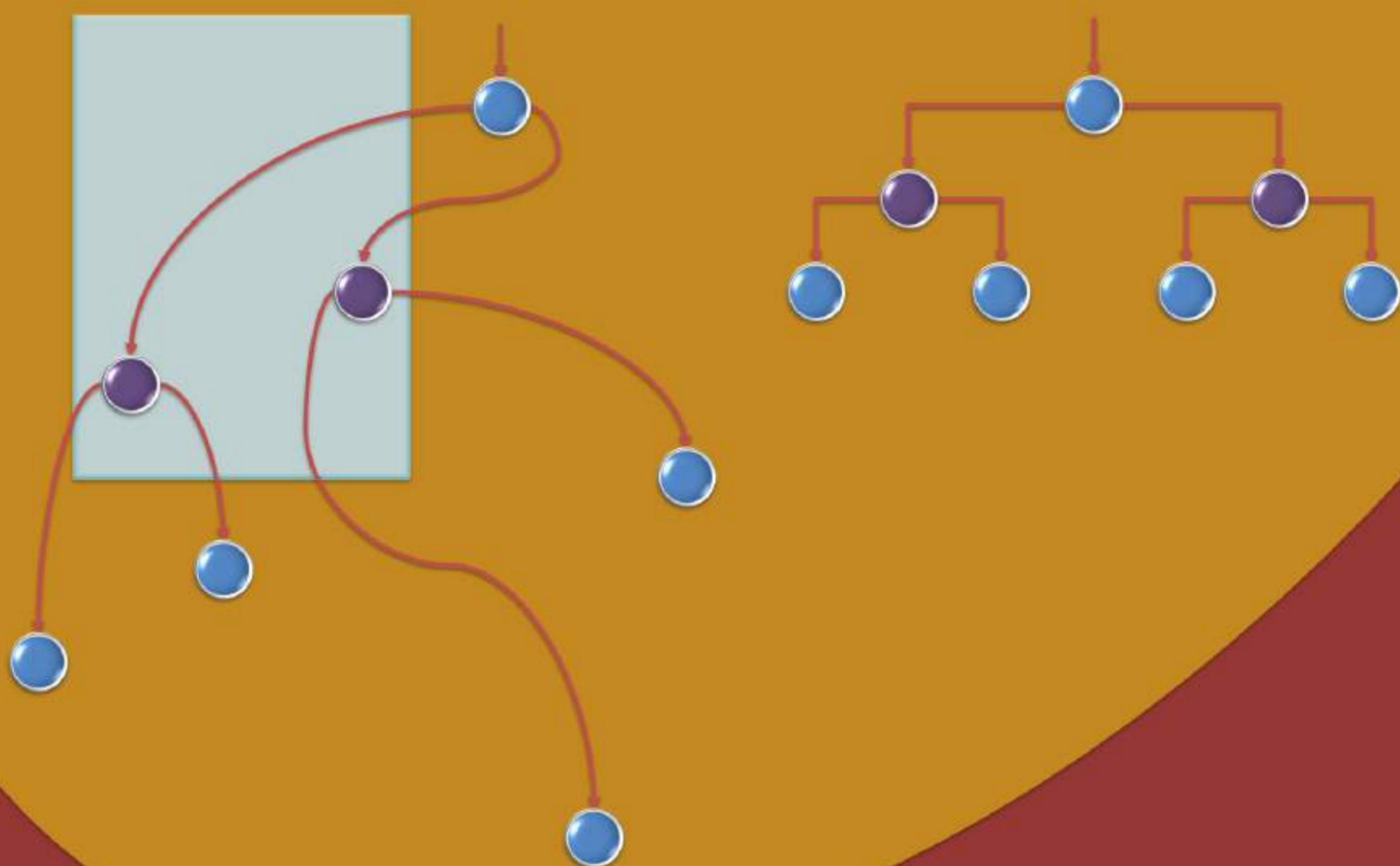
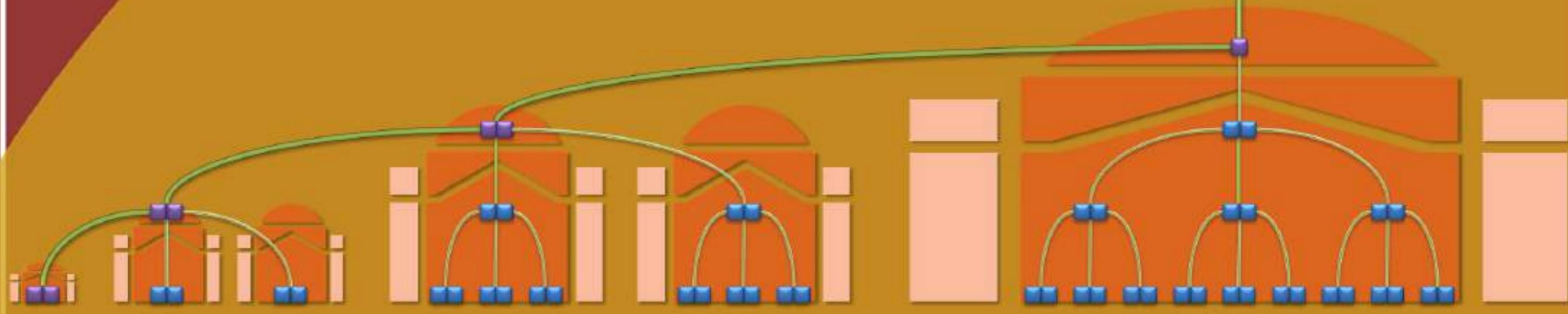


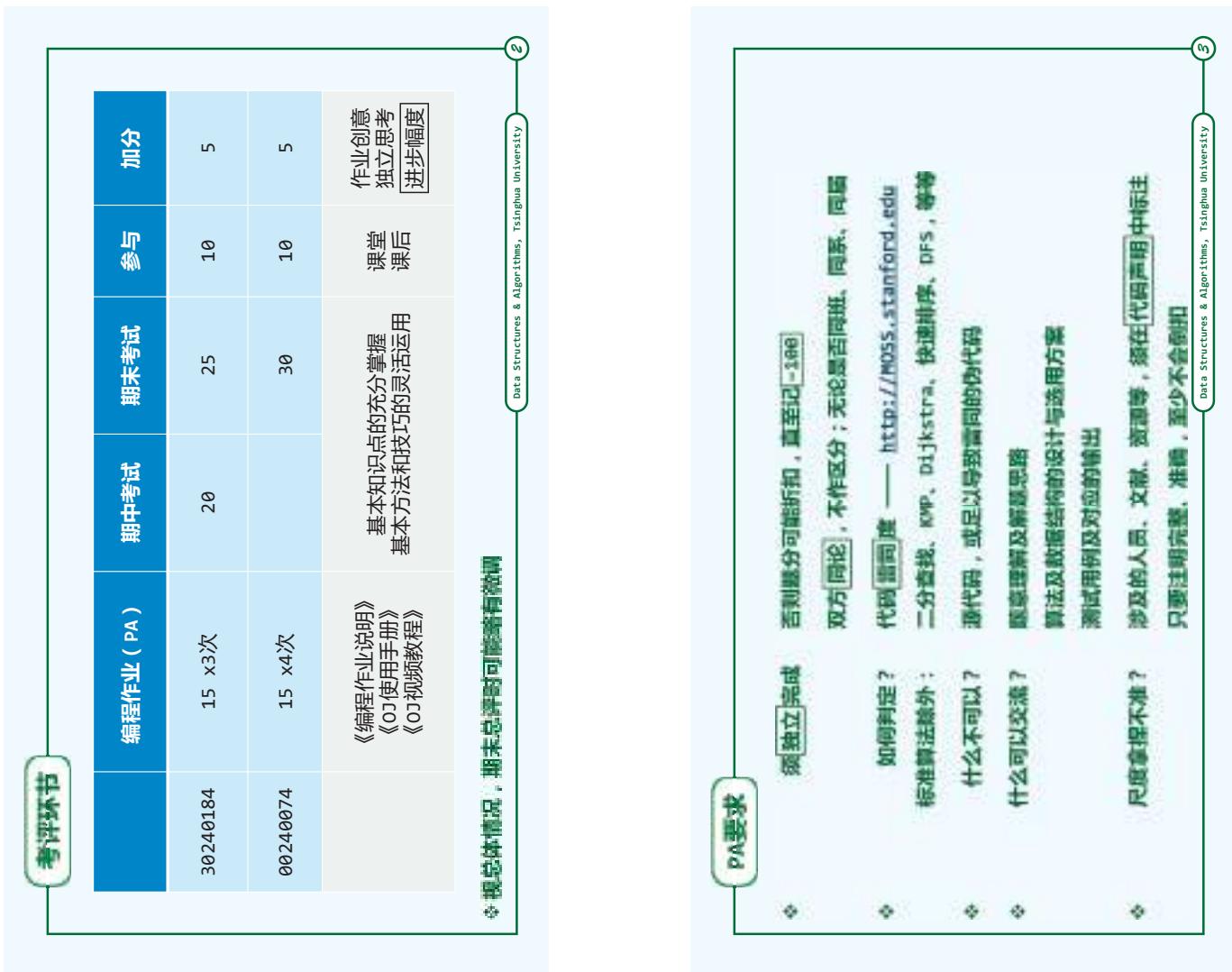
# 数据结构



清华大学 · 2016秋







## 教材 + 教辅

- 

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

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

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

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

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

Data Structures & Algorithms, Tsinghua University

⑤ Data Structures & Algorithms, Tsinghua University

## 基础知识

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

Data Structures & Algorithms, Tsinghua University

⑥ Data Structures & Algorithms, Tsinghua University

## 选修，还是不选修

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



Data Structures & Algorithms, Tsinghua University

⑦ Data Structures & Algorithms, Tsinghua University

## 平台 + 资源



Data Structures & Algorithms, Tsinghua University

⑧ Data Structures & Algorithms, Tsinghua University

# 目 录

01. 绪论		02. 向量	
	A. 计算	91	02. 最长公共子序列
1	01. 工具	99	XB. 局限
4	02. 算法	105	01. 缓存
8	03. 优劣	111	02. 字宽
	B. 计算模型		03. 随机数
12	01. 统一尺度	113	XC. 下界
16	02. 图灵机	122	01. 代数判定树
20	03. RAM		02. 归约
	C. 渐进分析		
26	01. 大 O 记号		A. 抽象数据类型
32	02. 多项式	127	01. 接口与实现
37	03. 指数	131	02. 从数组到向量
43	04. 复杂度层次	135	03. 模板类
	D. 算法分析		B. 可扩充向量
48	01. 级数	139	01. 算法
52	02. 迭代	144	02. 分摊
56	03. 正确性		C. 无序向量
62	04. 封底估算	149	01. 基本操作
	E. 迭代与递归	154	02. 查找
66	01. 减而治之	158	03. 去重
72	02. 分而治之	163	04. 遍历
77	03. Max2		D. 有序向量
82	04. 尾递归	166	01. 唯一化
	XA. 动态规划	173	02. 二分查找 ( A )
86	01. 记忆化	183	03. Fib 查找

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

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

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

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

## 09. 词典

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

## 10. 优先级队列

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

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

## 1. 绪论

### 计算 工具

从报纸而不从书，看风景之远方

<http://www.tsinghua.edu.cn>

### 算法

◆ 计算 = 信息处理  
通过某种工具，遵循一定规则，以精确而机械的形式进行

◆ 计算模型 = 计算机 + 信息处理工具

◆ 算法 = 即特定计算操作下，按照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，且在有限的时间内完成

有穷性 对于任何输入，经有限次基本操作，都可以得到输出

<http://www.tsinghua.edu.cn>

### 擒捉计算机及其算法

◆ 输入：任给圆周上的一点A

输出：经过点A的一条直线

◆ 算法 (2000 B.C., 古埃及人)

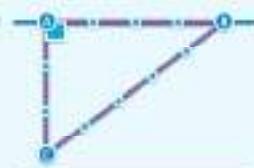
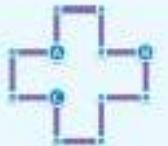
① 找两条长的线段，用圆规画成环

从圆心起，将圆周分成12等分并固定于圆

沿另一方向找到圆周相隔的端点C

移动点C，将剩余的1+5段继续伸直

◆ 这里的计算机是什么？



<http://www.tsinghua.edu.cn>

### 有穷性

◆ 定义  $Hailstone(n) = \begin{cases} \{1\} & n \leq 1 \\ \{n\} \cup Hailstone(\frac{n}{2}) & n \text{ 偶数} \\ \{n\} \cup Hailstone(3n+1) & n \text{ 奇数} \end{cases}$

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

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

$Hailstone(27) = \{27, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, \dots, 1\}$

<http://www.tsinghua.edu.cn>

### 有穷性

◆ `int hailstone( int n ) { //计算序列hailstone(n)的长度`

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

`while ( 1 < n ) { n = 3 * n + 1; length++; }`

`return length; //返回|hailstone(n)|`

`} //对于任意的n，总有|hailstone(n)| <= ?`

<http://www.tsinghua.edu.cn>

## 1. 绪论

### 计算 算法

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

- E. Dijkstra

邓伟峰  
<http://www.tsinghua.edu.cn>

## 1. 绪论

### 计算 优势

邓伟峰  
<http://www.tsinghua.edu.cn>

- 正向：符合语法，能够编译，能被正确处理简单的输入
- 能被正确处理大规模的输入
- 能被正确处理一般性的输入
- 能被正确处理退化的输入
- 能被正确处理任意合法的输入
- 健壮：能识别不合法的输入并做适当处理，而不能像正常退出
- 可读：结构化 + 准确命名 + 注释 + ...
- 效率：通常尽可能快；存储空间尽可能少

Algorithms + Data Structures = Programs  
(Algorithms + Data Structures) × Efficiency = Computation

- 两个主要方面 正确： 算法正确的与问题需求一致？  
数学证明？可不可以简单...  
成本： 执行时间。 所需存储空间  
如何度量？如何比较？
- 智慧： $T_A(n)$  = 对算法A解决特定问题的计算成本  
意义不大，毕竟... 可能出现的问题实例太多  
如何归结概念？
- 观察： 同类实例的规模，往往是决定计算成本的主要因素
- 速度： 程序接近，计算成本也接近  
规模扩大，计算成本亦上升

## 为什么要学？学什么？学习目标？

- 基础地位 在计算机科学专业课程体系中，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
- 课程范围 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
- 学习掌握结构，就是学会  
高效地利用计算机，有效地存储、组织、传输和转换数据  
掌握数据结构功能、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  
了解重要的数据结构应用的实用场景  
算法适用各种（基本）算法及到底的数据结构，解决实际问题

Data Structures & Algorithms, 1st edition, 2019

## 特定算法 · 不同实例

- 常数： $T_A(n)$  = 算法A解决某一问题规模为n的实例，所需的计算成本  
讨论特定算法A（及其对应的实例）时，可简记作 $T(n)$
- 然而： 逐一定义仍有问题...
- 规模： 同一问题规模的不同实例，计算成本不尽相同，甚至有实质差别...
- 例如： 在把平面上n个点，在它们生成的 $\binom{n}{3}$ 个三角形中，是否有某个的面积不超过1.07
- 能力： 最坏情况下高校使用有三角形；但运气好的话...
- 既如此，又该如何定义 $T(n)$ 呢？
- 分类起见，设 $T(n) = \max\{T(P) | |P| = n\}$
- 亦即，在规模同为n的所有实例中，只关注最坏（成本最高）者



## 内容纵览

- 数据结构用的ADT接口及其实现  
序列（向量、列表、栈、队列），树及搜索树（AVL树、伸展树、红黑树、B-树、kd-树）  
优先队列（堆）、字典（哈希表、链表）、图
- 构造有效算法模块的常用技巧  
顺序和二分查找、递归与排序、遍历  
模式匹配、枚举、几何查找
- 算法设计的经典策略与模式  
迭代、贪心、动态、分治、减治、试探-剪枝-回溯、动态规划
- 复杂度分析的基本方法  
渐进分析与相关记号  
递推关系、递归树  
分摊分析、归纳分析

Data Structures & Algorithms, 1st edition, 2019

## 特定问题 · 不同算法

- 同一问题通常有多种算法，如何评判哪个好？
- 实施统计是最直接的方法，但是以粗略反映算法的真正效率？不足够！  
不同的算法，可能更适合于不同的规模的输入
- 不同的算法，可能更适合于不同类型的输入
- 同一算法，可能由不同程序员，用不同语言实现，经不同编译器实现
- 同一算法，可能实现在运行于不同的体系结构、操作系统...
- 为此给出直观的评判，需要抽象出一个理想的平台或机器
- 不再依赖于上述种种具体的因素  
从而直接而准确地评估、调整并评价算法

To measure is to know.  
If you can not measure it,  
you can not improve it.  
— Lord Kelvin

## 1. 绪论

### 计算模型

### 统一尺度

单佳辉  
singlethink@sohu.com

## 1. 绪论

### 计算模型

### 图灵机

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

单佳辉  
singlethink@sohu.com

**Turing Machine**

- Tape 模拟物理纸带分为带单元格
- 带上存有某一字符，默认为'0'
- 空白带
- Alphabet 字符的种类有限
- Head 总是操作某一单元格，并可读取和改写其中的字符  
每经过一个节拍，可转向左侧或右侧的单元格
- State TM总是处于有限种状态中的某一种  
每经过一个节拍，可（按照规则）转向另一种状态

Java Simulation & Algorithms, Turing Machines

**Random Access Machine**

- 带存存储序编号，总数没有限制： $R[0], R[1], R[2], R[3], \dots$  //但带始址
- 通过索引，可以直接读取任意内存位置 //call-by-value
- 每一基本操作仅需常数时间 //循环及子程序本身非基本操作

$R[1] \leftarrow c$	$R[1] \leftarrow R[R[3]]$	$R[1] \leftarrow R[1] + R[2]$
$R[1] \leftarrow R[3]$	$R[R[1]] \leftarrow R[3]$	$R[1] \leftarrow R[1] - R[2]$
$\text{IF } R[1] = 0 \text{ GOTO 1}$	$\text{IF } R[1] > 0 \text{ GOTO 1}$	$\text{END 1}$
<b>STOP</b>		

Java Simulation & Algorithms, Random Access

**转换函数**

- Transition Function
- $(q, c; d, L/R, \sigma)$
- 起始状态为 $q_0$ ，起始读字符为 $c_0$ ，则
  - 将当前字符 $c$ 改写为 $d$
  - 转向左/右读移位
  - 插入 $\sigma$ 状态
- 特别地，一旦输入读到的状态“ $\overline{q}$ ”，则停机

Java Simulation & Algorithms, Turing Machines

**Random Access Machine**

- 与TM模型一样，RAM模型也是统一计算工具的简化与抽象  
使我们可以独立于具体的平台，对算法的效率做出可信的比较与评判
- 在这些模型中
  - 算法的运行时间 = 需要执行的基本操作次数
  - $T(n) =$  算法为求解规模为 $n$ 的问题，所需执行的基本操作次数

Java Simulation & Algorithms, Random Access

**Increase**

- 功能：将二进制串负整数加一
- 原理：
  - 全‘1’的细胞，翻转为全‘0’
  - 最低位的‘0’或‘1’翻转为‘1’
- $(s, -1, 0, l, r) //$ 左移， $-1 \rightarrow 0$   
 $(s, 0, 1, R, >) //$ 弹头， $0 \rightarrow 1$   
 $(s, R, 1, R, >) //$ ?  
 $(s, 0, 0, R, >) //$ 右移  
 $(s, 0, 0, l, R) //$ 翻位
- 题目 - 接口

Java Simulation & Algorithms, Random Access

**Floor Division**

- 对任意  $0 < c < d$  和  $0 < d$ ，计算
 
$$\begin{aligned} \lfloor c/d \rfloor &= \max\{x \mid dx \leq c\} \\ &= \max\{x \mid dx < 1 + c\} \end{aligned}$$
- 例如： $\lfloor 5/2 \rfloor = 2$        $\lfloor 2815/56 \rfloor = 50$   
 $\lfloor 6/3 \rfloor = 2$        $\lfloor 2816/56 \rfloor = 51$   
 $\lfloor 13/5 \rfloor = 2$
- 算法：最简单从  $R[0] = 1 + c$  中，减去  $R[1] = d$   
统计在下溢之前，所做减法的次数。

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

Java Simulation & Algorithms, Random Access

## 1. 绪论

计算机模型

RAM

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

单值算  
<http://neupg.ust.hk/~mcs/>

**Floor Division**

- 算法
  - $R[3] \leftarrow 1$  //increment
  - $R[0] \leftarrow R[0] + R[3]$  //add
  - $R[0] \leftarrow R[0] - R[1]$  //c  $\rightarrow$  d
  - $R[2] \leftarrow R[2] + R[3]$  //add
  - $\text{IF } R[0] > 0 \text{ GOTO 2}$  //if c > d, goto 2
  - $R[0] \leftarrow R[2] - R[3]$  //else x  $\leftarrow$  and
  - STOP** //return  $R[0] = x = a \lfloor c/d \rfloor$

Java Simulation & Algorithms, Random Access

**Floor Division**

- 算法
  - $R[3] \leftarrow 1$  //increment
  - $R[0] \leftarrow R[0] + R[3]$  //add
  - $R[0] \leftarrow R[0] - R[1]$  //c  $\rightarrow$  d
  - $R[2] \leftarrow R[2] + R[3]$  //add
  - $\text{IF } R[0] > 0 \text{ GOTO 2}$  //if c > d, goto 2
  - $R[0] \leftarrow R[2] - R[3]$  //else x  $\leftarrow$  and
  - STOP** //return  $R[0] = x = a \lfloor c/d \rfloor$

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

Java Simulation & Algorithms, Random Access

- 课后
- 随着问题实例规模增大，同一算法的求解时间可能波动甚至下降
  - 在渐进方面，现代电子计算机仍未达到RAM模型的要求？
  - 在T(n)、RAM模型中衡量算法效率，为何通常只需考虑运行时间？
  - 渐进分析法Increase中，以下这些操作可否忽略：  
-  $\Theta(n)$ ,  $\Omega(n)$ ,  $\mathcal{O}(n)$ ,  $\Omega(n^2)$
  - 设计一个函数f，实现对正整数的减一(Decrease)功能

Data Structures &amp; Algorithms, Lecture Notes 25

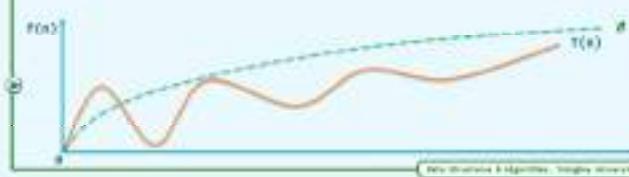
## 大O记号

+ big-O notation

// Paul Bachmann, 1894

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

$$\text{Ex: } \sqrt{2n \cdot [3n \cdot (n+2) + 4]} + 6 < \sqrt{2n \cdot [6n^2] + 4} + 6 < \sqrt{12n^3} + 6 < 6 \cdot n^{1.5} = \mathcal{O}(n^{1.5})$$



Data Structures &amp; Algorithms, Lecture Notes 29

## 1. 绪论

渐进分析  
大O记号

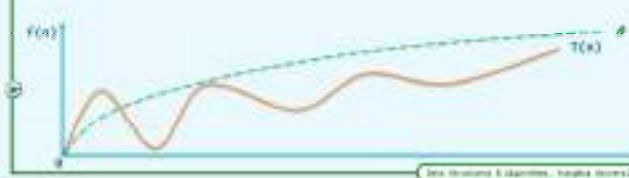
Mathematics is more in need  
of good notations than  
of new theorems.  
- A. Turing

邓伟群  
deng@tsinghua.edu.cn

Data Structures &amp; Algorithms, Lecture Notes 26

## 大O记号

+ 与T(n)相比，f(n)在形式上更为简洁，但依然反映前者增长趋势

常数级可忽略:  $\mathcal{O}[f(n)] = \mathcal{O}(c \cdot f(n))$ 低次级可忽略:  $\mathcal{O}(n^a + n^b) = \mathcal{O}(n^a), a \geq b > 0$ 

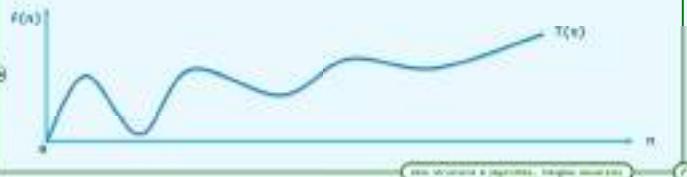
Data Structures &amp; Algorithms, Lecture Notes 30

渐进分析

+ 判断原元的问题：随着问题规模的增长，计算成本如何增长？

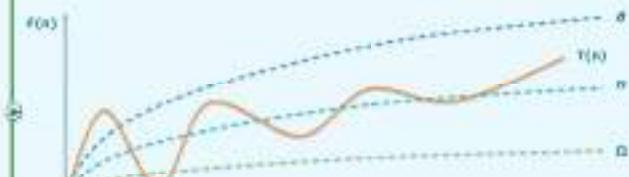
注意：这里更关心足够大的问题，注意考察成本的增长趋势

+ 在问题规模足够大时，计算成本如何增长？



Data Structures &amp; Algorithms, Lecture Notes 27

## 其它记号

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

Data Structures &amp; Algorithms, Lecture Notes 31

渐进分析

+ Asymptotic analysis

当  $n \rightarrow \infty$  时，对于规模为n插入，算法需执行的基本操作次数:  $\tau(n) + 2$ 需占用的存储单元数:  $s(n) + 2$ 

// 通常可不考虑，为什么？



Data Structures &amp; Algorithms, Lecture Notes 28

## 1. 绪论

渐进分析  
多项式

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

- A. Cobham &amp; J. Edwards

邓伟群  
deng@tsinghua.edu.cn

$\Theta(1)$

- 常数 (constant function) // 这段代码基本操作
  - $z = 2015 + 2015 + 2015 + \Theta(1)$ , 常数
  - $2015^{2015} = \Theta(1)$
- 最近距离，再大的常数，也要小于函数的参数
- [General twin prime conjecture, de Polignac 1849]
 

For every natural number  $k$ , there are infinitely many prime pairs  $p$  and  $q$  such that  $p - q = 2k$
- [Yitang Zhang, April 2013]  $k \leq 35,000,000$
- [Terence Tao, May 2013]  $k \leq 6,580,000$
- [Polymath Project, April 2014]  $k \leq 123$

1. 绪论  
渐进分析  
指教

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

- S. Cook

见注释  
[www.cse.lehigh.edu/~cse305/notes/np.pdf](http://www.cse.lehigh.edu/~cse305/notes/np.pdf)

$\Theta(1)$

- 该类算法的效率最高 // 越不能理解不切西能用
- 什么样的代码段对应于常数执行时间? // 应具体分析
  - 一定不会循环?
  - For ( $i = 0; i < n; i += n/2015 + 1$ );
  - For ( $i = 1; i < n; i += 1$ ); //  $\log n$ , 几乎常数
- 一定不含分支转向?
  - if ( $(n + m)^2 + (n + m) < 4 * n^2 + n$ ) goto UNREACHABLE; // 不考虑退出
  - 一定不能有 (语义) 错误?
  - if ( $2 == (n * n) \% n$ )



$\Theta(\log n)$

- 对数  $\Theta(\log)$  // 为何不理解常数?
  - $\ln n + \lg n + \log_{10} n + \log_{2} n = \Theta(\log n)$
- 对数无所谓
  - $\forall c > 0, \log_c n = \log_2 n / \log_2 c = \Theta(\log n)$
- 对数多项式 (poly-log function)
  - $123 * \log^{100} n + \log^{100}(n^2 + n + 1) = \Theta(\log^{100} n)$
- 该类算法非常有效，且必须无限接近于常数
  - $\forall c > 0, \log_c n = \Theta(n^c)$



$\Theta(n^r)$

- 多项式 (polynomial function)
  - $1000 + 300 = \Theta(n)$
  - $(100n + 500)(20n^2 + 300n + 2015) = \Theta(n \times n^3) = \Theta(n^4)$
  - $(2015n^2 + 2n)/(1235n + 1) = \Theta(n^2/n) = \Theta(n)$
- 一般地:  $a_0 n^r + a_1 n^{r-1} + \dots + a_r n + a_0 = \Theta(n^r), a_i > 0$
- 线性 (linear function): 所有  $\Theta(n)$  是函数
- 从  $\Theta(n)$  到  $\Theta(n^r)$ : 编程习惯主要提高的级别
  - 若  $\lfloor (n^{0.01} + 24n^{0.001})^{1.01} + 523n^{0.01} - 1978n^{0.001} \rfloor^{1.01} = \Theta(n^r)$
- 该类算法的效率通常认为已令人满意，然而...
  - 这个标准是否太高了?



1. 绪论  
渐进分析  
指教

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

- S. Cook

见注释  
[www.cse.lehigh.edu/~cse305/notes/np.pdf](http://www.cse.lehigh.edu/~cse305/notes/np.pdf)



### 2-Subset

- 问题...
- 总共有两位候选人
- 谁会赢得选举(269票)？

候选人	选票数
John Doe	120
Jane Smith	140
Mike Johnson	130
Sarah Lee	150
David Wilson	160
Emily Davis	170
Robert Green	180
Grace White	190
James Black	200
Sarah Lee	210
David Wilson	220
Emily Davis	230
Robert Green	240
Grace White	250
James Black	260
Sarah Lee	270
David Wilson	280
Emily Davis	290
Robert Green	300
Grace White	310
James Black	320
Sarah Lee	330
David Wilson	340
Emily Davis	350
Robert Green	360
Grace White	370
James Black	380
Sarah Lee	390
David Wilson	400
Emily Davis	410
Robert Green	420
Grace White	430
James Black	440
Sarah Lee	450
David Wilson	460
Emily Davis	470
Robert Green	480
Grace White	490
James Black	500

41

- 【直觉算法】
- 一枚枚举所有子集并统计其中元素总和



Data Structures &amp; Algorithms, Lecture 00000000

45

### 增长速度



Data Structures &amp; Algorithms, Lecture 00000000

### 2-Subset

• 定理:  $|2^n| = 2^{n+1} = 2^n$

- 例如: 直觉算法需要迭代 $2^n$ 轮, 并(在最坏情况下)至少需要花费这么多的时间
- 不甚理想!
- // 严格讲, 这只是程序, 而不是算法

• 还是直觉: 应该有更好的办法吧?

• 定理: 2-Subset is NP-complete

— 什么意思?

• 例如: 简单的计算模型而言, 不存在可在多项式时间内回答此问题的算法

— 就此定义而言, 上述的直觉算法已属拙劣

Data Structures &amp; Algorithms, Lecture 00000000

42

### 层次级别

时间复杂度	渐进表达式	常见算法	时间复杂度的基本操作
$O(1)$	常数级时间	遍历常数时间	常数级操作
$O(\log n)$	对数级时间	几乎任何操作	对数级操作
$O(n)$	线性级时间	线性时间	线性操作
$O(n \log n)$	线性对数级	几乎任何几乎线性操作	高级DP算法
$O(n^{\alpha})$	多项式级时间	几乎恒定性	多项式分解法
$O(n^2)$	平方级时间	矩阵乘法	矩阵乘法
$O(n^3)$	立方级时间	逆序对	矩阵乘法
$O(n^{\alpha})$ , $\alpha > 3$	多项式级时间	树状图	存在多项式解法的问题
$O(n^{\alpha})$ , $\alpha < 1$	指数级时间	更多时间的平凡操作, 再可忽略化	指数级时间, 但不存在解法

46

43

## 1. 绪论

### 渐进分析 复杂度层级

好读书, 不求甚解

每有会意, 便欣然忘食

邓佳辉

dengjiahui@zju.edu.cn

### 增长速度



Data Structures &amp; Algorithms, Lecture 00000000

44

## 1. 绪论

### 算法分析 进阶

相比较时间, 空间会更快耗去。

邓佳辉  
dengjiahui@zju.edu.cn

48

47

### 课后

+ 读物、推荐阅读: Fibonacci数  $F_{10}(n) = O(2^n)$

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

$$\log^2(n^{100} + 2^{100}n + 100) = O(?)$$

$$\log^2 n = O(n^2), \forall c > 0, d > 1$$

$$\log^{1.001} n = O(\log(n^{100}))$$

$$(n^2 + 1) / (2n + 3) = O(n)$$

$$n^{0.5} = O(n^2)$$

$$2^n = O(n!)$$

+ 2-Subset: 任给整数集S, 两两之间两两组成n个不交子集, 则解均为 $(2^n)/2$

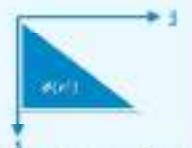
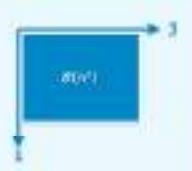
证明需要用到 $(k+1)-k\text{-subset}$ 的推导, 不要于k-Subset

+ Google: small-o notation

Data Structures &amp; Algorithms, Lecture 00000000

**算法分析**

- 两个主要任务：正确性（不变性  $\times$  单调性） $\times$  复杂度
- 为确保后者，直接需要将算法描述为HAN的基本指令，再统计累计的执行次数？不必！
- 从 $\leftrightarrow$ 等价地指出的基本指令，均等效于常数级cost的基本指令：在表达意义下，二者大体相当
- 分支转向：`goto` //循环的构造：出于结构化考虑，被隐含了
- 迭代循环：`for()`, `while()`, ... //本质上就是“if + goto”
- 通用 + 递归（自动展开）//本质上也是`goto`
- 复杂度分析的主要方法
  - 迭代：显式求和
  - 递归：递归树 + 递推方程
  - 插图 + 验证



**级数**

- 四边形数：与末项平方倒数  $T(n) = 1 + 2 + \dots + n = \binom{n+1}{2} = \frac{n(n+1)}{2} = O(n^2)$
- 五边形数：等差数列求和  $\sum_{k=1}^n k^2 \approx \int_0^n x^2 dx = \frac{x^{3/2}}{3/2} \Big|_0^n = \frac{n^{3/2}}{3/2} = O(n^{3/2})$
- $T_1(n) = \sum_{k=1}^n k^2 = 1^2 + 2^2 + 3^2 + \dots + n^2 = n(n+1)(2n+1)/6 = O(n^3)$
- $T_2(n) = \sum_{k=1}^n k^2 = 1^2 + 2^2 + 3^2 + \dots + n^2 = n^2(n+1)^2/4 = O(n^4)$
- $T_3(n) = \sum_{k=1}^n k^2 = 1^2 + 2^2 + 3^2 + \dots + n^2 = n(n+1)(2n+1)(3n^2+3n-1)/36 = O(n^5)$
- 几何级数：与末项商除
  - $T_4(n) = \sum_{k=1}^n k^3 = 1^3 + 2^3 + 3^3 + \dots + n^3 = \frac{n^3+1}{n+1} = O(n^3), \quad n < 0$
  - $T_5(n) = \sum_{k=0}^n 2^k = 1 + 2 + 4 + 8 + \dots + 2^n = 2^{n+1} - 1 = O(2^{n+1}) = O(2^n)$

**迭代 vs. 级数**

```

迭代
for (int i = 0; i < n; i++)
  for (int j = 0; j < i; j++)
    operation(i, j)

级数:

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


```



**级数**

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

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

$$\sum_{k=1}^{\infty} \frac{1}{k^2-1} = \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8} + \dots = 1 = O(1)$$
- 为什么讨论收敛级数？相信，基本操作次数、存储单元数可面前~~分段~~？是的，某种意义上！
- $(1-\lambda) \cdot [1 + 2\lambda + 3\lambda^2 + 4\lambda^3 + \dots] = 1/(1-\lambda) = O(1), \quad 0 < \lambda < 1$  //几何分布
- 可能未必收敛，然而长编有理
  - $b(n) = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n} = O(\log n)$  //调和级数
  - $\log n = \log 2 + \log 3 + \log 4 + \dots + \log n = \log(n!) = O(n \log n)$  //对数级数
- 有兴趣，不妨读读：Concrete Mathematics //见 2.2.2 Goldbach Theorem

**迭代 vs. 级数**

```

迭代
for (int i = 0; i < n; i++)
  for (int j = 0; j < i; j++)
    operation(i, j)

级数:

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

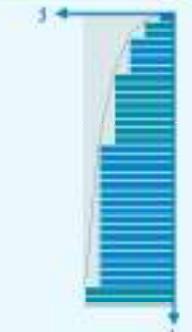

$$11 = 0, 1, 2, 3+4, 5+6, 7+8, \dots$$


$$= 0 + 0 + 1 + 2^2 + 3^2 + 4^2 + \dots$$


$$= T_{n-1, \log 2}(k=2^{k-1})$$


$$= O(\log n + 2^{n+1}) \quad (\text{CH page 833})$$


```



Go To Statement Considered Harmful.  
- E. Dijkstra, 1968

**1. 绪论**

"GOTO Considered Harmful" Considered Harmful.  
- F. Rubin, 1985

**算法分析**

"GOTO Considered Harmful" Considered Harmful.  
Considered Harmful.  
- Communications of the ACM, 1987

**迭代**

单链表  
[singlelinkedlist.com](http://singlelinkedlist.com)

**1. 绪论****算法分析****正确性**

Beware of bugs in the above code;

I have only proved it correct, not tried it.

- D. Knuth

单链表  
[singlelinkedlist.com](http://singlelinkedlist.com)

- 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，任意一对相邻元素的顺序。



- 过程交互：依次比较每一对相邻元素；如有必要，交换之。

- 若所有的操作都没有进行交换，则排序完成；否则，再做一趟的操作。

Java Standard & Advanced, Integer Sorter 6.0

- 最坏情况：插入数据及浮排列。

共 $n-1$ 趟归并交换。

每趟的效果，都等同于当前有效区间向左移一位。

最坏时间，得 $\Theta(n-1) = n(n-1)$ 次比较和 $(n-1)n/2$ 次移动。 $\Theta(n^2)$

$\text{avg} = 3 \times n(n-1)/2$

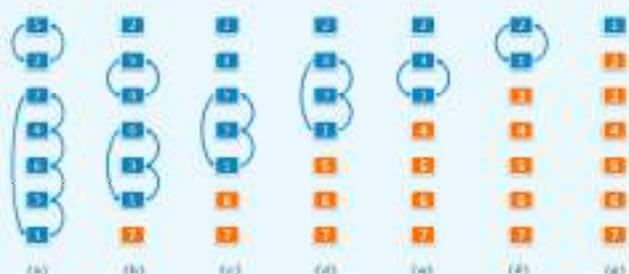
$T(n) = 4 \times n(n-1)/2 = \Theta(n^2)$

- 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： $T(n) = n-1 = \Omega(n)$

Java Standard & Advanced, Integer Sorter 6.0



Java Standard & Advanced, Integer Sorter 6.0

## 1. 绪论

算法分析

封闭估算

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

- Francois Arago

邓伟群

dengweiquan@ust.hk

```
void bubblesort(int A[], int n) { //第二步将进一步改进
    boolean sorted = false; [sorted = !sorted]; n-- ) //没有归并交换，每一趟...
    for (int i = 1; i < n; i++) //且在右，遍对数组A[1..n]内各相邻元素
        if (A[i - 1] > A[i]) { //若逆序，则
            swap(A[i - 1], A[i]); //令其互换，同时
            sorted = false; //消除(全局)排序标志
        }
    } //该算法...是否实现了所要的功能？必须由结束？且多圈迭代多少趟？
}
```

Java Standard & Advanced, Integer Sorter 6.0

## Back-Of-The-Envelope Calculation

今摩耶（孕妇）周长 =  $787 \times 399/7.2$

$$= 787 \times 58 = 45,358 \text{ km}$$

今3天 =  $360 \times 60 \text{min} \times 60 \text{sec}$

$$= 25 \times 4800 = 10^5 \text{ sec}$$



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



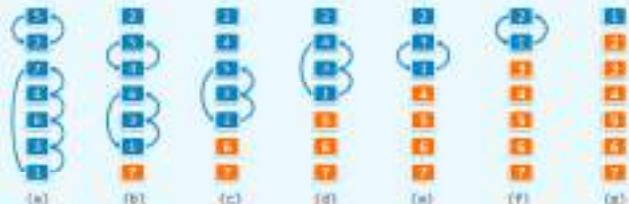
今“为祖国健康工作五十年” =  $1.6 \times 10^{10} \text{ sec}$

今“立生三世” =  $300 \text{ yr} = 10^{10} \text{ sec} = (1 \text{ geogel})^{1/(1/10)} \text{ sec}$

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

Java Standard & Advanced, Integer Sorter 6.0

- 不变性：经k趟归并操作后，最大的 $k+1$ 个元素必然是正确的。
- 单调性：经k趟归并操作后，数组中的值减至 $n-k$ 。
- 正确性：经整数n趟归并后，算法必然终止，因此输出正确解。



Java Standard & Advanced, Integer Sorter 6.0

## Back-Of-The-Envelope Calculation

今要算对全国人口

普查数据的排序

$n = 10^{9.9} \dots$

商速 $\sim$   
10^9 flops

天河1A  
千万亿次  
10^13 flops

bubblesort  
 $(10^{9.9})^2 / 10^{13}$

mergesort  
 $(10^{9.9}) \times \log(10^{9.9}) / 32 \times 10^9$

快  
慢

$10^{10} \text{ sec}$   
30 yr

$10^3 \text{ sec}$   
20 min

30 sec

$0.03 \text{ ms}$

Java Standard & Advanced, Integer Sorter 6.0

- 试按照“不变性+单调性”的模式，归纳证明本章包算法的正确性
- 试举例说明，`operation()`函数环障的限制可能有实际影响
- 学习不同开发环境提供的Profiler工具，并据此优化你的程序性能
- 习题[1-32]

Data Structures &amp; Algorithms, Python Version 2.1

```
def sum(A[], int n):
    return
    if n < 1:
        return A[n] + sum(A, n - 1)
    else:
        return A[n] + sum(A, n - 1)
```

## 通过跟踪分析

检查每个递归实例

统计所用时间（调用语句本身，计入对应的子实例）

算总和即算该执行时间

- 效率低，单个递归实例每次只用  $\Theta(1)$  时间
- $T(n) = \Theta(1) * (n + 1) = \Theta(n)$



## 1. 绪论

迭代与递归  
进而治之

迭代乃人工，递归乃神迹

To iterate is human,  
to recurse, divine.

郑佳辉

ding@tsinghua.edu.cn

+ 问题：计算任意n个整数之和

+ 实现：逐一取出每个元素，累加之

```
int Sum(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：

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

+ 你呢？

Data Structures &amp; Algorithms, Python Version 2.1

+ 任选数组A[0, n)，将其中的子区间A[lo, hi]翻转颠倒

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

+ if (lo &lt; 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 ); } // 递归版

+ next:

{ if (lo &lt; hi)
 { swap( A[lo], A[hi] ); lo++; hi--; goto next; } } // 迭代版

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

+ 为求解一个大规模的问题，可以

将问题分为两个子问题：再一平凡，另一规模减半

// 单调性

分别求解子问题

由子问题的解，得到原问题的解



Data Structures &amp; Algorithms, Python Version 2.1

凡治众如治寡，分崩离析

## 1. 绪论

## 迭代与递归

## 分而治之

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.

郑佳辉

ding@tsinghua.edu.cn

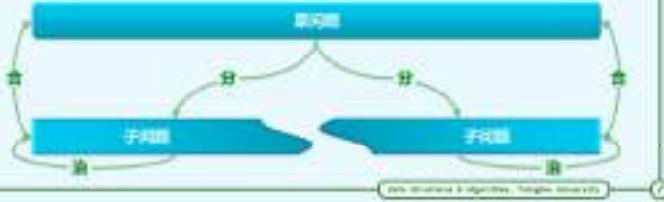
**Divide-and-conquer**

为求解一个大规模的问题，可以

将其划分若干（通常两个）子问题，规模大幅减小

分别求解子问题

由子问题的解，得到原问题的解

**1. 结论**

迭代与递归

Max2

堆栈

Data Structures & Algorithms, Trigoche Monika

**Binary Recursion**

```
+ void sum(int A[], int lo, int hi) { // 迭代求和sum(A[lo..hi])
    if (lo == hi) return A[lo];
    int mi = (lo + hi) / 2;
    return sum(A, lo, mi) + sum(A, mi + 1, hi); // 入口形式是sum(A, 0, n-1)
```

**迭代1**

从数组区间A[lo..hi]中找出最大的两个整数A[x1]和A[x2] // A[x1] > A[x2]

元素比较的次数，要求尽可能地少

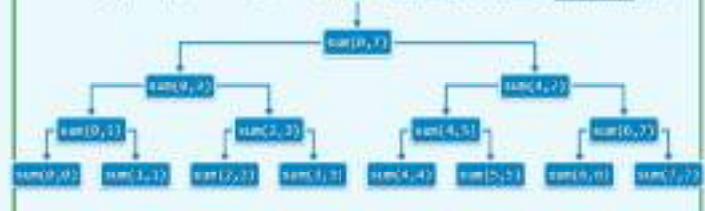


```
+ void max2(int A[], int lo, int hi, int &x1, int &x2) { // 1 <= lo <= hi <= n
    for (int i = lo, int j = lo + 1; i < hi; i++) // 扫描A[lo..hi]，提出A[i]
        if (A[i] > A[j]) x1 = i; // hi - lo - 1 = n - 1
    for (int i = lo, int j = lo + 1; i < hi; i++) // 扫描A[lo..hi]
        if (A[i] > A[j]) x2 = i; // hi - lo - 1
    for (int i = x1 + 1; i < hi; i++) // 扫描A[x1..hi]，提出A[i]
        if (A[i] > A[x1]) x2 = i; // hi - x1 - 1
    } // 无论如何，比较次数总是O((n-1))
```

Data Structures & Algorithms, Trigoche Monika

**Binary Recursion: Trace**

$T(n) = \Theta(\log n)$  // 退归跟据  
 $= \Theta(1) + (\Theta(2) + \Theta(2) + \dots + \Theta(2^{n-1}))$   
 $= \Theta(1) + (2^{n-1} - 1) = \Theta(n)$  // 更换底数，几项相加与最大/末项等价

**迭代1**

从数组区间A[lo..hi]中找出最大的两个整数x1和x2 // T(1) <= n-1

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

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

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

if (A[x1] < A[x2 + 1])

swap(x1, x2);



最好情况， $1 + (n-2)^{\lceil \frac{n}{2} \rceil} = n-1$

最坏情况， $1 + (n-2)^{\lceil \frac{n}{2} \rceil} = 3n-2$

比较次数是否进一步减少？分而治之！

Data Structures & Algorithms, Trigoche Monika

**Binary Recursion: Recurrence**

从递推的角度看，为求解sum(A, lo, hi)，需

- 递归调用sum(A, lo, mi)和sum(A, mi+1, hi)，进阶 //  $T(1)(n/2)$
- 将子问题的解累加 //  $\Theta(1)$

递推关系： $T(n) = 2T(n/2) + \Theta(1)$

$\Theta(1) = \Theta(1)$  // base: sum(A, k, k)

求解  $T(n) = 2^n T(n/2) + c_1$

$$T(n) + c_1 = 2^n (T(n/2) + c_1) = 2^n \cdot (T(n/4) + c_1) = \dots = 2^{\lfloor n/2 \rfloor} (T(1) + c_1) = n^{\lfloor n/2 \rfloor} (c_1 + c_2)$$

$$T(n) = (c_1 + c_2)n - c_1 = \Theta(n)$$

**递归 + 分治**

从数组区间A[lo..hi]中找出最大的两个整数x1和x2 // T(1) <= 2

if (hi - lo <= 1) { trivial(A, lo, hi, x1, x2); return; } // T(1) <= 2

int x1L, x2L; max2(A, lo, mi, x1L, x2L);

int x1R, x2R; max2(A, mi, hi, x1R, x2R);

if (A[x1L] > A[x1R]) {

x1 = x1L; x2 = A[x2L] > A[x2R] ? x2L : x2R;

} else {

x1 = x1R; x2 = A[x2L] > A[x2R] ? x2L : x2R;

} // 1 + 1 = 2

//  $T(n) = 2T(n/2) + 2 + 2n/3 - 2$ ；惯性数据结构，还可进一步优化（见10章）

Data Structures & Algorithms, Trigoche Monika

## Master Theorem

◆ [AMU-74], pg6, Theorem 2.1

Recurrence	Solution	Example
$T(n) = T(n-1) + 1$	$\Theta(n)$	向量求和之线性递归
$T(n) = T(n-1) + n$	$\Theta(n^2)$	列表或矩阵排序之线性递归
$T(n) = 2T(n-1) + 1$	$\Theta(2^n)$	Hanoi 塔, Fibonacci 数
$T(n) = 2T(n-1) + n$	$\Theta(2^n)$	
$T(n) = T(n/2) + 1$	$\Theta(\log n)$	向量的二分查找
$T(n) = T(n/2) + n$	$\Theta(n)$	列表的二分查找
$T(n) = 2T(n/2) + 1$	$\Theta(n)$	向量求和之二分递归
$T(n) = 2T(n/2) + n$	$\Theta(n \log n)$	归并排序

## 课后

◆ 在递归函数分析时，为什么递归树的节点数可以不统计？

◆ 使用递归树算法，分析  $f(n)$  二归并树的复杂度

通过递归树，解释该版本复杂度过高的原因

◆ 递归算法的空间复杂度，主要取决于什么因素？

◆ 本节给出求积问题的两个（线性和二分）递归算法

时间复杂度相同，空间呢？

## 1. 绪论

迭代与递归  
尾递归

邓佳群

dengj@tsinghua.edu.cn

## 1. 绪论

动态规划  
记忆法

请告诉我你是中国人。

告诉我，如何把记忆法写

请告诉我这个国家的伟大

轻轻的告诉你，不要喧哗

邓佳群

dengj@tsinghua.edu.cn

## Tail Recursion

◆ 尾递归法易于理解和实现，但空间（链型返回）效率低

在计算效率时，应将递归改写为等价的迭代形式

```

< fac(n) { return (1 > n) ? 1 : n * fac(n-1); }
< fac(n) {
    if (1 > n) return 1;
    else return n * fac(n-1); // tail recursion
}

```

◆ 尾递归：最后一步是递归调用

最简单的递归模式，可直接改写



## fib(): 递归

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

◆ int fib(n) // 为什这么慢？

**i** **return** (2 > n) ? n : fib(n-1) + fib(n-2); **1**◆ 复杂度：  $T(0) = T(1) = 1$        $T(n) = T(n-1) + T(n-2) + 1, \forall n > 1$     **令**  $S(n) = T(n) + 1/2$     **则**  $S(0) = 1 + T(0)$ ,     $S(1) = 1 + T(1)$     **加**  $S(n) = S(n-1) + S(n-2) = fib(n+1)$  $T(n) = 2 \cdot S(n) - 1 = 2 \cdot fib(n+1) - 1 = O(fib(n+1)) = O(n^n) = O(2^n)$ 其中  $\phi = (1 + \sqrt{5})/2 \approx 1.618$ 

## Tail Recursion

```

< fac(n) { /* 递归 */ > < fac(n) { /* 第一阶段为迭代 */ > < fac(n) { /* 简洁 */ >
    int f = 1; /* 记录子问题的解 */
    next: /* 拆分法，模拟递归调用
        if (1 > n) return f;
        else return n * fac(n-1);
        f *= n-1;
        goto next; /* 模拟递归调用 */
    } /* n > 1 */
    />n>1时同, <n>1同 > />n>1时同, <n>1同 > />n>1时同, <n>1同 >

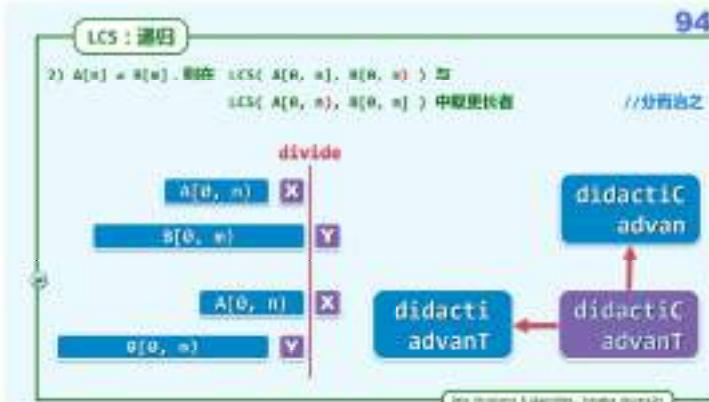
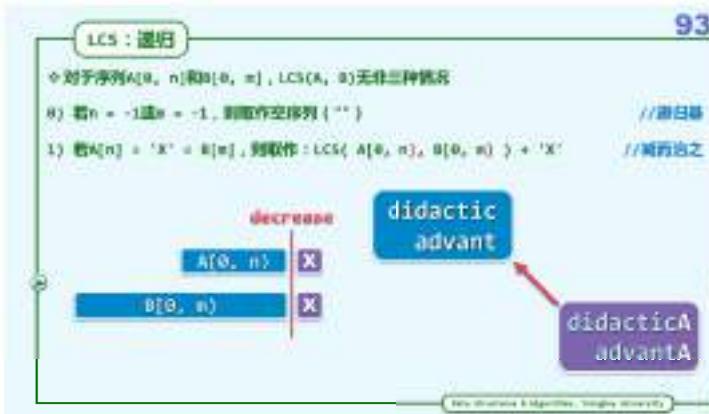
```

## 封闭估算

$\phi^{25} \approx 2^{75} \quad \phi^{43} \approx 2^{43} \approx 10^9 \text{ flo} = 1 \text{ sec}$

$\phi^3 \approx 10 \quad \phi^{42} \approx 10^{14} \text{ flo} = 10^6 \text{ sec} \approx 1 \text{ day}$

$\phi^{62} \approx 10^{18} \text{ flo} = 10^{18} \text{ sec} \approx 10^5 \text{ day} \approx 3 \text{ century}$



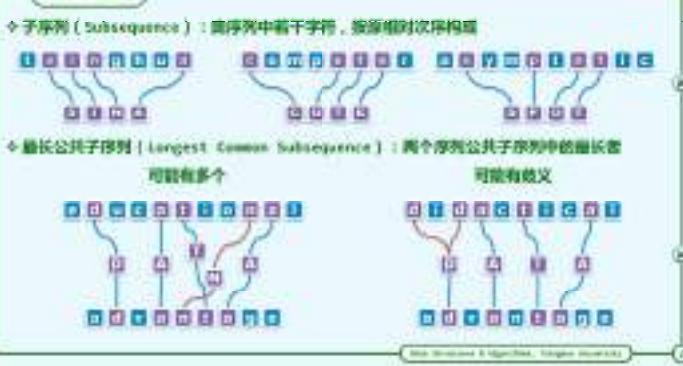
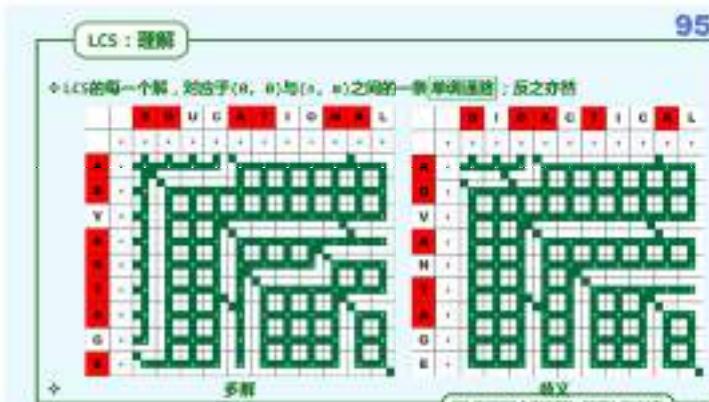
## 1. 绪论

### 动态规划

#### 最长公共子序列

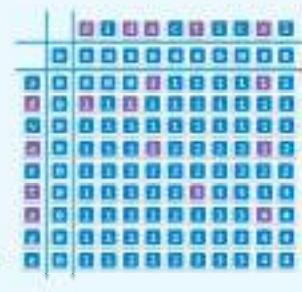
Make it work,  
make it right,  
make it fast.  
- Kent Beck

邓桂娟  
dengguijuan@csu.edu.cn



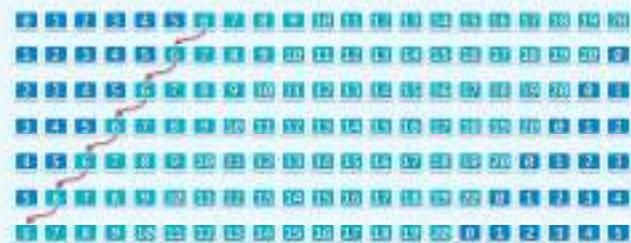
## LCS: 迭代

- 与fib()类似
- 该题也有大而复杂的递归实例（子问题）  
（最坏情况下）先地共计出现 $\Theta(2^n)$ 个
- 子问题，分别对应于A和B的某个前缀组合  
因此总共有 $\Theta(n^2)$ 个
- 采用动态规划的策略  
只要 $\Theta(n^2)$ 的矩阵可计算出所有子问题
- 为此，只需
  - 将所有子问题（假想地）列成一张表
  - 顺序计算方向，从 $LCS(A[0:n], B[0:m])$ 出发  
**依次**计算出所有项



## 暴力版

```
void shift1( int *A, int n, int k ) //假设以1为基准做k位左移
{ while ( n-- ) shift( A, n, B, k ); //共迭代n次, O(nk)
```



## 课后

- 复习：算法设计基础（第3版）之第11章（动态规划）
- 参考：Introduction to Algorithms, §15.1, §15.3, §15.4
- 本节所介绍的迭代版LCS算法，似乎需要记录每个子问题的解，从而导致空间复杂度激增。  
实际上，这既不现实，亦无必要。
- 试改进该算法，使得每个子问题只需常数空间，即可保证总的得到LCS的组成（而非仅仅长度）
- 考察序列 A = "Innovate" 和 B = "computer"

  - 它们的LCS是什么？
  - 该题的解是否唯一？是否有歧义？
  - 按照本节所给的算法，找出的解集中哪一个呢？

- 实现LCS算法的递归版和迭代版，并通过实测比较其运行时间
- 掌握imperialization策略，改进shift()与LCS算法的递归版

## 迭代版

```
int shift( int *A, int n, int k ) //n > k
{ int b = A[s]; int i = s, j = (s+k) % n; int mov = 0; //mov记录移位次数
  while ( i >= j ) //从A[s]出发，以k为周期，依次左移k位
    { A[i] = A[j]; i -= k; j += (j-k) % n; mov++; }
  A[i] = b; return mov + 1; //最后，初始元素转入到i的位置
```



$\Theta(n \cdot n)$ 由关于 $n$ 的 $\Theta(\text{GCD}(n, k))$ 个同余类组成 //包含 $n/k$ 个元素  
shift( $s, k$ )能够只扫描其中之一就位 //即 $n$ 所用的同余类

其他的同余类呢...

## 1. 绪论

规则  
缓存

不学诗，何以食  
不学礼，何以立

邓伟明  
dwm@ust.hk

## 迭代版

```
void shift1( int *A, int n, int k ) //很多轮迭代，实际数据循环左移n位，需 $\Theta(n)$ 
{ for ( int s = 0, mov = 0; mov < n; s++ ) //n/GCD(n, k) = O(n/k)/O(n, k)
  mov += shift( A, n, s, k );
}
```



## 循环位移

- //将数组 $A[0, n)$ 中元素向左循环移位k个单元
- void shift( int \*A, int n, int k )
- 比如：shift( A, 21, 4 );



## 倒置版

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



## 1. 绪论

尾图

字典

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

华生译

dengtingguoxiaoxia

常函数:  $\sigma(r)$ 

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

◆ 输入规模 =  $n$  的二进制位数 =  $r = \lceil \log_2(n+1) \rceil$ ◆ 复杂度主要取决于循环次数,  $T(r) = 1 + 1 + 4r + 1 = \sigma(r)$ ◆ 从  $O(2^r)$  到  $\sigma(r)$  的改进, 实际效果如何?幂函数:  $\theta(2^r)$ 

◆ 算法: 同一问题的不同算法, 整合在一起讨论

◆ 问题: 对任何给定的整数  $n > 0$ , 计算  $a^n$  ( $a$  为常数)◆ 常规实现:  $pow = 1; //O(1)$ 

```
while (n > 0) { //O(n)
    l pow *= a; n--;
}
```

◆ 复杂度与  $n$  成正比,  $T(n) = 1 + n + 1 = \theta(n)$ 

理性? 俗世!

◆ 所谓输入规模, 指的就是定义为可以描述输入所用的空间的量级

对于此类数值计算, 即使  $n$  的二进制位数  $r = \lceil \log_2(n+1) \rceil$ ◆ 复杂度与  $r$  成指数关系,  $T(r) = \theta(2^r) // 失去了$ 

## 辩论?

◆ 观察:  $power(a) = a^r$  的二进制展开宽度, 可以表示为  $\theta(r)$ ◆ 根据以上算法, 可在  $\theta(r = \log n)$  时间内计算出  $power(a)$ ◆ 然而, 循环逻辑直接打  $power(a)$ , 也至少需要  $\theta(n)$  时间

... 谁错了?

◆ 常规实现

▪ 均匀代价准则 (uniform cost criterion)

▪ 对数代价准则 (logarithmic cost criterion)

a<sup>10000</sup>

```
= a ^ [9 * 10^9 + 8 * 10^9 + 7 * 10^9 + 6 * 10^9 + 5 * 10^9]
= (a^10^9)^9 + (a^10^9)^8 + (a^10^9)^7 + (a^10^9)^6 + (a^10^9)^5
= pow( pow(a, 10^9), 9) + pow( pow( pow(a, 10^9), 10^9), 8)
= pow( pow( pow(a, 10^9), 10^9), 7) + pow( pow( pow(a, 10^9), 10^9), 6)
= pow( pow( pow(a, 10^9), 10^9), 5) + pow( pow(a, 10^9), 4)
= pow( pow( a, 10^9), 3)
```

◆ 能否在  $\theta(1)$  的时间内得到  $pow(a, n)$ ,  $0 \leq n \leq 10$ ◆ 对应于 (低) 向右 (高), 那个数位只影响  $\theta(1)$  的时间

## 1. 绪论

尾图

随机数

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

- A. Einstein

dengtingguoxiaoxia

a<sup>100000</sup>

```
= a ^ [1 * 2^9 + 0 * 2^8 + 1 * 2^7 + 1 * 2^6 + 0 * 2^5]
= (a^2^9)^1 + (a^2^8)^0 + (a^2^7)^1 + (a^2^6)^1 + (a^2^5)^0
= pow( pow(a, 2^9), 1) + pow( sqrt( pow(a, 2^8)), 0 )
= pow( pow(a, 2^7), 1) + pow( sqrt( pow(a, 2^6)), 1 )
= pow( pow(a, 2^1), 1) + pow( sqrt( pow(a, 2^0)), 1 )
= pow( pow(a, 2^0), 0)
```

$\theta(\text{pow}(x, 0)) = 1 // \theta(1)$   
 $\text{pow}(x, 1) = x // \theta(1)$   
 $\text{pow}(x, 2) = \text{sqrt}(x) // \theta(1)$

◆ 故对应于每个数位, 只需  $\theta(1)$  的时间!

## 随机数

◆ 将给一个数组  $A[0, n]$ , 随机地将其中元素的次序 **随机打乱**

```
// [(R. Fisher & F. Yates, 1938), [R. Durstenfeld, 1964], [D. E. Knuth, 1969]]
void shuffle( int A[], int n )
```

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



◆ 证明: 通过归档, 依次将各元素与随机选取的第一元素 (含自身) 交换

◆ 由此可以 **等概率** 地生成所有  $n!$  种排列?

dengtingguoxiaoxia

## 1. 绪论

### 下界 代数判定树

两个盗墓贼，又走了四五十里，却来到一市镇上。  
地名唤做扇子胡同，却是个三岔路口。宋江便问那店  
人道：“小人们欲投二龙山，请问路上，不知从那  
里能去？”

林冲醉

dongtinghuawu.com

## 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，能提高  
因此，首先应考虑最坏情况的算法 // worst-case optimal
- 排序所需的时间，主要取决于
 

关键码比较的次数 / # (key comparison)
元素交换的次数 / # (data swap)
- 就地 **In-place**：就地插入数据本身，只增加常数空间
- 稳定 **stability**：关键码相同的元素，在排序后相对位置保持

## 难度与下界

- 由前述实例可见，同一问题的不同算法，复杂度可能相差悬殊
- 在可解的前提下，可否谈论问题的难度？如何比较不同问题的难度？
- 如果“若存在算法，则所有算法中最低的复杂度称为”的难度
- 为什么是确定问题的难度？确定问题P，如何确定其难度？
- 两个方面着手：设计复杂度更低的算法 + 证明更高的无能算法下界
- 一旦算法的复杂度达到难度下界，则说明该算法已经是优
- 例如，排序问题下界为 $O(n \log n)$ ，而冒泡排序的...

114

Data Structures &amp; Algorithms, In-place Insertions

## 最坏情况最优 + 基于比较

- 排序算法，最快能快到多快？
  - 困难1：就是坏情况是优先的
  - 困难2：就算某一大类主流算法而言...
- 基于比较的算法（comparisons-based algorithm）
- 算法执行的进程，取决于一系列的比较（这里叫关键码）比对结果
 

比如，max()	和 bubbleSort()
----------	----------------
- 任何算法在最坏情况下，都需 $\Omega(n \log n)$ 时间才能完成排序

118

Data Structures &amp; Algorithms, In-place Insertions

## 排序

- 给定n个元素： $x_1, x_2, \dots, x_n$ ，对应关键码： $K_1, K_2, \dots, K_n$
- 需按某种次序排列 // i: 排序/全局
- 亦即，找出  $x_1, \dots, x_n$  的一个排列
 

$x_1, \dots, x_n$ 使得
$K_1 \leq K_2 \leq \dots \leq K_n$
- 例如： $\{3, 1, 4, 2, 5, 6, 7\}$   
 →  $\{1, 2, 3, 4, 5, 6, 7\}$
- 注意：此处的关键码需要是互异的 // 可能存在重复关键码
- 应用：25~300的数字都可以排序

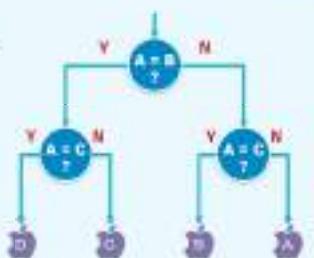
115

Data Structures &amp; Algorithms, In-place Insertions

## 判定树

- 每个TSA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，都对应于至少一四叶子（一条通往叶子的路径）
- 实例：经过2/4次决策  
 必可从A/16个苹果中  
 找出唯一的重的（不同重）
- 问题：决策次数可否减少？

119



## 算法分类

- 直接算法 直接移动元素本身 // 元素结构简单的通常采用
- 间接算法 下标 + 关键码 + 额外 // 元素结构复杂的通常采用
- 内部 / 外部 internal / external
- 脱机 / 在线 offline / online
- 串行 / 并行 sequential / parallel
- 确定性 / 随机 deterministic / randomized
- 基于比较式 / 散列式 comparison-based / hash-based

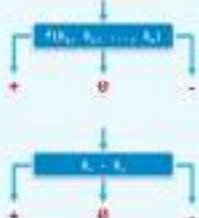
116

Data Structures &amp; Algorithms, In-place Insertions

## 代数判定树

- Algebraic Decision Tree
  - 针对“比较-判定”式算法的计算模型
  - 指定输入的规模，将所有可能的输入所对应的系列判断表示出来
- 代数判定：
  - 使用某一度数的代数多项式
  - 将任意一些关键码设为空值，对多项式求值
  - 根据结果的符号，确定算法执行方向
- Comparison Tree：最简单的ADT，二元一次多项式，用如： $x_1 = x_2$

120



Data Structures &amp; Algorithms, In-place Insertions

**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 //先, 中, 后
  - 从根节点到叶子的每一路径, 对应于算法的某次递归过程
  - 每层叶子对应一个输出 //在此, 指排好序的序列
  - 树高 = 最坏情况下所需的比较次数 //最坏情况复杂度
  - 树高的下界 = 所有树的时间复杂度下界
- 对  $n$  个元素进行排序的任务—堆排序 //时间至少为  $\Omega(n \log n)$ 
  - 叶子数  $\geq$  可能的输出 =  $n$  个元素可能的排列 =  $n!$
  - 树高  $\geq \log_2 n! = \log_2 n + O(\ln n) = \Omega(n \log n)$  //Stirling approximation
- 上述结论, 可进一步推广至概率平均情况。随机情况 (概率  $\geq 25\%$ )



- 热力学第二定律: 能量不会自发地从低温物体传向高温物体
- Shannon: 信息系统中蕴含的信息量, 可达 **信息熵**  $H$

$$H(S) = -\sum p_i \log_2 p_i \quad (H = S \text{ 可能的状态总数})$$

热系统的熵减少, 需要付出一定的能量

数据系统的 **信息熵** 减少, 也需要付出一定的计算量



不怕不识货, 就怕货比货

## 1. 绪论

### 下界 归约

郑伟群

zengwq@zjhu.edu.cn

### 熵与下界

#### Landauer's principle

信息的减少或丢失, 必伴随着熵的增加, 并放出相等的热量

就像耗功和耗能, AND 和 OR 逻辑门必劣于 NOT 门

◆ “幸福的人都是相似的, 不幸福的人却各有各的不幸”

从计算的角度看, 幸福的可能状态, 要远少于不幸

丽丽的痛苦小于丽丽, 故从后者丽丽者, 需要付出巨大的努力



Data Structures & Algorithms - Integer Sorting

### 属性归约

◆ 除了 (代数) 重归约, 俗叫 **reduction**, 也是确定下界的有力工具

$\Theta(n \log n)$ : linear-time reduction

NP-complete/P: polynomial-time reduction

P-SPACE complete: polynomial-time many-one reduction



## 2. 向量

### 抽象数据类型 接口与实现

郑伟群

zengwq@zjhu.edu.cn

### 实例

- 【Red-Green Matching】平面上任给  $n$  个红色点和  $n$  个蓝色点, 如何以当不相交的线段配对联接  
Sorting  $\leq_{\sim}$  Red-Green Matching
- 【Element Uniqueness】任给  $n$  个整数中, 是否包含重复? //下界为  $\Omega(n \log n)$   
EU  $\leq_{\sim}$  Closest Pair
- 【Integer Element Uniqueness】任给  $n$  个整数中, 是否包含相同? //下界亦是  $\Omega(n \log n)$   
IEU  $\leq_{\sim}$  Segment Intersection Detection
- 【Set Disjointness】任给一对集合  $A$  和  $B$ , 是否存在公共元素? //下界亦是  $\Omega(n \log n)$   
SD  $\leq_{\sim}$  Diameter

### Abstract Data Type vs. Data Structure

◆ 抽象数据类型 = 数据模型 + 定义在该模型上的一组操作

抽象定义      外部的逻辑特性

一种定义      不考虑实现细节

操作API

不涉及数据的存储方式

数据结构 = 基于某种物理组织, 实现ADT的一组操作方法

具体实现      内部的表示与实现

多种实现      与复杂度密切相关

高效的算法

要考虑数据的物理存储结构



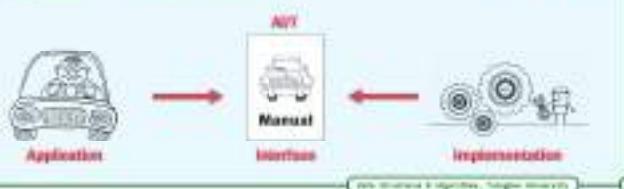
Data Structures & Algorithms - Integer Sorting

## Application = Interface x Implementation

在数据结构的~~具体实现~~与~~实现应用~~之间，ADT就分工与接口划出了统一的界面

**实现：**高效率地完成数据结构的ADT接口操作 //装水箱、进汽车

**应用：**便捷地通过操作接口使用数据结构 //喝水箱、开汽车



## 向量ADT接口

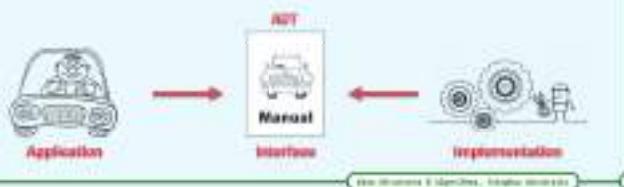
操作	功能	适用对象
size()	报告向量当前的规模(元素总数)	向量
get(r)	根据索引为r的元素	向量
put(r, e)	用e替换索引为r的元素值	向量
insert(r, e)	e作为秩为r的新插入，既插既改次而移	向量
remove(r)	删除秩为r的元素，返回该元素值	向量
disordered()	判断所有元素是否已按非降序排列	向量
sort()	调整各元素的位置，使之按非降序排列	向量
find(e)	查找目标元素e	向量
search(e)	查找e，返回不大于e且秩最大的元素	有序向量
deduplicate(), unique()	剔除重复元素	向量/有序向量
traverse()	遍历向量并统一处理所有元素	向量

## Application = Interface x Implementation

按照ADT设计：高级算法设计者与底层数据结构实现者可高效地分工协作

不同的算法与数据结构可以任意组合，便于构建最优解

每种操作接口只需一堆实现一次，代码复用增加，软件重用范围广



## 2. 向量

抽象数据类型  
从数组到向量

## 标准库

<http://cpluslang.com>

在C/C++语言中，数组A[]中的元素与[0, n)的索引一一对应

A[0], A[1], A[2], ..., A[n-1]

A[] 可以看作一个线性表

反之，每个元素均由~~其值~~偏移唯一标识，并可直接访问  
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)一一对应  
typedef int Rank; //函数访问 (call-by-rank)  
操作：管理维护更加规范化，统一与安全  
元素类型可灵活选取，便于定制和杂糅结构 //Vector<PFCTree\*> pfcForest;

## 2. 向量

抽象数据类型  
模板类

## 标准库

<http://cpluslang.com>

template<typename T> class Vector { //向量指代类

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

protected:

/\* ... 内部函数 \*/

public:

/\* ... 构造函数 \*/

/\* ... 拷贝函数 \*/

/\* ... 只读接口 \*/

/\* ... 可写接口 \*/

/\* ... 遍历接口 \*/

Vector

vector size capacity

interface

vector created insert remove ... traverse

applications

## 构造 + 析构

```

#define DEFAULT_CAPACITY 3 //默认起始容量(实际应用中可设置为更大)
vector<int> c = DEFAULT_CAPACITY;
    _elem = new T[_capacity <= c]; _size = 0; //默认
+vector<T const * A, Rank lo, Rank hi>; //数据区间复制
{ copyFrom(A, lo, hi); }
vector<Vector<T> const& V, Rank lo, Rank hi>;
{ copyFrom(V._elem, lo, hi); } //向量区间复制
vector<Vector<T> const& V>;
{ copyFrom(V._elem, 0, V._size); } //向量整体复制
-vector() { delete [] _elem; } //释放内部空间

```

Data Structures &amp; Algorithms, Highly Interactive

## 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; //根据清零
    while (lo < hi) //A[lo, hi)内的元素逐一
        _elem[_size++] = A[lo++]; //调用基类 elem[0, hi - lo]
} //O(hi - lo) = O(n)
    _elem [lo] [hi - lo] [hi]
    A[lo] [hi - lo] [hi]
    copy

```

Data Structures &amp; Algorithms, Highly Interactive

## 2. 向量

可扩充向量  
语法

空间要动态，不要假设所用  
量点来排队，而到你没座位

邓伟群  
dengwq@zjhu.edu.cn

## 静态空间管理

\_elem [size] [capacity]

开辟内部数组 \_elem[] 并使用一段地址连续的物理空间

\_capacity : 总容量

\_size : 当前的实际规模

若采用静态空间管理策略，将用\_capacity固定，则有明显的不足...

\_elem [size] [capacity]

开辟内部数组 \_elem[] 并使用一段地址连续的物理空间

\_capacity : 总容量

\_size : 当前的实际规模

若采用静态空间管理策略，将用\_capacity固定，则有明显的不足...

Data Structures &amp; Algorithms, Highly Interactive

## 空间效率

\_elem [size] [capacity]

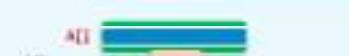
- 上溢(overflow) : \_elem[]不足以存放所有元素，尽管此时系统往往仍有足够的空间
  - 下溢(underflow) : \_elem[]中的元素寥寥无几
  - 调整因子(\_load\_factor) : 1. = \_size/\_capacity <= 0.95
- 更糟糕的是，一般的应用环境中难以准确预测空间的需求量
- 可否使得向量可随实际需求动态调整容量，并同时保证高效率？

Data Structures &amp; Algorithms, Highly Interactive

## 动态空间管理

## ① 基本型：

- 身体每经过一段时间的生长
- 以路无法为外壳容纳
- 需要丢弃的外壳，代之以...



## ② 在动态发生上溢时

## 适当扩大内部数据的容量



Data Structures &amp; Algorithms, Highly Interactive

## 扩容算法

```

template <typename T> void vector<T>::expand() { //向量空间不足时扩容
    if (_size < _capacity) return; //尚未满员时，不必扩容
    _capacity = max(_capacity, DEFAULT_CAPACITY); //不低于最小容量
    T* oldElem = _elem; _elem = new T[_capacity <= 1]; //智能指针
    for (int i = 0; i < _size; i++) //遍历原向量内容
        _elem[i] = oldElem[i]; //T为基本类型，或已重载赋值操作符“=”
    delete [] oldElem; //释放旧空间
} //得益于壳层的封装，尽管扩容之后数据的物理地址有所改变，却不必出现野指针

```

为什么必须采用智能指针策略呢？其它策略是否可行？

Data Structures &amp; Algorithms, Highly Interactive

## 2. 向量

可扩充向量  
分配

...在他的心理上，他总以为北平是天底下最可爱的大城，不能有什么灾难，到三个月必定完满辉煌，西西诸事大吉。北平的灾难恰似一个人免不了有些头痛脚痛，过几天你就会好了的。

邓伟群  
dengwq@zjhu.edu.cn

2. 向量  
可扩充向量  
分配

## 容量递增策略

```

 $\diamond \text{if } \text{oldSize} = \text{_elems}; \text{else } \text{newT}[ \text{_capacity} == \text{INCREMENT} ] \text{; } // \text{溢出预定增量}$ 
 $\diamond \text{最坏情况: 在初始容量为 } n \text{ 的空间里, 连续插入 } n + m^2 + 2 \text{ 个元素...}$ 
 $\diamond \text{于是, 在第 } 1, 2 + 1, 2\cdot 2 + 1, 2\cdot 2\cdot 2 + 1, \dots \text{ 次插入时, 都需扩容}$ 
 $\diamond \text{即每次扩容中语句的操作, 每次扩容过程中插入的代价成半径级为 } O(1) \text{ 的木箱数}$ 
 $\diamond \text{总代价 } = 1 + (n-1) * m/2 = \Theta(n^2), \text{ 每次扩容的分摊成本为 } \Theta(n)$ 


```

data structures & algorithms, taught interactively

## 2. 向量

无序向量  
基本操作

它构造, 它更新, 它表征, 它迭代, 它杂乱,  
它容纳, 它可靠, 它矩阵大的最初的北平

zhengtongguo@163.com

用起来

## 容量加倍策略

```

 $\diamond \text{if } \text{oldSize} = \text{_elems}; \text{else } \text{newT}[ \text{_capacity} <= 1 ] \text{; } // \text{容量加倍}$ 
 $\diamond \text{最坏情况: 在初始容量为 } 1 \text{ 的空间里, 连续插入 } n + 2^m + 2 \text{ 个元素...}$ 
 $\diamond \text{于是, 在第 } 1, 2, 4, 8, 16, \dots \text{ 次插入时都需扩容}$ 
 $\diamond \text{首次扩容过程中插入的向量的对角成本依次为 } O(1), O(2), O(4), O(8), \dots, O(2^m) \text{ 等}$ 
 $\diamond \text{总代价 } = \Theta(n), \text{ 每次扩容的分摊成本为 } \Theta(1)$ 


```

data structures & algorithms, taught interactively

## 元素访问

```

 $\diamond \text{似乎不是问题: 通过 } \text{v.get}(r) \text{ 和 } \text{v.set}(r, c) \text{ 接口, 当然可以做。与向量元素}$ 
 $\diamond \text{相似: 便携性而言, 还不如数组元素的访问方式: } \text{A}[r] \text{ } // \text{可直接用下标的访问方式?}$ 
 $\diamond \text{可以! 为此, 向量提供下标操作符 “[ ]”}$ 

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

 $\diamond \text{此后, 对外的 } \text{V}[r] \text{ 语义对应于内部的 } \text{V._elems}[r]$ 

右值:  $T \times = V[r] + i[n] * M[k]$   
左值:  $V[r] = (T)(2^k + 3)$

 $\diamond \text{为便于讲解, 这里采用了显式的处理意外和错误 (比如, 入口参数越界等)}$ 

```

data structures & algorithms, taught interactively

## 对比



## 平均分析 vs. 分摊分析

◆ 平均复杂度 / 预期复杂度 (average/expected complexity)  
根据数据结构各种操作出现概率的分布, 将对内的成本加权平均  
各种可能的操作, 作为独立事件分别考虑  
影响了操作之间的相关性和依赖性  
体谅不能准确地评判数据结构和算法的简洁性

◆ 分摊复杂度 (amortized complexity)  
对数据结构连续地实施过多操作, 所谓总体成本分摊至单一操作  
从实际可行的角度, 对一系列操作做整体的考量  
更加准确地刻画了可能出现的操作序列  
更大程度地利用数据结构和算法的简洁性

◆ 读者将看到更多, 更复杂的例子

data structures & algorithms, taught interactively

## 插入

```

 $\diamond \text{template <typename T> // } r \text{ 作为终点元素插入, } n \leftarrow r \leftarrow \text{size}$ 
 $\diamond \text{Rank Vector<T>::insert( Rank r, T const \& v ) \{ } / \Theta(n - r)$ 
 $\quad \text{expand(); // 若有必要, 扩容}$ 
 $\quad \text{for ( int i = \_size; i > r; i-- ) // 右移向右$ 
 $\quad \quad \text{_elems[i]} = \text{_elems}[i-1]; // 将原元素依次向右一个单元}$ 
 $\quad \quad \text{_elems[r]} = \text{v}; \text{size}++; \text{return r;} // 插入新元素, 更新容量, 返回秩}$ 
 $\} \quad \text{(a) } [ \text{v}, \text{v} + \text{size} - 1 ] \rightarrow \text{[v, n]}$ 
 $\quad \quad \text{(b) } [ \text{v}, \text{v} + \text{size} - 1 ] \rightarrow [ \text{v}, \text{v} + \text{size} ] \text{ expanded if necessary}$ 
 $\quad \quad \text{(c) } [ \text{v}, \text{v} + \text{size} ] \rightarrow [ \text{v}, \text{v} + \text{size} ]$ 
 $\quad \quad \text{(d) } [ \text{v}, \text{v} + \text{size} ] \rightarrow [ \text{v}, \text{v} + \text{size} ] \text{ right shift}$ 

```

data structures & algorithms, taught interactively

## 区间删除

```

 $\diamond \text{template <typename T> // 删除区间 } [\text{lo}, \text{hi}], 0 \leq \text{lo} \leq \text{hi} < \text{size}$ 
 $\diamond \text{int Vector<T>::remove( Rank lo, Rank hi ) \{ } / \Theta(n - h)$ 
 $\quad \text{if ( lo == hi ) return 0; // 出于效率考虑, 单独处理退化情况}$ 
 $\quad \text{while ( hi < \_size ), \_elems[lo++], \_elems[hi++]; } // [\text{hi}, \text{size})$ 
 $\quad \text{\_size} = \text{lo}; \text{shrink(); // 因此很慢, 若有必要则稍等}$ 
 $\quad \text{return hi - lo; // 返回被删除元素的数目}$ 
 $\} \quad \text{(a) } [ \text{v}, \text{v} + \text{size} ] \rightarrow [ \text{v}, \text{v} + \text{size} ] \text{ free}$ 
 $\quad \quad \text{(b) } [ \text{v}, \text{v} + \text{size} ] \rightarrow [ \text{v}, \text{v} + \text{size} ] \rightarrow [ \text{v}, \text{v} + \text{size} ] \text{ left shift}$ 
 $\quad \quad \text{(c) } [ \text{v}, \text{v} + \text{size} ] \rightarrow [ \text{v}, \text{v} + \text{size} ] \rightarrow [ \text{v}, \text{v} + \text{size} ] \text{ shrink if necessary}$ 
 $\quad \quad \text{(d) } [ \text{v}, \text{v} + \text{size} ] \rightarrow [ \text{v}, \text{v} + \text{size} ] \text{ free}$ 

```

data structures & algorithms, taught interactively

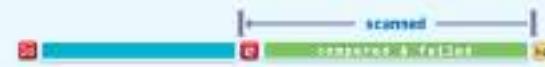
◆ 可以操作区间删除操作的特例： $[r] = [r, r + 1]$

```
+template <typename T> //删除向量中秩为1的元素， $0 \leq r \leq \text{size}$ 
T Vector<T>::remove( Rank r ) { // $\delta(n - r)$ 
    T e = _elem[r]; //暴力删除元素
    remove( r, r + 1 ); //调用区间删除算法
    return e; //返回被删除元素
}

◆ 反过来，基于remove(r)接口，通过反向的调用，实现remove(lo, hi)呢？
+每次循环耗时正比于删除区间的后缀长度。 $n - hi + lo = \delta(n)$ 
而循环次数等于区间距离。 $hi - lo = \theta(n)$ 
如此，将导致总体 $\Theta(n^2)$ 的时间度
```

+ template <typename T> //在向中多个元素时可调用`const`

```
Rank Vector<T>::const( Rank lo, Rank hi ) const { // $0 \leq lo \leq hi \leq \text{size}$ 
    while ( lo < hi-- ) if ( e[lo] == _elem[hi] ) ; //暴力查找
    return hi; //hi < lo意味着失败：否则hi指向向中元素的秩
} //Excel::match(e, range, type)
```



+ 输入敏感 (input-sensitive)：最好的 $\Theta(1)$ ，最坏 $\Theta(n)$

## 2. 向量

无序向量  
查找

他便站将起来，背着手踱来踱去，同时把那些人逐个个地招来，向中一个果然将领上挂着一寸来长粗的铁头。

郑佳群  
[zhangj@tsinghua.edu.cn](mailto:zhangj@tsinghua.edu.cn)

## 2. 向量

(c3) 无序向量  
· 去重

郑佳群

世面同而故号，何留壁之琳琳！

[zhangj@tsinghua.edu.cn](mailto:zhangj@tsinghua.edu.cn)

```
+template <typename K, typename V> struct Entry { //判断相等类
    K key; V value; //关键码、数据
    Entry( K k = K(), V v = V() ) : key(k), value(v) {} //默认构造函数
    Entry( Entry& e, const& e ) : key(e.key), value(e.value) {} //拷贝
    bool operator== ( Entry& e, const& e ) { return key == e.key; } //等于
    bool operator!= ( Entry& e, const& e ) { return key != e.key; } //不等
    ... ...
}
```

+ 应用实例：网格搜索的局部结果经过去重操作，汇总为最终报告

```
+template <typename T> //删除重复元素，调用顺序族向量成员
int Vector<T>::deduplicate() { //去重版 + 指针版
    int oldSize = _size; //记录原规模
    Rank l = 1; //从_elem[1]开始
    while ( l < _size ) //遍历向量逐一考察各元素_elem[l]
        find( _elem[l], 0, l ) < 0 ? //在前面寻找相同的
            l++ //若无相同则继续考察到尾端
        : remove(l); //否则删除该元素（且移一个？！）
    return oldSize - _size; //衡量规模变化量，即删除元素总数
}
```

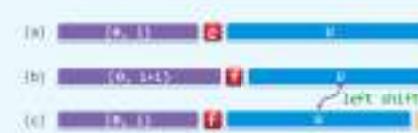
```
+template <typename K, typename V> struct Entry { //自然相等类
    K key; V value; //关键码、数据
    Entry( K k = K(), V v = V() ) : key(k), value(v) {} //默认构造函数
    Entry( Entry& e, const& e ) : key(e.key), value(e.value) {} //拷贝
    bool operator== ( Entry& e, const& e ) { return key == e.key; } //等于
    bool operator!= ( Entry& e, const& e ) { return key != e.key; } //不等
    bool operator< ( Entry& e, const& e ) { return key < e.key; } //小于
    bool operator> ( Entry& e, const& e ) { return key > e.key; } //大于
}; //得益于比较器和判等器，从此往后，不必严格区分谓词及其对应的关键词
```

+ 稳定：凡被删除者均为重复元素（不多）

故只需证明：算法不忽略重复元素（不少）

+ 不变性：在当前元素 $v[i]$ 的前缀 $v[0:i]$ 中，各元素彼此无序

+ 初始 $i = 1$ 时自然成立；后续的一般情况...





**复杂度**

- 遍历时间主要取决于while循环次数  
共计  $\sim n - 1$
- 在循环情况下，每次都需要调用remove()
  - 需耗时  $\delta(n-1) \sim \delta(1)$
  - 累计  $\delta(n^2)$
  - 尽管省去find()，总体尚坏
  - 无序向量的deduplicate()相同
- 反思：低效的原因在于，同一元素可作为被删除元素的后继多次调用
- 启示：若能以重叠区间为单位，减少对除重元素，性能必将改进...



## 2. 向量

有序向量

## 二分查找（版本A）

向山进发之后，达山被二郎菩萨点上火，烧尽了大木。他们继续赶路，钻进洞内，身子热得极，烧了自己，又害人灭洞，出来时，又因托尼作祟，想以种籽，刚外又害了一半。他们这一身，跟着的住在山中，这两年，又施过打雷的烧一半天。

林伟群

www.englishbooks.org

**高效算法**

```
#template <typename T> int Vector<T>::uniqueify() {
    Rank i = 0, j = 0; //暂对后界“相邻”元素的秩
    while (i < j + _size) //逐一扫描，直至末元素
    //跳过前两者：若树不同元素时，向后移至适于插入右侧
        if (_elem[i] != _elem[j]) _elem[i+1] = _elem[j];
    _size = ++i; shrink(); //直接裁掉尾部多余元素
    return j + 1; //内部操作优化，直接忽略元素总数
} //注意：通过remove(la, hi)批量删除，依然不能达到高效率
```

move forward

Data Structures &amp; Algorithms, English Version 2e

**实例与复杂度**

- 共  $i = 1$  次迭代，每次常数时间，累计  $\delta(n)$  时间



Data Structures &amp; Algorithms, English Version 2e

**课后**

- 较之无序向量，有序向量的唯一化可以更快速完成  
其中的原因，如何理解并解释？

Data Structures &amp; Algorithms, English Version 2e

## 统一接口

```
#template <typename T> //直接映射统一接口，o <- lo < hi < _size
Rank vector<T>::search(T const & e, Rank lo, Rank hi) const {
    return (rank() >= 2) ? //适合set的基本情况适用
        binSearch(_size, e, lo, hi) : //二分查找算法，或者
        fibSearch(_size, e, lo, hi); //Fibonacci查找算法
}
```



Data Structures &amp; Algorithms, English Version 2e

## 避而远之

从任意一元素  $x = S[mi]$  为界，即可将待查找区间分为三部分 //  $S[mi]$  为操作对象

→ 整体向量整体排序，则必有：

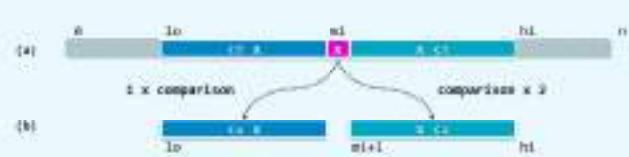
 $S[lo, mi] \ll S[mi] \ll S[mi, hi]$ 

Data Structures &amp; Algorithms, English Version 2e

## 避而远之

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

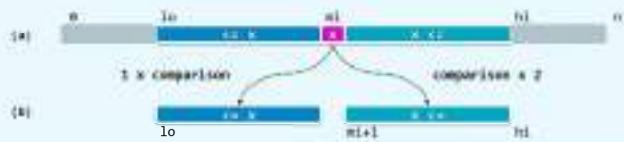
- $e < x$ : 则  $e$  必须属于左子区间  $S[lo, mi]$ ，故可递归深入
- $e > x$ : 则  $e$  必须属于右子区间  $S[mi, hi]$ ，亦可递归深入
- $e = x$ : 已在此处命中，可随即返回 // 若向多个，返回何者？



Data Structures &amp; Algorithms, English Version 2e

若枢点mid取作中点，则可经过至多两次比较

- 或者能解命中
- 或者将问题归结为一半

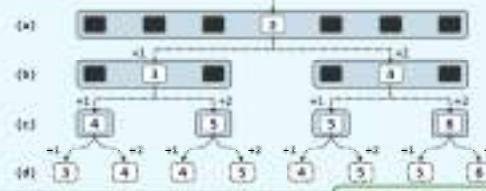


当  $n = 7$  时，各元素对应的成功查找长度为  $\{4, 3, 1, 2, 5, 4, 0\}$

在等概率情况下，平均成功查找长度  $= 29 / 7 = 4.14$

在共8种失败情况，查找长度分别为  $\{3, 4, 4, 5, 4, 3, 3, 6\}$

在等概率情况下，平均失败查找长度  $= 36 / 8 = 4.50$



完整代码实现  $T(n)$  在有序数组区间  $[lo, hi]$  内查找元素。

```
static Rank binSearch(T *A, T count, int lo, Rank lk, Rank rk) {
    while (lo < hi) { // 每步迭代可能要进行两次比较判断，有三个分支
        Rank mid = (lo + hi) / 2; // 取中点成为枢点
        if (A[mid] < lk) lk = mid; // 增入左半段 [lo, mid] 调整枢点
        else if (A[mid] > rk) hi = mid + 1; // 增入右半段 (mid, hi)
        else return mid; // 在 mid 处命中
    }
    return -1; // 找找失败
}
```



若待查找结果出现的概率不均衡时，查找长度应该如何定价计算？

$\oplus \text{S.search}(3, 0, 7)$ : 共经  $2 + 1 + 2 = 5$  次比较，在  $S[4]$  处命中

$\oplus \text{S.search}(3, 0, 7)$ : 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

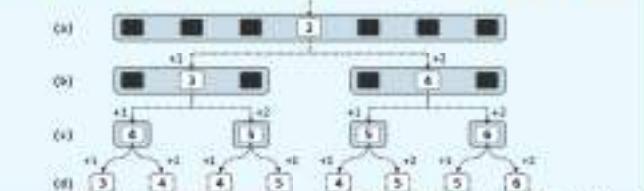
递归路径：轴点选取中点，递归深度  $\Theta(\log n)$ ；递归分割均耗时  $\Theta(1)$

如何更为精确地评估直接查找的性能呢？

考察关键码的比较次数，即直接长度 (search length)

通常，需分别针对成功与失败直接，从最好、最坏、平均等角度评估

比如，成功的平均直接长度约大致为  $\Theta(1.39 \cdot \log n)$  //详见教材。习题解析



二分查找根本上的效率仍没有改进余地，因为不断发掘

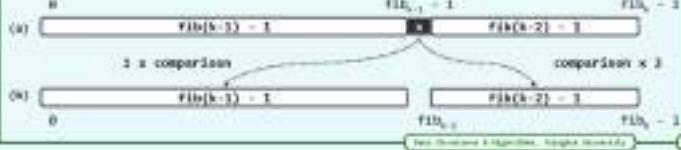
转向左、右分支的关健码比较次数不等，而且归根到底相等

若能通过递归深度的不均衡，对转向成本的不均衡进行补偿

平均直接长度应能进一步降低...

比如，若设  $k = \text{fib}(n) - 1$ ，则可取  $mid = \text{fib}(n - k - 1)$

于是，前、后子问题的长度分别为  $\text{fib}(k - 1) + 1, \text{fib}(k - 2) + 1$



## 2. 向量

## 有序向量

Fibonacci查找

## 矩阵链

long long fibs[100];

## 实现

```

<template <typename T> static Rank fibSearch( T *A, T const &e, Rank lo, Rank hi ) {
    Fib fib(hi - lo); //用O(log2n) = O(log2(hi - lo))时间构建fib数列
    while ( lo < hi ) {
        while ( hi - lo < fib.get() ) fib.prev(); //最多迭代几次? 整体要计几次?
        //通过向右顺序查找, 确定目标fib(n) - 1的值点 (分摊O(1))
        Rank mi = lo + fib.get() - 1; //按黄金比例切分
        if ( e < A[mi] ) hi = mi; //深入左半段 [lo, mi] 继续查找
        else if ( A[mi] < e ) lo = mi + 1; //深入右半段 [mi, hi]
        else return mi; //在E范围内
    }
    return -1; //查找失败
}

```

## 2. 向量

## 有序向量

## 二分查找 (版本B)

稍微风马牛不相干的书, 被本章的刀刀无双给, 把简陋的大师们给降维了。一气儿崩, 一气儿崩, 相同的一半, 使人感到麻痹, 另一半使人感到警觉。

林伟群

www.imooc.com/1000.html

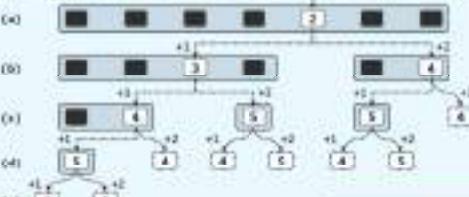
## 查找长度

◆ Fibonacci查找的ASL: (在常规的意义上) 低于二分查找 //理论的财, 实践的贼

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

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

$$\text{平均失败查找长度} = (4 + 5 + 2 + 4 + 5 + 4 + 5 + 4) / 8 = 35 / 8 = 4.38$$



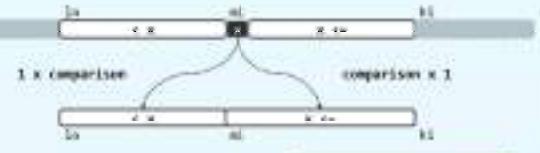
Data Structures & Algorithms, Python Version 2.0

## 改进思路

◆ 二分查找中左、右分支向代价不平等的问题, 也可直接解决

◆ 比如, 每次迭代 (或每个嵌套实例) 仅做 1 次关键码比较

如此, 所有分支只有 1 个方向, 而不再是 2 个



Data Structures & Algorithms, Python Version 2.0

## 通用策略

◆ 对于任何的  $A[\alpha, \beta]$ , 总想选取  $A[\alpha]$  作为轴点,  $0 \leq \alpha < 1$

比如: 二分查找对应的  $\alpha = 0, \beta = 1$ , Fibonacci查找对应的  $\alpha = 0, \beta = 0.6180339\dots$

◆ 在  $[0, 1]$  内,  $\lambda$  如何取值才能达到最优? 因平均查找长度为  $\alpha(\lambda) \cdot \log_2 n$ , 何时  $\alpha(\lambda)$  最小?

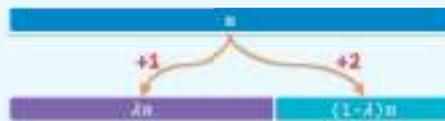


$\phi = 0.6180339\dots$

◆ 通解式:  $\alpha(\lambda) \cdot \log_2 n = \lambda \cdot [1 + \alpha(\lambda) \cdot \log_2 (\lambda n)] + [1 - \lambda] \cdot [2 + \alpha(\lambda) \cdot \log_2 ((1 - \lambda)n)]$

$$\text{◆ 极端解: } \frac{-\ln 2}{\alpha(\lambda)} = \frac{\lambda \ln \lambda + (1 - \lambda) \ln (1 - \lambda)}{2 - \lambda}$$

◆ 当  $\lambda = \phi$  时,  $\alpha(\lambda) = 1.449412\dots$  达到最小



Data Structures & Algorithms, Python Version 2.0

## 实现

```

<template <typename T> static Rank biseSearch( T *A, T const &e, Rank lo, Rank hi ) {
    while ( 1 < hi - lo ) { //有效查找区间始终存在时, 算法才会停止
        Rank mi = (lo + hi) / 2; //以中点为轴点, 比较后根据情况
        if ( e < A[mi] ) hi = mi; //深入左半段 [lo, mi] 或 [mi, hi]
        else if ( A[mi] < e ) lo = mi + 1; //深入右半段 [lo, mi] 或 [mi, hi]
        else return mi; //在E范围内
    }
    //相对于原本A, 偏好 (坏) 情况下循环 (好), 有冲突时下的S策略加锁后, 防冲突的数据稳定性
}

```



Data Structures & Algorithms, Python Version 2.0

## 语义约定

◆ 针对特殊情况，如何统一处理？比如

- 目标元素不存在；或反过来
- 目标元素同时存在多个

◆ 有序向量自身，如何健壮地维护？

比如：`V.insert( 1 + V.search(e), e )`

- 原理失败，但能给出新元素适当的插入位置 //有序性
- 若允许重复元素，则每一组由按其插入的次序排列 //稳定性

◆ 为高，需要更为精确、清晰、简明地定义search()的返回值

[Data Structures & Algorithms - Simple Searches](#)

不变性： $A[lo, hi] \Leftarrow e \in A[hi, n]$ 

◆ 在算法执行过程中的任意时刻

$A[lo - 1] / A[hi]$  总是（截至目前已确认的）不大于 $e$ 的最大者 / 大于 $e$ 的最小者

◆ 当算法终止时 ( $lo = hi$ )

$A[lo - 1] / A[hi]$  带括号 (空集) 不大于 $e$ 的最大者 / 大于 $e$ 的最小者



[Data Structures & Algorithms - Simple Searches](#)

## 语义约定

◆ 四种：search()把范围不大于 $e$ 的最后一个元素 //首尾相接，即大于 $e$ 的第一个元素  
 若  $-n < e < V[lo]$ ，则返回 $lo - 1$  //左闭右开。首元素的前驱  
 若  $V[hi - 1] < e < e$ ，则返回 $hi - 1$  //末元素。右闭右开的前驱  
 若  $V[k] < e < V[k + 1]$ ，则返回 $k$  // $e$ 可作为 $V[k]$ 的后继插入  
 若  $V[k] \leq e < V[k + 1]$ ，亦返回 $k$  // $e$ 可作为 $V[k]$ 的后继插入，因稳定性

◆ 二分查找版本C：`int bSearch()`，尚未严格兑现这一语义约定

◆ 谨慎：对函数操作调整，使之符合约定！对`bSearch()`操作调整，使之符合约定

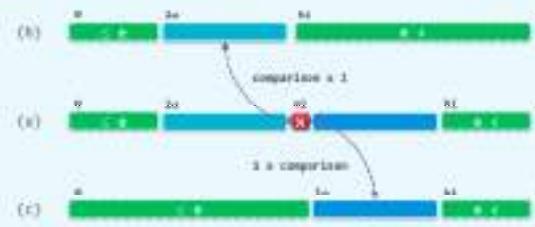
◆ 需要有更为简明的实现方式？

[Data Structures & Algorithms - Simple Searches](#)

不变性： $A[lo, hi] \Leftarrow e \in A[hi, n]$ 

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

◆ 故事目的：假设不变性一直保持至(s) //以下无印两种情况...



[Data Structures & Algorithms - Simple Searches](#)

## 2. 向量

## 有序向量

## 二分查找（版本C）

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

邓伟群

[zengwq@tsinghua.edu.cn](mailto:zengwq@tsinghua.edu.cn)

## 课后

◆ 针对二分查找，`#include <ACP/v3-18.2.1-ex_23.h>` 指出：

将三分支变为两分支后的改进效果

需要额外非常大 ( $2^{1720+100} = 2^{1820}$ ) 后方指针表

试阅读相关段落；这一结论，对当下的实际应用有何意义？

◆ template `cyberuse > static int bSearch( T *a, T const &e, Rank lo, Rank hi ) {`  
 `while ( lo < hi ) { //不变性：A[lo, hi] == e < A[hi, n]`  
 `Rank mi = (lo + hi) >> 1; //以中点为轴点，保证范围被缩小`  
 `e < A[mi] ? hi = mi : lo = mi + 1; // [lo, mi] 或 (mi, hi)`  
 `} //出口时，A[lo - hi] 为大于 $e$ 的完满的最大块`  
 `return --lo; //因 lo - 1 是不大于 $e$ 的完满的最大块`  
`}`

◆ 与版本B的区别

- 将查找区间宽度缩小到非 1 时，算法才结束 // $lo == hi$
- 转入右闭区间的最左，在边界取 $hi - 1$ 的和 $mi$  // $A[mi]$ 会被遗漏？
- 无论成功与否，返回的必须严格符合接口的语义约定... //如何证明其正确性？

[Data Structures & Algorithms - Simple Searches](#)

## 2. 向量

## 有序向量

## 数值查找

邓伟群

[zengwq@tsinghua.edu.cn](mailto:zengwq@tsinghua.edu.cn)

◆ 背景：已知有序数组中各元素随机分布的概率

比如：均匀独立的随机分布

◆ 于是： $[lo, hi]$  内各元素按大致按概率性地增长

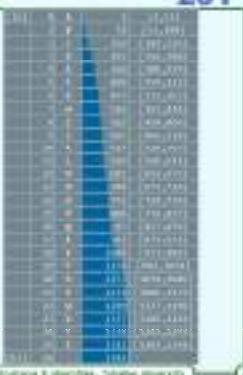
$$\frac{hi - lo}{hi - lo} = \frac{e - A[lo]}{A[hi] - A[lo]}$$

◆ 因此：通过遍历节点  $n_i$ ，可以极大地提高收敛速度

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

◆ 以英文词典为例：binary 大致位于  $1/26$  处

search 大致位于  $1/26^{26}$  处



◆ 关于逐值查找算法的平均性能分析，试阅读：

A. C. Yao & F. F. Yao

"The Complexity of Searching an Ordered Random Table"

Proc. of 17th FOCS (1976), 222-227

$lo = 0, hi = 99$

$lo = 0, hi = 18$  推理：  $ni = 0 + (18 - 0) * (99 - 1) / (92 - 1) \approx 9.3$   
赋值：  $ni = 9$   
比较：  $A[9] \approx 46 < e$

$lo = 18, hi = 18$  推理：  $ni = 18 + (18 - 18) * (99 - 49) / (92 - 49) \approx 18.2$   
赋值：  $ni = 19$   
比较：  $A[19] \approx 49 < e$

$lo = 18, hi = 18$  推理：  $ni = 18 + (18 - 18) * (99 - 51) / (92 - 51) \approx 18.8$   
赋值：  $ni = 19 < 19$   
直接完成 (not\_found)

## 2. 向量

### 链式排序

### 最佳解

http://tiny.cc/meyarw0

◆ 最坏：  $\sigma(hi - lo) = \sigma(n)$  //具体实例如何？

◆ 平均：每经一次比较，待排序区间宽度由  $n$  缩至  $\sqrt{n}$  // [Yao76, Fiat78], 习题解析 [2-24]

$n, \sqrt{n}, \sqrt{\sqrt{n}}, \sqrt{\sqrt{\sqrt{n}}}, \dots, 2$   
 $n, n^{1/2}, n^{1/4}, \dots, n^{1/2^k}, \dots, 2$

◆ 经多少次比较之后，有  $n^{1/2^k} < 2$ ，或等价地， $1/2^k \cdot \log_2 n < 1$ ？

不难解得： $k = \lceil \log_2 \log_2 n \rceil$  // 理解理解？

◆ 每经一次比较，待排序区间宽度的数值  $n$  开方，有效字长  $\log_2 n$  减半

- 直接查找 = 在字长意义上的“渐平查找”
- 二分查找 = 在字长意义上的“渐字查找”



```
template <typename T>
void vector<T>::sort( Rank lo, Rank hi ) { //区间[lo, hi]
    switch ( rand() % 5 ) { //可视具体问题的特点灵活选取或扩充
        case 1 : bubbleSort( lo, hi ); break; //冒泡排序
        case 2 : selectionSort( lo, hi ); break; //选择排序 (习题)
        case 3 : mergeSort( lo, hi ); break; //归并排序
        case 4 : heapSort( lo, hi ); break; //堆排序 (第10章)
        default: quickSort( lo, hi ); break; //快速排序 (第12章)
    }
}
```

//在此统一接口下，具体算法的不同实现，将在后续各章节陆续讲解

◆ 从  $\delta(\log n)$  到  $\delta(\log \log n)$ ，是否值得？

◆ 高级别的不用想——除非查找区间跨度极大，或然比较操作成本极高

比如， $n = 2^k (2^m) = 2^m 2^k = 4096$ ,  $\log_2 n = 12$ ,  $\log_2 (\log_2 n) = 5$

◆ 高级小技巧归纳和“家常”

◆ 常引入索引，降低运算

◆ 实际可行的方法

- 首先通过粗略查找，将查询范围缩小一定的尺度
- 然后再进行二分查找

```
template <typename T> void vector<T>::bubbleSort( Rank lo, Rank hi )
{
    while ( ! bubble( lo, hi - 1 ) ) ; //逐趟扫描归位，直至全局
}

template <typename T> bool vector<T>::bubble( Rank lo, Rank hi )
{
    bool sorted = true; //整体有序标志
    while ( ++lo < hi ) //自左向右，逐一检查被对换的元素
    {
        if ( _elems[lo - 1] > _elems[lo] ) { //若逆序，则
            sorted = false; //意味着尚未整体有序，升索引
            swap( _elems[lo - 1], _elems[lo] ); //交换
        }
    }
    return sorted; //返回排序标志
}

//如果等于 [lo, hi) 时，仍需  $O(n^2)$  时间——按照  $\delta(n)$  或已足够
```

## 再改进

```
template <typename T> void Vector<T>::bubbleSort( Rank lo, Rank hi ) {
    while ( lo < hi + bubble( lo, hi ) ); // 通过归并交换，直接全序
}

template <typename T> Rank Vector<T>::bubble( Rank lo, Rank hi ) {
    Rank last = lo; // 左右指针的起始对初始化为 (lo - 1, lo)
    while ( ++lo < hi ) // 从左向右，逐一检查待对相邻元素
        if ( _elem[lo - 1] > _elem[lo] ) { // 若逆序，则
            last = lo; // 更新最右的逆序的探测记录，并
            swap( _elem[lo - 1], _elem[lo] ); // 交换
        }
    return last; // 最后是右侧的记序对位置
} // 第一版本中的逻辑型缺 sorted，故为秩 last
```



## 分而治之

```
template <typename T>
void Vector<T>::mergeSort( Rank lo, Rank hi ) { // [lo, hi)
    if ( hi - lo + 1 ) return; // 单元则返回自然排序，否则...
    int mi = (lo + hi) / 2; // 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
}
```



## 综合评价

与第一版针对整数数组的基本相同，最好  $\Theta(n)$ ，最坏  $\Theta(n^2)$ 。  
插入含重复元素时，算法的稳定性 (stability) 是更为极致的要求。  
重复元素在插入，输出序列中的相对次序，是否保持不变？  
输入： 6, [7], 3, 2, [7], 1, 5, 8, [7], 4  
输出： 1, 2, 3, 4, 5, 6, [7], [7], [7], 8 //stable  
1, 2, 3, 4, 5, 6, [7], [7], [7], 8 //unstable  
以上排序和原算还是相同的吗？是的！为什么？  
在链表排序中，元素a和b的相对位置发生变化，只有一种可能。  
经分析与算已元素的安排，二者相比较应完全相同。  
往后看来一般归并交换中，二者顺序正好相反。  
在if一句的判断条件中，把“ $>$ ”换成“ $>=$ ”，将有等优化？

白玉堂的香料精，东阿春得均匀。  
蜂蜜蜂蜜真好，几滴蜂蜜水，没必要苦心。  
万绿千红也不改，任他随风飘舞。  
新华林笑本无邪。  
好闻好闻，送我上青云。

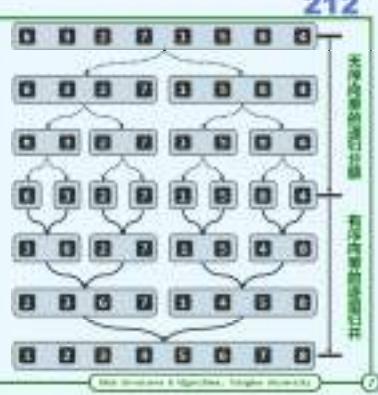
## 2. 向量

归并排序  
分而治之

邓佳群  
[dengjiaqun.sina.com](http://dengjiaqun.sina.com)

## 原理

// 分治策略  
// 每层与底层通用  
// J. von Neumann, 1945  
序列一分为二 // O(1)  
子序列归排序 // O(1)(O(2))  
合并有序子序列 // O(O(1))  
// 越直越对称，整体的运行成本由该层  
外 (nlogn)



## 基本实现

```
template <typename T> void Vector<T>::merge( Rank lo, Rank mi, Rank hi ) {
    T* A = _elem + lo; int lh = mi - lo; T* B = new T[2lh]; // [A, B, B+lh] = _elem[lo, hi]
    for ( Rank k = lo; k < mi; k++ ) A[k] = B[k]; // 复制前子序列到 A, B[0] = _elem[lo, mi]
    int lc = mi - mi + lh; T* C = _elem + mi; // 后子序列 C[0, lc] = _elem[mi, hi]
    for ( Rank k = lo, j = mi, l = mi + lh; l < hi; k++, j++ ) // [C, C+lh] = [B, B+lh] 中小者赋值的末尾
        if ( A[j] < B[l] ) A[k] = A[j++]; // A[1] < B[1] <= A[2]
        else ( B[l] < A[j] ) C[k] = B[l++]; // C[1] < B[1] <= A[2]
    // 为提高实现效率，假定效率很高，不知细分情况
    delete [] B; // 释放临时空间
}
```

## 分而治之

```
template <typename T>
void Vector<T>::mergeSort( Rank lo, Rank hi ) { // [lo, hi)
    if ( hi - lo + 1 ) return; // 单元则返回自然排序，否则...
    int mi = (lo + hi) / 2; // 以中点为界
    mergeSort( lo, mi ); // 对前半段排序
    mergeSort( mi, hi ); // 对后半段排序
    merge( lo, mi, hi ); // 归并
}
```



## 2. 向量

归并排序  
二路归并

## 邓佳群

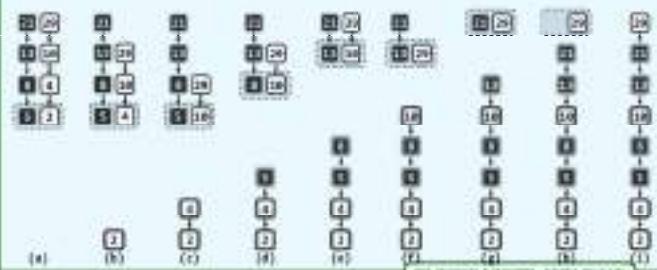
天下大物，分久必合，合久必分

[dengjiaqun.sina.com](http://dengjiaqun.sina.com)

## 二路归并

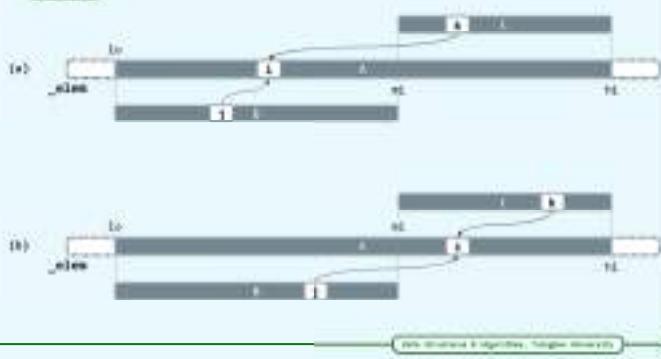
2-way merge：两个有序序列，合并为一个有序序列：

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



```
template <typename T> void Vector<T>::merge( Rank lo, Rank mi, Rank hi ) {
    T* A = _elem + lo; int lh = mi - lo; T* B = new T[2lh]; // [A, B, B+lh] = _elem[lo, hi]
    for ( Rank k = lo; k < mi; k++ ) A[k] = B[k]; // 复制前子序列到 A, B[0] = _elem[lo, mi]
    int lc = mi - mi + lh; T* C = _elem + mi; // 后子序列 C[0, lc] = _elem[mi, hi]
    for ( Rank k = lo, j = mi, l = mi + lh; l < hi; k++, j++ ) // [C, C+lh] = [B, B+lh] 中小者赋值的末尾
        if ( A[j] < B[l] ) A[k] = A[j++]; // A[1] < B[1] <= A[2]
        else ( B[l] < A[j] ) C[k] = B[l++]; // C[1] < B[1] <= A[2]
    // 为提高实现效率，假定效率很高，不知细分情况
    delete [] B; // 释放临时空间
}
```





◆ 算法的运行时间主要消耗于for循环，共有两个控制变量

初值： $j = 0, k = 0$

循环： $j < 1b, k < 1c$

条件： $j + k < 1b + 1c + 1L - 1a = n$

◆ 观察：每经过一次迭代， $j$ 和 $k$ 中至少有一个会加一（ $j+k$ 也跟着少加一）

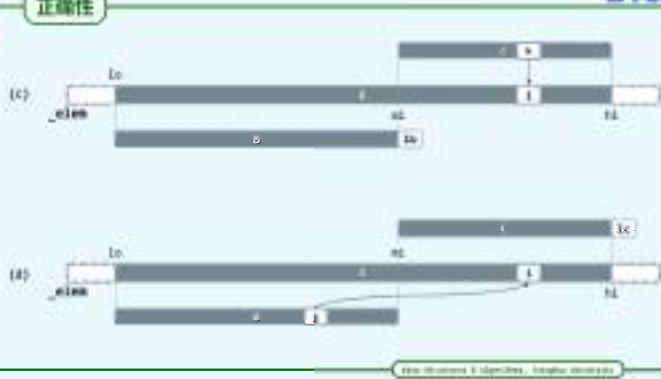
◆ 故知：merge()总操作不超过 $\sigma(n)$ 次，累计只能遍历时间！

◆ 这一结论与归并算法的 $O(n \log n)$ 下界并不矛盾——毕竟这里的 $a$ 和 $c$ 均已各自有序

◆ 注意：清归并子序列不必等长

亦即：允许 $1b > 1c, m = (1b + 1L)/2$

◆ 实际上，这一算法及结论也适用于另一种序列——列表（下一章）



#### 优点

实现在坏情况下是第 $\sigma(n \log n)$ 级别的第一个排序算法

不需随机数号，完全顺序结构——尤其适用于

列表之类的序列

困难之类的设备

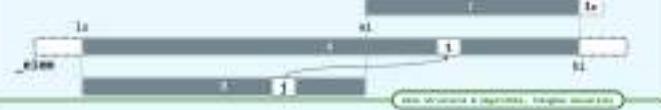
只要实现恰当，对保证稳定性——出现相同元素时，在例子中是优先  
可扩展性极佳，十分适合于外部排序——海量网页搜索引擎均归并  
属于并行化

#### 缺点

堆栈大，需要对每级调用的辅助空间——对函数体而言？

即使输入完全（或相近）有序，仍需 $\Omega(n \log n)$ 时间——改进...

```
+ for ( ; max(i = n, j = m, k = 0); k < 1L - 1 + i + j + k) {
    if (i < 1c && j < 1L - 1 + i) C[k] < B[j] -> A[i++], C[k++];
    if (i + 1L - 1c < j || B[j] < C[k] -> A[i++], B[j++];
} // 交换循环体内两句话次序，避免冗余逻辑
```



### 3. 列表

循位置访问

单链表

<http://tinyurl.com/3yqfjw>

### 2. 向量

归并排序

稳定性

I think there is a world market  
for about five computers.  
— T. J. Watson, 1943

单链表  
<http://tinyurl.com/3yqfjw>

◆ 数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织一般不变：get, search

2) 动态：需写入，数据结构的局部或整体将改变：insert, remove

◆ 与操作方式相对应地，数据元素的存储与组织方式也分为两种

1) 静态：数据空间整体创建或销毁

数据元素的物理存储次序与逻辑次序严格一致；可支持高效的静态操作  
比如物理，元素的物理地址与逻辑次序绝对对应

2) 动态：为数据元素动态地分配和回收物理空间

数据元素记录彼此的物理地址，在逻辑上形成一个整体；可支持高效的动态操作

## 从向量到列表

◆ 列表 (list) 是采用动态链存储结构的典型结构

其中的元素称作节点 (node)

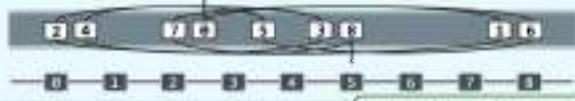
各节点通过前向或后向指针彼此联接，在逻辑上构成一个线性序列

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

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

前驱或后继都存在，则必然唯一

没有前驱/后继的 **头** (first) / **尾** (last/rear) 节点



## 从找到位置

◆ 直接使用指针访问 (call-by-pointer) 的方式

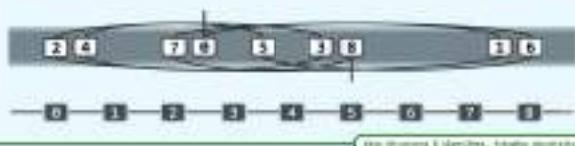
根据数据元素的秩，可在 $\delta(1)$ 时间内直接确定其物理地址

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

◆ 比如：假设在北京市海淀区的街道  $V$ ，各住户的地理位置均为：

则对于门牌号为  $i$  的住户，地理位置 =  $V + i \times \star\cdot\star$

◆ 这种高效的方式，是否能列表应用？



## 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，可设置并的深若干基本的操作接口

## 操作 功能

pred() 当前节点的前节点的位置

succ() 当前节点后继节点的位置

data() 当节点的所存数据对象

insertBefore(e) 插入前驱节点，存入被引用对象，返回新节点位置

insertAfter(e) 插入后继节点，存入被引用对象，返回新节点位置



## 列表节点：模板类

◆ `#include "ListNode.h"` //引入列表节点类 (150 C++11, template alias)

◆ `template <typename T> class ListNode {` //简洁起见，完全开放而不再过度封装

`struct ListNode {` //列表节点模板类 (以内部嵌套形式实现)

`T data; //数据`

`Posi(T) pred; //前驱`

`Posi(T) succ; //后继`

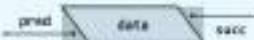
`ListNode() {} //对header和trailer的构造`

`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) insertAsSucce(T const& e); //后插入`



## 从找到位置

◆ 既然用线性序列，列表同样也可通过秩来定位节点：从头/尾端出社，沿前继/后继到哪...

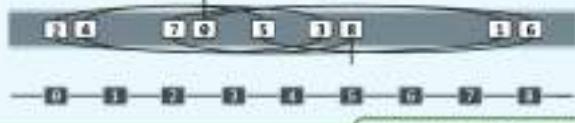
◆ 然而，此时的链接访问成本过高，已不合时宜 //List::operator[] (Rank r)，下节讲解

//理解两种访问方式的优劣 (exiplist, 第九章)

◆ 因此，应改用偏位遍历 (call-by-position) 的方式

亦即，转而利用节点之间的相对引用，找到特定的节点

◆ 比如：找到 我的朋友A 的朋友B 的朋友C 的... 的朋友D



## 列表节点：模板类

◆ `#include "ListNode.h"` //引入列表节点类 (150 C++11, template alias)

◆ `template <typename T> class ListNode {` //简洁起见，完全开放而不再过度封装

`struct ListNode {` //列表节点模板类 (以内部嵌套形式实现)

`T data; //数据`

`Posi(T) pred; //前驱`

`Posi(T) succ; //后继`

`ListNode() {} //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) insertAsSucce(T const& e); //后插入`

3. 列表  
接口与实现

两只老虎跑山来，九十九只在山上。  
两只老虎不见头，头朝山头出山来。

邓伟祥  
[www.dengweixiang.com](http://www.dengweixiang.com)

## 列表：ADT接口

操作接口 功能 遍历方法

`size()` 统计列表中的数据项 (节点总数) 列表

`first(), last()` 返回首、末节点的位置 列表

`insertBefore(e), insertLast(e)` 将*e*插入首、末节点之前或之后 列表

`insert(p, e), insert(r, e)` 将*e*插入节点*p*的直接后继、前驱插入 列表

`remove(p)` 删除节点*p*的直接后继、前驱引用 列表

`disorderAll()` 打乱所有节点是否已被排序的排列 列表

`sort()` 调整各节点的顺序，使之按升序排列 列表

`find(e)` 遍找数据元素*e*，失败时返回NULL 列表

`search(e)` 遍找*e*，直到不大于*e*且既最大的节点 有序列表

`deuplicate()`, `equality()` 去除重复节点 列表/有序列表

`traversal()` 遍历列表 列表



## 列表：模板类

◆ `#include "ListNode.h"` //引入列表节点类

◆ `template <typename T> class List {` //列表抽象类

`private:` int \_size; //数据

`Posi(T) header; Posi(T) trailer; //头、尾哨兵`

`protected: /* ... 内部函数 */`

`public: /* ... 构造函数、析构函数、只读接口、可写接口、遍历接口 */`

`};`

◆ 等效地，头、首、末、尾节点的秩可分别理解为-1, 0, n-1, n



```
template <typename T> void List<T>::init() { // 初始化，创建列表对象时使用
    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; // 头部
    trailer->pred = header; trailer->succ = NULL; // 尾部
    _size = 0; // 记录规模
}
```



在返回 p (可能指向 trailer) 的 rank(e) 调用中，找到等于 e 的最后者

```
template <typename T> //从外部调用时, 0 <= n < rank(e) < _size
Posi(T) List<T>::find( T const & e, int n, Posi(T) p ) const { //O(n)
    while ( 0 < n-- ) //顺序遍历：从右向左，逐个将当前项与 e 对比
        if ( e == ( p = p->pred )->data ) //这里假定类型 T 已重载操作符 “==”
            return p; // 跳出循环后返回结果
    return NULL; // 越过边界，意味着查找失败
} //header 的存在使得处理更为简便
```

### 3. 列表

#### 无序列表：随机访问

Don't lose the link.  
— Robin Miller

邓伟群  
deng@zjhu.edu.cn

\* 典型的调用模式：通过返回值判定

```
x = L.find( e, n, p );  
cout << x->data;  
cout << "not found";  
  
Posi(T) find( T const & e ) const { // 建立全局接口  
    return find( 0, _size, trailer ); // 从内部调用时, rank(trailer) == _size  
}
```

### 3. 列表

#### 无序列表：插入

邓伟群

deng@zjhu.edu.cn

\* 可否像链表那样随机访问方式？

\* 可以，比如，通过虚假下标操作符

```
template <typename T> //assert: 0 <= n < size
T List<T>::operator[]( Rank r ) const { //O(r) 效率低下，可模拟为之，却不宜使用
    Posi(T) p = first(); // 从首节点出发
    while ( 0 < r-- ) p = p->succ; // 遍历用 r 个节点即是
    return p->data; // 目标节点
} //在一节点的数，亦即其常数倍数
```

\* 时间复杂度为 O(r)，相当于子待访问节点的秩  
以均分布分为界，单次访问的期望复杂度为

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

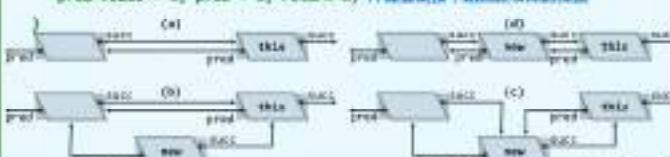
### 3. 列表

#### 无序列表：查找

邓伟群  
deng@zjhu.edu.cn

```
template <typename T> Posi(T) List<T>::insertBefore( Posi(T) p, T const & x )
{ _size++; return p->insertAsPred( x ); } // 0 作为前驱插入 (before)
```

```
template <typename T> //前插入算法 (后插入算法完全对称)
Posi(T) ListNode<T>::insertAsPred( T const & e ) { //O(1)
    Posi(T) x = new ListNode( e, pred, this ); // 创建 (耗时 O(1))
    pred->succ = x; pred = x; return x; // 建立链接，返回新节点的位置
```



### 3. 列表

#### 无序列表：基于复制的构造

单链表

http://tiny.cc/meyarw

```
◆ template <typename T> //基本插入
void List<T>::copyNodes( Posi(T) p, int n ) { //从n
    init(); //创建头，清除节点并被初始化
    while (n--) //将后n个节点依次作为末节点插入
        { insertLast( 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, Single Header

242

### 3. 列表

#### 无序列表：析构

单链表

http://tiny.cc/meyarw

```
◆ template <typename T> List<T>::~List() { //彻底析构
    { clear(); delete header; delete trailer; } //清空列表，释放头、尾指针结点
}

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

◆ remove( header->succ )操作流程( trailer->pred )呢？

```

Data Structures &amp; Algorithms, Single Header

243

### 3. 列表

#### 无序列表：删除

单链表

http://tiny.cc/meyarw

```
◆ template <typename T> //删除合法位置p处节点，返回其数据
T List<T>::remove( Posi(T) p ) { //O(1)
    T x = p->data; //返回待删除节点数据(设类型T可直接赋值)
    p->pred->succ = p->succ;
    p->succ->pred = p->pred;
    delete p; _size--; return x; //返回删除数据
}
```

244

247

### 3. 列表

#### 无序列表：去重

单链表

http://tiny.cc/meyarw

```
◆ template <typename T> int List<T>::deduplicate() { //删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; //记录当前规模
    Posi(T) p = first(); Rank [ ] = 1; //p从首节点起
    while ( !trailer != ( [ p = p->succ ] ) ) { //依次遍历末节点
        Posi(T) q = find( p->data, [ ] , p ); //在p的r个(首)的键中，查找与之相同的
        if ( q != remove(q) : [ p++ ] //若找到存在，则删除之；否则就递增——奇葩remove(p)！
            ) //assert: 循环过程中的任意时刻，p的所有数据互不相同
        return oldSize - _size; //彻底清空之后，再检测元素总数
    }
}

//正确性及效率分析的方法与结论，与vector::deduplicate()相同

```

Data Structures &amp; Algorithms, Single Header

248

## 3. 列表

无序列表：遍历

## 3. 列表

有序列表：查找

邓伟群

dengweiqun@zjhu.edu.cn

邓伟群

dengweiqun@zjhu.edu.cn

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

template <typename T>
template <typename VST>
void List<T>::traverse( VST & visit ) { //函数对象
    Posi(T) p = header;
    while ( ( p + p->succ ) != trailer ) visit( p->data );
}
```

Data Structures &amp; Algorithms, 第四版 学习手册

◎ 在有序列表内节点s的n个（幕）的圈中，找到不大于e的最后一个

◆ template &lt;typename T&gt;

```
Posi(T) List<T>::search( T const & s, int n, Posi(T) p ) const {
    while ( 0 <= n-- ) //对于p的最近n个邻居，从右向左
        if ( ( ( p + p->pred ) ->data ) <= s ) break; //逐个比较
    return p; //直至命中，数据越界或越过边界后，继续查找终止的位置
} //最好O(1)，最坏O(n)；平均O(n)，正比于区间宽度
```

Data Structures &amp; Algorithms, 第四版 学习手册

## 3. 列表

有序列表：唯一化

邓伟群

dengweiqun@zjhu.edu.cn

◆ template &lt;typename T&gt;

Posi(T) List&lt;T&gt;::search( T const &amp; e, int n, Posi(T) p ) const ;

- ◆ 通过与头结点相比较，便于插入操作等后续操作：`insertAt( search(s, r, p), s )`
- ◆ 为将来构造有序性提高查找效率？实现不妥，还是根本不可行？
- ◆ 按照低位进位的方式，物理存储地址与其逻辑次序无关

依赖移的随机访问无法高效实现，而只能依赖元素间的引用顺序访问

```
#template <typename T> int List<T>::uniquify() { //删除重复元素
    if ( _size < 2 ) return 0; //平凡列表自然无重复
    int oldSize = _size; //记录源长度
    ListNodePosi(T) p = first(); //p为被匹配节点
    ListNodePosi(T) q; //q为匹配项
    while ( trailer != ( q = p->succ ) ) //仅需考虑前者的邻居对(p, q)
        if ( p->data != q->data ) p = q; //若不同，则转向下一邻居
        else remove(q); //删除(图四)，移除后退
    return oldSize - _size; //减去空化量，即被删除元素总数
} //只需遍历整个列表一趟，O(n)
```

Data Structures &amp; Algorithms, 第四版 学习手册

## 3. 列表

选择排序

邓伟群

dengweiqun@zjhu.edu.cn

萧下又选了几样果品叫凤姐送去，  
凤姐儿也送了几样来。

## 回忆起泡排序...

◆ 算法需要  $\sigma(n^2)$  次网，是因为每趟扫描交换需要  $\sigma(n)$  次网： $\sigma(n)$  次比较 +  $\sigma(n)$  次交换  
◆  $\sigma(n)$  次比较或许无可厚非，但  $\sigma(n)$  次空跑趟时没有必要 // 大量无谓的徒劳移动



## 回忆起泡排序...

◆ 每趟扫描交换的头尾效率，无非就是通过比较找到当前的最大元素所在，并通过交换使之就位  
◆ 如此看来，在经  $\sigma(n)$  次比较遍历之后，仅需一次交换即足矣



## selectMax()

◆ template <typename T> // 从起始于位置 p 的 n 个元素中选出最大者，l <= p < r

Posi(T) max = p; // 最大者暂定为 p

for (Posi(T) cur = p; cur < n; cur++) // 遍历节点逐一与 max 比较

if (!max || cur->succ->data > max->data) // 若 > max

max = cur; // 到底最大元素位置记录

return max; // 返回最大节点位置

// See Selection & Insertion, Simple Iterators

## 稳定性

◆ 有多个重复元素同时命中时，往往需要按某种预定的次序，返回其中最大的某一个

◆ 比如，通常都归于「右后置优先级」

◆ 为此，必须采用比较函数 lt() 或 ge()，即等效于「右置优先」

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

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

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

2 6 3 8 1 5 6b 6c 7 8 9

◆ 如此即可保证，重复元素在列表中的次序与插入次序一致

// See Selection & Insertion, Simple Iterators

## 实例

迭代轮次	前缀无序子序列	后缀有序子序列	
0	5 2 9 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

## 性能分析

◆ 并演化为次，在第 k 次迭代中

selectMax() 为  $\theta(n-k)$

sweep() 为  $\sigma(k)$

故总体复杂度为  $\theta(n^2)$

// 算术进制

// 算 maxsel() + insertk()

◆ 想象如此，元素移动操作远远少于起泡排序

// 实际更为费时

也就是说， $\theta(n^2)$  主要来自于元素比较操作

// 成本相对更低

◆ 可否... 每轮只能  $\sigma(n)$  次比较，那就找出当前的最大元素？

◆ 可以！... 利用高级数据结构，selectMax() 可改进为  $\theta(\log n)$

// 前提分解

当然，如此立即可以得到  $\theta(n \log n)$  的排序算法

// 保持兴趣

## selectionSort()

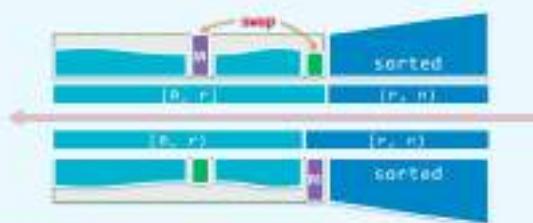
```
// 对列表中起始于位置 p 的连续 n 个元素做选择排序。valid(p) && rank(p) + n <= size
template <typename T> void list<T>::selectionSort(Posi(T) p, int n) {
    Posi(T) head = p->pred; Posi(T) tail = p; // 将操作区间(head, tail)
    for (int i = 0; i < n; i++) tail = tail->succ; // head/tail 可能是头/尾哨兵
    while (i < n) { // 从 head 起始从 (非平凡) 被操作区间内找出最大者，并将其放在区间首端
        insertk(tail, const selectMax(head->succ, n)); // 插入...
        tail = tail->pred; n--; // 移除尾区间。若尾区间的范围，当 i 步骤后
    }
}
```

3. 列表  
循环  
单链表  
<http://www.ust.hk/~cs201/>

- 任何一个序列  $s(0, n)$  都可以分解为若干个循环节 // 循环节之间可重次序
- 任何另一个序列  $s(0, n)$  都对应于一个有向图  $r(s, n)$  // 循环节之后
- 元素  $A[k]$  在  $s[]$  中到底的映射，记作  $r(A[k]) = r(k)$
- 元素  $A[k]$  所属的循环节是：  

$$A[k], A[r(k)], A[r(r(k))], A[r(r(r(k)))] \dots, A[r(\dots r(k) \dots)] = A[k]$$
- 每个循环节，长度均不超过  $n$
- 循环节之间，没有重复元素

- 因已到位，无需交换 —— 这种情况会出现在几次？



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

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

3	-	p	-	a	-	-	-	-	-	-	-	-	-	-	-	-
-	n	-	-	-	-	-	-	-	-	-	b	-	l	-	-	-
-	-	-	c	-	-	-	-	-	-	-	d	-	k	-	-	-
-	-	-	-	d	-	-	-	-	-	-	e	-	f	-	-	-

Data Structures & Algorithms, Python Version 2.0



一语未了，只见宝玉笑嘻嘻的折了一枝红梅进来。  
众丫鬟忙归接过，插入瓶内。

林黛玉

http://tiny.cc/meyarw

- 每迭代一步，所检测环节的长度都恰好减少一个单位 // 直到长度为 1
- 为什么？



• 假设  $r[A] = r$

Data Structures & Algorithms, Python Version 2.0

- 【不变性】序列总被操作为两部分：

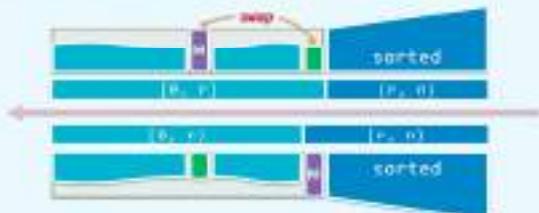
$S[0, r] \rightarrow O[r, n]$



- 【初始化】 $: S \leftarrow r \leftarrow \emptyset$  // 空序列无所谓有序无序

Data Structures & Algorithms, Python Version 2.0

- 因检测原属的循环节，自此成为一个长度为 1 的循环节



• 其余循环节保持不变

Data Structures & Algorithms, Python Version 2.0

- 反推法，特别  $O[r, n]$  的首元素  $e = L(r)$



- $O[0, r] \rightarrow$  确定 适当位置 // 有序序列的搜索
- 插入  $e$ ，得到  $O[r, n]$  // 有序序列的插入

Data Structures & Algorithms, Python Version 2.0

## 实例

迭代轮次	前缀有序子序列	当前元素	后缀无序子序列
0	-	-	9 2 7 4 6 3 1
1	-	1	2 7 4 6 3 1
2	(2)	2	7 4 6 3 1
3	(2) 5	3	4 6 3 1
4	2 5 (7)	4	6 3 1
5	2 4 (5) 7	6	3 1
6	2 (5) 4 3 6 7	3	1
7	(1) 2 3 4 5 6 7	-	-

Data Structures &amp; Algorithms, Python Version 2.0

//对列表中指定位置r的连接n个元素插入排序。call(p) && rank(p) + n < size

```
template <typename T> void List<T>::insertionSort( Posi(T) p, int n ) {
    for (int r = 0; r < n; r++) { //逐一引入新节点，由s得到s...
        insert( search( p->data, r, -1 ), p->data ); //查找 + 插入
        p = p->succ; remove( p->pred ); //转向下一节点
    } //n次迭代，每次O(r+1)
} //仅使用O(1)辅助空间，属于就地算法
```

◆ 虽然search()接口返回的位置之后插入当前节点，总能保持有序

◆ 防止各种情况下假正确定性，得益于新节点的作用：

- ① 中间有/不含与p相等的元素：s中的元素均严格小于/大于p

Data Structures &amp; Algorithms, Python Version 2.0

## 3. 列表

## 归并排序

因社孔早，六御在手  
棋盘摆中，胡棋摆缺  
龙棋之色，通以摆物

日再莫要必合弓，棋指移而致之？  
君九州以之游大号，当推指摆女？

http://tiny.cc/meyarw

## 性能分析

- ◆ 最好情况：完全（或几乎）有序
  - 每次迭代，只要1次比较，0次交换
  - 累计O(n)时间！

## ◆ 最坏情况：完全（或几乎）逆序

- 第k次迭代，需O(k)次比较，1次交换
- 累计O( $\frac{n^2}{2}$ )时间！

//改用纯递归？稍后分析

## ◆ 一般情况：完全3个步骤时

//inversion? 稍后添品！

- 第k次迭代，只需O(1)次比较，1次交换
- 累计O( $n + 2$ )！

//input-sensitive，对InplaceInsert至关重要

## ◆ 平均而言呢？

//当然，首先需要就过具体的随机分布...

Data Structures &amp; Algorithms, Python Version 2.0

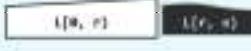
## 归并排序

List 

```
template <typename T> void List<T>::merge( Posi(T) p, int r, List<T> & L, Posi(T) q, int n ) {
    if ( r < 2 ) return; //待归并区间足够小则直接返回，否则...
    Posi(T) s = p; int m = r > 2 ? 2 : 1; //设m为界
    for ( int i = 0; i < m; i++ ) q = q->succ; //将分区调
    mergeSort( s, r-1, mergeSort( q, n-m ), n-m ); //子序列分别操作
    merge( p, m, q, n-m ); //归并
} //若归并可在原地附带完成，则总体运行时间为O(n log n)
```

## 平均性能：后向分析

- ◆ 假定：各元素的取值遵守均匀、独立分布
  - 于是：平均要被多少次元素比较？



## ◆ 考查：L[r]刚插入完成的那一时刻（穿越）



试问：此时的有序数组{0, r}中，哪个元素是此前的L[r]？

## ◆ 问题：其中的r+1个元素（均有可能），后概率均等于1/(r+1)

## ◆ 因此，在刚完成的这次迭代中，为引入L[r]所花费时间的数学期望为

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

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

## ◆ 再问：在n次迭代中，平均有多少次“无谓交换”？

//习题[1-10]

Data Structures &amp; Algorithms, Python Version 2.0

## 二路归并

//当前列表中由v组成的r个元素，与列表L中由v组成的n个元素归并

```
template <typename T> // (归并排序时为同一列表, this == L)
void List<T>::merge( Posi(T) p, int r, List<T> & L, Posi(T) q, int n ) {
    while ( r < n ) //在q尚未超出区间之前
        if ( ( 0 < r ) && ( p->data < q->data ) ) //若p仍在区间且v[p] < v[q]
            { if ( q == L ) p = p->succ; } break; n--; } //则将p直接后移
        else //若p已超出区间或v[q] < v[p]，则将q插入v之间
            { Insert( p, L.remove( [ q = q->succ ]->pred ) ); m--; }
} //每经过一次迭代n = m递减1，故整体运行时间为O(n log n)，理性正比于节点总数
```

Data Structures &amp; Algorithms, Python Version 2.0

## 3. 列表

逆序对

样例题

http://tanghuanhua.com

## 3. 列表

(xa) 游标实现

样例题

http://tanghuanhua.com

## Inversion

◎ 考查序列  $a[0, n]$ ，记录两数之间的比较大小令  $(i, j)$  表示一个逆序对，如果  $0 \leq i < j \leq n$  且  $A[i] > A[j]$ 

◎ 为便于统计，构造逆序对这一记数而画的图上

令例  $A[0] = \{5, 3, 1, 4, 2\}$  中，共有  $8 + 3 + 2 + 1 + 0 = 12$  个逆序对 $A[0] = \{3, 2, 1, 6, 5\}$  中，共有  $8 + 6 + 3 + 0 + 0 = 17$  个逆序对 $A[0] = \{5, 4, 3, 2, 1\}$  中，共有  $8 + 7 + 3 + 1 + 0 = 19$  个逆序对◎ 一般地，逆序对总数  $\leq \binom{n}{2} = O(n^2)$ 

282

Data Structures &amp; Algorithms, Design Patterns

## 动机与构思

◎ 基础的空闲位或链表中

或者不（直接）支持插队

或者不支持动态空间分配

此时，如何实现列表结构呢？

◎ 利用线性数组，以游标方式操作列表

elem[]：对外可见的数据

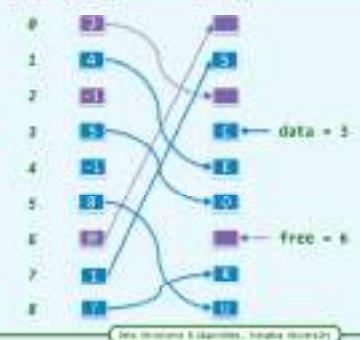
link[]：数据项之间的引用

◎ 避免逻辑上互补的列表 data 和 free

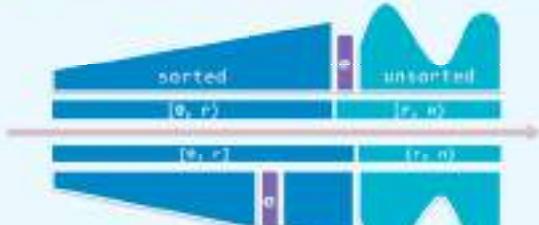
◎ 在插入或删除元素时，该如何调整？

286

rank link[] elem[]



## 在 insertionsort 中

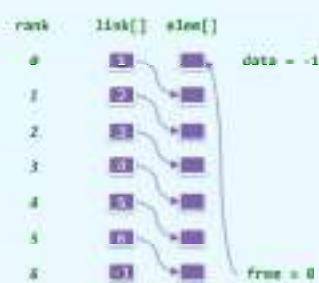
◎ 若  $e = A[r]$  上的逆序对共有  $I(r)$  个，则在接下来的一步迭代中，恰好需要做  $I(r)$  次比较◎ 每阶段含 1 个逆序对，则关键项比较次数为  $O(1)$ ，总行进趟数为  $O(n+1)$  //习题 [3-11]

283

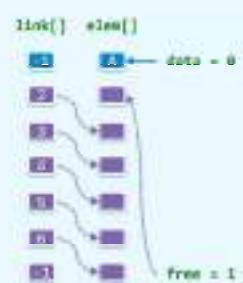
Data Structures &amp; Algorithms, Design Patterns

## 实例

init(7)



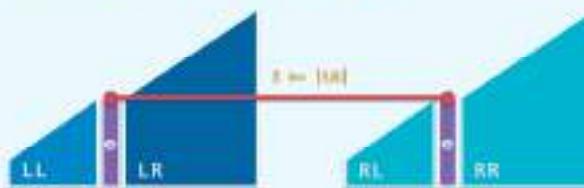
insert('A')



287

## 计数

◎ 任意给定一个序列，如何统计其中逆序对的总数？

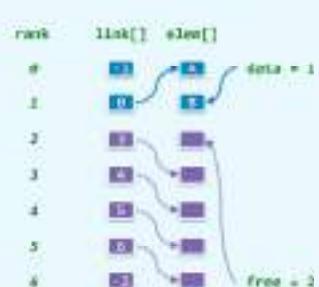
◎ 直接算法在最坏情况下，需要  $O(n^2)$  时间 //  $T = \Theta(\frac{n^2}{2})$ ◎ 使用归并排序的框架，仅需  $O(n \log n)$  时间 // 怎么做到的？

284

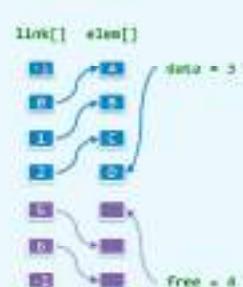
Data Structures &amp; Algorithms, Design Patterns

## 实例

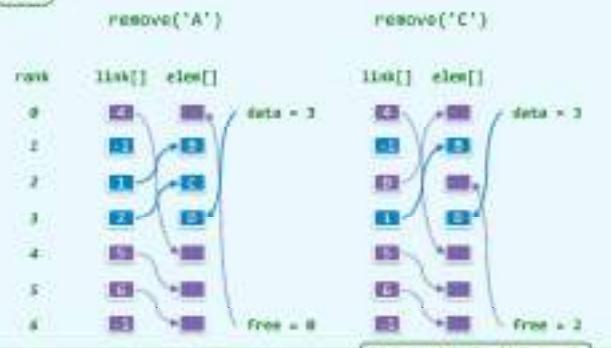
insert('B')



288

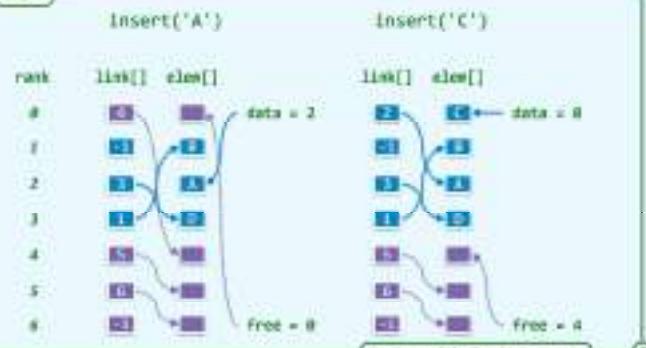
insert('C')  
insert('D')

## 实例



```
class Disk implements Geometry { // 实现Geometry接口的Disk类
    Point c; double r;
    public Disk(Point center, double radius) { // 构造方法
        c = center; r = radius;
    }
    public double perimeter() { return 2 * PI * r; } // 负责方法
    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;
    }
}
```

## 实例



## 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;
}
```

## 3. 列表

(ab) Java序列

荐读  
http://tiny.cc/meyarw0

## 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 (r < 0 || r > n) throw new ExceptionBoundaryViolation("out of range");
        if (n == N) throw new ExceptionBoundaryViolation("overflow");
        for (int i = n; i > r; i--) A[i] = A[i - 1];
        A[r] = obj; n++; return obj;
    }
    ...
}
```

## Interface: 定义

◆ Java支持API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
}

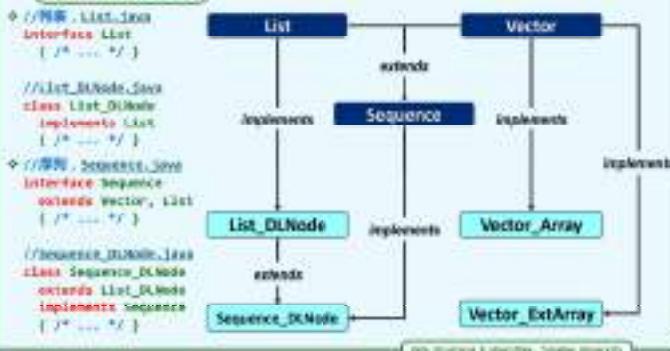
```

◆ Interface不能直接实例化为对象

得将Interface定义成任何类，都需要具体实现其中的接口方法

## 向量实现2: Vector\_ExArray.java

```
public class Vector_ExArray implements Vector {
    private int N = 0; // 数组的初始容量，可不断增加
    ...
    public Object insertAtRank(int r, Object obj) throws ExceptionBoundaryViolation {
        if (r < 0 || r > n) throw new ExceptionBoundaryViolation("out of range");
        if (N <= n) { // 空间用尽的处理
            N *= 2; Object B[] = new Object[N]; // 新建数组
            for (int i = n; i < N; i++) B[i] = A[i - n]; // 将旧元素依次后移
            A[0] = obj; n++; return obj;
        }
        ...
}
```



```

>>> 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
  
```

## 3. 列表

(xc) Python列表

胡伟群

wang@tonghuawu.com

```

>>> def reverse_1(L): # 遍历遍历法
...     for i in range(0, len(L)): # 对[0, n)内的每个i, 将次
...         L.insert(i, L.pop(0)) # 把末元素移到首位
...     return L # 遍历遍历得到倒置的列表
>>> def reverse_2(L): # 遍历访问法
...     i, j = 0, len(L) - 1 # 从第0, 末元素开始
...     while i < j: # 按次序交换L[i]及L[n-1-i]
...         L[i], L[j] = L[j], L[i] # 互换, 然后
...         i, j = i + 1, j - 1 # 重叠下一对元素
...     return L # 遍历访问得到倒置的列表
  
```

哪个版本效率更高？基础题目如何解题？

```

>>> # 在Python中, list属于内部的标准数据类型
>>> box = [ 'pencil', 'pen', 'ruler', 'rubber' ]; print box
['pencil', 'pen', 'ruler', 'rubber']
>>> for item in box: print item,
... pencil pen ruler rubber
>>> box.reverse()
>>> for item in box: print item,
... ruler ruler pen pencil
>>> box.sort()
>>> for item in box: print item,
... pen pencil rubber ruler
  
```

## 4. 栈与队列

栈接口与实现

如下用队列，如B站同，后来者居上。

胡伟群

wang@tonghuawu.com

```

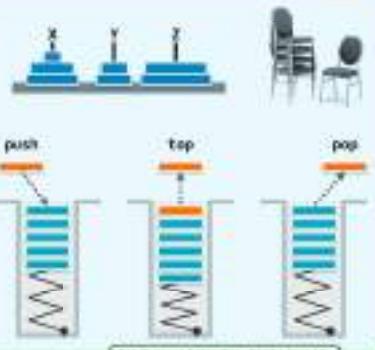
>>> for i in range(0, len(box)): # [0, n)
...     print box[i],
... pencil pen ruler rubber
>>> for i in range(len(box)-1, -1, -1): # [-1, 0-1]
...     print box[i],
... ruler rubber pencil pen
>>> for i in range(-1, -len(box)-1, -1): # [-1, -n-1]
...     print box[i],
... ruler rubber pencil pen
  
```

◆ 栈(stack)是受限的序列  
只能在顶端(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() 顶端

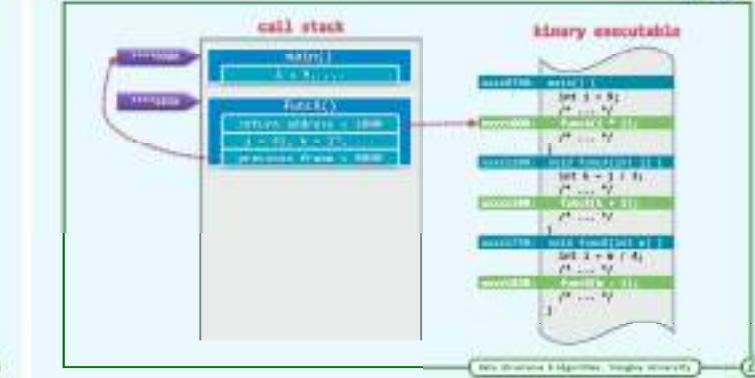
◆ 超进先出(LIFO)  
先进后出(FIFO)

◆ 扩展接口：getMax()...



操作	输出	栈 (左侧堆栈)		
stack()				
empty()	true			
push(5)				
push(3)		<table border="1"><tr><td>5</td></tr><tr><td>3</td></tr></table>	5	3
5				
3				
pop()	3	<table border="1"><tr><td>5</td></tr></table>	5	
5				
push(7)		<table border="1"><tr><td>7</td></tr><tr><td>5</td></tr></table>	7	5
7				
5				
push(3)		<table border="1"><tr><td>7</td></tr><tr><td>3</td></tr></table>	7	3
7				
3				
top()	3	<table border="1"><tr><td>7</td></tr><tr><td>3</td></tr></table>	7	3
7				
3				
empty()	false	<table border="1"><tr><td>7</td></tr><tr><td>3</td></tr></table>	7	3
7				
3				

操作	输出	栈 (左侧堆栈)						
push(11)		<table border="1"><tr><td>11</td><td>3</td><td>7</td><td>5</td></tr></table>	11	3	7	5		
11	3	7	5					
size()	4	<table border="1"><tr><td>11</td><td>3</td><td>7</td><td>5</td></tr></table>	11	3	7	5		
11	3	7	5					
push(6)		<table border="1"><tr><td>6</td><td>11</td><td>3</td><td>7</td><td>5</td></tr></table>	6	11	3	7	5	
6	11	3	7	5				
empty()	false	<table border="1"><tr><td>6</td><td>11</td><td>3</td><td>7</td><td>5</td></tr></table>	6	11	3	7	5	
6	11	3	7	5				
push(7)		<table border="1"><tr><td>7</td><td>6</td><td>11</td><td>3</td><td>7</td><td>5</td></tr></table>	7	6	11	3	7	5
7	6	11	3	7	5			
pop()	7	<table border="1"><tr><td>6</td><td>11</td><td>3</td><td>7</td><td>5</td></tr></table>	6	11	3	7	5	
6	11	3	7	5				
push(4)		<table border="1"><tr><td>4</td><td>6</td><td>11</td><td>3</td><td>7</td><td>5</td></tr></table>	4	6	11	3	7	5
4	6	11	3	7	5			
top()	11	<table border="1"><tr><td>4</td><td>6</td><td>11</td><td>3</td><td>7</td><td>5</td></tr></table>	4	6	11	3	7	5
4	6	11	3	7	5			
size()	4	<table border="1"><tr><td>4</td><td>6</td><td>11</td><td>3</td><td>7</td><td>5</td></tr></table>	4	6	11	3	7	5
4	6	11	3	7	5			

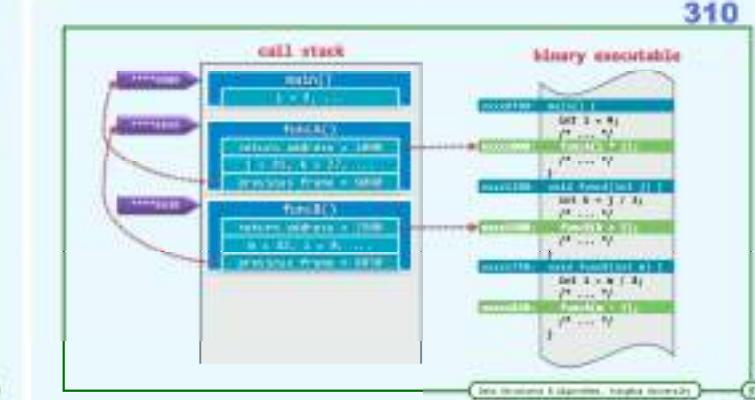


## 实践

少说些面子话别的事，我们由堆栈子类派生类来实现。

```
#template <typename T> class Stack: public Vector<T> { //由向量派生的栈模板类
public: //size(), empty()以及其它方法接口均可直接沿用
    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
}; //以向量基类/本端为线底/底——翻到过早呢？

//确认：如此实现的栈接口，均只需O(1)时间
//深思：基于列表，派生定义栈模板类
特别：你所实现的栈接口，效率如何？
```



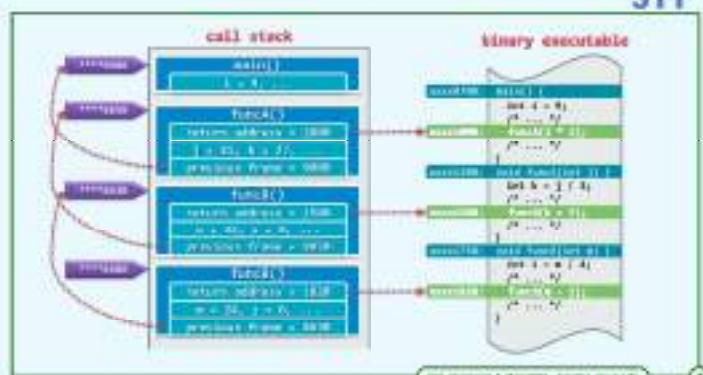
## 4. 栈与队列

调用栈：原理与算法

Yessiree. We do not doubt his word,  
as stack ourselves into the bus like flapjacks.

邓伟群

dengweiquan@ust.hk



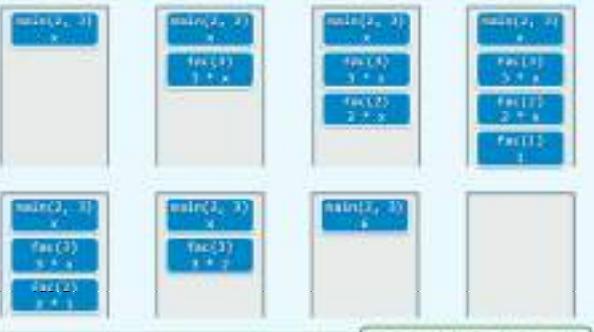
## 4. 栈与队列

调用栈：实例

邓伟群

dengweiquan@ust.hk

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



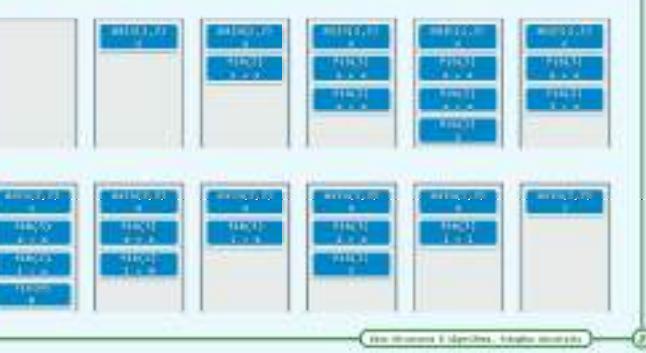
缺点：递归函数的空间复杂度，主要取决于最大递归深度，而非循环实现总步数

为提高效率的调用栈，需花费额外的处理时间

方法：显式地维护调用栈，将递归算法改写为迭代版本...

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

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



```
int fib(int n) {
    int f = 0, g = 1; //O(1)空间
    while (g < n++)
        { g += f; f = g - f; }
    return f;
}

void tailCalls(int n) { //O(1)空间
    while (1 < n)
        n = n % 2 + odd(n);
}
```

```
#include <iostream>
using namespace std;

int hailstone(int n) {
    if (2 > n) return;
    if (n % 2 == odd = 1) even(n);
    else odd(n);
}

int odd(int n) { hailstone(n / 2); }

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

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



Hickory, Dickory, Dock

The mouse ran up the clock

- Nursery Rhyme Medley

#### 4. 栈与队列

进制转换

#### 典型应用场合

##### 进序输出

- conversion
- 输出次序与处理过程顺序一致，逐项累加和输出长度不需预知

##### 延迟输出

- 线性扫描阅读模式中，在扫描至最长之后，方能推出可处理的数据

##### 递归嵌套

- 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确定

##### 模式计算

- DP
- 基于栈结构的特定计算模式

#### 4. 栈与队列

应用场景：进阶递归

邓伟峰

[dengweifeng.com](http://dengweifeng.com)



## 实例

```

bool solve( const char exp[], int lo, int hi ) { //exp[lo, hi]
    Stack> S; //使用栈记录未匹配的左括号

    for ( int i = lo; i < hi; i++ ) //逐一检查当前字符
        if ( '(' == exp[i] ) S.push( exp[i] ); //遇左括号：放进栈
        else if ( !S.empty() ) S.pop(); //遇右括号：若栈非空，则弹出左括号
        else return false; //否则（遇右括号时栈已空），必不匹配
    return S.empty(); //最终，栈空当且仅当匹配
}

```

Data Structures & Algorithms, 10th Edition (c) 2012

## 4. 栈与队列

栈操作

样例解

http://tiny.cc/meyarw

## 实例

◆ 实际上，我们考虑一种数据，只用一个计数器是么：始终忘掉：遍历位置  $i$  /  $S.size()$

(1) (2) (1) (2) (3) (2) (1) (2) (1) (0)

0 1 2 3 4 5 6 7 8 9 10 11

0 1 2 3 4 5 6 7 8 9 10 11

Data Structures & Algorithms, 10th Edition (c) 2012

## 拓展

◆ 以上思路及算法，可直接地推广至多种括号对称的情况

◆ 可否，使用多个计数器？不行，反例： ( ) ( )

◆ 其实，只要约定“括号”的通用格式，而不必事先固定括号的类型与数目

比如：<body> | </body>, <div> | </div>, <font> | </font>, <sp> | </sp>, <cols> | </cols>, ...

0 1 2 3 4 5 6 7 8 9 10 11

0 1 2 3 4 5 6 7 8 9 10 11

Data Structures & Algorithms, 10th Edition (c) 2012

## 拓展

◆ 按字典序，枚举由  $n$  对匹配括号组成的表达式

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

◆ 在由  $n$  对匹配括号组成的带有表达式中，按字典序输出第  $k$  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，随机平均地选择任选其一

ACP-v4-f4-p15, Algorithm R, D. E. Arnold & M. R. Sleep, 1968

◆ 能算法  $\mu$ ，不能可以直接实现算法  $\mu$  的功能吗？后面的真义何在？

Data Structures & Algorithms, 10th Edition (c) 2012

## 栈操作

◆ 常常地  $A = \{x_1, x_2, \dots, x_n\}$ ,  $B = \{y_1, \dots, y_m\}$

//压入为栈顶

◆ 只允许 将  $A$  的元素弹出并注入  $B$ ，或  
将  $B$  的元素弹出并注入  $A$

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

◆ 经过一系列以上操作后， $A$  中元素全部纳入  $B$  中

//压入为栈底

$B = \{x_{m1}, \dots, x_{mn}\}$   
即称之为  $A$  的一个栈排列 (stack permutation)

//压入为栈顶

$B = \{x_{m1}, \dots, x_{mn}\}$

$\leftarrow x_{m1}, x_{m2}, \dots, x_{mn} \right\} \in A$

Data Structures & Algorithms, 10th Edition (c) 2012

## 计算

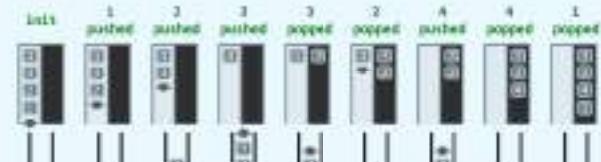
◆ 同一输入序列，可能多种栈顶次序

$\{1, 2, 3, 4, 1, 2, 1, 4, 3, 2, 1, 3, 1, 2, 4, 3, 2, \dots\}$

//显然，SP(n) > n!

◆ 长度为  $n$  的序列，可能的最小总数  $SP(n) = ?$

//当然，SP(n) > n!



Data Structures & Algorithms, 10th Edition (c) 2012

## 计算

◆  $SP(1) = 1$

◆ 等待，在第  $k$  次  $pop()$  之后首次底部变空

◆ 以下几种情况：

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

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

$$SP(2) = 4! / 3! / 2! = 2$$

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

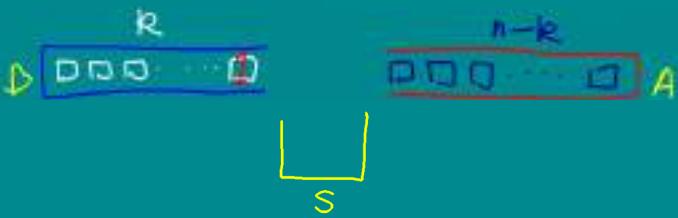
$$\dots$$

$$SP(6) = 12! / 7! / 5! = 132$$



Data Structures & Algorithms, 10th Edition (c) 2012

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



◆插入排序： $i = 2, 3, 4, \dots, n-1$  的任一插洞  $(p_1, p_2, p_3, \dots, p_n)$  是否为对称性？

◆简单情况： $\langle 1, 2, 3, \dots, n \rangle$

- 检测法共  $6! / 4! = 3! = 6$  种
- 全排列共  $3! = 6$  种 //少了一种...

◆  $\langle 3, 1, 2 \rangle$  //为什么是它？

◆ 规律：任意三个元素的后改某指针次序出现于乱洗中，与其它元素无关 //故可推广之...

◆ 对于任给  $1 \leq i < j \leq n$

$\boxed{1}, \dots, \boxed{i}, \dots, \boxed{j}, \dots, \boxed{k}, \dots, \boxed{n}$  必能挑到先

◆ 反过来，不存在「123？」模式的序列，一定是在混吗？

www.concursos.br Edital 00000 - Análise de conteúdo

◆ 定义： A permutation is a stack permutation [17].  
    (Knuth, 1968) It does NOT involve the permutation  $\begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}$ . //询问(a-1)

◆ 如此，可得一个  $\mathcal{O}(n^2)$  的检测算法 //进一步推...

◆  $[p_1, p_2, p_3, \dots, p_n] < [1, 2, 3, \dots, n]$  的检测法，尚待权衡  
    对于任意  $i < j$ ，不含模式  $\begin{bmatrix} \square & \square \\ \square & \square \end{bmatrix}$ ， $\begin{bmatrix} \square & \square \\ \square & \square \end{bmatrix}, \dots, \begin{bmatrix} \square & \square \\ \square & \square \end{bmatrix}$

◆ 如此，可得一个  $\mathcal{O}(n^2)$  的检测算法 //再进一步推...

◆  $\mathcal{O}(n)$  算法：遍历输出栈 A、B 和 S，检测进栈过程  
    每次  $S.pop()$  之前，检测 S 是否已空；或需弹出的元素不在 S 中，此时将元素

340

分析：每一组操作，都对栈的  $n$  次 push 与  $m$  次 pop 操作构成的一序列

(	)	(	)	(	)	(	)	(	)
+[1]	+[2]	+[3]	-[3]	-[2]	+[4]	-[4]	-[1]		
init	1 pushed	2 pushed	3 pushed	3 popped	2 popped	4 pushed	4 popped	1 popped	
									

↑ 3个元素的进退栈，等于子字符串的匹配

#### 4. 线与队列

中學英語教材

四

知实而不知名，知名而不知实，皆不如也。

[www.springerlink.com](http://www.springerlink.com)

第六章

令該審閱者正確的算不審訪式，計算與之對話的微積

```

$ echo $(( 0 + ( 1 + 23 ) / 4 * 5 * 67 - 8 + 9 ))
$ \v set J3={ 10 ^c< ( 1 - 2 + 3 * 4 ) } - 5 * ( 6 ^c 7 ) / ( 8 ^c 9 )
$ \${J3} 0 1 23 add 4 div 5 mul 67 mul add 8 sub 9 add =
$expr= $(echo "scale=0; $J3" | bc)
$word= NOT(0) + 12 + 34 * 56 + 7 + 89
$calc= 0 ! + 12 + 34 * 56 + 7 + 89 +
$calc= 0 ! + 12 + 34 * 56 + 7 + 89 +

```

2020 RELEASE UNDER E.O. 14176

第13课-第一单元

[View Details](#) | [Edit](#) | [Delete](#) | [Print](#)

课时作业设计

图 1-5-1 水管井(单孔)、水塔(单)、泵房(单)

◎ 家居设计

$$S = S_1 + S_2 + S_3$$

八周基础训练

$$\text{val}(S) = \text{val}(\text{SL} + \text{str}(w_i) + \text{SH})$$

4. 检与队列

中国司法观察

四

## 优先级

◆ 焦点：如何有效地识别可优先计算的子表达式，其对应的运算符？

◆ 与括号匹配代码类似，但亦不尽相同

- 不能简单地按“左先右后”次序处理各运算符
- 此时，需参考更多因素...

◆ { 的读取规则（优先级）： $1 + 2 * 3 ^ 4$  }

可执行改变次序的括号  $(( ( 1 + 2 ) * 3 ) ^ 4)$

[Data Structures & Algorithms, Highly Interactive](#)

## 主算法

```
float evalExpr( char* s, char* &opnd ) { // 中缀表达式求值：语法规则
    Stack<float> opnd; Stack<char> opr; // 运算操作符、运算数
    opr.push( '*' ); // 尾端的 '*' 也作为头端且首先入堆
    while ( !opr.empty() ) { // 堆顶运算符字符，直至运算符耗空
        if ( !isDigit( *s ) ) { // 若当前字符为操作符，则
            readOper( s, opr ); // 读入（可能嵌套的）操作符
        } else { // 若当前字符为运算符，则根据与栈顶运算符之优先级的高低
            switch( orderBetween( opr.top(), *s ) ) { // 分别处理 ...
            }
        }
    }
    return opnd.pop(); // 弹出并返回最后的计算结果
}
```

[Data Structures & Algorithms, Highly Interactive](#)

## 延迟缓冲

◆ 仅根据表达式的前缀，不足以确定各运算符的计算次序

只有在获得足够多的后续信息之后，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

为处理第一阶段，必须提醒预读并分析更长的前缀

◆ 为此，需借助某种支持弱识别缓冲的机制...



[Data Structures & Algorithms, Highly Interactive](#)

## 优先级表

优先级	操作符
I	- + * / % ^ ( ) , ;
II	+ - * / % ^ ( ) , ;
III	* / % ^ ( ) , ;
IV	( ) , ;
V	, ;
VI	,
VII	,
VIII	,
IX	,

[Data Structures & Algorithms, Highly Interactive](#)

## 求值算法 - 栈 + 遍历扫描

◆ 由右向左扫描表达式，用栈记录已扫描的部分（含已执行计算的结果）  
在每一字符处

```
while ( 栈的顶部存在可优先计算子表达式 ) // 循环判断
    读子表达式逻辑；计算其数值；计算结果进栈
    跳转字符逻辑，转入下一字符
```

◆ 只要语法正确，栈内最终应只剩一个元素 // 子表达式对内的数据



[Data Structures & Algorithms, Highly Interactive](#)

## 不同优先级处理方法

```
switch( orderBetween( opr.top(), *s ) ) {
    case '<': // 括号运算符优先级更高
        opr.push( '*' ); s++; break; // 计算推迟，当前运算符进堆
    case '=': // 优先级相等（当前运算符为右括号，或尾部特殊‘;’）
        opr.pop(); s++; break; // 则清除非接收下一个字符
    case '>': // 括号运算符优先级更高，要进行相应的计算，读入入栈
        char op = opr.pop(); // 括号运算符出栈，执行到底的计算
        if ( '*' == op ) opnd.push( calc( op, opnd.pop() ) );
        else { float opnd2 = opnd.pop(); opnd1 = opnd.pop();
            opnd.push( calc( opnd1, op, opnd2 ) );
        } // 实施计算，结果入栈
    } // 为树不底座：opnd.push( calc( op, opnd.pop(), op, opnd.pop() ) );
    break;
} // case '>'
```

[Data Structures & Algorithms, Highly Interactive](#)

## 4. 栈与队列

## 中缀表达式求值

## 算法

## 解 读 片

[www.guitinghua.com](http://www.guitinghua.com)

## 优先级表（理解）

优先级	操作符
I	- + * / % ^ ( ) , ;
II	+ - * / % ^ ( ) , ;
III	* / % ^ ( ) , ;
IV	( ) , ;
V	, ;
VI	,
VII	,
VIII	,
IX	,

[Data Structures & Algorithms, Highly Interactive](#)



$(x + 1)^{225} \times (x^2 - 5x + 7) \div (x^2 + 8x + 7) \div (x^2 - 5x + 4)$

32  
4236  
13600  
2004

Data Structures & Algorithms, 10th Edition (c) 2012

◆ 例如： $(0 + 1)^2 \times (2^2 - 3 + 4 - 5)$

假设：事先未就运算符之间的优先级关系做过任何约定

1) 用括号显式地表示操作级

$((0 + 1)^2 \times ((2^2 - 3 + 4 - 5)))$

2) 将该算式转换到对应的右括号后

$((0 + 1)^2) \times (((2^2 - 3 + 4) - 5))$

3) 去掉所有括号

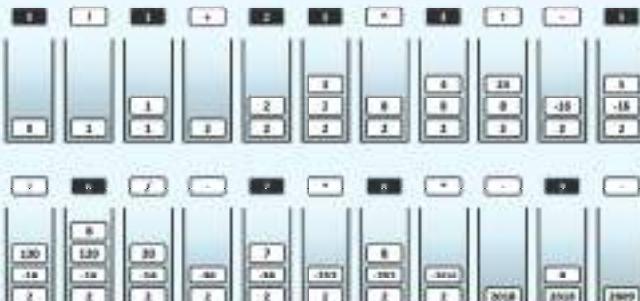
$0 + 1 ^ 2 \times (2^2 - 3 + 4) - 5$

4) 移除空格，即得

$0 + 1 ^ 2 \times 2^2 - 3 + 4 - 5$

◆ 中缀式求值算法evaluate()稍微扩展，亦可同时完成RPN转换...

$0 + 1 + 2 \cdot 3 \wedge 4 + - 5 + 6 / - 7 \cdot 8 + - 9 -$



$0 + 1 + 2 \cdot 3 \wedge 4 + - 5 + 6 / - 7 \cdot 8 + - 9 +$



◆ float evaluate(char\* s, char\*& optr) { // 手工转换

/\* ..... \*/

while (!optr.empty()) { // 每个处理一个字符，直至运算符耗空

if (isdigit(\*s)) // 若当前字符为操作数，则直接将其

{ readNumber(s, optr); append(SPN, optr.top()); } // 插入SPN

else // 若当前字符为运算符

switch( orderBetween(optr.top(), \*s) ) {

/\* ..... \*/

case '>': { // 既可直接执行，则在执行组合计算的同时将其

char op = optr.pop(); append(SPN, op); } // 插入SPN

/\* ..... \*/

} // case '>'

/\* ..... \*/

} // case '<'

/\* ..... \*/

◆ 这里的多处实例中，为何操作数都是数学表达式？

◆ “ $x + a$ ”除了可以转换为“ $x + a$ ”，是否也可转换为“ $a + x$ ”？

◆ 虽然evaluate()算法已经能够步进，同时完成RPN转换又有何意义？

◆ 相对于原表达式，存储RPN所需的空间是否更少？

◆ 在数学意义上完全对称的前缀表达式，为何在类似问题中很少应用？

#### 4. 栈与队列

逆波兰表达式  
- 转换

持彼古之，必固掌之。  
持彼夺之，必固予之。  
持彼灭之，必固学之。

邓伟群  
dengweiquan@ust.hk

#### 4. 栈与队列

逆波兰表达式  
- PostScript

邓伟群  
dengweiquan@ust.hk

## PostScript

◆ 诞生于1985，支持设备独立的图形描述

◆ `( 1 个输入器 + 5 个绘图 ) + RPN语法`

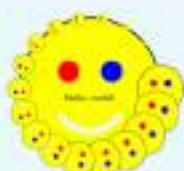
◆ operand stack：存放操作数及运算结果

◆ 一些通用操作符，如：

- 弹出相应类型的元素
- 实施计算，并
- 将（可选多个、一个或两个）结果入栈

◆ 实例：`[4 8 mul 5 5 mul add 7 mul 7 mul]`

◆ 提供基础且强大的图形功能，支持的新类型、变量、函数/宏……



## PostScript

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

## 4. 框与队列

## 队列接口与实现

邓伟群

dengwq@tongji.edu.cn

## 操作与接口

◆ 队列（queue）也叫“受限”的序列

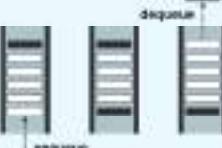
只能在队尾插入（塞进）： `enqueue()`  
`push()`

只能在队头删除（拿出）： `dequeue()`  
`pop()`

◆ 先进先出（FIFO）

后进后出（LIFO）

◆ 扩展接口：`getFront()`...



## 操作实例

操作	输出	队列（右侧为队头）
<code>Queue()</code>		
<code>empty()</code>	true	
<code>enqueue(5)</code>		5
<code>enqueue(3)</code>		3 5
<code>dequeue()</code>	5	3
<code>enqueue(7)</code>		7 3
<code>enqueue(3)</code>		3 7 3
<code>Front()</code>	3	3 7 3
<code>empty()</code>	false	3 7 3

操作	输出	队列（右侧为队头）
<code>enqueue(11)</code>		11 9 7 3
<code>size()</code>	4	11 9 7 3
<code>enqueue(6)</code>		6 11 9 7 3
<code>empty()</code>	false	6 11 9 7 3
<code>enqueue(7)</code>		7 6 11 9 7 3
<code>dequeue()</code>	1	7 6 11 9 7 3
<code>dequeue()</code>	2	7 6 11 9 7 3
<code>Front()</code>	3	7 6 11 9 7 3
<code>size()</code>	4	7 6 11 9 7 3

## 相位类

◆ 相位类属于特别的队列，故存放在接续子由继承类别派生

```
template <typename T> class Queue : public List<T> { //由列表派生的队列相位类
public: //size()与empty()由相位类
    void enqueue( T const & e ) { insertLast( e ); } //入队
    T dequeue() { return remove( first() ); } //出队
    T & front() { return first().data; } //队首
}; //以列表做/末端为队列头/尾——能快过单链？
```

◆ 检查：如此实现的队列接口，均只需O(1)时间

◆ 课堂：基于向量，派生定义队列相位类  
评论：你精实现的队列接口，效率如何？

## 4. 框与队列

## 队列应用

邓伟群

dengwq@tongji.edu.cn

## 4. 框与队列

## 队列应用

邓伟群

dengwq@tongji.edu.cn

## 操作与接口

◆ 队列（queue）也叫“受限”的序列

只能在队尾插入（塞进）： `enqueue()`  
`push()`

只能在队头删除（拿出）： `dequeue()`  
`pop()`

◆ 先进先出（FIFO）

后进后出（LIFO）

◆ 扩展接口：`getFront()`...

## 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，买微型成员共享打印机，……

◆ RoundRobin //循环分配器

`Queue< client> Q; //参与资源分配的所有客户组成队列`

`while (!ServiceClosed()) { //直到服务关闭之前，反复执行`

`c = Q.dequeue(); //令队首的客户出队，并`

`serve( c ); //给该服务，然后`

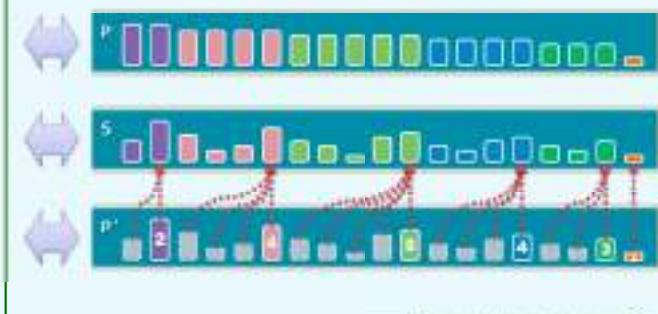
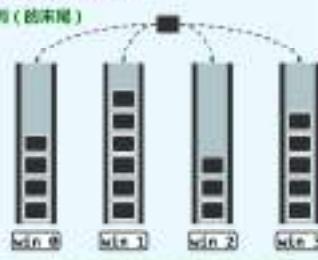
`Q.enqueue( c ); //重新入队`

`}`

◆ 利用队列改进进阶算法，找出最短的通路

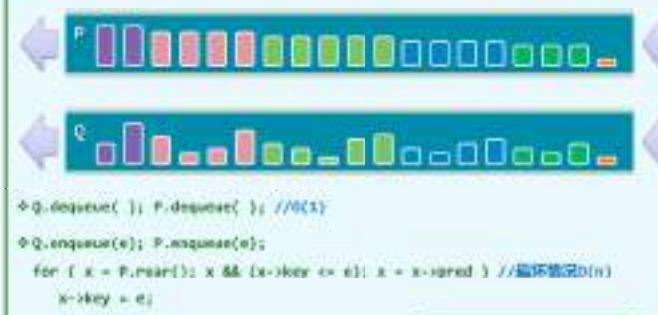


◆ 模型：提供n个服务窗口  
任一时刻，每个窗口最多接待一位顾客，其他顾客排队等候  
顾客到达后，自动地选择和加入最近队列（先进尾）  
参数：min //窗口（队列）数目  
servTime //营业时间  
  
struct Customer { //顾客类  
 int window; //所属窗口（队列）  
 unsigned int time; //服务时长  
};



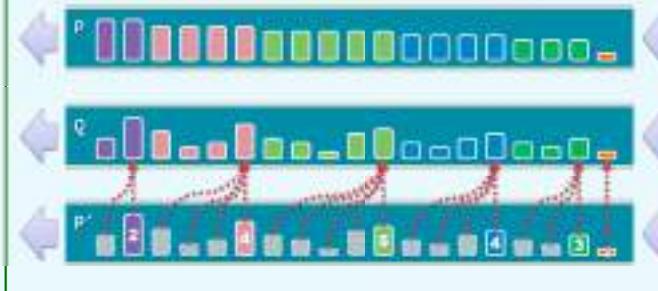
```
#include <iostream>
#include <queue>
using namespace std;

Customer* windows[5] = new Queue<Customer*>[5];
int main() {
    for (int now = 0; now < servTime; now++) { // 在下班之前，每段单位时间
        Customer c; c.time = 1 + rand() % 50; // 一位顾客到达，其服务时长随机生成
        c.window = bestWindow(windows, min); // 找出最佳（最短）服务窗口
        windows[c.window].enqueue(&c); // 顾客接入对应的队列
    }
    for (int i = 0; i < min; i++) // 分别检测
        if (!windows[i].empty()) // 若非空队列
            if (--windows[i].front().time == 0) // 从队首移除顾客
                windows[i].dequeue(); // 服务完毕则出列，由后继顾客接替
    // ...
    delete [] windows; // 销毁所有队列
}
```



```
if (Q.dequeue() || P.dequeue()); // O(1)
Q.enqueue(e); P.enqueue(e);
for (x = P.front(); x && (x->key < e); x = x->next) // 遍历链表 O(n)
    x->key = e;
```

```
#include <iostream>
#include <stack>
using namespace std;
```

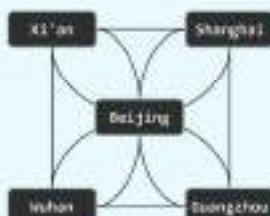


```
/* n, maxval, k, minmax, 200 */
int x, L, L1, L2, L3, M, min, max(); void main() {
    cout << "n, maxval, k, minmax, 200 ";
    cin >> n >> maxval >> k >> minmax;
    min = minmax; max = maxval;
    for (i = 0; i < n; i++) {
        for (j = 0; j < k; j++) {
            for (l = 0; l < maxval; l++) {
                if (min <= l <= max) {
                    cout << l << " ";
                }
            }
        }
    }
}
```

```
#include <iostream>
using namespace std;
```

## 指教操作

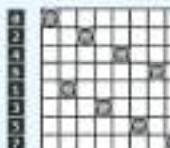
- 很多问题的解，形式上都可看作若干元素按一定次序构成的序列
- 以TSP问题为例，即  
给定n个城市之间总成本最低的环游路线
- 每一对列组合都是一个候选解  
往往构成一个很大的搜索空间
- 所以TSP为例，共有： $n! / n = (n-1)!$  =  $O(n!)$
- 若采用暴力策略求解  
需逐一生成可能的候选解，并检查其是否合理  
如此，必须无须将所有候选解都留在多项式以内



Data Structures &amp; Algorithms, Chapter 10 (Answers)

## 编译

- 规则：每行（列）有且仅有一个皇后
- 因此，每一布局（解）都可编码为数组 $[0, \dots, n-1]$ 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, Chapter 10 (Answers)

## 试探-回溯-剪枝

- 为解决迷宫，我们也是使用递归找解，但深搜法解出相对慢，并利用特有的便捷
- 事实上，根据候选解的某种 **剪枝特征**，即可判断解是否合理  
此时只要剪枝得当，便可成功地排除错误解
- 此即所谓剪枝（pruning）
- 试探回溯（probe-backtrack）模式  
从0开始，逐层地加候选解长度 //试探  
一旦发现定位失败，则  
收缩某前一长度，并 //回溯到那  
继续试探
- 待被剪的结点 - 结果 + 相关  
如何以数据结构的形式呈现？



Data Structures &amp; Algorithms, Chapter 10 (Answers)

## 暴力搜索

```

void nQueens_BruteForce() { //暴力搜索法
    int soln[4]; //解矩阵存储器
    for (soln[0] = 0; soln[0] < 4; soln[0]++)
        for (soln[1] = 0; soln[1] < 4; soln[1]++)
            for (soln[2] = 0; soln[2] < 4; soln[2]++)
                for (soln[3] = 0; soln[3] < 4; soln[3]++) { //枚举所有候选解
                    if (!collide(soln, 0)) continue;
                    if (!collide(soln, 1)) continue;
                    if (!collide(soln, 2)) continue;
                    if (!collide(soln, 3)) continue;
                    else++; displaySolution(soln, 4);
                }
}
    
```

//暴力搜索 $4^4 = O(n^4)$

Data Structures &amp; Algorithms, Chapter 10 (Answers)

## 八皇后

- 在 $n \times n$ 的棋盘上放 $n$ 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？
- $n = 2, 3, 4, 5, \dots$   
允许重解： 1, 0, 0, 1, 10, 4, 40, 92, ...  
不允许解： 1, 0, 0, 1, 2, 1, 8, 22, 48, 92, ...



Data Structures &amp; Algorithms, Chapter 10 (Answers)

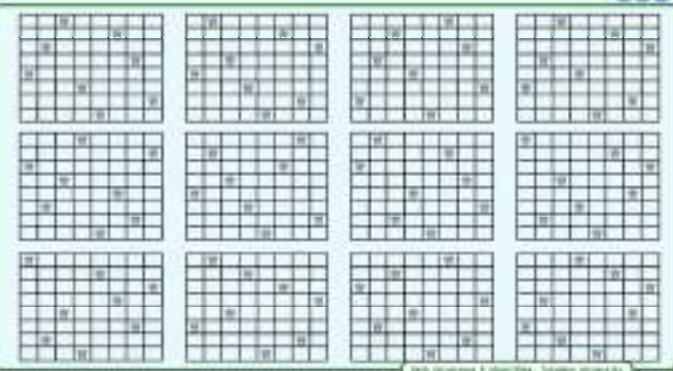
## 剪枝

```

void placeQueen() { //八皇后剪枝算法
    int soln[4]; //解矩阵存储器
    for (soln[0] = 0; soln[0] < 4; soln[0]++)
        if (!collide(soln, 0)) { //剪枝
            for (soln[1] = 0; soln[1] < 4; soln[1]++)
                if (!collide(soln, 1)) { //剪枝
                    for (soln[2] = 0; soln[2] < 4; soln[2]++)
                        if (!collide(soln, 2)) { //剪枝
                            for (soln[3] = 0; soln[3] < 4; soln[3]++)
                                if (!collide(soln, 3)) { //剪枝
                                    nSols++; displaySolution(soln, 4);
                                }
                            }
                        }
                    }
    }
    
```

//复杂度大大降低，但冒泡的副作用变高

Data Structures &amp; Algorithms, Chapter 10 (Answers)



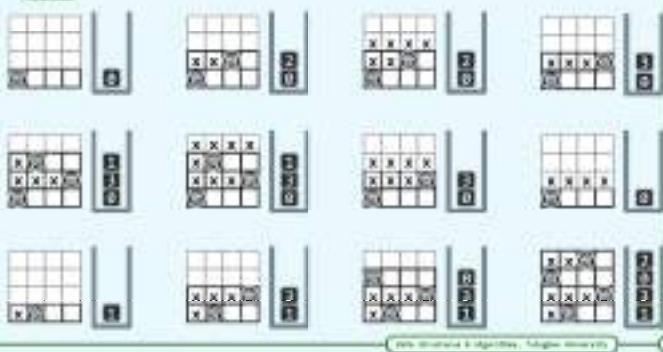
## 通用算法

```

void placeQueen(int N) { //N = 列数大小。皇后总数，向后的规模可任意
    Stack<Queen> soln; Queen q(0, 0); //存放(部分)解的线，从底向顶部出栈
    do { //粗略地讲，回溯
        if (0 == soln.size() || N >= soln.y) { //若已出界，则
            q = soln.pop(); //回溯一行，再继续进下一列
        } else { //否则，进接下来一行
            while (0 <= q.y < N && 0 == soln.push(q)) { //通过与已有皇后的位置
                q.y++; //尝试摆放可能的下一通后的列
            }
            if (N > q.x) { //若能在可摆放的列，摆线上再放置后
                soln.push(q); if (0 == soln.size()) soln++; //若所有的行已成全局解，则计数
                q.y++; q.y = 0; //转入下一行，从第0行开始，进解下一层后
            }
        }
    } while ((0 < q.x) || (q.y > N)); //直到所有分支均已检测完而回溯
}
    
```

Data Structures &amp; Algorithms, Chapter 10 (Answers)

## 实例



## 算法

```

bool isLegal(int cellIndex[Max], cellIndex[Max], cellIndex[Max], cellIndex[Max])
{
    Stack<Cell*> path; //用来记录遍历(Thessal)的结构
    int cNeighboring = UNKNOWN; cStatus = ROUTE; path.push(c);
    do { //不重结果，因为，直到到达终点，遍历过程都有可能
        Cell* c = path.back(); if (c == t) return true; //如果触碰，找到...
        while (!NO_WAY) { c->neighboring = nextESM(c->neighboring) + 1; //遍历另一
            if (AVAILABLE == neighbor(c)->status) break; //遍历试探的方向
            if (NO_WAY < c->neighboring) //若所有方向都已尝试，向内后退一步
                c->status = BACKTRACKED; c = path.pop(); } //从Thessal的接壤
        else //否则，向前试探一步
            { path.push(c->neighbor(c)); c->outgoing = UNKNOWN; c->status = ROUTE; }
    } while (!path.empty());
    return false;
}

```

## 数据结构

在上面的方法中的“`cellIndex[i]`”如何到用接（函数）的声明接口？

◆ 定义皇后类Queen，重新定义判断器，使之在语义上与冲突廉价

```

struct Queen { //皇后类
    int x, y; Queen( int xx = 0, int yy = 0 ) : x(xx), y(yy) {} //类型的构造
    bool operator==( Queen const & q ) { //直接判断操作
        return ( x == q.x ) //行冲突(不会发生, 可忽略)
            || ( y == q.y ) //列冲突
            || ( x + y == q.x + q.y ) //左对角线冲突
            || ( x - y == q.x - q.y ); //右对角线冲突
    }
    bool operator!=( Queen const & q ) { return !( *this == q ); }
}

```

## 数据结构

```

struct Cell { //单个单元
    int x, y; //坐标
    Status status; //类型
    ESM incoming, outgoing; //进入、退出方向
}

```

## 相关类型

```

//状态：可用，在当前路径上，所有方向均尝试失败后就通过。不可使用（堵）
typedef enum { AVAILABLE, ROUTE, BACKTRACKED, NULL } Status;
//单元的相对邻接方向：东定、东、南、西、北、无路可通
typedef enum { UNKNOWN, EAST, SOUTH, WEST, NORTH, NO_WAY } ESM;
//依次排列下一邻接方向
inline ESM nextESM( ESM esm ) { return ESM( esm + 1 ); }

```

## 4. 桥与队列

试探回溯法：迷宫寻径

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

邓佳辉

[dengjiahui.sina.com.cn](http://dengjiahui.sina.com.cn)

## 进一步的考虑

◆ 如何统计耗时情况的概率？

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？

环境：尽可能发现开路去

向路：尽可能发现开路去

贪心：优先方向优先试探

...

◆ 迷宫首法



## 5. 二叉树

邓

Two roads diverged in a yellow wood  
And sorry I could not travel both

邓佳辉

[dengjiahui.sina.com.cn](http://dengjiahui.sina.com.cn)

## 迷宫寻径

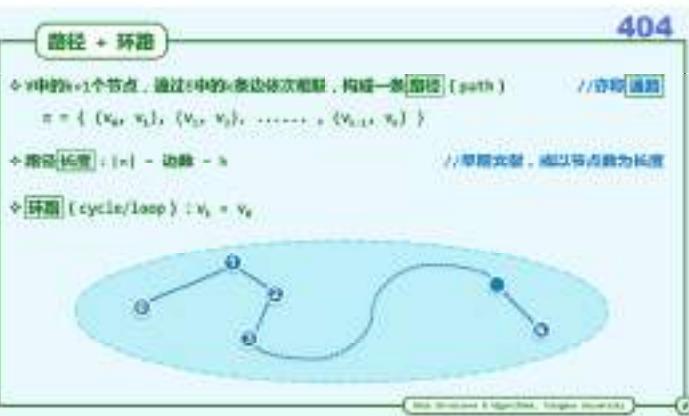
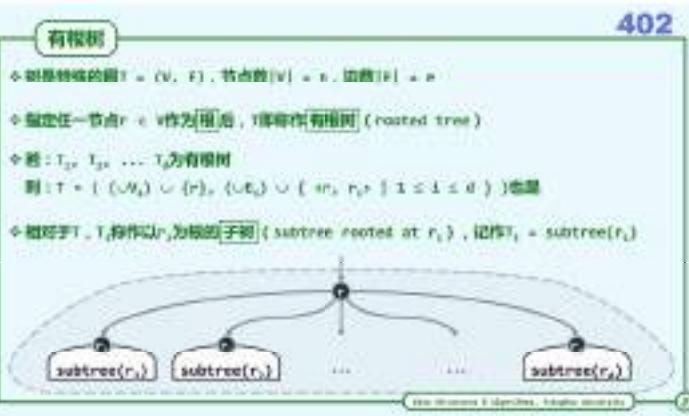
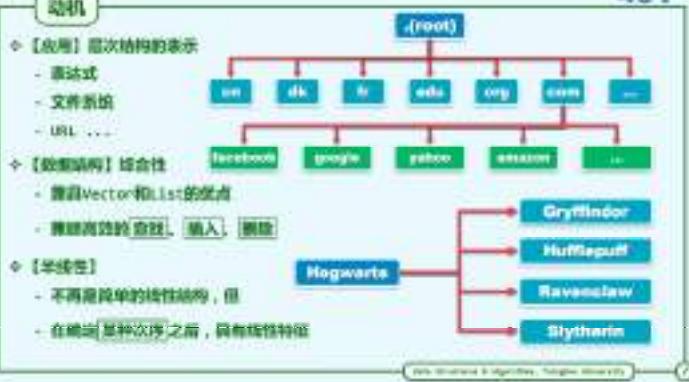
◆ 路径规划（path planning），系人工智能等领域的基本问题

◆ n\*n方格组成的迷宫，西南方格为起始墙，中间若干方格为障碍墙

机器人漫游其间，每步只能运动到东、南、西、北方向的某一邻格

◆ 确定的起点s和终点t，在其间找出一条西连通的通路（如果存在）



**5. 二叉树**

树的表示

单链表

zhangtinghua@csail.mit.edu

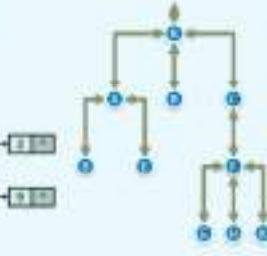
接口

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

Data Structures &amp; Algorithms, 10th Edition (c) 2012

父节点 + 孩子节点

	data	parent	children
0	A	-	1 2 3
1	B	0	-
2	C	0	-
3	D	-	-
4	E	0	5 6 7
5	F	4	-
6	G	4	-
7	H	4	-
8	I	-	-



Data Structures &amp; Algorithms, 10th Edition (c) 2012

父节点

◆ 父节点：除根外，任一节点有且仅有一个父节点

◆ 想象：将节点组织为序列，各节点分别记录

[data] 本身数据

[parent] 父节点的树的引用

◆ 根节点(0)也有“指向”的父节点

parent(0) = -1 或

parent(0) = NULL

id	data	parent	children
0	A	-1	1 2 3
1	B	0	-
2	C	0	-
3	D	0	-
4	E	1	-
5	F	4	-
6	G	4	-
7	H	4	-
8	I	0	-

Data Structures &amp; Algorithms, 10th Edition (c) 2012

长子 + 兄弟

◆ 每个节点均设两个引用

纵：firstChild()

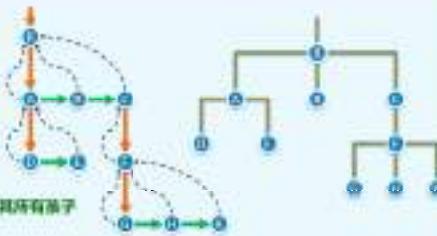
横：nextSibling()

◆ 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()函数也仅需 $O(1)$ 时间

Data Structures &amp; Algorithms, 10th Edition (c) 2012

父节点

◆ 空间复杂度： $O(n)$ 

◆ 时间性能

① parent():  $O(1)$ ② root():  $O(n)$ 或 $O(1)$ ③ firstChild():  $O(n)$ ④ nextSibling():  $O(n)$ 

◆ 动态操作姑且不论，首先

如何加速对孩子、兄弟的遍历？

id	data	parent	children
0	A	-1	1 2 3
1	B	0	-
2	C	0	-
3	D	0	-
4	E	1	-
5	F	4	-
6	G	4	-
7	H	4	-
8	I	0	-

Data Structures &amp; Algorithms, 10th Edition (c) 2012

## 5. 二叉树

有根有序树 = 二叉树

宝生抱怨不安本分之人，竟一味地随心所欲。因此又发了脾气，又对内蒙种植园说：“咱们俩个人一样的年纪，况又同胞，以前不必论亲臣，只论弟兄朋友就是了。”

平生好

http://tiny.cc/meyarw

孩子节点

◆ 同一节点的所有孩子，组织为一个序列

◆ 序列的长度，分别等于对应节点的孩子数

	data	children
0	A	1 2 3
1	B	-
2	C	4
3	D	-
4	E	5 6 7
5	F	-
6	G	-
7	H	-
8	I	-

Data Structures &amp; Algorithms, 10th Edition (c) 2012

二叉树

◆ 节点函数不相干的树

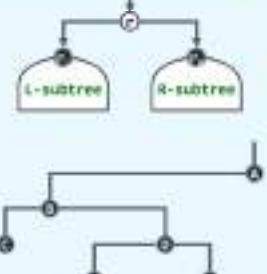
称作二叉树 (binary tree)

◆ 同一节点的孩子和子树，均以左/右区分

left() ~ lSubtree()

right() ~ rSubtree()

完全有序



Data Structures &amp; Algorithms, 10th Edition (c) 2012

## 基数

◆ 基数为k的节点，高度h个

◆ 高h个节点，高度为k的二叉树中

$$E \leq k \leq 2^{h-1}$$

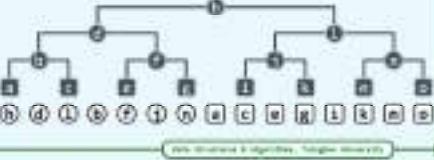
$$1) n = h + 1$$

转化为一条单链

$$2) n = 2^{h-1} - 1$$

即完全二叉树

(full binary tree)



## 5. 二叉树

## 二叉树实现

样例解

Data Structures &amp; Algorithms, Design Patterns

## 基数

◆ 基数为0、1和2的节点，高度n<sub>0</sub>、n<sub>1</sub>和n<sub>2</sub>个

$$\Delta \text{边数 } E = n_0 + n_1 + n_2 = n_0 + 2n_1$$

1/2度节点各对应于1/2度插入边

$$\Delta \text{叶节点数 } n_0 = n_1 + 1$$

n<sub>0</sub>与n<sub>1</sub>无关

h = 0时，1 = n + 1；此后，n<sub>0</sub>与n<sub>1</sub>同步递增

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

◆ 特别地，当n<sub>1</sub> = 0时，有

$$n = 2n_2 \text{ 和 } n_0 + n_1 + 1 = (n + 1)/2$$

此时，节点总数均为偶数，不含单分支节点...



## BinNode模板类

◆ addFirst BinNodePosi(T) BinNode<T>\* //节点位置

◆ template <typename T> BinNode<T>

BinNodePosi(T) parent, lc, rc; //父类，子类指针

T data; int height; int size(); //高度，子树规模

BinNodePosi(T) insertFirstChild(T const&); //作为左孩子插入新节点

BinNodePosi(T) insertSecondChild(T const&); //作为右孩子插入新节点

BinNodePosi(T) succ(); // (中序遍历情况下) 当前节点的直接后继

template <typename VST> void traverse(VST&); //子树层次遍历

template <typename VST> void traverseLeft(VST&); //子树先序遍历

template <typename VST> void traverseRight(VST&); //子树后序遍历



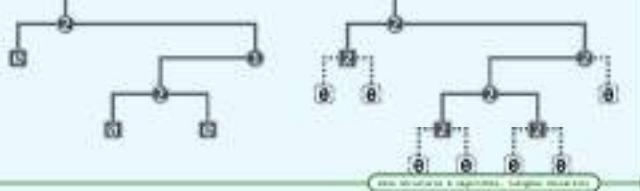
## 真二叉树

◆ 通过引入n<sub>0</sub> + 2n<sub>2</sub>个外部节点，可使原有节点都统一为2

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

◆ 注意：在此转换之后，全树自身的操作完全未变而增加

◆ 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



## BinNode接口实现

◆ template <typename T> BinNodePosi(T) BinNode<T>::insertLast(T const& e)

{ return lc = new BinNode<e>(e, this); }

◆ template <typename T> BinNodePosi(T) BinNode<T>::insertLastLeft(T const& e)

{ return rc = new BinNode<e>(e, this); }

◆ template <typename T>

int BinNode<T>::size() { //返回总数，亦即以其为根的子树总规模

int s = 1; //计入本身

if (lc) s += lc->size(); //递归计入左子树规模

if (rc) s += rc->size(); //递归计入右子树规模

return s;

} //O( n <= |size| )

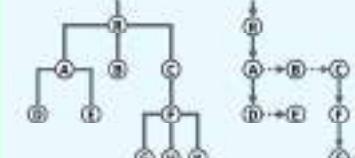


## 描述多叉树

◆ 二叉树是多叉树的特例，但在有根真二叉树时，其能效能力却足以超越所有

◆ 多叉树向可转化为二叉树——回忆长子-兄弟表示法...

+ 长子 - 左孩子 firstChild() ~ lc();  
兄弟 - 右孩子 nextSibling() ~ rc();



## 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) \*); //更新节点的高度

void updateHeightAbove(BinNodePosi(T) \*); //更新x及其祖先的高度

public:

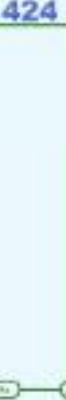
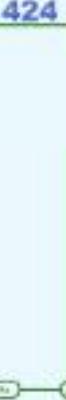
int size() const { return \_size; } //根根

bool empty() const { return !\_root; } //空空

BinNodePosi(T) root() const { return \_root; } //根根

/\* ... 子树插入、删除和分离接口 ... \*/

/\* ... 插入接口 ... \*/



**高度更新**

```

#define stature(p) ((p) ? (p)->height : -1) //节点高度——约定空树高度为-1

template <typename T> //更新节点x高度，具体规则因树不同而异
int BinTree<T>::updateHeight( #BinNodePos1(T) x ) {
    return x->height = 2 + max( stature( x->lc ), stature( x->rc ) );
} //此选择用常量二叉树规则，O(1)

template <typename T> //更新v及所有祖先的高度
void BinTree<T>::updateHeightAbove( #BinNodePos1(T) x ) {
    while (x) //可优化：一但高度未变，即可终止
        { updateHeight(x); x = x->parent; }
} //O( n = depth(x) )

```

◆ 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，面对分离出来的子树重新计算，并返回给上层调用者

```

template <typename T> BinTree<T>* BinTree<T>::secede( #BinNodePos1(T) x ) {
    FromParentTo( * x ) = NULL; //切断来自父节点的链接
    updateHeightAbove( x->parent ); //更新整棵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; //剪断以x为根
    S->size = x->size(); _size -= S->size; //更新规模
    return S; //返回对整树的子树
}

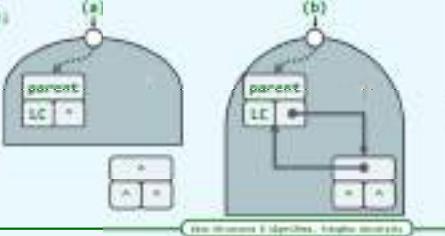
```

**节点插入**

```

template <typename T> #BinNodePos1(T)
BinTree<T>::insertAsRC( #BinNodePos1(T) x, T const& b ) { //insertAsRC()对称
    _size++; x->insertAsRC( b ); //祖先的高度可能增加，其余节点不变
    updateHeightAbove(x);
    return x->rc;
}

```



## 5. 二叉树

先序遍历  
遍历

单佳辉

dengtinghua@sohu.com

萧何曰：“西昌羽林擒卢山西侯，魏武子授元勋而称道，今谁无故苦勤之？”  
项目：“有，但当速归。”

**子树插入**

```

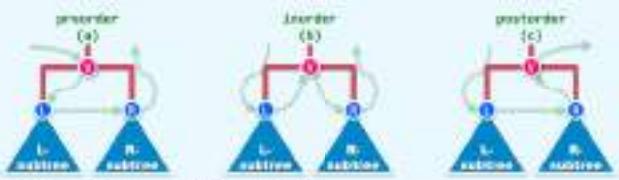
template <typename T>
#BinNodePos1(T) BinTree<T>::attachAsL( #BinNodePos1(T) x, BinTree<T>* S ) {
    if ( x->rc == S->root ) x->rc->parent = x; //插入
    _size += S->size; //更新规模
    updateHeightAbove(x); //更新祖先高度
    S->root = NULL; S->size = 0;
    release(S); S = NULL; //释放S
    return x; //返回插入位置
} //attachAsL()对称

```



◆ 按照某种次序访问所有子节点，每个节点都访问到一次：T = L ∪ R ∪ E

- 先序      中序      后序      层次（广度）
- L | R | E      L | R | E      L | R | E      层上而下，先左后右



◦ 遍历结果 = 遍历过程 = 遍历次序 = 遍历策略

**子树删除**

```

template <typename T>
int BinTree<T>::remove( #BinNodePos1(T) x ) { //子树插入的逆过程
    FromParentTo( * x ) = NULL; //切断来自父节点的链接
    updateHeightAbove( x->parent ); //更新祖先高度（其余节点不变）
    int n = remove(x); _size -= n; //通过删除x及其后代，更新规模
    return x; //返回被删除节点总数
}

template <typename T> static int removeAll( #BinNodePos1(T) x ) {
    if (!x) return 0; //禁止为空树，否则左、右遍历
    int n = 1 + removeAll( x->lc ) + removeAll( x->rc );
    release(x->data); release(x); return n; //释放被删除节点，并返回被删除节点总数
}

```

## 5. 二叉树

先序遍历  
算法A

单佳辉

dengtinghua@sohu.com

```
template <typename T, typename VST>
void traverse( BinNodePosi(T) x, VST & visit ) {
    if ( !x ) return;
    visit( x->data );
    traverse( x->Lc ), visit );
    traverse( x->rc ), visit );
} //T(n) = O(1) + T(n) + T(n - n - 1) = O(n)
```

先序输出文件树结构：[tiny.cc/meyarw](http://tiny.cc/meyarw)

挑战：不依赖遍历机制，能否实现先序遍历？如何实现？效率如何？

## 无遗漏：

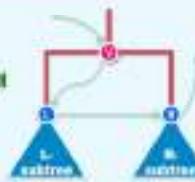
- 每个节点都会被访问到
- 归纳假设：若深度为*i*的节点都已经被正确访问，则深度为*i+1*的也属

## 祖先：

- 对于任一子树，必须访问而才会访问其它节点
- 只需注意到：若*x*是*y*的祖先，则*x*先于*y*被访问

## 左先右后：

- 同一节点的左子树，先于右子树被访问



[tiny.cc/meyarw](http://tiny.cc/meyarw)

先序遍历二叉树的过程，无非是：

先访问根节点

再先后遍历左子树和右子树

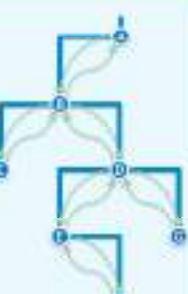
通过实现，对左、右子树的遍历调用

都类似于递归

就不难理解啦

想通：

二分递归 → 递归 + 单递归 → 递归 + 循



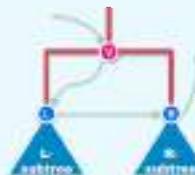
[tiny.cc/meyarw](http://tiny.cc/meyarw)

④  $\Theta(n)$

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

⑤ 以上的结论进行的思考不易推广

需要另外想法...



[tiny.cc/meyarw](http://tiny.cc/meyarw)

```
template <typename T, typename VST>
void traverse_ll( BinNodePosi(T) x, VST & visit ) {
    Stack<BinNodePosi(T)> S; // 领域性
    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 ( !x->Lc ) S.push( x->rc ); // 右孩子先入后出
        if ( !x->rc ) S.push( x->Lc ); // 左孩子后入先出
    } // 学会以上所有的技巧
}
```



[tiny.cc/meyarw](http://tiny.cc/meyarw)

## 5. 二叉树

先序遍历

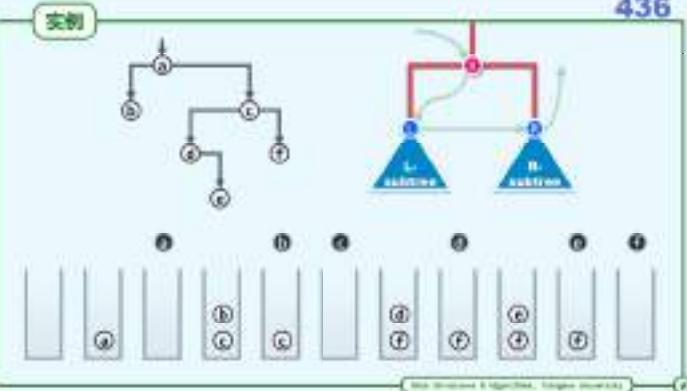
观察

一般书中的简短与美妙，通常并不放在显眼的地方。有图是在画面。只不过，一般人没有兴趣。要想成为一个好的程序员，首先要必须学会观察...

曾国华所言：“做点与众不同的事，不过这样来学习者甚之。下过一二雨则又反之。余吾重农之辈，则无所可教矣。”  
宋濂所言：倘若下过一二雨后，真就叫着“做点十二雨翻”，又一个叫着“做点十二双义翻”。

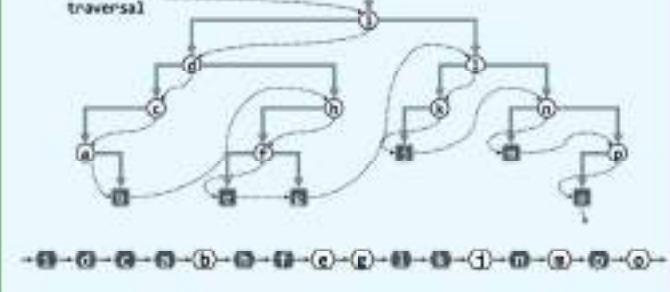
平生好

[tiny.cc/meyarw](http://tiny.cc/meyarw)



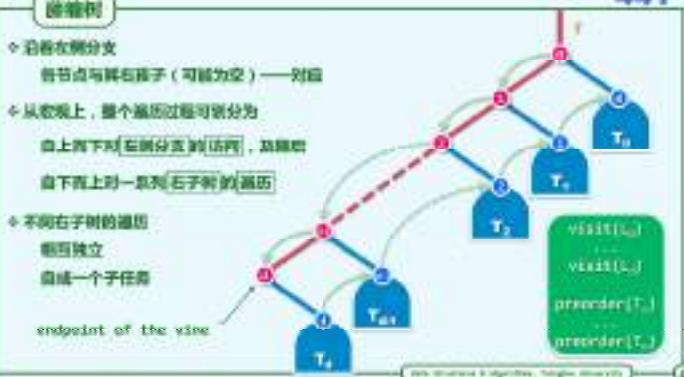
[tiny.cc/meyarw](http://tiny.cc/meyarw)

preorder traversal



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

[tiny.cc/meyarw](http://tiny.cc/meyarw)

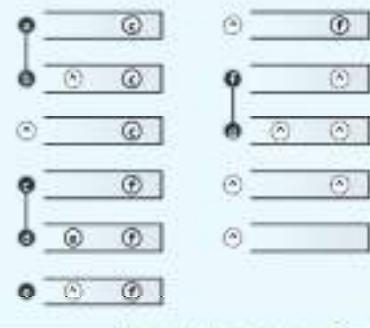
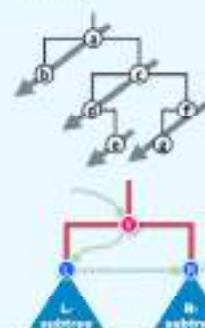


## 5. 二叉树

先序遍历  
算法B

邓佳群  
dengj@tsinghua.edu.cn

## 实列



## 5. 二叉树

中序遍历  
观察

邓佳群  
dengj@tsinghua.edu.cn

```
template <typename T, typename VST>
static void visitAlongLeftBranch
< BiNodePosi(T) x, VST & visit, Stack < BiNodePosi(T) > S > {
    while (x) { //反向地
        visit( x->data ); //访问当前节点
        S.push( x->Lc ); //右孩子（右子树）入栈（后来进先出线）
        x = x->Lc; //沿左指针下行
    } //只有右孩子，NULL可插入最后一项以断续后督，是否值得？
}
```

## 递归

```
template <typename T, typename VST>
void traversal( BiNodePosi(T) x, VST & visit ) {
    if (!x) return;
    traversal( x->Lc, visit );
    visit( x->data );
    traversal( x->Rc, visit );
} //T(n) = T(a) + O(1) + T(n-a-1) = O(n)
```

中序遍历文件树结构：printBisTree()  
挑战：不使用递归机制，能否实现中序遍历？如何实现？效率如何？

```
template <typename T, typename VST>
void travPre_L( BiNodePosi(T) x, VST & visit ) {
    Stack < BiNodePosi(T) > S; //辅助栈
    while ( true ) { //以「右」子树为单位，逆推访问节点
        visitAlongLeftBranch( x, visit, S ); //访问子树x的左附属，右子树入栈操作
        if ( S.empty() ) break; //栈空跳出
        x = S.pop(); //弹出下一子树的根
    } //pop = top = visit = O(n) = 分层点(1)
}
```

## 难点

难点在于
 

- 尽管右子树的遍历通过调用递归，但在子树却严格地不是

**解决方法**

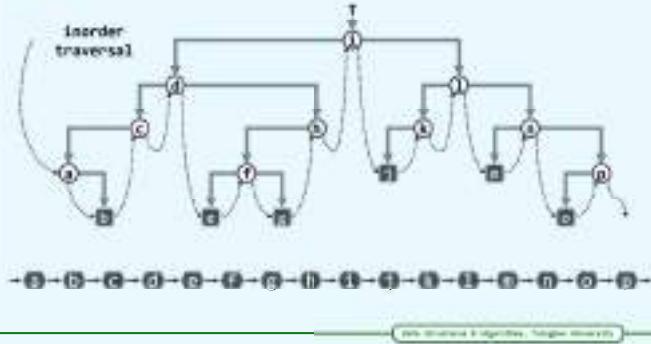
- 找到第一个遍历到的节点 //根据迭代的先序遍历算法
- 将其祖先用栈保存 //按照按访问过程的逆序

**进阶，原问题就被分解为**

- 依次对若干棵右子树的遍历问题 //为什么“次”？

**于是，首先能解决的问题就是：**

- 中序遍历任一二叉树T时
- 首先被访问的是哪个节点？如何识别它？



```
template <typename T, typename Vs>
void travIn_II( BiNodePosi<T> x, Vs visit ) {
    Stack < BiNodePosi<T> > S; //辅助栈
    while ( true ) { //无限循环
        goAlongLeftBranch( x, S ); //从当前节点出发，遍历左子树
        if ( S.empty() ) break; //当栈所有节点处理完毕
        x = S.pop(); //x的左子树将为空，或已遍历（等效于空），故可以
        visit( x->data ); //立即访问之
        x = x->rCh; //再指向其右子树（可能为空，留意处理手法）
    }
}
```

## 做题树

从根出发沿左分支下行，遇到最深的节点  
——它就是全树最先被访问者

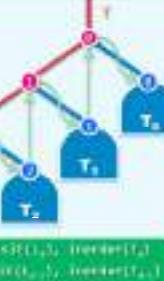
从宏观上，整个遍历过程可分为d+1步迭代  
访问L<sub>0</sub>，再遍历T<sub>1</sub>，k = d, ..., 2, 1, 0

不同右子树的遍历

相互独立

自成一个子任务

endpoint of the vine



## 5. 二叉树

中序遍历  
实例

李桂群

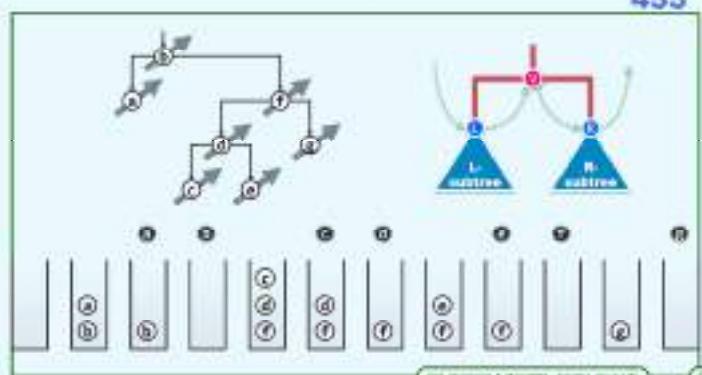
<http://www.csinghua.edu.cn>

5. 二叉树  
中序遍历  
迭代算法

李桂群

<http://www.csinghua.edu.cn>

```
template <typename T> static void
goAlongLeftBranch( BiNodePosi<T> x, Stack <BiNodePosi<T>> &S ) {
    while ( x ) {
        S.push( *x );
        x = x->lCh;
    }
} //反向地入栈，因为分支深入
```



5. 二叉树  
中序遍历  
分析

李桂群

<http://www.csinghua.edu.cn>

正确性

- ◆ 可归纳证明：
  - 每个节点出栈时
    - 其左子树（若存在）已被完全遍历，因二叉树尚未入栈
  - 于是，唯有有节点出栈，只阅读它，然后从其右孩子出发...



457

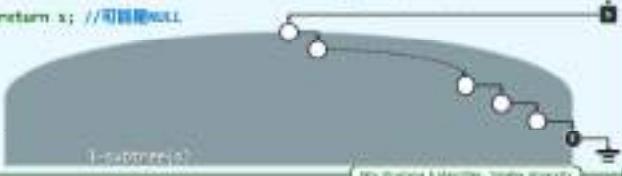
461

直接后继

- ```

    } else { //否则，后继指的是“把当前节点包含于其左子树中的最小值”
        while (!IsLeftChild(*x)) //根节点在左方
            x = x->parent; //逐级向右的分支，不断将左上方移动
        x = x->parent; //最后再向左上方移动一步，操作达后继（若存在）
    } //两种情况的运行时间分别为当前节点的高度与深度，不过 $O(h)$ 
    return x; //可能为NULL
}

```



效率

- ◆ 是否 $O(n)$ ，取决于以下条件
  - 1) 每次迭代，新选出一个节点出栈并被访问 //满足
  - 2) 每个节点入选一次且仅一次 //满足
  - 3) 每次迭代只需 $O(1)$ 时间 //不再满足，因为...
- ◆ 单次遍历`goAlongLeftBranch()`
  - 很可能需要做 $O(n)$ 次入栈操作，并非 $O(n)$ 时间
- ◆ 虽然如此，难道总体将需要... $O(n^2)$ 时间？
- ◆ 事实上，这个界远远不紧...
  - 使用先分摊原理，自行分析
- ◆ 更多的证明：`travIn_12()` + `travIn_13()` + `travIn_14()`

458

462

## 5. 二叉树

### 中序遍历 后继与前驱

邓伟群

<http://www.dengweiqun.com>

递归

- ```

template <typename T, typename VST>
void traverse(BinNodePosist(T)* x, VST & visit) {
    if (!x) return;
    traverse(x->lch, visit);
    traverse(x->rch, visit);
    visit(x->data);
} //T(n) = T(x) + T(n - x - 1) + O(1) + O(n)
//提醒：BinNode::size()
//BinTree::updateHeight()

```

//注意：

平体递归机制，能否实现后序遍历？如何实现？效率如何？



460

464

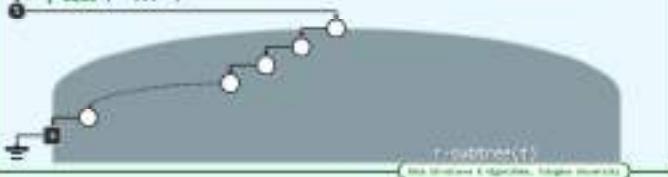
难点

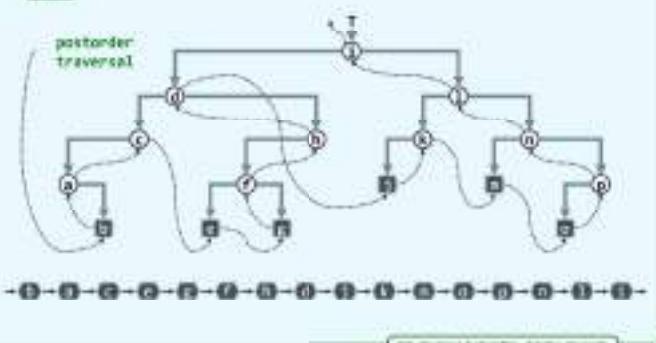
- ◆ 难度在于
  - 对左，右子树的递归调用，但不是遍历
- ◆ 解决方法
  - 找到第一个被访问的节点
  - 将祖先及其右兄弟（如果存在）用线保存
- ◆ 这样，原问题就被分解为
  - 依次对若干颗右子树的遍历问题 //同样地，这里应该叫什么“次”？
- ◆ 于是，首先能解决的问题初现
  - 后序遍历任一二叉树时，最先被访问的需要哪个节点？如何找到它？

```

template <typename Tr> //稍后将探讨BT::remove中的removeAt()调用
BinNodePosist(T) BinNode<Tr>::succ() { //在中序遍历语义下的直接后继
    BinNodePosist(T) s = this; //记录后继的临时变量
    if (!rc) { //若有右孩子，则直接后继必在右子树中，具体地就是
        s = rc; while (HasLChild(*s)) s = s->lch; //右子树中最小节点
    } else /* ... */
}

```





```
template <typename T, typename VST>
void travPost_I( BinNodePosi<T> x, VST & visit ) {
    stack < BinNodePosi<T> > S; // 链队列
    if ( !x || S.empty() ) { // 根节点非空且栈为空
        while ( !S.empty() ) { // x为当前节点
            if ( S.top() == x->parent ) // 检查x之父（则必为孩子）
                gotoHLVFL( S ); // 在x的右子树中，找到HVL
            x = S.pop(); // 弹出栈顶（当前一节点之后继）以更新x，并继续
            visit( x->data ); // 访问x
        }
    }
}
```



## 5. 二叉树

### 后序遍历 实例

祖父出来  
我就是你哥哥  
长长的，缠绵悱恻  
太缠绵悱恻的大树

邓佳群  
[dengjiaqun.sina.com.cn](http://dengjiaqun.sina.com.cn)



5. 二叉树  
后序遍历  
递归算法

```
template <typename T> static void gotoHLVFL( stack <BinNodePosi<T>> & S ) {
    while ( BinNodePosi<T> ( x = S.top() ) ) // 由深而下反向检查链接节点
        if ( HasLeft( * x ) ) { // 若可能向左，在此之前
            if ( HasRight( * x ) ) // 若有右孩子，则
                S.push( x->right ); // 先入右
            S.push( x->left ); // 然后转向左孩子
        } else // 不不得已
            S.push( x->right ); // 才转向右孩子
    S.pop(); // 遍历之后，弹出栈顶的空节点
}
```



## 5. 二叉树 后序遍历 分析

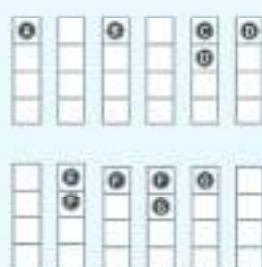
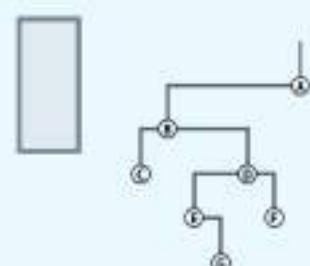
单佳辉  
[singlejiahui.com](http://singlejiahui.com)

## Expression Tree

◆ 由前缀表达式构建表达式树：

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

&lt;math



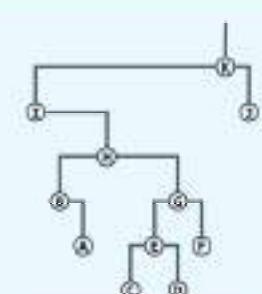
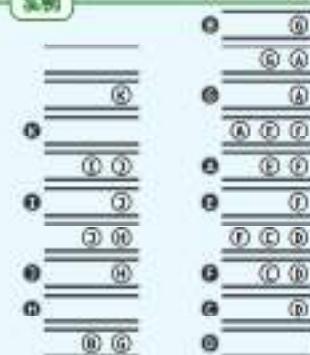
效率如何？

每次迭代

- 都有一个节点出队并接受访问
- 也可能有两个节点入队

更新队列

每个节点入、出队各恰好一次

整体效率 =  $O(n)$ 

## 5. 二叉树

层次遍历

完全二叉树

样例

## 5. 二叉树

层次遍历  
分析

样例

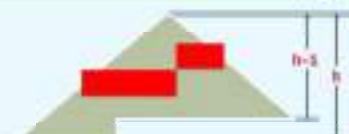
叶节点仅限于最低两层

底层叶子，均属于次底层叶子左侧

除末节点的父亲，内部节点均有双子

叶节点

- 不少于内部节点，但
- 最多出一个



- ◆ 可以见得：以上迭代算法符合广度优先遍历的规则...
- ◆ 每次迭代， 入队节点（若存在）都提出队节点的孩子，深度增加一层
- ◆ 任何时候， 队列中各节点按照的单向排列，而且
  - （相邻）节点的深度相差不超过1层
- ◆ 进一步地， 所有节点过早都会入队，而且
  - 更左/低的节点，更早/先入队
  - 更右/高的节点，更晚/后入队

考察遍历过程中的n步迭代...

第  $\frac{n}{2}$  - 1 步迭代中，都有  $\frac{1}{2}$  孩子入队第  $\frac{n}{2}$  步迭代中，都剩下  $\frac{1}{2}$  孩子入队累计至少  $n - \frac{n}{2}$  次入队

## 题次遍历

489

◆ 考察遍历过程中的 n 步迭代...

◆ 遍历队列的根根

- 先增后减, 单纯对称

- 最大深度 =  $\lceil \log_2 n \rceil$ (前  $\lceil \log_2 n \rceil - 1$  次取出入队)

- 最大深度

可能出现两次



Data Structures &amp; Algorithms, Tonghua University

## 5. 二叉树

Huffman 编码树

PFC 编码

初读之不解,熟之不释,或得其,或不得,  
小学而大悟。吾未见解也

林伟群

dongtang@tonghua.edu.cn

## 5. 二叉树

结构

林伟群

dongtang@tonghua.edu.cn

490

## [ 先序 | 后序 ] + 中序

逆序遍历

491

## [ 先序 + 后序 ] \* 真

逆序遍历

492

## 应用

◆ 通讯 / 编码 / 译码



◆ 二进制编码

- 将成数据文件的字符来自字符集:

- 字符被赋予互异的二进制串

"1010 0110 00" = "MATH"

◆ 文件的大小取决于

- 字符的数量。各字符编码的长短

◆ 通讯带宽有限时

- 如何对些字符串,使文件最小?

Data Structures &amp; Algorithms, Tonghua University

494

## 二叉编码树

◆ 树中的字符串构造一棵二叉树

以 0/1 表示左/右孩子

字符串 x 分别存于子树对应的叶子 v(x) 中

◆ 字符 x 的编码串 rps( v(x) ) = rps(x)

由根到 v(x) 的路径 (root-path) 确定

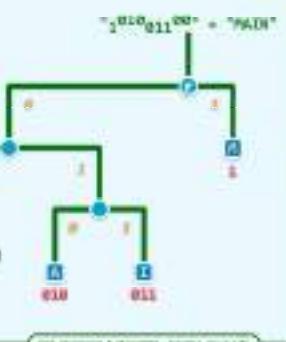
◆ 优点: 字符编码不必唯一,且

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,故不敢说又

◆ 缺点: 寻找发困难?



Data Structures &amp; Algorithms, Tonghua University

495

## 编码长度 vs. 叶节点平均深度

◆  $|rps(x)| = \text{depth}(v(x))$ ◆ 编码总长 =  $\sum_i \text{depth}(v(x_i))$ 

平均编码长度

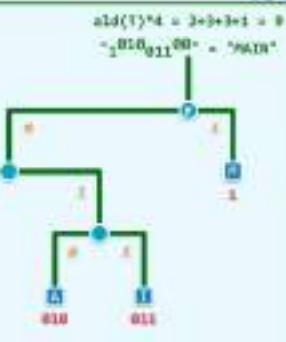
=  $\sum_i \text{depth}(v(x_i)) / |X|$ ◆ 叶节点平均深度  $ald(T)$ 

◆ 对于完全的!

 $ald(T)$  最小值即为最优编码树  $T_{opt}$ 

◆ 最优编码树必然存在,但不是唯一

它们都有相似特征?



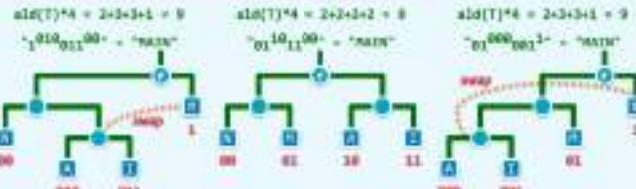
Data Structures &amp; Algorithms, Tonghua University

496

## 最优编码树

◆  $\forall v \in T_{opt}$ ,  $\deg(v) = 0$  [叶结点]  $\Leftrightarrow \text{depth}(v) \geq \text{depth}(T_{opt}) - 1$

即若，叶子只能出现在最底层，则—否则，通过节点交换即可...



◆ 特别地，**完全树**是**最优编码树**

◆ 事实上，字符的出现频率不相同时，例如  $w('t') > w('Z')$

## 白板画上，逐层构造

// 假想推论：频率低的字符优先引入，从而使得生成树短

为每个字符创建一棵单节点的树，组成森林F

按频率从低到高排序

while (F中的树不止一棵)

取出频率最小的两棵树  $T_1$  和  $T_2$

将它们合并成一棵新树  $T$ ，并令：

$lchild(T) = T_1$  且  $rchild(T) = T_2$

$w(\text{root}(T)) = w(\text{root}(T_1)) + w(\text{root}(T_2))$

// 非常幸运，就此推倒的，**能保证最优编码树之一**

## 带权编码长度 vs. 叶节点平均带权深度

+ 已知各字符的带权频率，如何构造最优编码树？

◆ 文符长度 = 平均带权深度

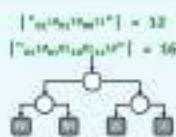
=  $wtd(T) = \sum_x w(x) \cdot \text{depth}(x) / n(x)$

◆ 此时，**完全树未必就是最优编码树**

比如，考虑“manan”和“numanina”...

$|^{\text{m} \text{a} \text{n} \text{a} \text{n}}|^{\text{m} \text{a} \text{n} \text{a} \text{n}}| = 12$

$|^{\text{n} \text{u} \text{m} \text{a} \text{n} \text{i} \text{n} \text{i} \text{n} \text{i} \text{n}}|^{\text{n} \text{u} \text{m} \text{a} \text{n} \text{i} \text{n} \text{i} \text{n} \text{i} \text{n}}| = 16$



$|^{\text{m} \text{a} \text{n} \text{a} \text{n}}|^{\text{m} \text{a} \text{n} \text{a} \text{n}}| = 12$

$|^{\text{n} \text{u} \text{m} \text{a} \text{n} \text{i} \text{n} \text{i} \text{n} \text{i} \text{n}}|^{\text{n} \text{u} \text{m} \text{a} \text{n} \text{i} \text{n} \text{i} \text{n} \text{i} \text{n}}| = 16$



## 5. 二叉树

## Huffman编码树

## 正确性

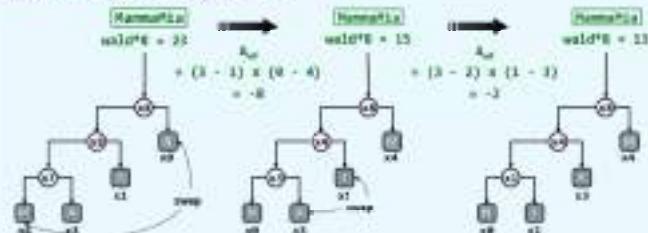
样例解

http://tinyurl.com/5yqjwz

## 最优带权识别树

◆ 同样，能识别/生成的（串）字符串，通常可能路在**西/北**处

◆ 故此，通过交换，同样可以构造  $wtd(T)$



## 正确性？

+ 色禁推论？

在多路合并并不适用

不见得能识别最长前缀

若单仅四字符(看图解)

// 拓展：推断属性

◆ Huffman树的构造采用了色禁策略，已是**最优编码树**？总是？

◆ 想见：任一指定频率的字符串，都存在对应的**最优编码树**

◆ 然而，**最优编码树可能不止一棵**

◆ 断言：Huffman树必是其中之一

// 为什么？

◆ 不然，先来考虑**非最优编码树**的特性...

## 5. 二叉树

Huffman编码树  
算法

两年的时间，在你看来，也许就是一段很的  
功夫，对不对？可对我来说，它实在长得很  
慢。我用不起来两年后的编程核心。

样例解

http://tinyurl.com/5yqjwz

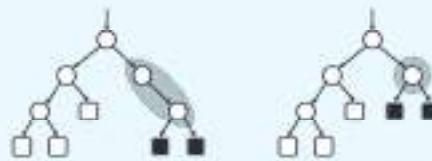
## 双子性

◆ 只要  $|T| > 1$ ，**最优编码树**中每一内部节点都有两个孩子，亦即

节点度数均为偶数 (④或②)

Huffman树必为**西二叉树**

◆ 而则，将**4度节点替换成其唯一的孩子**，实际构造  $wtd$  将更小



## 不唯一性

- ◆ 每一内部节点的左、右子树相对交换之后， $wd$ 不变  
//上述算法中，在右子树的次序可以随机选取，比如...
- ◆ 为消除这种歧义，可以（比如）明确规定 **左子树的概率要小**
- ◆ 不过，倘若它们（甚至更多节点）的概率恰好相等...



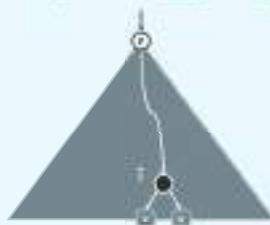
## 正确性

- ◆ 取**T**中频率最低的x和y  
//由层次性，仅考虑就成为兄弟的结点
- ◆  $\text{wd}(T) = \text{wd}(x, y) \cup \{z\}, w(z) = w(x) = w(y)$



## 层次性

- ◆ 在字符串中，**x和y是出现频率最低的两个字符**
- ◆ 则：若在某棵树先同时，**x和y在其中处于最底层**，且互为兄弟
- ◆ 为什么？



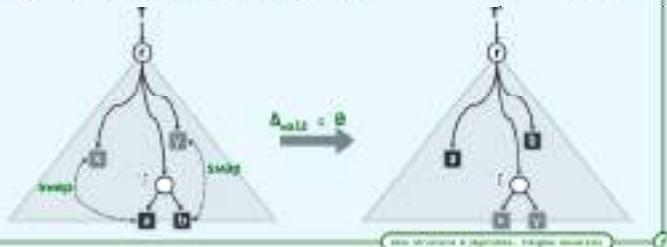
## 正确性

- ◆ 对于**T'**的任一结点时**T'**，只要为→添加结点→和y，即可  
得到**T'**的一棵新树**T**，且  $wd(T) = wd(T') \cup [w(x) + w(y)] = w(z)$
- ◆ 亦即，如此构造的**T**和**T'**， $wd$ 之差与**T**的具体形态无关



## 层次性

- ◆ 回顾一棵最优解如何  
在某温度下，任取一对兄弟x和y  
交换x和y，交换下标之后， $wd$ 绝对不会增加
- ◆ 注意**T**的对称性  
//同样，注意其对称性  
//正因如此才已看明



## 正确性

- ◆ 因此，只要**T'**是**T**的最优解树，则**T**也必是**T'**的最优解树（之一）
- ◆ 实际上，Huffman算法的过程，与上述白的过程完全一致



## 正确性

- ◆ 全部Huffman（算法所生成的）编码树，的确最优！
- ◆ 对|T|做归纳：|T|=1时显然  
设|T|<r的Huffman算法输出的最优编码，考虑|T|=r的情况...



## 5. 二叉树

Huffman编码和  
算法实现

单佳辉

zhangjiahui@ust.hk

```

#define N_CHAR (0x00 - 0x20) //仅以可打印字符为限

struct HuffChar { //Huffman(码)字节
    char ch; int weight; //字符、频度
    HuffChar (char c = '^', int w = 0) : ch(c), weight(w) {}
    bool operator< (HuffChar const& hc) //比较重
        { return weight > hc.weight; } //此的频度大小于彼
    bool operator== (HuffChar const& hc) //相等
        { return weight == hc.weight; }
};

#define HuffTree HuffTree< HuffChar *> //Huffman树，节点类即HuffChar
typedef List< HuffTree *> HuffForest; //Huffman森林

```

## 5. 二叉树

Huffman编码树  
改进

单链表

https://github.com/

## 构造编码树

```

HuffTree* generateTree(HuffForest * forest) { //Huffman编码树构造
    while (1 < forest->size()) { //反推迭代，直至森林中仅存一棵树
        HuffTree *t1 = minChar(forest), *t2 = minChar(forest);
        HuffTree *S = new HuffTree(); //创建新树，准备合并t1和t2
        S->insertASRoot(HuffChar(' ', //根节点权重，取作t1与t2之和
            t1->root()->data.weight + t2->root()->data.weight));
        S->attachASL(t1->root(), t1); S->attachASR(t2->root(), t2);
        forest->insertASLast(S); //t1与t2合并后，重新链接森林
    } //assert: 循环结束时，森林中唯一的一棵树即为Huffman编码树
    return forest->first()->data; //返回根结点之
}

```

Data Structures &amp; Algorithms, 第四版

## 优先级队列

- 方案1,  $\Theta(n^2)$ 
  - 初始化时，通过排序得到一个升序队列 //  $\Theta(n \log n)$
  - 每次（从队首）取出频率最低的两个节点 //  $\Theta(1)$
  - 将合并得到的新树插入向量，并保持有序 //  $\Theta(n)$
- 方案2,  $\Theta(n^2)$ 
  - 初始化时，通过排序得到一个升序队列 //  $\Theta(n \log n)$
  - 每次（从队首）取出频率最低的两个节点 //  $\Theta(1)$
  - 将合并得到的新树插入列表，并保持有序 //  $\Theta(n)$
- 方案3,  $\Theta(n \log n)$  //稀后聚类...保持中间
  - 初始化时，将所有树组织为一个优先级队列 //  $\Theta(n)$
  - 取出频率最低的两个节点，合并得到的新树插入队列 //  $\Theta(\log n) + \Theta(\log n)$

Data Structures &amp; Algorithms, 第四版

## 查找最小字符

```

//Huffman编码的常数效率，直接决定了minChar()的效率
以下版本仅达到  $\Theta(n)$ ，整体为  $\Theta(n^2)$ 

HuffTree* minChar(HuffForest * forest) {
    ListNodePos(HuffTree*) p = forest->first(); //从森林头出发
    ListNodePos(HuffTree*) minChar = p; //记录最小树的位置及其
    int minWeight = p->data->root()->data.weight; //树结点权重
    while (forest->valid(p = p->next)) //遍历所有节点
        if (minWeight > p->data->root()->data.weight) { //如必要
            minWeight = p->data->root()->data.weight; minChar = p; //则更新记录
        }
    return forest->remove(minChar); //从森林中移除该树，开始遍
}

```

Data Structures &amp; Algorithms, 第四版

## 预排序 x ( 插 + 队列 )

- 方案1
    - 所有字符按频率排序，构成一个队列 //  $\Theta(n \log n)$
    - 维护另一个频率队列... //  $\Theta(n)$
- 

## 构造编码表

```

#include "../Hashtable/hashtable.h" //用HashTable(第3章)实现
typedef Hashtable<char, char*> HuffTable; //Huffman编码表

static void generateCT //通过遍历映射各字符的编码
{
    BitMap* code, int length, HuffTable* table, BitNodePos(HuffChar) v {
        if (!IsLeaf(*v)) //若是叶节点（还有多种方法可以判断）
            if (table->put(v->data.ch, code->bits2String(length))) return;
        if (HasLChild(*v)) //Left != null, 进入递归
            if (code->clear(length)) generateCT(code, length + 1, table, v->l);
        if (HasRChild(*v)) //Right != null
            if (code->set(length)) generateCT(code, length + 1, table, v->r);
    } //尾部()
}

```

Data Structures &amp; Algorithms, 第四版

## 6. 图

概述

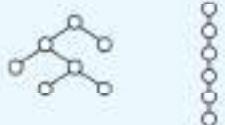
单链表

https://github.com/

## 基本术语

全图 =  $(V; E)$ vertex:  $n = |V|$ edge||arc:  $m = |E|$ 

◆ 同一端点连两个顶点，该边称重边[adjacency]



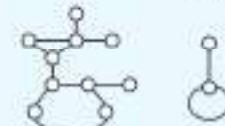
同一顶点自连称环，构成自环[self-loop]

不含重环，称为简单图[simple graph]

非简单图[non-simple]，即不讨论

◆ 顶点与其所连的边，通过关联[incidence]

度(degree/valency)：与同一顶点关联的边数



## 6. 图

邻接矩阵  
构思

用矩阵

https://tinyurl.com/yd2qzg4a

Data Structures &amp; Algorithms, Graphs (Part 1)

## 无向图 + 有向图

◆ 若每对顶点都有且仅有一条无向边



◆ 所有边均无方向的图，即无向图[undirected graph]



◆ 反之，有向图[digraph]中的边为有向边[directed edge]

u, v 分别称作边 (u, v) 的尾 (tail)、头 (head)



◆ 无向、有向边并存的图，称作混合图[mixed graph]

◆ 有向图通用性更强

极点主要针对有向图，介绍相关结构及算法

Data Structures &amp; Algorithms, Graphs (Part 1)

## Graph模板类

◆ template&lt;typename Tv, typename Tc&gt; class Graph { //所有顶点、边类型

private:

```
void mark() { //所有顶点、边的辅助信息预设
    for (int i = 0; i < n; i++) { //顶点
        states[i] = UNDISCOVERED; dTime[i] = fTime[i] = -1;
        parent[i] = -1; priority[i] = INT_MAX;
    }
}
```

public:

/\* ...，顶点操作、边操作、遍历法：见前如何实现，接口必须统一 ... \*/

Data Structures &amp; Algorithms, Graphs (Part 1)

## 路径 + 环路

◆ 路径 =  $\langle v_0, v_1, v_2, \dots, v_k \rangle$ 长度  $|v| = k$ 

◆ 简单路径：

 $v_1 = v_2$  预设  $i = 1$ ◆ 顶/环路： $v_0 = v_n$ 

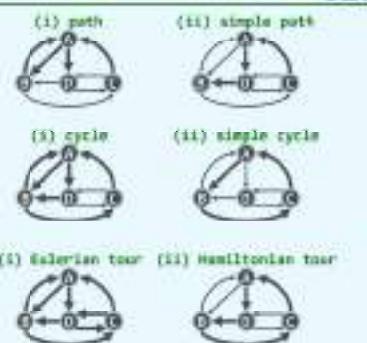
◆ 有向无环图(DAG)

◆ 欧拉环路： $|e| = |V|$ 

每边恰好出现一次

◆ 哈密尔顿环路： $|e| = |V|$ 

每顶点恰好出现一次



Data Structures &amp; Algorithms, Graphs (Part 1)

## 邻接矩阵 + 关联矩阵

◆ adjacency matrix：用二维矩阵记录顶点之间是否存在边

——对角：矩阵元素  $= 1$  表示点1与1之间存在一条边

= 0 表示

虽然只考虑简单图，对角线一起置为0

空间复杂度为  $O(n^2)$ ，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  $O(n^2) = O(n^2)$ 空间利用率  $< 1/n$ 

解决某些问题时十分有效

Data Structures &amp; Algorithms, Graphs (Part 1)

## 支撑树 + 带权网络 + 最小支撑树

◆ 图  $G = (V; E)$  的子图  $T = (V; F)$  是支撑树，即为其支撑树[spanning tree]

同一图的支撑树，通常并不唯一

◆ 若边  $e$  也有对应的权值  $w(e)$ ，则为带权网络[weighted network]

◆ 同一网络的支撑树中，总权重最小者为最小支撑树(MST)



spanning tree

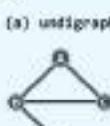
weighted network  
(triangle inequality?)

minimum spanning tree

Data Structures &amp; Algorithms, Graphs (Part 1)

## 实例

(a) undigraph



	A	B	C	D
A	0	1	1	1
B	1	0	1	1
C	1	1	0	1
D	1	1	1	0

redundancy

(b) digraph



	A	B	C	D
A	0	1	0	0
B	0	0	1	1
C	1	0	0	0
D	0	1	0	0

(c) network



	A	B	C	D
A	0	2	3	5
B	2	0	3	5
C	3	5	0	2
D	5	2	0	0

Data Structures &amp; Algorithms, Graphs (Part 1)

## 6. 图

邻接矩阵  
邻接表

## 邻接阵

dengjiaozhizhushu.cs

## 6. 图

邻接矩阵  
简单接口

## 邻接阵

dengjiaozhizhushu.cs

## Vertex

```
#include <iostream>
#include <vector>
using namespace std;

template <typename Tv> struct Vertex { //顶点对象(并不严格封装)
    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(遍历距离、根据两边等)
    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) {}
}
```

530

Data Structures &amp; Algorithms, Simple Interface

## 顶点的读写

```
#include <iostream> int i) { return V[i].data; } //数据
int indegree(int i) { return V[i].indegree; } //入度
int outDegree(int i) { return V[i].outDegree; } //出度
VStatus & status(int i) { return V[i].status; } //状态
int & dTime(int i) { return V[i].dTime; } //时间标签dTime
int & fTime(int i) { return V[i].fTime; } //时间标签fTime
int & parent(int i) { return V[i].parent; } //在遍历树中的父节点
int & priority(int i) { return V[i].priority; } //优先级
```

534

Data Structures &amp; Algorithms, Simple Interface

## Edge

```
#include <iostream>
enum { UNDETERMINED, TREE, CROSS, FORWARD, BACKWARD };
EType;
template <typename Tv> struct Edge { //边对象(并不严格封装)
    Tv data; //数据
    int weight; //权重
    EType type; //在遍历树中所属的类型
    Edge(Tv const & d, int w) : //构造新边
        data(d), weight(w), type(UNDETERMINED) {}
}
```

531

Data Structures &amp; Algorithms, Simple Interface

## 邻点的枚举

对于任意顶点i，如何枚举其所有的邻接顶点neighbor?

```
#include <iostream> int i, int j) { //若已枚举至邻居，则转向下一邻居
    while ((i+1 < j) && !exists(i, --j)) ; //正确的遍历顺序
    return j;
} //返回的邻居可提高到i + outDegree(i)
```

```
#int firstBeti int i) {
    return nextBeti(i, = 1); //假想哨兵
} //整个邻居。
```

535

Data Structures &amp; Algorithms, Simple Interface

## GraphMatrix

```
#include <iostream>
#include <vector>
template <typename Tv, typename Tv> class GraphMatrix : public Graph<Tv, Tv> {
private:
    Vector< Vertex<Tv> > V; //顶点集
    Matrix< Edge<Tv> > E; //边集
public:
    /* 操作接口：顶点相关, 边相关, ... */
    GraphMatrix() { n = e = 0; } //构造
    ~GraphMatrix() { //析构
        for (int j = 0; j < n; j++)
            for (int k = 0; k < n; k++)
                delete E[j][k]; //清除所有动态申请的边记录
    }
}
```

532

## 边的读写

```
#bool existal int i, int j) { //判断边(i, j)是否存在(边缘值)
    return (0 == i) && (1 <= i) && (0 == j) && (1 <= j) && (i < n) && (j < n) && E[i][j] != NULL;
} //以下假设exists(i, j)...
#Tv & edge( int i, int j ) //取(i, j)的边
    { return E[i][j]->data; } //O(1)
#EType & type( int i, int j ) //边(i, j)的类型
    { return E[i][j]->type; } //O(1)
#int & weight( int i, int j ) //边(i, j)的权重
    { return E[i][j]->weight; } //O(1)
```

536

Data Structures &amp; Algorithms, Simple Interface

6. 图  
邻接矩阵  
复杂接口

样例  
<http://tinyurl.com/3qyvz3>

```
+ void insert( Tv const & edge, int u, int v ) { // 插入(u, v, w)
    if ( exists(u, v) ) return; // 跳过已有的边
    E[u][v] = new EdgeType( edge, w ); // 邻接矩阵
    e++; // 更新边计数
    V[u].outDegree++; // 更新关联顶点的出度
    V[v].inDegree++; // 更新关联顶点的入度
}
```

538

## 顶点删除

```
+ Tv remove( int i ) { // 删除顶点及其关联边，返回该顶点的度数
    for ( int j = 0; j < n; j++ ) // 遍历所有出边
        if ( exists( i, j ) ) { delete E[i][j]; V[j].inDegree--; }
    E.remove(i); n--; // 删除边
    tv vBox = vertices( i ); V.remove( i ); // 移除之后，删除顶点
    for ( int j = 0; j < n; j++ ) // 遍历所有入边及尾结点
        if ( EdgeType * e = E[j].remove( i ) ) { delete e; V[j].outDegree--; }
    return vBox; // 返回删除时顶点的邻居
}
```

542

6. 图  
邻接矩阵  
性能分析

样例  
<http://tinyurl.com/3qyvz3>

```
+ void remove( int i, int j ) { // 删除顶点i和j之间的联边 (exists(i, j))
    tv eBox = edge(i, j); // 跳过边(i, j)的信息
    delete E[i][j]; E[i][j] = NULL; // 删除边(i, j)
    e--; // 更新边计数
    V[i].outDegree--; // 更新关联顶点的出度
    V[j].inDegree--; // 更新关联顶点的入度
    return eBox; // 返回被删除边的信息
}
```

539

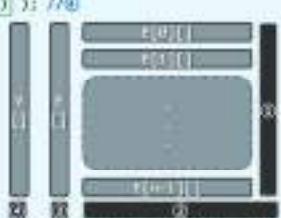
## 优点

- 直观，易于理解和实现
- 通用图形语法：digraph / network / cyclic / ...
- 尤其适用于稠密图（dense graph）
- 判断两点之间是否存在联边： $O(1)$
- 获取顶点的（出/入）度数： $O(1)$   
添加、删除边后更新度数： $O(1)$
- 扩展性（scalability）：  
得益于Vector良好的空间局部性  
空间进出等操作可“透明地”予以处理

540

顶点插入

```
+ int insert( Tv const & vertex ) { // 插入顶点，返回编号
    for ( int j = 0; j < n; j++ ) E[j].insert( NULL ); n++; // ①
    E.insert( Vector< EdgeType*>( n, n, NULL ) ); // ②
    return V.insert( VertexType( vertex ) ); // ③
}
```



540

缺点

- $O(n^2)$ 空间，与边数无关！
- 是否会有这么多的边呢？不妨考虑一类特定的图...
- 平面图（planar graph）：可嵌入于平面的图
- Father's formula (1750)：  
 $V - E + F - C = 1$ , for any PG
- 平面图： $e \leq 3n - 6 \leq O(n) \ll n^2$   
此时，空间利用率 =  $1/n$
- 稀疏图（sparse graph）  
空间利用率同样极低，可采用压缩存储技术



## 6. 图

## 邻接表

## 译注释

[zhangtinghua.com](http://zhangtinghua.com)

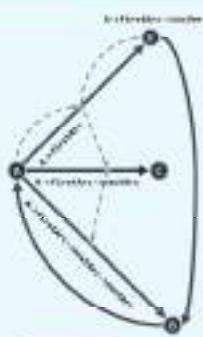
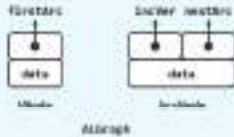
## 邻接表

◆ 邻接表的关键组成部分是什么？

- 将关联矩阵的各行组织为列表
- 只记录存在的边

◆ 算法上，每一顶点v对应于列表

$$L_v = \{ u \mid uv, uv \in E \}$$



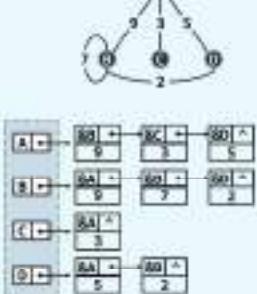
## 实例

◆ 4个顶点，5条边

◆ 不必占用 $4 \times 4 = 16$ 个单元

但还是占用了5个单元，另外4个单元

v	A	B	C	D
A		9	3	6
B	9	7		2
C	3			
D	6	2		



## 时间复杂度

◆ 检查顶点u, v是否存在边 $uv \in E$ 有向图：搜索u的邻接表， $\mathcal{O}(\deg(u)) = \mathcal{O}(n)$ 无向图：搜索u或v的邻接表， $\mathcal{O}(\max(\deg(u), \deg(v))) = \mathcal{O}(n)$ “并行”搜索： $\mathcal{O}(2 \times \min(\deg(u), \deg(v))) = \mathcal{O}(e)$ 能表达邻接矩阵的 $\mathcal{O}(1)$ 吗？

◆ 整列！如果遍历因子远超得当

- 例：expected $\sim\mathcal{O}(1)$ ，与邻接矩阵“相同”- 空间： $\mathcal{O}(n + e)$ ，与邻接矩阵相同

◆ 为什么仍使用邻接矩阵？仅仅因为实现简单？不，有更多原因！

如：可处理Euclidean graph和intersection graph之差的

稠密图（implicitly-represented graphs）

## 联合原则

◆ 完全/速度

◆ 顶点类型

- int
- arr
- float
- struct
- class
- ...

◆ 边类型（方向 / 权值）

◆ 图类型（稀疏 / 稠密）

邻接矩阵	邻接表
经常检测边的存在	经常计算顶点的度数
经常能边的插入/删除	联合题目不确定
图的规模固定	经常做遍历
稠密图	稀疏图

## 空间复杂度

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

注意：无向图要浪费存储

问题：如何改进？

◆ 邻接子集的图

◆ 平衡图 =  $\mathcal{O}(n + 3e) = \mathcal{O}(n)$ 

较之邻接矩阵，有很大改进



## 6. 图

广度优先搜索  
算法

## 译注释

[zhangtinghua.com](http://zhangtinghua.com)

## 时间复杂度

◆ 建立邻接表（链式存储）： $\mathcal{O}(n + e)$ 

// 如何实现

◆ 检查所有以顶点v为端点的邻居： $\mathcal{O}(1 + \deg(v))$ 

// 邻接表的邻接表

◆ 检查所有以顶点v为头的邻居： $\mathcal{O}(n + e)$ 

// 邻接表的邻接表

可能进退  $\mathcal{O}(1 + \deg(v))$ 

// 建立逆邻接表

为此，空间增加多少？

◆ 计算顶点v的度数/入度

增加邻接表表域： $\mathcal{O}(n)$ 附加空间

// 邻接表的邻接表

增加/删除时更新度数： $\mathcal{O}(1)$ 时间// 总体 $\mathcal{O}(e)$ 时间每次度数  $\mathcal{O}(1)$ 时间

Data Structures &amp; Algorithms, Chapter 6: Graphs

## Breadth-First Search

◆ 基本思想：广度优先搜索

访问顶点：

依次访问所有尚未访问过的邻接顶点

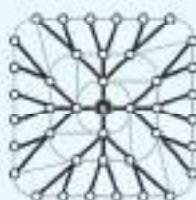
依次访问它们尚未访问的邻接顶点

如此反复

直到没有尚未访问的邻接顶点

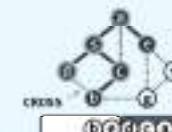
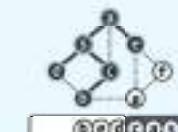
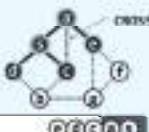
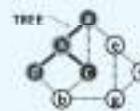
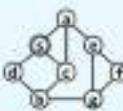
◆ 以上策略及过程完全等同于图的广度优先遍历

◆ 事实上，BFS也恰好构造出原图的一棵支撑树（BFS tree）



Data Structures &amp; Algorithms, English Version

## 无向图



Data Structures &amp; Algorithms, English Version

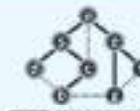
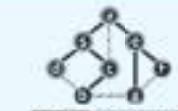
## Graph::BFS()

```
#include <queue>
#include <iostream>
using namespace std;

template <typename Tv, typename Tn>
void Graph<Tv, Tn>::BFS( int v, int &clock ) {
    queue<int> Q; status(v) = DISCOVERED; Q.enqueue(v); // 初始化
    while ( !Q.empty() ) { // 循环
        int v = Q.dequeue(); dTime(v) = ++clock; // 取出队首顶点v，并
        for ( int u = firstNbr(v); ~1 < u; u = nextNbr(v, u) ) // 遍历v的每一邻居
            if ( ... v 的状态， 分别处理 ... )
                status(u) = VISITED; // 此处，当被顶点访问时
    }
}
```

Data Structures &amp; Algorithms, English Version

## 无向图



Data Structures &amp; Algorithms, English Version

## Graph::BFS()

```
#include <queue>
using namespace std;

int v = Q.dequeue(); dTime(v) = ++clock; // 取出队首顶点v，并
for ( int u = firstNbr(v); ~1 < u; u = nextNbr(v, u) ) // 遍历v的每一邻居
if ( !DISCOVERED == status(u) ) { // 若u尚未被发现，则
    status(u) = DISCOVERED; Q.enqueue(u); // 将u置为待发现
    type(v, u) = IN; parent(u) = v; // 将入数加1
} else // 若u已被发现（正在队列中），或者已经已访问完毕（已出队列），则
    type(v, u) = OUT; // 将(v, u)归类于普通
status(v) = VISITED; // 此处，当被顶点访问时
}
```

Data Structures &amp; Algorithms, English Version

## 6. 图

广度优先搜索  
推广

## 单链表

http://tinyurl.com/3yqjwz

6. 图  
广度优先搜索  
实例单链表  
http://tinyurl.com/3yqjwz

## 连通分量 + 可达分量

## ◆ 算法

给定无向图，找出图中任一顶点v所在的连通分量

给定有向图，找出源自其中任一顶点v的可达分量

## ◆ 算法

从v出发做DFS

输出所能被发现的顶点

队列为空后立刻终止，无图考虑其它顶点



Data Structures &amp; Algorithms, English Version

```
graph::bfs()
+template <typename TV, typename Te> //类边图，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( n + e )
    do //逐一检查所有顶点，一旦遇到尚未发现的顶点
        if ( !DISCOVERED == status(v) ) //置计数器
            RESE(v, clock); //将从该顶点出发启动一次BFS
    while ( ++s = ( v = ( s + e - 1 ) ) ) //按序号访问，不漏不重
        //无论具有多少连通/可达分量...
    } //bfs()均可遍历它们，而自身累计仅需恒定时间...

```



**BFS树/森林**

- 对于每一条通路/可达分量，bfs(v)进入BFS(v)恰好1次 (v为该分量的**基础顶点**)
- 进入BFS(v)时，队列为空
- 所遍历分量内的每个顶点
  - 通常会以`UNDISCOVERED`状态进队1次
  - 由队首被移除为`DISCOVERED`状态，并生成一条新边
  - 通常会出队并转为`VISITED`状态
- 退出BFS(v)时，队列为空
- BFS(v)以v为根，生成一棵BFS树
- bfs()生成一个BFS森林包含
  - 二叉树：n - c 条树边和 e - c + c 条普通边

**复杂度**

考虑无向图...

• bfs() 的初始化 (reset())  $O(n + e)$

• bfs() 的循环  $O(n + 2e)$

外循环 (while (!Q.empty()))，每个顶点只进入1次，累计n次  
内循环 (枚举v的每一邻居)，每个邻居至多进入1次，累计 $\deg(v)$ 次

采用邻接矩阵  $O(s)$

采用邻接表  $O(1 + \deg(v))$

总耗： $O(\sum_{v=1}^n (1 + \deg(v))) = O(n + 2e)$

整个算法： $O(n + e) + O(n + 2e) = O(n + e)$

• 有向图呢？亦是如此！

Data Structures & Algorithms, Higher Edition**最短路径**

从无向图中顶点s出发，前往任一顶点t的所有路径中
 

- 长路数距离  $d(s, t) = ?$
- (最短) 距离  $\text{dist}(v) = |s(s, v)| = ?$
- 退化情况：可能有多条 //任取其一

在BFS过程中的任一时刻 //松弛操作
 

- 队列中顶点按 `dist()` 单调排列 //自内而外，由近及远
- 队列中相邻顶点，`dist()` 相等不修改
- 队首、队末顶点，`dist()` 相差不超过1 //模拟广搜遍历
- 由树边联接的顶点，`dist()` 增加1 //从亲到子
- 由普通边联接的顶点，`dist()` 增加相等 //若呈六边，遍历方法：从右至左，更像深度

BFS树中从s到t的路径，即  $s(t, t)$

**Erdős Number**

+ Describes the "collaborative distance" between mathematician Paul Erdős and another person, as measured by authorship of mathematical papers



五环 你比四环多一环  
五环 你比六环少一环  
对于每一天 你会被到七环  
做到七环怎么办  
你比五环多两环

## 6. 图

### 广度优先搜索 性质

邓桂娟  
[dengguijuan.com](http://dengguijuan.com)

**边分类**

• 按BFS层，所有边均确定方向，自然分为两类
  $\rightarrow$  //有向图有何不同？
   
tree edges + cross edges

• 一个树边  $(v, u)$   $\rightarrow$   $\rightarrow$   $v$

树边之前： $v \in \text{DISCOVERED}$  &  $u \in \text{UNDISCOVERED}$

树边之后： $v = parent(u)$

• 一个圈边  $(v, u)$   $\rightarrow$   $\rightarrow$   $?$   $\rightarrow$   $u$

无向图：只可能  $v$  和  $u$  均为 `UNDISCOVERED`

有向图：还可能  $v$  是 `UNDISCOVERED`,  $u$  是 `VISITED`

**Chow Number**

• `chow` (周嘉达) = 1
 

- [图论专家] (1)  $\rightarrow$  吴恩达、吴建达、姚士安、刘维华、关之琳、胡锦秋

• `chow` (周锐) = 2
 

- [没听说过] (2)  $\rightarrow$  马化、黄精莲、隋超
- [程序员] (3)  $\rightarrow$  吴恩达、吴建达、吴清宇、钟丽丽、伍冰清、黄晋鸣

• `chow` (周国) = 3
 

- [烹饪大师] (3)  $\rightarrow$  张本硕、孙敏、董慧、陈丽娟、于祖康、唐杰忠
- [物理自由] (2)  $\rightarrow$  张本硕、任达华、王智平、吴家丽
- [甲死者] (1)  $\rightarrow$  吴恩达、吴建达、吴家丽、董洁、宋晓峰、梅艳芳

• `chow ("Julia Roberts") = infinity`

## 6. 四

深度优先搜索  
算法

尚相逢之不称奇，遇忙乎若惊反。  
臣群车以纵唇舌，及行进之未尽。

林佳群

www.englishbookclub.ca

## 6. 四

深度优先搜索  
实例（无向图）

Keep it simple, stupid.

— R. Johnson

林佳群

www.englishbookclub.ca

## 570

## Depth-First Search

◆ DFS(v) //给定顶点v的深度优先搜索

访问顶点v

若v尚未被访问的邻居，则任取其一，递归执行DFS(u)

否则，返回

◆ 若此时图中内有顶点未被访问 //何时出现这一情况？

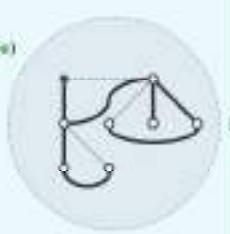
任选这样的一个顶点作为起始点

重复上述过程

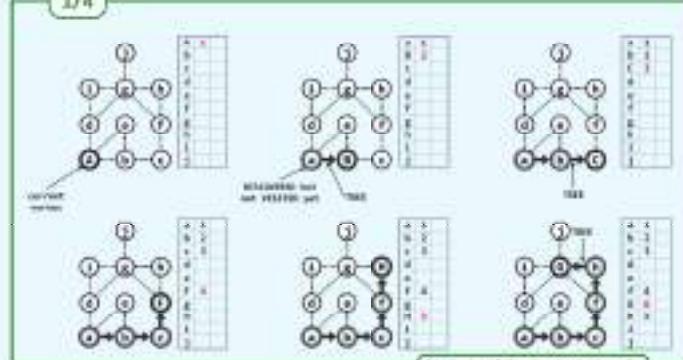
直至所有顶点都被访问

◆ 等效于树的先序遍历

事实上，DFS也的确构造出源图的一棵先序树 (DFS tree)



## 3/4



## 574

## Graph::DFS()

◆ template <typename Tv, typename Tc> //顶点类型、边类型

void Graph<Tv, Tc>::DFS( int v, int &clock ) {

    dtime(v) = ++clock; status(v) = DISCOVERED; //发现新的顶点

    for ( int u = firstNei(v); ~1 < u; u = nextNei(v, u) ) //枚举v的每一个邻接点

        // ... 情况的处理 ... //

        // ... 与DFS不同，需要重写 ... //

        status(v) = VISITED; FTIME(v) = ++clock; //至此，当前顶点v方面访问完毕

## 571

## Graph::DFS()

◆ for ( int u = firstNei(v); ~1 < u; u = nextNei(v, u) ) //枚举v的所有邻接点

    switch ( status(u) ) { //并根据状态分别处理

        case UNDISCOVERED: //尚未识别，意味着支撑树可在该处延伸

            type(v, u) = TREE; parent(u) = v; DFS( u, clock ); break; //递归

        case DISCOVERED: //u已被发现但尚未访问完毕，应将被置为尚未的祖先

            type(v, u) = BACKWARD; break;

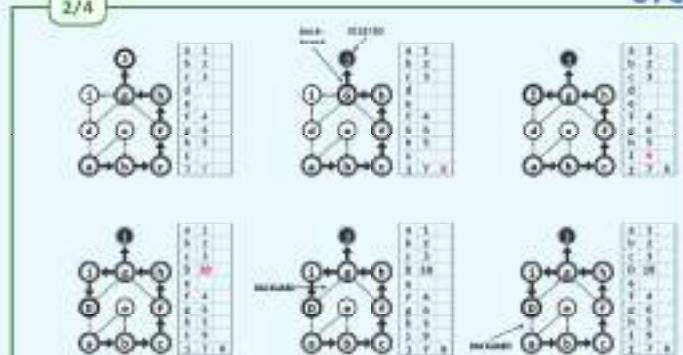
        default: //u已访问完毕 (VISITED, 有向图)，利用亲缘关系分为前向或跨边

            type(v, u) = dtime(v) < dtime(u) ? FORWARD : CROSS; break;

    } //switch

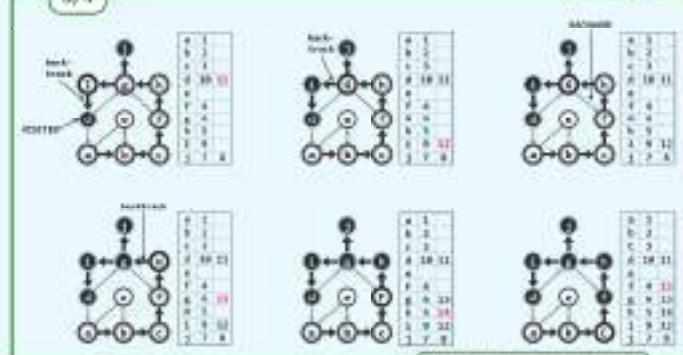
## 572

## 2/4



## 575

## 3/4



## Graph::DFS()

◆ for ( int u = firstNei(v); ~1 < u; u = nextNei(v, u) ) //枚举v的所有邻接点

    switch ( status(u) ) { //并根据状态分别处理

        case UNDISCOVERED: //尚未识别，意味着支撑树可在该处延伸

            type(v, u) = TREE; parent(u) = v; DFS( u, clock ); break; //递归

        case DISCOVERED: //u已被发现但尚未访问完毕，应将被置为尚未的祖先

            type(v, u) = BACKWARD; break;

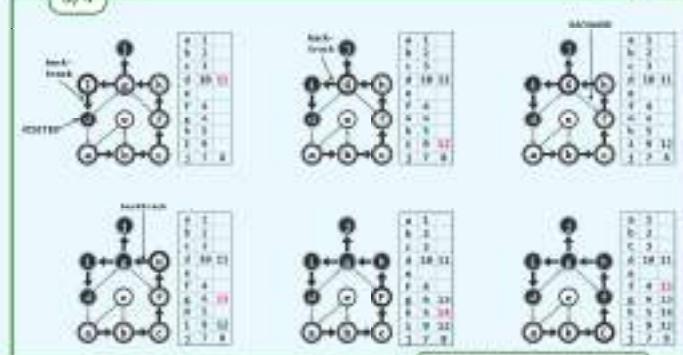
        default: //u已访问完毕 (VISITED, 有向图)，利用亲缘关系分为前向或跨边

            type(v, u) = dtime(v) < dtime(u) ? FORWARD : CROSS; break;

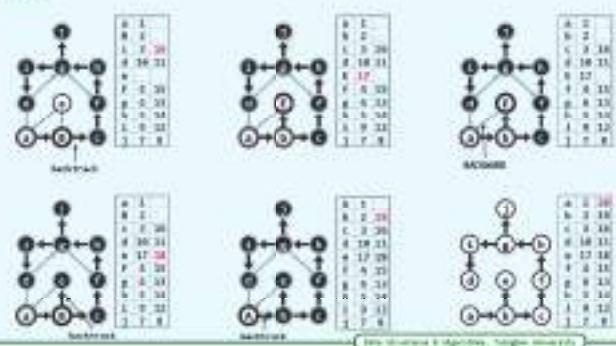
    } //switch

## 572

## 3/4



## 576



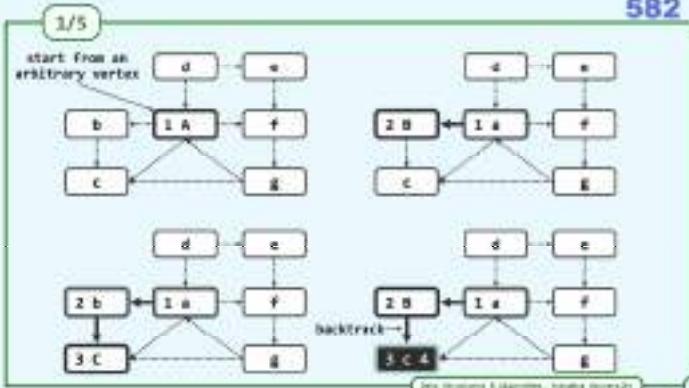
6. 图示

深度优先搜索  
实例（有向图）

郑佳辉

wangjiahui@zjhu.edu.cn

6. 图示  
深度优先搜索  
推广

郑佳辉  
wangjiahui@zjhu.edu.cn

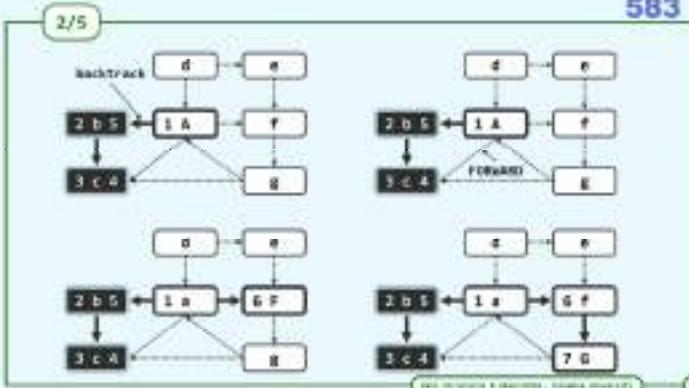
非连通

全局  $\text{dfs}(v)$  类似， $\text{dfs}(v)$  也可遍历  $v$  所属分量

- 若含多个分量呢？

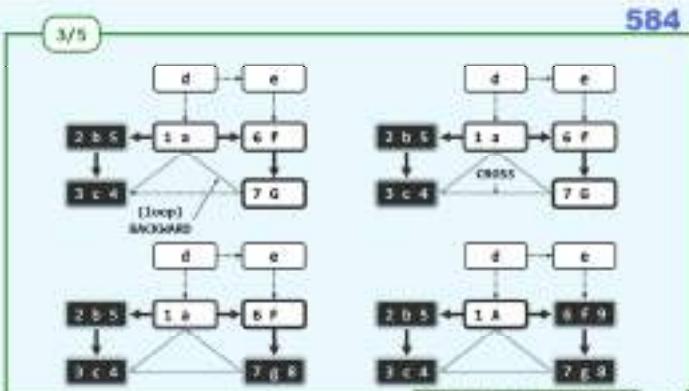
- 如  $\text{dfs}(s)$  类似（深搜带图素） $\text{dfs}(s)$  也可在预计  $O(n + e)$  时间内

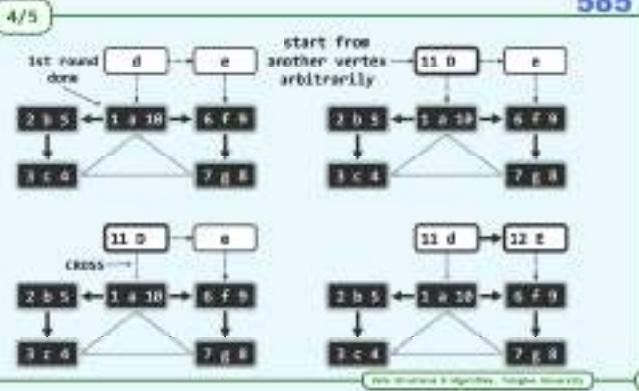
- 对于每一连通/可达分量

从起始顶点  $v$  进入  $\text{dfs}(v)$  恰好 1 次，并- 最终生成一个 DFS 林（包含  $\text{Euler}$ 、 $\text{BFS}$  树根边）

Graph::dfs()

```
# template <typename Tv, typename Tss> // 领域类型, 边类型
void Graph<Tv, Tss>::dfs( int s ) { // s 为起始节点
    reset(); int clock = 0; int v = s; // 初始化
    do // 遍历所有顶点, 一旦遇到尚未发现的顶点
        if ( !DISCOVERED == status(v) )
            DFS( v, clock ); // 从该顶点出发启动一次 DFS
    while ( s != ( v = ( ++v % n ) ) ); // 循环再访问, 防不漏不重
}
```





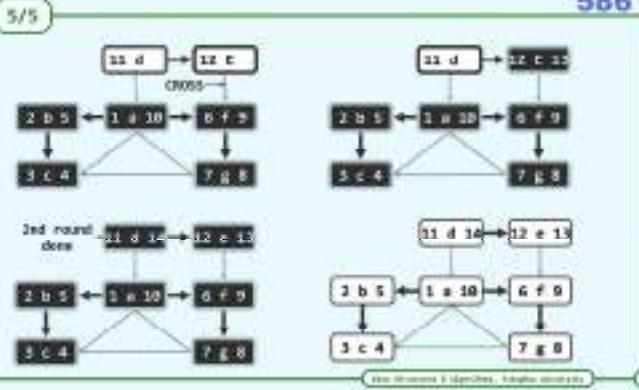
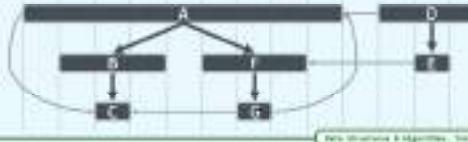
## 括号引理

• 选择规则:  $\text{active}[u] = (\text{dTime}[u], \text{fTime}[v])$

• Parenthesis Lemma: 指定有向图  $G = (V, E)$  及同样一DFS森林, 则

- $u$  是  $v$  的后代 iff  $\text{active}[u] \subset \text{active}[v]$
- $u$  是  $v$  的祖先 iff  $\text{active}[u] \supset \text{active}[v]$
- $u$  与  $v$  “无关” iff  $\text{active}[u] \cap \text{active}[v] = \emptyset$

今仅用  $\text{status}[], \text{dTime}[], \text{fTime}[]$ , 即可对每边分类...



## 边分类

• TREE( $v, u$ ): 可从当前  $v$  进入处于 DISCOVERED 状态的  $u$

• BACKWARD( $v, u$ ): 试想从当前  $v$  进入处于 DISCOVERED 状态的  $u$   
DFS发现后向边 iff 存在回路 //后向边  $\Rightarrow$  回路?

• FORWARD( $v, u$ ): 试想从当前顶点  $v$  进入处于 VISITED 状态的  $u$ , 且  $u$  第早被发现

• CROSS( $v, u$ ): 试想从当前顶点  $v$  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以区分, 那应没有

//为什么?

## 6. 图

### 深度优先搜索 性质

身后有余忘动手  
腹痛无病想回头

邓伟群  
<http://www.csail.mit.edu/~dvw/>

## 遍历算法的应用

遍历算法实现 (traverser tree)	RTS/RTS
非连通图的支持森林	RTS/RTS
连通性检测	RTS/RTS
无向图环路检测	RTS/RTS
有向图环路检测	RTS
顶点之间可达性检测/路径求解	RTS/RTS
顶点之间的最短距离	RTS
圈游	RTS
Edmonds tree	RTS
拓扑排序	RTS
无向图分量、强连通分量分量	RTS

## DFS树/森林

• 从顶点  $s$  出发的 DFS

- 在无向图中将访问与该顶点相邻的所有顶点 connected component
- 在有向图中将访问由  $s$  可达的所有顶点 reachable component

• 经 DFS 被访问的顶点, 不会构成回路

• 从  $s$  出发的 DFS, 相以  $s$  为根生成一棵 DFS 树, 所有 DFS 树, 进而构成 DFS 森林

• DFS 树及森林的 parent 字符串标注 (只不过所有边都是反向)

• DFS 之后, 我们已经知道森林乃至原图的全部信息了吗?

就某种意义上而言, 是的...

## 6. 图

### 拓扑排序 最小度算法

邓伟群  
<http://www.csail.mit.edu/~dvw/>

## 有向无环图

593

Directed Acyclic Graph

应用

资源生成和继承关系图中，是否存在循环依赖

操作系统中，相互等待的一组进程是否可调度，如何调度

给定一组相容依赖的课程，是否存在可行的培养方案

给定一组相容依赖的知识点，是否存在可行的教学进度方案

项目工程图中，是否存在可行的施工方案

cos12系的学分，是否存在满足转专业限制的选修



Data Structures &amp; Algorithms, Chapter 10(DAG)

## 算法A：顺序输出零入度顶点

597

将所有入度为零的顶点存入队列S，取空队列Q //O(n)

while (!S.empty()) { //O(n)}

Q.enqueue(S.pop()); //将S中的顶点v加入队列Q

for each edge(v, u) //v的邻接顶点u若入度为1

if (u.inDegree &lt; 2) S.push(u); //将入度为1的顶点u加入队列S

Q = Q \ { v }; //删除v及其关联边（邻接顶点入度减1）

} //迭代O(n + n)

return [Q]; Q := "NOT\_A\_DAG"; //判别物为空，当该队列可拓扑排序



Data Structures &amp; Algorithms, Chapter 10(DAG)

## 拓扑排序

594

任何也有向图G，不一定能TAS

尝试将其顶点排成一个线性序列

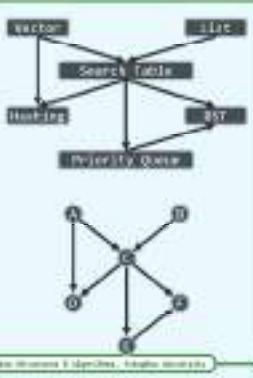
其次序表与原图相容，亦即

每一顶点都不会通过指向向(前驱)顶点

算法要求

若源顶点在图尾(即升序Down)，检查并报告

否则，给出一个相容的线性序列



Data Structures &amp; Algorithms, Chapter 10(DAG)

## 6. 图

598

## 拓扑排序

## 零出度算法

样例解

http://tiny.cc/meyarw

## 存在性

595

每个DAG对应于一个拓扑图；拓扑排序对后于一个全序图

所求的拓扑排序，即构造一个与指定顺序相容的全序图

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...

相容的序集必有极大/极大元素

任何DAG都存在(至少)一种拓扑排序

可归到证明，并直接导出一个算法...



Data Structures &amp; Algorithms, Chapter 10(DAG)

## 算法B：逆序输出零出度顶点

599

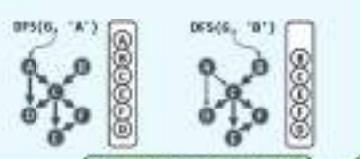
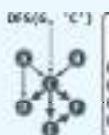
/\* 基于DFS，倒置DFS \*/

对图G做DFS，利用 //得到图GDFS森林的一系列DFS树

每棵树有该点被标记为VISITED，时将其压入S

一旦发现有逆内边，则退出DFS并输出

DFS遍历后，顺序输出S中的各个顶点

相容度与DFS相当，也是 $O(n + e)$ 

Data Structures &amp; Algorithms, Chapter 10(DAG)

## 存在性

596

1. 任给DAG，必有(至少一个)顶点入度为零 //记作v

2. 若  $DAG \setminus \{v\}$  在拓扑排序  $S = \langle v_0, v_1, \dots, v_{n-1} \rangle$  //SUBTRACTION则  $S' = \langle v_1, v_2, \dots, v_{n-1} \rangle$  为DAG的拓扑排序 //DAG子图仍为DAG

只要v唯一，拓扑排序也应唯一 //反之呢？



Data Structures &amp; Algorithms, Chapter 10(DAG)

## 实现(1/2)

600

&lt;template &lt;typename Tv, typename Tc&gt; //链表类型, 边类型

bool Graph&lt;Tv, Tc&gt;::BFS(int v, int &amp;clock, Stack&lt;Tv&gt; \*S)

dTime(v) = ++clock; status(v) = DISCOVERED; //发现顶点v

for (int u = firstNbr(v); -1 &lt; u; u = nextNbr(v, u)) //枚举v所有邻居u

{ /\* ... 邻接的邻居, 分别处理 ... \*/ }

status(u) = VISITED; S-&gt;push(vertex(u)); //顶点被标记为VISITED时入堆

return true;

Data Structures &amp; Algorithms, Chapter 10(DAG)

```

for (int u = firstNbr(v); -1 < u; u = nextNbr(v, u)) //枚举所有邻接点
    switch (status(u)) { //根据u的状态分别处理
        case UNDISCOVERED:
            parent(u) = v; type(v, u) = TREE; //树边(v, u)
            if (!TSearc(u, clock, 5)) return false; break; //从顶点u处理入
        case DISCOVERED: //一旦发现后向边(TBno)
            type(v, u) = BACKWARD; return false; //跳出不再深入
        default: //VISITED (digraphs only)
            type(v, u) = dTime(v) < dTime(u) ? FORWARD : CROSS; break;
    }
}

```

```

while (1) { //依次引入n-1个顶点(和n-1条边)
    for (int u = firstNbr(s); -1 < u; u = nextNbr(s, u)) //对s的邻接点
        priupdate(this, s, u); //更新堆中u的优先级及其父结点
    if (UNDISCOVERED == status(s)) //从未加入遍历树的顶点中
        if (shortest > priority(u)) //选出下一个
            if (shortest = priority(u); s = u); //优先级最高的顶点
        if (VISITED == status(s)) break; //直到所有顶点均已加入
    status(s) = VISITED; type(parent(s), s) = TREE; //将s加入遍历树
} //while

```

## 6. 图

优先级搜索:

郑伟群

weng@zjhu.edu.cn

执行时间主要消耗于内层循环体：检测所有内层环的，总开销  
 前一循环中，优先级更新的次数呈阶梯状数变化 $\{0, 0+1, \dots, 1, 1\}$ ，累加为 $\Theta(n^2)$   
 两项合计，为 $\Theta(n^2)$

后面将会看到：若采用优先队列，以上两项将分别为 $\Theta(\log n)$ 和 $\Theta(n \log n)$  //保持队列  
 两项合计，为 $\Theta((e+n) \cdot \log n)$

这是很大的改进——尽管对于稠密图而言，反而慢倒退 //已有接近于 $\Theta(e+n \log n)$ 的算法  
 基于这个统一框架，如何解决同样的应用问题...

各种遍历算法的区别，仅在于选取顶点进行访问的次序  
 广度/深度：优先级论及更早/更晚被发现的顶点和邻接者  
 ...

不同的遍历算法，取决于存放和提供顶点的数据结构——栈

较高的优先级，为每个顶点v维护一个优先级值priority(v)  
 每个顶点都有包括优先级；并可能能降低的优先级(树根)

通常的习惯是，优先级数据大/小，优先级低/高  
 特别地，priority(v) == INT\_MAX，意味着v的优先级最低

## 6. 图

Prim算法  
最小支撑树

“孩子会，艺舞翩之以浪漫？还有那盲目的雨歌。  
 你更在苦练这一身风姿长袖，把村里的每一户人家  
 都连通起来，哈哈。他以为，这样一来，盲济人就  
 可免除归宿雨淋之苦了。”

郑伟群  
weng@zjhu.edu.cn

```

template <typename Tv, typename Te> //顶点类型，边类型
template <typename Pv> //优先级更新器(函数对象)

void Graph<Tv, Te>::pfs( int s, Pv priupdate ) { //Pv把原结，遍历法改写
    priority(s) = 0; status(s) = VISITED; parent(s) = -1; //起点s如果Pv树中
    while (1) { //将下一顶点和边添加至S树中
        //...，依次引入n-1个顶点(和n-1条边)..., ...
    } //while
} //如何推广至非连通图？

```

连通图G = (V, E)的子图T = (V, F)

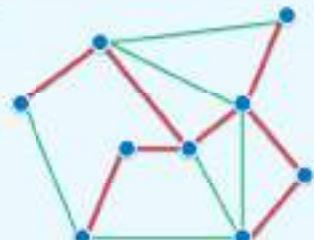
支撑(spanning) = 图G中所有顶点

树(tree) =

- 通过最短边， $|V| = |F| + 1$
- 加边出单环，再把同样边即权重为0的
- 环边不连通，再加那块边即权数为1

不连通图，同一树的支撑树不唯一

最小(minimum) = 各边总权重 $w(T) = \sum_{e \in T} w(e)$ 达到最小

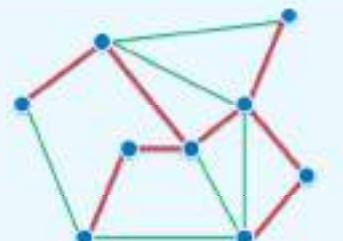


- 什么是MST？电信公司、网络设计师、VLSI布线的算法设计...
- 为什么重要？
  - 应用中常见的共性问题
  - 也是很多优化问题的基本模型
  - 自身可看作计算
  - 为许多MST提供出较好的近似解
- 比如，Euclidean TSP



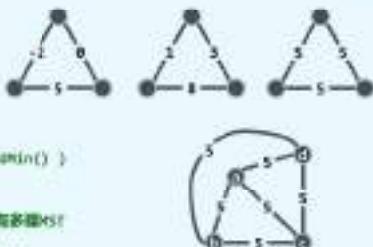
## 已有算法

- Boruvka-1926
- Jarník-1930
- Kruskal-1956
- Szemerédi-Klein-Tarjan-1995
- Chapelle-2000
- ...是否...存在 $(n + \epsilon)$ 算法？



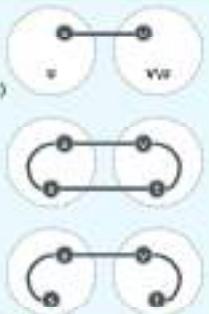
## 演化

- 极值的清晰正题？
- 化为图，会有什么帮助？
- 化为数论版呢？
- 所有支撑树所含的边数，必然相等
- 极小的一调整：increase( 1 - f(max()) )
- The minimum?
- A minimal 同一网路 $\tau$ 可能有多棵MST
- The minimum! 可强制消除歧义...
- 全奇数 (composite number) : { w(u, v), min(u, v), max(u, v) }
- $Sub = Sub < Sub < Sub < Sub < Sub$



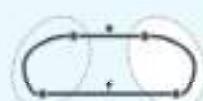
## 割 &amp; 极短跨边

- 若 $(u, v)$ 是 $M$ 的割 (cut)
- 【Cut Property-a】
  - 若 $(u, v)$ 是 $M$ 的极短跨边 (shortest crossing edge)
  - 则 $M$ 必存在一棵包含 $(u, v)$ 的MST
- 反正：假设 $(u, v)$ 未被任何MST采用...
- 【归取】一株MST，将 $(u, v)$ 加入其中，于是
  - 将此变成唯一的割边，且此割边必经过 $(u, v)$ 以及至少另一边 $(t, t')$
  - 现在，将原MST中的 $(t, t')$ 替换为 $(u, v)$ ...
- 【Cut Property-b】反之， $M$ 的任一MST也必通过割边而必经过每一割



## 环 &amp; 极长环边

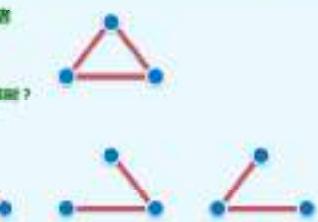
- 若 $t$ 是 $\tau$ 的一条边，而在 $\tau$ 中添加边 $t'$ 能得到 $\tau'$
- 【cycle Property】
  - 若 $t$ 沿着 $\tau$ 在 $\tau$ 中对应的环路， $t$ 为一圆长边
  - 则 $\tau' = \{t\} + \{x\}$ 即为 $\tau'$ 的一棵MST
- 若 $t$ 为环路上的最长边，则与前同理， $t$ 不可能属于 $\tau'$ 的MST
- 此时， $\tau' = \tau - \{t\} + \{x\} = \tau$ 仍然是 $\tau'$ 的MST
- 否则有： $|e| \leq |\tau|$ ；移除 $t$ 后 $\tau - \{t\}$ 一分为二，则属于 $M/M'$ 的割
- 在 $M/M'$ 中， $t/e$ 便是该割的极短跨边
- 此前在 $M/M'$ 中指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 $M'$ 的一棵MST



## 暴力算法

- 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...
- 多少个互异的顶点组成的图，可能有多少棵支撑树？
 

$n = 1$	1
$n = 2$	1
$n = 3$	3
$n = 4$	16
...	...
- 康尼公式：联接 $n$ 个互异顶点的树共有 $(n-1)!$ 棵；或等价地，剩余图 $G_n$ 有 $(n-1)!$ 棵支撑树
- 如何高效地构造MST呢？



## 算法

- 从 $T_0 = \{ (v_1) \cup \emptyset \}$ 开始，逐步构建 $T_1, T_2, \dots, T_k$ ，其中
  - $v_1$ 可以任选
  - $T_k = \{ V_{k+1}, E_k \}$
  - $|V_k| = k, |E_k| = k-1$
  - $V_k \subseteq V_{k+1}$
- 由以上分析，为由 $T_k$ 构造 $T_{k+1}$ ，只需
  - 将 $(V_k \cup \{v_{k+1}\})$ 操作深搜的一个割
  - 在该割的所有跨边中，找出极短边 $e_{k+1} \in \{ (v_k, v_{k+1}) \}$
  - $\oplus T_{k+1} = (V_{k+1} \cup E_k) \times (V_k \cup \{v_k\}, E_k \cup \{e_{k+1}\})$



6. 图  
Prim 算法  
实例

最佳解

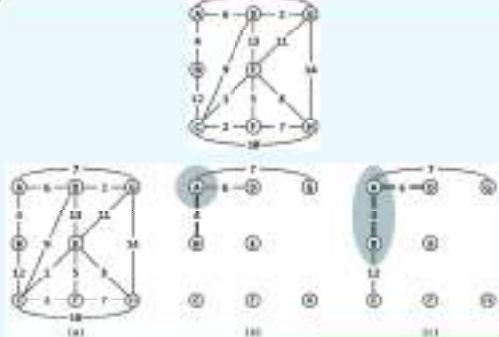
[www.jiangtinghua.com](http://www.jiangtinghua.com)

6. 图  
Prim 算法  
正确性

最佳解

[www.jiangtinghua.com](http://www.jiangtinghua.com)

1/3



(1)

(2)

(3)

618

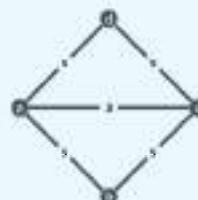
[www.jiangtinghua.com](http://www.jiangtinghua.com)

622

但是而非

设  $T_{\text{MST}}$  依次选取边  $(v_{1,2}, v_{1,3}, \dots, v_{1,n})$ ，构成支撑树  $T$ 。其中每一条边  $v_{1,i}$  的端点属于某棵 MST。

那么 MST 不唯一...

由此并不难确认，最终的  $T$  必是 MST (2-1)

由最短两边构成的支撑树，未必就是一棵 MST

反例...

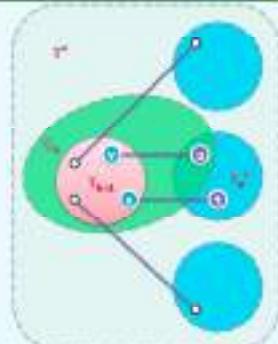
可行的证明方法

令在不增加总权重的前提下

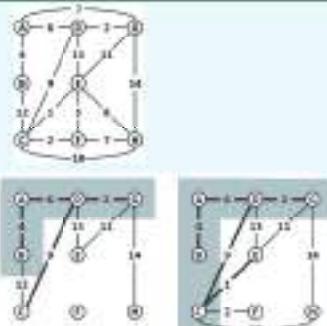
可以将任一 MST 转换为  $T$ 每一  $T_k$  都是某棵 MST 的子树， $1 \leq k \leq R$ 问题简化： $R=200$  题

数学归纳

623



2/3



(1)

(2)

(3)

619

[www.jiangtinghua.com](http://www.jiangtinghua.com)

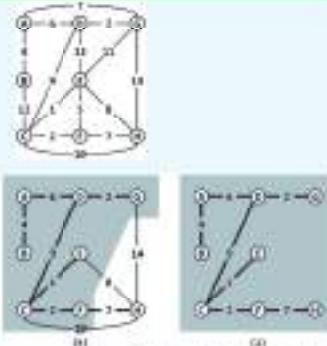
624

6. 图  
Prim 算法  
实践

借边一柄，圆睁环眼，凶猛虎爪，威武八蛇矛，飞马大叫：“三姓家奴休走！猪八戒在此！”吕布见了，弃了金枪戟，便欲要飞。

最佳解  
[www.jiangtinghua.com](http://www.jiangtinghua.com)

3/3



(1)

(2)

(3)

620

[www.jiangtinghua.com](http://www.jiangtinghua.com)

对于 $v_i$ 之外的每一顶点 $v$ , 令:

$\text{priority}(v) = \infty$  初值

于是有  $\text{优先级队列} \rightarrow$  空

为将 $v_i$ 扩充进 $V_{\text{cur}}$ , 可以

- 选出优先级最高的(即最小)顶点 $v_j$ , 及时对应该点 $v_j$ , 并将其加入 $V_{\text{cur}}$

- 然后, 更新 $v_{j+1}$ 之外每一顶点的优先级(数)



给定: 无向图 $G$ 及其中的顶点 $s$ 和 $t$

找出: 从 $s$ 到 $t$ 的最短路径及其长度

旅行者: 做经济的出行路线

路由器: 最快地把数据传送到目标位置

消防员: 多边形区域内的火灾报警器



注意: 优先级队列可能改变(降低)的顺序, 必须这样操作

因此, 只需

- 找新 $v_i$ 的每一邻接顶点 $v_j$ , 并按

-  $\text{priority}(v) = \min(\text{priority}(v), [\text{ok}, v])$

以上完全符合PFS的框架, 唯一要做工作无非是

按照`PriorityQueue::push`

编写一个优先级(数)更新器...



按问题的类型

- 无权图/有权图: BFS

- 单权有向图 //负权图?

单源点(Single-source)到各顶点的最短路径

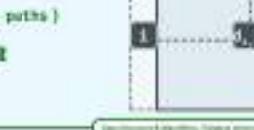
- 给定顶点 $s$ , 计算 $s$ 到其余各个顶点的最短路径及长度

- E. Dijkstra, 1959

所有顶点对之间的最短路径 (All shortest paths)

- 找出每对顶点 $s$ 和 $t$ 之间的最短路径及长度

- Floyd-Warshall, 1962



```
typedef PqFq< G, PriorityQueue, int> PQ; //从顶点s出发, 高级Priority类
template <typename Tv, typename Tg> //顶点类型, 边类型
struct Priority { //Priority类的顶点优先级更新器
    virtual void operator() (Graph<Tv, Tg*> g, int ok, int v) { //对v的相邻
        if (UNDISCOVERED == g->status(v)) return; //尚未被发现的邻居v, 接
        if (g->priority(v) > g->weight(ok, v)) { //如果
            g->priority(v) = g->weight(ok, v); //更新
            g->parent(v) = ok; //置父结点v -> ok = s
        }
    }
};
```



† Turing Award, 1972



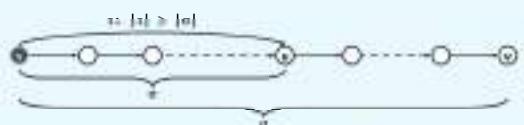
## 6. 图 Dijkstra算法 最短路径

## 6. 图 Dijkstra算法 最短路径树

△ 路径图中， $s$ 到每个顶点都有（至少）一条最短路径

// 并进化假设：每个顶点到  $s$  的最短路径唯一

△ 对同一组点而言，任何最短路径的前缀，也是一条最短路径

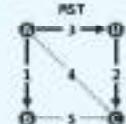
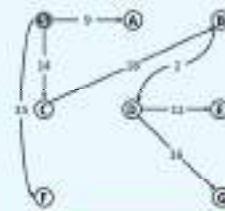
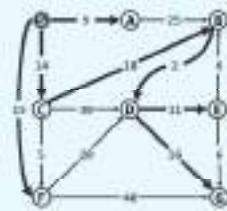


### Shortest Path Tree

△ 面向一起点  $s$  而言，所有最短路径唯一。

- 不会回路，因此

- 构成一棵树



△ 指明  $s$  是始节点，对所有的顶点应用

$$\text{dist}(s, u_1) \leq \text{dist}(s, u_2) \leq \dots \leq \text{dist}(s, u_n)$$

△ 最短距离最近邻  $u_1 = ?$

△ 沿任一最短路径，各顶点到  $s$  的距离更简单地变化

△  $u_1$  必与  $s$  直接相邻

$$\text{dist}(s, u_1) = w(s, u_1) < \infty$$

△  $w(u, s) < \infty$  仅当  $w(s, u_1) \leq w(s, u)$

△ 为找到  $u_1$ ，只需

在与  $s$  关联的各顶点中，找到对边权值最小者



### 6. 图示

Dijkstra算法  
算法

单链表

<http://www.csie.ntu.edu.tw/~u94004/teaching/algorithm/>

△  $u_1 = ?$ ,  $u_2 = ?$ , ...,  $u_k = ?$

△ 三角不等式： $\text{dist}(s, v) \leq \text{dist}(s, u) + w(u, v)$

△ 若设  $u_0 = s$ ，则有： $\text{dist}(s, u_i) = \min\{\text{dist}(s, u_j) + w(u_j, u_i) \mid 0 \leq j < k\}$

△ 算法？



$u_k = ?$

策略

641

令  $T_1 = \{(v_i)\}$  为开始，逐步构造  $T_2, T_3, \dots, T_n$ ，其中

- $v_1 \in S$
- $T_k = (V_k, E_k)$
- $|V_k| = k$
- $|E_k| = k+1, v_k \in V_{k+1}$



算法

642

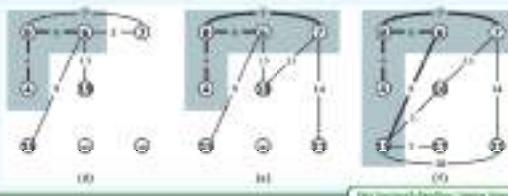
由以上分析，为由  $T_1$  构造  $T_{n+1}$ ，只需

- 将  $(V_1 \cup \{v_{n+1}\}, E_{n+1})$  视作原图的一个割
- 在该割的所隔(两)边中
- 找出最近者  $n_1 = (v_{n+1}, u_1) \in E_1$  (即  $u_1$  距  $v_{n+1}$  最近)
- 令  $T_{n+1} = (V_{n+1}, E_{n+1}) = (V_1 \cup \{u_1\}, E_1 \cup \{u_1\})$



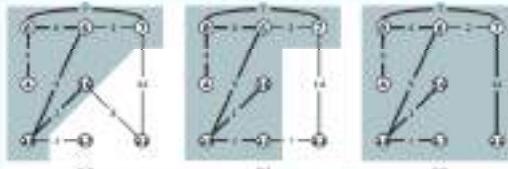
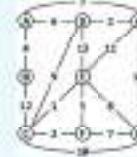
2/3

645



3/3

646



643

647

## 6. 图 Dijkstra 算法 实例

6. 图

Dijkstra 算法  
实例

邓伟群

[www.dengweiqun.com](http://www.dengweiqun.com)

6. 图

Dijkstra 算法  
实例

邓伟群

[www.dengweiqun.com](http://www.dengweiqun.com)

1/3

644

PFS

对于图 6.10 中各顶点  $v$ ，令：

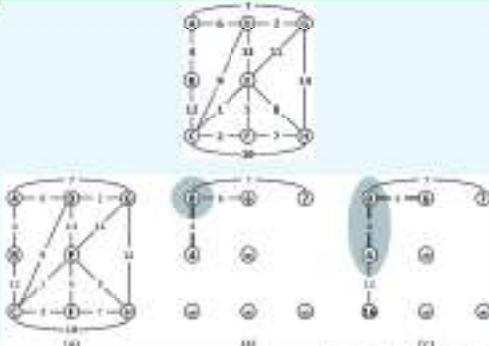
$\text{priority}(v) = (v, \text{最短})$  的距离

于是首先优先遍历图 6.10 中

为将  $T_1$  扩充为  $T_{n+1}$ ，可以

- 选出优先级最高的附近  $v$  (及其对应顶点  $u$ )，并将其加入  $T_1$
- 然后，更新  $V_{n+1}$  外所有顶点的优先级 (数)

648



△ 注意：优先级的值都可能改变（降低）的节点，必须立即更新

△ 因此，只用

- 枚举  $v$  的每一条接壤边，并赋
- $\text{priority}(v) = \min(\text{priority}(v), \text{priority}(v_i) + |u_i, v|)$

△ 以上同样完全符合DFS的框架，唯一要做的是工作无抖动

按照  $\text{prioritize}()$  模式

编写一个优先级（数）更新器...

[Data Structures & Algorithms, Integer Mathematics](#)

## 6. 图

双连通分量  
判定准则

带高亮之图部分，读者自拍图

邓桂科

[http://tangguikuo.com](#)

[Data Structures & Algorithms, Integer Mathematics](#)

## 实践

```
#include "DijkstraPriorityQueue.h"
#include "Graph.h" //从顶点出发，启动Dijkstra算法
template <typename Tv, typename Tvt> //顶点类型，边类型
struct DijkstraPU { //Dijkstra算法的带向量的优先级更新器
    virtual void operator()(Graph<Tv, Tvt*>* g, int uk, int v) { //对v的每个
        if (UNDISCOVERED != g->status(v)) return; //尚未被发现的邻居v，操作
        if (g->priority(v) > g->priority(uk) + g->weight(uk, v)) { //Dijkstra
            g->priority(v) = g->priority(uk) + g->weight(uk, v); //策略
            g->parent(v) = uk; //做标记
        }
    }
};
```

[Data Structures & Algorithms, Integer Mathematics](#)

## Brute-Force

△ 检验各内圈，如何标记各CC?

△ 参考很简单的版本：如何标记关节点？

△ 直接：对每一顶点*v*，遍历所有检查  $G(v)$  路径连通

△ 需要  $O(n^2 \cdot (n + e))$  时间，太慢！

而且，即便找出关节点，各CC仍需遍历

△ 改进：从任一顶点出发，构造DFS树

根据DFS树下的标记，甄别是否关节点

△ 比如，叶节点的不可能是关节点 //为什么？



[Data Structures & Algorithms, Integer Mathematics](#)

## 根顶点

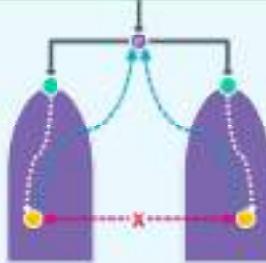
△ 根顶点是关节点 - 1F

树根至少有2棵树

△ 在DFS树中，不能检测

从根顶点*v*发出的树边数目

△ 那么，一般的内部顶点呢？



[Data Structures & Algorithms, Integer Mathematics](#)

也其十倍，不如将其一搬

6. 图  
双连通分量  
关节点

邓桂科

[http://tangguikuo.com](#)

## 关节点 + 双连通分量

△ 无双割路关节点：//articulation point, cut-vertex  
其删除之后，图别的连通分量增多 //connected components  
△ 无关节点的图，称作双（重）连通图 //bi-connectivity  
△ 较大的双连通子图，称作双连通分量 //Bi-Connected Components



[Data Structures & Algorithms, Integer Mathematics](#)

## 内顶点

△ 内顶点*v*是关节点

1FF

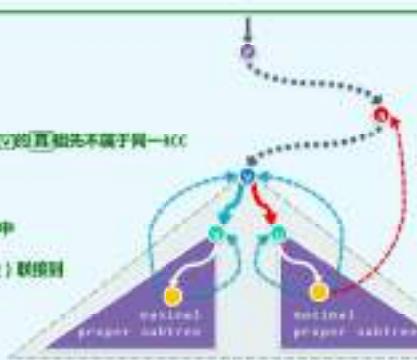
△ *v*的某棵根大直子树与*v*的直祖先不属于同一CC

1FF

△ 在*v*的某棵根大直子树中

没有顶点（经后向边）联接到

△ 直的直祖先

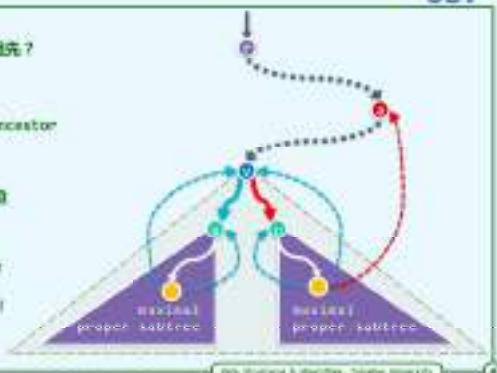


[Data Structures & Algorithms, Integer Mathematics](#)

祖先  
如何记录可联接的最高祖先？  
比如 $\text{hca}(A)$ ...

Highest Connected Ancestor  
 $\text{hca}(v) = \text{subtree}(v)$ 最后向上的  
能抵达的最高祖先

如何判断那个祖先更高？  
比如 $dTime$ ，越小越高！  
 $d(1)!!$



```
switch (status(u)) {
    case DISCOVERED:
        type(v, u) = BACKWARD;
        if (u != parent(v))
            hca(v) = min(hca(v), dTime(u)); //更新hca(v)，缩小范围
        break;
    default: //VISITED (digraphs only)
        type(v, u) = dTime(v) < dTime(u) ? FORWARD : CROSS;
        break;
}
```

### 6. 图 双连通分量 算法

邓佳群  
deng@tsinghua.edu.cn

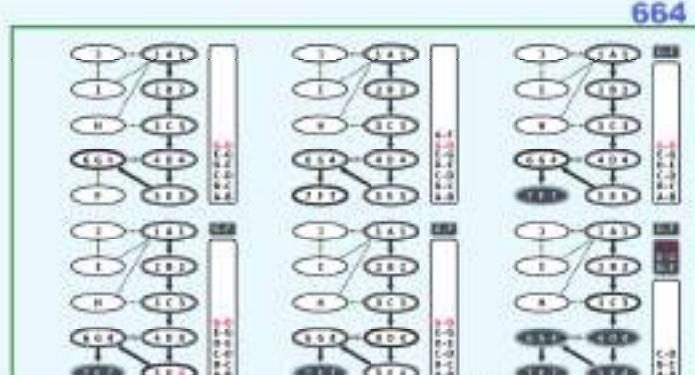
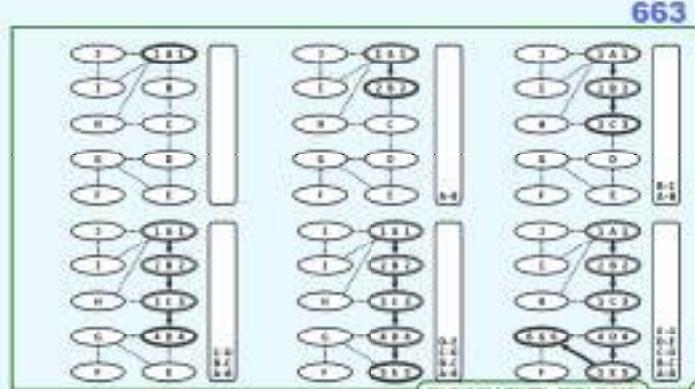
### 6. 图 双连通分量 实例

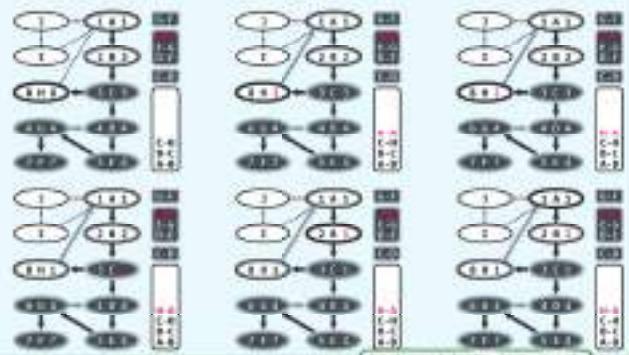
邓佳群  
deng@tsinghua.edu.cn

#### Graph::BCC()

```
#define hca(x) (dTime[x]) //利用此宏里面的dTime[]充当hca()
template classname Tv, typename Tc //顶点类型, 边类型
void Graph<Tv, Tc>::BCC(int v, int8 clock, Stack<int> S) {
    hca(v) = dTime[v] = ++clock; status(v) = DISCOVERED; //发现v
    S.push(v); //将v入栈, 以下枚举v的所有邻居u
    for(int u = firstNbr(v); -1 < u; u = nextNbr(v, u))
        switch (status(u)) {
            /* ... 模4的状态分别处理 ... */
            status(v) = VISITED; //对v的访问结束
        }
}
```

```
switch (status(u)) {
    case UNDISCOVERED:
        parent(u) = v; type(v, u) = TREEL; //拓展树边
        BCC(u, clock, S); //从u开始遍历, 跳回v
        if (hca(u) < dTime(v)) { //若v检测到指向u的直接边
            hca(v) = min(hca(v), hca(u)); //既奇必偶
        } else { //否则, 若v为关节点, u下辖为一个bcc, 且其中顶点均非正常集中于u的顶点
            while (v != S.pop()); //首先弹出所有该顶点, 逻辑错误需另求另解处理
            S.push(v); //最后一个顶点(关节点)重新入栈
        } //尽管同一顶点可作为(关节点)重复入栈, 但必须不第一次
        break;
}
```





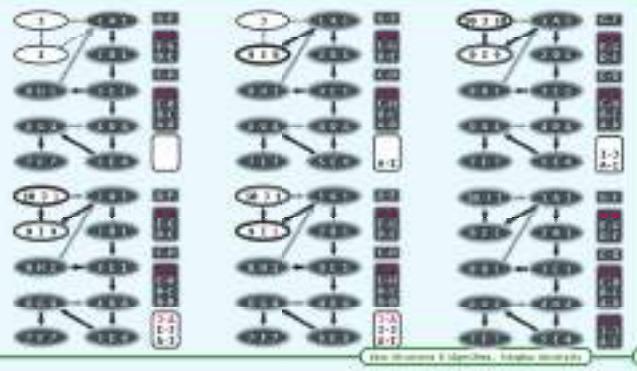
## 6. 图

Kruskal算法  
算法

想坚持作图，虽然以为什  
真在苦下笔，真在苦中读  
本想同君生，相教何太急

Junhui DENG

dengjh@zjhu.edu.cn



## 贪心策略

◆ Prim-Dijkstra算法：

代价最小的边，迟早会被采用

次小的边，亦是如此

再次小的，则未必 // 这路！

◆ Kruskal：贪心策略

根据代价，从小到大依次尝试选取

只要“安全”，就加入该边

◆ 但是，每步是否最优？ 全局最优？

◆ 确实，Kruskal很幸运...



6. 图  
双连通分量  
复杂度

邓桂科  
dengjh@zjhu.edu.cn

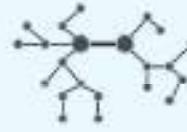
## 算法框架

◆ 给定G的一个森林： $F = \{V_i, E'_i\}$ ,  $G = V \cup E = \{V, E\}$

◆ 初始化：  $F = \{V\}$  (包含n棵树(各含1个顶点)和0条边)  
将所有边按明代价值排序

◆ 迭代： 检测当前最廉价的边

若e的两个点来自F中不同的树，则  
 $\{E' + E'_i\} \cup \{e\}$ , 然后  
将e相连的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

◆ 运行时间和常数的DFS遍历，也是 $O(n + n)$

自行验证：操作的复杂度也不过如此

◆ 除根节点外，还需一个容量为 $\sigma(n)$ 的堆存所有已访问的边  
为支撑该日，另需一个容量为 $\theta(n)$ 的结构体

◆ 最后： 该算法是否也适用于非连通图？

在有向图中如何实现对边的计算？

◆ Strongly-connected component:

Kosaraju's algorithm

Tarjan's algorithm

## 正确性

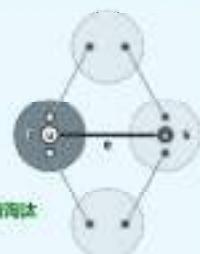
◆ 原理： Kruskal引入的每条边都属于某棵MST

◆ 设边 $e = (v, u)$ 的引入导致树T和S的合并

◆ 若： 将 $(T \cup S, E)$ 操作原操作 $\sigma$ 的倒  
则：  $e$ 检测该倒的一条假边

◆ 在确定应引入 $e$ 之前

该边的所有两边将被Kruskal忽略，且只可能用不小于 $e$ 而被忽略



Following the leader,  
the leader, the leader  
We're following the leader  
wherever he may go  
He won't be home till morning,  
till morning, till morning  
He won't be home till morning  
because he told us so

### Kruskal 算法 实现

Junhui DENG

dengjh@zjhu.edu.cn

## 6. 图

### Kruskal 算法 并查集

Junhui DENG

dengjh@zjhu.edu.cn

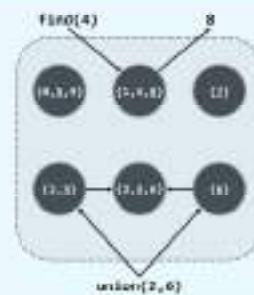
## 排序

- 贪心能排序吗？  
答：能，耗时  $\Theta(\text{edges}) = \Theta(n \log n)$  // 到目前为止上界
- 实际上，大多数情况下只需考虑  $\Theta(n)$  边
- 将所有边按优先级从低到高排序 // 比如，以加实现
  - 直接， $\Theta(n)$
  - 删除并恢复， $\Theta(\log n)$
  - 共迭代  $\Theta(n)$  次 // 实际中往往远小于  $n$ ，尤其是对于稠密图
- 总共 =  $\Theta(n) + \Theta(\log n) \times \Theta(n) = \Theta(n \log n)$

[Implementation Summary, Analysis and Review]

## Union-Find

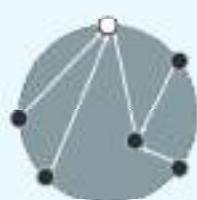
- Union-Find 问题  
给定一组互不相交的等价类  
由各自的一个成员作为代表
- Singleton  
初始时各包含一个元素
- Find(x)  
找到元素 x 所属等价类
- Union(x, y)  
合并 x 和 y 所属等价类
- Kruskal = Union-Find



## Union-Find

- Tarjan-SGI:  $\Theta(n \alpha(n))$  amortized time per Union/Find
- $\alpha(n)$ : inverse Ackermann function
  - $\log^* n = \Theta(\log \log \dots \log n)$  s.t.  $\log^*(\log^{\dots}(\log n)) < 2$
  - $\log^{**} n = \Theta(\log \log^* \dots \log^*(\log n))$  s.t.  $\log^{**}(\log^*(\dots(\log^* n))) < 2$
  - $\dots$
  - $\log^{k+1} n = \dots$
  - $\dots$
  - $\log^{k+2} n = \dots$
  - $\dots$
- $\alpha(n) = \Theta(\log^{**} n)$  s.t.  $\log^{**} n < 3$
- $\Theta(n)$  (目前可观察到宇宙范围内的粒子总数) =  $\Theta(10^{80}) < 4$

## 图检测



- 如何高效地检测回路，并且合并树？
- 领导节点 (leader node)
  - 每棵树选举出一个首领
  - 领节点指向 parent
  - parent 指针可找到对应的首领
  - leader.parent = NO\_PARENT
- 假设引入新边  $e = (u, v)$  时
  - 由 parent 指针，找到并比较 leader(u) 和 leader(v)
  - 若引入造成回路，则  $\text{leader}(u) = \text{leader}(v)$
- 因此检测稳定性可以合并，只须如何高效地合并呢？ [Memory, Space Intensity]

[Implementation Summary, Analysis and Review]

## 树合并

- 合并树  $T(u)$  和  $T(v)$  后， $\text{leader}(u).parent = \text{leader}(v)$
- 注意：合并后，树的深度 =  $\Theta(n)$   
总会不幸达到上界吗？很有可能！于是 ...
- 对 Leader 的  $\Theta(n)$  次查询，共需  $\Theta(n^2)$  时间
- 改进：  $\text{leader}.parent = -(\text{nodes})$ 

```
if (leader[u].parent > leader[v].parent)
    leader[u].parent = leader[v];
else leader[v].parent = leader[u];
```
- 使用改进后算法，树合并耗时度 =  $\Theta(\log n)$  // VMN, p.142
- 因此，查找 leader 的总耗时度 =  $\Theta(n \log n)$



[Implementation Summary, Analysis and Review]

## 复杂度

- 初始化森林， $\Theta(n)$
- 建立 PQ， $\Theta(n)$
- 迭代  $\Theta(n)$  次： 检出队首并调整 PQ,  $\Theta(\log n)$   
回路检测 + 边插入， $\Theta(\log n)$   
树合并， $\Theta(1)$
- 总共 =  $\Theta(\text{edges}) + \Theta(\log n)$
- 对于图 G，迭代次数 =  $c = \Theta(n)$ , 且  $\Theta(n \log n) < \Theta(n^2)$   
需要快吗？
- [Fredman & Tarjan-87]  
采用 Fibonacci Heap，对耗时可做到  $\Theta(n \log \log n)$

[Implementation Summary, Analysis and Review]

## 更多算法

681

• Boruvka (1926):

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...

• Vassotsky (1960):

每次增加一条边

如果出现回路，将回路上最长的边删除

Implementation: Boruvka, Minimum Spanning Tree

6. 图

Floyd-Marshall算法

用法

https://tinyurl.com/yd2qzv6x

## 更新结果

682

• Gabow, Tarjan, Spencer, & Tarjan:  $\Theta(n + \log(\beta(n, n)))$ Efficient algorithms for finding minimum spanning trees  
in undirected and directed graphs

Combinatorica, vol. 6, 1986, pp. 109-122

 $\beta(n, n) = \text{smallest } i \text{ s.t. } \log(\log(\log(\dots \log(n)\dots))) < n/n$   
where the logs are nested  $i$  times

• Freedman &amp; Willard: 极值均为不大于常数时，最坏情况仅需线性时间

Trans-dichotomous algorithms for  
minimum spanning trees and shortest pathsACM SIGART Symp. Foundations of Comp. Sci., 1988, pp. 719-725, <https://doi.org/10.1145/62212.62285>

Implementation: Boruvka, Minimum Spanning Tree

## 从Dijkstra到Floyd-Marshall

686

+ 假定图G，计算图中所有点对之间的最短距离

+ 应用：搜索图G中心点 center vertex

<中心的半径 radius(G, c) = 所有顶点到c的最大距离  
中心点 = 半径最小的顶点c

+ 跟踪：依次将顶点作为源点，使用Dijkstra算法

时间  $= n \cdot d(n^2) = O(n^3)$  —— 可否更快？

+ 思想：图矩阵 = 最短路径矩阵

+ 效率： $O(n^3)$ ，与执行n次Dijkstra相同 —— 既然如此，为何还要用n次？

+ 约定：形式简单，算法复杂，便于实现

允许负权边（尽管仍不能有负权环路）

Implementation: Dijkstra, Floyd-Warshall

## 更新结果

683

• YAO (1995):  $\Theta(n + \log \log n)$ 

• Karger, Klein, &amp; Tarjan: 随机算法，期望的运行时间

A randomized linear-time algorithm

to find minimum spanning trees

J. ACM, vol. 42, 1995, pp. 321-338

• CHAZELLE (2000): MST与union-find问题的复杂度相同

Implementation: Boruvka, Minimum Spanning Tree

## 问题特点

687

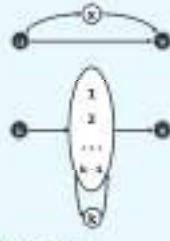
+ u和v之间的最短路径可能长

0) 不存在通路，或者

1) 直接连接，或者

2) 通过路径  $(u, [k], v)$  + 量短路径  $([k], v)$ + 指定有向点数映射  $: 1, 2, \dots, n$ + 假设  $: d^k(u, v)$ 

= 中途只经过图k个顶点，取u和v的量短路径长度

=  $d(u, v)$  (if k = 0)=  $\min(d^{k-1}(u, v), d^{k-1}(u, [k]) + d^{k-1}([k], v))$  (if k ≥ 1)

Implementation: Dijkstra, Floyd-Warshall

## 相关话题

684

• Proximity\_Shape //O(n log n)

Euclidean MST

Delanney Triangulation

Gabriel Graph

Relative Neighborhood Graph

• Steiner MST //NP-hard

Approximation

Implementation: Boruvka, Minimum Spanning Tree

## 暴力递归

688

weight dist( node \* u, node \* v, int [k] ) { //暴力递归

if ([k] &lt; 1) return w[u, v]; //递归基：中途不经过任何点

minDist = dist( u, v, [k - 1] ); //递归：中途必须经过 k - 1 个点

for each node [k] ≠ ( u, v ) { //枚举剩余节点，分别作为图 k 个点

u2vDist = dist( u, [k], [k - 1] ) + dist( [k], v, [k - 1] ); //递归

minDist = min( minDist, u2vDist ); //优化

}

return minDist;

Implementation: Dijkstra, Floyd-Warshall

◆  $T(n, k) = \begin{cases} 1 & k=0 \\ T(n-1) + 2 \times T(k-1) & k>0 \\ 2^{k+1} - (k-2)^{k+1} & k=0 \end{cases}$   
 $T(n, k) = O(n^k)$  // 递归仅只是一对节点所用的时间

◆ 注意：能力算法中，存在大量的重叠递归调用

◆ 挑战：如何优化计算？如何避免？你应该如何记得...?

◆ 动态规划 dynamic programming

维护一张表，记录需要反复计算的数据 // 因此，只需计算一次

◆ 根据事先约定的规则，从数据集合中指出符合特定条件的数据

◆ 对于算法的构建而言，属于最为基本且必要的静态操作

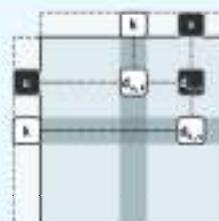
◆ 虽然然，基本的数据结构并不能高效地兼顾静态查找与动态修改

基础结构	查找	插入/删除
无序列表	$O(n)$	$O(n)$
有序列表	$O(\log n)$	$O(n)$
有序树	$O(n)$	$O(1)$
有序图	$O(n)$	$O(n)$

◆ 那么，能结合原有方法的优点？如何做到？

```
// Initialization
for (int u = 0; u < n; u++)
    for (int v = 0; v < n; v++)
        if (dist[u][v] == u[v][v]; min[u][v] = -1)

// Iteration
for (int k = 0; k < n; k++)
    for (int u = 0; u < n; u++)
        for (int v = 0; v < n; v++)
            if (dist[u][v] > dist[u][k] + dist[k][v])
                dist[u][v] = dist[u][k] + dist[k][v]; min[u][v] = k;
```



◆ 数据项之间，依照各自的关键码彼此区分

call-by-key

◆ 为此，关键码之间的必须支持

- 大小比较与

- 相等比较

◆ 数据集合中的数据项



统一表示和实现为类的 entry 形式

◆ 时间

$$\mathcal{O}(n^2) + \mathcal{O}(n^3) = \mathcal{O}(n^3)$$

与 n 次两两 DJKstra 相同

◆ 空间

存储一些  $n \times n$  的数据， $\mathcal{O}(n^2)$

每个单元为两个整数

◆ 对于桶选择和桶密图，你会分别选择哪种算法？

◆ 根据  $min[i][j]$  阶段，如何重构出  $u[i][j]$  之间的数据组织段？

为此，你需要多长时间？

◆ template <typename K, typename V> struct entry { // 可以根据类

K key; V value; // 关键码、数据

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

entry( entry& e ) : key(e.key), value(e.value) {} // 克隆

// 比较器、判断器（从此，不必严格区分谓语及其对应的关键码）

bool operator<( entry& e ) { return key < e.key; } // 小于

bool operator>( entry& e ) { return key > e.key; } // 大于

bool operator==( entry& e ) { return key == e.key; } // 等于

bool operator!= ( entry& e ) { return key != e.key; } // 不等于

};

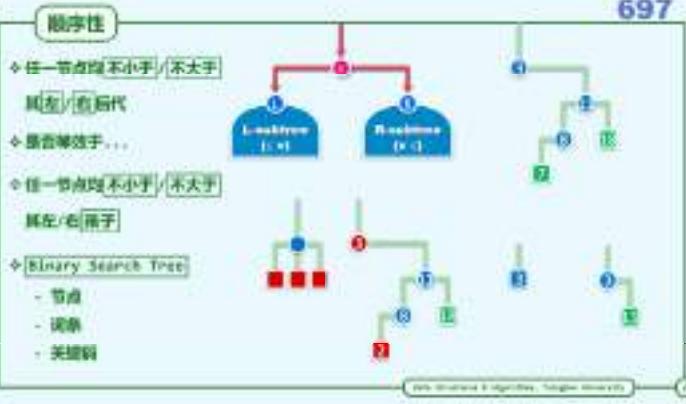
There's nothing in your head  
the sorting hat can't see.  
So try me on and I will tell  
you where you ought to be.



- Harry Potter and the Sorcerer's Stone

## 7. 二叉搜索树

## 7. 二叉搜索树



**对外接口**

```

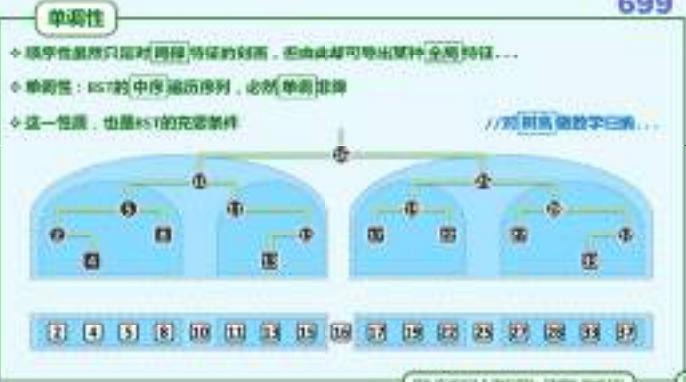
template <typename T> class BST : public BinTree<T> { //二叉搜索树
public:
    //无virtual修饰，以提高生成效率
    virtual BinNodePosi<T> search( const T& t ) const; //查找
    virtual BinNodePosi<T> insert( const T& t ); //插入
    virtual bool remove( const T& t ); //删除
protected:
    /* ..... */
};
```



**内部接口**

```

template <typename T> class BST : public BinTree<T> { //二叉搜索树
public:
    /* ..... */
protected:
    BinNodePosi<T> father; //某节点的父亲
    BinNodePosi<T> connectSA( /* 3 + 4 节点 */ );
    BinNodePosi<T>, BinNodePosi<T>, BinNodePosi<T>;
    BinNodePosi<T>, BinNodePosi<T>, BinNodePosi<T>, BinNodePosi<T>;
    BinNodePosi<T> establish( BinNodePosi<T> );
};
```



## 7. 二叉搜索树

算法及实现  
查找

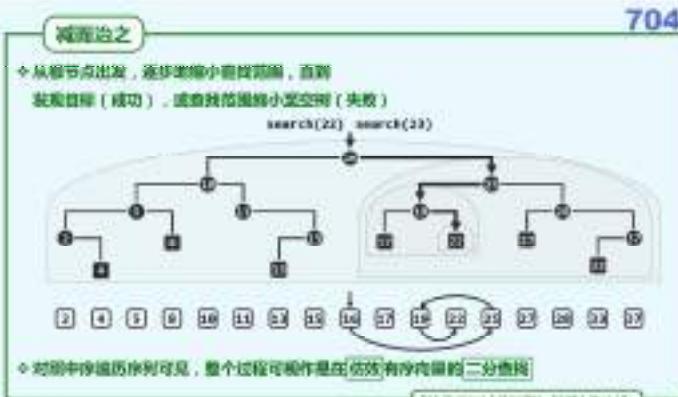
单链表

<http://www.codingbook.cn>

## 7. 二叉搜索树

概述  
接口

单链表  
<http://www.codingbook.cn>



```

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( //典型的尾递归，可改为迭代
    BinNodePosi(T) & v, const T & e, BinNodePosi(T) & hot) {
    //当前(子)树根，目标关键码，记忆热点
    if (!v || (e == v->data)) return v; //足以确定失败：成功，或者
    hot = v; //先记下当前(非空)节点，然后...
    return searchIn( (e < v->data ? v->left : v->right), e, hot );
} //运行时间正比于返回节点v的深度，不超过树高h(n)


```

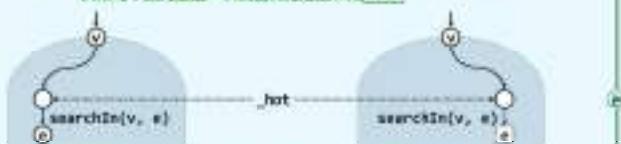
```

template <typename T> BinNodePosi(T) & BST<T>::insert(const T & e) {
    BinNodePosi(T) & x = search(e); //直接目标(插入结点的设置)
    if (!x) { //如果不存在元素，仅仅在查找失败时才实施插入操作
        x = new BinNodePosi(e, _hot); //将新结点插入树中，以_hot为父亲
        _size++; updateHeightAbove(x); //更新全树高度。更新x及其后代结点的高度
    }
    return x; //无论x是否存在子树中，至此必有x->data == e
} //验证：对于首个节点插入之类的边界情况，均可正确处理

//时间主要消耗于search(e)和updateHeightAbove(x)
//均摊性正比于插入节点e的深度，不超过树高h(n)


```

◆返回的引用值：成功时，指向一个关键码e的确实存在的节点  
失败时，指向最后一次试图插入的空节点NULL



◆失败时，不妨假想地将此空节点，转换为一个数值为e的新叶节点  
如此，依然满足es1的充要条件：而且更加强化...

◆无论成功与否：返回值总是成功地指向返回节点，而\_hot恰指明库中节点的父亲

## 7. 二叉搜索树

算法及实现  
插入

邓伟群

<http://www.hust.edu.cn>

## 7. 二叉搜索树

算法及实现  
删除

邓伟群

<http://www.hust.edu.cn>

◆先做search(e)确定插入位置及方向

再将新节点作为叶子插入

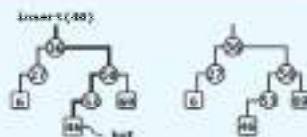
◆若e尚不存在，则

\_hot为新节点的父亲

v = search(e)为\_hot对新孩子的引用

◆于是，只需

\_hot通过v指向新节点



◆若e (即) 的第一子树为空

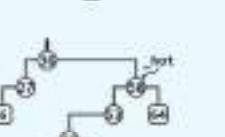
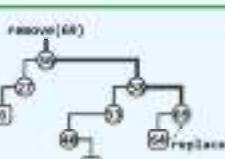
则可将其替换为另一子树 (64) //可能为空

◆验证：

如此操作之后，二叉搜索树的

有序性依然满足

相序性依然满足



```

*template <typename T> static BinNodePosi(T)
removeAt( BinNodePosi(T) x, BinNodePosi(T) & hot ) {
    BinNodePosi(T) w = x; //实际被删除的节点，初值同x
    BinNodePosi(T) succ = NULL; //实际被替换节点的接替者
    if ( ! HasLChild( "x" ) ) succ = x->LChild; //左子树为空
    else if ( ! HasRChild( "x" ) ) succ = x->RChild; //右子树为空
    else { /* ...左、右子树非空的情况，需要要处理... */ }
    hot->parent = w->parent; //记录实际被删除节点的父亲
    if ( ! succ ) succ->parent = hot; //将接替节点的级数指向hot相同
    release( w->data ); release( w ); //释放被删除节点
    return succ; //返回接替者
} //此步情况仅需O(1)时间


```

Data Structures & Algorithms, English Version 2e

- 由以上的实现与分析，BST主要接口search()、insert()和remove()的运行时间  
在最坏情况下，均性正比于树高 $O(n)$
- 因此，若不希望数据倾斜树高，则对实际的数据而言  
较之此前的向量和列表等数据结构，BST将无法体现出明显的优势
- 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低 $O(n)$ ，性正比于树（列表）的长度
- 那么，出现此类退化或极坏情况的概率有多大？  
或者，从平均数角度的角度看，二叉树数据的性究竟如何呢？
- 以下就两种常用的随机树 $T(n)$ ，即BST的平均性能做一个分析和对比

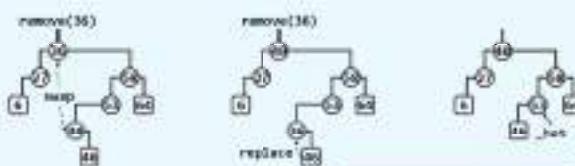
Data Structures & Algorithms, English Version 2e

小结： $x$  (36) 左、右子并存

时：调用BinNode::succ()找到 $x$ 的直接后继（必无左孩子）；交换 $x$  (36) 与 $w$  (40)

子树问题转化为删除 $w$ ，可按前一情况处理

尽量避免节点在中途一直不变，但是总必将重新调整

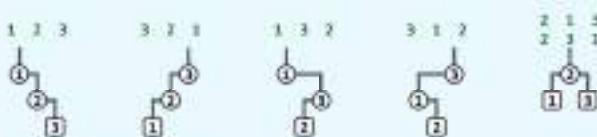


Data Structures & Algorithms, English Version 2e

考虑 $n$ 个互异键值 $\{n_1, n_2, \dots, n_n\}$ ，对任一排列 $\sigma = (n_{\sigma(1)}, n_{\sigma(2)}, \dots, n_{\sigma(n)})$ ...

从空树开始，反向调用insert()顺序将 $n$ 项依次插入，得到 $T(\sigma)$

与 $\sigma$ 相对应的 $T(\sigma)$ ，称作[随机生成] (randomly generated)



若固定任一序列 $\sigma$ 作为输入的键串中项 $\{n_i\}$

则当 $n$ 个互异键值随机生成的二叉搜索树，平均高度为 $O(\log n)$

```

*template <typename T> static BinNodePosi(T)
removeAt( BinNodePosi(T) & x, BinNodePosi(T) & hot ) {
    /* ... */
    else { //若x的左、右子树并存，则
        w = w->succ(); swap( w->data, x->data ); //若x与最近后继w直接相邻
        BinNodePosi(T) u = w->parent; //将问题降阶化，消除非二叉的节点u
        if ( u->left == w || u->right == w ) u->succ = w->succ; //继续递归情况：u的键值相等
    }
    /* ... */
}

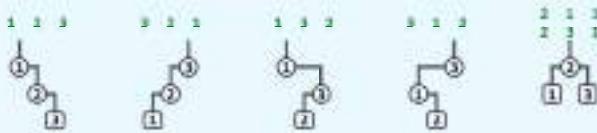

```

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 $O(n)$

Data Structures & Algorithms, English Version 2e

$n$ 个互异节点，在遵守键属性的前提下，可能构造出若干种链接关系

如此所得的BST，称作[随机组成] (randomly composed)



由 $n$ 个互异节点随机组成的BST，若共计 $f(n)$ 棵，则有

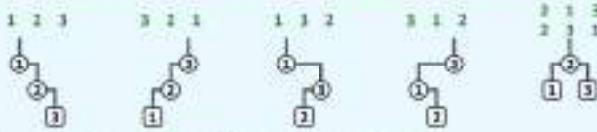
$$T(n) = \sum_{k=1}^n SP(k-1) \cdot SP(n-k) = \text{Catalan}(n) = (2n)! / (n+1)! / n!$$

假定所有BST等概率出现，则平均高度为 $O(\sqrt{n})$

平衡  
键树高度

单链表  
[tinyurl.com/bsc-sll](http://tinyurl.com/bsc-sll)

按两种接口所估计的平均性能，差异极大——谁更可信？孰更接近于真实情况？



第一口中，弱弱的 $\Theta(1)$ 被重写估计为多次——故难过于 $\Theta(n)$

若想算法因而使用succ()，则每棵BST都有越来越左偏的倾向

理想情况在实践中并不常见，关键因往往在“单键某些特性”随次序生成

因为 BST 数据生成不足为凭

Data Structures & Algorithms, English Version 2e

## 7. 二叉搜索树

平衡  
理想与适度

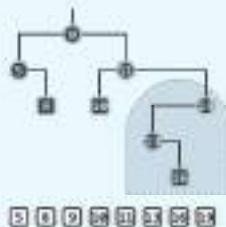
树 译 码

https://timguba.edu.cn

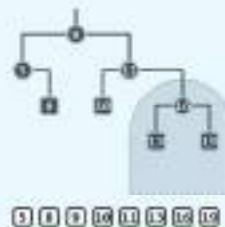
## 等价BST

△ 上下对称：数据类型不相同时，嵌套关系可对称倒置

△ 左右互换：中序遍历序列完全一致，全局单弱判断



1 2 3 4 5 6 7 8 9

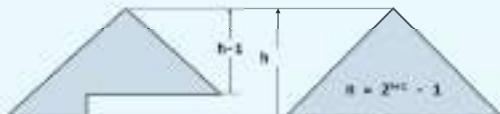


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Data Structures &amp; Algorithms - Height-balanced

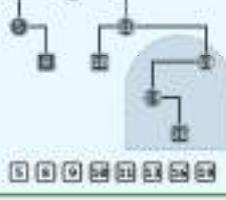
## 理想平衡

△ 节点数固定时，兄弟子树高度越接近（平衡），全树直径越小于离散

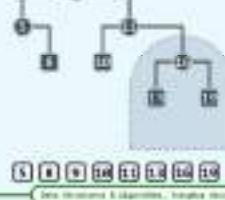
△ 由n个节点组成的二叉树，高度不低于 $\lceil \log_2 n \rceil$ ——恰为 $\lceil \log_2 n \rceil$ 时，称作**理想平衡**△ 大数相当于**完全树**，其最坏情况：叶节点只出现在子树底部的两端——条件过于苛刻

Data Structures &amp; Algorithms - Height-balanced

## 限制条件 + 局部性

△ 各种bst都可操作bst的某一子集，相应地满足精心设计的**限制条件**1) 单次修改操作后，至多 $O(\log n)$ 处局部不再满足限制条件2) 可在 $O(\log n)$ 时间内，使这些局部（以整全树）重新满足

1 2 3 4 5 6 7 8 9

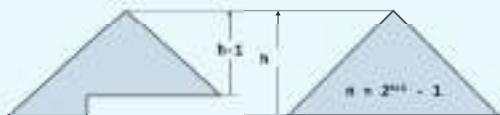


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Data Structures &amp; Algorithms - Height-balanced

## 适度平衡

△ 理想平衡出现概率极低，维护成本过高，故须采用适度平衡策略

△ 退一步海阔天空：高度差进块不超过 $O(\log n)$ ，即可称作**适度平衡**△ 适度平衡的bst，称作**字典二叉插入树**（DST）

Data Structures &amp; Algorithms - Height-balanced

## 等价变换 + 旋转调整

△ 限制类操作 $O(1)$ ，也可迅速转换为一等价的 $O(n)$ ——为此，只需 $O(\log n)$ 耗时 $O(1)$ 次旋转△ zig和zag：仅涉及 $O(2)$ 个节点，只要调整其间的数据关系；均属于**局部**操作，**基本**操作△ 调整之后： $n/2$ 深度加/减1，子（全）树高度的变化幅度，上下 $O(1)$ △ 实际上，经过不超过 $O(n)$ 次旋转，等价的bst将可相似简化 [习题解析 7-15]

## 7. 二叉搜索树

平衡  
等价变换

树 译 码

https://timguba.edu.cn

## 7. 二叉搜索树

AVL 树  
适度平衡

树 译 码

https://timguba.edu.cn

## 平衡因子

$\text{BalFact}(v) = \text{height}(\text{lc}(v)) - \text{height}(\text{rc}(v))$

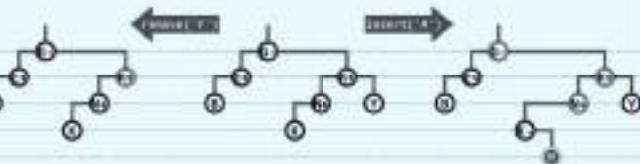
如 Adel'son-Vol'sky & E. Landis (1962):  $\forall v, |\text{BalFact}(v)| \leq 2$



Data Structures & Algorithms, Tongyu Ma et al.

## 失衡与重平衡

按AVL规则插入或删除节点之后，AVL平衡性可能破坏——如何恢复？



能力不足时，旋转的顺序交换

同层性：所有的旋转都在同层进行 //每次只调n(1)次

偶数性：在每一深度只向右旋转并旋转更多一次 //共 $O(\log n)$ 次

Data Structures & Algorithms, Tongyu Ma et al.

## AVL = 适度平衡

高度为h的AVL树，至少包含 $S(h) + fib(h+2) - 2$ 个节点

$\therefore S(h) = 1 + S(h-1) + S(h-2)$

$\therefore S(h) + 1 = [S(h-1) + 1] + [S(h-2) + 1]$

$\therefore fib(h+3) = fib(h+2) + fib(h+1)$

$$S(2) + 1 = 5 \\ = fib(5)$$

$$S(3) + 1 = 8 \\ = fib(6)$$

$$S(4) + 1 = 13 \\ = fib(8)$$

$$S(5) + 1 = 21 \\ = fib(13)$$

$$S(6) + 1 = 34 \\ = fib(21)$$

$$S(7) + 1 = 55 \\ = fib(34)$$

$$S(8) + 1 = 89 \\ = fib(55)$$

反过来，由n个节点构成的AVL树，高度至多为 $O(\log n)$

Data Structures & Algorithms, Tongyu Ma et al.

## 7. 二叉搜索树

AVL树  
插入

样例

样例  
样例  
样例

## 7. 二叉搜索树

AVL树  
重平衡

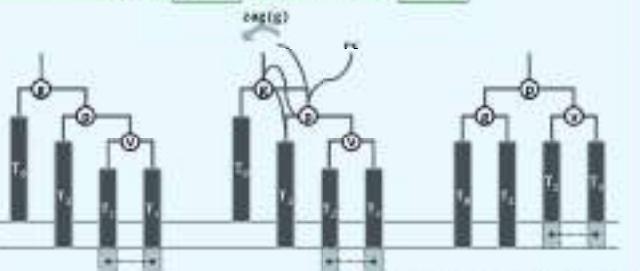
样例  
样例  
样例

Data Structures & Algorithms, Tongyu Ma et al.

## 单旋

同时可能多个失衡节点，操作按 $\alpha$ 不小于 $\beta$ 优先

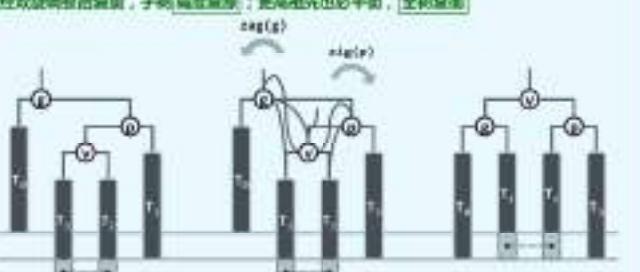
$\alpha$ 层单独调整后平衡，子树尚未平衡；更高祖先也必平衡，全树平衡



## 双旋

同时可能多个失衡节点，操作按 $\alpha$ 不小于 $\beta$ 优先

$\alpha$ 层单独调整后平衡，子树尚未平衡；更高祖先也必平衡，全树平衡



接口

```

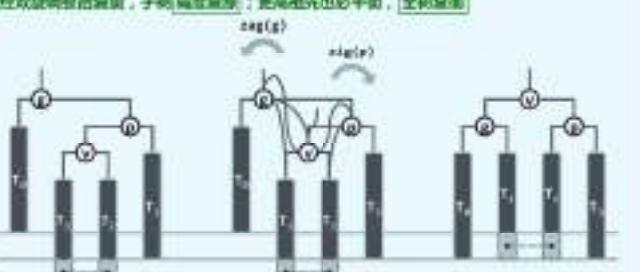
#define Balanced(x) (status((x).lc) == status((x).rc)) // 理想平衡
#define BalFact(x) (status((x).lc) - status((x).rc)) // 平衡因子
#define AvlBalanced(n) ((-2 < BalFact(n)) && (BalFact(n) <= 2)) // AVL平衡条件

template <typename T> class AVL : public BST<T> { // 由BST派生
public: // BST::search()等接口，可直接应用
    void insert(T const& t); // 插入操作
    void remove(T const& t); // 删除操作
};
```

Data Structures & Algorithms, Tongyu Ma et al.

同时可能多个失衡节点，操作按 $\alpha$ 不小于 $\beta$ 优先

$\alpha$ 层单独调整后平衡，子树尚未平衡；更高祖先也必平衡，全树平衡



Data Structures & Algorithms, Tongyu Ma et al.

```

    template <typename T> BinNodePosi(T) AVL::insert( const T & e ) {
        BinNodePosi(T) & x = search( e ); if ( !x ) return x; //若目标不存在
        BinNodePosi(T) xx = x + new BinNodePosi( e, _het ); _size++; //将目标节点
        //此时，若x的父结点_het过高，则祖父必有失衡。故以下从_het起，向上逐级检查各代祖先
        for ( BinNodePosi(T) g = _het; g; g = g->parent )
            if ( !g->isBalanced( *g ) ) { //一旦发现g失衡，则通过调整恢复平衡
                rotateLeft( -g ) = rotateLeft( tallerChild( tallerChild( g ) ) );
                break; //x是孙子，高祖父时高度必然减低，其祖先亦必如此，故调整结束
            } else //否则（在偶然平衡的祖先处），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）
        return xx; //返回新节点：最多只要一次调整
    }

```

## 实现

```

    template <typename T> bool AVL::remove( const T & e ) {
        BinNodePosi(T) & x = search( e ); if ( !x ) return false; //若目标不存在
        BinNodePosi(T) xx = x + new BinNodePosi( e, _het ); _size--; //将目标的值存在
        //从_x到_het逐级向上，依次检查各代祖先
        for ( BinNodePosi(T) g = _het; g; g = g->parent )
            if ( !g->isBalanced( *g ) ) { //一旦发现g失衡，则通过调整恢复平衡
                g = fromParent( *g ) + rotateLeft( tallerChild( tallerChild( g ) ) );
                updateHeight( g ); //并更新其高度
            } //可能需要进行O(logn)次调整；无论是自做过调整，全局高度均可能下降
        return true; //删除成功
    }

```

## 7. 二叉搜索树

AVL树

删除

邓佳辉

dengjiahui@zjhu.edu.cn

## 7. 二叉搜索树

AVL树

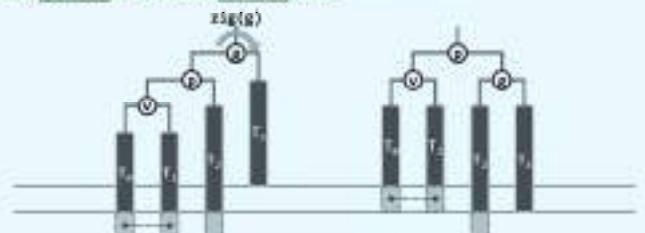
(3+4)-重构

邓佳辉

dengjiahui@zjhu.edu.cn

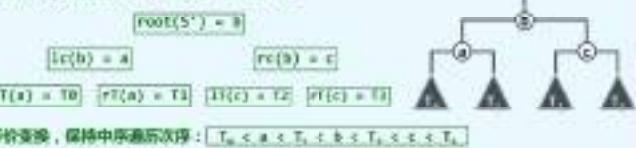
## 单旋

- 同时至多一个失衡节点x，首个可能就是e的父亲\_het
- e经单旋调整后偏倒，子树高度未必减低；更高祖先仍可能失衡
- 因有失衡修复规则，可能调用 $\sigma(\log n)$ 次调整



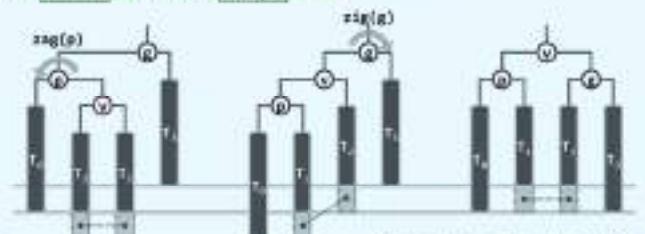
## 算法

- $\text{rot}_l(x)$ 为偏倒的失衡节点，有四种外三行： $x \sim p \sim y$
- 按中序遍历次序，将其重命名为： $x \sim b \sim c$
- 它们总共拥有互不相同的四棵（可能为空的）子树
- 按中序遍历次序，将其重命名为： $T_a, c, T_b, c, T_c$
- 将原以c为根的子树t，替换为一棵新子树s！



## 双旋

- 同时至多一个失衡节点x，首个可能就是e的父亲\_het
- e经单旋调整后偏倒，子树高度未必减低；更高祖先仍可能失衡
- 因有失衡修复规则，可能调用 $\sigma(\log n)$ 次调整



## 实现

```

    template <typename T> BinNodePosi(T) BST::connect124(
        BinNodePosi(T) a, BinNodePosi(T) b, BinNodePosi(T) c,
        BinNodePosi(T) T0, BinNodePosi(T) T1, BinNodePosi(T) T2, BinNodePosi(T) T3)
    {
        a->lc = T0; if ( T0->parent == a ) a->rc = T1;
        a->rc = T1; if ( T1->parent == a ) updateHeight(a);
        c->lc = T2; if ( T2->parent == c ) updateHeight(c);
        c->rc = T3; if ( T3->parent == c ) updateHeight(c);
        b->lc = a; a->parent = b;
        b->rc = c; c->parent = b; updateHeight(b);
        return b; //孩子树新的根节点
    }

```

```

统一调整

// template<typename T> BinNodePosi(T) BST<T>::rotateLeft( BinNodePosi(T) v ) {
    BinNodePosi(T) p = v->parent, g = p->parent; // 父亲、祖父
    if ( IsChildL( *v ) ) { // zig
        if ( IsChildL( *p ) ) { // zig-zig
            p->parent = g->parent; // 向上调整
            return connect34( v, p, g, v->lcl, v->rcl, p->lrc, g->rcc );
        } else { // zig-zag
            v->parent = g->parent; // 向上调整
            return connect34( p, v, g, p->lcl, v->lcl, v->rcl, g->rcc );
        }
    }
    else { /*... zig-zig & zig-zag ...*/ }
}

```

Data Structures & Algorithms, Design Patterns

**综合评价**

- 优点：无须旋转，插入或删除，最好情况下的时间复杂度均为 $O(\log n)$ 。
- 缺点：牺牲高度或平衡因子，为此需改造元素结构，或额外存储。实测跟深度与理论结构有差距。  
插入/删除后的旋转，成本不同。  
最坏情况，最多需旋转 $O(\log n)$ 次（Knuth：平均 $\log \log n$ 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，全局拓扑结构的变化量可能高达 $O(\log n)$ 。
- 有没有更好的结构呢？//保持兴趣

Data Structures & Algorithms, Design Patterns

↑ 节点v一旦被访问，随即转移至树根

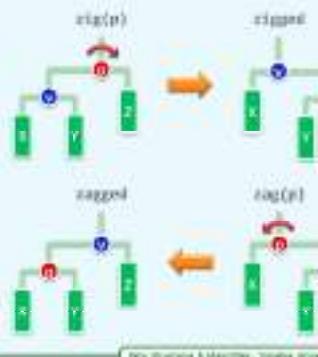
↑ 一步一步往上升

- 向下向上，逐层单旋

- zig( u->parent )

- zag( v->parent )

- 两两+带向树根的单旋



↑ 伸展过程的改善

a) 读取x之后：做zig(z)

b) 读取x之后：做zag(z)

↑ 就连延伸树的

策略而言

这取决于

- 树的初始形态和

- 节点的访问次序

c) 读取y之后：做zig(y)

d) 读取y之后：做zag(y)

Data Structures & Algorithms, Design Patterns

## 8. 高级搜索树

伸展树  
逐层伸展

我一步一步往上升  
在最高点乘着叶片往路飞

攀枝科  
<http://tiny.cc/meyarw>

↑ 每次数据(属性性)的(算不进数)演变：每一周期累计 $\Theta(n^2)$ ，空间 $\Theta(n)$

(a) 初始结构

(b) search(x)之后

(c) search(x)之后

(d) search(x)之后

(e) search(x)之后

(f) search(x)之后

(g) search(x)之后

Data Structures & Algorithms, Design Patterns

↑ locality：刚刚访问过的数据，很有可能要快地再次被访问  
这一现象在信息处理过程中很常见 // BST就是这样的一个例子

↑ BST：刚刚被访问过的节点，很有可能更快地再次被访问  
下一将要访问的节点，很有可能就在刚刚访问过节点的附近

↑ 通常的 $n$ 次遍历( $n \gg n = |\text{BST}|$ )，采用avl共需 $\Theta(n \log n)$ 时间

↑ 对局部性，精度快？ // 依次自底向链表

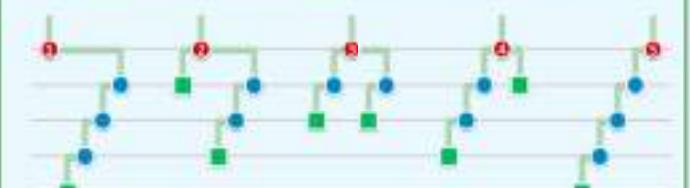
↑ 常数： 节点一旦被访问，随即调整至树根 // 如此，下次访问即可...

↑ 问题： 如何实现这种局部性？调整过程自身的局部性和树形？

Data Structures & Algorithms, Design Patterns

↑ 全树延伸伸展是单链表结构，等价于一般列表

↑ 被访问节点的深度，是局部性的算术级数演变： $\{n-1, n-2, n-3, \dots, 3, 2, 1\}$



↑ 问题的症结点已确定，便可针对性地改进...

Data Structures & Algorithms, Design Patterns

## 8. 高级搜索树

伸展树  
双层伸展

要领道：“不用全打开，怕惊起来倒霉事。  
你光惊均匀叫数第一招一招的吓收场。”

算法解  
[http://tinyurl.com/3qjwv4z](#)

## zig-zig / zig-zag

◆ 断链修复：一旦访问节点，到链表的长尾指针被重置

// 会翻转

◆ 循环情况不致持续发生！

单趟体操操作，分解为 $O(\log n)$ 时间！

// 具体见，详见习题 8-2



## zig / zag

◆ 谁是v只有父亲，没祖父呢？

◆ 此时必有 $\text{parent}(v) == \text{parent}(\text{parent}(v))$ ，且  
树中调整中，这种情况最多（在最后）出现一次

◆ 相具体形态，做首次旋转：zig(r)或zag(r)



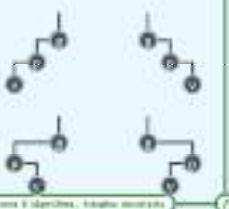
## 双层伸展

◆ D. Sleator &amp; R. E. Tarjan

[Self-Adjusting Binary Trees](#)

J. ACM, 32:652-686, 1985

◆ 梅思的暗语：向上追溯两层，而找一层



## zig-zag / zig-zig

◆ 增加树深度完全等效！

◆ 可设置伸展到无二段！

◆ 难道，就这样平淡无奇？



## zig-zig / zig-zag

◆ 调整次序之后，同样的操作差异，将彻底地改变整棵...



## 8. 高级搜索树

伸展树

算法实现

到了所在，往了脚，便把这驴和这一起折起来，再腰也只比腰低，放在巾箱里面。

算法解

[http://tinyurl.com/3qjwv4z](#)

## 伸展树接口

◆ template &lt;typename T&gt;

class Splay : public BST<T> { // 由 BST 派生  
protected: BiNodePos<T> splay(BiNodePos<T> v); // 特殊伸展至根  
public: // 伸展树的查找也会引起结构的结构性调整，故 search() 也需要写

BiNodePos&lt;T&gt; &amp; search(const T &amp; e); // 搜索 [重写]

BiNodePos&lt;T&gt; insert(const T &amp; e); // 插入 [重写]

bool remove(const T &amp; e); // 删除 [重写]

## 伸展算法

```

+ template <typename T> void rotate(T& p, const T& v) {
    if (!v) return NULL; RTreeNode<T> g; //父结点, 被父
    while (g = v->parent) g->parent = NULL; //自下而上, 逐级取消链接
    RTreeNode<T> gg = g->parent; //遍历之后, v将被设置为父
    gg->children[0] = v;
    if (IsLeftChild(g, p)) { // zig-zig }
    else if (IsLeftChild(g, p)) { // zig-zag }
    else if (IsRightChild(g, p)) { // zagi-zag }
    else { // zagi-zig }
    if (g->parent == NULL) //如果根结点是g, 则g成为新树根; 否则, gg成为新结点为左孩子
    else (g = gg->attachLeftChild(g, v); attachLeftChild(g, v));
    if (g->parent == NULL) attachRoot(g, v);
    //如果伸展结束时, 必有g == NULL, 但g可能非空
    if (p == v->parent) { // 如果非根结点, 只需在链表中插入(或删除一次) }
    v->parent = NULL; return v; //释放游标, 通过转换
}

```

Data Structures &amp; Algorithms, Higher Edition



## 伸展算法

```

+ if ( IsChild(p, v) ) {
    if ( IsLeftChild(p, v) ) { // zig-zig
        attachLeftChild(v, p->rc);
        attachLeftChild(v, v->rc);
        attachLeftChild(v, g);
        attachLeftChild(v, v);
    } else { // zagi-zig }
    else
        if ( IsLeftChild(p, v) ) { // zig-zagi-zig }
        else { // zagi-zagi }
}

```



Data Structures &amp; Algorithms, Higher Edition

```

+ template <typename T> RTreeNode<T> * Splay(T& p, const T& e) {
    // 调用Splay(e)的内部接口定位目标节点
    RTreeNode<T> p = searchIn(_root, e, _root == NULL);
    // 无论成功与否, 最后被访问的节点都将被置为根
    _root = splay(p ? p : _root); // 成功, 失败
    // 总是返回根节点
    return _root;
}

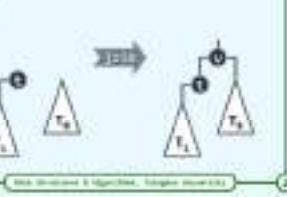
+ 伸展树的旋转操作, 与常规STL::ROTATE()不同
    可能会改变树的拓扑结构, 不再属于游标操作

```

Data Structures &amp; Algorithms, Higher Edition

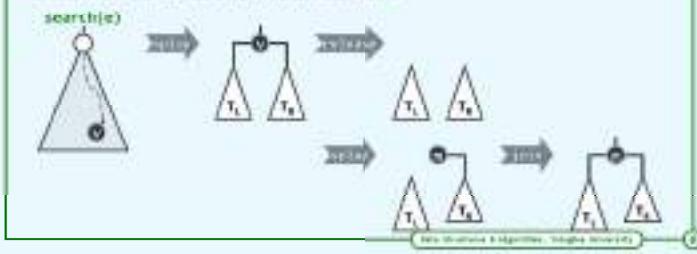
## 插入算法

实现方法：调用bst标准的插入算法，再将新节点伸展至根  
其中，首先调用Splay(e)；  
之后，Splay后的Splay::search()已集成了splay()操作  
直接（失败）之后，\_root即是根节点  
除此之外，何不把新节点在游标附近完成新节点的插入...。



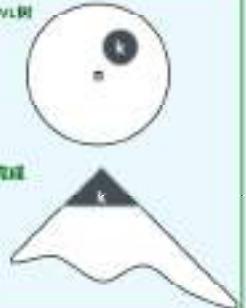
## 删除算法

实现方法：调用bst标准的删除操作，再将\_node伸展至根  
同样地，Splay::search()查找（成功）之后，删除节点即被移除  
除此之外，何不把新节点在游标附近完成新节点的插入...



## 综合评价

无须记录节点两侧或子集游标；操作实现简单易于——优于AVL树  
分摊复杂度 $O(\log n)$ ——与AVL树相当  
+ 期望性能。缓存命中率提高时 ( $\Theta(n) \ll n \ll \Theta(n^2)$ )  
效率显著可以提高——由 $\Theta(n)$ 到 $\Theta(\log n)$   
任何 $n$ 级别的 $n$ 次查找，都可在 $O(n\log n + n\log n)$ 时间内完成  
仍不能杜绝最差情况的出现  
不适合于对效率敏感的场合  
+ 算法的分析稍嫌复杂——好在有前缀的方法



## 8. 高级搜索树

64KB might be enough for anybody.  
— B. Gates, 1981

白面奸一听说不由火了：“又是个百日劫持！”  
他继续说：“西白又不懂事，你收当西白他才舒服。你就忍一忍，一两天很快过去了.....”

平生好

http://tiny.cc/meyarw

## 越来越小的内存

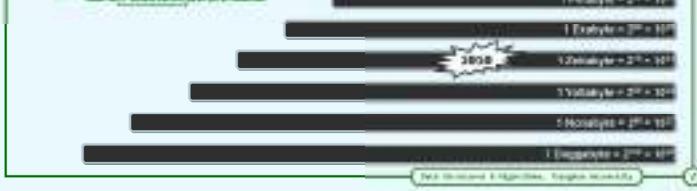
今世略： 存储器？不就是无限可数个寄存器吗？

Turing： 存储器？不就是无限长的纸带吗？

◎ ◎ ◎

是纸带存储器的增长速度

◎ ◎ ◎ 基于问题规模的增长速度



Data Structures &amp; Algorithms, Higher Edition

◆ 高级别的  
数据仓库 / 内存容量  
◆ 今天需要的数据集  
◆ 为单位容量  
1990 : 10MB / 1TB = 10  
2000 : 1TB / 1TB = 1000  
2005 : 50TB / Global climate  
500TB / Nuclear  
250TB / Turbulent combustion  
60TB / Parkinson's disease  
10TB / Protein folding

◆ 例如，相对而言... 内存容量是在... 不断减少！

◆ 为什么不把内存做得更大？

◆ 物理上，存储器的容量越大/小，访问速度就越慢/快

◆ 1970, E. Bayer & L. McCreight

◆ 平衡的多路 (multi-way) 搜索树

◆ 节点再合并，根跟非节点

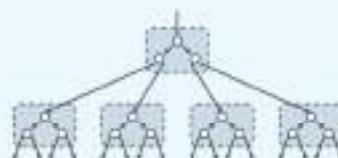
每2代合并：4路

每3代合并：8路

每d代合并： $n = 2^d$ 路， $n = 1$ 个关键码

◆ 基础上与AVL完全等价

——既然如此，为何还要引入B-树？



◆ 事实：不同容量的存储器，访问速度差异很大

◆ 以磁盘与内存为例：10G / 1G > 1000倍

◆ 第一次内存访问需要一秒，第一次外存访问则相当于一天

◆ 为避免1次外存访问，我们宁愿访问内存100次、200次，甚至...



◆ 多路存储系统，都是分级组织的——Caching

通常用的数据尽可能放在更高级、更大的存储器中  
实在找不到，才向低级别、更大的存储器索要

◆ 算法的I/O频度 = 频繁在不同存储级别之间的传输次数

算法的读写执行时间，往往主要取决于此

◆ 多路存储系统中如果n=时，可针对外形查找，大大减少I/O次数

◆ 难道，AVL还不够？比如，若有n=10个记录...

每次查询需要  $\log(2, 10^9) = 36$  次I/O操作，每次只能读出一个关键码，不得不先

◆ B-树又能如何？

充分利用外存对块级访问的高效支持，将此节点转化为优点

每下降一级，都以底层节点为单位，读入一组关键码

◆ 具体多大一组？视底层数组块大小而定， $n = \text{Keys} / \text{pg}$

比如，当前多数数据库系统的采用  $n = 256 \sim 300$

◆ 同时上例，若取  $n = 256$ ，则每次操作只需  $\log(256, 10^9) \approx 4$  次I/O

◆ 事实：从硬盘中读取10G，与读取1GB几乎一样快

◆ 批量式访问：以页 (page) 或块 (block) 为单位，使用缓冲区 // <std::vector<...> ...

```
define _BSIZE_ 512 // 缓冲区默认容量
int setup() { // 定制缓冲区
    FILE* fp; //流
    char* buf; //缓冲区
    int Node; // _10FBF | _10LB | _10RF
    size_t size; //缓冲区容量
    int flush(FILE* fp); //强制清空缓冲区
}
```

◆ 效果：单位字节的平均访问时间大大增加

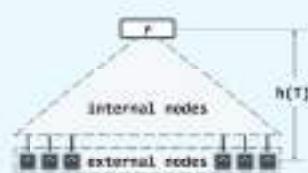
// <boost/thread/thread.hpp>

◆ 所谓“叶子”，即平衡搜索树 ( $n \geq 2$ )

◆ 外部节点的深度统一相等

所有叶节点的深度统一相等

◆ 则高h = 外部节点的深度



◆ 内部节点都有

不超过  $n - 1$  个关键码：  $K_1 < K_2 < \dots < K_n$

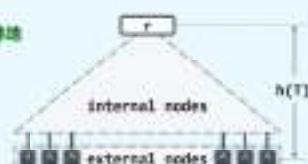
不超过  $n$  个分支：  $A_0 < A_1 < A_2 < \dots < A_n$

◆ 内部节点的分支数  $n + 1$  由不能太少，具体地

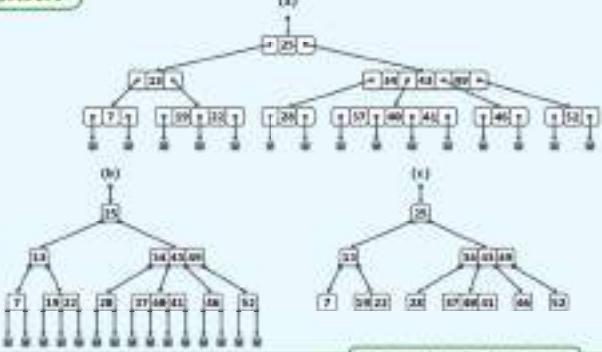
分支：  $n \leq n + 1$

剩余：  $[n/2] \leq n + 1$

◆ 放弃带作  $([n/2], n)$ -树



## 紧凑表示



## 8. 高级搜索树

B-树  
查找高坐天边望深海  
每寸搜素着这天下  
寻觅着那个“它”

...按照模型的该部署，用现时的最高计算能力根和百分之一秒的聚变过程，就算大约二十年时间。而研究过程中所遇到的模型被模型运行，这使得模型的实际应用成为不可能。

并行 科

www.imooc.com/244

## 实例

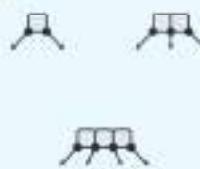
```
#include <B_tree.h>
// B-树
2-3-树, (2,3)-树, 最简单的B-树 // John Hopcroft, 1970
    各(内部)节点的分支数, 可能是2或3
    各节点所含key的数量, 可能是1或2
```

分支 = 4

2-3-4-树, (2,3,4)-树

各节点的分支数, 可能是2, 3或4
    各节点所含key的数量, 可能是1, 2或3

◆ 数据结构(4维n-树), 对称数据(红黑树)大有裨益



## 算法

◆ 指针节点作为当前节点 // 定义类

只遍历的节点非外部节点

在当前节点中遍历数据 // 算法头部

若找到目标关键码, 则

返回(查找成功)

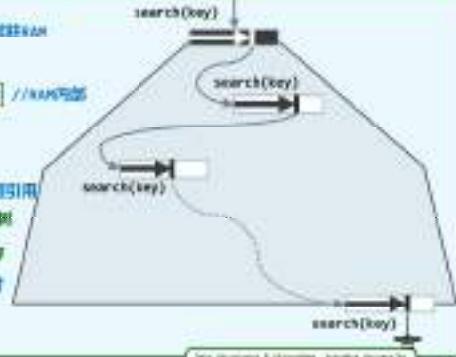
否则 // 此时第一对下层引用

沿引用, 转至对子树

将其根节点装入内存

// I/O, 然为耗时

更新当前节点



Data Structures &amp; Algorithms, Design Patterns

## BTNode

```
#template <typename T> struct BTNode { // B-树节点
    BTNodePosi(T) parent; //父
    vector<T> key; // 节点内键
    vector<BTNodePosi(T)> child; // 孩子节点 (其浓度总比key多一)
    BTNode() { parent = NULL; child.insert(0, NULL); }
    BTNode(T k, BTNodePosi(T) lc = NULL, BTNodePosi(T) rc = NULL) {
        parent = NULL; // 作为根节点, 莫得初始
        key.insert(0, k); // 仅一个关键码, 以及
        child.insert(0, lc); child.insert(1, rc); // 两个孩子
        if (lc) lc->parent = this; if (rc) rc->parent = this;
    }
}
```

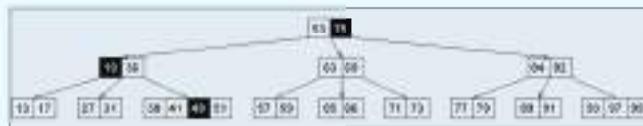
## 实例

◆ (2,3)-树: 55 37 38 69 41 75 23 94 77 79 51 37 29 31

92 93 17 73 13 66 59 49 63 65 71 69 27 33 36

成功查找: 75, 19, 49

失败查找: 5, 45



Data Structures &amp; Algorithms, Design Patterns

## BTree

```
#define BTNodePosi(T) BTNodePosi(T)* // B-树节点指针
#template <typename T> class BTree { // B-树
protected:
    int _size; int _order; BTNodePosi(T) _root; // 关键码总数, 阶次, 根
    BTNodePosi(T) _bot; // search()函数访问的非空节点位置
    void solveOverflow( BTNodePosi(T) ); // 因插入而上溢后的分离处理
    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
    bool insert(const T & e); // 插入
    bool remove(const T & e); // 删除
}
```

## 实现

```
#template <typename T> BTNodePosi(T) BTree::search( const T & e ) {
    BTNodePosi(T) v = _root; _bot = NULL; // 从根节点出发
    while (v) { // 通常查找
        Rank r = v->key.search( e ); // 在当前节点对应的向量中顺序查找
        if (r < 0 || r == v->key.size) return v; // 若成功, 则返回; 否则...
        _bot = v; v = v->child[r+1]; // 沿引用指向对应的下层子树, 并装入其根
    } // 若真 to 跳出, 则意味着抵达外部节点
    return NULL; // 失败
}
```

Data Structures &amp; Algorithms, Design Patterns

## 性質

△ 約定：根節點非空

△ 數據存儲中的產地

△ 運行時間上操作子[1/0次數]

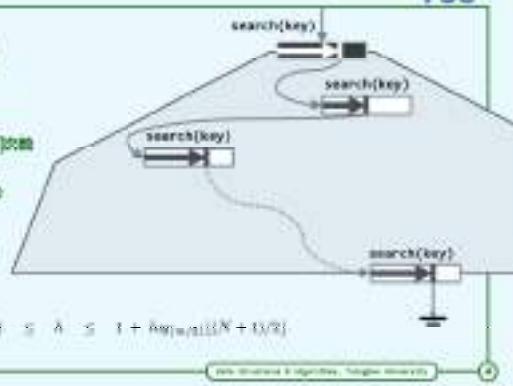
△ 在每一深度至多一次 I/O

△ 評運行時間

- O(停止節點的深度)

-  $O(h)$

△ 可以推導： $\log_2(N+1) \leq h \leq 1 + \log_{\sqrt{2}}((N+1)/2)$



## 8. 高級搜索樹

B-樹  
插入

出遇見，在這夢幻國度，孤獨的一首

清音讓你，分離再分離

那樣

www.englishbookclub.org

## 最大樹高

△ 有  $n$  個關鍵碼的二叉B-樹，最大高度 = ?

△ 為此，內部節點應尽可能“瘦”

$$n_k \geq 2 \times \lceil n/2 \rceil^{k-1}, \forall k > 0$$

△ 考慮外部節點所在層

$$\begin{aligned} N+1 - n_0 &\geq 2 \times \lceil n/2 \rceil^{h-1} \\ h \leq 1 + \log_{\sqrt{2}}(N+1)/2 &= O(\log_2 N) \end{aligned}$$

△ 相對於 BST： $\log_{\sqrt{2}}(N/2) / \log_2 N = 1 / (\log_2 N - 1)$

範例  $n = 256$ ，樹高  $\leq 1/0$  次數 ) 的降低率  $1/7$

// 用 4 年上完大學，還是 28 年？



## 算法

△ template <typename T>

bool Btree<T>::insert( const T & e ) {

    Node\* root = \_root; v = search( e );

    if ( v ) return false; // 認為 e 不存在

    Rank r = \_root->key.search( \* ); // 在節點\_root 中確定插入位置

    root->key.insert( r + 2, e ); // 將新關鍵碼插入到 root 节點

    root->child.insert( r + 2, NULL ); // 建立一個空子樹指針

    size++; checkOverflow( root ); // 檢查上溢，擇機分裂

    return true; // 插入成功



## 最小樹高

△ 有  $n$  個關鍵碼的二叉B-樹，最小高度 = ?

△ 為此，內部節點應尽可能“胖”

$$n_k \leq m^k, \forall k \geq 0$$

△ 考慮外部節點所在層：

$$\begin{aligned} N+1 - n_0 &\leq m^h \\ h \geq \log_m(N+1) &= O(\log_m N) \end{aligned}$$

△ 相對於 BST： $(\log_m N - 1) / \log_2 N = \log_2 m - \log_2 2 \approx 1 / \log_2 m$

範例  $n = 256$ ，樹高  $\leq 1/0$  次數 ) 的降低率  $1/8$



## 分裂

△ 上溢節點中的關鍵碼依次為  $x_0, \dots, x_{m-1}$

△ 取中位數  $y = \lceil m/2 \rceil$ ，以關鍵碼  $y$  为界劃分

$$[x_0, \dots, x_{y-1}], [x_y], [x_{y+1}, \dots, x_{m-1}]$$

△ 將關鍵碼  $y$  上升一層，并

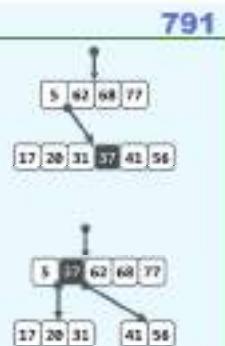
分裂(split)：以所得的兩個節點作為左、右孩子

△ 請認：如果分有問題

左、右孩子所含關鍵碼數必然符合二叉B-樹的條件

$$y = \lceil m/2 \rceil \geq \lceil m/2 \rceil - 1$$

$$y + y - 1 \leq m - \lceil m/2 \rceil - 1 = \lceil m/2 \rceil - 1$$



## 意義與價值

△ BST 究竟可貴多麼？

△ 常數高度為  $h$  的 BST ( 俗稱 AVL ) 可以令節點的數目  $f$

$$f = 32 \cdot 2^h, f + 2 \approx 32 \cdot 2^h + 2 \approx 2^h \cdot 10 \approx 99 \quad // 全球人口$$

$$f = 77 \cdot 2^h, f + 2 \approx 77 \cdot 2^h + 2 \approx 2^h \cdot 10 \approx 23 \quad // 全球人口的家數總數$$

$$f = 133 \cdot 2^h, f + 2 \approx 133 \cdot 2^h + 2 \approx 2^h \cdot 10 \approx 40 \quad // 國際象棋可能的局面總數$$

$$f = 266 \cdot 2^h, f + 2 \approx 266 \cdot 2^h + 2 \approx 2^h \cdot 10 \approx 78 \quad // 国際象棋每步由中括號表示總數$$

△ 從此可知， $h$  的大小，能夠更多地體現在實用方面

通過適應適當的節點指標 ( $m$ )，各子樹高度之間巨大的進度差異

△ 另外，在算法方面， $m$ -樹也有其獨特的價值與地位...

## 消分裂

△ 上溢節點的父節點本已饱和，對在接續被提升的關鍵碼之後，出現上溢此類，大可重用前述，兩端分裂

△ 上溢可能時候產生，並非兩端上溢後

纵然樹狀情況，亦不必割捨 // 若果真極端割捨...

△ 可令相應的關鍵碼自成節點，作為新的樹根

這也是  $m$ -樹裏的唯一弱點 // 極率多大？

△ 注意：新的樹根沒有兩個分支

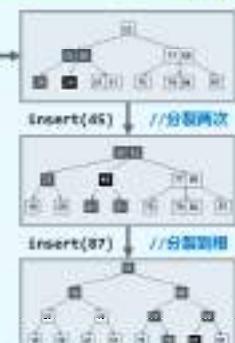
// 可見樹根的右不能遵守下限  $m/2$

△ 總體執行時間複雜度比手動分裂次數，不超過  $O(h)$



实例  
+ 2-3-树：

55 97 26 89 41 75 19 34 77 79 51



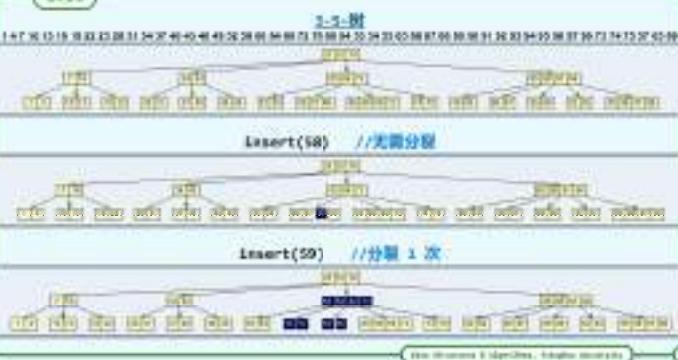
## 上溢修整

```

    if (u->child[0] == null) //若u的孩子们非空，则将u设为父节点
        for (Rank j = 0; j < _order - 1; j++) u->child[j + 1]->parent = u;
    BTNodePos1(T) v = v->parent; //当前的父节点
    if (!v) //若v为空，将调整之（变成长满一枝，新节点点放好两度）
        (_root = p = new BTNode<T>()); p->child[0] = v; v->parent = p;
    Rank r = 1 + p->key.size(); v->key.remove(r); //p中指针r的指向的键
    p->key.insert(r, v->key.remove(r)); //节点关键码上升
    p->child[r].insert(r + 1, u); u->parent = p; //新节点u与父节点p且函数overflow(p); //上升一级，如仍必要则继续分裂——至多进级2次
    }
}

```

## 实例



## 8. 高级搜索树

B-树  
删除

别影，空了影，反光镜。

流动了情，也要合井，候归了零。

也不经不生不灭不离的相影。

拜佳客

http://tinyurl.com/34944a

## 实例

Insert(59) //分裂 1 次



## 算法

```

+ template <typename T>
bool BTnode<T>::comove(const T & e) {
    BTNodePos1(T) v = search(e);
    if (!v) return false; //确认e存在
    rank = v->key.search(e); //确定e在v中的秩
    if (v->child[0]) { //若v有子，即
        BTNodePos1(T) u = v->child[0]; //在右子树中一直向左，即可
        while (u->child[0]) u = u->child[0]; //找到e的后继（必属于某叶节点）
        v->key[r] = u->key[0]; u->key[0] = e; r++; //并与之交换位置
    } //至此，v必然位于最底层，且其中第一个关键码就是待插入者
    v->key.remove(r); v->child.remove(r + 1); _size--;
    solveoverflow(v); return true; //如尚必要，需递归地调用
}

```

## 上溢修整

```

+ template <typename T> void BTnode<T>::valuetOverflow(BTNodePos1(T) v) {
    if (_order >= v->child.size()) return; //返回基：不再上溢
    Rank s = _order / 2; //确定(此时_order = key.size() + child.size() + 1)
    BTNodePos1(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++) { //分出最底层节点u (效率低可改进)
        u->child.insert(j, v->child.remove(s + 1)); //将键_order-s-1个孩子
        u->key.insert(j, v->key.remove(s + 1)); //在键_order-s-1个关键码
    }
    u->child[_order - s - 1] = v->child.remove(s + 1); //移出最底层的孩子
}

```

## 旋转

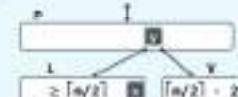
◆节点p下溢时，必恰好包含  $\lceil n/2 \rceil - 2$  个关键码 +  $\lceil n/2 \rceil - 1$  个分支

◆根其左、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  $\lceil n/2 \rceil$  个关键码

    将关键码y从L移至p中（作为最小关键码）

    将关键码x从R移至p中（替代其叶端关键码y）

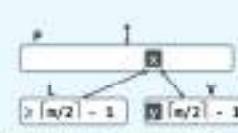


◆如此操作之后，树形乃至整棵树都重新满足B-树条件

下溢修整完毕

2) 若R存在，且至少包含  $\lceil n/2 \rceil$  个关键码

    需舍弃L



## 合并

③ L树的根不存在，键值所含的关键码均不足  $\lceil n/2 \rceil$  个

注意，L和R仍必有其一，且恰有  $\lceil n/2 \rceil - 1$  个关键码（不妨以L为例）

◆从L中抽出介于L和R之间的关键码

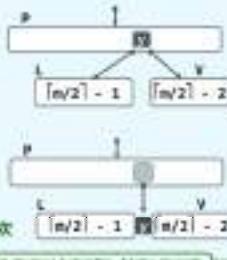
通过y的右孩子，以L的左子树做一个节点  
同时合并此前y的孩子引用

◆如此“拉开”之后，离底部处的下溢得以修复

但可能要做更高级的下溢

此时，大可套用前述，继续替换结合

◆下溢可能持续发生，升降级向上传播；但至多不过  $O(\log n)$  次



## 下溢修复：旋转

◆情况1：向左兄弟相关键码——情况2完全对称

◆if ( $0 < r$ ) { //若v不是p的第一个孩子，则

```
    if ( $\lfloor \text{order} + 1 \rfloor / 2 < \text{ls}->\text{child}.size()$ ) { //若该兄弟足够“胖”，则
        v->key.insert(0, p->key[r-1]); //p抽出一个关键码给v（作为最小关键码）
        p->key[r-1] = ls->key.remove( $\lfloor \text{ls}->\text{key.size}() - 1 \rfloor$ ); //ls的最大关键码插入p
        v->child.insert(0, ls->child.remove( $\lfloor \text{ls}->\text{child.size}() - 1 \rfloor$ ));
        //同时ls的最右孩子过继给v（作为v的最左孩子）
    }
    if ( $v->\text{child}[0] < v->\text{child}[0]-\text{parent} = \text{vs}$ )
        return; //至此，通过本轮已完成当前层级（以及所有层）的下溢处理
}
```

## 实例：底层节点

◆2-3-树

53 97 36 89 41 75 19 34 77 79 51



## 下溢修复：合并

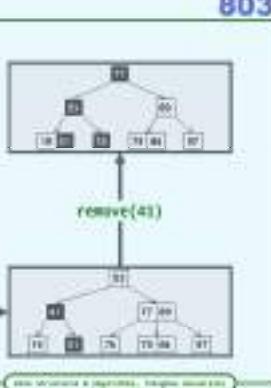
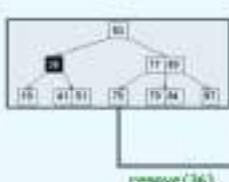
◆if ( $0 < r$ ) { //与左兄弟合并

```
    BTreePosi(T) ls = p->child[r-1]; //左兄弟必存在
    ls->key.insert(0, ls->key.size(), p->key.remove(r-1));
    p->child.remove(r-1); //p的第r-1个关键码接入ls，v不再是p的第r个孩子
    ls->child.insert(0, ls->child.size(), v->child.remove(0));
    if ( $ls->\text{child}[ls->\text{child.size}() - 1]$ ) //若是在右孩子往ls注入ls的最右孩子
        ls->child[ls->child.size() - 1]->parent = ls;
    /* ... TBC ... */
} else /* 与右兄弟合并，完全对称 */;
```

## 实例：非底层节点

◆2-3-树

53 97 36 89 41 75 19 34 77 79 51



## 下溢修复：合并

◆if ( $0 < r$ ) { //与左兄弟合并

```
    /* ... */
    while (!lv->key.empty()) { //v剩余的关键码和孩子，依次转入ls
        ls->key.insert(0, ls->key.size(), v->key.remove(0));
        ls->child.insert(0, ls->child.size(), v->child.remove(0));
        if ( $ls->\text{child}[ls->\text{child.size}() - 1]$ )
            ls->child[ls->child.size() - 1]->parent = ls;
    }
    release(v); //释放v
} else /* 与右兄弟合并，完全对称 */;
```

## 下溢修复

```
#template <typename T> void BTree<T>::solveUnderflow(BTreePosi(T) v) {
    if ( $\lfloor \text{order} + 1 \rfloor / 2 < v->\text{child}.size()$ ) return; //溢出点：v升上来下溢
    BTreePosi(T) p = v->parent; if (!p) { /* 递归基：已到根节点 */
        Rank r = 0; while (p->child[r] == v) r++; //确定p是v的第一个孩子
        if ( $0 < r$ ) { /* 情况1：若v的左兄弟存在，且... */
            if ( $p->\text{child}.size() - 1 < r$ ) { /* 情况2：若v的右兄弟存在，且... */
                if ( $0 < r$ ) { /* 与左兄弟合并 */ } else { /* 与右兄弟合并 */ } //情况3
                solveUnderflow(p); //上升一级，继续分裂——最多四次O(log n)层——典型尾递归
            }
        }
    }
}
```

## 习题解析

◆举个说明，在遍历情况下，一次插入操作会引发  $O(\log n)$  次分裂

在连续的插入操作过程中，发生这种情况的概率有多大？

或直接定义而言，平均每次插入操作平均会引发多少次分裂？

从数据源而言，与下溢修复一样，上溢的矩阵可做链接，也可做分裂

试扩充 `BTree::solveOverflow()` 接口，插入这种策略

这一对称的策略，如何实现？若遇奇数呢？

## k-Tree

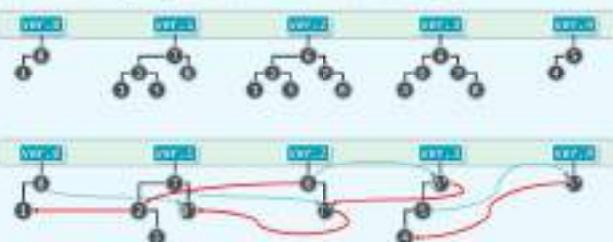
## 从归并分析到联合分析

节点上出现未必独自分析，由 $k$ 个值相的兄弟构成解关键码

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

如此，可将空间使用率从 $50\%$ 提高至 $k / (k+1)$

为此，能附带结构的拓扑内存，相邻版本之间的差异不能超过 $O(1)$



如图，AVL, Splay等BST均不具备这一性质：须另辟蹊径...

## 8. 高级搜索树

## 红黑树

## 一致性

As she looks at the blood on the snow,  
she says to herself, "Oh, how I wish  
that I had a daughter that had skin  
WHITE as snow, lips RED as blood, and  
hair BLACK as ebony".

李佳辉

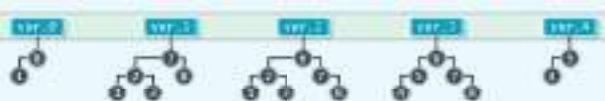
lijianghai@zjhu.edu.cn

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

## Persistent structure：支撑对历史版本的访问

`T.search(0); T.insert(0); T.remove(0); T.key(0)`

能力实现：每个版本独立保存；各版本入口组成一个搜索结构

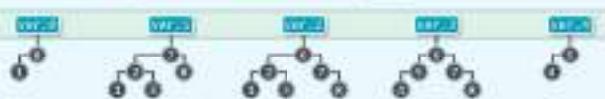


每次操作 $O(\log n + \log m)$ ，累计 $O(mn)$ 时间/空间

// $n = |\text{history}|$

挑战：是否将修改限制在 $O(n + \log m)$ 内？

可以！为此需利用相邻版本之间的相关性...

大量共享，少量更新：每个版本的新数据条目，仅 $O(\log m)$ 

能否进一步提高，比如总存 $O(n + k)$ ，单版本 $O(1)$ ？可以！

1972, R. Bayer

symmetric binary B-tree

1973, L. Guibas & R. Sedgewick

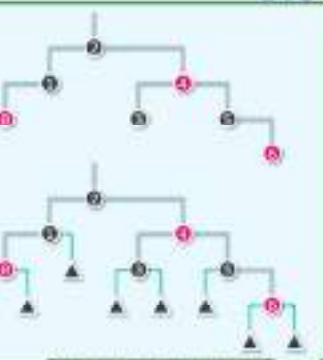
red-black tree

1980, H. Olivie

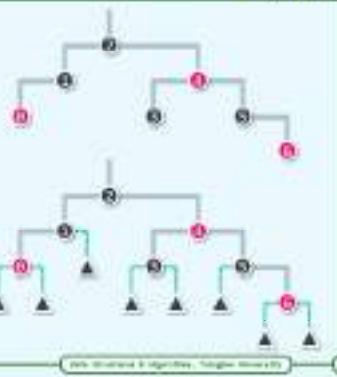
half-balanced binary search tree

由红、黑两类节点组成的BST

统一假设外部节点null，使之成为真二叉树



- 规则**
- 1) 根结点必为黑色
  - 2) 外部节点均为黑色
  - 3) 红色节点：若为红，则只能有黑孩子 //红之子，之父必黑
  - 4) 外部节点数相等：途中黑节点数目相等 //平衡性



- ◆ 节点的颜色，只能显示记录？
- ◆ 以上定义能为平衡，有逻辑解释吗？
- ◆ 如此定义的RBT，也是BST？

## 8. 高级搜索树

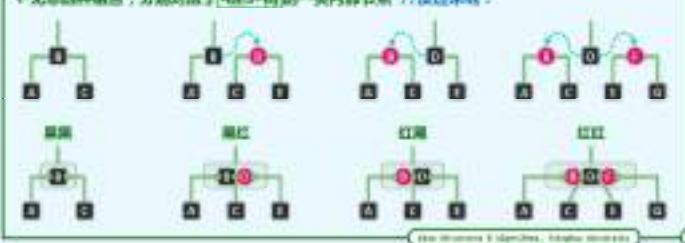
红黑树  
插入黄杏报喜，黄鹂报鸟  
黑白转换，共享同车

样例

Data Structures &amp; Algorithms, Design Patterns

## (2,4)树 == 红黑树

- ◆ 由于每红节点，使之与树（即）父亲等高——于是每根红黑树，都时由至一棵  $(2, 4)$ -树
- ◆ 将黑节点与纯红孩子操作（关键的操作行为）[替换节点]...
- ◆ 无此操作组合，分别对后于4的3-树的一类内部节点 //通过策略：



## 红黑树 == BST

- ◆ 由零阶性，既然B-树是平衡的，红黑树自然也应是
- ◆ 定理：若有n个内部节点的红黑树T，高度  $h = \lceil \log_2(n+1) \rceil$  //最严证明...
- ◆ 若  $T$  高度为  $H$ ，红/黑高度为  $R/H$ 
  - 则  $h = R + H \leq 2H$
- ◆ 若  $T$  所对应的  $n$ -树为  $T_n$ ，  
则  $T_n$  的高度
- ◆  $T_n$  的每个节点，都含其自身的一个黑节点
- ◆ 于是  $L(H) \leq \log_2(n+1) \leq \log_2(n+1)$



## 算法

- ◆ 按x的常模操作，插入关键码x //若x->insert(x)必为末端节点
- ◆ 不然没x的父亲  $p = x->parent$  存在 //否则，即平凡的首次插入
- ◆ 将x染红（除非已墨化）// $x->color = isLeaf(x) ? \text{BLACK} : \text{RED}$
- ◆ 纯红(double-red) // $p->color == x->color == \text{RED}$
- ◆ 考虑：x的祖父  $g = p->parent$  // $g = \text{NULL}$  且  $g->color == \text{RED}$
- ◆ x的兄弟  $u = g->lc ? g->rc : g->lc$  // $u$ 的祖父
- ◆ 红白的颜色，分两种情况处理...

Data Structures &amp; Algorithms, Design Patterns

## 实现

```

* template <typename T> class BinNodePosi(T) : public BinNode<T>::BinNodePosi(T) {
    // 确认目标节点不存在 (前置对_tvr的设置)
    BinNodePosi(T) & x = search( * ); if ( !x ) return x;
    // 前置红节点x，以_xtvr为父，用高斯
    x = new BinNodePosi( e, _xtvr, NULL, NULL, -1 ); _size++;
    // 如有必要，需做双红修正
    ResolveDoubleRed( x );
    // 返回插入的节点
    return x ? x : _root->parent;
    // 无论树中是否存在e，返回时总有x->data == e

```

## RedBlack

```

* template <typename T> class RedBlack : public BST<T> { //红黑树
public:
    //BST::search()等其他接口可直接沿用
    BinNodePosi(T) insert( const T & e ); //插入 (重写)
    bool remove( const T & e ); //删除 (重写)
protected:
    void solveDoubleRed( BinNodePosi(T) * x ); //双红修正
    void solveDoubleBlack( BinNodePosi(T) * x ); //双黑修正
    int updateHeight( BinNodePosi(T) * x );
}
* template <typename T> int RedBlack<T>::updateHeight( BinNodePosi(T) * x ) {
    x->height = max( stature( x->lc ), stature( x->rc ) );
    if ( !isBlack( x ) ) x->height++; return x->height; //只计算节点
}

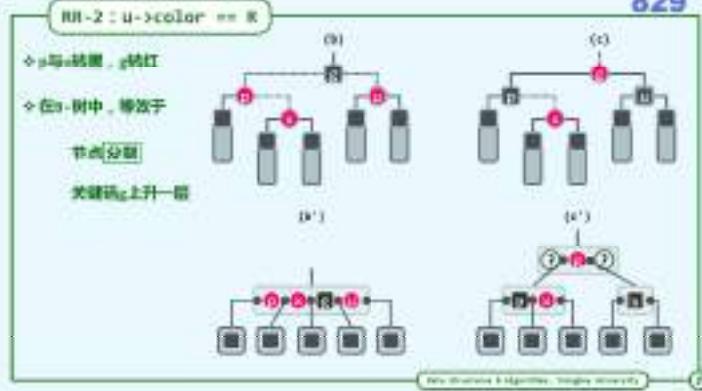
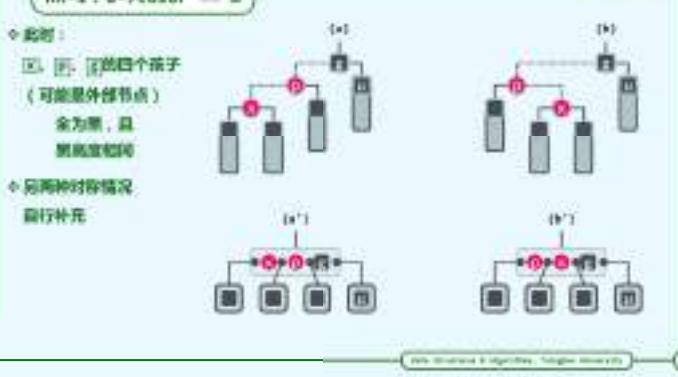
```

## 双红修正

```

* template <typename T> void RedBlack<T>::solveDoubleRed( BinNodePosi(T) * x ) {
    if ( !IsRoot( *x ) ) { //若已 (溢出) 将至树根，将将其转回，整树高度也随之递增
        _root->color = RED_BLACK; _root->height++; return; } //否则...
    BinNodePosi(T) p = x->parent; //考虑x的父亲p (必存在)
    if ( !IsBlack( p ) ) return; //若p为红，则可停止调整；否则
    BinNodePosi(T) g = p->parent; //x祖父必存在，且必须
    BinNodePosi(T) u = uncle( *x ); //以下根据父p的颜色分别处理
    if ( !isBlack( u ) ) { //... (为黑 (即NULL) ... )
    }
    else { //... (为红 ... )
    }

```

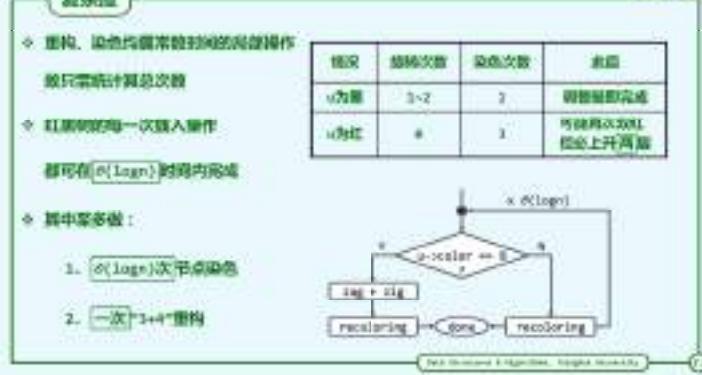
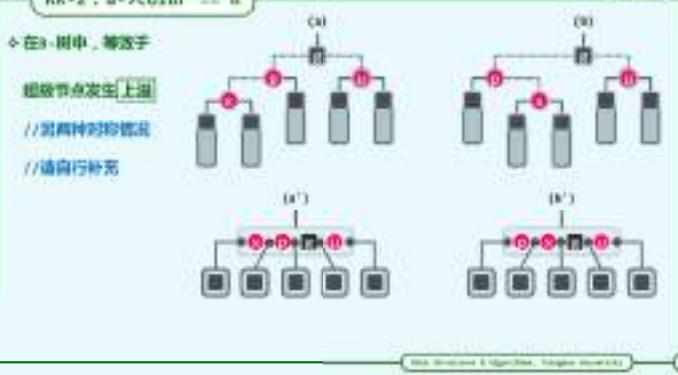


RR-1 : 实现

```
+ template <typename T> void RedBlack<T>::solveDoubleRed( BinNodePos1(T) * > {
    /* ... */
    if ( ! IsBlack( u ) ) { //u为根或NULL
        // 若s与p同黑，两由红转黑，>保持红；否则，>由红转黑，p保持红
        if ( IsBlack( *s ) == IsBlack( *p ) ) p->color = BB_BLACK;
        else
            s->color = BB_BLACK;
        g->color = BB_NOD; //g必定由跟转红
        BinNodePos1(T) g = e->parent; //great-grand parent
        BinNodePos1(T) r = ErrParentPos1( *g ) = errdata&1( * );
        r->parent = g; //调整之后的父子结，同与跟的祖父链接
    } else { /* ... u为红 ... */
    }
}
```

RR-2 : 实现

```
+ template <typename T> void RedBlack<T>::solveDoubleRed( BinNodePos1(T) * > {
    /* ... */
    if ( ! IsBlack( u ) ) { /* ... u为黑（或NULL）... */
    } else { /* u为红
        p->color = BB_BLACK; p->height++; //p由红转黑，提高
        g->color = BB_BLACK; g->height++; //g由红转黑，提高
        if ( ! IsRoot( *g ) ) g->color = BB_RED; //g若非根则转红
        solveDoubleRed( g ); //继续调整g（类似于尾递归，可优化）
    }
}
```



## 8. 高级搜索树

红黑树  
图解

空白以为黑色，斜上以为红色

对称解  
http://tiny.cc/meyarw

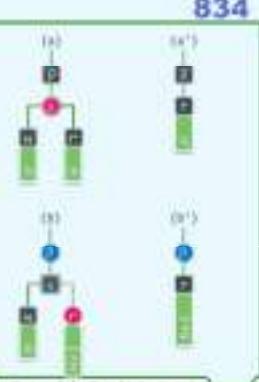
## 算法

- 首先将树转成平衡树法，执行
 

```
return removeR(r, _hot);
```
- ④为孩子②的前替，另一孩子记作③
 

首次必为NULL，但可能在随后的调整过程中逐渐上升——如何统一？
- 更清晰、更直接的等效理解：
 

因为一棵左三右高的子红黑树，且颜色一致要删除
- 操作①和②必然满足：操作③不用再用
- 在原树中，平衡已满足...
- 若二者之一为红，则②和④不相满足 //删除操作完成



834

## 实现

```
// 父节点（及祖先）依然平衡，既无需要调整
if (!BlackHeightBalanced(r->parent) || !r->parent) return true;
// 此处，必失衡
// 若替代节点r为红，则只需简单地翻转颜色
if (IsRed(r)) { r->color = RS_BLACK; r->height++; return true; }
// 而此，r以及被替换的x均为黑色
solveDoubleBlack(r); //双黑调整（入口处必有 r == NULL）
return true;
}
```

837

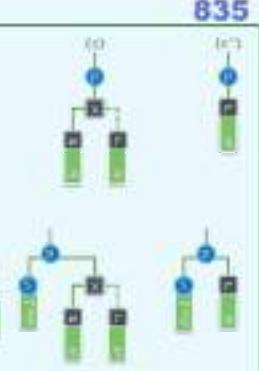
## 双黑修正

```
#template <typename T> void RedBlack::solveDoubleBlack( BinNodePosi(T)* r ) {
  BinNodePosi(T)* p = r->parent; if (!p) return; //r的父亲
  BinNodePosi(T)* s = (r == p->lc) ? p->rc : p->lc; //r的兄弟
  if (IsBlack(s)) { //兄弟s为黑
    BinNodePosi(T)* t = NULL; //s的红孩子（若无，右孩子插红，左者优先；否则时为NULL）
    if (IsRed(s->rc)) t = s->rc;
    if (IsRed(s->lc)) t = s->lc;
    if (t) { /* ... 节点有红孩子 [RR-1] ... */ }
    else { /* ... 节点无红孩子 [RR-2]或[RR-3] ... */ }
  } else { /* ... 兄弟s为红 [RR-2] ... */ }
}
```

838

## 算法

- 既然与已均衡的binblack  
则不然...
- 摘除④并代之以平衡  
全局最深度不再统一 //高效率...
- 原B-树中④所带节点下退
- 在新树中，考虑
  - ④的父辈 p = r->parent //新平衡树中r的父辈
  - ④的兄弟 s = (r == p->lc) ? p->rc : p->lc
- 以下四种情况处理...

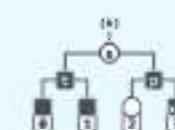


835

## BB-1: ④为黑，且至少有一个红孩子⑤

④为平衡；⑤, ⑥, ⑦为平衡；④, ⑤, ⑥, ⑦为平衡

④保持黑；⑤和⑥换色；⑦继承④的颜色



839

如此，红黑树性质在全局深以恢复——递归完成！ //zig-zag等类似

在对应的B-树中，以上操作等效于...

836

实现

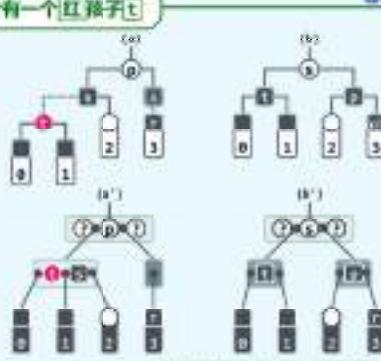
```
#template <typename T> bool RedBlack::remove( const T& e ) {
  BinNodePosi(T)* x = search( e ); if (!x) return false; //查找失败
  BinNodePosi(T)* r = removeR( x, _hot ); //删除_red的某孩子，r指向间接替者
  if (!r || --_size) return true; //若替季后为空时，可直接返回
  if (!r->parent) { //若被替时的祖辈，即
    _root->color = RS_BLACK; //将其置黑，并
    updateHeight( _root ); //更新（全树）高度
    return true;
  } //至此，因x（即r）的非根
```

## BB-1: ④为黑，且至少有一个红孩子⑤

通过关键码的替换

消除数据节点的下退

④为平衡  
可同时存在  
颜色不变



840

## BB-1：实现

```

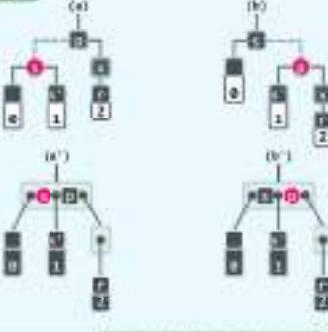
if ( IsBlack( *t ) ) { //兄弟s为黑
    /* ... */
    if ( *t ) { /* s有红孩子 : BB-1 */
        s->color = p->color; //替换了颜色，并对s、父亲、祖父
        fixNodePosi( t ) b = EncapsParent( *t ) = rotateLeft( *t ); //旋转
        if ( maxChild( *b ) ) //新子树左子为黑后
            ( b->lc->color = BB_BLACK; updateHeight( b->lc ) );
        if ( maxChild( *b ) ) //新子树右子为黑后
            ( b->rc->color = BB_BLACK; updateHeight( b->rc ) );
        b->color = oldColor; updateHeight( b ); //恢复原有颜色的黑色
    } else { /* ... s无红孩子 : BB-2或BB-3 ... */ }
} else { /* ... 兄弟s为红 : BB-1 ... */ }

```

Data Structures &amp; Algorithms, 10th Edition 2013

BB-3： $s$ 为红，且两个孩子均为黑； $p$ 为红

↑ zig(s)或zig(p)：红色替换，黑变转红  
↑ 调度依然正常，但...  
↑ 有了一个额外的兄弟s  
故转化为前述情况，而且...  
↑ 父亲s已转红，接下来  
谁不会是黑色BB-2  
而只能是BB-1或BB-2  
↑ 于是，再经一轮调整之后  
红黑树逻辑必然是全局正确



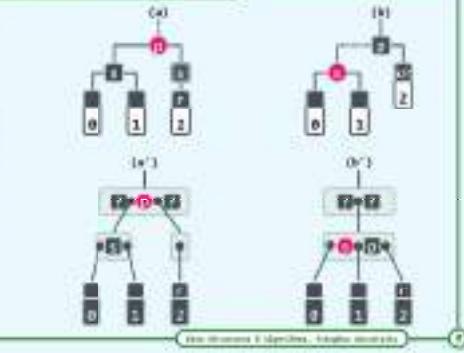
Data Structures &amp; Algorithms, 10th Edition 2013

BB-2R： $s$ 为黑，且两个孩子均为黑； $p$ 为红

↑ 保持原s-树不变

↑ 在对应的p-树中，等效于  
下溢节点与兄弟合并

↑ 红黑树性质在全局树以依赖

↑ 关键关键结点，上层节点  
是否倒退下溢？不会！↑ 合并之前，在p之左或右侧  
还应有（同号）关键结点  
必为黑色  
并且仅剩一个

Data Structures &amp; Algorithms, 10th Edition 2013

## BB-3：实现

```

if ( IsBlack( *t ) ) { //兄弟s为黑
    if ( *t ) { /* s有红孩子 : BB-1 ... */ }
    else { /* ... s无红孩子 : BB-2或BB-3 ... */ }
} else { /* ... 兄弟s为红 : BB-1 */
    s->color = BB_BLACK; p->color = BB_RED; //s转黑，p转红
    fixNodePosi( t ) t = ISLChild( *t ) ? s->lc : s->rc; //s与其父s-树
    lhot = p; EncapsParent( *p ) = rotateLeft( *t ); //对s及其父亲，祖父做平衡调整
    solveDoubleBlack( p ); //修正修正——这时p已转红，故前结只能是BB-1或BB-2
}

```

Data Structures &amp; Algorithms, 10th Edition 2013

BB-2B： $s$ 为黑，且两个孩子均为黑； $p$ 为黑↑ s转红； $p$ 与 $s$ 保持不变

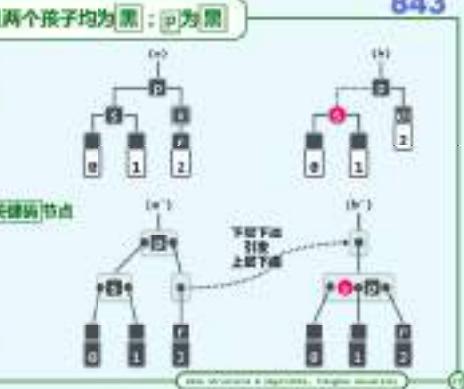
↑ 红黑树性质在全局树以依赖

↑ 在对应的p-树中，等效于  
下溢节点与兄弟合并

↑ 合并之前，p结点对来自于单关键结点

↑ 关键关键结点，上层节点  
上溢节点必倒退下溢

↑ 好可在可继续分情况处理

高度递增，最多 $O(\log n)$ 步

Data Structures &amp; Algorithms, 10th Edition 2013

## 复杂度

↑ 红黑树使用一趟旋操作

即可在 $O(\log n)$ 时间内完成

↑ 同中，更多样

1.  $\delta(\log n)$ 次遍历操作

2. 一次“3+6”重构

3. 一次旋转



Data Structures &amp; Algorithms, 10th Edition 2013

```

if ( IsBlack( *t ) ) { //兄弟s为黑
    /* ... */
    if ( *t ) { /* ... s有红孩子 : BB-1 ... */ }
    else { /* ... s无红孩子 */
        s->color = BB_RED; s->height--; //s转红
        if ( IsRed( p ) ) //BB-2B: p转黑，但高度不变
            ( p->color = BB_BLACK );
        else //BB-2B: p保持黑，但高度下降；进行修正
            ( p->height--; solveDoubleBlack( p ) );
    }
} else { /* ... 兄弟s为红 : BB-1 ... */ }

```

Data Structures &amp; Algorithms, 10th Edition 2013

## Advanced Balanced Search Tree

Range Query

1-Dimensional Range Query

译者注

dengtinghua@sohu.com

## Query Types

849

Let  $P = \{p_1, \dots, p_n\}$  be a set of  $n$  points on the  $x$ -axis



For any given interval  $I = (x_1, x_2]$  //left open & right closed

- Counting**: how many points of  $P$  lies in the interval?

- Reporting**: enumerate all points in  $I \cap P$  (if not empty)

## Lower Bound?

853

In the worst case,

the interval contains up to  $\Theta(n)$  points,

which need  $\Theta(n)$  time to enumerate



However, how if we

- ignore the time for **enumerating** and
- count only the **searching** time?

## Online Query

850

or  $I$  is **fixed** while  $x$  is **randomly and repeatedly** given



How to **preprocess**  $P$  into a certain data structure s.t.

the queries can be answered efficiently!

## Geometric Range Search

Range Query

Binary Search

解説

douglashuang@zjhu.edu.cn

851

## Advanced Balanced Search Tree

Range Query

Brute Force

When all else fails, try brute-force.

- Anonymous

#18

douglashuang@zjhu.edu.cn

## Preprocessing

855

Sort all points into a sorted vector and

add an extra sentinel point  $p[0] = -\infty$

## Brute-Force

852

For each point  $p \in P$ , test if  $p \in (x_1, x_2]$

Thus each query can be answered in  $\Theta(n)$  time



Can we do it faster?

It seems we can't, for ...

## Binary Search

856

For any interval  $I = (x_1, x_2]$

- find  $i = \text{search}(x_1) = \min\{k \mid p[k] \leq x_1\} // O(\log n)$
- Traverse the vector **backward** from  $p[i]$  and report each point //  $O(r)$
- until escaping from  $I$  at point  $p[s]$
- return  $[r = i - s]$  // output size

//  $i$  is the index of the first point in  $I$ .

**Advanced Balanced Search Tree**  
**Range Query**  
**Output Sensitivity**

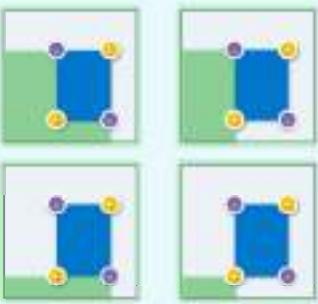
# [E]

dengtingguo@cs.cuhk.edu.hk

**Preprocessing**For each point  $(x, y)$ , let

$$n(x, y) = |\{(b, x) = (0, y) \cap P\}|$$

This requires

 $\Theta(n^2)$  time/space

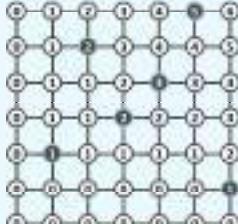
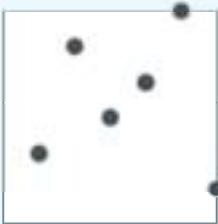
Data Structures &amp; Algorithms, Chapter 10

- + A **counting** query can be answered in  $\Theta(r + \log n)$  time
- $p(x)$  can also be found by binary search in  $\Theta(\log n)$  time
- Hence for **counting** query,  $\Theta(\log n)$  time is enough // independent to  $r$



- Can this simple strategy be extended to **planar** range query?
- TRYONE, unfortunately, no!

Data Structures &amp; Algorithms, Chapter 10

**Domination**A point  $(x, y)$  is called to be **dominated** by point  $(x', y')$  if  
 $x' \leq x$  and  $y' \leq y$ 

Data Structures &amp; Algorithms, Chapter 10

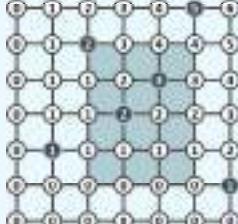
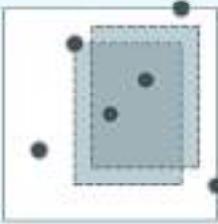
**Advanced Balanced Search Tree**  
**Range Query**  
- Planar Range Query

# [E]

dengtingguo@cs.cuhk.edu.hk

**Inclusion-Exclusion Principle**For any rectangular range  $R = [x_1, x_2] \times [y_1, y_2]$ , we have

$$|R \cap P| = n(X_1, Y_1) + n(X_2, Y_2) - n(X_1, Y_2) - n(X_2, Y_1)$$



Data Structures &amp; Algorithms, Chapter 10

Let  $P = \{p_1, \dots, p_n\}$  be a planar set.Given  $R = [x_1, x_2] \times [y_1, y_2]$ 

- **Counting** :  $|R \cap P| = ?$
- **Reporting** :  $P \cap R \rightarrow ?$

Binary search

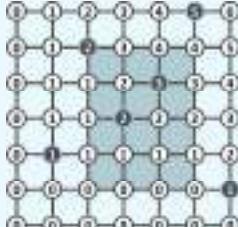
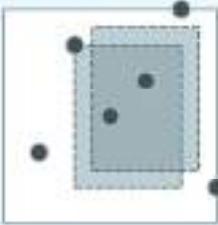
doesn't help this kind of query

You might consider to expand the counting method using

the **Inclusion-Exclusion Principle****How If ...**

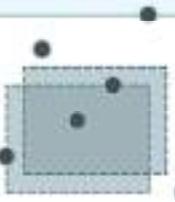
the query range is closed?

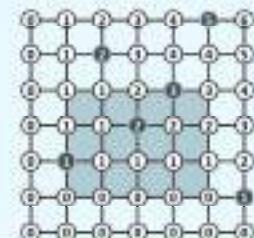
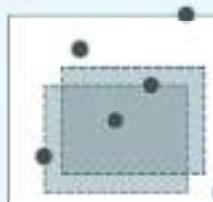
there exist points sharing a same x or y coordinate?



Data Structures &amp; Algorithms, Chapter 10

**Performance**

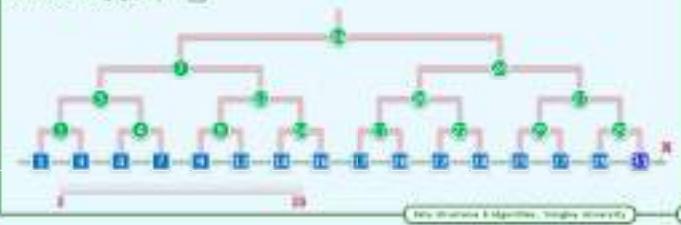
- Each query needs only  $\Theta(\log n)$  time
- Uses  $\Theta(n^2)$  storage and even more for higher dimensions
- To figure out a better solution, let's go back to the  case



Data Structures &amp; Algorithms: Chapter 10

**Lowest Common Ancestor**

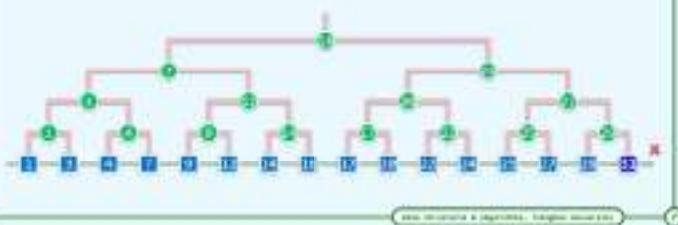
- Consider, as an example, the query for  $(1, 14)$
- Binary search:
  - $\text{search}(2).succ() = 3.succ() = [3]$
  - $\text{search}(14) = [14]$

 $\Rightarrow \text{lca} = \text{lca}(3, 14) = [3]$ **Advanced Balanced Search Tree**

**1d-Tree**  
- Structure

AB (1.4)  
abg@tonghu.edu.cn

- $\forall v, v.key = \max\{ u.key \mid u \in \text{L-Tree}(v) \} = v.\text{pred}().key$
- $\forall v \in \text{L-Tree}(v), v(x) \sqsubseteq v(y)$
- $\text{search}(x) : \text{Returns the maximum key not greater than } x$

**Advanced Balanced Search Tree**

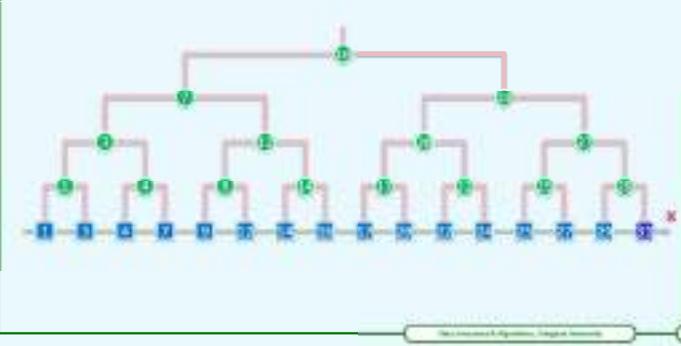
**1d-Tree**  
Query

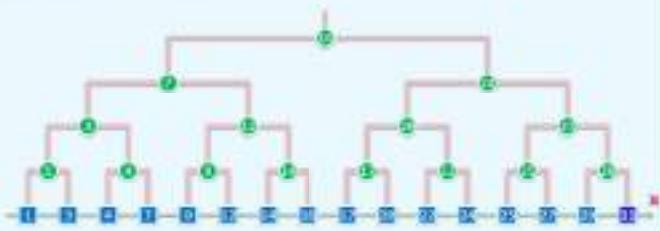
AB (1.4)  
abg@tonghu.edu.cn

**Advanced Balanced Search Tree**

**1d-Tree**  
Complexity

AB (1.4)  
abg@tonghu.edu.cn





♦ To make it work,

- each partition should be done

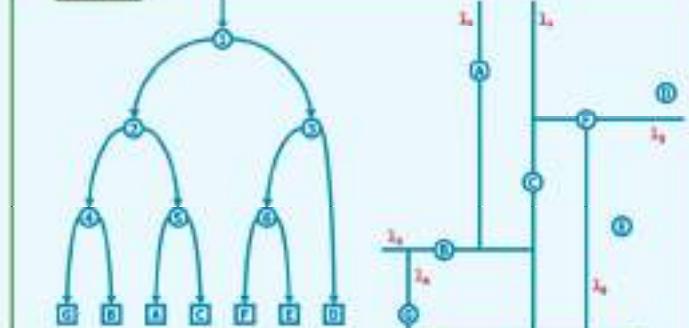
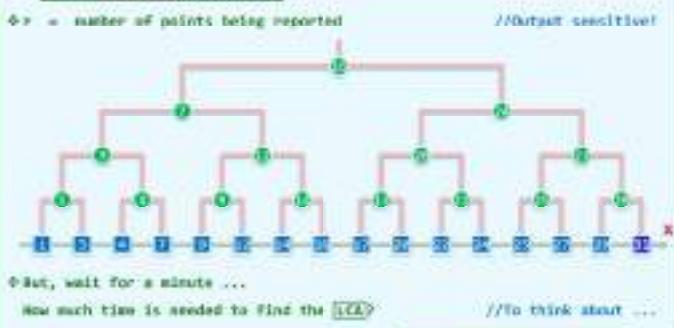
as evenly as possible (at median)

- each region is defined to be open/closed

on the left-lower/right-upper sides

♦ Degeneracy assumption:

no two input points lie on a same vertical/horizontal line



## Advanced Balanced Search Tree

2d-Tree  
Structure

凡局字数，如得分，即平分一半为上卦，一半为下卦。  
卦，如字数不均，如少一卦为上卦，取天机清之义，  
或多一卦为下卦，取地幽之义。

卦 体 球

douyitongguo@sohu.com

## Advanced Balanced Search Tree

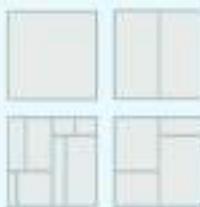
2d-Tree  
Construction

God found himself by creating.

douyitongguo@sohu.com

♦ To extend the BST method to planar QRS, we

- divide the plane recursively and
- partition the regions into a kd-tree



♦ Start with a single region (the entire plane).

Partition the region vertically/horizontally at each even/odd level.

Partition the sub-regions recursively.

```

// construct a 2d-(sub)tree for point (sub)set P at depth d
IF ( P == {p} ) return CreateLeaf(p) //base
root = CreateKdNode()
root->splitDirection = Even(d) ? VERTICAL : HORIZONTAL
root->splitLine = FindMedian( P, root->splitDirection, P ) //O(n)
( P1, P2 ) = Divide( P, root->splitDirection, root->splitLine ) //DNC
root->lChild= buildKdTree( P1, d + 1 ) //recuse
root->rChild= buildKdTree( P2, d + 1 ) //recuse
return( root )

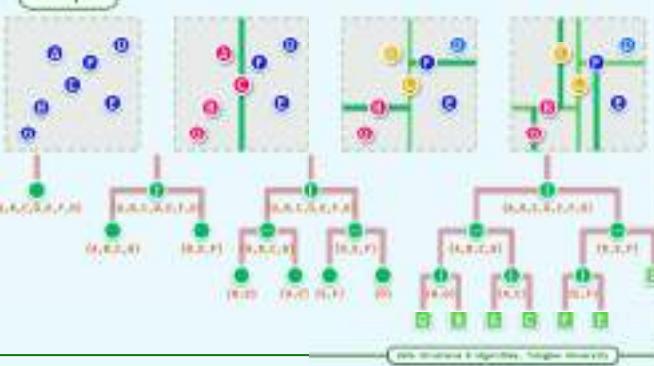
```

## Advanced Balanced Search Tree

2d-Tree  
Query

胡佳群

dongtinghuang@zjhu.edu.cn



## Advanced Balanced Search Tree

2d-Tree

Canonical Subsets

当小虫爬着爬到末边，只觉得上方布上打满了  
几千枚弹花针，几乎快碎裂，几乎快碎裂成一幅完整无缺的  
大地图，难得的是几千片碎皮拼在一起，既没多出  
一片，也沒少了一片。

胡佳群

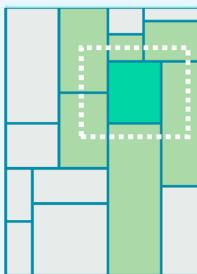
dongtinghuang@zjhu.edu.cn

kdSearch( v, R )

```

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

```



- Each node corresponds to
  - a rectangular sub-region of the plane, as well as
  - the subset of points contained in the sub-region
- Each of these subsets is called a canonical set
- For each internal node X with children L and R,
  $\text{region}(X) = \text{region}(L) \cup \text{region}(R)$
- Sub-regions of nodes at a same depth
  - never intersect with each other, and
  - their union covers the entire plane
- We will see soon that
  - each 2D QRS can be answered by
  - the union of a number of CS's

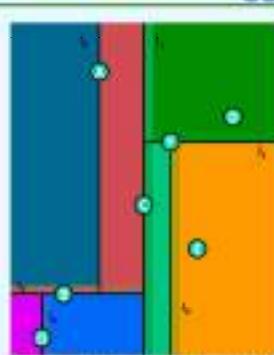
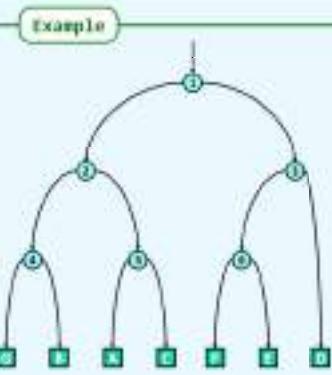
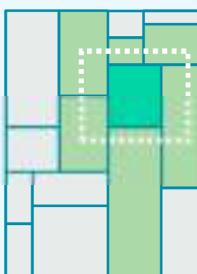


kdSearch( v, R )

```

if ( region( v->lc ) ⊂ R )
    reportSubtree( v->lc );
else if ( intersect( region( v->lc ), R ) )
    kdSearch( v->lc, R );

```

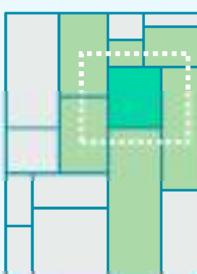


kdSearch( v, R )

```

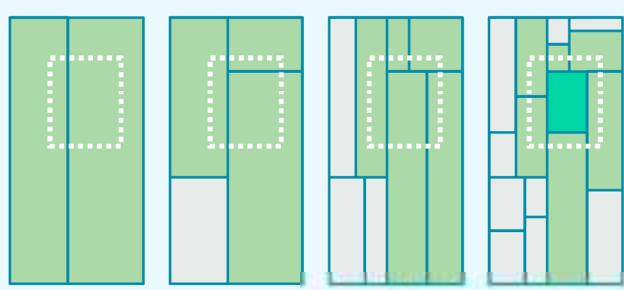
if ( region( v->rc ) ⊂ R )
    reportSubtree( v->rc );
else if ( intersect( region( v->rc ), R ) )
    kdSearch( v->rc, R );

```



Example

889



Advanced Balanced Search Tree

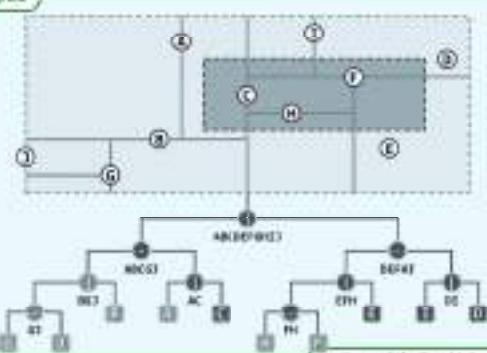
2d-Tree  
Complexity

解説

http://algotree.csie.ntu.edu.tw

Example

890



Data Structures &amp; Algorithms - Integer Interview

Preprocessing

- ❖  $T(n)$
- =  $2^k \pi(n/2) + \theta(n)$
- =  $\theta(n \log n)$



Data Structures &amp; Algorithms - Integer Interview

894

Advanced Balanced Search Tree

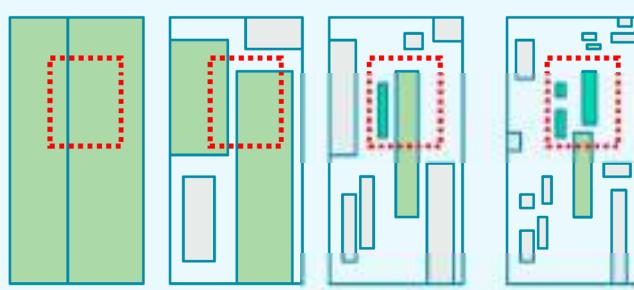
2d-Tree  
- Optimization

解説

http://algotree.csie.ntu.edu.tw

Bounding Box

892



Data Structures &amp; Algorithms - Integer Interview

Storage

- ❖ The tree has a height of  $\theta(\log n)$
- ❖  $\pi = 3$
- + 2
- + 4
- + ...
- =  $\theta(2^{\log n})$
- =  $\theta(n)$



Data Structures &amp; Algorithms - Integer Interview

895

Query Time

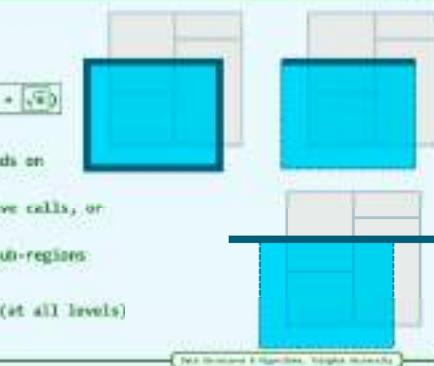
Case:

Report + Search =  $\Theta(r + \sqrt{n})$ 

❖ The searching time depends on

- the number of recursive cells, or
- $\Theta(r)$ , the number of sub-regions

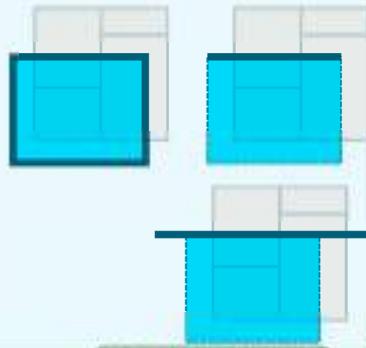
[Intersecting with] R (at all levels)



Data Structures &amp; Algorithms - Integer Interview

Query Time

- after each node, no more than 2 of its 4 grandchildren will recurse



Recurrence

$$\begin{aligned} & Q(1) = \Theta(1) \\ & Q(n) = 2 + 2^2 Q(n/4) \end{aligned}$$

Solve to  $Q(n) = \Theta(\sqrt{n})$

Advanced Balanced Search Tree  
2d-Tree  
Quadtree

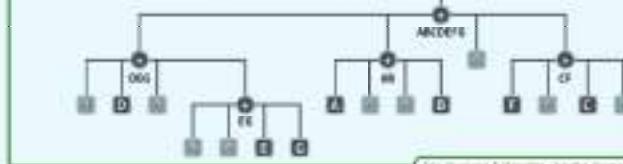
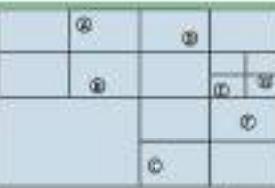
见注释

http://tiny.cc/meyarw4

Advanced Balanced Search Tree  
2d-Tree  
Beyond 2D

见注释  
http://tiny.cc/meyarw4

Quadtree



See Advanced Data Structures, Chapter 10 notes 10.1

kd-Tree

Can 2d-tree be extended to kd-tree and help higher dimensional GRS?

If yes, how efficiently can it help?

A kd-tree in k-dimensional space

is constructed by:

recursively divide  $\mathbb{R}^k$

along the  $x^1, x^2, \dots, x^k$  dimensions

Advanced Balanced Search Tree

Multi-level Search Tree  
x-Query + y-Query

见注释

http://tiny.cc/meyarw4

$\Theta(r + n^{1 - 1/k})$

An orthogonal range query on a set of n points is  $\Theta(n)$

- can be answered in  $\Theta(r + n^{1 - 1/k})$  time,
- using a kd-tree of size  $\Theta(n)$ , which
- can be constructed in  $\Theta(n \log n)$  time

2D Range Query = x-Query  $\sqcap$  y-Query

Is there any structure

which answers range query

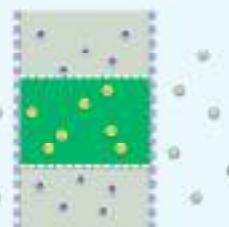
faster than kd-tree?

An  $m \times n$  orthogonal range query

can be answered by

the intersection of

= 1-D queries



See Advanced Data Structures, Chapter 10 notes 10.1

Q

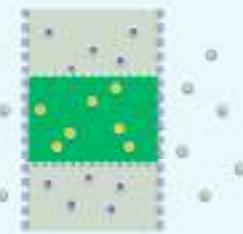
For example ...

• A 2D range query

can be divided into

two 1D range queries:

- find all points
- in  $[x_1, x_2]$  and then
- find from these candidates
- those lying in  $[y_1, y_2]$



Data Structures & Algorithms, Lecture Slides 01

This idea can be implemented

in the following manner:

- build a 1D RBT (called  $\langle x \rangle$ -tree)
- for the first range query ( $\langle x \rangle$ -query);
- for each node  $v$  in the  $\langle x \rangle$ -tree,
- build a  $y$ -coordinate RBT (called  $\langle y \rangle$ -tree),
- containing the canonical subset associate with  $v$



Data Structures & Algorithms, Lecture Slides 01

### Advanced Balanced Search Tree

#### Multi-Level Search Tree

##### Worst Case

AB (1/4)

ding@tonghu.edu.cn

### Tree Of Trees

Hence we have built

an  $\langle x \rangle$ -tree of (a number of)  $\langle y \rangle$ -trees,

which is called

a multi-level search tree

? How to answer range queries

with such an MLST?



Data Structures & Algorithms, Lecture Slides 01

Using kd-trees,

we need  $O(1 + \sqrt{n})$  time

but here ...

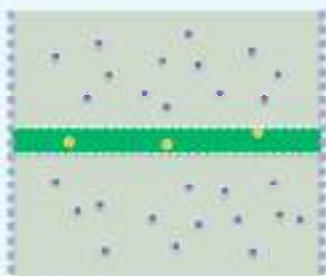
The  $x$ -query returns

(almost) all points whereas

the  $y$ -query projects

(almost) all

We spent  $O(n)$  time before getting  $r = 8$  points



Data Structures & Algorithms, Lecture Slides 01

### Advanced Balanced Search Tree

#### Multi-level Search Tree

##### Query

AB (1/4)

ding@tonghu.edu.cn

### Geometric Range Search

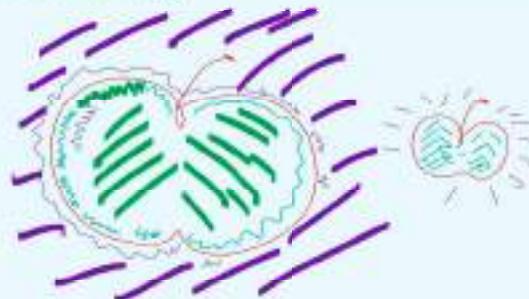
#### Multi-Level Search Tree

##### $x$ -Query \* $y$ -Queries

AB (1/4)

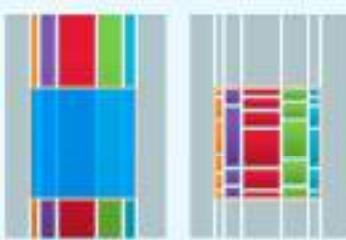
ding@tonghu.edu.cn

### Painters' Strategy



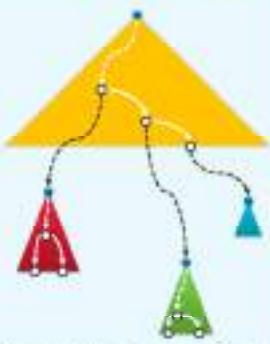
Data Structures & Algorithms, Lecture Slides 01

⇒ Query Time =  $\Theta(r + \log n) = \Theta(r + \log n)$



Data Structures & Algorithms, Chapter 10 (page 913)

- ⇒ A 2-level search tree
- for  $n$  points in the plane
- can be built
- in  $\Theta(n \log n)$  time

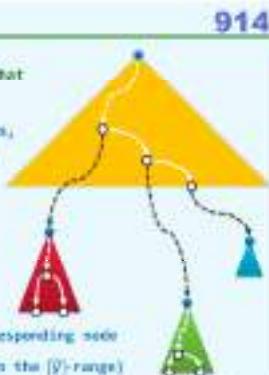


Data Structures & Algorithms, Chapter 10 (page 917)

### Algorithm

- Determine the canonical subsets of points that satisfy the first query

```
// there will be  $\Theta(\log n)$  such canonical sets,
// each of which is just represented as
// a node in the  $\mathbb{Y}$ -tree
2. Find out from each canonical subset
    which points lie within the  $\mathbb{Y}$ -range
    // To do this,
    // for each canonical subset,
    // we access the  $\mathbb{Y}$ -tree for the corresponding node
    // this will be again a  $\mathbb{Y}$ -range search (on the  $\mathbb{Y}$ -range)
```



Data Structures & Algorithms, Chapter 10 (page 914)

⇒ Claim:

- A 2-level search tree
- For  $n$  points in the plane
- answers each planar range query
- in  $\Theta(r + \log n)$  time



Data Structures & Algorithms, Chapter 10 (page 918)

### Advanced Balanced Search Tree

#### Multi-level Search Tree Complexity

$\Theta(\log n)$

<http://tiny.cc/meyarw>

### Storage

- Each input point is stored in  $\Theta(\log n)$   $\mathbb{Y}$ -trees
- A 2-level search tree for  $n$  points in the plane needs  $\Theta(n \log n)$  space



Data Structures & Algorithms, Chapter 10 (page 916)

- Let  $S$  be a set of  $n$  points in  $\mathbb{R}^d$ ,  $d \geq 2$

- A  $d$ -level tree for  $S$  uses  $\Theta(d n \log^d n)$  storage
- such a tree can be constructed in  $\Theta(d n \log^d n)$  time
- each rectangular range query of  $S$  can be answered in  $\Theta(r + \log n)$  time, where  $r$  is the number of reported points



Data Structures & Algorithms, Chapter 10 (page 920)

## Advanced Balanced Search Tree

Range Tree  
Y-Lists

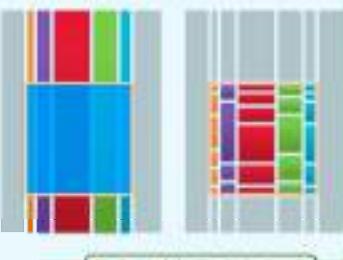
At 1:45

dmitriengithubio

observe further that, for each query

- we need to **repeatedly** search **different** lists/y-trees.
- but always with the **same** key

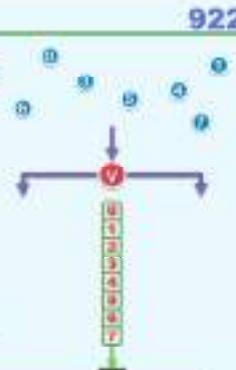
However, such an essential fact is not used yet



Data Structures &amp; Algorithms: Range Queries 1

## BBST&lt;BBST&lt;T&gt;&gt;

while we descend the search in the **x-tree**,  
for each node, we need to  
search the corresponding **y-tree**  
⇒ it is this **combination** that leads to  
the squaring of the logarithms  
⇒ if each **y-tree** can be searched in  $\Theta(1)$  time,  
the squared **log(n)** factor will be eliminated!



## Advanced Balanced Search Tree

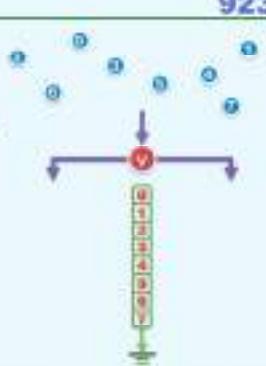
Range Tree  
Idea

At 1:45

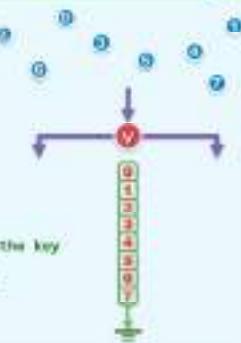
dmitriengithubio

## BBST&lt;Vector&lt;T&gt;&gt;

for easier visualization,  
let's regard each **y-tree**, equivalently,  
as a sorted array (called **y-list**)  
actually, this is also  
a more efficient way  
to implement all **y-trees**.



so the idea for an improvement is that  
we merge all the different lists  
into a **single massive** list  
thus, to answer a planar query, we can  
do a global search in this list  
in  $\Theta(\log n)$  time, and then  
use the information about the location of the key  
to answer each of the remaining **y-queries**  
in  $\Theta(1)$  time



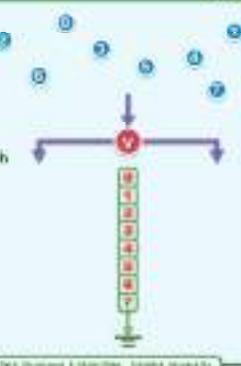
## Advanced Balanced Search Tree

Range Tree  
Coherence

At 1:45

dmitriengithubio

In our case, the massive list on which  
we will do one search is  
the entire set of points,  
sorted by their **y**-coordinates  
we will do one expensive ( $\Theta(\log n)$  time) search  
on the **y-list** for the root  
// after this, however, we claim that ...  
while descending the **x-tree**, we can  
keep track of the position of **y-range**  
in each auxiliary list  
in **constant** time



Data Structures &amp; Algorithms: Range Queries 1

## Advanced Balanced Search Tree

Range Tree  
Fractional Cascading

930

dmitriy@mit.edu.cs

## Merge/Split

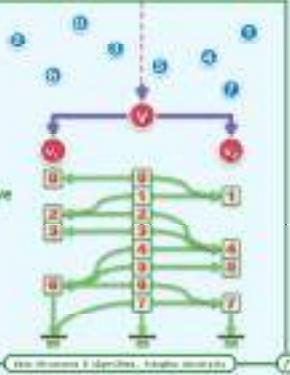
Given  $v$  be an internal node in the  $\mathbb{B}$ -tree with  $v_L/v_R$  its left/right child resp.

Let  $A_v$  be the  $\mathbb{B}$ -list for  $v$  and

$A_L/A_R$  be the  $\mathbb{B}$ -lists for its children

Assuming no duplicate  $y$ -coordinates, we have

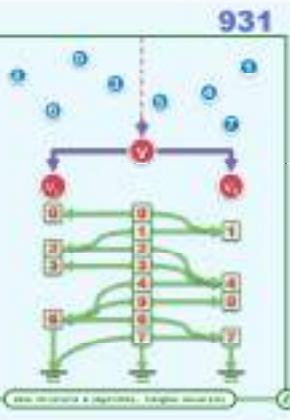
- $A_v$  is the disjoint union of  $A_L$  and  $A_R$ , and hence
- $A_v$  can be obtained by merging  $A_L$  and  $A_R$  (in linear time)



## Structure

For each item in  $A_v$ , we store two pointers to the item of  $\text{equal or larger}$  value in  $A_L$  and  $A_R$  resp.

When there is no such item, the pointer is NULL.



931

## Complexity

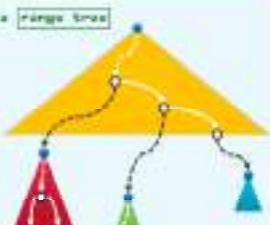
An MBLT with fractional cascading is called a **range tree**.

At the root of the main tree,

- we need to perform a binary search with all the  $y$ -values
- to determine which points lie within this interval, and

- it requires  $\mathcal{O}(\log n)$  time

For all subsequent levels, once we know where the  $y$ -interval falls w.r.t. to the  $x$ -axis here, we can drop down to the next level in  $\mathcal{O}(1)$  time



934

## Complexity

Thus, as with fractional cascading,

the time for cascading searches is  $\mathcal{O}(2 \log n)$ ,

rather than  $\mathcal{O}(\log^2 n)$ .

Gives a set of  $n$  points in the plane, orthogonal range queries

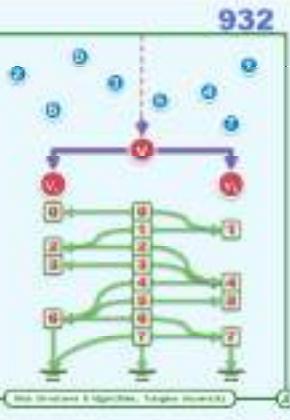
- can be answered in  $\mathcal{O}(r + \log n)$  time
- from a data structure of size  $\mathcal{O}(n \log n)$ ,
- which can be constructed in  $\mathcal{O}(n \log n)$  time



935

## Fractional Cascading

For any  $y$ -query with  $q_y$ , once we know its entry in  $A_v$ , we can determine its entry in either  $A_L$  or  $A_R$  in  $\mathcal{O}(1)$  additional time



932

## Beyond 2D

Unfortunately, it turns out that the trick of fractional cascading can only be applied to the last level

of the search structure, because all other levels need

the full tree search

to compute canonical sets

Given a set of  $n$  points in  $\mathbb{R}^d$ , an orthogonal range query

- can be answered in  $\mathcal{O}(r + \log^{d-1} n)$  time
- from a data structure of size  $\mathcal{O}(n \log^{d-1} n)$ ,
- which can be constructed in  $\mathcal{O}(n \log^{d-1} n)$  time

936



**Advanced Balanced Search Tree**  
**Interval Tree**  
**Stabbing Query**

#H (14)

dmitriykhakimov.ru

**Median**

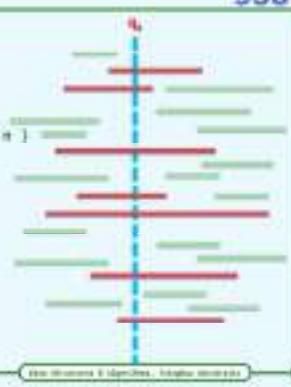
- Let  $S = \{s_1, \dots, s_n\}$  be the set of intervals
- Let  $P = 2S$  be the set of all endpoints
  - // by general position assumption,  $|P| = 2n$
- Let  $\text{median}(P) = x_{\text{med}}$  be the median of  $P$



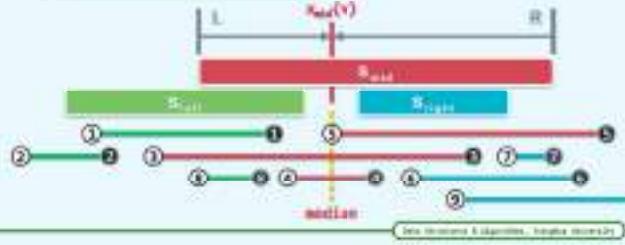
938

**Input**

- A set of **intervals** in general position on the **x-axis**
- $S = \{s_i \mid (x_i, x'_i) \mid 1 \leq i \leq n\}$
- A query point **q**:

**Partitioning (1)**

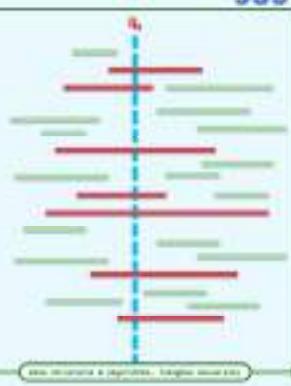
- All intervals can be then categorized into 3 subsets
- $S_{\text{left}} = \{s_i \mid x_i < x_{\text{med}}\}$  //intervals lying entirely **left** to  $x_{\text{med}}$
- $S_{\text{right}} = \{s_i \mid x_{\text{med}} < x_i\}$  //intervals lying entirely **right** to  $x_{\text{med}}$
- $S_{\text{med}} = \{s_i \mid x_i \leq x_{\text{med}} \leq x'_i\}$  //intervals that **contain**  $x_{\text{med}}$



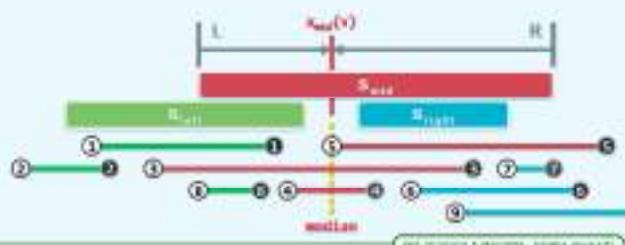
939

**Output**

- All intervals that contain  $q$ ,
- $\{s_i \mid (x_i, x'_i) \mid x_i \leq q, q \leq x'_i\}$
- To solve this query,
- we will use
- a structure named
- interval tree** ...

**Partitioning (2)**

- $S_{\text{interv}}[v]$  will be recursively partitioned until they are empty (leaves)



940

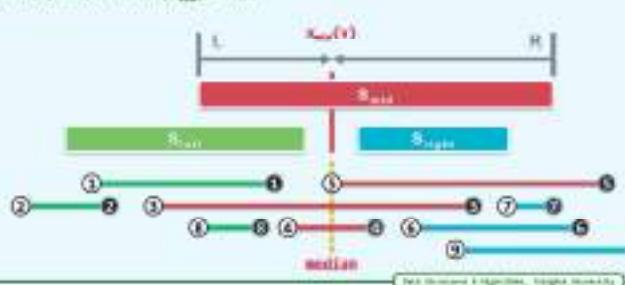
**Advanced Balanced Search Tree**  
**Interval Tree**  
**Construction**

#H (14)

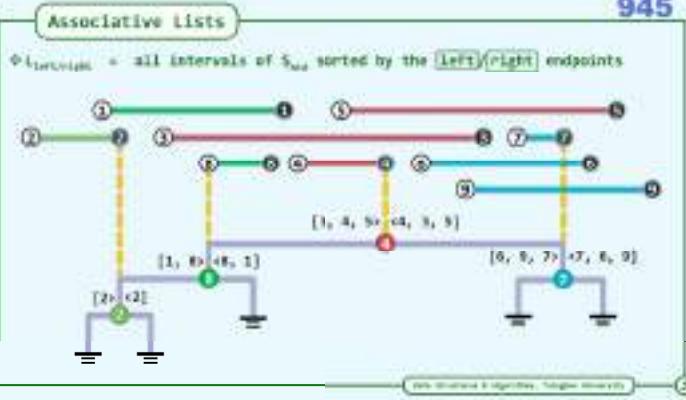
dmitriykhakimov.ru

**Balance**

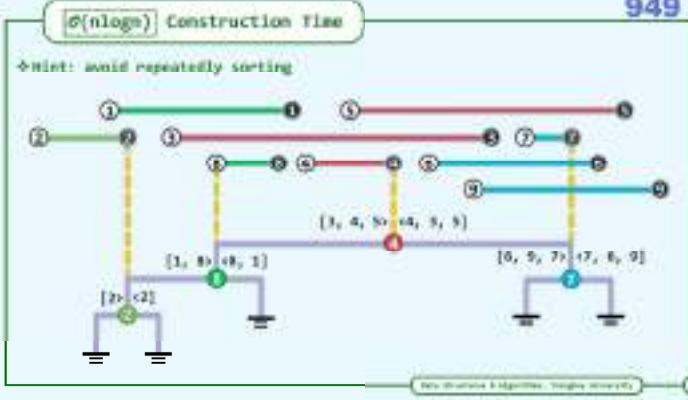
- $\max(\|S_{\text{left}}\|, \|S_{\text{right}}\|) \leq n/2$
- Best/worst case:  $\|S_{\text{med}}\| = n/2$



945



949



946

### Advanced Balanced Search Tree

Interval Tree  
Complexity (1)

苏伟峰  
dong@tsinghua.edu.cn

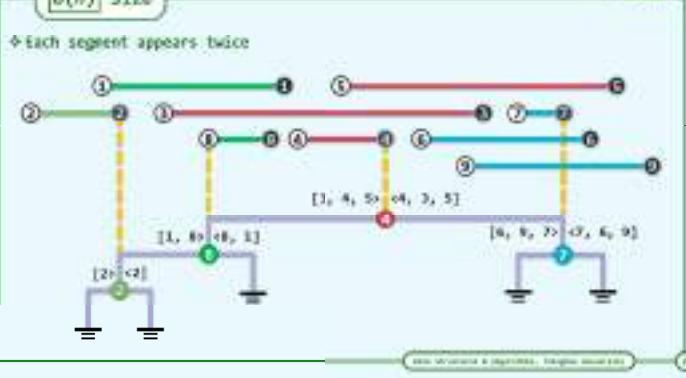
950

### Windowing Query

Interval Tree  
Query

苏伟峰  
dong@tsinghua.edu.cn

947



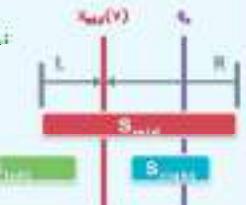
951

queryIntervalTree(  $v_i, q_e$  )

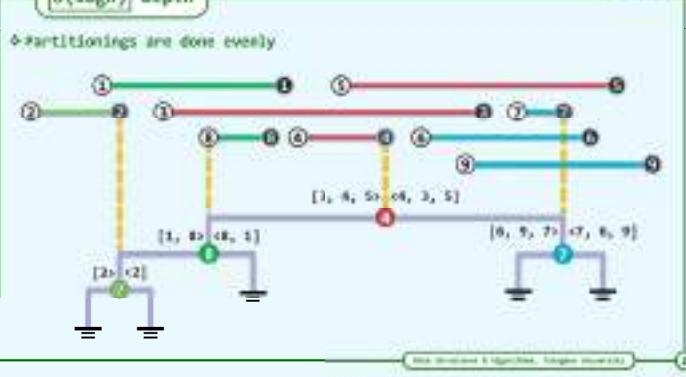
```

if ( !v ) return; //base
if (  $q_e < x_{\text{left}}(v)$  ) {
    report all segments of  $S_{\text{int}}(v)$  that contain  $q_e$ ;
    //by a scan of  $L_{\text{int}}(v)$ 
    queryIntervalTree( lc(v),  $q_e$  );
    //rc(v) can be ignored
}
...

```



948



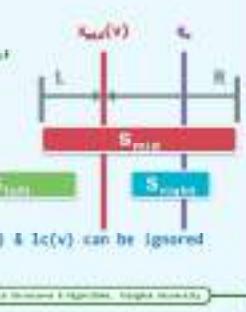
952

queryIntervalTree(  $v_i, q_e$  )

```

...
} else if (  $x_{\text{right}}(v) < q_e$  ) {
    report all segments of  $S_{\text{int}}(v)$  that contain  $q_e$ ;
    //by a scan of  $R_{\text{int}}(v)$ 
    queryIntervalTree( rc(v),  $q_e$  );
    //lc(v) can be ignored
} else //with a probability ~ 0
    report all segments of  $S_{\text{int}}(v)$ ; //both rc(v) & lc(v) can be ignored

```



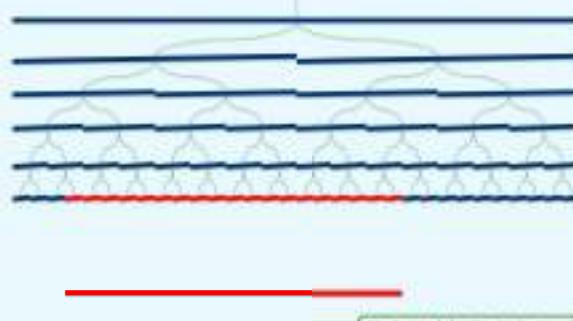


## Advanced Balanced Search Tree

Segment Tree  
BBST

At \$14  
dengtinghua@126.com

Still, Up To  $O(n)$  Space For Each Interval



## Replacing The Sorted Vector With A BBST

- For each leaf  $v$ ,
  - denote the corresponding elementary interval as  $EI(v)$
  - denote the subset of intervals containing  $EI(v)$  as  $Int(v)$  and
  - store  $Int(v)$  at  $v$



## 1D Stabbing Query

- To find all intervals containing  $q_i$ , we can
  - find the  $EI(v)$  containing  $q_i$ .  $\lceil d \log n \rceil$  time for a BBST
  - and then report  $Int(v)$   $\lceil d(1 + k) \rceil$  time



## Advanced Balanced Search Tree

Segment Tree  
Cousin Ancestor

邓伟华  
dengtinghua@126.com

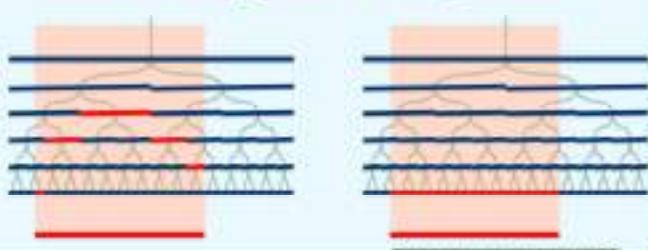
## Advanced Balanced Search Tree

Segment Tree  
Worst Case

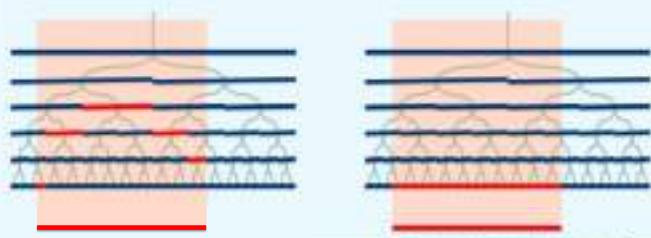
At \$14  
dengtinghua@126.com

If some of the leaves share a  $CA$ , then

a search path ends in any of them iff the  $CA$  is visited



So, instead, why not just store a **single** copy at the **leaf**?



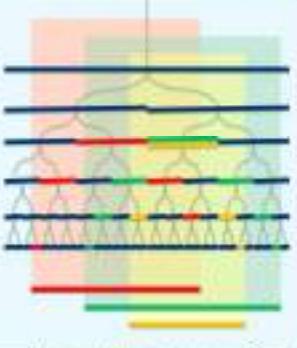
Data Structures & Algorithms, Highly Interactive

Consider any input interval  $x$

- $x$  is stored no more than twice on each level
- $x$  costs  $\Theta(\log n)$  space

Such an optimized BST

is called a **segment tree**



Data Structures & Algorithms, Highly Interactive

### Advanced Balanced Search Tree

Segment Tree  
Canonical Subsets

周廷朝

deng@zjhu.edu.cn

### Advanced Balanced Search Tree

Segment Tree  
Construction

周廷朝

deng@zjhu.edu.cn

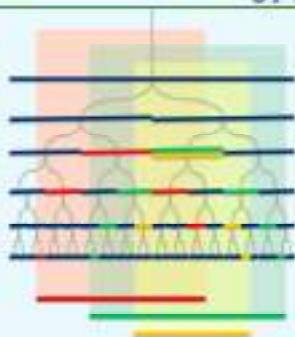
For each internal node  $v$

- let  $\text{EI}(v) = \text{EI}(v.\text{lc}) + \text{EI}(v.\text{rc})$
- denote the  $\text{CS}$  for  $v$  as  $\text{Int}(v)$  ...

For an input interval  $s$

belongs to  $\text{Int}(v)$

- iff
- it covers  $\text{EI}(v)$  but
- not  $\text{EI}(v.\text{parent})$

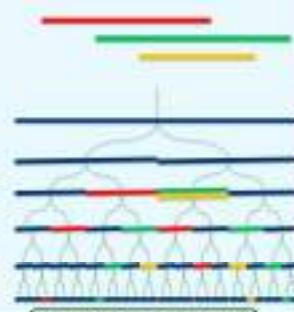


Data Structures & Algorithms, Highly Interactive

**BuildSegmentTree( I )**

```

// Construct a segment tree on
// a set I of n intervals
Sort all endpoints in I before
determining all EI's //O(nlogn)
Create T a BST on all the EI's //O(n)
Determine Int(v) for each node v
//O(n) if done in a bottom-up manner
For each s of I
    call insertSegmentTree( T.root , s )
  
```



Data Structures & Algorithms, Highly Interactive

### Advanced Balanced Search Tree

Segment Tree  
-  $\Theta(n\log n)$  Space

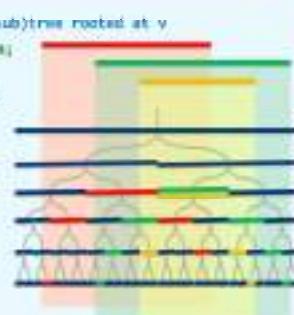
周廷朝

deng@zjhu.edu.cn

**insertSegmentTree( v , s )**

```

// Insert an interval s into a segment (sub)tree rooted at v
if ( Int(v) ⊂ s ) store s at v and return
if ( Int( lc(v) ) ⊂ s ) = Ø //recurse
    InsertSegmentTree( lc(v) , s )
if ( Int( rc(v) ) ⊂ s ) = Ø //recurse
    InsertSegmentTree( rc(v) , s )
at each level, 2 nodes are
visited (2 stores) + 2 recursions
Θ(log n) time
  
```

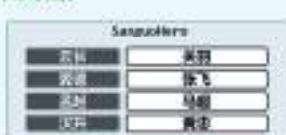


Data Structures & Algorithms, Highly Interactive



在这里，无论对外的访问方式，还是内部的存储方式都跟直接使用数据对象自身的键值

key与value地位等同，不必区分



↑ 调用访问`call-by-value`：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, Chapter 10, exercise 11

```
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 is order
end
```

```
import java.util.*;
public class Hash {
  public static void main(String[] args) {
    HashMap HM = new HashMap(); // Map
    HM.put("东岳", "泰山"); HM.put("西岳", "华山"); HM.put("南岳", "衡山");
    HM.put("北岳", "恒山"); HM.put("中岳", "嵩山"); System.out.println(HM);
    Hashtable HT = new Hashtable(); // Dictionary
    HT.put("东岳", "泰山"); HT.put("西岳", "华山"); HT.put("南岳", "衡山");
    HT.put("北岳", "恒山"); HT.put("中岳", "嵩山"); System.out.println(HT);
  }
}
```

Data Structures & Algorithms, Chapter 10, exercise 11

◆ 了解Java中HashMap与Hashtable的区别

◆ 安装JRE (<http://www.java.com>)

◆ 安装Perl (<http://www.perl.org>)

◆ 安装Python (<http://www.python.org>)

◆ 安装Ruby (<http://www.ruby-lang.org>)

◆ 安装Dash Table

```
#由字符串(string)修饰的一般无序字典(scalar) // 法国沙龙
my $hero = ("丘长"=>"关羽", "翼德"=>"张飞", "子龙"=>"赵云", "武圣"=>"关羽");
foreach $style (keys $hero) # Hash类型的变量由$修饰
  { print "$style => $hero{$style}\n"; }
$hero{"汉升"} = "黄忠";
foreach $style (keys $hero)
  { print "$style => $hero{$style}\n"; }
foreach $style (reverse sort keys $hero)
  { print "$style => $hero{$style}\n"; }
```

Data Structures & Algorithms, Chapter 10, exercise 11

## 9. 字典

散列

原理

书籍：微机。

秘书先做坏掉，任他空挂，发回书2。

单佳辉

<http://singlejiahui.sina.com.cn>

```
# beauty = dict # Python dictionary (hashable)
# ( ("刘强东", "西施"), ("落霞", "西施"), ("凤阙", "西施"), ("百花", "玉环") )
print beauty

# beauty["红颜"] = "西施"
print beauty

for alias, name in beauty.items():
  print alias, ":", name

for alias, name in sorted(beauty.items()):
  print alias, ":", name

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

Data Structures & Algorithms, Chapter 10, exercise 11

**General Inquiries**  
Tel: Toll Free: 1-800-IBM-4YOU  
E-mail: [sakshi@viet.ibm.com](mailto:sakshi@viet.ibm.com)  
[www.ibm.com/us](http://www.ibm.com/us)

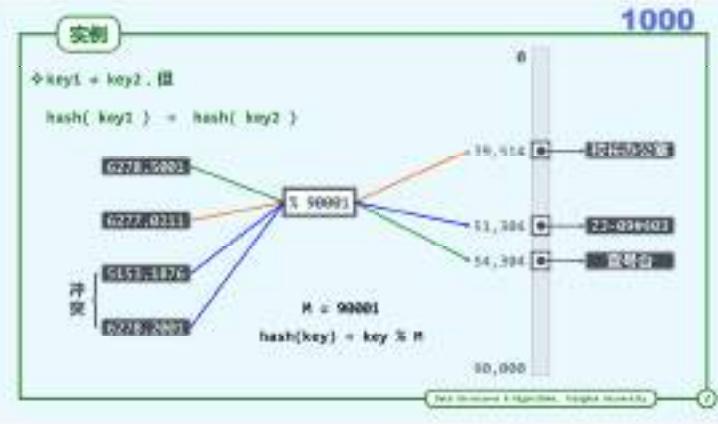
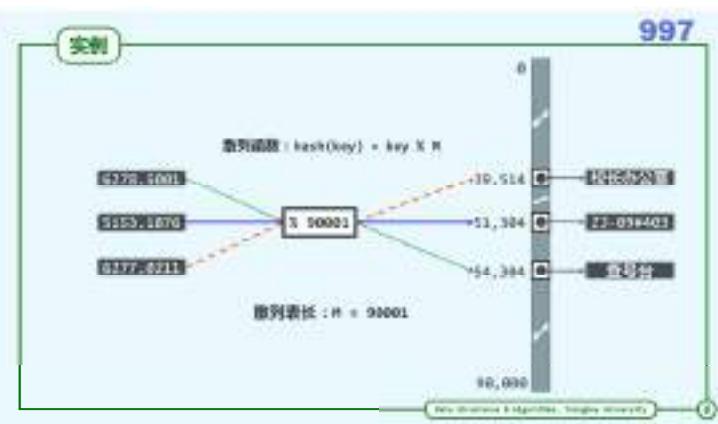
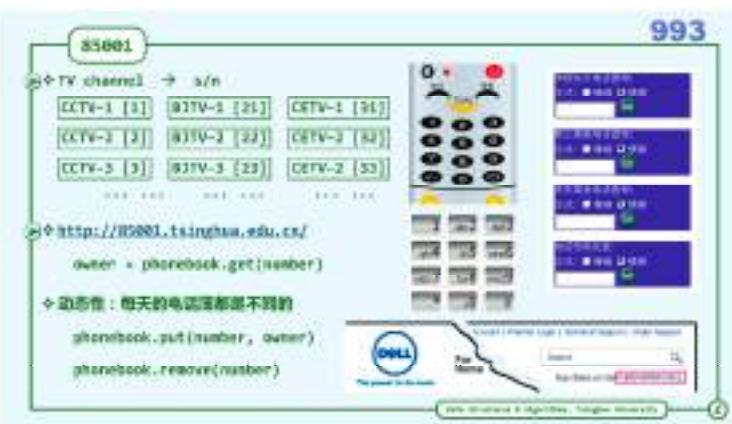
**Shopping**  
Tel: Toll Free: 1-865-SHOP-IBM

**Sales Center**  
1-865-2-LENOVO (1-865-253-0888)  
Mon - Fri: 9am-8pm (EST)  
Sat - Sun: 9am-8pm (EST)

**Customer Service**  
1-865-2-LENOVO (1-865-253-0888)  
Mon - Fri: 9am-8pm (EST)  
Sat - Sun: 9am-8pm (EST)



Data Structures & Algorithms, Chapter 10, exercise 11



## 完美散列

- 是否存在某种地址方法，能保证不出现冲突？  
散列、散列函数等等于一个散列 injection？
- 在关键码满足某些条件时，的确实可以实现单射式散列，比如...
- 对已知范围内的关键码集（比如ID）  
可实现完美散列 perfect hashing
- 采用两相散列模式  
仅需  $O(n)$  空间  
关键码之间互不冲突  
时间在最坏情况下，查找时间也不过  $O(1)$  时间
- 不过，在一般情况下，向散列无法保证存在...

## 两项基本任务

- 近似的单射，往往可行...

因此，我们需要...

## 1) 本节：精心设计散列表及散列函数

以尽可能降低冲突的概率；需要...

## 2) 下节：制造可行的散列

以便在发生冲突时，能够尽快予以缓解



## 生日悖论

- 将在座同学（对所有的词典）按生日（月/日）编散列存储  
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  $P_{\text{conflict}}(n) = ?$
- // 相率论与数据组织 讲义第一章，吉林大学数学系王晓峰
- $P_{\text{conflict}}(21) \approx 44.4\%$ ,  $P_{\text{conflict}}(22) \approx 47.0\%$ , ...,  $P_{\text{conflict}}(23) \approx 50.7\%$  //  $23/365 \approx 6.3\%$
- 约 180 人的概率： $1 - P_{\text{conflict}}(189) \approx 0.000000123$   
自 7 岁起，不吃不喝、无休无怨，每小时参加四次  
到 180 岁，才有可能遇到一次没有冲突的聚会
- 因此，在装填因子确定之后，散列函数的选择非常关键。散列函数的设计由得而讲究...

## 评价标准与设计原则

- 什么样的散列函数 hash() 更好？

- 确定 determinism：同一关键码总被映射至同一地址
- 快速 efficiency：expected- $O(1)$
- 同时 separation：尽可能充分地利用整个散列空间
- 均匀 uniformity：关键码映射到散列表各位置的概率尽量接近

可能造成类聚现象 clustering

## 9. 字典

## 散列函数

## 基本

此例散列的优点上，方法是将从 0 到 255 的字节转换成 32 位的整数  
的映射可以做三步操作，那么前两步正好对应于好的散列  
且：预散列不好，那就自己好。是下本是二十六步，但  
做的最后一步缩小，这样就正好做了二十七步。

## 散列码

http://www.hanwang.org

## 除余法

- $\text{hash}(\text{key}) = \text{key} \% M$

的命中，为何选  $M = 360001$ ？

- 若取  $M = 2^n$ ，其效果相当于

散列 key 的最低  $k$  位 (bit)，前面的  $n-k$  位对地址没有影响

$$M-1 = [0 \ 0 \ 0 \ \dots \ 0] 1 \ 1 \ 1 \ \dots \ 1$$

$$\text{key} \% M = \text{key} \% (M-1)$$

结论：发生冲突 1/F，且随 k 变大 // 发生冲突的概率大

- 若取  $M = 2^k$ ，缺陷和所改善

当 M 为 质数 时，散列函数先质数函数 充分，分布更均匀 // 好的散列

## 无法杜绝的冲突

- 散列函数  $\text{hash}(i) : S \rightarrow A$ ，不可能是单射

// 地址空间 S ~ 可能的词条

$$|S| = 2^{32 \times 8} = |A|$$

// 地址空间 A ~ 散列表



Elements	76726663	zebra
Facilit	382543	duckie
zaiti/d	7862473	penisise
terrie	536643	lemon
pietold	7432253	ridable

## PAD 法

- 散列值的映射...

- 不结点：无论表长 n 取值如何，总有  $\text{hash}(0) = 0$

- 平均均匀： $[0, n]$  的关键码，平均分配到 n 个桶；但相邻关键码的散列地址也必须相

- 一致均匀：邻近的关键码，散列地址不相邻

// 更高阶的均匀性呢？

- MAD = multiply-add-divide

取  $n$  为常数， $a > 0, b > 0, c \leq n, d \leq n$

$$\text{hash}(\text{key}) = (a \times \text{key} + b) \% n$$

- 当然，某些特定情况下，未必需要高的均匀性

## 更多散列函数

◆ 数字分析：selecting digits

提取key中的某几位，构成地址

比如，取十进制数的低奇数位

 $hash(1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9) = 1\ 3\ 5\ 7\ 9$ 

◆ 在平方根中 mid-square

取key的中间若干位，构成地址

 $hash(123) = 512 // 保留 key^2 = 123^2 = 15123 的中间几位$  $hash(1234567) = 556 // 1234567^2 = 1524125677489$ 

Data Structures &amp; Algorithms - Hashing Methods

## 更多散列函数

◆ 折叠法（Folding）：将key分割成等宽的若干组，取其总和作为地址

 $hash(123456789) = 1238 // 123 + 456 + 789$ , 直接相加 $hash(123456789) = 1066 // 123 + 056 + 789$ , 往前折进

◆ 按位或法（XOR）：将key分隔成等宽的二进制段，按异或运算得到地址

 $hash(1100110111) = 110_1 // 110 + 011 \wedge 011$ , 直接相加 $hash(1100110111) = 011_1 // 110 \wedge 110 \wedge 011$ , 往前折进

◆ ...

◆ 总之，越复杂，越是没有规律，越妙

1010

1011

## 9. 字典

## 散列函数

更多

邓伟强

deng@tonghuaxue.com

有趣的齐与有趣的文化，错过一万年。

## (伪) 随机数发生器

ANSI:  $rand(x+1) = [x \times rand(x)] \% R$  // 整数,  $x \leq R$  $R = 2^31 - 1 = 1073741823$  $x = 2^{31} - 1 = 2,147,483,647 \quad 01111111111111111111111111111111$ 

## (伪) 随机数校正

编程:  $hash(key) = rand(key) + [rand(0) \times R^{0.5}] \% R$ 种子:  $rand(0) = T$ 

◆ 随机数校正伪随机数发生器，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

到底的散列表 可移植性差——故要慎用此法！

1012

## (伪) 随机数法

```
unsigned long int next = 1; //The C Programming Language (2nd edn), p46
void srand(unsigned int seed) { next = seed; }
int rand(void) { //1103515245 = 3*5*7*7*129749
    next = next + 1203515245 + 12345;
    return (unsigned int)(next/45536) % 32768;
}
rand() = next / 2^15 = 2^15
```

◆ 另类随机数 (单) 随机

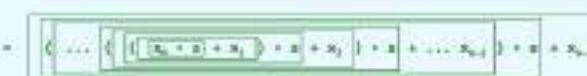
int rand() { int uninitialized; return uninitialized; }

char\* rand( t\_size n ) { return (char\*) malloc( n ); }

◆ 以上方法，可应用于嵌入？

Data Structures &amp; Algorithms - Hashing Methods

## 多项式法

◆  $hash(x = [x_0, x_1, \dots, x_{n-1}]) = x_0 R^{n-1} + x_1 R^{n-2} + \dots + x_{n-2} R^1 + x_{n-1}$ 

◆ static size\_t hashCode( char s[] ) { // 这是多项式，但更快速

unsigned int h = 0;

for( size\_t n = strlen(s), i = 0; i &lt; n; i++ )

{ h = ((h &lt;&lt; 5) | (h &gt;&gt; 27)); h += (int)s[i]; }

return (size\_t)h;

} //有必要如此复杂吗？

◆ 别告诉我用简单的就好，比如...

Data Structures &amp; Algorithms - Hashing Methods

## 多项式法

◆ 字符自定义映射为数字:  $f(c) = code(UPPER(c)) - 64$  // 'A' - 'Z' = [1, 26]再简单相加:  $hash(s) = \sum f(c)$  $hash = 0 + 1 + 19 + 3 + 26$ 

◆ 字符相加的次序很重要，将引发大量冲突

I am Lord Voldemort.

Tom Marvolo Riddle.



◆ 如果字符不同，数目不等...

He's Harry Potter.

◆ Key to improving your programming skills:

Learning Tonghuaxue Data Structures &amp; Algorithms

1015

## Java: hashCode()

◆ hashCode() 方法

适用于 Java 中的所有对象

将任意类型的对象转换为 (32 位 int 型) 整数

对于非空型的 key，先转换为整数（散列码），然后再做操作

◆ hashCode()：效果如何？效率如何？是如何实现的？

◆ object.hashCode() = object 内存中的地址

◆ 问题：相关的对象地址也相近，冲突概率高 // 重播的是...

◆ 散列码与对象的内容无关

比如，完全相同的两个字符串对象，散列码仍然不同

◆ 有无替代方案？

1016

## 9. 字典

## 9. 字典

相冲突  
开放散列

相冲突  
开放散列

Every mistake I've ever made  
Has been rebashed and then replayed  
As I get lost along the way.

相冲突

http://tinyurl.com/4zqjw2

相冲突  
开放散列

相冲突  
开放散列

Every mistake I've ever made  
Has been rebashed and then replayed  
As I get lost along the way.

相冲突

相冲突  
开放散列

相冲突  
开放散列

## 多指位

## 1018

## multiple slots

简单元组分派于指位 slot

导致：与同一单元冲突的词条

只要槽位数目不多

依然可以保证  $\Theta(1)$  的时间效率

但是，需要为每个桶配备多少个指位，才能保证  $\Theta(1)$ ？ //难以预测

预留过多，空间浪费

无论预留多少，极端情况下仍有可能不等



bucket[n + 1]  
bucket[n]  
bucket[n - 1]

http://tinyurl.com/4zqjw2 http://tinyurl.com/4zqjw2

## 开放定位

## 1022

## closed hashing

当然对应于 open addressing：

只要有必要，任何散列桶都可以接纳任何词条

## probing sequence/chain:

为每个词条，都要事先约定若干空闲桶，优先级逐次下降

查找：沿虚线找，逐个映射下一插槽单元，直到...

- 命中成功，或者

- 遇达一个空槽（已遇过所有冲突的词条）失败

具体情况，查找策略如何约定？

http://tinyurl.com/4zqjw2 http://tinyurl.com/4zqjw2

## 独立链

## 1019

## linked-list chaining / separate chaining

每个桶存放一个指针

冲突的词条，组织成列表

优点：无需为每个桶准备多个槽位

任意多次的冲突都可解决

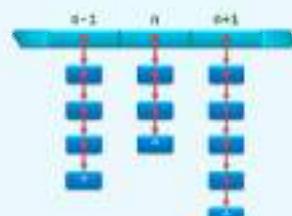
查找操作实现简单，统一

但是：查找需要额外空间

节点需要动态申请

更重要的是...

空间未必连贯分布，系统内存几乎失效



http://tinyurl.com/4zqjw2 http://tinyurl.com/4zqjw2

## 线性试探：策略

## 1023

## Linear probing

一旦冲突，测试探后一空闲单元

| hash(key) + ① | % M

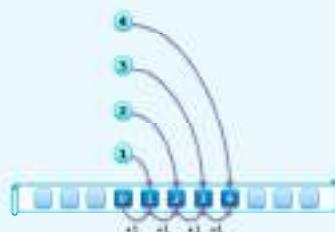
| hash(key) + ② | % M

| hash(key) + ③ | % M

| hash(key) + ④ | % M

...

直到命中成功，或遇达空槽失效



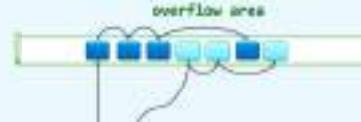
http://tinyurl.com/4zqjw2 http://tinyurl.com/4zqjw2

## 公共溢出区

## 1020

## overflow area

单独开辟一块连接空间



发生冲突的词条，暂存入此区域

结构简单

算法易于实现

但是，不冲突而已，一旦发生冲突

最坏情况下，处理冲突所需的时间正比于溢出区的规模

http://tinyurl.com/4zqjw2 http://tinyurl.com/4zqjw2

## 线性试探：评估

## 1024

## 在散列表内部解决冲突

- 无须附加的（链表、链表溢出区等）空间

- 能将本身保持简洁

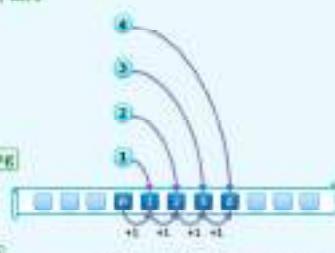
只要还有空槽，记早会找到

但是，已发生过（并已消除）的冲突

会导致本不必发生的冲突 clustering

好在，查找具有局部性

可充分利用系统缓存，有效减少 I/O



http://tinyurl.com/4zqjw2 http://tinyurl.com/4zqjw2

插入 + 删除

◆ 插入：新词条若尚不存在，则存入查找~~终止点的空桶~~

◆ 查找：可能因为被~~截断~~

◆ 删除：简单地清除表中的值？

- [某些]查找将可能因此被~~切断~~，而该词条可能~~消失~~——明明存在，却访问不到

- [多路]查找将可能因此被~~切断~~...

◆ 另外说，此时的~~查找结果如何可~~表示~~~~？

Data Structures &amp; Algorithms, Chapter 10 (Answers)

## 9. 词典

消除冲突

平方法探

我原以为

这样何尝不是一种所谓的解脱

要背负的辛苦又有谁能够清楚

内心的冲突

郑佳群

wangzhang@tsinghua.edu.cn

## 9. 词典

消除冲突

数据删除

郑佳群

wangzhang@tsinghua.edu.cn

Data Structures &amp; Algorithms, Chapter 10 (Answers)

链路

◆ [lazy removal]：仅做~~删除标记~~，而按~~键不必被移除~~

◆ 此后，若有链路标记的桶所指向的~~格子~~，因固体的操作类型而异：

- [查找]向标记

被操作“~~必不匹配的非空桶~~”，查找将在必须以~~转移~~

- [插入]向标记

被操作“~~必匹配到的空桶~~”，可以用来存取新词条

◆ 具体过程的实现...

Data Structures &amp; Algorithms, Chapter 10 (Answers)

```
template <typename K, typename V> int HashTable<K, V>::probeHLL(const K &k) {
    int r = hashCode(k) % M; // 从首个桶起沿途检测，跳过所有冲突的和碰撞指针指向的桶
    while ((ht[r] == K || ht[r] == key) || (ht[r] && !isInitiallyRemoved(r))) {
        r = (r + 1) % M; // 按位试探（注意并列判断的次序，命中可能性更大哦朋友）
    }
    return r; // 利用 hashCode[r] 是否为空，即可判断查找是否成功
}
```

```
template <typename K, typename V> int HashTable<K, V>::probeFirst(const K &k) {
    int r = hashCode(k) % M; // 从首个桶起
    while ((ht[r] == K) || (r + 1) % M == key) // 会直接检测到第一个空桶（无论是否带有链接）
        return r; // 利用 hashCode[r] 是否为空，即可判断查找是否成功
}
```

Data Structures &amp; Algorithms, Chapter 10 (Answers)

## 优点、缺点及疑惑

数据就来提供有所缓解

直观上，各桶间的~~线性碰撞~~

一些冲突，可~~利用桶的向量非碰撞~~

若涉及~~外存~~，I/O 的激增

只要~~有空桶~~，就...一查就...找出来吗？

1031

Data Structures & Algorithms, Chapter 10 (Answers)

装填因子，须足够小！

$\{0, 1, 2, 3, 4, 5, \dots\}^2 \% 11 = \{0, 1, 4, 9\}$

N若为合数： $n^2 \% N$ 可能的取值必然少于 $N/2$ 种——此时，只要对底数均非空...

$\{0, 1, 2, 3, 4, 5, \dots\}^2 \% 13 = \{0, 1, 4, 9, 5, 12\}$

N若为质数： $n^2 \% N$ 可能的取值恰好会有 $N/2$ 种——此时，性质需要检测前 $N/2$ 项数据

个原理：若 $N$ 是质数，且 $A < 0.5$ ，就一定能够找出；否则，不一定得

Data Structures & Algorithms, Chapter 10 (Answers)

查找倍数的倍，必定够长！

1033

◆ 证明：假设存在  $a \leq b < 2k + M/2$ ，使得

后者查找倍，而 a 和 b 未被此冲突

◆ 于是： $a^2 \equiv b^2 \pmod{M}$  的某一两条线，亦即

$$a^2 \equiv b^2 \pmod{M}$$

$$b^2 - a^2 \equiv (b-a)(b+a) \equiv 0 \pmod{M}$$

◆ 然而： $b-a \leq b+a \leq M$  —— 这与 M 为质数矛盾

◆ 那么，另一半的倍，是否也能用起来呢...

$4k+3$

◆ 两两直除： 3 5 7 11 13 17 19 23 29 31 ...

◆ 假设操作策略  $M = 4k+3$ ，必然可以保证查找的前  $M$  项均无冲突

$\pm 1^2$	-36	-25	-16	-9	-4	-1	0	1	4	9	16	25	36
5				1	4	9	16	25	36				
7			5	3	6	8	1	4	2				
11	8	6	2	7	10	0	1	4	9	5	3		
13	3	1	10	4	9	12	0	1	4	9	3	12	18

◆ 反之， $M = 4k+1$  时... 必然不可使用？

1034

## 9. 词典

指针冲突

双向平方试探

单链表

dong@tonghu.edu.cn

## Two-Square Theorem of Fermat

◆ 每一自然数均可表示为一对整数的平方和，当且仅当  $x \equiv 4 \pmod{4}$

◆ 只要注意到：

$$(u^2 + v^2)(s^2 + t^2) = (us + vt)^2 + (ut - vs)^2$$

$$= (vs + ut)^2 + (vt - us)^2$$

$$(1^2 + 3^2)(5^2 + 8^2) = (19 + 24)^2 + (16 + 15)^2$$

$$= (35 + 36)^2 + (26 - 10)^2$$



◆ 就不难推知：

每一自然数均可表示为一对整数的平方和，当且仅当

在其素分量中，假如  $M = 4k+1$  的每一素因子均为偶数次方

指针

1035

◆ 四种冲突源，依次向后过滤

| hash(key) + 3 | X M

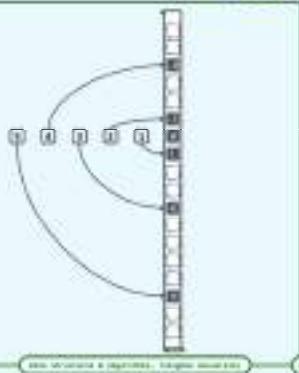
| hash(key) - 3 | X M

| hash(key) + 2 | X M

| hash(key) - 2 | X M

| hash(key) + 1 | X M

| hash(key) - 1 | X M



1039

## 9. 词典

指针冲突

再散列

单链表

dong@tonghu.edu.cn

查找性，彼此独立？

1036

◆ 正向同进向的子溢出结，假设有  $M/2$  个相同的值

$-M/2, -\dots, -2, -1, 0, 1, 2, \dots, M/2$

## Double Hashing

◆ 先执行第二散列函数：hash2()

◆ 一旦发生冲突，以 hash2(key) 为偏移增量，重新确定地址

| hash(key) + [1] \* hash2(key) | X M

| hash(key) + [2] \* hash2(key) | X M

| hash(key) + [3] \* hash2(key) | X M

...

直到发现一个空槽

◆ 比如，hash2() = 1 时，即呈线性试探

1040

◆ 除了 0，这两个序列是否有... 其它公共的模？

1040

## 9. 字典

相容冲突  
重映射

坏习惯

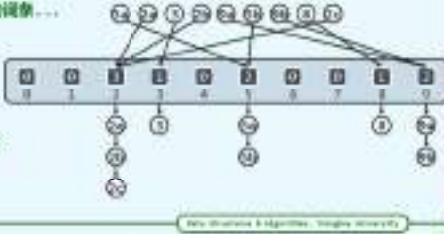
dengtingguoxiaoxue.com

## 一般情况

进一步地，若为许多键值对， $m \ll n$ ，甚至可能  $m \ll n$

比如，清华大学2013级本科学生按生日排序，设有  $n = 3300$ ,  $m = 365$

劣势使用散列表，相互冲突的键值...



组成独立链

空间浪费

· 链表长  $\rightarrow$  所有链表总长

$= O(n + m)$  //在用到散列

## Rehashing

```
+template <typename K, typename V> //随着散列因子增大，冲突概率、相容冲突率都将增加
void HashTable<K, V>::rehash() { //此时，不能“集体搬迁”至一个更大的散列表
    int old_capacity = m; Entry<K, V>* old_ht = ht; h = p;
    ht = new Entry<K, V>[m = primeMTC(2 * m)]; //新容量至少加倍
    memset(ht, 0, sizeof(Entry<K, V>) * m); //初始化指针
    release(lazyRemoval); lazyRemoval = new BitMap(m); //初始化位图，容量至少加倍
    for (int i = 0; i < old_capacity; i++) //扫描原散列表
        if (old_ht[i]) //将非空槽中的键移走
            putt(old_ht[i].key, old_ht[i].value); //插入空散列
    release(old_ht); //释放原散列表
}
```

## 9. 字典

桶排序  
算法

Don't put all your eggs  
in one basket.

坏习惯

dengtingguoxiaoxue.com

## 一般情况

+ initialization: 初始化散列表（开新空间，设置首指针的头） //如有必要，可以优化

distribution: 分派键值对，选择并指派对存储的键值 //插入位置有讲究

collection: 回填数据，串接所有非空链表 //串接次序和方向也有讲究

↓ 只要实现理由，必须保证稳定性，即相同键条的次序与插入相同 //就是索性，还想底数

↓ 时间复杂度  $= O(n) + O(n) + O(n) = O(n + m)$

↓ 大量向量搬移时  $= O(n)$ ，性能相当于线性

↓ 等效的均匀分布时，还是随机

## 简单情况

↓ 给定  $[0, n]$  内的  $n$  个互异整数，如何高效地排序？ //必有  $n \leq m$

↓ 假设散列表  $t[0, n]$  //将元素保留在  $n$  个 bin

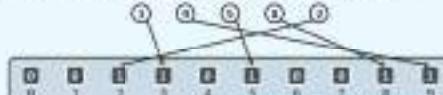
initialization: for  $i = 0$  to  $m - 1$ , let  $t[i] = 0$  // $O(n)$ ，可优化至  $O(1)$

distribution: for each key in the input, let  $t[key] = 1$  // $O(n)$

enumeration: for  $i = 0$  to  $m - 1$ , output 1 if  $t[i] = 1$  // $O(n)$

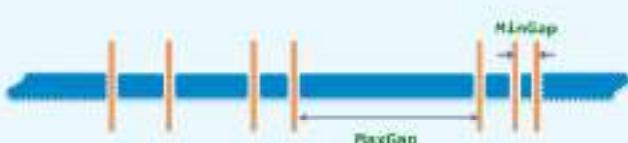
↓ 空间:  $O(n)$

时间:  $O(n + m)$



## MaxGap

↓ 任意  $n$  个互异点均将其轴分为  $n - 1$  份有序区间，其中哪一段最长？

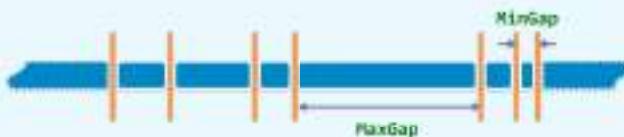


↓ 如果不追求效率，最直觉的方法是...

## 平凡算法

1049

◆ 则排序点(插入) //最好情况下  $\Theta(n \log n)$   
依次计算各相邻点的间距，保留最大者  $\Theta(n^2)$



◆ 可否更快？

◆ 采用分桶策略，可改进至  $\Theta(n)$  时间...

## 词典序

1053

◆ 有时，关键词由多个字母组成： $s_1, s_{1+1}, \dots, s_k$

◆ 将字符串操作(字典)，则关键由即单词

◆ 于是，可按词典序方式排序 lexicographic order

◆  $\text{AA} > \text{AK} > \text{AQ} > \text{AJ} > \text{AO} > \dots > \text{A2} >$   
 $\text{KA} > \text{KQ} > \text{KJ} > \text{KO} > \dots > \text{K2} >$   
 $\text{QA} > \text{QK} > \text{QJ} > \text{QO} > \dots > \text{Q2} >$   
 $\text{JA} > \text{JK} > \text{JO} > \text{JQ} > \dots > \text{J2} >$   
 $\text{OA} > \text{OK} > \text{OJ} > \text{OO} > \dots > \text{O2} >$

## 线性算法

1050

◆ 找到最左点、最右点  $\Theta(n)$  //一维扫描法  
将有效范围均匀地划分为  $n-1$  段 ( $n$  个桶)  $\Theta(n)$  //相当于数列法  
通过映射，将各点归入对应的桶  $\Theta(n)$  //模余法  
在各桶中，动态记录最左点、最右点  $\Theta(n)$  //可能相同甚至没有  
距离相等(非空)桶之内的“距离”  $\Theta(n)$  //一维遍历法  
最大的距离即 MaxGap  $\Theta(n)$  //赢家阶段



## 算法

1054

◆ 逐位优先：从  $1$  到  $k$ ，依次以各进制为序，做一遍桶排序

3200	5112	5113	4320	6441
4320	71214	73114	3200	5276
4320	51214	43210	6441	4320
5112	51200	64211	52212	7234
5120	43200	51210	72114	4698
5120	64411	32000	52210	3280
7214	64411	46980	52112	

## 正确性

1051

◆ 正确性：MaxGap是少与相邻的两个桶相交  
换句话说，定义 MaxGap 的点不可比属于同一个桶  
◆ 对称的 MaxGap 问题： $n-1$  距有界区间中，何谓最短？



## 9.词典

## 基础排序

## 算法

## 单链表

[www.yuvalinghuo.com](http://www.yuvalinghuo.com)

## 9.词典

基础排序  
算法

## 单链表

[www.yuvalinghuo.com](http://www.yuvalinghuo.com)

## 正确性

1056

◆ Radixsort的正确性何以见得？自学归纳！

◆ 归的假设：在经过前  $i$  位的桶排序之后，所有问题关于  $i$  位的  $10^i$  位有序——第  $i$  位意见成立

◆ 假设  $i-1$  位均成立，试考察第  $i$  位排序之后的对称

无非两种情况：1) 凡第  $i$  位不同的词语：即使此前已是排序，现在亦必已转化为有序

2) 凡第  $i$  位相同的词语：由于此排序的稳定性，必保持原有次序

◆ 由此也可看出，只要实现排序，被数焯序同样成立

## 时间成本

- ◆ 基数排序的总时间为  $n + 2m$
- ◆  $n = 2m$
- ◆  $\dots$
- ◆  $n = 2m$ , //  $m$  为整数的取值范围
- ◆  $\sigma(n) \times (n + m) / m = \max\{m_1, \dots, m_k\}$

◆ 因此  $\sigma(n)$  是常数级， $\sigma(n) = 1$

◆ 在一些特定场合，基数排序非常高效，比如...

## 基数排序算法

- ◆ 插述题：将所有元素转换为  $n$  进制形式：

$$x = \{x_0, \dots, x_1, x_k\}$$

◆ 于是，每个元素均转化为  $k$  个域，故可能要使用 Radixsort 算法

◆ 排序时间  $= \Theta(n + m) = \Theta(n)$  // “突破”了此题规定的下界！

◆ 考虑

- 整数 取值范围有限制
- 不再是 基于比较的计算模式

◆ 相等：将处理本身需要多少时间？回忆一下，此前的相关内容...

## 整数排序

- ◆ 对于  $[0, n^d]$  内的任取  $d$  个整数，如何高效排序？ // 常数  $d > 1$
- ◆ 解释： $d/d = \log(n) / \log(n^d) //$  将时数放宽，实际应用中不推荐
- ◆ 插述题：将所有关键转换为  $n$  进制形式  $x = \{x_0, \dots, x_1, x_k\}$
- ◆ 于是，将问题转化为  $d$  个域的基数排序问题，可套用前述算法
- ◆ 排序时间  $= \sigma(n + m) = \Theta(n)$  // “突破”了此题规定的下界！
- ◆ 考虑：1) 整数 取值范围 有限制；2) 不再是 基于比较 的计算模式
- ◆ [将处理本身需要计入，它本身需要多少时间？回忆一下，此前的相关内容...]

## 9.词典

## 计数排序

## 解 例

[tiny.cc/meyarw](http://tiny.cc/meyarw)

## 9.词典

基础排序  
整数排序

## 解 例

[tiny.cc/meyarw](http://tiny.cc/meyarw)

◆ 相信：基础排序中反复做的操作...

◆ 亦属 小集合 + 大数据 类型，是否可以更快？

◆ 仍以扑克排序为例  $n = 13, m = 4$ ，假设已按 直数 排序，以下对花色排序

1) 扫描 (方向 无序 ) 所有牌，统计各种花色的 数量

$\Theta(n)$



## 常对数密度的数据集

◆ 设  $1/c$  为常数，考虑取自  $[0, n^2]$  内的  $n$  个整数

$$\text{常数密度} = \frac{n}{n^2} = \frac{1}{n} \rightarrow 0$$

$$\text{对数密度} = \frac{\ln n}{\ln n^2} = \frac{1}{2} = O(1)$$

◆ 意即，这类型数据的对数密度 不超过常数

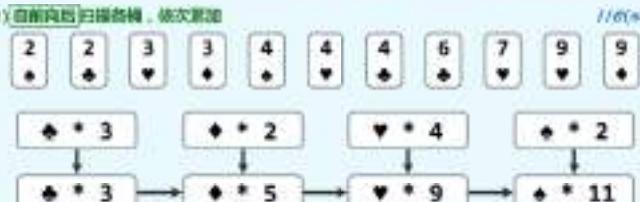
◆ 该一特征条件，在实际应用中不难满足

◆ 例如  $c = 4$ ，对数便限制 64 位整数而言，也只要  $n > (2^{64})^{1/4} > 2^{16} = 65,536$

◆ 对于这类型数据，有无速率  $= (n \log n)$  的排序算法？

2) 自前向后扫描各横，依次累加

$\Theta(n)$



3) 自后向前扫描所物统计

$\Theta(n)$

对扫描的计数 等

相对对白梅的 计数，就应该排在最后有序序列中到达的 种

2) 自前向后扫描各横，依次累加

$\Theta(n)$



1065

实例

Java Simulation & Algorithm, Intelligent Memory

1069

实例

Java Simulation & Algorithm, Intelligent Memory

1066

实例

Java Simulation & Algorithm, Intelligent Memory

1070

分析

- 时间复杂度 =  $O(n + m + n) = O(n)$
- 高效处理大规模数据
- 空间复杂度 =  $O(n)$
- 充分利用多维矩阵特点

是否一步的扫描次序，可否改为自前向后？

Java Simulation & Algorithm, Intelligent Memory

1067

实例

Java Simulation & Algorithm, Intelligent Memory

1071

## 9.词典

翻转表  
结构

去沿江上下，或二十里，或三十里，造南亭北榭一排火台，每台用五十炬守之。

平准书

http://www.gutenberg.org

1068

实例

Java Simulation & Algorithm, Intelligent Memory

1072

动机与思路

能否综合向量与列表的优势，高效地实现词典接口？  
具体地，双向使得各接口的效率均为 $O(\log n)$ ？

**skip lists** [Mitsuharu Nagai, 1990] Help Lists: A Probabilistic Alternative to Balanced Trees

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.

本节介绍等概率型链表，即定型 (deterministic) 翻转表词典

Java Simulation & Algorithm, Intelligent Memory

↑ 分层次，相邻结点的多个位置： $s_1, s_2, s_3, \dots, s_n$  // 纵向  $\rightarrow n$

↑  $S_i$  的操作 [top]  $s_1$   $s_2$   $s_3$   $s_4$   $s_5$   
↓  $S_i$  的操作 [bottom]  $s_1$   $s_2$   $s_3$   $s_4$   $s_5$

↑ 节点在多棵树的四个引用：

- 指向父层 (level) : prev(), next() // 语义头尾，用哨兵
- 纵向连接 (lower) : above(), below()

Data Structures & Algorithms, Chapter 10

```
template <typename K, typename V> class Skiplist { // 多重地址表
public: Dictionary<K, V>; public List<QuadListEntry<K, V>> *qlist;
protected:
    bool skipSearch( ListHeader<QuadListEntry<K, V>> *qlist,
                      QuadListEntry<K, V> *entry, K key );
public:
    int size() { return empty(); } // 空表
    int level() { return list.size(); } // 层高，即 quadList 总数
    bool add( K, V ); // 插入 (Skiplist 先调用插入，插入后成功)
    V * get( K ); // 读取 (基于 skipSearch() 直接实现)
    bool remove( K ); // 删除
}
```

Data Structures & Algorithms, Chapter 10

空间性能

↑ 故事之家庭的单层列表

每次操作过程中，要访问的节点是否会实际地增多？

每个节点都最多可能更新n次，空间复杂度是否因此有实测影响？ // 为单向表而生...

↑ 生长概率决策概率： $S_i$  中的每个关键码，在 $S_{i+1}$  中依然出现的概率，均为  $p = 1/2$

——暂且假设成立，稍后说明如何保证

↑ 可见，高度符合几何分布：

$$\Pr\{ h = k \} = p^{k-1}(1-p)$$

于是，期望的塔高  $E(h) = 1/(1-p) = \frac{1}{1-p}$

↑ 什么，没有学过概率？不要紧，有初等的解释...

Data Structures & Algorithms, Chapter 10

只见秦淮老歌女子连哭带唱的站起身来，向众人拱了拱手，缓步走到窗中，微皱双眉，在足探出，已落在欧阳闪躺在雪地的身子之上，揭开被子，...，把一包巧打连绵的“苏荷春”递了出来，脚下疾速如飞，每一步都踩在赵庭的身子之上。

http://tiny.cc/meyarw

↑ 空间性能

↑ 故事之：由看戏到 - 由高到底

$S_1$   $S_2$   $S_3$   $S_4$   $S_5$

↑ 实例：成功 {21, 34, 1, 39}，失败 {38, 8, 39}

↑ 规律：直接的问接决定于  $h$ 。询问的累计询问次数

那么，是否可能既  $h$  次过多，首先导致从头询问过多？

Data Structures & Algorithms, Chapter 10

QuaList

```
template <typename T> class QuaList { // 四联表
private: int _size; QuaListNode<T> header, trailer; // 领域，哨兵
protected: void init(); int clear(); // 初始化、清除所有节点
public: QuaListNode<T> first() const { return header->succ; } // 首节点
        QuaListNode<T> last() const { return trailer->pred; } // 尾节点
        T remove( QuaListNode<T> p ); // 删除 p
        QuaListNode<T> insertAfterAbove(); // 插入
        T const & e; // 数据项 e，使之成为
        QuaListNode<T> p; // p 的邻接，以及
        QuaListNode<T> b = NULL; // p 的上层
}
```

Data Structures & Algorithms, Chapter 10

```
template <typename K, typename V> bool Skiplist<K, V>::skipSearch(
    ListHeader<QuadListEntry<K, V>> *qlist, // 从指定 qlist 的
    QuadListEntry<K, V> *entry, K key ) { // 节点 p 出发，向右，向下面找
    while ( true ) { // 在每一层从右向左查找，直到出现更大的 key，或者溢出至 trailer
        while ( p->succ && ( p->entry.key >= key ) ) p = p->succ; p = p->pred;
        if ( p->pred && ( key == p->entry.key ) ) return true; // 找中则成功返回
        qlist = qlist->succ; // 否则进入下一层
        if ( ! qlist->succ ) return false; // 若已到最底层，则意味着失败；否则...
        p = p->pred ? p->below : qlist->data->first(); // 移至当前层的下一节点
    } // 确认：无论成功或失败，遍历的 p 均为其中不大于 key 的最大者？
} // 地位：地位于精英的设置，略胜环节简化了？


```

Data Structures & Algorithms, Chapter 10

## 双向链表 / 遍历

• 理论：一堆堆高度（不等于  $\lfloor \log n \rfloor$ ）的频率  $= p^k + (1-p)^k$

• 引理：随机n的增加， $S_k$ 为 $\emptyset$ /非空的概率总和 $\uparrow$ 上升/ $\downarrow$ 下降

$$\Pr[S_k = 0] = (1-p^k)^n \geq (1-n \cdot p^k)$$

$$\Pr[S_k > 0] \leq n \cdot p^k$$

• 结论：跳转表高度  $\lambda = O(\log n)$  的概率极大

• 比如：若  $p = 1/2$ ，则  $\Pr[\lambda = \lfloor \log n \rfloor]$  跳转表（高度仅高  $\lambda \geq k$ ）的概率为

$$\Pr[S_k > 0] \leq n \cdot p^k = n \cdot n^{-1} = 1/n^2 \rightarrow 0$$

• 简记：查找过程中，访问跳转的高度，累计不超过  $\text{expected-}\delta(\lfloor \log n \rfloor)$

## 算法 &amp; 实现

```

while (rand() & 1) { // 轮询碰运气，若碰运气再长命，则先找出不低于比高度的...
    while (qlist->data->valid(p) || p->above) p = p->pred; // 跳至前驱
    if (!qlist->data->valid(p)) { // 跳过的链表 header
        if (qlist == first()) // 目前已是最高层，插入节点必须
            insertAsFirst( new Quadrant<Entry<K, V>> ); // 先进链表首，再
        p = qlist->pred->data->first()->pred; // 将p停靠上一层的header
    } else p = p->above; // 否则，可轻而易举提升查询速度
    qlist = qlist->pred; // 上升一层，并在该层停驻节点
    h = qlist->data->insertAfterAbove( s, p, h ); // 跳过p之后，h之上
} // while (rand() & 1)
return true; // SkipList允许重复元素，故插入必成功
    
```

## 横向跳转 / 邻接矩阵

• 那么：横向跳转操作可能重多次？比如  $\text{w}[logn]$ ，遍历  $\Omega(n)$ ？

• 理论：在同一水平列表中，横向跳转所经节点必然依次紧邻，而且每次抵达都是随机

• 于是：若相邻两横列表跳转的次数记作  $(Y)$ ，则有 **几何分布**：

$$\Pr[Y = i] = (1-p)^{i-1} \cdot p$$

• 推理： $E(Y) = (1-p)/p = (1-0.5)/0.5 = 1$  次

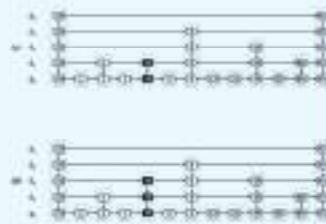
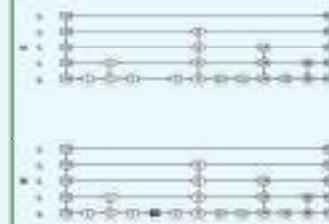
• 例题：列表中相邻的跳转节点，平均不过  $i+1 = \lfloor \log n \rfloor$  个

• 简论：查找过程中横向跳转所需时间 =  $\text{expected-}\delta(\lfloor \log n \rfloor) = \text{expected-}\delta(\log n)$

• 简记：跳转表的每次操作可在  $\text{expected-}\delta(\log n)$  时间内完成

## 案例

• 定制：一般  $(A, B, M)$ ，边缘  $(a, b)$



## 9. 字典

跳转表  
插入

如果一个人遇到不可解之事，把脑子想穿了，也找不到脚中的原因，怎么办呢？他或许会去盗版书店，把自己的难题交给菲尔先生，求人他们别摆布。

菲尔哥

[www.foolishgo.com](http://www.foolishgo.com)

## 9. 字典

跳转表  
删除

菲尔哥

[www.foolishgo.com](http://www.foolishgo.com)

## 算法 &amp; 实现

```

template <typename K, typename V> bool SkipList<K, V>::insert( K k, V v ) {
    Entry<K, V> * e = entry( K, V >> v ); // 增加新结点进入多个副本的跳跃表
    if (!empty()) insertAsFirst( new Quadrant<Entry<K, V>> ); // 首个Entry
    ListNodes<Quadrant<Entry<K, V>> >> qlist = first(); // 从顶部列表的
    Quadrant<Entry<K, V>> * p = qlist->data->first(); // 首节点开始
    if (!skipSearch( qlist, p, k )) // 跳过语句的插入位置——若已有相同词条，则
        while (p->below) p = p->below; // 跳到特别暗黑
    qlist = last();
    Quadrant<Entry<K, V>> * h = qlist->data->insertAfterAbove( s, p );
    /* ... 以下，至于s的右侧，以该节点h为基准，四维向上逐级长出一座座塔 ... */
    
```

## 算法 &amp; 实现

```

// 插入的过程

template <typename K, typename V> bool skipList<K, V>::remove( K k ) {
    if (!empty()) return false; // 空集
    ListNodes<Quadrant<Entry<K, V>> >> qlist = first(); // 从顶部Quadrant
    Quadrant<Entry<K, V>> * p = qlist->data->first(); // 首节点开始
    if (!skipSearch( qlist, p, k )) // 目标词条不存在，则
        return false; // 跳过语句
    /* ... */
    
```

```

do { //若目标节点存在，删除该节点并返回该结果
    if(list->data->val == k) {
        lower = p->below; //记住下一层节点
        list->data->remove(p); //删除当前层的节点后，再
        p = lower; qlist = qlist->succ; //转入下一层
    }
} while (!qlist->succ); //以上不断推移，直到顶层
while (!empty() && first()->data->empty()) //逐一地
    list->remove(first()); //清空已可能不含可删的顶层qalist
return true; //删除操作成功完成
}
//体会：得益于哨兵的设置，删除环节被简化了？

```

```

void set(int k) { expand(k); } //k > 31 || 0x80 >= k & 0x07) && 1
void clear(int k) { expand(k); } //k > 31 || k - 0x80 >= k & 0x07) && 1
bool test(int k) { expand(k); return k < 31 || 0x80 >= k & 0x07) && 1
void expand(int k) { //Bitmap[k]边界时扩容，分摊O(1)
    if (k < 0 || k > 31) return; //0x80在界内，无需扩容
    int oldM = M; char * oldM = M;
    M = new char[8 + ((k + 8) / 8)]; //与向量类似，加倍策略
    memory_S(M, M, oldM, oldM); delete [] oldM;
}

```

## 9. 字典

位图

数据结构

邓佳辉

deng@tsinghua.edu.cn

## 9. 字典

位图

典型应用

邓佳辉

deng@tsinghua.edu.cn

◆集合，也可操作一种抽象数据类型  
◆这里不过考虑其中的特例——整数集  
◆枚举类的思想推广，所有离散集  
或者整式类本身就是整数集  
注意离式类可转换为整数集

```

void set( int k ); //将整数k加入整数集合
void clear( int k ); //从整数集合中删除整数k
bool test( int k ); //判断整数k是否属于当前集合

```

◆如何具体实现？看接口的效率如何？

◆老问题：int A[ 31 ] 的元素均取自 [ 0, 31 ]，如何删除其中的重叠部分？  
◆使用vector::default\_delete()改进版：先排序，再扫描 —— O(nlog n + n) —— 基本无压力  
◆新特点：数据量很大，但单组内存高 —— 想想我们和脑里的MP3，不难理解  
◆比如， $2^{24} \approx 16 \times 10^6$  个 24 位无符号整数  
亦即，16,000,000 个 24 位无符号整数  
◆如果采用内部排序方法，你至少需要  $4 \times 31 = 488$  内存  
——否则，简单的 I/O 将导致整体效率的低下  
◆那么， $m < n$  的条件，又该如何加以利用？

◆使用物理地址连续的一段空间，各元素依次对应一个比特位

```

class Bitmap {
    private:
        char * M; //以char(8比特)为单位的比特位图
        int N; //位图长度(以sizeof(char)为单位)
    public:
        Bitmap( int n = 8 ) { M = new char[ N = (n+7)/8 ]; memset( M, 0, N ); }
        ~Bitmap() { delete [] M; M = NULL; }
        void set( int k ); void clear( int k ); bool test( int k ); //ADT
}

```

◆Bitmap( 8 ); //0x00  
for ( int i = 0; i < 8; i++ ) B.set( A[i] ); //O(8)  
for ( int k = 0; k < 31; j++ ) if ( B.test( k ) ) /\* ... \*/; //O(31)  
◆总体运行时间 = O(8 + 31) = O(39)  
◆空间 = O(8) —— 就上例而言，降低：n/8 + 2^21 = 240 < 4800  
即使 n = 2^32，也不过：2^29 = 536  
◆拓展：搜索引擎的使用频率亦是如此，词表规模不大，但密度极高  
——如何从中剔除重复的索引词？  
◆关键在于，如何将查询语义转换为某一集合——操作习题

## 算法

1097

◆如何计算出 $(a, n)$ 之间的所有质数？

◆与其说是计算，不如说是质数筛选。

```

void printPrimes( int n, char * file ) {
    Bitset B[ n + 1 ]; // 判断位置
    B.set( 0 ); B.set( 1 ); // 0和1都不是质数
    for ( int i = 2; i < n; i++ ) // 从第一位开始，从下一
        if ( !B.test( i ) ) // 可认定的质数（粗
            for ( int j = 2 * i; j < n; j += i ) // 以j为周期
                B.set( j ); // 将下一个数标记为合数
    B.dump( file ); // 将筛选出的质数输出至文件，以便日后使用
}

```

[Java 8 Stream API 算法 | 第九章 筛选](#)

## 改进

◆循环起始 $i + 1$ ，可进一步改写 $i + i$ 。

——为什么？

◆如此，每次内循环的长度，由

$\sigma(n/2)$

降至

$\sigma(\max(1, n/2 - 1))$

◆效率有所提高！

——从原书的角度看，是否实施改进？

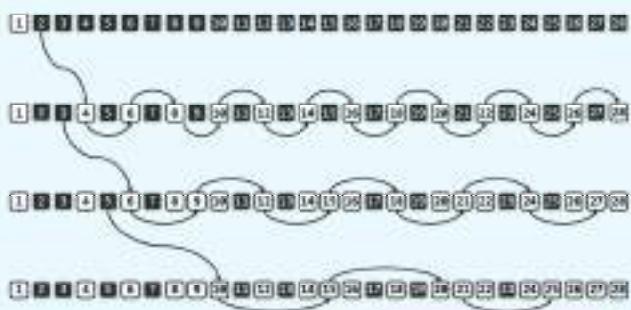
◆基于以上，如何实现primeLT(int low)？

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25
26	27	28	29	30

1101

## 实例

1098

[Java 8 Stream API 算法 | 第九章 筛选](#)

1102

## 9. 字典

位图

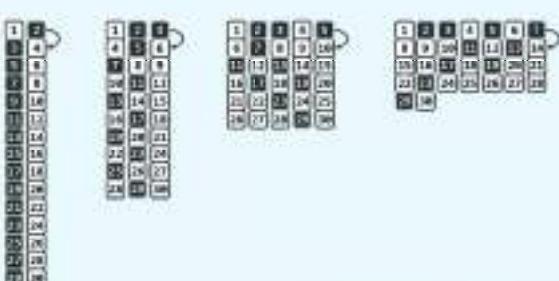
快速初始化

## 最佳解

[http://tiny.cc/meyarwadde](#)

## 实例

1099

[Java 8 Stream API 算法 | 第九章 筛选](#) $\sigma(n) \sim \sigma(1)$ 

◆Bitmap的构造函数中，通过newSet( N, 0, N )统一清零

这一步只用 $\sigma(1)$ 时间？不，实际上仍等效于高位清零， $\sigma(N) = \sigma(n)$ ！

◆尽管这并不会影响上面的渐进复杂度，但并非所有函数都是如此

◆有时，对于大规模的数据源，初始化的速度直接影响到实际性能

例如：遍历单个`int`类型的向量时，需要 $\sigma(|I| * n) = \sigma(|I| * N)$ 时间  
而当直接对`int`类型的初始化，则可严格地保证是 $\sigma(|I|)$ 

◆有时，甚至会影响到算法的整体渐进复杂度

例如，为从 $n = 10^{10}$ 个`int`型整数中找出质数者，可用的质数算法... // 相比要慢很多  
因此，若能省去Bitmap的初始化，则只要 $\sigma(n)$ 时间

1103

## 效率

1100

◆不计内循环，外循环自身每次仅一次加法。两次判断，累计 $\sigma(n)$ ◆内循环遍历迭代 $\sigma(n/i)$ 步，由素数定理(更多 $n/\ln(n)$ )知，累计耗时不过

$$n/2 + n/3 + n/5 + n/7 + n/11 + \dots$$

$$< n/2 + n/2 + n/4 + n/3 + n/6 + \dots + n/\lceil n/\ln(n) \rceil$$

$$= \sigma(n * \lfloor n/\ln(n) \rfloor) - \sigma(1)$$

$$= \sigma(n\ln(n)) - \sigma(n)$$

$$= \sigma(n\log(n))$$

[Java 8 Stream API 算法 | 第九章 筛选](#)

## 算法

◆// [J. Hopcroft, 1974] 为 $n = 1$ 增加一对等长的`top`型指针Rank`[k = 1]`; // 向量`front`Rank`[T = 1]`; // 向量`tail` toRank`.top = 0;` // `tail`的极限// 指示空闲链的首结点为`0(n)`◆bool Bitmap::test( int k ) {  $\sigma(1)$  }{ return `B <= T[k]` || `BB[T[k]] < top` || `BB[T[k]] == k`; }◆void Bitmap::set( int k ) {  $\sigma(1)$  }{ if ( !B.test( k ) ) { `T[top] = k`; `B[k] = top++`; } }◆void Bitmap::clear( int k ) { /\* 为简化，习题[2-34] \*/ } //  $\sigma(1)$ 

1104

[Java 8 Stream API 算法 | 第九章 筛选](#)

## 9. 字典

MD5

以小见大，见一时落霞知岁之将暮。  
触瓶中之冰，知天下之寒。

单佳群

daogt@zjhu.edu.cn

## 散列指针

◆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？相抵冲突！

I am Lord Voldemort = Tom Marvolo Riddle = He's Harry Potter

◆问题：序列越多越好吗？

技巧：在足以覆盖文档的前提下，如何选取最少的序列？

◆单向函数： $f(x)$  可快速计算，但  $f^{-1}(y)$  的计算却十分...十分地耗时◆散列算法： $f(x, y) = x * y \mod 10^9 + 7$  —— RAR 算法 $f^{-1}(x * y) = \alpha(x, y) \mod 10^{18}$  —— 散列校验

Data Structures &amp; Algorithms, Single-Minded

MD5

◆网站下载：除了数据本身，你是否还注意到对应的 MD5？

MD5 有何作用？

```
< bash : /etc/passwd
$ cat /etc/passwd
root:x:0:0:Super User:/bin/sh
daemon:x:1:1:FTP user:/usr/spool/ftp:
nobody:x:65534:65534:Student:/bin/csh
...

```

◆**password**，是如何计算出来的？如何验证的？反过来，会解密原文吗？

## 算法

◆Message-Digest Algorithm version 5, 1991

MIT Lab For Computer Science &amp; RSA Data Security Inc.

将消息块一分为二块情况，即  $512 = 16 \times 32$  bit 作为一个处理分组经过四轮迭代运算，最终得到一个  $128$  bit 的大整数，即为 MD5 的◆初始化：就有必要，添加一个 1 及多个 0，使总长为  $512+8 = 64$ 将原消息的长度附加到末尾，使总长为  $512+8$ ◆ $128$  bit，足够复杂？ $2^{128} = 3.4 \times 10^{38}$ 

数据块（碰撞）符合侧碰指纹的初始值，似乎不那么理想...

## 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 冲突的原因，则意味着什么？

◆了解 MD5 安全性的最深结论 //google("MD5 'SHA-1'")

◆提交作业时，尝试同时提交 MD5 指纹

◆了解分布式哈希列表的

原理、方法与 //Overlay Network, Distributed Hash Table

实践 //Napster, Gnutella; Chord, CAN, Tapestry, Pastry, ...

## 课后

◆在很多场合，MD5 都可大显身手...

◆数字签名：授权文档一旦篡改就失效 //电子形式签署的法律文书

◆身份验证：QQ 或应用软件，根据口令做出可授权 //充分性不必

◆隐私通知：“你知道我是谁，别人不知道我是谁” //手机号码

◆资源共享：文档之间的（局部、逻辑）复制关系 //学术论文链接

◆痕迹检验：带宽有限时，快速确认大致相同一致性 //分布式存储、云存储

◆病毒检测：系统文件是否被病毒修改？被何种病毒修改？ //特征

...  
MD5

## 10. 优先级队列

概述

需求与动机

I cannot choose the best.

The best chooses me.

单佳群

daogt@zjhu.edu.cn

## 夜间门诊



1113

## 多任务调度



1114

## 优先级队列

```


template <typename T> struct PQ { //priority queue
    virtual void insert(T) = 0; //插入优先级次序进入队列
    virtual T getMax() = 0; //取出优先级最高的元素
    virtual T delMax() = 0; //删除优先级最高的元素
};

//与其说PQ是数据结构的，不如说它是ADT；不同的实现方式，效率及适用场合也各不相同

//Stl和queue_，都是PQ的特例——优先级完全取决于元素的插入次序

//Stack和queue_，也是PQ的特例——插入和删除的后进先出


```

1117

## 18. 优先级队列

概述

基本实现

推荐阅读

<http://www.iteye.com/topic/1000000>

## 应用、算法与特点

- 应用：高效率并行处理  
操作系统：任务调度、中断处理、GUI的队列...  
输入法：词典树是
- 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  
平衡二叉树中的事件队列  
...
- 数据元素：动态变化、快速地定位  
集合组成：可动态变化  
元素优先级：可动态变化

1115

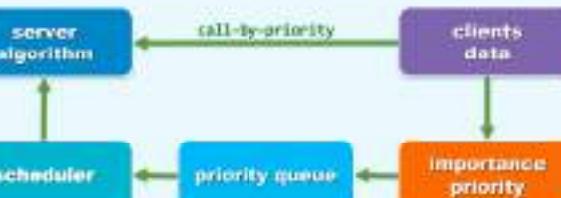
## Vector



getMax()	delMax()	insert()
$\Theta(n)$	$\Theta(n) + \Theta(n) = \Theta(n)$	$\Theta(1)$
traverse()	remove(traverse(), i)	insertAtLast(e)

1119

## 问题模式



1116

## Sorted Vector



getMax()	delMax()	insert()
$(n - 1)$	$(n - 1)$	$\Theta(\log n) + \Theta(k) = \Theta(n)$

1120

## Java面试题

List



getMax()	delMax()	insert()
traverse()	remove(traverse())	insertasFirst(e)
$\Theta(n)$	$\Theta(n) + \Theta(1) = \Theta(n)$	$\Theta(1)$

Data Structures &amp; Algorithms, Python Edition

## 10. 优先级队列

完全二叉堆  
结构进阶题：“何人问乱石堆？”  
如何乱石堆中有杀气冲天？

解法

www.ChinaSource.cc

Sorted List

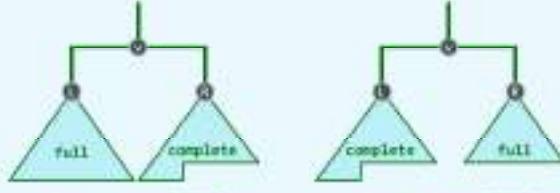


getMax()	delMax()	insert()
first()	remove(first())	insert(search(e), e)
$\Theta(1)$	$\Theta(1)$	$\Theta(n) + \Theta(1) = \Theta(n)$

Data Structures &amp; Algorithms, Python Edition

完全二叉树

◆ Complete Binary Tree：单根因子外的完全的AVL，而且...



BBST

+ AVL, Splay, Red-black：三个接口均只要  $\Theta(\log n)$  时间

但是，BBST的功能远远超出了PQ的需求...

- +  $EQ = 1 \times insert() + 0.5 \times search() + 0.5 \times remove() = \frac{1}{2} \times BBST$
- + 若只需要插入元素，则不必维护所有元素之间的全序关系，偏序足矣
- + 因此有理由相信，存在某种更为简单、相对成本更低的实现方式
- + 需要自功能接口，时间复杂度依然为  $\Theta(\log n)$ ，而且
- + 实际效率更高
- + 当然，既具体情况而言，这类实现方式已属最佳——为什么？

Data Structures &amp; Algorithms, Python Edition

结构性

◆ 逻辑上，等同于完全二叉树

物理上，直接借助向量实现

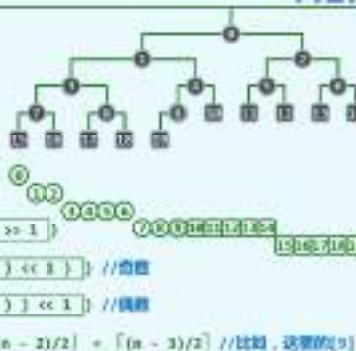
◆ 节点节点与物理元素

物理次遍历次序依次对应：

#define Parent(i) ((i - 1) &gt;&gt; 1) //父节点

#define LChild(i) ((i + (1) &lt;&lt; 1)) //左孩子

#define RChild(i) ((i + (1) ) &lt;&lt; 1) //右孩子

◆ 共  $n$  个节点时，内部节点的最大数 =  $\lceil (n - 2)/2 \rceil + \lceil (n - 1)/2 \rceil$  //比如，这颗的 9

统一测试

```
#template <typename T> void testBBST( int n ) {
    T A = new T[ 2 * n / 3 ]; //创建容量是2n/3的数组，并
    for ( int i = 0; i < 2 * n / 3; i++ ) A[i] = dice( (T) 2 * n ); //随机化
    PQ heap( A + n / 6, n / 3 ); delete [] A; //Robert Floyd
    while ( !heap.size() < n ) //随机测试
        if ( dice( 100 ) < 70 ) heap.insert( dice( (T) 2 * n ) ); //70%概率插入
        else if ( !heap.empty() ) heap.delMax(); //30%概率删除
    while ( !heap.empty() ) heap.delMax(); //清空
}
```

Data Structures &amp; Algorithms, Python Edition

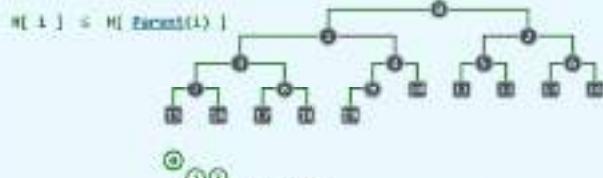
PQ\_complHeap = PQ + Vector

```
#template <typename T> class PQ_complHeap : public PQ<T>, public Vector<T> {
protected:
    Rank percolateDown( Rank n, Rank i ); //下沉
    Rank percolateUp( Rank i ); //上升
    void heapify( Rank n ); //Floyd建堆算法
public:
    PQ_complHeap( T* A, Rank n ); //构造构造
        { copyfrom( A, 0, n ); heapify( n ); }
    void insert( T ); //根据比较器确定的优先级次序，插入问题
    T getMax(); //返回优先级最高的问题
    T delMax(); //删除优先级最高的问题
}
```

33

## 堆序性

1129

△数据上，只要  $i < 1$ ，必满足△根  $\leq i \leq n$  都服从最大元素

```
template <typename T>
T PQ_CompilHeap<T>::getMax() { return _elem[0]; }
```

## 实现

1133

```
template <typename T> void PQ_CompilHeap<T>::insert(T e) //插入
{ Vector<T>::insert(e); percolateUp(_size + 1); }
```

```
template <typename T> //对第i个词条实施上浮,  $i < _size$ 
Rank PQ_CompilHeap<T>::percolateUp(Rank i) {
    while (ParentValid(i)) { //只要i有父亲(尚未抵达根部), 则
        Rank j = Parent(i); //将i之父操作
        if (_elem[i] < _elem[j]) break; //一旦父子不再逆序, 上浮即需停止
        swap(_elem[i], _elem[j]); i = j; //否则, 交换父子位置, 并上升一层
    } //while
    return i; //返回上浮终止的祖先
```

## 10. 优先级队列

完全二叉堆  
插入

时还看见土路南后一棵大柏树，便把两只腿夹住，一节节地搭上去架头顶，骑马儿坐在枝柯上。

李佳群  
[long@tinghuawx.com](mailto:long@tinghuawx.com)

## 算法

1131

△为插入词条e，只需将e作为末元素插入内部

//结构性自然保持

//若堆序性也未被破坏，则完成

△否则 //只能是e与父节点违反堆序性

→与其父节点换位 //若堆序性因此恢复，则完成

△否则 //依然只能是e与其(新的)父节点...

→再与父节点换位

△不断重复...直到

→与其父亲恢复堆序性，或者

→到达堆顶(没有父亲)



## 10. 优先级队列

完全二叉堆  
删除

I have scaled the peak and

found no shelter in

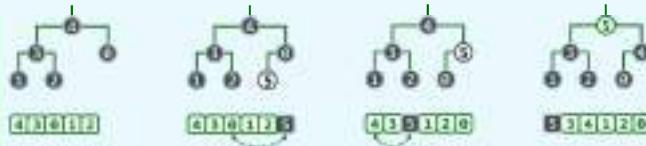
Fame's bleak and barren height.

李佳群

<http://long@tinghuawx.com>

## 实现

1132



## 算法

1136

△最大元素始终在堆顶，故为删除之，只需...

△删除后置首元素，代之以末元素

→结构性保持：若堆序性依然保持则清理

△否则 //因e与孩子们的违背堆序性

→与孩子中的大者换位

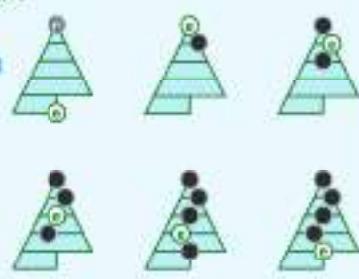
→若堆序性因此恢复，则完成

△否则 //因e与具有(新的)孩子的们...

→再次与孩子中的大者换位

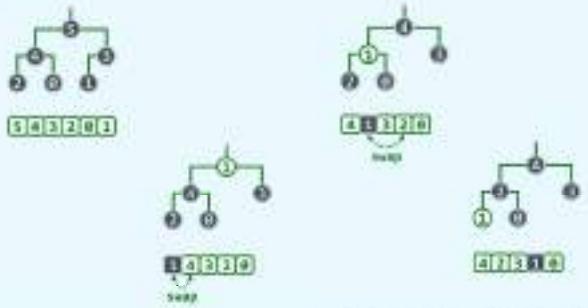
△不断重复...直到

→满足堆序性，或者已是叶子



实例

1137



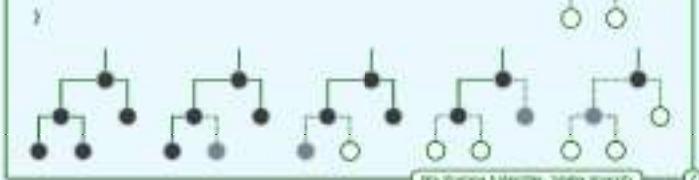
自上而下的上堆

1141

```

◆ PQ_CompHeap<T> A; Rank n = 3; if copyFrom( A, 0, n = 11 ); heapify( 6 ); //如何实现?
◆ template <typename T> void PQ_CompHeap<T>::heapify( Rank n ) { //暴力
    for ( int i = n - 1; i < n; i++ ) //按秩(此次遍历)次操作
        percolateUp( i ); //将上述插入的节点
}

```



实操

1138

```

◆ template <typename T> T PQ_CompHeap<T>::delMax() { //删除
    T maxItem = _elem[0]; _elem[0] = _elem[--_size]; //摘掉堆顶，代之以末尾项
    percolateDown( _size, 0 ); //对剩余项实施下堆
    return maxItem; //返回此前标记的最大值
}

◆ template <typename T> //对前n个元素中的第i个实施下堆, i < n
Rank PQ_CompHeap<T>::percolateDown( Rank n, Rank i ) {
    Rank j; //i是j(至多两个)孩子中, j是父
    while ( 1 != ( j = ProperParent( _elem, n, i ) ) ) //只要i的j, 则
        swap( _elem[i], _elem[j] ); i = j; //换位, 并继续考察
    return i; //返回下堆起始的位置(亦即i)
}

```

Data Structures &amp; Algorithms: Design &amp; Analysis

效率

1142

极端情况下

每个节点都向上递进

时间成本正比于其深度

即使只考虑底层

n/2个叶节点, 深度均为 $\Theta(\log n)$ 总计耗时 $\Theta(n \log n)$ 

这样长的时间, 本足以坚持住!

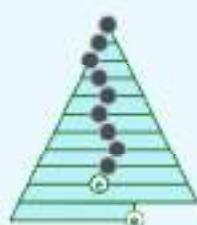


应该, 能够更快的...

效率

1139

在下述的过程中  
每经过一次交换  
n的高度都降低一级  
故此, 在每层至少需要一次交换  
两次后, 由于堆顶尚余, 故耗时为 $\Theta(\log n)$   
结论: 通过下堆, 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: Design &amp; Analysis

自下而上的下堆

1143

往堆顶进堆  $H_0$  和  $H_1$ , 以及节点 p为根的堆  $H_0 \cup \{p\} \cup H_1$ , 只需将  $H_0$  和  $H_1$  当作 p 的孩子, 对 p 下堆

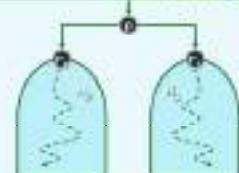
◆ template &lt;typename T&gt;

void PQ\_CompHeap&lt;T&gt;::heapify( Rank n ) { //Robert Floyd, 1964

for ( int i = LastInternal( n ); i = i - 1; ) //从下而上, 依次

percolateDown( n, i ); //下堆各内部节点

} //可理解为子堆的逐级合并——由以上推导, 先序性遍历必然在全是空堆



实例

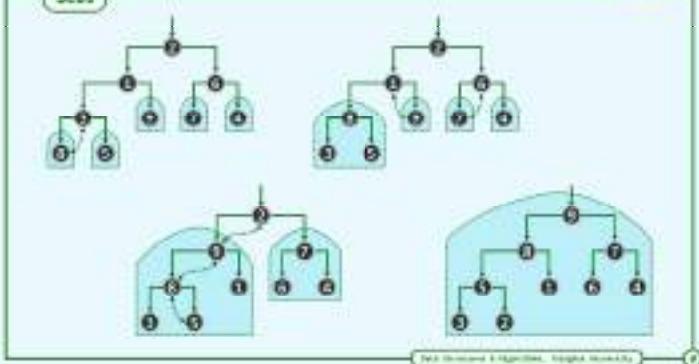
1140

## 10. 优先级队列

完全二叉堆  
数组堆单链表  
[algomaster.csail.mit.edu](http://algomaster.csail.mit.edu)

实例

1144



Data Structures &amp; Algorithms: Design &amp; Analysis

效率

1145

每个内部节点所用的调整时间，正比于其高度的非深浅

不考虑一般性，考虑满树： $n = 2^{d+1} - 1$  $S(n) = \text{所有节点的高度总和}$ 

$$= \sum_{i=0, i \neq d} (0 + 1) \times 2^i$$

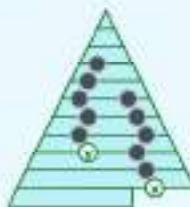
$$= d \times \sum_{i=0, i \neq d} 2^i = T(n)$$

$$= d \times (2^{d+1} - 1) - [ (d + 1) \times 2^{d+1} - 2 ]$$

$$= 2^{d+1} - (d + 2)$$

$$= n - \log_2(n + 1)$$

$$= \Theta(n)$$



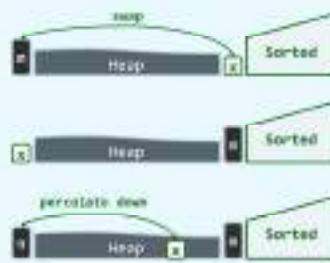
Data Structures &amp; Algorithms, Integer Heaps v1

1149

就地

物理上

完全二叉堆 增删 取数



自然此有：

$$x = H[0]$$

$$x = S[-1]$$

不如这样：

$$\text{swap}(n, x) = H.\text{insert}(x) + S.\text{insert}(x)$$

Data Structures &amp; Algorithms, Integer Heaps v1

课后

1146

在 insert()：循环情况下效率为  $\Theta(\log n)$ ，平均情况呢？

在 heapify()：根据次序被插入，为什么需要准备多样性地操作？

这一算法在哪些场合不适用？

扩壳操作：decrease(i, delta) //在一元素 elem[i] 的数值减小 delta  
increase(i, delta) //在一元素 elem[i] 的数值增加 delta  
remove(i) //删除任一元素 elem[i]调整完全堆，在  $\Theta(\log n)$  时间内构造 Huffman 树在大数组中，delMin() 操作是否能在  $\Theta(\log n)$  时间内完成？

难道，为此需要对维护一个小区间？

1147

## 10. 优先级队列

堆排序

堆结构

http://tiny.cc/meyarw

选取

1148

在 selectionSort() 中...



+ 相 U 列为 H

+ J. WILLEMS, 1960

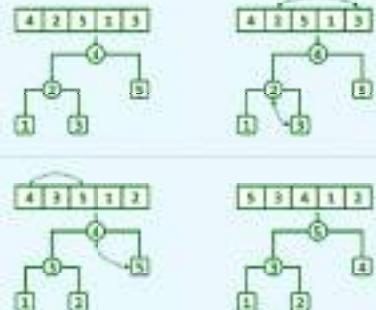
初始化： heapify(),  $\Theta(n)$ , 建堆迭代： delMax(),  $\Theta(\log n)$ , 取出堆顶并调整数据不变性：  $i \leq n - 5$ 

+ 等效于带堆选择排序，正确无误

+  $\Theta(n) + n \times \Theta(\log n) = \Theta(n \log n)$ 

实例：建堆

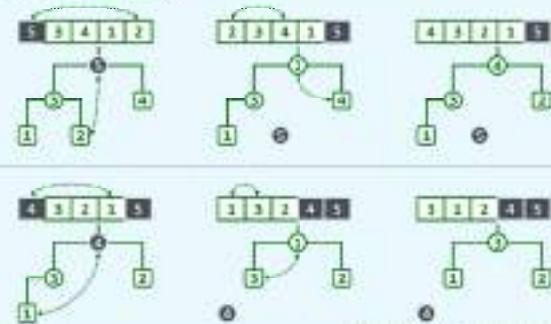
1151



Data Structures &amp; Algorithms, Integer Heaps v1

实例：选取 + 调整

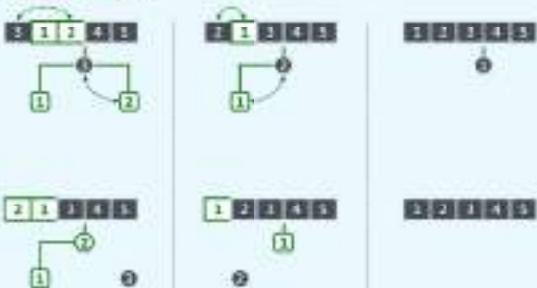
1152



Data Structures &amp; Algorithms, Integer Heaps v1

## 实例：选取 + 调整

1153



## 算法

↑TournamentSort()

CREATE a tournament tree for the input list

while there are active leaves

REMOVE the root

RETRACE the root down to its leaf

DEACTIVATE the leaf

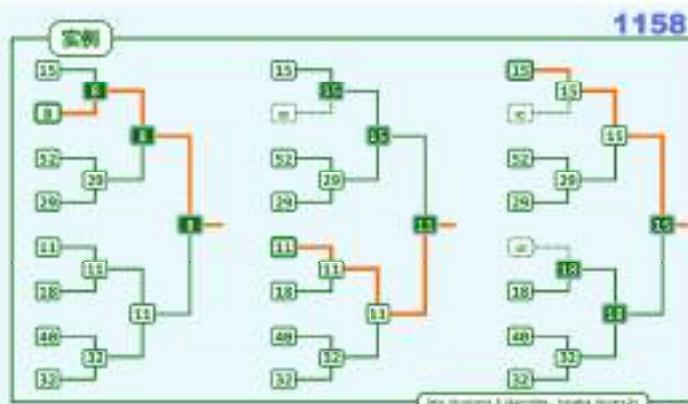
REPLAY along the path back to the root

1157

## 评价

1154

- 易于理解，便于实现 //完全基于二叉堆的插入及删除操作接口
- 快速高效 //尤其适用于大规模数据
- 可就地旋转
- 不完全排序即可找出前k个词条 // $O(k\log n)$  的selection算法
- 不稳定 //为什么？何曾克服？
- 均衡： 平用就地策略，是否值得？
- 虽然可以节省一定的空间
- 但对操作来说必须涉及两个完整的树条，操作的是位或不增加

1155  
10. 优先队列锦标赛排序  
锦标赛树

老奶奶：“怎么叫做分而治之？”

小姑娘：“如今把拥中大小横纵，点快起来，干中选百，四中选十，十中只选三个...”

邓桂科

dengguikao@163.com

1159

## 效率

↑空间： $\Theta(\text{节点数}) = \Theta(\text{叶节点数}) = \Theta(n)$ ↑扫描：仅需  $n(n)$  时间

↑更新：每次数据全体重置(replay)？

谁上一代胜者(祖先)，才有必要参加！

↑为此：只要从所处叶节点出发，逐层上升直到根

如此：为确定各轮优胜者，总共所需时间仅  $\Theta(\log n)$ ↑时间： $n^2 = \Theta(\log n) = \Theta(n \log n)$ ，达到下界

## 锦标赛树

1156

↑ 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：小数为胜

↑create() // $\Theta(n)$ remove() // $\Theta(\log n)$ insert() // $\Theta(\log n)$ 

↑时间：总用全场冠军——类似于最小堆

## 选取

↑从n个元素中提出最小的k个， $\underline{k < n}$  //即它第1章的max2()，扩展  $k > 2, \underline{k < n}$ ↑锦标赛锦标赛： 初始化： $\Theta(n)$  // $n - 1$ 次比较选择k步： $\Theta(k \log n)$  //即它1章的比较

与小顶堆相似？

↑渐进意义上，的确如此

但就常数而言，区间不小...

↑ Floyd算法中的percolateDown()，在每一层需要 $\Theta$ 次比较，累计最少于  $\Theta(n)$  次

deleteMax()中的percolateDown()，亦是如此

1160

## 10. 优先级队列

锦标赛排序  
堆树

树结构

http://tinyurl.com/3qyqjwz

## 优先级搜索

◆ 困惑图的优先级搜索以及的一框架：`a->push()`...◆ 无论何种算法，差异仅在于所采用的优先级更新器 `PriorityUpdate()`Prim算法：`a->push( 0, PrimEU() );`Dijkstra算法：`a->push( 0, DijkstraEU() );`

◆ 每一节点引入进优先队列，都调用

【图斯】用外层点的优先级（粗），得

【图出】原的优先级是浅青

◆ 若果每条搜索，两类操作的累计时间，分别为  $O(n + e)$  和  $O(n^2)$ 

◆ 哪个更快呢？

http://tinyurl.com/3qyqjwz, http://tinyurl.com/3qyqjwz

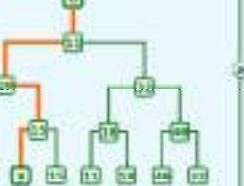
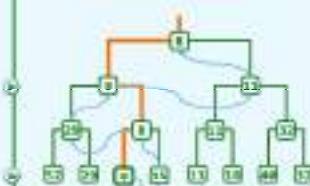
## 堆树

◆ 重置过程中，除了沿途各节点，还需要将所有兄弟

//如何避免这类问题？

◆ [inner tree]：内部节点，记录对应比赛的~~较弱~~

锦标赛的“父节点”，记录冠军



http://tinyurl.com/3qyqjwz, http://tinyurl.com/3qyqjwz

## 优先级队列

◆ 直接地，PQ中的元素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 $O(n)$ `delMax():` 顶层优先级最高（强项）删除( $v, w$ ) 总计 $O(\boxed{n} + \log n)$ `increase():` 更新所有关联顶点到 $v$ 的权值，提高优先级 总计 $O(\boxed{n} + \log n)$ ◆ 总体运行时间为  $O((n + e) + \log n)$ 

对于升强图，处理效率很高；对于降强图，是否不如常规实现的版本

◆ 没有更好的办法？如果PQ的接口效率足够高的话...

◆ 不太现实？异想天开？不妨先试试...

## 课后

◆ 可用锦标赛树，可否实现稳定的排序？

◆ 利用锦标赛树，如何实现多路归并？

http://tinyurl.com/3qyqjwz, http://tinyurl.com/3qyqjwz

## 多叉堆

◆ `heapify(): O(n)` 不可能再快了 //直接可入，你不过如此`delMax(): O(log n)` 实际就是`percolateDown()` //已是根换了`increase(): O(log n)` 实际就是`percolateUp()` //似乎仍有余地

## 10. 优先级队列

多叉堆

树结构

http://tinyurl.com/3qyqjwz

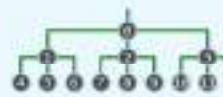
## 多叉堆

◆ 将二叉堆改成多叉堆 ( $d$ -heap)判堆底障壁  $O(\log n)$ 

◆ 上山爬坡下山冲：

上坡成本可降低  $\log_2 d$ ，但

下坡成本却提高

 $d + \log_2 d + (d + \lceil \log_2 d \rceil + 1) \cdot \log_2 n$ 

◆ 对于弱强图，两类操作的次数相差甚殊——没弱利大于我...

http://tinyurl.com/3qyqjwz, http://tinyurl.com/3qyqjwz

## 多叉堆

1169

如图，PFS的运行时间是 $\lceil n/d \rceil \log n + c \cdot \log n = (n/d + c) \cdot \log n$

再稍权衡，大概取  $d = n/n+2$  时

总体性能达到最优的  $O(c \cdot \log_{n+2} n)$

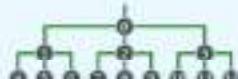
对于 **插入** 而深保修改：

$$c \cdot \log_{n+2} n \approx n \cdot \log_{n+2} n \approx O(n \log n)$$

对于 **删除** 而深保修改：

$$c \cdot \log_{n+2} n \approx n^2 \cdot \log_{n+2} n \approx n^2 = O(n)$$

对于一般的插入，会直接地实现最优



Data Structures & Algorithms, Chapter 16 (16.9)

## 多叉堆

1170

实现方面，依然可以基于内部

`parent(k) = ⌈(k - 1) / d⌉`

`child(k, i) = kd + i, 0 < i < d`

当然， $d$ 不再限于2的幂时

将不再能够借助多位加进位的换取

不过反过来，特别适用于

不主要解决操作子耗较高效率的场合

比如，数据规模大到需要经常存取最近次时 // 相比上，与d的完全一致



Data Structures & Algorithms, Chapter 16 (16.10)

## 堆合并

1173

`H = merge(A, B)`：将堆A和B合为一 // 不妨设  $|A| = n \geq |B|$

方法一：`A.insert( B.delMax() )`

$$\delta(n) = (\lceil \log n + \log(n+m) \rceil)$$

$$= \delta(n + \log(n+m))$$



方法二：`union(A, B).heapify(n+n)`

$$\delta(n+n)$$

有没有更好的办法？比如...

可惜受限于...`\log(n)`...时间内实现`merge()`？



Data Structures & Algorithms, Chapter 16 (16.11)

## Fibonacci堆

1171

在底堆  $x$  (新的上或最近 + 剩余合并)

直接接口的分摊复杂度

`delMax()`  $\Theta(\log n)$

`insert()`  $\Theta(1)$

`merge()`  $\Theta(1)$

`increase()`  $\Theta(1)$

于是，基于PFS的算法采用Fibonacci堆后，运行时间自然满足

$$n \cdot O(\log n) + c \cdot O(1) = O(n + n \log n)$$

Data Structures & Algorithms, Chapter 16 (16.11)

## 空节点路径长度

1175

引入所有的外部节点

消除一度节点

转为满二叉树

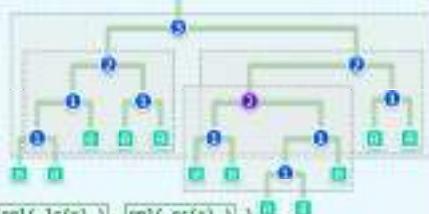
WLL Path Length

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

$$1) \text{ wpl}( x ) = 1 + \min( \text{wpl}( \text{lc}(x) ), \text{wpl}( \text{rc}(x) ) )$$

结论：`wpl(x) = x到叶节点的最近距离`

`wpl(x) = 以x为根的最大子树的高度`



Data Structures & Algorithms, Chapter 16 (16.12)

God's right hand is gentle  
but terrible is his left hand

君子以尚柔，用兵尚勇

古事记左，凶事记右  
他将军弱左，上将军强右

## 16. 优先级队列

左式堆  
结构

单链表  
[tiny.cc/meyarw](http://tiny.cc/meyarw)

## 左属性 &amp; 左式堆

1176

左属性：对任何内节点  $x$ ，都有 `mp1([lc(x)]) > mp1([rc(x)])`

结论：对任何内节点  $x$ ，都有 `mp1([l]) = [l] + mp1([rc(x)])`

满足在满足 **leftist property** 的堆，即是 **左式堆**

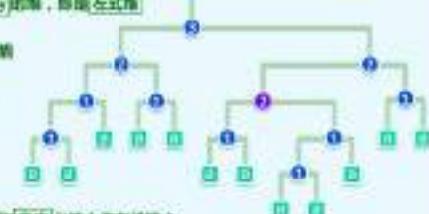
左属性与顺序性，相容而不矛盾

左式堆的子堆，必须左式堆

左式堆倾向于

新节点分布于左侧分支

这是合理的吗，左子堆的规模和高度必然大于右子堆？



Data Structures & Algorithms, Chapter 16 (16.13)

右树链

1177

◆ `rChain(s)`: 从节点s出发，一直沿右分支推进◆ 指明数，`rChain(root)`的终点，必为全树中最浅的外部节点`npl(r) = ⌈ rChain(r) ⌉ - d`

高度为d的子树树

◆ 右侧链长为d的左式堆，至少包含

[ $2^d - 1$ ]个内部节点， $[2^{d+1} - 1]$ 个节点

◆ 反之，在仅有n个节点的左式堆中

右侧链的长度  $d \leq \lfloor \log_2(n + 1) \rfloor - 1 = d(\log_2)$ 

实例

◆ `template <typename T>`

```
static BinNodePosi(T) merge( BinNodePosi(T) a, BinNodePosi(T) b ) {
    if ( ! a ) return b; // 假归零
    if ( ! b ) return a; // 假归零
    if ( lt( a->data, b->data ) ) swap( a, b ); // 一般情况：首先确保b不大于a
    a->rc = merge( a->rl, b ); // 将a的右子堆与b合并
    a->parent = a; // 重新设父子关系
    if ( ! a->rc || (a->rc)->rpl < (a->rc)->lpl ) // 若需必要
        swap( a->rl, a->rc ); // 交换位置。若子堆，以确保右子堆的rpl不大于lpl
    a->rpl = (a->rc) ? (a->rc)->rpl + 1 : 0; // 更新a的rpl
    return a; // 返回合并后的堆顶
}
```

1181

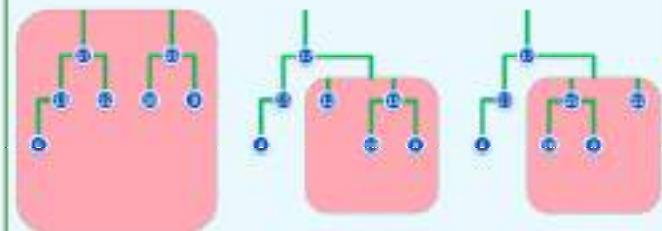
左之左之，右子右之  
右之右之，右子右之

## 10. 优先级队列

左式堆  
合并孙伟群  
[ding@tsinghua.edu.cn](mailto:ding@tsinghua.edu.cn)

1178

实例 (1/4)



LeftHeap

1179

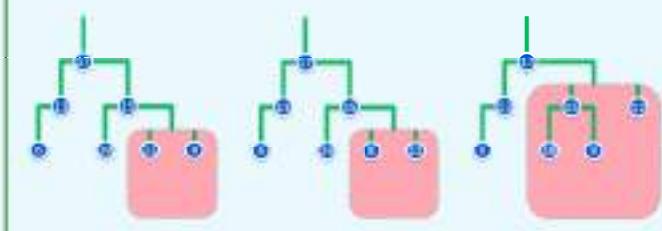
◆ `template <typename T>` // 基于二叉树，以左式堆形式实现的优先级队列

```
class PQ_LeftHeap : public PQ<T>, public BinTree<T> {
public:
    void insert(T); // (按比较器指定的优先级次序) 插入元素
    T getMax() { return _root->data; } // 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...
```

◆ `template <typename T>`

实例 (2/4)

1183



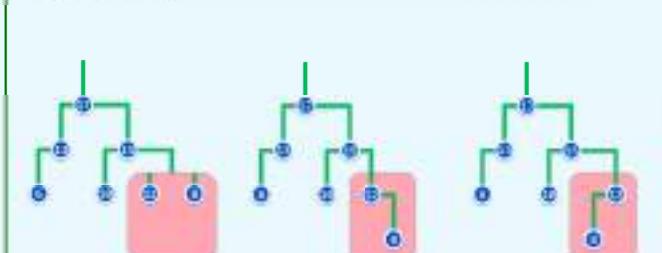
算法

1180



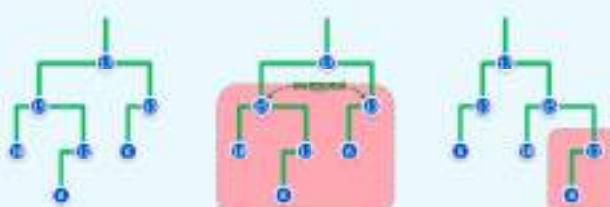
实例 (3/4)

1184



实例(4/4)

1185



Data Structures &amp; Algorithms - Height-balanced

AVL::merge()

1189

◆设 $T_1$ 和 $T_2$ 为两棵AVL树，且 $\max(T_1) < m = \min(T_2)$ 

如何便捷地将其合并为一棵AVL树？

◆ $height(T_1) \geq height(T_2)$ 

Data Structures &amp; Algorithms - Height-balanced

## 10. 优先级队列

左式堆  
插入与删除

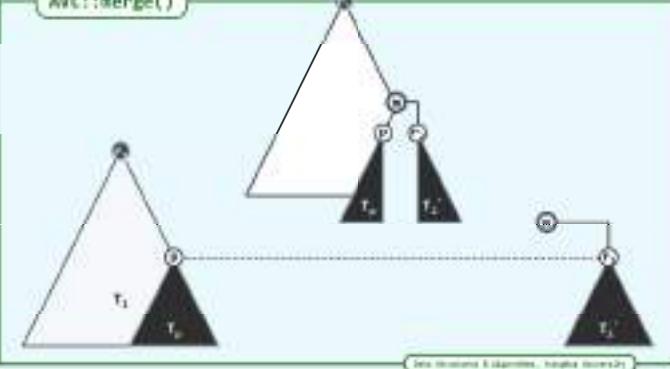
一指针，吃饱天

邓伟群  
deng@zjhu.edu.cn

1186

1190

AVL::merge()

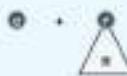


Data Structures &amp; Algorithms - Height-balanced

insert()

1187

```
#template <typename T>
void PQ_leftHeap<T>::insert(T v) { //O(logn)
    BinNodePosi(T) w = new BinNode<T>(< v); //为v创建一个二叉树节点
    _root = merge(_root, w); //通过合并将新节点的插入
    _root->parent = NULL; //既然此时非空，还需相应设置父子链接
    _size++; //更新规模
}
```



Data Structures &amp; Algorithms - Height-balanced

## 10. 优先级队列

优先级搜索树

邓伟群

deng@zjhu.edu.cn

1191

delMax()

1188

```
#template <typename T> T PQ_leftHeap<T>::delMax() { //O(logn)
    BinNodePosi(T) lHeap = _root->lC; //左子堆
    BinNodePosi(T) rHeap = _root->rC; //右子堆
    T t = _root->data; //输出堆顶处的最大元素
    delete _root; _size--; //删除根节点
    _root = merge(lHeap, rHeap); //原左、右子堆合并
    if (! _root) _root->parent = NULL; //更新父子链接
    return t; //返回堆顶节点的数据值
}
```



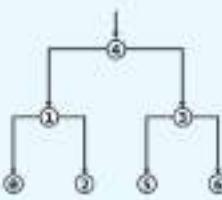
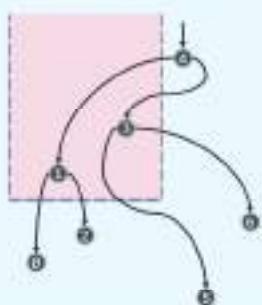
Data Structures &amp; Algorithms - Height-balanced

Priority Search Tree = BST + PQ

1192



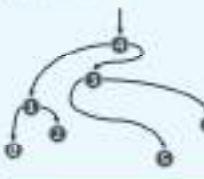
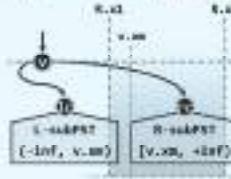
Data Structures &amp; Algorithms - Height-balanced



Data Structures &amp; Algorithms, Chapter 11

- 相等:  $S[0, n] = T[0, m]$   
长度相等 ( $n = m$ )，且对应字符均相同 ( $S[i] = T[i]$ )
- 子串:  $S.substr(i, k) = S[i, i+k], 0 \leq i \leq n, k \geq 0$   
表示从  $S[1]$  起的连续  $k$  个字符  $\boxed{S[1]} \dots \boxed{S[i+k]}$
- 前缀:  $S.prefix(k) = S.substr(0, k) = S[0, k], 0 \leq k \leq n$   
表示  $S$  中前  $k$  位的  $k$  个字符  $\boxed{S[1]} \dots \boxed{S[k]}$
- 后缀:  $S.suffix(k) = S.substr(n-k, k) = S[n-k, n], 0 \leq k \leq n$   
表示  $S$  中后  $k$  位的  $k$  个字符  $\boxed{S[n-k+1]} \dots \boxed{S[n]}$
- 截取:  $S.substr(i, k) = S.prefix(i+k).suffix(k)$
- 空串:  $S[0, n=0]$ ，也是任何串的子串、前缀、后缀

```
function queryPST(PSTNode v, SegInRange R, I)
    if (!v) || (R.y < v.y) return; // prune
    if ((R.x1 < v.x) && (v.x < R.x2)) output(v); // hit
    if ((R.x1 < v.xl) && queryPST(v.lc, R)); // recursion
    if ((v.xr < R.x2) && queryPST(v.rc, R)); // recursion
}
```



Data Structures &amp; Algorithms, Chapter 11

length()	$[0, \infty]$
charAt(i)	$[0, n]$
substr(i, n)	$[0, 1] \dots [1, n]$
prefix(k)	$[0, k]$
suffix(k)	$[0, n-k]$
concat(T)	$S$
equal(T)	$T$
indexOf(P)	$[0, n+k+1]$
S	$\#(0, n)$
T	

Data Structures &amp; Algorithms, Chapter 11

dengtinghua@sohu.com

由来自字母表的字符所构成的有限序列:

T s i n g h u a

 $S = \{a_0, a_1, a_2, \dots, a_{n-1}\} \neq \emptyset$ 

类似地, 为什么不能用序列来实现呢?

通常, 字符的种类不多, 因而长  $= n \gg |S|$ 比如: 英文文章:  $\{('A'-'Z') \cup ('a'-'z') \cup ('.' ',', '!', '?', ';', '!', '---')\}^*$ C/C++程序:  $\{('95个可打印字符') \cup ('IF', 'OR')\}^*$ 天然蛋白质:  $\{21种氨基酸\}^*$ DNA:  $\{A, C, G, T\}^*$ RNA:  $\{A, C, G, U\}^*$ 二进制:  $\{0, 1\}^*$ 

Data Structures &amp; Algorithms, Chapter 11

dengtinghua@sohu.com

- “data structures”.length() = 15
- “data structures”.charAt(0) = ‘d’
- “data structures”.prefix(4) = “data”
- “data structures”.suffix(10) = “structures”
- “data structures”.concat(“ I algorithms”) = “data structures & algorithms”
- “algorithms”.equal(“data\_structures”) = False
- “data structures and algorithms”.indexOf(“string”) = -1
- “data structures and algorithms”.indexOf(“algorithme”) = 20
- 字符串处理的常见函数: strlen(), strcpy(), strcat(), strcmp(), strstr()
- 以下, 直接利用字符串实现字符串, 而非重点讨论出匹配算法

Data Structures &amp; Algorithms, Chapter 11

## 模式匹配

```

文本 T = now is the time for all good people to come
模式 P = people

△ 若 n = |T| 和 m = |P|，通常有  $T \gg P \gg 1$  //比如，100,000 >> 100 >> 1

△ 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？
enumeration: 什么出现在哪里？ //find /c "2013" students.txt

```

## 构思

△ 由右向左，以字符为单位，依次移动模式串

直到某个位置，发现匹配



△ 如何：确定串T和P每次对比时的字符串位置？并

在发现某对字符不匹配，调整位置以继续对比？

## 模式匹配

```

示例：T = "I O U E I A T T I G A T T I C H H I"
P = "I A T T"

△ 应用：文本编辑器，数据匹配，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：最近的匹配在什么位置？

HBB_HUMAN vs. HBB_HUMAN
图 S A Q V R K G H E E X V I A D R L T N A V A K R Y F E Q D P M A I S A C T S D I I A H
图 R P R V K D R A H E E X V I A D R L T N A V A K R Y F E Q D P M A I S A C T S D I I A H C D

```

## 高级

△ 实现1：[ ] 和 [ ] 分别指网T[] 和 P[] 中待匹配的字符

```

if (T[i] == P[j]) { i++; j++; } //若匹配，则同时右移
else { i = j - 1; j = 0; } //若不匹配，则 P 固定，i 回溯

```

△ 实现2：[ ] + [ ] 分别指网T[] 和 P[] 中待匹配的字符

```

if (T[i+1] == P[j]) { i++; j++; } //若匹配，同时右移
else { i++; j = 0; } //若不匹配，则 P 固定，j 回溯

```

△ 两种实现方法，各有什么优缺点？

## 算法评测

△ 如何直观地理解字符串匹配算法的复杂度？具体采用什么原理与策略？

△ 随机P + 随机T？不要！

△ 以  $X = \{0, 1\}^n$  为例  
 | (长度为 |P| 的 P ) | =  $2^n$   
 | (长度为 |T| 在 T 中出现的 P ) | =  $n - n + 1 < n$

匹配成功的概率 =  $\pi/2^n \ll 100,000 / 2^{10} \ll 10^{-16}$

如此，将无法对算法做充分测试

△ 随机P，对成功失败的匹配分别测试

成功：在 T 中，随机取出长度为 n 的子串作为 P；分析平均匹配度

失败：采用随机的 P；统计平均匹配度

## 版本1

```

int match( char * P, char * T ) {
    size_t n = strlen(T), i = 0;
    size_t m = strlen(P), j = 0;
    while ( i < n && i < n ) //一直在看这个比对字符
        if ( T[i] == P[j] ) { i++; j++; } //若匹配，则转到下一组字符
        else { i = j - 1; j = 0; } //否则，T回退，P定位
    return i - j;
}

```

## 11.串

## 模式匹配

## 暴力匹配

## 解 读

http://tiny.cc/meyarw

## 版本2

```

int match( char * P, char * T ) {
    size_t n = strlen(T), i = 0; //T[i]与P[0]对齐
    size_t m = strlen(P), j = 0; //T[i+m]与P[1]对齐
    for ( i = 0; i < n - m + 1; i++ ) { //从第1个字符起，与
        for ( j = 0; j < m; j++ ) //P中相对的字符串逐个比对
            if ( T[i+j] != P[j] ) break; //若失败，P相对右移一个字符，重新比对
        if ( i == j ) break; //找到匹配子串
    }
    return i;
}

```

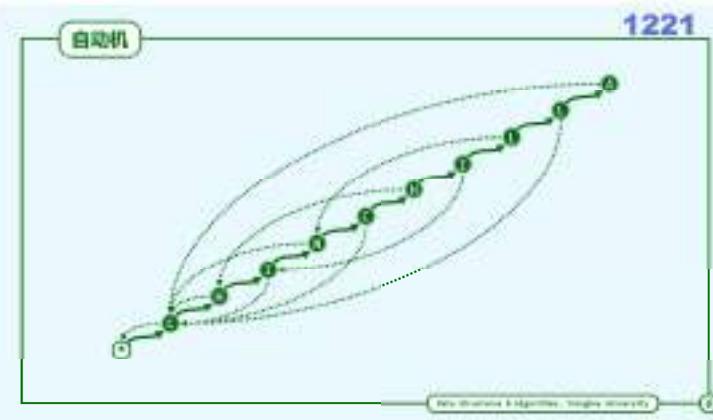


**事先确定 1**

```
#include <iostream>
using namespace std;
char next[80], T[80];
int i = 0, j = 0;
int main() {
    cin >> next >> T;
    int n = strlen(T), l = 0; //文本串指针
    int m = strlen(next), j = 0; //模式串指针
    while (j < n && l < m) {
        if (T[l] == next[j]) { //匹配成功
            l++; j++; //向右移动
        } else { //否则，P匹配，T不匹配
            l = next[j]; //从头开始
            j++;
        }
    }
    cout << l - j;
}
```

该函数实现 `next[i][n]`；在任一位置 `T[i]` 处失败之后，将 `j` 调整为 `next[1]`。

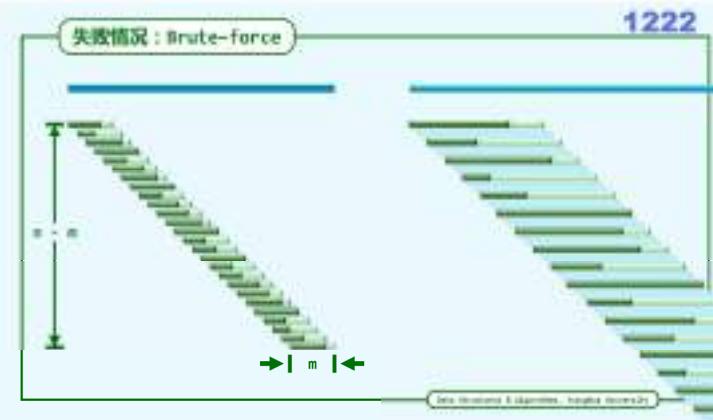
与其说是相等，不如说是做好充分的准备。



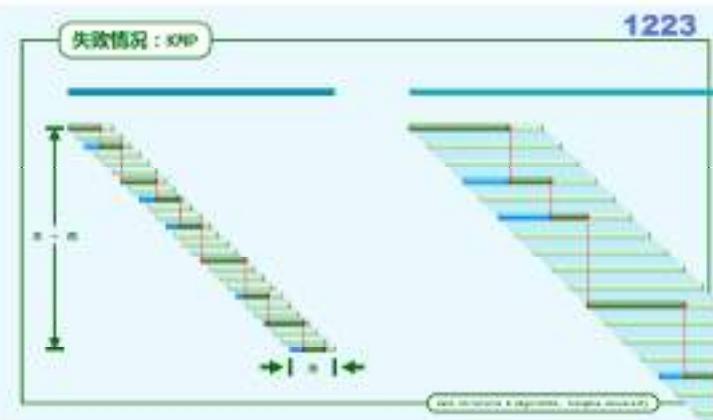
**KMP算法**

```
#include <iostream>
using namespace std;
char next[80], T[80];
int *next = buildNext(P); //构建next表
int n = strlen(T), l = 0; //文本串指针
int m = strlen(P), j = 0; //模式串指针
while (j < n && l < m) {
    if (T[l] == P[j]) { //匹配成功
        l++; j++; //向右移动
    } else { //否则，P匹配，T不匹配
        j = next[j];
    }
}
delete [] next; //释放next表
return l - j;
```

该函数实现 `next[i][n]`；在任一位置 `T[i]` 处失败之后，将 `j` 调整为 `next[1]`。



**实例**



**自动机**

```
#include <iostream>
using namespace std;
char next[80], T[80];
int n = strlen(T); int l = -1; //文本串的序位置
int i = 0;
if (T[0] == next[1]) goto s1; if (n == m+1) return -1; // S
s1: if (T[1] == next[2]) goto s2; if (n == m+1) return -1; // S1
s2: if (T[2] == next[3]) goto s3; if (n == m+1) return -1; // S2
s3: if (T[3] == next[4]) goto s4; if (n == m+1) return -1; // S3
s4: if (T[4] == next[5]) goto s5; if (n == m+1) return -1; // S4
s5: if (T[5] == next[6]) goto s6; if (n == m+1) return -1; // S5
s6: if (T[6] == next[7]) goto s7; if (n == m+1) return -1; // S6
s7: if (T[7] == next[8]) goto s8; if (n == m+1) return -1; // S7
s8: if (T[8] == next[9]) goto s9; if (n == m+1) return -1; // S8
s9: if (T[9] == next[10]) goto s10; if (n == m+1) return -1; // S9
s10: if (T[10] == next[11]) goto s11; if (n == m+1) return -1; // S10
s11: if (T[11] == next[12]) goto s12; if (n == m+1) return -1; // S11
s12: if (T[12] == next[13]) goto s13; if (n == m+1) return -1; // S12
s13: if (T[13] == next[14]) goto s14; if (n == m+1) return -1; // S13
s14: if (T[14] == next[15]) goto s15; if (n == m+1) return -1; // S14
s15: if (T[15] == next[16]) goto s16; if (n == m+1) return -1; // S15
s16: if (T[16] == next[17]) goto s17; if (n == m+1) return -1; // S16
s17: if (T[17] == next[18]) goto s18; if (n == m+1) return -1; // S17
s18: if (T[18] == next[19]) goto s19; if (n == m+1) return -1; // S18
s19: if (T[19] == next[20]) goto s20; if (n == m+1) return -1; // S19
s20: if (T[20] == next[21]) goto s21; if (n == m+1) return -1; // S20
s21: if (T[21] == next[22]) goto s22; if (n == m+1) return -1; // S21
s22: if (T[22] == next[23]) goto s23; if (n == m+1) return -1; // S22
s23: if (T[23] == next[24]) goto s24; if (n == m+1) return -1; // S23
s24: if (T[24] == next[25]) goto s25; if (n == m+1) return -1; // S24
s25: if (T[25] == next[26]) goto s26; if (n == m+1) return -1; // S25
s26: if (T[26] == next[27]) goto s27; if (n == m+1) return -1; // S26
s27: if (T[27] == next[28]) goto s28; if (n == m+1) return -1; // S27
s28: if (T[28] == next[29]) goto s29; if (n == m+1) return -1; // S28
s29: if (T[29] == next[30]) goto s30; if (n == m+1) return -1; // S29
s30: if (T[30] == next[31]) goto s31; if (n == m+1) return -1; // S30
s31: if (T[31] == next[32]) goto s32; if (n == m+1) return -1; // S31
s32: if (T[32] == next[33]) goto s33; if (n == m+1) return -1; // S32
s33: if (T[33] == next[34]) goto s34; if (n == m+1) return -1; // S33
s34: if (T[34] == next[35]) goto s35; if (n == m+1) return -1; // S34
s35: if (T[35] == next[36]) goto s36; if (n == m+1) return -1; // S35
s36: if (T[36] == next[37]) goto s37; if (n == m+1) return -1; // S36
s37: if (T[37] == next[38]) goto s38; if (n == m+1) return -1; // S37
s38: if (T[38] == next[39]) goto s39; if (n == m+1) return -1; // S38
s39: if (T[39] == next[40]) goto s40; if (n == m+1) return -1; // S39
s40: if (T[40] == next[41]) goto s41; if (n == m+1) return -1; // S40
s41: if (T[41] == next[42]) goto s42; if (n == m+1) return -1; // S41
s42: if (T[42] == next[43]) goto s43; if (n == m+1) return -1; // S42
s43: if (T[43] == next[44]) goto s44; if (n == m+1) return -1; // S43
s44: if (T[44] == next[45]) goto s45; if (n == m+1) return -1; // S44
s45: if (T[45] == next[46]) goto s46; if (n == m+1) return -1; // S45
s46: if (T[46] == next[47]) goto s47; if (n == m+1) return -1; // S46
s47: if (T[47] == next[48]) goto s48; if (n == m+1) return -1; // S47
s48: if (T[48] == next[49]) goto s49; if (n == m+1) return -1; // S48
s49: if (T[49] == next[50]) goto s50; if (n == m+1) return -1; // S49
s50: if (T[50] == next[51]) goto s51; if (n == m+1) return -1; // S50
s51: if (T[51] == next[52]) goto s52; if (n == m+1) return -1; // S51
s52: if (T[52] == next[53]) goto s53; if (n == m+1) return -1; // S52
s53: if (T[53] == next[54]) goto s54; if (n == m+1) return -1; // S53
s54: if (T[54] == next[55]) goto s55; if (n == m+1) return -1; // S54
s55: if (T[55] == next[56]) goto s56; if (n == m+1) return -1; // S55
s56: if (T[56] == next[57]) goto s57; if (n == m+1) return -1; // S56
s57: if (T[57] == next[58]) goto s58; if (n == m+1) return -1; // S57
s58: if (T[58] == next[59]) goto s59; if (n == m+1) return -1; // S58
s59: if (T[59] == next[60]) goto s60; if (n == m+1) return -1; // S59
s60: if (T[60] == next[61]) goto s61; if (n == m+1) return -1; // S60
s61: if (T[61] == next[62]) goto s62; if (n == m+1) return -1; // S61
s62: if (T[62] == next[63]) goto s63; if (n == m+1) return -1; // S62
s63: if (T[63] == next[64]) goto s64; if (n == m+1) return -1; // S63
s64: if (T[64] == next[65]) goto s65; if (n == m+1) return -1; // S64
s65: if (T[65] == next[66]) goto s66; if (n == m+1) return -1; // S65
s66: if (T[66] == next[67]) goto s67; if (n == m+1) return -1; // S66
s67: if (T[67] == next[68]) goto s68; if (n == m+1) return -1; // S67
s68: if (T[68] == next[69]) goto s69; if (n == m+1) return -1; // S68
s69: if (T[69] == next[70]) goto s70; if (n == m+1) return -1; // S69
s70: if (T[70] == next[71]) goto s71; if (n == m+1) return -1; // S70
s71: if (T[71] == next[72]) goto s72; if (n == m+1) return -1; // S71
s72: if (T[72] == next[73]) goto s73; if (n == m+1) return -1; // S72
s73: if (T[73] == next[74]) goto s74; if (n == m+1) return -1; // S73
s74: if (T[74] == next[75]) goto s75; if (n == m+1) return -1; // S74
s75: if (T[75] == next[76]) goto s76; if (n == m+1) return -1; // S75
s76: if (T[76] == next[77]) goto s77; if (n == m+1) return -1; // S76
s77: if (T[77] == next[78]) goto s78; if (n == m+1) return -1; // S77
s78: if (T[78] == next[79]) goto s79; if (n == m+1) return -1; // S78
s79: if (T[79] == next[80]) goto s80; if (n == m+1) return -1; // S79
s80: if (T[80] == next[81]) goto s81; if (n == m+1) return -1; // S80
s81: if (T[81] == next[82]) goto s82; if (n == m+1) return -1; // S81
s82: if (T[82] == next[83]) goto s83; if (n == m+1) return -1; // S82
s83: if (T[83] == next[84]) goto s84; if (n == m+1) return -1; // S83
s84: if (T[84] == next[85]) goto s85; if (n == m+1) return -1; // S84
s85: if (T[85] == next[86]) goto s86; if (n == m+1) return -1; // S85
s86: if (T[86] == next[87]) goto s87; if (n == m+1) return -1; // S86
s87: if (T[87] == next[88]) goto s88; if (n == m+1) return -1; // S87
s88: if (T[88] == next[89]) goto s89; if (n == m+1) return -1; // S88
s89: if (T[89] == next[90]) goto s90; if (n == m+1) return -1; // S89
s90: if (T[90] == next[91]) goto s91; if (n == m+1) return -1; // S90
s91: if (T[91] == next[92]) goto s92; if (n == m+1) return -1; // S91
s92: if (T[92] == next[93]) goto s93; if (n == m+1) return -1; // S92
s93: if (T[93] == next[94]) goto s94; if (n == m+1) return -1; // S93
s94: if (T[94] == next[95]) goto s95; if (n == m+1) return -1; // S94
s95: if (T[95] == next[96]) goto s96; if (n == m+1) return -1; // S95
s96: if (T[96] == next[97]) goto s97; if (n == m+1) return -1; // S96
s97: if (T[97] == next[98]) goto s98; if (n == m+1) return -1; // S97
s98: if (T[98] == next[99]) goto s99; if (n == m+1) return -1; // S98
s99: if (T[99] == next[100]) goto s100; if (n == m+1) return -1; // S99
s100: if (T[100] == next[101]) goto s101; if (n == m+1) return -1; // S100
s101: if (T[101] == next[102]) goto s102; if (n == m+1) return -1; // S101
s102: if (T[102] == next[103]) goto s103; if (n == m+1) return -1; // S102
s103: if (T[103] == next[104]) goto s104; if (n == m+1) return -1; // S103
s104: if (T[104] == next[105]) goto s105; if (n == m+1) return -1; // S104
s105: if (T[105] == next[106]) goto s106; if (n == m+1) return -1; // S105
s106: if (T[106] == next[107]) goto s107; if (n == m+1) return -1; // S106
s107: if (T[107] == next[108]) goto s108; if (n == m+1) return -1; // S107
s108: if (T[108] == next[109]) goto s109; if (n == m+1) return -1; // S108
s109: if (T[109] == next[110]) goto s110; if (n == m+1) return -1; // S109
s110: if (T[110] == next[111]) goto s111; if (n == m+1) return -1; // S110
s111: if (T[111] == next[112]) goto s112; if (n == m+1) return -1; // S111
s112: if (T[112] == next[113]) goto s113; if (n == m+1) return -1; // S112
s113: if (T[113] == next[114]) goto s114; if (n == m+1) return -1; // S113
s114: if (T[114] == next[115]) goto s115; if (n == m+1) return -1; // S114
s115: if (T[115] == next[116]) goto s116; if (n == m+1) return -1; // S115
s116: if (T[116] == next[117]) goto s117; if (n == m+1) return -1; // S116
s117: if (T[117] == next[118]) goto s118; if (n == m+1) return -1; // S117
s118: if (T[118] == next[119]) goto s119; if (n == m+1) return -1; // S118
s119: if (T[119] == next[120]) goto s120; if (n == m+1) return -1; // S119
s120: if (T[120] == next[121]) goto s121; if (n == m+1) return -1; // S120
s121: if (T[121] == next[122]) goto s122; if (n == m+1) return -1; // S121
s122: if (T[122] == next[123]) goto s123; if (n == m+1) return -1; // S122
s123: if (T[123] == next[124]) goto s124; if (n == m+1) return -1; // S123
s124: if (T[124] == next[125]) goto s125; if (n == m+1) return -1; // S124
s125: if (T[125] == next[126]) goto s126; if (n == m+1) return -1; // S125
s126: if (T[126] == next[127]) goto s127; if (n == m+1) return -1; // S126
s127: if (T[127] == next[128]) goto s128; if (n == m+1) return -1; // S127
s128: if (T[128] == next[129]) goto s129; if (n == m+1) return -1; // S128
s129: if (T[129] == next[130]) goto s130; if (n == m+1) return -1; // S129
s130: if (T[130] == next[131]) goto s131; if (n == m+1) return -1; // S130
s131: if (T[131] == next[132]) goto s132; if (n == m+1) return -1; // S131
s132: if (T[132] == next[133]) goto s133; if (n == m+1) return -1; // S132
s133: if (T[133] == next[134]) goto s134; if (n == m+1) return -1; // S133
s134: if (T[134] == next[135]) goto s135; if (n == m+1) return -1; // S134
s135: if (T[135] == next[136]) goto s136; if (n == m+1) return -1; // S135
s136: if (T[136] == next[137]) goto s137; if (n == m+1) return -1; // S136
s137: if (T[137] == next[138]) goto s138; if (n == m+1) return -1; // S137
s138: if (T[138] == next[139]) goto s139; if (n == m+1) return -1; // S138
s139: if (T[139] == next[140]) goto s140; if (n == m+1) return -1; // S139
s140: if (T[140] == next[141]) goto s141; if (n == m+1) return -1; // S140
s141: if (T[141] == next[142]) goto s142; if (n == m+1) return -1; // S141
s142: if (T[142] == next[143]) goto s143; if (n == m+1) return -1; // S142
s143: if (T[143] == next[144]) goto s144; if (n == m+1) return -1; // S143
s144: if (T[144] == next[145]) goto s145; if (n == m+1) return -1; // S144
s145: if (T[145] == next[146]) goto s146; if (n == m+1) return -1; // S145
s146: if (T[146] == next[147]) goto s147; if (n == m+1) return -1; // S146
s147: if (T[147] == next[148]) goto s148; if (n == m+1) return -1; // S147
s148: if (T[148] == next[149]) goto s149; if (n == m+1) return -1; // S148
s149: if (T[149] == next[150]) goto s150; if (n == m+1) return -1; // S149
s150: if (T[150] == next[151]) goto s151; if (n == m+1) return -1; // S150
s151: if (T[151] == next[152]) goto s152; if (n == m+1) return -1; // S151
s152: if (T[152] == next[153]) goto s153; if (n == m+1) return -1; // S152
s153: if (T[153] == next[154]) goto s154; if (n == m+1) return -1; // S153
s154: if (T[154] == next[155]) goto s155; if (n == m+1) return -1; // S154
s155: if (T[155] == next[156]) goto s156; if (n == m+1) return -1; // S155
s156: if (T[156] == next[157]) goto s157; if (n == m+1) return -1; // S156
s157: if (T[157] == next[158]) goto s158; if (n == m+1) return -1; // S157
s158: if (T[158] == next[159]) goto s159; if (n == m+1) return -1; // S158
s159: if (T[159] == next[160]) goto s160; if (n == m+1) return -1; // S159
s160: if (T[160] == next[161]) goto s161; if (n == m+1) return -1; // S160
s161: if (T[161] == next[162]) goto s162; if (n == m+1) return -1; // S161
s162: if (T[162] == next[163]) goto s163; if (n == m+1) return -1; // S162
s163: if (T[163] == next[164]) goto s164; if (n == m+1) return -1; // S163
s164: if (T[164] == next[165]) goto s165; if (n == m+1) return -1; // S164
s165: if (T[165] == next[166]) goto s166; if (n == m+1) return -1; // S165
s166: if (T[166] == next[167]) goto s167; if (n == m+1) return -1; // S166
s167: if (T[167] == next[168]) goto s168; if (n == m+1) return -1; // S167
s168: if (T[168] == next[169]) goto s169; if (n == m+1) return -1; // S168
s169: if (T[169] == next[170]) goto s170; if (n == m+1) return -1; // S169
s170: if (T[170] == next[171]) goto s171; if (n == m+1) return -1; // S170
s171: if (T[171] == next[172]) goto s172; if (n == m+1) return -1; // S171
s172: if (T[172] == next[173]) goto s173; if (n == m+1) return -1; // S172
s173: if (T[173] == next[174]) goto s174; if (n == m+1) return -1; // S173
s174: if (T[174] == next[175]) goto s175; if (n == m+1) return -1; // S174
s175: if (T[175] == next[176]) goto s176; if (n == m+1) return -1; // S175
s176: if (T[176] == next[177]) goto s177; if (n == m+1) return -1; // S176
s177: if (T[177] == next[178]) goto s178; if (n == m+1) return -1; // S177
s178: if (T[178] == next[179]) goto s179; if (n == m+1) return -1; // S178
s179: if (T[179] == next[180]) goto s180; if (n == m+1) return -1; // S179
s180: if (T[180] == next[181]) goto s181; if (n == m+1) return -1; // S180
s181: if (T[181] == next[182]) goto s182; if (n == m+1) return -1; // S181
s182: if (T[182] == next[183]) goto s183; if (n == m+1) return -1; // S182
s183: if (T[183] == next[184]) goto s184; if (n == m+1) return -1; // S183
s184: if (T[184] == next[185]) goto s185; if (n == m+1) return -1; // S184
s185: if (T[185] == next[186]) goto s186; if (n == m+1) return -1; // S185
s186: if (T[186] == next[187]) goto s187; if (n == m+1) return -1; // S186
s187: if (T[187] == next[188]) goto s188; if (n == m+1) return -1; // S187
s188: if (T[188] == next[189]) goto s189; if (n == m+1) return -1; // S188
s189: if (T[189] == next[190]) goto s190; if (n == m+1) return -1; // S189
s190: if (T[190] == next[191]) goto s191; if (n == m+1) return -1; // S190
s191: if (T[191] == next[192]) goto s192; if (n == m+1) return -1; // S191
s192: if (T[192] == next[193]) goto s193; if (n == m+1) return -1; // S192
s193: if (T[193] == next[194]) goto s194; if (n == m+1) return -1; // S193
s194: if (T[194] == next[195]) goto s195; if (n == m+1) return -1; // S194
s195: if (T[195] == next[196]) goto s196; if (n == m+1) return -1; // S195
s196: if (T[196] == next[197]) goto s197; if (n == m+1) return -1; // S196
s197: if (T[197] == next[198]) goto s198; if (n == m+1) return -1; // S197
s198: if (T[198] == next[199]) goto s199; if (n == m+1) return -1; // S198
s199: if (T[199] == next[200]) goto s200; if (n == m+1) return -1; // S199
s200: if (T[200] == next[201]) goto s201; if (n == m+1) return -1; // S200
s201: if (T[201] == next[202]) goto s202; if (n == m+1) return -1; // S201
s202: if (T[202] == next[203]) goto s203; if (n == m+1) return -1; // S202
s203: if (T[203] == next[204]) goto s204; if (n == m+1) return -1; // S203
s204: if (T[204] == next[205]) goto s205; if (n == m+1) return -1; // S204
s205: if (T[205] == next[206]) goto s206; if (n == m+1) return -1; // S205
s206: if (T[206] == next[207]) goto s207; if (n == m+1) return -1; // S206
s207: if (T[207] == next[208]) goto s208; if (n == m+1) return -1; // S207
s208: if (T[208] == next[209]) goto s209; if (n == m+1) return -1; // S208
s209: if (T[209] == next[210]) goto s210; if (n == m+1) return -1; // S209
s210: if (T[210] == next[211]) goto s211; if (n == m+1) return -1; // S210
s211: if (T[211] == next[212]) goto s212; if (n == m+1) return -1; // S211
s212: if (T[212] == next[213]) goto s213; if (n == m+1) return -1; // S212
s213: if (T[213] == next[214]) goto s214; if (n == m+1) return -1; // S213
s214: if (T[214] == next[215]) goto s215; if (n == m+1) return -1; // S214
s215: if (T[215] == next[216]) goto s216; if (n == m+1) return -1; // S215
s216: if (T[216] == next[217]) goto s217; if (n == m+1) return -1; // S216
s217: if (T[217] == next[218]) goto s218; if (n == m+1) return -1; // S217
s218: if (T[218] == next[219]) goto s219; if (n == m+1) return -1; // S218
s219: if (T[219] == next[220]) goto s220; if (n == m+1) return -1; // S219
s220: if (T[220] == next[221]) goto s221; if (n == m+1) return -1; // S220
s221: if (T[221] == next[222]) goto s222; if (n == m+1) return -1; // S221
s222: if (T[222] == next[223]) goto s223; if (n == m+1) return -1; // S222
s2
```

## 自匹配 - 快速右移

1225

- 当对齐后，考虑组合： $R(P, t) = \{0 \leq i \leq |P|, 0 \leq j \leq |T|, P[i, t] == P[i-t, j]\}$
- 亦即，在 $P[j]$ 的前面 $P[0, t]$ 中，所有匹配的起始位置和对应的长度

因此，一旦 $t=1 \neq P[0]$ ，可从 $R(P, 1)$ 中断（某个 $i$ ），令 $P[i]$ 对准 $T[1]$ ，并继续比对



Java实现见《Algorithm》第4章例程 12.1

## 进阶

↑ 那么已知 $\text{next}[0, 1]$ ，如何高效的计算 $\text{next}[j+1]$ ？

↑ 所谓 $\text{next}[1]$ ，即是在 $P[0, 1]$ 中，最大自匹配的起始位置和对应的长度

↓ 故： $\text{next}[j+1] = \text{next}[j] + 1$  // 这里假设 $P[0, j] == T[0, \text{next}[j]]$ 时数等号



↑ 一般地， $P[0, j] == P[0, \text{next}[j]]$ 时，又该如何得到 $\text{next}[j+1]$ ？

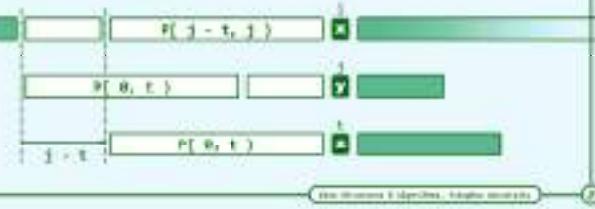
## 最长自匹配 - 快速右移 + 填不相退

1226

↑  $|R(P, 1)| > 1$  时，难道需要遍历到中间的一个 $t$ ？不必！

↓ 理由：位移量  $= |j - t|$ ，与 $t$ 成反比

↓ 因此，若使用最大的 $t$ ，则必然最安全



Java实现见《Algorithm》第4章例程 12.1

## 算法

↑  $\text{next}[j+1]$ 的解决策

依次计算：

$1 + \text{next}[j]$   
 $1 + \text{next}[\text{next}[j]]$   
 $1 + \text{next}[\text{next}[\text{next}[j]]]$   
 $\dots$

↓ 这个操作严格递减，且

必须等于 $1 + \text{next}[0] = n$

↑ 以上递推过程，即为 $\text{next}$ 的自匹配过程，故只须对 KMP 程序略做修改...

1230

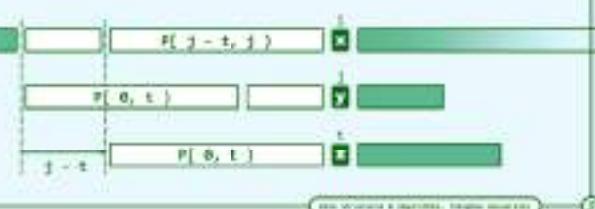
 $\text{next}[0]$ 

1227

↑ 其实 $j > n$ ，也有 $|R(P, j)| > |R(P, 1)|$  // 空串是任何非空串的最长子串

↓ 但假 $j = 0$ ，则有 $|R(P, 0)| = 0$  // 空串没有最长子串

↓ 不过取 $\text{next}[0] = -1, \dots$  // 跟原式算法一样有效！你何理解？



Java实现见《Algorithm》第4章例程 12.1

## 实现

↑ int \* buildNext1( char \* P ) { // 构造按字母序的 next[] 数

size\_t n = strlen(P), j = 0; // "生" 原串

int \* N = new int[n]; // next[] 数

int T = N[0] = -1; // 指示前缀符 (P[-1]) 通常符

while ( j < n - 1 )

if ( [0, j+1] || P[j] == P[T] ) // 匹配，向左讲

N[ ++j ] = ++T; // 将 N[0] 替换为 T

else // 失配，则滑动

T = N[T];

return N;

1231



1228

11. 基  
KMP算法  
构造next[]表

单链表  
[gepingtinghua.com](http://gepingtinghua.com)

## 实例

china chinachin

-1 0

china chinachin

-1 0

china chinachin

-1 0 0

1232

Java实现见《Algorithm》第4章例程 12.1

实例



1233

 $\Theta(n + m)$ ?

```

 $\Delta P[i] = [j+1 - j]$  //具体含义，详见习题12-4

while (j < n && i < m) //i必须迭代而j要递增，故也是迭代参数的上界
    if (P[i] == T[j]) //如果相等，做i恰好加1
        { i++; j++; } //i, j同时加1，故i恰好加1
    else
        { j = next[j]; } //不变，j至少减1，故i至少加1

```

↑i的初值为0；算法结束时，必有：  
 $i = 2*j + j \leq 2(n+1) - (i-1) = 2n + 1$

1237

11.串

KMP算法  
时间分析

样例解

失之东隅，收之桑榆

1234

11.串

KMP算法  
时间分析

样例解

失之东隅，收之桑榆

1238

11.串

KMP算法  
再改进

样例解

要求：提前想出，能一点忘记了怎么投诉

要求：从此以后，不要犯同一个错误

样例解

前车之鉴，后车之鉴

 $\Omega(n * m)$ ?

1235

△观察：KMP算法的键可以节省多次比对

△然而：我指进双义两面

有实质的节省吗？

△观察：在每一个  $T[i+1]$  处下键可能比对  $\Omega(n)$  次△于是，假设  $\Theta(n)$  个  $T[i+1]$  如此...

反例

1239

 $\Theta(T + P)$ ?

P = 0 0 0 0 1 0 0 0 0 1

△  $P[1]$  :与  $P[1]$  比对，失败与  $P[2] = P[next[1]]$  比对，失败与  $P[3] = P[next[2]]$  比对，失败与  $P[4] = P[next[3]]$  比对，失败最终，才匹配到  $P[4]$ 。 $\Omega(n * m)$ ?

1236

△难道，总体复杂度还跟

 $\Omega(n + m)$ ?

△然而，更细致的分析将表明

即使是最坏情况，也不过

 $O(n)$  时间△同样，建立  $next[]$  也只用 $O(n)$  时间

根除

1240

△无需T串，你可在事前搞定：

P[3] =

P[2] =

P[1] =

P[0] = ②

题目如此...

△在满足  $P[3] = P[1]$  之后

为何还要一通再通？

△事实上，后三次比对本来都是可以避免的！



改进

1241

```

int * buildNext( char * P ) {
    size_t n = strlen(P), j = 0; // "主"串指针
    int * N = new int[n]; // next表
    int t = N[0] = -1; // 模式串指针
    while( j < n - 1 ) {
        if( 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;
}

```

经验 + 教训

1245

◆ 原因： $\rightarrow$  一次失配的对齐  $\rightarrow$  0/1次成功，0次失败

◆ 与匹配要加速匹配，不知道是加速失配 —— 寻找问题失配的对齐

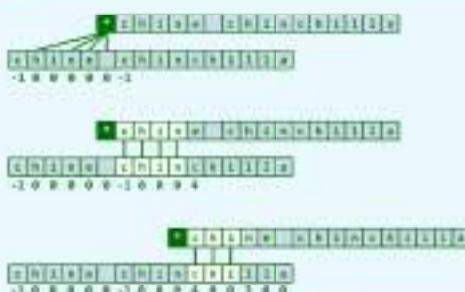
+ 单个对齐拉远的对齐距离  
平均仅需常数次对比（只要 $t \ll n$  不要太小，常数对比成功率足够高）  
且具体的匹配位数/失败率无所谓

◆ 然而此阶段要多遍历对齐位置的匹配  
不同的对比位置/失败率，作用影响极大

◆ 通常，越强距离/后的匹配，作用越小/大

实例

1242



善待教训

1246



小结

1243

◆ 充分利用以往的对比所提供的信息  
模式串快速右移，文本串无用扫描

◆ 提醒：以住商对的对比 ——  $T[i - j, i]$  是什么  
翻译：以住失配的对比 ——  $T[i]$  不是什么

◆ 特别适用于简单字符串

◆ 常数匹配概率很大 ( $|T| \ll n$ ) 的情况下，优势越明显  
否则，与暴力算法的性能相差无几...

前轻后重

1247



1244

11.串

逆向法：从后往前  
以块为单位

串结构

Begin with the end in mind.

以块为始

1248

◆ 既然如此，每一组对比都应该

从末字符开始

自后向前，自右向左

令  $n = 4 \times 12$ 

以终为始 ◊ [ Boyer + Moore, 1977 ] A fast string searching algorithm

△ 指处理：根据模式串 $P$ ，预先计算 $T[i:j]$ 中哪位 $b \in P$ 不真

优化：在字符串 $T$ 依次比对字符，找到最大的匹配后缀

若完全匹配，则返回位置

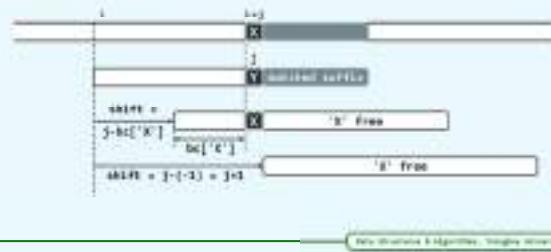
否则，根据 $T[i:j]$ 和 $b \in P$ ， $i$ 适当右移，并重新自右向左比对



## Bad-Character Shift

△ 若 $T$ 包含 $[多个]X$ ，假其中 $X$ 都正确 // 此时可直接丢弃后缀

△ 若 $T$ 不含任何 $X$ ，将 $T$ 整体右移 $i+1$  // 与KMP类似，效率较高。与滑动的匹配 $\neq$ 对齐



## 11.串

BM算法：BC策略

坏字符

哪儿了？

大王被贼偷儿啊

怎么这么困难？

杀谁的狗？

谁被狗咬？

五六十岁

一惊一跳

谁咬谁两块二呀

谁咬谁一惊一跳

谁咬谁两块二呀

谁咬谁两块二呀

李佳辉

新郎新娘一派六十六枝红花二分吉祥

新娘一下枝一枝，剩下多枝嫁妆！

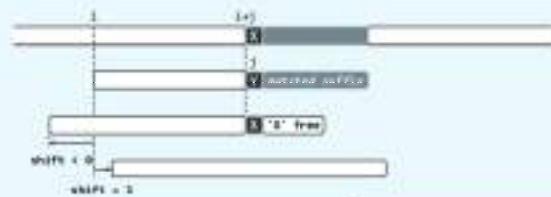
zhang@tonghuaxu.edu.cn

## Bad-Character Shift

△ 那便 $'X'$ 在 $T$ 中出现，检测右脚部 $过字偏右$ （以该子位移量为 $l$ ）呢？

△ 指 $T[i:j]$ 右部一个字符！

△ 为何...不选相 $T[i:j]$ 左脚部 $过字偏左$ 的那个 $l$ ？



## Bad-Character

△ 基础归结中，一旦发现

$T[i:j] = X \neq Y = P[i:j]$  //  $Y$ 即作坏字符  
则 $T$ 粗白地右移，并启动新一轮扫描比对 // 具体如何右移？



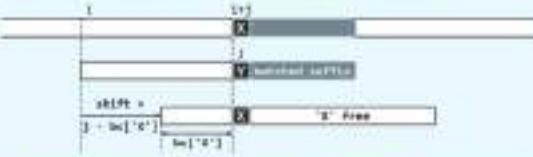
△ 必要条件：至少一个坏字符本身应能以检测匹配——因此...

## 实例



## Bad-Character Shift

△ 只需：找出 $P$ 中的 $X$ ，使之与 $T[i:j] = X$ 对准，并做新的一轮比对



△ 注意：位移量取决于失配位置 $i$ ，以及 $X$ 在 $P$ 中的 $rank$ ，而与 $T$ 和 $j$ 无关！ // 与KMP类似

△ 退： $\oplus bc[X] = rank[X] = j - shift$

△  $bc[]$ 总计有 $s = |P|$ 项，且项事先计算，并静态存储

## 11.串

BM算法：BC策略

构造 $bc[]$

李佳辉

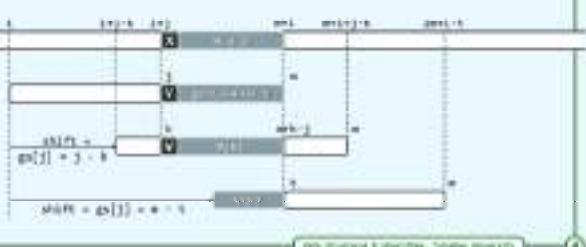
zhang@tonghuaxu.edu.cn



## Good-Suffix Shift

1265

- 在所有后缀  $P[0, j-1]$  中，与  $P[0, j-1]$  匹配的最长前缀 //注意：有可能  $j = 0$
- 无论如何，位移量仅取决于  $P$  和  $S$  本身——亦可预先计算，并制定查询



**实例**

字符串  $S$  和  $P$  分别为：

$S: \text{非 日 静 也 舌 改 静 出}$   
 $P: \text{也 静 也 舌 改 静 出}$

构造  $gS[i]$  表：

1	2	3	4	5	6	7	8	9	10	11	12	13	14
也	静	也	舌	改	静	出							

构造  $gS[i]$  表：

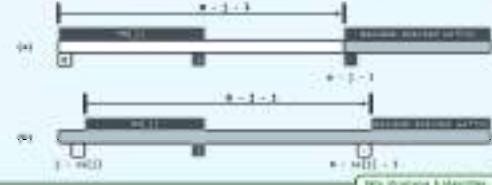
1	2	3	4	5	6	7	8	9	10	11	12	13	14
也	静	也	舌	改	静	出							

从右向左逐位扫描，只用  $O(n)$  时间 —— 习题 [11-6]

 $ss[] \rightarrow gs[]$ 

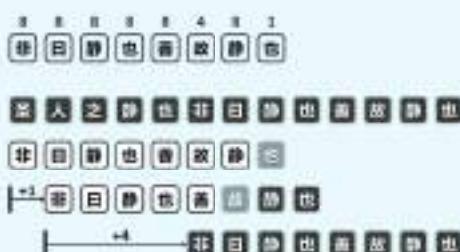
1269

- a) 若  $ss[j] = j + 1$ ，则 对于任一字符  $P[i]$  ( $1 \leq i \leq j - 1$ )  
 $i = j - 1$  必是  $gs[i]$  的一个候选
- b) 若  $ss[j] < j$ ，则 对于字符  $P[m - ss[j] - 1]$   
 $m - j - 1$  必是  $gs[m - ss[j] - 1]$  的一个候选



1266

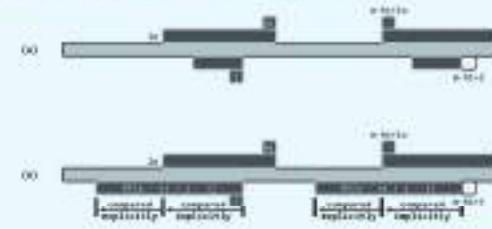
## 实例

构造  $ss[]$ 

1270

- 若能暴力匹配，每个字符都需一掠扫描，总计  $O(n^2)$  时间

- 从右向左逐位扫描，只用  $O(n)$  时间 —— 习题 [11-6]



1267

11. 索  
KMP算法：GS策略  
构造GS表

覃伟群  
<http://www.csust.edu.cn/~tian/>

1271

11. 索  
KMP算法：GS策略  
综合性能

覃伟群  
<http://www.csust.edu.cn/~tian/>

 $ps[] \rightarrow ss[]$ 

1268

- $ps[j] : ps[0, j]$  所有后缀中，与  $P$  的某一后缀匹配的最长前缀
- $ps[j] = ps[0, j] = \max\{0 \leq i \leq j+1 \mid P[j-i, j] = P[n-i, n]\} \leq j+1$



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

实际上， $ps[]$  表中蕴含了  $gs[]$  表的所有信息

/ 无关两种情况 ... /

## 性质

- 空间  
 $|ts| + |gs| = \Theta(|T| + n)$

- 时间复杂度  
 $\Theta(|T| + n)$

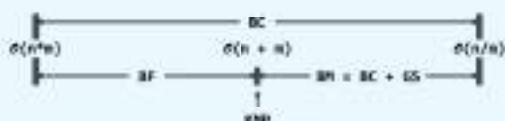
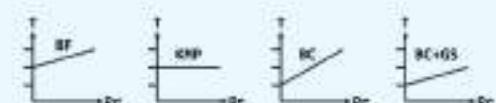
- 查找

最好 =  $\Theta(n / m)$

最差 =  $\Theta(n + m)$

- // 多种O(n)算法的分析

1272



## 11. 算术

Karp-Rabin算法

串即是数

All things are numbers.  
—Pythagoras (570 – 495 BC)God made the integers;  
all else is the work of man.  
—L. Kronecker (1823 – 1891)郑伟群  
zengwq@zjhu.edu.cn

Godel numbering

逻辑系统的形式、表达式、公式、命题、定理、公推等  
均可以不同的自然数表示素数序列： $p(x) =$  第x个质数

2, 3, 5, 7, 11, ...,

Godel's theorem：每个有限的自然数内函，都对应于一个唯一的数

$$\langle n_1, n_2, \dots, n_k \rangle = p(1)^{n_1} \times p(2)^{n_2} \times \dots \times p(k)^{n_k}$$

$$= p^1[n_1] \times p^2[n_2] \times p^3[n_3] \times \dots \times p^k[n_k]$$

Godel's theorem：给定可数字母表，有限长度的字符串均有一对对应于自然数

$$\begin{aligned} & \{ A, B, C, \dots, Z, \dots \} \\ & = \{ 1, 2, 3, \dots, 26, \dots \} \\ \text{godel: } & = 2^{1+1} \times 3^{1+1} \times 5^{1+1} \times 7^{1+1} \times 11^{1+1} \\ & = 139,869,568,618,664,817,887,943,919,288,000 \end{aligned}$$

若能直接从字符串计算，字长无关

则只要一个寄存器即可...

☆ 扩张素数，由6001素数可否唯一确定质数串？

☆ Book IX of The Elements of Geometry (ca 300 B.C.)

Facilitate every number factors uniquely into primes

☆ 因此，由合法的相等，经典因子分解再排序，都可唯一确定质数串

☆ 质因子分解，至今尚未有效算法！

——不幸？幸运？

☆ Cantor numbering



☆ cantor([1], [1]) = ((1+1)^2 + 2^1 + 1) / 2

cantor([1], [2]) = ((2+2)^2 + 2^2 + 2) / 2 = 17

cantor([2], [2]) = ((3+2)^2 + 2^3 + 2) / 2 = 38

☆ cantor\_m1([a1, ..., an], [m1, ..., mn]) =

cantor([a1, ..., an], cantor([m1, ..., mn]))



☆ 长度有限的字符串，都可操作 [0, 1] 内的自然数

decade = 453145<sub>(10)</sub>

// 10 = 1 + (10 - 1) = 10



☆ 长度无限的字符串，都可操作 [0, 1] 内的无限小数

signature[0..] = 0.271828182845...

fad(x) = 0.414[1] = 0.4144000... = fad111...



☆ Cantor：集合是否可数，与基数无关

有理数集与自然数集一样可数 // 分子、分母 // N

无理数集不可数 // Cantor's diagonal // N<sub>0</sub> [0, 1] R<sub>1</sub>

Godel's theorem：给定可数字母表，有限长度的字符串均有一对对应于自然数

$$\begin{aligned} & \{ A, B, C, \dots, Z, \dots \} \\ & = \{ 1, 2, 3, \dots, 26, \dots \} \\ \text{godel: } & = 2^{1+1} \times 3^{1+1} \times 5^{1+1} \times 7^{1+1} \times 11^{1+1} \\ & = 139,869,568,618,664,817,887,943,919,288,000 \end{aligned}$$

若能直接从字符串计算，字长无关

则只要一个寄存器即可...

☆ 十进制，可直接操作自然数 // 指纹 (Fingerprint)，等效于多项式语

p = 123456789 T = 2718281828459452353602874715527

☆ 一般地，随便对字符串 { 0, 1, 2, ..., d - 1 } // 设d = |T|

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

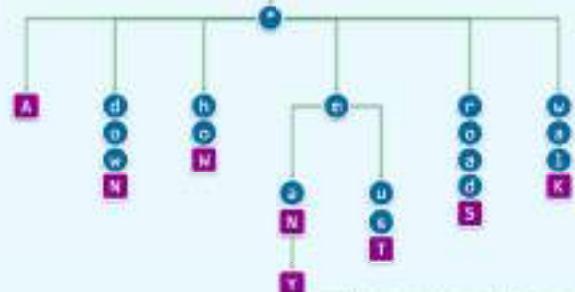
CAT = 2 0 19<sub>(10)</sub> = 1371<sub>(10)</sub> // { A, B, C, ..., Z }ABBA = 0 1 1 0<sub>(10)</sub> = 702<sub>(10)</sub>

☆ T 在 T 中出现 // 当 T 中某一子串与 p 相等 // 为什么不能是？

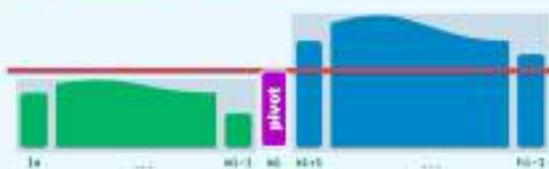
☆ 这，不已经就是个回文吗？ // 具体如何实现？

☆ 到处似乎解决得很快利，果真如此简单吗？ // 是复杂？



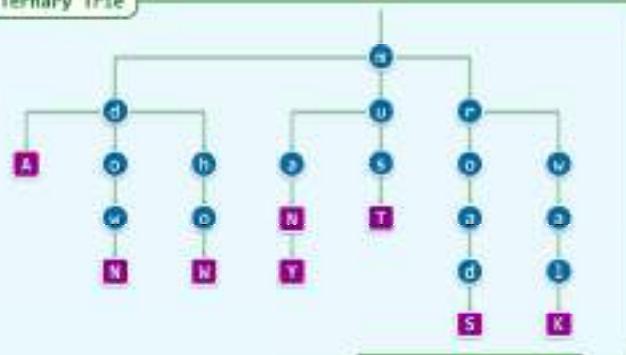


◆ pivot：左/右端的元素，不把它变大/小



◆ 以轴点为界，用序列的划分自然实现：

$\{l_0, h_1\} = \{l_0, m\} + \{m\} + \{m, h_1\}$



\* template <typename T> void quickSort(T& rank, lo, hi) {

if (hi - lo < 2) return; // 单元格区间自然有序，否则

rank[mi] = partition( {lo, hi - 1} ); // 先构造轴点，再

quickSort( {lo, mi} ); // 左侧排序

quickSort( {mi + 1, hi} ); // 右侧排序



左单峰之若瑟芬  
右花盆之维维

翠雀科

http://tiny.cc/meyarw

## 12. 排序

### 快速排序

#### 算法A

◆ 环消息：在原始序列中，轴点未分布...。

◆ 必要条件：轴点必须已然~~锁定~~ // 轴点反之不然

◆ derangement：2 3 4 ... n - 1

◆ 特别地：在有序序列中，所有元素皆为轴点；反之亦然

◆ 快速排序：就是将所有元素逐个调换为轴点的过程

◆ 好消息：通过适当~~交换~~，可使任一元素转换为轴点

◆ 问题：如何交换？成本多高？

◆ 将序列分为两个子序列： $S_1 = S_0 + S_2 + S_3 \dots / \theta(n)$

操作最小： $\max(|S_1|, |S_2|) < n$

操作独立： $\max(S_1) \leq \min(S_2)$

◆ 在子序列的递归归并排序之后，原序列自然有序

$\text{sorted}(S) = \text{sorted}(S_1) + \text{sorted}(S_2)$

◆ 平凡部：只要一个元素时，本身就是有序

\* mergesort的计算量和渐近在于~~合~~，而quicksort在于~~分~~ // 如何实现上述划分？



C.A. Hoare  
1980-1981

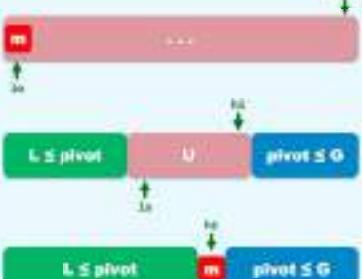
◆ 任取一枢轴值 (如 m )

◆ ① 分偏移，② 3个子序列

前偏移 L：③ 枢轴值，初始为空

后偏移 R：④ 枢轴值，初始为空

中偏移 M：⑤ 枢轴值，初始为全集





## 12. 排序

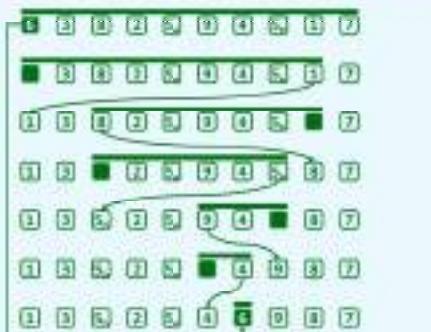
## 快速排序

性能分析(1): 基本

单链表

https://tinyurl.com/yd4gqjwz

## 不变性 + 单调性

 $lo < pivot < hi$ :  $0 \leq [lo, hi] \leq [lo][hi]$  为轴空间

## 非时间性能

swapable:  $lo/hi$  的移动方向相反, 故/看不大/小枢轴元素可能高/低 overheadin-place: 只需  $O(1)$  跳转空间overhead for recursion: 最坏情况下需要  $O(n)$  为空间greedy (smaller first) + tail\_recursion removal: 可控制在  $O(\log n)$ , Sedgewick

## 时间性能

最好情况: 每次划分都(接近)平均, 轴点总是(接近)中央

$$T(n) = 2 \cdot T\left(\frac{n-1}{2}\right) + O(n) = O(n \log n) \quad // \text{算法下界!}$$

最坏情况: 每次划分都极不均衡

// 比如, 轴点总是最小/大元素

$$T(n) = T(n-1) + T(0) + O(n) = O(n^2) \quad // \text{与冒泡排序相当!}$$

即如果用随机选取, (unlucky) 三面倒中之央的困难

也可能降低最坏情况的概率, 而无法杜绝

当然如此, 为何称作快速排序?

## 12. 排序

## 快速排序

性能分析(2): 固定策略 + 随机序列

单链表

https://tinyurl.com/yd4gqjwz

## 难题中

◆ 平均情况：最佳情况 ( $\Omega(n \log n)$ ) 退化深度  $\lambda$

平均情况 ( $O(n \log n)$ ) 退化深度  $\lambda$  提高

◆ 实际上：除非过于糟糕的 pivot，都会有对称的退化深度

$$(1 - \lambda) / 2 \quad \text{width} = \lambda = \Pr_{\pi} \quad (1 + \lambda) / 2$$

◆ 难题中：pivot 的跌落宽度为  $\text{width}$  的概率区间 // 也是这种情况出现的频率

◆ 在一进位树上，最多出现  $\log_{1/\lambda} n$  个 困难子树 ...

[Data Structures & Algorithms, Design Patterns](#)

## 12. 排序

## 快速排序

性能分析(3)：固定序列 + 随机策略

样例 3

[Data Structures & Algorithms, Design Patterns](#)

## 难题深入

◆ 每递归一步，都取  $\lambda / 1 - \lambda$  的概率 困难子树 | 困难策

◆ 输入  $\frac{1}{\lambda} \cdot \log_{1/\lambda} n$  级后，即可期望出现  $\log_{1/\lambda} n$  次 困难子树，且有退化的概率出现

◆ 相应情况的概率  $< (1 - \lambda)^{\lfloor \frac{1}{\lambda} \log_{1/\lambda} n \rfloor}$

$$(1 - \lambda) / 2 \quad \text{width} = \lambda = \Pr_{\pi} \quad (1 + \lambda) / 2$$

◆ 以  $\lambda = 1/3$  和  $n = 328$  为例，此概率  $< \left(\frac{2}{3}\right)^{21 \log_3 328} < \left(\frac{2}{3}\right)^{21 \log_3 328} < \left(\frac{2}{3}\right)^{14} < 0.45$

◆ 因此有极高的概率，递归深度不超过  $\frac{1}{\lambda} \cdot \log_{1/\lambda} n = 3 \cdot \log_2 n$

[Data Structures & Algorithms, Design Patterns](#)

## BST

◆ 随机选取 pivot，随机选择优先递归左子树

◆ quicksort 的每一次运行，都对应于一棵 BST // 假设递归实例执行次序固定，BFS 或 DFS

◆ 序列中被选取的结点，对应于全局的根节点；左、右子序列，对应于左、右子树

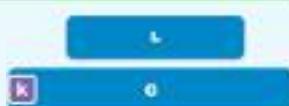
◆ 子序列中被选取的结点，对应于子树的根节点；.....



[Data Structures & Algorithms, Design Patterns](#)

## 平均性能

◆  $\Theta(n \log n)$  —— 以均匀独立分布为例...



$$\begin{aligned} T(n) &= (n+1) + \frac{1}{n} \times \sum_{k=0}^{n-1} [T(k) + T(n-k-1)] && // \text{对称} \\ &= (n+1) + \frac{2}{n} \times \sum_{k=0}^{n-1} T(k) && // \text{通过 } n \dots \end{aligned}$$

$$\begin{aligned} n \cdot T(n) &= n \cdot (n+1) + 2 \times \sum_{k=0}^{n-1} T(k) && // \text{下推图 } n-1 \dots \\ (n-1) \cdot T(n-1) &= (n-1) \cdot n + 2 \times \sum_{k=0}^{n-2} T(k) && // \text{相减 } \dots \end{aligned}$$

[Data Structures & Algorithms, Design Patterns](#)

## 难题比较次数

◆ 在 quicksort 过程中执行的比较操作次数，等于

相对应的随机序列生成该 BST 过程中，执行的比较操作次数

◆ quicksort 所有可能的运行过程中，所做比较次数的平均|期望值，等于

通过随机输入序列构造 BST 过程中，所做比较次数的平均|期望值



[Data Structures & Algorithms, Design Patterns](#)

## 平均性能

◆  $n \cdot T(n) = (n-1) \cdot T(n-1) + 2 \cdot n + 2 \times T(n-1)$  // 整理 ...

$n \cdot T(n) = 2 \cdot n + (n+1) \cdot T(n-1)$  // 除以  $n(n+1)$  ...

$$\begin{aligned} T(n)/(n+1) &= 2/(n+1) + T(n-1)/n \\ &= 2/(n+1) + 2/n + T(n-2)/(n-1) \\ &= 2/(n+1) + 2/n + 2/(n-1) + \dots + 2/2 + T(0)/1 \\ &< 2 \cdot \ln n \end{aligned}$$

$$T(n) \approx 2 \cdot n \cdot \ln n = (2 \cdot \ln 2) \cdot n \log n = \mathcal{O}(n \log n)$$

[Data Structures & Algorithms, Design Patterns](#)

## 数学期望的线性律

◆ 生成 BST 的随机序列，记作：  $(v_1 | v_2 | v_3 | \dots | v_n)$

◆ 相应的计算成本（比较操作的次数），记作：  $T = \sum_{i=1}^n \sum_{j < i} I(i, j)$

◆ 其中： $I(i, j) = \begin{cases} 1 & \text{若在插入过程中，结点 } j \text{ 被比较；或等价地， } v_j \text{ 为 } v_i \text{ 的祖先} \\ 0 & \text{否则} \end{cases}$

◆ 于是： $E(T) = \sum_{i=1}^n \sum_{j < i} E(I(i, j))$  // by Linearity of expectation  
 $= \sum_{i=1}^n \sum_{j < i} P(I(i, j) = 1)$

[Data Structures & Algorithms, Design Patterns](#)

## 平均性状

- 因此：任意一对  $v_i$  与  $v_j$ 。
- 思路：介于二者之间的所有节点 //它们存在与否，仅  $v_i$  与  $v_j$  是否相同比较无关
- 于是：作为等效的此前最后一个插入者， $v_i$  必是叶子
- 证明： $\{v_1, v_2, v_3, \dots, v_n\}$  将整个数据区间分为  $j+1$  个子区间 //无论是内部排序
- 证明： $v_i$  与  $v_j$  通过比较  $i < j$  //  $v_i$  与落在以  $v_j$  为边界的两个子区间之一
- 因此： $P(v_i = v_j) = \frac{2}{j+1}$  //从  $v_1 \rightarrow v_n$  亦是随机排列，各子区间概率均等
- 所是： $E(T) = \sum_i (\sum_{j>i} P(v_i = v_j)) = \sum_i (\sum_{j>i} \frac{2}{j+1}) = \sum_i O(\log n) = O(n \log n)$

见《算法》第 1 版第 1 章第 1 节

## 算法 81

```
template <typename T> Rank Vector<T>::partition( Rank lo, Rank hi ) { // [lo, hi]
    swap( _elem[ lo ], _elem[ lo + rand() % ( hi - lo + 1 ) ] ); // 随机交换
    T pivot = _elem[ lo ]; // 以上交换，等效于随机选取枢纽点
    while ( lo < hi ) { // 从两端交替指向中间归位，彼此对调
        while ( ( lo < hi ) && pivot >= _elem[ hi - 1 ] ) hi--; // 向左拓展
        if ( lo < hi ) _elem[ lo++ ] = _elem[ hi - 1 ]; // 凡不大于枢纽者，暂归入
        while ( ( lo < hi ) && _elem[ lo ] <= pivot ) lo++; // 向右拓展
        if ( lo < hi ) _elem[ hi-- ] = _elem[ lo ]; // 凡不小于枢纽者，暂归入
    } // assert: lo == hi
    _elem[ lo ] = pivot; return lo; // 枢纽始点归位：返回其值
}
```

## 12. 排序

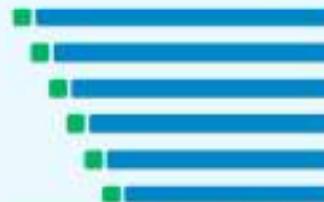
快速排序  
重复元素

孙伟群  
[dwg@zjhu.edu.cn](mailto:dwg@zjhu.edu.cn)

左见弓鹤图，右暗含呼鹤

## 重复元素

- 大量重复元素时
- 轴点位置必须挑日子
- 序序列的划分很不均匀
- 二分法归化为线性遍历
- 递归深度接近于  $O(n)$
- 运行时间依赖于  $O(n^2)$



在移动  $lo$  和  $hi$  的过程中，同时比较相邻元素：若属于相邻的重复元素，则不再深入进行。但一般情况下，如此计算量反而增加，得不偿失。

• 对算法 81 做些调整，即可解决此题——为便于对比，先给出等价形式 A1...

见《算法》第 1 版第 1 章第 1 节

## 算法 8

```
template <typename T> Rank Vector<T>::partition( Rank lo, Rank hi ) { // [lo, hi]
    swap( _elem[ lo ], _elem[ lo + rand() % ( hi - lo + 1 ) ] ); // 随机交换
    T pivot = _elem[ lo ]; // 以上交换，等效于随机选取枢纽点
    while ( lo < hi ) { // 从两端交替指向中间归位，彼此对调
        while ( lo < hi ) if ( pivot >= _elem[ hi - 1 ] ) hi--; // 向左拓展
        else { _elem[ lo++ ] = _elem[ hi - 1 ]; break; } // 将其归入
        while ( lo < hi )
            if ( _elem[ lo ] <= pivot ) lo++; // 向右拓展，直至遇到不小于枢纽者
            else { _elem[ hi-- ] = _elem[ lo ]; break; } // 将其归入
    } // assert: lo == hi
    _elem[ lo ] = pivot; return lo; // 枢纽始点归位：返回其值
}
```

## 性能

- 可以正确处理一匹情况，但性能会提升至  $O(n^2)$  级别
- 处理重复元素时
  - $lo$  和  $hi$  会交替停顿
  - 二者移动的距离大致相当
- 估计，轴点被安置于  $\lfloor (lo + hi) / 2 \rfloor$  处，实现粗分细归
- 相对于算法 A1 的易于理解、易于交换，转为易于拓展、易于交换，因此：
  - 交换操作有所增多——尤其是相同元素，在版本 A 中多不移动
  - 不稳定

```
template <typename T> Rank Vector<T>::partition( Rank lo, Rank hi ) { // [lo, hi]
    swap( _elem[ lo ], _elem[ lo + rand() % ( hi - lo + 1 ) ] ); // 随机交换
    T pivot = _elem[ lo ]; // 以上交换，等效于随机选取枢纽点
    while ( lo < hi ) { // 从两端交替指向中间归位，彼此对调
        while ( ( lo < hi ) && pivot >= _elem[ hi - 1 ] ) hi--; // 向左拓展
        if ( lo < hi ) _elem[ lo++ ] = _elem[ hi - 1 ]; // 凡不大于枢纽者，暂归入
        while ( ( lo < hi ) && _elem[ lo ] <= pivot ) lo++; // 向右拓展
        if ( lo < hi ) _elem[ hi-- ] = _elem[ lo ]; // 凡大于枢纽者，暂归入
    } // assert: lo == hi
    _elem[ lo ] = pivot; return lo; // 枢纽始点归位：返回其值
}
```

见《算法》第 1 版第 1 章第 1 节

## 12. 排序

快速排序  
变种

The great walks with the small without fear.  
The mind keeps alone.

孙伟群

[dwg@zjhu.edu.cn](mailto:dwg@zjhu.edu.cn)

## 不变性

◆ 质疑点： $S = [l, m] \rightarrow i(l, m) \rightarrow S(m, k) \rightarrow S(k, m)$   
 $l < pivot < m$



## 单调性

◆  $(k)$  不少于始点  $i$ ，而大于终点  $j$ ：② 读数前移，③ 换序

$pivot < S[i]$  :  $i++$  ;  $i = swap(S[i] \leftrightarrow m), S[i \leftrightarrow j]$



```
template <typename T> Rank<vector<T>>::partition( Rank<lo, Rank<hi> ) { // [lo, hi]
    swap( _elem[ lo ], _elem[ lo + rand() % ( hi - lo + 1 ) ] ); // 随机交换
    int pivot = _elem[ lo ]; int mi = lo;
    for ( int k = lo + 1; k < hi; k++ ) // 找到剩余待选的 k
        if ( _elem[ k ] < pivot ) // 若(k)小于始点，则将其
            swap( _elem[ ++mi ], _elem[ k ] ); // 与[mi]交换，向右扩展
    swap( _elem[ lo ], _elem[ mi ] ); // 指定锚点归位（从西向东实现）
    return mi; // 返回锚点的数
}
```

## 实例



## 12. 排序

选取

众数

中位数

https://tinyurl.com/yd2qzv4t

## 选取与中位数

◆ **k-selection** 在任意一组可比较大小的元素中，如何由小到大，找到第*k*大的？

奇偶，在这些元素的非降序序列中，找出*S[k]*

// Excel: large( range, rank )

◆ **median** 长度为*n*的有序序列*S*中，元素*S[ ⌊ n/2 ⌋ ]*称作中位数 // 数组上可能有重叠

在任意一组可比较大小的元素中，如何找到中位数？

// Excel: median( range )





◆ **中位数** *k*-选取的一个特例：最小的数，也是其中响应最大者

https://tinyurl.com/yd2qzv4t

## 众数

◆ **majority** 无序数组中，若有一半以上元素同为*m*，则称之为众数

在( 1, 3, 2, 1, 1 )中，众数为 1；然而

在( 1, 3, 2, 1, 1, + )中，却无众数

◆ **平凡算法** 排序 + 扫描

但进一步地，若限制时间不超过  $\Theta(n)$ ，则如何空间不超过  $\Theta(1)$  呢？

◆ **必要性** 众数存在，则必为中位数

◆ **事实上** 只要能够找出中位数，即不能保证它便是众数

```
template <typename T> bool majority( Vector<T> A, T & maj )
{
    return majCheck( A, maj = median( A ) );
}
```

## 必要条件

◆ **首先** 在高效的中位数算法未知之前，如何确定众数的候选呢？

◆ **node** 众数若存在，则必为众数 // Excel: mode( range )

```
template <typename T> bool majority( Vector<T> A, T & maj )
```

```
{ return majCheck( A, maj = mode( A ) ); }
```

◆ **同样地** *mode()* 算法难以兼顾时间、空间的高效

◆ **可行思维** 假设更高级计时成本更高的必要条件，通过逐一排除法

```
template <typename T> bool majority( Vector<T> A, T & maj )
```

```
{ return majCheck( A, maj = modeCandidate( A ) ); }
```

## 减而加之

◆ 若在向量A的前缀P ( |P| 为偶数) 中, 元素n 出现的次数恰占半数, 则 A 的众数位置, 对应的后缀 A - P 有众数n, 故 A 的众数



◆ 虽然这里主要考虑  $n < m$  时的情况, 故而只要考虑 A 的偶数有众数的两种情况:

1. 若  $n = m$ , 则在排除前缀 P 之后, n 与其它元素在数组上的位置(见图)完全不变。从浓度 50% 的薪水中得出 (n-1) 的一部分, 剩余部分的浓度仍为 50%;
2. 若  $n < m$ , 则在排除前缀 P 之后, n 与其它元素在数组上的位置(见图)不相同。

[Java 代码实现 & 算法题、面试题练习](#)

## 算法

```
◆ template <typename T> T majorElCandidate( Vector<T> &A ) {
    T maj; //众数候选者
    // 线性扫描: 统计计数器, 记录maj 与其它元素的数量差值
    for ( int c = 0, i = 0; i < A.size(); i++ )
        if ( 0 == c ) ( // 遇到归零, 需要将此时的前缀可以删除。
            maj = A[i]; c = 1; // 众数候选者改为新的当前元素
        ) else // 否则
            maj += A[i]; c++; c--; // 相应地更新计数器
    return maj; // 至此, 前缀的众数若存在, 则只能是maj —— 否则反之不然
}
```

◆ 将众数由  $m$  变从一半以上改作至少一半, 算法有何调整?

[Java 代码实现 & 算法题、面试题练习](#)

## 等长子向量: 构思

◆ 假设:  $s_1 = S_1[\lfloor n/2 \rfloor], s_2 = S_1[\lceil n/2 \rceil - 1] = S_1[\lfloor (n-1)/2 \rfloor]$



◆ 若  $n = m$ , 则它们同时是  $S_1, S_2$  和 S 的中位数

◆ 若  $n < m$ , 则无论 n 为奇, 偶数区间或者不是 S 的中位数; 或者与  $n_1$  或  $n_2$  因为 S 的中位数这样讲着, 剔除这些区间之后, S 中位数的数值保持不变

◆ 总之, 每经一次比较, 该问题的规模就大些简单——最终不过  $O(\log n)$

[Java 代码实现 & 算法题、面试题练习](#)

## 等长子向量: 实现

```
◆ template <typename T> // 用递归, 可证明为迭代形式
    T median( Vector<T> &S1, Set<int> &S2, Vector<T> &S3, int lo1, int hi1, int n ) {
        if ( n < 3 ) return trivialMedian( S1, lo1, n, S2, lo2, hi2 ); // 递归基
        int m1 = lo1 + n/2, m2 = hi2 + (n-1)/2; // 选定两半
        if ( S1[ m1 ] < S2[ m2 ] ) // 取 S1 左半, S2 右半
            return median( S1, m1, S2, m2, lo2, n + hi2 - m2 );
        else if ( S1[ m1 ] > S2[ m2 ] ) // 取 S1 右半, S2 左半
            return median( S1, lo1, S2, m2, hi2, n + lo2 - m1 );
        else
            return S1[ m1 ];
    }
```

[Java 代码实现 & 算法题、面试题练习](#)

## 12. 排序

## 进阶

## 中位数

华为了特别忙二件事, 在冉冉地地工作的日子快了段落。  
首先忙了, 叫名屋檐和白开了卷子来, 他们再睡觉, 就  
不能睡觉了。由你不知道忙了几次, 不说他们不懂得,  
但你没有听说过这睡觉的事一样。

早睡早起

[http://tiny.cc/meyarw](#)

## 任选子向量: 实现 (1/2)

```
template <typename T>
    T median( Vector<T> &S1, int m1, int n1, Vector<T> &S2, int m2, int n2 ) {
        if ( m1 > m2 )
            return median( S2, m2, n2, S1, m1, n1 ); // 交换 m1 <= n2
        if ( m2 < m1 )
            return trivialMedian( S1, m1, n1, S2, m2, n2 );
        if ( 2 * m1 < n2 )
            return median( S1, m1, n1, S2, m2, n2 + (n2-n1-1)/2, n1+2-(n2-n1)/2 );
        else
            return median( S1, m1, n1, S2, m2, n2 + (n2-n1-1)/2, n1+2-(n2-n1)/2 );
    }
```

[Java 代码实现 & 算法题、面试题练习](#)

## 任选子向量: 实现 (2/2)

```
int m1 = lo1 + n1/2, m2a = lo2 + (n2 - 1)/2, m2b = lo2 + n2 - 1 - n1/2;
if ( S1[ m1 ] > S2[ m2b ] ) // 取 S1 左半, S2 右半
    return median( S1, [m1], n1/2 + 1, S2, [m2b], n2 - (n1 - 1)/2 );
else if ( S1[ m1 ] < S2[ m2a ] ) // 取 S1 右半, S2 左半
    return median( S1, [m1], n1/2 + 1, S2, [m2a], n2 - n1/2 );
else // S1 没摆, S2 左右对称摆着
    return median( S1, [lo1], n1, S2, [m2a], n2 - (n1 - 1)/2 );
} //  $\Theta(\log(n_1, n_2, n_3))$  — 可见, 实际上等长版本才屈服速度大的
```

[Java 代码实现 & 算法题、面试题练习](#)

## 归并向量的中位数

◆ 任选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?

◆ 重点: 归并  $S_1$  和  $S_2$ , 得到有序向量 S

输出  $S[\lfloor (n_1 + n_2) / 2 \rfloor]$ , 即指 S 的中位数

◆ 由此, 共需  $\Theta(|S_1| + |S_2|)$  时间

◆ 这一效率很不自然, 但毕竟本题充分利用  $S_1$  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, 依然采用归并之策略...

[Java 代码实现 & 算法题、面试题练习](#)

1337

## 12. 排序

### 面试

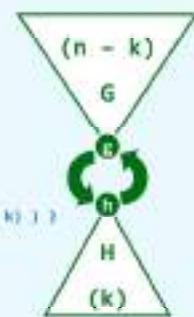
QuickSelect

世纪的译名，第所教的。在世纪数万人潮头选出来说很困难的。至于他乃谓得译者译人，译同名，译者玷辱了这两个字。

邓伟群  
dengwq@tongji.edu.cn

1341

尝试：堆 (c)

◆ ①：仅取  $k$  个元素，组织为大顶堆 //  $\Theta(k)$ ②：其余  $n - k$  个元素，组织为小顶堆 //  $\Theta(n - k)$ ◆ 技术点：比较运算 //  $\Theta(1)$ 如有必要，交换之 //  $\Theta(2 + (\log k + \log(n - k)))$ 直到：  $|k| \leq g$  //  $\Theta(\min(|k|, n - k))$ 

1338

尝试：暴力

◆ 遍历数组 //  $\Theta(n \log n)$ 从前往后遍历 //  $\Theta(k) \approx \Theta(n)$ 

1339

尝试：计数排序

1342



尝试：堆 (A)

1339

◆ 将所有元素组织为小顶堆 //  $\Theta(n)$ 直接调用  $n$  次  $\text{delMin}()$  //  $\Theta(n \log n)$ 

下界与最优

1343

◆ 是否存在更快的算法？

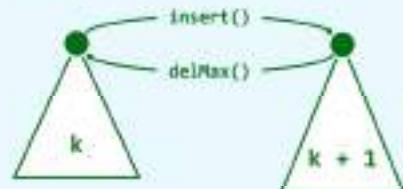
◆  $\Omega(n)$  ↑◆ 所谓弱  $\Theta$ ，指相对于序列整体而言

在访问每个元素至少一次之后，块不可忽略

◆ 反过来，是否存在  $\Theta(1)$  的算法？

尝试：堆 (n)

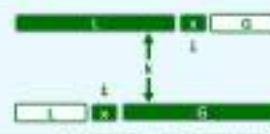
1340

◆ 任务（比例值） $k$  个元素，组织为大顶堆 //  $\Theta(k)$ 对于剩余的  $n - k$  个元素，各调用一次  $\text{insert}()$  和  $\text{delMax}()$  //  $\Theta(2^*(n - k) \log k)$ 

quickSelect()

```
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 ) { //  $\Theta(hi - lo + 1) = \Theta(n)$ 
            while ( i < j && pivot <= A[i] ) i++; A[i] = A[j];
            while ( i < j && A[j] <= pivot ) j++; A[j] = A[i];
        } // assert: i == j
        A[i] = pivot;
        if ( k < i ) hi = i - 1;
        if ( k > i ) lo = i + 1;
    } // A[k] is now a pivot
}
```

1344



## 12. 排序

## 选取

## LinearSelect

## 样例解

<http://tinyurl.com/3qg7tq>

## 复杂度

$$\Delta T(n) = \Theta(n) + \Theta(n/2) + \Theta(3n/4)$$

为使之解得线性函数，只要保证

$$n/2 + 3n/4 < n$$

或等价地

$$1/2 + 3/4 < 1$$

比如，若假定  $k=5$ ，则存在常数  $c$ ，使得

$$T(n) = cn + T(n/5) + T(3n/4)$$

$$T(n) = \Theta(20c n) = \Theta(n)$$



## linearSelect()

```

let Q be a small constant
0. If (n + |A| < Q) return trivialSelect(A, k)
1. else divide A [evenly] into  $n/Q$  subsequences (each of size  $\bar{Q}$ )
2. Sort each subsequence and determine  $n/Q$  medians //e.g. by insertion sort
3. Call linearSelect to find  $M$ , median of the medians //by recursion
4. Let  $L = E = \{x \mid \bar{Q} / 2 \leq x < n\}$ 
5. if ( $k \leq |L|$ ) return linearSelect( $L, k$ )
6. if ( $|L| < k \leq |E|$ ) return  $M$ 
7. return linearSelect( $E, k - |L| - |E|$ )

```

1346

## 12. 排序

## 希尔排序

## 框架+实例

## 样例解

<http://tinyurl.com/3qg7tq>

## 复杂度

将 linearSelect() 转换的运行时间记作  $T(n)$ 

- ◆ 第0步:  $\Theta(1) = \Theta(Q \log Q)$  //基函数: 序列长度  $|A| < Q$
- ◆ 第1步:  $\Theta(n)$  //子序列划分
- ◆ 第2步:  $\Theta(n) = \Theta(1) \times n/Q$  //子序列各自排序，并找到中位数
- ◆ 第3步:  $\Theta(n/Q)$  //从  $n/Q$  个中位数中，进行地间断全局中位数
- ◆ 第4步:  $\Theta(n)$  //分子做  $\Theta(n/Q)$ ，并分树计算——一箭双雕是真
- ◆ 第5步:  $\Theta(3n/4)$  //为什么...

## Shellsort

Donald L. Shell, 1959: 整个序列操作一个矩阵，逐列地自序  $\omega$ -sorting递减增量  $(\text{dilishing increment})$ 进阶思想：使用插屏，使用更窄，再次遍历相邻  $\omega$ -ordered进阶步骤：如此往复，直至矩阵变成一列  $\omega$ -sorting生长序列  $(\text{tree sequence})$ ：由台阶状增长而成的逐列

$$\Psi = \{M_0, M_1, M_2, M_3, \dots, M_{k-1}, \dots\}$$

正确性：最后一次迭代，等同于全排序

$$\omega\text{-ordered} = \text{sorted}$$

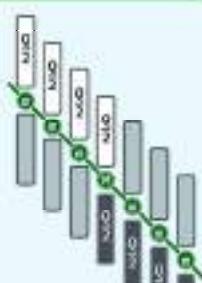
## 复杂度

在某种意义上，如上所讲矩阵必须不同不同

至少首有  $n/4$  个元素，不小于 / 不大于  $n/4$ 至少有  $n/4$  个中位数，最少半数不小于  $n/4$ 而它们又各自不小于至少  $n/2$  个元素

$$\lceil n/2 \rceil / 2 \geq \lceil n/2 \rceil = \lceil n/4 \rceil$$

$$\max(|L|, |C|) \leq 3n/4$$

实例:  $M_0 = 8$ 

89 23 19 40 85 1 18 92 71 8 96 46 12

89 23 19 40 85 1 18 92  
71 8 96 46 1271 8 19 40 12 1 18 92  
89 23 96 46 85

71 8 19 40 12 1 18 92 88 23 96 46 85

实例 :  $m_1 = 5$ 

1353

1 8 19 48 12 71 18 85 88 23 96 46 92

71	8	19	48	11
1	18	92	88	23
96	46	85		

1	8	19	48	12
71	18	85	88	23
96	46	92		

71 8 19 48 12 1 18 92 88 23 96 46 85

Data Structures &amp; Algorithms - Insertion Sort

12. 排序

希尔排序

输入敏感性

堆栈

http://www.csail.mit.edu/

实例 :  $m_1 = 3$ 

1354

1 8 19 48 12 71 18 85 88 23 96 46 92

1	8	19
48	12	71
18	85	88
23	96	46
92		

1	8	19
18	12	46
23	85	71
46	96	88
92		

1 8 19 18 12 46 23 85 71 46 96 88 92

Data Structures &amp; Algorithms - Insertion Sort

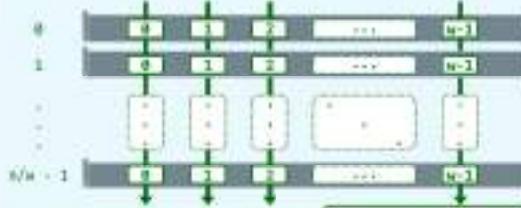
Call-by-rank

1358

如何实现归并？真非，需要使用二倍内存？实际上，借助一些向量足矣。

在每步迭代中，若当前的矩阵宽度为 $m$ ，则

$$R[1][1] \leftarrow A[(2m+1)] \quad \text{或} \quad A[k] \leftarrow R[k/m](k \% m)$$

实例 :  $m_1 = 2$ 

1355

1 8 12 18 19 48 23 46 71 88 92 85 96

1	8
19	18
12	46
23	85
71	88
96	80
92	

1	8
12	18
19	46
23	85
71	88
96	85
92	

1 8 12 18 19 48 23 85 71 46 96 88 92

Data Structures &amp; Algorithms - Insertion Sort

Input-sensitivity

1359

何等内部的排序如何实现？

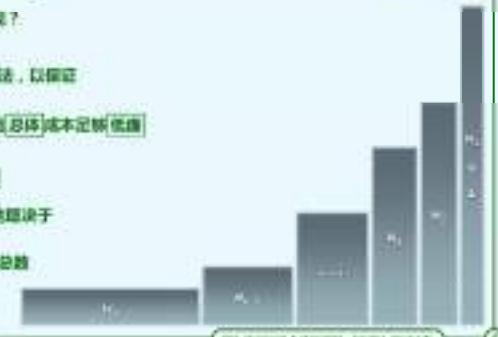
必须采用 输入敏感的算法，以保证

有序性可持恒不变，且总体成本足够低廉

比如，Insertionsort

其实实际运行时间，要多地取决于

输入序列所含 逆序对 的总数

实例 :  $m_1 = 1$ 

1356

1 8 12 18 19 48 23 46 71 88 92 85 96

1
8
12
18
19
48
23
46
71
88
92
85
96

Data Structures &amp; Algorithms - Insertion Sort

Step Sequences

1360

+ ShellSort 的总体效率

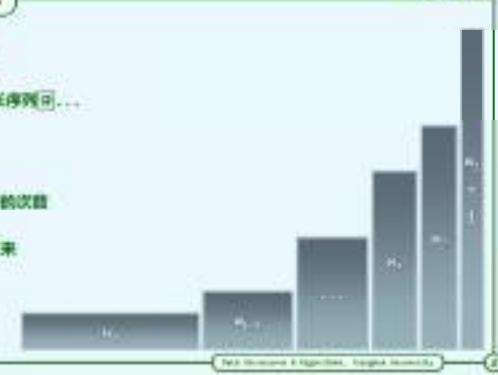
取决于具体使用何种步长序列...

+ 主要考虑和评测

1. 比较操作、移动操作的次数

2. 效率的提高，或反过来

迭代的轮数



## 12. 排序

希尔排序

Shell序列

算法课

https://tinyurl.com/2zgqfjwz

## Linear combination

◆ Let  $g, h \in \mathbb{N}$ ◆ For any  $n, m \in \mathbb{N}^*$ ,  $n \cdot g + m \cdot h$  is called a **linear combination** of  $g$  and  $h$ ◆ Denote:  $C(g, h) = \{ng + mh \mid n, m \in \mathbb{N}\}$  $N(g, h) = \mathbb{N} \setminus C(g, h)$  // numbers that are **not** combinations of  $g$  and  $h$  $x(g, h) = \max\{N(g, h)\}$  // always exists?◆ Theorem: When  $g$  and  $h$  are **relatively prime**, we have

$$x(g, h) = (g-1)(h-1) - 1 = gh - g - h$$

◆ e.g.,  $x(3, 7) = 11$ ,  $x(4, 9) = 28$ ,  $x(4, 15) = 35$ ,  $x(5, 14) = 51$ 

Data Structures &amp; Algorithms, Single Variable CS

## Shell's Sequence

◆  $\varphi_{\text{shell}} = \{1, 2, 4, 8, \dots, 2^k, \dots, 1\}$  // Shell 1959◆ 采用  $\varphi_{\text{shell}}$  在最坏情况下需要进行  $\mathcal{O}(n^2)$  次操作◆ 考虑由子序列  $A = \text{unsort}[0, 2^{k-1}]$  和  $B = \text{unsort}[2^{k-1}, 2^k]$  交错而成的：
$$11 \quad 4 \quad 18 \quad 3 \quad 18 \quad 8 \quad 25 \quad 1 \quad 9 \quad 6 \quad 8 \quad 7 \quad 13 \quad 2 \quad 12 \quad 5$$
◆ 在 **i-sorting** 阶段时， $A$  和  $B$  必然各自有序：
$$8 \quad 9 \quad 5 \quad 1 \quad 18 \quad 2 \quad 13 \quad 3 \quad 12 \quad 4 \quad 13 \quad 5 \quad 18 \quad 6 \quad 13 \quad 7$$
◆ 其中的逆序对仍然很多，**i-sorting** 仍需  $1 + 2 + \dots + 2^{k-1} = \mathcal{O}(n^2)$  时间◆ 根据定义， $\varphi_{\text{shell}}$  中各项并不互素，甚至相邻项也互互素

## 12. 排序

希尔排序

逆序对

算法课

https://tinyurl.com/2zgqfjwz

## h-sorting &amp; h-ordered

◆ Let  $n = N$ . A sequence  $\langle g[0, n] \rangle$  is called **h-ordered** if
$$\{g[i] \leq g[i+h]\} \text{ holds for } 0 \leq i \leq n-h$$
◆ A **1-ordered** sequence is sorted◆ **h-sorting**: an h-ordered sequence is obtained by

- arranging  $S$  into a 2D matrix with  $\lceil \frac{n}{h} \rceil$  columns and
- sorting each column respectively



## Theorem K

◆ [Knuth, ACP Vol.3 p.99]

//习题解析[12-12, 12-13]

A **g-ordered** sequence **REMAINS** g-ordered after being **h-sorted**.

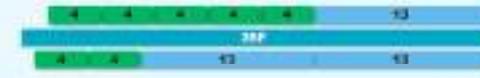
## Postage Problem

◆ The postage for a letter is 28F, and a postcard is 35F.

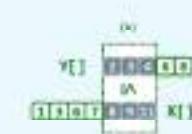
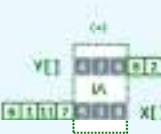
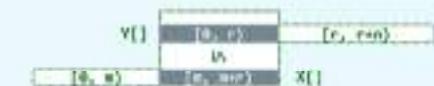
But there are only stamps of 1F and 13F available.

◆ Possible to stamp the letter and the postcard **EXACTLY**?

◆ How about other postages?

◆ For each postage  $P$ , determine whether  $P \in \{n \cdot 1 + m \cdot 13 \mid n, m \in \mathbb{N}\}$ 

## Lemma L



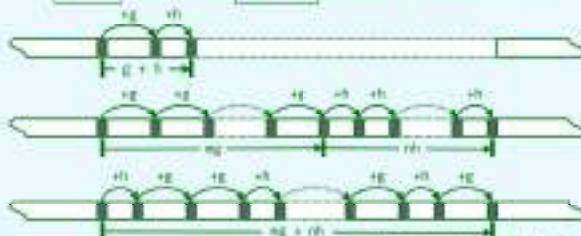
Data Structures &amp; Algorithms, Single Variable CS

**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  $(hg+rh)$ -ordered for any  $n, m \in \mathbb{N}$



See [Introduction to Algorithms, Integer Sequences](#)

**Inversion**

⊕ Let  $S[0, n]$  be a  $(g, h)$ -ordered sequence, where  $g \& h$  are relatively prime

⊕ Then for all elements  $S[1]$  and  $S[2]$ , we have

$$j-i \geq \text{sg}(g, h) + 1 = (g-1)(h-1) \quad \text{only if} \quad S[i] \leq S[j]$$

⊕ This implies that to the **right** of each element,

only the next  $\text{sg}(g, h)$  elements could be smaller



⊕ There would be no more than  $n \cdot \text{sg}(g, h)$  inversions altogether.

**12. 排序****希尔排序****PS序列**

见 [\[1\]](#)

[tinyurl.com/ps-seq](http://tinyurl.com/ps-seq)

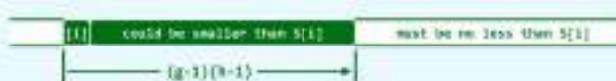
**d-Sorting an  $\mathcal{O}(d)$ -Ordered Sequence in  $\mathcal{O}(dn)$  Time**

⊕ If  $[g]$  and  $[h]$  are relatively prime and are both in  $\mathcal{O}(d)$

we can d-sort the sequence in  $\mathcal{O}(dn)$  time ...

- re-arrange the sequence as a 2D matrix with  $[g]$  columns
- each element is swapped with  $O((g-1) \cdot (h-1)/d) = O(d)$  elements

⊕ since this holds for all elements,  $\mathcal{O}(dn)$  steps are enough



See [Introduction to Algorithms, Integer Sequences](#)

**PS Sequence**

⊕ Paperov & Stasevic, 1965

// also called Hibbard's sequence

$$\text{Key} = \text{P}_{\text{Hib}} - 1 = \{2^k - 1 \mid k \in \mathbb{N}\} = \{1, 3, 7, 15, 31, 63, 127, 255, \dots\}$$

⊕ different items **may not** be relatively prime, e.g.,  $b_{10} = b_5 \cdot (b_2 + 2)$

⊕ adjacent items **must** be, since  $b_{i+1} - 2 \cdot b_i = 1$

⊕ ShellSort with  $P_n$  needs:

-  $\mathcal{O}(\log n)$  outer iterations and

-  $\mathcal{O}(n^{1/2})$  time to sort a sequence of length  $n$

// why ...

See [Introduction to Algorithms, Integer Sequences](#)

 **$t < k$** 

⊕ Let  $b_t$  be the  $b$  closest to  $\sqrt{n}$  and hence  $b_t \approx \sqrt{n} = \mathcal{O}(n^{1/2})$

$b_t < t$

1) Consider those iterations for  $\{b_k \mid t < k\} = \{b_{t+1}, b_{t+2}, \dots, b_n\}$

$b_t < b_k$

∴ there would be  $\mathcal{O}(n/b_t)$  elements in each of the  $b_t$  columns

∴ we can **insertionsort** each column in  $\mathcal{O}(n^2/b_t)$  time

2) each  $b_t$ -sorting costs  $\mathcal{O}(n^2/b_t)$  time

3) all these iterations cost time of

$$\mathcal{O}(2 \times n^2/b_t) = \mathcal{O}(n^{3/2})$$

$t < k$

$b_t < b_k$

See [Introduction to Algorithms, Integer Sequences](#)

$k < t$

$b_t < b_k$

$k < t$

$R_{\text{pratt}} = \{2^p \cdot 3^q \mid p, q \in \mathbb{N}\} = \{1, 2, 3, 4, 6, 8, 9, 12, 16, \dots\}$

With  $R_{\text{pratt}}$ :

- shellsort sorts a sequence of length  $n$  in  $O(\log^2 n)$  time. //why?
- And, wait a minute, but ...
- adjacent numbers in  $R_{\text{pratt}}$
- are NOT all relatively prime ...

One-dimensional algorithms, Integer sequences

$\Theta(n \log n)$

& from  $(2, 3)$ -ordered to  $(1)$ -ordered

- $x(2, 3) = [1]$
- to the right of each element in a  $(2, 3)$ -ordered sequence,  
only the next element can be smaller
- it costs  $\Theta(n)$  time to sort such a sequence



& from  $(2n_1, 3n_1)$ -ordered to  $(1)$ -ordered

- divide 5 into 5 subsequences, each of which is  $(2, 3)$ -ordered
- it costs altogether  $\Theta(n)$  time to sort them resp.
- there are altogether  $\Theta(\log n)$  iterations //why
- we need time  $\Theta(n) + \Theta(\log n) = \Theta(n \log n)$

One-dimensional algorithms, Integer sequences

## 12. 排序

希尔排序

Sedgewick序列

见注释

[sedg@csail.mit.edu](mailto:sedg@csail.mit.edu)

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 Pratt's sequence with Pratt's

$$\{1, 3, 19, 41, 109, 309, 929, 2901, 2903, 8929, 16801, 36289, 61789, \dots\}$$

$$\{2 \times 3^k - 3 \times 2^k + 1 \mid k \geq 0\} \cup \{4^k - 3 \times 2^k + 1 \mid k \geq 1\}$$

- worst  $\Theta(n^{4/3})$  & average  $\Theta(n^{1/3})$
- best performance in practice

& Is there a step-sequence with  $\Theta(n \log n)$  worst-case performance?

One-dimensional algorithms, Integer sequences

