

Computer Networking

From: James W. Kurose

Nobody

May 26, 2021

PART I

Data Structure and Algorithms

CHAPTER 1

Introduction

Algorithms + Data Structure = Programs

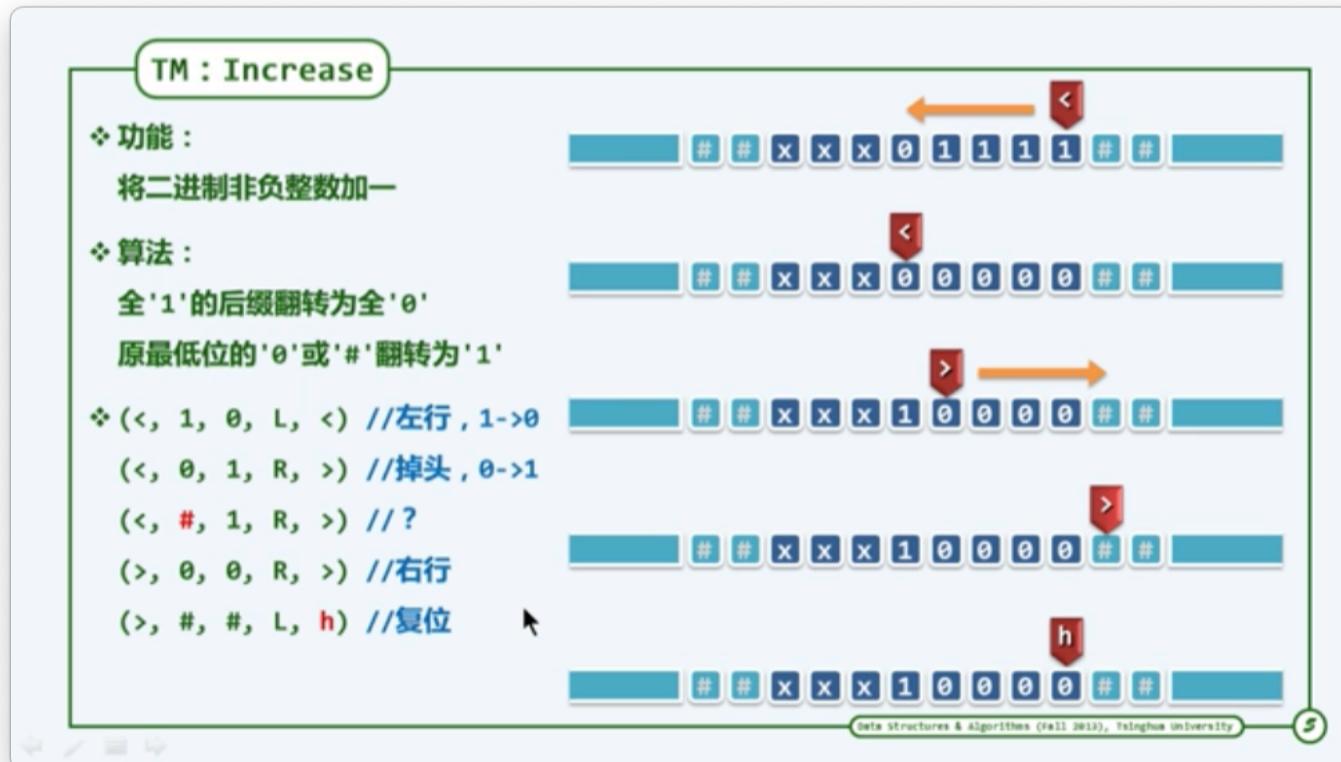
Algorithms + Data Structure × efficiency = Computation

1.1 computational models

1.1.1 measure

- 正确性
- 成本
 - 时间成本 $T_A(n) = \max \{T_0(P) | |P| = n\}$
 - 空间成本: 除了输入之外, 计算所要用的空间

Turning machine



Random Access Machine(RAM)

- 寄存器顺序编号, 总数没有限制
- 每一个基本操作仅需常数时间 (constant time)

TM 和 RAM 都是对于一般计算模型的抽象和简化，可独立于具体的平台，对算法的效率进行可信的比较和评判。
将运行时间转换成需要执行的基本操作次数

RAM: Floor

◆ 功能：向下取整的除法， $0 \leq c, 0 < d$

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

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

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

```

0   R[3] <- 1           //increment
1   R[0] <- R[0] + R[3] //c++
2   R[0] <- R[0] - R[1] //c -= d
3   R[2] <- R[2] + R[3] //x++
4   IF R[0] > 0 GOTO 2  //if c > 0 goto 2
5   R[0] <- R[2] - R[3] //else x-- and
6   STOP    //return R[0] = x = ⌊c/d⌋

```

Step IR R[0] R[1] R[2] R[3]

0	0	12	5	0	0
1	1	^	^	^	1
2	2	13	^	^	^
3	3	8	^	^	^
4	4	^	^	1	^
5	2	^	^	^	^
6	3	3	^	^	^
7	4	^	^	2	^
8	2	^	^	^	^
9	3	0	^	^	^
10	4	^	^	3	^
11	5	^	^	^	^
12	6	2	^	^	^

Data structures & algorithms (Fall 2013), Tsinghua University

1.1.2 渐进分析

big Ω 记号

- $T(n) = \Omega(f(n))$:
- $\exists c > 0$, 当 $n \gg 2$ 后, 有 $T(n) < c \cdot f(n)$

big Θ 记号

- $T(n) = \Theta(f(n))$:
- $\exists c_1 > c_2 > 0$, 当 $n \gg 2$ 后, 有 $c_1 \cdot f(n) > T(n) > c_2 \cdot f(n)$
- Θ 是 Ω 和 O 的组合

大 O 记号 (big-O notation)

- 随着计算规模的增长, 计算成本考察的是增长趋势, 而不考虑具体的操作次数和存储单元 → Asymptotic analysis
 - $T(n) = O(f(n))$ iff $\exists c > 0$, 当 $n \gg 2$ 后, 有 $T(n) < c \cdot f(n)$
 - $\sqrt{5n \cdot [3n \cdot (n+2) + 4] + 6} < \sqrt{5n \cdot [6n^2 + 4] + 6} < \sqrt{35n^3 + 6} < 6 \cdot n^{1.5} = O(n^{1.5})$
 - 常系数可忽略
 - 低次项可忽略
 - upper-bound 在 n 比较小的时候, 不一定会在 $T(n)$

(1) constant function

- $2 = 2013 = 2013 \times 2013 = O(1)$ 甚至 $2013^{2013} = O(1)$

(2) 对数 $O(\log n)$

- 常底数无所谓: $\forall a, b > 0, \log_a n = \log_a b \cdot \log_b n = O(\log_b n)$

- 常数次幂无所谓: $\forall c > 0, \log n^c = c \cdot \log n = O(\log n)$
- 对数多项式 (poly-log function): $123 * \log^{321} n + \log^{105} (n^2 - n + 1) = O(\log^{321} n)$
- 这类算法非常有效, 复杂度无限接近与常数: $\forall c > 0, \log n = O(n^c)$

(3) 多项式 (polynomial function)

- 一般地: $a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0 = O(n^k), a_k > 0$
- tractable
- 线性 (linear function): 所有 $O(n)$ 类函数

(4) 指数 (exponential function): $T(n)$

- $\forall c > 1, n^c = O(2^n)$
- 这类算法的计算成本增长极快, 通常认为不可忍受
- intractable, incomputable
- NP-complete: 不存在多项式时间内回答此问题的算法

(5) 从 $O(n^c)$ 到 $O(2^n)$, 是从有效算法到无效算法的分水岭

(6) 很多问题的 $O(2^n)$ 算法往往显而易见。然而, 设计出 $O(n^c)$ 算法却极其不易。甚至, 有时注定地只能是徒劳无功

1.2 算法分析

正确性 (不变性 \times 单调性) + 复杂度

1.2.1 C++ 指令类型

- C++ 基本指令, RAM 的基本指令
- 分支转向: goto
- 迭代循环: for(), while()
- 调用 + 递归 \rightarrow goto

1.2.2 复杂度分析的主要方法

- 迭代: 级数求和
- 递归: 递归跟踪 + 递归方程
- 猜测 + 验证

级数

- 算数级数: 与末项平方同阶
 $T(n) = 1 + 2 + \dots + n = n(n+1)/2 = O(n^2)$
- 幂方级数: 比幂次高出一阶
 $\sum_{k=0}^n k^d \approx \int_0^n x^{d+1} dx = \frac{1}{d+1} x^{d+1} \Big|_0^n = \frac{1}{d+1} n^{d+1} = O(n^{d+1})$
- 几何级数: 与末项同阶
 $T_a(n) = a^0 + a^1 + \dots + a^n = (a^{n+1} - 1)/(a - 1) = O(a^n)$
 $1 + 2 + 4 + \dots + 2^n = 2^{n+1} - 1 = O(2^{n+1}) = O(2^n)$
- 收敛级数
 $1/1/2 + 1/2/3 + 1/3/4 + \dots + 1/(n-1)/n = 1 - 1/n = O(1)$
 $1 + 1/2^2 + \dots + 1/n^2 < 1 + 1/2^2 + \dots = \pi^{2/6} = O(1)$
 $1/3 + 1/7 + 1/8 + 1/15 + 1/24 + 1/26 + 1/31 + 1/35 + \dots = O(1)$

- 可能未必收敛，但是长度有限

调和级数: $h(n) = 1 + 1/2 + 1/3 + \dots + 1/n = \Theta(\log n)$

对数级数: $\log 1 + \log 2 + \log 3 + \dots + \log n = \log(n!) = \Theta(n \log n)$

循环 vs. 级数

$$1 < \log \log n < \log n < n^\epsilon < n^c < n^{\log n} < c^n < n^n < c^{c^n}, 0 < \epsilon < 1 < c.$$

循环 vs. 级数

```

for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        O10peration(i, j);

```

算术级数：

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


```

for (int i = 0; i < n; i++)
    for (int j = 0; j < i; j++)
        O10peration(i, j);

```

算术级数：

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

Data Structures & Algorithms (Fall 2013), Tsinghua University

循环 vs. 级数

```

for (int i = 0; i < n; i++)
    for (int j = 0; j < i; j += 2013)
        O10peration(i, j);

```

算术级数：...


```

for (int i = 1; i < n; i <<= 1)
    for (int j = 0; j < i; j++)
        O10peration(i, j);

```

几何级数：

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

$$= \sum_{k=0}^{\lfloor \log_2(n-1) \rfloor} 2^k \quad (\text{let } k = \log_2 i)$$

$$= 2^{\lceil \log_2 n \rceil} - 1 = O(n)$$

Data Structures & Algorithms (Fall 2013), Tsinghua University

Back-of-The-Envelope Calculation

- 1 day $\approx 25 \times 4000 = 10^5$ sec
- 1 lifetime / century $\approx 100 \times 365 = 3 \times 10^4 \times 10^5 = 3 \times 10^9$ sec
- 三生三世 $\approx 300 \text{yr} = 10^{10}$ sec

迭代与递归

减而治之

❖ 【Decrease-and-conquer】
为求解一个大规模的问题，可以
将其划分为两个子问题：其一**平凡**，另一规模**缩减**
分别求解子问题
由子问题的解，得到原问题的解

//单调性

Data structures & Algorithms (Fall 2013), Tsinghua University

数组求和：线性递归

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

❖ 递归跟踪 (recursion trace) 分析
检查每个**递归实例**
累计所需时间 (调用语句本身，计入对应的子实例)
其总和即算法执行时间

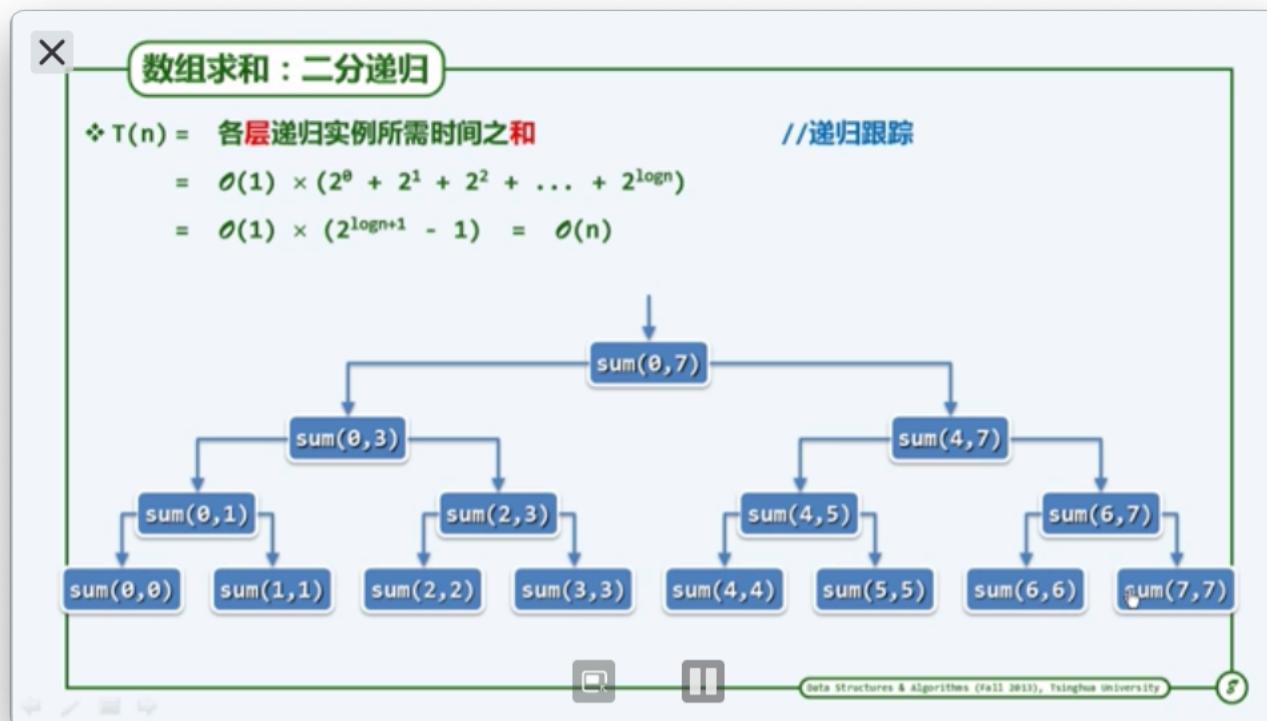
❖ 本例中，单个递归实例自身只需 $\mathcal{O}(1)$ 时间
 $T(n) = \mathcal{O}(1) * (n+1) = \mathcal{O}(n)$

Data structures & Algorithms (Fall 2013), Tsinghua University

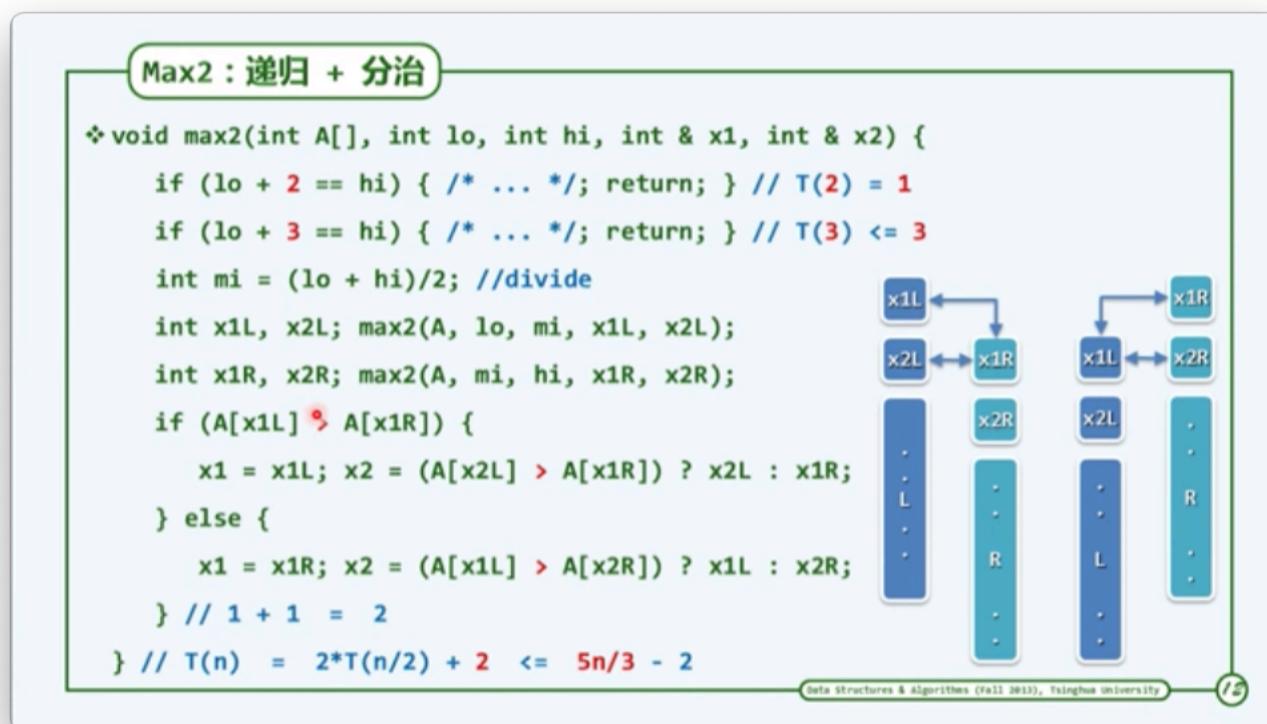
分而治之

❖ 【Divide-and-conquer】
为求解一个大规模的问题，可以
将其划分为若干（通常两个）子问题，规模大体**相当**
分别求解子问题
由子问题的解，得到原问题的解

Data structures & Algorithms (Fall 2013), Tsinghua University



☞ 几何级数可知，其 big O 和最后一个数值的级数相同，而最后一个数就是数组的个数 n



☞ 普通的求解达到 $O(2n)$ ，上述情况最坏的情况降为 $\frac{5}{3}$

1.3 Dynamic programming

fibonacci 的求和两种简便方法：

- memoization: 将已计算过示例的结果制表备查
- Dynamic programming:

```

1   f = 0; g = 1;
2   while(0<n--){
3       g = g + f;
4       f = g - f;
5   }
6   return g;

```

- 公共子序列 (subsequence): 由序列中若干字符，按原相对次序构成

- 最长公共子序列 (longest common subsequence), 可能有多个，但是长度是一定的

1.3.1 LCS: 递归

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

- (1) 若 $n = -1$ 或 $m = -1$, 则取作空序列 ("")
- (2) 若 $A[n] = 'x' = B[m]$, 则取作 $LCS(A[\theta, n], B[\theta, m]) + 'X'$ \Rightarrow decrease
- (3) 若 $A[n] \neq B[m]$, 则在两个字符中取一个稳定的，另外一个单独的字符和稳定的字符比较，相同保留，不同去掉（两种取法，每一个都对应两种取法，可以发散出很多种） \Rightarrow devide
- (4) $O(2^n)$
- (5) 使用动态规划，可以消除其中的重复计算

CHAPTER 2

vector

2.1 接口与实现

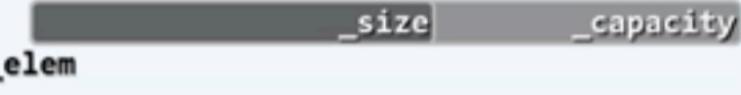
2.1.1 abstract data type

- 抽象数据类型: 数据模型 (抽象定义) + 一组操作
- 数据结构: 基于特定语言 (具体实现), 实现 ADT 的一整套算法
- 向量是数组的泛化
 - 一组元素按照线性次序封装而成 (call by rank)
 - 各元素与 rank 意义对应
 - 元素类型不限于基本类型
 - 提供 ADT 接口操作向量

2.2 可扩充向量

2.2.1 静态空间管理

静态空间管理

- ◆ 开辟内部数组 `_elem[]` 并使用一段地址连续的物理空间
- `_capacity` : 总容量
`_size` : 当前的实际规模 n
- `_elem` 
- ◆ 若采用静态空间管理策略，容量 `_capacity` 固定，则有明显的不足
 1. 上溢 (overflow) : `_elem[]` 不足以存放所有元素
尽管此时系统仍有足够的空间
 2. 下溢 (underflow) : `_elem[]` 中的元素寥寥无几
装填因子 (load factor) $\lambda = \frac{\text{_size}}{\text{_capacity}} << 50\%$
- ◆ 更糟糕的是，一般的应用环境中难以准确预测空间的需求量
- ◆ 可否使得向量可随实际需求动态调整容量，并同时保证高效率？

Navigation icons: back, forward, search, etc.

Data Structures & Algorithms (Fall 2013), Tsinghua University

2.2.2 动态空间管理

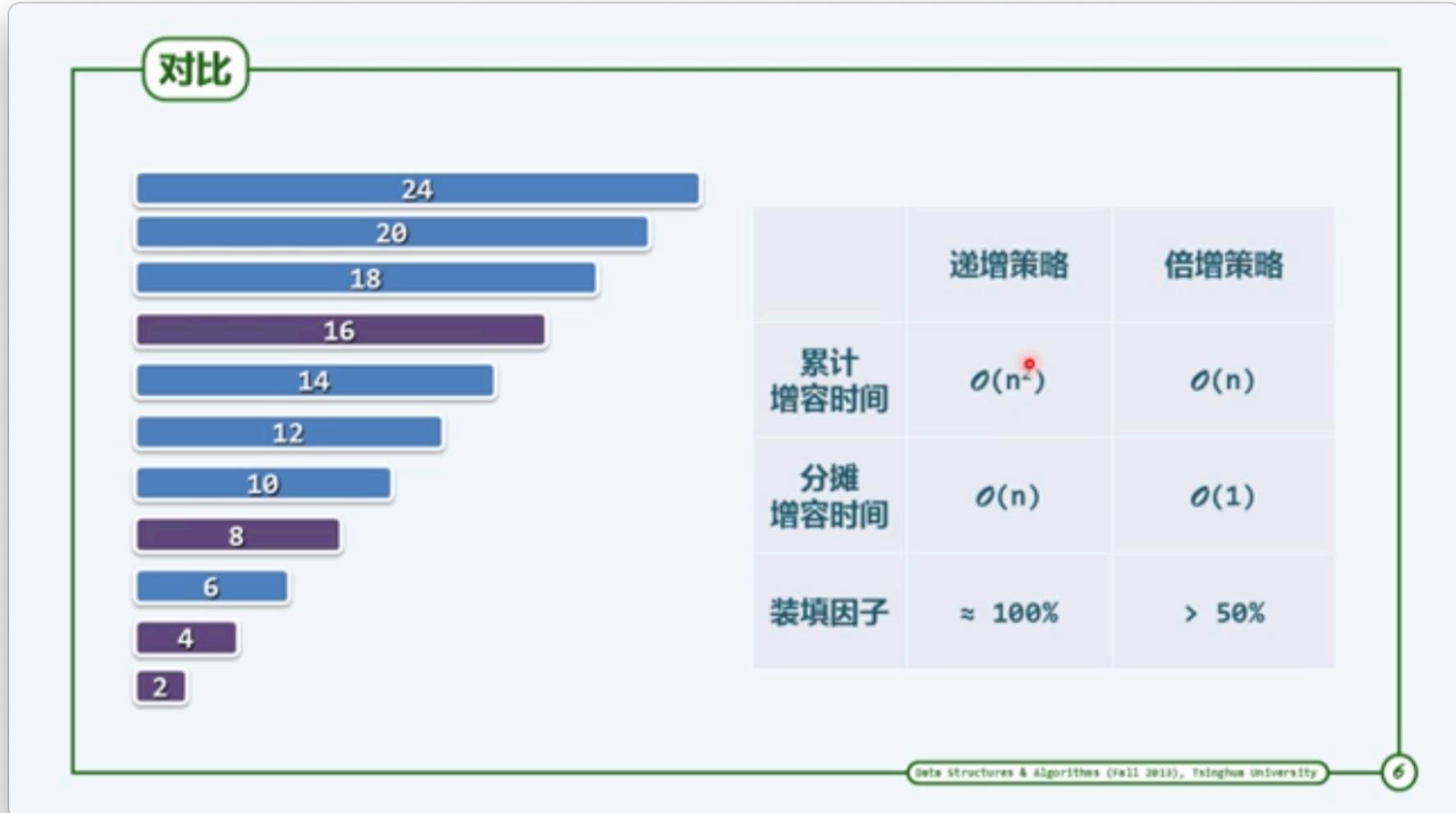
- 即将发生上溢的时候，重新开辟大的内存空间，数据 copy 过来，原本空间释放
- 问题：以前的指针可能会出现问题

递增式扩容

每加入 1 个元素就需要扩容一次，在 1、2l、3l... m*l 个元素时扩容，扩容时间 = $I * (m - 1) * m / 2 = O(n^2)$ ，每次扩容的分摊成本 $O(n)$ 。

加倍式扩容

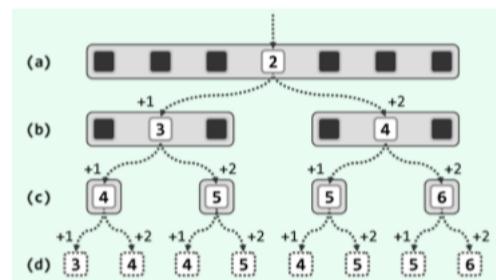
在连续插入 $1, 2, 4, 8 \dots 2^m$ 的时候扩容，总体耗时 = $O(n)$ ，每次扩容的分摊成本是 $O(1)$



2.2.3 分析方法

- 平均分析
- 分摊分析

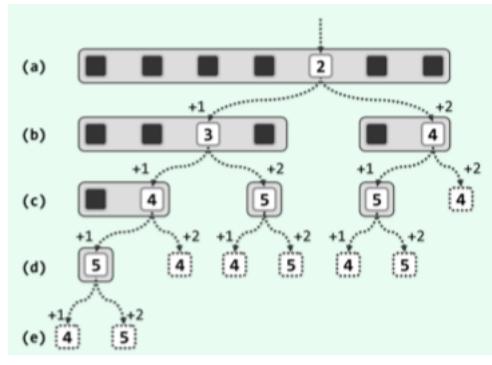
2.3 Binary Search



两次比较是因为有两次小于的比较， $e < s[mid]$, $s[mid] < e$, $O(1.5)lgn$

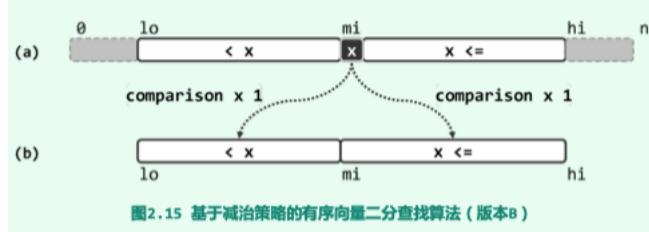
2.4 Fibonacci Search

- BS 的左侧代价小，将左侧的长度设置更大，抵消右侧的开销
- $O(1.44 \cdot \lg_2 n)$



2.5 二分查找 B 版

- 三分支到两分支， mid 被归到右侧区间，不进行二次小于号比对， $O(\log n)$



2.6 二分查找 C 版

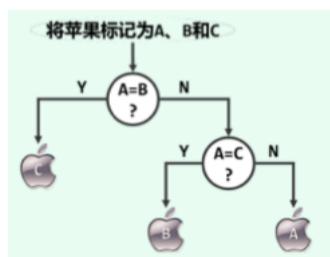
- $e < mid, hi = mi$
- $e \leq mid, hi = mi$

2.7 排序及其分类

- 内部排序
 - 处理数据规模不大，可以在内存中处理
- 外部排序
 - 数据规模大，需要分布式存储器
- offline Algorithms: 待排序数据批处理形式
- online algorithms: 待排序数据网络生成

2.7.1 比较树 (comparison tree)

基于比较式算法 (comparison-based algorithm)，简称 CBA 式算法



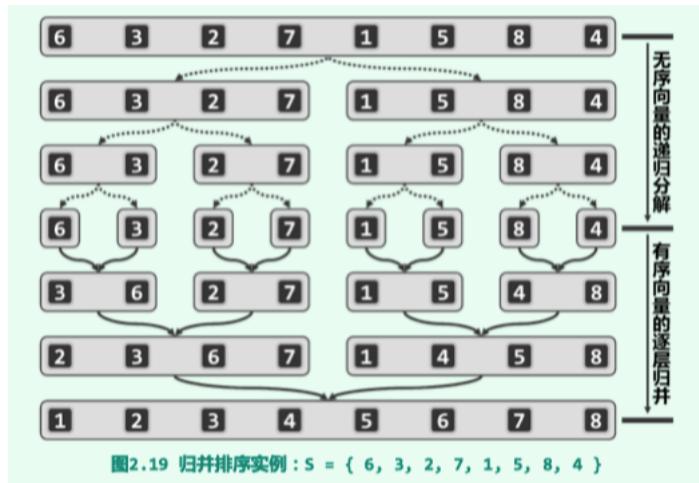
2.7.2 bubble sort

改进

记录最后右侧反转的位置，缩小右侧的反转空间；反向排序记录最左侧的反转位置，缩小左侧的空间，从而不断地缩小需要排序的序列。

2.8 mergesort

- $O(n \log n)$
- 两个有序向量合并一个有序向量：先 sort 再 merge
 - 递归可不断划分为子向量直到规模抵达递基
 - 递归返回的过程中不断归并



2.8.1 改进

判断条件的改进，对于前一个资源对比完成的对比组，后一个剩余的资源无需再更改，直接与原数组是对应的。

2.9 Bitmap

bitmap 中存储的是 bool 值，其对应的

CHAPTER 3

列表

- vector 是 call-by-rank
 - 对应逻辑和物理秩序
 - 读写操作容易，插入和移除操作难
- 列表是 call-by-position/call-by-link,
 - 因为放置的物理位置不连续，元素之间通过索引链接，仅仅对应逻辑秩序
 - 读写操作难，插入和移除操作容易

3.1 insertionsort

- 算法分为有序的前缀和无序的后缀
- 反复将后缀元素转移至前缀

3.2 selectionsort

- 无序前缀，有序后缀
- 后缀一直有序且不小于前缀

3.3 mergesort

- 复杂度 $O(n \log n)$

CHAPTER 4

栈与队列

- 线性列表
- 操作仅限于逻辑上某端
- 基本数据结构以硬件形式实现
- 结构更为简化和紧凑，可以视作向量和列表的特例
- eg: 网页浏览，文本编辑

4.1 栈 (stack)

- last-in-first-out, LIFO

4.1.1 栈与递归

- 所需空间量取决于递归深度，最深的时候达到最多的递归实例
- 递归调用和被调用函数实例关系存储在栈中
 - 调用栈
 - * 跟踪统一程序的所有函数
 - * 记录相互之间的调用关系
 - * 执行完后保证准确返回
 - * 基本单位是帧，每次函数调用创建一帧：返回地址、传入参数、局部变量
 - * 当期间发生新的调用会将新的调用压入栈顶，结束之后被中断的调用重新成为栈顶
 - 执行栈

⚠ 尽量避免递归的使用，各种参数悉数入站难以统一优化，空间利用率非常低

4.2 栈的典型应用

- 解以现行序列给出，序列逆序计算输出
- 输入输出规模不确定

4.2.1 逆序输出

4.2.2 递归嵌套

括号匹配算法

4.2.3 延迟缓冲

4.2.4 逆波兰表达式-RPN

4.3 试探回溯法

4.3.1 剪枝 pruning

根据候选解的某些局部特征，以候选解子集为单位批量地排除。

4.3.2 试探 probing

从零开始，尝试逐步增加候选解的长度。更准确地，这一过程是在成批地考查具有特定前缀的所有候选解。从长度上逐步向目标解靠近的尝试

4.3.3 回溯 backtracking

作为解的局部特征，特征前缀在试探的过程中一旦被发现与目标解不合，则收缩到此前一步的长度，然后继续试探下一可能的组合。

4.4 队列 queue

对象只能从一端进入 (enqueue)，另一端删除 (dequeue)，first-in-first-out, FIFO

CHAPTER 5

二叉树

semi-linear structure

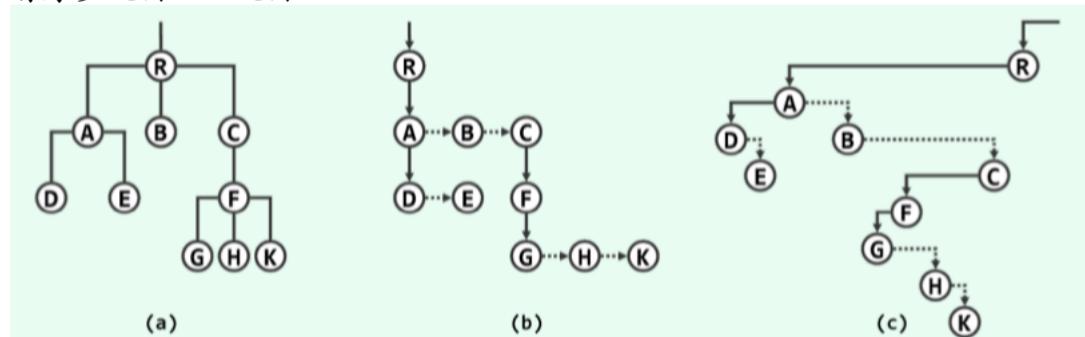
- 连通无环图：顶点 + 边，有根 (rooted tree)
- depth, 深度: 节点到根节点的边数
- ancestor, descendant, proper ancestor(除自身外), proper descendant
- parent, child
- degree(v 的孩子总数), leaf(叶节点, 无孩子的节点)
- internal node(非 root, leaf 节点)
- subtree
- height(深度最大值)

5.1 二叉树 binary tree

5.2 多叉树 k-ary tree

每个节点的孩子均不超过 k 个的有根树

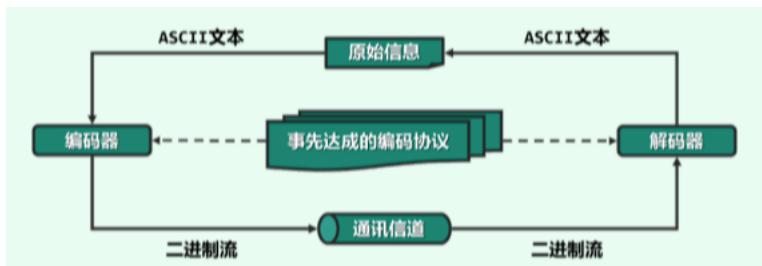
有序多叉树 = 二叉树



5.3 编码树

5.3.1 二进制编码

- 编码: 信息被转换为二进制形式
- 解码: 二进制编码恢复原始信息
- 前缀无歧义编码 (prefix-free code), PFC 编码

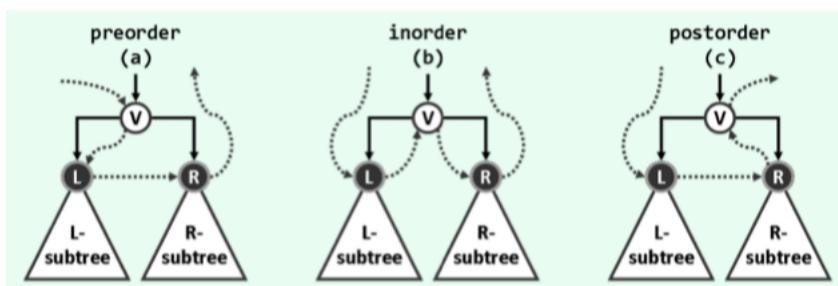


5.3.2 二叉编码树

PFC, 所有的字符对应于叶子节点

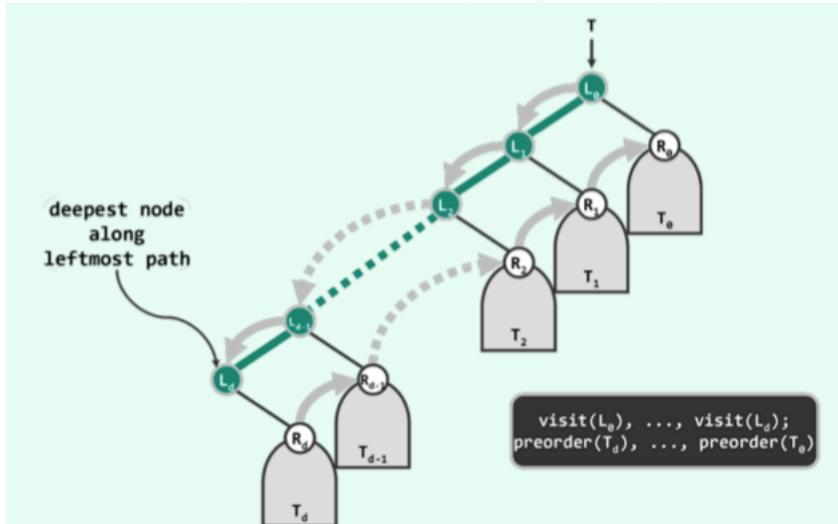
5.4 二叉树实现

5.4.1 遍历



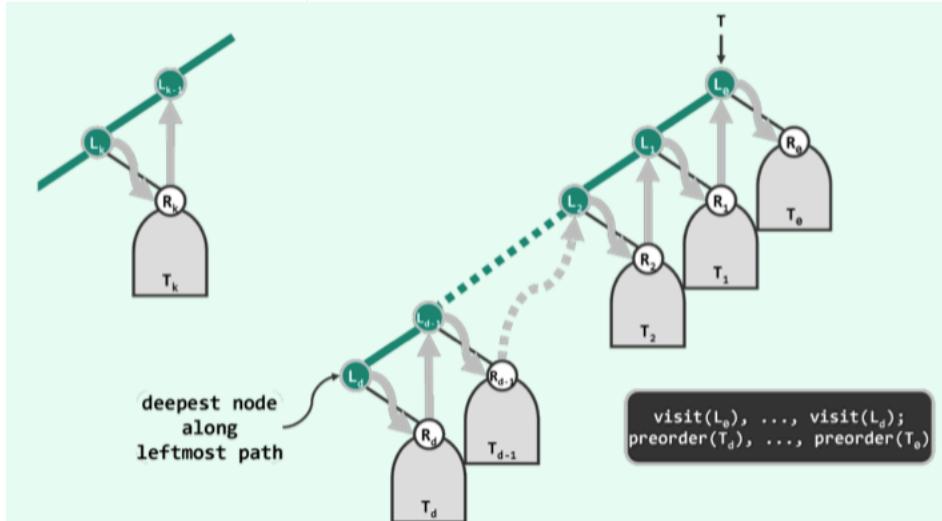
5.4.2 迭代版先序遍历

自顶而下访问左节点，每一个左节点都是当前节点的一部分，对应的右节点入栈，在自底而上访问右节点



5.4.3 迭代版中序遍历

自顶而下沿着左节点深入，但是并不访问对应节点，而是直接入栈，直到无左孩子，再弹出入栈节点，访问该节点并转向右子树，重复左节点的方法，遍历完成右子树



此方法的缺陷是，最坏的情况可能出现所有节点入栈，极大的空间消耗

5.4.4 改进迭代版中序遍历

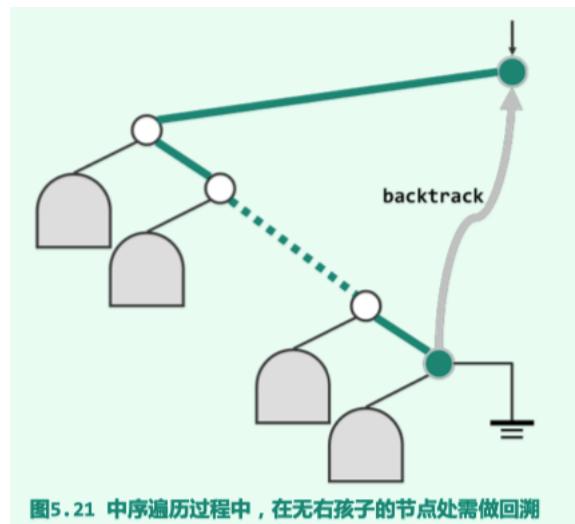


图5.21 中序遍历过程中，在无右孩子的节点处需做回溯

之前的节点都实施了入栈的行为，耗费空间，可以直接深入左节点直至最后一个，此时无左子树直接访问该节点，再判断有无右子树，如果有右子树，则需要重复之前的步骤进行遍历同时关闭回溯标志；如果右子树为空，则回溯同时设置回溯标志位为真，此时相当于直接访问回溯的

5.4.5 迭代版后序遍历

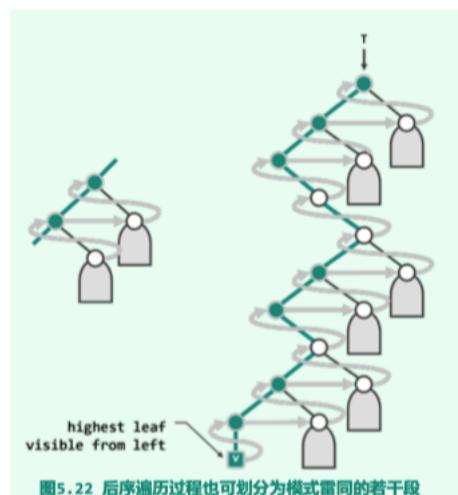


图5.22 后序遍历过程也可划分为模式相同的若干段

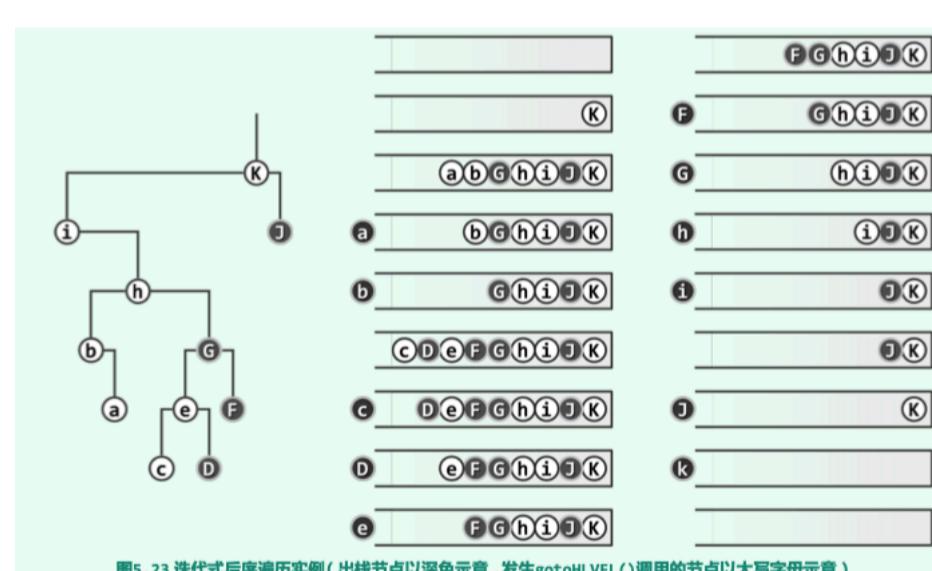


图5.23 迭代式后序遍历实例(出栈节点以深色示意 ,发生gotoHLVFL()调用的节点以大写字母示意)

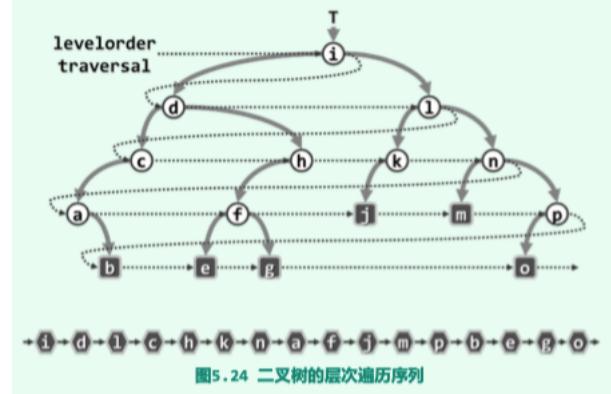
- 入根节点(栈顶非根节点之父)

- 左侧有子节点
 - * 右侧有子节点，先入右侧再入左侧
 - * 右侧无子节点，直接入左侧
- 左侧无，直接入右侧
- 循环条件是不断赋值栈顶，只有当没有节点可深入才结束(最后结束是入左侧，那么左侧是重点，无左侧则右侧是重点)

- 弹出终端节点进行访问，当栈顶又是非空右侧节点时，重复类似根节点的工作

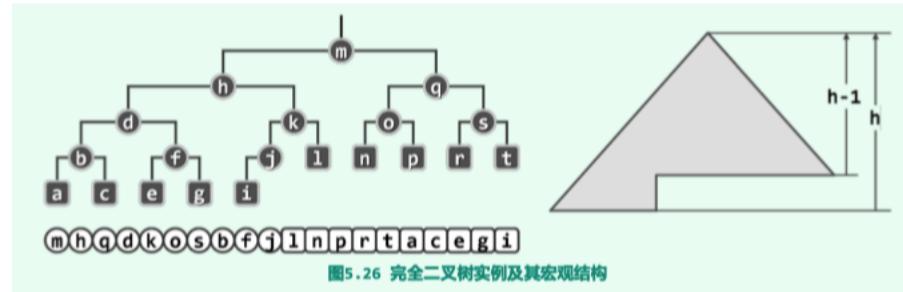
5.4.6 层次遍历

广度优先遍历/层次遍历 (level-order traversal): 先上后下、先左后右



- 访问父节点
- 子节点(左右)入队列，先访问左子节点，左子节点的子节点入队列，以此类推，直至到达底部返回

完全二叉树



次底层之上所有父节点都有完全子节点，最底层叶节点都处于次底层叶节点的左侧，规模 $2^h \sim 2^{h+1} - 1$

满二叉树

最底层的节点全满，完全二叉树的特例，有 $2^{h+1} - 1$ 个节点

5.5 huffman 编码

5.5.1 PFC 编码及解码

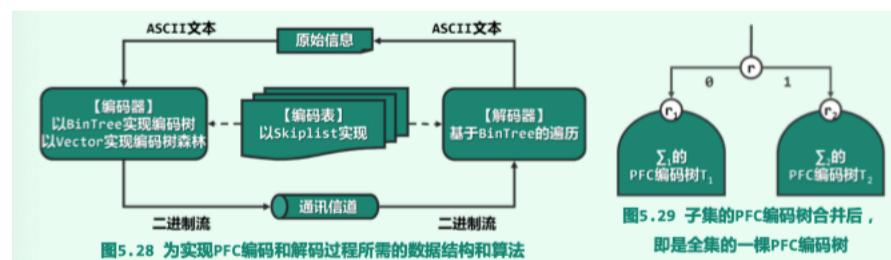


图5.28 为实现PFC编码和解码过程所需的数据结构和算法

图5.29 子集的PFC编码树合并后，即是全集的一棵PFC编码树

5.5.2 最优编码树 optimal encoding tree

$$\text{ald}(T) = \sum_{x \in \Sigma} |\text{rps}(x)| / |\Sigma| = \sum_{x \in \Sigma} \text{depth}(x) / |\Sigma|$$

- 双子性：真二叉树，左右孩子双全
- 层次性：深度之差不得超过 1
- $2|\Sigma| - 1$ 的完全二叉树 T ，再将 Σ 中的字符任意分配给 T 的 $|\Sigma|$ 个叶节点

5.5.3 Huffman 编码树

- 字符出现有概率
- 带权平均编码长度与叶节点带权平均深度 (weighted average leaf depth, WALD): $\text{wald}(T) = \sum_{x \in \Sigma} p(x) \cdot |\text{rps}(x)|$
- 完全二叉树不能保证 WALD 最短
- 最优带权编码树
 - 双子性: 概率最低的是最底层的兄弟节点
 - 层次性:

算法

对于 Σ 个已知字符，分别建立 Σ 棵树，每棵树对应的权重为该字符出现的频率，形成一个森林。从森林中取出权重最小的两棵树作为左、右子树，创建一个新的节点，权重为两棵子树的权重之和。反复迭代，每轮减少一棵树，最终形成一整棵高树，即 huffman 编码树。总体运行时间: $O(n) + o(n - 1) + \dots + o(2) = o(n^2)$

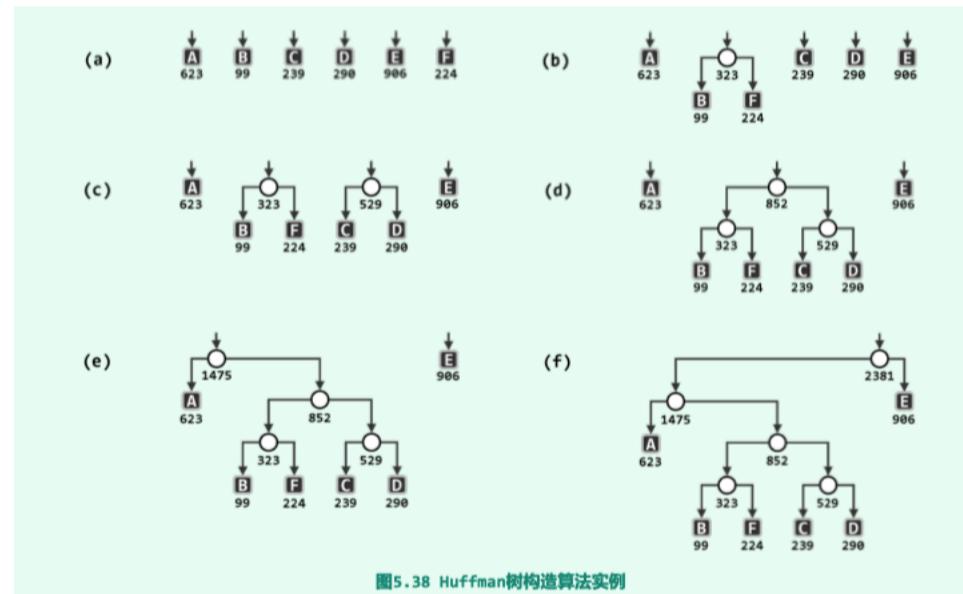


图5.38 Huffman树构造算法实例

CHAPTER 6

图

non-linear structure

通过遍历转化成 semi-linear structure, 再通过树结构转化成线性结构

- graph: $G = (V, E)$
 - undirected graph/undigraph
 - directed graph/digraph
 - mixed graph
 - simple graph: 不含任何自环的图
 - directed acyclic graph, DAG
 - weighted graph/ network
- vertex: 顶点, 规模 $n = |V|$
- edge: $E, e = |E|$
 - undirected edge: (u, v) 无次序
 - directed edge: (u, v) 和 (v, u) 不对等
- degree: 与顶点相连的边数
 - in-degree: incoming edge
 - out-degree: outgoing edge
- simple path: 沿途顶点互异
- Eulerian tour: 欧拉环路, 经过途中各边一次且恰好一次的环路

6.1 邻接矩阵 adjacency matrix

- 访问时间 $O(1)$
- 顶点动态操作接口耗时
- 空间冗余大, vector 装填因子不低于 50%, 单次操作耗时不过 $O(n^2)$

6.2 邻接表 adjacency list

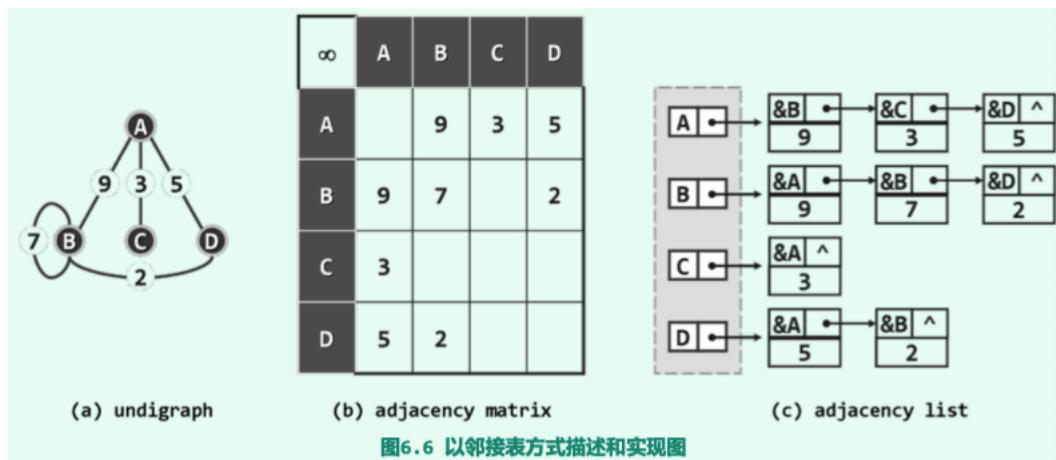


图6.6 以邻接表方式描述和实现图

- 空间总量 $O(n + e)$
- 时间复杂度，反问单条边的效率不高，但是擅长批量处理方式
- 总体效率比邻接矩阵要高

6.3 图算法

graph search: 对处于特定状态顶点的甄别与查找

6.3.1 广度优先搜索 breath-first search, BFS

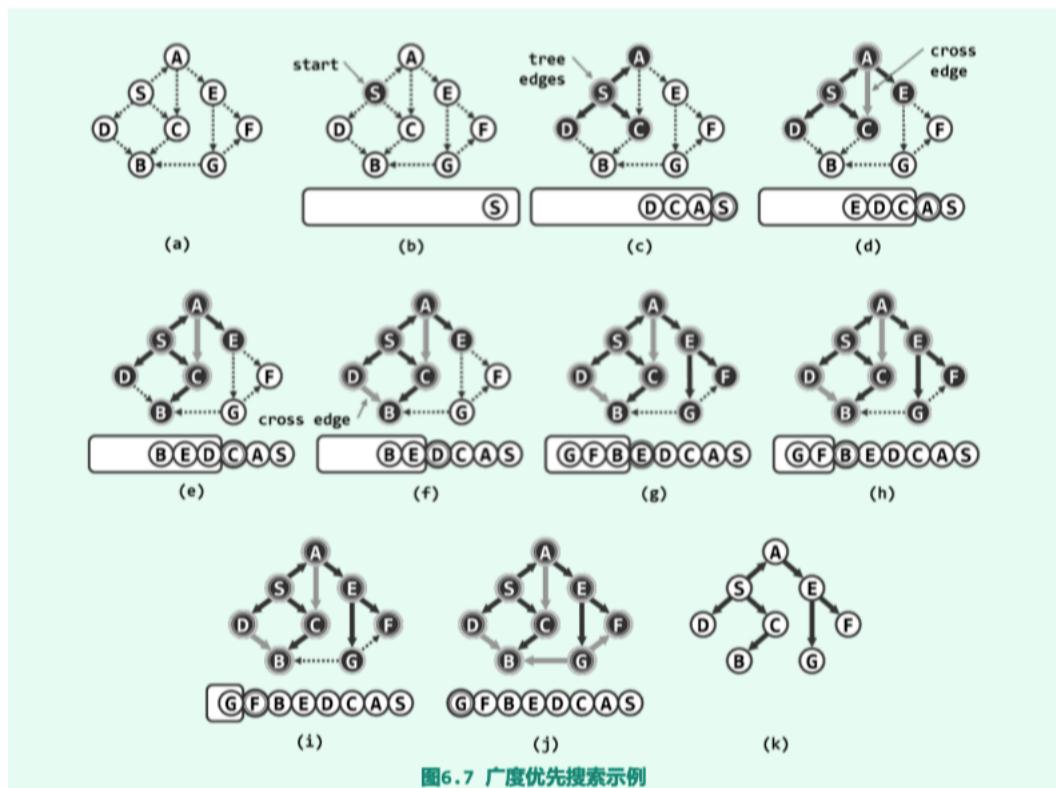


图6.7 广度优先搜索示例

- 层次遍历
- frontier, 前沿集中的所有顶点的深度相差不超过 1
- 覆盖无向图 S 所属的连通分量
- 覆盖有向图 S 为起点的可达分量
- 总体时间复杂度 $O(n + e)$

6.3.2 深度优先搜索 depth-first search, DFS

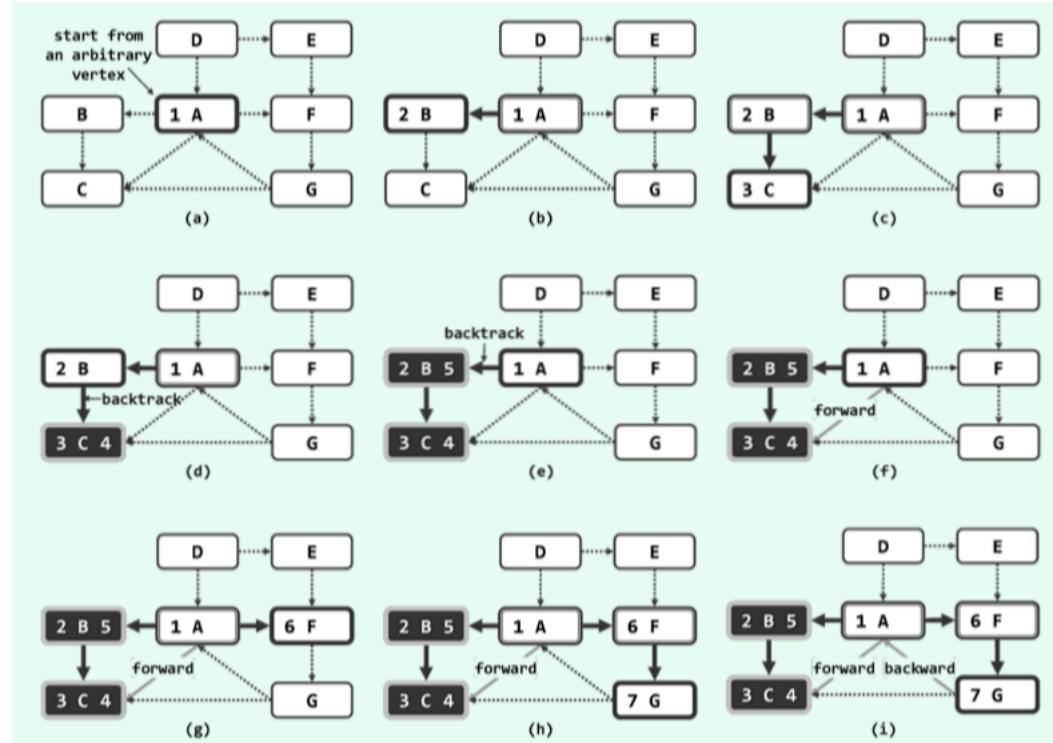


图6.8 深度优先搜索实例 (粗边框白色, 为当前顶点; 细边框白色、双边框白色和黑色, 分别为处于 UNDISCOVERED、DISCOVERED和VISITED状态的顶点; dTime和fTime标签, 分别标注于各顶点的左右)

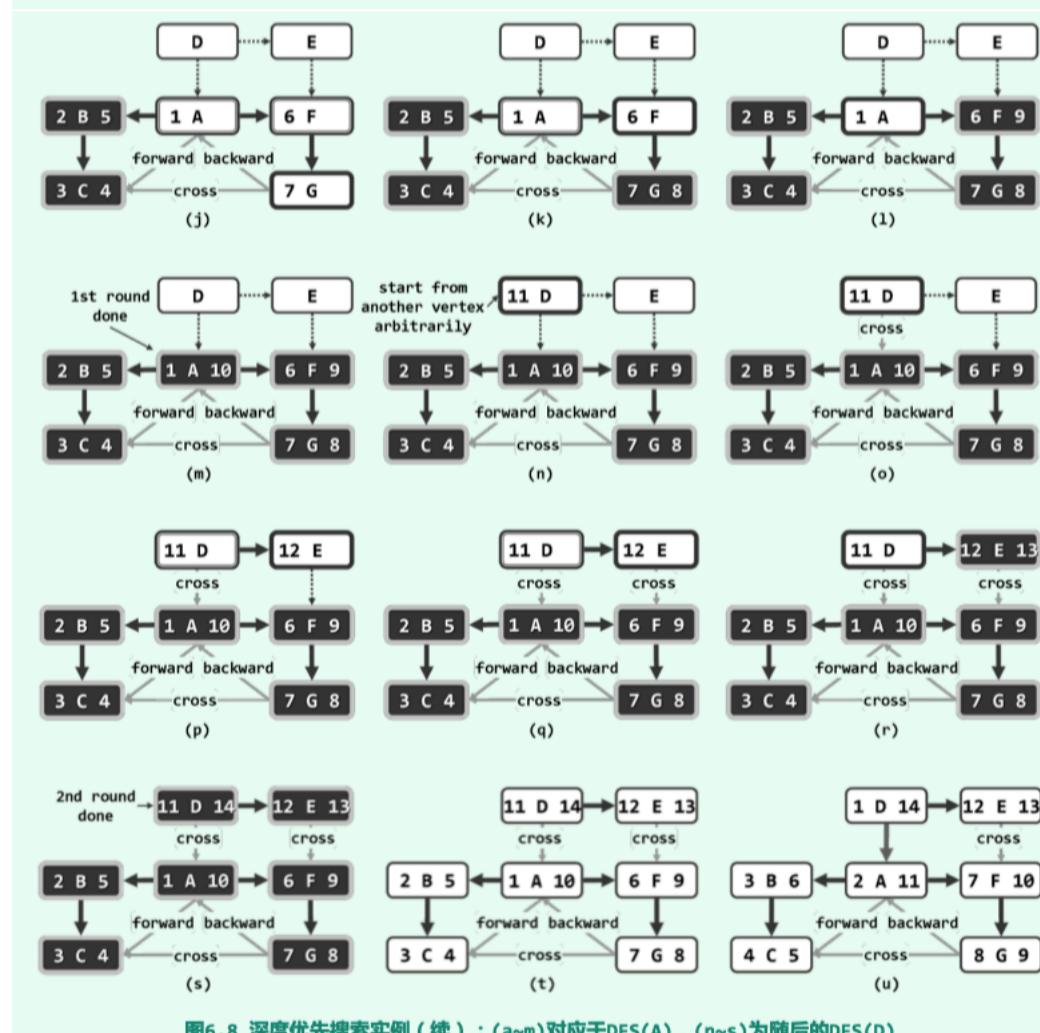
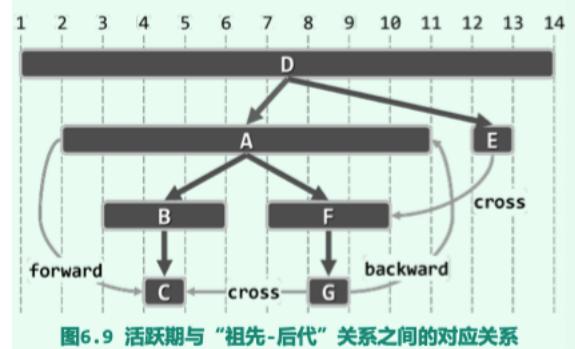


图6.8 深度优先搜索实例 (续) : (a-m)对应于DFS(A), (n-s)为随后的DFS(D)

有两棵DFS树，起点不同，形成的DFS树也会不同



- 时间复杂度 $O(n + e)$
- 空间复杂度，需要耗费一定量空间

6.4 拓扑排序 topological sorting

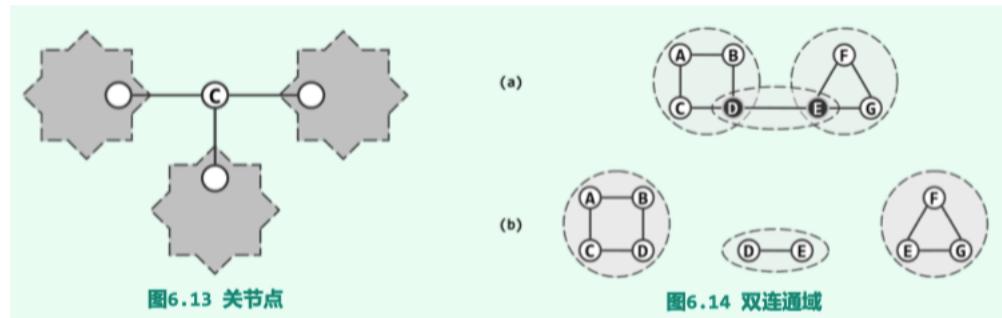
每一顶点都不会通过边，指向其在此序列中的前驱顶点。

6.4.1 有向无环图

- 必然有拓扑排序
- 必有入度为 0 的顶点(起点)，极大顶点
- 必有出度为 0 的顶点(终点)，极小顶点

6.5 双连通域分解

- 切割节点/关节点：删除顶点 v 后 G 所包含的连通域增多
- 双连通图：不含关节点的图



6.5.1 蛮力算法

- BFS 或者 DFS 找到连通域的数目
- 枚举删除顶点 v ，重新计算连通域
- 连通域变化则是关节点，否则不是

6.5.2 可行算法

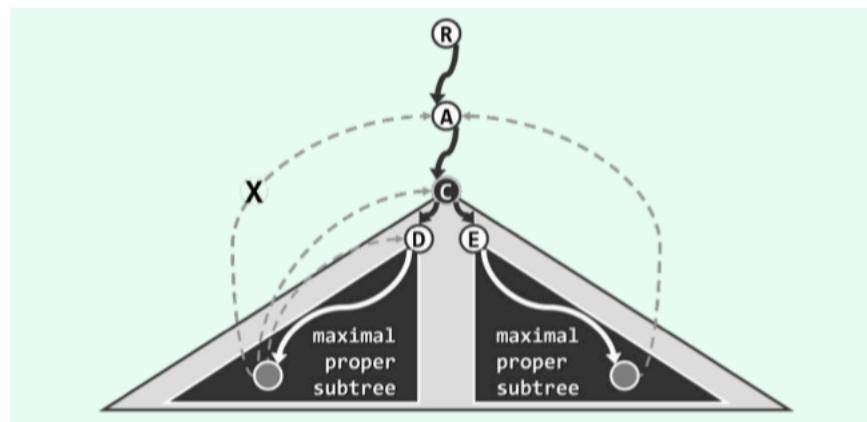


图6.16 内部节点C是关节点，当且仅当C的某棵极大真子树不(经后向边)联接到C的真祖先

- 无向图 DFS 树中，在搜索过程中，更新各顶点所能连同的最高祖先 (highest connected ancestor, HCA)，即可认定关节点
- $hca[u] \geq dTime[v]$, 说明 u 及其后代无法通过后向边与 v 的真祖先连同，故 v 为关节点。
- $hca[u] < dTime[v]$, 说明 u 及其后代可通过后向边与 v 的真祖先连同，故 v 不为关节点。

6.6 优先级搜索

priority-first search, PFS/ best-first search, BFS

以迭代的方式逐步引入顶点和边，最终构造出一颗遍历树，每次都引入当前优先级最高的顶点s，按照不同的策略更新其邻接顶点的优先级数。

6.7 最小支撑树

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



所有 spanning tree 中，成本最低者(边权重之和)为最小支撑树(minimum spanning tree, MST)，不一定唯一

6.7.1 蛮力算法

由n个互异顶点组成的完全图共有 n^{n-2} 棵支撑树，更新成本记录需要 $O(n^{n-2})$

6.7.2 Prim 算法

图 $G=(V; E)$ ，顶点集 V 的任一非平凡自己 U 及其补集 $V \setminus U$ ，构成图 G 的一个割 cut，其中两个子集之间相连接的边为 crossing edge(跨越边/桥)。

贪心迭代

阴影中的子图内的顶点始终至少与集合内至少一点相连，其余过大的边就删除掉，逐步扩充子图的顶点个数，不停迭代。

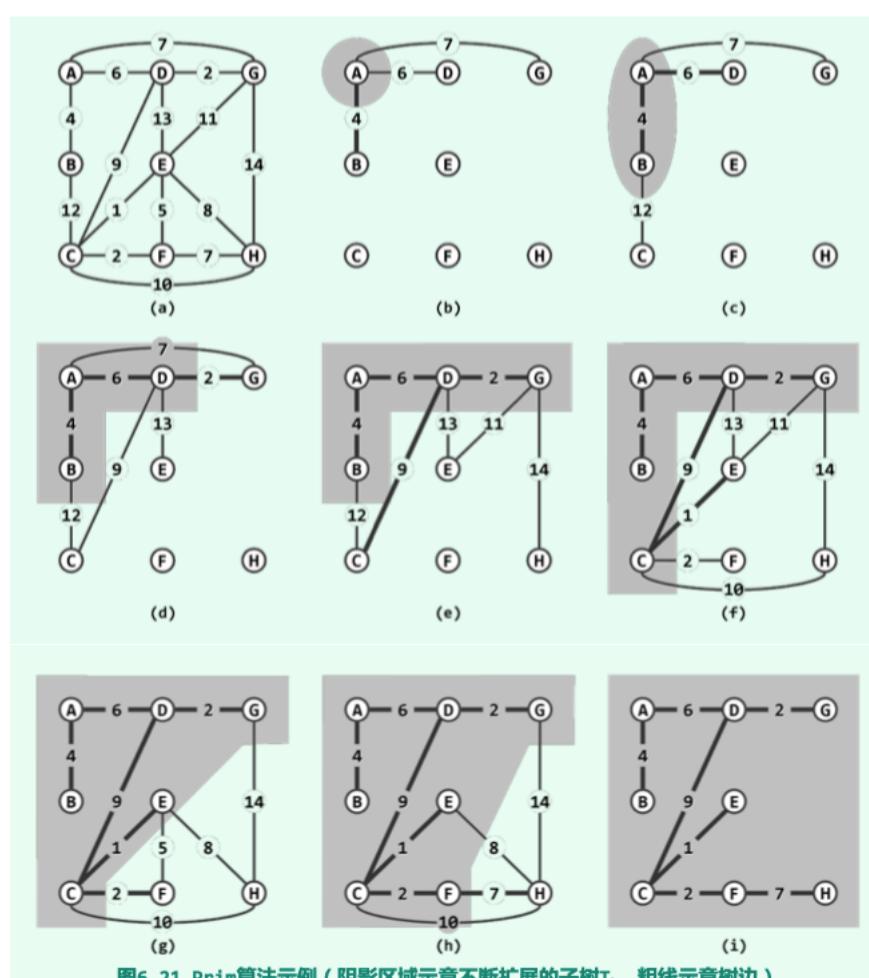


图6.21 Prim算法示例 (阴影区域示意不断扩展的子树 T_k ，粗线示意树边)

6.8 最短路径

最短路径的任一前缀也是最短路径

6.8.1 Dijkstra 算法

与 prim 相比，Dijkstra 算法关注的是每一个顶点到 s 距离最近者，而前者关注的是到 T_k 这个整体的最近者。

CHAPTER 7

搜索树

7.1 二叉搜索树

7.1.1 循关键码查询 call-by-key

查找算法及其实现

时间复杂度: $O(n)$ 最坏(当数字是顺序排列, 就成了列表形式, 查找最大值就是在最长的部分进行搜索)

插入算法

在 search 算法的基础上进行, 插入之后需要更新对应的祖先的高度, 时间复杂度取决于 search() 和 updateHeightAbove() 的

删除算法

单分支, 直接替换即可; 双分支需要找到合适的节点

7.2 平衡二叉搜索树

7.2.1 随机生成 vs 随机组成

- **随机生成:** 将 n 个值进行随机排列, 因为顺序一般都会被打乱, 那么最终生成的二叉树的深度不太可能是最坏的情况, 平均高度是 $\Theta(\log n)$
- **随机组成:** 遵循一定顺序性的前提下, 随机确定他们之间的拓扑连接, 平均查找长度为 $\Theta(\sqrt{n})$
- 随机生成的高度不同于随机组成的高度, 是因为随机生成的序列有很多序列不同、结构一样的二叉树被重复统计

7.2.2 理想平衡 vs 适度平衡

- **理想平衡:** 高度恰好为 $\lfloor \log_2 n \rfloor$
- **适度平衡:** 树高渐进地不超过 $O(\log n)$

7.2.3 等价交换

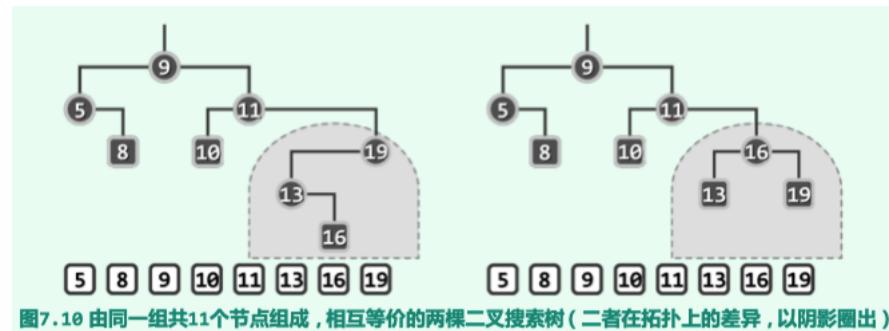


图7.10 由同一组共11个节点组成，相互等价的两棵二叉搜索树（二者在拓扑上的差异，以阴影圈出）

调整手段

- zig

- zag

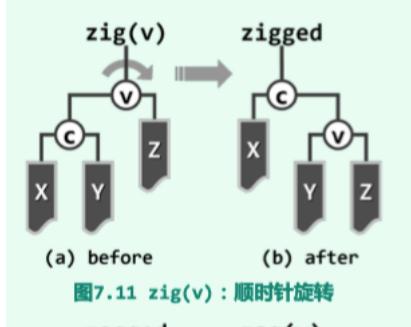


图7.11 zig(v)：顺时针旋转

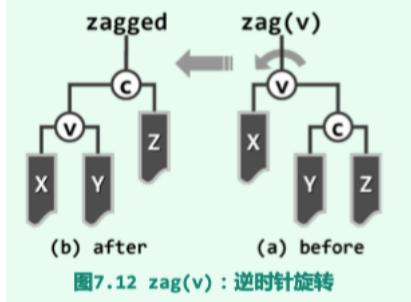


图7.12 zag(v)：逆时针旋转

7.3 AVL树

- 平衡因子: $\text{balFac}(v) = \text{height}(\text{lc}(v)) - \text{height}(\text{rc}(v))$
- 失衡: 因节点的插入或删除而暂时失衡的节点构成失衡节点集, $UT(x)$
- 重失衡:

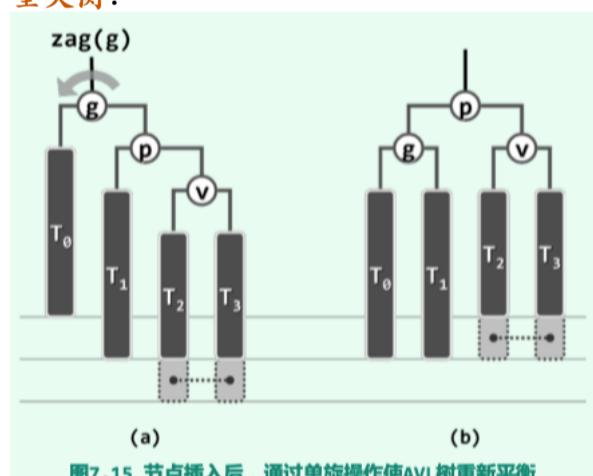


图7.15 节点插入后，通过单旋操作使AVL树重新平衡

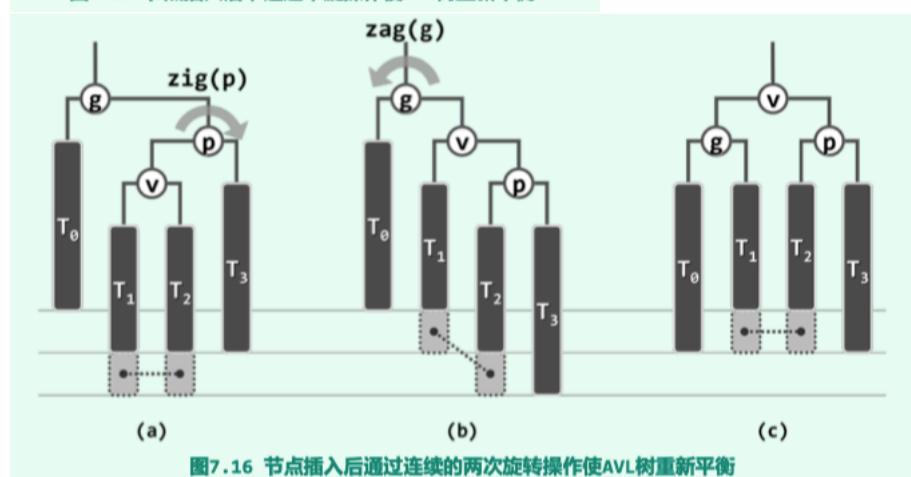
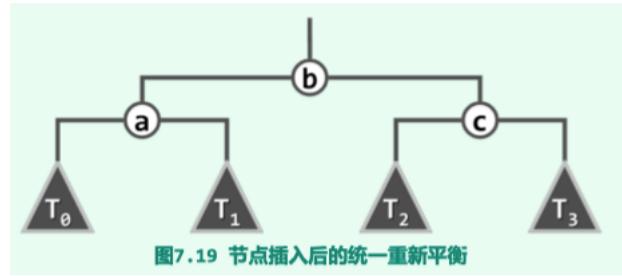


图7.16 节点插入后通过连续的两次旋转操作使AVL树重新平衡

7.3.1 统一重平衡算法

上诉的重构涉及的较为复杂，可以根据“3+4”进行重构，失衡节点和对应的子树直接重新组装而不是旋转



CHAPTER 8

高级搜索树

8.1 伸展树 splay tree

- 数据局部性 (data locality)
 - 刚被访问的元素，可能不久之后再次被访问到
 - 将被访问的下一元素，可能处于不久之前就被访问过的某个元素的附近

8.1.1 逐层伸展

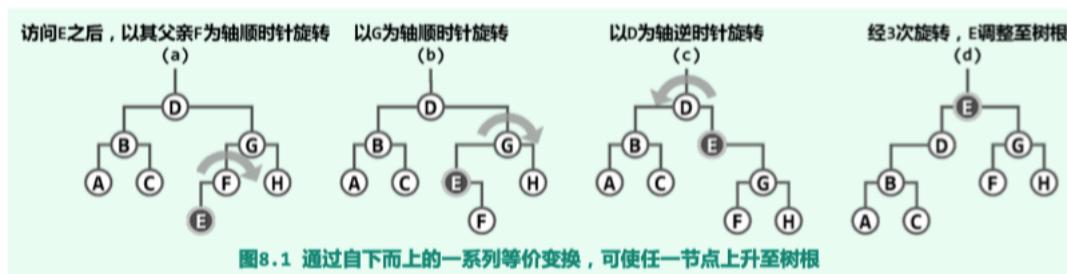


图8.1 通过自下而上的一系列等价变换，可使任一节点上升至树根

- 平均访问时间 $\Omega(n)$

8.1.2 双层伸展

- zig-zig/zag-zag

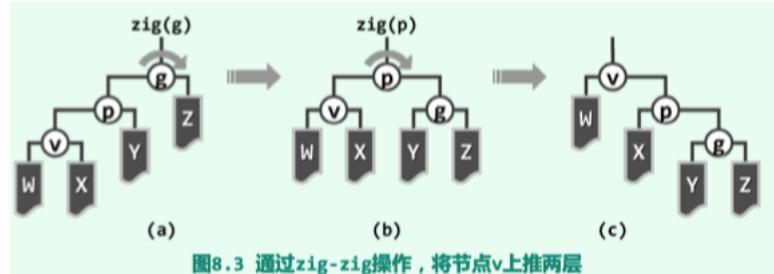


图8.3 通过zig-zig操作，将节点v上推两层

- zig-zag/zag-zig

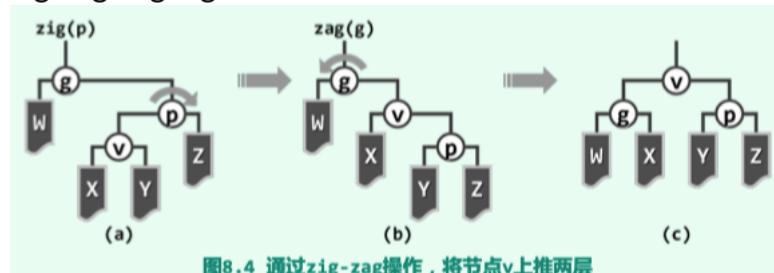


图8.4 通过zig-zag操作，将节点v上推两层

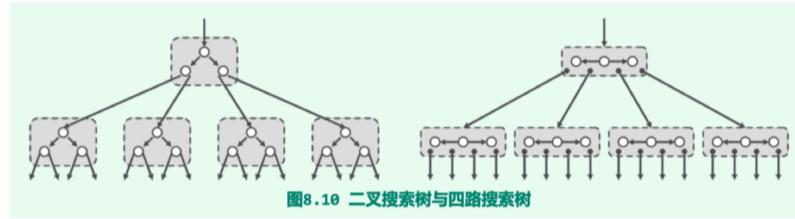
- 双层伸展和单旋相结合可以完成节点的移动
- 时间复杂度 $O(\log n)$

8.2 B-树

8.2.1 分级存储

- 内存高速度
- 外存大容量
- 内、外之间的数据传输: I/O 操作

8.2.2 四路搜索树



- 将二叉树中的两层转化为四路“大节点”
- k 层为间隔进行重组，可将二叉搜索树转化为 2^k 路搜索树，多路搜索树 multi-way search tree

8.2.3 多路平衡搜索树

m 阶 B-树 (B-tree)

关键码查找

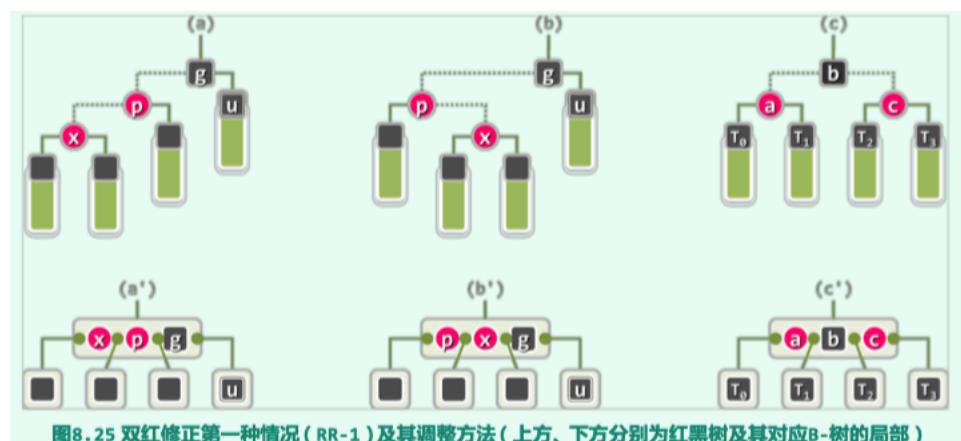
- B-树适宜在相对更小的内存中，实现对大规模数据的高效操作
-

上溢与分裂

- 关键码的删除会引发节点的上溢
- 上溢节点提升为父节点，如果发生上溢上传，就一直传递，当传至根节点时，可以自成一个新的树根

下溢和合并

关键码的插入会引发节点的下溢



考查如图8.20(a)所示的3阶B-树。

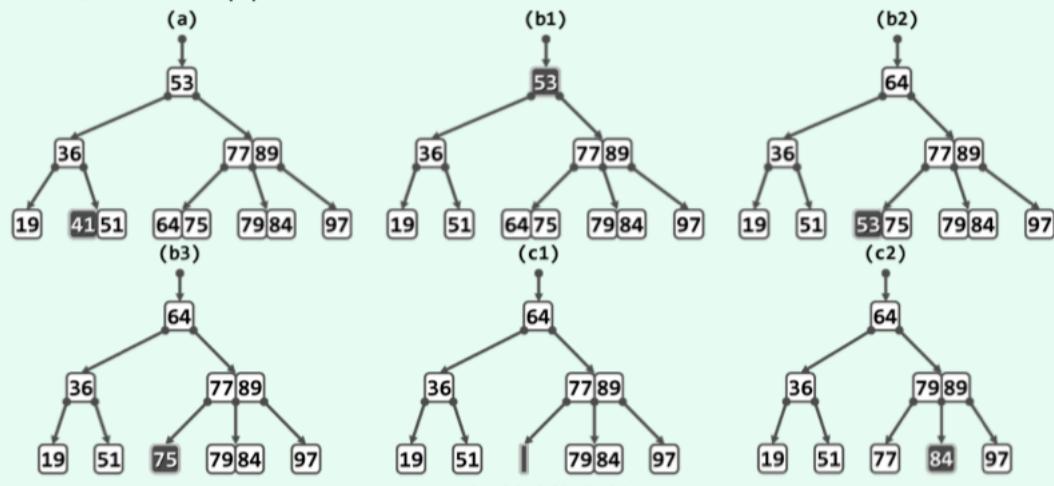


图8.20 3阶B-树删除操作实例 (I)

首先执行remove(41): 因关键码41来自底层叶节点, 且从中删除该关键码后未发生下溢, 故无需修复, 结果如图(b1)所示。接下来执行remove(53): 因关键码53并非来自底层叶节点, 故在将该关键码与其直接后继64交换位置之后, 如图(b2)所示关键码, 53必属于某底层叶节点; 在删除该关键码之后, 其所属节点并未发生下溢, 故亦无需修复, 结果如图(b3)所示。

然后执行remove(75): 关键码75来自底层叶节点, 故被直接删除后其所属节点如图(c1)所示发生下溢; 在经父节点中转, 从右侧兄弟间接借得一个关键码之后, 结果如图(c2)所示。

8.3 红黑树

- 树根始终为黑色
- 外部节点均为黑色
- 其余节点若为红色, 则其孩子节点必为黑色
- 从任一外部节点到根节点的沿途, 黑节点的数目相等

红黑树到4阶B树的等效转换

- 节点中黑码仅包含1个



平衡性

因为红黑树中任一通路都不包含相邻的红节点, 那么搜索树的高度不会超过黑高度的两倍, 虽然无法做到理想平衡, 到那会其高度依然可控制在 $O(\log n)$ 。此中的树高指的实际是红黑树的黑高度

查找算法

查找节点, 如果目标节点不存在, 则在查找终止的位置创建节点, 并随即将其染成红色 (除非全树仅此一个节点才染黑)

插入算法

引入红节点之后，可能出现双红现象

- 双红修正 RR-1(另外有两种对称情况), u 为黑色, 出现双红的时候, x 的孩子兄弟和 u 的黑高度均相等, 需要修正

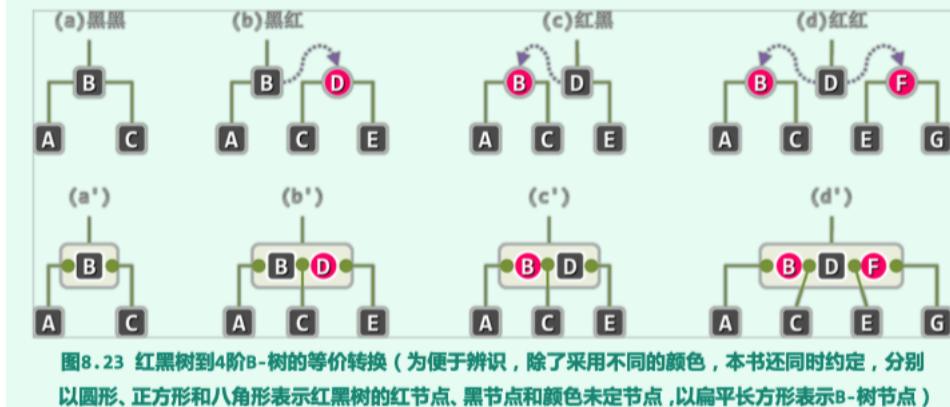
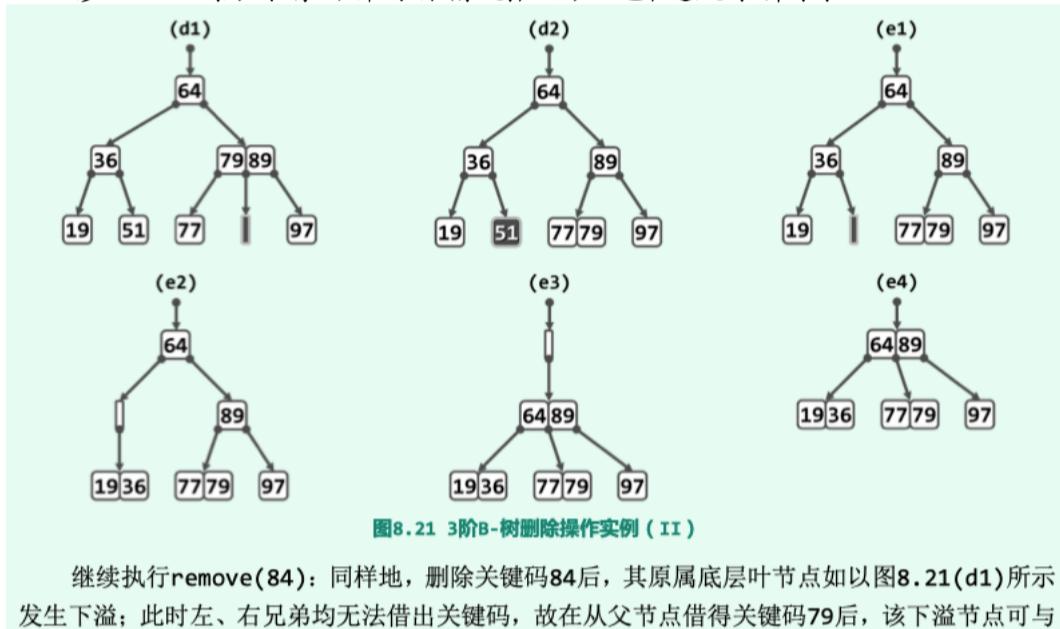


图8.23 红黑树到4阶B-树的等价转换 (为便于辨识, 除了采用不同的颜色, 本书还同时约定, 分别以圆形、正方形和八角形表示红黑树的红节点、黑节点和颜色未定节点, 以扁平长方形表示B-树节点)

- 双红修正 RR-1(另外有两种对称情况), u 为红色, 违反条件 (3)



继续执行remove(84): 同样地, 删除关键码84后, 其原属底层叶节点如以图8.21(d1)所示发生下溢; 此时左、右兄弟均无法借出关键码, 故在从父节点借得关键码79后, 该下溢节点可与其左侧兄弟合并; 父节点借出一个关键码之后尚未下溢, 故结果如图(d2)所示。

最后执行remove(51): 删除关键码51后, 其原属底层叶节点如图(e1)所示发生下溢; 从父节点借得关键码36后, 该节点可与左侧兄弟合并, 但父节点如图(e2)所示因此发生下溢; 从祖父(根)节点借得关键码64后, 父节点可与其右侧兄弟合并, 但祖父节点如图(e3)所示因此发生下溢。此时已抵达树根, 故直接删除空的根节点, 如图(e4)所示全树高度降低一层。

修正的复杂度:

节点删除算法

- ...

8.4 kd-树

8.4.1 范围搜索

- 1D: 平衡二叉树
- 2D 多维: 递归定义的平衡二叉搜索树 – kd 树

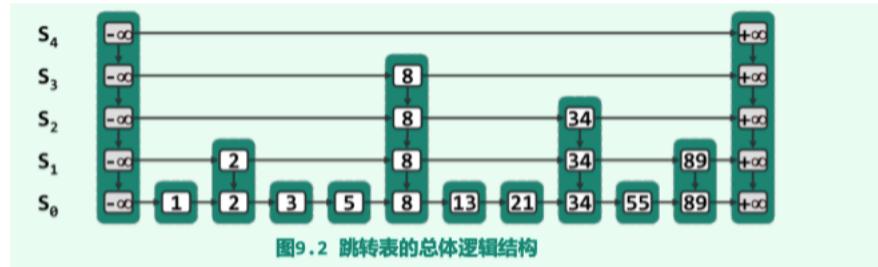
CHAPTER 9

词典

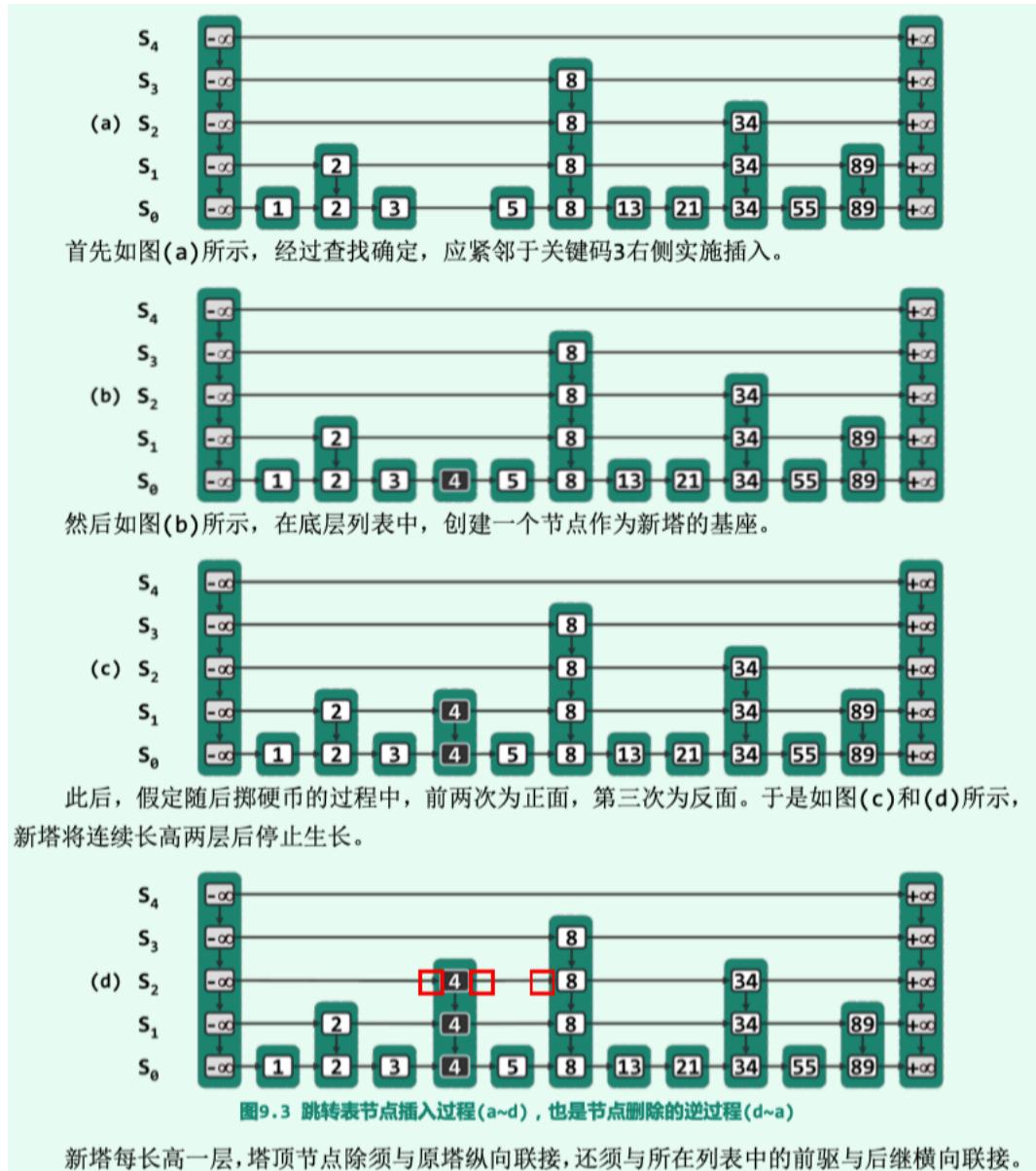
dictionary 和 map 的区别是 dict 运行多个词条拥有相同的关键码，而 map 则不允许拥有相同的关键码

9.1 skip list 跳转表

9.1.1 四联表 quadlist



- 纵向的生长，依据生长概率逐层减半的原则
- 第 k 层列表所含节点的期望数目: $E(|S_k|) = n \times 2^{-k}$
- 空间总体消耗量的期望值: $E(\sum_k |S_k|) = \sum_k E(|S_k|) = n \times (\sum_k 2^{-k}) < 2n = O(n)$
- 时间复杂度:



9.2 散列表 hashtable

- 可存放词条或引用的单元组成 (bucket)，线性结构，通常由向量来实现
- bucket array
- hash function: 词条与桶地址之间约定某种映射关系， $\text{hash}(): \text{key} \rightarrow \text{hash}(\text{key}): \text{hashing address}$

9.2.1 hash function

division method

mod，除数必须是素数，降低冲突的风险

MAD multiply-add-divide method

$(a \times \text{key} + b) \bmod M$, $a > 0, b > 0, M$ 为素数

- 数字分析法
- 平方取中法
- folding
- xor
- 分割后各区段的方向往复折返式
- 伪随机数发
- ...

9.2.2 冲突及其排解

multiple slots

~	~	~	~	~	~	~	~	~	~	~	~	~	~	~	~	~	~	~	~
1120	~	~	~	~	~	~	~	~	~	~	~	~	~	~	~	1135	~	~	~
1060	~	~	~	~	1105	~	~	~	~	~	1090	~	~	~	~	1075	~	~	~
1000	~	~	~	~	1045	~	~	~	~	~	1030	~	~	~	~	1015	~	~	~
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

图9.10 通过槽位细分排解散列冲突

- 大部分 slots 都处于空闲状态， k 个槽位的散列表可以将装填因子降至之前的 $\frac{1}{k}$

separate chaining

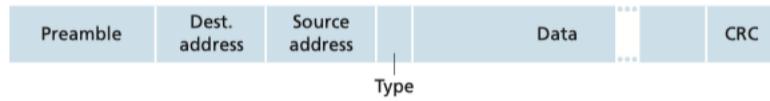


Figure 6.20 ◆ Ethernet frame structure

- 子字典通过词典实现，而不是采用列表（向量）
- 有效降低空间消耗，查找过程中发生冲突，需要遍历整个列表，导致查找成本的增加

公共溢出区 overflow area

- 一旦插入词条发生冲突就转入公共缓冲池
- 独立链等策略便捷而紧凑，但绝非上策。比如，因需要引入次结构，实现相关算法的代码自身的复杂程度和出错概率都将大大增加。反过来，因不能保证物理上的关联性，对于稍大规模的词条集，查找过程中将需做更多的 I/O 操作。

闭散列策略

只允许在散列表内部为其寻找另一空桶，closed hashing。之前在列表外开辟空间，称为 open hashing

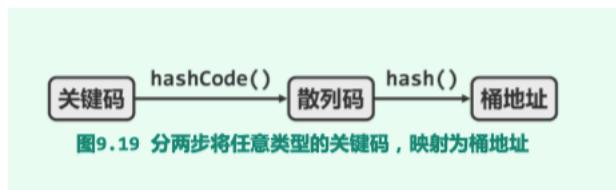
- linear probing: 若 $hash(key)$ 被占用，转而试探 $hash(key)$ 之后的桶，直至找到空桶为止
 - 装填因子: $\lambda = N/M < 0.5$
 - 查找链: 从重复的位置开始
 - 局部性: 当散列表规模不小，装填因子不大的时候，闭散列对 I/O 负担的降低是更好的选择
 - 懒惰删除: 当词条直接删除，会导致后续所有查找的失败，如果需要删除对象，直接将桶的标志位 $ht[r]$ 表姐为 `lazilyRemoved(t)`
 - 删除查找: 当桶为空且没有 `remove` 标志才算“查找失败”
 - 插入查找: 当桶为空或带有 `remove` 标志，则可以用于插入
 - 重散列: 当装填因子越过某一个阈值时，调用 `rehash()` 算法

更多闭散列策略

- linear probing 会加剧聚集现象
- quadratic probing: 缓解聚集现象
 - 确保试探终止: linear probing 试探一遍必然停止；quadratic probing 要保证 $\lambda \leq 50\%$ 才能保证试探终止于某个空桶

- pseudo-random probing
- double hashing: 发现 $ht[hash(key)]$ 被占用之后, 以 $hash_2(key)$ 为偏移量进行尝试: $[hash(key) + i \times hash_2(key)] \% M$

散列码转换



- 不超过 32 位的转成 32 位
- 超过 32 位的分成高低 32 位求和
- 字符串采用多项式散列码 (polynomial hash code)

9.3 散列应用

9.3.1 桶排序 bucketsort

- 非重复情况直接使用关键码对应秩置位
- 重复情况, 使用独立链表, 重复的置于链表之中

9.3.2 基数排序 radixsort

- least significant digit first: 低位字段优先
- 时间复杂度: $O(t * (n + M))$

表9.3 基数排序实例							
输入序列	4 4 1	2 7 6	3 2 0	2 1 4	6 9 8	2 8 0	1 1 2
以个位排序	3 2 0	2 8 0	4 4 1	1 1 2	2 1 4	2 7 6	6 9 8
以十位排序	1 1 2	2 1 4	3 2 0	4 4 1	2 7 6	2 8 0	6 9 8
以百位排序	1 1 2	2 1 4	2 7 6	2 8 0	3 2 0	4 4 1	6 9 8

CHAPTER 10

优先级队列

- 搜索树: 显式全序结构 (full-order)
- 散列表: 隐式全序结构 (full-order), 类似牺牲空间加快搜索速度
- dictionary: 只要求关键码可以判等
- priority queue: 要求关键码可以比较大小

10.1 priority

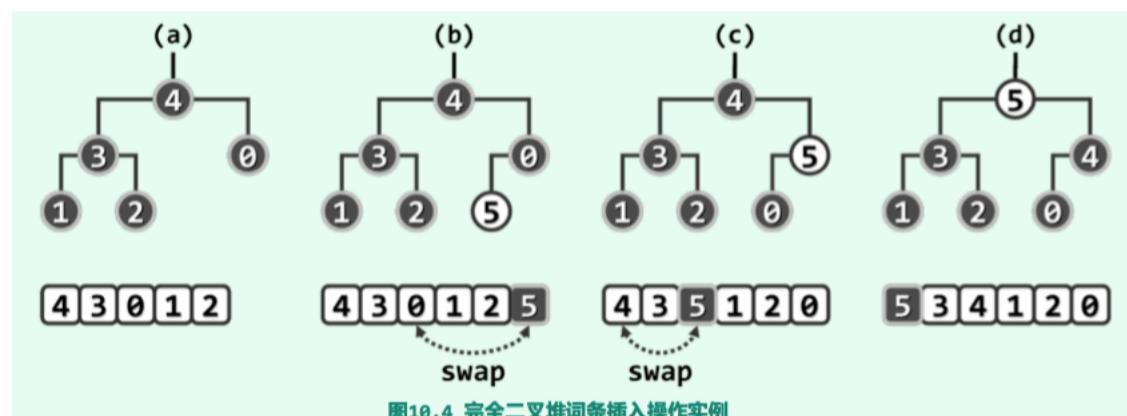
10.1.1 huffman 编码树

列表和向量对于优先级的理解过于机械, 始终都保存了全体词条之间的全序关系, 难以提高效率

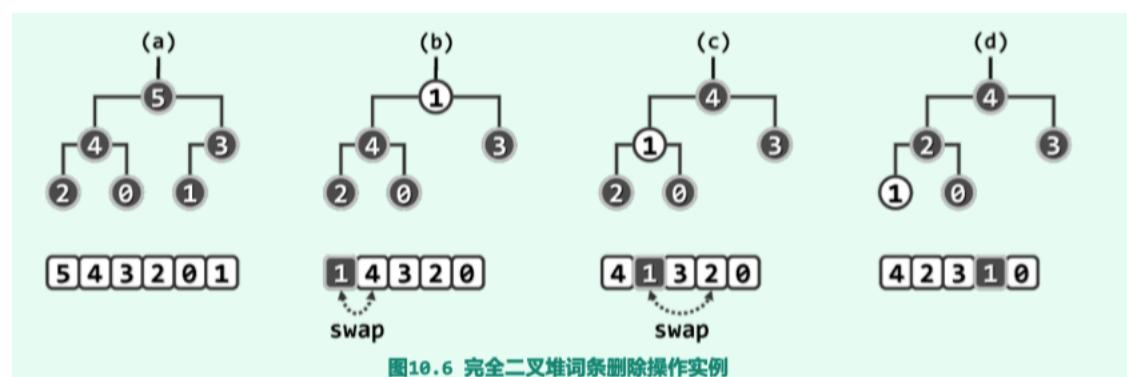
10.2 堆 heap

10.2.1 完全二叉堆

插入—上滤



删除一下滤



建堆 heapification

- 蛮力算法: 时间复杂度 $O(n \log n)$
- Floyd 算法: 堆合并操作, 自下而上的下滤, 时间复杂度 $O(n)$

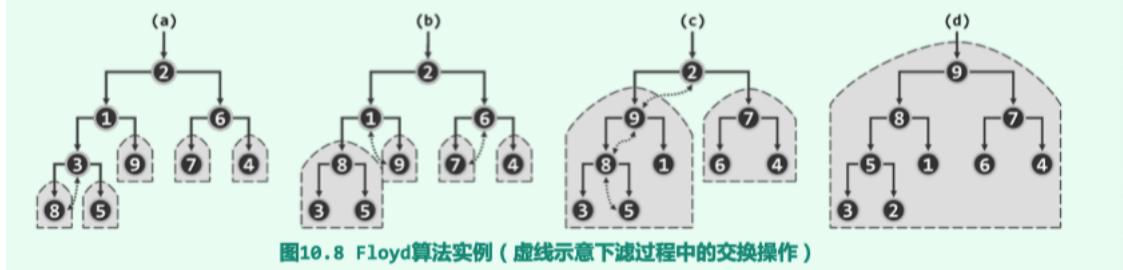


图10.8 Floyd算法实例 (虚线示意下滤过程中的交换操作)

10.2.2 就地堆排序 in-place heapsort

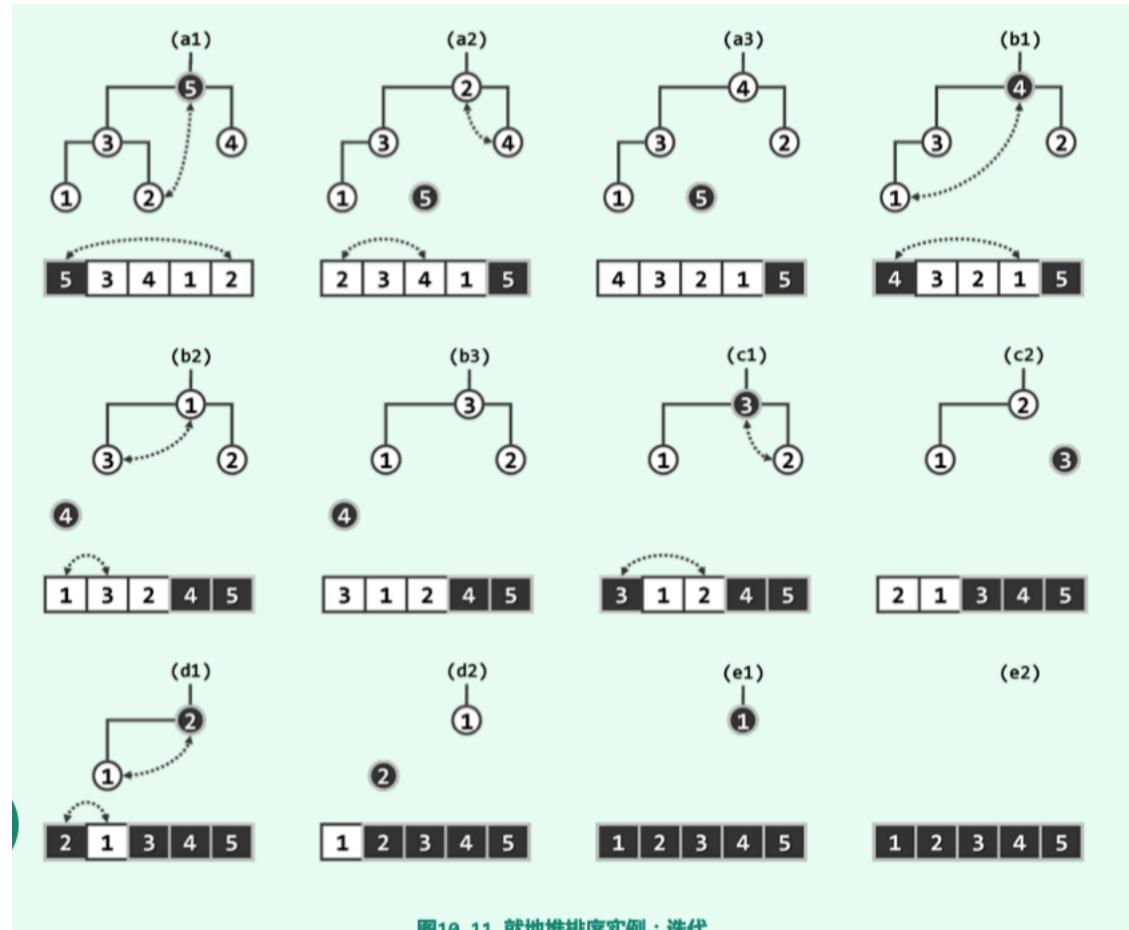


图10.11 就地堆排序实例：迭代

10.3 左式堆

两个堆进行合并组成一个堆

10.3.1 单侧倾斜

leftist heap

左倾性: 任一内部节点 x 都满足左孩子不小于其右孩子, 但前者的高度可能小于后者的高度

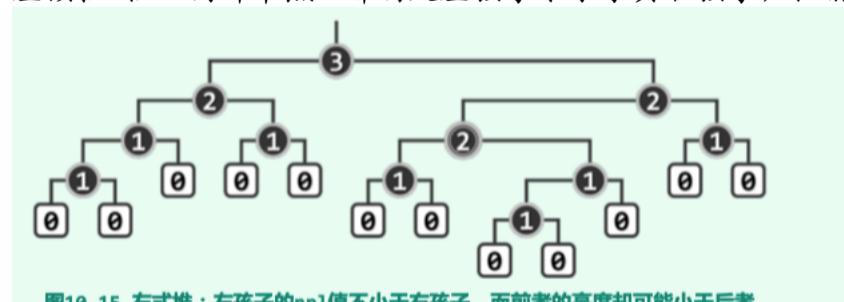
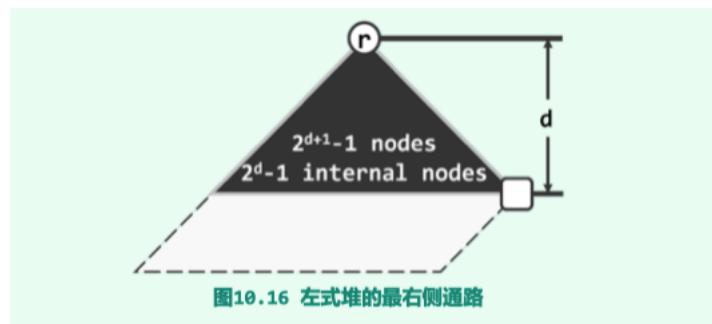


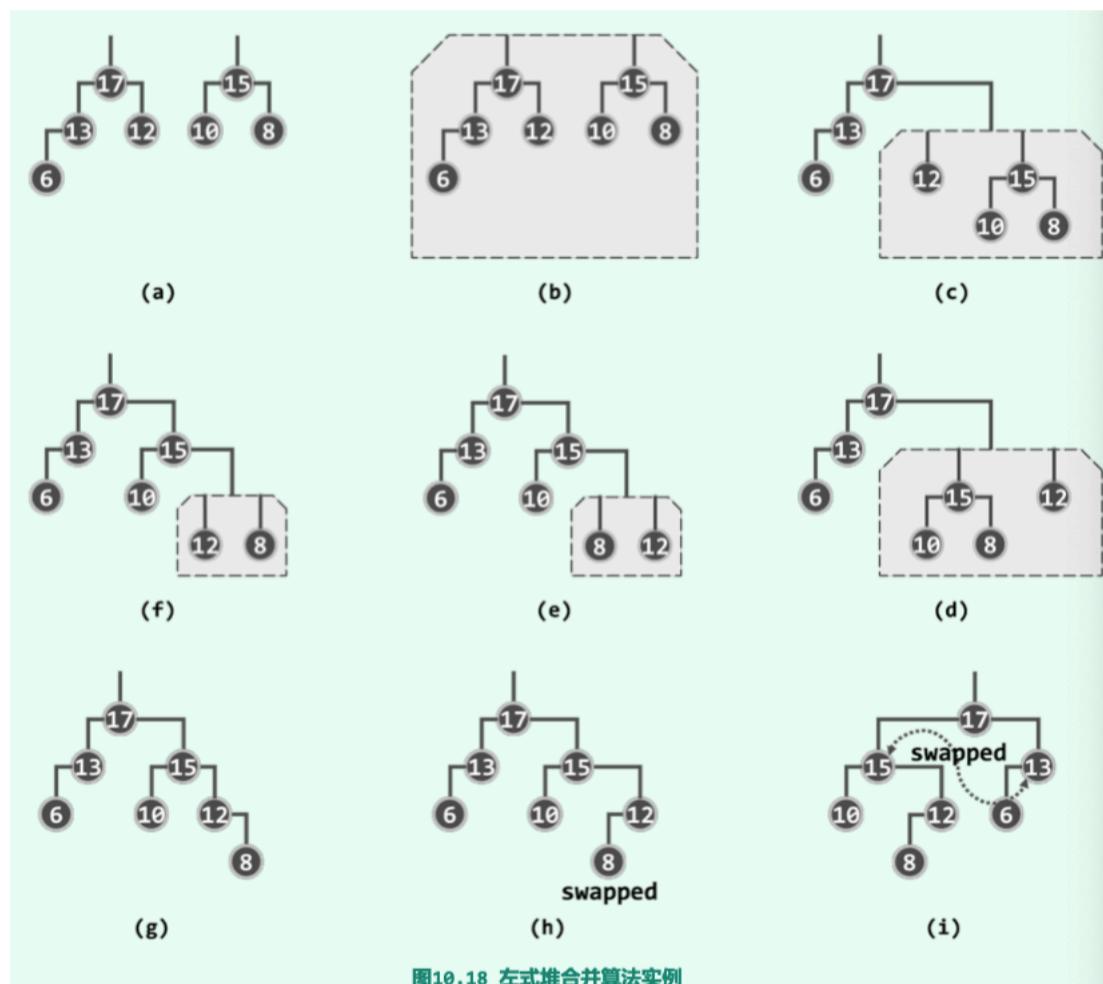
图10.15 左式堆：左孩子的np1值不小于右孩子，而前者的高度却可能小于后者

最右侧通路 rightmost path

根节点的最右侧通路是全堆中深度最小的外部节点



合并算法



CHAPTER 11

串

- call-by-pattern
- 模式检测 (pattern detection)
- 模式定位 (pattern location)
- 模式计数 (pattern counting)
- 模式枚举 (pattern enumeration)

11.1 蛮力算法

- 需要 $n-m+1$ 次比对，每次比对需要比对至多 m 次，总体最差情况消耗时间 $O(n * m)$

11.2 KMP 算法



- 11.4 中可以看出，利用 KMP 不需要回退 i ，而是保持 i 不变，根据 P 的特性， j 只需要回退到 $j-1$ 即可
- 11.5 可以看出，不需要回退 i ，而是将 j 回退到 1
- 这种方法利用了匹配的 substring 找到 proper prefix 和 proper suffix 重叠的长度 t ，使得 $j = j-t, i$ 保持不变
$$next[j] = \max(N(P, j)) = \max\{\theta \leq t < j \mid P[\theta, t) = P[j - t, j)\}$$
- 这个 $next$ 表格可以由 pattern p 提前计算得出

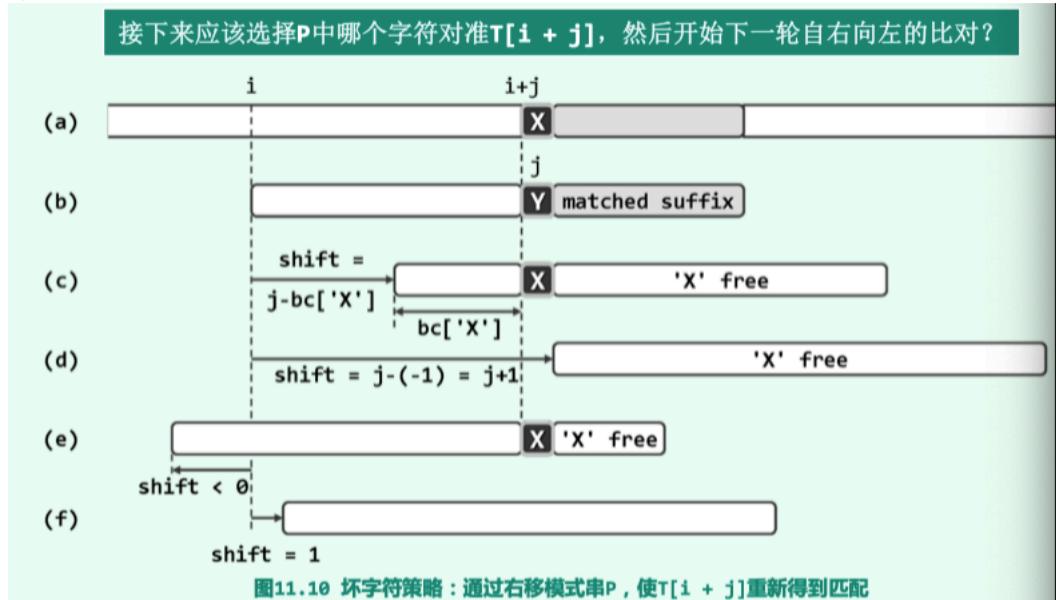
11.2.1 KMP 改进

由于 $T[i] \neq p[j]$ 导致的错配，那么 $p[t]=p[j]$ 时，对应的字符只会匹配失败，所以判定条件改为 $N(P, j) = \{\theta \leq t < j \mid P[\theta, t) = P[j - t, j)\}$

表11.5 改进后的next表实例						
rank	-1	0	1	2	3	4
$P[]$	*	0	0	0	0	1
$next[]$	N/A	-1	-1	-1	-1	3

11.3 BM 算法

采用自右向左匹配的方式



其中 $T[i+j]$ 对应的是错配的字符，在后续的未匹配字符中如果没有 $T[i+j]$ ，那么字符串可以直接移动全长，否则移动到 $T[i+j]$ 出现的位置再次匹配

表11.6 模式串P = "DATA STRUCTURES"及其对应的BC表																
rank	-1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
P[]	*	D	A	T	A	~	S	T	R	U	C	T	U	R	E	S
char	~	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

char	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
bc[]	4	3	-1	9	0	13	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	12	14	10	11	-1	-1	-1	-1	

bc 表的制作方法，char 只存储最右侧的x 对应的序号，与比对的方向相一致，最坏情况 $O(n * m)$

改进

不论 $T[i+j]$ 与未匹配的前缀的匹配结果，最终还是要已匹配的后缀和未匹配的前缀相结合，所以应该对照 pattern 制作一个表格表明在该匹配位置出错之后 pattern 应该如何向右移动。最差运行时间 $O(n+m)$

表11.7 模式串P = "ICED RICE PRICE"对应的GS表															
j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
P[j]	I	C	E	D	~	R	I	C	E	~	P	R	I	C	E
gs[j]	12	12	12	12	12	12	12	12	12	12	6	12	15	15	1

gs[] 表构造原则

ss[] 标识真后缀的长度

表11.8 模式串P = "ICED RICE PRICE"对应的SS表															
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
P[i]	I	C	E	D	~	R	I	C	E	~	P	R	I	C	E
ss[i]	0	0	3	0	0	0	0	0	4	0	0	0	0	0	15

11.4 karp-rabin 算法