目录

[第一章 绪论 2](#_Toc522720854)

[1.1 计算机与算法 2](#_Toc522720855)

[1.2复杂度度量 3](#_Toc522720856)

[1.3 复杂度分析 4](#_Toc522720857)

[1.4 递归 5](#_Toc522720858)

[1.5 抽象数据类型 6](#_Toc522720859)

[第二章 向量（Vector） 6](#_Toc522720860)

[2.1 从数组到向量 6](#_Toc522720861)

[2.2 接口 6](#_Toc522720862)

[2.3 构造与析构 8](#_Toc522720863)

[2.4 动态空间管理 8](#_Toc522720864)

[2.5 常规向量 9](#_Toc522720865)

[2.6 有序向量 10](#_Toc522720866)

[2.7 排序与下界 10](#_Toc522720867)

[2.8 排序器 11](#_Toc522720868)

[第三章 列表（List） 11](#_Toc522720869)

[3.1 从向量到列表 11](#_Toc522720870)

[3.2 接口 12](#_Toc522720871)

[3.3 列表 13](#_Toc522720872)

[3.4 有序列表 13](#_Toc522720873)

[3.5 排序器 13](#_Toc522720874)

[第四章 栈与队列 13](#_Toc522720875)

[4.1 栈（Stack） 13](#_Toc522720876)

[4.2 栈与递归 14](#_Toc522720877)

[4.3 栈的典型应用 14](#_Toc522720878)

[4.4 试探回溯法 14](#_Toc522720879)

[4.5 队列（Queue） 15](#_Toc522720880)

[4.6 队列应用 15](#_Toc522720881)

[第五章 二叉树 15](#_Toc522720882)

[5.1 二叉树及其表示 15](#_Toc522720883)

[5.2 编码树 16](#_Toc522720884)

[5.3 二叉树的实现 16](#_Toc522720885)

[5.4 遍历（traversal） 17](#_Toc522720886)

[5.5 Huffman编码 17](#_Toc522720887)

[第六章 图 18](#_Toc522720888)

[6.1 概述 18](#_Toc522720889)

[6.2 抽象数据类型 19](#_Toc522720890)

[6.3 邻接矩阵（adjacency matrix） 19](#_Toc522720891)

[6.4 邻接表（adjacency list） 20](#_Toc522720892)

[6.5 图遍历算法概述 20](#_Toc522720893)

[6.6 广度优先策略（Breadth-first search, BFS） 20](#_Toc522720894)

[6.7 深度优先搜索（Depth-First Search, DFS） 20](#_Toc522720895)

[6.8 拓扑排序 21](#_Toc522720896)

[6.9 双连通域分解 21](#_Toc522720897)

[6.10 优先级搜索 22](#_Toc522720898)

[6.11 最小支撑树 22](#_Toc522720899)

[6.12 最短路径 22](#_Toc522720900)

[第七章 搜索树 23](#_Toc522720901)

[7.1 查找 23](#_Toc522720902)

[7.2 二叉搜索树（binary search tree） 23](#_Toc522720903)

[7.3 平衡二叉搜索树（balanced binary search tree, BBST） 24](#_Toc522720904)

[7.4 AVL树 24](#_Toc522720905)

[第八章 高级搜索树 25](#_Toc522720906)

[8.1 伸展树（splay tree） 25](#_Toc522720907)

[8.2 B-树 26](#_Toc522720908)

[8.3 红黑树（red-black tree） 28](#_Toc522720909)

[8.4 kd-树（kd-tree） 29](#_Toc522720910)

[第九章 词典（Dictionary） 30](#_Toc522720911)

[9.1 词典ADT 30](#_Toc522720912)

[9.2 跳转表（skip list） 30](#_Toc522720913)

[9.3 散列表（hashtable） 31](#_Toc522720914)

[9.4 散列应用 34](#_Toc522720915)

[第十章 优先级队列（Priority Queue） 35](#_Toc522720916)

[10.1 优先级队列ADT 35](#_Toc522720917)

[10.2 堆（heap） 36](#_Toc522720918)

[10.3 左式堆（leftist heap） 37](#_Toc522720919)

[第十一章 串（String） 38](#_Toc522720920)

[11.1 串及串匹配 38](#_Toc522720921)

[11.2 蛮力算法 38](#_Toc522720922)

[11.3 KMP算法 39](#_Toc522720923)

[11.4 BM算法 40](#_Toc522720924)

[11.5 Karp-Rabin算法 41](#_Toc522720925)

[第十二章 排序 41](#_Toc522720926)

[12.1 快速排序（quicksort） 41](#_Toc522720927)

[12.2 选取与中位数 42](#_Toc522720928)

[12.3 希尔排序（Shellsort） 43](#_Toc522720929)

# 第一章 绪论

## 1.1 计算机与算法

1.1.1 古埃及人的绳索

1.1.2 欧几里得的尺规

1.1.3 起泡排序（sorting）

①局部有序与整体有序

②扫描交换

③起泡排序（bubblesort）

④实现

1.1.4 算法

•算法：基于特定的计算模型，旨在解决某一信息处理问题而设计的一个指令序列，必须具备以下要素

①输入与输出

②基本操作、确定性与可行性

•算法应可描述为由若干语义明确的基本操作所组成的指令序列，且每一基本操作在对应的计算模型中均可兑现。

③有穷性与正确性（finiteness和correctness）

•证明算法有穷性和正确性的一个重要技巧就是从适当的角度审视整个计算过程，并找出其中所具有的某种不变形与单调性。

•单调性：问题的有效规模会随着算法的推进不断递减

•不变性：从算法的初始状态到最终的正确性，算法存在着一种稳定的状况，随着算法进行这种稳定状况逐渐扩大到全局成为正确性

④退化（degeneracy）与鲁棒性（robustness）

•退化情况：非一般性输入实例，或各种极端的输入实例

•鲁棒性：应对极端异常情况仍能正常返回不致出现异常

⑤重用性：算法框架可以推广至其他场合，比如可以适用于不同类型的数据

1.1.5 算法效率

①可计算性（computability）

②难解性（intractability）

•不可计算或者不可解，即没有必然终止的算法

•难解，即虽然满足有穷性，但是花费的时间成本太高

③计算效率

•非“不可解和难解”的问题，称为一般性问题，同样具有一种效率属性

④数据结构

•定义：以“数据”这一信息的表现形式为研究对象，旨在建立支持高效算法的数据信息处理策略、技巧与方法。

## 1.2复杂度度量

•封底估算（back-of-the-envelope calculation）：直译为信封底部即可进行的简单估算，指根据数据结构和算法的渐进复杂度，凭借在实际计算环境中积累的经验，针对计算过程主要部分进行的粗略估算。

1.2.1 时间复杂度

•时间复杂度（time complexity）：执行时间（T(n)）的变化趋势可表示为输入规模（n）的一个函数，称作该算法的时间复杂度；在规模为n的所有输入中，选择执行时间最长的为T(n)，作为该算法的时间复杂度

1.2.2 渐进复杂度

•渐进分析（asymptotic analysis）：着眼长远，更为注重时间复杂度总体变化趋势和增长速度的策略与方法；不纠结于小规模问题的处理效率比较

①大O记号（big-O notation）

•若存在正的常数c和函数f(n)，使得对任何n>>2都有T(n)<=c\*f(n)，则可认为n足够大之后，f(n)给出了T(n)增长速度的一个渐进上界，即为T(n)=O(f(n))

性质1：对于任意常数c>0，有O(f(n))=O(c\*f(n)) //忽略常数系数

性质2：对任意常数a>b>0，有O(na+nb)=O(na) //忽略多项式低次项

②环境差异

•有必要按照超脱于具体硬件平台和软件环境的某一客观标准来度量时间复杂度

③基本操作

•基本操作：算术运算，比较，分支，子程序调用与返回；大多数实际计算环境中，每一次这类基本操作都可以在常数时间内完成

•一般将T(n)定义为算法所执行基本操作的总次数

•基本操作种类：

a.从特定单元读取元素一个

b.输出元素一个

c.比较大小一次

d.算数运算，位运算一次

④最坏最好与平均情况

•算法执行时间的最复杂最简单和最准确的估计，分别用大O，大Ω，大Θ表示

⑤大Ω记号

•若存在正的常数c和函数g(n) ，使得对任何n>>2都有T(n)>=c\*g(n)，则可认为n足够大之后，g(n)给出了T(n)增长速度的一个渐进下界，即为T(n)=Ω(g(n))

⑥大Θ记号

•若存在正的常数c1<c2和函数h(n) ，使得对任何n>>2都有c1\*h(n)<=T(n)<=c2\*h(n)，则可认为n足够大之后，h(n)给出了T(n)增长速度的一个确界，即为T(n)= Θ(h(n))；对于规模为n的任何输入，算法的运行时间都与Θ(h(n))同阶

1.2.3空间复杂度（space complexity）

•除非特别声明，空间复杂度通常不计入原始输入本身占用的空间，其他的如转储、中转、索引、映射、缓冲等消耗的空间都应计入。

•算法所需存储空间；估计方法与时间复杂度类似；一般情况下，只考虑时间复杂度；时间复杂度是空间复杂度的天然上界（因为每次基本操作所占用或访问的存储空间不一定都是新开辟的，存在着可回收利用性，但是时间不存在回收利用性）；不过有时仍需考虑空间复杂度，比如对于空间效率非常在乎的场合。

## 1.3 复杂度分析

1.3.1常数O(1)

•常数时间复杂度算法（constant-time algorithm）：最为理想的算法，一般仅含一次或常数次基本操作的算法，一般不含循环，分支，子程序调用等

•就地算法（in-place algorithm）：仅需O(1)辅助空间的算法

•通常O(loglogn)可以视为常数复杂度，因为这个函数的增长极其缓慢

1.3.2对数O(logn)

•对数时间复杂度（logarithmic-time algorithm）：略

•对数多项式时间复杂度（polylogarithmic-time algorithm）：即O(logcn)，效率虽然不如常数复杂度理想，不过从多项式的角度看仍能无限接近于后者，也是极为高效的

•对数复杂度的常底数具体取值无所谓，都写作O(logn)

•对于任何ε>0，都有logn=O(nε)，因此对数复杂度一定比线性复杂度好

1.3.3线性O(n)

•线性时间复杂度（linear-time algorithm）：略；效率亦足以令人满意

1.3.4多项式O(polynomial(n))

•多项式时间复杂度（polynomial-time algorithm）：即O(f(n))，f(x)为多项式；线性复杂度是其特例；效率一般被认为是可接受的或可忍受的，而被解决的问题是可有效求解或易解的（tractable）

1.3.5指数O(2n)

•指数时间复杂度算法（exponential-time algorithm）：效率急剧降低，难以忍受

•n! = O(nn)

1.3.6 复杂度层次

•复杂度层次：由低至高O(1), O(log\*n), O(loglogn), O(logn), O(sqrt(n)), O(n), O(nlog\*n), O(nloglogn), O(nlogn), O(n2), O(n3), O(nc), O(2n)

1.3.7 输入规模

•输入规模：输入规模的界定应“用以描述输入所需的空间规模”，一般用输入参数的二进制展开宽度代替参数本身的数值更为合理；用数值本身为输入规模得出复杂度，称为“伪……复杂度”

## 1.4 递归

•C++语言中，递归调用（recursive call）就是某一方法调用自身，递归算法一般都是非常简洁高效的算法

1.4.1 线性递归

•递归基（base case of recursion）：首先判断并处理n=0之类的平凡情况，以免因无限递归而导致系统溢出，这些平凡情况即为递归基；递归基可能有多种，至少为一种

•线性递归（linear recursion）：每一递归实例对自身的调用至多一次，每一层次上至多只有一个实例，且构成一个线性的次序关系；一般问题可分解为两个独立子问题，一个是直接求解的，另一个与结构与原问题相同，二者经过简单合并（比如相加）即为总解；

•特点是减而治之（decrease-and-conquer），递归每深入一层，待求解问题的规模都缩减一个常数，直至最终蜕化为平凡的小（简单）问题。即调用参数单调线性递减，调用次数一定有限，抵达递归基时，执行非递归运算。

1.4.2 递归分析

•递归分析：计算递归算法的时间复杂度

①递归跟踪（recursion trace） p38/17

•将递归算法的执行过程整理为图的形式

②递推方程（recurrence equation）p40/19

•根据条件和情况，得到T(n)的递推方程组，然后根据由递归基确定的边界条件，求解T(n)表达式，进而可以得到复杂度。

1.4.3 递归模式

①多递归基

•显式或隐式地同一算法有多个递归基，往往是因为具体实际情况不同，最终的平凡情况有不止一种可能性。

②多向递归

•递归公式在每一步递归时可能存在分歧，但是每步仍然只有一种选择，因此仍为线性递归

1.4.4 递归消除

•递归消除即改写递归语句使之成为等效的非递归版本，递归算法虽然精巧但是对空间成本往往消耗更多。

①尾递归及其消除

•尾递归（tail recursion）即递归调用在递归实例中以最后一步操作的形式出现。属于尾递归形式的算法，均可以简单的转换为等效的迭代版本。

1.4.5 二分递归

①分而治之

•特点是分而治之（divide-and-conquer），每一递归实例都可能做多次递归，又称多路递归（multi-way recursion）。无论分解为2个或更多子问题，对算法总体渐进复杂度并无实质影响。

•递归深度：任意时刻的活跃递归实例的总数。

•对于二分递归，一般来说，对每一个递归实例都是单独依次处理的，而不是多层并行处理；当一个递归实例是递归基时，完成计算后该递归实例便消逝，再继续执行其他递归实例。在递归跟踪中，同一时间每一层只有一个递归实例被激活。因此时间和空间复杂度的计算都必须依照此种方式。

②效率

•为了保证效率：

a.分解问题和合并结果的过程可以高效实现

b.子问题之间相互独立，没有额外造成复杂度增加

③优化策略

•制表或记忆（tabulation/memorization）：借助一定量的辅助空间，在各子问题求解之后，及时记录下其对应的解答。每遇到一个子问题，首先检验是否已经计算过。

•动态规划（dynamic programming）：从递归基出发，自底而上递推得到解答。不一定使用递归算法。

## 1.5 抽象数据类型

•抽象数据类型（abstract data type，ADT）即被提供一致且标准的对外接口的数据集合的类型。是面向对象程序设计的基础。

# 第二章 向量（Vector）

•数据结构按照逻辑次序分类：

①线性结构:各数据项按照一个线性次序构成一个整体;最基本的线性结构统称为序列(sequence),进一步分为向量(vector)与列表(list)。向量的数据物理位置与逻辑次序吻合，此时逻辑次序称为秩（rank）。

②半线性结构

③非线性结构

## 2.1 从数组到向量

2.1.1 数组

•所有某元素之前的元素都称为前驱（predecessor），之后的元素叫后继（successor），最近的前后元素叫直接前驱/后继（intermediate predecessor/successor）；所有前驱称为前缀（prefix），所有后继称为（suffix）

•一般的数组每个元素占据相同的空间，并且线性排列，也称为线性数组（linear array）

2.1.2 向量

•向量是线性数组的一种抽象与泛化，向量特有的元素访问方式叫做循秩访问（call-by-rank），因为秩和元素线性一一对应。

•向量虽然不限制元素的类型，可以是基本类型也可以是自定义类型，但是各个元素必须是同一个类型。

## 2.2 接口

2.2.1 ADT接口

•表2.1 向量ADT支持的操作接口

|  |  |  |
| --- | --- | --- |
| 操作接口 | 功能 | 适用对象 |
| size() | 报告向量当前的规模（元素总数） | 向量 |
| get(r) | 获取秩为r的元素 | 向量 |
| put(r,e) | 用e替换秩为r元素的数值 | 向量 |
| insert(r,e) | e作为秩为r元素插入，原后继元素依次后移 | 向量 |
| remove(r) | 删除秩为r的元素，迒回该元素中原存放的对象 | 向量 |
| disordered() | 刞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) | 查找等亍e且秩最大的元素 | 向量 |
| search(e) | 查找目标元素e，返回不大于e且秩最大的元素 | 有序向量 |
| deduplicate() | 剔除重复元素 | 向量 |
| uniquify() | 剔除重复元素 | 有序向量 |
| traverse() | 遍历向量并统一处理所有元素，处理方法由函数对象指定 | 向量 |

2.2.2 操作实例

2.2.3 Vector模板类

|  |
| --- |
| 1 **typedef int** Rank; //秩 |
| 2 **#define** DEFAULT\_CAPACITY 3 //默认的初始容量（实际应用中可设置为更大） |
| 3 |
| 4 **template** <**typename** T> **class** Vector { //向量模板类 |
| 5 **protected:** |
| 6 Rank \_size; **int** \_capacity; T\* \_elem; //规模、容量、数据区 |
| 7 **void** copyFrom(T **const\*** A, Rank lo, Rank hi); //复制数组区间A[lo, hi) |
| 8 **void** expand(); //空间不足时扩容 |
| 9 **void** shrink(); //装填因子过小时压缩 |
| 10 **bool** bubble(Rank lo, Rank hi); //扫描交换 |
| 11 **void** bubbleSort(Rank lo, Rank hi); //起泡排序算法 |
| 12 Rank max(Rank lo, Rank hi); //选取最大元素 |
| 13 **void** selectionSort(Rank lo, Rank hi); //选择排序算法 |
| 14 **void** merge(Rank lo, Rank mi, Rank hi); //归并算法 |
| 15 **void** mergeSort(Rank lo, Rank hi); //归并排序算法 |
| 16 Rank partition(Rank lo, Rank hi); //轴点构造算法 |
| 17 **void** quickSort(Rank lo, Rank hi); //快速排序算法 |
| 18 **void** heapSort(Rank lo, Rank hi); //堆排序（稍后结合完全堆讲解） |
| 19 **public:** |
| 20 // 极造函数 |
| 21 Vector(int c = DEFAULT\_CAPACITY, **int** s = 0, T v = 0) //容量为c、规模为s、所有元素初始为v |
| 22 { \_elem = **new** T[\_capacity = c]; **for** (\_size = 0; \_size < s; \_elem[\_size++] = v); } //s <= c |
| 23 Vector(T **const\*** A, Rank lo, Rank hi) { copyFrom(A, lo, hi); } //数组区间复制 |
| 24 Vector(T **const\*** A, Rank n) { copyFrom(A, 0, n); } //数组整体复制 |
| 25 Vector(Vector<T> **const&** V, Rank lo, Rank hi) { copyFrom(V.\_elem, lo, hi); } //向量区间复制 |
| 26 Vector(Vector<T> **const&** V) { copyFrom(V.\_elem, 0, V.\_size); } //向量整体复制 |
| 27 // 析构函数 |
| 28 ~Vector() { **delete** [] \_elem; } //释放内部空间 |
| 29 // 只读访问接口 |
| 30 Rank size() **const** { **return** \_size; } //规模 |
| 31 **bool** empty() **const** { **return** !\_size; } //判空 |
| 32 **int** disordered() **const;** //判断向量是否已排序 |
| 33 Rank find(T **const&** e) **const** { **return** find(e, 0, \_size); } //无序向量整体查找 |
| 34 Rank find(T **const&** e, Rank lo, Rank hi) **const;** //无序向量区间查找 |
| 35 Rank search(T **const&** e) **const** //有序向量整体查找 |
| 36 { **return** (0 >= \_size) ? -1 : search(e, 0, \_size); } |
| 37 Rank search(T **const&** e, Rank lo, Rank hi) **const;** //有序向量区间查找 |
| 38 // 可写访问接口 |
| 39 T& **operator[](Rank** r) **const;** //重载下标操作符，可以类似于数组形式引用各元素 |
| 40 Vector<T> & **operator=(Vector<T> const&);** //重载赋值操作符，以便直接克隆向量 |
| 41 T remove(Rank **r);** //删除秩为r的元素 |
| 42 **int** remove(Rank lo, Rank hi); //删除秩在区间[lo, hi)之内的元素 |
| 43 Rank insert(Rank r, T **const&** e); //插入元素 |
| 44 Rank insert(T **const&** e) { **return** insert(\_size, e); } //默认作为末元素插入 |
| 45 **void** sort(Rank lo, Rank hi); //对[lo, hi)排序 |
| 46 **void** sort() { sort(0, \_size); } //整体排序 |
| 47 **void** unsort(Rank lo, Rank hi); //对[lo, hi)置乱 |
| 48 **void** unsort() { unsort(0, \_size); } //整体置乱 |
| 49 **int** deduplicate(); //无序去重 |
| 50 **int** uniquify(); //有序去重 |
| 51 // 遍历 |
| 52 **void** traverse(void (\*)(T&)); //遍历（使用函数指针，只读或局部性修改） |
| 53 **template** <**typename** VST> **void** traverse(VST&); //遍历（使用函数对象，可全局性修改） |
| 54 }; //Vector |

•向量的扩充接口：判空接口empty(), 区间删除remove(lo,hi),区间查找 find(e,lo,hi)，是由基本接口的扩展或变型。

## 2.3 构造与析构

•向量结构在内部维护一个元素类型为T的私有数组\_elem[]，数组容量由私有变量\_capacity表示，有效元素的数量（即向量当前的实际规模），由\_size指示。

2.3.1 默认构造方法

2.3.2 基于复制的构造方法

2.3.3 析构方法

## 2.4 动态空间管理

2.4.1 静态空间管理

•无法调整向量占用空间，即静态空间管理策略，效率很低，容易造成上溢(overflow)

•向量实际规模与其内部数组容量的比值（\_size/\_capacity），称作装填因子（load factor），是衡量空间利用率的重要指标，也可以直观理解为满载程度

2.4.2 可扩充向量（extendable vector）

•可行策略（算法）：当向量原有的空间满时，重新申请一个更大新空间，将原空间中的向量复制过来，并释放原空间，之后在新空间中操作。

2.4.3 扩容

•调用insert()方法前，都必须先调用扩容方法，如果\_size<\_capacity就直接返回。

•新数组的容量总是取原数组的两倍

2.4.4 分摊分析

①时间代价

②分摊复杂度

•分摊运行时间（amortized running time）：对某一对象进行足够多次连续操作，并将其间消耗的时间，平均分摊至所有操作。

•平均运行时间（average running time）：按照某种假定的概率分布，对各种情况下所需的执行时间的加权平均，也称期望运行时间（expected running time）

③O(1)分摊时间（计算方法p35/56）

④其他扩容策略

•早期扩容策略采用追加固定数目的单元，这种策略无论采用的固定常数多大，在最坏情况下，单次操作分摊时间复杂度高达Ω(n)

2.4.5 缩容

•当装填因子低于某一阈值时，数组可称为发生下溢（underflow）

## 2.5 常规向量

2.5.1 直接引用元素

2.5.2 置乱器

2.5.3 判等器与比较器

•比对：判断两个对象是否相等

•比较：判断两个对象的相对大小

2.5.4 无序查找

①判等器

•无序向量（unsorted vector）：仅支持比对，未必支持比较的向量

②顺序查找（sequential search）

•即依次逐个比对的查找方式

③复杂度

•复杂度为O(n)，但是对于规模相同，内部组成不同的输入，渐进运行时间有本质区别，此类算法称为输入敏感的算法（input sensitive）

2.5.5 插入

2.5.6 删除

①区间删除

②单元素删除

•单元素删除是区间删除的特例，而不能认为或者试图用单元素删除反过来实现区间删除，虽然方法可行，但是删除之后伴随着后继元素移动的操作，显然对于一般的区间删除，单元素删除的移动操作效率较低。

③复杂度

•删除操作实际上是被删除区间的后继元素向前移动覆盖，并没有严格意义上的删除操作，因为\_size变量随后更新，即便前移后有一部分空间还有残留数据，但是也不再是向量实际规模的一部分了。所以删除操作花费的时间全部在于移动后继元素。

④错误及意外处理

•操作接口对于输入参数有一定的限制，如果出现例如超出合法范围的问题，称为典型错误（error）或意外（exception）。可以将入口参数合法性检查统一交由上层调用例程。

2.5.7 唯一化

2.5.8 遍历

## 2.6 有序向量

•向量中的所有元素不仅按线性次序存放，而且其数值大小也按此次序单调分布，称作有序向量（sorted vector）。通常约定其中的元素自前向后或自左向右构成一个非降序列

2.6.1 比较器

2.6.2 有序性甄别

•有序向量因为元素之间的有序性，使得查找和唯一化等操作可以采用效率更高的算法，因此有必要在采用高效接口之前检验有序性

2.6.3 唯一化

2.6.4 查找

•有序查找的接口命名为search()，与无序查找的find()区别

2.6.5 二分查找（版本A）

①减而治之

②查找长度

推导过程：p51/72

•对于向量元素的类型，不一定都是基本类型，可能是非常复杂的自定义类型，这些数据的比较可能未必能在常数时间内完成。因此比较向量元素大小的操作次数，即查找长度（search length）是对于查找算法的性能评估很重要的一个因素

③不足

2.6.6 Fibonacci查找

•和二分查找原理类似，只是每次不平均划分子向量，而是按照黄金分割比例，以平衡前后两个子向量中所需的总查找长度

查找长度推导：p54/75

2.6.7 二分查找（版本B）

2.6.8 二分查找（版本C）

•使用版本C的查找算法，可以为有序向量的插入操作提供便捷。一般有序向量的插入不再是以插入位置的秩为要求，因为向量的有序性，对于特定的值可行的插入位置是有限的，并且也要尽量减少插入操作造成的元素后移。另外也需要在查找失败时，返回一个合适的位置以便插入对应值的元素。

## 2.7 排序与下界

2.7.1 有序性

•因为有序向量的算法效率较高，因此解决很多问题时，先排序再调用其他操作

2.7.2 排序及其分类

①内部排序和外部排序：内存中即可处理的数据规模|内存中只能容纳数据的一小部分

②离线（脱机）和在线：待排序数据以整体形式存在|数据需要实时生成，算法启动后数据才陆续到达

③串行和并行

④确定式和随机式

2.7.3 下界

•复杂度下界（lower bound）：任一问题在最坏情况下的最低计算成本，如果某一算法达到这个下界，意味着它已经是最坏情况下最优的（worst-case optimal）

2.7.4 比较树

•性质定义：

①每一内部节点各对应于一次比对操作；

②内部节点的左、右分支，分别对应于在两种比对结果下的执行方向；

③叶节点（或等效地，根到叶节点的路径）对应于算法某次执行的完整过程及输出；

④反过来，算法的每一运行过程都对应于从根到某一叶节点的路径。

按上述规则与算法相对应的树，称作比较树（comparison tree）。算法称为基于比较式算法（comparison-based algorithm），简称CBA式算法。

2.7.5 估计下界

①最小树高

•叶节点的深度：对应叶节点和根节点的距离

•树的高度：比较树中所有叶节点的最大深度

•若某一问题的输出结果不少于N种，那么叶节点也不能少于N，树高不能低于log2N

②CBA式排序

•n个元素的排序输出有n！种，则最小树高为h>=log3n!=Ω(nlogn)

## 2.8 排序器

2.8.1 统一入口

2.8.2 起泡排序

①重复元素与稳定性

•将向量A转换为有序向量S之后，设A[i]对应于S[ki]。若对于A中每一对重复元素A[i]=A[j]（相应的S[ki]=S[kj]），都有i<j当且仅当ki<kj，即重复元素之间的相对次序在排序前后保持一致，则称该排序算法是稳定算法（stable algorithm），否则这种排序算法为不稳定算法（unstable algorithm）

•起泡算法是稳定算法

2.8.3 归并排序

①有序向量的二路归并

•归并算法的核心是二路归并（2-way merge），即将两个有序向量组合成一个有序向量；通过将二分策略和二路归并结合使用（先分割至每个子向量只含有一个元素，即每个子向量必然有序，再反复归并结合成原规模的有序向量），即可完成对一个无序向量的排序

②归并时间

•二路归并的时间复杂度为O(n)，注意只是单次二路归并而不是排序总时间

③排序时间（推导过程p64/85）

•二路归并算法效率稳定，实则为Θ(n)，因此最终的归并排序时间复杂度为Θ(nlogn)

# 第三章 列表（List）

•列表是循位置访问（call-by-position）或称循链接访问（cal-by-link），元素的物理地址任意。

## 3.1 从向量到列表

3.1.1 从静态到动态

•数据结构支持的操作，分为动态和静态，静态操作用于获取信息，如size()和get()；动态操作用于修改数据结构的内容，如insert()与remove()。存储策略也分为静态和动态。静态存储策略，如向量可以短时间内进行静态操作，但是动态操作有很大不便（如插入删除操作需要移位等）。动态存储策略，如列表，可以大大降低动态操作成本。

3.1.2 由秩到位置

•因为采用动态存储策略，静态操作如循秩访问效率变得底下，因此应该多使用列表中的相对位置信息进行元素的定位与访问。

3.1.3 列表

•列表和向量一样，也是具有线性逻辑次序的一组元素构成的集合，因此也有前驱后继前后缀等定义。列表中的元素称为节点（node）。

## 3.2 接口

•列表中的节点和向量中的元素不同，必须保存列表前后的信息，因此单独成类

3.2.1 列表节点（ListNode）

•表3.1 列表节点ADT支持的操作接口

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

|  |
| --- |
| 1 **typedef int** Rank; //秩 |
| 2 **#define** ListNodePosi(T) ListNode<T>\* //列表节点位置 |
| 3 |
| 4 **template** <**typename** T> **struct** ListNode { //列表节点模板类（以双向链表形式实现） |
| 5 // 成员 |
| 6 T data; ListNodePosi(T) pred; ListNodePosi(T) succ; //数值、前驱、后继 |
| 7 // 构造函数 |
| 8 ListNode() {} //针对header和trailer的构造 |
| 9 ListNode( T e, ListNodePosi(T) p = NULL, ListNodePosi(T) s = NULL) |
| 10 : data(e), pred(p), succ(s) {} //默认构造器 |
| 11 // 操作接口 |
| 12 ListNodePosi(T) insertAsPred(T **const&** e); //紧靠当前节点之前插入新节点 |
| 13 ListNodePosi(T) insertAsSucc(T **const&** e); //紧随当前节点之后插入新节点 |
| 14 }; |

3.2.2 列表

•表3.2 列表ADT支持的操作接口

|  |  |  |
| --- | --- | --- |
| 操作接口 | 功能 | 适用对象 |
| size() | 报告列表当前的规模（节点总数） | 列表 |
| first()、last() | 返回首、末节点的位置 | 列表 |
| insertAsFirst(e)  insertAsLast(e) | 将e当作首、末节点插入 | 列表 |
| insertBefore(p, e)  insertAfter(p, e) | 将e当作节点p的直接前驱、后继插入 | 列表 |
| remove(p) | 删除位置p处的节点，返回其数值 | 列表 |
| disordered() | 判断所有节点是否已按非降序排列 | 列表 |
| sort() | 调整各节点的位置，使之按非降序排列 | 列表 |
| find(e) | 查找目标元素e，失败时返回NULL | 列表 |
| search(e) | 查找目标元素e，返回不大于e且秩最大的节点 | 有序列表 |
| deduplicate() | 剔除重复节点 | 列表 |
| uniquify() | 剔除重复节点 | 有序列表 |
| traverse() | 遍历并统一处理所有节点，处理方法由函数对象指定 | 列表 |

•List模板类定义见p69/90

## 3.3 列表

3.3.1 头尾节点

•头尾节点称为header/trailer，对外不可见，对外可见的节点第一个和最后一个节点称为fist node和last node，即首节点与末节点。被封装后外部不可见的节点，称为哨兵节点（sentinel node）

3.3.2 默认构造方法

3.3.3 由秩到位置的转换

3.3.4 查找

3.3.5 插入

3.3.6 基于复制的构造

3.3.7 删除

3.3.8 析构

3.3.9 唯一化

3.3.10 遍历

## 3.4 有序列表

•有序列表：sorted list，概念和有序向量类似，包括相关约定

3.4.1 唯一化

3.4.2 查找

•有序列表的查找算法并不比无序列表效率更高，因为动态存储策略导致物理地址并不必须按照逻辑次序排列，因此无法使用有序向量一般的减治算法。

## 3.5 排序器

3.5.1 统一入口

3.5.2 插入排序（insertionsort）

•适用于包括向量与列表在内的任何序列结构

•插入排序属于稳定算法

3.5.3 选择排序（selectionsort）

•适用于包括向量与列表在内的任何序列结构

•选择排序属于稳定CBA算法

3.5.4 归并排序

•归并排序列表与向量的时间复杂度相同，虽然在计算时有些许不同，但是按照整体复杂度而言还是一样的

•虽然不能循秩访问，在列表中找到二分点要比向量中花费更多时间，但是从整体复杂度而言，这仍是效率最高的做法

# 第四章 栈与队列

## 4.1 栈（Stack）

4.1.1 ADT接口

•盲端：栈是一种按照线性逻辑次序存放数据元素的容器，但是只能在一端插入和删除操作，另外一端禁止操作，称为盲端

•栈顶（stack top）和栈底（stack bottom），遵循后进先出（last-in-first-out, LIFO）原则

•表4.1 栈ADT支持的操作接口

|  |  |
| --- | --- |
| 操作接口 | 功能 |
| size() | 报告栈的规模 |
| empty() | 判断栈是否为空 |
| push(e) | 将e插至栈顶 |
| pop() | 删除栈顶对象 |
| top() | 引用栈顶对象 |

4.1.2 操作实例

•一般top()方法作为引用栈顶，pop()作为删除栈顶数据，二者都可以输出栈顶数据

4.1.3 Stack模板类

•因为栈是序列的一个特例，故可以派生自向量类，当然也可派生自列表类

|  |
| --- |
| 1 **#include** "../Vector/Vector.h" //以向量为基类，派生出栈模板类 |
| 2 **template** <**typename** T> **class** Stack: **public** Vector<T> { //将向量的首/末端作为栈底/顶 |
| 3 **public:** //size()、empty()以及其它开放接口，均可直接沿用 |
| 4 **void** push(T **const&** e) { insert(size(), e); } //入栈：等效于将新元素作为向量的末元素插入 |
| 5 T pop() { **return** remove(size() - 1); } //出栈：等效于初除向量的末元素 |
| 6 T& top() { **return** (\*this)[size() - 1]; } //取顶：直接返回向量的末元素 |
| 7 }; |

## 4.2 栈与递归

4.2.1 函数调用栈

•windows等大部分操作系统中，每个运行中的二进制程序都配有一个调用栈（call stack）或执行栈（execution stack）。调用栈的基本单位是帧（frame），每次函数调用都会创建一个相应的帧，用来存储相应的数据和地址，并且以栈的形式保存。因此调用栈中存储的都是活跃函数实例（active function instance），位于栈底的必然是入口主函数。

4.2.2 避免递归

•递归一般仍然是空间效率很低的的算法种类，有时需要避免；递归本身的实现是通过栈结构来实现的，因此可以人工模仿栈结构运算过程，实现等效的算法功能。

## 4.3 栈的典型应用

4.3.1 逆序输出

•举例：十进制转换任意进制数

4.3.2 递归嵌套

•举例：栈混洗，括号匹配

4.3.3 延迟缓冲

•举例：表达式求值（运算符优先级判定）及RPN转换

4.3.4 逆波兰表达式（reverse Polish notation, RPN）

•是数学表达式的一种，RPN属于后缀表达式（postfix），而一般常用的表达式为中缀表达式（infix）。RPN有优秀的运算符优先级表述能力，因此直接对其进行求值的算法更加高效。

•手工转换方法：p98/119

## 4.4 试探回溯法

4.4.1 试探与回溯

•剪枝（pruning）与回溯（backtracking），试探（probing）：是一种优化的算法策略，一般从零开始沿某一个方向逐步完善候选解，当到达一个位置所有的进一步试探都导致与目标不合的情况下，回溯到上一步做另外一个方向的尝试，以此类推直到发现解法

4.4.2 八皇后问题p100/121

4.4.3 迷宫寻径p102/123

## 4.5 队列（Queue）

4.5.1 概述

•队列只能从固定的一端插入元素，而同时只允许从另一端删除元素。允许取出元素的一端为队头（front），允许插入的称为队尾（rear）。对应的插入和删除元素，称为入队（enqueque）和出队（dequeue）

•队列与栈的区别特点在于，先进先出（first-in-first-out, FIFO）

4.5.2 ADT接口

•表4.4 队列ADT支持的操作接口

|  |  |
| --- | --- |
| 操作 | 功能 |
| size() | 报告队列的规模（元素总数） |
| empty() | 判断队列是否为空 |
| enqueue(e) | 将e插入队尾 |
| dequeue() | 删除队首对象 |
| front() | 引用队首对象 |

4.5.3 操作实例

4.5.4 Queue模板类

•因为队列两端都存在插入或删除操作，因此派生自列表类更佳

|  |
| --- |
| 1 **#include** "../List/List.h" //以List为基类 |
| 2 **template** <**typename** T> **class** Queue: **public** List<T> { //队列模板类（继承List原有接口） |
| 3 **public:** //size()、empty()以及其它开放接口均可直接沿用 |
| 4 **void** enqueue(T **const&** e) { insertAsLast(e); } //入队：尾部插入 |
| 5 T dequeue() { **return** remove(first()); } //出队：首部删除 |
| 6 T& front() { **return** first()->data; } //队首 |
| 7 }; |

## 4.6 队列应用

4.6.1 循环分配器

•即资源循环分配器，更直观的说就是排队做某件事。而出队的元素之后会回到队尾，称为轮值（round robin）。这种方法有一个关键点在于确定每个元素单次被处理花费的时间，过长会导致其他元素响应速度下降，过短会导致占有权切换耗费更多比例的时间因而降低效率。必须通过实测决定最佳值。

•更复杂条件和需求下的调度分配算法，可参考排队论（queuing theory）的资料

4.6.2 银行服务模拟

# 第五章 二叉树

•基本数据结构分为两种类型，基于数组实现或基于链表实现。二者各有优劣。

•树不属于纯粹的线性结构（linear structure）而属于半线性结构（semi-linear structure）

## 5.1 二叉树及其表示

5.1.1 树

•树等价于连通无环图，树由一组顶点（vertex）和若干条边（edge）组成。树的某一特定顶点可以定义称为根（root），则为有根树（rooted tree）。另外顶点有时也称为节点（node）。

•深度（depth）：沿每个节点v到根r的唯一通路所经过边的数目，根节点深度为0，即第0层。

•任一节点v在通往树根沿途所经过的每个节点都是其祖先（ancestor），v是它们的后代（descendant）。特别地，v的祖先/后代包括其本身，而v本身以外的祖先/后代称作真祖先（proper ancestor）/真后代（proper descendant）。

•节点v历代祖先的层次，自下而上以1为单位逐层递减；在每一层次上，v的祖先至多一个。特别地，若节点u是v 的祖先且恰好比v 高出一层， 则称u 是v 的父亲（parent），v是u的孩子（child）。v的孩子总数，称作其度数或度（degree），记作deg(v)。无孩子的节点称作叶节点（leaf），包括根在内的其余节点皆为内部节点（internal node）。

•v所有的后代及其之间的联边称作子树（subtree），记作subtree(v)。在不致歧义时，我们往往不再严格区分节点（v）及以之为根的子树（subtree(v)）。

•树T中所有节点深度的最大值称作该树的高度（height），记作height(T)。不难理解，树的高度总是由其中某一叶节点的深度确定的。特别地，本书约定，仅含单个节点的树高度为0，空树高度为-1。推而广之，任一节点v所对应子树subtree(v)的高度，亦称作该节点的高度，记作height(v)。特别地，全树的高度亦即其根节点r的高度，height(T) = height(r)。

5.1.2 二叉树

•因此在二叉树中，同一父节点的孩子都可以左、右相互区分。此时，亦称作有序二叉树（ordered binary tree）。本书所提到的二叉树，默认地都是有序的（两者间的有序不一定是大小关系，可以是某种性质的不同，只要可以区分即可）。特别地，不含一度节点的二叉树称作真二叉树（proper binary tree）。

5.1.3 多叉树

•树中各节点的孩子数目并不确定。每个节点的孩子均不超过k个的有根树，称作k叉树（k-ary tree）。

•仿照有序二叉树的定义，凡符合同一节点的所有孩子之间必须具有某一线性次序这一条件的多叉树也称作有序树（ordered tree）

•有序多叉树可以转化为二叉树，具体方法是为每个多叉树的节点设置两个指针（或称为分支），分别指向其“长子”和下一个“兄弟”。便可将多叉树改写为一种特殊形式的有序二叉树。

## 5.2 编码树

5.2.1 二进制编码

•编码解码（encoding & decoding）：信息被转换为二进制形式的过程，反之由二进制编码恢复原始信息的过程称为解码

•前缀无歧义编码（prefix-free code，PFC）：即各字符的编码串互相不为前缀，则解码时，对每个二进制编码组合的转换不存在歧义

5.2.2 二叉编码树

•定义：任一编码方案都可描述为一棵二叉树，从根节点出发，每次向左（右）都对应于一个0（1）比特位。由从根节点到每个节点的唯一通路，可以为各节点v赋予一个互异的二进制串，称作根通路串（root path string），记作rps(v)。当然，|rps(v)| = depth(v)就是v的深度。

•PFC编码树：PFC二叉编码树的特点则是，每个字符对应的节点都是叶节点，即没有字符节点是其他字符节点的parent即可。PFC编码的解码不必等所有编码串集齐后进行，可以一个个边接收边进行，因此属于在线算法。

•使用二叉树这种数据结构来存储编码表，并进行解码非常高效。

## 5.3 二叉树的实现

5.3.1 二叉树节点（BinNode模板类）

模板类定义和基本操作的宏定义见p117/138

•节点模板类的成员还有深度和子树规模等，但是这些成员必须随着树的结构调整时时更新，应该根据具体情况看是否应该带有这些成员，或者在需要时直接计算需要的量

5.3.2 二叉树节点操作接口

5.3.3 二叉树（BinTree模板类）

•\_root指向树根，\_size动态记录树的规模，\_root = NULL当且仅当\_size = 0

•列表，二叉树等模板类并不包含全部的节点对象成员，只是包含一个头尾哨兵或者是根节点即可，因为都是可以链式查找的。

## 5.4 遍历（traversal）

5.4.1 递归式遍历

•遍历的次序种类中，左兄弟优先于右兄弟，节点本身的访问次序可以有三种选择，即VLR，LVR和LRV。分别称为先序遍历，中序遍历和后序遍历（preorder，inorder，postorder）

•中序遍历对节点的访问次序和按照有序树定义确定的全局左右次序完全吻合，即完全按照从左到右的顺序访问节点。

5.4.2 迭代版先序遍历

5.4.3 迭代版中序遍历

5.4.4 迭代版后序遍历

5.4.5 层次遍历

•完全二叉树：若在对某棵二叉树的层次遍历过程中，前次迭代中都有左孩子入队，且前次迭代中都有右孩子入队，则称之为完全二叉树（complete binary tree）。完全二叉树的高度，n为节点总数。

•满二叉树（full binary tree）：所有叶节点同处于最底层（非底层节点均为内部节点）

## 5.5 Huffman编码

5.5.1 PFC编码及解码

•构造PFC编码树：由每一字符分别构造一棵单节点二叉树，将所有二叉树视为一个森林，然后反复从森林中取出两棵树并将其合二为一。

•具体算法代码：p137/158

•优化：编码解码过程的效率体现在由原始字符文本生成的二进制编码串总长，或者说从小到大越能完全利用各种二进制数来代替字符，效率越高。

5.5.2 最优编码树

•平均编码长度与叶节点平均深度（average leaf depth，ald(T)）是一个概念。

•同一字符集所有编码方案中，平均编码长度最小者称作最优方案，即最优二叉编码树（optimal encoding tree）。

①双子性：最优二叉编码树必为真二叉树，和完全二叉树或满二叉树含义不同

②层次性：最优编码树中，叶节点深度之差不能超过1

•节点交换是尝试优化编码树的一种方法

•一种构造最优编码树的方法是：创建一棵规模为2n-1的完全二叉树T，再将其中的n个叶节点分配给n个字符。

5.5.3 Huffman编码树

•因为一般文本中的字符不是等概率出现的，所以上述构造算法并不常用。

•带权平均编码长度或叶节点带权平均深度（weighted average leaf depth）：用字符出现的概率作为权重的平均深度，写作wald(T)

•最优带权编码树，同样是wald(T)最小的编码树，但是不一定是完全二叉树。不过仍然有双子性。

•若字符x和y的出现概率在所有字符中最低，则必然存在某最优带权编码树，使二者在其中同处最底层，且互为兄弟。（新的层次性）

5.5.4 Huffman编码算法

•算法：对于字符出现概率已知的任一字符集，对应每个字符先分别建立一棵单个节点的树，这些树构成一个森林。然后从其中选出权重最小的两棵树合并到一个新节点上创建一棵新树。新树的权重为两棵子树的权重之和。重复这个过程直到只剩下一棵树为止。生成的树称为Huffman encoding tree.

•Huffman编码树只是最优带权编码树中的一棵。

•代码实现：p144/165

•编码encode用二进制串（如Bitmap）和编码表（如HashTable），解码用二进制串和编码树进行。

•构造Huffman编码树的时间复杂度O(n^2)

# 第六章 图

## 6.1 概述

•所谓的图（graph），可定义为G = (V, E)。其中，集合V中的元素称作顶点（vertex或node）；集合E中的元素分别对应于V中的某一对顶点(u, v)，表示它们之间存在某种关系，故亦称作边（edge或arc）。对于V和E的规模，n=|V|，e=|E|

•无向边：若边(u, v)所对应顶点u和v的次序无所谓，则称作无向边（undirected edge）

•有向边：反之若u和v不对等，则称(u, v)为有向边（directed edge）。无向边(u, v)也可记作(v, u)，而有向的(u, v)和(v, u)则不可混淆。这里约定，有向边(u, v)从u指向v，其中u称作该边的起点（origin）或尾顶点（tail），而v称作该边的终点（destination）或头顶点（head）。

•无向图：若E中各边均无方向，则G称作无向图（undirected graph，简称undigraph）。

•有向图：若E中只含有向边，则G称作有向图（directed graph，简称digraph）。

•混合图：若E同时包含无向边和有向边，则G称作混合图（mixed graph）。有向图的通用性更强，因为无向图和混合图都可转化为有向图。

•度：对于任何边e = (u, v)，称顶点u和v彼此邻接（adjacent），互为邻居；而它们都与边e彼此关联（incident）。在无向图中，与顶点v关联的边数，称作v的度数（degree），记作deg(v)。对于有向边e = (u, v)，e称作u的出边（outgoing edge）、v的入边（incoming edge）。v的出边总数称作其出度（out-degree），记作outdeg(v)；入边总数称作其入度（in-degree），记作indeg(v)。

•自环与简单图：联接于同一顶点之间的边，称作自环（self-loop）。不含任何自环的图称作简单图（simple graph）

•通路：所谓路径或通路（path），就是由m + 1个顶点与m条边交替而成的一个序列：π= { v0, e1, v1, e2, v2, ..., em, vm }且对任何0 < i ≤ m都有ei = (vi-1, vi)。也就是说，这些边依次地首尾相联。尽管通路上的边必须互异，但顶点却可能重复。沿途顶点互异的通路，称作简单通路（simple path）。

•环路：对于长度m ≥ 1的通路，若起止顶点相同（即v0 = vm），则称作环路（cycle）。，不含任何环路的有向图，称作有向无环图（directed acyclic graph, DAG）。若沿途除v0 = vm外所有顶点均互异，则称作简单环路（simple cycle）特别地，经过图中各边一次且恰好一次的环路， 称作欧拉环路（Eulerian tour）。经过图中各顶点一次且恰好一次的环路，称作哈密尔顿环路（Hamiltonian tour）。

•带权网络：若为每一边e指定一个权重（weight），比如wt(e)即为边e的权重。各边均带有权重的图，称作带权图（weighted graph）或带权网络（weighted network），有时也简称网络（network），记作G(V, E, wt())。

•复杂度：问题的输入规模，也应该以顶点数与边数的总和（n + e）来度量。对于无向图，每一对顶点至多贡献一条边，故总共不超过n(n - 1)/2条边，且这个上界由完全图达到。对于有向图，每一对顶点都可能贡献（互逆的）两条边，因此至多可有n(n - 1)条边。总而言之，必有e = O(n2)。

## 6.2 抽象数据类型

6.2.1 操作接口

•图ADT支持的边操作接口

|  |  |
| --- | --- |
| 操作接口 | 功能描述 |
| e() | 边总数|E| |
| exist(v,u) | 判断联边(v, u)是否存在 |
| insert(v,u) | 引入从顶点v到u的联边 |
| remove(v,u) | 删除从顶点v到u的联边 |
| type(v,u) | 边在遍历树中所属的类型 |
| edge(v,u) | 边所对应的数据域 |
| weight(v,u) | 边的权重 |

•图ADT支持的顶点操作接口

|  |  |
| --- | --- |
| 操作接口 | 功能描述 |
| n() | 顶点总数|V| |
| insert(v) | 在顶点集V中插入新顶点v |
| remove(v) | 将顶点v从顶点集中删除 |
| inDegree(v)  outDegree(v) | 顶点v的入度和出度 |
| firstNbr(v) | 顶点v的首个邻接顶点 |
| nextNbr(v,u) | 在v的邻接顶点中，u的后继 |
| status(v) | 顶点v的状态 |
| dTime(v), fTime(v) | 顶点v的时间标签 |
| parent(v) | 顶点v在遍历树中的父节点 |
| priority(v) | 顶点v在遍历树中的权重 |

6.2.2 Graph模板类

p153/174

## 6.3 邻接矩阵（adjacency matrix）

6.3.1 原理

•邻接矩阵：是图ADT最基本的实现方式，使用方阵A[n][n]表示由n个顶点构成的图，方阵的每个元素，表示一对顶点之间可能存在的邻接关系（即一条有向边或无）。元素的具体取值可以作为边的权重，取值无穷或0代表不存在该有向边

6.3.2 实现Graph

p155/176

6.3.3 时间性能

•由于邻接矩阵实现的图结构基于向量结构，所以兼备了向量的优劣势。但通常顶点的动态操作远小于其它操作。就分摊意义而言，单次操作的耗时亦不过O(n)

6.3.4 空间性能

•空间总量渐进地不超过O(n2)

•无向图的邻接矩阵一般为对称，可以进一步提高利用率

## 6.4 邻接表（adjacency list）

6.4.1 原理

•稀疏图（sparse graph）

•邻接表：用列表代替向量实现图结构，邻接矩阵的每一行转化为一个列表，即共需n个列表，而每个列表中只保存确实存在的边

6.4.2 复杂度

•空间总量为O(n+e)，与图自身的规模相当

•exist(v,u)需要O(n)，插入O(1)，删除O(e)；邻接表的自身结构决定了便于处理同一顶点的所有关联边

## 6.5 图遍历算法概述

•核心思想：将非线性结构转化为半线性结构

•遍历树（traversal tree）：即按照遍历算法的策略将图转化为树结构形成的支撑树

•图搜索（graph search）：图的遍历更加强调对处于特定状态顶点的甄别与查找，故也称作图搜索，图搜索即是图算法的基石，一般这些算法自身仅需O(n+e)时间

## 6.6 广度优先策略（Breadth-first search, BFS）

6.6.1 策略

•策略特点：越早被访问到的顶点，其邻居越优先被选用；效果等同于树结构的层次遍历

•核心思想： 反复从波峰集（frontier）中找到最早被访问到顶点v，若其邻居均已访问到，则将其逐出波峰集；否则，随意选出一个尚未访问到的邻居，并将其加入到波峰集中

•始自图中顶点s的BFS搜索，将首先访问顶点s；再依次访问s所有尚未访问到的邻居；再按后者被访问的先后次序，逐个访问它们的邻居；...；如此不断。在所有已访问到的顶点中，仍有邻居尚未访问者，构成所谓的波峰集（frontier）。对于无向图必能覆盖s所属的连通分量（connected component），对于有向图必能覆盖以s为起点的可达分量（reachable component）

6.6.2 实现

•BFS算法：p160/181

•广度优先搜索树：或简称BFS tree，即按照该策略给出的原图的某一连通或可达域的一棵遍历树

•BFS森林（BFS forest）：由多个BFS tree组成的结构；一个图因为结构不同，起始顶点选取不同可能无法一次性用一个BFS tree遍历

6.6.3 实例

6.6.4 复杂度

•时间和空间复杂度均为O(n+e)

6.6.5 应用

•连通域分解

•最短路径

## 6.7 深度优先搜索（Depth-First Search, DFS）

6.7.1 策略

•策略核心：优先选取最后一个被访问到的顶点的邻居，各个顶点被访问到的次序，类似于树的先序遍历，各顶点被访问完毕的次序，类似于树的后序遍历

6.7.2 实现

p162/183

•活跃期（active duration）：一个顶点v被发现和访问完成的时刻区间，即[dTime(v), fTime(v)]。任意顶点v和u之间是否存在祖先/后代的“血缘”关系，完全取决于二者的活跃期是否相互包含。更进一步，两个顶点的活跃期如果存在重叠，则两顶点必存在包含和被包含关系。因此对于DFS和基于DFS的算法，时间标签非常重要。

6.7.3 实例

6.7.4 复杂度

•时间与空间复杂度均为O(n+e)

6.7.5 应用

•也可用于连通分量的分解和有向无环图的判定

•拓扑排序和双连通域分解

## 6.8 拓扑排序

6.8.1 应用概述

•定义：给定描述某一实际应用的有向图，如何在与该图“相容”的前提下，将所有顶点排成一个线性序列。相容的概念是，每一顶点都不会通过边，指向其在此序列中的前驱顶点。这样的序列或操作，称为原有向图的一个拓扑排序（topological sorting）。

6.8.2 有向无环图

•有向无环图一定存在拓扑排序，反之亦然。有向无环图对应于偏序关系，而拓扑排序则对应于全序关系。在顶点数目有限时，与任一偏序相容的全序必然存在。

•在任一有限偏序集中，必有极值元素（尽管未必唯一）；相应地，任一有向无环图，也必包含入度为零的顶点。有限偏序集中也必然存在极小元素（同样，未必唯一）。该元素作为顶点，出度必然为零。

6.8.3 算法

•算法一：不断将入度为0的顶点及其关联边从图中取出，即可构成

•算法二：DFS遍历过程中各顶点被标记为VISITED的次序按照逆序给出了原图的拓扑排序

6.8.4 实现

p167/188，精简版：习题6-19

6.8.5 实例

6.8.6 复杂度

时间与空间复杂度均为O(n+e)

## 6.9 双连通域分解

6.9.1 关节点与双连通域

•关节点（articulation point）或切割节点（cut vertex）：若删除顶点v后G所包含的连通域增多，则v称作切割节点

•双连通图：不含任何关节点的图称作双连通图。

•双连通域（bi-connected component）：任一无向图都可视作由若干个极大的双连通子图组合而成，这样的每一子图都称作原图的一个双连通域。（每个子图之间可以有重复的顶点，但是不能缺少原图中存在的边）

6.9.2 蛮力算法

•耗时O(n(n+e))

6.9.3 可行算法

6.9.4 实现

p170/191

6.9.5 实例

6.9.6 复杂度

•时空复杂度均为O(n+e)

## 6.10 优先级搜索

6.10.1 优先级与优先级数

•包括BFS和DFS在内的几乎所有图搜索，都可纳入统一的框架，即根据某种事先约定的优先级次序访问顶点。鉴于优先级在其中所扮演的关键角色，故亦称作优先级搜索（priority-first search, PFS），或最佳优先搜索（best-first search, BFS）。

•优先级数（priority number）：顶点的一种属性。

6.10.2 基本框架

p173/194

6.10.3 复杂度

•若不借助优先级队列，PFS的总体时间复杂度为O(n2)

## 6.11 最小支撑树

6.11.1 支撑树

•连通图G的某一无环连通子图T若覆盖G中所有的顶点，则称作G的一棵支撑树或生成树（spanning tree）。

•支撑树既是“禁止环路”前提下的极大子图，也是“保持连通”前提下的最小子图。尽管同一幅图可能有多棵支撑树，但由于其中的顶点总数均为n，故其采用的边数也均为n - 1。

6.11.2 最小支撑树

•若图G为一带权网络，则每一棵支撑树的成本（cost）即为其所采用各边权重的总和。在G的所有支撑树中，成本最低者称作最小支撑树（minimum spanning tree, MST）。

6.11.3 歧义性

•总体成本最低的支撑树却未必唯一，故严格说来，此类支撑树应称作极小支撑树（minimal spanning tree）。当然，通过强制附加某种次序即可消除这种歧义性。

6.11.4 蛮力算法

•然而根据Cayley公式，由n个互异顶点组成的完全图共有nn-2棵支撑树，即便忽略掉构造所有支撑树所需的成本，仅为更新最低成本的记录就需要*O*(nn-2)时间。

6.11.5 Prim算法

•基本概念：图G = (V; E)中，顶点集V的任一非平凡子集U及其补集V\U都构成G的一个割（cut），记作(U : V\U)。若边uv满足uU且vU，则称作该割的一条跨越边（crossing edge）。因此类边联接于V及其补集之间，故亦形象地称作该割的一座桥（bridge）。

•思想：最小支撑树总是会采用联接每一割的最短（权重最小）跨越边。当然不一定每种cut之间都只有一条跨越边。

•贪心迭代：每一步迭代之前，假设已经得到最小支撑树T的一棵子树Tk = (Vk; Ek)，其中Vk包含k个顶点，Ek包含k - 1条边。于是，若将Vk及其补集视作原图的一个割，则在找到该割的最短跨越边ek = (vk, uk)（vkVk且ukVk）之后，即可将Tk扩展为一棵更大的子树Tk+1 = (Vk+1; Ek+1)，其中Vk+1 = Vk{uk}，Ek+1 = Ek{ek}。最初的T1不含边而仅含单个顶点，故可从原图的顶点中任意选取。

•复杂度：若不借助优先级队列，则时间复杂度为O(n2)

## 6.12 最短路径

•问题描述：给定带权网络G = (V, E)，以及源点（source）sV，对于所有的其它顶点v，s到v的最短通路有多长？该通路由哪些边构成？

6.12.1 最短路径树

•问题简化：带权网络G内各边权重均大于零，若非如此则会造成难以解决的歧义性

•最短路径树：由源点S到所有其他顶点的最短路径组成的树

6.12.2 Dijkstra算法

•贪心算法：只要能够确定uk+1，便可反过来将Tk扩展为Tk+1。如此，便可按照到s距离的非降次序，逐一确定各个顶点{ u1, u2, ..., un }，同时得到各棵最短路径子树，并得到最终的最短路径树T = Tn。每一个顶点uk+1都是在Tk之外，距离s最近者。若将此距离作为各顶点的优先级数，每次将uk+1加入Tk并将其拓展至Tk+1后，需要且只需要更新那些仍在Tk+1之外，且与Tk+1关联的顶点（其实就是uk+1的邻居的优先级会有影响）的优先级数。

•实现：p179/200

•复杂度：和最小支撑树一致

# 第七章 搜索树

•二叉搜索树是一种类似但是不同于二叉树的新的数据结构，兼顾动态修改和静态查找的总体效率。

## 7.1 查找

7.1.1 循关键码访问（call-by-key）

7.1.2 词条（entry）

•entry的概念包括关键码（key）与数值（value），前者即是有数值意义或可排序的标签数据，而value则是不一定或不必需可排序甚至无数值意义的实际存储数据。key好比身份证号码，value好比公民的名字或其他个人信息。

•词条模板类p183/204

7.1.3 序与比较器

•对于算法而言，value的具体内容其实没有多大意义，只有key才是查找所需的指标，所以一般讨论算法时不对key和entry进行区分。

## 7.2 二叉搜索树（binary search tree）

7.2.1 顺序性

•二叉搜索树：对于一个二叉树，任一节点r的左（右）子树中，所有节点（若存在）均不大于（不小于）r

7.2.2 中序遍历序列

•任何一颗二叉树是二叉搜索树，当且仅当其中序遍历序列单调非降

7.2.3 BST模板类p185/206

•二叉搜索树的所有变种，都必须支持三个基本接口，search(), insert(), remove()

7.2.4 查找算法及其实现

•算法：原理上和有序向量二分查找一样，只不过是采用了树结构，从根节点到叶节点逐层比较递进而已。

•语义约定：对于查找失败的情况，仍然按照类似查找成功的方式处理，只是返回的是一个假想的哨兵节点，其内容为空。这种处理方式可以兼容删除和插入操作的各种情况，因为二者都需要查找作为第一步。

•效率：最好时O(1)，最坏时Ω(n)，因为最坏时BST可退化为单链即一维有序列表。此时的查找变为效率较低的顺序查找。因此要控制BST的高度以控制最坏查找时间。

7.2.5 插入算法及其实现

•BST插入算法有其特殊性，即插入的节点必然是叶节点，因为BST有顺序性，插入之前查找失败后，失败位置必然是一处空孩子处。

•效率：节点插入操作所需的时间，主要消耗于对算法search()及updateHeightAbove()的调用。后者与前者一样，在每一层次至多涉及一个节点，仅消耗O(1)时间，故其时间复杂度也同样取决于新节点的深度，在最坏情况下不超过全树的高度。

7.2.6 删除算法及其实现

•BST删除算法是后续所有高级BST删除算法的核心，这个算法的理念是若待删除节点同时含有左右子树，则将待删除的节点或关键码和其直接后继（也可以和直接前驱做对称的操作）交换，然后再删除之，因交换后其必不含左子树方便处理（之后若有右子树可直接承继被删除节点，若无右子树则直接删除即可）。

•效率：删除操作所需的时间，主要消耗于对search()、succ()和updateHeightAbove()的调用。在树中的任一高度，它们至多消耗O(1)时间。故总体的渐进时间复杂度，亦不超过全树的高度。

## 7.3 平衡二叉搜索树（balanced binary search tree, BBST）

7.3.1 树高与性能

•实际应用中，构建BST时，关键码往往是单调录入的，而删除操作的过多使用也会导致树的倾斜。

7.3.2 理想平衡与适度平衡

•理想平衡：包含n个节点的二叉树，高度不可能小于log2n。若树高恰好为log2n，则称作理想平衡树

•适度平衡：将树高限制为“渐进地不超过*O*(logn)”

7.3.3 等价变换

•等价：若两棵二叉搜索树的中序遍历序列相同，则称它们彼此等价；反之亦然。特点可概括为“上下可变，左右不乱”，它也是以下等价变换的基本特性。

•平衡二叉搜索树的适度平衡性，都是通过对树中每一局部增加某种限制条件来保证的。

•局部性：

1）经过单次动态修改操作后，至多只有*O*(1)处局部不再满足限制条件

2）总可在*O*(logn)时间内，使这*O*(1)处局部（以至全树）重新满足限制条件

•刚刚失去平衡的二叉搜索树，必然可以迅速转换为一棵等价的平衡二叉搜索树。等价二叉搜索树之间的上述转换过程，也称作等价变换。

7.3.4 旋转调整

•zig和zag：p193/p214

•zig和zag旋转均属局部操作，仅涉及常数个节点及其之间的联接关系，故均可在常数时间内完成。就与树相关的指标而言，经一次zig或zag旋转之后，节点v的深度加一，节点c的深度减一；这一局部子树（乃至全树）的高度可能发生变化，但上、下幅度均不超过一层。

## 7.4 AVL树

•AVL：由G. M. Adelson-Velsky和E. M. Landis于1962年发明，以他们名字的首字母命名

7.4.1 定义及性质

•平衡因子：任一节点v的平衡因子（balance factor）定义为“其左、右子树的高度差”，空树高度取-1，单节点子树（叶节点）高度取0。

•AVL：所谓AVL树，即平衡因子受限的二叉搜索树..其中各节点平衡因子的绝对值均不超过1。高度为h的AVL树至少包含fib(h + 3) - 1个节点。

•失衡节点集：因节点x的插入或删除而暂时失衡的节点，UT(x)，若x为被摘除的节点，则UT(x)仅含单个节点；但若x为被引入的节点，则UT(x)可能包含多个节点。

7.4.2 节点插入

•新引入节点x后，UT(x)中的节点都是x的祖先，且高度不低于x的祖父。因为x的父节点在此前于x处的子树高度为-1（空树），且原来本已平衡，则另一子树至多有一个根节点（高度为0），则x的引入无论如何不会影响到x的父节点的平衡。而x的引入使得其父节点高度至多+1，如果导致其祖先失衡，则x必处于该祖先节点两棵子树中本已经较高的一棵（该祖先的两子树原先必不等高）。而对于未失衡的祖先，若该祖先的祖先中有失衡的，那么从最深失衡祖先到x经历的所有祖先节点，虽然未失衡但是全部因为x的引入导致高度+1，即原来都是两子树等高的节点。

•两种局部调整方案：单旋和双旋p196/217~p197/218，只需调整因x引入影响到平衡的最深节点，亦即x的某位祖先（深度要低于x的祖父）。

•无论单旋或双旋，经局部调整之后，不仅g(x)能够重获平衡，而且局部子树的高度也必将复原。这就意味着，g(x)以上所有祖先的平衡因子亦将统一地复原。换而言之，在AVL树中插入新节点后，仅需不超过两次旋转，即可使整树恢复平衡。

•效率：AVL树的节点插入操作可以在O(logn)时间内完成。

7.4.3 节点删除

•在摘除节点x后，以及随后的调整过程中，失衡节点集UT(x)始终至多只含一个节点。而且若该节点g(x)存在，其高度必与失衡前相同。另外还有一点重要的差异是，g(x)有可能就是x的父亲。

•节点删除只有一个失衡节点，因为因删除导致的失衡只可能是两子树不等高且删除导致较低子树高度-1所致，而失衡节点的高度仍由较高子树决定，因此失衡节点集只有一个。因为失衡，则较高子树高度至少为1，因此g(x)、较高子树根节点p和p的子节点v（高度至少为1则必存在）构成了类似节点插入平衡调整的可控结构。显然一般情况下，v应选取p的较高子树根节点，若p的两子树等高，则应对应p和g(x)的关系选取以便仅用单旋即可调整平衡。

•失衡传播：节点删除重平衡与节点插入重平衡不一样，后者重平衡前后不会影响局部子树高度，而前者有可能会，因此可能会导致深层节点重平衡后，祖先节点失衡的情况。只需逐层遍历x的所有祖先并重复重平衡操作即可。

•效率：同样是O(logn)时间，因为单次删除导致的直接失衡节点只有一个，而失衡传播最多传递logn层至树根，因此和节点插入一致。

7.4.4 统一重平衡算法

•综合以上的各种重平衡情况，可以写出一个统一的方法来专门处理各种g,p,v组合并平衡

见p200/221, 3+4重构, 3节点+4子树结构

# 第八章 高级搜索树

•持久性结构（persistent structure）：一种高级数据结构

•四叉树（quadtree）八叉树（octree）

## 8.1 伸展树（splay tree）

•伸展树也是平衡二叉搜索树，虽然没有定义平衡因子和相关的平衡约束。它的平衡是通过“伸展”来实现的。

•显然没有维护平衡约束会导致伸展树在最坏情况下的运行效率很低，但是“伸展”功能使得其分摊意义上效率更高，并且考虑到了数据局部性影响。

8.1.1 局部性

•数据局部性（data locality）：

1）刚刚被访问过的元素，极有可能在不久之后再次被访问到

2）将被访问的下一元素，极有可能就处于不久之前被访问过的某个元素的附近

•将刚被访问的节点，及时地“转移”至树根（附近），即可加速后续的操作。这种转移操作通过等价变换实现，而转移过程中树结构的“伸展”动作为这种数据结构获名。

8.1.2 逐层伸展

•逐层伸展是一种简易的伸展策略，即将每次访问过后的节点，通过形如围绕其父节点旋转的等价变换，逐层上升至根节点。这种策略单次访问的分摊时间复杂度在极端情况下高达Ω(n)

•对于规模为任意n的伸展树，只要按关键码单调的次序，周期性地反复进行查找，则无论总的访问次数m >> n有多大，就分摊意义而言，每次访问都将需要Ω(n)时间！

8.1.3 双层伸展

•双层伸展：每次都从当前节点v向上追溯两层（而不是仅一层），并根据其父亲p以及祖父g的相对位置，进行相应的旋转。双层伸展的特点是，每次将某个底层节点转移至根部都会使该分支的整体高度大致减半，从而增加了下一次从该分支查找的效率。

•三种情况：zig-zig/zag-zag, zig-zag/zag-zig, zig/zag，p206/227

•Tarjan等人采用势能分析法（potential analysis）业已证明，在改用“双层伸展”策略之后，伸展树的单次操作均可在分摊的O(logn)时间内完成

8.1.4 伸展树的实现

•伸展树接口定义p208/229

•伸展算法的实现p209/230

•查找算法的实现p210/231（查找算法包括了伸展算法，而插入删除算法则包括查找算法）

•插入算法的实现p210/231

•删除算法的实现p212/233

•有两种实现插入删除的算法，一种是沿用一般的二叉搜索树算法，之后进行伸展，另一种即将伸展算法包括进查找算法。

## 8.2 B-树

•由R. Bayer和E. McCreight于1970年合作发明。

•I/O操作：两个相邻存储级别之间的数据传输，统称I/O操作。

•各级存储器的访问速度相差悬殊，故应尽可能地减少I/O操作。在衡量相关算法的性能时，基本可以忽略对内存的访问，转而更多地关注对外存的访问次数。

•磁盘之类外部存储器的另一特性：就时间成本而言，读取物理地址连续的一千个字节，与读取单个字节几乎没有区别。外部存储器更适宜于地址连续的批量式访问。

•不妨通过时间成本相对极低的多次内存操作，来替代时间成本相对极高的单次外存操作。

8.2.1 多路平衡查找

•多路搜索树（multi-way search tree）：一般地，以k层为间隔，将每k层中所有最根部的节点与其所有子孙组成一个“大节点”，如此重组，可将二叉搜索树转化为等价的2k路搜索树。而每个大节点中2k-1个关键码存储在一起。比如，2层为间隔，一个大节点包括一个节点及其两个孩子，还有它们外延的四个分支。

•不考虑硬件因素，多路搜索树和原二叉树效率一致，但是每次从外存读取数据是以“大节点”为单位进行的一次地址连续的批量访问。大节点每组关键码的最佳数目，取决于不同外存的批量访问特性，可根据磁盘扇区的容量和关键码的大小，经换算得出每组关键码的最佳规模。

•多路平衡搜索树：所谓m阶B-树，即m路平衡搜索树（m ≥2）。对于分支数目的要求：除根以外的所有内部节点，都应满足：n + 1≥ ，而在非空的B-树中，根节点应满足：n + 1≥ 2（n为单个节点关键码数量，之所以根节点的分支要求不一样，跟上溢处理有关）。因为各节点的分支数介于至m之间，故m阶B-树也称作(, m)-树，如(2, 3)-树、(3, 6)-树或(7, 13)-树等。

•B-树所有外部节点均深度相等，且同处于最底层。则其最深的内部节点同样具有类似的性质。可以简单来看，B-树就是一棵满多叉树（分叉数目不定但是有上下限），因此其外部节点数目比关键码数目始终多1。

•计算B-树高度时，还需要计入其最底层的外部节点。因为B-树往往是用于不同级存储系统之间的数据结构，也就是说一整棵树可能不存储在同一个区域，其最底层往往关联其他级存储系统。

8.2.2 ADT接口及其实现

•同一节点的所有孩子组织为一个向量，各相邻孩子之间的关键码也组织为一个向量。当然，按照B-树的定义，孩子向量的实际长度总是比关键码向量多一。

B-树节点和B树的接口p215/236~p216/237

8.2.3 关键码查找

•算法：和一般的二叉搜索树类似，区别在于在大节点内部进行比较搜索时，因为大节点内部的关键码成向量结构，并且数量通常在128~512之间。可以使用顺序查找。

•实现p217/238

•查找结果由返回的节点位置指代：成功时返回目标关键码所在的节点，上层调用过程可在该节点内进一步查找以确定准确的命中位置；失败时返回对应外部节点，其父节点则由变量\_hot指代。

8.2.4 性能分析

•树高：若存有N个关键码的m阶B-树高度为h，则必有

即存有N个关键码的m阶B-树的高度h = Θ(**log**mN)。

•复杂度：渐进意义上O(logmN)，虽然趋势和一般的BST一样，但是这是以外存读取操作为基准的。实际上比原BST算法将I/O操作次数降低为原先的1/log2m

8.2.5 关键码插入

•实现p219/240

•\_hot所指的节点中增加了一个关键码。若该节点内关键码的总数依然合法（即不超过m - 1个），则插入操作随即完成。否则，称该节点发生了一次上溢（overflow），此时需要通过适当的处理，使该节点以及整树重新满足B-树的条件。

8.2.6 上溢与分裂

•上溢修复：刚发生上溢的节点，应恰好含有m个关键码，取其中间关键码s=，将原节点分割成两个子节点（节点分裂split），并将中间码s上移一层，成为两子节点父亲并归入适当位置。处理上溢时的算法操作，可能会发生上溢传递。

•实现：p220/241

•复杂度：时间通常主要消耗于对目标关键码的查找，O(logmN)

8.2.7 关键码删除

•算法：p222/243

•过程：首先查找需要删除的关键码e所在节点v，并进一步确定其秩，若v为叶节点，可进行删除；若非叶节点，将e与其直接后继（或前驱）交换，即可将e移至叶节点。之后将e（及其左侧的外部空节点）从v中删去。

•若该节点所含关键码的总数依然合法（即不少于 - 1），则删除操作随即完成。否则，称该节点发生了下溢（underflow），

8.2.8 下溢与合并

•下溢修复：2种情况，第一种从父节点借取父关键码，父节点再从其左或右兄弟节点借取最大或最小的关键码（前提是兄弟节点不会因此发生下溢）补充；第二种将两兄弟节点和其父关键码合并为一个节点（如果兄弟节点无法完成第一种修复）。节点合并（merge）可能会发生下溢传递。

•实现：p224/245

•复杂度：在存有N个关键码的m阶B-树中的每次关键码删除操作，都可以在*O*(logmN)时间内完成。另外同样地，因某一关键码的删除而导致Ω(logmN)次合并操作的情况也极为罕见，单次删除操作过程中平均只需做常数次节点的合并。

## 8.3 红黑树（red-black tree）

•红黑树是AVL树的改进版，尽管最坏情况下需对多达Ω(logn)个节点重染色，但就分摊意义而言仅为*O*(1)个。

•适度平衡标准：任一节点左、右子树的高度，相差不得超过两倍。

8.3.1 概述

•为了便于理解与分析，通常引入假想外部节点，使得二叉树变为真二叉树。红黑树中对于叶节点有特别定义，即叶节点就是外部节点（或红黑树中不再提及叶节点）。而B-树中，叶节点指非外部且最靠近外部的一层顶点。

•红黑树定义：由红、黑两色节点组成的二叉搜索树若满足以下条件，即为红黑树

(1) 树根始终为黑色

(2) 外部节点均为黑色

(3) 其余节点若为红色，则其孩子节点必为黑色

(4) 从任一外部节点到根节点的沿途，黑节点的数目相等

•在从根节点通往任一节点的沿途，黑节点都不少于红节点。除去根节点本身，沿途所经黑节点的总数称作该节点的黑深度（black depth）根节点的黑深度为0，其余依此类推。

•从外部节点开始通往任一节点的沿途，所经黑节点的总数称作该节点的黑高度（black height），所有外部节点的黑高度均为0，其余依此类推。

•根节点的黑高度亦称作全树的黑高度，在数值上与外部节点的黑深度相等。

•每棵红黑树都等价于一棵(2,4)-树（B-树）；转换过程为：将每个红节点所带子树与其父节点（必黑）合并成为一个B-树大节点。换言之，红节点不贡献黑高度，而黑节点不然，因此转换前后，（黑）高度是等价的。

•红黑树的每一节点都对应于(2,4)-树的一个关键码。(2,4)-树中的每个节点应包含且仅包含一个黑关键码，同时红关键码不得超过两个。而且，若某个节点果真包含两个红关键码，则黑关键码的位置必然居中。

•平衡性：包含n个内部节点的红黑树T的高度h有以下范围

虽然平衡性不如AVL树，但是仍然保持适度平衡

8.3.2 红黑树接口定义

p230/251

8.3.3 节点插入算法

•插入新节点都取红色（也可以取黑色，这里求简便）因此可能存在双红现象（double red）

•双红修正存在两种情况：RR-1, RR-2，前者需要重构（旋转）和染色，但不会传递双红现象，后者只需重染色，但可能会传递双红。

•复杂度：O(logn)

•双红修正实现：p233/254

8.3.4 节点删除算法

•节点删除的几种情况：实际待删除节点为x，并且已经和原位置的直接后继替换，故其左孩子必为外部节点。

①若x为红，则其右孩子r为外部节点，可直接删除x及左孩子，并替换为r

②若x为黑，其右孩子r若为红，则r的双子均为外部节点，可删去x，并替换为r，再将r染为黑色。

③若x为黑，其右孩子r为外部节点，则会出现所谓的双黑现象（double black），此时直接删除x，会导致该分支的黑高度变化进而影响全局平衡。

•双黑修正：假定s为x兄弟，p为二者父亲p235/256

BB-1：s为黑，s有一个红孩子，p颜色不限

BB-2-R：s为黑，s只有黑孩子，p为红色

BB-2-B：s为黑，s只有黑孩子，p为黑色

BB-3：s为红，p为黑色，则s的双子必为黑色节点，且所有孙子均为外部节点

注意：BB-3处理之后问题转化为BB-1或BB-2-R，才能解决。双黑现象不一定只是节点删除直接导致的，它只是一种红黑树的失衡状态，由BB-2-B导致的下一次双黑现象中r就不可能再是外部节点了。

•复杂度：全部操作时间总和最差不超过O(logn)，而唯一的需要迭代的BB-2-B情况，一旦某一次迭代不再是BB-2-B情况，则可常数时间内完成。双黑修正的整个过程，也仅涉及常数次的拓扑结构调整操作。

•算法实现：p238/259

## 8.4 kd-树（kd-tree）

8.4.1 范围查询（range query）

•批处理（batch）或离线方式（offline）：输入点集P通常会在相当长的时间内保持相对固定。数据的这种给出及处理方式称为批处理。

•对于同一输入点集，往往需要针对大量的随机定义的区间R，反复地进行查询。数据的这种给出及处理方式称作在线（online）方式。

•输出敏感的（output sensitive）算法：需要同时根据问题的输入规模和输出规模进行估计。一般地，时间复杂度可以这种形式给出的算法称为输出敏感。

•用平衡二叉搜索树解决一维范围查询问题，这种方式其意义在于可以推广至更高维。而BBST不一定是一维条件下的最佳解法。

•一维范围查询效率：每次查询都可在O(r+logn)时间内完成，算法效率和输出规模也有关。

8.4.2 kd-树

•kd-树：k-dimensional tree的缩写。在任何的维度下，kd-树都是一棵递归定义的平衡二叉搜索树。

•2d-树：树中的每个节点，都对应于二维平面上的某一矩形区域（还包括里面的输入点集），且其边界都与坐标轴平行。同层节点各自对应的矩形区域，经合并之后恰好能够覆盖整个平面（彼此没有交集），特别的，根节点一个点即代表整个平面。对于矩形区域有一种约定：每个矩形区域的左边和底边开放，右边和顶边封闭。

•构造2-d树：p242/263；对于每个节点所对应矩形区域的划分，按照每个节点只包含其父节点包含的输入点集的一半来划分。若当前节点深度为偶（奇）数，则沿垂直（水平）方向切分，所得子区域随同包含的输入点分别构成左、右孩子。如此不断，直至子区域仅含单个输入点。每次切分都在中位点（median point）（即按对应的坐标排序居中者）处进行，以保证全树高度不超过*O*(logn)。

8.4.3 基于2d-树的范围查询

•算法实现：p244/265

•复杂度：平面范围查询与一维情况不同，在同一深度上可能递归两次以上，并报告出多于两棵子树。但更精细的分析表明，被报告的子树总共不超过*O*()棵，累计耗时*O*()。

# 第九章 词典（Dictionary）

•词典结构，是由一组数据构成的集合，其中各元素都是由关键码和数据项合成的词条（entry）。映射（map）结构与词典结构一样，也是词条的集合。二者的差别仅仅在于，映射要求不同词条的关键码互异，而词典则允许多个词条拥有相同的关键码（有的地方可能正好反过来定义）。除了静态查找，映射和词典都支持动态更新，二者统称作符号表（symbol table）。因为差别不大，可以不必区分二者。

•词典可以实现为跳转表和散列表。对于后者，将转而依据数据项的数值，直接做逻辑查找和物理定位。此类结构所支持的这种新的数据访问方式，即所谓的循值访问（call-by-value）。

•搜索树中的key与value和词典（或称映射）中的key与value的区别在于：搜索树中的key必须能够比较大小，每个节点按照其key中序遍历非降次序排列。因此搜索树的key往往是数字。而词典并不强制要求这一点，顾名思义，词典的key可以是任何形式的，不必比较大小（虽然实际中词典里面的词条确实可按照某种次序排列，比如首字母顺序，但是即便没有这种次序，也可以工作）。

•循值访问不属于CBA算法的范畴，因为不再需要比较大小。因此也不适用CBA算法的相关结论

## 9.1 词典ADT

9.1.1 操作接口

|  |  |
| --- | --- |
| 操作接口 | 功能描述 |
| get(key) | 若词典中存在以key为关键码的词条，则返回该词条的数据对象；否则，返回NULL |
| put(key, value) | 插入词条(key, value)，并报告是否成功 |
| remove(key) | 若词典中存在以key为关键码的词条，则删除之并返回true；否则，返回false |

•在很多高级语言中，词典作为基本的数据类型，统称作关联数组（associate array）

9.1.2 操作实例

•本书中约定，跳转表允许同时保留多个关键码雷同的词条，查找时任意返回其一；散列表则维持原词条不变，返回插入失败标志，也就是说，更接近于映射的规范。而在Python和Perl在内的众多语言中，如果插入的关键码相同，词条内容则直接被替换。换句话说，它们不允许多个关键码雷同的词条，但put操作一定成功。

9.1.3 接口定义

•Dictionary模板类中的Entry类只需支持判等操作

9.1.4 实现方法

•跳转表和散列表为例介绍词典结构的两种实现方法。尽管它们都在底层引入了某种“序”，但这类“序”只是内部的一种约定；从外部接口来看，依然只有“相等”的概念。即关键码只需支持判等操作即可，不必要支持比较大小。

## 9.2 跳转表（skip list）

9.2.1 Skiplist模板类

p249/270

9.2.2 总体逻辑结构

•跳转表即一个普通列表，该列表的每个元素都是一个四联表（仍然是条形列表，但每个元素和上下前后四个实体有联系）。

•跳转表其内部由沿横向分层、沿纵向相互耦合的多个列表（称作四联表）组成，h称作跳转表的高度。每一水平列表称作一层（level），其中S0和Sh分别称作底层（bottom）和顶层（top）。同层节点之间可定义前驱与后继关系；层次不同的节点（但是关键码相同）可能沿纵向组成塔（tower），同一塔内的节点以高度为序也可定义前驱与后继关系。高层列表总是低层列表的子集，其中特别地，S0包含词典中的所有词条，而Sh除头、尾哨兵外不含任何实质的词条。不难看出，跳转表的层高h必然决定于最大的塔高。

9.2.3 四联表（quadlist）

•定义：，跳转表内各节点沿水平和垂直方向都可定义前驱和后继，支持这种联接方式的表称作四联表（quadlist）

•四联表模板类p250/271

•四联表节点p251/272

•初始化构造p252/273

9.2.4 查找

•查找算法：get()和skipSearch()接口  
查找x，从最顶层的头哨兵p节点出发，先向右查找，若遇到比x大的节点或尾哨兵仍未找到x，则从不大于x的最大节点向下一层，再继续向右查找，如此重复直至无法继续进行报告失败或锁定x报告成功。

9.2.5 空间复杂度

•为保证跳转表的效率，跳转表的生长过程必须被控制，要满足“生长概率逐层减半”条件：

对于任意0≤ k < h，Sk中任一节点在Sk+1中依然出现的概率，始终为1/2

即S0中任一关键码依然在Sk中出现的概率，等于2-k

•空间总体消耗量的期望值应为O(n)

9.2.6 时间复杂度

•期望高度，纵向跳转次数和横向跳转次数的估算：p254/275

•时间复杂度为：O(logn)

9.2.7 插入

•put()实现：p255/276

•新塔每长高一层，塔顶节点除须与原塔纵向联接，还须与所在列表中的前驱与后继横向联接。

•insertAfterAbove() p257/278

•insertAsSuccAbove() p257/278

•时间复杂度：O(logn)，运行时间期望值的上界

9.2.8 删除

Skiplist::remove()和Quadlist::remove() p258/279

•删除的过程和插入相反，从塔顶依次删除，直到彻底删除关键码

•跳转表词条删除操作所需的时间不超过*O*(h) = *O*(logn)

## 9.3 散列表（hashtable）

9.3.1 完美散列

•散列表（hashtable）是散列方法的底层基础，逻辑上由一系列可存放词条（或其引用）的单元组成，故这些单元也称作桶（bucket）或桶单元；与之对应地，各桶单元也应按其逻辑次序在物理上连续排列。因此，这种线性的底层结构用向量来实现再自然不过。为简化实现并进一步提高效率，往往直接使用数组，此时的散列表亦称作桶数组（bucket array）。若桶数组的容量为R，则其中合法秩的区间[0, R)也称作地址空间（address space）。

•散列函数（hash function）：一组词条在散列表内部的具体分布，取决于所谓的散列（hashing）方案，事先在词条与桶地址（或散列地址hashing address）之间约定的某种映射关系，可描述为从关键码空间到桶数组地址空间的函数。

•完美散列（perfect hashing）：时间性能保证查找插入或删除均在常数时间内，空间性能保证连续的每个桶恰好存放一个所需数据，无空余和重复，称作完美散列，然而并不常见。

9.3.2 装填因子与空间利用率

•常见问题：尽管词典中实际需要保存的词条数N（远远少于可能出现的词条数R，但R个词条中的任何一个都有可能出现在词典中。

•装填因子（load factor）：将散列表中非空桶的数目与桶单元总数的比值称作装填因子。

9.3.3 散列函数

•散列函数hash()的作用可理解为，将关键码空间[0, R)压缩为散列地址空间[0, M)。通常有R>>M>N。（假定关键码均为[0, R)范围内的整数。将词典中的词条数记作N，散列表长度记作M）

•散列函数具备的条件：

①确定性：无论所含的数据项如何，词条E在散列表中的映射地址hash(E.key)必须完全取决于其关键码E.key

②简单性：映射过程自身不能过于复杂，唯此方能保证散列地址的计算可快速完成，从而保证查询或修改操作整体的*O*(1)期望执行时间

③广阔性：所有关键码经映射后应尽量覆盖整个地址空间[0, M)，唯此方可充分利用有限的散列表空间。也就是说，函数hash()最好是满射。

④均匀性：关键码映射到各桶的概率应尽量接近于1/M

•散列冲突（collision）：因定义域规模R远远大于取值域规模M，hash()不可能是单射。这就意味着，关键码不同的词条被映射到同一散列地址的情况。散列冲突无法彻底避免，可以通过冲突排解解决，但是仍应以散列函数的设计降低发生几率从而提高效率。

•词条聚集（clustering）：词条集中到散列表内少数若干桶中（或附近）的现象

(1)除余法（division method）

hash(key) = key mod M

•将key映射至关于散列表长度整除的余数，应将M取作素数，并和词条的key的周期（有时候key以一定的规律存在，这种周期表现为关键码有一定的间隔T）的素因子避开。

(2)MAD法（multiply-add-divide method）

hash(key) = (a \* key + b) mod M，M仍为素数，a > 0，b > 0，且a mod M ≠ 0

•改进的除余法，可以消除散列的连续性。对于原除余法，显然连续的关键码映射的散列地址往往也是连续的，违背了均匀性原则。

(3)更多散列函数p264/285

•为保证上述函数取值落在合法的散列地址空间以内，通常都还需要对散列表长度M再做一次取余运算

(4)（伪）随机数法

hash(key) = rand(key) mod M

•这一策略的原理也可理解为，将“设计好散列函数”的任务，转换为“设计好的（伪）随机数发生器”的任务。幸运的是，二者的优化目标几乎是一致的

•需特别留意的是，由于不同计算环境所提供的（伪）随机数发生器不尽相同，故在将某一系统中生成的散列表移植到另一系统时，必须格外小心

9.3.4 散列表

•Hashtable模板类 p264/285

•散列表的构造 p265/286

•散列表析构 p266/287

9.3.5 冲突及其排解

•经过计算，实际问题中，至少发生一次散列地址冲突的概率是很高的。

(1) 多槽位法（multiple slots）

•令相互冲突的每组词条构成小规模的子词典，为每个散列地址开辟出更多的槽位，好比为每个地址建造更多的房间容纳地址冲突的关键码。缺点是每个地址建造数量一致的空余房间，效率极低。

(2) 独立链法（separate chaining）

•与多槽位法类似，只是采用列表来实现子词典，用多少建造多少。缺点是一旦发生冲突，仍然需要遍历列表来查找。

(3) 公共溢出区法（overflow area）

•在原散列表之外另设一个词典结构，专门存储发生冲突的词条。在冲突不频繁的场合可以使用。

9.3.6 闭散列策略

•开放定址（open addressing）：每个桶单元都有可能存放任一词条，散列地址空间对所有词条开放的策略特点。严格遵循散列函数，不能随意存放词条的策略称为封闭定址（closed addressing）

•闭散列（closed hashing）：可用的散列地址仅限于散列表所覆盖的范围之内。上一小节采用的另构建其他附加数据结构存储冲突词条的策略，称为开散列（open hashing）。

•采用闭散列方法时，因不得使用附加空间，装填因子需要适当降低，通常都取λ≤0.5。装填因子是影响闭散列策略效率的最重要因素。而对于比如独立链法，装填因子λ＜0.9。

•闭散列策略比开散列方法应用更为广泛。

(1) 线性试探法（linear probing）

•插入时遇到冲突的关键码则向下一地址进行试探，直到找到一个空位置存放。遇到存储向量尾端时返回前端。查找时则从冲突地址开始顺序查找，直到找到目标或遇到空桶单元宣告失败。查找序列称为查找链（probing chain）。

•一条查找链可能因为多个冲突的关键码位置相近，因此多个冲突的关键码相互交叠。这确实会导致效率降低，因此装填因子不能取太大。由此省略的I/O操作仍然是决定效率的关键。

•懒惰删除（lazy removal）：是指在遇到删除情况时，查找链会因为中间桶单元清空而断裂。设置一个标记使得该空桶单元仍能被查找链识别，同时也能作为空桶插入新的关键码。

(2) 重散列（rehashing）

•即当因插入操作导致散列表的装填因子越过上界时，将原有的所有词条重新散列至新的扩容散列表中。新的散列表容量至少x2。其道理和向量扩容类似。这一策略也可使重散列所耗费的时间，在分摊至各次操作后可以忽略不计。

9.3.7 查找与删除

•get(), probe4Hit(), remove()算法实现 p271/292

9.3.8 插入

•put(), probe4Free()算法实现 p272/293

9.3.9 更多闭散列策略

(1) 平方试探法（quadratic probing）

•线性试探是从冲突地址向后连续试探，容易造成数据聚集。平方试探也是从冲突地址开始向后试探，但是采用依据公式的跳跃式试探。按如下规则确定第j次试探的桶地址：

(hash(key) + j2) mod M, j = 0,1,2,…

•鉴于目前常规的I/O页面规模已经足够大，只有在查找链极长的时候，才有可能引发额外的I/O操作。因此不必担心平方试探逐步增加的跳跃距离会破坏数据局部性。

•可以证明，只要散列表长度M为素数且装填因子λ≤0.5，则平方试探迟早必将终止于某个空桶。因此保证这两条基本条件就不必担心平方试探法无法终止。

(2) （伪）随机试探法（pseudo-random probing）

rand(j) mod M

•道理同伪随机散列函数

(3) 再散列法（double hashing）

•选取一个适宜的二级散列函数hash2()，一旦在插入词条(key, value)时发现ht[hash(key)]已被占用，则以hash2(key)为偏移增量继续尝试，直到发现一个空桶。

[hash(key) + 1×hash2(key)] % M

[hash(key) + 2×hash2(key)] % M

[hash(key) + 3×hash2(key)] % M

9.3.10 散列码转换

•概念：利用某一种散列码转换函数hashCode()，将关键码key统一转换为一个整数，称作散列码（hash code），然后，再利用散列函数将散列码映射为散列地址。通过这种方式来支持任意类型的关键码。也就是说实际上的，从关键码key到散列地址address的映射分为两步。

•散列码转换函数应具备的条件：

(1) 为支持后续尺度不同的散列空间，以及种类各异的散列函数，作为中间桥梁的散列码，取值范围应覆盖系统所支持的最大整数范围。

(2) 各关键码经hashCode()映射后所得的散列码，相互之间的冲突也应尽可能减少.

(3) hashCode()也应与判等器保持一致。也就是说，被判等器判定为相等的词条，对应的散列码应该相等；反之亦然。

•散列码转换方法举例：

①强制转换为整数

对于byte、short、int和char等本身即可表示为不超过32位整数的数据类型，可直接将它

们的这种表示作为其散列码。即使用强制类型转换。

②对成员对象求和

long long和double之类长度超过32位的基本类型，将高32位和低32位分别看作两个32位整数，将二者之和作为散列码。这一方法，可推广至由任意多个整数构成的组合对象。比如，可将其成员对象各自对应的整数累加起来，再截取低32位作为散列码。

③多项式散列码（polynomial hash code）

用于字符串类型关键码的转换，对于x0x1…xn-1的字符串，取常数a≥2，散列码可表示为：

x0an-1 + x1an-2 + ... + xn-2a1 + xn-1

其中的常数a非常关键，为尽可能多地保留原字符串的信息以减少冲突，其低比特位不得全为零。另外，针对不同类型的字符串，应通过实验确定a的最佳取值。实验表明，对于英语单词之类的字符串，a = 33、37、39或41都是不错的选择。

## 9.4 散列应用

9.4.1 桶排序（bucketsort）

•定义：借助一组桶单元实现对一组关键码的分拣。即使用散列表数据结构进行排序。

•算法：只需将待排序的关键码用适当的散列函数映射到适合的位置，即可自动完成排序。对于重复的关键码，采用独立链法处理即可。前提是，必须知道待排序的关键码或散列码的范围，才能创建对应长度的散列表。

•效率：M为散列表长度，也即待排序数据可能的取值情况数，n为待排序数据总量。算法所用散列表共占O(M)空间。散列表的创建和初始化耗时O(M)，将所有关键码插入散列表耗时O(n)，依次读出非空桶中的关键码耗时O(M)，故总体运行时间为O(n + M)。一般情况下n>>M，则可近似为O(n)，突破了CBA排序的Ω(nlogn)。

9.4.2 最大间隙（maximum gap）

•描述：任意n个互异点都将实轴切割为n + 1段，除去最外侧无界的两段，其余有界的n - 1段中何者最大？

•使用桶排序代替传统排序来提高效率，其他部分算法类似。

•效率：空间方面，除了输入本身这里只需维护一个散列表，共占用*O*(n)的辅助空间。，该算法的每一步均耗时*O*(n)。故累计不超过*O*(n)。

9.4.3 基数排序（radixsort）

•字典序（lexicographical order）：任意两个关键码之间的大小关系，取决于它们第一个互异的域，一般左边的域或字段优先级更高。

•基数排序：即低位优先（least significant digit first）的多趟桶排序，低位指优先级低的字段或域或位。比如，两位数十位比个位优先级高，个位是低位；纸牌花色的优先级高于数字等。只需按照优先级递增的次序针对每一字段各做一趟桶排序，即可实现按整个关键码字典序的排序。对于某一次桶排序大小相同的情况，因为二者前次已经比较过低位大小，则插入顺序必定是较大者靠后，亦即较大者更靠近独立链末端。

•效率：设各字段取值范围为[0, Mi)，1<i<t。若记

M = max{ M1, M2, ..., Mt }

则总体运行时间不超过：

O(n + M1) + O(n + M2) + ... + O(n + Mt)

= O(t \* (n + M))

# 第十章 优先级队列（Priority Queue）

•循优先级访问（call-by-priority）：根据数据对象之间相对优先级对其进行访问的方式

•搜索树结构和词典结构，都支持覆盖数据全集的访问和操作。其中存储的每一数据对象都可作为查找和访问目标。因此，搜索树结构需要在所有元素之间定义并维护一个显式的全序（full order）关系。而词典结构中的数据对象之间，尽管不必支持比较大小，但在散列表之类的具体实现中，都从内部强制地在对象的数值与其对应的秩之间，建立起某种关联（尽管实际上这种关联通常越“随机”越好），从而隐式地定义了一个全序次序。

## 10.1 优先级队列ADT

10.1.1 优先级与优先级队列

•优先级（priority）：数据项的某种属性只要可以相互比较大小，则这种关系即可称为。

•优先级队列：按照事先约定的优先级，可以始终高效查找并访问优先级最高数据项的数据结构

•效率：优先级队列的所有ADT操作均可在O(logn)时间内完成。

10.1.2 关键码，比较器与偏序关系

•优先级队列中数据项也称作词条（entry）并具备关键码（key），而key必须可以比较大小

10.1.3 操作接口

|  |  |
| --- | --- |
| size() | 报告优先级队列规模 |
| insert() | 插入指定词条 |
| getMax() | 返回优先级最大词条 |
| delMax() | 删除优先级最大词条 |

•本章允许在同一优先级队列中出现关键码雷同的多个词条，故insert()操作必然成功，因此该接口自然不必返回操作成功标志。

10.1.4操作实例：选择排序

10.1.5 接口定义

p284/305

10.1.6 应用实例：Huffman编码树

## 10.2 堆（heap）

•优先级队列虽然名称上有队列二字，但是实际实现依靠的是堆结构。

•堆有多种实现形式，CBH只是最基本的。

10.2.1 完全二叉堆（complete binary heap）

•完全二叉堆的条件：

①结构性：其逻辑结构须等同于完全二叉树。

②堆序性：就优先级而言，堆顶以外的每个节点都不高（大）于其父节点。

•大顶堆与小顶堆：堆顶是优先级最高的词条的堆为大顶堆，反之为小顶堆。

•完全二叉堆按照层次遍历的次序从根节点到最后一个节点顺序组成一个向量，若将节点v的编号（秩）记作i(v)，则

①若v有左孩子，则i(lchild(v)) = 2∙i(v) + 1；

②若v有右孩子，则i(rchild(v)) = 2∙i(v) + 2；

③若v有父节点，则i(parent(v)) ==

•宏定义 p288/309

•PQ\_ComplHeap模板类 p288/309

注意完全二叉堆没有继承二叉树只是继承了向量和PQ接口

•getMax() p288/309

10.2.2 元素插入

•算法步骤：

①调用向量的标准插入接口，将新词条接至向量的末尾

②调用percolateUp()函数，对新接入的词条做适当调整（即与其父节点交换位置），在保持结构性的前提下恢复整体的堆序性，这个过程称为上滤（percolate up）

•效率：最多交换次数不超过全堆高度，每次交换只需常数时间，即O(logn)。而最坏情况往往很少发生。

•算法实现：p290/311 改进：习题[10-3],[10-4]

10.2.3 元素删除

•delMax()步骤：

①删除堆顶，并将最末尾词条转移至堆顶

②调用percolateDown()函数调整新堆顶（即与其优先级更加接近根部的孩子交换位置），在保持结构性的前提下，恢复整体的堆序性

•效率：最多交换次数不超过全堆高度，每次交换只需常数时间。

•算法实现：p292/313 改进：习题[10-3],[10-4]

10.2.4 建堆（heapification）

•蛮力算法：对任何一棵完全二叉树，只需自顶而下、自左向右地针对其中每个节点实施一次上滤，即可使之成为完全二叉堆。效率为O(nlogn)

•Floyd算法：自下而上、由深而浅地遍历所有内部节点，并对每个内部节点分别调用一次下滤算法percolateDown()。效率为O(n)

•算法效率区别在于，节点上滤的时间最坏正比于其深度，下滤时最坏正比于其高度。而显然深度大高度小的节点一般是最多的。

10.2.5 就地堆排序（in-place heapsort）

•算法的总体思路和策略与选择排序算法基本相同：将所有词条（以向量形式给出）分成未排序和已排序两类，不断从前一类中取出最大者，顺序加至后一类中。而从开始排序之前，就始终将未排序部分构建为一个堆。

•实现：p297/318

•效率：仍为O(nlogn)，但是在常系数尺度上效率高于其他O(nlogn)算法。

## 10.3 左式堆（leftist heap）

10.3.1 堆合并

•第一种算法：将两个堆中规模较小的一个不断取delMax()，插入进较大的堆中，效率为O(mlogn)。其中，n和m为二者规模，且n≥m

•第二种算法：Floyd算法，道理同建堆过程，两种算法的缺点都在于无法利用二堆中本已各自存在的偏序性。

10.3.2 单侧倾斜

•左式堆是优先级队列的另一种实现方式，和完全二叉堆相比，它仍然保持堆序性，但是打破了结构性，而大致形状始终向左倾斜。

10.3.3 PQ\_LeftHeap模板类

p298/319 左式堆继承二叉树和PQ接口

10.3.4 空节点路径长度（null path length）

•定义：节点x的npl等于该节点x到外部节点的最近距离，记作npl(x)。若x为外部节点，则npl等于0。若为内部节点，则有递归定义npl(x) = 1 + min(npl(lc(x)), npl(rc(x)))。npl也等于以x为根的最大满子树的高度。

10.3.5 左倾性与左式堆

•左倾性：即任一内部节点x都满足npl(lc(x)) ≥ npl(rc(x))

•左式堆即满足左倾性（取代完全二叉堆的结构性）的二叉堆，注意左倾性不能直观从树高来判断。左式堆中可以存在左孩子高度低于右孩子的节点。

•由左倾性和npl定义，左式堆中的任一内节点x满足：npl(x) = 1 + npl(rc(x))，即左式堆中每个节点的npl值，仅取决于其右孩子。

10.3.6 最右侧通路（rightmost path）

•定义：从x出发沿右侧分支一直前行直至空节点，经过的通路。记作rPath(x)。每个节点的npl值，应恰好等于其最右侧通路的长度。

•对于根节点r，rPath(r)的终点必为全堆中深度最小的外部节点，若rPath(r)=d，则该堆深度最大的满二叉子树高度即为d。该满二叉树至少应包含2d+1 -1个节点、2d - 1个内部节点。在包含n个节点的左式堆中，最右侧通路必然不会长于 = *O*(logn)

10.3.7 合并算法 p300/321

10.3.8 实例p301/322

10.3.9 合并操作的实现 p302/323

10.3.10 复杂度 p302/323 用迭代代替递归的改进：习题[10-15]

10.3.11 基于合并的插入和删除

•由于merge()算法的简洁性，左式堆的插入和删除操作可以不用之前常规的算法，而基于合并来实现。

(1) delMax() p303/324

删除节点之后将左右子堆merge即可，效率仍为O(logn)，但接口实现更简洁。

(2) insert() p303/324

直接将待插入节点视作一个堆和另一个堆merge即可，效率仍为O(logn)。

# 第十一章 串（String）

## 11.1 串及串匹配

11.1.1 串

•一般地，由n个字符构成的串记作：S = "a0 a1 ... an-1"， 其中，ai，0≤i<n，这里的Σ是所有可用字符的集合，称作字符表（alphabet）。字符串S所含字符的总数n，称作S的长度，记作|S| = n。这里只考虑长度有限的串，n <∞。特别地，长度为零的串称作空串（null string）。

•字符串中任一连续的片段，称作其子串（substring）。具体地，对于任意的0≤i≤i +k < n，由字符串S中起始于位置i的连续k个字符组成的子串记作：S.substr(i, k) = "ai ai+1 ... ai+k-1" = S[i, i + k)

•有两种特殊子串：起始于位置0、长度为k的子串称为前缀（prefix），而终止于位置n - 1、长度为k的子串称为后缀（suffix），分别记作：

prefix(S, k) = S.substr(0, k) = S[0, k)

suffix(S, k) = S.substr(n - k, k) = S[n - k, n)

•空串是任何字符串的子串，也是任何字符串的前缀和后缀；任何字符串都是自己的子串，也是自己的前缀和后缀。此类子串、前缀和后缀分别称作平凡子串（trivial substring）、平凡前缀（trivial prefix）和平凡后缀（trivial suffix）。反之，字符串本身之外的所有非空子串、前缀和后缀，分别称作真子串（proper substring）、真前缀（proper prefix）和真后缀（proper suffix）。

•串ADT支持的操作接口 p307/328

11.1.2 串匹配

•串模式匹配（string pattern matching），简称串匹配

•问题描述1：如何在字符串数据中，检测和提取以字符串形式给出的某一局部特征

•问题描述2：对基于同一字符表的任何文本串T（|T| = n）和模式串P（|P| = m）：

判定T中是否存在某一子串与P相同；若存在（匹配），则报告该子串在T中的起始位置。一般2<<m<<n

•问题分类：

①模式检测（pattern detection）问题：我们只关心是否存在匹配而不关心具体的匹配位置，比如垃圾邮件的检测。

②模式定位（pattern location）问题：若经判断的确存在匹配，则还需确定具体的匹配位置，比如带毒程序的鉴别与修复。

③模式计数（pattern counting）问题：若有多处匹配，则统计出匹配子串的总数，比如网络

热门词汇排行榜的更新。

④模式枚举（pattern enumeration）问题：在有多处匹配时，报告出所有匹配的具体位置，比如网络搜索引擎。

11.1.3 测评标准与策略

•常用的时间复杂度评估对于这里不适用，有效涵盖成功匹配情况的一种简便策略是，随机选取文本串T，并从T中随机取出长度为m的子串作为模式串P。这也是本章将采用的评价标准。

## 11.2 蛮力算法

11.2.1 算法描述 p309/330

•即将T与P逐个进行比对，失败时，使P沿着T向后移动一个字符继续从头比对

11.2.2 算法实现 p309/330

11.2.3 时间复杂度 p310/331

## 11.3 KMP算法

11.3.1 构思

•如果有比对成功的情况，则必然显示了T中的实际信息，而可以针对性的调整P的下一次比对位置，而没有必要一次一步的顺次位移

11.3.2 next表

•若模式串P经适当右移之后，能够与T的某一（包含T[i]在内的）子串完全匹配，则一项必要条件就是：P[0, t) = T[i - t, i) = P[j - t, j)。亦即，在P[0, j)中长度为t的真前缀，应与长度为t的真后缀完全匹配，故t必来自集合：N(P, j) = { 0≤t<j | P[0, t) = P[j - t, j) }

•该集合可能包含多个这样的t。但需要特别注意的是，其中具体由哪些t值构成，仅取决于模式串P以及前一轮比对的首个失配位置P[j]，而与文本串T无关！

•当有多个值得试探的右移方案时，应该保守地选择其中移动距离最短者。于是，next[j] = max( N(P, j) )

•对于任一模式串P，不妨通过预处理提前计算出所有位置j所对应的next[j]值，并整理为表格以便此后反复查询

11.3.3 KMP算法

•Knuth和Pratt师徒，与Morris几乎同时发明了这一算法。他们稍后联合署名发表该算法，并以其姓氏首字母命名

•KMP主算法（待改进版） p313/334

11.3.4 next[0] = -1

•若在某一轮比对中首对字符即失配，则应将P直接右移一个字符，然后启动下一轮比对。不妨假想地在P[0]的左侧“附加”一个P[-1]，且该字符与任何字符都是匹配的。就实际效果而言，这一处理方法完全等同于“令next[0] = -1,P[-1]=万用字符”。

11.3.5 next[j+1]

•已知next[0,j]，可以递推next[j+1]。道理上，如果next[j]=t，即P[0,t)=P[j-t,j)，即P[0,j)范围内存在前后缀相同关系。若P[t]=P[j]，则next[j+1]=next[j]+1=t+1，说明前缀和后缀的相同关系延续。反之若P[t]≠P[j]，则一般来说前后缀相同的关系破裂，则next[j+1]=0。除非P[0,t]内仍然存在前后缀相同关系，可由next[next[j]]=next[t] > 0判定，则继续检查这种前后缀相同关系是否延续，或直到范围内不存在前后缀相同关系为止。

11.3.6 构造next表

•next表的构造算法与KMP算法几乎完全一致。这一构造过程完全等效于模式串的自我匹配，因此两个算法在形式上的近似亦不足为怪。 p314/335

11.3.7 性能分析

•尽管可能有Ω(n)个对齐位置，但就分摊意义而言，在每一对齐位置仅需O(1)次比对（习题[11-4]）。KMP算法本身的运行时间不超过*O*(n)。，next表的构造仅需O(m)时间。综上可知，KMP算法的总体运行时间为*O*(n + m)。

11.3.8 继续改进

•原方法只是记忆了成功的匹配经验，适当跳过已经匹配成功并获得信息的部分。但是没有吸取失败匹配的经验。原方法每一次失败匹配后，接替P[j]下一次和T[i]进行比较的将是P[next[j]]=P[i]，若P[i]=P[j]，则这种比较没有意义。因此可以适当改进next表，集合N(P, j)的定义修改为：N(P, j) = { 0≤t<j | P[0, t) = P[j - t, j) 且 P[t]≠P[j] }。实际运行时，每次在P[0, j)中发现长度为t的真前缀和真后缀相互匹配之后，还需进一步检查P[j]是否等于P[t]。唯有在P[j]≠P[t]时，才能将t赋予next[j]；否则，需转而代之以next[t]。

•改进后的next表，看似不能再采用之前的next[j+1]递推法。实际上不然，仍然用t保存最大的自匹配长度，而当P[j+1]=P[t+1]时用next[t+1]代替t+1，并没有改变t的值。

•改进的next表构造算法：p316/337

## 11.4 BM算法

•由R. S. Boyer和J. S. Moore于1977年发明

11.4.1 思路与框架

•BM算法中，模式串P与文本串T的对准位置依然“自左向右”推移，而在每一对准位置却是“自右向左”地逐一比对各字符。一旦发生失配，则通过特有的启发式策略确定最大的P的位移。然后重复操作。

11.4.2 坏字符策略

•若模式串P当前在文本串T中的对齐位置为i，且在这一轮自右向左将P与substr(T, i, m)的比对过程中，在P[j]处首次发现失配：T[i + j] = 'X' ≠ 'Y' = P[j]则将'X'称作坏字符（bad character）。

•策略：若失配，找出P中的每一字符 'X'，分别与T[i + j] = 'X'对准，并执行一轮自右向左的扫描比对。P的位移量仅取决于原失配位置j，以及'X'在P中的秩，而与T和i无关！

•bc[]表：即处理失配坏字符情况时，返回T[i+j]和下一个P中的比对位置k。作用显然和next表类似。具体定义如下：

即为了不致漏掉可能匹配并且有多种k值的情况，k值选取最靠右的；若k=bc[c]＞j，则此时，下一次跳转会导致P实际向左移动，而这并不必要，此时可以简单的将P串向后移动一位（即相当于i++）继续从右至左比对即可。

•bc表构造算法 p319/340；若将BC表比作一块画布，则其中各项的更新过程，就犹如画家在不同位置堆积不同的油彩。而画布上各处最终的颜色，仅取决于在对应位置所堆积的最后一笔。这类算法，也因此称作“画家算法”（painter's algorithm）。

•bc表构造算法复杂度：O(|Σ|+m)，前者为字符表规模，m为P的规模。

11.4.3 好后缀策略

•每轮比对中的若干次（连续的）成功匹配，都对应于模式串P的一个后缀，称作“好后缀”（good suffix）。

•单纯依靠坏字符策略，无法将P右侧（后缀）中已经匹配成功的部分经验考量进来。

•策略与gs表：若发现失配，则P跳转之后的位置后的一段子串须与其失配时已经匹配的好后缀相同。若这样的位置多于一个，仍然选取最靠右的优先。但是还需保证P[k]≠P[j]，才能使下一次比较有意义。k作为P中新的比对位置，但是gs表中记录的是gs[j] = j - k，为P的向右位移量。如果找不到这样的子串，则从P的所有前缀中，找出可与U的某一（真）后缀相匹配的最长者，作为V(k)，并取gs[j] = m - |V(k)|。

•好后缀策略GS可以和坏字符策略BC同时使用，当两者都满足条件时，才确认为下一次跳转位置。否则直接完全跳过原P所覆盖的全部区段，这也是BM算法的特点。算法中可能出现失配位置的P的秩为大于1的负数，所以gs表使用的是P的首字符在T中对应位置i的位移。

•效率：对于匹配失败的情况，总体比对的次数不致超过O(n)。完全匹配时，则该算法在最坏情况下的效率，有可能退化至与蛮力算法相当。所幸，只要做些简单的改进，依然能够保证总体的比对次数不超过线性（习题[11-7]）。在兼顾了坏字符与好后缀两种策略之后，BM算法的运行时间为*O*(n + m)。

11.4.4 gs[]表构造算法

•MS[]串与ss[]表：对于任一整数j ∈ [0, m)，在P[0, j]的所有后缀中，考查那些与P的某一后缀匹配者。若将其中的最长者记作MS[j]，则ss[j]就是该串的长度|MS[j]|。特别地，当MS[j]不存在时，取ss[j] = 0。综上所述，可定义ss[j]如下：ss[j] = max{ 0 ≤ s ≤ j + 1 | P(j - s, j] = P[m - s, m) }特别地，当j = m - 1时，必有s = m，此时，有P(-1, m - 1] = P[0, m)。

•由ss[]表构造gs[]表：

①ss[j]=j+1：对于P[m - j - 1]左侧的每个字符P[i]而言，对应于前缀和好后缀部分匹配情况，m - j - 1都应该是gs[i]取值的一个候选。

②ss[j]≤j：对于字符P[m - ss[j] - 1]而言，对应于一般情况（P中其他区段与好后缀匹配），若同时还满足：P[m - ss[j] - 1] ≠ P[ j - ss[j] ]，则m - j - 1也应是gs[m - ss[j] - 1]取值的一个候选。

最后，因每一位置i所对应的gs[i]值只可能来自于以上候选。且gs[i]的最终取值是上述候选中的最小（最安全）者，故仿照构造bc[]表的画家算法，累计用时将不超过O(m)。

•ss表构造原理 p324/345

•gs表构造算法 p325/346

11.4.5 算法纵览

•BF（蛮力法），KMP，BM（仅BC），BM（BC+GS）四种典型算法的时间复杂度对比：p326/347；其中最后一种算法最优。

•基于单次比对成功概率进行的效率分析：p326/347；单次比对成功概率越高，一般越增加算法消耗的时间。而一般情况下，字符串所含字符种类较多，这一概率往往较低。因此更加体现出BC+GS算法的优势。而同时，蛮力算法效率也不太低。

•算法适用场合：习题[11-10]

## 11.5 Karp-Rabin算法

11.5.1 构思

•遵循凡物皆数（All things are numbers）原理，这一原理也类似的应用在散列技术上，将所有字符串中的字符和自然数之间建立联系。若字母表规模|Σ| = d，则任一字符串都将对应于一个d + 1进制的整数。

•选择d+1进制的数，因为字符不能对应数字0，否则0代表的字符无法置于首位。因此多增加一位制，而0不参与映射。

11.5.2 算法与实现

•以上散列并非满射，但不含'0'的任一d + 1进制值自然数，均唯一地对应于某个字符串。字符串经如此转换所得的散列码，称作其指纹（fingerprint）。

①数位与字长

②散列压缩

③散列冲突

④快速指纹更新

# 第十二章 排序

## 12.1 快速排序（quicksort）

12.1.1 分治策略

•归并排序的计算量主要消耗于有序子向量的归并操作，而子向量的划分却几乎不费时间。快速排序恰好相反，它可以在*O*(1)时间内，由子问题的解直接得到原问题的解；但为了将原问题划分为两个子问题，却需要*O*(n)时间

•该算法并不能保证最坏情况下的*O*(nlogn)时间复杂度，但易于实现，代码结构紧凑简练，而且对于按通常规律随机分布的输入序列，快速排序算法实际的平均运行时间较之同类算法更少。

12.1.2 轴点（pivot）

•定义：考查任一向量区间S[lo, hi)。对于任何lo ≤ mi < hi，以元素S[mi]为界，都可分割出前、后两个子向量S[lo, mi)和S(mi, hi)。若S[lo, mi)中的元素均不大于S[mi]，且S(mi, hi)中的元素均不小于S[mi]，则元素S[mi]称作向量S的一个轴点（pivot）。

•轴点位置mi的充要条件：设向量S经排序可转化为有序向量S'

① S[mi] = S'[mi]

② S[lo, mi)和S'[lo, mi)的成员完全相同

③ S(mi, hi)和S'(mi, hi)的成员完全相同

12.1.3 快速排序算法

•算法接口：p335/356

•原理：，轴点的位置一旦确定，则只需以轴点为界，分别递归地对前、后子向量实施快速排序；子向量的排序结果就地返回之后，原向量的整体排序即告完成。

12.1.4 快速划分算法（quick partitioning）

•思路：p336/357+p337/358

•代码实现：p336/357

•重复的元素也将按颠倒的次序转移至相对的一端，因而不再保持其原有的相对次序。由此可见，如此实现的快速排序算法并不稳定。

12.1.5 复杂度

•原有快速排序算法效率为O(n2)，其效率居然低到与起泡排序相近。因为无法保证采用原有的快速划分算法可以使所得的子任务规模上彼此相近，有很大几率选取了很大或很小的备选轴点而导致子问题失衡。

•降低最坏情况（备选轴点过大或过小）的几率：

①随机法：partition()算法在入口处增加了swap()一句，在区间内任选一个元素与\_elem[lo]交换。就其效果而言，这使得后续的处理等同于随机选择一个候选轴点

②三者取中法：从待排序向量中任取三个元素，将数值居中者作为候选轴点。这种方法优于前者。

•快速排序算法平均运行时间估算：p338/359

•正因为其良好的平均性能，加上其形象直观和易于实现的特点，快速排序算法自诞生起就一直受到人们的青睐，并被集成到Linux和STL等环境中。

12.1.6 应对退化

•退化情况即输入数据中含有大量重复元素的情况。原轴点构造算法无法高效处理这种情况。

•针对退化情况改进的轴点构造算法partition()：p340/361

•性能：退化的输入向量能够始终被均衡的切分，如此反而转为最好情况，排序所需时间为*O*(nlogn)。单趟partition()算法需做更多的元素交换操作。好在这并不影响该算法的线性复杂度。后一种算法倾向于反复交换重复的元素，故它们在原输入向量中的相对次序更难保持，快速排序算法稳定性的不足更是雪上加霜。

## 12.2 选取与中位数

12.2.1 概述

•k-选取：从与一组元素对应的有序序列S中，找出秩为k的元素S[k]，称作选取（selection）问题。若将目标元素的秩记作k，则亦称作k-选取（k-selection）问题

•作为k-选取问题的特例，0-选取即通常的最小值问题，而(n - 1)-选取问题即通常的最大值问题。在允许元素重复的场合，其中任何一个都可作为解答输出。

•中位数：在长度为n的有序序列S中，位序居中的元素S[]称作中值或中位数（median）。或对于尚未排序的序列，在对原数据集排序之后，对应的有序序列的中位数

•所谓中位数查找问题，也可以理解为是选取问题在k = 时的特例。中位数查找问题既是选取问题的特例，同时也是选取问题中的难度最大者。

12.2.2 众数

•定义：在任一无序向量A中，若有一半以上元素的数值同为m，则将m称作A的众数（majority）。若众数存在，则必然同时也是中位数。

•众数查找策略：减而治之p342/363

设P为向量A中长度为2m的前缀。若元素x在P中恰好出现m次，则A有众数仅当后缀A-P拥有众数，且A-P的众数就是A的众数。

•对于向量的每个秩i，该算法迭代且仅迭代一步。故其运行时间，因线性正比于向量规模。

12.2.3 归并向量的中位数

•问题描述：任给有序向量S1, S2，如何找出它们归并后所得有序向量S = S1S2的中位数？

•思想：采用减而治之的策略，S1的中位数m1 = S1[]和S2的逆向中位数m2 = S2[ - 1] =S2[]，并比较其大小。若m1 = m2，则二者均为S的中位数，若m1<m2（另外一种情况是对称的），则删去S1中前半部分和S2中后半部分之后，S的中位数不会发生变化。如此，问题规模缩减为一半，如此反复即可获得结果。

•算法实现：p344/365

•效率：O(log(min(n1,n2)))，子向量长度相等或接近时，此类问题的难度更大

12.2.4 基于优先级队列的选取

•蛮力算法的效率之所以无法令人满意，是因为花费足以全排序的计算成本，却仅得到了少量的局部信息，未免得不偿失。

•基于优先级队列的选取有三种算法：p346/367

①小顶堆删除法：O(n+klogn)

②大顶堆插入删除法：O(k+2(n-k)logk)

③大顶堆小顶堆交换法：O(n-k)+O(k)+min(k,n-k)\*2\*(O(logk+log(n-k)))

•在目标元素的秩很小或很大（即|n/2 - k| ≈ n/2）时，上述算法的性能都还不错。比如，k ≈ 0时，前两种算法均只需*O*(n)时间。然而很遗憾，当k ≈ n/2时，以上算法的复杂度均退化至蛮力算法的*O*(nlogn)

12.2.5 基于快速划分的选取

•思路：任做一次快速划分，获得一个轴点S[i]，如果i=k，则轴点即为被选取目标。如果i≠k，可以根据二者的大小关系，深入i的左或右侧继续递归寻找。

•算法实现：p347/368

•效率：尽管内循环仅需*O*(hi - lo + 1)时间，但很遗憾，外循环的次数却无法有效控制。与快速排序算法一样，最坏情况下外循环需执行Ω(n)次（习题[12-11]），总体运行时间为*O*(n2)。

12.2.6 k-选取算法 p348/369

•效率：O(20n)，常系数较大的线性复杂度

## 12.3 希尔排序（Shellsort）

12.3.1 递减增量（diminishing increment）策略

•思路：将待排序向量A[]重排为宽度为w（也称作增量increment）的二维向量B[][]。然后使w从某一极值开始缩小直到1，中间过程中对每个w的值进行一次二维向量重排（实际无需更改物理地址），并对每一列B[]进行一次排序。最后w=1时，B和A完全等价，相当于一次A的正常排序，但A此前已经由局部有序趋近于全序，所以排序效率将大大提高。

•底层排序算法：能够有效支持希尔排序的底层排序算法，必须是输入敏感的，比如3.5.2节所介绍的插入排序算法。

•重排：无需更改地址，只需按照公式关系来访问即可

对于任一固定的矩阵宽度w，A与B中元素之间总有一一对应关系：

B[i][j] = A[i + jw] 或 A[k] = B[k % w][k / w]

•尽管该算法在最坏情况下需要运行*O*(n2)时间，但随着向量的有序性不断提高（即逆序对的不断减少），运行时间将会锐减。

12.3.2 增量序列

•增量序列即选取的w的序列，最大的w值不能超过A的规模n。

(a)h-有序（h-ordered）与h-排序

•h-有序：在向量S[0, n)中，若S[i] ≤ S[i + h]对任何0 ≤ i < n - h均成立，则称该向量h-有序。也就是说，其中相距h个单元的每对元素之间均有序。

•h-排序：将原序列折叠为宽度为h的矩阵（或二维向量），即得到h个n/h维的一维子向量，对每个子向量分别排序即可。

•已经g-有序的向量，再经h-排序之后，依然保持g-有序，此时该向量既是g-有序的，也是h-有序的，称作(g, h)-有序。进一步的，必然(mg + nh)-有序。

•对于任意自然数g和h，只要m和n也是自然数，则f = mg + nh都称作g和h的一个组合（combination）。不能由g和h组合生成出来的最大自然数记作x(g, h)。如果g和h互素，则必有x(g, h) = (g - 1)∙(h - 1) - 1 = gh - g - h

(b)增量序列种类

①Shell序列：

*H*shell = { 1, 2, 4, 8, 16, 32, ..., 2k, ... }

在*H*shell中，首项之外的其余各项均为偶数。因此，在最后一步迭代之前，所有元素的秩依然保持最初的奇偶性不变。

②Papernov-Stasevic序列：

*H*ps = { 1, 3, 7, 15, 31, 63, ..., 2k - 1, ... }

其中相邻各项互素。我们将看到，采用这一增量序列，希尔排序算法的性能可以改进至*O*(n3/2)，其中n为待排序向量的规模。

③Pratt序列：

*H*pratt = { 1, 2, 3, 4, 6, 8, 9, 12, 16, ..., 2p3q, ... }

可见，其中各项除2和3外均不含其它素因子。可以证明，采用*H*pratt序列，希尔排序算法至多运行*O*(nlog2n)时间

④Sedgewick序列：

*H*sedgewick = { 1, 5, 19, 41, 109, 209, 505, 929, 2161, 3905, 8929, ... }

其中各项，均为：9∙4k - 9∙2k + 1

或4k - 3∙2k + 1的形式。

如此改进之后，希尔排序算法在最坏情况下的时间复杂度为*O*(n4/3)，平均复杂度为*O*(n7/6)。

更重要的是，在通常的应用环境中，这一增量序列的综合效率最佳。