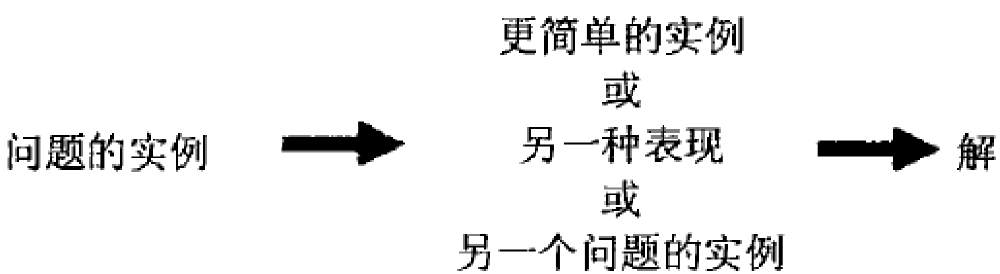
数据分析与算法设计
第6章 变治法
（Transform-and-Conquer）

李旻
百人计划研究员
浙江大学 信息与电子工程学院
Email: min.li@zju.edu.cn

变治法

- 变：将问题实例变得更容易求解
- 治：对变换后的实例进行求解
- 变治思想3种类型
  - 实例化简（instance simplification）—> 同样问题
  - 改变表现（representation change）—> 同样实例
  - 问题化简（problem reduction） —> 另一问题



由难化易
由繁化简

目录

- 实例化简
  - 预排序
  - 高斯消元法
  - 平衡查找树：AVL树
- 改变表现
  - 平衡查找树：2-3树
  - 堆排序
  - 霍纳法则和二进制幂
- 问题化简
  - 简化为图问题等

预排序

- 思想：对于某些以数组为数据结构的问题，若可以对数组进行预先排序，则相应的问题求解往往会更容易
- 实例1：检验数组中元素的唯一性
  - 蛮力法

算法 UniqueElements($A[0..n-1]$)
```
//验证给定数组中的元素是否全部唯一
//输入：数组 A[0..n-1]
//输出：如果 A 中的元素全部唯一，返回 true
//      否则，返回 false
for i ← 0 to n-2 do
    for j ← i+1 to n-1 do
        if A[i] = A[j] return false
return true
```

$\Theta(n^2)$复杂度

  - 预排序：先排后检

算法 PresortElementUniqueness($A[0..n-1]$)
```
//先对数组排序来解元素唯一性问题
//输入：n 个可排序元素构成的一个数组 A[0..n-1]
//输出：如果 A 没有相等的元素，返回 true，否则返回 false
对数组 A 排序
for i ← 0 to n-2 do
    if A[i] = A[i+1] return false
return true
```

$$T(n) = T_{sort}(n) + T_{scan}(n) \in \Theta(n \log n) + \Theta(n) = \boxed{\Theta(n \log n)}$$

预排序

- 实例2：模式计算
  - 模式：数组中出现频率最高的元素
    - 数组[5, 1, 5, 7, 6, 5, 7]的模式是5
  - 蛮力法：逐个元素遍历+ 统计频率 + 频率比较
    - $\Theta(n^2)$（最差效率）
  - 预排序：先对输入排序，那么所有相等的数值都会邻接在一起，则只需求出邻接次数最多的等值元素

```
算法 PresortMode(A[0..n-1])
     //先对数组排序来计算它的模式
     //输入：可排序元素构成的数组 A[0..n-1]
     //输出：该数组的模式
     对数组 A 排序
     i ← 0                    //当前一轮从位置 i 开始
     modefrequency ← 0   //目前为止求出的最高频率
     while i ≤ n-1 do
          runlength ← 1;  runvalue ← A[i]
          while i + runlength ≤ n-1 and A[i + runlength] = runvalue
               runlength ← runlength + 1
          if runlength > modefrequency
               modefrequency ← runlength;  modevalue ← runvalue
          i ← i + runlength
     return modevalue
```

$\Theta(n\log n)$

预排序

- 实例3：查找指定元素
  - 蛮力法：顺序查找，n次比较
  - 预排序：先排序，后折半查找

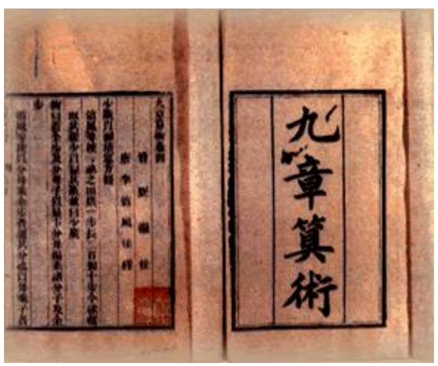
$$T(n) = T_{sort}(n) + T_{search}(n) = \Theta(n \log n) + \Theta(\log n) = \Theta(n \log n)$$

目录

高斯消元法

- 线性方程组的求解

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\
&\vdots \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n
\end{aligned}
$$




九章算術卷八

方程以御錯糅正負

今有上禾三秉中禾二秉下禾一秉實三十九

斗上禾二秉中禾三秉下禾一秉實三十四斗

上禾一秉中禾二秉下禾三秉實二十六斗問

上中下禾實一秉各幾何

答曰

$$
\begin{cases}
3x + 2y + z = 39 \\
2x + 3y + z = 34 \\
x + 2y + 3z = 26
\end{cases}
$$

《九章算术》：世界上最早的线性方程组及“消元”解法（公元前150年左右）

高斯消元法

- 基本思想
  - 将原始方程组化简为一个系数矩阵为上三角形式的等价方程组，则可采用反向替换的方法计算方程组的解

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\
&\vdots \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n
\end{aligned}
\quad \Longrightarrow \quad
\begin{aligned}
a'_{11}x_1 + a'_{12}x_2 + \cdots + a'_{1n}x_n &= b'_1 \\
a'_{22}x_2 + \cdots + a'_{2n}x_n &= b'_2 \\
&\vdots \\
a'_{nn}x_n &= b'_n
\end{aligned}
$$

$$Ax = b \qquad\qquad A'x = b'$$

矩阵表示

初等变换

$(\lambda E_i) \to E_i$

$E_i + (\lambda E_j) \to E_i$

$E_i \leftrightarrow E_j$

$$
A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix},\ b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}
\quad \Longrightarrow \quad
A' = \begin{bmatrix} a'_{11} & a'_{12} & \dots & a'_{1n} \\ 0 & a'_{22} & \dots & a'_{2n} \\ \vdots & & & \\ 0 & 0 & \dots & a'_{nn} \end{bmatrix},\ b = \begin{bmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_n \end{bmatrix}
$$

高斯消元法

- 前向消元：举例(n=3)

$$\underbrace{\begin{bmatrix} \boxed{a_{11}^{(1)}} & a_{12}^{(1)} & a_{13}^{(1)} \\ a_{21}^{(1)} & a_{22}^{(1)} & a_{23}^{(1)} \\ a_{31}^{(1)} & a_{32}^{(1)} & a_{33}^{(1)} \end{bmatrix}}_{A^{(1)}=A} \xrightarrow[M^{(1)}=\begin{bmatrix} 1 & 0 & 0 \\ -m_{21} & 1 & 0 \\ -m_{31} & 0 & 1 \end{bmatrix}]{\substack{E_2-m_{21}E_1\to E_2 \quad (m_{21}=a_{21}^{(1)}/a_{11}^{(1)}) \\ E_3-m_{31}E_1\to E_3 \quad (m_{31}=a_{31}^{(1)}/a_{11}^{(1)})}} \underbrace{\begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} \\ 0 & \boxed{a_{22}^{(2)}} & a_{23}^{(2)} \\ 0 & a_{32}^{(2)} & a_{33}^{(2)} \end{bmatrix}}_{A^{(2)}=M^{(1)}A}$$

主元

比例因子

$$\xrightarrow[M^{(2)}=\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -m_{32} & 1 \end{bmatrix}]{E_3-m_{32}E_2\to E_3 \quad (m_{32}=a_{32}^{(2)}/a_{22}^{(2)})} \underbrace{\begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} \\ 0 & 0 & a_{33}^{(3)} \end{bmatrix}}_{A'=A^{(3)}=M^{(2)}A^{(2)}}$$

高斯消元法

$$
\begin{aligned}
2x_1 - x_2 + x_3 &= 1 \\
4x_1 + x_2 - x_3 &= 5 \\
x_1 + x_2 + x_3 &= 0.
\end{aligned}
$$

$$
\begin{array}{l}
\left[\begin{array}{rrrr} 2 & -1 & 1 & 1 \\ 4 & 1 & -1 & 5 \\ 1 & 1 & 1 & 0 \end{array}\right]
\end{array}
\begin{array}{l} \\ \text{row 2} - \frac{4}{2}\ \text{row 1} \\ \text{row 3} - \frac{1}{2}\ \text{row 1} \end{array}
$$

$$
\left[\begin{array}{rrrr} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & \frac{3}{2} & \frac{1}{2} & -\frac{1}{2} \end{array}\right]
\begin{array}{l} \\ \\ \text{row 3} - \frac{1}{2}\ \text{row 2} \end{array}
$$

$$
\left[\begin{array}{rrrr} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 0 & 2 & -2 \end{array}\right]
\Longrightarrow
$$

前向消元

$$
\begin{aligned}
x_3 &= (-2)/2 = -1 \\
x_2 &= (3 - (-3)x_3)/3 = 0 \\
x_1 &= (1 - x_3 - (-1)x_2)/2 = 1
\end{aligned}
$$

后向替换

高斯消元法

- 前向消元的一些改进
  - 主元不能为0，因此，若第i次迭代中A[i,i]=0，则需要用下面的某行与第i行进行交换
  - 主元不能太小，否则当A[j,i]与A[i,i]数量级差别比较大时，比例因子$m_{ji} = A[j,i]/A[i,i]$会非常大，容易导致计算误差以及相应的误差传递。一种改进的方法是部分选主元法（Partial Pivoting），其思路是每次迭代都去找第i列系数的绝对值最大的行，把它作为第i次迭代的基点，这样可以保证消元过程中比例因子的绝对值永远不会大于1

高斯消元法

算法 BetterForwardElimination($A[1..n, 1..n], b[1..n]$)

```
//用部分选主元法实现高斯消去法
//输入：矩阵 A[1..n, 1..n]和列向量 b[1..n]
//输出：一个代替 A 的上三角形等价矩阵图，相应的右边的值位于第(n + 1)列中
for i ← 1 to n do  A[i, n+1] ← b[i]  //把 b 作为最后一列添加到 A 中
for i ← 1 to n−1 do
    pivotrow ← i
    for j ← i+1 to n do
        if |A[j,i]| > |A[pivotrow,i]| pivotrow ← j
    for k ← i to n+1 do
        swap(A[i, k], A[pivotrow, k])
    for j ← i+1 to n do
        temp ← A[j,i] / A[i,i]
        for k ← i to n+1 do
            A[j,k] ← A[j,k] − A[i,k] * temp
```

高斯消元法

- 算法效率
  - 前向消元
    - 基本操作：乘法（乘法的实现开销一般比加减法要大）
    - 执行次数

$$
\begin{aligned}
C(n) &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\sum_{k=i}^{n+1} 1 = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}(n+1-i+1) = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}(n+2-i) \\
&= \sum_{i=1}^{n-1}(n+2-i)(n-(i+1)+1) = \sum_{i=1}^{n-1}(n+2-i)(n-i) \\
&= (n+1)(n-1)+n(n-2)+\cdots+3\cdot 1 \\
&= \sum_{j=1}^{n-1}(j+2)j = \sum_{j=1}^{n-1} j^2 + \sum_{j=1}^{n-1} 2j = \frac{(n-1)n(2n-1)}{6} + 2\frac{(n-1)n}{2} \\
&= \frac{n(n-1)(2n+5)}{6} \approx \frac{1}{3}n^3 \in \Theta(n^3).
\end{aligned}
$$

  - 后向替换： $\Theta(n^2)$（为什么?）

$\Longrightarrow \Theta(n^3)$

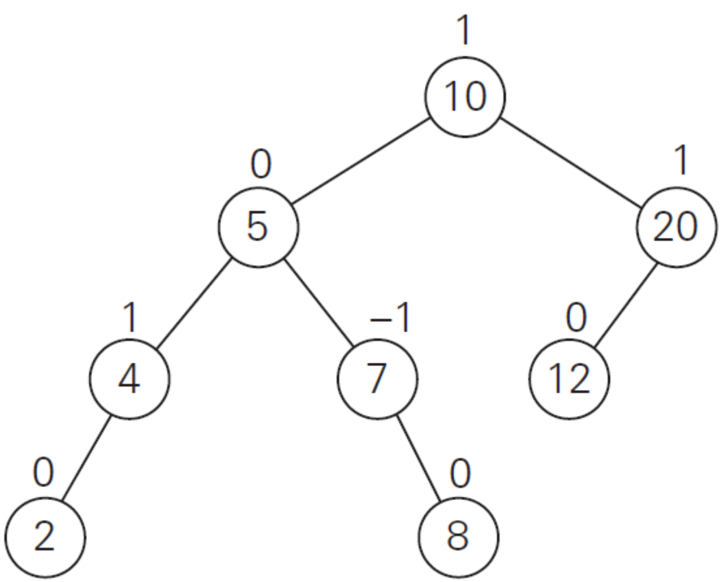
目录

- 实例化简
  - 预排序
  - 高斯消元法
  - 平衡查找树：AVL树
- 改变表现
  - 平衡查找树：2-3树
  - 堆排序
  - 霍纳法则和二进制幂
- 问题化简
  - 简化为图问题等

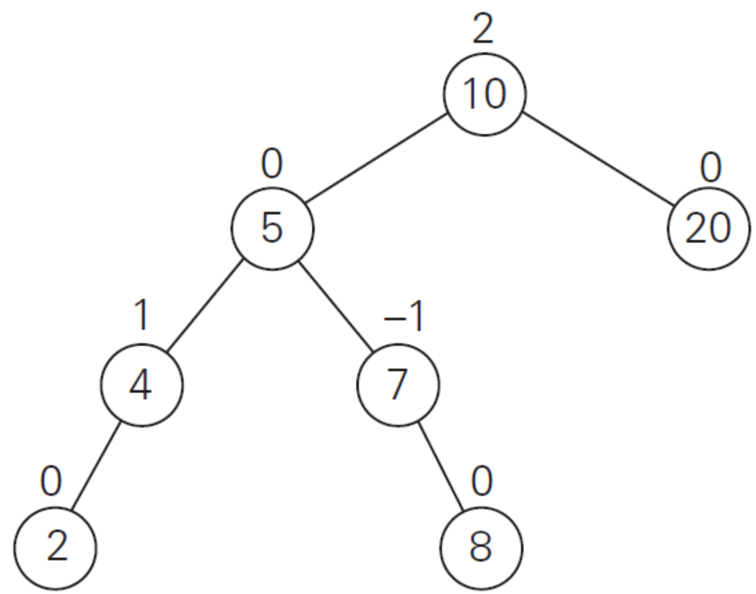
平衡查找树

- 二叉查找树的平衡因子
  - 对于查找树上的任意节点，它的平衡因子是该节点的左子树与右子树的高度差
- 二叉查找树的效率取决于平衡因子的大小，因此，如果我们把一棵不平衡的查找树转变为平衡的形式，那么查找的效率会更高
- AVL树
  - 由G.M. Adelson-Velsky和E.M. Landis发明
  - 每个节点的平衡因子为0，+1或−1

AVL树



AVL树



非AVL树

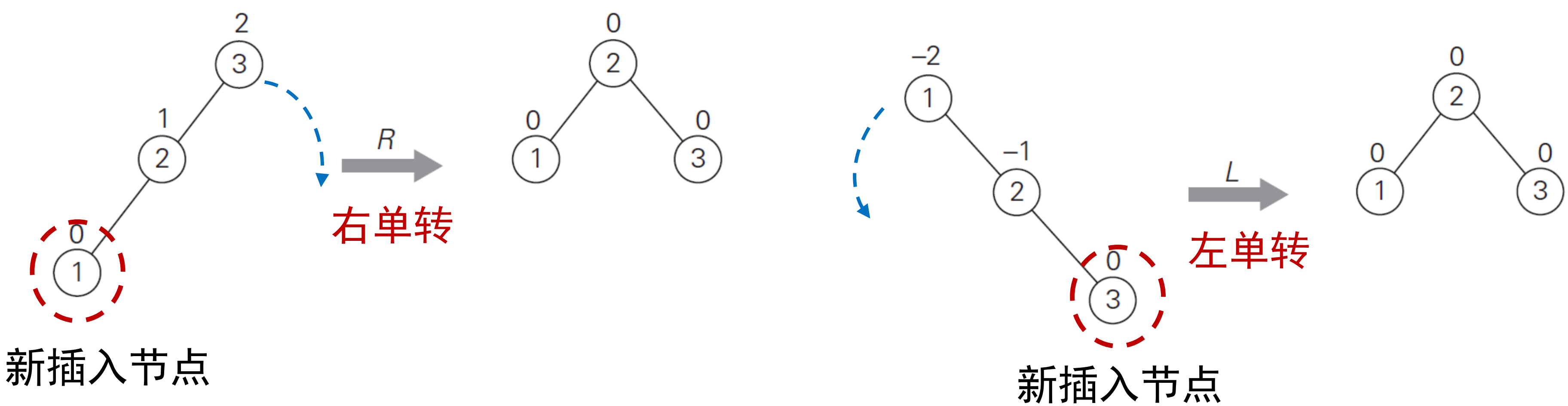
AVL树

- 对于AVL树的一些操作，如插入一个节点，会破坏左右子树的平衡性，需要通过AVL树的平衡旋转化，使其重新平衡

- 4种平衡旋转类型
  - 右单转
  - 左单转
  - 左右双转
  - 右左双转

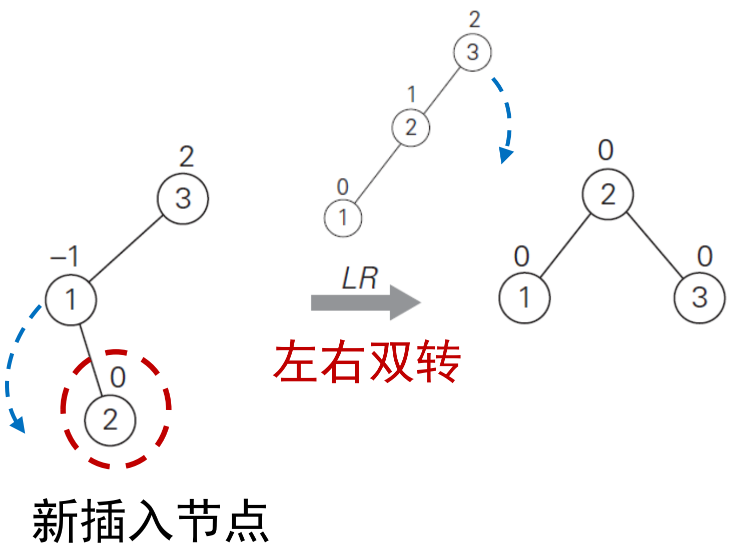
AVL树

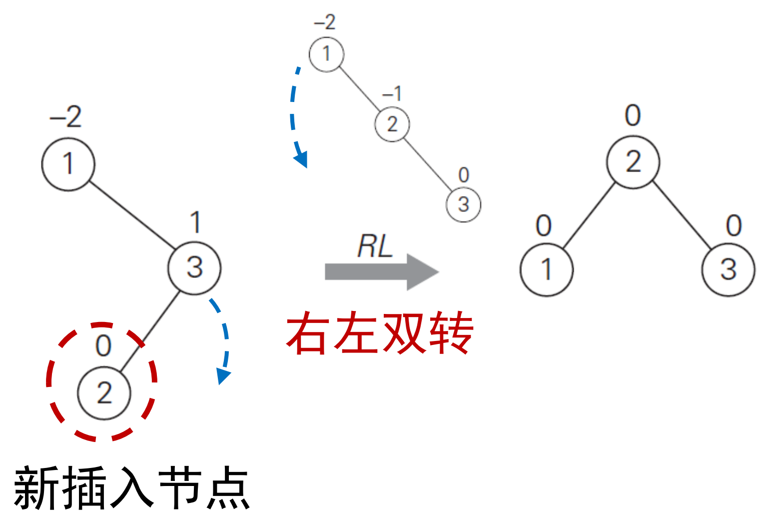
- 平衡旋转（3个节点的AVL树）



AVL树

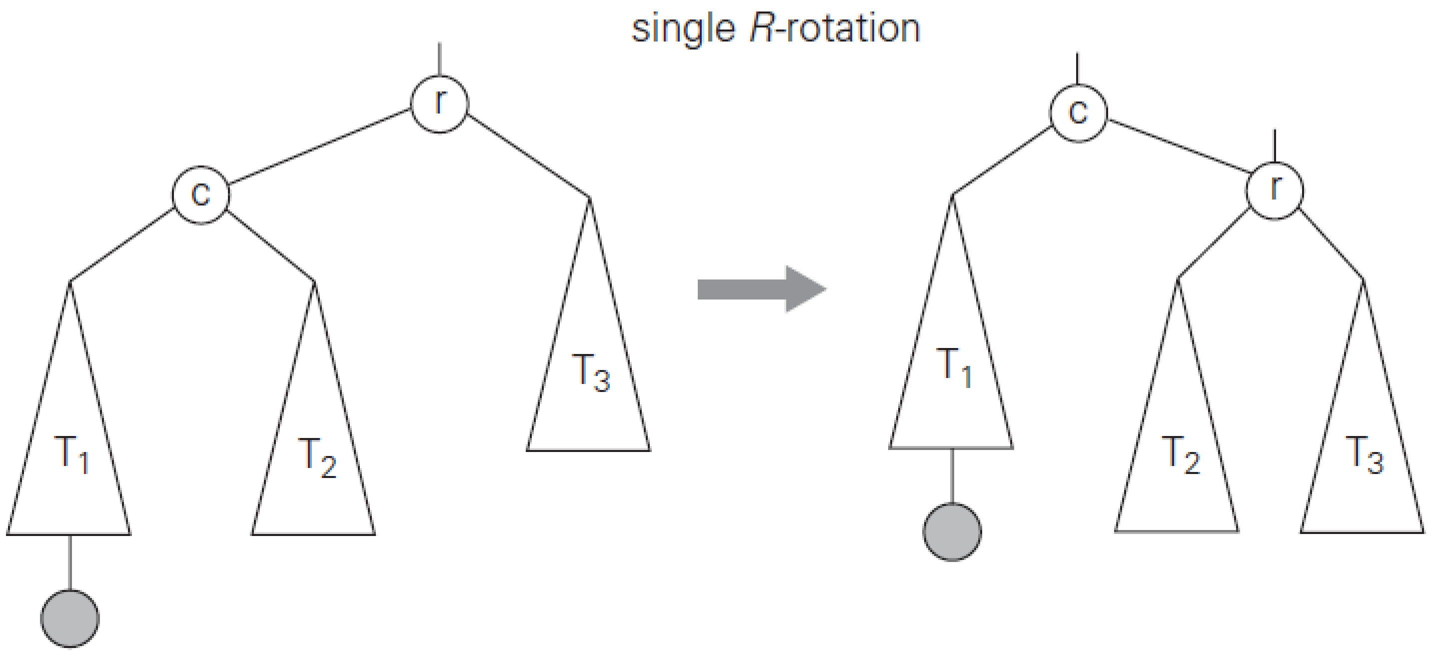
- 平衡旋转（3个节点的AVL树）





AVL树

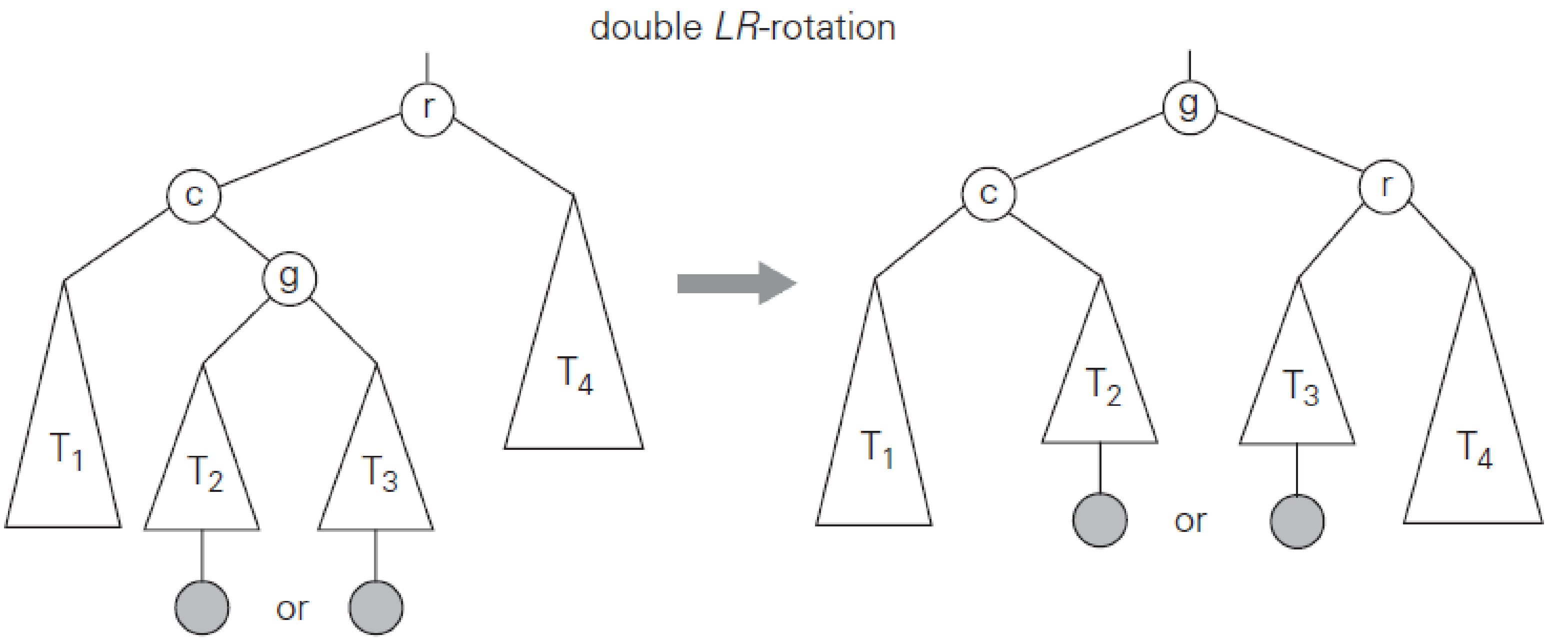
- 平衡旋转（一般性形式）
  - 右单转



  - 左单转为右单转的镜像

AVL树

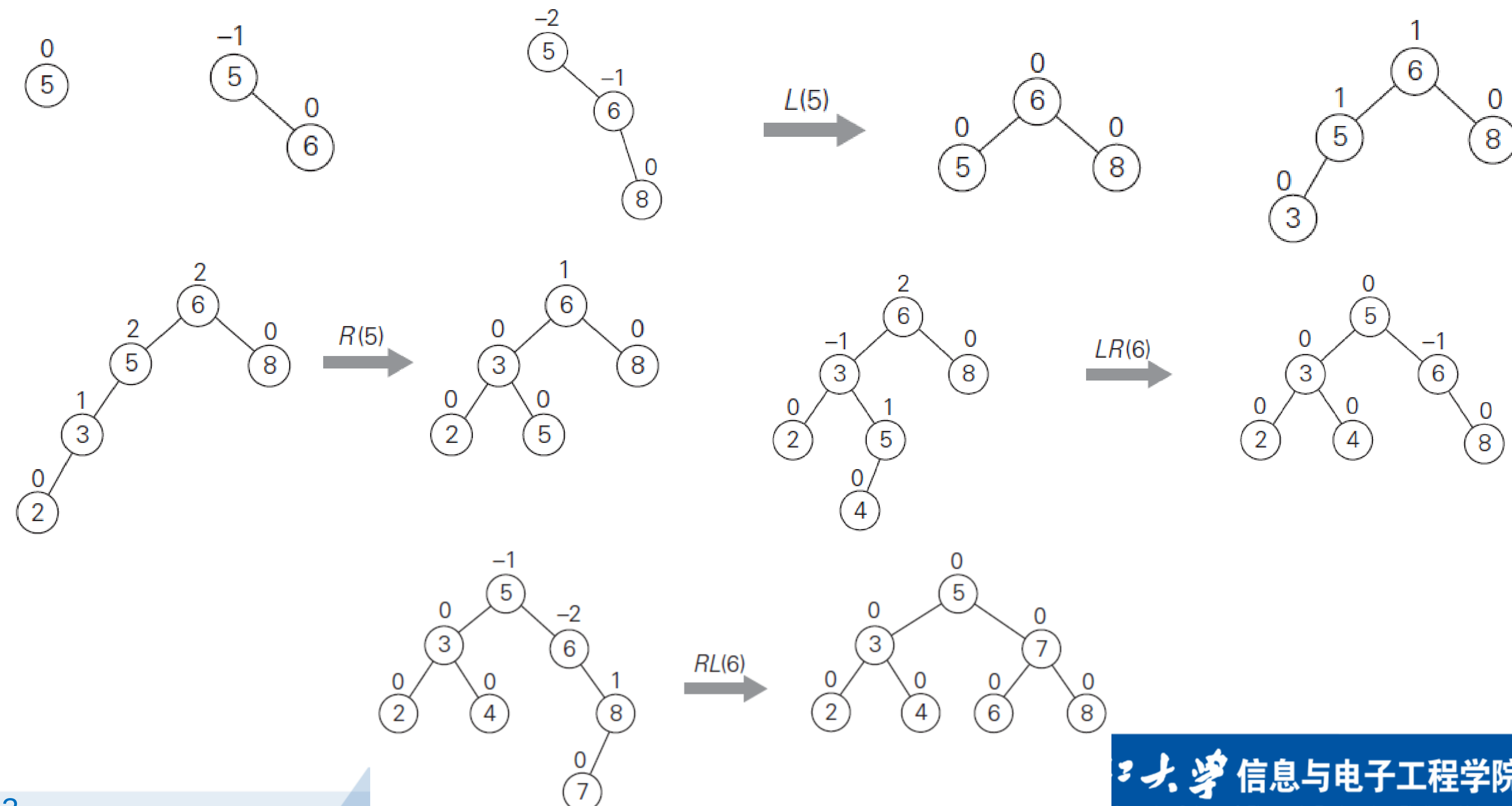
- 平衡旋转（一般性形式）
  - 左右双转



  - 右左双转为左右双转的镜像

AVL树

- 采用平衡旋转，为列表5,6,8,3, 2, 4,7构造一棵AVL树



AVL树

- AVL树的效率
  - 取决于树的高度
  - 包含n个节点的AVL树的高度h满足

$$\lfloor \log_2 n \rfloor \le h < 1.4405 \log_2(n+2) - 1.3277$$

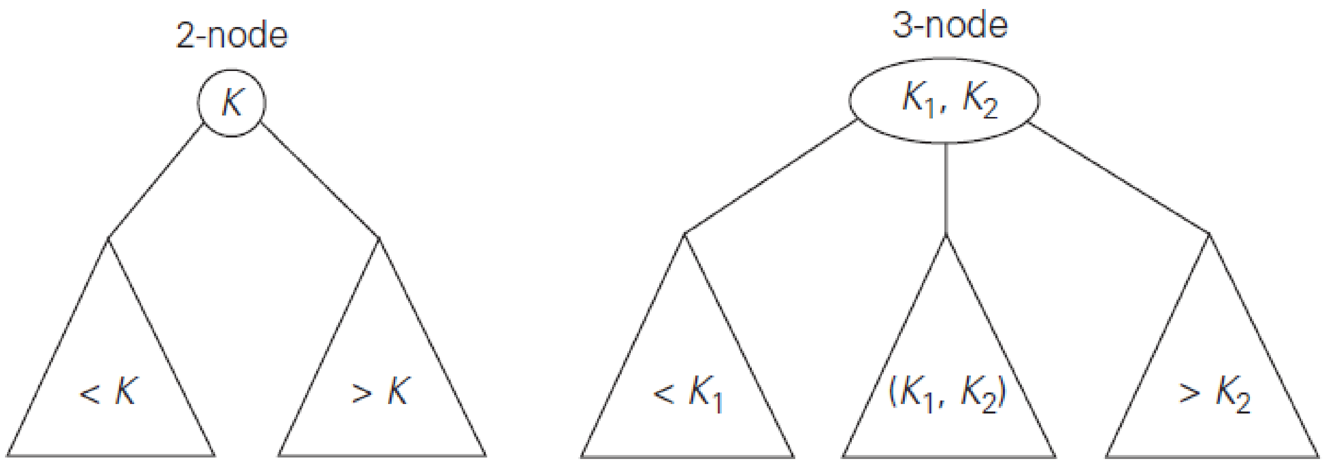
  - 查找、插入和删除操作的时间效率类型均为$\Theta(\log n)$
  - 高效率的代价是：频繁的旋转、维护树的节点的平衡及总体上的复杂性

目录

- 实例化简
  - 预排序
  - 高斯消元法
  - 平衡查找树：AVL树
- 改变表现
  - 平衡查找树：2-3树
  - 堆排序
  - 霍纳法则和二进制幂
- 问题化简
  - 简化为图问题等

2-3树

- 通过“改变表现”来平衡一棵查找树
  - 允许一个节点不止包含一个键
- 2-3树：包含两种类型节点的树
  - 2节点：一个键K和两个子女
  - 3节点：两个有序的键K_1和K_2和3个子女
  - 所有叶子必须位于同一层（总是高度平衡）

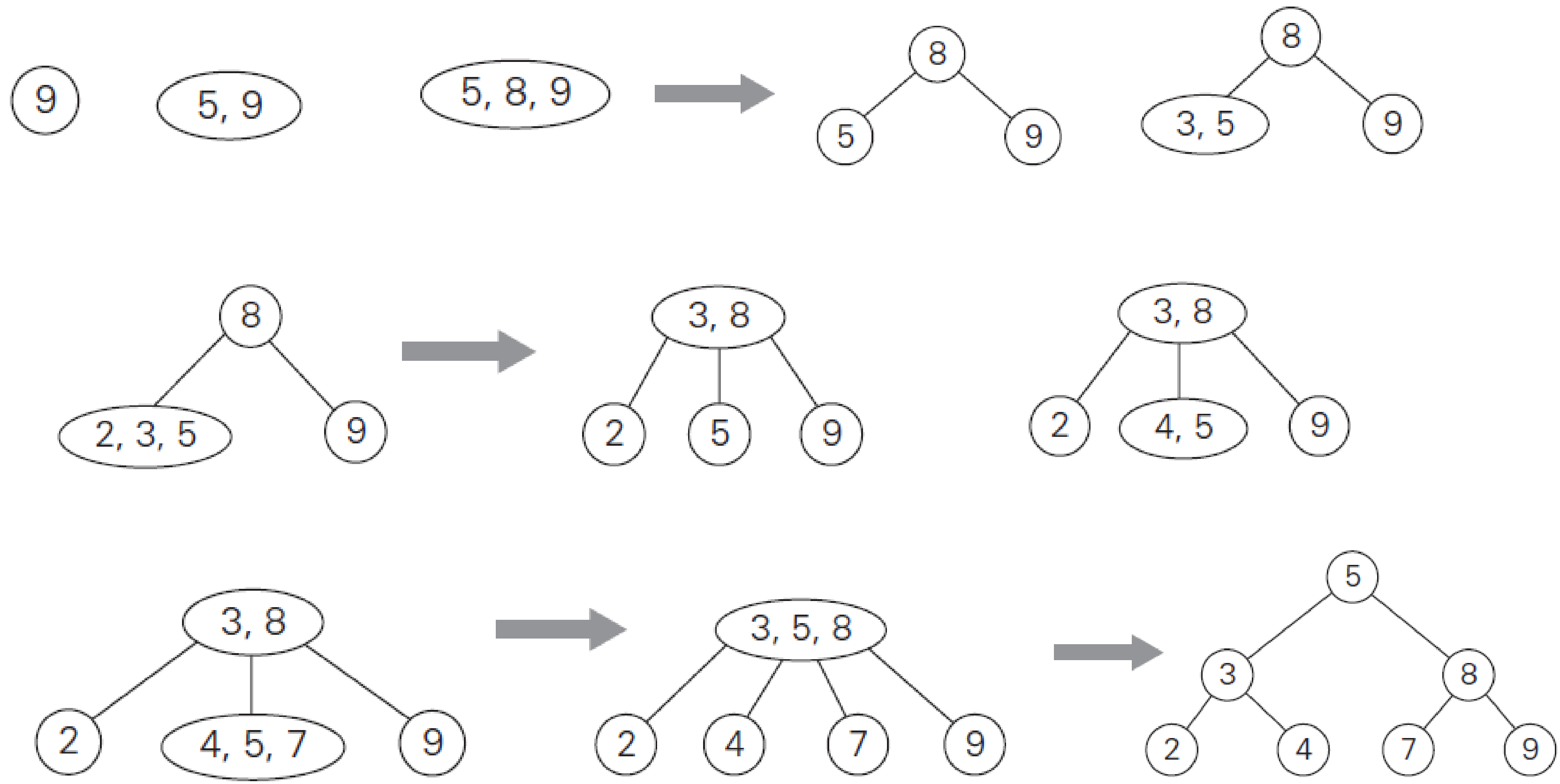


2-3树

- 2-3树的插入操作
  - 总是把一个新的键K插入一个叶子里
  - 如果找到的叶子是一个2节点，根据K是小于或大于节点中原来的键，我们把K作为第一个键或第二个键插入
  - 如果找到的叶子是一个3节点，则把叶子分裂成2个节点：3个键（2个原来的键和1个新键）中最小的放到第一个叶子中，最大的放到第二个叶子中，同时中间的键提升到原来叶子的父母中

2-3树

- 通过插入操作为列表9, 5, 8, 3, 2, 4, 7构造一棵2-3树



2-3树

- 2-3树的查找操作
  - 根是2节点：当作一个二叉查找树处理，比较要查找的键值K与根的键值，就可决定是查找停止，或分别从左子树或右子树继续查找
  - 根是3节点：在不超过两次比较后，就可决定是查找停止，或在根的某一棵子树中继续查找

- 2-3树的效率
  - 包含n个节点的2-3树的高度h满足

$$\log_3(n+1)-1 \le h \le \log_2(n+1)-1,$$

  - 查找、插入和删除的时间类型均为$\Theta(\log n)$

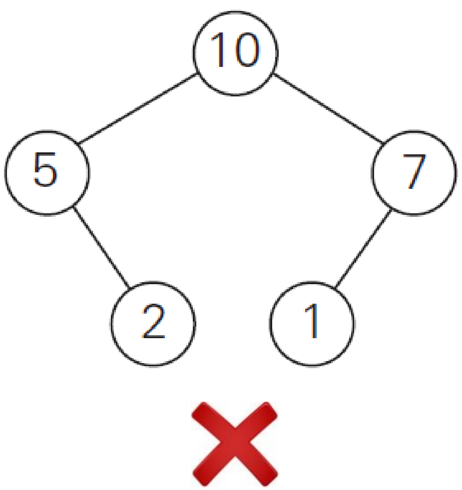
目录

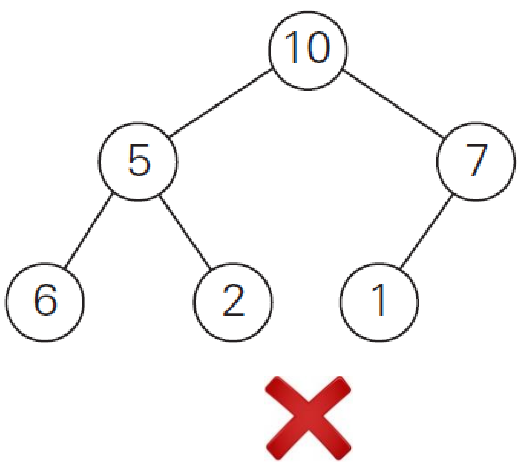
- 实例化简
  - 预排序
  - 高斯消元法
  - 平衡查找树：AVL树
- 改变表现
  - 平衡查找树：2-3树
  - 堆排序
  - 霍纳法则和二进制幂
- 问题化简

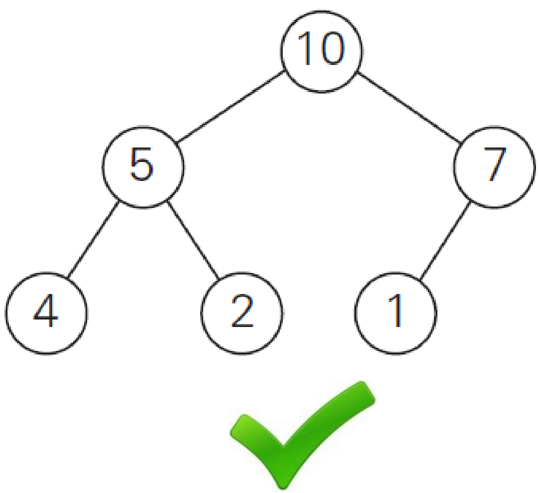
  简化为图问题等

堆(Heap)

- 堆是一棵满足以下条件的二叉树：
  ① 基本完备（essentially complete）：树的每一层都是满的，除了最后一层最右边的元素有可能缺位
  ② 父母优势（parental dominance）：每个节点键值都要大于或等于它子女的键

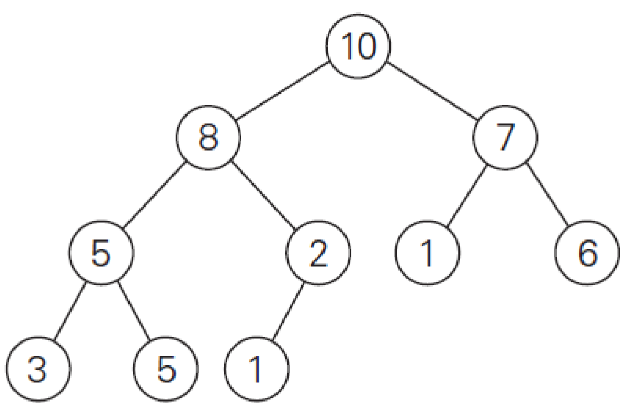






堆 (Heap)

- 重要特性
  - 只存在一棵n个节点的完全二叉树，高度为$\lfloor log_2 n \rfloor$
  - 堆的根总是包含了堆的最大元素
  - 堆的一个节点以及该节点的子孙也是一个堆
  - 可以用数组来实现堆：从上到下、从左到右的方式来记录堆的元素



the array representation

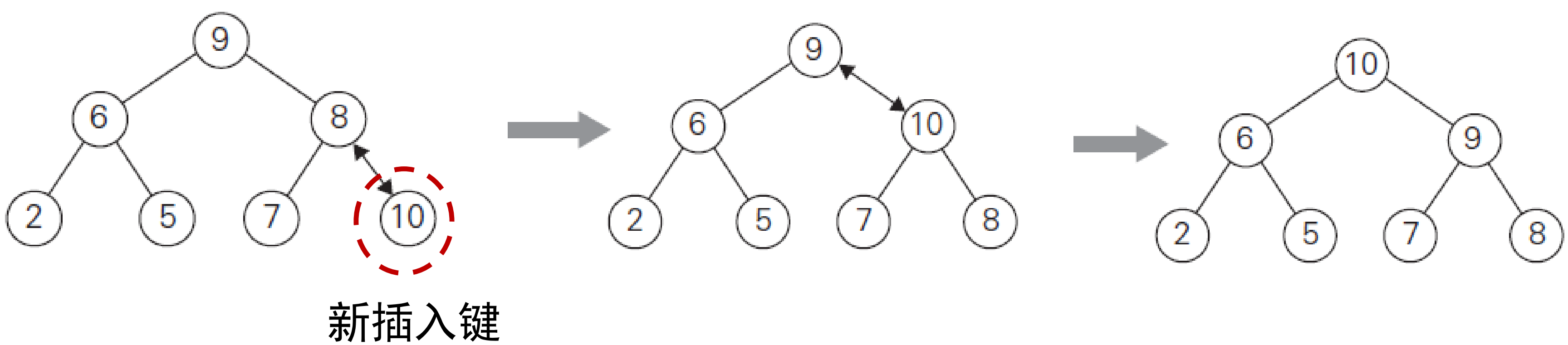
index	0	1	2	3	4	5	6	7	8	9	10
value		10	8	7	5	2	1	6	3	5	1

parents | leaves

位于父母位置i的键，其子女位于2i和2i+1

堆的构造

- 算法1：自顶向下堆构造(键的连续插入)
  ① 把包含键K的新节点附加在当前堆的最后一个叶子后
  ② 通过将K和它的父母键做比较，把K换到合适的位置
  - 后者大于K，算法停止
  - 否则交换这两个键并把K和它的新父母键比较，直到K不大于它的最后一个父母键，或是达到了树的根为止



堆的构造

- 算法2：自底向上堆构造

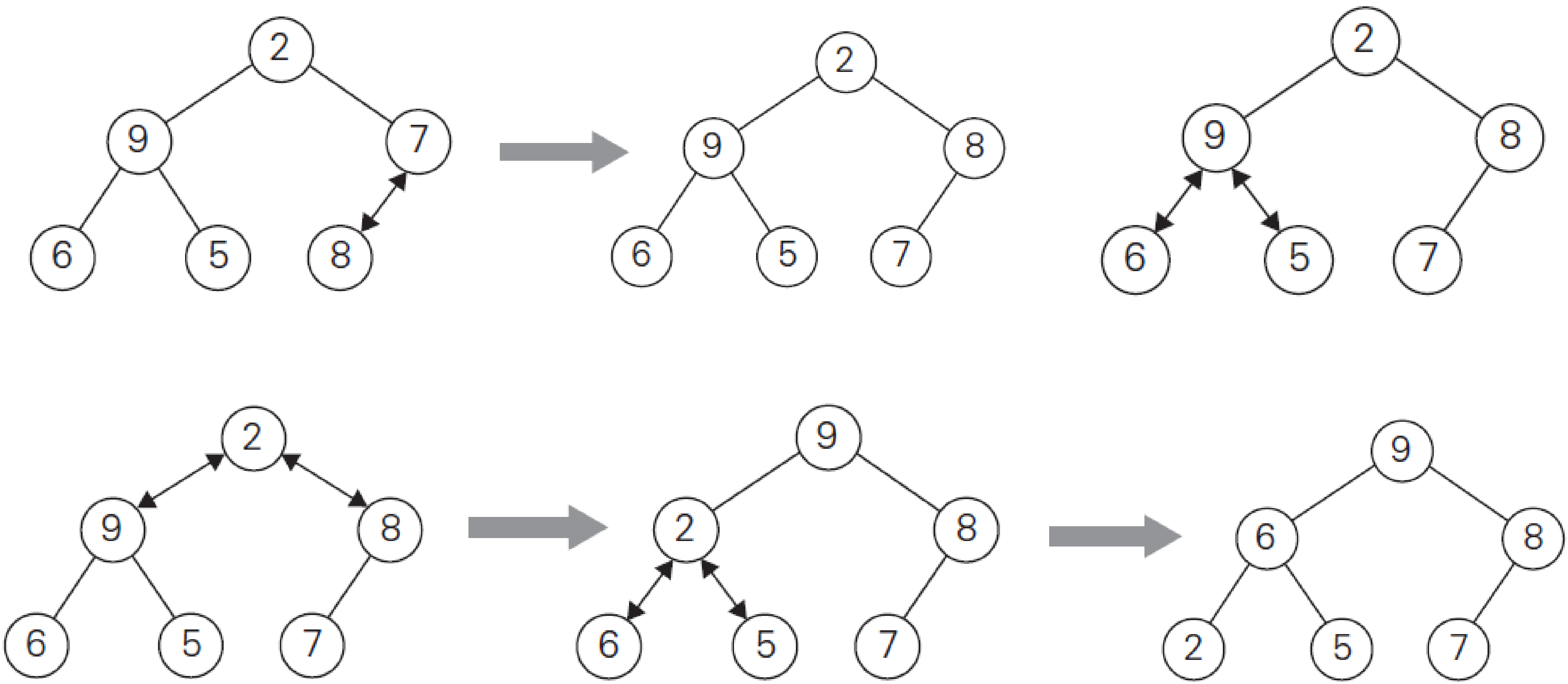
① 按照给定的顺序初始化包含n个节点的完全二叉树

② 按照自下而上，从右至左的顺序，逐个考查父母节点是否比子女节点大，如果不满足，交换它们的键值，并继续在新位置检查父母优势要求

③ 在当前节点根的子树完成“堆化”后，对该节点的直接前趋进行同样的操作，直到对树的根完成该操作

堆的构造

- 举例：对于列表2, 9, 7, 6, 5, 8自底向上构造堆



算法 HeapBottomUp($H[1..n]$)

```
//用自底向上算法，从给定数组的元素中构造一个堆
//输入：一个可排序元素的数组 H[1..n]
//输出：一个堆 H[1..n]
for i ← ⌊n/2⌋ downto 1 do
    k ← i; v ← H[k]
    heap ← false
    while not heap and 2*k≤n do
        j ← 2*k
        if j < n  // 存在两个子女
            if H[j] < H[j+1] j ← j+1
        if v ≥ H[j]
            heap ← true
        else H[k] ← H[j]; k ← j
H[k] ← v
```

堆的构造

- 自底向上算法的效率
  - 假设$n = 2^{k-1}$，则树的高度$h = \lfloor log_2 n \rfloor$
  - 最差情况：假设每个位于第i层的键都会移动到叶子层h，而移动到下一层需要两次比较（找出较大的子女；确定是否需交换），因此需$2(h-i)$次比较。所有层总的比较次数为

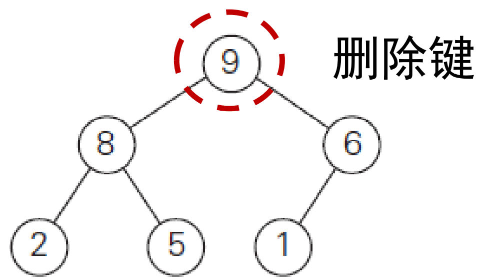
$$C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{\text{level } i \text{ keys}} 2(h-i) = \sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n+1))$$

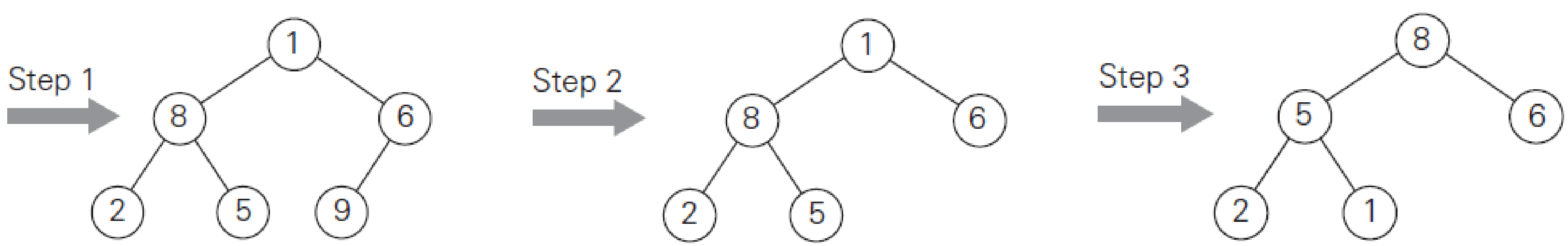
$$\in O(n)$$

浙江大学 信息与电子工程学院
College of Information Science & Electronic Engineering, Zhejiang University

最大键的删除

- 算法思想
  - ① 根的键和堆的最后一个键做交换
  - ② 堆的规模减1
  - ③ 采用自底向上堆构造算法对该树进行“堆化”





堆排序（Heapsort）

- 算法思想
  ① 第一步：为给定的数组构造一个堆（“改变表现”）
  ② 第二步：删除最大键，即对剩下的堆应用n−1次根删除操作

Stage 1 (heap construction)

输入数组 →

2 9 **7** 6 5 8

2 **9** 8 6 5 7

2 9 8 6 5 7

9 **2** 8 6 5 7

9 6 8 2 5 7

自底向上
构建堆

Stage 2 (maximum deletions)

9 6 8 2 5 7

7 6 8 2 5 | **9**

8 6 7 2 5

5 6 7 2 | **8**

7 6 5 2

2 6 5 | **7**

6 2 5

5 2 | **6**

5 2

2 | **5**

2

堆排序

- 算法效率
  - 第一步：$O(n)$
  - 第二步：把堆的规模从n减到2，为了消去根的键，需要的键值比较次数为

$$C(n) \le 2\lfloor \log_2(n-1) \rfloor + 2\lfloor \log_2(n-2) \rfloor + \cdots + 2\lfloor \log_2 1 \rfloor \le 2\sum_{i=1}^{n-1} \log_2 i$$

$$\le 2\sum_{i=1}^{n-1} \log_2(n-1) = 2(n-1)\log_2(n-1) \le 2n\log_2 n. \quad \in O(n\log n)$$

  - 总效率为：$O(n\log n)$ （可进一步证明$\in \Theta(n\log n)$ ）

未完待续…

排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	空间复杂度	稳定性
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
希尔排序	$O(n^{1.3})$	$O(n^2)$	$O(n)$	$O(1)$	不稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
快速排序	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	$O(nlog_2n)$	不稳定
归并排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(n)$	稳定
计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	稳定
桶排序	$O(n+k)$	$O(n^2)$	$O(n)$	$O(n+k)$	稳定
基数排序	$O(n*k)$	$O(n*k)$	$O(n*k)$	$O(n+k)$	稳定

目录

- 实例化简
  - 预排序
  - 高斯消元法
  - 平衡查找树：AVL树
- 改变表现
  - 平衡查找树：2-3树
  - 堆排序
  - 霍纳法则和二进制幂
- 问题化简
  - 简化为图问题等

霍纳法则

- 用于计算多项式的值的算法

$$p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$$

- “改变表现”：不断地把x作为公因子从降次以后的剩余多项式中提取出来

$$p(x) = (\cdots (a_n x + a_{n-1})x + \cdots)x + a_0.$$

- 实例

$$\begin{aligned} p(x) &= 2x^4 - x^3 + 3x^2 + x - 5 \\ &= x(2x^3 - x^2 + 3x + 1) - 5 \\ &= x(x(2x^2 - x + 3) + 1) - 5 \\ &= x(x(x(2x - 1) + 3) + 1) - 5. \end{aligned}$$

$p(3)$=?

coefficients	2	-1	3	1	-5
$x = 3$	2	$3 \cdot 2 + (-1) = 5$	$3 \cdot 5 + 3 = 18$	$3 \cdot 18 + 1 = 55$	$3 \cdot 55 + (-5) = 160$

霍纳法则

- 算法伪代码

```
算法  Horner(P[0..n], x)
  //用霍纳法则求一个多项式在一个给定点的值
  //输入：一个 n 次多项式的系数数组 P[0..n](从低到高存储)，以及一个数字 x
  //输出：多项式在 x 点的值
  p ← P[n]
  for  i ← n−1 downto 0 do
       p ← x * p + P[i]
  return p
```

- 算法效率
  - 乘法和加法的次数相同：$M(n) = A(n) = \sum_{i=0}^{n-1} 1 = n.$
  - 对于一般的多项式，优于“蛮力法”

二进制幂

- 针对幂运算a^n，霍纳法则直接求解退化为“蛮力法”，需要n-1次乘法
- 改进：把指数n用二进制位串来表示，则有“从左到右二进制幂”算法，或“从右到左二进制幂”算法

$$a^n = a^{p(2)} = a^{b_I 2^I + \cdots + b_i 2^i + \cdots + b_0}$$

- 从左到右算法

Horner's rule for the binary polynomial $p(2)$	**Implications for** $a^n = a^{p(2)}$
$p \leftarrow 1$ //the leading digit is always 1 for $n \geq 1$	$a^p \leftarrow a^1$
for $i \leftarrow I - 1$ **downto** 0 **do**	**for** $i \leftarrow I - 1$ **downto** 0 **do**
$p \leftarrow 2p + b_i$	$a^p \leftarrow a^{2p+b_i}$

But

$$a^{2p+b_i} = a^{2p} \cdot a^{b_i} = (a^p)^2 \cdot a^{b_i} = \begin{cases} (a^p)^2 & \text{if } b_i = 0, \\ (a^p)^2 \cdot a & \text{if } b_i = 1. \end{cases}$$

二进制幂

- 从左到右算法

ALGORITHM *LeftRightBinaryExponentiation*$(a, b(n))$

//Computes a^n by the left-to-right binary exponentiation algorithm
//Input: A number a and a list $b(n)$ of binary digits $b_I, \ldots, b_0$
// in the binary expansion of a positive integer n
//Output: The value of a^n
$product \leftarrow a$
for $i \leftarrow I-1$ **downto** 0 **do**
  $product \leftarrow product * product$
  if $b_i = 1$ $product \leftarrow product * a$
return $product$

乘法次数

$$I \le M(n) \le 2I$$

$$(I = \lfloor \log_2 n \rfloor)$$

– 举例：计算a^{13}

n 的二进制位	1	1	0	1
累乘器	a	$a^2 \times a = a^3$	$(a^3)^2 = a^6$	$(a^6)^2 \times a = a^{13}$

二进制幂

- 从右到左算法

$$a^n = a^{b_I 2^I + \cdots + b_i 2^i + \cdots + b_0} = a^{b_I 2^I} \cdot \ldots \cdot a^{b_i 2^i} \cdot \ldots \cdot a^{b_0}.$$

$$a^{b_i 2^i} = \begin{cases} a^{2^i} & \text{if } b_i = 1, \\ 1 & \text{if } b_i = 0, \end{cases}$$

ALGORITHM *RightLeftBinaryExponentiation*$(a, b(n))$

```
//Computes a^n by the right-to-left binary exponentiation algorithm
//Input: A number a and a list b(n) of binary digits b_I, ..., b_0
//         in the binary expansion of a nonnegative integer n
//Output: The value of a^n
term ← a   //initializes a^(2^i)
if b_0 = 1  product ← a
else product ← 1
for i ← 1 to I do
    term ← term * term
    if b_i = 1  product ← product * term
return product
```

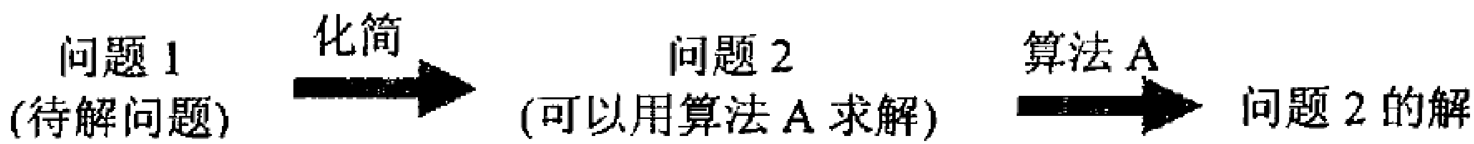
计算a^{13}

1	1	0	1	binary digits of n
a^8	a^4	a^2	a	terms a^{2^i}
$a^5 \cdot a^8 = a^{13}$	$a \cdot a^4 = a^5$		a	product accumulator

目录

- 实例化简
  - 预排序
  - 高斯消元法
  - 平衡查找树：AVL树
- 改变表现
  - 平衡查找树：2-3树
  - 堆排序
  - 霍纳法则和二进制幂
- 问题化简
  - 简化为图问题等

问题化简



简化为图问题

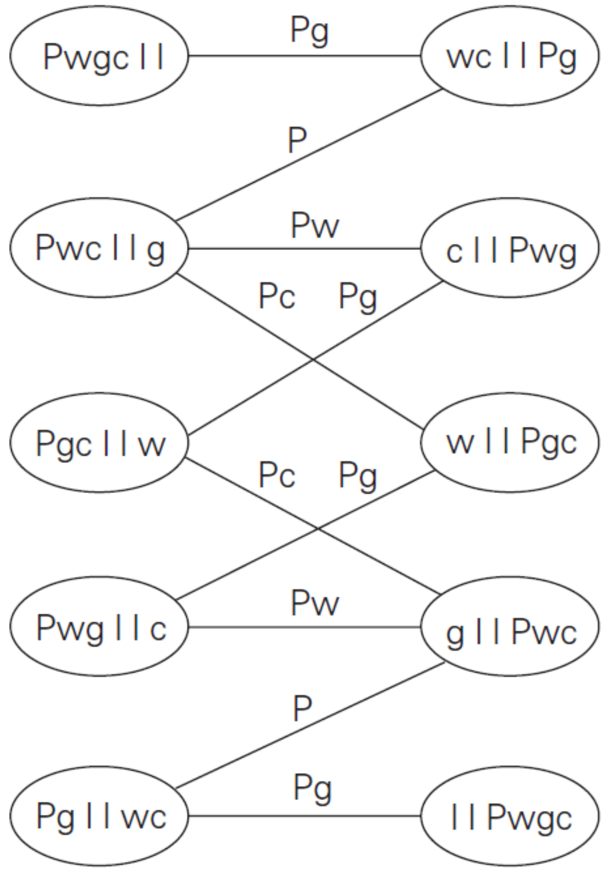
- 许多问题用图表示后，求解很容易。通常用图的顶点表示问题的状态，边表示状态之间的可能转变。表示问题的图称为状态空间图。

- 例如，过河问题：

一个农夫希望用一条小船把一只狼、一头羊、一篮白菜从河的北岸渡到河的南岸，由于船小，只能够容纳人、狼、羊、白菜中的两个。需考虑的约束条件是：在没有人的情况下，狼和羊不能在一起，羊和白菜不能单独在一起。求解一个渡船方案，把狼、羊、白菜都运过去。

简化为图问题

初始状态



结束状态

- P： 农夫
- w： 狼
- g： 羊
- c：白菜
- ||： 河

问题求解化简为：
寻找一条从初始状态到结束状态之间的路径

状态空间图

讨论:称量水果

- 在果园工作的送货员，给一家罐头加工厂送了10箱桃子。每个桃子重500克，每箱装20个。正当他送完货回果园的时候，接到了从果园打来的电话，说由于分类错误，这10箱桃子中有一箱装的是每个400克的桃子，要送货员把这箱桃子带回果园以便更换。但是，怎样从10箱桃子中找出到底是哪一箱的分量不足呢？你有什么办法可以找出那箱桃子呢？能否有只需要称量一次的方法？
  - 蛮力法
  - 减治法
  - 变治法

课后作业

章 X	节 X.Y	课后作业题 Z	思考题 Z
6	6.1	6	9
	6.2	2	11
	6.3	4	9
	6.4	1	12
	6.5	4	12
	6.6	9	11,12

注：只需上交“课后作业题”；以“学号姓名_chX.pdf ”规范命名，提交到“学在浙大”指定文件夹。DDL：2024年4月9日

课后阅读（1）：高斯消元法的副产物

- LU分解
  - 如果采用高斯消元法对Ax=b化简求解时无需任何行变换操作，则矩阵A可以分解成下三角矩阵L和上三角矩阵U的乘积，其中L的非零元素由消元过程中的比例因子构成，而U则为消元后的上三角矩阵

$$A = LU$$

$$L = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ m_{21} & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ m_{n1} & \cdots & m_{n,n-1} & 1 \end{bmatrix} \qquad U = \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & \ddots & \vdots \\ \vdots & \ddots & \ddots & a_{n-1,n}^{(n-1)} \\ 0 & \cdots & 0 & a_{nn}^{(n)} \end{bmatrix}$$

高斯消元法的副产物

- LU分解
  - 如果A的LU分解已知，则可用更低的复杂度来求解线性方程组Ax=b

$$A = LU$$

$$L\mathbf{y} = \mathbf{b} \qquad U\mathbf{x} = \mathbf{y}$$

正向或反向替换：$\Theta(n^2)$

$$A\mathbf{x} = \mathbf{b}$$

高斯消元法的副产物

- 矩阵求逆
  - 思路：把矩阵求逆转成n个线性方程组求解问题，则可采用高斯消元LU分解来求解

$$\underbrace{\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & & & \\ x_{n1} & x_{n2} & \dots & x_{nn} \end{bmatrix}}_{A^{-1}} = \underbrace{\begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & & & \\ 0 & 0 & \dots & 1 \end{bmatrix}}_{I}$$

n 个线性方程组

$$\Longrightarrow Ax^j = e^j \ (1 \le j \le n)$$

LU分解

$$\Longrightarrow LUx^j = e^j$$

课后阅读（2）：线性规划

- 许多决策优化问题可以转化为线性规划问题
- 算法：单纯形法、Karmarkar算法等

- 背包问题的线性规划表示：

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} v_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} w_j x_j \le W \\ & x_j \in \{0, 1\} \quad \text{for } j = 1, \ldots, n. \end{aligned}$$

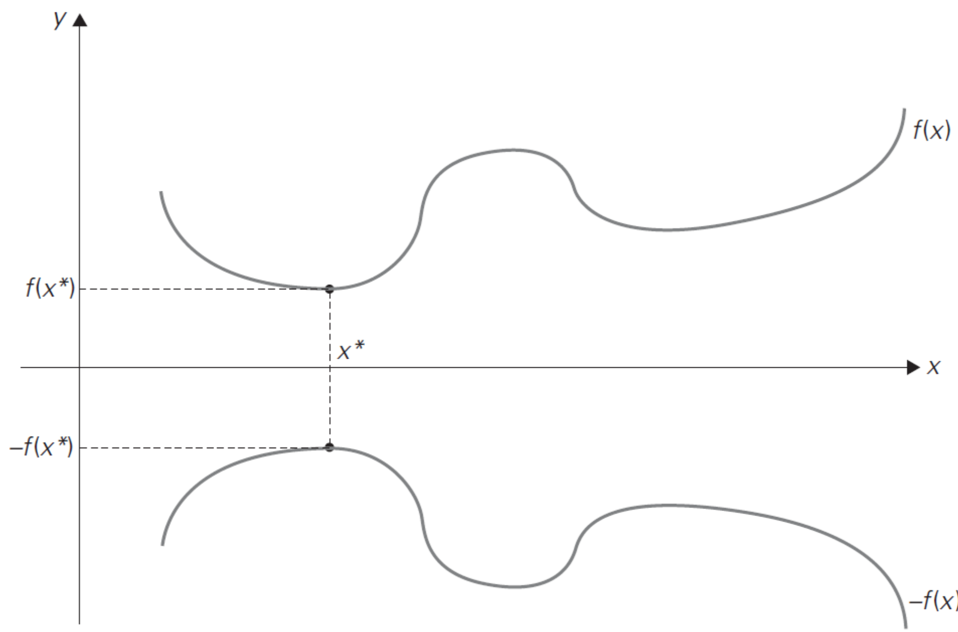
离散版本

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} v_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} w_j x_j \le W \\ & 0 \le x_j \le 1 \quad \text{for } j = 1, \ldots, n. \end{aligned}$$

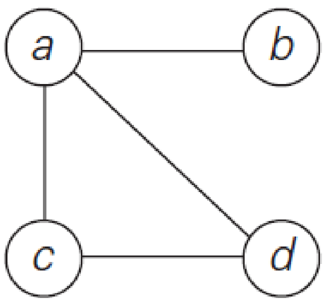
连续版本

课后阅读（3）：优化问题的化简

- 最小化问题与最大化问题的关系：
  min f(x) = -max[-f(x)]； max f(x) = -min[-f(x)]
- 函数最优化：把最优化问题转化为函数极值问题，再由f'(x)=0求临界点



课后阅读（4）：计算图中的路径数量



$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array} \qquad A^2 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{bmatrix} \end{array}$$

一个图，它的邻接矩阵 $\boldsymbol{A}$ 及其平方 $\boldsymbol{A}^2$ 分别指出了长度分别为 1 和 2 的路径的数量