fall 2012), Tsinghua University

```
template <typename T> //e作为秩为r元素插入，0 <= r <= 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++; //置入新元素，更新容量
 return r; //返回秩
}
```

(a) may be full

(b) expanded if necessary right shift

(c) right shift

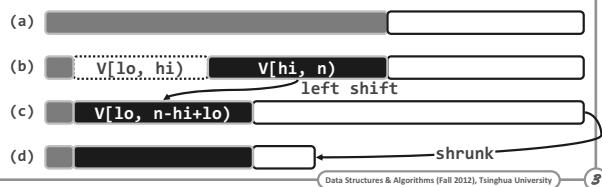
(d) e

Data Structures & Algorithms (Fall 2012), Tsinghua University

g

## 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hi-lo个单元
 _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
}
```



## 删除

❖ 单元素删除操作，可视作区间删除操作的特例，重载实现

```
❖ template <typename T> //删除向量中秩为r的元素，0<=r<size
T Vector<T>::remove(Rank r) { //O(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
}
```

❖ 如果反过来，基于单元素删除接口实现区间删除，效果如何？

每次循环耗时正比于删除区间的后缀长度 =  $n - hi = O(n)$

循环次数等于区间宽度 =  $hi - lo = O(n)$

如此，将导致总体  $O(n^2)$  的复杂度

Data Structures & Algorithms (Fall 2012), Tsinghua University

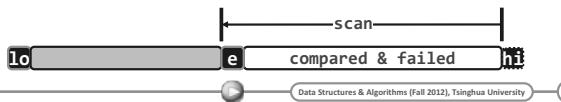
## 查找

❖ 判等器与比较器

无序向量：假定 T 为可判等的基本类型，或已重载操作符“=”或“!=”  
有序向量：假定 T 为可比较的基本类型，或已重载操作符“<”或“>”

例如：稍后介绍的词条类 `Entry`

```
❖ template <typename T> //0 <= lo < hi <= _size
Rank Vector<T>::find(T const& e, Rank lo, Rank hi) const
{ //O(hi - lo) = O(n)，在命中多个元素时可返回秩最大者
 while ((lo < hi--) && (_elem[hi] != e)); //逆向查找
 return hi; //hi < lo意味着失败；否则hi即命中元素的秩
}
```



Data Structures & Algorithms (Fall 2012), Tsinghua University

## 唯一化：正确性及复杂度

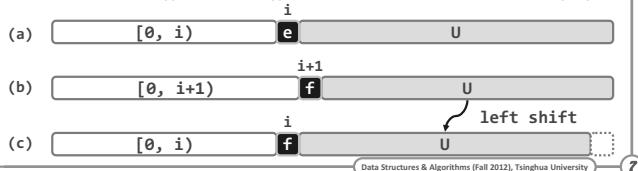
❖ 不变性：当前元素  $V[i]$  前缀  $V[0, i)$  中的元素均互异

初始  $i = 1$  时自然成立

❖ 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

❖ 每轮迭代中 `find()` 和 `remove()` 累计耗费线性时间，故总体为  $O(n^2)$



Data Structures & Algorithms (Fall 2012), Tsinghua University

## 遍历：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
}
```

## 唯一化：算法

```
❖ 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]
 //在其前缀中寻找与之雷同者（至多一个）
 (find(_elem[i], 0, i) < 0) ?
 //若无雷同则继续考查其后继，否则删除雷同者
 i++ : remove(i);
 return oldSize - _size; //向量规模变化量，即删除元素总数
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

## 2. 向量

## (d) 有序向量

群猴道：“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。”

郑俊辉

deng@tsinghua.edu.cn

Data Structures & Algorithms (Fall 2012), Tsinghua University

9

## 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，逆序相邻元素的数目，可用以度量向量的**逆序程度**

◆ template <typename T> //返回逆序相邻元素对的总数

```
int Vector<T>::disordered() const {
 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
}
```

◆ 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

Data Structures & Algorithms (Fall 2012), Tsinghua University

## 唯一化（低效版）：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() { //逐个剔除重复元素，效率低
 int oldSize = _size; int i = 0; //从首元素开始
 while (i < _size - 1) //从前向后，逐一比对各对相邻元素
 //若雷同，则删除后者；否则，转至后一元素
 (_elem[i] == _elem[i + 1]) ? remove(i + 1) : i++;
 return oldSize - _size; //向量规模变化量，即删除元素总数
}
```

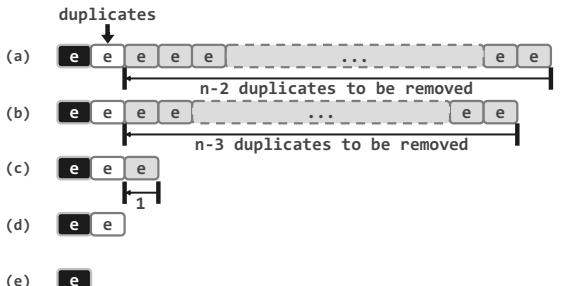
Data Structures & Algorithms (Fall 2012), Tsinghua University

## 唯一化（低效版）：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：每次需都调用一次remove()，耗时  $\Theta(n-1) \sim \Theta(1)$

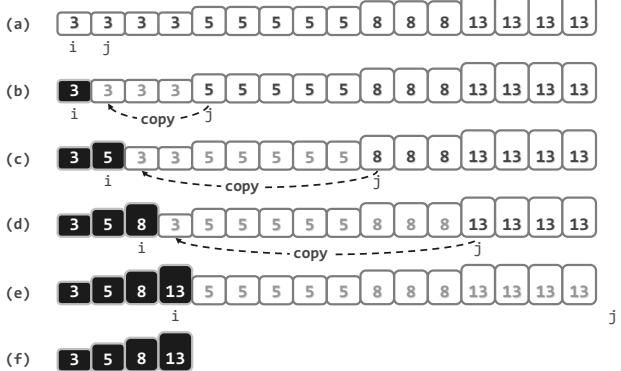
累计  $\Theta(n^2)$  — 尽管省去find()，总体竟与**无序向量**相同



Data Structures & Algorithms (Fall 2012), Tsinghua University

## 唯一化：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



Data Structures & Algorithms (Fall 2012), Tsinghua University

## 二分查找（版本A）：原理

◆ 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

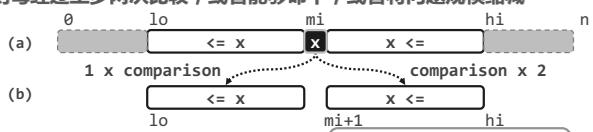
$S[lo, mi] \leq S[mi] \leq S[mi+1, hi]$  //  $S[mi]$  称作轴点

◆ 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：

- 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)  $x < e$ ：则  $e$  若存在必属于右侧子区间  $S[mi+1, hi]$ ，亦可递归深入
- 3)  $e = x$ ：已在此处命中，可随即返回

◆ 若轴点  $mi$  总是取作中点 //二分或折半策略

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



Data Structures & Algorithms (Fall 2012), Tsinghua University

## 唯一化（高效版）：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() { //成批剔除重复元素，效率高
 int oldSize = _size; int i = 0; //从首元素开始
 while (i < _size - 1) //从前向后，逐一比对各对相邻元素
 //若雷同，则删除后者；否则，转至后一元素
 (_elem[i] == _elem[i + 1]) ? remove(i + 1) : i++;
 return oldSize - _size; //向量规模变化量，即删除元素总数
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

## 唯一化（高效版）：算法

◆ 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

若能以重复区间为单位，成批删除雷同元素，性能必将改进



◆ template <typename T>

```
int Vector<T>::uniquify() { //成批剔除重复元素，效率更高
 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
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

## 查找

◆ 功能：在有序向量区间  $V[lo, hi]$  中，确定不大于  $e$  的**最后一个**元素

$-\infty < e < V[lo]$  时，返回  $lo-1$  (左侧哨兵)

$V[hi-1] < e < +\infty$  时，返回  $hi-1$  (右侧哨兵左邻)

如此，可便于有序向量的维护： $V.insertAfter(V.search(e), e)$



◆ template <typename T> //查找算法统一接口

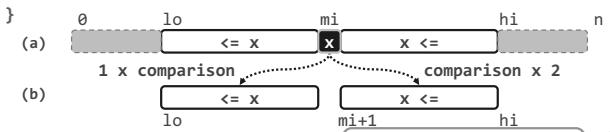
```
Rank Vector<T>::search(T const& e, Rank lo, Rank hi) const
{ //0 <= lo < hi <= _size
 return (rand() % 2) ? //按各50%的概率随机选用
 binSearch(_elem, e, lo, hi) //二分查找算法，或者
 : fibSearch(_elem, e, lo, hi); //Fibonacci查找算法
} //约定返回“不小于e的最前一个元素”呢？
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

## 二分查找（版本A）：实现

◆ template <typename T>

```
static Rank binSearch(T* A, T const& e, Rank lo, Rank hi) {
 while (lo < hi) {
 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+1, hi)
 else return mi; //在mi处命中
 }
 return -1; //查找失败
}
```

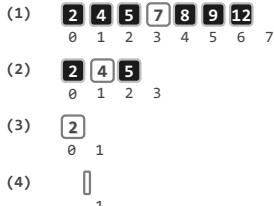
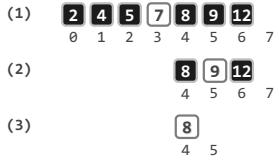


Data Structures & Algorithms (Fall 2012), Tsinghua University

## 二分查找(版本A)：实例与复杂度

❖ `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： $T(n) = T(n/2) + \Theta(1) = \Theta(\log n)$ ，大大优于顺序查找

递归跟踪：轴点总取中点，递归深度 $\Theta(\log n)$ ；各递归实例均耗时 $\Theta(1)$

Data Structures & Algorithms (Fall 2012), Tsinghua University

9

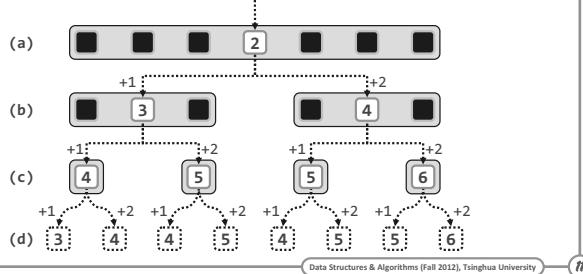
## 二分查找(版本A)：查找长度

❖  $n = 7$ 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 =  $36/8 = 4.50$



Data Structures & Algorithms (Fall 2012), Tsinghua University

11

## Fibonacci查找：实现

```
❖ template <typename T>
static Rank fibSearch(T* A, T const& e, Rank lo, Rank hi) {
 Fib fib(hi - lo); //用O(log_phi(n=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？
 //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+1, hi]
 else return mi; //在mi处命中
 }
 return -1; //查找失败
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

12

## 二分查找(版本B)：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

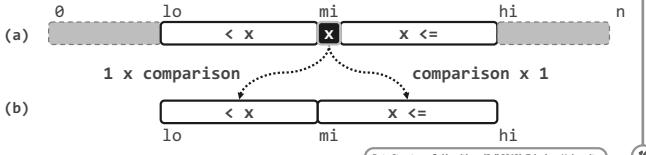
如此，所有分支只有2个方向，而不再是3个

❖ 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



Data Structures & Algorithms (Fall 2012), Tsinghua University

13

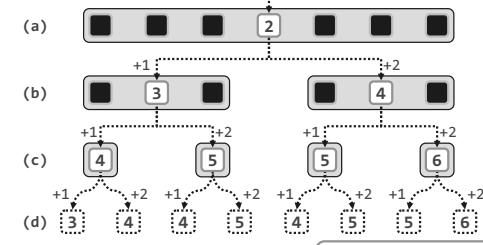
## 二分查找(版本A)：查找长度

❖ 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，平均成功查找长度为 $\Theta(1.50 \cdot \log n)$  //详见教材



Data Structures & Algorithms (Fall 2012), Tsinghua University

## Fibonacci查找：改进思路及原理

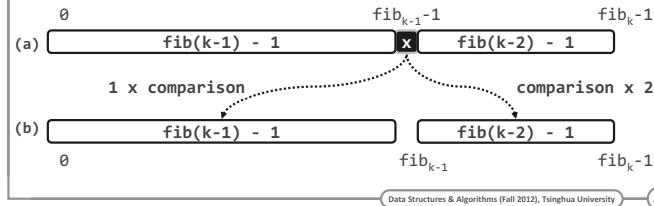
❖ 以上二分查找的效率仍有改进余地，因为不难发现

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

❖ 若能利用递归深度对比较次数做补偿，平均查找长度应该能够缩短

❖ 比如，若假设 $n = \text{fib}(k) - 1$ ，则可取 $mi = \text{fib}(k-1) - 1$

于是，前、后子向量的长度分别为 $\text{fib}(k-1) - 1, \text{fib}(k-2) - 1$



Data Structures & Algorithms (Fall 2012), Tsinghua University

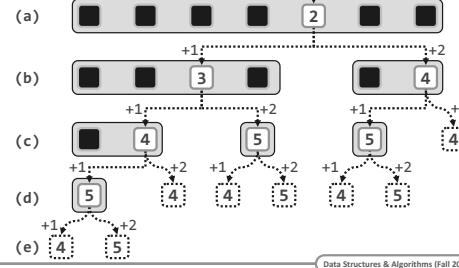
## Fibonacci查找：查找长度

❖ 平均成功查找长度为 $\Theta(1.44 \cdot \log n)$ ，略优于二分查找 //详见教材

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

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

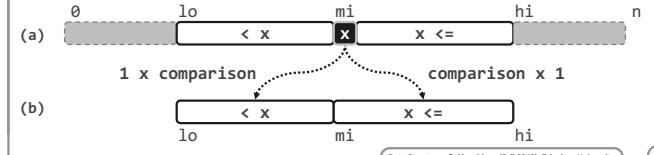
平均失败查找长度 =  $(4+5+4+4+5+4+5+4)/8 = 35/8 = 4.38$



Data Structures & Algorithms (Fall 2012), Tsinghua University

## 二分查找(版本B)：实现

```
❖ template <typename T>
static Rank binSearch(T* A, T const& e, Rank lo, Rank hi) {
 while (1 < hi - lo) {
 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; //若[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
}
```



Data Structures & Algorithms (Fall 2012), Tsinghua University

16

## 二分查找(版本B)：性能及进一步要求

❖ 有效查找区间宽度缩短至1时，算法才会终止

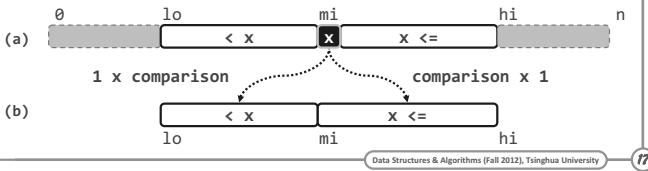
最好情况下效率不如版本A；作为补偿，最坏情况下效率有所提高  
各种情况下的查找长度更加接近，整体性能更趋稳定

❖ 以上二分查找及Fibonacci查找算法

并未严格兑现search(e)接口“返回不大于e且秩最大的元素”的要求

1) 当有多个命中元素时，必须返回最靠后者

2) 失败时，应返回小于(亦是不大于)e的最大者



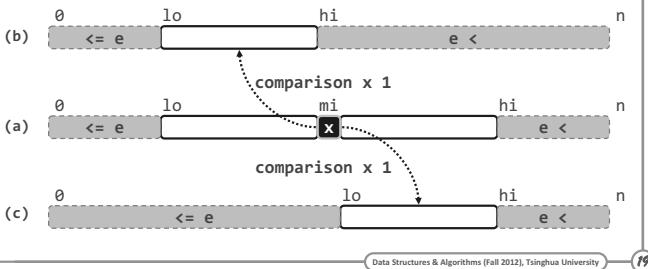
## 二分查找(版本C)：正确性

❖ 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…



## 插值查找：实例

❖  $e=50$ 

|   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 5 | 10 | 12 | 14 | 26 | 31 | 38 | 39 | 42 | 46 | 49 | 51 | 54 | 59 | 72 | 79 | 82 | 86 | 92 |
| 0 | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |

❖  $lo=0, hi=18$  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)

Data Structures & Algorithms (Fall 2012), Tsinghua University 21

## 插值查找：性能

❖ 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范围

然后再进行二分查找

Data Structures & Algorithms (Fall 2012), Tsinghua University 22

## 二分查找(版本C)：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
 if (e < A[mi]) hi = mi; else lo = mi + 1; // [lo, mi]或[mi+1, hi]
 }
 return --lo; //故lo-1即不大于e的元素的最大秩
}
```

❖ 与版本B的差异

1) 待查找区间宽度缩短至0而非1时，算法才结束

2) 转入右侧子向量时，左边界取作 $mi+1$ 而非 $mi$  —  $A[mi]$ 被遗漏？

3) 无论成功与否，统一返回 $lo-1$

Data Structures & Algorithms (Fall 2012), Tsinghua University 18

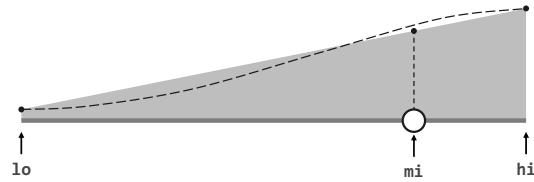
## 插值查找

❖ 假设：有序向量中各元素均匀且独立随机分布

❖ 于是， $[lo, hi]$ 内各元素大致按照线性趋势增长

$$(mi-lo)/(e-A[lo]) \approx (hi-lo)/(A[hi]-A[lo]) \\ mi \approx lo + (hi-lo)*(e-A[lo])/(A[hi]-A[lo])$$

❖ 以查阅英文词典为例，binary肯定非常靠前，search则非常靠后



❖ 因此，将轴点mi取作上述估计值，将极大地提高查找的收敛速度

Data Structures & Algorithms (Fall 2012), Tsinghua University 20

## 插值查找：性能

❖ 最坏情况： $\Theta(hi - lo) = \Theta(n)$

//具体实例？

❖ 平均情况：每经一次比较，n缩至 $\sqrt{n}$

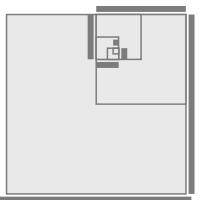
//[Yao76]

❖ 于是，待查找区间宽度将按以下趋势缩减：

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

❖ 经多少次比较之后，有 $n^{(1/2)^k} < 2$ ？

$$(1/2)^k \times \log n < 1 \\ k \log(1/2) + \log \log n < 0 \\ k > \log \log n \\ \therefore \Theta(\log \log n)$$



Data Structures & Algorithms (Fall 2012), Tsinghua University 22

## 课后

❖ fibSearch()的内层while循环，至多能够连续执行几次？

❖ 试证明

1) binSearch()版本A的平均失败查找长度为 $\Theta(1.50 * \log(n+1))$

2) 即便在最坏情况下，Fibonacci查找也只需 $\Theta(\log n)$ 时间

3) 关键码均匀独立分布时，插值查找中子问题规模按平方根速度递减

❖ 改进本节所给的实现，使Fibonacci查找严格符合search()接口

Data Structures & Algorithms (Fall 2012), Tsinghua University 24

## 2. 向量

### (e) 起泡排序

邓俊辉

deng@tsinghua.edu.cn

### 起泡排序

```
◆ template <typename T> //0 <= lo < hi <= size
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) { //一趟扫描交换
 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]); //通过交换使局部有序
 }
 return sorted; //返回有序标志
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

## 2. 向量

### (f) 归并排序

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

- T. J. Watson, 1943

邓俊辉

天下大势，分久必合，合久必分

deng@tsinghua.edu.cn

### 归并排序：实现

```
◆ 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
}
```



Data Structures & Algorithms (Fall 2012), Tsinghua University

### 排序器：统一入口

```
◆ template <typename T>
void Vector<T>::sort(Rank lo, Rank hi) { //区间[lo, hi)
 switch (rand() % 4) { //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 : mergeSort(lo, hi); break; //归并排序
 case 3 : heapSort(lo, hi); break; //堆排序 (稍后)
 default: quickSort(lo, hi); break; //快速排序 (稍后)
 }
}
```

### 综合评价

- ◆ 效率与第一章针对整数数组的版本相同，最好 $O(n)$ ，最坏 $O(n^2)$
- ◆ 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, 7<sub>a</sub>, 3, 2, 7<sub>b</sub>, 1, 5, 8, 7<sub>c</sub>, 4
  - 输出： 1, 2, 3, 4, 5, 6, 7<sub>a</sub>, 7<sub>b</sub>, 7<sub>c</sub>, 8 //stable
  - 1, 2, 3, 4, 5, 6, 7<sub>a</sub>, 7<sub>c</sub>, 7<sub>b</sub>, 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？

Data Structures & Algorithms (Fall 2012), Tsinghua University

## 2. 向量

### (f) 归并排序

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

- T. J. Watson, 1943

邓俊辉

天下大势，分久必合，合久必分

deng@tsinghua.edu.cn

### 归并排序：原理

◆ //分治策略  
//向量与列表通用  
//J. von Neumann, 1945

序列一分为二 //O(1)  
子序列递归排序 //2xT(n/2)  
合并有序子序列 //O(n)

◆ 例1：mergesort  
例2：xSortLab

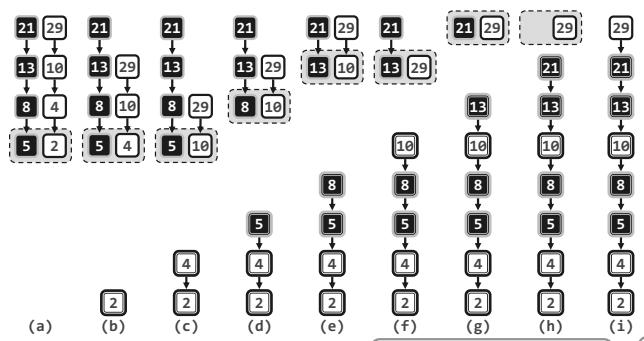
◆ 若真能如此，归并排序的运行时间应该是 $O(n \log n)$

Data Structures & Algorithms (Fall 2012), Tsinghua University

### 二路归并：原理

- ◆ 2-way merge：将两个有序序列合并为一个有序序列

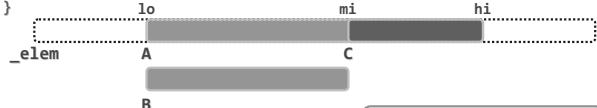
$$A[p, r) = A[p, q) + A[q, r)$$



Data Structures & Algorithms (Fall 2012), Tsinghua University

## 二路归并：实现

```
template <typename T> void Vector<T>::merge(Rank lo, Rank mi, Rank hi) {
 T* A = _elem + lo; //A[0..hi-lo] = _elem[lo..hi]
 int lb = mi - lo; T* B = new T[lb]; //B[0..lb] = A[lo..mi)
 for (Rank i = 0; i < lb; B[i] = A[i++]); //复制B
 int lc = hi - mi; T* C = _elem + mi; //C[0..lc) = A[mi..hi)
 Rank i = 0, j = 0, k = 0;
 while ((j < lb) && (k < lc)) { //反复比较B和C的首元素，将小者续至A末尾
 while ((j < lb) && B[j] <= C[k]) A[i++] = B[j++];
 while ((k < lc) && C[k] <= B[j]) A[i++] = C[k++];
 } //出口时，B或C为空，即j = lb或k = lc
 while (j < lb) A[i++] = B[j++]; //B非空时须将剩余部分续至A末尾—C非空呢？
 delete [] B; //释放临时空间B
}
```



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

4

## 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 — PageRank

易于并行化

## 缺点

非就地，需要对等规模的辅助空间

即使在最好情况（已经有序或接近有序），仍需 $\Theta(n \log n)$ 时间

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

6

## 3. 列表

## (a) 接口与实现

邓俊辉

Don't lose the link.

- Robin Milner (1934~2010)

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将不变：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，数据元素的存储于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

逻辑上相邻的元素记录彼此的物理地址，形成一个整体

可支持高效的动态操作

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

7

## 从秩到位置

## 向量支持循秩访问 (call-by-rank) 的方式

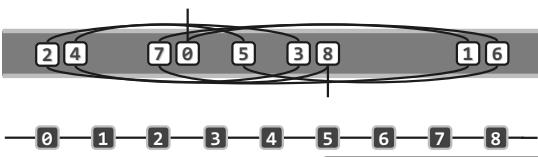
根据数据元素的秩，可在 $\Theta(1)$ 时间内直接确定其物理地址

$V[i]$ 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

则门牌号为i的住户的地理位置 =  $V + i \times s$

## 这种高效的方式，可否被列表沿用？



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

8

## 二路归并：复杂度

◆ 算法的运行时间主要消耗于两重while循环，莫非是平方量级？

◆ 观察其两个while内循环：每做一次迭代，i必加一，j + k也必加一

◆ 既然： $j + k < lb + lc = hi - lo = n$

故知：总体迭代不过 $O(n)$ 次，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B和C均已有序

◆ 注意：待归并子序列不必等长，即允许 $lb \neq lc$ ,  $mi \neq (lo+hi)/2$

◆ 实际上，这一算法及结论也适用于另一类序列 — 列表（下一章）



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

5

## 3. 列表

## (a) 接口与实现

邓俊辉

Don't lose the link.

- Robin Milner (1934~2010)

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将不变：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，数据元素的存储于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

逻辑上相邻的元素记录彼此的物理地址，形成一个整体

可支持高效的动态操作

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

6

## 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)

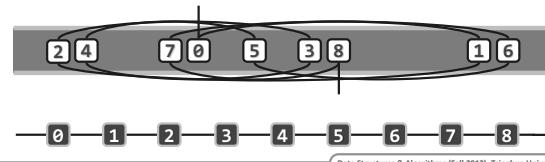
各节点通过指针或引用彼此联接，构成一个逻辑上的线性序列

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

## ◆ 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/后继的节点称作首 (first/front) /末 (last/rear) 节点



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

8

## 由秩到位置

## 由秩到位置

## ◆ 既然同属线性序列，列表固然也可通过秩找到对应的元素

为找到秩为1的元素，须从头（尾）端出发，沿引用前进（后退）1步

## ◆ 然而因为成本过高，此时的循秩访问已不合时宜

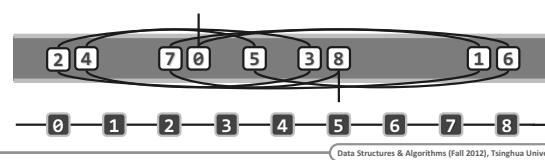
//下节详解

以平均分布为例，单次访问的期望复杂度为 $(n+1)/2 = \Theta(n)$

## ◆ 因此，应改用循位置访问 (call-by-position) 的方式访问列表元素

也就是说，应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...的同学Z



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

9

## 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                      |
|-----------------|-------------------------|
| pred()          | 当前节点前驱节点的位置             |
| succ()          | 当前节点后继节点的位置             |
| data()          | 当前节点所存数据对象              |
| insertAsPred(e) | 插入前驱节点，存入被引用对象e，返回新节点位置 |
| insertAsSucc(e) | 插入后继节点，存入被引用对象e，返回新节点位置 |

Data Structures & Algorithms (Fall 2012), Tsinghua University

5

## 列表：ADT接口

155

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

Data Structures & Algorithms (Fall 2012), Tsinghua University

7

## 构造

157

```
#include "listNode.h" //引入列表节点类
void List<T>::init() { //列表初始化，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
}
```

等效地，头、首、末、尾节点的秩可以分别理解为-1、0、n-1、n

Data Structures & Algorithms (Fall 2012), Tsinghua University

9

## 3. 列表

## (b) 无序列表

邓俊辉

deng@tsinghua.edu.cn

159

## 列表节点：ListNode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
};
```

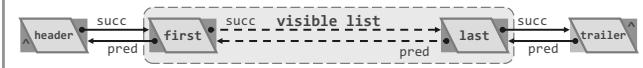


Data Structures & Algorithms (Fall 2012), Tsinghua University

156

## 列表：List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; Posi(T) header, trailer; //规模、头哨兵、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 */
};
```



Data Structures & Algorithms (Fall 2012), Tsinghua University

158

## 析构

```
template <typename T>
int List<T>::clear() { //清空列表
 int oldSize = _size;
 while (0 < _size) remove(header->succ); //反复删除首节点
 return oldSize; //直至列表变空
}

template <typename T>
List<T>::~List() { //列表析构器
 clear(); //清空列表
 delete header; delete trailer; //释放头、尾哨兵节点
}
```



Data Structures & Algorithms (Fall 2012), Tsinghua University

160

## 秩到位置

通过重载下标操作符，可仿照向量的方式循秩访问

```
template <typename T> //assert: 0 <= r < size
Posi(T) List<T>::operator[](int r) const { //O(r)
 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; //目标节点
}
```

时间复杂度为O(r)，线性正比于待访问节点的秩

以平均分布为例，单次访问的期望复杂度为

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

效率低下，不宜使用

Data Structures & Algorithms (Fall 2012), Tsinghua University

## 查找

```

❖ template <typename T> //顺序查找，复杂度O(n)
Posi(T) List<T>:: //在p的最近的n个前驱中查找e
find(T const& e, int n, Posi(T) p) const {
 while (0 < n--) //对于p的最近的n个前驱，从右向左逐个比对
 if (e == (p = p->pred)->data) return p; //查找成功
 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
L.find(e, n, p) ? cout << p->data : cout << "not found";

❖ Posi(T) find(T const& e) const //重载全局查找接口
{ return find(e, _size, trailer); }

```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

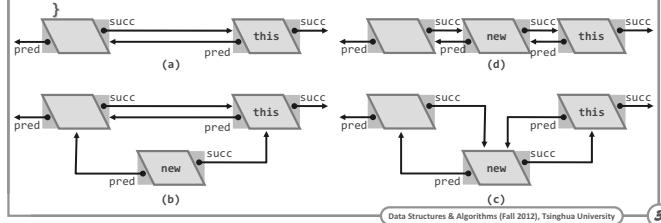
## 插入

```

❖ template <typename T>
ListNodePosi(T) List<T>::insertBefore(ListNodePosi(T) p, T const& e)
{ _size++; return p->insertAsPred(e); } //e当作p的前驱插入

❖ 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
}

```



## 基于复制的构造

```

❖ 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); }

```

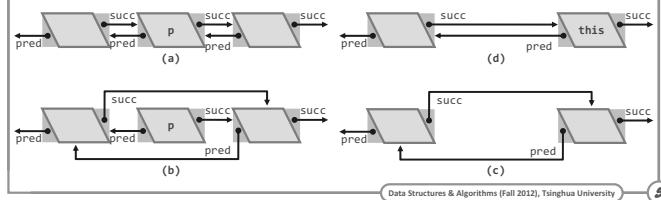
Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 删除

```

❖ 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
}

```



## 析构

```

❖ template <typename T>
List<T>::~List() //列表析构器：清空列表，释放头、尾哨兵
{ clear(); delete header; delete trailer; }

❖ template <typename T>
int List<T>::clear() { //清空列表：反复删除首节点，直至表空
 int oldSize = _size;
 while (0 < _size) 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？

```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 唯一化

```

❖ 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); //在前驱中查找雷同者
 q ? remove(q) : r++; //若的确存在，则删除之；否则秩加一
 } //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

## 遍历

```

❖ 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);
}

```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 3. 列表

## (c) 有序列表

邓俊辉

deng@tsinghua.edu.cn

## 唯一化

```

❖ template <typename T>
int List<T>::unique() { //或批剔除重复元素，效率更高
 if (_size < 2) return 0; //平凡列表自然无重复
 int oldSize = _size; //记录原规模
 Posi(T) q = first(); //从首节点出发
 while (trailer != q->succ) { //只要q合法
 Posi(T) p = q; q = p->succ; //每对紧邻节点
 if (p->data == q->data) //若雷同
 { remove(q); q = p; } //则删除后者
 }
 return oldSize - _size; //规模变化量，即被删除元素总数
}

```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 查找

- ❖ 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
- ❖ 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
- ❖ 始终返回不大于e的最大元素，如此可便于插入排序等后续操作  
得益于哨兵的存在，后续处理大为简化

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 3. 列表

## (d) 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

## 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 |

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 实现：selectMax()

```

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

```

❖ 在有重复元素时，采用比较器“!lt()”或“ge()”等效于“后者优先”  
如此，可保证上层选择排序算法的稳定性

|   |    |   |    |   |   |    |   |    |   |   |   |
|---|----|---|----|---|---|----|---|----|---|---|---|
| 2 | 6a | 4 | 6b | 3 | 0 | 6c | 1 | 5  | 7 | 8 | 9 |
| 2 | 6a | 4 | 6b | 3 | 0 | 1  | 5 | 6c | 7 | 8 | 9 |

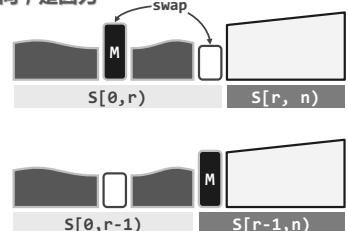
Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

- ❖ 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
- ❖ 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
- ❖ 始终返回不大于e的最大元素，如此可便于插入排序等后续操作  
得益于哨兵的存在，后续处理大为简化

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 构思

- ❖ 回忆起泡排序 ...
- ❖ 每经一趟扫描交换，当前的最大元素必然就位 //其效果也等效于...  
从未排序元素中挑选出最大元素，并使之就位 //选择排序！不过...
- ❖ 起泡排序之所以需要O( $n^2$ )时间，是因为  
为挑选每个最大元素M，做了  
O(n)次比较和O(n)次交换
- ❖ O(n)次比较无可厚非，但  
O(n)次交换没有必要
- ❖ 经O(n)次比较即可确定M  
此后，一次交换足矣



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 实现：selectionSort()

```

//通用版本：对列表中起始于位置p的连续n个元素做选择排序
template <typename T> //valid(p) && rank(p) + n <= size
void List<T>::selectionSort(Posi(T) p, int n) {
 //待排序区间为(head, tail)——因设有头、尾哨兵，可简捷而统一地表示
 Posi(T) head = p->pred; Posi(T) tail = p;
 for (int i = 0; i < n; i++) tail = tail->succ;
 while (1 < n) { //在至少还剩两个节点之前，在待排序区间内
 //找出最大者，将其移至有序区间前端
 insertBefore(tail, remove(selectMax(head->succ, n)));
 //待排序区间、有序排序区间的范围、宽度同步更新
 tail = tail->pred; n--;
 }
}

```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 性能

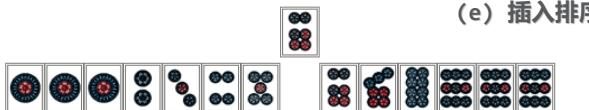
- ❖ 共迭代n次，第k次迭代中  
selectMax()为Θ(n-k)  
remove()和insertBefore()均为O(1)  
故总体复杂度应为Θ(n<sup>2</sup>)
- ❖ 尽管如此，（实际更费时的）元素移动操作远远少于起泡排序  
也就是说，Θ(n<sup>2</sup>)主要来自于元素的数值比较操作
- ❖ 可否 ... 每轮只做O(n)次比较，即找出当前的最大元素？
- ❖ 可以！  
利用高级数据结构，selectMax()可改进至O(logn) //稍后分解
- ❖ 当然，如此立即可以得到O(nlogn)的排序算法 //保持兴趣

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

5

## 3. 列表

## (e) 插入排序



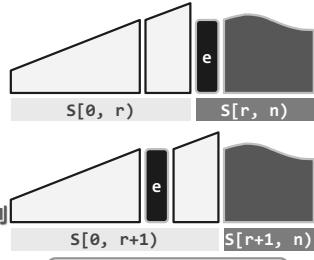
花荣便道：“前面必有强人。”把枪  
带住，取弓箭来整顿得端正，再插放  
飞鱼袋内。

邓俊辉

deng@tsinghua.edu.cn

## 构思

- 将输入看成两部分：已排序 + 未排序 =  $S_r + S \setminus S_r$  //  $|S_r| = r$
- 【初始化】置  $S_0$  为（长度为零的）空序列 // 空序列自然有序
- 【迭代】在有序的  $S_r = S[0, r]$  中确定适当位置 // 有序序列的查找  
插入  $S[r]$ ，得到有序的  $S_{r+1} = S[0, r+1]$  // 有序序列的插入
- 如此，可逐步得到：  
 $S_0, S_1, \dots, S_n$   
最终， $S_n = S[0, n]$  即排序序列
- 正确性基于以下不变性：  
 $\forall 0 \leq r \leq n$ ,  
 $S_r$  = 前  $r$  个元素组成的有序序列



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 实例

## 实现

| 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 | ^    | ^             |

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

//通用版本：对列表中起始于位置p的连续n个元素做插入排序

```
template <typename T> //valid(p) && rank(p) + n <= size
void List<T>::insertionSort(ListNodePosi(T) p, int n) {
 for (int r = 0; r < n; r++) { //逐一引入各节点，由S_r得到S_{r+1}
 insertAfter(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)
} //除输入列表，仅使用O(1)辅助空间，故属于就地算法
```

◆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

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 平均性能

## 3. 列表

- 若各元素的数值均匀、独立分布，平均要做多少次元素比较？
- 考查： $S[r]$ 刚插入到  $S[0, r]$  后的那一时刻  
试问：在此时已有序的  $S[0, r]$  中，哪个元素是原先的  $S[r]$ ？
- 观察：其中的  $r+1$  个元素都有可能，且概率均等于  $1/(r+1)$
- 因此，在刚完成的这次迭代中，为引入  $S[r]$  所花费时间的期望值量为  

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

$$= r/2 + 1$$
- 于是，总体时间的期望值为  

$$[0 + 1 + \dots + (n-1)] / 2 + 1 = O(n^2)$$
- 改用向量，查找效率可优化至  $O(\log k)$ ，但总体性能会否相应地提高？

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 归并排序

## (f) 归并排序

四牡孔阜，六辔在手  
骐骥是中，騄駠是骖  
龙盾之合，鋈以觶韁

邓俊辉

deng@tsinghua.edu.cn

List p q  
lo mi hi  
= = =  
lo + n/2 lo + n

//对起始于位置p的n个元素排序

```
template <typename T> //valid(p) && rank(p) + n <= size
void List<T>::mergeSort(Posi(T)& p, int n) {
 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，则总体运行时间为O(n log n)
```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 二路归并

List p q  
lo mi hi  
= = =  
lo + n/2 lo + n

//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（归并排序时为同一列表）

```
template <typename T> //O(n+m)
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--; } //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
 { insertBefore(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
```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

### 3. 列表

#### (x1) 游标实现

邓俊辉

deng@tsinghua.edu.cn

#### 动机与构思

◆ 有些语言或者不支持指针，或者不能动态申请空间

此时，如何实现列表结构呢？

◆ 可以游标方式实现

每个列表占用固定量、地址连续的数据空间，故亦称作“静态链表”

内部维护两个列表，共用同一地址空间

list：存放实际节点，外部可见

space：记录可分配节点，仅供内部使用

◆ 两个表头节点，应如何确定？

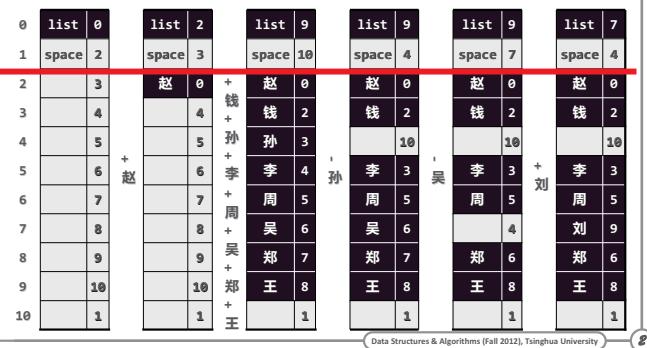
当列表发生变化（比如有节点插入或删除）时，应如何调整？

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

#### 原理与实例

◆ 可理解为，将整个空间L[]组织为有两条链表：

- 1) 空余单元表space：头结点L[1]，首结点L[1].next，1指示表尾
- 2) 数据单元表list：头结点L[0]，首结点L[0].next，0指示表尾



#### (x2) Java序列

### 3. 列表

#### 3. 列表

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

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

#### Interface : 实现

```
class Disk implements Geometry { //符合Geometry接口的圆盘类
 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;
 }
}
```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

#### 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;
}
```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

#### 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("越界");
 if (n >= N) throw new ExceptionBoundaryViolation("溢出");
 for (int i = n; i > r; i--) A[i] = A[i-1];
 A[r] = obj; n++; return obj;
 }
 /* */
}
```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

向量实现2 : Vector\_ExtArray.java

```

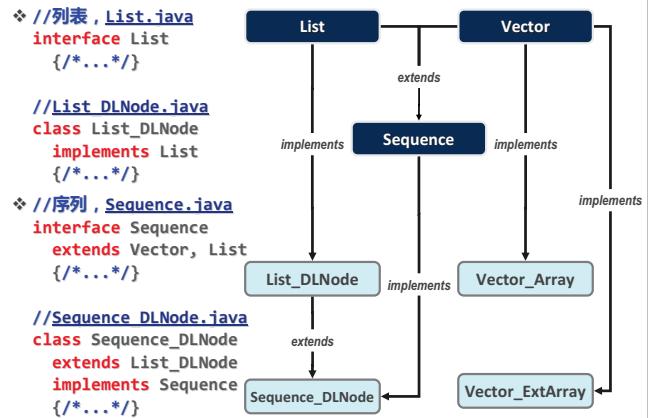
public class Vector_ExtArray implements Vector {
 private int N = 8; //数组的初始容量，可不断增加
 /* */
 public Object insertAtRank(int r, Object obj)
 throws ExceptionBoundaryViolation {
 if (0 > r || r > n)
 throw new ExceptionBoundaryViolation("越界");
 if (N<=n) { //空间溢出的处理
 N *= 2; Object B[] = new Object[N]; //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;
 }
 /* */
}

```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

5

## 序列接口及实现



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

6

## 3. 列表

## (x3) Python列表

邓俊辉

deng@tsinghua.edu.cn

## 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

```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

8

## reverse() : 循位置访问 ? 循秩访问 ?

```

❖ def reverse_1(L):
 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):
 i, j = 0, len(L)-1 # 从首、末元素开始
 while i < j: # 对于每对对称的元素L[i]及L[n-1-i]
 L[i], L[j] = L[j], L[i] # 交换之, 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?

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

9

## 4. 栈与队列

## (a) 栈接口与实现

邓俊辉

deng@tsinghua.edu.cn

## Python List

## 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

```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

9

## 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

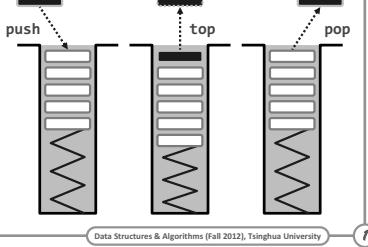
`pop()` 出栈

`top()` 查顶

◆ 后进先出 (LIFO)

先进后出 (FILO)

◆ 扩展接口 : `getMax()` ...



Data Structures & Algorithms (Fall 2012), Tsinghua University

## 模板类

◆ 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？
```

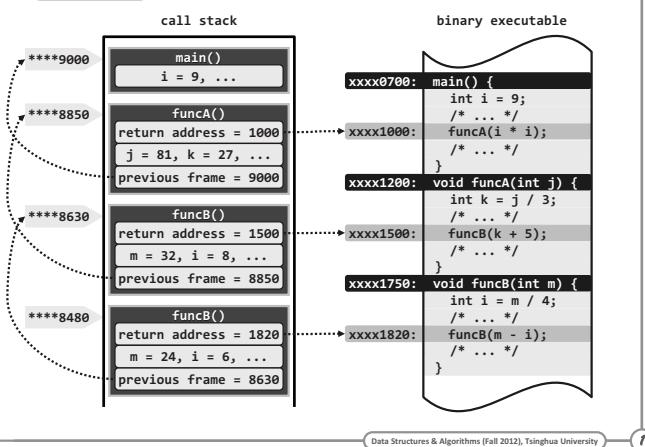
◆ 确认：如此实现的栈接口，均只需O(1)时间

◆ 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？

Data Structures & Algorithms (Fall 2012), Tsinghua University

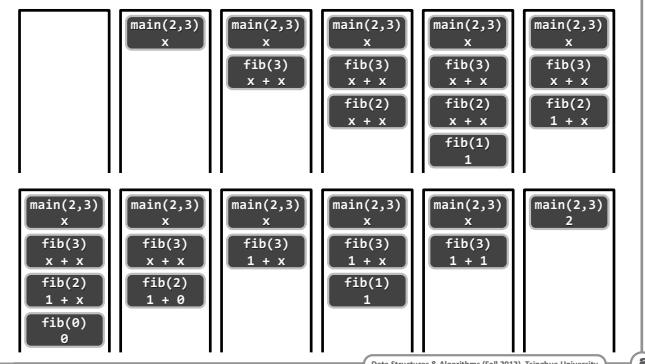
## 函数调用栈



Data Structures & Algorithms (Fall 2012), Tsinghua University

## 实例 : fib()

```
◆ int fib(int n) { return (2 > n) ? n : fib(n-1) + fib(n-2); }
main(int argc, char* argv[]) { fib(atoi(argv[1])); }
```



Data Structures & Algorithms (Fall 2012), Tsinghua University

3

## 操作实例

| 操作      | 输出    | 栈 (左侧为栈顶) |
|---------|-------|-----------|
| Stack() |       |           |
| empty() | true  |           |
| push(5) |       | 5         |
| push(3) |       | 3 5       |
| pop()   | 3     | 5         |
| push(7) |       | 7 5       |
| push(3) |       | 3 7 5     |
| top()   | 3     | 3 7 5     |
| empty() | false | 3 7 5     |

| 操作       | 输出    | 栈 (左侧为栈顶)    |
|----------|-------|--------------|
| push(11) |       | 11 3 7 5     |
| size()   | 4     | 11 3 7 5     |
| push(6)  |       | 6 11 3 7 5   |
| empty()  | false | 6 11 3 7 5   |
| push(7)  |       | 7 6 11 3 7 5 |
| pop()    | 7     | 6 11 3 7 5   |
| pop()    | 6     | 11 3 7 5     |
| top()    | 11    | 11 3 7 5     |
| size()   | 4     | 11 3 7 5     |

Data Structures & Algorithms (Fall 2012), Tsinghua University

## 4. 栈与队列

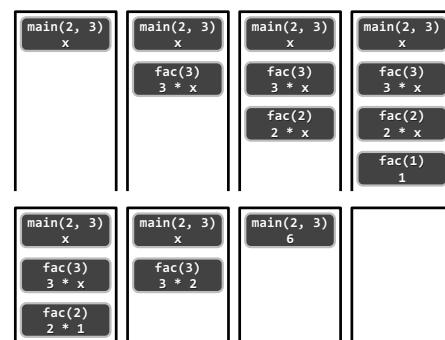
## (b) 栈与递归

邓俊辉

deng@tsinghua.edu.cn

## 实例 : fac()

```
◆ int fac(int n) { return (2 > n) ? 1 : n*fac(n-1); }
main(int argc, char* argv[]) { fac(atoi(argv[1])); }
```



Data Structures & Algorithms (Fall 2012), Tsinghua University

## 实例 : hailstone()

```
◆ 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    | call stack     |
|---------------|----------------|
| main(2, 10)   | main(2, 27)    |
| hailstone(10) | hailstone(27)  |
| even(10)      | odd(27)        |
| hailstone(5)  | hailstone(82)  |
| odd(5)        | even(82)       |
| hailstone(16) | hailstone(41)  |
| even(16)      | odd(41)        |
| hailstone(8)  | hailstone(124) |
| even(8)       | odd(124)       |
| hailstone(4)  | hailstone(62)  |
| even(4)       | odd(62)        |
| hailstone(2)  | hailstone(31)  |
| even(2)       | odd(31)        |
| hailstone(1)  | hailstone(94)  |
|               | ...            |

Data Structures & Algorithms (Fall 2012), Tsinghua University

4

## 避免递归

- ◆ 动机：递归函数的空间复杂度，主要取决于**最大递归深度**  
为了**隐式地**维护调用栈，也需花费额外的处理时间
- ◆ 方法：将递归算法改写为迭代版本
- ◆ int fac(int n)
 

```
{ int f = 1; while (n > 1) f *= n--; return f; }
```
- ◆ int fib(int n)
 

```
int f = 1, g = 0; while (0 < n--) { f = f + g; g = f - g; }
```
- ◆ void hailstone(int n)
 

```
{ while (1 < n) n = (n % 2) ? (3*n + 1) : (n / 2); }
```
- ◆ 更为复杂的算法，其迭代版本往往需要**显式地**维护一个栈 //第五章

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

5

## 4. 栈与队列

## (c1) 栈应用：进制转换

Hickory, Dickory, Dock  
The mouse ran up the clock  
- Nursery Rhyme Medley

邓俊辉

deng@tsinghua.edu.cn

## 典型应用场合

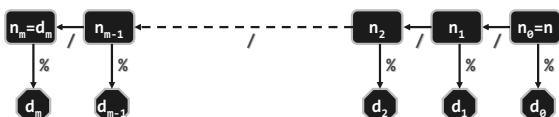
- ◆ 以下数节，分别结合实际问题介绍栈的几种典型应用场合
- ◆ 逆序输出 conversion  
输出次序与处理过程**颠倒**，递归深度和输出长度不易预知
- ◆ 递归嵌套 permutation + parenthesis  
具有自相似性的问题可递归描述，但分支位置和嵌套深度不固定  
栈操作天然具有递归嵌套性，可用以有效控制此类算法的复杂度
- ◆ 延迟缓冲 evaluation  
线性扫描算法模式中，在预读足够长之后，方能确定可处理的前缀
- ◆ 栈式算法 RPN  
基于栈结构的特定计算模式

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

211

## 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  $d_i = n_i \% \lambda$  //初始  $n_0 = n$
- ◆ 构思：n对λ反复取模、整除，即可**自低到高**得出λ进制的各位



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

213

## 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
 '0', '1', '2', '3', '4', '5', '6', '7',
 '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'
 }; //可视base取值范围适当扩充
 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（对应的数位）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
}
```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

5

## 4. 栈与队列

## (c2) 栈应用：括号匹配

邓俊辉

deng@tsinghua.edu.cn

212

## 问题

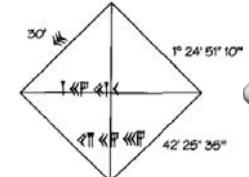
- ◆ 描述：给定任一10进制非负整数，将其转换为λ进制表示形式

```
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...

- ◆ 长方形的对角线

$$\begin{aligned} 1^{\circ}24'51''10''' \\ = 1 + 24/60 + 51/60^2 + 10/60^3 \\ = 1.41421296296296\dots \end{aligned}$$



- ◆ 误差

$$|1^{\circ}24'51''10''' - \sqrt{2}| < 0.0000006 = 0.6 \times 10^{-6}$$

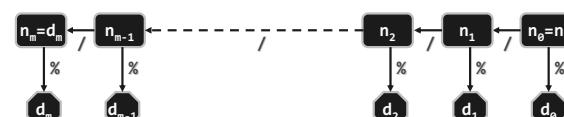
即便边长为1km，误差亦不足1mm

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

214

## 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，则扩充策略必须得当  
若使用列表，则多数接口均被闲置
- ◆ 使用栈，即可满足以上要求，亦可有效控制计算成本



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

216

## 算法实现

215

```
◆ void convert(Stack<char>& S, __int64 n, int base) {
 static char digit[] = { //新进制下的数位符号
 '0', '1', '2', '3', '4', '5', '6', '7',
 '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'
 }; //可视base取值范围适当扩充
 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（对应的数位）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
}
```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

5

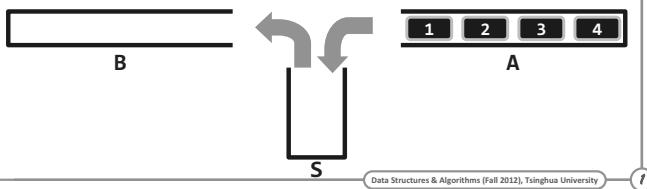
## 4. 栈与队列

## (c2) 栈应用：括号匹配

邓俊辉

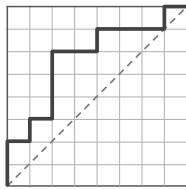
deng@tsinghua.edu.cn

- 考查栈A = { $a_1, a_2, \dots, a_n$ }、B = S =  $\emptyset$  //左端为栈顶
- 只允许 将A的顶元素弹出并压入S，或 //S.push(A.pop())
   
将S的顶元素弹出并压入B //B.push(S.pop())
- 若在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)



## 栈混洗：计数

- 定理： $SP(n) = \text{Catalan}(n) = (2n)!/(n+1)!/n!$
- $SP(2) = 4! / 3! / 2! = 2$
- $SP(3) = 6! / 4! / 3! = 5$
- ...
- $SP(6) = 12! / 7! / 6! = 132$



Data Structures & Algorithms (Fall 2012), Tsinghua University

## 栈混洗：甄别

- 反过来，不存在“312”模式的序列，一定是栈混洗吗？
- 充要性 (Knuth, 1968)：  
 A permutation is a stack permutation iff  
 it does NOT involve the permutation 312
- 如此，可得一个 $O(n^3)$ 的甄别算法 //进一步地...
- { $p_1, p_2, p_3, \dots, p_n$ }是{1, 2, 3, ..., n}的栈混洗，当且仅当  
 对于任意*i* < *j*，不含模式{..., *j*+1, ..., *i*, ..., *j*, ...}
- 如此，可得一个 $O(n^2)$ 的甄别算法 //再进一步地...
- $O(n)$ 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，检测是否S已空，或需弹出的元素在S中却非顶元素

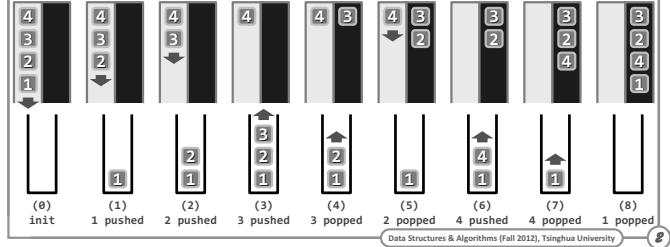
Data Structures & Algorithms (Fall 2012), Tsinghua University

## 括号匹配

- (a[i-1][j+1]) + a[i+1][j-1] \* 2 不匹配 ([ ] [ ] [ ] [ ] )  
 (a[i-1][j+1] + a[i+1][j-1]) \* 2 匹配 ([ ] [ ] [ ] [ ] )
- 观察：除各种括号，其它符号可暂忽略
- 递归描述：0) 无括号的表达式是匹配的
  - E匹配，仅当"(" + E + ")"匹配
  - E和F均匹配，仅当E + F匹配
- 然而，根据以上描述，不易得到高效的递归实现 //根源在于...
- 1) 和2) 仅为必要性  
 而为使问题有效降解，必须借助于充分性  
 反例："((())())" = "(" + "(()()" + ")" = "((()" + "())()"

Data Structures & Algorithms (Fall 2012), Tsinghua University

- 同一输入序列，可有多种栈混洗  
 $\{1, 2, 3, 4\}, \{4, 3, 2, 1\}, \{3, 2, 4, 1\}, \dots$
- 长度为n的序列，可能的混洗总数 $SP(n) = ?$
- 显然， $SP(n) < n!$  //更精确地...



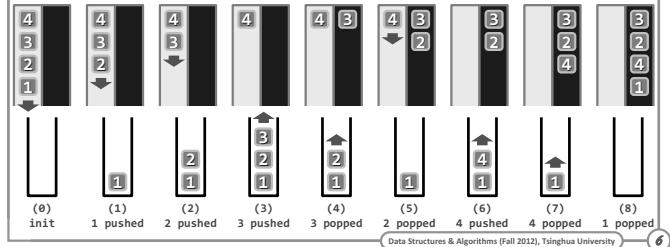
## 栈混洗：甄别

- 给定输入序列{1, 2, 3, ..., n}，对于任一排列 $\{p_1, p_2, p_3, \dots, p_n\}$ ，如何判定它是否栈混洗？
- 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  
 推而广之，可得栈混洗的必要条件...
- 不存在  $1 \leq i < j < k \leq n$ ，使得  
 栈混洗 $\{p_1, \dots, p_n\} = \{\dots, k, \dots, i, \dots, j, \dots\}$

Data Structures & Algorithms (Fall 2012), Tsinghua University

## 栈混洗：甄别

- 观察：每一栈混洗，都对应于栈S的n次push与n次pop操作构成的序列  
 $\text{push}(1) \text{ push}(2) \text{ push}(3) \text{ pop}(3) \text{ pop}(2) \text{ push}(4) \text{ pop}(4) \text{ pop}(1)$
- push/pop序列对应于输入/输出序列：{1, 2, 3, 4}/{3, 2, 4, 1}；反之不然
- 栈混洗的操作序列中，任一前缀所含push不少于pop；反之亦然



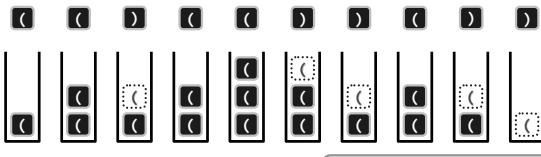
## 括号匹配：构思

- n个元素的栈混洗，等价于n对括号的匹配  
 为看出这一点，只需将混洗操作序列中的push/pop替换为左/右括号
- 颠倒以上思路可知：消去一对紧邻的左右括号，不影响全局的匹配判断  
 亦即： $S_L + "()" + S_R$  匹配，当且仅当  $S_L + S_R$  匹配
- 那么，如何找到这对括号？又，如何使问题的这种降解得以持续进行？
- 顺序扫描表达式，用栈记录已扫描的部分
- 若扫描至"时栈内为 $S_L + "("$ ，则弹出栈顶的"("，并继续扫描
- 实际上，栈中只需（只能）保留左括号

Data Structures & Algorithms (Fall 2012), Tsinghua University

## 括号匹配：实现

```
bool paren(const char exp[], int lo, int hi) {
 Stack<char> S; // 使用栈记录已发现但尚未匹配的左括号
 int i = -1;
 while (exp[++i] //逐一检查当前字符
 if ('(' == exp[i]) S.push(exp[i]); //左括号：则进栈
 else //右括号：栈已空，则不匹配；否则弹出栈顶（左括号）
 S.empty() ? return false : S.pop();
 return S.empty(); //最后，栈空当且仅当匹配
}
```



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

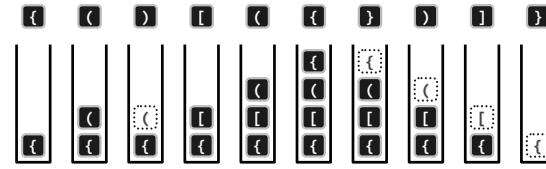
9

## 括号匹配：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

◆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数目  
比如，HTML文件中的tag标志：

```
<body> + </body>, <h1> + </h1>
<center> + </center>, <p> + </p>
 + , +
```



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

10

## 括号匹配：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 &amp; M. R. Sleep, 1980

◆由算法U，不是可以直接实现算法W的功能吗？后者存在的意义何在？

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

H

## 4. 栈与队列

## (c3) 栈应用：中缀表达式求值

邓俊辉

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 + 23) / 4 * 5 * 67 - 8 + 9
PostScript
GS> 0 1 23 add 4 div 5 mul 67 mul add 8 sub 9 add =
Excel:= 0 + (1 + 23) / 4 * 5 * 67 - 8 + 9
Word:= 0 + (1 + 23) / 4 * 5 * 67 - 8 + 9
calc:= 0 ! + (1 + 23) / 4 * 5 * 67 - 8 + 9 =
calc: (0 ! + 1) * 2 ^ (3 ! + 4) - 5 ! + 67 + 8 + 9 =
y
```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

I

## 表达式求值

◆表达式的求值，可视作（子）符号串与对应数值交替转换的过程

◆str(v)：数值v对应的（十进制）符号串

val(S)：符号串S对应的（十进制）数值

◆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

val(S)

=

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

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

E

## 延迟缓冲

◆难点：如何高效地找到可优先计算的 $S_\theta$ （亦即，其对应的运算符）？  
◆与括号匹配等应用不同，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 ) !

◆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...

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

B

## 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

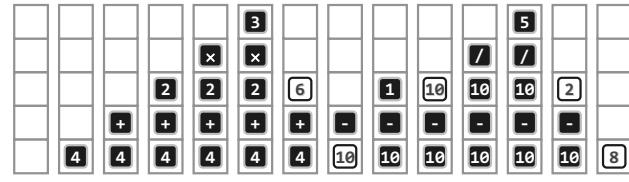
while (栈的顶部存在可优先计算的子表达式) //如何判断？

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 + 2 × 3 - 1 0 / 5 \0



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

C

## 实现：主算法

```
* float evaluate(char* S, char*& RPN) { //中缀表达式求值
 Stack<float> opnd; Stack<char> oprt; //运算数栈、运算符栈
 oprt.push('\0'); //尾哨兵'\0'也作为头哨兵首先入栈
 while (!oprt.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(oprt.top(), *S)) {/* 分别处理 */}
 } //while
 return opnd.pop(); //弹出最后的计算结果
}
```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

5

## 实现：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 */ '|', '<', '<', '<', '<', '<', '<', '<', '|', '='
};
```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

6

## 实现：不同优先级处理方法

```
* switch(orderBetween(oprt.top(), *S)) {
 case '<': //栈顶运算符优先级更低
 oprt.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')
 oprt.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 = oprt.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 { //二元运算符
 float pOpnd2 = opnd.pop(), pOpnd1 = opnd.pop();
 opnd.push(calcu(pOpnd1, op, pOpnd2));
 } //为何不直接：opnd.push(calcu(opnd.pop(), op, opnd.pop()));
 break;
 } //case '>'
} //switch
```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

7

```
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 */ '|', '<', '<', '<', '<', '<', '|', '|', '|', '='
};
```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

8

## 实例

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		运算符'^'入栈

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

9

## 实例

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		操作数9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

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

11

## 4. 栈与队列

## (c4) 栈应用：逆波兰表达式

郑俊辉

deng@tsinghua.edu.cn

## RPN

逆波兰表达式 ( Reverse Polish Notation )

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 )

Data Structures & Algorithms ( Fall 2012 ), Tsinghua University

1

## RPN求值：实例

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

630  
4230  
1880  
2004

Data Structures & Algorithms ( Fall 2012 ), Tsinghua University

2

## infix到postfix：手工转换

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

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

2)将运算符移到对应的右括号后

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

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...

Data Structures & Algorithms ( Fall 2012 ), Tsinghua University

3

## PostScript

诞生于 1985 , 支持设备独立的图形描述

一个解释器 + 五个栈 + 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

dictionary/execution/graphics state/clipping path stacks

实例：4 4 mul 5 5 mul add 7 mul 7 mul // 执行 | 编辑

提供基础且强大的图形功能，支持数据类型、变量、函数/宏 ...

newpath 300 600 80 22.5 -22.5 arc stroke



Data Structures & Algorithms ( Fall 2012 ), Tsinghua University

4

## 4. 栈与队列

## (d1) 试探回溯法：八皇后

```
/* D. Osvolanski & 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&&!l)?printf(2+"\\n\\n%c"-(
!l<<!j),".Q"[1~?((1~j)&1~2))&&+1||a[i]<s&&v>i+j&&v+i-
j))&&(1~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));}
```

邓俊辉

deng@tsinghua.edu.cn

## RPN求值：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;

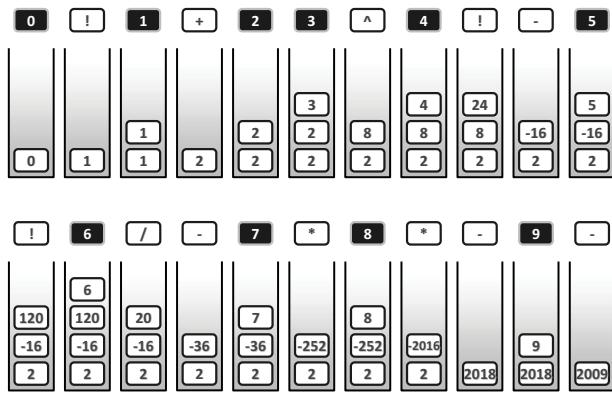
    } // else

} // while

返回栈顶； // 也是栈底

Data Structures & Algorithms ( Fall 2012 ), Tsinghua University

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



Data Structures & Algorithms ( Fall 2012 ), Tsinghua University

## 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, op); } // 读出并接入 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 '>'

/\* ..... \*/

Data Structures & Algorithms ( Fall 2012 ), Tsinghua University

6

## 4. 栈与队列

## (d1) 试探回溯法：八皇后

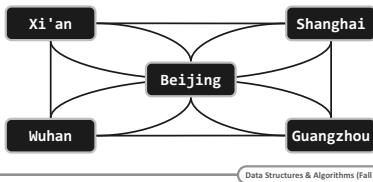
```
/* D. Osovanski & B. Nissenbaum, 1990 */
int v,i,j,k,l,s,a[99];void main(){for(scanf("%d",&s);*a-s;
v=a[j=v]-a[i],k=i<s,j+=(v=j<s&&(!k&&!l)?printf(2+"\\n\\n%c"-(
!l<<!j),".Q"[1~?((1~j)&1~2))&&+1||a[i]<s&&v>i+j&&v+i-
j))&&(1~s),v||(i==j)?a[i+=k]=0:+a[i]]>=s*k&&++a[-i]);}

/* 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));}
```

邓俊辉

deng@tsinghua.edu.cn

- 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

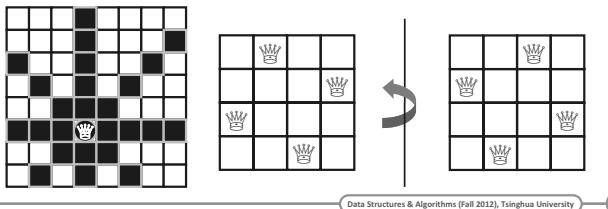


Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 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？

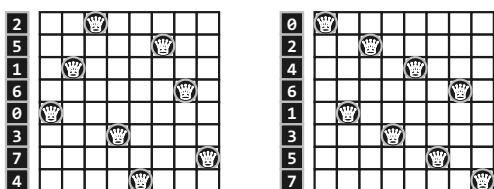
- $n = 1, 2, 3, 4, \dots$   
允许重复： 1, 0, 0, 2, 10, 4, 40, 92, ...  
不许重复： 1, 0, 0, 1, 2, 1, 6, 12, 46, 92, ...



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 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）



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 剪枝

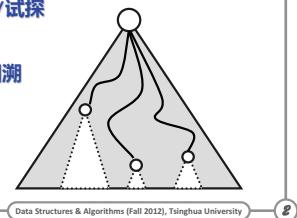
- ```

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
}

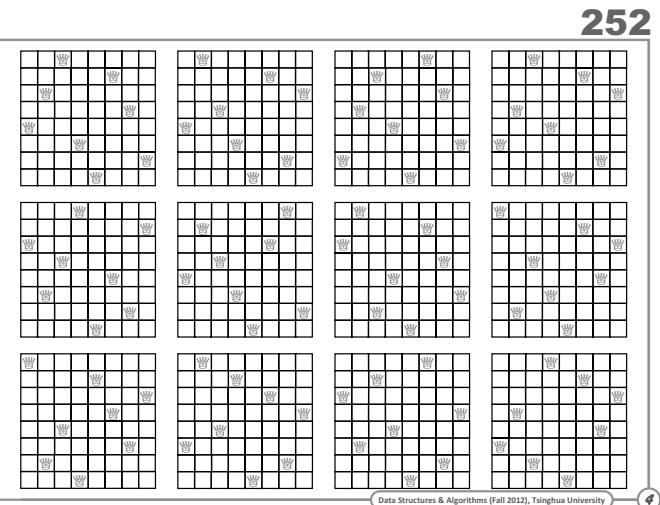
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

- 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？



Data Structures & Algorithms (Fall 2012), Tsinghua University



Data Structures & Algorithms (Fall 2012), Tsinghua University

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)

```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 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，并继续试探下一行
        } 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
                solu.push(q); //摆上当前皇后，并
                if (N <= solu.size()) nSolu++; //若部分解已成为全局解，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
}

```

Data Structures & Algorithms (Fall 2012), Tsinghua University

数据结构

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

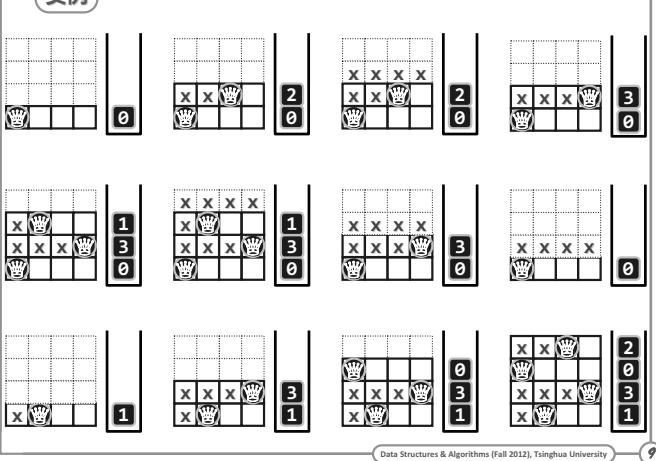
❖ 定义皇后类Queen，重新定义判等器，使之在语义上与“冲突”等价

❖ class Queen { //皇后类

```
public:
    int x, y; //皇后在棋盘上的位置坐标
    Queen(int xx = 0, int yy = 0) : x(xx), y(yy) {}
    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); }
};
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

10



259

4. 栈与队列

(d2) 试探回溯法：迷宫寻径

When all else fails, try brute-force.

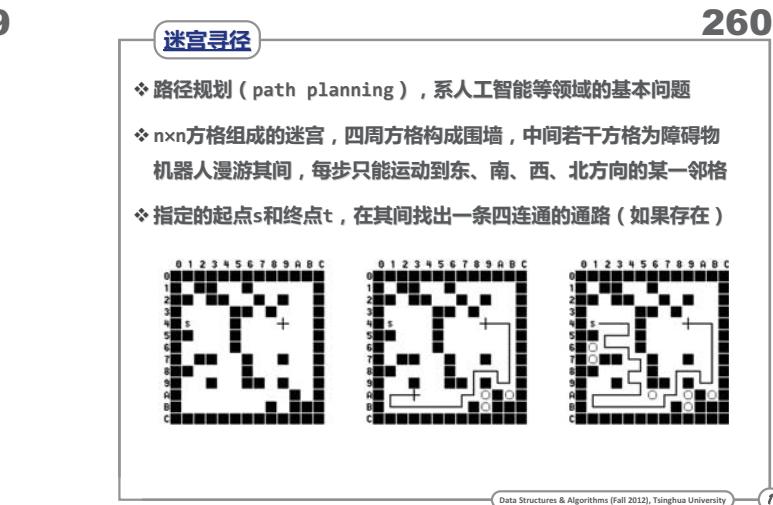
- Anonymous

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

- "No Matter What", Boyzone

邓俊辉

deng@tsinghua.edu.cn



Data Structures & Algorithms (Fall 2012), Tsinghua University

11

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;
        } //else
    } while (!path.empty());
    return false;
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

261

数据结构

❖ class Cell { //迷宫单元

```
public:
    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); }
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

12

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？

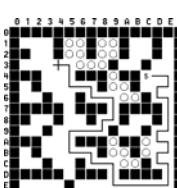
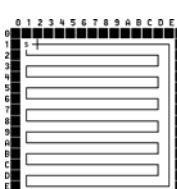
环路：尽可能发现并消去

弯路：尽可能发现并消去

贪心：终点方向优先试探

...

❖ 通用算法



Data Structures & Algorithms (Fall 2012), Tsinghua University

263

4. 栈与队列

(e) 队列接口与实现

邓俊辉

deng@tsinghua.edu.cn

264

操作与接口

❖ 队列 (queue) 也是受限的序列

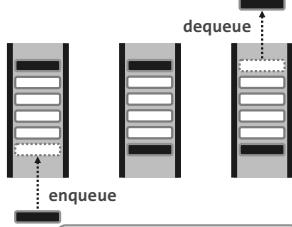
只能在队尾 (rear) 插入 enqueue()

只能在队头 (front) 删除和查询 dequeue() / front()

❖ 先进先出 (FIFO)

后进后出 (LILO)

❖ 扩展接口 : getMax() ...



Data Structures & Algorithms (Fall 2012), Tsinghua University

1

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 | 3 | |
| empty() | false | 6 11 | 3 | 7 | 3 | |
| enqueue(7) | | 7 6 11 | 3 | 7 | 3 | |
| dequeue() | 3 | 7 6 11 | 3 | 7 | | |
| dequeue() | 7 | 7 6 11 | 3 | | | |
| front() | 3 | 7 6 11 | 3 | | | |
| size() | 4 | 7 6 11 | 3 | | | |

Data Structures & Algorithms (Fall 2012), Tsinghua University

268

4. 栈与队列

(f) 队列应用

邓俊辉

deng@tsinghua.edu.cn

模板类

❖ 队列既然属于序列的特例，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？

Data Structures & Algorithms (Fall 2012), Tsinghua University

3

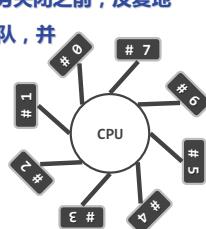
资源循环分配

❖ 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，车轮大战

❖ RoundRobin { //循环分配器

```
Queue Q(clients); //参与资源分配的所有客户组成队列Q
while (!ServiceClosed()) { //在服务关闭之前，反复地
    e = Q.dequeue(); //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

Data Structures & Algorithms (Fall 2012), Tsinghua University

1

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; //有一位新顾客到达
        c.time = 1 + rand() % 50; //随机指定新顾客的服务时长1~50
        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
    } //while
    delete [] windows; //释放所有队列（是否需要显式地调用clear()分别清空？）
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

3

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）

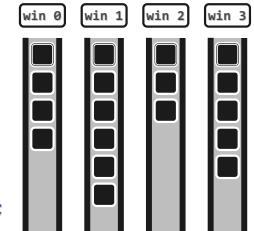
❖ 参数：nWin //窗口（队列）数目

servTime //营业时长

❖ class Customer { //顾客类

public:

```
    int window; //所属窗口（队列）
    unsigned int time; //服务时长
};
```



Data Structures & Algorithms (Fall 2012), Tsinghua University

270

4. 栈与队列

(x) Steap + Queap

邓俊辉

deng@tsinghua.edu.cn

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; //有一位新顾客到达
        c.time = 1 + rand() % 50; //随机指定新顾客的服务时长1~50
        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
    } //while
    delete [] windows; //释放所有队列（是否需要显式地调用clear()分别清空？）
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

3

4. 栈与队列

(x) Steap + Queap

邓俊辉

deng@tsinghua.edu.cn

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; //有一位新顾客到达
        c.time = 1 + rand() % 50; //随机指定新顾客的服务时长1~50
        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
    } //while
    delete [] windows; //释放所有队列（是否需要显式地调用clear()分别清空？）
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

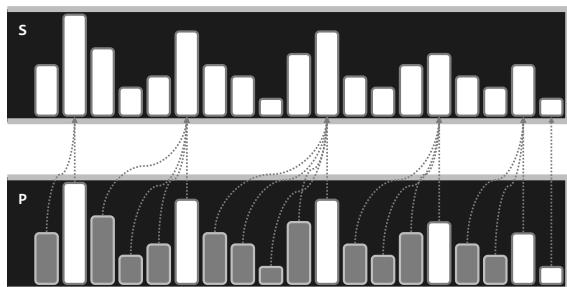
3

4. 栈与队列

(x) Steap + Queap

邓俊辉

deng@tsinghua.edu.cn



Data Structures & Algorithms (Fall 2012), Tsinghua University

f

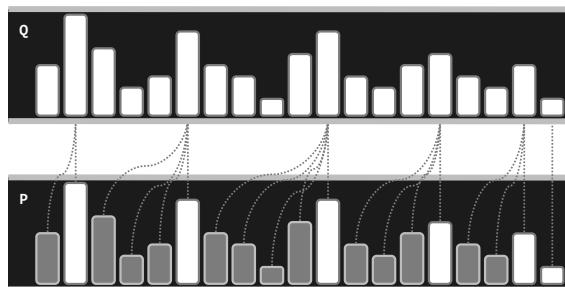
5. 二叉树

(a) 概述

Two roads diverged in a yellow wood
And sorry I could not travel both
- Robert Frost, 1915

邓俊辉

deng@tsinghua.edu.cn

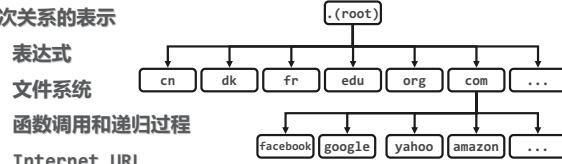


Data Structures & Algorithms (Fall 2012), Tsinghua University

g

应用

◆ 层次关系的表示



◆ 综合性：集成Vector和List的优点

1. 既可快速查找
2. 亦可快速插入、删除

◆ 半线性：不再是简单的线性结构，但

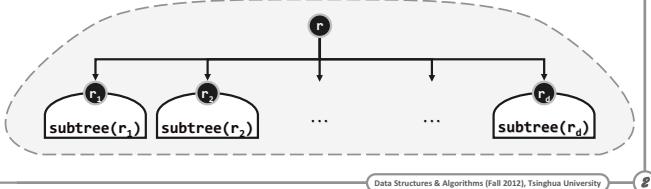
在确定某种次序之后，具有线性特征

Data Structures & Algorithms (Fall 2012), Tsinghua University

f

有根树

- ◆ 树是特殊的图 $T = (V, E)$, 节点数 $|V| = n$, 边数 $|E| = m$
- ◆ 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 $T_i = \text{subtree}(r_i)$

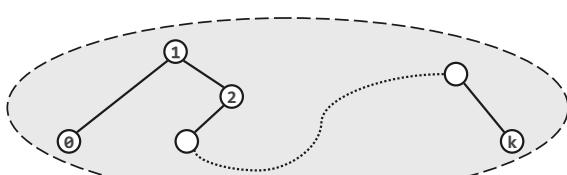


Data Structures & Algorithms (Fall 2012), Tsinghua University

g

路径与环路

- ◆ V 中的 $k+1$ 个节点, 通过 E 中的 k 条边首尾相联, 构成一条**路径** (path)
 $\pi = \{(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)\}$
 $|\pi| = k$ 称作路径长度 // 有的文献以节点数为长度
若 $v_k = v_0$, 则称作**环路** (cycle/loop)
- ◆ 任意节点之间均有路径的图, 称作**连通图** (connected graph)
不含环路的图, 称作**无环图** (acyclic graph)



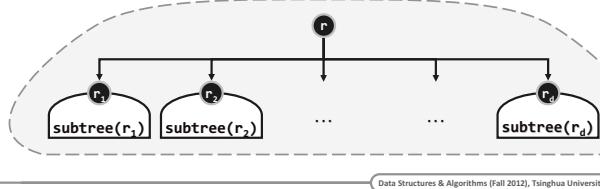
Data Structures & Algorithms (Fall 2012), Tsinghua University

279

279

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: $m = \text{节点度数总和} = n-1$, 即节点数与边数相当
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)

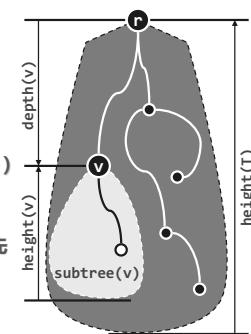


Data Structures & Algorithms (Fall 2012), Tsinghua University

g

深度与层次

- ◆ 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
记作 $\text{path}(v, r) = \text{path}(v)$
- ◆ $\text{depth}(v) = |\text{path}(v)|$
称作 v 的**深度** (depth)
- ◆ $\text{path}(v)$ 上节点均为 v 的**祖先** (ancestor)
 v 是它们的**后代** (descendent)
- ◆ 除自身以外的祖先, 是**真** (proper) 祖先
- ◆ 在任一深度: v 的祖先若存在则必唯一
 v 的后代不见得唯一



Data Structures & Algorithms (Fall 2012), Tsinghua University

g

◆ 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)

◆ 叶子的最大深度称作(子)树的高度

也是(子)树根节点的高度(height)

$$\text{height}(v) = \text{height}(\text{subtree}(v))$$

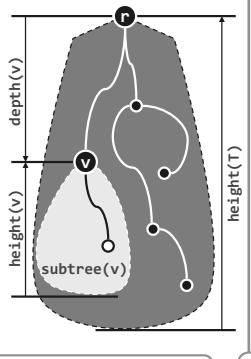
◆ 特别地，空树的高度取作-1

◆ $\text{depth}(v) + \text{height}(v) \leq \text{height}(T)$

何时取等号？

◆ 不致歧义时

路径、节点和子树可相互指代



Data Structures & Algorithms (Fall 2012), Tsinghua University

5.二叉树

(b) 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 |

Data Structures & Algorithms (Fall 2012), Tsinghua University

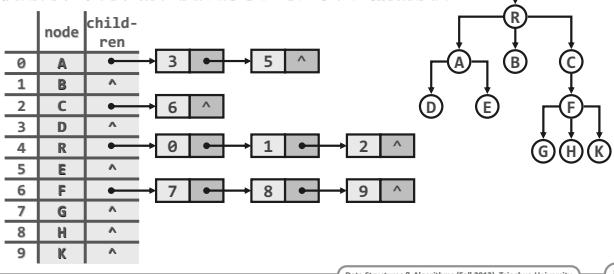
孩子节点

◆ 每个节点各用一个序列存放其所有孩子的引用

◆ 使用不可扩充的向量，向量的长度统一为最大度数d

$$\textcircled{⑧} \text{ 空间使用率} = (n-1) / (n \cdot d) < 1/d$$

◆ 使用列表，列表的长度分别等于对应节点当前的度数



Data Structures & Algorithms (Fall 2012), Tsinghua University

长子 + 兄弟

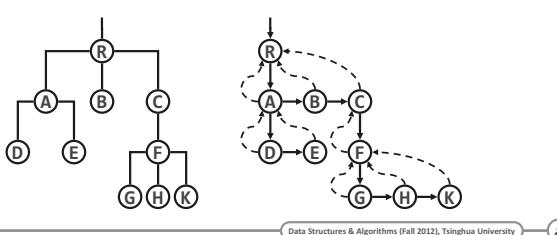
◆ 每个节点均设两个引用

纵：`firstChild()`

横：`nextSibling()`

◆ 如此，对于度数为d的节点，可在 $O(d+1)$ 时间内遍历其所有孩子

◆ 若再设置`parent`引用，则`parent()`接口也仅需 $O(1)$ 时间



Data Structures & Algorithms (Fall 2012), Tsinghua University

父节点

◆ 观察：除根外，任一节点有且仅有1个父节点

◆ 构思：将节点组织为一个序列，各节点保存

node：本身的信息，以及

parent：父节点的秩或位置

◆ 树根也有一个“虚构的”父节点-1或NULL

◆ 空间性能： $O(n)$

◆ 时间性能

$$\textcircled{⑨} \quad \text{parent(): } O(1)$$

$$\textcircled{⑨} \quad \text{root(): } O(n) \text{ 或 } O(1)$$

$$\textcircled{⑨} \quad \text{firstChild(): } O(n)$$

$$\textcircled{⑨} \quad \text{nextSibling(): } O(n)$$

◆ 如何加速对孩子、兄弟的查找？

| rank | node | 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 |

Data Structures & Algorithms (Fall 2012), Tsinghua University

父节点 + 孩子节点

◆ 现在，所有孩子都可很快找出，但`parent()`却很慢

◆ 改进：父节点 + 孩子节点

| node | parent | child-refs |
|------|--------|------------|
| 0 A | 4 | 3 5 |
| 1 B | 4 | 6 |
| 2 C | 4 | 6 |
| 3 D | 0 | 8 |
| 4 R | -1 | 0 1 2 |
| 5 E | 0 | 9 |
| 6 F | 2 | 7 8 |
| 7 G | 6 | 9 |
| 8 H | 6 | |
| 9 K | 6 | |

Data Structures & Algorithms (Fall 2012), Tsinghua University

5.二叉树

(c) 二叉树概述

邓俊辉

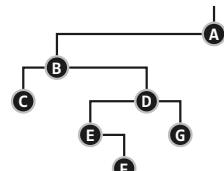
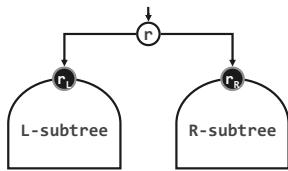
deng@tsinghua.edu.cn

◆ 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 //暗藏有序

lChild() + rChild()

lSubtree() + rSubTree()



Data Structures & Algorithms (Fall 2012), Tsinghua University

◆ 设度数为0、1和2的节点各有 n_0 、 n_1 和 n_2 个，则

$$\text{边数 } e = n - 1 = n_1 + 2n_2$$

$$\text{叶节点数 } n_0 = n_2 + 1$$

$$\text{节点数 } n = n_0 + n_1 + n_2 = 1 + n_1 + 2n_2 = 1 + e$$

◆ 特别地，当 $n_1 = 0$ 时，有

$$e = 2n_2 \text{ 和 } n_0 = n_2 + 1 = (n + 1)/2$$

此时，节点度数均为偶数，不含单分支节点...



Data Structures & Algorithms (Fall 2012), Tsinghua University

◆ 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

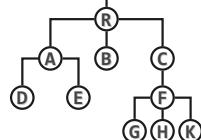
◆ 实际上，多叉树均可转化并表示为二叉树

— 回忆“长子、兄弟”表示法...

◆ 为此只需旋转45度：将长子、兄弟与左、右孩子等效地相互对应

`firstChild() = lChild()`

`nextSibling() = rChild()`



Data Structures & Algorithms (Fall 2012), Tsinghua University

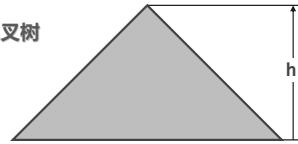
◆ 考查包含n个节点、高度为h的二叉树

◆ 其中，深度为k的节点至多 2^k 个

◆ $h < n < 2^{h+1}$

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)



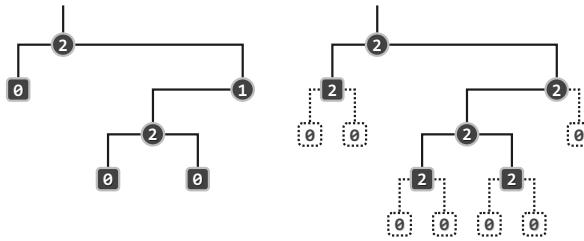
Data Structures & Algorithms (Fall 2012), Tsinghua University

◆ 通过引入 $n_0 + 2n_1$ 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



Data Structures & Algorithms (Fall 2012), Tsinghua University

5. 二叉树

(d) 二叉树实现

邓俊辉

deng@tsinghua.edu.cn

```
◆ template <typename T>
class BinNode {
    parent
    data
    lChild rChild
public:
    T data; BinNodePosi(T) parent, lChild, rChild; //父亲、孩子
    int height; int size(); //高度、子树规模
    BinNodePosi(T) insertAsLC(T const& e); //作为左孩子插入新节点
    BinNodePosi(T) insertAsRC(T const& e); //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
};
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

◆ template <typename T>

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

◆ template <typename T>

BinNodePosi(T) BinNode<T>::insertAsRC(T const& e)
{ return rChild = 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 (lChild) s += lChild->size(); //递归计入左子树规模
    if (rChild) s += rChild->size(); //递归计入右子树规模
    return s;
}
```



Data Structures & Algorithms (Fall 2012), Tsinghua University

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() { return _size; } //规模
    bool empty() const { return !_root; } //判断空
    BinNodePosi(T) & root() { return _root; } //树根
    /* ... 子树接入、删除和分离接口 ... */
    /* ... 遍历接口 ... */
}

```

Data Structures & Algorithms (Fall 2012), Tsinghua University

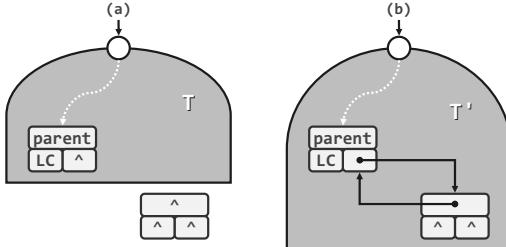
3

节点插入

```

◆ template <typename T> BinNodePosi(T) BinTree<T>::insertAsRC(BinNodePosi(T) x, T const& e) { //insertAsRC()对称
    _size++; x->insertAsRC(e);
    updateHeightAbove(x); //x祖先的高度可能增加，其余节点必然不变
    return x->rChild;
}

```



Data Structures & Algorithms (Fall 2012), Tsinghua University

5

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->lChild) + removeAt(x->rChild);
    release(x->data); release(x); //释放被摘除节点，并
    return n; //返回被删除节点总数
}

```

Data Structures & Algorithms (Fall 2012), Tsinghua University

7

5. 二叉树

(e1) 先序遍历

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。”

郑俊辉

deng@tsinghua.edu.cn

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->lChild), stature(x->rChild));
}

◆ 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; }
}

```

Data Structures & Algorithms (Fall 2012), Tsinghua University

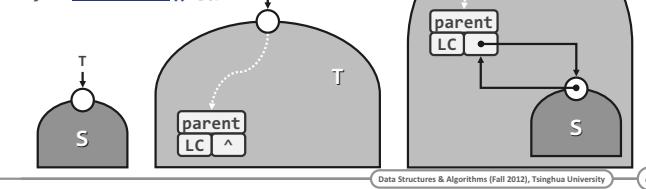
6

子树接入

```

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

```



Data Structures & Algorithms (Fall 2012), Tsinghua University

6

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
}

```

Data Structures & Algorithms (Fall 2012), Tsinghua University

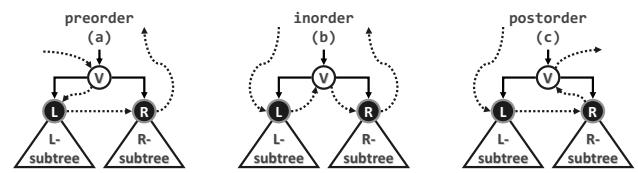
8

◆ $T = V \cup L \cup R = \{root\} \cup L_subtree(T) \cup R_subtree(T)$
按照某种次序访问树中各节点，每个节点被访问恰好一次

◆ 遍历结果 ~ 遍历过程 ~ 遍历次序 ~ 遍历策略

| | | | |
|----|----|----|----|
| 先序 | 中序 | 后序 | 层次 |
|----|----|----|----|

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



Data Structures & Algorithms (Fall 2012), Tsinghua University

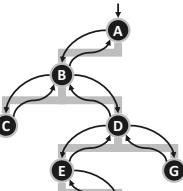
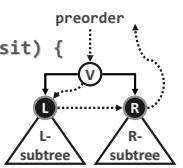
f

```
* template <typename T, typename VST>
void traverse(BinNodePosi(T) x, VST& visit) {
    if (!x) return;
    visit(x->data);
    traverse(x->lChild, visit);
    traverse(x->rChild, visit);
}
```

❖ 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？



Data Structures & Algorithms (Fall 2012), Tsinghua University

```
* 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->rChild); // 右孩子先入后出
        if (HasLChild(*x)) S.push(x->lChild); // 左孩子后入先出
    } // 体会以上两句的次序
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

❖ 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

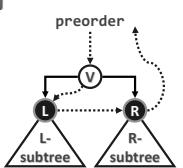
> 根先：对于任一子树，根被访问后才会访问其它节点

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： $\Theta(n)$

- 1) 每步迭代，都有一个节点出栈并被访问
- 2) 每个节点入/出栈一次且仅一次
- 3) 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...

Data Structures & Algorithms (Fall 2012), Tsinghua University

❖ 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

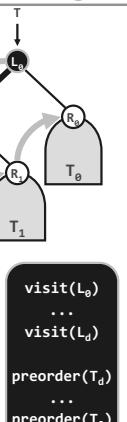
❖ 不同右子树的遍历

相互独立

自成一个子任务

deepest node along left branch
L_{d-1}
R_{d-1}
T_{d-1}
L_d
R_d
T_d

T



Data Structures & Algorithms (Fall 2012), Tsinghua University

❖ 先序遍历任一二叉树T时

首先访问根节点r

接下来，先后递归地遍历T_L和T_R

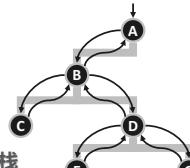
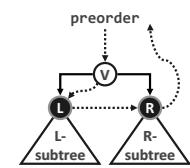
❖ 递归实现中，对左、右子树的递归遍历

都类似于尾递归

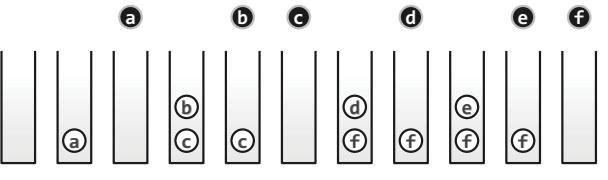
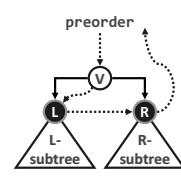
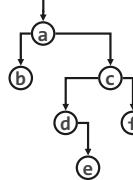
故可以某种方式直接消除

❖ 思路：

二分递归 → 迭代 + 单递归 → 迭代 + 栈

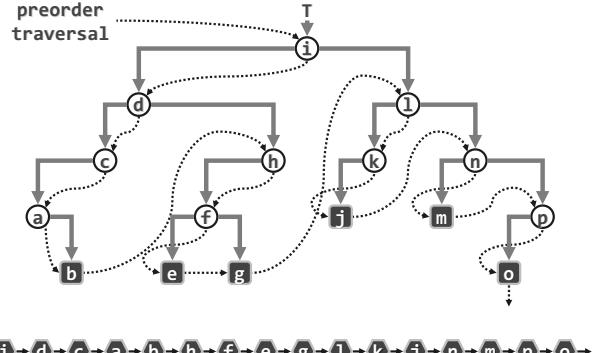


Data Structures & Algorithms (Fall 2012), Tsinghua University



Data Structures & Algorithms (Fall 2012), Tsinghua University

preorder traversal



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

Data Structures & Algorithms (Fall 2012), Tsinghua University

❖ template <typename T, typename VST>

```
static void visitAlongLeftBranch // 分摊 O(1)
(BinNodePosi(T) x, VST& visit, Stack<BinNodePosi(T)>& S) {
    while (x) // 沿左侧链访问至底，沿途右孩子依次入栈 (将来逆序出栈)
        { visit(x->data); S.push(x->rChild); x = x->lChild; }
}

❖ void travPre_I2(BinNodePosi(T) x, VST& visit) {
    Stack<BinNodePosi(T)> S; // 辅助栈
    while (true) { // 逐批访问节点，直至辅助栈空
        visitAlongLeftBranch(x, visit, S); // 每批均从当前节点x出发
        if (S.empty()) break; x = S.pop(); // 弹出下一批的起点
    }
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

5. 二叉树

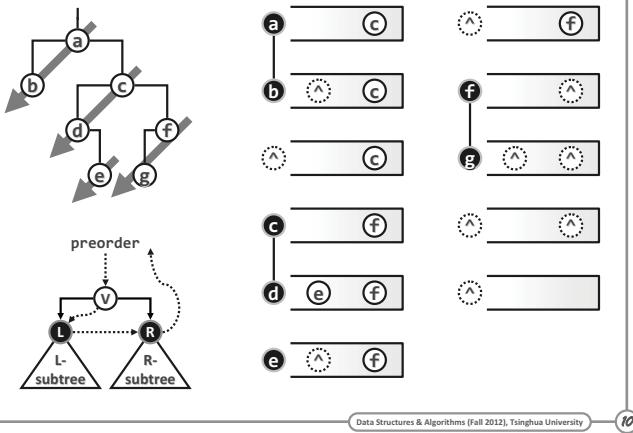
(e2) 中序遍历

邓俊辉

deng@tsinghua.edu.cn

迭代2：实例

313



Data Structures & Algorithms (Fall 2012), Tsinghua University

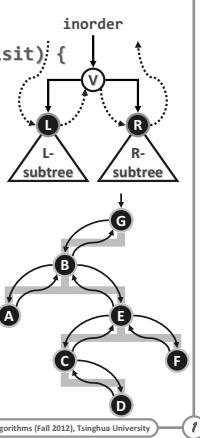
递归

315

```
template <typename T, typename VST>
void traverse(BinNodePosi(T) x, VST& visit) {
    if (!x) return;
    traverse(x->lChild, visit);
    visit(x->data);
    traverse(x->rChild, visit);
}

中序输出文件树结构
printBinTree()

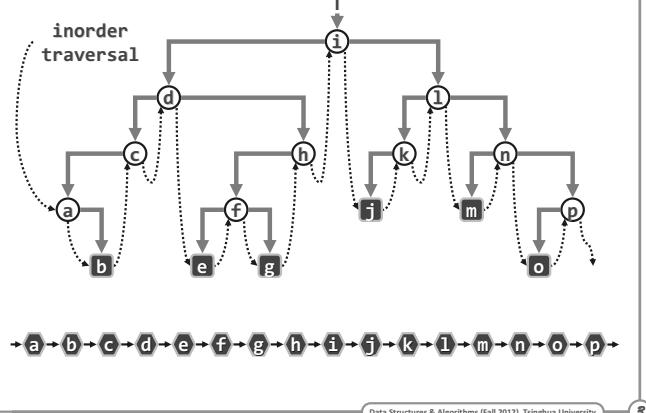
挑战：不依赖递归机制，能否实现中序遍历？
如何实现？效率如何？
```



Data Structures & Algorithms (Fall 2012), Tsinghua University

迭代：思路

317



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

Data Structures & Algorithms (Fall 2012), Tsinghua University

迭代：实现

319

```
template <typename T> static void goAlongLeftBranch
(BinNodePosi(T) x, Stack<BinNodePosi(T)>& S) //反复地
{ while (x) { S.push(x); x = x->lChild; } } //入栈，沿左分支深入

template <typename T, typename VST>
void traverseIn_I1(BinNodePosi(T) x, VST& 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); //立即访问它
        x = x->rChild; //再转向其右子树（可能为空，留意处理手法）
    }
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

迭代：难点

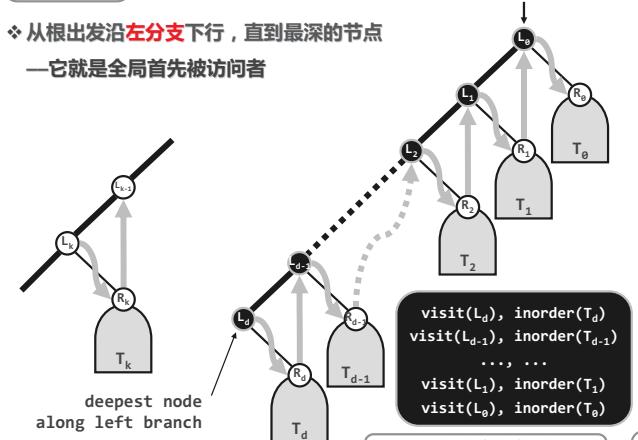
- 难度在于 尽管右子树的递归遍历是尾递归，但左子树却严格地不是
- 解决方法 找到第一个被访问的节点 //仿照迭代的先序遍历算法
将其祖先用栈保存 //按照被访问过程的逆序
- 这样，原问题就被分解为 依次对若干棵右子树的遍历问题 //依什么“次”？
- 于是，首先要解决的问题就是： 中序遍历任一二叉树T时
首先被访问的是哪个节点？如何找到它？

Data Structures & Algorithms (Fall 2012), Tsinghua University

迭代：思路

318

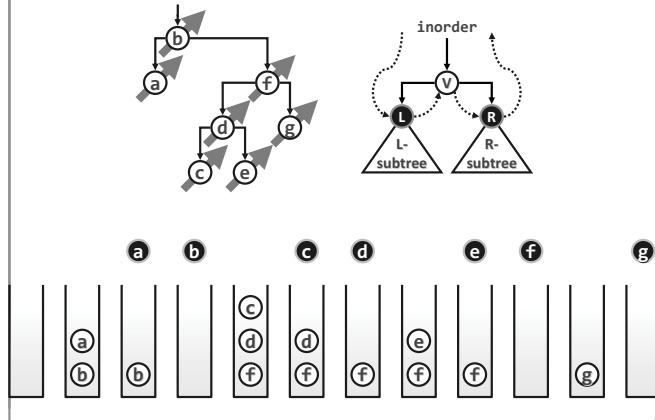
- 从根出发沿左分支下行，直到最深的节点 —— 它就是全局首先被访问者



Data Structures & Algorithms (Fall 2012), Tsinghua University

迭代：实例

320

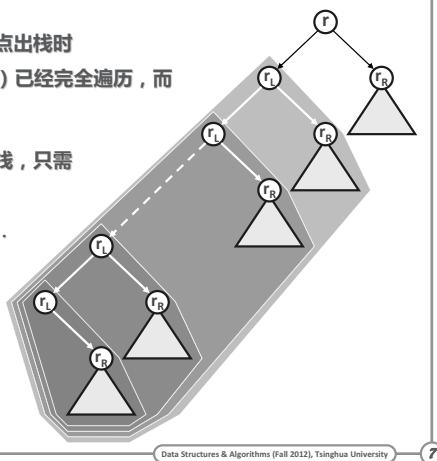


Data Structures & Algorithms (Fall 2012), Tsinghua University

321

迭代：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...



Data Structures & Algorithms (Fall 2012), Tsinghua University

7

322

迭代：效率

- 是否 $\Theta(n)$ ，取决于以下条件
 - 每次迭代，都恰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`goAlongLeftBranch()`
就可能需做 $\Omega(n)$ 次入栈操作，共需 $\Omega(n^2)$ 时间
- 既然如此，难道总体将需要... $\Theta(n^2)$ 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()`

Data Structures & Algorithms (Fall 2012), Tsinghua University

8

323

直接后继

```
template <typename T>
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 (rChild) { //若有右孩子，则直接后继必在右子树中，具体地就是
        s = rChild; //右子树中
        while (HasLChild(*s)) s = s->lChild; //最靠左（最小）节点
    } 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))
            s = s->parent; //逆向地沿右向分支，不断朝左上方移动
        s = s->parent; //最后再朝右上移动一步，即抵达后继（如果存在）
    } //两种情况的运行时间为当前节点的高度与深度，不过 $\Theta(h)$ 
    return s; //可能是NULL
} //稍后的搜索树节点删除操作removeAt()需借助这一接口
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

9

5. 二叉树

(e3) 后序遍历

邓俊辉

deng@tsinghua.edu.cn

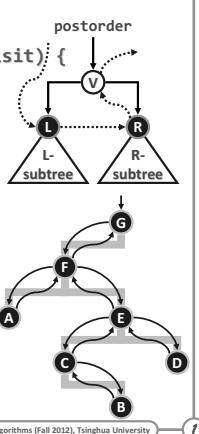
324

递归

```
template <typename T, typename VST>
void traverse(BinNodePosi(T) x, VST& visit) {
    if (!x) return;
    traverse(x->lChild, visit);
    traverse(x->rChild, visit);
    visit(x->data);
}
```

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？



Data Structures & Algorithms (Fall 2012), Tsinghua University

10

325

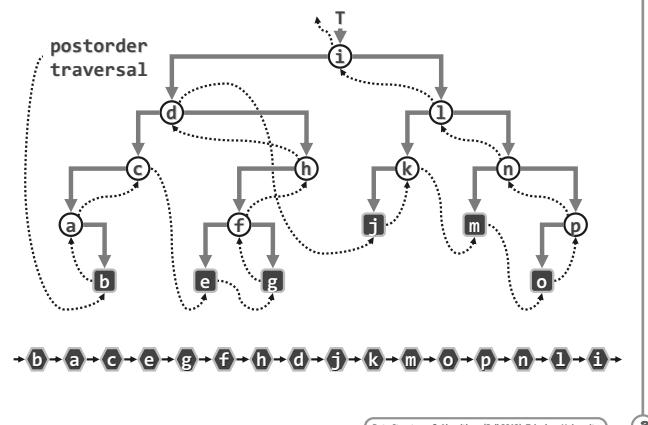
迭代：难点

- 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
后序遍历一二叉树T时
首先被访问的是哪个节点？如何找到它？

Data Structures & Algorithms (Fall 2012), Tsinghua University

11

迭代：思路



Data Structures & Algorithms (Fall 2012), Tsinghua University

12

327

迭代：思路

- 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



Data Structures & Algorithms (Fall 2012), Tsinghua University

13

迭代：实现

```

◆ 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->rChild); //优先入栈
            S.push(x->lChild); //然后转向左孩子
        } else //实不得已
            S.push(x->rChild); //才转向右孩子
    S.pop(); //返回之前，弹出栈顶的空节点
}

```

Data Structures & Algorithms (Fall 2012), Tsinghua University

5

```

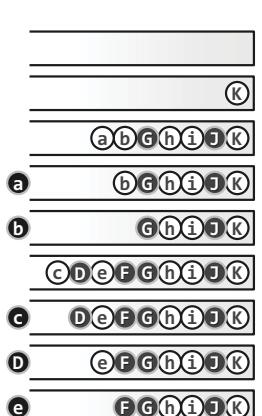
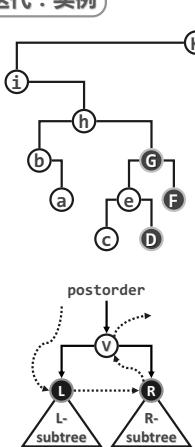
◆ template <typename T, typename VST>
void travPost_I(BinNodePosi(T) x, VST& visit) {
    Stack<BinNodePosi(T)> S; //辅助栈
    if (x) S.push(x); //根节点非空则首先入栈
    while (!S.empty()) { //x为当前节点
        if (S.top() != x->parent) //栈顶非x之父（则必为其右兄）
            gotoHLVFL(S); //在x的右子树中，找到HLVFL
        x = S.pop(); //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
    }
}

```

Data Structures & Algorithms (Fall 2012), Tsinghua University

6

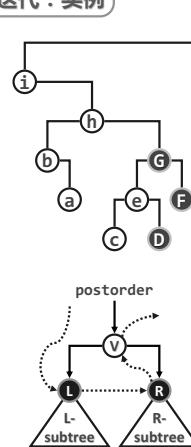
迭代：实例



Data Structures & Algorithms (Fall 2012), Tsinghua University

7

迭代：实例

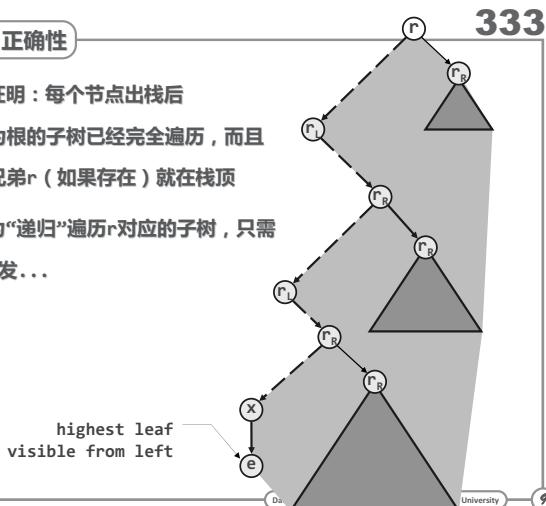


Data Structures & Algorithms (Fall 2012), Tsinghua University

8

迭代：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...



9

5. 二叉树

(e4) 层次遍历

邓俊辉

deng@tsinghua.edu.cn

迭代：效率

- 是否 $\mathcal{O}(n)$ ，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 $\mathcal{O}(1)$ 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... $\mathcal{O}(n^2)$ 时间？
- 同样地，事实上这个界远远不紧...

Data Structures & Algorithms (Fall 2012), Tsinghua University

10

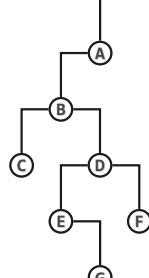
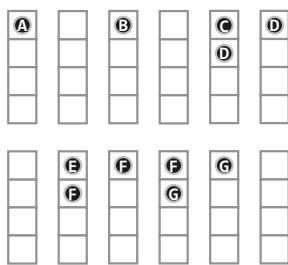
```

◆ template <typename T> template <typename VST>
void BinNode<T>::travLevel(VST& visit) { //二叉树层次遍历
    Queue<BinNodePosi(T)> Q; //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->lChild); //左孩子入队
        if (HasRChild(*x)) Q.enqueue(x->rChild); //右孩子入队
    }
}

```

Data Structures & Algorithms (Fall 2012), Tsinghua University

f



Data Structures & Algorithms (Fall 2012), Tsinghua University

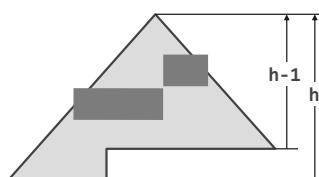
分析

- ❖ 正确性何以见得？
- ❖ 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差不超过1
- ❖ 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
- ❖ 故此，迭代算法符合广度优先遍历的次序要求
- ❖ 效率 = $\Theta(n)$
每个节点入、出队各一次且仅一次
每经过一次迭代，都有一个节点出队并接受访问

Data Structures & Algorithms (Fall 2012), Tsinghua University

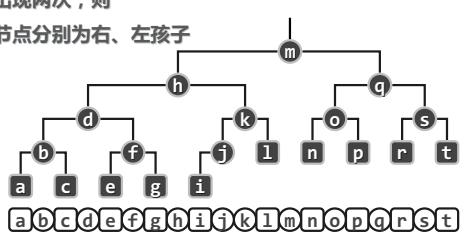
完全二叉树

- ❖ complete binary tree



- ❖ 考查辅助队列的规模...
- ❖ 在层次遍历的n次迭代中
 - 1) 每一规模至多出现两次
 - 2) 若最大规模出现两次，则

当时的队末节点分别为右、左孩子



Data Structures & Algorithms (Fall 2012), Tsinghua University

问题

- ❖ 通讯 / 编码 / 译码

原始信息 → 编码器 → 事先达成的编码协议 → 通讯信道 → 解码器 → 译码 / 预测
- ❖ 二进制编码

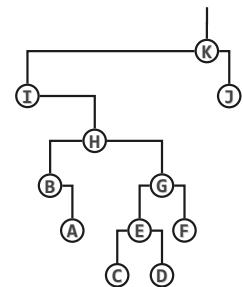
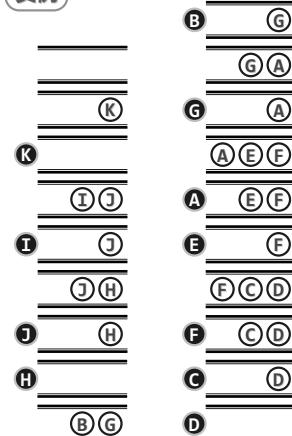
组成数据文件的字符来自字符集 Σ
字符被赋予互异的二进制串
- ❖ 文件的大小取决于

字符的数量 × 各字符编码的长短

| | | | |
|---|-----|-----|----|
| M | A | I | N |
| 1 | 010 | 011 | 00 |
- ❖ 通讯带宽有限时

如何对各字符编码，使文件最小？

Data Structures & Algorithms (Fall 2012), Tsinghua University



Data Structures & Algorithms (Fall 2012), Tsinghua University

应用：表达式树

- ❖ 表达式树 (expression tree) : 由前缀表达式创建表达式

 $a + b * (c - d) - e / f$

- ❖ postorder (postfix = RPN)

 $a\ b\ c\ d\ -\ *\ +\ e\ f\ /$

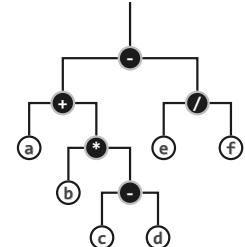
- ❖ preorder (prefix, 求值并不便捷)

 $-+a*\ b- c\ d/ef$

- ❖ inorder (infix, 无优先级, 有歧义)

 $a+b*c-d-e/f$

- ❖ breadth-first (?)

 $-+/\ a*\ e\ f\ b- c\ d$ 

Data Structures & Algorithms (Fall 2012), Tsinghua University

5. 二叉树

(f) PFC编码树

邓俊辉

deng@tsinghua.edu.cn

二叉编码树

- ❖ 将 Σ 中字符组织成一棵二叉树

以 0、1 表示左、右孩子

各字符 c 分别存放于对应的叶子 $v(c)$ 中

- ❖ c 的编码串 $rps(v(c))$

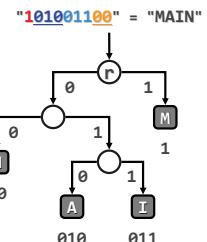
由根到 $v(c)$ 的通路 (root path) 确定

- ❖ 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？



Data Structures & Algorithms (Fall 2012), Tsinghua University

编码长度 vs. 叶节点平均深度

◆ 字符c的编码长度 $|rps(v(c))|$

=

v(c)的深度 $depth(v(c))$

◆ 编码总长 = $\sum_c depth(v(c))$

平均编码长度

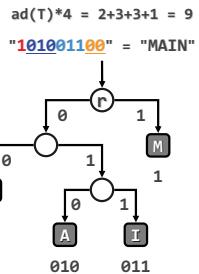
$$= \sum_c depth(v(c)) / |\Sigma|$$

= 叶节点平均深度 $ad(T)$

◆ 对于特定的 Σ , 以上指标最小者即为

最优编码树 $T_{opt} = (V_{opt}, E_{opt})$

◆ 最优编码树具有哪些特征?



Data Structures & Algorithms (Fall 2012), Tsinghua University

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

"011001100011" = "Mamani", wad*6=12
"0110010110011110" = "MammaMia", wad*8=16

如何构造最优编码树?

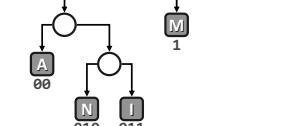
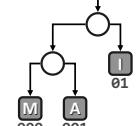
◆ 文件长度 \propto 平均带权深度

$$= wad(T) = \sum_c w(c) \times depth(v(c))$$

◆ 此时, 完全树不见得是最优编码树

"000001000001101" = "Mamani", wad*8=15
"00000100000000100001001" = "MammaMia", wad*8=23

"10010001010011" = "Mamani", wad*6=12
"1001100101100" = "MammaMia", wad*8=13



Data Structures & Algorithms (Fall 2012), Tsinghua University

5. 二叉树

(g) Huffman树

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?

Data Structures & Algorithms (Fall 2012), Tsinghua University

最优编码树

◆ $\forall v \in V_{opt}$, $deg(v) = 0$ only if $depth(v) \geq depth(T_{opt}) - 1$

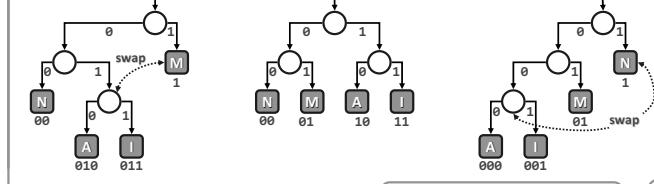
亦即, 叶子只能出现在倒数两层内 — 否则, 通过节点交换可以...

◆ 特别地, 真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 $w('E') >> w('Z')$

$ad(T)*4 = 2+3+3+1 = 9$ $ad(T)*4 = 2+2+2+2 = 8$ $ad(T)*4 = 2+3+3+1 = 9$

"101001100" = "MAIN" "01101100" = "MAIN" "010000011" = "MAIN"

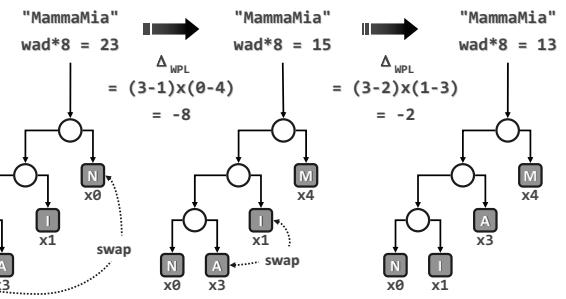


Data Structures & Algorithms (Fall 2012), Tsinghua University

最优编码树

◆ 同样, 频率高(低)的字符, 应尽可能放在高(低)处

◆ 故此, 通过交换, 同样可以缩短 $wad(T)$



Data Structures & Algorithms (Fall 2012), Tsinghua University

策略与算法

◆ 自下而上构造Huffman树 //稍后可见, 它的确是一棵最优编码树

//贪婪策略: 在构造编码树的过程中, 频率低的字符优先引入

//贪心目标: 在构造出来的编码树中, 频率低的字符位置更低

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(非降)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, 满足:

 lchild(T) = T_1 且 rchild(T) = T_2

 w(root(T)) = w(root(T_1)) + w(root(T_2))

} //<http://thudsa.3322.org/~deng/ds/demo/huffman/>

Data Structures & Algorithms (Fall 2012), Tsinghua University

双子性

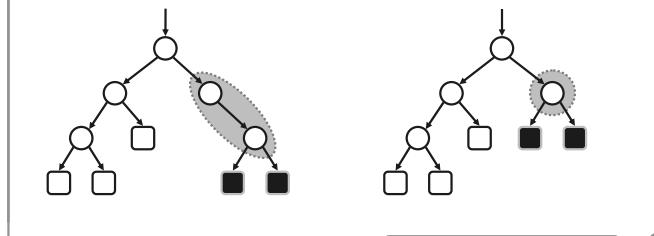
◆ 只要 $|\Sigma| > 1$, Huffman树中每一内部节点都有两个孩子, 亦即

节点度数均为0或2

//偶数

Huffman树必为真二叉树

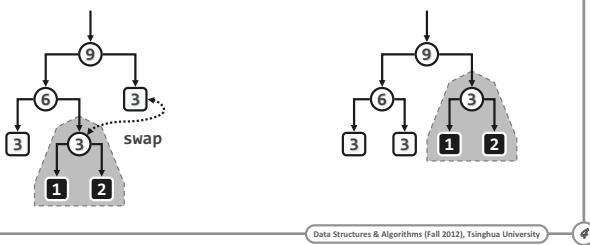
◆ 否则, 将1度节点替换为其唯一的子, 则新树的WAD将更小



Data Structures & Algorithms (Fall 2012), Tsinghua University

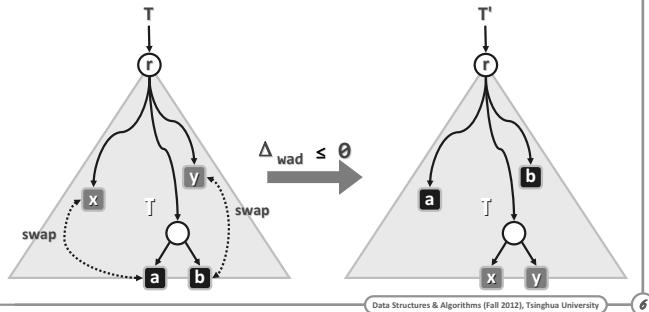
不唯一性

- 任一内部节点的左、右子树相互交换后，WAD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

- 任取一棵最优编码树T
在其最底层，任取一对兄弟a和b
分别交换a和x、b和y后，WAD绝不会增加
- //注意T的存在性
//同样，注意其存在性
//正此前已看到的



正确性！

- 由 Huffman 编码算法生成的编码树，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
 - 取 Σ 中频率最低的 x 和 y // 由层次性，仅考虑其在最底层互为兄弟的情形
 - 令 $\Sigma' = (\Sigma \setminus \{x, y\}) \cup \{z\}$, $w(z) = w(x) + w(y)$
 - 对于 Σ' 的任一编码树 T' ，只要为 z 添加孩子 x 和 y，即可得到 Σ 的一棵编码树 T ，且 $WAD(T) - WAD(T') = w(x) + w(y) = w(z)$
 - 亦即，如此对应的 T 和 T'，WAD 之差与 T 的具体形态无关
 - 因此，只要 T' 是 Σ' 的最优编码树，则 T 也必是 Σ 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

Data Structures & Algorithms (Fall 2012), Tsinghua University 5

实现：构造编码树

```
#define HuffmanBinTree HuffChar // Huffman 树，节点类型 HuffChar
typedef List<HuffmanBinTree*> HuffmanForest; // Huffman 森林

HuffmanBinTree* generateTree(HuffmanForest* forest) { // Huffman 编码算法
    while (1 < forest->size()) { // 反复迭代，直至森林中仅含一棵树
        HuffmanBinTree* T1 = minHChar(forest), *T2 = minHChar(forest);
        HuffmanBinTree* S = new HuffmanBinTree(); // 创建新树，准备合并 T1 和 T2
        S->insertAsRoot(HuffChar('^'), // 根节点权重为 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
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University 8

实现：搜索最小超字符

```
HuffmanBinTree* minHChar(HuffmanForest* forest) {
    ListNodePosi(HuffmanBinTree*) p = forest->first(); // 从首节点出发
    ListNodePosi(HuffmanBinTree*) 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) { // 如有必要
            minWeight = p->data->root()->data.weight; // 则更新记录
            minChar = p;
        }
    return forest->remove(minChar); // 从森林中摘除该树，并返回
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University 9

实现：构造编码表

```
#include "../Hashtable/Hashtable.h" // 用 Hashtable (第9章) 实现
typedef Hashtable<char> HuffTable; // Huffman 编码表

static void generateCT(Bitmap* code, // 通过遍历获取各字符的编码
                      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
        code->clear(length);
        generateCT(code, length + 1, table, v->lChild); // 深入遍历
    }
    if (HasRChild(*v)) { // Right = 1
        code->set(length);
        generateCT(code, length + 1, table, v->rChild);
    }
} // 总体 O(n)
```

Data Structures & Algorithms (Fall 2012), Tsinghua University 10

改进

- | | | |
|----------------------|---------------|--------------------|
| 方案1 | $\Theta(n^2)$ | |
| > 初始化时，通过排序得到一个非升序向量 | | $\Theta(n \log n)$ |
| > 每次（从后端）取出频率最低的两个节点 | | $\Theta(1)$ |
| > 将合并得到的新树插入有序向量 | | $\Theta(n)$ |
| 方案2 | $\Theta(n^2)$ | |
| > 初始化时，通过排序得到一个非降序列表 | | $\Theta(n \log n)$ |
| > 每次（从前端）取出频率最低的两个节点 | | $\Theta(1)$ |
| > 将合并得到的新树插入有序列表 | | $\Theta(n)$ |

Data Structures & Algorithms (Fall 2012), Tsinghua University 11

改进

◆ 方案3, $\mathcal{O}(n \log n)$

初始化时, 将所有树组织为一个优先队列 // $\mathcal{O}(n)$
 取出频率最低的两个节点 // $\mathcal{O}(\log n)$
 将合并得到的新树插入队列 // $\mathcal{O}(\log n)$
 对Huffman树做一次遍历, 即可导出编码表 // $\mathcal{O}(n)$

◆ 方案4, $\mathcal{O}(n)$

(如果所有字符已经按频率非降排序)
 使用 $\mathcal{O}(n)$ 空间, 维护两个有序队列...

// 稍后第10章...保持兴趣

6. 图

(a) 概述

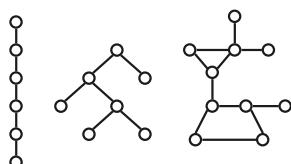
邓俊辉

deng@tsinghua.edu.cn

Data Structures & Algorithms (Fall 2012), Tsinghua University

12

基本术语

◆ 线性 \rightarrow 半线性 \rightarrow 非线性◆ $G = (V; E) = (\text{顶点集}; \text{边集})$

◆ 顶点 (vertex)

◆ 边 (edge) / 弧 (arc)

◆ 习惯约定: $n = |V|, e = |E|$

◆ 邻接关系 (adjacency): 定义同一条边的两个顶点之间

自环 (self-loop): 同一顶点自我相邻

简单图 (simple graph) 不含自环, 非简单图暂不讨论

◆ 关联关系 (incidence): 顶点与其所属的边之间

度 (degree): 于同一顶点关联的边数

Data Structures & Algorithms (Fall 2012), Tsinghua University

13

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: $v_0 = v_k$

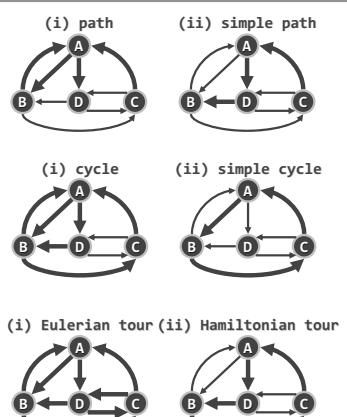
◆ 有向无环图 (DAG)

◆ 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



Data Structures & Algorithms (Fall 2012), Tsinghua University

14

6. 图

(b) 接口与实现

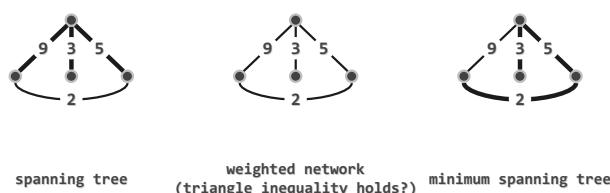
邓俊辉

deng@tsinghua.edu.cn

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不唯一◆ 各边 e 均有对应的权值 $wt(e)$, 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 holds?)

minimum spanning tree

Data Structures & Algorithms (Fall 2012), Tsinghua University

Graph模板类

```

template <typename Tv, typename Te> // 顶点类型、边类型
class Graph {
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)) status(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

```

Data Structures & Algorithms (Fall 2012), Tsinghua University

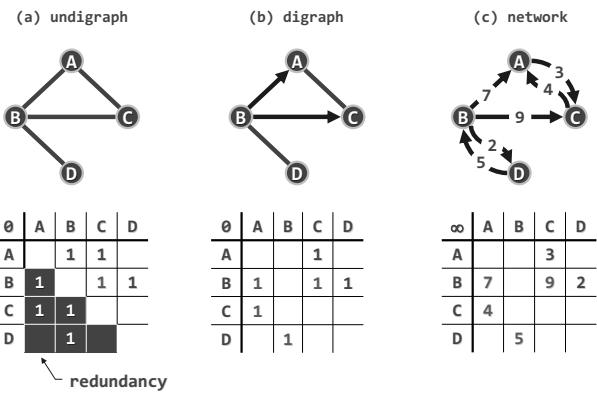
15

邻接矩阵与关联矩阵

- 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 $\Theta(n \cdot e) = \Theta(n^3)$
空间利用率 < $2/n$
解决某些问题时十分有效

Data Structures & Algorithms (Fall 2012), Tsinghua University

邻接矩阵: 实例



Data Structures & Algorithms (Fall 2012), Tsinghua University

邻接矩阵: Vertex

```

typedef enum { UNDISCOVERED, DISCOVERED, VISITED } VStatus;
template <typename Tv> struct Vertex { //顶点对象 (不再严格封装)
    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) {}
};
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

邻接矩阵: Edge

```

typedef
    enum { UNDETERMINED, TREE, CROSS, FORWARD, BACKWARD } EStatus;
    EStatus;

template <typename Te> struct Edge { //边对象 (不再严格封装)
    Te data; //数据
    int weight; //权重
    EStatus status; // (如上五种) 状态
    Edge(Te const& d, int w) : //构造新边
        data(d),
        weight(w),
        status(UNDETERMINED) {}
};
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

邻接矩阵: GraphMatrix

```

template <typename Tv, typename Te> //顶点类型、边类型
class GraphMatrix : public Graph<Tv, Te> {
private: //顶点集 (向量), 边集 (邻接矩阵)
    Vector<Vertex<Tv*>*> V; Vector<Vector<Edge<Te*>*>> E;
public:
    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
    }
    /* 操作接口: 顶点查询, 顶点修改, 边查询, 边修改 */
};
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

邻接矩阵: 顶点查询接口

```

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
int firstNbr(int i) { return nextNbr(i, n); } //首个邻居
int nextNbr(int i, int j) //逆向线性试探 (改用邻接表可提高效率)
{ while ((-1 < j) && (!exists(i, --j))); return j; } //下一邻居
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
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

邻接矩阵: 边查询接口

```

bool exists(int i, int j) { //判断边(i, j)是否存在
    return
        (0 <= i) && (i < n) && (0 <= j) && (j < n) &&
        E[i][j] != NULL;
}
Te edge(int i, int j) //边(i, j)的数据
{ return E[i][j]->data; }
EStatus& status(int i, int j) //边(i, j)的状态
{ return E[i][j]->status; }
int& weight(int i, int j) //边(i, j)的权重
{ return E[i][j]->weight; }
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

邻接矩阵: 顶点修改接口

```

int insert(Tv const& vertex) { //插入顶点, 返回编号
    E.insert(Vector<Edge<Te*>*>());
    for (int j = 0; j < n; j++) E[n].insert(NULL); n++;
    //对应的边初始为空
    for (int j = 0; j < n; j++) E[j].insert(NULL);
    //各顶点多出一条潜在关联边
    return V.insert(new Vertex<Tv>(vertex));
    //顶点向量增加一个顶点
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

邻接矩阵：顶点修改接口

```

◆ 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++) { //更新度数
        if (exists(i, j)) { e--; V[j]->inDegree--; } //出
        if (exists(j, i)) { e--; V[j]->outDegree--; } //入
    }
    if (exists(i, i)) e++; //追回可能重复的边计数（自环）
    Tv d = V[i]->data; //删除顶点i
    E.remove(i); n--; //删除第i行，以及
    for (int j = 0; j < n; j++) E[j].remove(i); //第i列
    return d;
}

```

Data Structures & Algorithms (Fall 2012), Tsinghua University 10

邻接矩阵：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： $\mathcal{O}(1)$
- 获取顶点的（出/入）度数： $\mathcal{O}(1)$
添加、删除边后更新度数： $\mathcal{O}(1)$
- 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

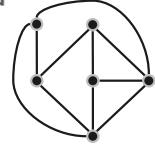
Data Structures & Algorithms (Fall 2012), Tsinghua University 11

邻接矩阵：缺点

- 空间复杂度为 $\mathcal{O}(n^2)$ ，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)：
 $v - e + f - c = 1$, for any PG

- 平面图： $e \leq 3n - 6 = \mathcal{O}(n) \ll n^2$
此时，空间利用率 $\approx 1/n$



- 对于一般的稀疏图（sparse graph）

空间利用率同样很低

此时，可采用压缩存储技术予以改进

邻接矩阵：缺点

- 空间复杂度为 $\mathcal{O}(n^2)$ ，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)：
 $v - e + f - c = 1$, for any PG
- 平面图： $e \leq 3n - 6 = \mathcal{O}(n) \ll n^2$
此时，空间利用率 $\approx 1/n$
- 对于一般的稀疏图（sparse graph）
空间利用率同样很低
此时，可采用压缩存储技术予以改进

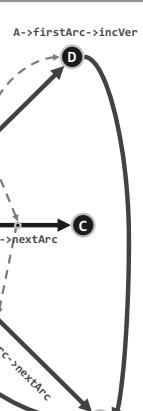
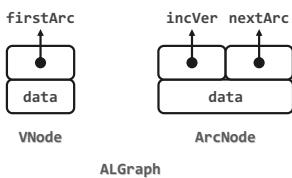
邻接表

- 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

$$L_v = \{ u \mid \langle v, u \rangle \in E \}$$



Data Structures & Algorithms (Fall 2012), Tsinghua University 14

邻接表：空间复杂度

- 有向图 = $\mathcal{O}(n + e)$
- 无向图 = $\mathcal{O}(n + 2xe) = \mathcal{O}(n + e)$

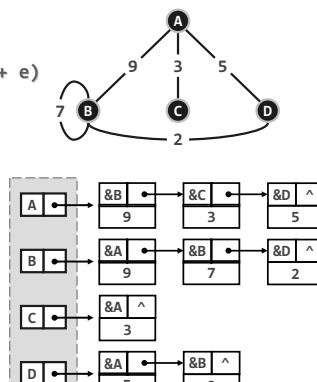
注意：无向弧被重复存储

问题：如何改进？

- 适用于稀疏图

$$\text{平面图} = \mathcal{O}(n + 3xn) = \mathcal{O}(n)$$

较之邻接矩阵，有极大改进



Data Structures & Algorithms (Fall 2012), Tsinghua University 15

邻接表：接口&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

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！

邻接表：空间复杂度

- 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

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！

Data Structures & Algorithms (Fall 2012), Tsinghua University 17

邻接表：接口&复杂度

◆ 给定顶点u和v，判断是否 $u, v \in E$

有向图：搜索u的邻接表， $\Theta(\deg(u)) = \Theta(e)$

无向图：搜索u或v的邻接表， $\Theta(\max(\deg(u), \deg(v))) = \Theta(e)$

“并行”搜索 $\Theta(2\min(\deg(u), \deg(v))) = \Theta(e)$

能够达到邻接矩阵的 $\Theta(1)$ 吗？

◆ 散列！如果装填因子选取得当 //保持兴趣

弧的判定：expected- $\Theta(1)$ ，与邻接矩阵“相同”

空间： $\Theta(n+e)$ ，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intersection graph之类的

隐式图（implicitly-represented graphs）

Data Structures & Algorithms (Fall 2012), Tsinghua University 18

6. 图

(c) 广度优先搜索

邓俊辉

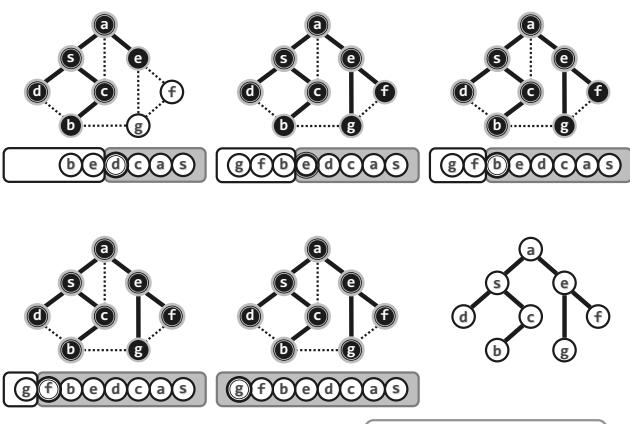
deng@tsinghua.edu.cn

Graph::BFS()

```
◆ template <typename Tv, typename Te> //顶点类型、边类型
void Graph<Tv, Te>::BFS(int v, int& clock) {
    Queue<int> Q; status(v) = DISCOVERED; Q.enqueue(v); //初始化
    while (!Q.empty()) { //不断取出队首顶点v，并考察v的所有邻居u
        int v = Q.dequeue(); dTime(v) = ++clock;
        for (int u = firstNbr(v); -1 < u; u = nextNbr(v, u))
            if (UNDISCOVERED == status(u)) { //若u尚未被发现，则
                status(u) = DISCOVERED; Q.enqueue(u); //发现该顶点
                status(v, u) = TREE; parent(u) = v; //引入树边
            } else //若u已被发现，或者甚至已访问完毕，则
                status(v, u) = CROSS; //将(v, u)归类于跨边
        status(v) = VISITED; //至此，当前顶点访问完毕
    }
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University 19

实例（无向图）



Data Structures & Algorithms (Fall 2012), Tsinghua University 19

取舍原则

◆ 空间/速度

◆ 顶点类型（bit / int / float / struct）

◆ 弧类型（方向 / 权值）

◆ 图类型（稀疏 / 稠密）

| | 邻接矩阵 | 邻接表 |
|------|------------|-----------|
| 适用场合 | 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 |

Data Structures & Algorithms (Fall 2012), Tsinghua University 19

算法

◆ BFS(s) //始自顶点s的广度优先搜索（Breadth-First Search）

访问顶点s

依次访问s所有尚未访问的邻接顶点

（按这些顶点被访问的先后次序）依次访问它们的邻接顶点
直至s所在连通分量（或s所属可达分量）中的所有顶点都被访问到

◆ 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

◆ 对于树而言，等同于广度优先遍历

事实上，BFS也的确会构造出原图的一棵支撑树（BFS tree）

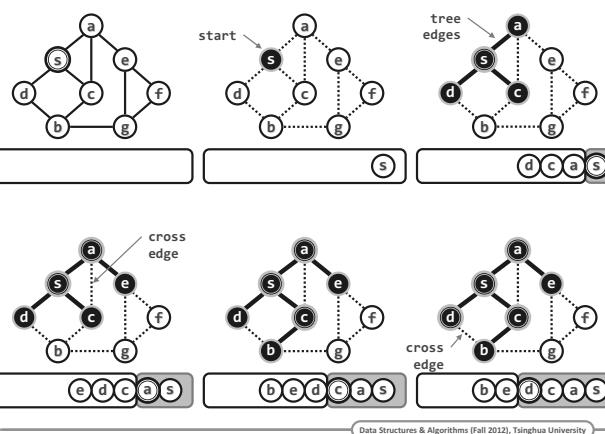
Data Structures & Algorithms (Fall 2012), Tsinghua University 19

Graph::BFS()

```
◆ template <typename Tv, typename Te> //顶点类型、边类型
void Graph<Tv, Te>::BFS(int v, int& clock) {
    Queue<int> Q; status(v) = DISCOVERED; Q.enqueue(v); //初始化
    while (!Q.empty()) { //不断取出队首顶点v，并考察v的所有邻居u
        int v = Q.dequeue(); dTime(v) = ++clock;
        for (int u = firstNbr(v); -1 < u; u = nextNbr(v, u))
            if (UNDISCOVERED == status(u)) { //若u尚未被发现，则
                status(u) = DISCOVERED; Q.enqueue(u); //发现该顶点
                status(v, u) = TREE; parent(u) = v; //引入树边
            } else //若u已被发现，或者甚至已访问完毕，则
                status(v, u) = CROSS; //将(v, u)归类于跨边
        status(v) = VISITED; //至此，当前顶点访问完毕
    }
}
```

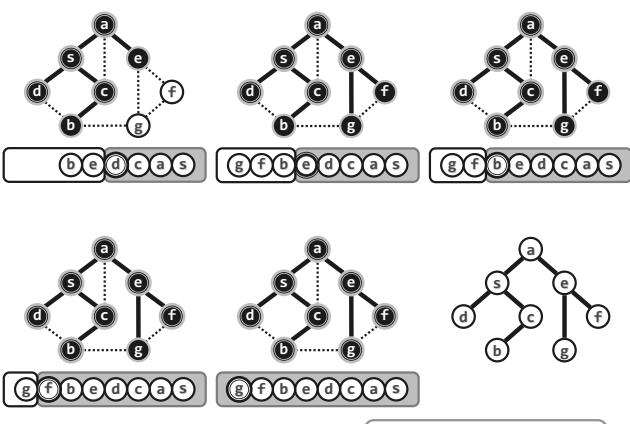
Data Structures & Algorithms (Fall 2012), Tsinghua University 19

实例（无向图）



Data Structures & Algorithms (Fall 2012), Tsinghua University 19

实例（无向图）



Data Structures & Algorithms (Fall 2012), Tsinghua University 19

19

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

do //逐一检查所有顶点，一旦遇到尚未发现的顶点

if (UNDISCOVERED == status(v))

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，故不漏不重

◆ 无论共有多少连通/可达分量，bfs()自身仅耗时 $\Theta(n)$

Data Structures & Algorithms (Fall 2012), Tsinghua University 19

19

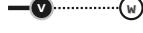
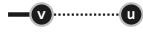
复杂度

- 初始化顶点 $\Theta(n)$
- 初始化边 $\Theta(e)$
- 迭代 $\Theta(n+2e)$
 - 外循环 //while (!Q.empty())
每个顶点只进入1次，总共n次
 - 内循环 //考察v的每一邻居u
每个邻居点（至多）进入1次，总共 $\Theta(1+\deg(v))$ 次 //1 = ?
总共 = $\Theta(\sum_v (1+\deg(v))) = \Theta(n+2e)$
- 整个算法 $\Theta(n+e) + \Theta(n+2e) = \Theta(2n+2e) = \Theta(n+e)$

Data Structures & Algorithms (Fall 2012), Tsinghua University

6

树边 & 跨边

- 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, w)  
- $v == \text{parent}(w)$
 $v \text{ DISCOVERED} \& w \text{ UNDISCOVERED}$
- 跨边(v, u)  
- 均为DISCOVERED //当然，也都不是VISITED
 $v \text{ DISCOVERED} \& u \text{ VISITED}$ //何时可能出现？
- http://thudsa.3322.org/~deng/ds/demo/graph_traversal/
- 注意：树边集合中不含回路 //但只要再加上任一跨边 ...

Data Structures & Algorithms (Fall 2012), Tsinghua University

7

BFS树/森林

- 从s出发的BFS
 - 进入遍历主循环1次 //为什么？
 - 进入主循环时，队列为空 //为什么？
 - s所在连通分量内的每个顶点
 - 迟早会进队1次 //可能多次吗？
 - 进队后立即被标注为DISCOVERED，并生成一条树边
 - 迟早会出队 //可能再次进入吗？
 - 退出主循环时，队列空 //为什么？
 - 生成n-c条树边，e-n+c条跨边
 - $\text{parent}(\text{root}) == \text{parent}(s) == \text{NO_PARENT}$
- 从s出发的BFS，将以s为根生成一棵BFS树；所有BFS树构成BFS森林

Data Structures & Algorithms (Fall 2012), Tsinghua University

8

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 = $\text{dist}_s(v) = |\pi(s, v)| = ?$
 - 退化情况：可能有多条 //任取其中一条
- 在BFS过程中的任一时刻
 - 队列中顶点按 $\text{dist}_s()$ 单调排列
 - 队列中相邻顶点， $\text{dist}_s()$ 相差不超过1 //而且...
 - 队首、队末顶点， $\text{dist}_s()$ 相差不超过1
 - 由树边联接的顶点， $\text{dist}_s()$ 恰好相差1
 - 由跨边联接的顶点， $\text{dist}_s()$ 至多相差1 //至少相差2的情况呢？
- BFS树中从s到v的路径，即是 $\pi(s, v)$

Data Structures & Algorithms (Fall 2012), Tsinghua University

10

算法

- DFS(s) //始自顶点s的深度优先搜索 (Depth-First Search)
 - 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
- 对于树而言，等同于先序遍历
 - 事实上，DFS也的确会构造出原图的一棵支撑树 (DFS tree)

Data Structures & Algorithms (Fall 2012), Tsinghua University

11

6. 图

(d) 深度优先搜索

邓俊辉

deng@tsinghua.edu.cn

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
                status(v, u) = TREE; parent(u) = v; DFS(u, clock); break;
            case DISCOVERED: //u已被发现但尚未访问完毕，应属被后代指向的祖先
                status(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
                status(v, u) = dTime(v) < dTime(u) ? FORWARD : CROSS; break;
        }
    status(v) = VISITED; fTime(v) = ++clock; //至此，当前顶点v方告访问完毕
}

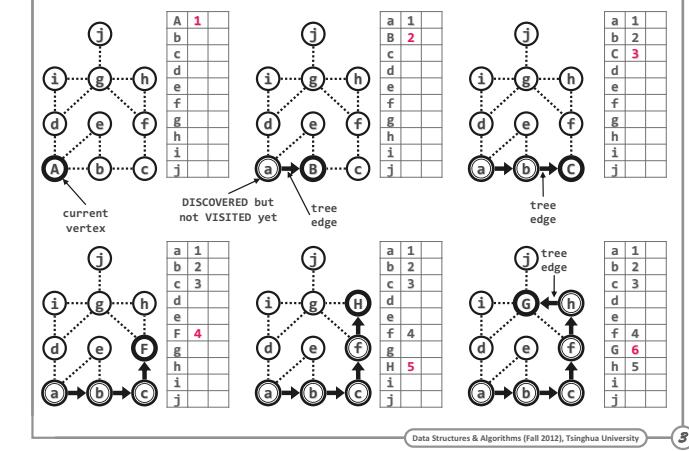
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

12

401

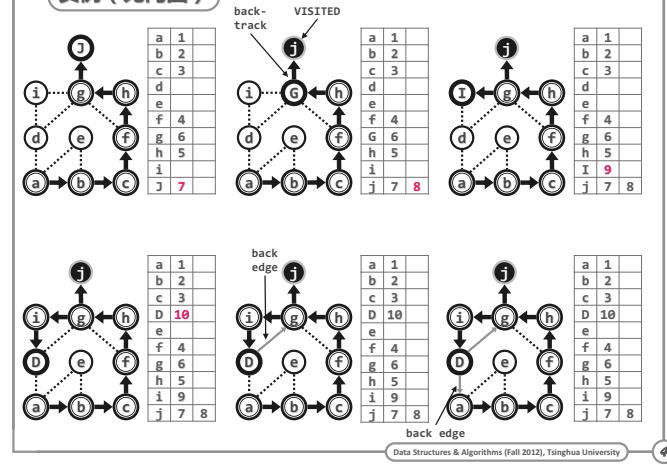
实例(无向图)



Data Structures & Algorithms (Fall 2012), Tsinghua University

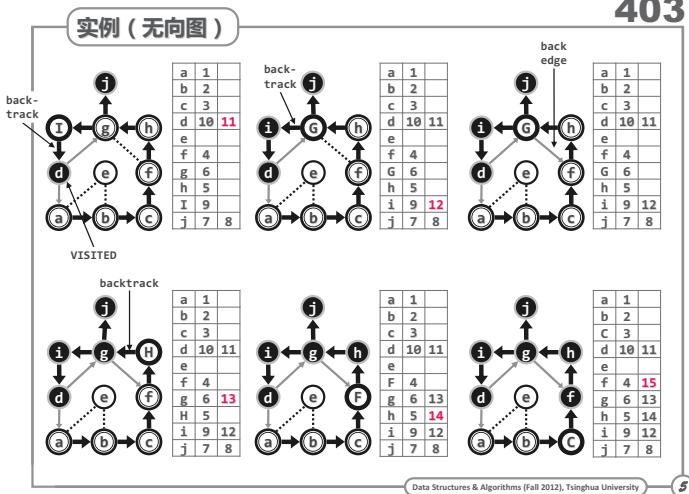
402

实例(无向图)



Data Structures & Algorithms (Fall 2012), Tsinghua University

403

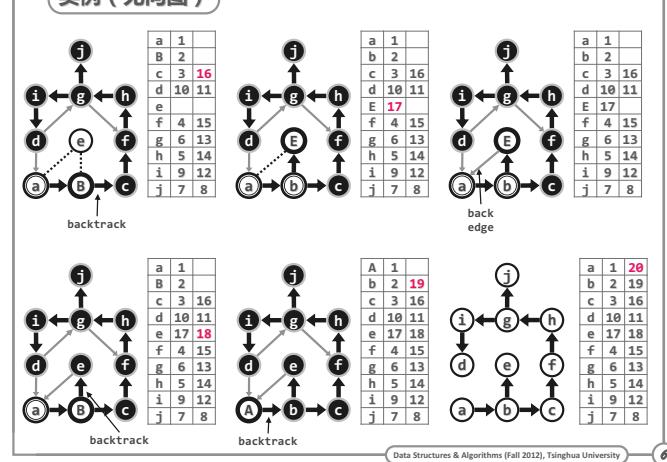


Data Structures & Algorithms (Fall 2012), Tsinghua University

Data Structures & Algorithms (Fall 2012), Tsinghua University

404

实例(无向图)



Data Structures & Algorithms (Fall 2012), Tsinghua University

Data Structures & Algorithms (Fall 2012), Tsinghua University

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
}
```

同样，无论共有多少连通/可达分量，dfs()自身仅耗时 $\Theta(n)$

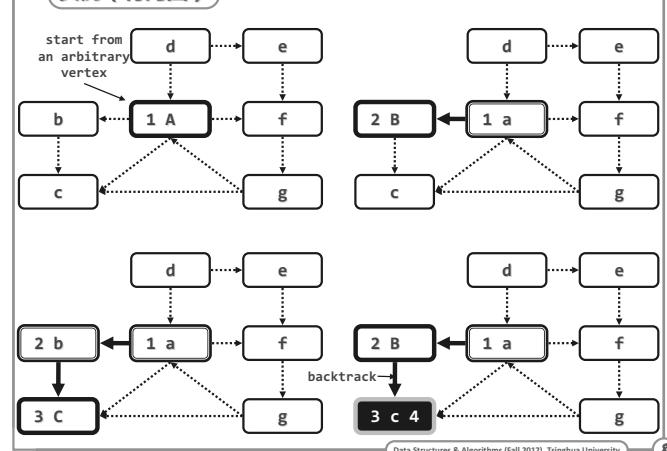
Data Structures & Algorithms (Fall 2012), Tsinghua University

Data Structures & Algorithms (Fall 2012), Tsinghua University

405

实例(有向图)

实例(有向图)

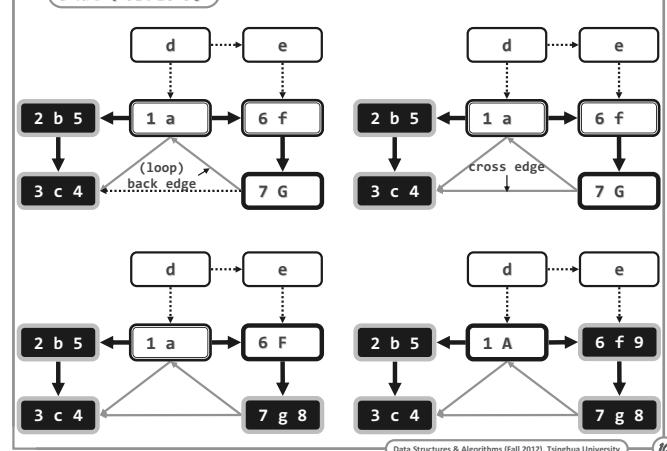


Data Structures & Algorithms (Fall 2012), Tsinghua University

407

实例(有向图)

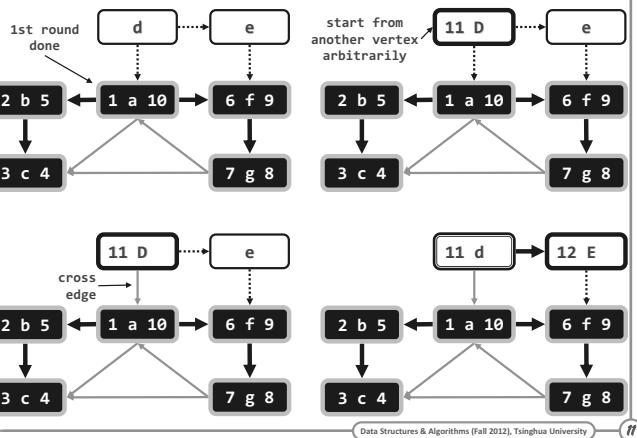
实例(有向图)



Data Structures & Algorithms (Fall 2012), Tsinghua University

Data Structures & Algorithms (Fall 2012), Tsinghua University

实例(有向图)



Data Structures & Algorithms (Fall 2012), Tsinghua University

复杂度

- 考查有向图
 - 每个顶点各对应1次DFS()调用 //只计当前层，不计递归
 - 每次DFS(G, u)调用需要检查outDeg(u)条邻边

- 邻接矩阵 检查u的所有邻边， $\Theta(n)$

$$\text{总共 } \sum_u (1 + n) = n + n^2 = \Theta(n^2)$$

- 邻接表 检查u的所有邻边， $\Theta(1+outDeg(u))$

$$\begin{aligned} \text{总共} \quad & n + \sum_u (1+outDeg(u)) \\ & = 2n+e = \Theta(n+e) = \Theta(n^2) \end{aligned}$$

- 无向图呢？

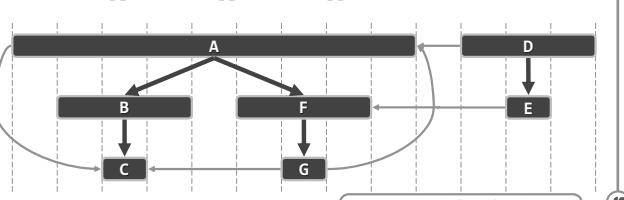
...

Data Structures & Algorithms (Fall 2012), Tsinghua University

- 顶点的活动期： $active[u] = (dTime[u], fTime[u])$

- 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 $active[u] \subseteq active[v]$
 - u 是 v 的祖先 iff $active[u] \supseteq active[v]$
 - u 和 v “无关” iff $active[u] \cap active[v] = \emptyset$

- 借助status[] + dTime[] + fTime[]，即可对各边分类...



Data Structures & Algorithms (Fall 2012), Tsinghua University

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

两个顶点之间可达性检测/路径求解

DFS/BFS

两个顶点之间的最短距离

BFS

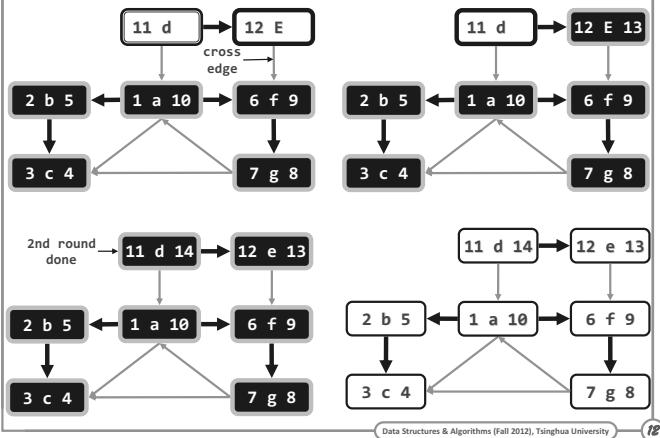
Eulerian tour

DFS

...

Data Structures & Algorithms (Fall 2012), Tsinghua University

实例(有向图)



Data Structures & Algorithms (Fall 2012), Tsinghua University

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

知道森林的“全部”信息了吗？

知道原图的“全部”信息了吗？

就某种意义而言，是的...

Data Structures & Algorithms (Fall 2012), Tsinghua University

边分类

- TREE(v, u) : 

可以从当前顶点v进入处于UNDISCOVERED状态的u

- BACKWARD(v, u) : 

试图从当前顶点v进入处于DISCOVERED状态的u //允许u==v，即自环

DFS发现后向边 iff 存在回路 //后向边数==回路数？

- FORWARD(u, v) : 

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(u, v) : 

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？

Data Structures & Algorithms (Fall 2012), Tsinghua University

6. 图

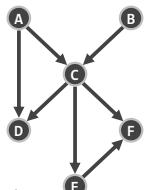
(e) 拓扑排序

邓俊辉

deng@tsinghua.edu.cn

有向无环图

- DAG
(Directed Acyclic Graph)



应用

- 类派生和继承关系图中，是否存在循环定义
- 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
- ...

Data Structures & Algorithms (Fall 2012), Tsinghua University

I

存在性

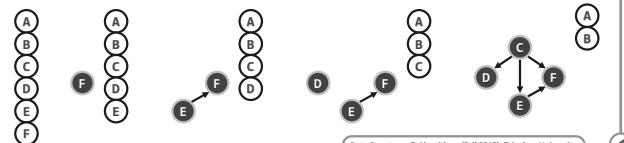
- 每个DAG对应于一个偏序集；拓扑排序对应于一个全序集
所谓的拓扑排序，即构造一个与指定偏序集相容的全序集
- 可以拓扑排序的有向图，必定无环 //反之
任何DAG，都存在（至少）一种拓扑排序？是的！ //为什么...
- 有限偏序集必有极小/极大元素
任何DAG都存在（至少）一种拓扑排序 //归纳证明...
- 1. 任何DAG必有（至少一个）顶点入度为零 //记作m
- 2. 若DAG\{m}存在拓扑排序S = <u_{k1}, ..., u_{k(n-1)}> //subtraction
则S' = <m, u_{k1}, ..., u_{k(n-1)}>即为DAG的拓扑排序 //DAG子图亦为DAG
- 只要m不唯一，拓扑排序也应不唯一 //反之呢？

Data Structures & Algorithms (Fall 2012), Tsinghua University

B

算法A：顺序输出零入度顶点

将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 (inDegree(u) < 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 |Q| < |G| ? "NOT_DAG" : Q; //子图非DAG，超图必非DAG

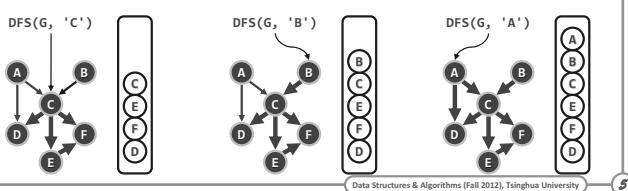


Data Structures & Algorithms (Fall 2012), Tsinghua University

4

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O(n+e)



Data Structures & Algorithms (Fall 2012), Tsinghua University

5

6. 图

(f) 优先级搜索

郑俊辉

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：优先访问与更早被发现的顶点相邻接者
深度：优先访问与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priority(v)
每个顶点都有初始优先级数；并可能随算法的推进而变化
- 通常的习惯是，优先级数越大/小，优先级越低/高
比如：priority(v) == INT_MAX意味着v的优先级最低

Data Structures & Algorithms (Fall 2012), Tsinghua University

f

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
    reset(); //初始化复位
    priority(s) = 0; status(s) = VISITED; parent(s) = -1; //s加入遍历树
    for (int i = 1; i < n; i++) { //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
        if (status(w) == UNDISCOVERED) //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
        status(s) = VISITED; status(parent(s), s) = TREE; //将s加入遍历树
    }
}

```

Data Structures & Algorithms (Fall 2012), Tsinghua University

6. 图
(g) Prim算法

邓俊辉

deng@tsinghua.edu.cn

MST

◆ 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 //具体算法稍后介绍
为许多NP问题提供足够好的近似解 //比如，TSP

算法

Boruvka-1926, Jarnik-1930/Prim-1956, Kruskal-1956, ...
Karger-Klein-Tarjan, 1995
B. Chazelle, 2000
...是否存在 $\Theta(n+e)$ 线性算法？

Data Structures & Algorithms (Fall 2012), Tsinghua University

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...
◆ 包含n个互异顶点的图，对应的支撑树可能有多少棵？

| | |
|-------|-----|
| n = 1 | 1 |
| n = 2 | 1 |
| n = 3 | 3 |
| n = 4 | 16 |
| ... | ... |



◆ Cavlev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

Data Structures & Algorithms (Fall 2012), Tsinghua University

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 $\Theta(n+e)$
- ◆ 后一循环的迭代长度，呈算术级数变化：n, n-1, ..., 3, 2, 1
累计为 $\Theta(n^2)$
- ◆ 两项合计，为 $\Theta(n^2)$
- ◆ 后面我们将会看到
若采用优先队列，以上两项将分别是 $\Theta(e\log n)$ 和 $\Theta(n\log n)$
- ◆ 这将是很大的改进，尤其是对稀疏图而言 //保持兴趣；继续改进
- ◆ 基于这个统一框架，如何解决具体的应用问题...

Data Structures & Algorithms (Fall 2012), Tsinghua University

最小 + 支撑 + 树

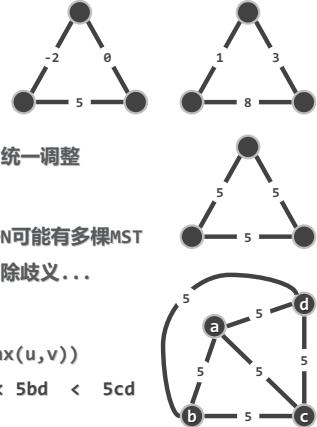
- ◆ 连通网络N = (V; E)的子图T = (V; F)
- 1) 支撑/spanning
覆盖N中所有顶点
- 2) 树/tree
连通且无环， $|V| = |F| + 1$
加边出单环，再删同环边即恢复为树
删边不连通，再加联接边即恢复为树
不难验证，同一网络的支撑树不唯一
- 3) 最小/minimum
各边总权重wt(T) = $\sum_{e \in F} wt(e)$ 达到最小



Data Structures & Algorithms (Fall 2012), Tsinghua University

退化

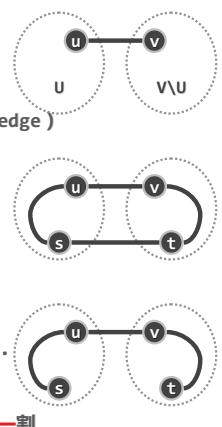
- ◆ 权值必须是正数？
允许为零或负数有什么影响？
- ◆ 所有支撑树中边数相等
通过elevate(1-findMin()) 统一调整
- ◆ The minimum?
A minimal 同一网络N可能有多棵MST
The minimum! 可强制消除歧义...
- ◆ 合成数 (composite number)
($w(u,v)$, $\min(u,v)$, $\max(u,v)$)
 $5ab < 5ac < 5ad < 5bc < 5bd < 5cd$



Data Structures & Algorithms (Fall 2012), Tsinghua University

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 (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



Data Structures & Algorithms (Fall 2012), Tsinghua University

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

Data Structures & Algorithms (Fall 2012), Tsinghua University

6

实例

435

(a)
(b)
(c)

Data Structures & Algorithms (Fall 2012), Tsinghua University

8

实例

437

(g)
(h)
(i)

Data Structures & Algorithms (Fall 2012), Tsinghua University

10

实现

439

对于 V_k 外各顶点 u ，令 $\text{priority}(u) = u$ 到 V_k 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} 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

$\text{priority}(v) = \min(\text{priority}(v), |u_k, v|)$

为此，需按照`prioUpdater()`模式，编写一个

优先级（数）更新器...

Data Structures & Algorithms (Fall 2012), Tsinghua University

12

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 $e_k = (v_k, u_k)$ 中，找出极短者

令 $T_{k+1} = (V_{k+1}; E_{k+1}) = (V_k \cup \{u_k\}; E_k \cup \{e_k\})$

Data Structures & Algorithms (Fall 2012), Tsinghua University

436

实例

(d)
(e)
(f)

Data Structures & Algorithms (Fall 2012), Tsinghua University

438

正确性

设Prim依次选取边 $\{e_2, e_3, \dots, e_n\}$ ，构造出树 T

则其中每一条边 e_k 都属于某棵MST

— 的确如此，但在MST不唯一时并不充分

— 满足这一性质的一组边，未必构成一棵MST

反例...

可行的证明方法...

1) 在不增加总权重的前提下，可以将任一MST转换为 T
数学归纳

2) 每一 T_k 都是某棵MST的子树， $1 \leq k \leq n$

数学归纳

Data Structures & Algorithms (Fall 2012), Tsinghua University

440

实现

`g->pfs(0, PrimPU<char, int>()); //调用`

`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) {
    if(g->status(v) == UNDISCOVERED)
```

`// 对uk的每个尚未被发现的邻居v，按Prim策略做松弛`

```
    if(g->priority(v) > g->weight(uk, v)) {
```

```
        g->priority(v) = g->weight(uk, v);
```

```
        g->parent(v) = uk;
```

```
}
```

```
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

13

6. 图

(h) Dijkstra 算法

邓俊辉

deng@tsinghua.edu.cn



Data Structures & Algorithms (Fall 2012), Tsinghua University

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?

❖ 单源点到各顶点的最短路径

Single-source shortest paths

给定顶点x, 计算x到其余各个顶点的最短路径及长度

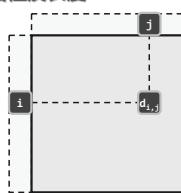
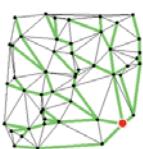
E. Dijkstra, 1959

❖ 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

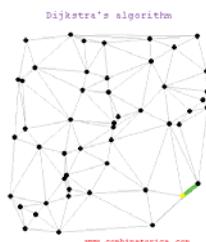
Floyd-Warshall, 1962



Data Structures & Algorithms (Fall 2012), Tsinghua University

E. W. Dijkstra

❖ Turing Award, 1972

E. W. Dijkstra
(1930 - 2002)

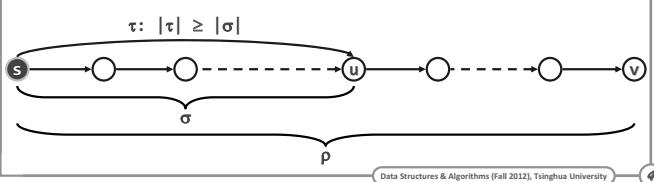
SPT

1) 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

2) 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

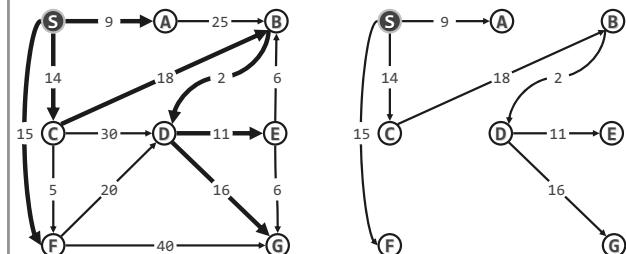
3) 就同一起点s而言, 所有最短路径的并, 不含回路



Data Structures & Algorithms (Fall 2012), Tsinghua University

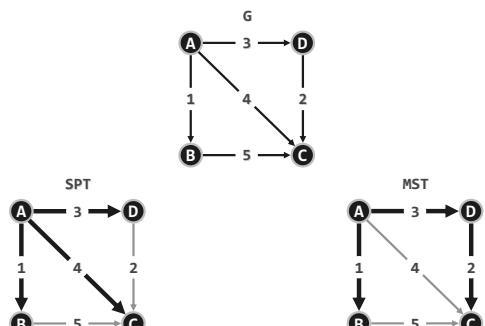
SPT

❖ 因此, 所有最短路径的并, 构成一棵树 (shortest path tree)



Data Structures & Algorithms (Fall 2012), Tsinghua University

SPT ≠ MST



Data Structures & Algorithms (Fall 2012), Tsinghua University

 u_1

❖ 按照到s的最短距离, 对其余的顶点排序

$$\text{dist}(u_1, s) \leq \text{dist}(u_2, s) \leq \dots \leq \text{dist}(u_{n-1}, s)$$

❖ 最短距离最短者 $u_1 = ?$

❖ 沿任一最短路径, 各顶点到s的最短距离单调变化

❖ u_1 必与s直接相联

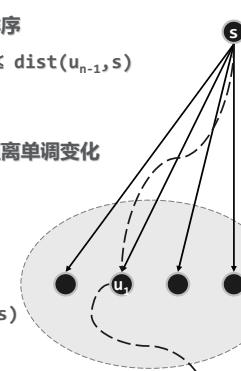
$$\text{dist}(s_1, x) = w(s_1, x) < \infty$$

❖ $\forall u \neq s,$

$$w(u, s) < \infty \text{ 仅当 } \text{dist}(u_1, s) \leq w(u, s)$$

❖ 为找到 u_1 , 只需

在与s关联的各顶点中, 找到对应边权值最小者

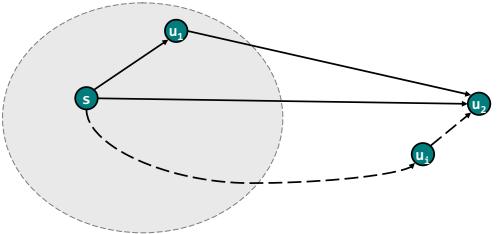


Data Structures & Algorithms (Fall 2012), Tsinghua University

u_2

❖ 最短距离次小的顶点 $u_2 = ?$

❖ 断言: $\text{dist}(u_2, s) = \min\{ w(u_2, s), \text{dist}(u_1, s) + w(u_2, u_1) \}$



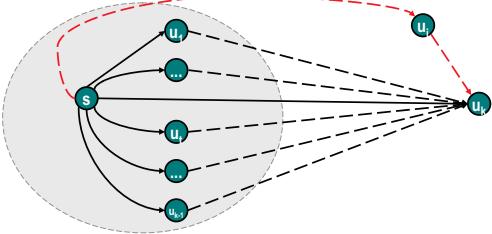
Data Structures & Algorithms (Fall 2012), Tsinghua University

 u_k

❖ $u_3 = ?, u_4 = ?, \dots, u_k = ?$

❖ 断言: $\text{dist}(u_k, s) = \min\{ \text{dist}(u_i, s) + w(u_k, u_i) \mid 0 \leq i < k \}$
其中, $u_0 = s$

❖ 算法?



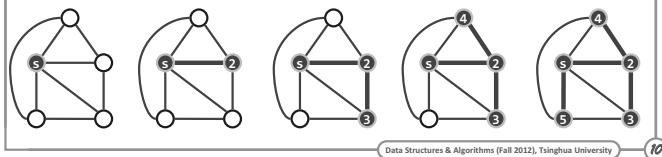
Data Structures & Algorithms (Fall 2012), Tsinghua University

算法

❖ 从 $T_1 = (\{v_1\}; \emptyset)$ 开始, 逐步构造 T_2, T_3, \dots, T_n , 其中

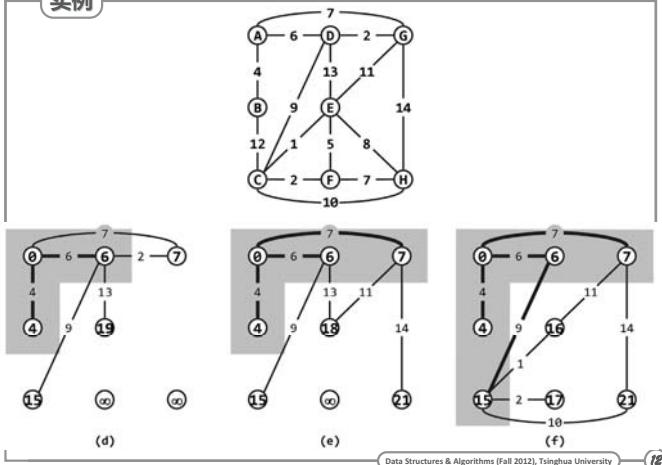
 $v_1 = s$ $T_k = (V_k; E_k), |V_k| = k, |E_k| = k-1, V_k \subset V_{k+1}$

❖ 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 $e_k = (v_k, u_k)$ 中, 找出到 s 距离最近的 u_k 令 $T_{k+1} = (V_{k+1}; E_{k+1}) = (V_k \cup \{u_k\}; E_k \cup \{e_k\})$ 

Data Structures & Algorithms (Fall 2012), Tsinghua University

实例



Data Structures & Algorithms (Fall 2012), Tsinghua University

实现

❖ 对于 V_k 外各顶点 u , 令 $\text{priority}(u) = u$ 到 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

 $\text{priority}(v) = \min(\text{priority}(v), \text{priority}(u_k) + |u_k, v|)$

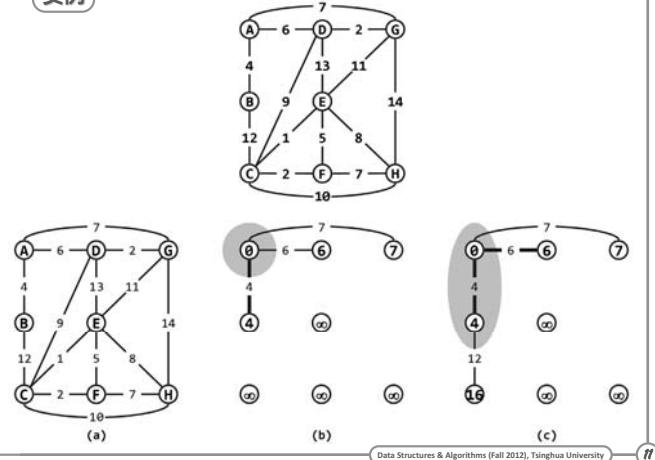
❖ 为此, 需按照 `prioUpdater()` 模式, 编写一个

优先级 (数) 更新器...

Data Structures & Algorithms (Fall 2012), Tsinghua University

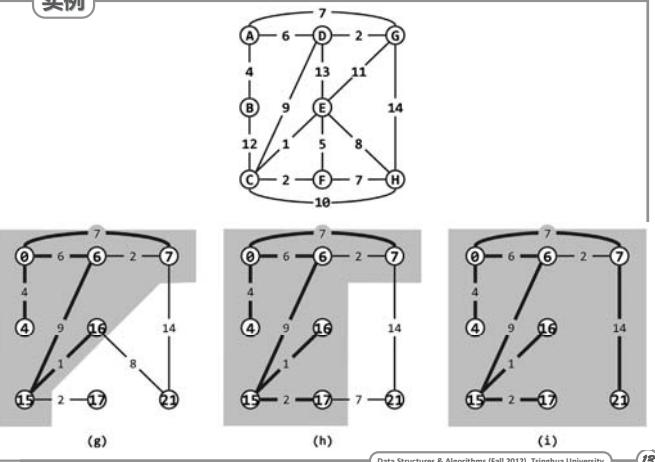
Data Structures & Algorithms (Fall 2012), Tsinghua University

实例



Data Structures & Algorithms (Fall 2012), Tsinghua University

实例



Data Structures & Algorithms (Fall 2012), Tsinghua University

```
❖ g->pfs(0, DijkstraPU<char, int>()); //调用
```

```
❖ 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) {
    if(g->status(v) != UNDISCOVERED) return;
    // 对uk的每个尚未被发现的邻居v, 按Dijkstra策略做松弛
    if(g->priority(v) > g->priority(uk) + g->weight(uk, v)) {
        g->priority(v) = g->priority(uk) + g->weight(uk, v);
        g->parent(v) = uk;
    }
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

Data Structures & Algorithms (Fall 2012), Tsinghua University

Data Structures & Algorithms (Fall 2012), Tsinghua University

6. 图

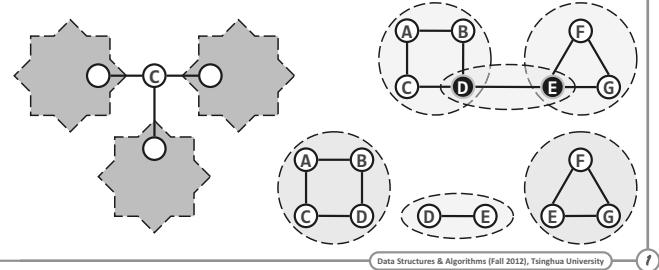
(x1) 关节点与双连通分量

邓俊辉

deng@tsinghua.edu.cn

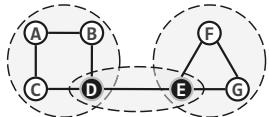
关节点 & 双连通分量

- ❖ 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



Brute-Force

- ❖ 给定无向图，如何确定各BCC？如何确定关节点？ //先考查后一问题

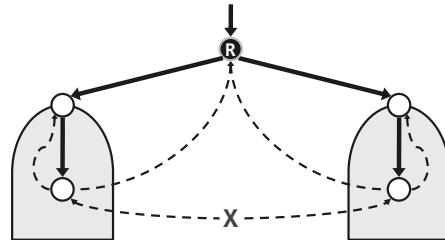


- ❖ 蛮力：对每一顶点 v ，通过遍历检查 $G \setminus \{v\}$ 是否连通
共需 $\Theta(n(n+e))$ 时间，太慢！
- ❖ 改进：从任一顶点出发，构造DFS树
根据DFS留下的标记，甄别是否关节点
- ❖ 比如，叶顶点绝不可能是关节点 //为什么？
- ❖ 那么，一般的顶点呢？

Data Structures & Algorithms (Fall 2012), Tsinghua University

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

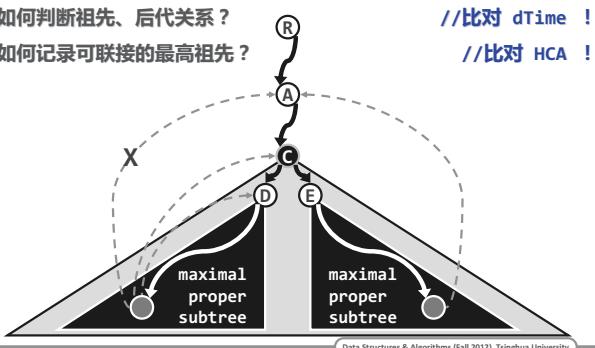


Data Structures & Algorithms (Fall 2012), Tsinghua University

内顶点

- ❖ 内顶点 C 是关节点 iff C 的某棵极大真子树与 C 的真祖先不属于同一BCC
在 C 的某棵极大真子树中，没有顶点（经后向边）联接到 C 的真祖先

- ❖ 如何判断祖先、后代关系？
如何记录可联接的最高祖先？



switch (status(u))

- ```

❖ case UNDISCOVERED:
 // 从顶点u处深入
 parent(u) = v; status(v, u) = TREE; BCC(u, clock, S);
// 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以下即是一个双连通域，且其中的顶点此时正集中记录于栈S的顶部
 //故可依次弹出当前BCC中的节点，或根据实际需求转存至其它结构
 while (v != S.pop());
 S.push(v); //最后一个顶点（关节点）重新入栈—总计至多两次
}
break;

```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## switch (status(u))

- ```

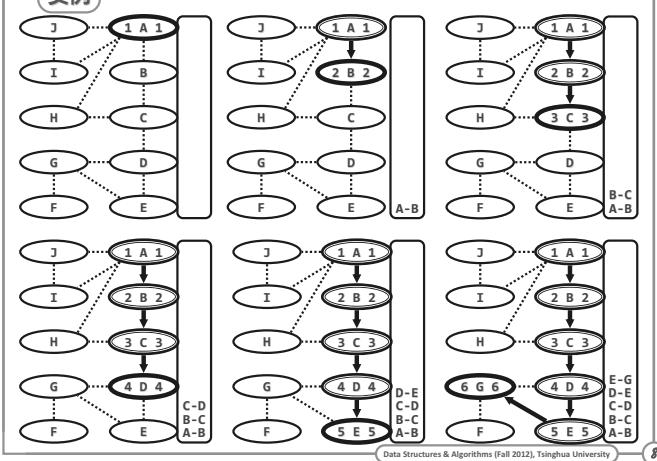
❖ case DISCOVERED:
    status(v, u) = BACKWARD;
    if (u != parent(v))
        hca(v) = min(hca(v), dTime(u)); //更新hca[v]—越小越高
    break;
default: //VISITED (digraphs only)
    status(v, u) = (dTime(v) < dTime(u)) ? FORWARD : CROSS;
    break;

```

Data Structures & Algorithms (Fall 2012), Tsinghua University

465

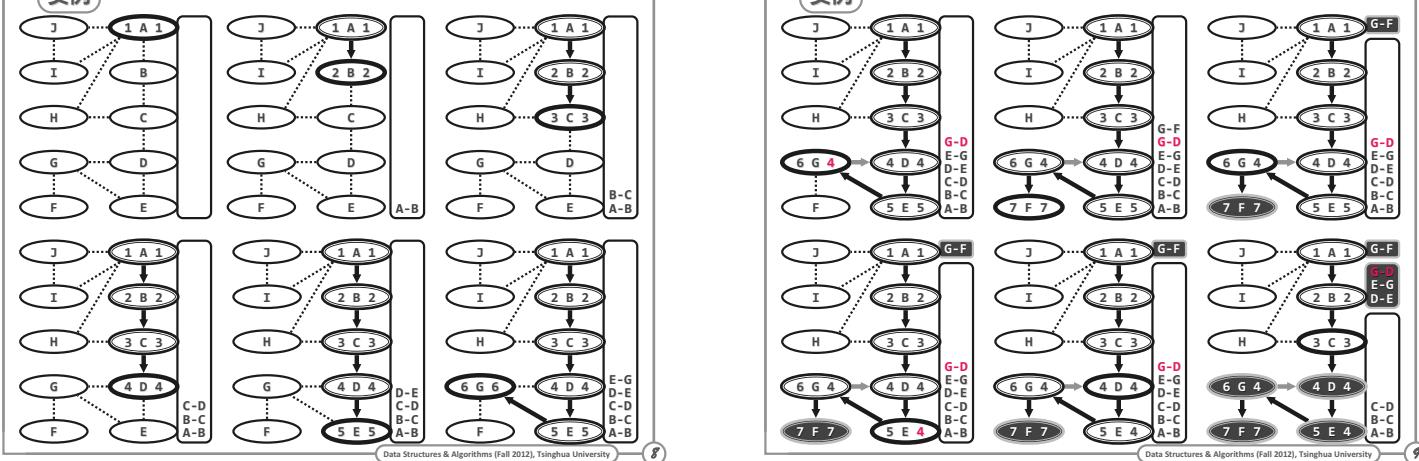
实例



Data Structures & Algorithms (Fall 2012), Tsinghua University

8

466

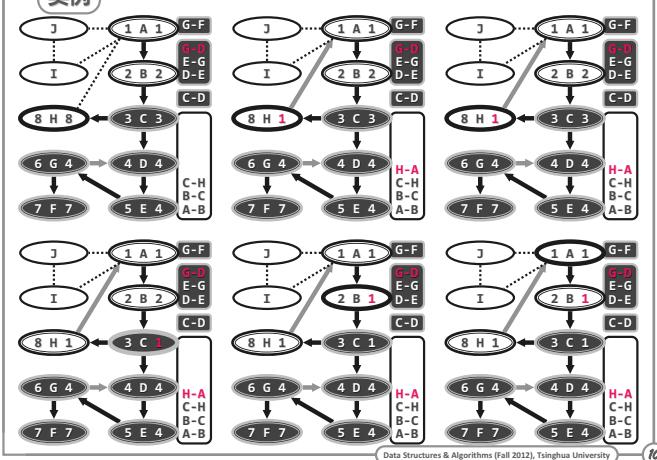


Data Structures & Algorithms (Fall 2012), Tsinghua University

9

467

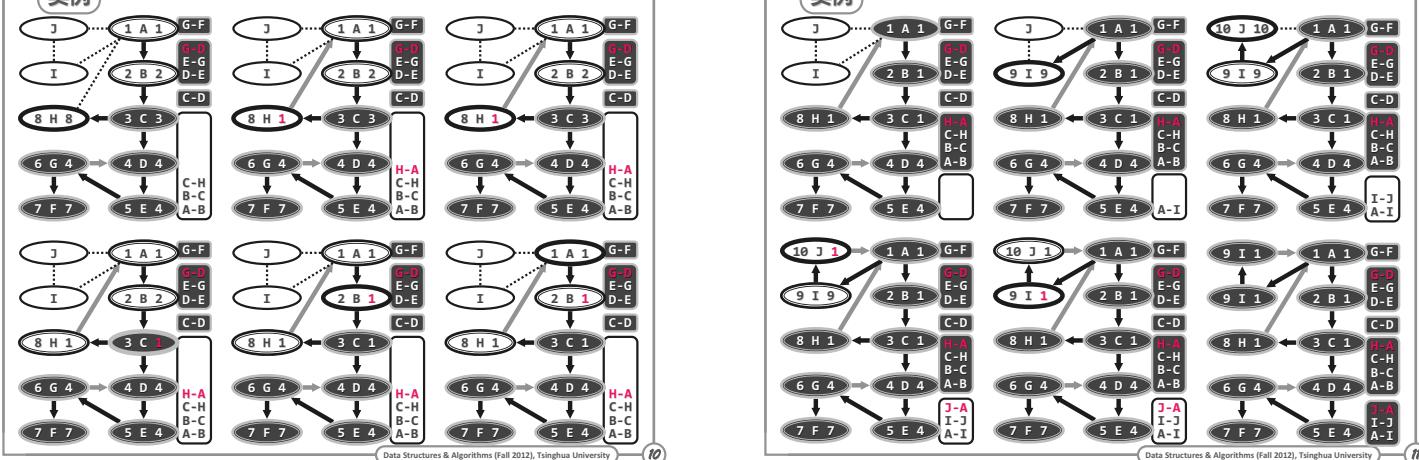
实例



Data Structures & Algorithms (Fall 2012), Tsinghua University

10

468



Data Structures & Algorithms (Fall 2012), Tsinghua University

11

469

复杂度

- 运行时间与常规的DFS相同，也是 $\mathcal{O}(n+m)$
- 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 $\mathcal{O}(m)$ 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？

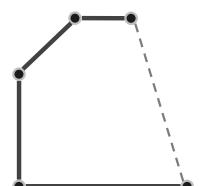
//strongly-connected component

Data Structures & Algorithms (Fall 2012), Tsinghua University

12

贪心策略

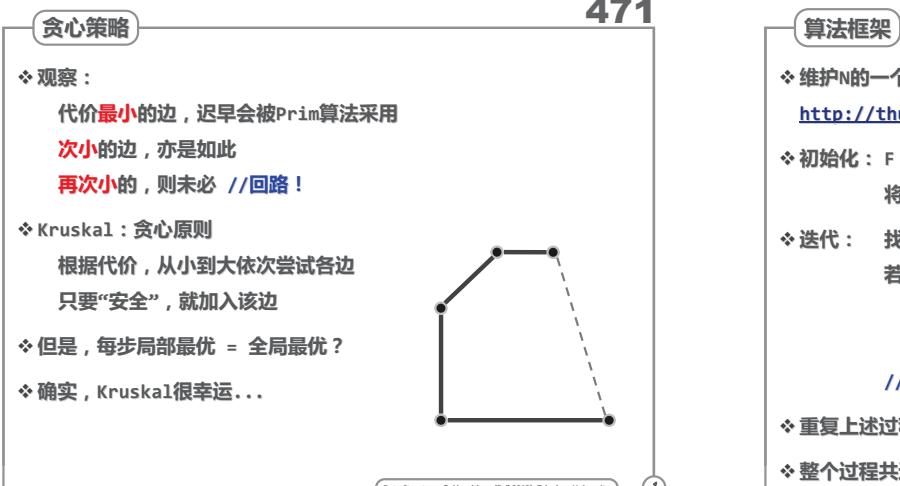
- 观察：
代价**最小**的边，迟早会被Prim算法采用
次小的边，亦是如此
再次小的，则未必 //回路！
- Kruskal：贪心原则
根据代价，从小到大依次尝试各边
只要“安全”，就加入该边
但是，每步局部最优 = 全局最优？
确实，Kruskal很幸运...



Data Structures & Algorithms (Fall 2012), Tsinghua University

471

实例



Data Structures & Algorithms (Fall 2012), Tsinghua University

13

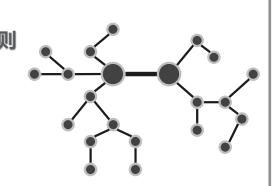
472

算法框架

- 维护N的一个森林： $F = (V; E') \subseteq N = (V; E)$
<http://thudsa.3322.org/~deng/ds/demo/mst/>
- 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

Data Structures & Algorithms (Fall 2012), Tsinghua University

14



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

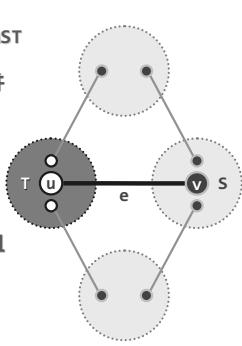
❖ 在确定应引入 e 之前

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



Data Structures & Algorithms (Fall 2012), Tsinghua University

3

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

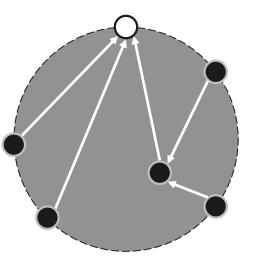
沿parent指针可找到对应的首领

$\text{leader.parent} = \text{NO_PARENT}$

❖ 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？



Data Structures & Algorithms (Fall 2012), Tsinghua University

5

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

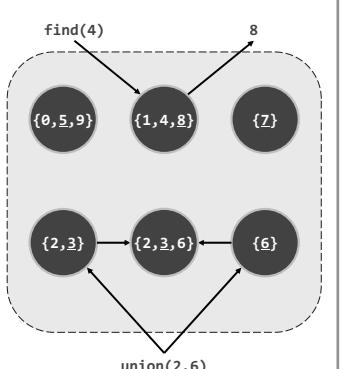
$\text{Find}(x)$

找到元素 x 所属等价类

$\text{Union}(x, y)$

合并 x 和 y 所属等价类

❖ Kruskal = Union-Find



Data Structures & Algorithms (Fall 2012), Tsinghua University

7

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 = $\mathcal{O}(elog e) = \mathcal{O}(elog n)$

❖ 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Fibonacci堆，对稀疏图可做到 $\mathcal{O}(elog \log n)$

Data Structures & Algorithms (Fall 2012), Tsinghua University

9

排序

❖ 需做全排序吗？

若做，将耗时 $\mathcal{O}(elog e) = \mathcal{O}(n^2 \log n)$ //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， $\mathcal{O}(e)$

删除并还原， $\mathcal{O}(\log e)$

共迭代 $\mathcal{O}(e)$ 次 //实际中往往远小于 e ，尤其在稠密图时

总共 = $\mathcal{O}(e) + \mathcal{O}(\log e) \times \mathcal{O}(e) = \mathcal{O}(elog e)$

Data Structures & Algorithms (Fall 2012), Tsinghua University

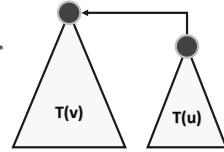
树合并

❖ 合并树 $T(u)$ 和 $T(v)$ 后，令 $\text{leader}(u).\text{parent} = \text{leader}(v)$

❖ 注意：合并后，树的深度 = $\mathcal{O}(n)$

总会不幸达到上界吗？很有可能！于是 ...

❖ 对leader的 $\mathcal{O}(n)$ 次查询，共需 $\mathcal{O}(n^2)$ 时间



❖ 改进： $\text{leader.parent} = -(\#\text{nodes})$

```
if ( $\text{leader}(u).\text{parent} > \text{leader}(v).\text{parent}$ )
     $\text{leader}(u).\text{parent} = \text{leader}(v);$ 
else  $\text{leader}(v).\text{parent} = \text{leader}(u);$ 
```

❖ 使用改进后算法，树合并后深度 = $\mathcal{O}(\log n)$ //YAN, p.142

❖ 因此，查询leader的总复杂度 = $\mathcal{O}(n \log n)$

Data Structures & Algorithms (Fall 2012), Tsinghua University

Union-Find

❖ [Tarjan-83] : $\mathcal{O}(\alpha(n))$ amortized time per Union/Find

❖ $\alpha(n)$: inverse Ackermann function

$\log^* n = \# \text{logs } s.t. \log(\log(\dots(\log n)\dots)) < 2$

$\log^{**} n = \# \text{logs}^* s.t. \log^*(\log^*(\dots(\log^* n)\dots)) < 2$

...

$\log^{***} n = \dots$

...

❖ $\alpha(n) = \# \text{stars} s.t. \log^{***} n < 3$

❖ $\alpha(\text{目前可观察到宇宙范围内的粒子总数}) < 4$

Data Structures & Algorithms (Fall 2012), Tsinghua University

其它算法

❖ 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

Data Structures & Algorithms (Fall 2012), Tsinghua University

10

更新结果

- ❖ Gabow, Galil, Spencer, & Tarjan : $\Theta(m \times \log(\beta(m, n)))$
 Efficient algorithms for finding minimum spanning trees
 in undirected and directed graphs
Combinatorica, vol. 6, 1986, pp. 109–122
 $\beta(m, n) = \text{smallest } i$
 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
31st IEEE Symp. Foundations of Comp. Sci., 1990
 pp. 719–725

Data Structures & Algorithms (Fall 2012), Tsinghua University

17

更新结果

- ❖ YAO (1995) : $\Theta(\epsilon \log \log n)$
 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

Data Structures & Algorithms (Fall 2012), Tsinghua University

18

相关话题

- ❖ Euclidean MST
 Delaunay Triangulation
 Gabriel Graph
 Relative Neighborhood Graph
 $\Omega(n \log n)$
- ❖ Steiner MST
 NP-hard
 Approximation

Data Structures & Algorithms (Fall 2012), Tsinghua University

19

6. 图

(x3) Floyd-Warshall算法

邓俊辉

deng@tsinghua.edu.cn

多起点：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radius(G, s) = 所有顶点到s的最大距离
 中心点 = 半径最小的顶点s
- ❖ Dijkstra : 依次将各顶点作为源点，调用Dijkstra算法
 时间 = $n \times \Theta(n^2) = \Theta(n^3)$ — 可否更快？
- ❖ 思路：图矩阵 ~ 最短路径矩阵
- ❖ 效率： $\Theta(n^3)$ ，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）

Data Structures & Algorithms (Fall 2012), Tsinghua University

20

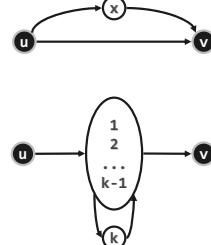
```
weight dist(node* u, node* v, int k) { //蛮力递归
  if (k < 1) return w(u,v); //递归基：中途不经过任何点
  minDist = dist(u, v, k-1); //递归：中途可经过前 k-1 个点
  foreach (node x ∈ {u, v}) { //枚举其余各点，分别作为第 k 个点
    u2x2vDist = dist(u, x, k-1) + dist(x, v, k-1); //递归
    minDist = min(minDist, u2x2vDist); //优化
  }
  return minDist;
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

21

多起点：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, x) + 最短路径(x, v)
- ❖ 将所有顶点随意编号：
 1, 2, ..., 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 $k = 0$)
= $\min(d^{k-1}(u, v), d^{k-1}(u, k) + d^{k-1}(k, v))$ (if $k \geq 1$)



Data Structures & Algorithms (Fall 2012), Tsinghua University

22

动态规划

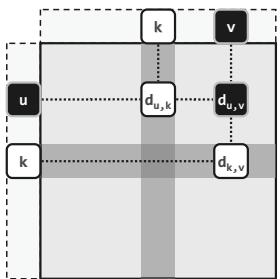
- ❖ 复杂度
 $T(n, 0) = 1$
 $T(n, k) = (n-2) \times 2 \times T(k-1) = 2^k (n-2)^k$
 $T(n, n-2) = 2^{n-2} (n-2)^{n-2}$
= $\Theta(n^n)$ //这还仅仅只是一对节点所需的时间
- ❖ 注意：蛮力算法中，存在大量的重复递归调用
- ❖ 能否避免这些重复计算？如何避免？
- ❖ 动态规划 (dynamic programming)
 维护一张表，记录需要多次计算的数值 //如此，只需计算一次

Data Structures & Algorithms (Fall 2012), Tsinghua University

23

算法

```
//Initialization
for u = 1 to n
for v = 1 to n {
    dist[u][v] = w[u][v];
    midV[u][v] = 0;
}
//Iteration
for k = 1 to n
for u = 1 to n
for v = 1 to n
if (dist[u][v] > dist[u][k] + dist[k][v])
{ dist[u][v] = dist[u][k] + dist[k][v]; midV[u][v] = k; }
```



Data Structures & Algorithms (Fall 2012), Tsinghua University

5

7. 二叉搜索树

(a) 接口

There's nothing in your head
the sorting hat can't see.
So try me on and I will tell
you where you ought to be.



- Harry Potter and the Sorcerer's Stone

邓俊辉

deng@tsinghua.edu.cn

循关键码访问

- 数据对象统一地表示和实现为词条 (entry) 形式
- 不同词条之间依照各自的关键码项 (key) 相互区分：call-by-key
- 关键码之间支持 (大小) 比较与 (相等) 比对

```
template <typename K, typename V> // (key, value)
struct Entry { //词条模板类
    K key; V value; //关键码、数值
    Entry(K k, 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; } //小于
    bool operator>(Entry<K, V> const& e) { return key > e.key; } //大于
    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
};
```

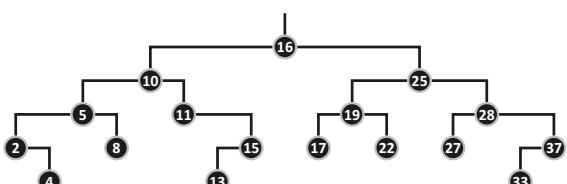
Data Structures & Algorithms (Fall 2012), Tsinghua University

6

中序遍历序列

由以上顺序性，BST的中序遍历序列完全顺序

这一性质，也是BST的充要条件



2 4 5 8 10 11 13 15 16 17 19 22 25 27 28 33 37

Data Structures & Algorithms (Fall 2012), Tsinghua University

6

复杂度

时间 $\Theta(n^2) + \Theta(n^3) = \Theta(n^3)$

与n次调用Dijkstra相同

空间

存储一张 $n \times n$ 的表格， $\Theta(n^2)$

每个单元为两个整数

对于稀疏图和稠密图，你会分别选择哪种算法？

根据 $midV[]$ 矩阵

如何重构出u和v之间的最短路径？需要多长时间？

可在 $\Theta(n)$ 时间内完成 //具体算法…

Data Structures & Algorithms (Fall 2012), Tsinghua University

查找

按照事先约定的规则，在一组数据对象中找到符合特定条件者

属于构建算法的一项基本而重要的操作

若需兼顾静态查找与动态修改，能否综合现有方法的优点？

| 采用结构 | 查找 | 插入/删除 |
|------|------------------|-----------------------|
| 无序向量 | $\Theta(n)$ | $\Theta(1)/\Theta(n)$ |
| 有序向量 | $\Theta(\log n)$ | $\Theta(n)$ |
| 无序列表 | $\Theta(n)$ | $\Theta(1)$ |
| 有序列表 | $\Theta(n)$ | $\Theta(n)$ |

Data Structures & Algorithms (Fall 2012), Tsinghua University

二叉搜索树

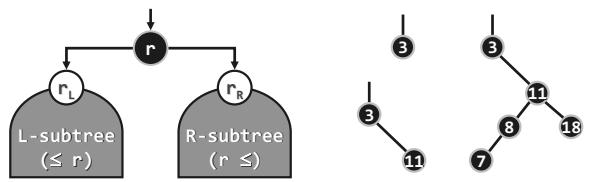
Binary Search Tree

依照词条类所定义的比较器和判等器

任一节点均不大（大）于其左（右）后代

为简化起见，禁止相等的词条并存

任一节点均大（小）于其左（右）后代



Data Structures & Algorithms (Fall 2012), Tsinghua University

BST模板类

```
template <typename T>
class BST : public BinTree<T> { //由BinTree派生
public: //以virtual修饰，以便派生类重写
    virtual BinNodePosi(T) & search(const T& e); //查找
    virtual BinNodePosi(T) insert(const T& e); //插入
    virtual bool remove(const T& e); //删除
protected:
    BinNodePosi(T) _hot; //指向search()最后访问的非空节点
    BinNodePosi(T) rotateAt(BinNodePosi(T) x); //对x做旋转调整
    BinNodePosi(T) connect34(); //按照“3+4”结构联接3个节点及四棵子树
        BinNodePosi(T), BinNodePosi(T), BinNodePosi(T),
        BinNodePosi(T), BinNodePosi(T),
        BinNodePosi(T), BinNodePosi(T));
};
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

6

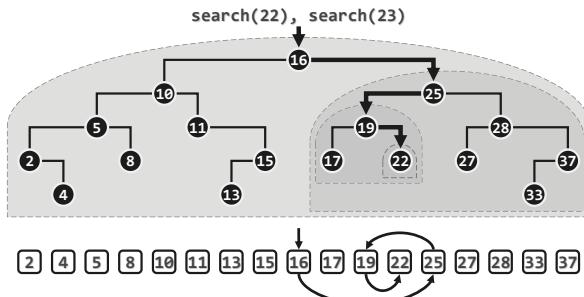
7. 二叉搜索树

(b) 算法及实现

邓俊辉

deng@tsinghua.edu.cn

- ❖ 从根节点出发，递归地不断缩小查找范围，直到发现目标词条（查找成功），或查找范围缩小至空树（查找失败）



- ❖ 由中序遍历序列可见，查找过程与有序向量的**二分查找**过程完全等效

查找：实现

❖ 演示：<http://thudsa.3322.org/~deng/ds/demo/avl/>

```
❖ template <typename T>
BinNodePosi(T) & BST<T>::search(const T& e)
{ return searchIn(_root, e, _hot = NULL); } //从根节点启动查找

❖ static BinNodePosi(T) & searchIn
(BinNodePosi(T) & v, const T& e, BinNodePosi(T) & hot) {
    if (!v || (e == v->data)) return v; //足以确定成功、失败，或者
    hot = v; //先记下当前（非空）节点，然后再
    return //递归查找
    searchIn(((e < v->data) ? v->lChild : v->rChild), e, hot);
} //运行时间正比于最终返回节点v的深度，不超过树高O(h)
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

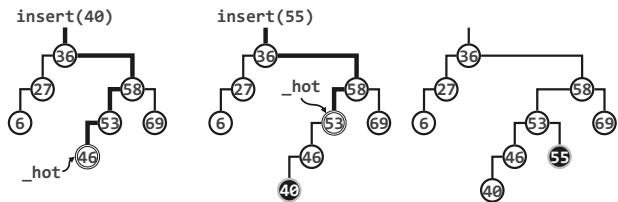
插入：算法

- ❖ 先借助search(e)确定插入位置及方向，再将新节点作为**叶子**插入

- ❖ 若e尚不存在，则

_hot为新节点的父亲
v = search(e)为_hot对新孩子的引用

- ❖ 于是，只需令_hot通过v指向新节点



插入：实现

```
❖ template <typename T>
BinNodePosi(T) BST<T>::insert(const T& e) {
    BinNodePosi(T) & v = search(e); //查找目标（留意_hot的设置）
    if (!v) { //既禁止雷同元素，故仅在查找失败时才实施插入操作
        v = new BinNode<T>(e, _hot); //在v处创建新节点，以_hot为父
        _size++; //更新全树规模
        updateHeightAbove(v); //更新v及其历代祖先的高度
    }
    return v; //无论e是否存在于原树中，至此总有v->data == e
}

// 验证：对于首个节点插入之类的边界情况，以上实现亦可正确处置
// 时间主要消耗于search(e)和updateHeightAbove(v)
// 正比于最终返回节点v的深度，不超过树高O(h)
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

删除：算法

- ❖ 借助search(e)确定目标节点的位置v（不妨设v非空）

分两大类情况，具体实施删除（此后，_hot指向*v原先的父亲）
更新全树规模，并从_hot出发更新历代祖先的高度

- ❖ 时间主要消耗于search()、updateHeightAbove()和succ()，O(h)

```
❖ template <typename T> bool BST<T>::remove(const T& e) {
    BinNodePosi(T) & v = search(e); //定位目标节点
    if (!v) return false; //确认目标存在
    removeAt(v, _hot); //实施具体的删除操作
    _size--; //更新全树规模
    updateHeightAbove(_hot); //更新祖先高度
    return true;
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

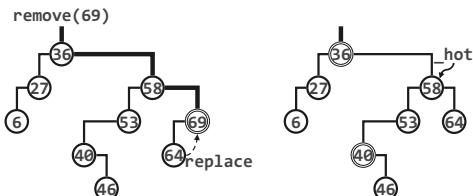
删除：情况一

❖ 若*v (69) 的某一子树为空，则可将其替换为另一子树 (64)

❖ 验证：如此操作之后，二叉搜索树的

拓扑结构依然完整

顺序性依然满足



Data Structures & Algorithms (Fall 2012), Tsinghua University

删除：情况一

```
❖ template <typename T> static BinNodePosi(T)
removeAt(BinNodePosi(T) & v, BinNodePosi(T) & hot) {
    BinNodePosi(T) w = v; //实际被摘除的节点，初值同v
    BinNodePosi(T) succ = NULL; //实际被删除节点的接替者
    if (!HasLChild(*v)) succ = v = v->rChild; //左子树为空
    else if (!HasRChild(*v)) succ = v = v->lChild; //右子树为空
    else { /*...左、右子树并存的情况，略微复杂些...*/ }
    hot = w->parent; //记录实际被删除节点的父亲
    if (succ) succ->parent = hot; //将被删除节点的接替者与hot相联
    release(w->data); release(w); //释放被摘除节点
    return succ; //返回接替者
}

// 此类情况仅需O(1)时间
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

删除：情况二

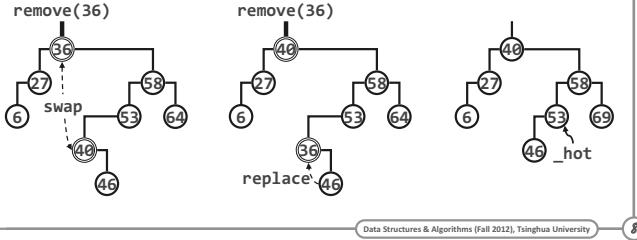
若 $*v$ (36)左、右孩子并存，则

调用BinNode::succ()找到 v 的直接后继 w (必无左孩子)

交换 $*v$ (36)与 $*w$ (40)

于是问题转化为删除 w ，可按前一情况处理

尽管顺序性曾在中途一度不合，但最终必将重新恢复



Data Structures & Algorithms (Fall 2012), Tsinghua University

7. 二叉搜索树

(c) 平衡性与等价性

邓俊辉

deng@tsinghua.edu.cn

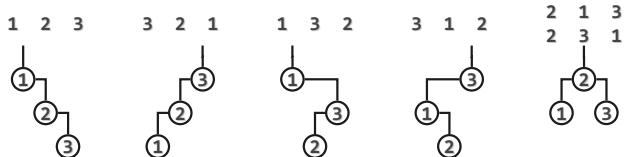
随机生成

考查 n 个互异词条 $\{e_1, e_2, \dots, e_n\}$

对任一排列 $\sigma = (e_{i1}, e_{i2}, \dots, e_{in})$

从空树开始，反复调用insert()接口将各词条依次插入，得到 $T(\sigma)$

与 σ 相对应的 $T(\sigma)$ ，称由 σ “随机生成”(randomly generated by)



若假定任一排列 σ 作为输入的概率均等($1/n!$)

则由 n 个互异词条随机生成的二叉搜索树，平均高度为 $\Theta(\log n)$

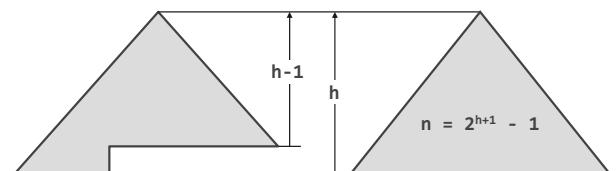
Data Structures & Algorithms (Fall 2012), Tsinghua University

理想平衡与适度平衡

节点数目固定时，左、右子树高度越接近，全树整体也更平衡

由 n 个节点组成的二叉树，高度不低于 $\lfloor \log_2 n \rfloor$

反之，高度恰为 $\lfloor \log_2 n \rfloor$ 时，称作**理想平衡** — 比如，完全树甚至满树



理想平衡很少出现，更难动态维护，故须适当地放宽标准

一般地，高度为 $\Theta(\log n)$ 时，称作**适度平衡**

(适度)平衡的BST，称作**平衡二叉搜索树(BBST)**

Data Structures & Algorithms (Fall 2012), Tsinghua University

删除：情况二

```
template <typename T> static BinNodePosi(T) removeAt(BinNodePosi(T) & v, BinNodePosi(T) & wot) {
    /* ..... */
    else { //若v的左、右子树并存，则
        w = w->succ(); //找到v的直接后继w (pred()亦可，完全对称)
        swap(v->data, w->data); //交换*v和*w的数据元素
        BinNodePosi(T) u = w->parent; //原问题转化为，摘除非二度的w
        ((u == v) ? u->rChild : u->lChild) = succ = w->rChild;
    }
    /* ..... */
}

//此类情况下，时间主要消耗于succ()，正比于v的高度
//更精确地，search()与succ()总共不过O(h)
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

树高

由以上的实现与分析

主要接口search()、insert()和remove()的运行时间

在最坏情况下均线性正比于二叉搜索树的高度

因此，若不能有效地控制树高，则就实际的性能而言

二叉搜索树较之此前的向量和列表等数据结构将无法体现出明显优势

比如在最坏情况下二叉搜索树可能彻底地退化为列表

此时的查找效率甚至会降至 $O(n)$ ，线性正比于树(列表)的规模

那么，出现此类最坏或较坏情况的概率有多大？

或者，从平均复杂度的角度看，二叉搜索树的性能究竟如何呢？

以下按两种常用的随机统计口径，就BST的平均性能做一分析和对比

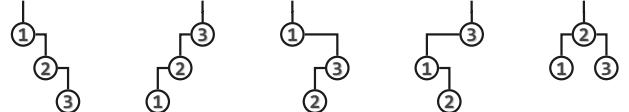
Data Structures & Algorithms (Fall 2012), Tsinghua University

随机组成

n 个互异节点，在遵守顺序性的前提下，可随机确定拓扑联接关系

如此所得BST，称由这组节点“**随机组成**”(randomly composed of)

由 n 个互异节点组成的BST，共计Catalan(n) = $(2n)!/n!(n+1)!$ 棵



若假定这些BST出现的概率均等，则其平均高度为 $\Theta(\sqrt{n})$

前一统计口径中，越低的BST被重复的次数越多 — 故嫌过于乐观

若删除算法固定使用succ()，则每棵BST都有越来越失衡的倾向

理想随机在实际中并不常见，关键码往往按单调甚至等间距的次序出现
因此频繁出现极高的BST也就不足为怪

Data Structures & Algorithms (Fall 2012), Tsinghua University

等价BST

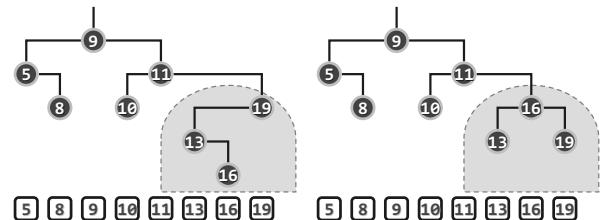
各种BBST的基本构思大致相同：

给出“适度平衡”的具体标准，并依此对所有的搜索树做**等价类划分**

1) 每一等价类都包含至少一棵BBST

2) 经动态修改之后不再平衡的任一搜索树

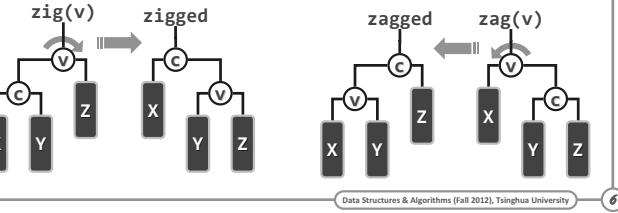
总可以低廉的时间、空间成本，变换为与之等价的一棵BBST



Data Structures & Algorithms (Fall 2012), Tsinghua University

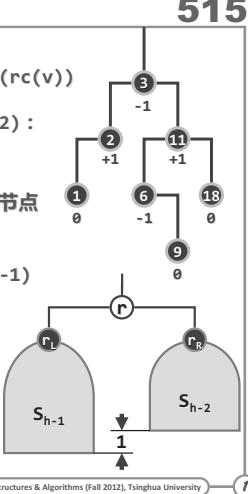
等价变换与局部调整

- ◆ zig和zag均属局部操作，仅涉及常数个节点及其之间的联接关系
- ◆ 调整之后，局部子树的高度可能改变，但上、下幅度不超过1
- ◆ 由同一组共n个关键码组成的BST，经 $\Theta(n)$ 次旋转之后，都可相互转化
- ◆ 只要“平衡性”界定得当，任一刚刚失衡的BBST，都可迅速复衡
—为此只需 $\Theta(\log n)$ 甚至 $\Theta(1)$ 次旋转



平衡因子与AVL平衡

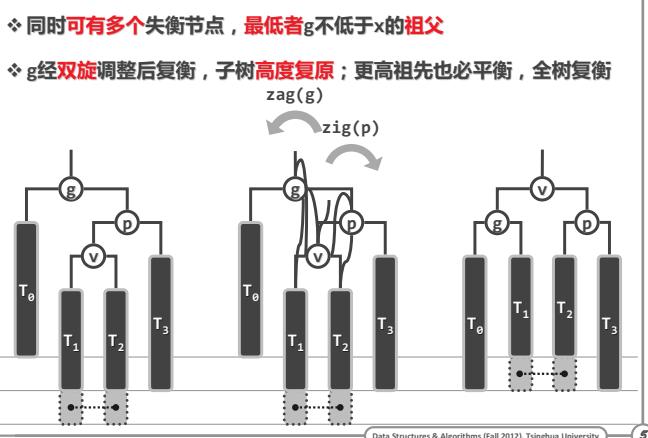
- ◆ $\text{balFac}(v) = \text{height}(\text{lc}(v)) - \text{height}(\text{rc}(v))$
- ◆ G. Adelson-Velsky & E. Landis (1962)：
 $\forall v, |\text{balFac}(v)| \leq 1$
- ◆ 高度为h的AVL树，至少包含 $\text{fib}(h+3)-1$ 个节点
 $|S| \geq 1 + |S_L| + |S_R|$
 $\geq 1 + (\text{fib}(h+2)-1) + (\text{fib}(h+1)-1)$
- ◆ 反过来
由n个节点构成的AVL树，高度为 $\Theta(\log n)$
- ◆ 也就是说
就渐进意义而言，AVL树的确（适度）平衡



失衡与重平衡

- ◆ 按BST规则插入或删除节点之后，AVL平衡性可能破坏 — 如何恢复？
- ◆ 蛮力：由中序遍历序列，按层次遍历次序重构完全树 — 代价太大！
- ◆ 等价变换：时间取决于各次旋转所需时间和旋转次数 — 如何保证？
- ◆ 局部性：所有的旋转都在局部进行
//每次只需 $\Theta(1)$ 时间
- ◆ 快速性：在每一深度只需检查并旋转至多一次
//共 $\Theta(\log n)$ 次

插入：双旋



AVL : 接口

```

#define Balanced(x) \
    (stature((x).lChild) == stature((x).rChild)) //理想平衡
#define BalFac(x) \
    (stature((x).lChild) - stature((x).rChild)) //平衡因子
#define AvlBalanced(x) \
    ((-2 < BalFac(x)) && (BalFac(x) < 2)) //AVL平衡条件

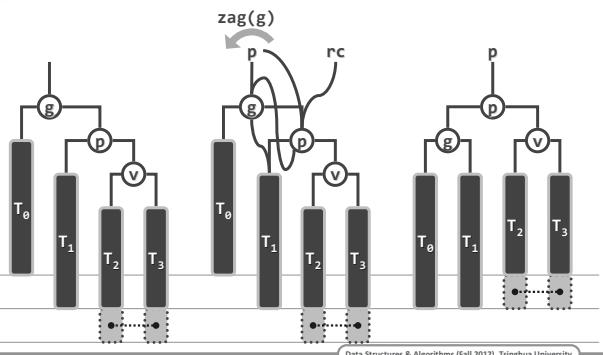
template <typename T>
class AVL : public BST<T> { //由BST派生
public:
    BinNodePosi(T) insert(const T&); //插入(重写)
    bool remove(const T& e); //删除(重写)
    // BST::search()等其余接口可直接沿用
};

```

邓俊辉
deng@tsinghua.edu.cn

插入：单旋

- ◆ 同时可有多个失衡节点，最低者g不低于x的祖父
- ◆ g经单旋调整后复衡，子树高度复原；更高祖先也必平衡，全树复衡



插入：实现

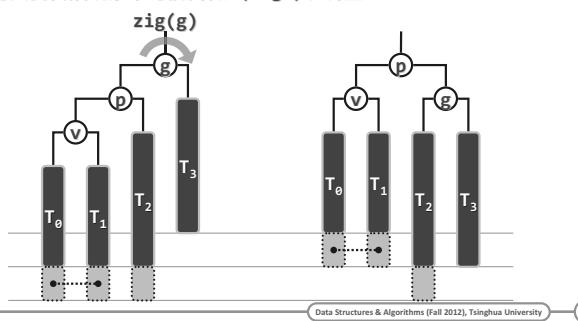
```

template <typename T> BinNodePosi(T) AVL<T>::insert(const T& e) {
    // 查找目标节点(留意其中对 _hot的设置)
    BinNodePosi(T) & x = search(e); if (x) return x;
    // 若目标节点尚不存在，则创建以 _hot为父的新节点x
    x = new BinNode<T>(e, _hot); _size++;
    // 以下从x的父亲出发，逐层向上，依次检查各代祖先
    for (BinNodePosi(T) g = x->parent; g; g = g->parent)
        if (!AvlBalanced(*g)) { //一旦发现g失衡，则通过调整恢复平衡
            FromParentTo(*g) = rotateAt(tallerChild(tallerChild(g)));
            break; //复衡后，局部子树高度必然复原；其祖先亦必如此，故调整结束
        } else //否则(在依然平衡的祖先处)，只需简单地
            updateHeight(g); //更新其高度(平衡性虽不变，高度却可能改变)
    // 至多只需一次调整；若果真做过调整，则全树高度必然复原
    return x; //返回新节点
}

```

删除：单旋

- 同时至多一个失衡节点g，首个可能就是x的父亲_hot
- g经单旋调整后复衡，子树高度不见得复原；更高祖先仍可能失衡
- 针对失衡传播现象，可能需做 $\Theta(\log n)$ 次调整



```

template <typename T> bool AVL<T>::remove(const T& e) {
    // 查找目标节点(留意其中对_hot的设置)
    BinNodePosi(T) & x = search(e); if (!x) return false;
    // 若目标节点的确存在，则先按BST规则删除之
    removeAt(x, _hot); _size--;
    // 以下从_hot出发，逐层向上，依次检查各代祖先g
    for (BinNodePosi(T) g = _hot; g; g = g->parent) {
        if (!AvlBalanced(*g)) //一旦发现g失衡，则通过调整恢复平衡
            g = FromParentTo(*g) = rotateAt(tallerChild(tallerChild(g)));
        updateHeight(g); //并更新其高度
    }
    // 可能需做过O(logn)次调整；无论是否做过调整，全树高度均可能下降
    return true; //删除成功
}

```

Data Structures & Algorithms (Fall 2012), Tsinghua University 9

3+4重构：实现

```

template <typename T> BinNodePosi(T) BST<T>::connect34(
    BinNodePosi(T) a, BinNodePosi(T) b, BinNodePosi(T) c,
    BinNodePosi(T) T0, BinNodePosi(T) T1,
    BinNodePosi(T) T2, BinNodePosi(T) T3)
{
    a->lChild = T0; if (T0) T0->parent = a;
    a->rChild = T1; if (T1) T1->parent = a; updateHeight(a);
    c->lChild = T2; if (T2) T2->parent = c;
    c->rChild = T3; if (T3) T3->parent = c; updateHeight(c);
    b->lChild = a; a->parent = b;
    b->rChild = c; c->parent = b; updateHeight(b);
    return b; //该子树新的根节点
}

```

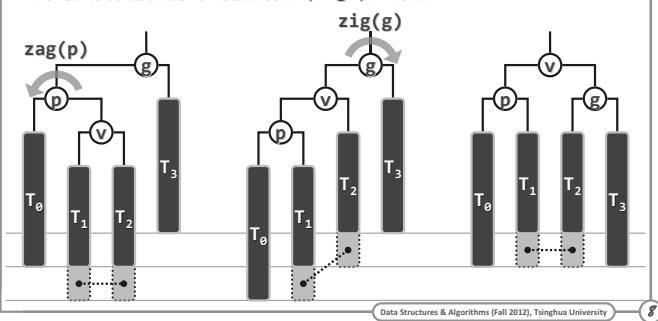
Data Structures & Algorithms (Fall 2012), Tsinghua University 11

综合评价

- 优点** 无论查找、插入或删除，最坏情况下的复杂度均为 $\Theta(\log n)$
- $\Theta(n)$ 的存储空间
- 符合Dictionary/Map的ADT
- 缺点** 借助高度或平衡因子，为此需改造元素结构或额外封装
- 实测复杂度与理论值尚有差距
 - 插入/删除后的旋转，成本不菲
 - 删除操作后的平衡化，最坏情况下需旋转 $\Omega(\log n)$ 次
 - 若需频繁进行插入/删除操作，未免得不偿失
 - 单次动态调整后，全树拓扑结构的变化量可能高达 $\Omega(\log n)$
- 有没有更好的结构呢？ //保持兴趣

删除：双旋

- 同时至多一个失衡节点g，首个可能就是x的父亲_hot
- g经双旋调整后复衡，子树高度不见得复原；更高祖先仍可能失衡
- 针对失衡传播现象，可能需做 $\Theta(\log n)$ 次调整



3+4重构：算法

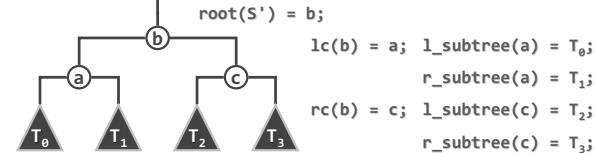
设g(x)为最低的失衡节点，考察祖孙三代：g ~ p ~ v

按中序遍历次序，将其重命名为：a < b < c

它们总共拥有互不相交的四棵（可能为空的）子树

按中序遍历次序，将其重命名为： $T_0 < T_1 < T_2 < T_3$

将原先以为根的子树S，替换为一棵新子树'S'



等价变换，保持中序遍历次序： $T_0 < a < T_1 < b < T_2 < c < T_3$

统一调整：实现

```

template <typename T> BinNodePosi(T) BST<T>::rotateAt(BinNodePosi(T) v){
    BinNodePosi(T) p = v->parent, g = p->parent; //父亲、祖父
    if (IsLChild(*p)) //zig
        if (IsLChild(*v)) { //zig-zig
            p->parent = g->parent; //向上联接
            return connect34
                (v, p, g,
                 v->lChild, v->rChild, p->rChild, g->rChild);
        } else { //zig-zag
            v->parent = g->parent; //向上联接
            return connect34
                (p, v, g,
                 p->lChild, v->rChild, v->lChild, g->rChild);
        }
    else { /*.. zag-zig & zag-zag ..*/ }
}

```

Data Structures & Algorithms (Fall 2012), Tsinghua University 12

8. 高级搜索树

(a) 伸展树

我要一步一步往上爬
在最高点乘着叶片往前飞
——《蜗牛》·周杰伦

郑俊辉

deng@tsinghua.edu.cn

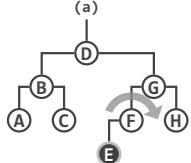
局部性

- Locality: 刚被访问过的数据，极有可能很快地再次被访问
这一现象在信息处理过程中屡见不鲜 //BST就是这样的一个例子
- BST的局部性：
刚被访问过的节点，极有可能很快地再次被访问
- 考虑m次连续的查找 ($m \gg n = |BST|$)
若采用AVL，总共需要 $\mathcal{O}(m \log n)$ 时间
- 利用局部性，能否更快？ //仿效自适应链表
- 策略：节点一旦被访问，随即调整至树根 //如此，下次访问即可...
- 问题：如何实现这种调整？
调整过程自身的复杂度如何控制？

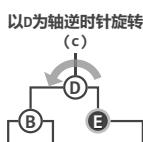
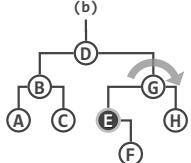
Data Structures & Algorithms (Fall 2012), Tsinghua University

逐层伸展：实例

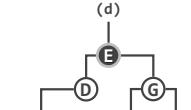
访问E之后，以其父亲F为轴顺时针旋转



以G为轴顺时针旋转



以D为轴逆时针旋转



Data Structures & Algorithms (Fall 2012), Tsinghua University

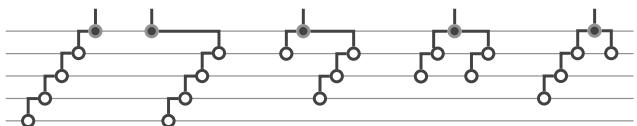
低效率的根源

最坏情况，问题出在哪里？

1)全树拓扑结构等价于一维列表

2)树高呈周期性的算术级数演变，最大 $n-1$ ，最小 $(n-1)/2 = \Omega(n)$

于是，若持续地运气不佳...

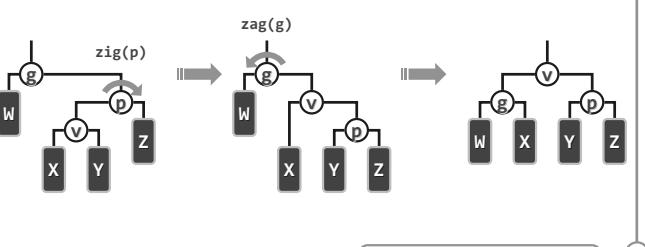


问题的症结既已确定，便可针对性地改进...

Data Structures & Algorithms (Fall 2012), Tsinghua University

zig-zag / zag-zig

- 与AVL树双旋完全等效！
- 与逐层伸展别无二致！
- 难道，就这样平淡无奇？



Data Structures & Algorithms (Fall 2012), Tsinghua University

逐层伸展

节点v一旦被访问，随即调整至树根

自下而上，逐层单旋

zig(parent(v))

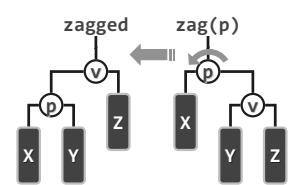
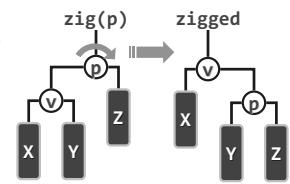
zag(parent(v))

直到v被“推送”至根

效率取决于

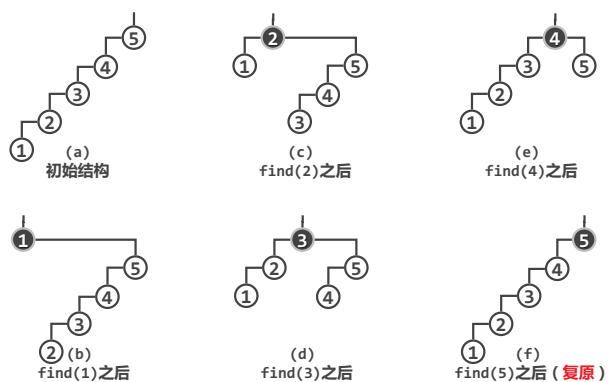
树的初始形态和

节点的访问次序



Data Structures & Algorithms (Fall 2012), Tsinghua University

最坏情况

旋转次数呈周期性的算术级数演变，每一周期累计 $\Omega(n^2)$ ，分摊 $\Omega(n)$ ！

Data Structures & Algorithms (Fall 2012), Tsinghua University

双层伸展

D. D. Sleator & R. E. Tarjan

Self-Adjusting Binary Trees

J. ACM, 32:652-686, 1985



构思的精髓：向上追溯两层，而非一层

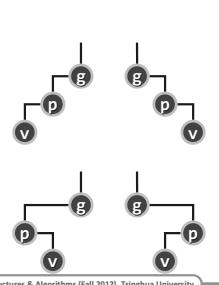
反复考察“祖孙”三代节点

g = parent(p), p = parent(v), v

根据它们的相对位置，经两次旋转使得
v上升两层，成为(子)树根

如此，性能的确会有改善？

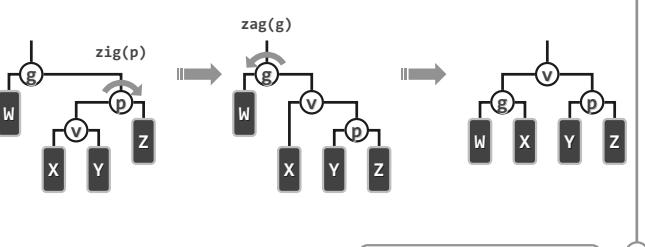
具体地，应该如何旋转？



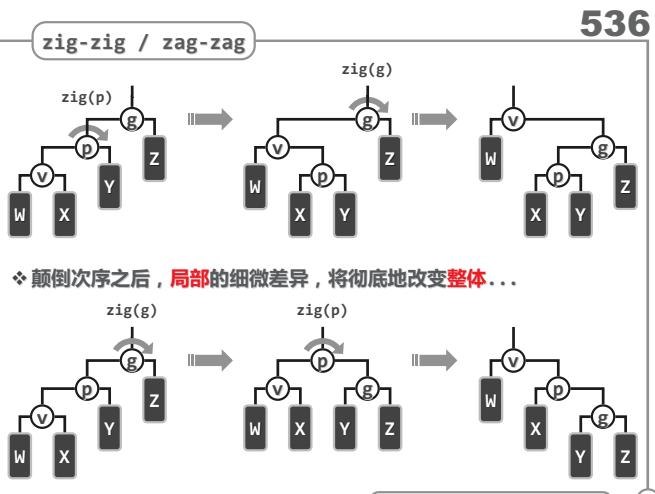
Data Structures & Algorithms (Fall 2012), Tsinghua University

zig-zig / zag-zag

- 与AVL树双旋完全等效！
- 与逐层伸展别无二致！
- 难道，就这样平淡无奇？



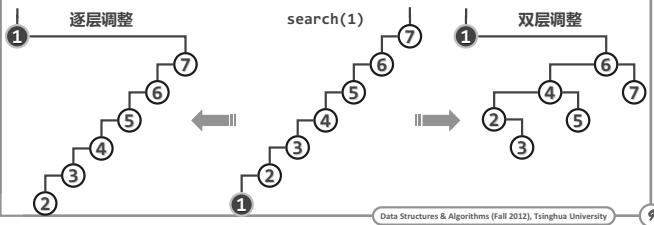
Data Structures & Algorithms (Fall 2012), Tsinghua University



Data Structures & Algorithms (Fall 2012), Tsinghua University

zig-zig / zig-zag

- ◆ 折叠效果：一旦访问坏节点，对应分支的长度将随即减半
- ◆ 最坏情况不致持续发生！甚至只能偶尔发生！
- ◆ 单趟伸展操作，分摊 $O(\log n)$ 时间！ //严格证明详见教材
- ◆ 局部性：若查找范围集中于某 k 个节点 ($k \ll n \ll m$)，则任何连续的 m 次查找，都可在 $O(m \log k + n \log n)$ 时间内完成



Data Structures & Algorithms (Fall 2012), Tsinghua University

539 实现：伸展树接口

```

◆ template <typename NodePosi> //在节点*p与*lc或*rc之间建立父子关系
inline void attachAsLChild(NodePosi p, NodePosi lc)
{ p->lChild = lc; if (lc) lc->parent = p; }
inline void attachAsRChild(NodePosi p, NodePosi rc)
{ p->rChild = rc; if (rc) rc->parent = p; }

◆ template <typename T> class Splay : public BST<T> { //由BST派生
protected:
    BinNodePosi(T) splay(BinNodePosi(T) v); //将v伸展至根
public: //伸展树的查找也会引起整树的结构调整，故search()也需要重写
    BinNodePosi(T) & search(const T& e); //查找（重写）
    BinNodePosi(T) & insert(const T& e); //插入（重写）
    bool remove(const T& e); //删除（重写）
};

```

Data Structures & Algorithms (Fall 2012), Tsinghua University

540 实现：伸展算法

```

◆ template <typename T> BinNodePosi(T) Splay::splay(BinNodePosi(T) v) {
    BinNodePosi(T) p; BinNodePosi(T) g; //父亲、祖父
    while ((p = v->parent) && (g = p->parent)) { //自下而上，反复双层伸展
        BinNodePosi(T) r = g->parent; //每轮之后，*v都将以原曾祖父为父
        if (IsLChild(*v))
            if (IsLChild(*p)) { /* zig-zig */ } else { /* zig-zag */ }
        else if (IsRChild(*p)) { /* zag-zag */ } else { /* zag-zig */ }
        if (!r) v->parent = NULL; //若无曾祖父r，则*v现应为树根
        else //否则，*r此后应以*v作为左或右孩子
            (g == r->lChild) ? attachAsLChild(r, v) : attachAsRChild(r, v);
        updateHeight(g); updateHeight(p); updateHeight(v);
    } //双层伸展结束时，必有g == NULL，但p可能非空
    if (p = v->parent) { /* 单旋（至多一次） */ }
    v->parent = NULL; return v; //伸展完成，v抵达树根
}

```

Data Structures & Algorithms (Fall 2012), Tsinghua University

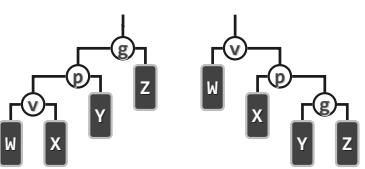
540 实现：伸展算法

541 实现：伸展算法

```

◆ if (IsLChild(*v))
    if (IsLChild(*p)) { /* zIg-zIg */
        attachAsLChild(g, p->rChild); attachAsLChild(p, v->rChild);
        attachAsRChild(p, g); attachAsRChild(v, p);
    } else { /* zIg-zAg */
        attachAsLChild(p, v->rChild); attachAsRChild(g, v->lChild);
        attachAsLChild(v, g); attachAsRChild(v, p);
    }
else
    if (IsRChild(*p)) {
        /* zAg-zAg */
    } else {
        /* zAg-zIg */
    }

```



Data Structures & Algorithms (Fall 2012), Tsinghua University

542 实现：查找算法

```

◆ template <typename T>
BinNodePosi(T) & Splay::search(const T& e) {
    // 调用标准BST的内部接口定位目标节点
    BinNodePosi(T) p = searchIn(_root, e, _hot = NULL);
    // 无论成功与否，最后被访问的节点都将成为伸展至根
    _root = splay((p ? p : _hot)); //成功、失败
    // 总是返回根节点
    return _root;
}

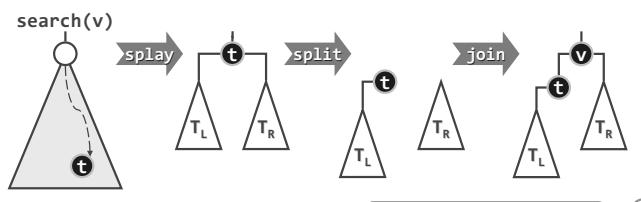
◆ 伸展树的查找操作，与常规BST::search()不同
很可能改变树的拓扑结构，不再属于静态操作

```

Data Structures & Algorithms (Fall 2012), Tsinghua University

543 实现：插入算法

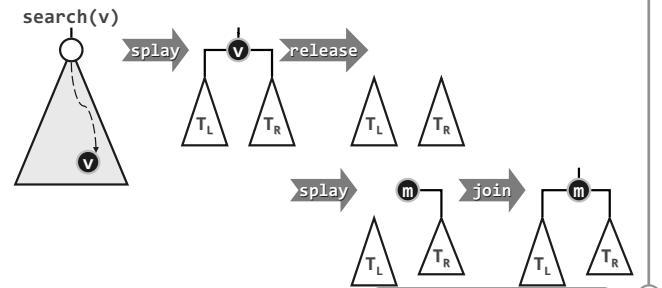
- ◆ 直观方法：调用BST标准的插入算法，再将新节点伸展至根
- 其中，首先需调用 `BST::search()`
- ◆ 重写后的 `Splay::search()` 已集成了 `splay()` 操作
- 查找（失败）之后，`_hot` 即是根节点
- ◆ 既如此，何不随即就在树根附近完成新节点的接入...



Data Structures & Algorithms (Fall 2012), Tsinghua University

544 实现：删除算法

- ◆ 直观方法：调用BST标准的删除算法，再将 `_hot` 伸展至根
- ◆ 同样地，`Splay::search()` 查找（成功）之后，目标节点即是树根
- ◆ 既如此，何不随即就在树根附近完成目标节点的摘除...



Data Structures & Algorithms (Fall 2012), Tsinghua University

综合评价

优点：

分摊复杂度 $\mathcal{O}(\log n)$ — 与AVL树相当 //局部性强时，甚至更低
无需记录节点高度或平衡因子 //优于AVL树
编程实现简单易行 //优于AVL树

缺点：

仍不能保证单次最坏情况的出现 //不适用于对效率敏感的场合
复杂度的分析稍嫌复杂 //不如AVL树 — 好在有初等的方法

Data Structures & Algorithms (Fall 2012), Tsinghua University

17

8. 高级搜索树

(b) B-树

邓俊辉

640K ought to be enough for anybody.

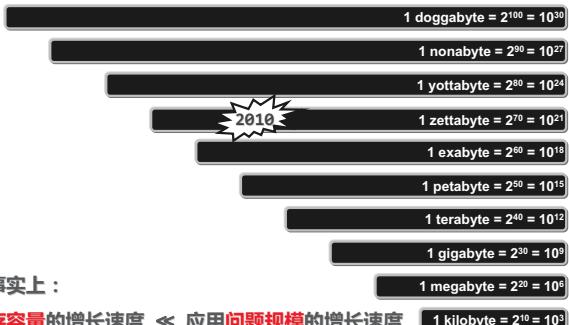
- B. Gates, 1981

deng@tsinghua.edu.cn

越来越小的内存

◆ Turing：存储器？不就是无限长的纸带吗？

RAM：存储器？不就是无限可数个寄存器吗？



Data Structures & Algorithms (Fall 2012), Tsinghua University

18

但事实上：

内存容量的增长速度 << 应用问题规模的增长速度

1 megabyte = 2^20 = 10^6

高速缓存

◆ 事实1：不同容量的存储器，访问速度差异悬殊

◆ 以磁盘与内存为例： $ms/ns > 100,000 > 24 \times 60 \times 60$

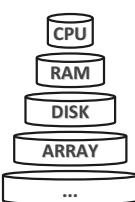
◆ 将一次内存访问比作从书桌上拿起钢笔，则一次外存访问就好比...

◆ 因此，为避免1次外存访问

我们宁愿访问内存10次、100次，甚至...

◆ 多数存储系统，都是分级组织的 — Caching

最常用的数据尽可能放在更高层、更小的存储器中
实在找不到，才向更低层、更大的存储器索取

◆ 算法的I/O复杂度 \propto 数据在不同存储级别之间的传输次数

算法的实际运行时间，往往主要取决于此

Data Structures & Algorithms (Fall 2012), Tsinghua University

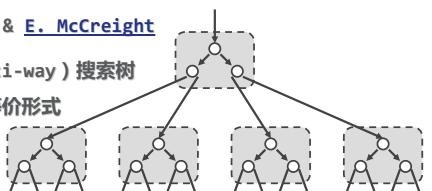
19

B-Tree

◆ 1970, R. Bayer & E. McCreight

◆ 平衡的多路 (multi-way) 搜索树

可理解为BBST的等价形式



◆ 4路：每个节点与其左、右孩子合并

8路：与其下2层后代合并

...
与其下d层后代合并后，超级节点含有 $m = 2^{d+1}$ 路分支， $m-1$ 个关键码
既然逻辑上与BBST完全等价，为何还要引入B-树？

Data Structures & Algorithms (Fall 2012), Tsinghua University

20

B-Tree

◆ 多级存储系统中使用B-树，可针对外部查找，大大减少磁盘操作次数

◆ 难道，AVL还不够？比如，若有 $n = 16$ 个记录每次查找需要 $\log_2 10^9 = 30$ 次I/O操作

每次只读出一个关键码，得不偿失

◆ B-树又能如何？

充分利用外存对批量访问的高效支持，将此特点转化为优点

每下降一层，都以超级节点为单位，读入一组关键码

◆ 具体多大一组？视磁盘的数据块大小而定， $m = \#keys/pg$ 比如，目前多数数据库系统采用 $m = 200 \sim 300$ ◆ 回到上例，若取 $m = 256$ ，则每次查找只需 $\log_{256} 10^9 \leq 4$ 次I/O

Data Structures & Algorithms (Fall 2012), Tsinghua University

21

B-Tree

553

- ◆ 所谓m阶B-树即m路平衡搜索树 ($m \geq 2$)
- ◆ 所有外部节点的深度统一相等
- 所有叶节点的深度统一相等
- ◆ 树高 h 需计入外部节点
 $h = \text{外部节点的深度}$
- ◆ 内部节点各有不超过 $m-1$ 个关键码，以及不超过 m 个分支，具体地
若存有 n 个关键码 $\{K_1 < K_2 < \dots < K_n\}$
则同时还应有 $n+1$ 个引用 $\{A_0, A_1, A_2, \dots, A_n\}, n+1 \leq m$
- ◆ 内部节点的分支数 ($n+1$) 也不能太少，具体地
树根以外的节点应满足 $n+1 \geq \lceil m/2 \rceil$ ，故亦称作 $(\lceil m/2 \rceil, m)$ -树
非空B-树的根节点应满足 $n+1 \geq 2$

Data Structures & Algorithms (Fall 2012), Tsinghua University

实例

555

◆ http://thudsa.3322.org/~deng/ds/demo/b_tree/

◆ $m = 3$

2-3-树，(2,3)-树，最简单的B-树 //John Hopcroft, 1970

各（内部）节点的子树数目有两种可能：2或3

各节点含key的数目有两种可能：1或2

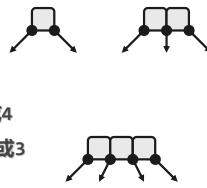
◆ $m = 4$

2-3-4-树，(2,4)-树

各节点之子树数目有三种可能：2、3或4

各节点含key的数目有三种可能：1、2或3

留意把玩，对稍后理解红黑树大有裨益



Data Structures & Algorithms (Fall 2012), Tsinghua University

BTree

557

```
#define BTNodePosi(T) BTNode<T>* //B-树节点位置
template <typename T> class BTree { //B-树
protected:
    int _order, _size; //阶次、关键码总数
    BTNodePosi(T) _root; //根
    BTNodePosi(T) _hot; //search()最后访问的非空节点
    void solveOverflow(BTNodePosi(T)); //因插入而上溢后的分裂处理
    void solveUnderflow(BTNodePosi(T)); //因删除而下溢后的合并处理
public:
    BTNodePosi(T) search(const T& e); //查找
    BTNodePosi(T) insert(const T& e); //插入
    bool remove(const T& e); //删除
};
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

查找：实现

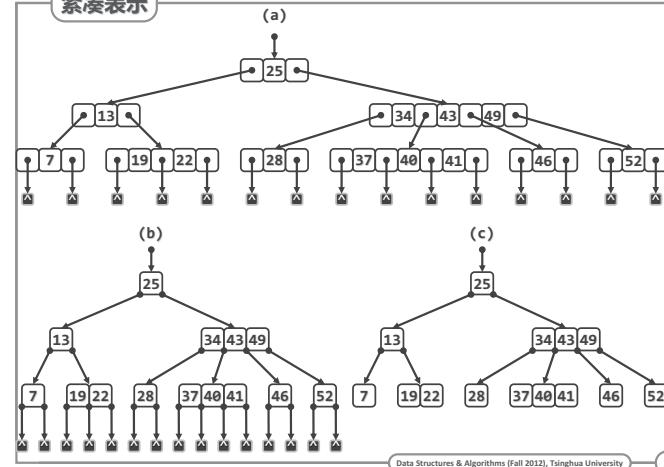
559

```
template <typename T>
BTNodePosi(T) BTree<T>::search(const T& e) {
    BTNodePosi(T) v = _hot = _root; //从根节点出发
    while (true) { //逐层查找
        if (!v) return v; //失败：最终抵达外部节点
        //在当前节点对应的向量中二分查找目标关键码
        Rank r = v->data.search(e); //对于通常的阶次m，可直接顺序查找
        //成功：停止于合法的秩，且关键码相等
        if ((0 <= r) && (e == v->data[r])) return v;
        //否则，沿引用转至对应的下层子树，并将其根节点读入内存
        _hot = v; v = v->child[r+1]; //I/O，最为耗时
    }
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

紧凑表示

554



Data Structures & Algorithms (Fall 2012), Tsinghua University

BTNode

556

◆ template <typename T> class BTNode { //B-树节点

```
public:
    BTNodePosi(T) parent; //父节点
    Vector<T> data; //数值向量
    Vector<BTNodePosi(T)> child; //孩子向量（其长度总比data多一）
    BTNode() { parent = NULL; child.insert(0, NULL); }
    BTNode(T e, BTNodePosi(T) lc = NULL, BTNodePosi(T) rc = NULL) {
        parent = NULL; //作为根节点，而且初始时
        data.insert(0, e); //只有一个关键码，以及
        child.insert(0, lc); child.insert(1, rc); //两个孩子
        if (lc) lc->parent = this; if (rc) rc->parent = this;
    }
};
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

查找：算法

558

◆ 二分查找 + 顺序查找

将根节点作为当前节点 //常驻RAM

只要当前节点非外部节点 {

在当前节点中顺序查找 //RAM内部

若找到目标关键码，则

返回“查找成功”

否则 //停止于指向下层的某一引用

否则，沿引用转至对应的下层子树

将其根节点读入内存 //I/O，最为耗时

更新当前节点

}

返回“查找失败”



Data Structures & Algorithms (Fall 2012), Tsinghua University

查找：实例

560

◆ (3,5)-树

53 97 36 89 41 75 19 84 77 79 51 57 99 91 92 93 17 73 13 66 59 49 63 65 71 69 27 31 38

成功查找 : 75, 19, 49

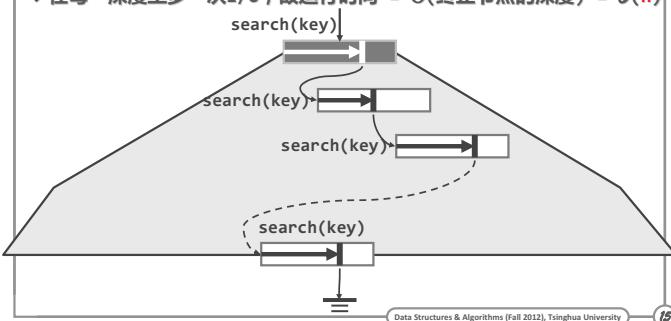
失败查找 : 5, 45



Data Structures & Algorithms (Fall 2012), Tsinghua University

查找：复杂度

- 约定：根节点常驻RAM
- 忽略内存中的查找，运行时间主要取决于I/O次数
- 在每一深度至多一次I/O，故运行时间 = $\Theta(\text{终止节点的深度}) = \Theta(h)$



最小树高

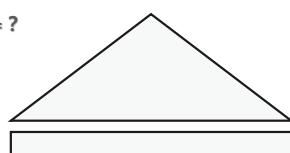
- 含N个关键码的m阶B-树，**最小高度**=？

为此，内部节点应尽可能“胖”

各层节点数依次为

$$n_0 = 1, n_1 = m, n_2$$

$$n_3 = m^3, \dots, n_{h-1} = m^{h-1}, n_h = m^h$$



考查**外部节点所在层**：

$$N+1 = n_h \leq m^h$$

$$h \geq \log_m(N+1) = \Omega(\log_m N)$$

相对于BBST：

$$(\log_m N - 1)/\log_2 N = \log_m 2 - \log_2 N \approx 1/\log_2 m$$

若取m = 256，树高(I/O次数)约降低至1/8

Data Structures & Algorithms (Fall 2012), Tsinghua University 17

意义与价值

- BBST，究竟可能多高？

考查高度为h的BBST（如AVL）可包含节点的数目f

$$h=33 \text{ 时, } f = 2^{33} = 8.6 \times 10^9$$

//全球人口

$$h=77 \text{ 时, } f = 2^{77} = 1.5 \times 10^{23}$$

//全球人口的体细胞总数

$$h=133 \text{ 时, } f = 2^{133} = 1.1 \times 10^{40}$$

//国际象棋可能的局面总数

$$h=260 \text{ 时, } f = 2^{260} = 1.8 \times 10^{78}$$

//目前可观测宇宙中基本粒子总数

由此可见，B-树的价值，的确更多地体现在实用方面

通过选取适当的节点规模(m)，弥合存储层级之间**巨大的速度差异**

另外，在算法方面，B-树也有其独特的价值与地位...

Data Structures & Algorithms (Fall 2012), Tsinghua University 18

插入：算法

```
template <typename T>
BTNodePosi(T) BTTree<T>::insert(const T & e) {
    BTNodePosi(T) v = search(e); //查找目标节点
    if (!v) {
        Rank r = _hot->data.search(e); //在节点_hot中确定插入位置
        _hot->data.insert(r + 1, e); //将新关键码插至对应的位置
        _hot->child.insert(r + 1, NULL); //创建一个空子树指针
        _size++; //更新全树规模
        solveOverflow(_hot); //如有必要，需做分裂
    }
    return v; //无论e是否存在于原树中，至此总有v->data == e
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University 19

上溢修复：算法

设上溢节点中的关键码依次为{k₀, ..., k_{m-1}}

令s = ⌊m/2⌋，以关键码k_s为界划分为

$$\{k_0, \dots, k_{s-1}\}, \{k_s\}, \{k_{s+1}, \dots, k_{m-1}\}$$

关键码k_s上升一层，并以分裂(split)出的两个节点作为左、右孩子



确认：如此分裂后，左、右孩子所含**关键码数**依然符合m阶B-树的条件

$$s = \lfloor m/2 \rfloor \geq \lceil m/2 \rceil - 1$$

$$m - s - 1 = m - \lfloor m/2 \rfloor - 1 = \lceil m/2 \rceil - 1$$

Data Structures & Algorithms (Fall 2012), Tsinghua University 20

上溢修复：算法

若上溢节点的父亲**本已饱和**，接纳被提升的关键码之后，也将上溢。此时，大可套用前法，继续分裂。



上溢可能持续发生，并逐层向上传播；纵然最坏情况，不过到根

若果真抵达树根，可令被提升关键码自成节点，作为新的树根

这也是B-树增高的唯一可能
(具体的概率多大?)



总体执行时间线性正比于分裂次数，不超过O(h)

Data Structures & Algorithms (Fall 2012), Tsinghua University 21

上溢修复：实现

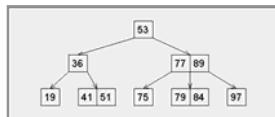
```
template <typename T> void BTTree<T>::solveOverflow(BTNodePosi(T) v) {
    if (_order >= v->child.size()) return; //递归基：不再上溢
    Rank s = _order/2; //轴点(此时_order = data.size() = child.size()-1)
    BTNodePosi(T) u = new BTNode<T>(); //注意：新节点已有一个空孩子
    for (Rank j = 0; j < _order-s-1; j++) { /*复制v右半侧，分裂出节点u*/ }
    u->child[_order-s-1] = v->child.remove(s+1); //移动v最靠右的孩子
    if (u->child[0]) //若u的孩子们非空，则令其父节点统一指向u
        for (Rank j = 0; j < _order-s; j++) u->child[j]->parent = u;
    BTNodePosi(T) p = v->parent; //v当前的父节点
    if (!p) //若p为空，则创建之
        _root = p = new BTNode<T>(); p->child[0] = v; v->parent = p;
    Rank r = 1 + p->data.search(v->data[0]); //p中指向u的指针的秩
    p->data.insert(r, v->data.remove(s)); //轴点关键码上升
    p->child.insert(r+1, u); u->parent = p; //新节点u与父节点p互联
    solveOverflow(p); //上升一层，如有必要则继续分裂 — 至多递归O(logn)层
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University 22

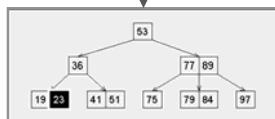
插入：实例

❖ 2-3-树：

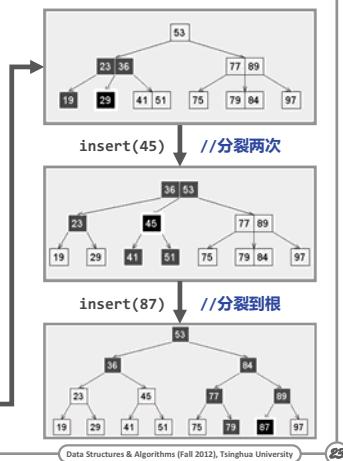
53 97 36 89 41 75 19 84 77 79 51



insert(23) //无需分裂



insert(29) //分裂一次



Data Structures & Algorithms (Fall 2012), Tsinghua University

28

删除：算法

```

❖ template <typename T> bool BTTree<T>::remove(const T& e) {
    BTNodePosi(T) v = search(e); if (!v) return false; //查找
    Rank r = v->data.search(e); //确定目标关键码在节点v中的秩
    if (v->child[0]) { //若v非叶子，则e的后继必属于某叶节点
        BTNodePosi(T) u = v->child[r+1]; //在右子树中一直向左，即可
        while (u->child[0]) u = u->child[0]; //找出e的后继
        v->data[r] = u->data[0]; v = u; r = 0; //并与之交换位置
    } //至此，v必然位于最底层，且其中第r个关键码就是待删除者
// 删除e以及对应的子树（实际上此时其左、右子树皆空，可任删其一）
    v->data.remove(r); v->child.remove(r); _size--;
    solveUnderflow(v); //如有必要，需做旋转或合并
    return true;
}

```

Data Structures & Algorithms (Fall 2012), Tsinghua University

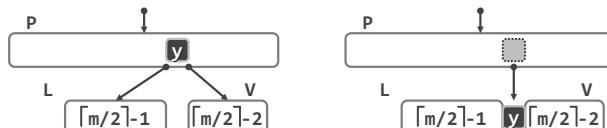
29

下溢修复：算法

3) L和R或者不存在，或者所含的关键码均不足 $\lceil m/2 \rceil$ 个注意，L和R仍必有其一，且恰含 $\lceil m/2 \rceil - 1$ 个关键码（不妨以L为例）

从P中抽出介于L和V之间的关键码

通过该关键码将L和V“粘接”成一个节点



如此合并之后，原高度的下溢得以修复，但可能导致更高处的P下溢。此时，大可套用前法，继续旋转或合并。

下溢可能持续发生，并逐层向上传播；但至多不过 $O(h)$ 次

Data Structures & Algorithms (Fall 2012), Tsinghua University

27

下溢修复：实现：情况1、2

❖ 情况1：向左兄弟借关键码 — 情况2完全对称

```

❖ if (0 < r) { //若v不是p的第一个孩子，则
    BTNodePosi(T) ls = p->child[r-1]; //左兄弟必存在
    if ((_order + 1) / 2 < ls->child.size()) { //若该兄弟足够“胖”，则
        // p借出一个关键码给v（作为最小关键码）
        v->data.insert(0, p->data[r-1]);
        // ls的最大关键码转入p
        p->data[r-1] = ls->data.remove(ls->data.size() - 1);
        // 同时ls的最小孩子过继给v（作为v的最左侧孩子）
        v->child.insert(0, ls->child.remove(ls->child.size() - 1));
        if (v->child[0]) v->child[0]->parent = v;
    // 至此，通过右旋已完成当前层（以及所有层）的下溢处理
        return;
    }
}

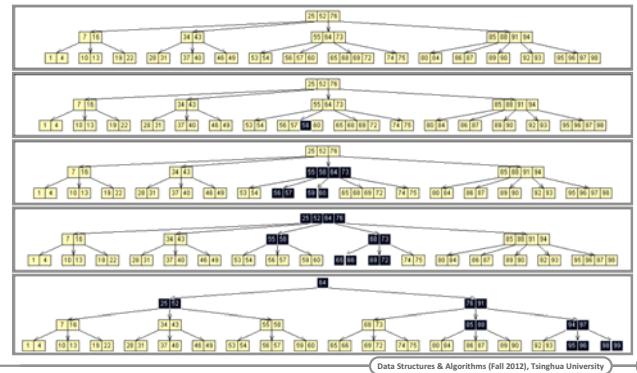
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

29

插入：实例

❖ 3-5-树：
14 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 56 60 64 68 72 76 80 84 53 55 58 66 87 88 89 90 91 92 93 94 95 96 97 98 73 74 75 76 69
无需分裂insert(58) 分裂一次insert(59) 分裂两次insert(66) 分裂到根insert(99)



Data Structures & Algorithms (Fall 2012), Tsinghua University

29

下溢修复：算法

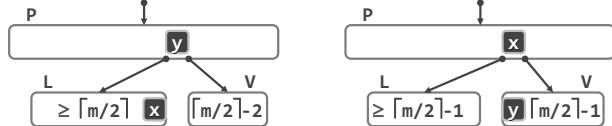
❖ 节点v下溢时，必包含 $\lceil m/2 \rceil - 2$ 个关键码、 $\lceil m/2 \rceil - 1$ 个分支

❖ 视其左、右兄弟L、R所含关键码的数目，可分三种情况处理

1) 若L存在，且至少包含 $\lceil m/2 \rceil$ 个关键码

将关键码y从P移至V中（作为最小关键码）

将关键码x从L移至P中（取代其中原关键码y）



❖ 如此旋转之后，局部乃至整树都重新满足B-树条件，下溢修复完毕

2) 若R存在，且至少包含 $\lceil m/2 \rceil$ 个关键码 — 完全对称

Data Structures & Algorithms (Fall 2012), Tsinghua University

29

下溢修复：实现

```

❖ template <typename T> void BTTree<T>::solveUnderflow(BTNodePosi(T) v) {
    if ((_order + 1) / 2 <= v->child.size()) return; //递归基
    BTNodePosi(T) p = v->parent;
    if (!p) { /* 递归基：已到根节点，没有孩子的下限 */ }
    Rank r = 0; while (p->child[r] != v) r++; //确定v是p的第r个孩子
    // 情况1：向左兄弟借关键码
    if (0 < r) { /* 若v的左兄弟存在，且... */ }
    // 情况2：向右兄弟借关键码
    if (p->child.size() - 1 > r) { /* 若v的右兄弟存在，且... */ }
    // 情况3：左、右兄弟要么为空（但不可能同时），要么都太“瘦”
    if (0 < r) { /* 与左兄弟合并 */ } else { /* 与右兄弟合并 */ }
    // 上升一层，继续分裂 — 至多递归O(logn)层
    solveUnderflow(p);
    return;
}

```

Data Structures & Algorithms (Fall 2012), Tsinghua University

29

下溢修复：实现：情况3

```

❖ if (0 < r) { //与左兄弟合并
    BTNodePosi(T) ls = p->child[r-1]; //左兄弟必存在
    ls->data.insert(ls->data.size(), p->data.remove(r - 1));
    p->child.remove(r); //p的第r-1个关键码转入ls，v不再是p的第r个孩子
    ls->child.insert(ls->child.size(), v->child.remove(0));
    if (ls->child[ls->child.size() - 1]) //v的最左侧孩子过继给ls做最右侧孩子
        ls->child[ls->child.size() - 1]->parent = ls;
    while (!v->data.empty()) { //v剩余的关键码和孩子，依次转入ls
        ls->data.insert(ls->data.size(), v->data.remove(0));
        ls->child.insert(ls->child.size(), v->child.remove(0));
        if (ls->child[ls->child.size() - 1])
            ls->child[ls->child.size() - 1]->parent = ls;
    }
    release(v); //释放v
} else { /* 与右兄弟合并，完全对称 */ }

```

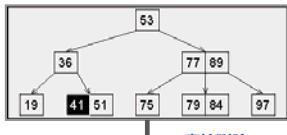
Data Structures & Algorithms (Fall 2012), Tsinghua University

30

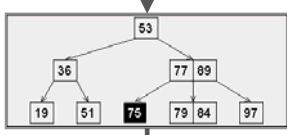
删除：实例：底层节点

❖ 2-3-树：

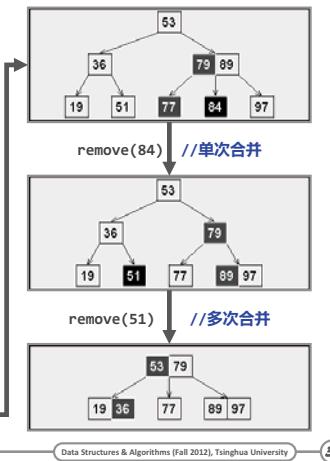
53 97 36 89 41 75 19 84 77 79 51



remove(41) //直接删除



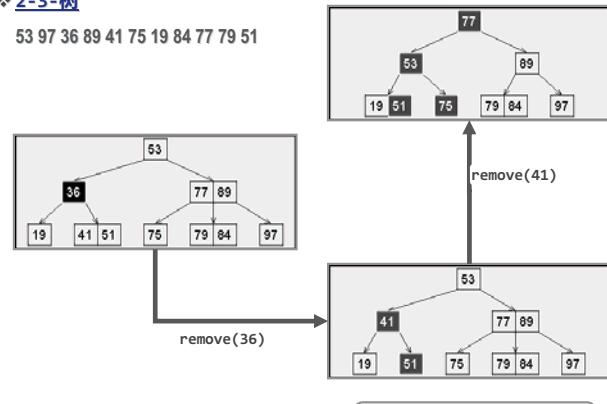
remove(75) //旋转



删除：实例：非底层节点

❖ 2-3-树

53 97 36 89 41 75 19 84 77 79 51



课后

❖ 旋转，还是分裂？

观察：通过旋转可以减少分裂

如此，可以减少页面的申请量，提高空间利用率

然而，这一策略却没有被普遍采用—原因何在？

❖ 独自分裂，还是联合分裂？

节点上溢后未必独自分裂，也可由n个邻接兄弟“均摊”新关键码

得到n+1个相邻节点，各含有至少 $\lfloor n \times (m-1)/(n+1) \rfloor$ 个关键码

如此，可将B-树的空间使用率从50%提高至n/(n+1)

google("B"-tree")

8. 高级搜索树

(x1) 红黑树

邓俊辉

deng@tsinghua.edu.cn

莫赤匪孤，莫黑匪乌

惠而好我，携手同车

Data Structures & Algorithms (Fall 2012), Tsinghua University

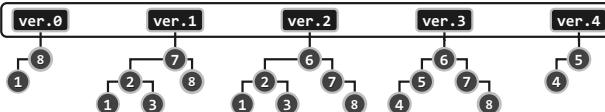
580

一致性结构

❖ Persistent structure：支持对历史版本的访问 //ephemeral

T.search(ver, key); T.insert(ver, key); T.remove(ver, key)

❖ 蛮力实现：每个版本独立保存；各版本入口自成一个搜索结构



❖ 单次操作的时间 = O(logh + logn) //h = |history|

然而，所有版本的构造累计需要O(h*n)时间，且共需O(h*n)空间

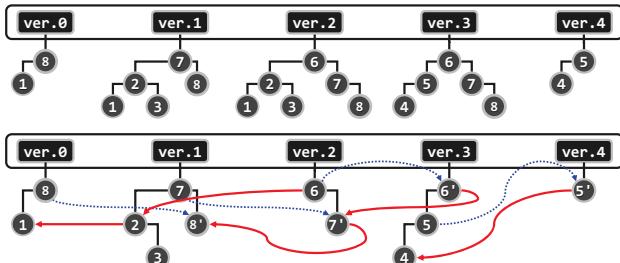
❖ 挑战：以上两项指标，可否控制在O(n + h*logn)？

❖ 可以！为此需利用相邻版本之间的关联性...

Data Structures & Algorithms (Fall 2012), Tsinghua University

o(1)重构

❖ 可大量共享，仅少量更新 — 每个版本新增复杂度仅为O(logn)



❖ 能否进一步提高，比如总体O(n + h)、单版本O(1)？可以！

为此，就树形结构的拓扑而言，相邻版本之间的差异不能超过O(1)

❖ 很遗憾，AVL、Splay等BBST均不具备这一性质；必须另辟蹊径...

Data Structures & Algorithms (Fall 2012), Tsinghua University

红 vs. 黑

❖ 1972, R. Bayer, "symmetric binary B-tree"

1978, L. Guibas & R. Sedgewick, "red-black tree"

1982, H. Olivie, "half-balanced binary search tree"

❖ 由红、黑两类节点组成的BST //也有给边染色的
(统一增设外部节点NULL，使之成为真二叉树)

(1) 树根：必为黑色

(2) 外部节点：均为黑色

(3) 内节点：若为红，则只能有黑孩子 //红之子、父必黑

(4) 外部节点到根：途中黑节点数目相等 //黑深度（高度）统一

❖ 以上定义令人费解，有直观解释吗？

如此定义的BST，也是BBST？

Data Structures & Algorithms (Fall 2012), Tsinghua University

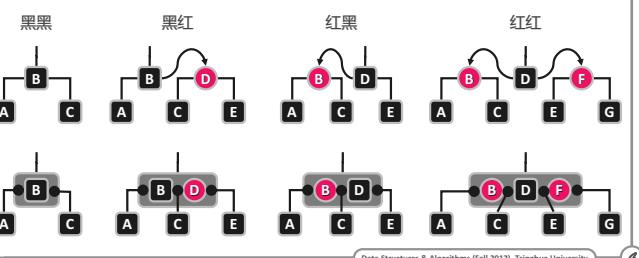
(2,4)树 vs. 红黑树

❖ 提升各红节点，使之与其（黑）父亲等高，于是

每棵红黑树，都对应于一棵(2,4)-树：.../ds/demo/bst/

❖ 将黑节点与其红孩子视作（关键码并合并为）超级节点...

❖ 无非四种组合，分别对应于4阶B-树的一类内部节点 //反过来呢？



Data Structures & Algorithms (Fall 2012), Tsinghua University

红黑树 ∈ BBST

❖ 定理：包含n个内部节点的红黑树T，高度 $h = \Theta(\log n)$

//更严格地有： $\log_2(n+1) \leq h \leq 2\log_2(n+1)$

❖ 左侧的“≤”显然

右侧的“≤”呢？

❖ 若T高度为h，黑深度为d

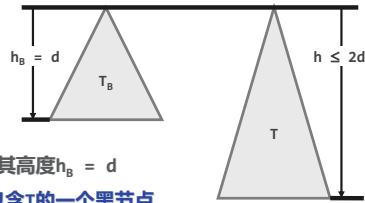
则 $h \leq 2d$

❖ 若T所对应的B-树为 T_B ，则其高度 $h_B = d$

// T_B 的每个节点包含且仅包含T的一个黑节点

❖ 于是， $d = h_B \leq \log_{4/3}((n+1)/2) + 1 = \log_2(n+1)$

❖ 实际上由等价性，既然B-树是平衡的，红黑树自然也应是

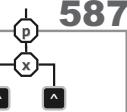


Data Structures & Algorithms (Fall 2012), Tsinghua University

5

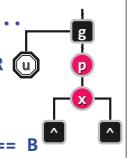
插入：算法

❖ 观察插入关键码key //不妨设T中本不含key



❖ 按BST的常规算法，插入之 // $x = insert(T, key)$ 必为末端节点
不妨设x的父亲p = x->parent存在 //否则，即平凡的首次插入

❖ 将x染红（除非它是根） // $x->color = isRoot(x) ? R : B$
条件(1)、(2)和(4)依然满足 //但(3)不见得，有可能...



❖ 双红 (double-red) // $p->color == x->color == R$

❖ 考查：

x的祖父g = p->parent // $g != null \&& g->color == B$
p的兄弟u = ($p == (g->lchild)$) ? g->rc : g->lc //即x的叔父

❖ 视u的颜色，分两种情况处理...

Data Structures & Algorithms (Fall 2012), Tsinghua University

7

插入：实现

❖ template <typename T>

```
void RedBlack<T>::solveDoubleRed(BinNodePosi(T) x) {
    if (IsBlack(x)) return; //“新”插入节点实为雷同节点的情况
    if (IsRoot(*x)) { //若已(递归)转至树根，则
        _root->color = RB_BLACK; //将其转黑
        _root->height++; return; //整树黑高度也随之递增
    }
    BinNodePosi(T) p = x->parent; //x的父亲(必非空)
    if (IsBlack(p) || IsRoot(*p)) return; //递归基：到黑节点或根
    BinNodePosi(T) g = p->parent; //p既为红，则x祖父必存在且为黑色
    BinNodePosi(T) u = uncle(x); //以下，视x叔父u的颜色分别处置
    if (IsBlack(u)) { /* ... u为黑(或NULL) ... */ }
    else { /* ... u为红 ... */ }
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

9

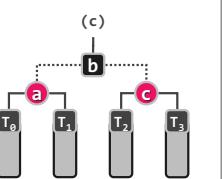
双红修正：算法

1. 参照AVL树算法，做局部“3+4”重构

将节点x、p、g及其四棵子树

按中序组合为

$T_0 < a < T_1 < b < T_2 < c < T_3$



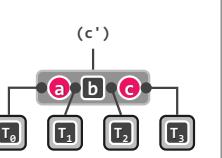
2. 染色：b转黑，a或c转红

❖ 从B-树的角度，如何理解这一情况？

1. 调整前之所以非法，是因为

在某个三叉节点中插入红关键码，使得

原黑关键码不再居中 //RRB或BRR，出现相邻的红关键码



2. 调整之后的效果相当于

//B-树的拓扑结构不变，但

在新的四叉节点中，三个关键码的颜色改为RBR

Data Structures & Algorithms (Fall 2012), Tsinghua University

11

RedBlack

❖ template <typename T> class RedBlack : public BST<T> { //红黑树
protected:

void solveDoubleRed(BinNodePosi(T) x); //双红修正

void solveDoubleBlack(BinNodePosi(T) x); //双黑修正

int updateHeight(BinNodePosi(T) x); //更新节点x的高度

public: //BST::search()等其余接口可直接沿用

BinNodePosi(T) insert(const T& e); //插入(重写)

bool remove(const T& e); //删除(重写)

}

❖ template <typename T>

int RedBlack<T>::updateHeight(BinNodePosi(T) x) {

x->height = max(stature(x->lChild), stature(x->rChild));

if (IsBlack(x)) x->height++; return x->height;

(Data Structures & Algorithms (Fall 2012), Tsinghua University)

6

插入：实现

❖ template <typename T>

BinNodePosi(T) RedBlack<T>::insert(const T& e) {

//确认目标节点不存在(留意对_hot的设置)

BinNodePosi(T) & x = search(e);

if (x) return x;

//创建红节点x，以_hot为父，黑高度-1

x = new BinNode<T>(e, _hot, NULL, NULL, -1);

_size++;

//经双红修正后，即可返回

solveDoubleRed(x);

return x;

} //无论原树中是否存有e，返回时总有x->data == e

(Data Structures & Algorithms (Fall 2012), Tsinghua University)

8

双红修正：实现

❖ template <typename T>

```
void RedBlack<T>::solveDoubleRed(BinNodePosi(T) x) {
    if (IsBlack(x)) return; //“新”插入节点实为雷同节点的情况
    if (IsRoot(*x)) { //若已(递归)转至树根，则
        _root->color = RB_BLACK; //将其转黑
        _root->height++; return; //整树黑高度也随之递增
    }
    BinNodePosi(T) p = x->parent; //x的父亲(必非空)
    if (IsBlack(p) || IsRoot(*p)) return; //递归基：到黑节点或根
    BinNodePosi(T) g = p->parent; //p既为红，则x祖父必存在且为黑色
    BinNodePosi(T) u = uncle(x); //以下，视x叔父u的颜色分别处置
    if (IsBlack(u)) { /* ... u为黑(或NULL) ... */ }
    else { /* ... u为红 ... */ }
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

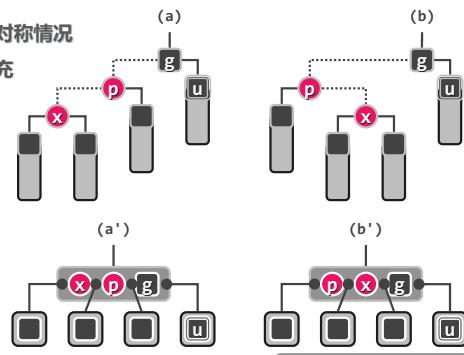
9

双红修正(1) : u->color == B

❖ 此时；x、p和g的四个孩子(可能是外部节点)全为黑，且
黑高度相同

❖ 另两种对称情况

自行补充



(Data Structures & Algorithms (Fall 2012), Tsinghua University)

10

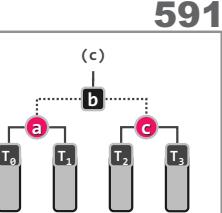
双红修正(1) : u->color == B

1. 参照AVL树算法，做局部“3+4”重构

将节点x、p、g及其四棵子树

按中序组合为

$T_0 < a < T_1 < b < T_2 < c < T_3$



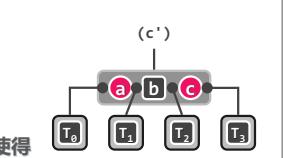
2. 染色：b转黑，a或c转红

❖ 从B-树的角度，如何理解这一情况？

1. 调整前之所以非法，是因为

在某个三叉节点中插入红关键码，使得

原黑关键码不再居中 //RRB或BRR，出现相邻的红关键码



2. 调整之后的效果相当于

//B-树的拓扑结构不变，但

在新的四叉节点中，三个关键码的颜色改为RBR

Data Structures & Algorithms (Fall 2012), Tsinghua University

11

双红修正(1) : 实现

❖ template <typename T>

void RedBlack<T>::solveDoubleRed(BinNodePosi(T) x) {

/* */

if (IsBlack(u)) { //u为黑或NULL

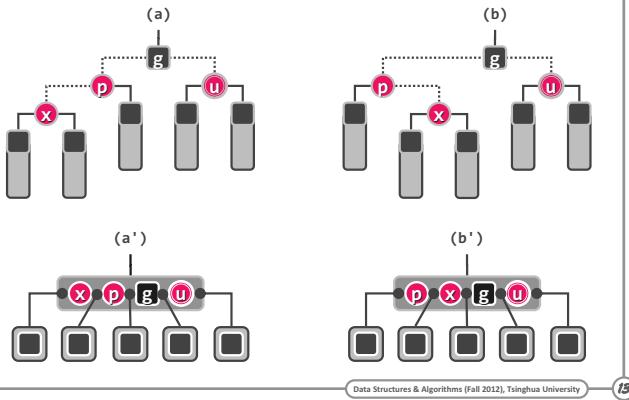
```
//若x与p同侧，则p由红转黑，x保持红；否则，x由红转黑，p保持红
if (IsLChild(*x) == IsLChild(*p)) p->color = RB_BLACK;
else x->color = RB_BLACK;
g->color = RB_RED; //g必定由黑转红
BinNodePosi(T) gg = g->parent; //grand-great parent
BinNodePosi(T) r = FromParentTo(*g) = rotateAt(x); //调整
r->parent = gg; //调整之后的新子树，需与原曾祖父联接
} else { /* ... u为红 ... */ }
```

(Data Structures & Algorithms (Fall 2012), Tsinghua University)

12

双红修正(2) : u->color == R

在B-树中，等效于超级节点发生上溢 //另两种对称情况，自行补充



Data Structures & Algorithms (Fall 2012), Tsinghua University

双红修正(2) : u->color == R

既然是分裂，也应有可能继续向上传递
//亦即，g与parent(g)再次构成双红

果真如此，可

等效地将g视作新插入的节点
区分以上两种情况，如法处置

直到所有条件满足（即不再双红）
或者抵达树根

g若果真到达树根，则
1. 强行将g转为黑色
2. 整树（黑）高度加一

Data Structures & Algorithms (Fall 2012), Tsinghua University

双红修正：复杂度

重构、染色均属常数时间的局部操作 //故只需统计其总次数

红黑树的每一次插入操作，都可在 $\mathcal{O}(\log n)$ 时间内完成

其中

1. 至多做 $\mathcal{O}(\log n)$ 次节点染色
2. 至多做一次“3+4”重构

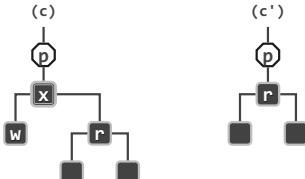
| 情况 | 旋转次数 | 染色次数 | 此后 |
|-----|------|------|------------------|
| u为黑 | 1~2 | 2 | 调整随即完成 |
| u为红 | 0 | 3 | 可能再次双红
但必上升两层 |

Data Structures & Algorithms (Fall 2012), Tsinghua University

删除：算法

若x与r均黑（double-black），则不然...

摘除x并代之以r后，全树黑深度不再统一 //B-树中x所属节点下溢



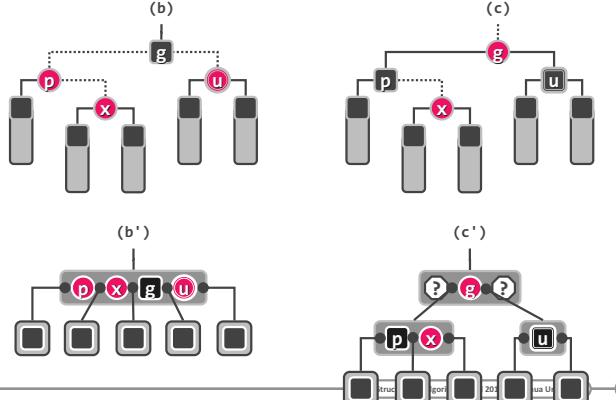
在新树中，记 r的父亲为p = r->parent //亦即原树中x的父亲
r的兄弟为s = (r == (p->lc) ? p->rc : p->lc)

以下分四种情况处理...

Data Structures & Algorithms (Fall 2012), Tsinghua University

双红修正(2) : u->color == R

p与u转黑，g转红 — 在B-树中，等效于节点分裂，关键码g上升一层



Data Structures & Algorithms (Fall 2012), Tsinghua University

双红修正(2) : 实现

```
template <typename T>
void RedBlack<T>::solveDoubleRed(BinNodePosi(T) x) {
    /* ..... */
    if (IsBlack(u)) { /* ... u为黑（或NULL）... */ }
    else { //u为红色
        p->color = RB_BLACK; p->height++; //p由红转黑，高度递增
        u->color = RB_BLACK; u->height++; //u由红转黑，高度递增
        IsRoot(*g) ? g->height++ //g若为根，则高度递增
                    : g->color = RB_RED; //否则g转为红
        solveDoubleRed(g); //（递归地）继续调整g
    }
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

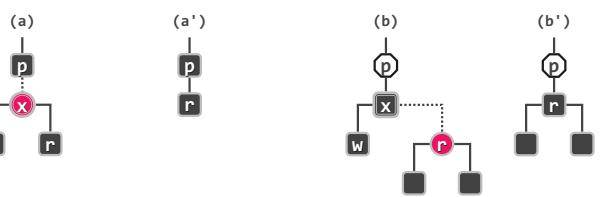
删除：算法

按BST常规算法，删除key //设实际被摘除节点x = delete(T, key)

于是，x至少拥有一个外部节点孩子w //另一孩子记作r

条件(1~2)依然满足；但(3~4)不见得 //在原树中，考查x与r...

若二者之一为红，则(3~4)不难满足 //删除遂告完成！



Data Structures & Algorithms (Fall 2012), Tsinghua University

删除：实现

```
template <typename T>
bool RedBlack<T>::remove(const T& e) {
    BinNodePosi(T) &x = search(e); if (!x) return false; //查找定位
    BinNodePosi(T) r = removeAt(x, _hot); _size--; //实施删除
    if (0 == _size) return true; //若删除后为空树，可直接返回
    // _hot的某孩子被删除，r指向其接替者。以下检查是否失衡，并做必要调整
    if (!_hot->color == RB_BLACK) updateHeight(_root); return true; }
    // (至此，原x（现r）必非根) 若父亲（及祖先）依然平衡，则无需调整
    if (BlackHeightUpdated(*_hot)) return true;
    // (至此，必失衡) 若替代节点r为红，则只需简单地翻转其颜色
    if (IsRed(r)) { r->color = RB_BLACK; r->height++; return true; }
    // (至此，r以及被其替代的x均为黑色) 做双黑调整（入口处必有 r == NULL）
    solveDoubleBlack(r); return true;
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

双黑修正：算法

```

◆ template <typename T> //r为被删除节点的替代者
void RedBlack<T>::solveDoubleBlack(BinNodePosi(T) r) {
    BinNodePosi(T) p = r ? r->parent : _hot; if (!p) return;
    BinNodePosi(T) s = (r == p->lChild) ? p->rChild : p->lChild;
    //以下，p、s分别为r的父亲、兄弟
    if (IsBlack(s)) { //兄弟s为黑
        BinNodePosi(T) t = NULL; //以下将t取作s的红孩子
        if (HasLChild(*s) && IsRed(s->lChild)) t = s->lChild;
        else if (HasRChild(*s) && IsRed(s->rChild)) t = s->rChild;
        if (t) { /* ... 黑s有红孩子：BB-1 ... */ }
        else { /* ... 黑s无红孩子：BB-2R或BB-2B ... */ }
    } else { /* ... 兄弟s为红：BB-3 ... */ }
}

```

Data Structures & Algorithms (Fall 2012), Tsinghua University

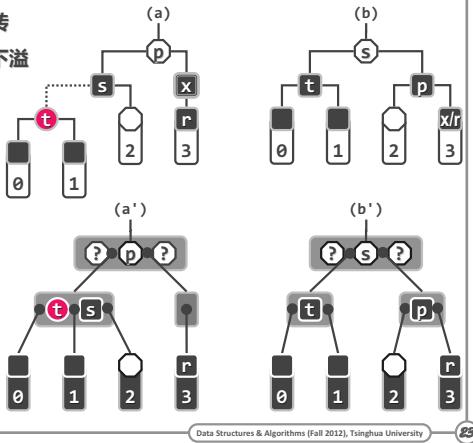
27

双黑修正(1)：s为黑，且至少有一个红孩子t

◆ 通过关键码的旋转

消除超级节点的下溢

◆ 问号节点

可同时存在
颜色不定

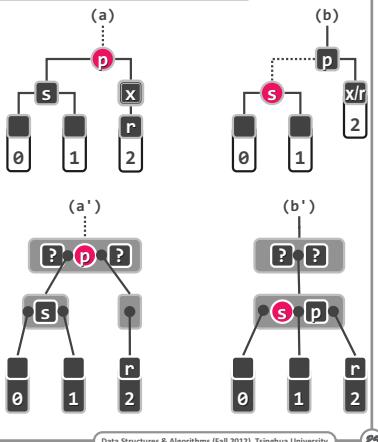
Data Structures & Algorithms (Fall 2012), Tsinghua University

28

双黑修正(2R)：s为黑，且两个孩子均为黑；p为红

◆ r保持黑；s转红；p转黑

◆ 红黑树性质在全局得以恢复

◆ 在对应的B-树中，等效于
下溢节点与兄弟合并◆ 失去关键码p后，上层节点
会否继而下溢？不会！◆ 合并之前，在p之左或右侧
还应有（问号）关键码
必为黑色
有且仅有一个

Data Structures & Algorithms (Fall 2012), Tsinghua University

29

```

/* .... */
if (IsBlack(s)) { //兄弟s为黑
    /* .... */
    if (t) { /* ... 黑s有红孩子：BB-1 ... */ }
    else { /* ... 黑s无红孩子：BB-2R或BB-2B ... */ }

    s->color = RB_RED; s->height--; //s转红
    if (IsRed(p)) //BB-2R：p转黑，但黑高度不变
        { p->color = RB_BLACK; }
    else //BB-2B：p保持黑，但黑高度下降；递归修正
        { p->height--; solveDoubleBlack(p); }

} else { /* ... 兄弟s为红：BB-3 ... */ }

```

Data Structures & Algorithms (Fall 2012), Tsinghua University

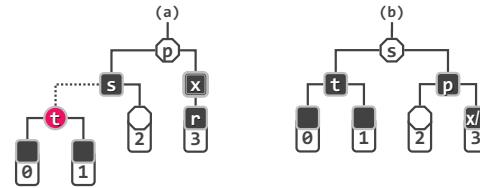
27

双黑修正(1)：s为黑，且至少有一个红孩子t

◆ 对t、s和p做“3+4”重构

这三个节点将重命名为a、b和c

r保持黑；a和c染黑；b继承p的原色



◆ 如此，红黑树性质在全局得以恢复——删除完成！ //zig-zag等类似

◆ 在对应的B-树中，以上操作等效于...

Data Structures & Algorithms (Fall 2012), Tsinghua University

28

双黑修正(1)：实现

```

/* .... */
if (IsBlack(s)) { //兄弟s为黑
    /* .... */
    if (t) { /* ... 黑s有红孩子：BB-1 */
        RBColor oldColor = p->color; //备份p颜色，并对t、父亲、祖父
        BinNodePosi(T) b = FromParentTo(*p) = rotateAt(t); //旋转
        //以下，设置旋转后的新子树
        if (HasLChild(*b)) b->lChild->color = RB_BLACK; //左子染黑
        if (HasRChild(*b)) b->rChild->color = RB_BLACK; //右子染黑
        updateHeight(b->lChild); updateHeight(b->rChild);
        b->color = oldColor; updateHeight(b); //根继承原根的颜色
    } else { /* ... 黑s无红孩子：BB-2R或BB-2B ... */ }
} else { /* ... 兄弟s为红：BB-3 ... */ }

```

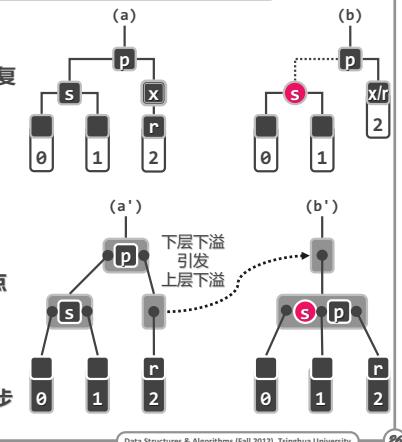
Data Structures & Algorithms (Fall 2012), Tsinghua University

29

双黑修正(2B)：s为黑，且两个孩子均为黑；p为黑

◆ s转红；r与p保持黑

◆ 红黑树性质在局部得以恢复

◆ 在对应的B-树中，等效于
下溢节点与兄弟合并◆ 合并之前，p和s
均对应于单关键码节点◆ 失去关键码p后，上层节点
必然继而下溢◆ 好在可继续分情况处理
高度递增，至多O(logn)步

Data Structures & Algorithms (Fall 2012), Tsinghua University

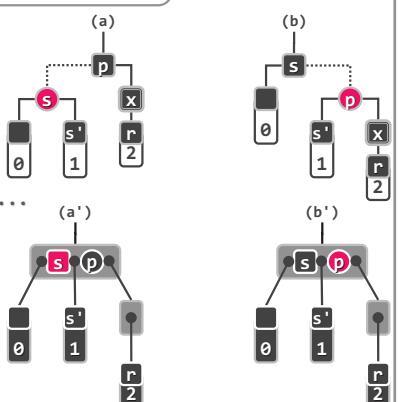
29

双黑修正(3)：s为红（其孩子均为黑）

◆ zig(p)或zag(p)

红s转黑，黑p转红

◆ 黑高度依然异常，但...

◆ r有了一个新的黑兄弟s'
故转化为前述情况，而且...◆ 既然p已转红，接下来
绝不会是情况(2B)
而只能是(1)或(2R)◆ 于是，再经一轮调整之后
红黑树性质必然全局恢复

Data Structures & Algorithms (Fall 2012), Tsinghua University

29

双黑修正(3)：实现

```

    /* ..... */
    if (IsBlack(s)) { //兄弟s为黑
        /* ..... */
        if (t) { /* ... 黑s有红孩子 : BB-1 ... */ }
        else { /* ... 黑s无红孩子 : BB-2R或BB-2B ... */ }
    } else { //兄弟s为红 : BB-3
        s->color = RB_BLACK; p->color = RB_RED; //s转黑，p转红
    }
    // 取t与其父s同侧
    BinNodePosi(T) t = IsLChild(*s) ? s->lChild : s->rChild;
    // 对t及其父亲、祖父做平衡调整
    _hot = p; FromParentTo(*p) = rotateAt(t);
    // 继续修正r处双黑—此时的p已转红，故后续只能是BB-1或BB-2R
    solveDoubleBlack(r);
}
}

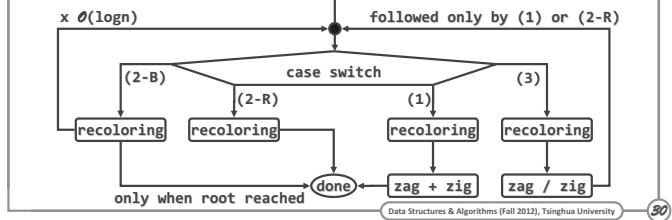
```

Data Structures & Algorithms (Fall 2012), Tsinghua University 89

双黑修正：复杂度

- ❖ 红黑树的每一删除操作都可在 $\Theta(\log n)$ 时间内完成其中
 1. 至多做 $\Theta(\log n)$ 次重染色
 2. 至多做一次“3+4”重构
 3. 至多做一次单旋

| 情况 | 旋转次数 | 染色次数 | 此后 |
|----------------|------|------|--------------|
| (1) 黑s有红子t | 1~2 | 3 | 调整随即完成 |
| (2R) 黑s无红子, p红 | 0 | 2 | 调整随即完成 |
| (2B) 黑s无红子, p黑 | 0 | 1 | 必然再次双黑但必上升一层 |
| (3) 红s | 1 | 2 | 转为(1)或(2R) |



Data Structures & Algorithms (Fall 2012), Tsinghua University 90

611

java.util.TreeMap

```

import java.util.*;
public class TestTreeMap {
    public static void main(String[] args) {
        TreeMap scarborough = new TreeMap();
        scarborough.put("P", "parsley");
        scarborough.put("S", "sage");
        scarborough.put("R", "rosemary");
        scarborough.put("T", "thyme");
        System.out.println(scarborough);
        // {P = parsley, R = rosemary, S = sage, T = thyme}
    }
}

```

8. 高级搜索树

(x2) kd-树

邓俊辉

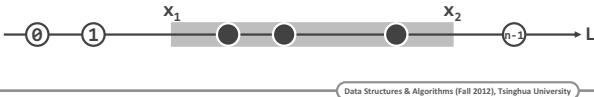
deng@tsinghua.edu.cn

612

范围查找

- ❖ 在直线L上，给定点集 $P = \{p_0, \dots, p_{n-1}\}$
- ❖ 在任意区间 $R = [x_1, x_2]$ 内：有多少个点？//计数 (counting)
有哪些点？//报告 (reporting)
- ❖ 限制：点集规模 (n) 非常大，以致于需要借助外存
因此：通过遍历整个点集，逐一测试各点将非常耗时，应尽量避免
- ❖ 问题特点：

[Offline] P相对固定，可以离线方式预处理
 [Online] R数量巨大，以非确定方式反复给出，须在线处理

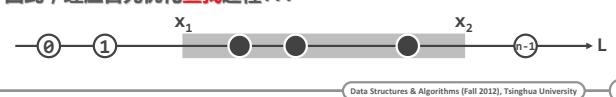


Data Structures & Algorithms (Fall 2012), Tsinghua University 91

613

蛮力

- ❖ 依次检查P中的每个点
若位于指定区间之内，则计数（或将其加至查找结果中）
- ❖ $\Theta(n)$ 时间——能否更快？就渐进意义而言似乎不能，因为...
- ❖ 最坏情况下，命中的点数 $r = \Omega(n)$
即便直接输出（枚举、打印）它们，也要花费 $\Omega(n)$ 时间
- ❖ 实际上，相对于查找过程本身，输出过程的效率不甚重要
对于简单的Counting版，不必输出命中子集
蛮力查找过程需反复I/O， $\Theta(n)$ 的常系数很大
因此，理应首先优化查找过程...



Data Structures & Algorithms (Fall 2012), Tsinghua University 92

614

二分查找

- ❖ 预处理：点集排序后转换为有序向量（哨兵 $p_{-1} = -\infty$ ）// $\Theta(n \log n)$
- ❖ 查找（针对任意区间 $R = [x_1, x_2]$ ）
 1. $t = \text{search}(x_2) = \max\{i \mid x_2 \geq p_i\}$ // $\Theta(\log n)$
 2. 从 p_t 出发，自右向左检查各点，直至首次离开查询区间的 p_s
只要当前点在范围之内，则报告之 // $\Theta(r+1) = \Theta(t-s+1)$
- ❖ 优势：时间复杂度 = $\Theta(r + \log n)$ ，输出敏感
仅涉及 $r+1$ 个点，且它们依次毗邻，I/O大大节省
对于Counting版，还可进一步优化至 $\Theta(\log n)$
- ❖ 这个方法似乎不错，但是...

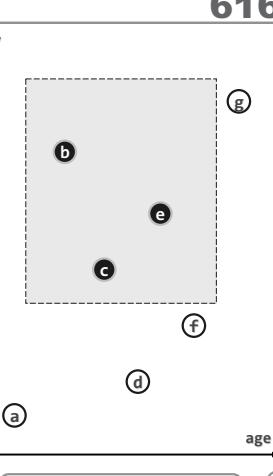


Data Structures & Algorithms (Fall 2012), Tsinghua University 93

615

平面

- ❖ 给定平面点集 $P = \{p_0, \dots, p_{n-1}\}$
- ❖ 矩形范围查找
(Rectangular Range Search)
任意矩形 $R = [x_1, x_2] \times [y_1, y_2]$ 内
Counting: 有多少个点
Reporting: 有哪些点
- ❖ 此时，计数法还能行得通吗？
- ❖ 否则，有无其它方法？
- ❖ 还有，要是空间维度更高呢？
- ❖ 回到一维情况，找出可扩展的方法...

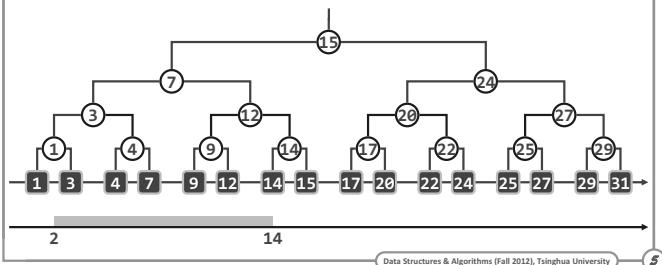


Data Structures & Algorithms (Fall 2012), Tsinghua University 94

616

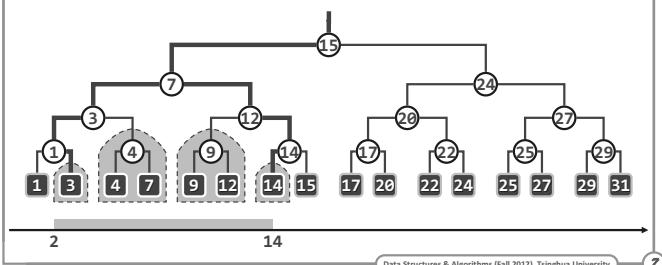
BBST : 结构

- 每个内部节点v，记录相应的划分位置x(v)
- 有序性： $LTree(v) \leq x(v) < RTree(v)$
- 比如 $x(v) = 12$ ，有 $\max\{9, 12\} \leq 12 < \min\{14, 15\}$
- 以 $R = [2, 14]$ 为例，范围查找如何实现？

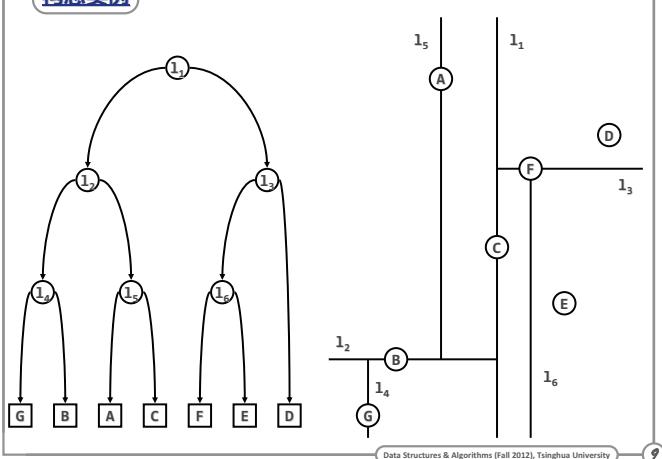


BBST : 效率

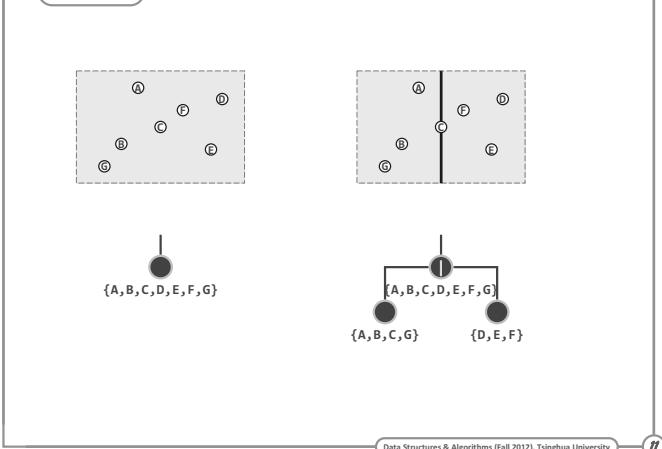
- 预处理 $\mathcal{O}(n \log n)$
- 空间 $\mathcal{O}(n)$
- 查询 $\mathcal{O}(r + \log n)$ // r = 查找命中的点数，输出敏感



构思实例

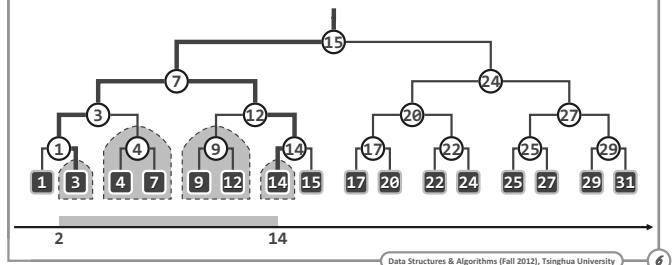


构造实例



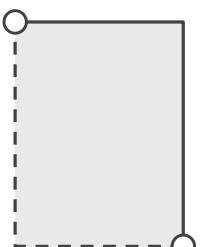
BBST : 查找

- 分别查找2和14，终止于3和14 //其最低共同祖先LCA(3, 14) = 7
- 从LCA起向下，重走一遍path(3)和path(14)
- 沿path(3)/path(14)，忽略右/左拐，一旦左/右拐则输出右/左子树
- 最后，还要单独检查path(3)和path(14)的终点



构思

- 上述数据结构，如何扩展到二维？
- 通过递归的平面划分，构造kd-树！
- 偶/奇数深度层：做垂直/水平划分**
- 两个子集规模尽可能接近
- 半平面：左开右闭、下开上闭
- 非退化约定：各点x坐标互异、y坐标互异
- 中值 (median)：非降序列 $\{a_0, a_1, \dots, a_{n-1}\}$ 中的 $a_{\lceil n/2 \rceil - 1}$
- // $\mathcal{O}(n)$ 算法——稍后讲解



构造算法

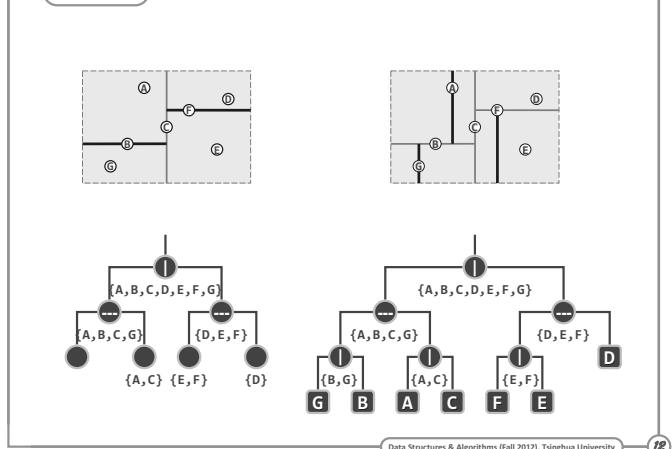
```

//Input. 点集P，及其所对应子树（根节点）的深度depth
//Output. 指针，指向存储P的一棵kd-子树（的根节点）

KdTree* createKdTree(P, depth) {
    //自顶而下，递归地逐层构造
    if (P == {p}) return createLeaf(p); //区域缩小至仅含一个点p
    root = createKdNode(); //创建节点
    root->splitDirection = depth%2 ? HORIZONTAL : VERTICAL; //切分方向
    root->splitLine = median(root->splitDirection, P); //切分位置
    (P1, P2) = divide(P, root->splitDirection, root->splitLine); //切分
    root->lChild = createKdTree(P1, depth+1); //递归构造左或上子树
    root->rChild = createKdTree(P2, depth+1); //递归构造右或下子树
    return root;
}
  
```

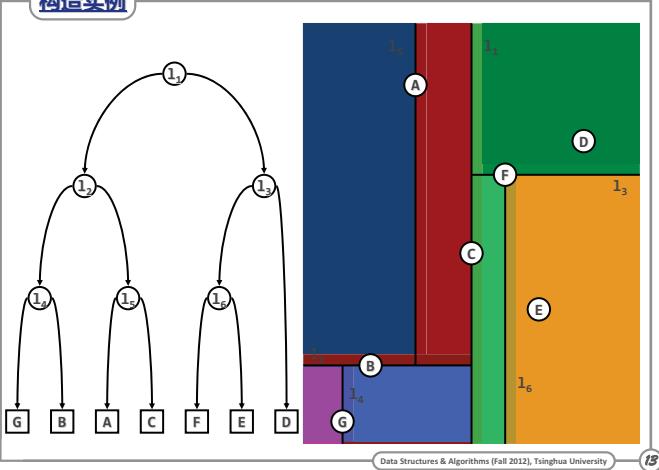
Data Structures & Algorithms (Fall 2012), Tsinghua University 10

构造实例



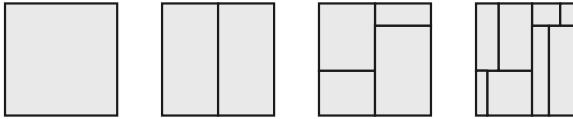
构造实例

625



正则子集

- ◆ 树节点 ~ 矩形子区域
- ◆ 同一节点左、右孩子所对应子区域之并，即该节点对应的子区域
- ◆ 同一层的所有节点对应的子区域
互不相交，而且
其并恰好覆盖整个平面



查找算法

627

```
//Input. kd-树的根节点v, 以及区域R
//Output. 所有以v为祖先、位于R之内的叶子所对应的点

kdSearch(v, R) {
    if (isLeaf(v)) { if (inside(v, R)) report(v); return; }

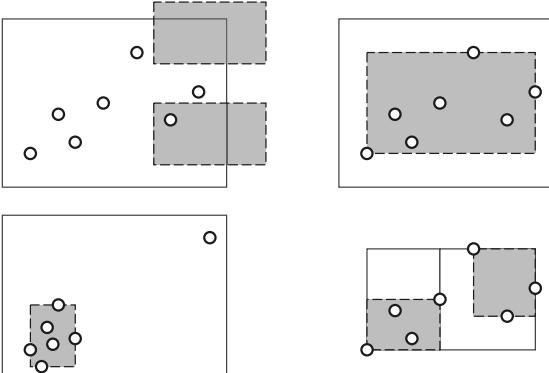
    if (region(v->lC) ⊆ R) reportSubtree(v->lC);
    else if (intersect(region(v->lC), R)) kdSearch(v->lC, R);

    if (region(v->rC) ⊆ R) reportSubtree(v->rC);
    else if (intersect(region(v->rC), R)) kdSearch(v->rC, R);
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University 15

包围盒

629



Data Structures & Algorithms (Fall 2012), Tsinghua University 17

高维推广

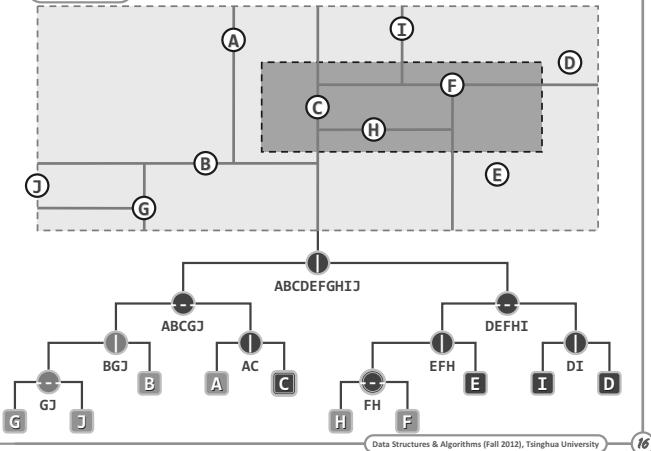
631

- ◆ 在更高维空间中，kd-树是否依然适用？ //kd = k-dimensional
- ◆ 构造 深度 = 0时，沿第0维划分
深度 = 1时，沿第1维划分
...
深度 = d-1时，沿第d-1维划分
深度 = d时，沿第d维划分
...
深度 = k时，沿第k % d维划分
- ◆ 空间： $\mathcal{O}(n)$
- ◆ 预处理： $\mathcal{O}(n \log n)$
- ◆ 查询： $\mathcal{O}(n^{1-1/d} + r)$

Data Structures & Algorithms (Fall 2012), Tsinghua University 19

查找实例

628



效率

- ◆ 空间 $\mathcal{O}(n)$
预处理 $\mathcal{O}(n \log n)$
查询 $\mathcal{O}(\sqrt{n} + r)$
- ◆ 其中前一部分（用于查找的）时间决定于
递归发生的次数，或
待查找区域边界与子区域相交的数目 $Q(n)$
- ◆ 递推关系： $Q(1) = \mathcal{O}(1)$, $Q(n) = 2 + 2Q(n/4)$
- ◆ 解得： $Q(n) = \mathcal{O}(\sqrt{n})$

Data Structures & Algorithms (Fall 2012), Tsinghua University 18

632

课后

- ◆ 在高维情况下，kd-树的效率如何评价？
- ◆ 如何消除（多点共垂线、水平线等）退化情况？
- ◆ 不借助median算法，如何构造2d-树？你所给算法的效率如何？
- ◆ 若只需计数，kd-树可在多少时间内给出解答？为此需对树做何调整？
- ◆ 适用于低（2~3）维数据的简化变种
quadtree, octree
- ◆ 三维以上的情况，有无更好的结构？
multi-level search tree, range tree
interval tree, priority search tree, segment tree

Data Structures & Algorithms (Fall 2012), Tsinghua University 20

633

8. 高级搜索树

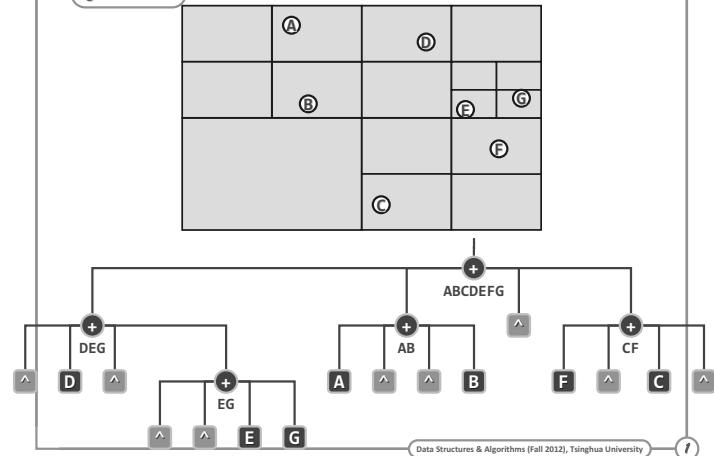
(x3) 更多变种

邓俊辉

deng@tsinghua.edu.cn

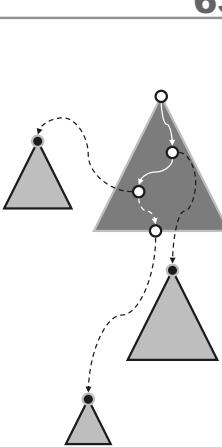
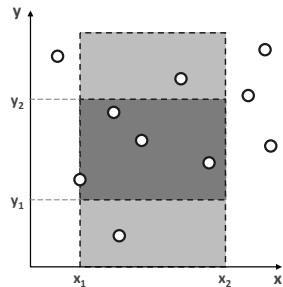
634

Quadtree



635

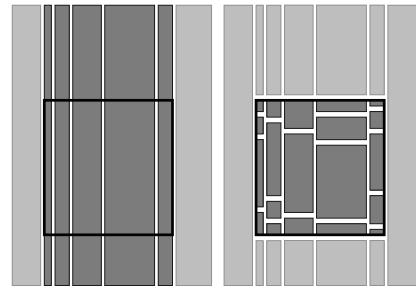
Range Tree



Data Structures & Algorithms (Fall 2012), Tsinghua University

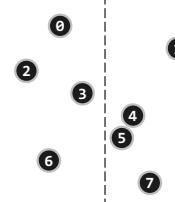
636

Range Tree

◆ 空间: $\mathcal{O}(n \log n)$ ◆ 时间: $\mathcal{O}(r + \log^2 n) \sim \mathcal{O}(r + \log n)$ 

Data Structures & Algorithms (Fall 2012), Tsinghua University

Range Tree

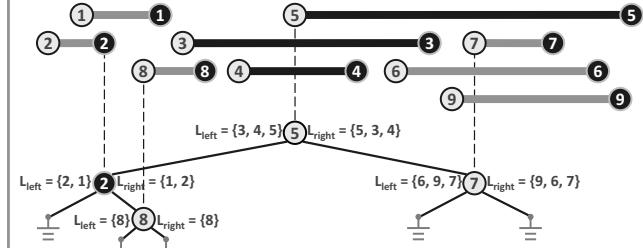


Data Structures & Algorithms (Fall 2012), Tsinghua University

637

Interval Tree

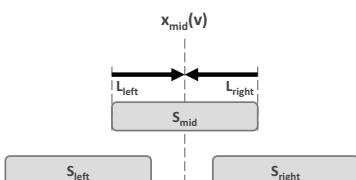
Interval Tree



Data Structures & Algorithms (Fall 2012), Tsinghua University

639

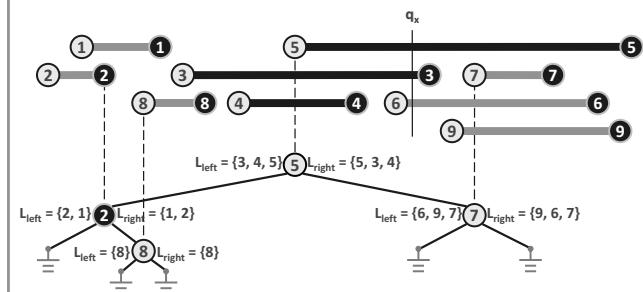
Interval Tree



Data Structures & Algorithms (Fall 2012), Tsinghua University

640

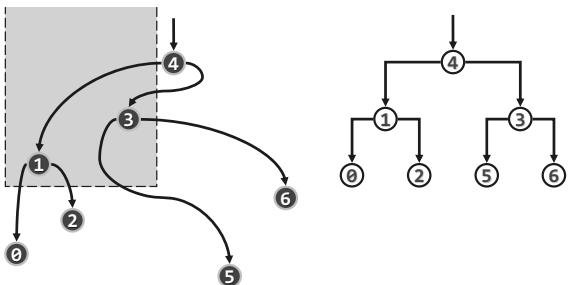
Interval Tree



Data Structures & Algorithms (Fall 2012), Tsinghua University

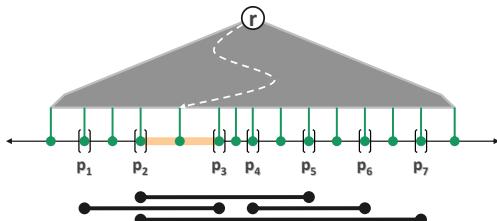
641

Priority Search Tree



Data Structures & Algorithms (Fall 2012), Tsinghua University

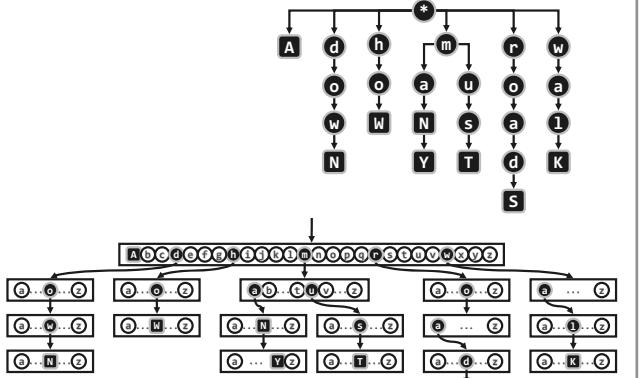
Segment Tree



Data Structures & Algorithms (Fall 2012), Tsinghua University

643

Trie = reTRIEval



Data Structures & Algorithms (Fall 2012), Tsinghua University

647

9.词典

(a) 循值访问

We are shaped by our thoughts;
we become what we think.

- Buddha

Man's thought is shaped
by his tongue.
- Anonymous

The diversity of languages is not
a diversity of signs and sounds, but
a diversity of views of the world.
- Wilhelm von Humboldt, 1820

邵俊辉

deng@tsinghua.edu.cn

映射 + 词典 = 符号表

- ❖ 词条 : entry = (key, value)
- ❖ 映射 (Map) = 词条的集合 //其中各词条的关键码互异
ADT : get(key), put(key, value), remove(key)
- ❖ 较之BST, 关键码之间不一定可比较 (call-by-value)
较之PQ, 查找对象更广泛, 不限于最大、最小词条
- ❖ 词典 (Dictionary)
与映射基本相同, 但允许多个词条具有相同的key
Sorted dictionary : 关键码之间可定义全序关系的词典
- ❖ 映射和词典都是动态的, 统称符号表 (symbol table)

Data Structures & Algorithms (Fall 2012), Tsinghua University

Dictionary

- ❖ template <typename K, typename V> //key、value
class Dictionary { //Dictionary模板类
public:
 virtual int size() = 0; //查询当前的词条总数
 virtual bool put(K, V) = 0; //插入词条(key, value)
 virtual V* get(K k) = 0; //查找以key为关键码的词条
 virtual bool remove(K k) = 0; //删除以key为关键码的词条
};
- ❖ 尽管诸如Java::TreeMap等实现仍然要求支持比较器, 但实际上
词典中的词条, 只需支持比对等操作, 而不必支持大小比较

Data Structures & Algorithms (Fall 2012), Tsinghua University

Dictionary

- ❖ 在这里, 无论对外的访问方式, 还是内部的存储方式
都更直接地依据数据对象自身的取值
key与value地位等同, 不必区分
- get("Yide") → 
- ❖ 循值访问 (call-by-value)
访问方式更为自然, 适用范围也更广泛
回忆一下, 初次接触程序设计时, 你首先想到的应该就是这种方式

Data Structures & Algorithms (Fall 2012), Tsinghua University

Perl: %Hash Type

- ❖ 由字符串 (string) 标识的一组无序标量 (scalar) //亦即MAP
my %hero = (# Hash类型的变量由%引导
 "yunchang"=>"Guan Yu", "yide"=>"Zhang Fei",
 "zilong"=>"Zhao Yun", "mengqi"=>"Ma Chao");
\$hero{"hansheng"} = "Huang Zhong";
foreach \$style (keys %hero)
{ print "\$style => \$hero{\$style}\n"; }
foreach \$style (reverse sort keys %hero)
{ print "\$style => \$hero{\$style}\n"; }

Data Structures & Algorithms (Fall 2012), Tsinghua University

Ruby: Hash Table

- ```
scarborough = { # declare and initialize a hash table
 "P"=>"parsley", "S"=>"sage",
 "R"=>"rosemary", "T"=>"thyme"
}

puts scarborough # output the hash table
for k in scarborough.keys # output hash table items
 puts k + "=>" + scarborough[k] # 1-by-1
end

for k in scarborough.keys.sort # output hash table items
 puts k + "=>" + scarborough[k] # 1-by-1 in order
end
```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## Java: HashMap + Hashtable

```
import java.util.*;
public class Hash {
 public static void main(String[] args) {
 HashMap HM = new HashMap(); //Map
 HM.put("east", "Mt. Tai"); HM.put("west", "Mt. Hua");
 HM.put("south", "Mt. Heng"); HM.put("north", "Mt. Heng");
 HM.put("central", "Mt. Song"); System.out.println(HM);
 Hashtable HT = new Hashtable(); //Dictionary
 HT.put("east", "Mt. Tai"); HT.put("west", "Mt. Hua");
 HT.put("south", "Mt. Heng"); HT.put("north", "Mt. Heng");
 HT.put("central", "Mt. Song"); System.out.println(HT);
 }
}
```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## Python: Dictionary Class

```
beauty = dict({ # Python dictionary (hashtable)
 "X":"Xi Shi", "D":"Diao Chan",
 "W":"Wang Zhaojun", "Y":"Yang Yuhuan"})
beauty["C"] = "Chen Yuanyuan"
print(beauty)

for id,name in beauty.items():
 print(id,":",name,"")
for id,name in sorted(beauty.items()):
 print(id,":",name,"")
for id in sorted(beauty.keys(),reverse=True):
 print(id,":", beauty[id],"")
```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 课后

- ❖ 了解HashMap与Hashtable的异同
- ❖ 安装JDK (<http://www.java.com>), 尝试HashMap和Hashtable类
- ❖ 安装Perl (<http://www.perl.org>), 尝试%Hash类型
- ❖ 安装Python (<http://www.python.org>), 尝试Dictionary类
- ❖ 安装Ruby (<http://www.ruby-lang.org>), 尝试Hash Table

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University



- ❖ 是否存在某种定址方法，能保证不出现冲突？  
亦即，散列函数等效于一个单射 (injection) ?
- ❖ 在关键码满足某些条件时，的确可以实现单射式散列，比如...
- ❖ 对已知且固定的关键码集，可实现完美散列 (perfect hashing)
  - 采用两级散列模式
  - 仅需  $\Theta(n)$  空间
  - 关键码之间互不冲突
  - 即便在最坏情况下，查找时间也不过  $\Theta(1)$  时间
- ❖ 不过，在一般情况下，完美散列无法保证存在...

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

8

- ❖ 将在座同学（对应的词条）按生日（月/日）做散列存储  
散列表长固定为  $M = 365$ ，装填因子 = 在场人数  $N / 365$
- ❖ 冲突（至少有两位同学生日相同）的可能性  $P_{365}(n) = ?$   
//《概率论与数理统计》讲义第一章，清华大学数学系王晓峰  
 $P_{365}(21) = 44.4\%$ ,  $P_{365}(22) = 47.6\%$ , ...  
 $P_{365}(23) = 50.7\%$  //此时，装填因子 =  $23/365 = 6.3\%$
- ❖ 100人的集会： $1 - P_{365}(100) = 0.000031\%$   
自7岁起，不吃不喝、无休无息，每小时参加四次  
到100岁，才有可能遇到一次没有冲突的集会
- ❖ 因此，在装填因子确定之后  
散列策略的选取将至关重要，散列函数的设计也很有讲究...

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

9

## 9. 词典

## (c) 散列表——散列函数

邓俊辉

deng@tsinghua.edu.cn

- ❖  $hash(key) = key \% M$  //前例中，为何选  $M = 90001$  ?
- ❖ 若取  $M = 2^k$   
//其效果，相当于截取key的最后k位 (bit)  
 $M-1 = 0\ 0\ 0\ \dots\ 0\ 1\ 1\ 1\ \dots\ 1$   
 $key \% M = key \& (M-1)$   
观察：前面的  $n-k$  位对地址没有影响  
推论：发生冲突 iff 最后k位相同 //发生冲突的概率大
- ❖ 若取  $M \neq 2^k$ ，效果有所改进
- ❖  $M$  为素数时，数据对散列表的覆盖最充分，分布最均匀 //为什么
- ❖ 还有哪些散列策略呢？

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

8

- ❖ 在无法实现单射的前提下，什么样的散列函数  $hash()$  更“好”？
  - 1) 确定 (determinism)  
同一关键码总是被映射至同一地址
  - 2) 快速计算 (efficiency)  
expected- $\Theta(1)$
  - 3) 满射 (surjection)  
尽可能充分地覆盖整个散列空间
  - 4) 均匀 (uniformity)  
关键码映射到散列表各位置的概率尽量接近  
可有效地避免聚集 (clustering) 现象

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

f

- ❖ MAD (multiply-add-divide) 法  
取  $M$  为素数， $a > 0, b > 0, a \% M \neq 0$   
 $hash(key) = (a \times key + b) \% M$
- ❖ 数字分析法/selecting digits  
抽取key中的某几位，构成地址  
比如，取十进制表示的奇数位  
 $hash(1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9) = 1\ 3\ 5\ 7\ 9$
- ❖ 平方取中法 (mid-square)  
取  $key^2$  的中间若干位，构成地址  
 $hash(123) = 512$  //保留  $key^2 = 123^2 = 15129$  的中间3位  
 $hash(1234567) = 556$  // $1234567^2 = 1524155677489$

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

g

- ❖ 折叠法/folding  
将key分割成等宽的若干段，取其和作为地址  
 $hash(123456789) = 1368$  //123 + 456 + 789，自左向右  
 $hash(123456789) = 1566$  //123 + 654 + 789，往复折返
- ❖ 位异或法/XOR  
将key分割成等宽的二进制段，经异或运算得到地址  
 $hash(110011011_b) = 110_b$  //110 ^ 011 ^ 011，自左向右  
 $hash(110011011_b) = 011_b$  //110 ^ 110 ^ 011，往复折返
- ❖ ...
- ❖ 总之，越是随机，越是沒有规律，越好

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

g

- ❖ (伪) 随机数发生器  
循环： $rand(x+1) = [a \times rand(x)] \% M$  // $M$  素数， $a \% M \neq 0$   
例如： $a = 7^5 = 16,807 = 10000110100111_b$   
 $M = 2^{31} - 1 = 2,147,483,647$   
 $= 01111111\ 11111111\ 11111111\ 11111111_b$
- ❖ (伪) 随机数法  
径取： $hash(key) = rand(key) = [rand(0) \times a^{key}] \% M$   
种子： $rand(0) = ?$
- ❖ 把难题推给伪随机数发生器，但是...
- ❖ (伪) 随机数发生器的实现，因具体平台、不同历史版本而异  
创建的散列表可移植性差——故需慎用此法！

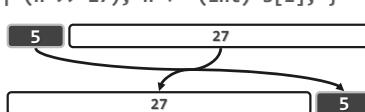
Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

g

## 多项式法

❖ hash(s = "x<sub>0</sub>x<sub>1</sub>...x<sub>n-1</sub>")  
 $= x_0a^{n-1} + x_1a^{n-2} + \dots + x_{n-2}a^1 + x_{n-1}$   
 $= (\dots((x_0a + x_1)a + x_2)a + \dots x_{n-2})a + x_{n-1}$

❖ static size\_t hashCode(char s[]) {  
 int h = 0;  
 for (size\_t n = strlen(s), i = 0; i < n; i++)  
 { h = (h << 5) | (h >> 27); h += (int) s[i]; }  
 return (size\_t) h; }



❖ 有必要如此复杂吗？能否使用更简单的散列，比如...

## (伪)随机数法

```
❖ [KR88].The C Programming Language (2nd edn), p46

❖ unsigned long int next = 1;

void srand(unsigned int seed) { next = seed; }

int rand(void) { //1103515245 = 3^5 * 5 * 7 * 129749

 next = next * 1103515245 + 12345;

 return (unsigned int)(next/65536) % 32768;

}

next 2^32

rand 2^15
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

6

## 多项式法

❖ 字符分别映射为数字：f(c) = CODE(UPPER(c)) - 64

['A', 'Z'] ~ [1, 26]

再简单相加：hash(s) =  $\sum_{c \in s} f(c)$

"hash" ~ 8 + 1 + 19 + 8 = 36



❖ 字符相对次序信息丢失，将引发大量冲突

"I am Lord Voldemort"

= "Tom Marvolo Riddle"

❖ 即便字符数目不等...

= "He's Harry Potter"



❖ "Key to improving your programming skills"

= "Learning Tsinghua Data Structures & Algorithms"

Data Structures & Algorithms (Fall 2012), Tsinghua University

8

## Java:hashCode()

❖ hashCode()方法

适用于Java中的所有对象

将任意类的对象转换为(32位int型)整数

对于非整型的key，先转换为整数(散列码)，然后再做散列

❖ hashCode()：效果如何？效率如何？是如何实现的？

❖ object.hashCode() = object在内存中的地址

❖ 问题：相关的对象地址也相近，冲突概率高 //更糟糕的是...

❖ 散列码与对象的内容无关

比如，完全相等的两个字符串对象，散列码居然不同

❖ 有何替代方案？

Data Structures & Algorithms (Fall 2012), Tsinghua University

9

## 9.词典

## (d) 散列表——排解冲突

邓俊辉

deng@tsinghua.edu.cn

## 普遍存在的冲突

❖ 一般情况下，无法完全杜绝冲突

词条空间S ~ 可能的词条

地址空间A ~ 散列表

散列函数hash(): S → A

既然|S| = R >> M = |A|，hash()就不可能是单射

❖ 因此，我们除了需要采用以上策略

精心设计散列表及散列函数，以尽可能降低冲突的概率；更需要

制定可行的预案，以便在发生冲突时能够尽快予以排解甚至预防...

Data Structures & Algorithms (Fall 2012), Tsinghua University

10

## 多槽位法

❖ Multiple slots

桶单元细分成若干槽位(slot)

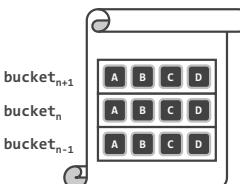
存放(与同一单元)冲突的词条

❖ 优点：只要槽位数目不多，依然可以保证O(1)的操作时间

❖ 但是，需要为每个桶配备多少个槽，方能保证O(1)？ //难以预测

预留的槽位造成空间浪费

无论预留多少个槽位，极端情况下仍有可能不够



Data Structures & Algorithms (Fall 2012), Tsinghua University

11

## 独立链法

❖ Linked-list chaining / separate chaining

每个桶存放一个指针

冲突的词条，组织成一个动态链表

❖ 优点 无需为每个桶预备多个槽位

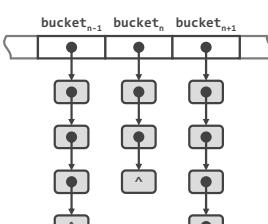
任意多次的冲突都可解决

删除操作实现简单、统一

❖ 但是 指针需要额外空间

结构本身实现复杂

链表长度没有上界



最坏情况下，每次操作所需时间正比于表长，不再是O(1)

Data Structures & Algorithms (Fall 2012), Tsinghua University

12



## 双向平方法探法

◆ 表长取作素数  $M = 4xk + 3 \quad // 3 \ 5 \ 7 \ 11 \ 13 \ 17 \ 19 \ 23 \ 29 \ 31 \dots$

即可保证查找链的前  $M$  项均互异，从而将冲突的概率降至最低

◆ 定理 (Two-Square Theorem of Fermat) :

任一素数  $p$  可表示为一对整数的平方和

当且仅当  $p \% 4 = 1$

◆ 只要注意到：

$$(u^2 + v^2)(s^2 + t^2) = (us + vt)^2 + (ut - vs)^2$$

$$(2^2 + 3^2)(5^2 + 8^2) = (10 + 24)^2 + (16 - 15)^2$$



◆ 就不难推知：

任一自然数  $n$  可表示为一对整数的平方和，当且仅当在其素分解中形如  $p = 4m + 3$  的每一素因子均为偶数次方

Data Structures & Algorithms (Fall 2012), Tsinghua University

17

## 再散列法

◆ Double hashing

第二散列函数：hash2()

发生冲突后，以 hash2(key) 为偏移增量，重新确定地址

$$[\text{hash}(\text{key}) + 1 \times \text{hash2}(\text{key})] \% M$$

$$[\text{hash}(\text{key}) + 2 \times \text{hash2}(\text{key})] \% M$$

$$[\text{hash}(\text{key}) + 3 \times \text{hash2}(\text{key})] \% M$$

...直到发现一个空桶

◆ hash2() 为常值函数时，即退化为...

Data Structures & Algorithms (Fall 2012), Tsinghua University

18

## 预防冲突：重散列

◆ template <typename K, typename V> //装填因子增大，将导致冲突激增  
void Hashtable<K, V>::rehash() { //必要时，需“集体搬家”至更大的表  
 int old\_capacity = M; Entry<K, V>\*old\_ht = ht; N = 0;  
 ht = new Entry<K, V>\*[M = primeNLT(2\*M)]; //新表容量至少加倍  
 memset(ht, 0, sizeof(Entry<K, V>)\*M); //初始化各桶  
 release(lazyRemoval); //释放原懒惰删除标记位图  
 lazyRemoval = new Bitmap(M); //新建位图（容量同样至少加倍）  
 for (int i = 0; i < old\_capacity; i++) //扫描原桶数组  
 if (old\_ht[i]) //将非空桶中的词条逐一  
 put(old\_ht[i]->key, old\_ht[i]->value); //插入至新表  
 release(old\_ht); //释放原桶数组  
}

Data Structures & Algorithms (Fall 2012), Tsinghua University

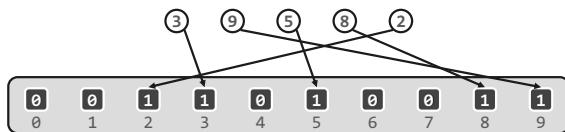
19

## 桶排序：简单情况

◆ 给定  $[0, m]$  内的  $n$  个互异整数 //必有  $n \leq m$

如何高效地排序？ //将这些整数视作词条的关键码

◆  $\Theta(m)$  空间 +  $\Theta(n+m)$  时间...



◆ 用散列表  $E[0..m-1]$  //各元素仅需 1 个 bit

```
for i = 0 to m-1, let E[i] = 0 // $\Theta(m)$, 可优化至 $\Theta(1)$
for each key in the input, let E[key] = 1 // $\Theta(n)$
for i = 0 to m-1, output i if E[i] = 1 // $\Theta(m)$
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

## 桶排序：一般情况

◆ 算法：初始化散列表（开辟空间、设置各桶的表头）

扫描各词条，散列并插至对应桶的链表 //插入位置有讲究

扫描各桶，串接所有非空链表 //串接次序和方向也有讲究

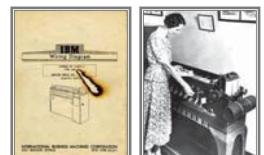
◆ 只要实现得当，必能保证稳定性，即雷同词条的次序与输入相同

◆ 时间复杂度：

$$\Theta(m) + \Theta(n) + \Theta(m) = \Theta(n+m)$$

◆ 若大量词条重复 ( $m \ll n$ )

则性能接近于线性



## 桶排序：一般情况

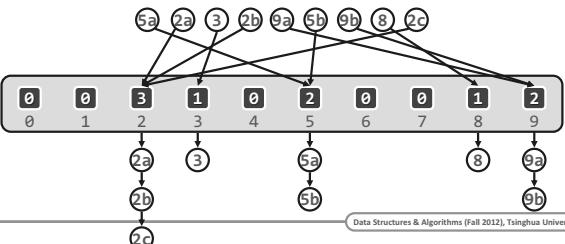
◆ 进一步地，若允许关键码重复 //此时未必  $n \leq m$ ，甚至可能  $m \ll n$

比如，清华大学2011级本科生按生日排序，则有  $n = 3300$ ,  $m = 365$

◆ 思路：依然使用散列表 //每个桶可容纳多个词条

相互冲突的词条，组织为一个链表 //独立链法

◆ 空间复杂度 = 散列表长 + 所有链表总长 =  $\Theta(m+n)$  //改用向量呢



Data Structures & Algorithms (Fall 2012), Tsinghua University

20

Data Structures & Algorithms (Fall 2012), Tsinghua University

21

## MaxGap

- 任意n个互异点均将实轴分为n-1段有界区间 //另加两段无界的其中，哪一段最长？ //MaxGap
- 平凡算法  
对所有点排序 //最坏情况下 $\Omega(n \log n)$   
依次计算各相邻点对的间距，保留最大者 // $\Theta(n)$
- 可否更快？



4

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University 5

- 有时，关键码由多个域( $k_t, k_{t-1}, \dots, k_1$ )组成  
若将各域视作字母，则关键码即单词，于是可按照词典方式排序
- 例：  
 $\spadesuit A > \spadesuit K > \spadesuit Q > \spadesuit J > \spadesuit 10 > \dots > \spadesuit 2 >$   
 $\heartsuit A > \heartsuit K > \heartsuit Q > \heartsuit J > \heartsuit 10 > \dots > \heartsuit 2 >$   
 $\diamond A > \diamond K > \diamond Q > \diamond J > \diamond 10 > \dots > \diamond 2 >$   
 $\clubsuit A > \clubsuit K > \clubsuit Q > \clubsuit J > \clubsuit 10 > \dots > \clubsuit 2 >$
- 算法：自 $k_1$ 到 $k_t$ ，依次以各域为序，做一趟桶排序 //低位优先！
- 实例：  
441 276 320 214 698 280 112  
320 280 441 112 214 276 698  
112 214 320 441 276 280 698  
112 214 276 280 320 441 698

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University 6

6

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University 7

## 基数排序：正确性与性能

- 以上算法的正确性何以见得？ //数学归纳...
- 归纳假设：在经过算法的前*i*趟之后，所有词条关于最低的*i*位有序
- 第一趟易见成立；现假设前*i-1*趟均成立，第*i*趟后无非两种情况
  - 第*i*位不同的词条：经第*i*趟后，相互之间必然变为有序
  - 第*i*位相同的词条：得益于桶排序的稳定性，必保持原有次序
- 由以上分析也可发现，基数排序同样是稳定的 //只要实现得法
- 运行时间
  - 各趟桶排序所需时间之和
  - $= (n+2m_1) + (n+2m_2) + \dots + (n+2m_t)$  // $m_k$ 为各域的取值范围
  - $= O(t \times (n+m))$  // $m = \max\{m_1, \dots, m_t\}$

## 整数排序

- 对于 $[0, n^d]$ 内的任意n个整数，如何高效排序？ //常数 $d > 1$
- 理解： $1/d = \log n / \log(n^d)$  //常对数密度，实际应用中不难满足
- 预处理：将所有关键码转换为n进制形式 $x = (x_d, \dots, x_2, x_1)$
- 于是，原问题转化为d个域的基数排序问题，可套用前述算法
- 排序时间 =  $d(n+n) = O(n)$  //“突破”了此前确定的下界！
- 原因：1) 整数取值范围有限制  
2) 不再是基于比较的计算模式
- 预处理的用时尚未计入，它本身需要多少时间？  
回忆一下，此前的相关内容...

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University 8

## 9.词典

(x1) 跳转表

去沿江上下，或二十里，或三十里，  
选高阜处置一烽火台，每台用五十军  
守之。

邓俊辉

deng@tsinghua.edu.cn

## 动机与思路

- 可否综合向量与列表的优势，高效地实现词典接口？  
具体地，如何使得各接口的效率均为 $O(\log n)$ ？
- William Pugh, 1989  
*Skip Lists: A Probabilistic Alternative to Balanced Trees*  
Proc. Workshop on Algorithms and Data Structures, 437-449
 

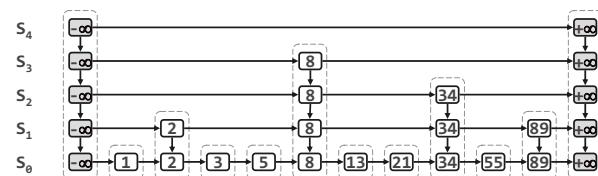
Skip lists are data structures that use probabilistic balancing rather than strictly enforced balancing  
Algorithms for insertion and deletion in skip lists are much simpler and significantly faster than equivalent algorithms for balanced trees
- 本节介绍非确定型跳转表，确定型(deterministic)跳转表可自学



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University 8

## 结构：设计

- 分层次、相互耦合的多个列表： $s_0, s_1, s_2, \dots, s_h$  //层高 =  $h$   
<http://thudsa.3322.org/~deng/ds/demo/bst/>
- $s_h/s_0$ 称作顶层 (top) / 底层 (bottom)
- 各节点至多拥有四个指针
  - 横向为层 (level) : prev()、next() //设有头、尾哨兵
  - 纵向成塔 (tower) : above()、below()



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University 9

## 空间性能

- 思考：较之单列表，需要访问的节点数目是否有实质增加？  
每个节点都可能重复多达 $h$ 次，空间复杂度因此有实质增加？  
首先来回答后者…
- “生长概率逐层减半”条件：  
 $S_{k-1}$ 中任一关键码在 $S_k$ 中依然出现的概率，均为 $1/2$   
——暂且假设成立，稍后说明如何保证
- 推论： $S_k$ 中任一关键码在第 $k$ 层依然出现的概率均为 $2^{-k}$   
 $k$ 层节点数的期望值 $E(|S_k|) = n \times 2^{-k} = n/2^k$
- 定理：跳转表所需空间为 $\Theta(n)$   
空间总量，即各层列表空间之和  
 $= E(\sum_k |S_k|) = \sum_k E(|S_k|) = n \times (\sum_k 2^{-k}) < 2n = \Theta(n)$

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

4

## 结构：QuadList

```
template <typename T> class QuadList { //四联表
private: int _size; QuadlistNodePosi(T) header, trailer; //规模、哨兵
protected: void init(); int clear(); //初始化、清除所有节点
public:
 QuadlistNodePosi(T) first() const
 { return header->succ; } //首节点
 QuadlistNodePosi(T) last() const
 { return trailer->pred; } //末节点
 T remove(QuadlistNodePosi(T) p); //删除p
 QuadlistNodePosi(T) insertAfterAbove(T const& e, //数据项e, 使之成为
 QuadlistNodePosi(T) p, //p的后继, 以及
 QuadlistNodePosi(T) b = NULL); //b的上邻
};
```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 结构：SkipList

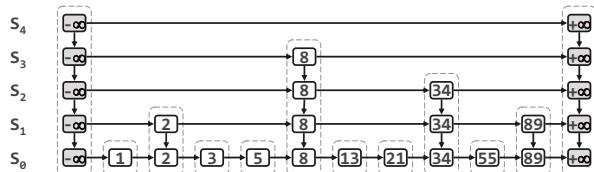
```
template <typename K, typename V> class SkipList {
public Dictionary<K, V>,
public List<QuadList<Entry<K, V>>> { //多重继承
protected:
 bool skipSearch(//从指定层qlist的首节点p出发
 ListNode<QuadList<Entry<K, V>>> *qlist,
 QuadlistNode<Entry<K, V>>* &p, K& k);
 public:
 int size() //词条总数, 即底层Quadlist的规模
 { return empty() ? 0 : last()->data->size(); }
 int level() { return List::size(); } //层高, 即Quadlist总数
 bool put(K k, V v); //插入 (SkipList允许词条重复, 故必然成功)
 V* get(K k); //读取 (基于skipSearch()直接实现)
 bool remove(K k); //删除
};
```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

6

## 查找：实例

- 实例：成功（21, 34, 1, 89），失败（80, 0, 99）



- 体会：得益于哨兵的设置，哪些环节被简化了？  
无论成功或失败，如何保证 $p$ 为其中不大于 $e$ 的最大者？
- 思考：查找时间取决于横向、纵向的累计跳转次数  
那么，是否可能因层次过多，首先导致纵向跳转过多？…

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

8

## 查找：纵向跳转/层高

- 引理：随着 $k$ 的增加，第 $k$ 层为空的概率急剧上升  
准确地， $\Pr(|S_k| = 0) \geq 1 - n/2^k$
- 于是：第 $k$ 层不为空，当且仅当 $n$ 个节点中至少有一个在其中出现  
亦即， $\Pr(|S_k| > 0) \leq n \times 2^{-k}$
- 推论：跳转表高度 $h = \Theta(\log n)$ 的概率极大
- 比如： $\Pr(h < 3\log n) \geq 1 - n^{-2}$   
考察第 $k = 3\log n$ 层  
该层为空（当且仅当  $h < k$ ）的概率  
 $\Pr(h < k) \geq \Pr(|S_k| = 0) \geq 1 - n/2^k = 1 - n/n^3$
- 结论：查找过程中，纵向跳转的累计次数不过 $\Theta(\log n)$

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 查找：横向跳转/紧邻塔顶

- 那么：横向跳转是否可能很多次？比如 $\omega(\log n)$ ，甚至 $\Omega(n)$ ？
- 观察：在同一高度，横向跳转所经过的节点必然依次紧邻，而且每次抵达的节点都是塔顶
- 于是：沿同层列表跳转的期望次数不超过其中相互紧邻的塔顶的期望数目
- 定理：同层列表中，紧邻塔顶的期望数目为2  
(由“生长概率逐层减半”条件，任一节点是塔顶的概率均为 $1/2$ …)
- 推论：查找过程中横向跳转所需时间  $\propto$  其间的横向跳转总次数  
 $\leq \text{expected}(2h) = \Theta(\log n)$
- 结论：跳转表的每次查找可在 $\Theta(\log n)$ 时间内完成

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

10

## 插入：put()

```
template <typename K, typename V>
bool SkipList<K, V>::put(K k, V v) {
 Entry<K, V>* e = new Entry<K, V>(k, v);
 if (empty()) insertAsFirst(new QuadList<Entry<K, V>>); //首个Entry
 ListNode<QuadList<Entry<K, V>>> *qlist = first(); //从顶层列表的
 QuadlistNode<Entry<K, V>>* p = qlist->data->first(); //首节点开始
 if (skipSearch(qlist, p, k)) //查找适当的插入位置
 while (p->below) p = p->below; //若已有雷同词条，则需强制转到塔底
 qlist = last(); //以下，紧邻于p的右侧，一座新塔将自底而上逐层生长
 QuadlistNode<Entry<K, V>>* b = NULL; //将e插入于
 = qlist->data->insertAfterAbove(e, p); //p的右侧 (作为新塔的基座)
/* TBC */
```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

11

## 插入 : put() (续)

```

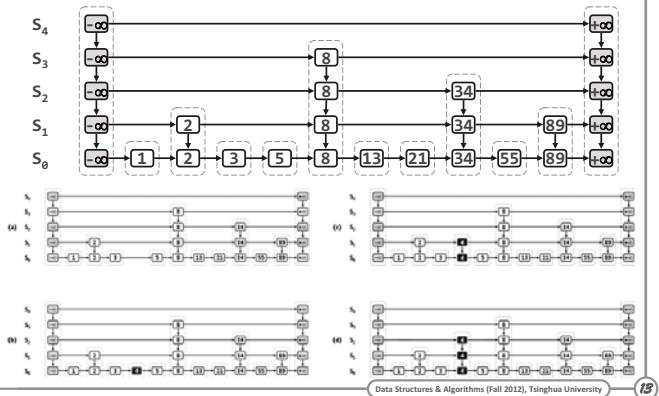
while (rand() % 2) { //经投掷硬币，若确定新塔需要再长高一层，则
 while (qlist->data->valid(p) && !p->above) //找出不低于此高度的
 p = p->pred; //最近前驱
 if (!qlist->data->valid(p)) { //若该前驱是header
 if (qlist == first()) //且当前已是最顶层，则意味着必须
 insertAsFirst(new Quadlist<Entry<K, V*>); //先创建新层，再
 p = qlist->pred->data->first()->pred; //将p转至上一层的header
 } else //否则，可径自
 p = p->above; //将p提升至该高度
 qlist = qlist->pred; //上升一层，并在该层将新节点
 b = qlist->data->insertAfterAbove(e, p, b); //插至p之后、b之上
}
return true; //SkipList允许重复元素，故插入必成功
} //留意：得益于哨兵的设置，哪些环节被简化了？

```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 插入 : 实例

❖ 实例：一般 (4, 20, 40)，边界 (0, 99)



## 9.词典

(x2) MD5

邓俊辉

deng@tsinghua.edu.cn

## 删除 : remove()

```

template <typename K, typename V>
bool SkipList<K, V>::remove(K k) {
 if (empty()) return false; //空表情况
 ListNode<Quadlist<Entry<K, V*>>* qlist = first(); //从顶层Quadlist
 QuadlistNode<Entry<K, V*>* p = qlist->data->first(); //从首节点开始
 if (!skipSearch(qlist, p, k)) return false; //目标词条不存在，直接返回
 do { //若目标词条存在，则逐层拆除与之对应的塔
 QuadlistNode<Entry<K, V*>* lower = p->below; //记住下一层节点
 qlist->data->remove(p); //删除当前层节点
 p = lower; qlist = qlist->succ; //转入下一层
 } while (qlist->succ); //直到塔基
 while (!empty() && first()->data->empty()) //反复
 List::remove(first()); //清除已可能不含词条的顶层Quadlist
 return true; //删除操作成功完成
} //留意：得益于哨兵的设置，哪些环节被简化了？

```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## MD5

❖ 网络下载：除数据包本身，是否还注意过对应的MD5？

❖ MD5 有何作用？

❖ UNIX系统：/etc/passwd文件

```

#LOGINNAME:PASSWORD:UID:GID:USERINFO:HOME:SHELL
root:zPDeHbougaPpA:0:0:Super User:/bin/sh
ftp:xyDfccTrt180xMy8:3:3:FTP User:/usr/spool/ftp:
pat:xmotTVoyumjls:6:4:DS Student:/bin/csh
#.....

```

❖ PASSWORD域，是如何计算出来的？如何验证的？

反过来，会暴露原文吗？

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

❖ 在很多场合，MD5都可大显身手...

- ❖ 数字签名：授权文档一经篡改随即失效 //电子形式签署的法律文书
- ❖ 身份验证：OS或应用软件，根据口令做账号授权 //充分而不必要
- ❖ 隐私通知：“你我知道是指谁，别人不知道是指谁” //手机尾号
- ❖ 寻源探源：文档之间的（局部、逻辑）复制关系 //学术诚信鉴别
- ❖ 信道校验：带宽有限时，快速确认大数据的一致性 //分布式存储
- ❖ 鉴别赝品：系统文件是否被病毒篡改？被何种病毒篡改？ //特征
- ❖ ...

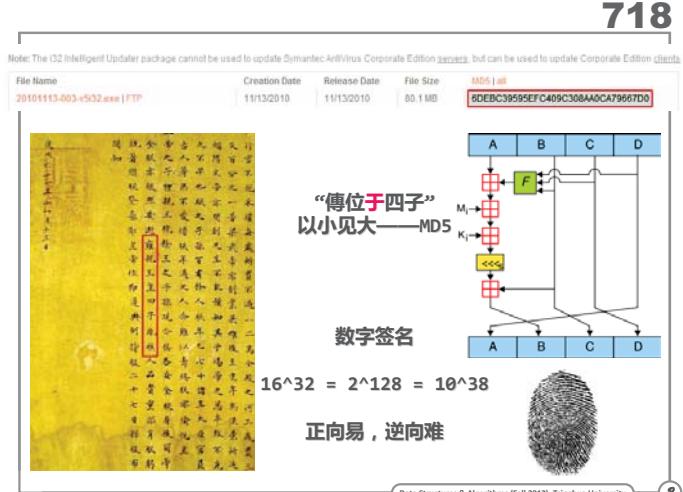
Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 9.词典

(x2) MD5

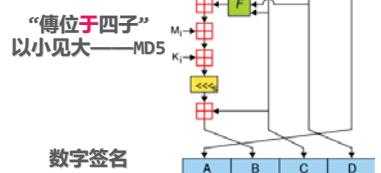
邓俊辉

deng@tsinghua.edu.cn



Note: The 32 Intelligent Updater package cannot be used to update Symantec AntiVirus Corporate Edition servers, but can be used to update Corporate Edition clients.

File Name	Creation Date	Release Date	File Size	MD5 Hash
20101113-003-e532.exe   FTP	11/13/2010	11/13/2010	60.1 MB	8DEB03959EFC0409C308AA0CA79567D0



$$16^{32} = 2^{128} = 10^{38}$$

正向易，逆向难



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 散列指纹

❖ 方法：在文档中选取若干序列，通过散列生成指纹

❖ 简单的累加？极易冲突！

此前的实例：“I am Lord Voldemort”

```

= "Tom Marvolo Riddle"
= "He's Harry Potter"

```

❖ 问题：序列越多越好吗？

技巧：在足以刻画文档的前提下，如何选取更少的序列？

❖ 单向函数：f()可快速计算，但f<sup>-1</sup>()的计算却十分...十分地耗时

❖ 例如，整数乘法： f(x, y) = x \* y //O(1)时间——RAM模型

$$f^{-1}(x * y) = \langle x, y \rangle //O(?) —— \text{质因数分解}$$

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## MD5算法

Message-Digest Algorithm version.5, 1991

MIT Lab for Computer Science & RSA Data Security Inc.

将信息统一视作比特流，每 $512 = 16 \times 32$  bit作为一个处理分组  
经过四轮位运算变换

最终得到一个128bit的大整数，即MD5指纹

## 初始化

如有必要，追加一个1及多个0，使总长形如 $512k - 64$   
将原信息长度附加到末尾，使总长形如 $512k$

128bit指纹，足够复杂？ $/2^{128} = 3.4 \times 10^{38}$

要重构（破译）符合指定指纹的原始信息，似乎不那么简单...

Data Structures & Algorithms (Fall 2012), Tsinghua University

5

## 10. 优先级队列

## (a) 基本实现

邓俊辉

deng@tsinghua.edu.cn

## 应用、算法与特点

## 应用 离散事件模拟

操作系统：任务调度、中断处理、GUI的MRU、...

输入法：词频调整

## 作为底层数据结构所支持的高效操作，是很多高效算法的基础

内部、外部、在线排序

贪心算法：Huffman编码、Kruskal算法

平面扫描算法中的事件队列

...

## 全序？偏序！

元素之间或者不能直接比较大小，或者不能低成本地进行比较

Data Structures & Algorithms (Fall 2012), Tsinghua University

8

## 列表实现

同样地，通常insert()操作多于delMax()操作

故相对而言，采用无序列表反而更有效

能否兼顾两种列表的优点？

	无序列表	有序列表
insert()	$\Theta(1)$ 直接作为首元素插入	$\Theta(n)$ 顺序搜索确定插入位置 $\Theta(n)$ 插入新元素 $\Theta(1)$
delMax()	$\Theta(n)$ 遍历以确定最大元 $\Theta(n)$ 摘除最大元 $\Theta(1)$	$\Theta(1)$ 直接摘除首元素

Data Structures & Algorithms (Fall 2012), Tsinghua University

9

## 课后

了解MD5算法的更多细节 //google("MD5 algorithm")

学习MD5相关软件的使用 //google("MD5 tool")

验证：某个文件的MD5，可能与另一个文件的MD5雷同

思考：如果有人掌握了MD5冲突的原理，则意味着什么？

了解MD5安全性的最新结论 //google("MD5 'WANG Xiaoyun'")

提交作业包时，尝试同时提交MD5指纹

了解分布式散列表的

原理、方法与 //Overlay Network, Distributed Hash Table

实现 //Napster, Gnutella; Chord, CAN, Tapestry, Pastry, ...

Data Structures & Algorithms (Fall 2012), Tsinghua University

6

## 优先级队列

还记得Huffman编码吗？

(超)字符 ~ 树 ~ 森林

取出权值最小的两棵树，合二为一并重新插入森林

重复上述过程，直到只剩下一棵树

基本操作 (<http://thudsa.3322.org/~deng/ds/demo/pqueue/>)

getMax() 返回优先级最高的元素

delMax() 删除优先级最高的元素

insert(x) 插入元素x

支持以上接口的数据结构，即所谓优先级队列 (priority queue)

栈和队列，都是优先级队列的特例——优先级取决于元素插入的次序

更一般情况下，优先级如何确定？

Data Structures & Algorithms (Fall 2012), Tsinghua University

7

## 应用、算法与特点

## 应用 离散事件模拟

操作系统：任务调度、中断处理、GUI的MRU、...

输入法：词频调整

## 作为底层数据结构所支持的高效操作，是很多高效算法的基础

内部、外部、在线排序

贪心算法：Huffman编码、Kruskal算法

平面扫描算法中的事件队列

...

## 全序？偏序！

元素之间或者不能直接比较大小，或者不能低成本地进行比较

Data Structures & Algorithms (Fall 2012), Tsinghua University

8

## 向量实现

通常，insert()操作多于delMax()操作

故相对而言，采用无序向量反而更为有效 //适用于哪些场合？

	无序向量	(非降)有序向量
insert()	$\Theta(1)$ 直接作为末元素插入	$\Theta(n)$ 二分搜索确定插入位置 $\Theta(\log n)$ 最坏时需移动所有元素 $\Theta(n)$
delMax()	$\Theta(n)$ 遍历以确定最大元素 $\Theta(n)$ 最坏时需移动所有元素 $\Theta(n)$	$\Theta(1)$ 直接摘除末元素

Data Structures & Algorithms (Fall 2012), Tsinghua University

9

## 更好的实现

## 更好的实现

BST : AVL、Splay、Red-black、...

insert()和delMax()都仅需 $\Theta(\log n)$ 时间

但是，BST的功能远远超出了优先级队列的要求

比如，这里并不要求提供search()接口

另外，这里也不要求维护所有元素之间的全序关系

因此，或许有结构更为简单、维护成本更低的方法

使得两个基本接口的

渐进时间复杂度依然为 $\Theta(\log n)$ ，而且

实际效率更高

就最坏情况而言，这类结构已属最优——为什么？

Data Structures & Algorithms (Fall 2012), Tsinghua University

5

```

❖ template <typename T>
class PQ { //优先级队列
public:
 virtual int size() = 0; //查询当前规模
 bool empty() { return !size(); } //判断队列是否为空
 virtual void insert(T) = 0; //按照优先级次序插入词条
 virtual T getMax() = 0; //取出优先级最高的词条
 virtual T delMax() = 0; //删除优先级最高的词条
};

}

```

❖ 与其说PQ是数据结构，不如说是一种ADT  
其不同的实现，效率及适用场合也各不相同

Data Structures & Algorithms (Fall 2012), Tsinghua University

```

❖ template <typename H, typename T> //堆类型、词条类型
void testHeap(int n) { //统一测试过程
 T* elem = new T[n/3]; //创建由n/3个随机元素组成的数组
 for (int i = 0; i < n/3; i++) elem[i] = dice((T)3 * n);
 H* heap = new H(elem, n/3); //Robert Floyd建堆算法
 delete [] elem;
 while (heap->size() < n) //随机测试
 if (dice(100) < 70)
 heap->insert(dice((T)3*n)); //70%概率插入
 else
 if (!heap->empty()) heap->delMax(); //30%概率删除
 while (!heap->empty()) heap->delMax(); //清空
 release(heap);
}

```

Data Structures & Algorithms (Fall 2012), Tsinghua University

## 10. 优先级队列

### (b) 完全二叉堆

逊问曰：“何人将乱石作堆？如何乱  
石堆中有杀气冲起？”

邓俊辉

deng@tsinghua.edu.cn

```

❖ template <typename T>
class PQ_CmplHeap : public PQ<T> { //基于向量的完全二叉堆
private:
 Vector<T*>* heap; //存放各词条的向量
public:
 PQ_CmplHeap(T* E, int n) //批量构造
 { heap = new Vector<T>(); heapify(E, n, n); }
 int size() { return heap->size(); } //查询当前规模
 bool empty() { return heap->empty(); } //判断是否堆空
 void insert(T); //按照比较器确定的优先级次序插入词条
 T getMax() { return (*heap)[0]; } //直接取出优先级最高的词条
 T delMax(); //删除优先级最高的词条
};

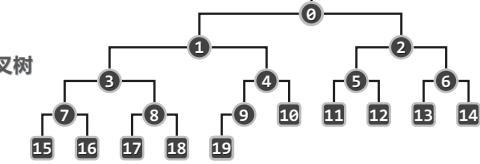
}

```

Data Structures & Algorithms (Fall 2012), Tsinghua University

### ❖ 拓扑结构

等同于完全二叉树



①

②

③

④

⑤

⑥

⑦

⑧

⑨

⑩

⑪

⑫

⑬

⑭

⑮

⑯

⑰

⑱

⑲

⑳

### ❖ 借助向量实现，无需指针

```

#define Parent(i) ((i - 1) >> 1)
#define LChild(i) (1 + ((i) << 1))
#define RChild(i) ((1 + (i)) << 1)

```

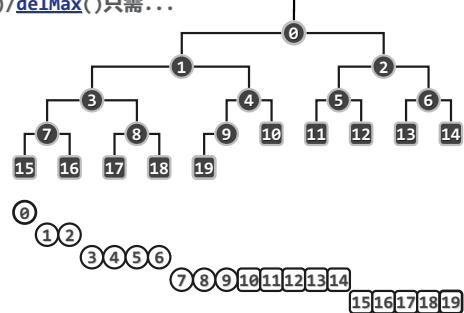
❖ 共n个节点时，内部节点的最大秩 =  $\lfloor (n - 2)/2 \rfloor = \lceil (n - 3)/2 \rceil$

Data Structures & Algorithms (Fall 2012), Tsinghua University

❖  $H[i] \geq max(H[lc(i)], H[rc(i)])$

❖ 堆顶 $H[0]$ 即是全局最大元素——于是

$getMax()$ / $delMax()$ 只需...



①

②

③

④

⑤

⑥

⑦

⑧

⑨

⑩

⑪

⑫

⑬

⑭

⑮

⑯

⑰

⑱

⑲

⑳

Data Structures & Algorithms (Fall 2012), Tsinghua University

## 插入与上滤：算法

- ❖ 为插入词条e，可以...
- ❖ 将e作为末元素接入向量
- ❖ //结构性自然保持
- ❖ //若堆序性也未破坏，则完成
- ❖ //否则 //只能是e与其父节点违反堆序性  
e与其父节点换位 //若堆序性因此恢复，则完成
- ❖ //否则 //依然只可能是e与其（新的）父节点...  
e再与父节点换位
- ❖ //不断重复...直到  
e与其父亲满足堆序性，或者  
e到达堆顶（没有父亲）



Data Structures & Algorithms (Fall 2012), Tsinghua University

## 插入与上滤：实现

```

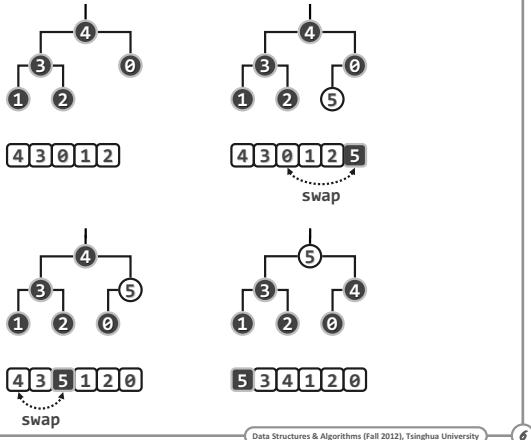
❖ template <typename T> void PQ_CmplHeap<T>::insert(T e) //插入
 { heap->insert(size(), e); percolateUp(heap, size() - 1); }

❖ template <typename T>
static Rank percolateUp(Vector<T*>* A, Rank i) { //对A[i]做上滤
 while (ParentValid(i)) {
 Rank j = Parent(i); //j为i之父
 if (lt(Node(*A, i), Node(*A, j))) //一旦当前父子不再逆序
 break; //上滤过程旋即完成
 swap(Node(*A, i), Node(*A, j)); //否则，交换父子位置
 i = j; //并上升一层
 } //while
 return i; //返回上滤最终抵达的位置
}

```

Data Structures & Algorithms (Fall 2012), Tsinghua University

## 插入与上滤：实例

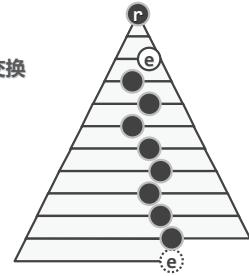


Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

6

## 插入与上滤：效率

- e与父亲的交换，每次只需 $\Theta(1)$ 时间，且每经过一次交换，e都会上升一层
- 在插入新节点e的整个过程中，只有e的祖先们，才有可能需要与之交换
- 这里的堆以完全树实现，必平衡，故e的祖先至多 $\Theta(\log n)$ 个
- 结论：通过上滤，可在 $\Theta(\log n)$ 时间内插入一个新节点，并整体重新调整为堆

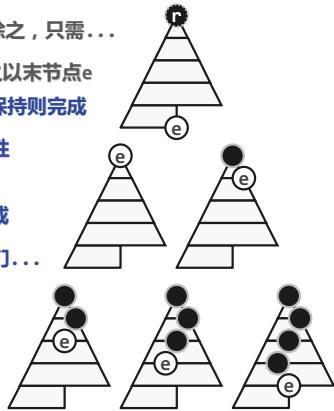


Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

7

## 删除与下滤：算法

- 最大元素始终在堆顶，故为删除之，只需...
- 摘除向量首元素 $heap[0]$ ，代之以末节点  
//结构性保持；若堆序性依然保持则完成
- 否则 //即e与孩子们违背堆序性  
e与孩子中的大者换位  
//若堆序性因此恢复，则完成
- 否则 //即e与其（新的）孩子们...  
e再次与孩子中的大者换位
- 不断重复...直到  
e满足堆序性，或者  
e已是叶子



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

8

## 删除与下滤：实现

```

template <typename T> T PQ_CmplHeap<T>::delMax() { //删除
 T maxElem = (*heap)[0]; //备份堆顶
 (*heap)[0] = heap->remove(size() - 1); //将末词条转移至堆顶
 percolateDown(heap, size(), 0); //对新堆顶实施下滤调整
 return maxElem; //返回此前备份的最大词条
}

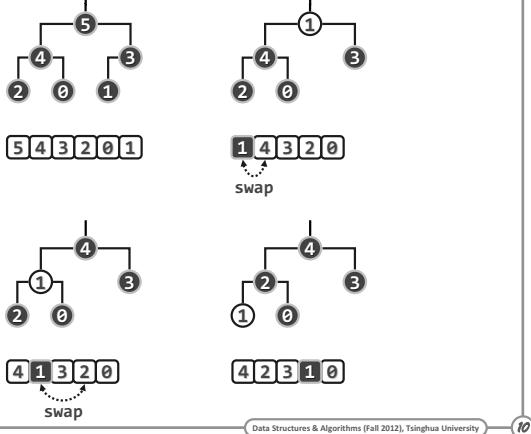
template <typename T>
static Rank percolateDown(Vector<T>* A, Rank n, Rank i) { //下滤
 Rank j; //i及其（至多两个）孩子中，堪为父者
 while (i != (j = ProperParent(*A, n, i))) //只要非j，则
 { swap(Node(*A, i), Node(*A, j)); i = j; } //换位，并继续考察i
 return i; //返回下滤抵达的位置（亦i亦j）
}

```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

9

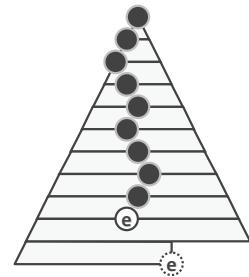
## 删除与下滤：实例



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

10

- 在下滤的过程中  
每经过一次交换  
e的高度都降低一层
- 故此，在每层至多需要一次交换
- 再次地，由于堆是完全树，故  
其高度 =  $\Theta(\log n)$
- 结论：  
通过下滤，可在 $\Theta(\log n)$ 时间内  
删除堆顶节点，并  
整体重新调整为堆



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

11

## 批量构造：自上而下的上滤：蛮力

- PQ\_CmplHeap(T\* E, int n) //将成批给出的一组词条组织为堆  
{ heap = new **Vector**<T>(heapify(E, n), n); } //如何实现？
- 蛮力算法

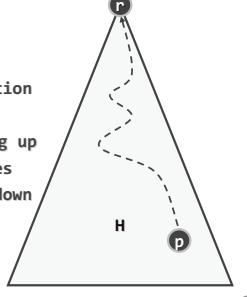
从空堆开始，依次插入各节点

## 最坏情况

每个新节点都需上滤至根  
累计耗时 $\Theta(n \log n)$ 这样长的时间  
足以全排序！

应该，能够更快的...

heapification  
= percolating up  
all nodes  
from top down



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

12

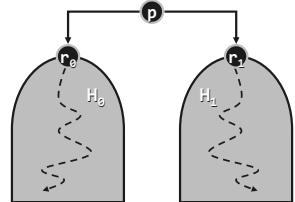
## 批量构造：自下而上的下滤：算法及实现

- 任意给定堆 $H_0$ 和 $H_1$ ，以及节点p
- 为得到堆 $H_0 \cup \{p\} \cup H_1$ ，只需  
将 $r_0$ 和 $r_1$ 当作p的孩子，对p下滤
- 【Robert Floyd, 1964】  
自下而上对各内部节点实施下滤
- template <typename T>
 

```

T* heapify(T* E, Rank lo, Rank hi) { //由区间E[lo, hi)建堆
 for (int i = LastInternal(hi - lo); i >= 0; i--) //自下而上
 percolateDown(E + lo, hi - lo, i); //依次下滤各内部节点
 return E;
} //可理解为子堆的逐层合并，由以上性质，最终堆序性必然全局恢复

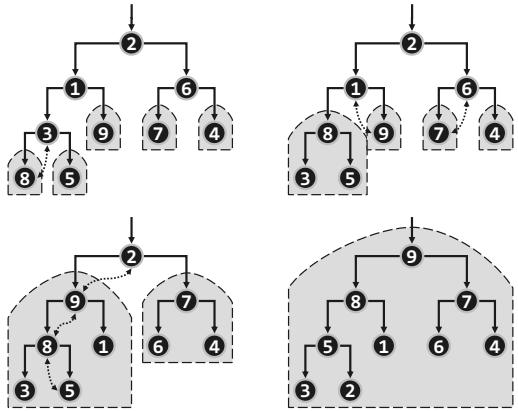
```



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

13

## 批量构造：自下而上的下滤：实例



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University 14

## 课后

- ❖ insert()：最坏情况下效率为  $\Theta(\log n)$ ，平均情况呢？
- ❖ heapify()：构造次序颠倒后，为什么复杂度会实质性地降低？  
这一算法在哪些场合不适用？
- ❖ 扩充接口：
 

```
decrease(i, delta) //任一元素H[i]的数值减小delta
increase(i, delta) //任一元素H[i]的数值增加delta
remove(i) //删除任一元素H[i]
```
- ❖ 借助完全堆，在  $\Theta(n \log n)$  时间内构造 Huffman 树
- ❖ 在大顶堆中，delMin() 操作能否也在  $\Theta(\log n)$  时间内完成？  
为此，是否需要单独实现一个...小顶...堆？

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University 16

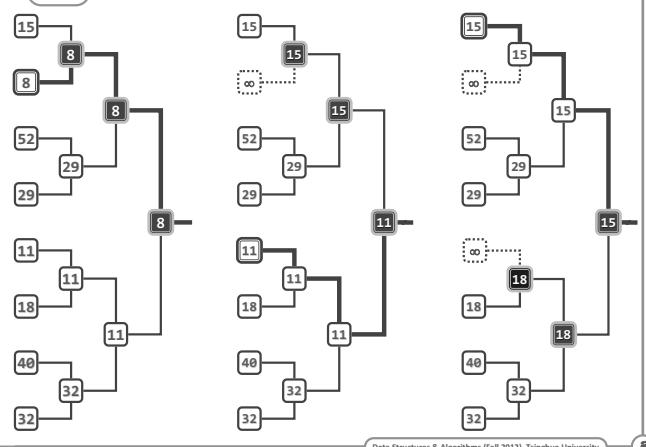
## 锦标赛树

- ❖ Tournament Tree：完全二叉树  
叶节点：待排序元素（选手）  
内部节点：孩子中的胜者  
胜负判定原则：小者为胜
- ❖ 操作：
 

```
create() //O(n)
remove() //O(log n)
insert() //实现稍难，幸亏这里不用
```
- ❖ 性质：树根处保存的总是全局优胜（最小）者——类似于最小堆  
树根到对应优胜者所在叶节点的路径存在且唯一  
该路径上所有节点的关键码都相同 (= 最小关键码)

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University 17

## 实例



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University 18

## 批量构造：自下而上的下滤：效率

- ❖ 每个内部节点所需的调整时间，正比于其高度而非深度

- ❖ 不失一般性，考查满树：

$$n = 2^{d+1} - 1$$

- ❖  $S(n)$

= 所有节点的高度总和

$$= \sum_{i=0..d} ((d-i) \times 2^i)$$

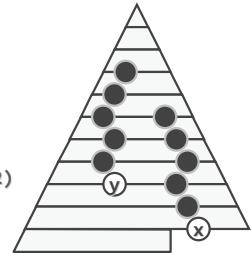
$$= d \times \sum_{i=0..d} 2^i - T(n)$$

$$= d \times (2^{d+1}-1) - ((d-1) \times 2^{d+1} + 2)$$

$$= 2^{d+1} - (d+2)$$

$$= n - \log_2(n+1)$$

$$= \Theta(n)$$



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University 19

## 10. 优先级队列

## (c) 锦标赛排序

老妖道：“怎么叫做分瓣梅花计？”  
小妖道：“如今把洞中大小群妖，点将起来，千中选百，百中选十，十中只选三个...”

邓俊辉

deng@tsinghua.edu.cn

## 算法

- ❖ Tournamentsort()

```
CREATE a tournament tree for the input list;
```

```
while there are active leaves
```

```
REMOVE the root; //current global minimum
```

```
RETRACE the root down to its leaf;
```

```
DEACTIVATE the leaf; //not necessary for sorting
```

```
REPLAY; //along the path back to the root
```

- ❖ <http://thudsa.3322.org/~deng/ds/demo/tournamentsort/>

## 实现与效率

- ❖ 空间： $\Theta(\text{节点数}) = \Theta(\text{叶节点数}) = \Theta(n)$

- ❖ 初始化：锦标赛树可在  $\Theta(n)$  时间内构建  
具体方法

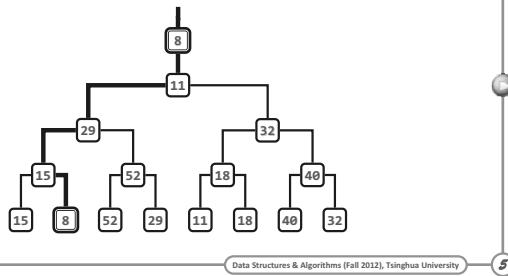
- ❖ 更新：每次都须全体重赛 (replay) ?  
上一优胜者的祖先处才有必要！

- ❖ 为此：只需从其所在叶节点出发，逐层上溯直到树根  
如此：为确定各轮优胜者，总共所需时间仅  $\Theta(\log n)$  //不再是  $\Theta(n)$

- ❖ 排序： $n \text{ 轮} \times \Theta(\log n) = \Theta(n \log n)$   
与原始 selectionsort() 的  $\Theta(n^2)$  有天壤之别，达到下界！

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University 20

- 利用锦标赛树，可否实现稳定的排序？
- 利用锦标赛树，如何实现多路归并（ $n$ 个长度为 $m$ 的有序序列的归并）？
- 锦标赛排序的效率可进一步提高——败者树（loser tree）
- 逐层重赛过程中，沿途不必访问兄弟节点



## 10. 优先级队列

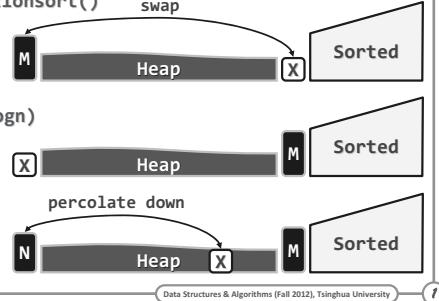
## (d) 堆排序

邓俊辉

deng@tsinghua.edu.cn

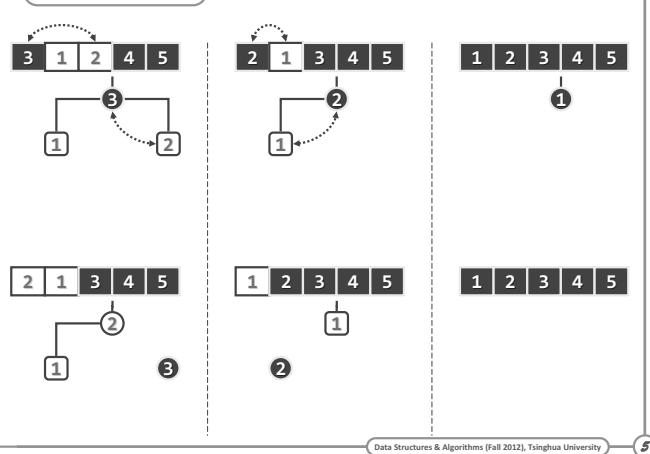
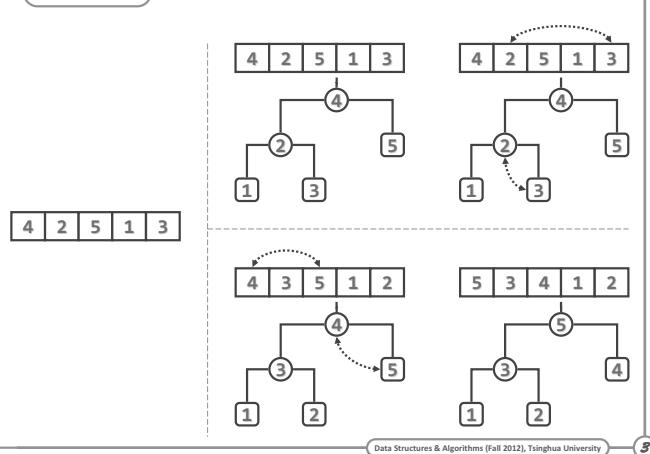
- J. Williams, 1964  
初始化：建堆，heapify()， $\Theta(n)$   
迭代：取出堆顶并调整复原，remove()， $\Theta(\log n)$

◆ 等效于常规selectionsort()



```
template <typename T> //对向量区间[lo, hi)做就地堆排序
void Vector<T>::heapSort(Rank lo, Rank hi) {
 heapify(_elem, lo, hi); //建堆， $O(hi-lo+1)$ 时间
 while (hi - lo > 1) { //迭代至(或初始即)不足两个元素
 hi--; //堆区间收缩一个单元，有序部分相应扩大
 swap(_elem[lo], _elem[hi]); //堆顶与末元素对换
 percolateDown(_elem, hi - lo, lo); //新“堆顶”下滤
 }
 percolate down
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University



## ◆ Pros

易于理解，便于实现 //完全基于二叉堆结构及其操作接口  
快速高效 //尤其适用于大规模数据  
可实现就地运转 //in-space,  $O(1)$ 附加空间  
不需全排序即可找出前 $k$ 个词条 // $O(k\log n)$ 的selection算法

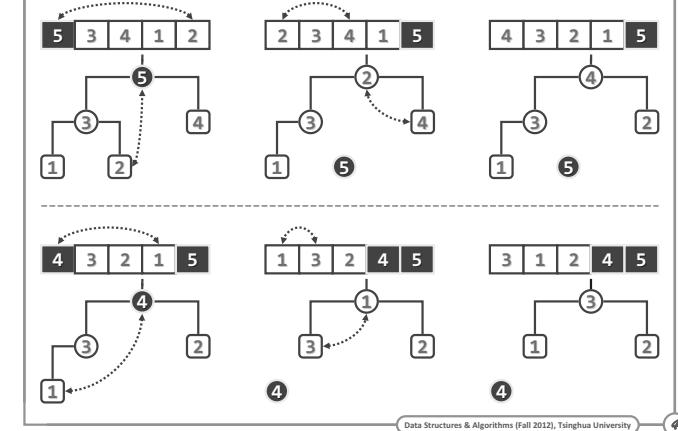
## ◆ Cons

不稳定 //为什么？可否克服？

## ◆ 权衡：采用就地策略，是否值得？

固然可以节省一定的空间

但对换操作因此须涉及两个完整的词条，操作的单位成本增加



## ◆ Pros

易于理解，便于实现 //完全基于二叉堆结构及其操作接口  
快速高效 //尤其适用于大规模数据  
可实现就地运转 //in-space,  $O(1)$ 附加空间  
不需全排序即可找出前 $k$ 个词条 // $O(k\log n)$ 的selection算法

## ◆ Cons

不稳定 //为什么？可否克服？

## ◆ 权衡：采用就地策略，是否值得？

固然可以节省一定的空间

但对换操作因此须涉及两个完整的词条，操作的单位成本增加

## 10. 优先级队列

### (x1) 左式堆

左之左之，君子宜之  
右之右之，君子有之

邓俊辉

deng@tsinghua.edu.cn

### 单侧倾斜

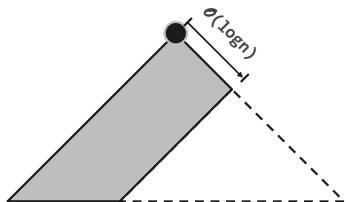
◆ C. A. Crane, 1972

◆ 思路：在保持堆序性的前提下，附加新条件，使得  
在堆合并过程中，只需调整很少部分 ( $O(\log n)$ ) 的节点

◆ 新条件：单侧倾斜

节点分布偏向于左侧  
合并操作只涉及右侧

◆ 具体实现方法...



### 763

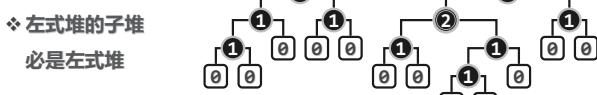
### 左倾性 & 左式堆

◆ 左倾：对任何内节点  $x$ ，都有  $npl(lc(x)) \geq npl(rc(x))$   
推论：对任何内节点  $x$ ，都有  $npl(x) = 1 + npl(rc(x))$

◆ 满足左倾性 (leftist property) 的堆，称作左式堆

◆ 左倾性与堆序性

相容而不矛盾



◆ 左式堆的子堆必是左式堆  
更多节点分布于左侧分支

◆ 这是否意味着，左子树的规模和高度必然大于右子树？

### 765

### LeftHeap

```
template <typename T>
class PO_LeftHeap : public PO<T> { //以左式堆形式实现的优先级队列
private:
 BinTree<T>* heap; //各元素在内部组织为二叉树
public:
 int size() { return heap->size(); } //查询当前规模
 bool empty() { return heap->empty(); } //判断是否堆空
 void insert(T); //按比较器确定的优先级次序插入元素
 T getMax(); //取出优先级最高的元素
 T delMax(); //删除优先级最高的元素
}; //主要接口均基于统一的合并操作实现...
```

### 767

### 堆合并

◆  $H = \text{merge}(A, B)$ ：将堆  $A$  和  $B$  合二为一 //不妨设  $|A|=n \geq m=|B|$

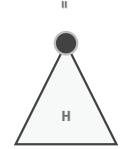
◆ 方法一：逐一取出  $B$  中节点插入  $A$  中

$m$  次  $A.insert(B.delMax())$   
共  $O(m * (\log m + \log(n+m)))$   
 $= O(m * \log(n+m))$



◆ 方法二： $O(m+n)$

合并  $A$  和  $B$  的节点  
调用 Robert Floyd 算法



◆ 有没有更好的办法？比如...

◆ 可否奢望在...

$O(\log n)$ ...时间内实现  $\text{merge}()$  ?

Data Structures & Algorithms (Fall 2012), Tsinghua University

### 764

### 空节点路径长度

◆ 引入所有外部节点

消除一度节点  
转为真二叉树

◆ Null Path Length

0) 外部节点

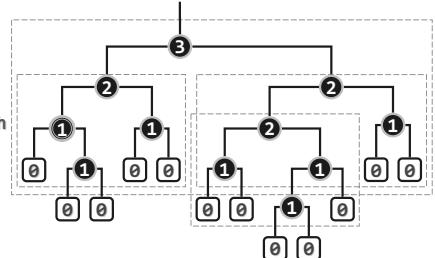
$npl(\text{NULL}) = 0$

1) 内部节点  $x$

$npl(x) = 1 + \min(npl(lc(x)), npl(rc(x)))$

◆ 验证： $npl(x) = x$  到外部节点的最近距离

$npl(x) = \text{以 } x \text{ 为根的最大满子树的高度}$



Data Structures & Algorithms (Fall 2012), Tsinghua University

### 766

### 右侧链

◆ 右侧链  $rChain(x)$ ：从节点  $x$  出发，一直沿右分支前进

◆ 特别地对于堆顶  $r$

$rChain(r)$  的终点必为全堆中最“浅”的外部节点

$npl(r) = |rChain(r)| = d$

存在一棵以  $r$  为根、高度为  $d$  的满子树

◆ 【定理】

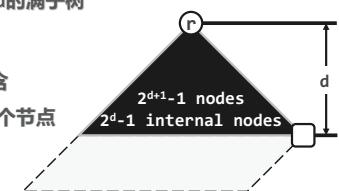
右侧链长  $d$  的左式堆，至少包含

$2^d - 1$  个内部节点、 $2^{d+1} - 1$  个节点

◆ 【推论】

在包含  $n$  个节点的左式堆中

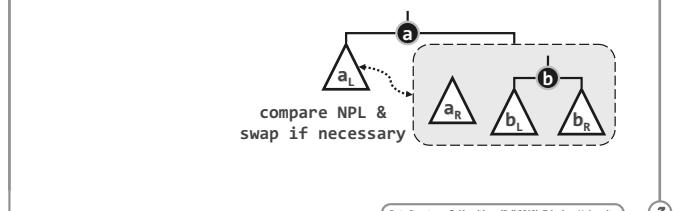
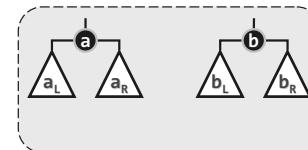
右侧链的长度  $\leq \lfloor \log_2(n+1) \rfloor - 1 = O(\log n)$



Data Structures & Algorithms (Fall 2012), Tsinghua University

### 768

### 合并：算法



Data Structures & Algorithms (Fall 2012), Tsinghua University

```

◆ template <typename T>
static BinNodePosi(T) merge(BinNodePosi(T) a, BinNodePosi(T) b) {
 if (!a) return b; if (!b) return a; //退化情况
 return lt(a->data, b->data) ? merge1(b, a) : merge1(a, b); //一般情况
} //根据待合并子堆的相对优先级，确定合并次序

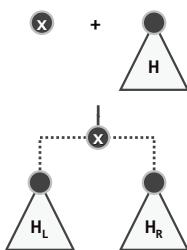
◆ static BinNodePosi(T) merge1(BinNodePosi(T) a, BinNodePosi(T) b) {
 if (!HasLChild(*a)) { a->lChild = b; b->parent = a; } //a无左孩子时
 else { //否则
 a->rChild = merge(a->rChild, b); //将b与a的右子堆合并
 a->rChild->parent = a; //并更新父子关系
 if (a->lChild->npl < a->rChild->npl) //必要时
 exchangeChildren(a); //交换a的左、右子堆，确保右子堆的npl更小
 a->npl = a->rChild->npl + 1; //更新a的npl
 }
 return a; //返回合并后的堆顶
}

```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

◆ merge()可视作一项基本操作  
其余主要接口均可基于merge()实现

1. insert(H, x)  
插入x后  
 $H' = \text{merge}(\text{heapify}(x), H)$
2. delMax(H)  
设 $x = \text{delMax}(H)$ ，则删除x后  
 $H' = \text{merge}(H.\text{lChild}(), H.\text{rChild}())$



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 10. 优先级队列

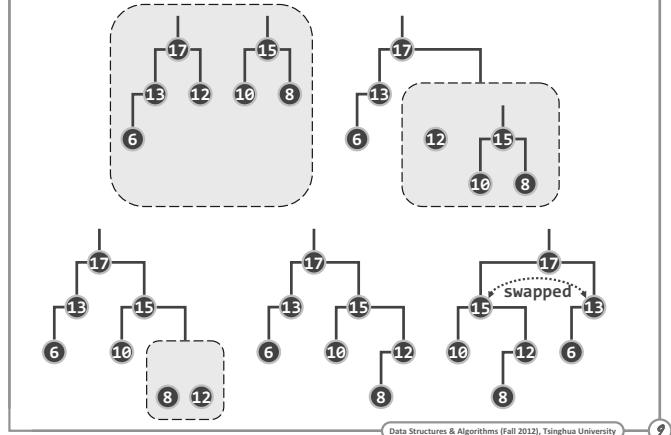
## (x2) 多叉堆

邓俊辉

deng@tsinghua.edu.cn

- ◆ 自然地，PFS中的各顶点可组织为**优先级队列**形式
- ◆ 为此需要使用PQ接口
  - heapify(): 由n个顶点创建初始PQ 总计 $\Theta(n)$
  - delMax(): 取优先级最高（极短）跨边(u,w) 总计 $\Theta(n \cdot \log n)$
  - decrease(): 更新w的所有关联顶点到u的距离 总计 $\Theta(e \cdot \log n)$
- ◆ 总体运行时间 =  $\Theta(n \cdot \log n)$ 
  - 对于稀疏图，处理效率很高
  - 对于稠密图，反而不如常规实现的版本
- ◆ 有无更好的算法？如果PQ的接口效率能够更高的话...
- ◆ 不太现实？异想天开？不妨先试试...

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

```

◆ template <typename T> void PQ_Heapify<T>::insert(T e) { //插入
 BinNodePosi(T) v = new BinNode<T>(e); //为e创建一个二叉树节点
 heap->root() = merge(heap->root(), v); //通过合并完成新节点的插入
 heap->root()->parent = NULL; //既然此时堆非空，还需相应设置父子链接
 heap->size++; //更新规模
}

◆ template <typename T> T PQ_Heapify<T>::delMax() { //删除
 BinNodePosi(T) lHeap = heap->root()->lChild; //左子堆
 BinNodePosi(T) rHeap = heap->root()->rChild; //右子堆
 T e = heap->root()->data; //备份堆顶处的最大元素
 delete heap->root(); heap->size--; //删除根节点
 heap->root() = merge(lHeap, rHeap); //原左右子堆合并
 if (heap->root()) heap->root()->parent = NULL; //更新父子链接
 return e; //返回原根节点的数据项
}

```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

◆ 回顾图的优先级搜索以及**统一框架**：g->pfs()...

◆ 无论何种算法，差异仅在于

所采用的优先级更新器 prioUpdater()

Prim算法：g->pfs(0, PrimPU());

Dijkstra算法：g->pfs(0, DijkstraPU());

◆ 每一节点引入遍历树后，都需要

**更新**树外顶点的优先级（数），并

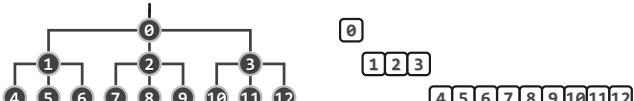
**选出**新的优先级最高者

◆ 若采用邻接表，两类操作的累计时间分别为 $\Theta(n+e)$ 和 $\Theta(n^2)$

◆ 能否更快呢？

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

- ◆ heapify():  $\Theta(n)$  不可能再快了 //直接写入亦不过如此
- ◆ delMax():  $\Theta(\log n)$  实质就是percolateDown() //已是极限了
- ◆ decrease():  $\Theta(\log n)$  实质就是percolateUp() //似乎仍有余地



◆ 若将**二叉堆**改成**多叉堆** (d-heap)，则堆高降至 $\Theta(\log_d n)$

**上滤**时间可降至 $\Theta(d \cdot \log_d n)$ ，但

**下滤**时间却增至 $\Theta(d^2 \cdot \log_d n) > (d \cdot \ln 2 / \ln d) \cdot \log_2 n$

◆ 对于稠密图，因两类操作的次数相差悬殊，故而**利大于弊**...

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 多叉堆

❖ 如此，PFS的运行时间将是

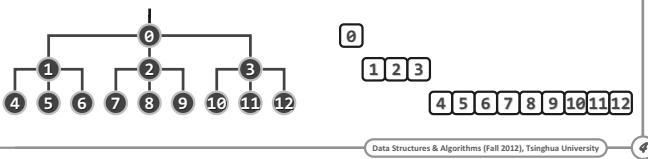
$$n * d * \log_d n + e * \log_d n = (n * d + e) * \log_d n$$

❖ 两相权衡，大致取  $d = e/n + 2$  时

总体性能达到最优的  $\mathcal{O}(e * \log_{e/n+2} n)$

❖ 对于稀疏图，接近于  $\mathcal{O}(n \log n)$  //保持高效

对于稠密图，接近于  $\mathcal{O}(e)$  //改进极大



## Fibonacci堆

❖ 左式堆  $\times$  (新的上滤算法 + 懒惰合并)

❖ 各接口的分摊复杂度

```
delMax() O(log n)
insert() O(1)
merge() O(1)
decrease() O(1)
```

❖ 于是，基于PFS框架的算法采用Fibonacci堆后，运行时间自然就是

$$n * O(\log n) + e * O(1) = O(e + n \log n)$$

## 11.串

(a) ADT

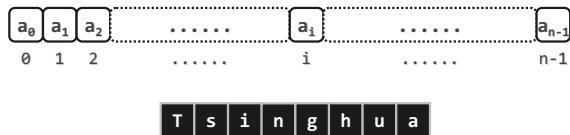
邓俊辉

deng@tsinghua.edu.cn

## 定义

❖ 字母表： $\Sigma$ ——通常就是ASCII字符集

❖ 字符串：由来自 $\Sigma$ 的字符所组成的有限序列  $S = "a_0 a_1 a_2 \dots a_{n-1}"$



❖ 既然如此，为何不直接用序列结构实现串？

❖ 通常，字符的种类不多（127 ~ 255），而串长 =  $n \gg |\Sigma|$

## ADT

`length()` 查询串的长度

`charAt(i)` 返回第*i*个字符

`substr(i, k)` 返回从第*i*个字符起、长度为*k*的子串

`prefix(k)` 返回长度为*k*的前缀

`suffix(k)` 返回长度为*k*的后缀

`equals(T)` 判断 *T* 是否与当前字符串相等

`concat(T)` 将 *T* 串接在当前字符串之后

`indexOf(P)` 若 *P* 是子串，则返回该子串的起始位置；否则返回 -1

## 术语

❖  $S = "a_0 a_1 a_2 \dots a_{n-1}"$ ,  $T = "b_0 b_1 b_2 \dots b_{m-1}"$

❖ 相等： $S = T$  iff  $n = m$  &  $a_i = b_i$  for all  $0 \leq i < n$   
亦即，长度相等，且对应的字符分别相同

❖ 子串： $\text{substr}(S, i, k) = "a_i a_{i+1} \dots a_{i+k-1}"$ ,  $0 \leq i < n$ ,  $0 \leq k$   
亦即，从  $S[i]$  起的连续  $k$  个字符

❖ 前缀： $\text{prefix}(S, k) = \text{substr}(S, 0, k)$ ,  $0 \leq k \leq n$   
亦即，*S* 的前  $k$  个字符

❖ 后缀： $\text{suffix}(S, k) = \text{substr}(S, n-k, k)$ ,  $0 \leq k \leq n$   
亦即，*S* 的最后  $k$  个字符

❖ 空串： $n = 0$ ，亦即 ""，也是任何串的子串、前缀、后缀

## 实例

```
"data structures".length() = 15
"data structures".charAt(5) = 's'
"data structures".prefix(4) = "data"
"data structures".suffix(10) = "structures"
"data structures".concat("and algorithms")
= "data structures and algorithms"
"algorithms".equals("data structures") = false
"data structures and algorithms".indexOf("string") = -1
"data structures and algorithms".indexOf("algorithm") = 20
```

❖ <string.h> 中的对应功能：

`strlen()`、`strstr()`、`strcpy()`、`strcat()`、`strcmp()`、

❖ 以下，直接利用字符数组实现字符串，转而重点讨论串匹配算法

## 11. 串

## (b) 串匹配

郑俊辉

deng@tsinghua.edu.cn

## 串匹配

◆ 歧义： $T = "1001\textcolor{red}{101101011011}1001"$  $P = "\textcolor{red}{1011}"$ 

◆ 应用：文本编辑器、数据库检索、C++模板匹配、模式识别、搜索引擎、...

◆ 应用：生物序列分析 (biological sequence analysis)

通常不能完全匹配

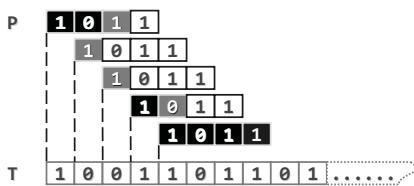
——alignment：最接近的匹配在什么位置？

HBA\_HUMAN vs. HBB\_HUMAN

**GSAQVKHGKKVADALTNAVAHVDDMPNALSALSDLHAHKL**  
**GNPKVKAHGKKVLGAFSDGLAHLDNLKGTFATLSELHCDKL**

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 蛮力：构思

◆ <http://thudsa.3322.org/~deng/ds/demo/pattern/>◆ 自左向右，以字符为单位，依次移动模式串  
直到在某个位置，发现匹配

◆ 如何：确定串T和P每次做比对的字符位置？并在发现某对字符失配后，调整位置以继续比对？

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 蛮力：版本1

```
◆ int match(char* P, char* T) {
 size_t n = strlen(T), i = 0; //主串长度、当前字符位置
 size_t m = strlen(P), j = 0; //模式串长度、当前字符位置

 while (j < m && i < n) //自左向右逐个比对字符
 if (T[i] == P[j]) //若匹配
 { i++; j++; } //则转到下一对字符
 else //否则，主串回退、模式串复位
 { i -= j - 1; j = 0; }

 return i - j;
} //如何通过返回值，判断匹配结果？
```

## 串匹配

```
% grep <pattern> <text>
主串 T = "now is the time for all good people to come"
模式 P = "people"
```

◆ 习惯及边界条件：记  $n = |T|$ ，记  $m = |P|$ 通常， $n \gg m \gg c > 0$ 比如： $100,000 \gg 100 \gg 5 > 0$ 

◆ Pattern matching

detection：P是否出现？

location：首次在哪里出现？ //本章主要讨论的问题

counting：共有几次出现？ //find /c "2011" students.txt

enumeration：各出现在哪里？ //find "2011" students.txt

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 评测标准与策略

◆ 如何客观地测量与评估任一串匹配算法的性能？

◆ 随机T + 随机P？不妥！

以  $\Sigma = \{0, 1\}^*$  为例|{长度为m的P}| =  $2^m$ |{长度为m且在T中出现的P}| =  $n - m + 1 < n$ 匹配成功的概率 =  $n/2^m \ll 100,000/2^{100} < 10^{-25}$ 

如此，将无法对算法做充分测试

◆ 随机T，对成功、失败的匹配分别测试

成功：在T中随机取出长度为m的子串作为P；分析平均复杂度

失败：采用随机的P；统计平均复杂度

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 蛮力：实现

◆ 实现1：i和j分别指向T[]和P[]中待比对的字符

```
if (T[i] == P[j]) { i++; j++; } //若匹配，则i和j同时右移
else { i -= j-1; j = 0; } //若失配，则T回退、P复位
```

◆ 实现2：i+j和j分别指向T[]和P[]中待比对的字符

```
if (T[i+j] == P[j]) { j++; } //若匹配，则i+j和j同时右移
else { i++; j = 0; } //若失配，则i右移，j回溯
```

◆ 这两种实现方法，各有什么优缺点？

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 蛮力：版本2

```
◆ int match(char* P, char* T) {
 size_t n = strlen(T), i = 0; //T长度、与P首字符的对齐位置
 size_t m = strlen(P), j; //P长度、当前接收比对字符的位置

 for (i = 0; i < n - m + 1; i++) { //T从第i个字符起，与P首字符对齐
 for (j = 0; j < m; j++) //P中对应的字符逐个比对
 if (T[i + j] != P[j]) //若失配，则
 break; //整体右移一个字符，再做一轮比对
 if (j == m) break; //找到匹配子串
 }
 return i;
} //如何通过返回值，判断匹配结果？
```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 蛮力：复杂度

❖ 最好情况：只经过一轮比对，即可确定匹配

#比对 = m =  $\Theta(m)$

❖ 最坏情况：每轮都比对至P的末字符，且反复如此

每轮循环：#比对 = m-1(成功) + 1(失败) = m

循环次数 = n - m + 1

一般有  $m \ll n$

故总体地，#比对 =  $m \times (n-m+1) = \Theta(nm)$

❖ 最坏情况，真会出现？ 0 0 0 0 0 0 0 0 0 0 ... 0 0 1

是的！ 0 0 0 0 1

❖  $|\Sigma|$  越小，最坏情况出现的概率越高

$m$  越大，最坏情况的后果更严重

Data Structures & Algorithms (Fall 2012), Tsinghua University

## 11.串

## (c) KMP算法

吴用再使时迁扮作伏路小军，去曾头市寨中，探听他不出何意，所有陷坑，暗暗地记着，离寨多少路远，总有些处。时迁去了一日，都知备细，暗地使了记号，回报军师。

邓俊辉

deng@tsinghua.edu.cn

## 改进思路

❖ 蛮力算法中，T回退、P复位之后



此前比对过的字符，将再次参与比对



如此，在最坏情况下



T/P中每个字符平均参加  $m/n$  次比对



❖ 于是，只要这类局部匹配很多，效率必将很低

❖ 其实，这类比对大多是不必要的，因为...

❖ 断言：在任何时刻， $\text{substr}(T, i - j, j) = \text{prefix}(P, j)$

亦即，我们已掌握了  $\text{substr}(T, i - j, j)$  的全部信息

——其中每个字符是什么

❖ 既然如此，只要我们记忆力足够好，则在失败后的下一轮比对中，就不必再次比对它们，而可以直...

Data Structures & Algorithms (Fall 2012), Tsinghua University

## KMP算法

```
❖ int match(char* P, char* T) {
 int* next = buildNext(P); //构造next表
 int n = (int) strlen(T), i = 0; //主串指针
 int m = (int) strlen(P), j = 0; //模式串指针

 while (j < m && i < n) //自左向右逐个比对字符串
 if (0 > j || T[i] == P[j]) //若匹配（或P已移出最左侧）
 { i++; j++; } //则转到下一字符
 else //否则，模式串右移（主串不用回退）
 j = next[j];
}

delete [] next; //释放next表
return i - j;
}
```



D. E. Knuth J. H. Morris V. R. Pratt

Data Structures & Algorithms (Fall 2012), Tsinghua University

## next[j]的含义：避免回溯

❖ 考察集合  $N(P, j) = \{0 \leq t < j \mid$

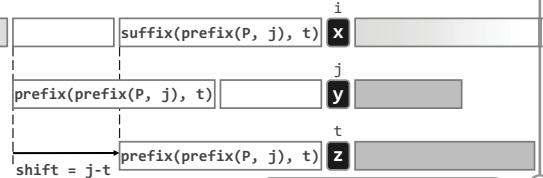
$\text{prefix}(\text{prefix}(P, j), t) = \text{suffix}(\text{prefix}(P, j), t)\}$

亦即，t为  $\text{prefix}(P, j)$  中，所有匹配真前缀和真后缀的长度

❖ P右移之后，若与T在此局部仍然匹配，则t必来自  $N(P, j)$  ——因此...

❖ 若  $T[i] \neq P[j]$ ，可取某个t，将P[t]对准T[i]，继续比对后续字符

❖ 观察：位移量 =  $j - t$ ，与t成反比 //t值越大越好



Data Structures & Algorithms (Fall 2012), Tsinghua University

## 改进思路

❖ 如此，T指针将完全不必回退！

若比对成功，则前进一个字符

否则，只需直接与P中某个字符继续比对



❖ 即便是比上例更为复杂的情况，依然可行



❖ 关键：如何确定这个字符，从而让你的记忆力发挥作用？

为此，你需要花费多少时间和空间？

更重要地，能否在事先就确定该字符？

❖ 事实上，不仅能够在事先确定，而且...

❖ 仅根据模式串P本身即可确定——与主串T的具体内容无关！

❖ 方法：根据P，构造一张查询表next[]

Data Structures & Algorithms (Fall 2012), Tsinghua University

## KMP算法：实例

chin chilla  
-1 0 0 0 1 2 3 0 0

chin chilla

chin chil \* |



chin chilla

chin chilla

Data Structures & Algorithms (Fall 2012), Tsinghua University

## next[j]的含义：不致遗漏

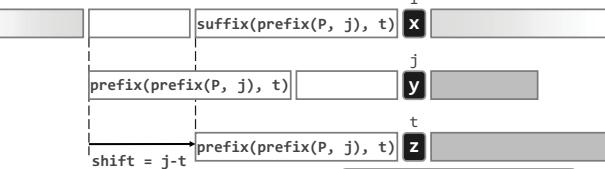
❖  $|N(P, j)| > 1$  时，难道需要遍历其中的每一个t？不必...

❖ 若取  $\text{next}[j] = \max(N(P, j))$  //真前缀、真后缀的最大匹配长度，则既可避免回溯，也不致遗漏匹配 //最短（最安全）位移量

❖ 只要  $j > 0$ ，必有  $0 \in N(P, j)$  //此处的0意味着什么？

❖ 但若  $j = 0$ ，则有  $N(P, 0) = \emptyset$  ——此时...

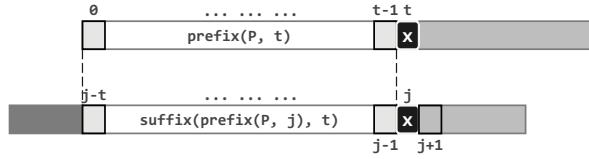
❖ 不妨取  $\text{next}[0] = -1$  //行之有效，但如何理解？慢慢体会



Data Structures & Algorithms (Fall 2012), Tsinghua University

## next[]的构造：递推

- 根据已知的 $\text{next}[0..j]$ , 如何高效地计算 $\text{next}[j + 1]$ ?
- 若:  $\text{next}[j] = t$   
则: 在 $\text{prefix}(P, j)$ 中, **自匹配的真前缀和真后缀的最大长度为** $t$
- 故:  $\text{next}[j + 1] \leq t + 1$   
——特别地, 当且仅当 $P[j] = P[t]$ 时取等号
- 一般地,  $P[j] \neq P[t]$ 时, 又该如何得到 $\text{next}[j + 1]$ ?



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University 7

## next[]的构造：算法

```

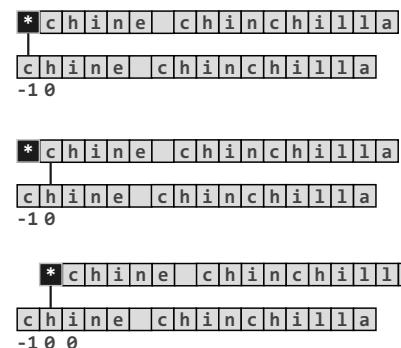
* 以上递推过程即是P的自匹配过程, 故只需对KMP框架略做修改...

* int* buildNext(char* P) { //构造模式串P的next[]表
 size_t m = strlen(P), j = 0; //“主”串指针
 int* N = new int[m]; //next[]表
 int t = N[0] = -1; //模式串指针 (next[]最左侧添加通配符)
 while (j < m - 1)
 if (0 > t || P[j] == P[t]) { //匹配
 j++; t++; N[j] = t; //N[++j] = ++t;
 } else //失配
 t = N[t];
 return N;
}

```

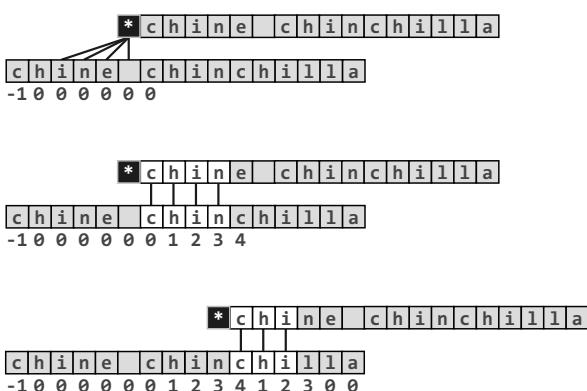
Data Structures &amp; Algorithms (Fall 2012), Tsinghua University 9

## next[]的构造：实例



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University 10

## next[]的构造：实例



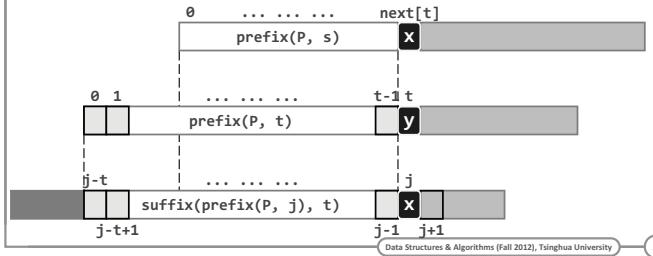
Data Structures &amp; Algorithms (Fall 2012), Tsinghua University 11

- 观察: KMP算法的确可以节省很多次比对
- 然而: 就渐进复杂度的意义而言  
这种“节省”是实质性的吗?
- 观察: 在主串的每一字符处  
模式串都可能(因失败而)回退 $\Omega(m)$ 次  
于是, 倘若这类情况出现 $\Omega(n)$ 次...
- 难道, 总体复杂度还是 $\Omega(n \times m)$ ?
- 借助更细致的分析可知, 即便是最坏情况, 也不过 $\mathcal{O}(n)$ ...

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University 12

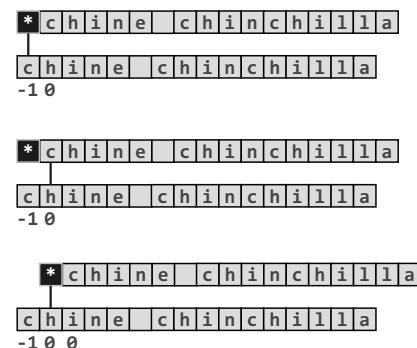
## next[]的构造：递推

- 若 $P[j] \neq P[t]$ , 则根据定义,  $\text{next}[j+1]$ 的下一候选者应该是 $\text{next}[\text{next}[j]] + 1, \text{next}[\text{next}[\text{next}[j]]] + 1, \dots$
- 因此, 只需反复考察 $t = \text{next}[t]$  //t严格递减, 为保证收敛...
- 另外, 还需统一约定 $\text{next}[0] = -1$



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University 8

## next[]的构造：实例



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University 10

## String-Search Compiler

```

* int match(char* T) { //P[] = "chinchilla"固定
 int i = -1; //主串对齐位置
 s0: if (T[i] != 'c') goto s_; i++;
 s1: if (T[i] != 'h') goto s0; i++;
 s2: if (T[i] != 'i') goto s0; i++;
 s3: if (T[i] != 'n') goto s0; i++;
 s4: if (T[i] != 'c') goto s0; i++;
 s5: if (T[i] != 'h') goto s1; i++;
 s6: if (T[i] != 'i') goto s2; i++;
 s7: if (T[i] != 'l') goto s3; i++;
 s8: if (T[i] != 'l') goto s0; i++;
 s9: if (T[i] != 'a') goto s0; i++;
 return i - 10; //模式串对齐位置呢?
}

```

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University 12

## 复杂度

- 观察: KMP算法的确可以节省很多次比对
- 然而: 就渐进复杂度的意义而言  
这种“节省”是实质性的吗?
- 观察: 在主串的每一字符处  
模式串都可能(因失败而)回退 $\Omega(m)$ 次  
于是, 倘若这类情况出现 $\Omega(n)$ 次...
- 难道, 总体复杂度还是 $\Omega(n \times m)$ ?
- 借助更细致的分析可知, 即便是最坏情况, 也不过 $\mathcal{O}(n)$ ...

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University 12

## 复杂度

- 分摊分析: 考察整个匹配过程, while循环累计执行 $\mathcal{O}(n)$ 次
- 令 $k = 2i - j$ , 观察k在算法过程中的变化  
初始时,  $k = 0$   
每经过一次循环,  $k$ 至少增加1  
若if判断为true, 则 $i, j$ 同时加1, 故 $k$ 加1  
否则,  $i$ 不变,  $j$ 至少减1, 故 $k$ 也至少加1  
算法结束时,  $k = 2i - j \leq 2(n-1) - (-1) = 2n-1$
- 同理, 建立next[]也只需 $\mathcal{O}(m)$ 时间
- 空间:  $\mathcal{O}(n+m)$

Data Structures &amp; Algorithms (Fall 2012), Tsinghua University 14

## 再改进：思路

例  $T = "000100001"$   
 $P = "00001"$   
 $\text{next[]} = \{-1, 0, 1, 2, 3\}$

$\diamond T[3]$

与  $P[3]$  比对，失败

与  $P[\text{next}[3]] = P[2]$  继续比对，失败

与  $P[\text{next}[2]] = P[1]$  继续比对，失败

与  $P[\text{next}[1]] = P[0]$  继续比对，失败

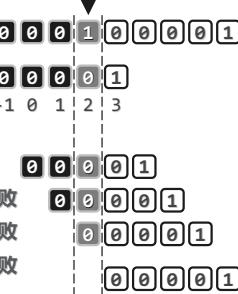
最终，才前进到  $T[4]$

$\diamond P[0] = P[1] = P[2] = P[3] = '0'$  // 无需  $T$  串即可在事先确定！

既然如此，在发现  $T[3] \neq P[3]$  之后，为什么还要...？

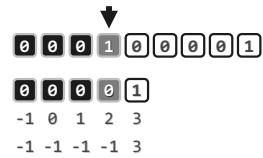
$\diamond$  事实上，后三次比对本来都是可以避免的！

Data Structures & Algorithms (Fall 2012), Tsinghua University 15



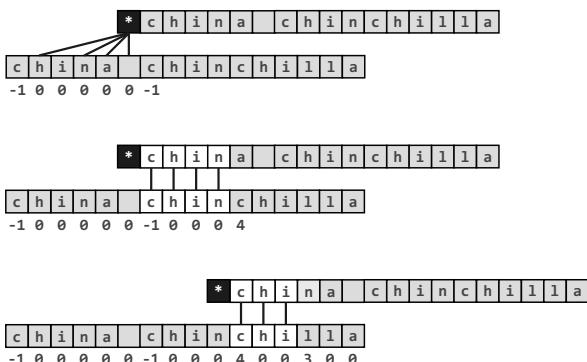
## 再改进：算法

```
int* buildNext(char* P) {
 size_t m = strlen(P), j = 0; // "主"串指针
 int* N = new int[m]; // next表
 int t = N[0] = -1; // 模式串指针
 while (j < m - 1)
 if (0 > t || P[j] == P[t]) { // 匹配
 j++; t++;
 N[j] = (P[j] != P[t]) ? t : N[t];
 } else // 失配
 t = N[t];
 return N;
}
```



Data Structures & Algorithms (Fall 2012), Tsinghua University 16

## 再改进：实例



Data Structures & Algorithms (Fall 2012), Tsinghua University 17

## 11. 串

## (d) BM 算法

邓俊辉

deng@tsinghua.edu.cn

## 构思

R. S. Boyer + J. S. Moore

A fast string searching algorithm

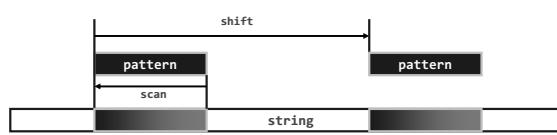
Comm. of ACM. 20:762-772, 1977

预处理：根据模式串  $P$ ，预先构造  $GS[]$  表和  $BC[]$  表

迭代：自右向左依次比对字符，找到极大的匹配后缀

若完全匹配，则返回位置

否则，根据  $GS[]$  和  $BC[]$ ，适当右移  $P$ ，并重新自右向左比对



Data Structures & Algorithms (Fall 2012), Tsinghua University 18

## 实例



Data Structures & Algorithms (Fall 2012), Tsinghua University 19

## Bad-Character Shift

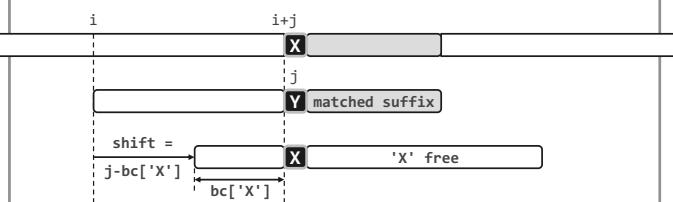
自右向左扫描比对过程中

一旦发现  $T[i+j] = 'X' \neq 'Y' = P[j]$  // 'Y' 称作坏字符

则右移  $P$  并重新扫描比对

哪些右移量值得试探？或者反过来，哪些无需试探？

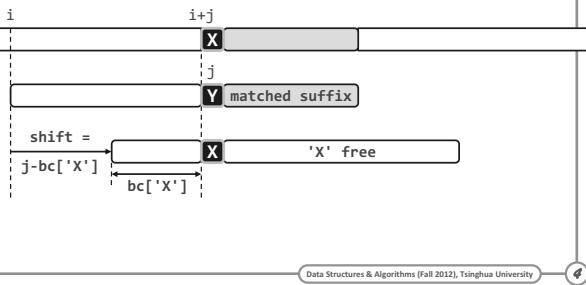
必要条件：至少坏字符本身应得以恢复匹配——因此...



Data Structures & Algorithms (Fall 2012), Tsinghua University 20

## Bad-Character Shift

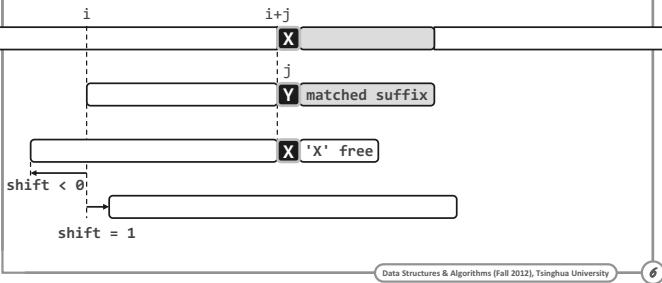
- 只需：找出P中每一字符‘X’，并分别使之与T[i+j] = ‘X’对准，然后再做一轮比对
- 注意：位移量取决于失配位置j和‘X’在P中的秩，而与T和i无关！因此， $bc['X'] = j + shift$ 可以事先计算，并制表待查



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

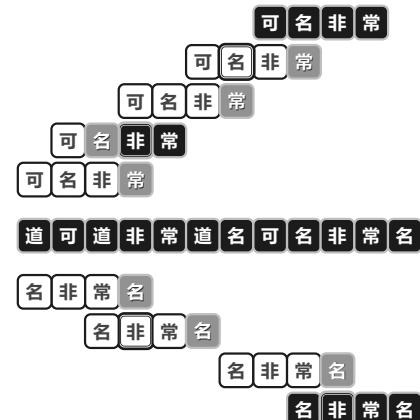
## Bad-Character Shift

- 即使存在‘X’，但最右侧者位置太靠右（以至于位移量为负）呢？
- 将P[]右移一个字符！
- 为何...不选用“在P[j]左侧并与之最近”的那个‘X’？



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## 实例



Data Structures &amp; Algorithms (Fall 2012), Tsinghua University

## BC[]表的构造：算法

- ```

int* buildBC(char* P) {
    int* bc = new int[256]; //bc[]表，与字母表等长
    for (size_t j = 0; j < 256; j++) bc[j] = -1; //初始化
    for (size_t m = strlen(P), j = 0; j < m; j++) //画家算法
        bc[P[j]] = j; //刷新P[j]的出现位置记录
    return bc;
}

与KMP同理，未出现字符的BC值取作-1，相当于在P的左侧添加通配符
自左向右的扫描，可保证各字符对应的BC值单调非降，最终取最大值
附加空间 = |BC[]| = O(|Σ|)
时间 = O(|Σ| + m)

```

Data Structures & Algorithms (Fall 2012), Tsinghua University

BC算法：查找时间

- 最好 = $\Theta(n / m)$
- 除法？没错！
- 比如： $T = "xxxx1xxxx1xxxx1...xxxx1xxxx"$
 $P = "00000"$
- 一般地，只要T的当前字符未在P中出现，即可直接移动m个字符
此类情况下，仅需单次比较，即可排除m个对齐位置
- 单次匹配概率越小的场合，相对于蛮力算法的性能优势越明显
(比如ACSI串、汉字、UNICODE串之类， $|\Sigma|$ 很大的场合)
- P越长，这类移动的效果越明显

Data Structures & Algorithms (Fall 2012), Tsinghua University

BC算法：查找时间

- 最坏 = $\Theta(n \times m)$
- 等效于蛮力算法？是的！
- 比如： $T = "00000...00"$
 $P = "10000"$
- 一般地，每轮迭代都要在扫过整个P之后，方能确定右移一个字符
此类情况下，须经m次比较，方能排除单个对齐位置
- 单次匹配概率越大的场合，性能越接近于蛮力算法
(比如位图、DNA序列之类， $|\Sigma|$ 很小的场合)

Data Structures & Algorithms (Fall 2012), Tsinghua University

Bad-Character Shift：不足

- 观察：若仅使用BC[]表，有时移动得仍不够快

$T = "xxxxCABCaabC"$
 $P = "cabcAABC"$
- $P[4] = 'A'$ 与 $T[4] = 'C'$ 失配后，P只右移1个字符

$T = "xxxxcabcaAbc"$
 $P = "cabcaabc"$
- 实际上，此前的成功比对已给出了足够的信息 //匹配的后缀"ABC"
根据这些信息，可以直接将P右移4个字符

$T = "xxxxCABCaabC"$
 $P = CABCAABC"$

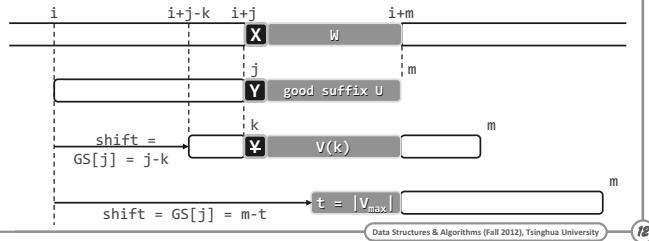
Data Structures & Algorithms (Fall 2012), Tsinghua University

Good-Suffix Shift

扫描比对的过程，尽管可能中断于 $T[i+j] = 'X' \neq 'Y' = P[j]$
但此时 $U = \text{postfix}(P, m-j)$ 必然已经匹配 //好后缀

于是，下一轮的对齐位置必须

- 1) 使 U 重新与 $V(k) = \text{substr}(P, k+1, m-j-1)$ 匹配，而且
- 2) $P[k] \neq 'Y' = P[j]$

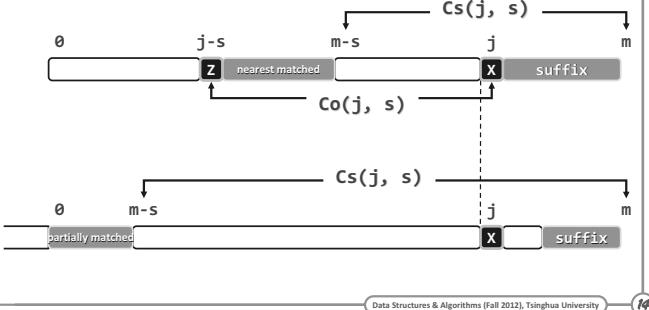


12

GS[]表的构造

条件 $CS(j, s)$ ：对于任何 $j < k < m$ ，若 $k > s$ ，则 $P[k-s] = P[k]$
条件 $CO(j, s)$ ：若 $s < j$ ，则 $P[j-s] \neq P[j]$

$GS[j] = \min\{s > 0 \mid CS(j, s) \text{ 且 } CO(j, s)\}$



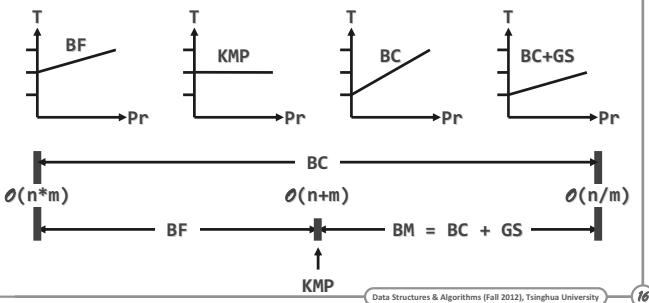
14

BC + GS：性能分析

空间 = $|BC| + |GS| = \Theta(|\Sigma| + m)$

预处理时间 = $\Theta(|\Sigma| + m)$

查找时间：最好 = $\Theta(n/m)$ ，最差 = $\Theta(n+m)$ //参照KMP算法的分析



16

凡物皆数

Gödel numbering

逻辑系统的符号、表达式、命题、公理等
均可以不同的自然数标识

素数序列：

$$p(k) = \text{第 } k \text{ 个素数} // 2, 3, 5, 7, 11, \dots$$



K. Gödel：每个有限维的自然数向量，唯一对应于一个自然数

$$\langle a_1, a_2, \dots, a_n \rangle = p(1)^{1+a_1} \times p(2)^{1+a_2} \times \dots \times p(n)^{1+a_n}$$

$$\langle 3, 1, 4, 1, 5, 9, 2, 6 \rangle = 2^4 \times 3^2 \times 5^5 \times 7^2 \times 11^6 \times 13^{10} \times 17^3 \times 19^7$$

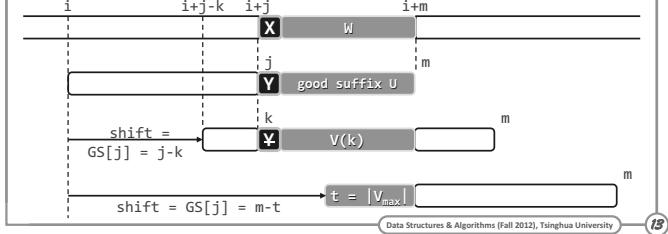
Good-Suffix Shift

若 P 中的确存在这样的子串 $V(k)$

则可选择其中最长者，并通过右移使之与 U 对齐

否则，在所有前缀 $\text{prefix}(P, t)$ 中，取与 U 的后缀匹配的最长者
注意：有可能 $t = 0$

无论如何，位移量仅取决于 j 和 P 本身——亦可预先计算，并制表待查



实例

8 8 8 8 8 4 8 1
非 日 静 也 善 故 静 也

圣 人 之 静 也 非 日 静 也 善 故 静 也

非 日 静 也 善 故 静 也

+1 → 非 日 静 也 善 故 静 也

+4 → 非 日 静 也 善 故 静 也

11.串

(e) Karp-Rabin算法

All things are numbers.

- Pythagoras (570 ~ 495 BC)

God made the integers;

all else is the work of man.

- L. Kronecker (1823 ~ 1891)

邓俊辉

deng@tsinghua.edu.cn

凡物皆数

K. Gödel：给定可数字母表，有限长度的字符串均唯一对应于自然数

$$\{a, b, c, \dots, z, \dots\} = \{1, 2, 3, \dots, 26, \dots\}$$

$$\text{"godel" } = 2^{1+7} \times 3^{1+15} \times 5^{1+4} \times 7^{1+5} \times 11^{1+12}$$

$$= 139 869 560 310 664 817 087 943 919 200 000$$

若果真如RAM模型所假设，字长无限，则只需一个寄存器即可...

反过来，由Gödel编号可否唯一确定原字符串？

Book IX of The Elements of Geometry (ca 300 B.C.)

Euclid: every number factors uniquely into primes

因此，由合法的编号，经素因子分解再排序，即可唯一确定原字符串

素因子分解，至今尚无有效算法！——不幸？幸运？

串亦为数

❖ 十进制串可直接视作自然数 //称作指纹 (fingerprint)

$$P = 82818$$

$$T = 271\textcolor{red}{82818}284590452353602874713527$$

❖ 一般地，随意对字符编号 {0, 1, 2, ..., d-1} //设d = |Σ|

于是，每个字符串都对应于一个d进制自然数 //尽管不是单射

$$\text{"CAT"} = 2 \ 0 \ 19_{(26)} = 1371_{(10)} // \Sigma = \{A, B, C, \dots, Z\}$$

$$\text{"ABBA"} = 0 \ 1 \ 1 \ 0_{(26)} = 702_{(10)}$$

❖ P在T中出现 仅当 T中某一子串与P“相等” //为什么不是“当”？

❖ 这，不已经就是一个算法吗？！ //具体如何实现？

❖ 问题似乎解决得很顺利，果真如此简单吗？ //复杂度？

Data Structures & Algorithms (Fall 2012), Tsinghua University

凡物皆数

833

❖ Cantor numbering

$$\begin{aligned} \text{cantor}_2(a_1, a_2) &= ((a_1 + a_2)^2 + 3a_1 + a_2) / 2 \\ \text{cantor}_{n+1}(a_1, \dots, a_n, a_{n+1}) &= \\ \text{cantor}_n(a_1, \dots, a_{n-1}, \text{cantor}_2(a_n, a_{n+1})) \end{aligned}$$



❖ 长度有限的字符串都可视作d进制自然数 //d = 1 + |Σ|

$$\text{"decade"} = 453145_{(10)} //d = 1 + |\{\text{'a'} \sim \text{'i'}\}| = 10$$

❖ 长度无限的字符串都可视作[0, 1]内的d进制小数

$$\text{"bgahbhahbhei..."} = 0.2718281828459\dots$$

$$\text{"fade"} = 0.614\textcolor{blue}{5} = 0.614\textcolor{red}{4999}\dots$$

❖ Cantor：集合是否可数，与其维度无关



有理数集与自然数集一样可数 //分子，分母，N₀

无理数集不可数 //Cantor's diagonal, N₁ > N₀

Data Structures & Algorithms (Fall 2012), Tsinghua University

数位溢出

835

❖ 如果|Σ|很大，模式串P较长，其对应的自然数将很大

可以将串P理解为，|P|位的|Σ|进制自然数

❖ 仍以ASCII字符集为例 //|Σ| = 128 = 2⁷

只要|P| > 9，则指纹的长度将至少是 $7 \times 10 = 70$ bits

❖ 然而，目前的字长一般也不过64位 //存储不便

❖ 而且，指纹计算及比对，将不能在 $\Theta(1)$ 时间内完成 //RAM！？

准确地说，需要 $\Theta(|P|/64) = \Theta(m)$ 时间

总体需要 $\Theta(n*m)$ 时间 //与蛮力算法相同

❖ 有何高招？

散列压缩！

Data Structures & Algorithms (Fall 2012), Tsinghua University

837

Karp-Rabin：散列冲突

❖ 注意：hash()值相等，并非匹配的充分条件... //好在必要

❖ 因此，通过hash()筛选之后，还须经严格比对方可最终确定是否匹配

❖ 如 P = 18284 //仍取M = 97，则hash(18284) = 48

$$T = 2 \ 7 \ 1 \ 8 \ 2 \ 8 \ \textcolor{red}{1 \ 8 \ 2 \ 8 \ 4} \ 5 \ 9 \ 0 \ 4 \ 5 \ 2 \ 3 \ 5 \ 3 \ 6$$

[2 2]

[4 8] //冲突可能导致误判

...

[4 8]

❖ 既然是通过散列压缩，指纹冲突就在所难免

❖ 适当选取散列函数，即可大大降低冲突可能 //如，选尽可能大的M

Data Structures & Algorithms (Fall 2012), Tsinghua University

839

12. 排序

(a) 快速排序

郑俊辉

左朱雀之芨芨兮，右苍龙之躍躍

deng@tsinghua.edu.cn

840

构思

❖ C. A. R. Hoare, 1962, 分治策略

❖ 将元素序列分为两个子序列：S = S_L + S_R // $\Theta(n)$ 时间

子序列规模缩小，而且

各自内部的排序互相独立：

$$\max(S_L) \leq \min(S_R)$$

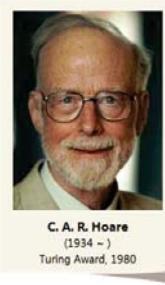
❖ 递归地对子序列分别排序

$$T(n_L) + T(n_R) \approx 2 \times T(n/2)$$

❖ 平凡解：只剩单个元素时，本身就是解

❖ 合并结果

$$\Theta(1)$$

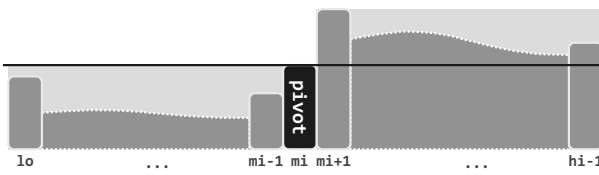


C. A. R. Hoare
(1934 ~)
Turing Award, 1980

Data Structures & Algorithms (Fall 2012), Tsinghua University

轴点

- ◆ pivot：序列中特殊的一类元素
其左/右侧的元素都不比它更大/小



- ◆ 亦即，轴点就是已经“就位”的元素——于是...
- ◆ 以轴点为界，原序列被分成左、右两个独立子序列 //递归划分
如果子序列已各自排好，那么也就完成了整体排序 //左右串接
- ◆ 实际上，一个序列完全有序，当且仅当其中所有元素皆为轴点

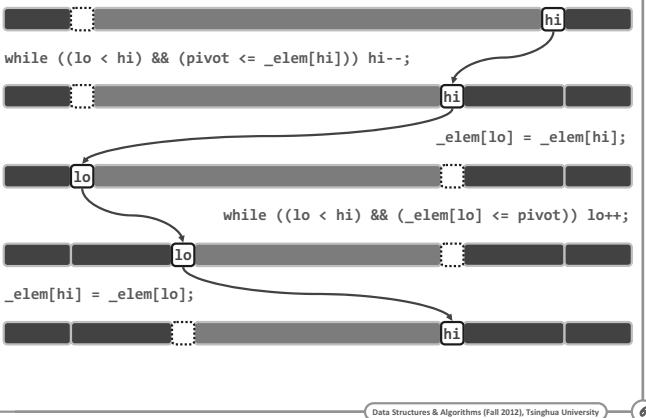
Data Structures & Algorithms (Fall 2012), Tsinghua University

构造轴点

- ◆ 任取一候选者（比如[0]）
- ◆ 两个指针：将序列分为3段
 - 左段L：≤ 轴点
 - 右段G：≥ 轴点
 - 中段U：待确定
- ◆ 初始：L = G = NULL
- ◆ 交替向内移动lo和hi，并
检查所指元素：若更小/大，则转移归入L/G
- ◆ 当lo = hi时，只需将候选者嵌入于L/G之间，它即是轴点！
- ◆ 整个过程中，各元素最多移动一次（候选者两次）——累计 $\theta(n)$ 时间

Data Structures & Algorithms (Fall 2012), Tsinghua University

算法A：过程



Data Structures & Algorithms (Fall 2012), Tsinghua University

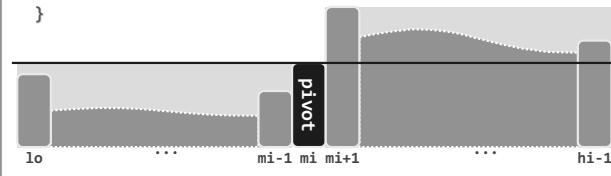
算法A：性能分析

- ◆ 不稳定：lo/hi的移动方向相反，左/右侧的大/小重复元素被逆序转移
- ◆ 就地：只需 $\theta(1)$ 附加空间——时间呢？
- ◆ 最好情况：每次划分都（接近）平均，轴点总是（接近）中央
 $T(n) = 2T((n-1)/2) + \theta(n) = \theta(n\log n)$ //到达下界！
- ◆ 最坏情况：每次划分都极不均衡 //比如，轴点总是最小/大元素
 $T(n) = T(n-1) + T(0) + \theta(n) = \theta(n^2)$
与起泡排序相当！
- ◆ 即便采用随机选取、（Unix）三者取中之类的策略
也只能降低最坏情况的概率，而无法杜绝
- ◆ 既然如此，为何还称作“快速”排序？

Data Structures & Algorithms (Fall 2012), Tsinghua University

快速排序

```
template <typename T>
void Vector<T>::quickSort(Rank lo, Rank hi) {
    if (hi - lo < 2) return; //单元素区间自然有序，否则
    Rank mi = partition(lo, hi - 1); //构造轴点
    quickSort(lo, mi); quickSort(mi + 1, hi); //前缀、后缀递归排序
}
```



◆ 坏消息：在原始序列中，轴点不见得总是存在 //试举一例

◆ 好消息：通过交换，可使任一元素成为轴点——如何交换？最多需做多少次？

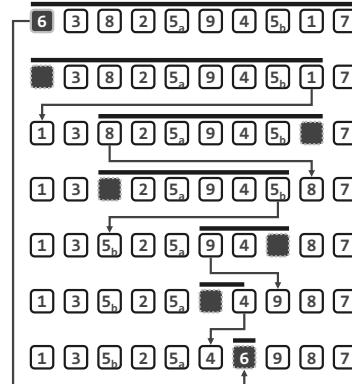
Data Structures & Algorithms (Fall 2012), Tsinghua University

算法A：实现

```
template <typename T>
Rank Vector<T>::partition(Rank lo, Rank hi) {
    swap(_elem[lo], _elem[lo + rand() % (hi-lo+1)]); //随机交换
    T pivot = _elem[lo]; //经以上交换，等效于随机选取候选轴点
    while (lo < hi) { //从两端交替地向中间扫描
        while ((lo < hi) && (pivot <= _elem[hi])) hi--; //向左拓展G
        _elem[lo] = _elem[hi]; //小于轴点者归入L
        while ((lo < hi) && (_elem[lo] <= pivot)) lo++; //向右拓展L
        _elem[hi] = _elem[lo]; //大于轴点者归入G
    } //assert: lo == hi
    _elem[lo] = pivot; return lo; //候选轴点归位；返回其秩
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

算法A：实例



Data Structures & Algorithms (Fall 2012), Tsinghua University

平均性能

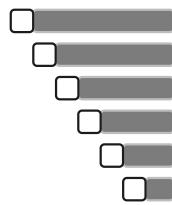
◆ 快速排序的平均性能为 $\theta(n\log n)$ ——以均匀独立分布为例...

$$\begin{aligned}
 T(n) &= (n+1) + (1/n) \times \sum_{k=0, \dots, n-1} (T(k) + T(n-1-k)) \\
 &= (n+1) + (2/n) \times \sum_{k=0, \dots, n-1} T(k) \\
 n &\times T(n) = (n+1)n + 2 \times \sum_{k=0, \dots, n-1} T(k) \\
 (n-1) \times T(n-1) &= n(n-1) + 2 \times \sum_{k=0, \dots, n-2} T(k) \\
 nT(n) - (n-1)T(n-1) &= 2n + 2T(n-1) \\
 T(n)/(n+1) &= 2/(n+1) + T(n-1)/n \\
 &= 2/(n+1) + 2/n + T(n-2)/(n-1) \\
 &= 2/(n+1) + 2/n + 2/(n-1) + T(n-3)/(n-2) \\
 &= 2/(n+1) + 2/n + 2/(n-1) + \dots + 2/2 + T(0)/1 \\
 &= 1.39 \times \log n
 \end{aligned}$$

Data Structures & Algorithms (Fall 2012), Tsinghua University

重复元素

- 大量甚至全部元素重复时
 - 轴点位置总是接近于 lo
 - 子序列的划分极不均匀
 - 二分递归退化为线性递归
 - 递归深度接近于 $O(n)$
 - 运行时间接近于 $O(n^2)$



- 可以考虑：移动 lo 和 hi 的过程中，同时比较相邻元素
 - 若属于相邻的重复元素，则不再深入递归
 - 但一般情况下，如此计算量反而增加，得不偿失
- 对算法A略做调整，即可解决问题
 - 为便于对比，先给出等价形式A1...

Data Structures & Algorithms (Fall 2012), Tsinghua University 10

(Data Structures & Algorithms (Fall 2012), Tsinghua University 11)

算法B1：实现

```
template <typename T>
Rank Vector<T>::partition(Rank lo, Rank hi) {
    swap(_elem[lo], _elem[lo + rand() % (hi-lo+1)]); //随机交换
    T pivot = _elem[lo]; //经以上交换，等效于随机选取候选轴点
    while (lo < hi) { //从两端交替地向中间扫描
        while ((lo < hi) && (pivot < _elem[hi])) hi--; //向左拓展G
        if (lo < hi) _elem[lo++] = _elem[hi]; //小于轴点者归入L
        while ((lo < hi) && (_elem[lo] < pivot)) lo++; //向右拓展L
        if (lo < hi) _elem[hi--] = _elem[lo]; //大于轴点者归入G
    } //assert: lo == hi
    _elem[lo] = pivot; return lo; //候选轴点归位；返回其秩
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University 12

(Data Structures & Algorithms (Fall 2012), Tsinghua University 13)

算法B：性能分析

- 可以正确地处理一般情况，而且复杂度没有实质的提高
- 处理重复元素时
 - lo 和 hi 会交替移动
 - 二者移动的距离相当
 - 最终，轴点被安置于 $(lo + hi)/2$ 处，实现划分均匀
- 相对于算法A的“勤于拓展、懒于交换”，转为“懒于拓展、勤于交换”
 - 因此
 - 1) 交换操作有所增多
 - 2) 更不稳定

Data Structures & Algorithms (Fall 2012), Tsinghua University 14

(Data Structures & Algorithms (Fall 2012), Tsinghua University 15)

算法C：实现

```
template <typename T>
Rank Vector<T>::partition(Rank lo, Rank hi) {
    swap(_elem[lo], _elem[lo + rand() % (hi-lo+1)]);
    T pivot = _elem[lo]; int mi = lo;
    for (int k = lo + 1; k <= hi; k++) //自左向右考察[k]
        if (_elem[k] < pivot) //若[k]小于轴点，则将其
            swap(_elem[++mi], _elem[k]); //与[mi]交换，使L向右扩展
    swap(_elem[lo], _elem[mi]); //候选轴点归位
    return mi; //返回轴点的秩
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University 16

(Data Structures & Algorithms (Fall 2012), Tsinghua University 17)

算法A1：实现

```
template <typename T>
Rank Vector<T>::partition(Rank lo, Rank hi) {
    swap(_elem[lo], _elem[lo + rand() % (hi-lo+1)]); //随机交换
    T pivot = _elem[lo]; //经以上交换，等效于随机选取候选轴点
    while (lo < hi) { //从两端交替地向中间扫描
        while ((lo < hi) && (pivot <= _elem[hi])) hi--; //向左拓展G
        if (lo < hi) _elem[lo++] = _elem[hi]; //小于轴点者归入L
        while ((lo < hi) && (_elem[lo] <= pivot)) lo++; //向右拓展L
        if (lo < hi) _elem[hi--] = _elem[lo]; //大于轴点者归入G
    } //assert: lo == hi
    _elem[lo] = pivot; return lo; //候选轴点归位；返回其秩
}
```

(Data Structures & Algorithms (Fall 2012), Tsinghua University 11)

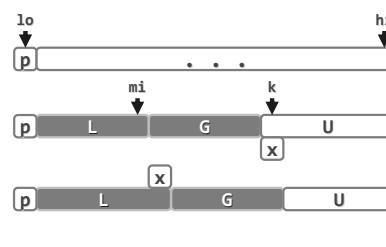
算法B：实现

```
Rank Vector<T>::partition(Rank lo, Rank hi) {
    /* ... */
    while (lo < hi) { //从两端交替地向中间扫描
        while (lo < hi)
            if (pivot < _elem[hi]) hi--; //向左拓展G
            else //直至遇到不大于轴点者
                { _elem[lo++] = _elem[hi]; break; } //将其归入L
        while (lo < hi)
            if (_elem[lo] < pivot) lo++; //向右拓展L
            else //直至遇到不小于轴点者
                { _elem[hi--] = _elem[lo]; break; } //将其归入G
    } //assert: lo == hi
    /* ... */
}
```

(Data Structures & Algorithms (Fall 2012), Tsinghua University 13)

算法C：思路

- $[lo, hi] = [lo] + L(lo, mi] + R(mi, k) + U[k, hi]$
- 考察 $[k]$ ：
 - 若小于轴点，则 $swap(mi+1, k)$ 后， L 拓展、 R 平移
 - 否则，直接拓展 R



(Data Structures & Algorithms (Fall 2012), Tsinghua University 15)

12. 排序

(b) 中位数

中也者，天下之大本也
和也者，天下之达道也

郑俊辉

deng@tsinghua.edu.cn

选取与中位数

❖ k-选取 (k-selection)

在任意一组可比较大小的元素中，如何找到由小到大次序为k者？

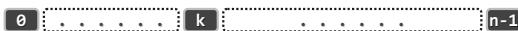
亦即，在这组元素的非降排序序列S中，找出 $S[k]$

Excel : large(range, rank)

❖ 长度为n的有序序列S中，元素 $S[\lfloor n/2 \rfloor]$ 称作中位数 (median)

在任意一组可比较大小的元素中，如何找到中位数？

Excel : median(range) //就数值而言，可能有多个重复



❖ 中位数是k-选取的一个特例；稍后将看到，也是其中难度最大者

Data Structures & Algorithms (Fall 2012), Tsinghua University

1

Data Structures & Algorithms (Fall 2012), Tsinghua University

2

主流数：减而治之

❖ 若在向量A的前缀P (|P|为偶数) 中，元素x出现的次数恰占半数，则

A有主流数仅当，对应的后缀A-P有主流数m，且m就是A的主流数



❖ 既然最终总要在O(n)时间内，核对m为主流数的充分性

故而只需考虑A的确含有主流数的两种情况

1. 若 $x = m$ ，则在排除前缀P之后，m与其它元素数量的差距保持不变

2. 若 $x \neq m$ ，则在排除前缀P之后，m与其它元素数量的差距不致缩小

❖ 因此，可采用“减而治之”策略，逐步缩减原问题的规模，直至平凡

Data Structures & Algorithms (Fall 2012), Tsinghua University

3

Data Structures & Algorithms (Fall 2012), Tsinghua University

4

归并向量的中位数

❖ 任给已经排序的有序向量 S_1 和 S_2

如何快速找出有序向量 $S = S_1 \cup S_2$ 的中位数？

❖ 蛮力算法：

归并向量 S_1 和 S_2 ，得到有序向量S

取出 $S[(|S_1| + |S_2|) / 2]$ ，即是S的中位数

❖ 如此，共需 $\Theta(|S_1| + |S_2|)$ 时间

❖ 这一效率虽不算低，但毕竟未能充分利用 S_1 和 S_2 的有序性

❖ 以下，先介绍 $|S_1| = |S_2| = n$ 情况下的算法

然后，再将该算法推广至一般情况

❖ 新的算法，依然采用“减而治之”策略...

Data Structures & Algorithms (Fall 2012), Tsinghua University

5

Data Structures & Algorithms (Fall 2012), Tsinghua University

6

归并向量的中位数：等长子向量：实现

❖ template <typename T> //尾递归，可改写为迭代形式

```
T median(Vector<T>& S1, int lo1,
          Vector<T>& S2, int lo2, int n) {
    if (n < 3) //递归基
        return trivialMedian(S1, lo1, n, S2, lo2, n);
    int mi1 = lo1 + ⌈n/2⌉, mi2 = lo2 + ⌈(n-1)/2⌉; //长度减半
    if (S1[mi1] < S2[mi2]) //取S1右半、S2左半
        return median(S1, mi1, S2, lo2, n + lo1 - mi1);
    else if (S1[mi1] > S2[mi2]) //取S1左半、S2右半
        return median(S1, lo1, S2, mi2, n + lo2 - mi2);
    else
        return S1[mi1];
}
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

7

Data Structures & Algorithms (Fall 2012), Tsinghua University

8

归并向量的中位数：任意子向量：实现

```
❖ template <typename T> T median
(Vector<T>& S1, int lo1, int n1, Vector<T>& S2, int lo2, int n2) {
    if (n1 > n2) return median(S2, lo2, n2, S1, lo1, n1); //确保n1 <= n2
    if (n2 < 6) return trivialMedian(S1, lo1, n1, S2, lo2, n2); //递归基
    if (2 * n1 < n2)
        return median(S1, lo1, n1, S2, lo2 + (n2-n1-1)/2, n1+2-(n2-n1)%2);
    int mi1 = lo1 + n1/2, mi2a = lo2 + (n1-1)/2, mi2b = lo2+n2-1 - n1/2;
    if (S1[mi1] > S2[mi2b]) //取S1左半、S2右半
        return median(S1, lo1, n1 / 2 + 1, S2, mi2a, n2 - (n1 - 1) / 2);
    else if (S1[mi1] < S2[mi2a]) //取S1右半、S2左半
        return median(S1, mi1, (n1 + 1) / 2, S2, lo2, n2 - n1 / 2);
    else //S1保留，S2左右同时缩短
        return median(S1, lo1, n1, S2, mi2a, n2 - (n1 - 1) / 2 * 2);
} //复杂度O(log(min(n1, n2))——可见，实际上等长版本才是难度最大的
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

9

12. 排序

(x1) 选取

邓俊辉

deng@tsinghua.edu.cn

Simple Algorithms

- ❖ Brute-force: begins with sorting S , and then $\Theta(n \log n)$
scans the sorted sequence $\Theta(k) = O(n)$
halts after k steps
- ❖ Heap selection (A): creates a min-heap of size n , and $\Theta(n)$
calls `delMin()` for k times $\Theta(k \log n)$
- ❖ Heap selection (B): creates a max-heap of size k , and $\Theta(k)$
calls `insert()` and `delMax()` for $n-k$ times $\Theta(2(n-k) \log k)$

Data Structures & Algorithms (Fall 2012), Tsinghua University

Median & Selection

- ❖ Create a max-heap H for k elements
arbitrarily chosen in S $\Theta(k)$
- Create a min-heap G for the rest $n-k$ elements $\Theta(n-k)$
- Repeat
 - comparing h and g , the two tops, and $\Theta(1)$
 - exchanging if necessary $\Theta(2 \times (\log k + \log(n-k)))$
- Until $h \leq g$ $\Theta(\min(k, n-k))$ iterations
- ❖ Can we hope to make it even faster?
- ❖ The k^{th} item can't be determined before all items have been visited at least once
- ❖ Can we hope for such an $\Theta(n)$ selection algorithm?

Data Structures & Algorithms (Fall 2012), Tsinghua University

QuickSelect(A[], n, k)

```
template <typename T>
void quickSelect(Vector<T> & A, Rank k) {
    for (Rank lo = 0, hi = A.size() - 1; lo < hi; ) {
        Rank i = lo, j = hi; T pivot = A[lo];
        while (i < j) { hi = hi - 1; }
        while ((i < j) && (pivot <= A[j])) j--;
        while ((i < j) && (A[i] <= pivot)) i++;
        A[j] = A[i];
    } //assert: i == j
    A[i] = pivot;
    if (k <= i) hi = i - 1;
    if (i < k) lo = i + 1;
} //A[k] is now a pivot
```

Data Structures & Algorithms (Fall 2012), Tsinghua University

SeqSelect(S, k)

- //Let Q be a small constant
1. if ($n=|S| < Q$)
 - sort S and return the k -th element;
 - else
 - subdivide S into n/Q sublists of Q elements each;
 2. Sort each sublist and determine its median;
 - //e.g. by `selectionsort`
 3. Call `SeqSelect` to find //by recursion
 - M , median of the n/Q medians found in step#2;

Data Structures & Algorithms (Fall 2012), Tsinghua University

SeqSelect(S, k)

```
4. Let
  L = {  $x < M$  |  $x \in S$  }; L
  E = {  $x = M$  |  $x \in S$  }; E
  G = {  $x > M$  |  $x \in S$  }; G
  k
```

5. if ($k \leq |L|$)
 return `SeqSelect(L, k); //recursion`
 else if ($k \leq |L|+|E|$)
 return M ;
 else
 return `SeqSelect(G, k-|L|-|E|); //recursion`

Data Structures & Algorithms (Fall 2012), Tsinghua University

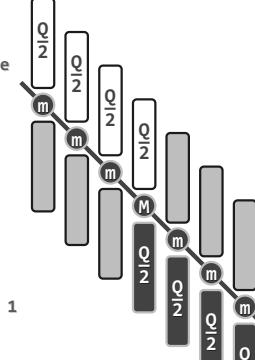
Sequential Select: Complexity

- ❖ Let $T(n) =$
time required in the worst case to
find the k -th smallest item in a list of n items
- ❖ Step 1: $\Theta(n)$
 - $\Theta(Q \log Q) = \Theta(1)$ //sort a list S of length $\leq Q$
 - $\Theta(n)$ //subdivision
- ❖ Step 2: $\Theta(1) \times n/Q = \Theta(n)$ //sort sublists & find medians
- ❖ Step 3: $\Theta(n/Q)$ //find the median of n/Q medians recursively
- ❖ Step 4: $\Theta(n)$ //constructing & counting L/E/G
//requires only one sequential scan of S

Data Structures & Algorithms (Fall 2012), Tsinghua University

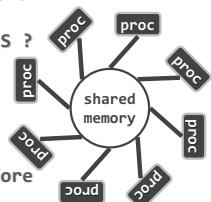
Sequential Select: Complexity

- ❖ Step 5: recursion : $T(3n/4)$
- ❖ Claim: $n/4$ items will be $\geq M$, since
there are $\geq n/Q/2$ medians $\geq M$,
each has $\geq Q/2$ items $\geq M$
- ❖ $\min(|L|, |G|) + |E| \geq n/4$
 $\max(|L|, |G|) \leq 3n/4$
- ❖ $T(n) = \Theta(n) + T(n/Q) + T(3n/4)$
solves to a linear function if
 $n/Q + 3n/4 < n$, or, $1/Q + 3/4 < 1$
- ❖ Let $Q = 5$
 - $T(n) = cn + T(n/5) + T(3n/4)$, c is a constant
 - $T(n) = \Theta(20cn) = \Theta(n)$

Data Structures & Algorithms (Fall 2012), Tsinghua UniversityData Structures & Algorithms (Fall 2012), Tsinghua University

Parallel Select: Problem

- Let S be a list of n items in random order
- We have a **shared-memory** system with p processors
- How to find the k^{th} smallest item in S ?
How fast can we do it?
- Lower bound: $\Omega(n/p)$
The k^{th} item cannot be determined before all items have been visited at least once (by at least one of the p processors)
- Can we hope for an $\mathcal{O}(n/p)$ parallel selection algorithm?



Data Structures & Algorithms (Fall 2012), Tsinghua University

8

ParallelSelect(S, k)

- 4a) (in parallel)
Construct $L_i = \{ \text{elements of } S_i \text{ that are } < m \}$;
 $E_i = \{ \text{elements of } S_i \text{ that are } = m \}$;
 $G_i = \{ \text{elements of } S_i \text{ that are } > m \}$;
- 4b) (in parallel, L/E/G in shared-memory)
Construct $L = \{ \text{elements of } S \text{ that are } < m \}$;
 $E = \{ \text{elements of } S \text{ that are } = m \}$;
 $G = \{ \text{elements of } S \text{ that are } > m \}$;
//Perform a pre-scan on $|L_0|, \dots, |L_{p-1}|$ to get
// $0, |L_0|, |L_0|+|L_1|, |L_0|+|L_1|+|L_2|, \dots$
//Processor P_i places its L_i starting in location
// $|L_0| + |L_1| + \dots + |L_{i-1}|$ of array L
//Do the same for E & G

Data Structures & Algorithms (Fall 2012), Tsinghua University

10

Parallel Select: Complexity

- Let $T(n, p) =$ time required in the worst case to select in a list of n items using p processors & with shared-memory
- Step 1: subdivision
trivial sorting (when $n \leq 5$): $\mathcal{O}(1)$
broadcasting n and $\&S$: $\mathcal{O}(\log p)$
- Step 2: medians
find m_i 's (using SeqSelect()): $\mathcal{O}(n/p)$
- Step 3: global median
find m among the m_i 's (recursively): $T(p, p)$

Data Structures & Algorithms (Fall 2012), Tsinghua University

12

12. 排序

(x2) 希尔排序

郑俊辉

deng@tsinghua.edu.cn

ParallelSelect(S, k)

- if ($n=|S| < 5$)
sort S and return the k^{th} element;
- else
subdivide S into p sublists of n/p elements each;
assign sublist S_i to P_i ;
- for each i in $[0, p)$ do (in parallel)
each processor P_i determines m_i the median of S_i
//by SeqSelect($S_i, |S_i|/2$)
set $M[i] = m_i$;
 $M[0, p)$ is an array in the shared memory
- find m , the median in $M[]$
//by ParallelSelect($M, p/2$)

Data Structures & Algorithms (Fall 2012), Tsinghua University

ParallelSelect(S, k)

- if ($k \leq |L|$)
return ParallelSelect(L, k);
- else if ($k \leq |L|+|E|$)
return m ;
- else
return ParallelSelect($G, k-|L|-|E|$);

Data Structures & Algorithms (Fall 2012), Tsinghua University

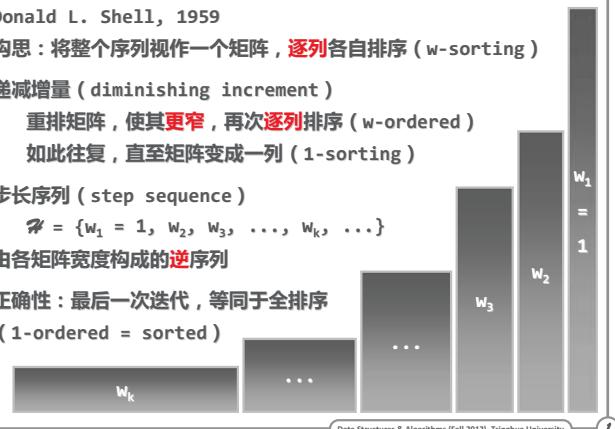
Parallel Select: Complexity

- Step 4a: construct $L_i/E_i/G_i$ in parallel
1 linear pass of S_i , $\mathcal{O}(n/p)$
- Step 4b: construct L/E/G from $(L_i/E_i/G_i)$'s
determine starting locations in L/E/G
(3 hierarchical scans), $\mathcal{O}(3 \times \log p)$
generate L/E/G (1 sequential scan of S_i), $\mathcal{O}(n/p)$
- Step 5: recursion
again, $\max(|L|, |G|) \leq 3n/4$, $T(3n/4, p)$
- $T(n, p)$
= $\mathcal{O}(\log p) + \mathcal{O}(n/p) + T(p, p) + [\mathcal{O}(3\log p) + \mathcal{O}(2n/p)] + T(3n/4, p)$
= $\mathcal{O}(4\log p) + \mathcal{O}(3n/p) + T(p, p) + T(3n/4, p) = \mathcal{O}(n/p)$

Data Structures & Algorithms (Fall 2012), Tsinghua University

Shellsort

- Donald L. Shell, 1959
构思：将整个序列视作一个矩阵，逐列各自排序 (w-sorting)
- 递减增量 (diminishing increment)
重排矩阵，使其更窄，再次逐列排序 (w-ordered)
如此往复，直至矩阵变成一列 (1-sorting)
- 步长序列 (step sequence)
 $\mathcal{W} = \{W_1 = 1, W_2, W_3, \dots, W_k, \dots\}$
由各矩阵宽度构成的逆序列
- 正确性：最后一次迭代，等同于全排序
(1-ordered = sorted)



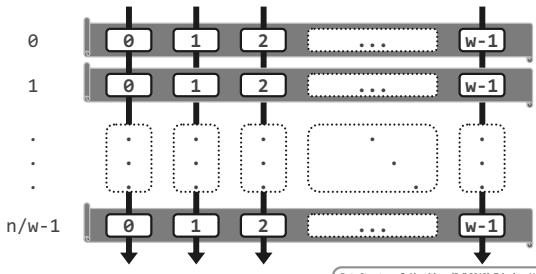
Data Structures & Algorithms (Fall 2012), Tsinghua University

f

Call-by-rank

- 如何实现矩阵重排？莫非，需要使用二维向量？
- 实际上，所有记录可以始终组织为一个一维向量
在每次迭代中，若当前的矩阵宽度为 w ，则

第*i*列中的记录依次就是： $a[i + kw]$, $k = 0, 1, \dots, n/w-1$



Data Structures & Algorithms (Fall 2012), Tsinghua University

Insertionsort

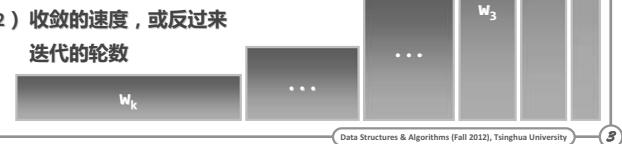
- 各列内部的排序如何实现？不必过于讲究！
比如，可使用insertionsort，因为该算法是输入敏感的...
- 回忆：在insertionsort算法过程中
比较操作次数不超过逆序对 (inversion) 总数
- Shellsort的总体效率
取决于具体使用何种步长序列...

主要考察和评测

1) 比较操作、移动操作的次数

2) 收敛的速度，或反过来

迭代的轮数



Data Structures & Algorithms (Fall 2012), Tsinghua University

Shell's Sequence

- With $\pi_{\text{shell}} = \{1, 2, 4, 8, \dots, 2^k, \dots\}$, //Shell 1959
Shellsort runs in $\Omega(n^2)$ time in the worst cases
- Take a random sequence of $1 \sim 2^n$
with SMALLER/LARGER numbers in EVEN/ODD positions
e.g. 11 4 14 3 10 8 15 1 9 6 16 7 13 2 12 5
- Since all increments except 1 are even, the k^{th} number
will be in position $2k \% (2^n+1)$ after 2-sorting
i.e. 9 1 10 2 11 3 12 4 13 5 14 6 15 7 16 8
- 1-sorting requires $1 + 2 + \dots + 2^{N-1} = \Omega(n^2)$ comparisons
- This is because the increments are not RELATIVELY PRIME

Data Structures & Algorithms (Fall 2012), Tsinghua University

Postage Problem

- The postage for a letter is 16F, and a postcard 11F
But there are only stamps of 3F and 7F available
- Possible to stamp the letter and the postcard EXACTLY?
How about other postages?
- Is it possible to
represent the numbers 16 and 11 as linear combinations:
 $3m + 7n$, where $m, n \in N = \{0, 1, 2, \dots\}$
- Check it now ...

3 7 3 3

Data Structures & Algorithms (Fall 2012), Tsinghua University

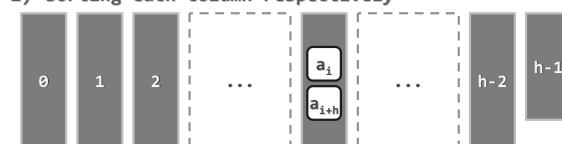
Linear Combination

- Let $g, h \in N$
Then for any $m, n \in N$
 $f = mg + nh$ is called a (linear) combination of g and h
- Let $g, h \in N$ be relatively prime
 $N(g, h) = \{\text{numbers that are NOT combinations of } g \text{ and } h\}$
- Let $x(g, h) = \max(N(g, h))$
i.e. the largest number in $N(g, h)$
- Theorem: $x(g, h) = (g-1)(h-1) - 1$
e.g. $x(3, 7) = 11$, $x(4, 9) = 23$
- Now, it's time to solve the postage problem now ...

Data Structures & Algorithms (Fall 2012), Tsinghua University

h-sorting & h-ordered

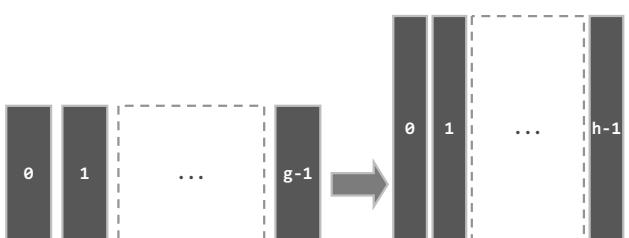
- Let $h \in N$
A sequence $S = \{a_1, \dots, a_n\}$ is called **h-ordered** if
 $a_i \leq a_{i+h}$ holds for $i = 1, \dots, n-h$
- A 1-ordered sequence is sorted
- h-sorting:** an h-ordered sequence is obtained by
 - arranging S into a 2D matrix with h columns and
 - sorting each column respectively



Data Structures & Algorithms (Fall 2012), Tsinghua University

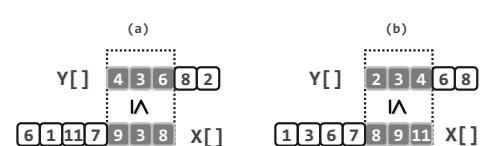
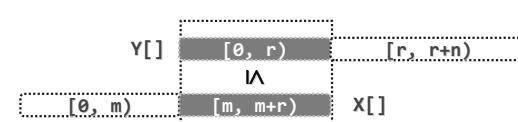
Theorem K

- [Knuth, ACP Vol.3 p.90]
A **g**-ordered sequence REMAINS **g**-ordered
after being **h**-sorted



Data Structures & Algorithms (Fall 2012), Tsinghua University

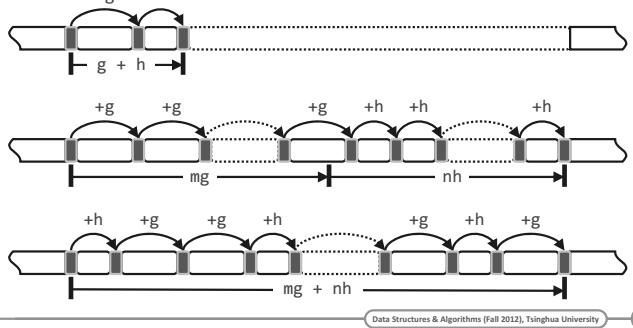
Lemma L



Data Structures & Algorithms (Fall 2012), Tsinghua University

Linear Combination

- A sequence that is both **g**-ordered and **h**-ordered is called **(g, h)**-ordered, which must be both **(g + h)**-ordered and **(mg + nh)**-ordered, $m, n \in \mathbb{N}$

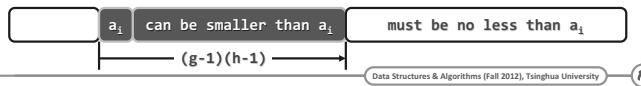


Data Structures & Algorithms (Fall 2012), Tsinghua University

890 d-Sorting an $\Theta(d)$ -Ordered Sequence in $\Theta(dn)$ Time

- If both g and h are in $\Theta(d)$, we can **d-sort** the sequence in $\Theta(dn)$ time
 - re-arrange the sequence as a 2D matrix with d columns
 - each a_i is swapped with $\Theta((g-1)(h-1)/d) = \Theta(d)$ elements

\because this holds for all elements
 $\therefore \Theta(dn)$ steps are enough



Data Structures & Algorithms (Fall 2012), Tsinghua University

891 PS Sequence

- Let h_t be the h closest to $n^{1/2} // h_t = \Theta(n^{1/2})$

- Consider those iterations where $k > t$
 - \therefore there would be $\Theta(n/h_k)$ elements in each of the h_k columns
 - \therefore we can sort each column (say, by insertionsort) in $\Theta(n/h_k)^2$ time
 - \therefore each h_k -sorting costs $\Theta(n^2/h_k)$ time
 - \therefore all these iterations cost $\Theta(2xn^2/h_t) = \Theta(n^{3/2})$

time



Data Structures & Algorithms (Fall 2012), Tsinghua University

892 PS Sequence

PS Sequence

- Papernov & Stasevic, 1965

$$\mathcal{H}_{ps} = \{1, 3, 7, 15, 31, 63, \dots, 2^k - 1, \dots\}$$

<http://thudsa.3322.org/~deng/ds/demo/shellsort/>

- With \mathcal{H}_{ps} ,

Shellsort sorts a sequence of length n in $\Theta(n^{3/2})$ time

- Why ...

893

PS Sequence

- Consider those iterations where $k \leq t$

$\therefore h_{k+1}$ and h_{k+2}

are relatively prime and are both in $\Theta(h_k)$

\therefore each h_k -sorting costs $\Theta(h_k n)$ time

\therefore all these iterations cost

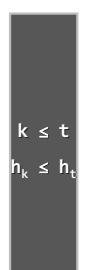
$$\Theta(n \times 2h_t) = \Theta(n^{3/2}) \text{ time}$$

- This upper bound is tight

- How about the average cases?

$\Theta(n^{5/4})$ based on simulations, but

not proved yet



894

PS Sequence

- Consider those iterations where $k > t$

$\therefore h_{k+1}$ and h_{k+2}

are relatively prime and are both in $\Theta(h_k)$

\therefore each h_k -sorting costs $\Theta(h_k n)$ time

\therefore all these iterations cost

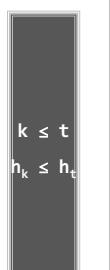
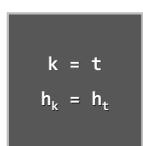
$$\Theta(n \times 2h_t) = \Theta(n^{3/2}) \text{ time}$$

- This upper bound is tight

- How about the average cases?

$\Theta(n^{5/4})$ based on simulations, but

not proved yet



Data Structures & Algorithms (Fall 2012), Tsinghua University

895

Pratt's Sequence

- Pratt, 1971

$$\mathcal{H}_{pratt} = \{1, 2, 3, 4, 6, 8, 9, 12, 16, \dots, 2^p 3^q, \dots\}$$

<http://thudsa.3322.org/~deng/ds/demo/shellsort/>

- With \mathcal{H}_{pratt} ,

Shellsort sorts a sequence of length n in $\Theta(n \log^2 n)$ time
 \therefore Why?

- And, wait a minute, but ...

consecutive numbers in \mathcal{H}_{pratt}
 \therefore are NOT all relatively prime ...

Data Structures & Algorithms (Fall 2012), Tsinghua University

896

Pratt's Sequence

- From $(2, 3)$ -ordered to 1-ordered

$$\therefore g(2, 3) = 1$$

\therefore to the right of each element in a $(2, 3)$ -ordered sequence, only the next element can be smaller

\therefore it costs $\Theta(n)$ time to sort such a sequence

- From $(2h_k, 3h_k)$ -ordered to h_k -ordered

"divide" the sequence into h_k subsequences, each of which is $(2, 3)$ -ordered

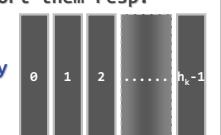
\therefore it costs altogether $\Theta(n)$ time to sort them resp.

- Altogether

\therefore there are $\Theta(\log^2 n)$ iterations //why

\therefore we need time

$$\Theta(n) \times \Theta(\log^2 n) = \Theta(n \log^2 n)$$



Data Structures & Algorithms (Fall 2012), Tsinghua University

17

Sedgewick's Sequence

- ❖ Pratt's sequence performs best asymptotically but requires too many iterations and hence is not good for pre-sorted sequences
- ❖ Sedgewick's sequence:
a combination of PS's sequence with Pratt's
 $\{1, 5, 19, 41, 109, 209, 505, 929, 2161, 3905, 8929, \dots\}$
of form $9 \times 4^k - 9 \times 2^k + 1$ or $4^k - 3 \times 2^k + 1$
worst $\mathcal{O}(n^{4/3})$, average $\mathcal{O}(n^{7/6})$,
best performance in practice
- ❖ Is there a step sequence with $\mathcal{O}(n \log n)$ worst-case performance? //not found yet

详细目录

| | | | |
|-------|-------------------------|-------|----------------------------------|
| 1 | 00.Syllabus | 34 | 级数和 |
| 2 | 教师与助教 | 35~37 | 循环 vs. 级数和 |
| 3 | 选修，还是不选修 | 38 | 取非极端元素 |
| 4~5 | 考评及要求 | 39~41 | 起泡排序 |
| 6 | 教材与教辅 | 42~43 | Back-Of-The-Envelope Calculation |
| 7 | 学习资源 | 44 | 课后 |
| 8 | 01.Introduction | 45 | 01.D.Iteration+Recursion |
| 8 | 01.A.Computation | 46 | 数组求和：迭代 |
| 9 | 绳索计算机及其算法 | 47 | 减而治之 |
| 10 | 尺规计算机及其算法 | 48 | 数组倒置 |
| 11 | 算法 | 49~50 | 数组求和：线性递归 |
| 12 | 好算法 | 51 | 分而治之 |
| 13 | 为何要学？学什么？学习目标？ | 52~54 | 数组求和：二分递归 |
| 14 | 内容纵览 | 55 | fib() |
| 15 | 01.B.Big_O | 56~58 | Hanoi 塔 |
| 16 | 算法分析 | 59 | Hanoi 塔：递归跟踪 |
| 17 | 问题规模 vs. 计算成本 | 60 | Hanoi 塔：递推方程 |
| 18 | 算法 vs. 计算效率 | 61 | 典型的递推方程 |
| 19 | RAM | 62 | 递归，还是不递归？ |
| 20~21 | 渐进分析 | 63 | 尾递归及递归消除 |
| 22 | $O(1)$ | 64 | 课后 |
| 23 | $O(\log^c n)$ | 65 | 01.X1.Limitation |
| 24 | $O(n^c)$ | 66 | 循环移位：蛮力版 |
| 25 | $O(2^n)$ | 67 | 循环移位：迭代版 |
| 26~27 | 2-Subset | 68 | 循环移位：倒置版 |
| 28~29 | 增长速度 | 69 | 幂函数： $O(2^{r^n})$ |
| 30 | 复杂度层次 | 70 | a^{98765} |
| 31 | 课后 | 71 | a^{10110_b} |
| 32 | 01.C.Algorithm_analysis | 72 | 幂函数： $O(r^n)$ |
| 33 | 算法分析 | 73 | 幂函数：悖论 |

| | | | |
|-----|--|---------|----------------------|
| 74 | 随机置乱 | 105 | 区间删除 |
| 75 | 01.X2.Sorting+Lower_Bound | 106 | 删除 |
| 76 | 难度与下界 | 107 | 查找 |
| 77 | 排序 | 108 | 唯一化：算法 |
| 78 | 算法分类 | 109 | 唯一化：正确性及复杂度 |
| 79 | 时空性能、稳定性 | 110 | 遍历 |
| 80 | 最坏情况最优 + 基于比较 | 111 | 遍历：实例 |
| 81 | 判定树 | 112 | 02.D.Sorted_Vector |
| 82 | 代数判定树 | 113 | 有序性及其甄别 |
| 83 | 下界： $\Omega(n \log n)$ | 114 | 唯一化（低效版）：算法 |
| 84 | 熵与下界 | 115 | 唯一化（低效版）：复杂度 |
| 85 | 课后 | 116 | 唯一化（高效版）：算法 |
| 86 | 02.Vector | 117 | 唯一化：实例与复杂度 |
| 86 | 02.A.Interface+Implementation | 118 | 查找 |
| 87 | Abstract Data Type vs. Data Structure | 119 | 二分查找（版本 A）：原理 |
| 88 | Application = Interface × Implementation | 120 | 二分查找（版本 A）：实现 |
| 89 | 从数组到向量 | 121 | 二分查找（版本 A）：实例与复杂度 |
| 90 | 向量 ADT 接口 | 122~123 | 二分查找（版本 A）：查找长度 |
| 91 | ADT 操作实例 | 124 | Fibonacci 查找：改进思路及原理 |
| 92 | Vector 模板类 | 125 | Fibonacci 查找：实现 |
| 93 | 构造与析构 | 126 | Fibonacci 查找：查找长度 |
| 94 | 基于复制的构造 | 127 | 二分查找（版本 B）：改进思路 |
| 95 | 02.B.extendable_vector | 128 | 二分查找（版本 B）：实现 |
| 96 | 静态空间管理 | 129 | 二分查找（版本 B）：性能及进一步要求 |
| 97 | 动态空间管理 | 130 | 二分查找（版本 C）：实现 |
| 98 | 扩容算法实现 | 131 | 二分查找（版本 C）：正确性 |
| 99 | 容量递增策略 | 132 | 插值查找 |
| 100 | 容量加倍策略 | 133 | 插值查找：实例 |
| 101 | 平均分析 vs. 分摊分析 | 134~135 | 插值查找：性能 |
| 102 | 02.C.unsorted_Vector | 136 | 课后 |
| 103 | 元素访问 | 137 | 02.E.Bubblesort |
| 104 | 插入 | 138 | 排序器：统一入口 |
| | | 139 | 起泡排序 |
| | | 140 | 综合评价 |

| | | | |
|---------|-------------------------------|---------|-------------------------------|
| 141 | 02.F.Mergesort | 174 | 实现 : selectionSort() |
| 142 | 归并排序 : 原理 | 175 | 实现 : selectMax() |
| 143 | 归并排序 : 实现 | 176 | 性能 |
| 144 | 二路归并 : 原理 | 177 | 03.E.Insertionsort |
| 145 | 二路归并 : 实现 | 178 | 构思 |
| 146 | 二路归并 : 复杂度 | 179 | 实例 |
| 147 | 综合评价 | 180 | 实现 |
| 148 | 03.List | 181 | 平均性能 |
| 148 | 03.A.interface+Implementation | 182 | 03.F.Mergesort |
| 149 | 从静态到动态 | 183 | 归并排序 |
| 150 | 从向量到列表 | 184 | 二路归并 |
| 151~152 | 从秩到位置 | 185 | 03.X1.Cursor |
| 153 | 列表节点 : ADT 接口 | 186 | 动机与构思 |
| 154 | 列表节点 : ListNode 模板类 | 187 | 原理与实例 |
| 155 | 列表 : ADT 接口 | 188 | 03.X2.Java_Sequence |
| 156 | 列表 : List 模板类 | 189 | Interface : 定义 |
| 157 | 构造 | 190 | Interface : 实现 |
| 158 | 析构 | 191 | 向量接口 : Vector.java |
| 159 | 03.B.Unsorted_list | 192 | 向量实现 1 : Vector_Array.java |
| 160 | 秩到位置 | 193 | 向量实现 2 : Vector_ExtArray.java |
| 161 | 查找 | 194 | 序列接口及实现 |
| 162 | 插入 | 195 | 03.X3.Python_List |
| 163 | 基于复制的构造 | 196~198 | Python List |
| 164 | 删除 | 199 | reverse() : 循位置访问 ? 循秩访问 ? |
| 165 | 析构 | | |
| 166 | 唯一化 | 200 | 04.Stack+Queue |
| 167 | 遍历 | 200 | 04.A.stack-ADT+implementation |
| 168 | 03.C.Sorted_list | 201 | 操作与接口 |
| 169 | 唯一化 | 202 | 操作实例 |
| 170 | 查找 | 203 | 模板类 |
| 171 | 03.D.Selectionsort | 204 | 04.B.stack-execution_stack |
| 172 | 构思 | 205 | 函数调用栈 |
| 173 | 实例 | 206 | 实例 : fac() |

| | | | |
|---------|--|---------|--|
| 207 | 实例 : fib() | 247 | PostScript |
| 208 | 实例 : hailstone() | 248 | 04.D1.Probe_Backtrack-8queens |
| 209 | 避免递归 | 249 | 指数爆炸 |
| 210 | 04.C1.stack-Application-conversion | 250 | 试探-回溯-剪枝 |
| 211 | 典型应用场合 | 251~252 | 八皇后 |
| 212 | 问题 | 253 | 编码 |
| 213 | 思路 | 254 | 蛮力搜索 |
| 214 | 难点及解决方法 | 255 | 剪枝 |
| 215 | 算法实现 | 256 | 通用算法 |
| 216 | 04.C2.stack-Application-parentheses | 257 | 实例 |
| 217 | 栈混洗 | 258 | 数据结构 |
| 218~219 | 栈混洗 : 计数 | 259 | 04.D2.Probe_Backtrack-maze |
| 220~222 | 栈混洗 : 颁别 | 260 | 迷宫寻径 |
| 223 | 括号匹配 | 261 | 算法 |
| 224 | 括号匹配 : 构思 | 262 | 数据结构 |
| 225 | 括号匹配 : 实现 | 263 | 进一步的考虑 |
| 226~227 | 括号匹配 : 拓展 | 264 | 04.E.Queue-ADT+implementation |
| 228 | 04.C3.stack-Application-infix_expression | 265 | 操作与接口 |
| 229~230 | 表达式求值 | 266 | 操作实例 |
| 231 | 延迟缓冲 | 267 | 模板类 |
| 232 | 求值算法 = 栈 + 线性扫描 | 268 | 04.F.Queue-Application |
| 233 | 实现 : 主算法 | 269 | 资源循环分配 |
| 234 | 实现 : 优先级表 | 270~271 | 银行服务模拟 |
| 235 | 实现 : 不同优先级处理方法 | 272 | 04.X.Steap+Queap |
| 236 | 回顾 : 优先级表 | 273 | Steap = Stack + Heap = push + pop
+ getMax |
| 237~239 | 实例 | 274 | Queap = Queue + Heap = enqueue +
dequeue + getMax |
| 240 | 04.C4.stack-Application-rpn | 275 | 05.Tree |
| 241 | RPN | 275 | 05.A.Introduction |
| 242 | RPN 求值 : 算法 | 276 | 应用 |
| 243 | RPN 求值 : 实例 | 277 | 有根树 |
| 244 | 0 ! 1 + 2 3 ^ 4 ! - 5 ! 6 / - 7 *
8 * - 9 - | 278 | 有序树 |
| 245 | infix 到 postfix : 手工转换 | | |
| 246 | infix 到 postfix : 转换算法 | | |

| | | | |
|---------|---------------------|---------|----------------------|
| 279 | 路径与环路 | 315 | 递归 |
| 280~281 | 深度与层次 | 316 | 迭代：难点 |
| 282 | 05.B.Representation | 317~318 | 迭代：思路 |
| 283 | 接口 | 319 | 迭代：实现 |
| 284 | 父节点 | 320 | 迭代：实例 |
| 285 | 孩子节点 | 321 | 迭代：正确性 |
| 286 | 父节点 + 孩子节点 | 322 | 迭代：效率 |
| 287 | 长子 + 兄弟 | 323 | 直接后继 |
| 288 | 05.C.Introduction | 324 | 05.E3.Postorder |
| 289 | 二叉树 | 325 | 递归 |
| 290~291 | 基数 | 326 | 迭代：难点 |
| 292 | 真二叉树 | 327~328 | 迭代：思路 |
| 293 | 描述多叉树 | 329~330 | 迭代：实现 |
| 294 | 05.D.Implementation | 331~332 | 迭代：实例 |
| 295 | BinNode 模板类 | 333 | 迭代：正确性 |
| 296 | BinNode 接口实现 | 334 | 迭代：效率 |
| 297 | BinTree 模板类 | 335 | 05.E4.LevelOrder |
| 298 | 高度更新 | 336 | 实现 |
| 299 | 节点插入 | 337~338 | 实例 |
| 300 | 子树接入 | 339 | 分析 |
| 301 | 子树删除 | 340 | 应用：表达式树 |
| 302 | 子树分离 | 341 | 完全二叉树 |
| 303 | 05.E1.Preorder | 342 | 05.F.Encoding_Tree |
| 304 | 遍历 | 343 | 问题 |
| 305 | 递归 | 344 | 二叉编码树 |
| 306 | 迭代 1：思路 | 345 | 编码长度 vs. 叶节点平均深度 |
| 307 | 迭代 1：实现 | 346 | 最优编码树 |
| 308 | 迭代 1：实例 | 347 | 带权编码长度 vs. 叶节点平均带权深度 |
| 309 | 迭代 1：分析 | 348 | 最优编码树 |
| 310~311 | 迭代 2：思路 | 349 | 05.G.Huffman_Tree |
| 312 | 迭代 2：实现 | 350 | 策略与算法 |
| 313 | 迭代 2：实例 | 351 | 正确性？ |
| 314 | 05.E2.Inorder | 352 | 双子性 |
| | | 353 | 不唯一性 |
| | | 354~355 | 层次性 |

| | | | |
|---------|-------------------------------|---------|------------------|
| 356 | 正确性 ! | 393 | 复杂度 |
| 357 | 实现 : 构造编码树 | 394 | 树边 & 跨边 |
| 358 | 实现 : 搜索最小超字符 | 395 | BFS 树 / 森林 |
| 359 | 实现 : 构造编码表 | 396 | 连通分量与可达分量 |
| 360~361 | 改进 | 397 | 最短路径 |
| 362 | 06.Graph | 398 | 06.D.DFS |
| 362 | 06.A.Introduction | 399 | 算法 |
| 363 | 基本术语 | 400 | Graph::DFS() |
| 364 | 无向图 / 有向图 | 401~404 | 实例 (无向图) |
| 365 | 路径 / 环路 | 405 | Graph::dfs() |
| 366 | 支撑树 / 带权网络 / 最小支撑树 | 406~410 | 实例 (有向图) |
| 367 | 06.B.Interface+Implementation | 411 | 复杂度 |
| 368 | Graph 模板类 | 412 | DFS 树 / 森林 |
| 369 | 邻接矩阵与关联矩阵 | 413 | 时间标签 |
| 370 | 邻接矩阵 : 实例 | 414 | 边分类 |
| 371 | 邻接矩阵 : Vertex | 415 | 遍历算法的应用 |
| 372 | 邻接矩阵 : Edge | 416 | 06.E.TS |
| 373 | 邻接矩阵 : GraphMatrix | 417 | 有向无环图 |
| 374 | 邻接矩阵 : 顶点查询接口 | 418 | 拓扑排序 |
| 375 | 邻接矩阵 : 边查询接口 | 419 | 存在性 |
| 376~377 | 邻接矩阵 : 顶点修改接口 | 420 | 算法 A : 顺序输出零入度顶点 |
| 378 | 邻接矩阵 : 边修改接口 | 421 | 算法 B : 逆序输出零出度顶点 |
| 379 | 邻接矩阵 : 优点 | 422 | 算法 B : 实现 |
| 380 | 邻接矩阵 : 缺点 | 423 | 06.F.PFS |
| 381 | 邻接表 | 424 | 通用算法 |
| 382 | 邻接表 : 实例 | 425 | 统一框架 |
| 383 | 邻接表 : 空间复杂度 | 426 | 复杂度 |
| 384~385 | 邻接表 : 接口 & 复杂度 | 427 | 06.G.Prim |
| 386 | 取舍原则 | 428 | 最小 + 支撑 + 树 |
| 387 | 06.C.BFS | 429 | MST |
| 388 | 算法 | 430 | 退化 |
| 389 | Graph::BFS() | 431 | 蛮力算法 |
| 390~391 | 实例 (无向图) | 432 | 割 & 极短跨边 |
| 392 | Graph::bfs() | 433 | 环 & 极长环边 |
| | | 434 | 算法 |

| | | | |
|---------|----------------------|---------|---------------------------------|
| 435~437 | 实例 | 485 | 多起点：从 Dijkstra 到 Floyd-Warshall |
| 438 | 正确性 | 486 | 多起点：问题特点 |
| 439~440 | 实现 | 487 | 多起点：递归 |
| 441 | 06.H.Dijkstra | 488 | 动态规划 |
| 442 | 问题描述 | 489 | 算法 |
| 443 | 问题分类 | 490 | 复杂度 |
| 444 | E. W. Dijkstra | | |
| 445~446 | SPT | 491 | 07.BST |
| 447 | $SPT \neq MST$ | | |
| 448 | u1 | 491 | 07.A.Interface |
| 449 | u2 | 492 | 查找 |
| 450 | uk | 493 | 循关键码访问 |
| 451 | 算法 | 494 | 二叉搜索树 |
| 452~454 | 实例 | 495 | 中序遍历序列 |
| 455~456 | 实现 | 496 | BST 模板类 |
| 457 | 06.X1.BCC | | |
| 458 | 关节点 & 双连通分量 | 497 | 07.B.algorithms+implementation |
| 459 | Brute-Force | 498 | 查找：算法 |
| 460 | 根顶点 | 499 | 查找：实现 |
| 461 | 内顶点 | 500 | 插入：算法 |
| 462 | Graph::BBC() | 501 | 插入：实现 |
| 463~464 | switch (status(u)) | 502 | 删除：算法 |
| 465~468 | 实例 | 503~504 | 删除：情况一 |
| 469 | 复杂度 | 505~506 | 删除：情况二 |
| 470 | 06.X2.Kruskal | | |
| 471 | 贪心策略 | 507 | 07.C.balance+equivalence |
| 472 | 算法框架 | 508 | 树高 |
| 473 | 正确性 | 509 | 随机生成 |
| 474 | 排序 | 510 | 随机组成 |
| 475 | 回路检测 | 511 | 理想平衡与适度平衡 |
| 476 | 树合并 | 512 | 等价 BST |
| 477~478 | Union-Find | 513 | 等价变换与局部调整 |
| 479 | 复杂度 | 514 | 07.D.AVL |
| 480 | 其它算法 | 515 | 平衡因子与 AVL 平衡 |
| 481~482 | 更新结果 | 516 | AVL：接口 |
| 483 | 相关话题 | 517 | 失衡与重平衡 |
| 484 | 06.X3.Floyd-Warshall | 518 | 插入：单旋 |

| | | | |
|---------|-------------------|---------|-------------------------|
| 519 | 插入 : 双旋 | 559 | 查找 : 实现 |
| 520 | 插入 : 实现 | 560 | 查找 : 实例 |
| 521 | 删除 : 单旋 | 561 | 查找 : 复杂度 |
| 522 | 删除 : 双旋 | 562 | 最大树高 |
| 523 | 删除 : 实现 | 563 | 最小树高 |
| 524 | 3+4 重构 : 算法 | 564 | 意义与价值 |
| 525 | 3+4 重构 : 实现 | 565 | 插入 : 算法 |
| 526 | 统一调整 : 实现 | 566~567 | 上溢修复 : 算法 |
| 527 | 综合评价 | 568 | 上溢修复 : 实现 |
| | | 569~570 | 插入 : 实例 |
| 528 | 08.ABST | 571 | 删除 : 算法 |
| | | 572~573 | 下溢修复 : 算法 |
| 528 | 08.A.Splay_Tree | 574 | 下溢修复 : 实现 |
| 529 | 局部性 | 575 | 下溢修复 : 实现 : 情况 1、2 |
| 530 | 逐层伸展 | 576 | 下溢修复 : 实现 : 情况 3 |
| 531 | 逐层伸展 : 实例 | 577 | 删除 : 实例 : 底层节点 |
| 532 | 最坏情况 | 578 | 删除 : 实例 : 非底层节点 |
| 533 | 低效率的根源 | 579 | 课后 |
| 534 | 双层伸展 | | |
| 535 | zig-zag / zag-zig | 580 | 08.X1.Red-Black_Tree |
| 536~537 | zig-zig / zag-zag | 581 | 一致性结构 |
| 538 | zig / zag | 582 | O(1)重构 |
| 539 | 实现 : 伸展树接口 | 583 | 红 vs. 黑 |
| 540~541 | 实现 : 伸展算法 | 584 | (2,4)树 vs. 红黑树 |
| 542 | 实现 : 查找算法 | 585 | 红黑树 ∈ BBST |
| 543 | 实现 : 插入算法 | 586 | RedBlack |
| 544 | 实现 : 删除算法 | 587 | 插入 : 算法 |
| 545 | 综合评价 | 588 | 插入 : 实现 |
| 546 | 08.B.B-Tree | 589 | 双红修正 : 算法 |
| 547~548 | 越来越小的内存 | 590~591 | 双红修正(1) : u->color == B |
| 549~550 | 高速缓存 | 592 | 双红修正(1) : 实现 |
| 551~553 | B-Tree | 593~595 | 双红修正(2) : u->color == R |
| 554 | 紧凑表示 | 596 | 双红修正(2) : 实现 |
| 555 | 实例 | 597 | 双红修正 : 复杂度 |
| 556 | BTNode | 598~599 | 删除 : 算法 |
| 557 | BTree | | |
| 558 | 查找 : 算法 | 600 | 删除 : 实现 |

| | | | |
|---------|-----------------------------------|---------|------------------------------------|
| 601 | 双黑修正 : 算法 | 641~642 | Priority Search Tree |
| 602~603 | 双黑修正(1) : s 为黑 , 且至少有一个红孩子 t | 643~644 | Segment Tree |
| 604 | 双黑修正(1) : 实现 | 645 | Trie = reTRIEval |
| 605 | 双黑修正(2R) : s 为黑 , 且两个孩子均为黑 ; p 为红 | 646 | Trie: PATRICIA Tree & Ternary Trie |
| 606 | 双黑修正(2B) : s 为黑 , 且两个孩子均为黑 ; p 为黑 | 647 | 09.Dictionary |
| 607 | 双黑修正(2R+2B) : 实现 | 648 | 联合数组 |
| 608 | 双黑修正(3) : s 为红 (其孩子均为黑) | 649 | 映射 + 词典 = 符号表 |
| 609 | 双黑修正(3) : 实现 | 650~651 | Dictionary |
| 610 | 双黑修正 : 复杂度 | 652 | Java: HashMap + Hashtable |
| 611 | java.util.TreeMap | 653 | Perl: %Hash Type |
| 612 | 08.X2.Kd-Tree | 654 | Python: Dictionary Class |
| 613 | 范围查找 | 655 | Ruby: Hash Table |
| 614 | 蛮力 | 656 | 课后 |
| 615 | 二分查找 | 657 | 09.B.Hashtable_hashing |
| 616 | 平面 | 658 | 85001 |
| 617 | BBST : 结构 | 659 | Tsinghua |
| 618 | BBST : 查找 | 660 | IP Dictionary |
| 619 | BBST : 效率 | 661 | 原理 |
| 620 | 构思 | 662 | 实例 |
| 621 | 构思实例 | 663 | 时空效率 |
| 622 | 构造算法 | 664 | 冲突 |
| 623~625 | 构造实例 | 665 | 完美散列 |
| 626 | 正则子集 | 666 | 生日悖论 |
| 627 | 查找算法 | 667 | 09.C.Hashtable_Hashing-Function |
| 628 | 查找实例 | 668 | 评价标准与设计原则 |
| 629 | 包围盒 | 669 | 除余法 |
| 630 | 效率 | 670~671 | 更多散列函数 |
| 631 | 高维推广 | 672~673 | (伪) 随机数法 |
| 632 | 课后 | 674~675 | 多项式法 |
| 633 | 08.X3.More_Search_Trees | 676 | Java: hashCode() |
| 634 | Quadtree | 677 | 普遍存在的冲突 |
| 635~637 | Range Tree | 678 | 09.D.Hashtable_Solving-Collision |
| 638~640 | Interval Tree | 679 | 多槽位法 |
| | | 680 | 独立链法 |
| | | 681 | 公共溢出区法 |

| | | | |
|---------|---------------------------|-----|---------------------------|
| 682 | 开放定址策略 | 723 | 10.A.Basic_Implementation |
| 683 | 线性试探法 | 724 | 优先级队列 |
| 684~685 | 懒惰删除 | 725 | 应用、算法与特点 |
| 686~687 | 平方试探法 | 726 | 向量实现 |
| 688~689 | 双向平方试探法 | 727 | 列表实现 |
| 690 | (伪)随机试探法 | 728 | 更好的实现 |
| 691 | 再散列法 | 729 | PQ |
| 692 | 预防冲突：重散列 | 730 | 统一测试 |
| 693 | 09.E.Bucketsort+Radixsort | 731 | 10.B.Complete_Binary_Heap |
| 694 | 桶排序：简单情况 | 732 | 结构性 |
| 695~696 | 桶排序：一般情况 | 733 | PQ_CmplHeap |
| 697~698 | MaxGap | 734 | 堆序性 |
| 699 | 基数排序：策略与算法 | 735 | 插入与上滤：算法 |
| 700 | 基数排序：正确性与性能 | 736 | 插入与上滤：实现 |
| 701 | 整数排序 | 737 | 插入与上滤：实例 |
| 702 | 09.X1.Skiplist | 738 | 插入与上滤：效率 |
| 703 | 动机与思路 | 739 | 删除与下滤：算法 |
| 704 | 结构：设计 | 740 | 删除与下滤：实现 |
| 705 | 空间性能 | 741 | 删除与下滤：实例 |
| 706 | 结构：QuadList | 742 | 删除与下滤：效率 |
| 707 | 结构：Skiplist | 743 | 批量构造：自上而下的上滤：蛮力 |
| 708 | 查找：skipSearch() | 744 | 批量构造：自下而上的下滤：算法及实现 |
| 709 | 查找：实例 | 745 | 批量构造：自下而上的下滤：实例 |
| 710 | 查找：纵向跳转/层高 | 746 | 批量构造：自下而上的下滤：效率 |
| 711 | 查找：横向跳转/紧邻塔顶 | 747 | 课后 |
| 712~713 | 插入：put() | 748 | 10.C.Tournamentsort |
| 714 | 插入：实例 | 749 | 锦标赛树 |
| 715 | 删除：remove() | 750 | 算法 |
| 716 | 09.X2.Md5 | 751 | 实例 |
| 717~719 | MD5 | 752 | 实现与效率 |
| 720 | 散列指纹 | 753 | 课后 |
| 721 | MD5 算法 | 754 | 10.D.Heapsort |
| 722 | 课后 | 755 | 算法 |
| | | 756 | 实现 |

| | | | |
|---------|--------------------|---------|------------------------|
| 757 | 实例：建堆 | 793 | 蛮力：复杂度 |
| 758~759 | 实例：选取+调整 | 794 | 11.C.KMP |
| 760 | 综合评价 | 795~796 | 改进思路 |
| 761 | 10.X1.Leftist_Heap | 797 | KMP 算法 |
| 762 | 堆合并 | 798 | KMP 算法：实例 |
| 763 | 单侧倾斜 | 799 | next[j]的含义：避免回溯 |
| 764 | 空节点路径长度 | 800 | next[j]的含义：不致遗漏 |
| 765 | 左倾性 & 左式堆 | 801~802 | next[]的构造：递推 |
| 766 | 右侧链 | 803 | next[]的构造：算法 |
| 767 | LeftHeap | 804~805 | next[]的构造：实例 |
| 768 | 合并：算法 | 806 | String-Search Compiler |
| 769 | 合并：实现 | 807~808 | 复杂度 |
| 770 | 合并：实例 | 809 | 再改进：思路 |
| 771 | 合并：基本操作 | 810 | 再改进：算法 |
| 772 | 合并：实现插入、删除接口 | 811 | 再改进：实例 |
| 773 | 10.X2.d-heap | 812 | 小结 |
| 774 | 优先级搜索 | 813 | 11.D.BM |
| 775 | 优先级队列 | 814 | 构思 |
| 776~778 | 多叉堆 | 815 | 实例 |
| 779 | Fibonacci 堆 | 816~819 | Bad-Character Shift |
| | | 820 | 实例 |
| | | 821 | BC[]表的构造：算法 |
| 780 | 11.String | 822~823 | BC 算法：查找时间 |
| 780 | 11.A.ADT | 824 | Bad-Character Shift：不足 |
| 781 | 定义 | 825~826 | Good-Suffix Shift |
| 782 | 术语 | 827 | GS[]表的构造 |
| 783 | ADT | 828 | 实例 |
| 784 | 实例 | 829 | BC + GS：性能分析 |
| 785 | 11.B.PM | 830 | 11.E.Karp-Rabin |
| 786~787 | 串匹配 | 831~833 | 凡物皆数 |
| 788 | 评测标准与策略 | 834 | 串亦为数 |
| 789 | 蛮力：构思 | 835 | 数位溢出 |
| 790 | 蛮力：实现 | 836 | Karp-Rabin：散列压缩 |
| 791 | 蛮力：版本 1 | 837 | Karp-Rabin：散列冲突 |
| 792 | 蛮力：版本 2 | 838 | Karp-Rabin：快速指纹计算 |

| | | | |
|---------|-------------------------------|---------|---|
| | | 877~878 | Parallel Select: Complexity |
| 839 | 12.Sorting | 879 | 12.X2.Shellsort |
| 839 | 12.A. Quicksort | 880 | Shellsort |
| 840 | 构思 | 881 | Call-by-rank |
| 841 | 轴点 | 882 | Insertionsort |
| 842 | 快速排序 | 883 | Shell's Sequence |
| 843 | 构造轴点 | 884 | Postage Problem |
| 844 | 算法 A : 实现 | 885 | Linear Combination |
| 845 | 算法 A : 过程 | 886 | h-sorting & h-ordered |
| 846 | 算法 A : 实例 | 887 | Theorem K |
| 847 | 算法 A : 性能分析 | 888 | Lemma L |
| 848 | 平均性能 | 889 | Linear Combination |
| 849 | 重复元素 | 890 | Inversion |
| 850 | 算法 A1 : 实现 | 891 | d-Sorting an O(d)-Ordered Sequence
in O(dn) Time |
| 851 | 算法 B1 : 实现 | 892~894 | PS Sequence |
| 852 | 算法 B : 实现 | 895~896 | Pratt's Sequence |
| 853 | 算法 B : 性能分析 | 897 | Sedgewick's Sequence |
| 854 | 算法 C : 思路 | | |
| 855 | 算法 C : 实现 | | |
| 856 | 12.B.Median | | |
| 857 | 选取与中位数 | | |
| 858 | 主流数 | | |
| 859 | 主流数 : 减而治之 | | |
| 860 | 主流数 : 算法 | | |
| 861 | 归并向量的中位数 | | |
| 862 | 归并向量的中位数 : 等长子向量 : 构思 | | |
| 863 | 归并向量的中位数 : 等长子向量 : 实现 | | |
| 864 | 归并向量的中位数 : 任意子向量 : 实现 | | |
| 865 | 12.X1.Selection | | |
| 866 | Simple Algorithms | | |
| 867 | Median & Selection | | |
| 868 | QuickSelect(A[], n, k) | | |
| 869~870 | SeqSelect(S, k) | | |
| 871~872 | Sequential Select: Complexity | | |
| 873 | Parallel Select: Problem | | |
| 874~876 | ParallelSelect(S, k) | | |