北京理工大學 BEIJING INSTITUTEOF TECHNOLOGY

程序设计方法与实践

第6章:

变 治 法

主讲人：高广宇，李立杰

变治法

- **变治法是基于变换的思想**
  - “变”的阶段：出于这样或那样的原因，把问题的实例变得更容易求解
  - “治”的阶段：对实例进行求解
- **变治法的3种类型**
  - 变换为同问题的更简单或更方便的实例——称为**实例化简**
  - 变换为同样实例的不同表现——称之为**改变表现**
  - 变换为另问题实例，该问题的解法已知——称为**问题化简**

变治法

- 6.1 预排序(Presorting)
- 6.2 高斯消去法(Gaussian Elimination)
- 6.3 平衡查找树(Balanced Search Trees)
- 6.4 堆和堆排序(Heaps and Heapsort)
- 6.5 霍纳法则和二进制幂(Horner’s Rule)
- 6.6 问题简化(Problem Reduction)

预排序

- 预排序基于：如果列表有序的话，许多关于列表的问题更容易求解
  - 例1 检验数组中元素的唯一性
  - 例2 模式计算
  - 例3 查找问题

预排序

- 如果列表是有序的话，许多关于列表的问题更容易解决。
- 例1：检查数组中元素的唯一性(每个元素都不一样)
  - 蛮力法：2.3节（参见例2）
  - 对数组排序，再检查连续元素(有相同元素则一定相互挨着)

ALGORITHM *PresortElementUniqueness*$(A[0..n-1])$

```
//Solves the element uniqueness problem by sorting the array first
//Input: An array A[0..n − 1] of orderable elements
//Output: Returns "true" if A has no equal elements, "false" otherwise
sort the array A
for i ← 0 to n − 2 do
    if A[i] = A[i + 1] return false
return true
```

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

预排序

模式：给定数字列表中最经常出现的一个数值，称为模式！

例2 模式计算(Computing a mode)

蛮力法如何做？

蛮力法：扫描列表，计算所有不同值出现的频率

- 维护辅助列表存储已经出现的值和出现的次数。
- 该算法最差输入是没有相等元素的列表，第i个元素要和辅助列表的 $i-1$ 个元素比较后作为新元素加入辅助列表。

$$C(n)=\sum_{i=1}^{n}(i-1)=0+1+\cdots+(n-1)=\frac{(n-1)n}{2}\in\Theta(n^2)$$

预排序

北京理工大學

模式：给定数字列表中最经常出现的一个数值，称为模式！

预排序方法：

ALGORITHM *PresortMode*$(A[0..n-1])$

//Computes the mode of an array by sorting it first
//Input: An array $A[0..n-1]$ of orderable elements
//Output: The array's mode
sort the array A
$i \leftarrow 0$ //current run begins at position i
$modefrequency \leftarrow 0$ //highest frequency seen so far
while $i \le n-1$ **do**
  $runlength \leftarrow 1$; $runvalue \leftarrow A[i]$
  while $i + runlength \le n-1$ **and** $A[i + runlength] = runvalue$
    $runlength \leftarrow runlength + 1$
  if $runlength > modefrequency$
    $modefrequency \leftarrow runlength$; $modevalue \leftarrow runvalue$
  $i \leftarrow i + runlength$
return $modevalue$

预排序

- 例3 查找问题
  - 对于一个可排序项构成的查找问题，蛮力法在最坏情况下需要进行n次比较。
  - 如果先进行排序，就可利用二分查找，在最坏情况下需要进行$\lfloor \log_2 n \rfloor + 1$次比较。
  - $T(n)=T_{sort}(n)+T_{search}(n)=\Theta(n\log n)+\Theta(\log_2 n)=\Theta(n\log n)$
  - 这个效率要比顺序查找差
  - 但是要在同一个序列中查找多次，在排序上花的时间也是值得的。

预排序——本节习题

- 求一个n元数组的最大元素和最小元素的值.
  - a. 设计一个基于预排序的算法并确定它的效率类型。
  - b. 比较一下3种算法的效率：

    1）蛮力算法； 2）基于预排序的算法；3）分治算法。

解答提示

a. 排序数组，然后返回它的第一和最后元素。假设排序的效率是$O(nlogn)$, 则该算法效率。$O(nlogn) + \Theta(1) + \Theta(1) = O(nlogn)$

b. 蛮力和分治都是线性的，所以优于基于预排序的算法.

（Why?）

预排序——本节习题

- **用分治法求一个n元数组的最大元素和最小元素的值.**

```
算法 MaxMin(A[l..r],Max,Min)
 //该算法利用分治技术得到数组A中的最大值和最小值
//输入：数值数组A[l..r]
//输出：最大值Max和最小值Min
{if(r=l) Max←A[l]; Min←A[l]; //只有一个元素时
 else
     if r－l=1  //有两个元素时
         if A[l]≤A[r]
            Max←A[r]; Min←A[l]
         else
            Max←A[l]; Min←A[r]
     else  //r－l>1
         MaxMin(A[l,(l+r)/2],Max1,Min1); //递归解决前一部分
         MaxMin(A[(l+r/)2..r],Max2,Min2); //递归解决后一部分
         if Max1＜Max2   Max= Max2   //从两部分的两个最大值中选择大值
         if Min2<Min1  Min=Min2; //从两部分的两个最小值中选择小值
}
```

预排序——本节习题

- 用分治法求一个n元数组的最大元素和最小元素的值.

假设 $n=2^k$,比较次数的递推关系式:

$$C(n)=2C(n/2)+2 \qquad \text{for } n>2$$

$$C(1)=0, \qquad C(2)=1$$

$C(n)=C(2^k)=2C(2^{k-1})+2$

$=2[2C(2^{k-2})+2]+2$

$=2^2C(2^{k-2})+2^2+2$

$=2^2[2C(2^{k-3})+2]+2^2+2$

$=2^3C(2^{k-3})+2^3+2^2+2$

...

$=2^{k-1}C(2)+2^{k-1}+2^{k-2}+...+2$ //C(2)=1

$=2^{k-1}+2^{k-1}+2^{k-2}+...+2$ //后面部分为等比数列求和

$=2^{k-1}+2^k-2$ //$2(^{k-1})=n/2,2^k=n$

$=n/2+n-2$

$=3n/2-2$

预排序——本节习题

- **用蛮力法求一个n元数组的最大元素和最小元素的值.**

```
算法 simpleMaxMin(A[l..r])
//用蛮力法得到数组A最大值和最小值
//输入：数值数组A[l..r]
//输出：最大值Max和最小值Min
Max=Min=A[l];
for i=l+1 to r do
if A[i]>Max  Max←A[i];
else if A[i]<Min Min←A[i]
return Max, Min
}
```

蛮力法时间复杂度
$n-1$，属于$\Theta(n)$;

算法MaxMin（分治法）时间复杂度为
$3n/2-2$，属于$\Theta(n)$;

但比较一下发现，
MaxMin的速度要比simpleMaxMin快!

预排序——本节习题

1. 考虑这样一个问题：它要找出 n 个数字构成的一个数组中两个最接近数的距离(两个数 x 和 y 之间的距离定义为 $|x-y|$)。

   a. 设计一个基于预排序的算法来解该问题并确定它的效率类型。

   b. 将该算法的效率类型和蛮力算法的效率类型进行比较(参见习题 1.2 中的第 9 题)。

预排序——本节习题

2. 假设 $A=\{a_1,\cdots,a_n\}$ 和 $B=\{b_1,\cdots,b_m\}$ 是两个数字集合。考虑一下对它们求交集的问题，也就是说，集合 C 中的所有数字都是既属于 A 又属于 B 的。

   a. 设计一个蛮力算法来解该问题并确定它的效率类型。

   b. 设计一个基于预排序的算法来解该问题并确定它的效率类型。

预排序——本节习题

4. 请估计一下，如果用合并排序做预排序，用折半查找做查找，要做多少次查找才能使得对一个由 10^3 个元素构成的数组所做的预排序是有意义的(我们可以假设，所要查找的都是数组中的元素)。如果是一个由 10^6 个元素构成的数组呢？

预排序——本节习题

4. 请估计一下，如果用合并排序做预排序，用折半查找做查找，要做多少次查找才能使得对一个由 10^3 个元素构成的数组所做的预排序是有意义的(我们可以假设，所要查找的都是数组中的元素)。如果是一个由 10^6 个元素构成的数组呢？

设k为所需的最小查找次数，以便比按顺序搜索（平均成功搜索）进行的比较次数更少。假设排序算法平均进行$n\log n$比较，并使用折半查找（比较次数约$\log_2 n$）和顺序查找（约$n/2$），得到如下不等式

$$\boldsymbol{n\log_2 n + k\log_2 n \le \frac{kn}{2}}$$

因此，

$$\boldsymbol{k \ge \frac{n\log_2 n}{\frac{n}{2} - \log_2 n}}$$

如果$\boldsymbol{n = 10^3}$，则$\boldsymbol{k_{min} = 21}$，如果$\boldsymbol{n = 10^6}$，则$\boldsymbol{k_{min} = 40}$.

变治法

- 6.1 预排序(Presorting)
- 6.2 高斯消去法(Gaussian Elimination)
- 6.3 平衡查找树(Balanced Search Trees)
- 6.4 堆和堆排序(Heaps and Heapsort)
- 6.5 霍纳法则和二进制幂(Horner's Rule)
- 6.6 问题简化(Problem Reduction)

高斯消去法

➢ 需要解一个包含n个方程的n元联立方程组：

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

高斯消去法

- 算法思路：把n个线性方程构成的n元联立方程组变换成一个等价的方程组，该方程组有一个**上三角的系数矩阵**，这种矩阵的主对角线下方元素全部为0

高斯消去法

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

$$\begin{cases} a_{11}'x_1 + a_{12}'x_2 + \ldots + a_{1n}'x_n = b_1' \\ \qquad a_{22}'x_2 + \ldots + a_{2n}'x_n = b_2' \\ \qquad \vdots \\ \qquad a_{nn}'x_n = b_n' \end{cases}$$

$$A' = \begin{bmatrix} a_{11}' & a_{12}' & \cdots & a_{1n}' \\ 0 & a_{22}' & \cdots & a_{2n}' \\ & & \vdots & \\ 0 & 0 & \cdots & a_{nn}' \end{bmatrix}$$

$$\boxed{Ax = b \Rightarrow A'x = b'}$$

高斯消去法

- **为什么具有上三角系数矩阵的方程组要好于任意系数矩阵的方程组呢?**
  - 可以对具有上三角系数矩阵的方程组进行变换:
    - 从最后一个方程中可以立即求出x_n的值
    - 然后把这个值代入倒数第二个方程求出x_{n-1}
    - 依此类推，直到求出x_1的值

高斯消去法

北京理工大学

➤ 例 用高斯消去法解方程组

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

矩阵A为:

$$A = \begin{bmatrix} 2 & -1 & 1 & 1 \\ 4 & 1 & -1 & 5 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

$$A \Rightarrow \begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & \frac{3}{2} & \frac{1}{2} & -\frac{1}{2} \end{bmatrix} \Rightarrow \begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 0 & 2 & -2 \end{bmatrix}$$

高斯消去法

北京理工大学

ALGORITHM *GaussElimination(*

//Applies Gaussian elimination

//augmented with vector b of th

//Input: Matrix $A[1..n, 1,...n]$ an

//Output: An equivalent upper e

//corresponding right-hand side values in the $(n+1)$st column

```
for i ← 1 to n do A[i, n+1] ← b[i]   //augments the matrix
for i ← 1 to n − 1 do
    for j ← i + 1 to n do
        for k ← i to n + 1 do
            A[j, k] ← A[j, k] − A[i, k] ∗ A[j, i] / A[i, i]
```

$$
\begin{cases}
a_{11}x + a_{12}x + \quad + a_{1n}x_n = b_1 \\
a_{22}x + \quad + a_{2n}x_n = b_2 \\
\vdots \\
a_{nn}x_n = b_n
\end{cases}
$$

缺点：$A[i,i]=0$；$A[i,i]$非常小，导致$A[j,i]/A[i,i]$非常大，$A[j,k]$的新值会有舍入误差；内层循环效率非常低。

高斯消去法——改进

ALGORITHM *BetterGaussElimination*$(A[1..n, 1..n], b[1..n])$

```
//Implements Gaussian elimination with partial pivoting
//Input: Matrix A[1..n, 1,..n] and column-vector b[1..n]
//Output: An equivalent upper-triangular matrix in place of A and the
//corresponding right-hand side values in place of the (n + 1)st column
for i ← 1 to n do A[i, n + 1] ← b[i]   //appends b to A as the last column
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
```

部分选主元法

每次寻找第 i 列系数的绝对值最大的行，把它作为第 i 次迭代的基点。保证比例因子的绝对值永远不会大于1。

高斯消去法

北京理工大学

➢ 算法效率

高斯消去的第2阶段：反向替换的效率属于$\Theta(n^2)$

- 最内层循环只有一行语句：

$$A[j,k] \leftarrow A[j,k] - A[i,k] * temp$$

$$C(n)=\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\sum_{k=i}^{n+1}1=\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}(n+1-i+1)=\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}(n+2-i)$$

$$=\sum_{i=1}^{n-1}(n+2-i)(n-(i+1)+1)=\sum_{i=1}^{n-1}(n+2-i)(n-i)$$

$$=(n+1)(n-1)+n(n-2)+\cdots+3\times 1$$

$$=\sum_{j=1}^{n-1}(j+2)j=\sum_{j=1}^{n-1}j^2+\sum_{j=1}^{n-1}2j=\frac{(n-1)n(2n-1)}{6}+2\frac{(n-1)n}{2}$$

$$=\frac{n(n-1)(2n+5)}{6}\approx\frac{1}{3}n^3\in\Theta(n^3)$$

高斯消去法

- 习题6.2-4 判断

$$
\begin{aligned}
C(n) &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\sum_{k=i}^{n+1}1 = \sum_{i=1}^{n-1}(n+2-i)(n-i) \\
&= \sum_{i=1}^{n-1}[(n+2)n - i(2n+2) + i^2] \\
&= \sum_{i=1}^{n-1}(n+2)n - \sum_{i=1}^{n-1}(2n+2)i + \sum_{i=1}^{n-1} i^2.
\end{aligned}
$$

Since $s_1(n) = \sum_{i=1}^{n-1}(n+2)n \in \Theta(n^3)$, $s_2(n) = \sum_{i=1}^{n-1}(2n+2)i \in \Theta(n^3)$, and $s_3(n) = \sum_{i=1}^{n-1} i^2 \in \Theta(n^3)$, $s_1(n) - s_2(n) + s_3(n) \in \Theta(n^3)$.

高斯消去法

习题6.2-4（解答）

- **不正确。** $f_1(n) \in \Theta(n^3)$, $f_2(n) \in \Theta(n^3)$, and $f_3(n) \in \Theta(n^3)$

  并不意味着：$f_1(n) - f_2(n) + f_3(n) \in \Theta(n^3)$,

  例如：$f_1(n) = n^3 + n$, $f_2(n) = 2n^3$, and $f_3(n) = n^3$.

  相减的结果：$f_1(n) - f_2(n) + f_3(n) = n \in \Theta(n)$.

高斯消去法

北京理工大学

➢ 习题6.2-5

反向替换

Algorithm $GaussBackSub(A[1..n, 1..n+1])$
//Implements the backward substitution stage of Gaussian elimination
//by solving a given system with an upper-triangular coefficient matrix
//Input: Matrix $A[1..n, 1, ..n+1]$, with the first n columns in the upper-
//triangular form
//Output: A solution of the system of n linear equations in n unknowns
//whose coefficient matrix and right-hand side are the first n columns
//of A and its $(n+1)$st column, respectively
for $i \leftarrow n$ **downto** 1 **do**
    $temp \leftarrow 0.0$
    for $j \leftarrow n$ **downto** $i+1$
        $temp \leftarrow temp + A[i, j] * x[j]$
    $x[i] \leftarrow (A[i, n+1] - temp)/A[i, i]$

高斯消去法

- **习题6.2-5（解答）**
  - 基本操作是乘法，算法效率为：

$$
\begin{aligned}
M(n) &= \sum_{i=1}^{n}\sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n}(n-(i+1)+1) = \sum_{i=1}^{n}(n-i) \\
&= (n-1)+(n-2)+\ldots+1 = \frac{(n-1)n}{2} \in \Theta(n^2).
\end{aligned}
$$

高斯消去法——总结

北京理工大学

- 不难看出高斯消去法的时间复杂度为$\Theta(n^3)$
- （反向替换属于$\Theta(n^2)$）
- 理论上来说，高斯消去法要么在一个线性方程组有唯一解时生成它的精确解，要么确定该方程组不存在这样的解(要么无解，要么有无穷多个解)。
- 在实践中，用该方法最主要的困难在于如何防止舍入误差的累积。

变治法

北京理工大学

- 6.1 预排序(Presorting)
- 6.2 高斯消去法(Gaussian Elimination)
- 6.3 平衡查找树(Balanced Search Trees)
- 6.4 堆和堆排序(Heaps and Heapsort)
- 6.5 霍纳法则和二进制幂(Horner's Rule)
- 6.6 问题简化(Problem Reduction)

平衡查找树

- 二叉查找树是一种实现字典的重要数据结构
- 二叉查找树的节点所包含的元素来自可排序项的集合。每个节点一个元素，并使得所有左子树中的元素都小于子树根节点的元素，而所有右子树的元素都大于它。
- 把一个集合变换成一棵二叉查找树，就是典型的“改变表现”技术
- 将之与字典的简单实现（如数组）有何优势?

平衡查找树

- **二叉查找树对于查找，插入和删除的时间效率都属于$\Theta(logn)$，但这仅在平均效率下成立，最差情况退化为$\boldsymbol{\Theta(n)}$，因为树可能是严重不平衡的树。**
  - “实例化简”：把一棵不平衡的二叉查找树转变为平衡的形式。如AVL树，红黑树
  - “改变表现”：它允许一个查找树的单个节点中不止包含一个元素。如2-3树，2-3-4树，B树

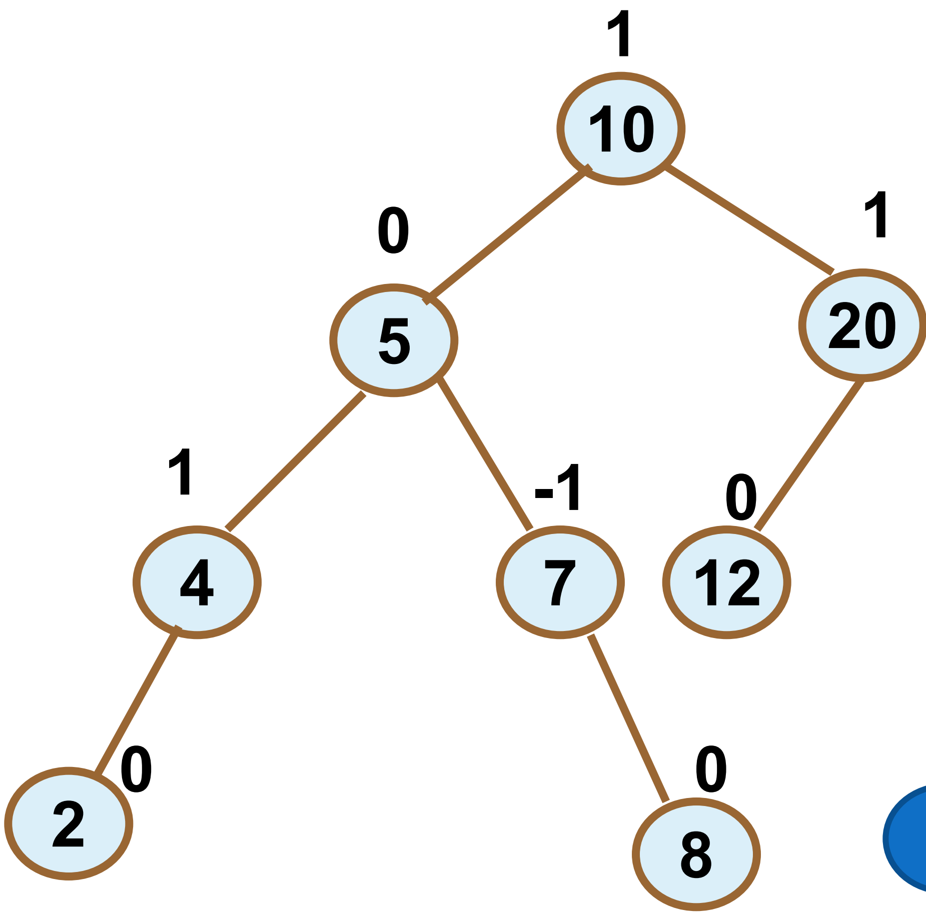
2-3树或B树与平衡二叉查找树的区别在于查找树的单个节点中能够容纳的元素个数

红黑树能够容忍同一节点的一棵子树的高度是另一棵子树的两倍

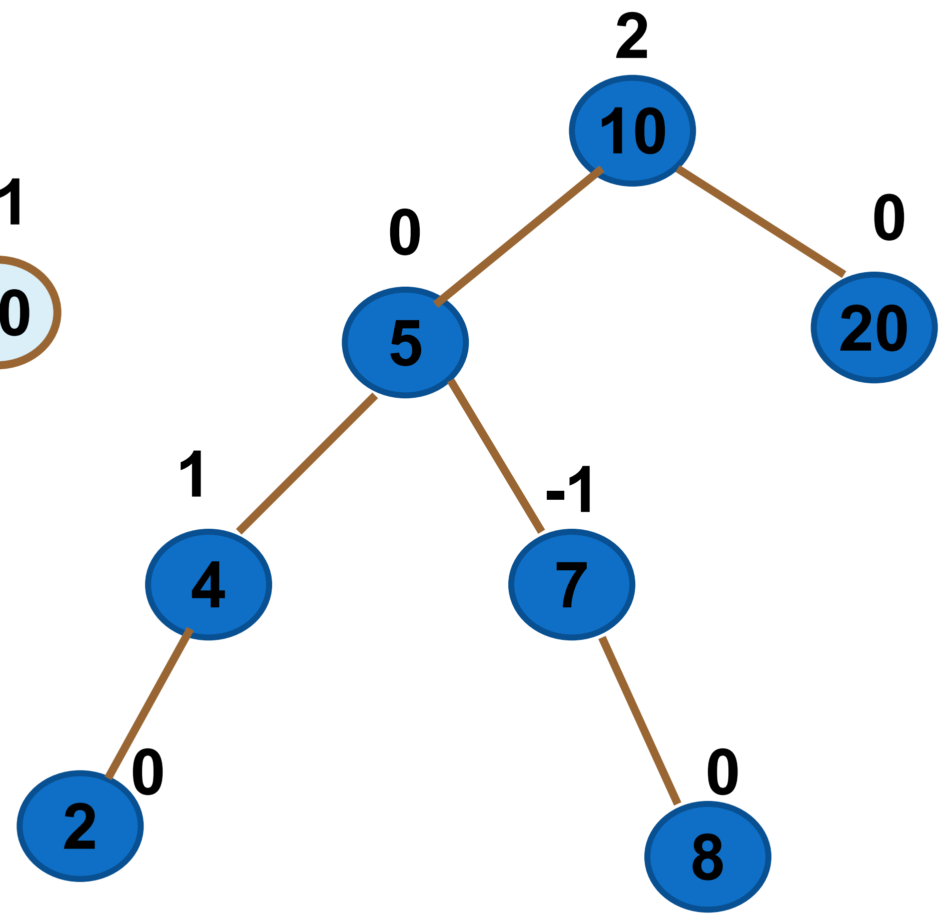
平衡查找树——AVL树

- AVL树是**一棵二叉查找树**，其中每个节点的**平衡因子定义为**该节点左子树和右子树的高度差，这个平衡因子要么为0，要么为+1或者-1(一棵空树的高度定义为-1)
- 如果插入一个新节点使得一个AVL树失去平衡，我们用**旋转**对这棵树做一个变换

平衡查找树——AVL树

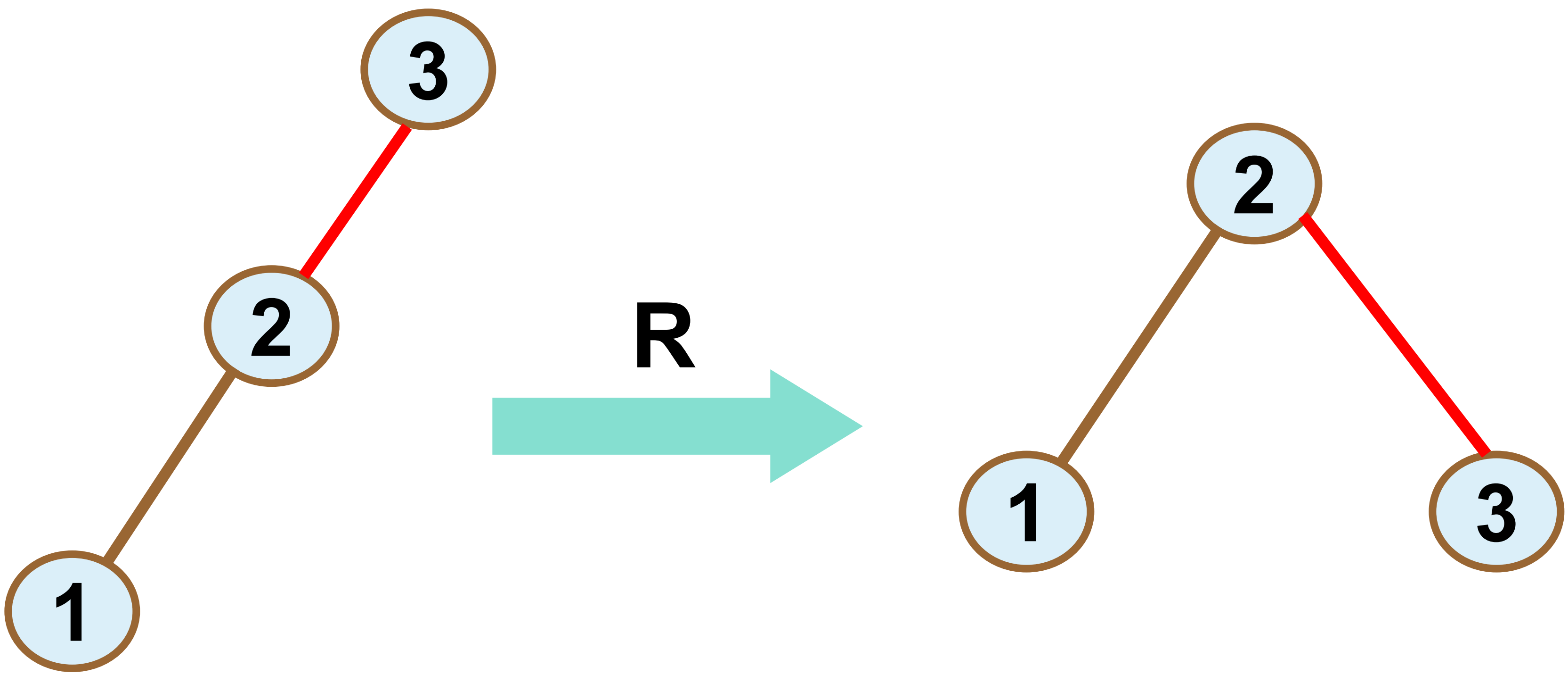


AVL树



非AVL树

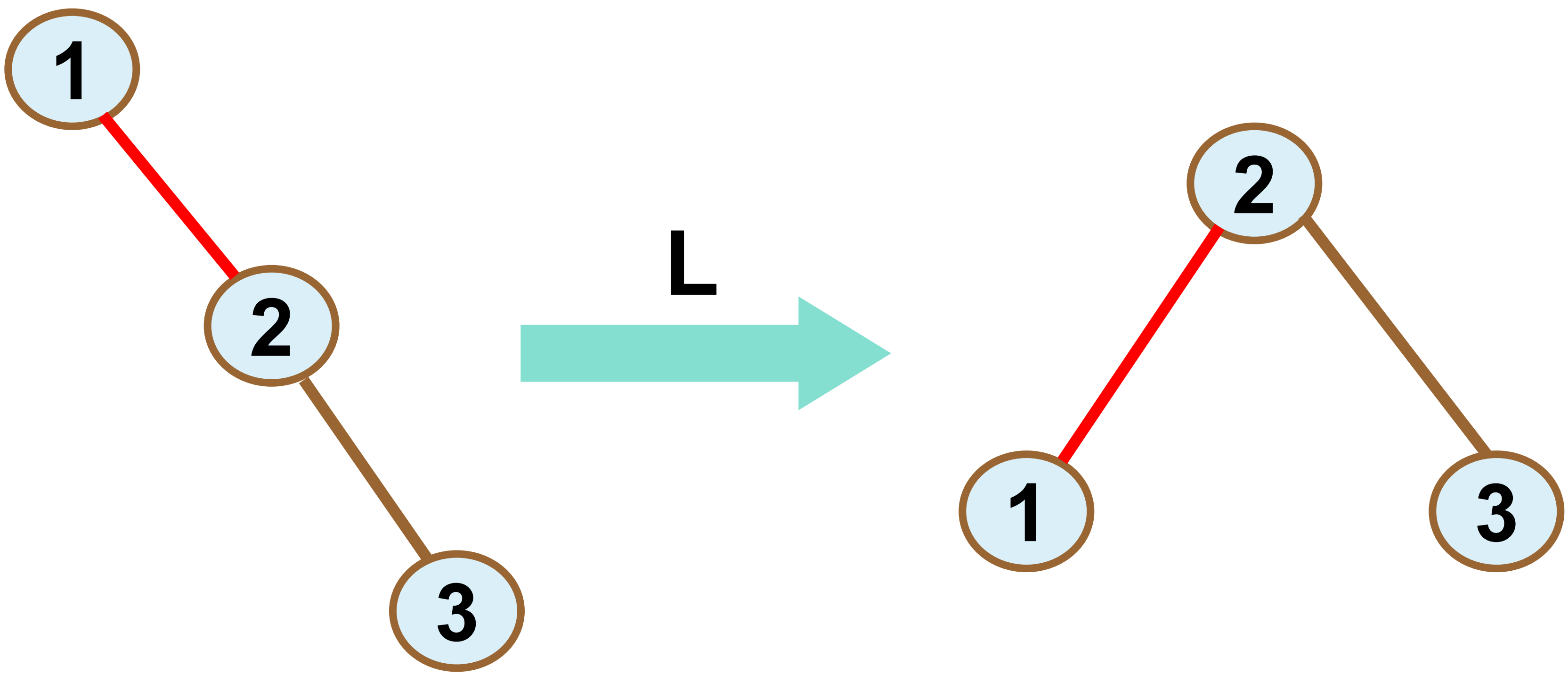
连接根节点和它左子女的边向右旋转



平衡查找树——AVL树

北京理工大学

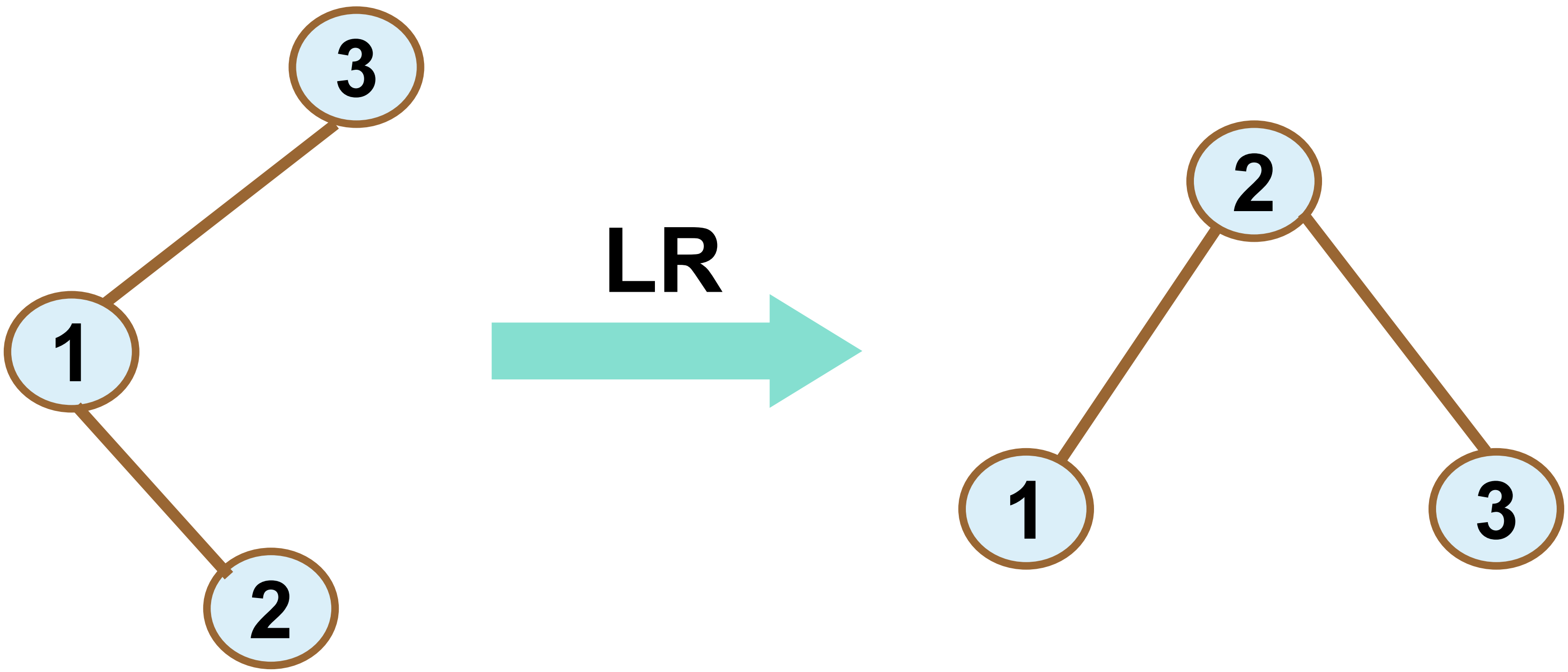
连接根节点和它右子女的边向左旋转



平衡查找树——AVL树

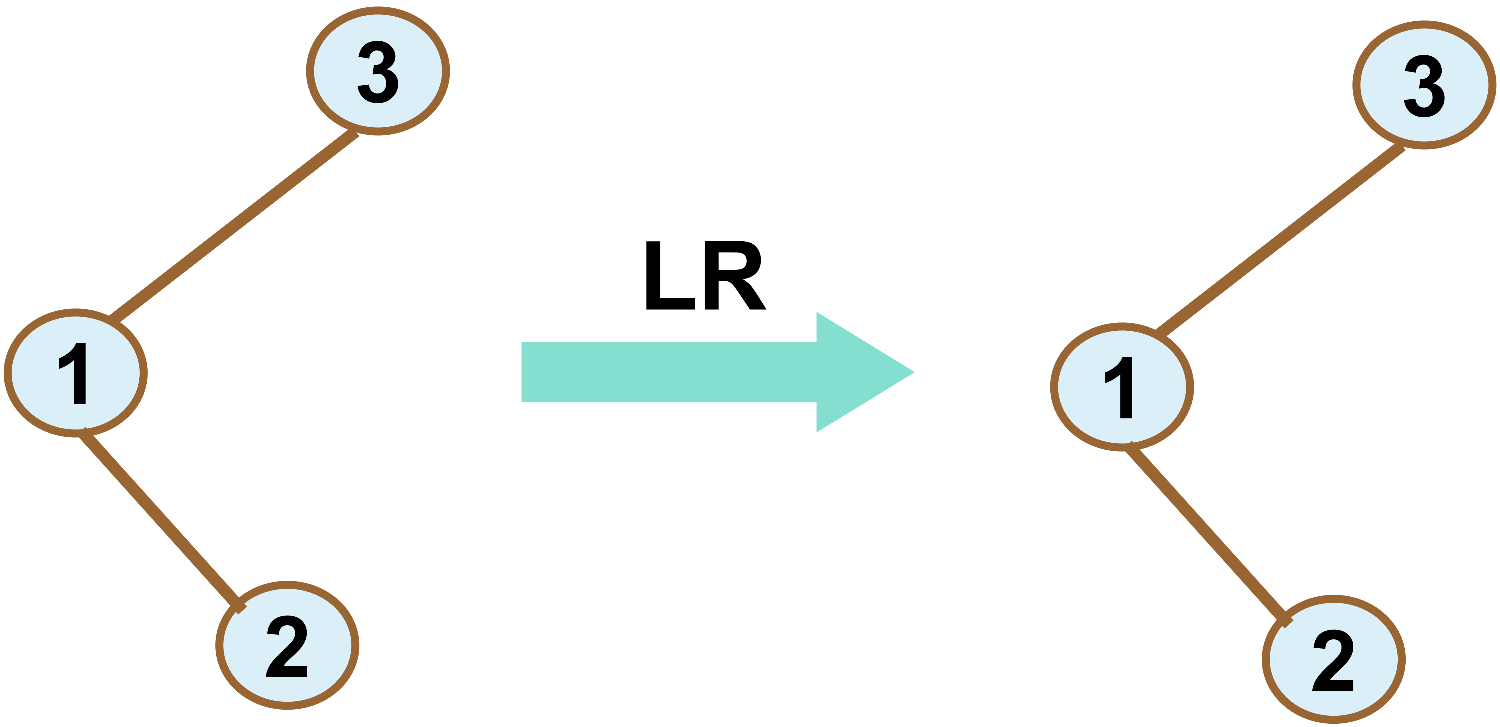
北京理工大学

左右双转：对根r的左子树进行左旋，在对这颗以r为根的新树进行右旋！

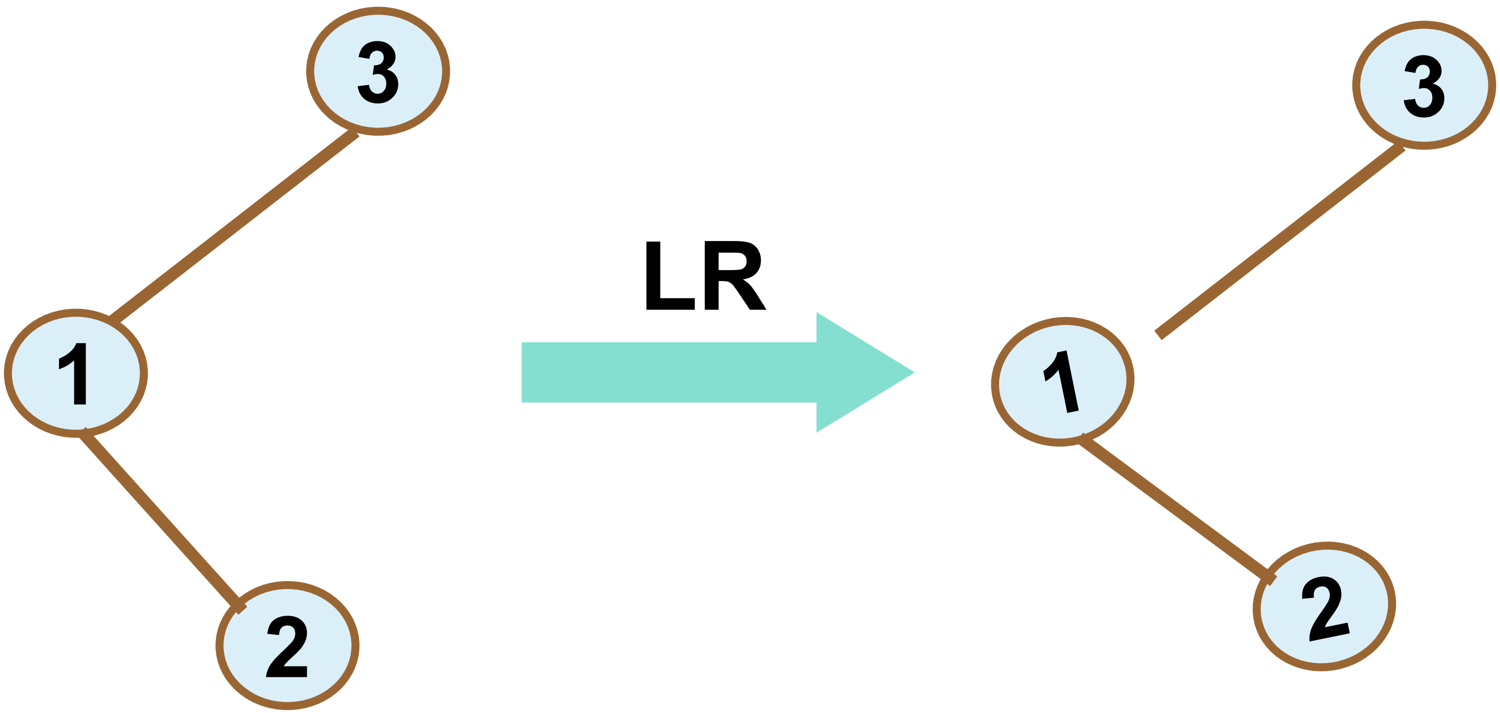


平衡查找树——AVL树

左右双转：对根r的左子树进行左旋，在对这颗以r为根的新树进行右旋！

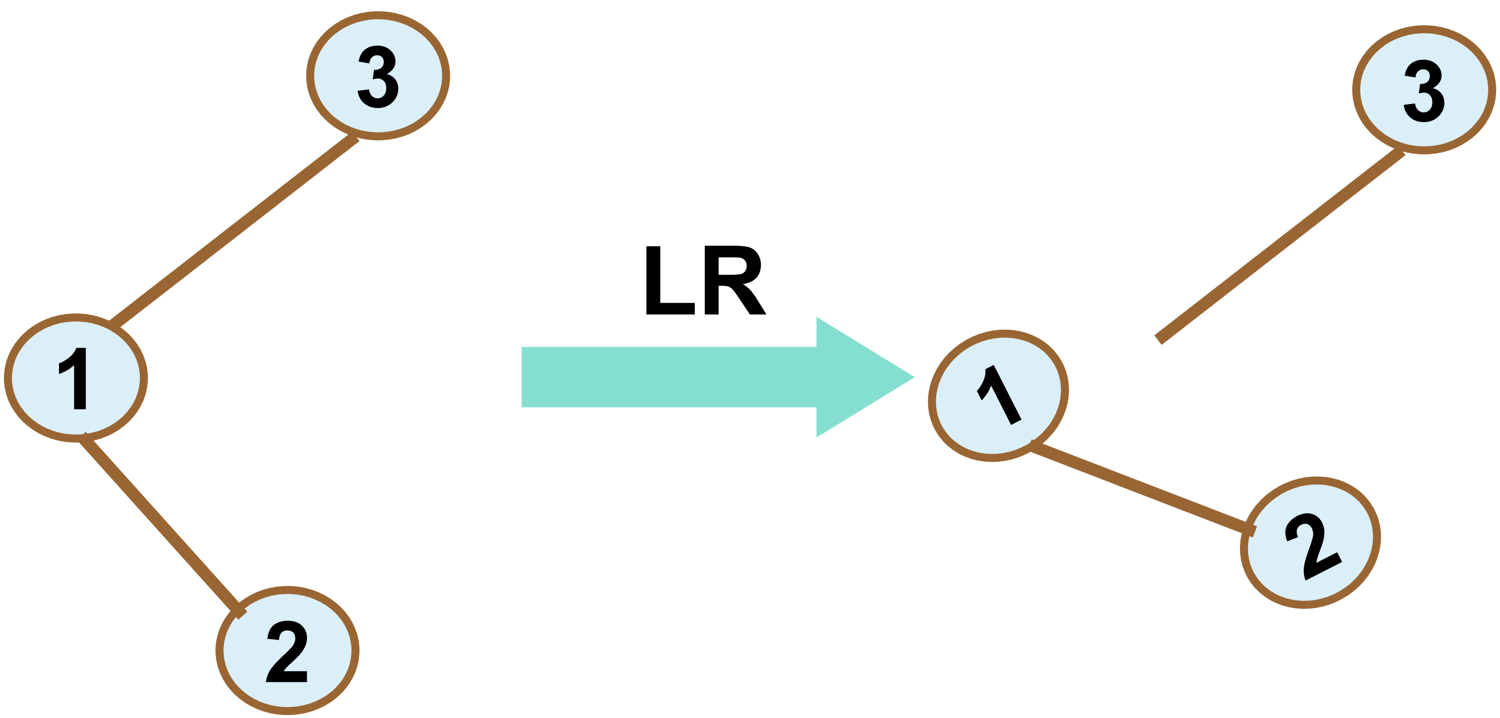


左右双转：对根r的左子树进行左旋，在对这颗以r为根的新树进行右旋！



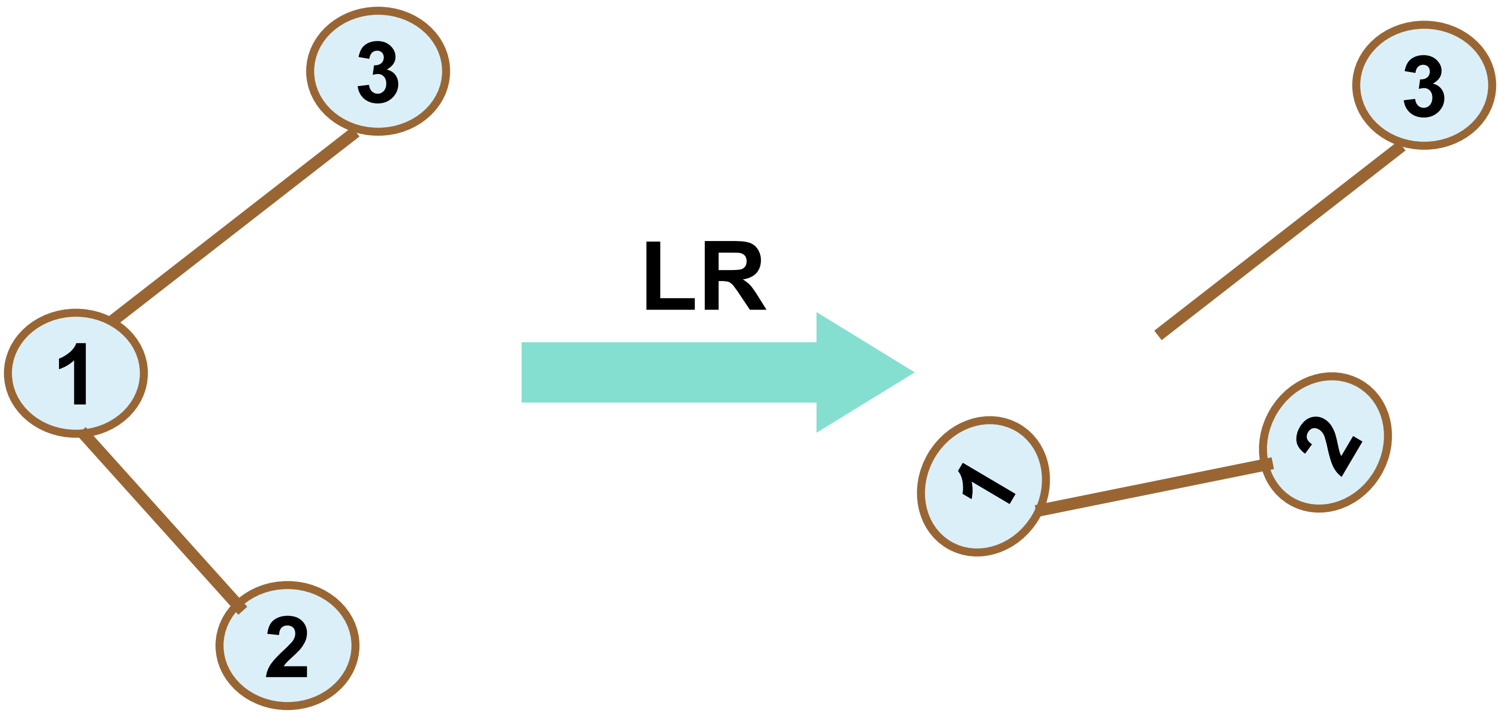
平衡查找树——AVL树

左右双转：对根r的左子树进行左旋，在对这颗以r为根的新树进行右旋！



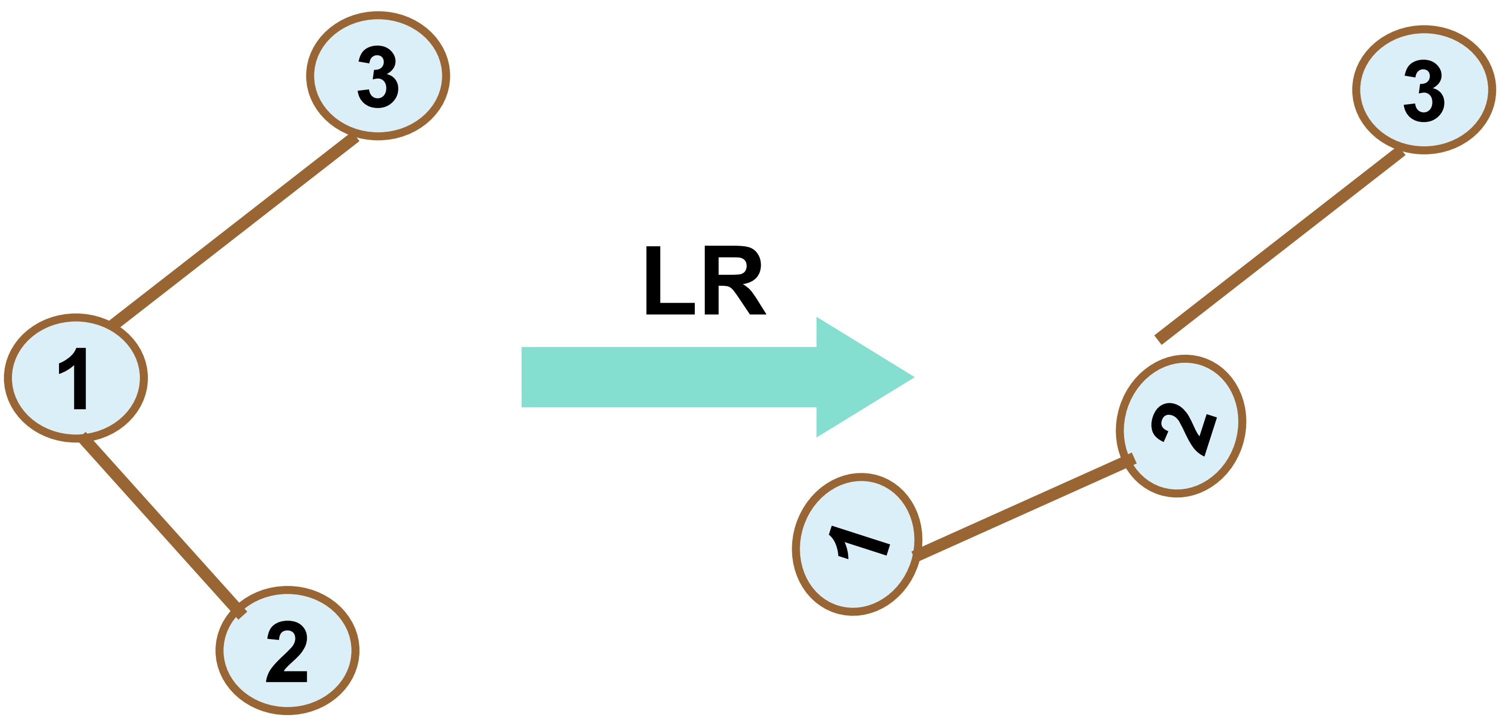
平衡查找树——AVL树

左右双转：对根r的左子树进行左旋，在对这颗以r为根的新树进行右旋！



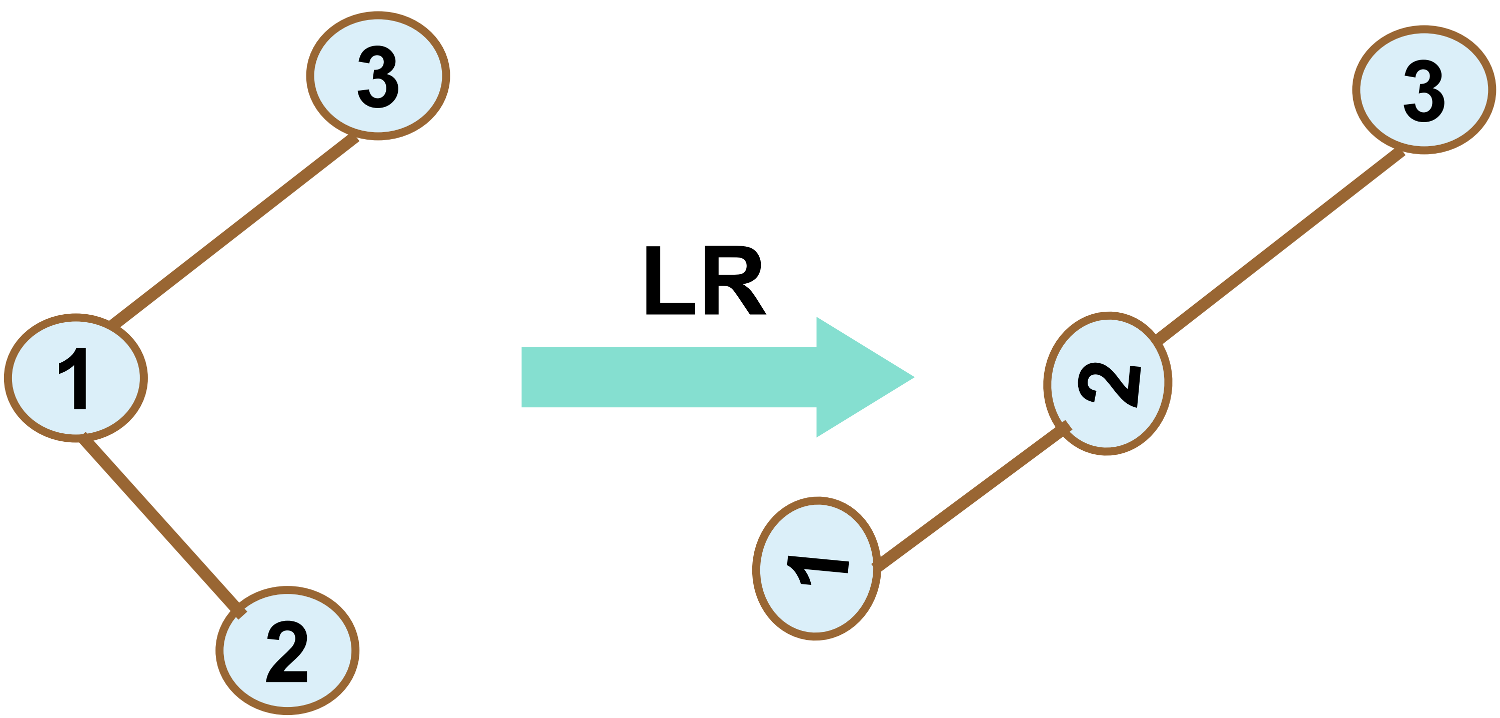
平衡查找树——AVL树

左右双转：对根r的左子树进行左旋，在对这颗以r为根的新树进行右旋！



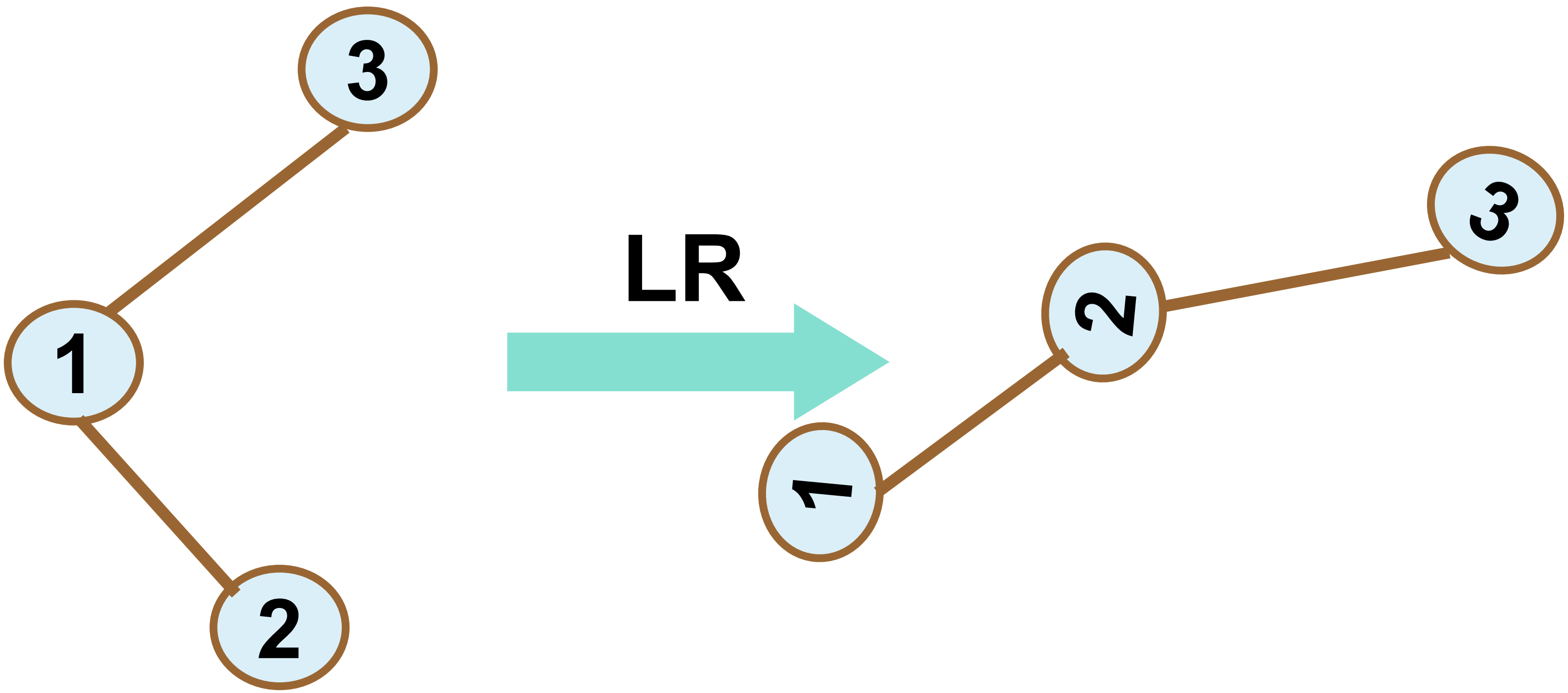
平衡查找树——AVL树

左右双转：对根r的左子树进行左旋，在对这颗以r为根的新树进行右旋！



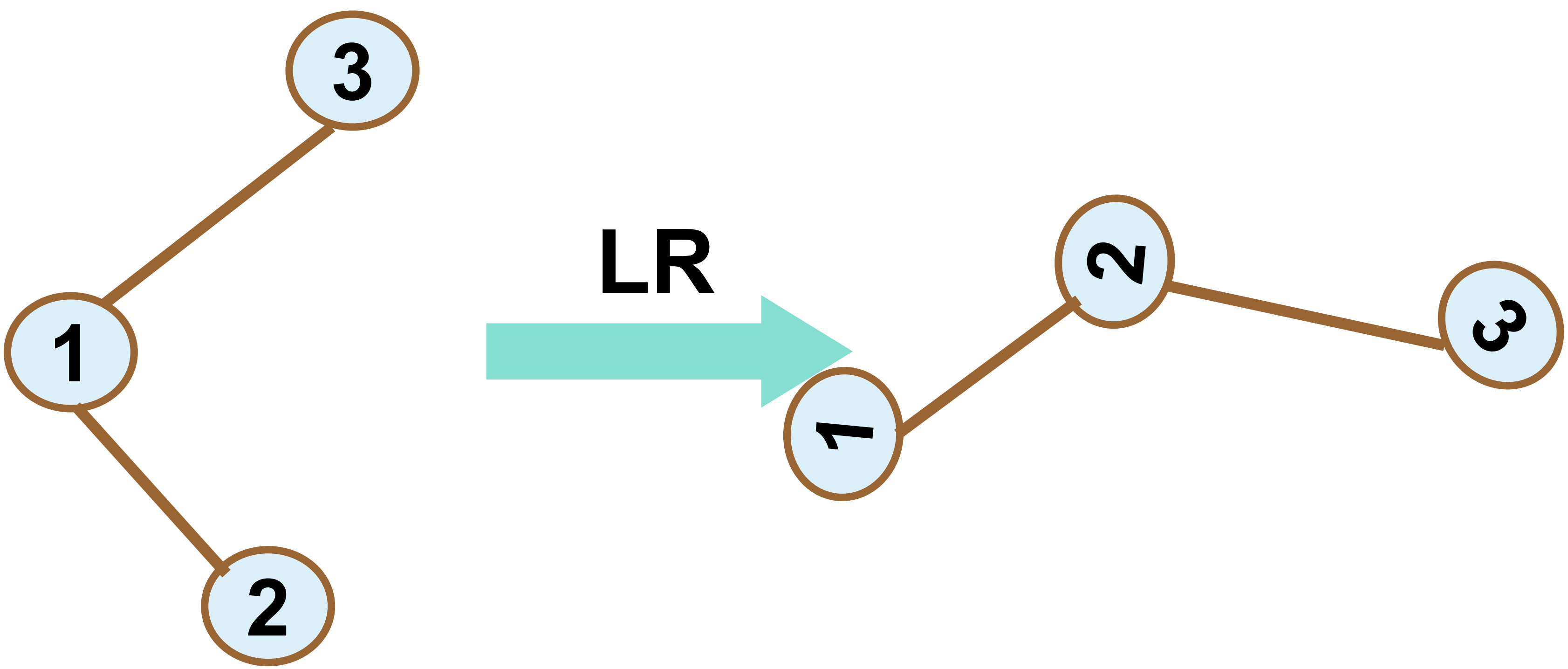
平衡查找树——AVL树

左右双转：对根r的左子树进行左旋，在对这颗以r为根的新树进行右旋！



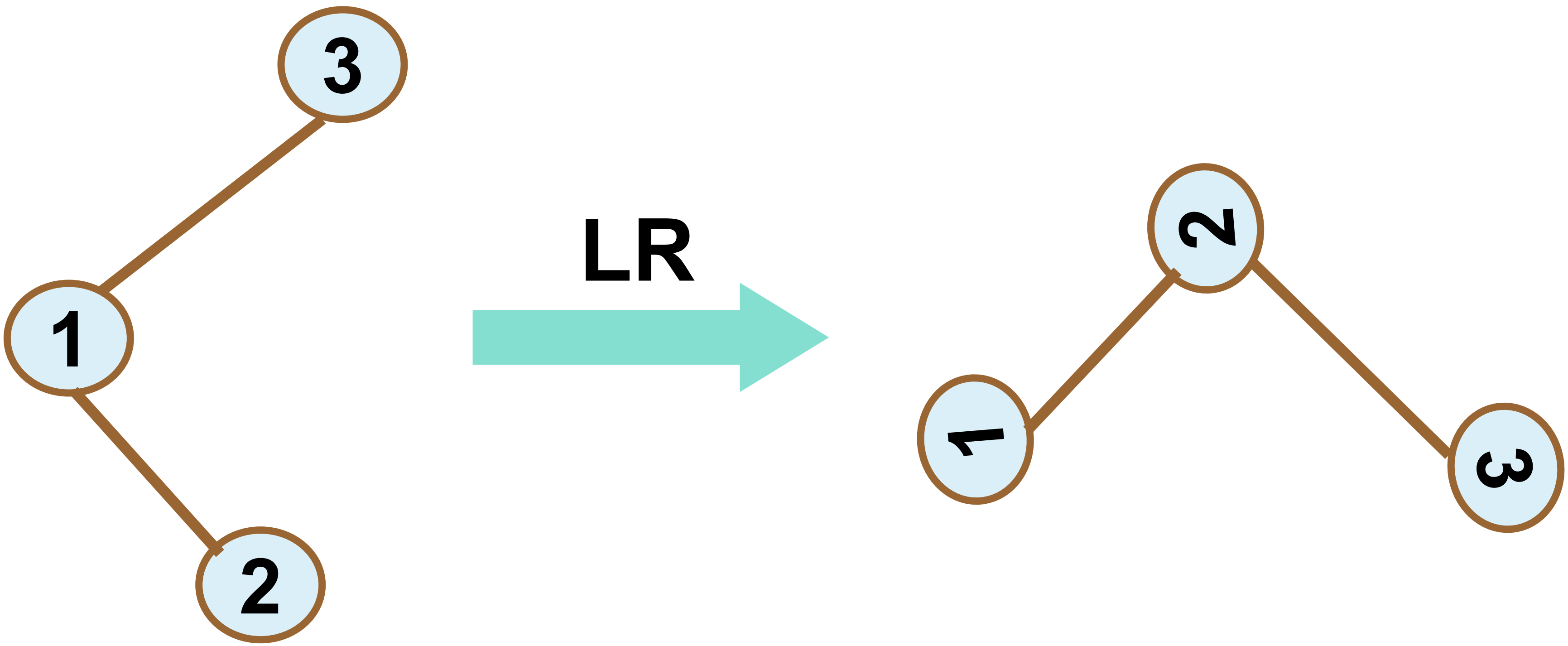
平衡查找树——AVL树

左右双转：对根r的左子树进行左旋，在对这颗以r为根的新树进行右旋！



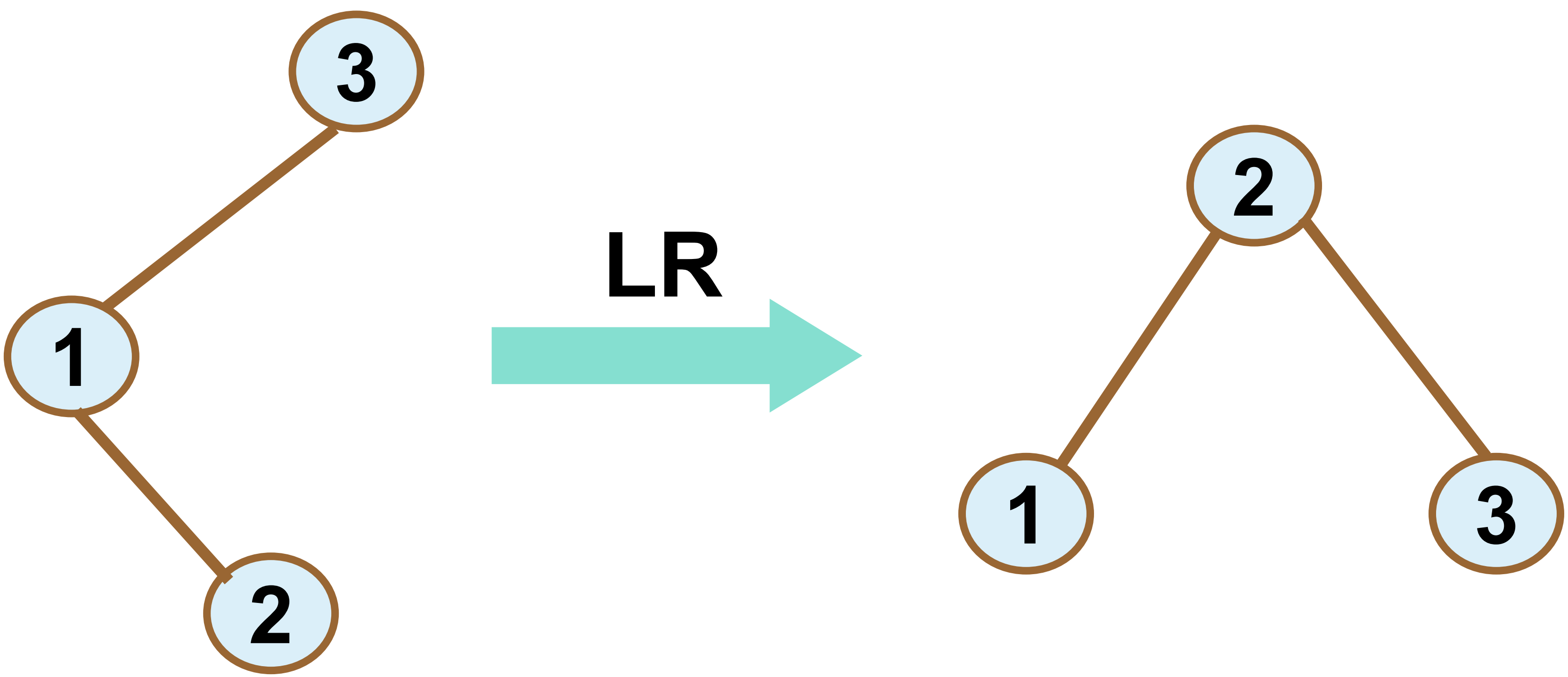
平衡查找树——AVL树

左右双转：对根r的左子树进行左旋，在对这颗以r为根的新树进行右旋！



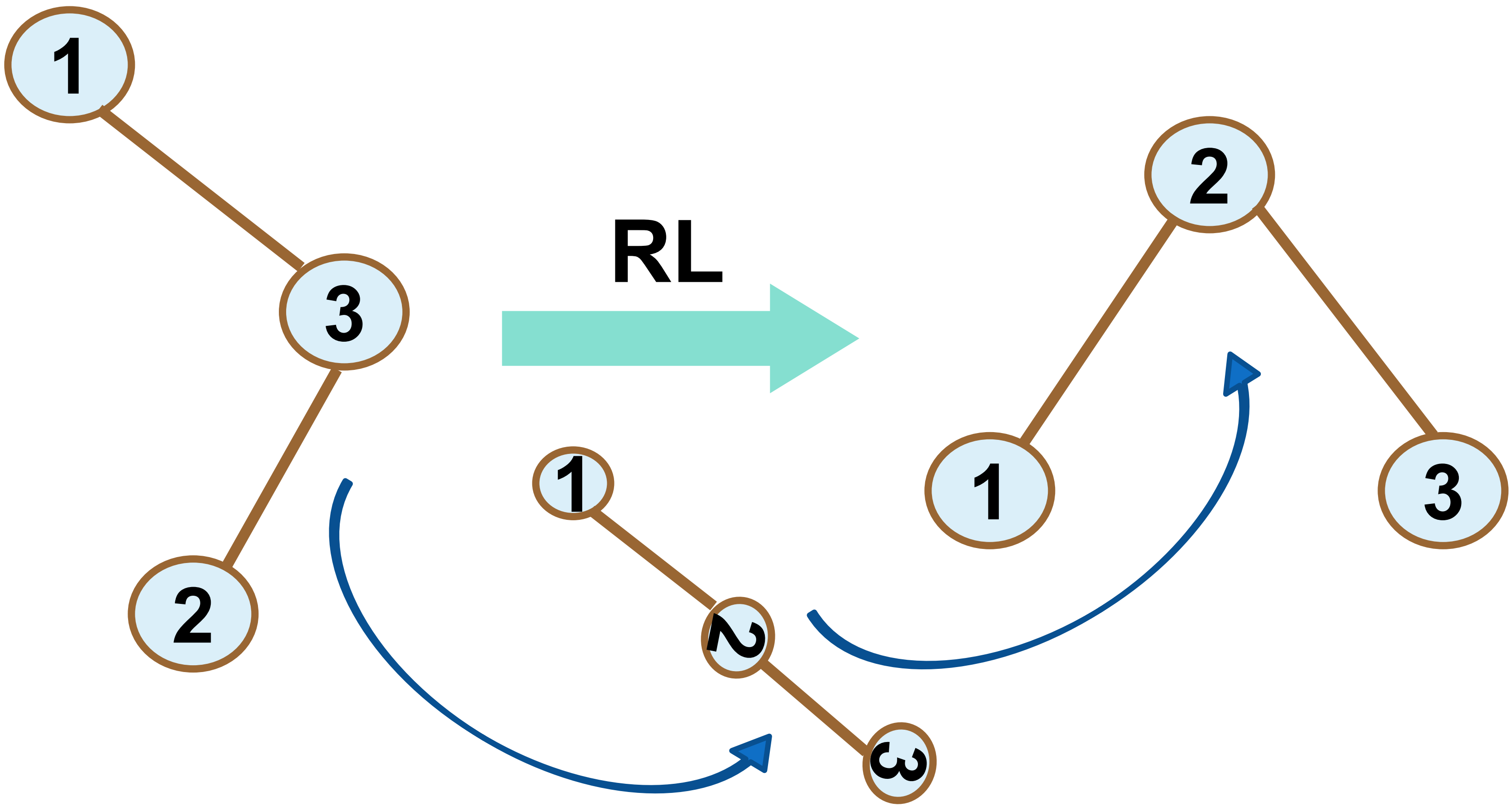
平衡查找树——AVL树

左右双转：对根r的左子树进行左旋，在对这颗以r为根的新树进行右旋！

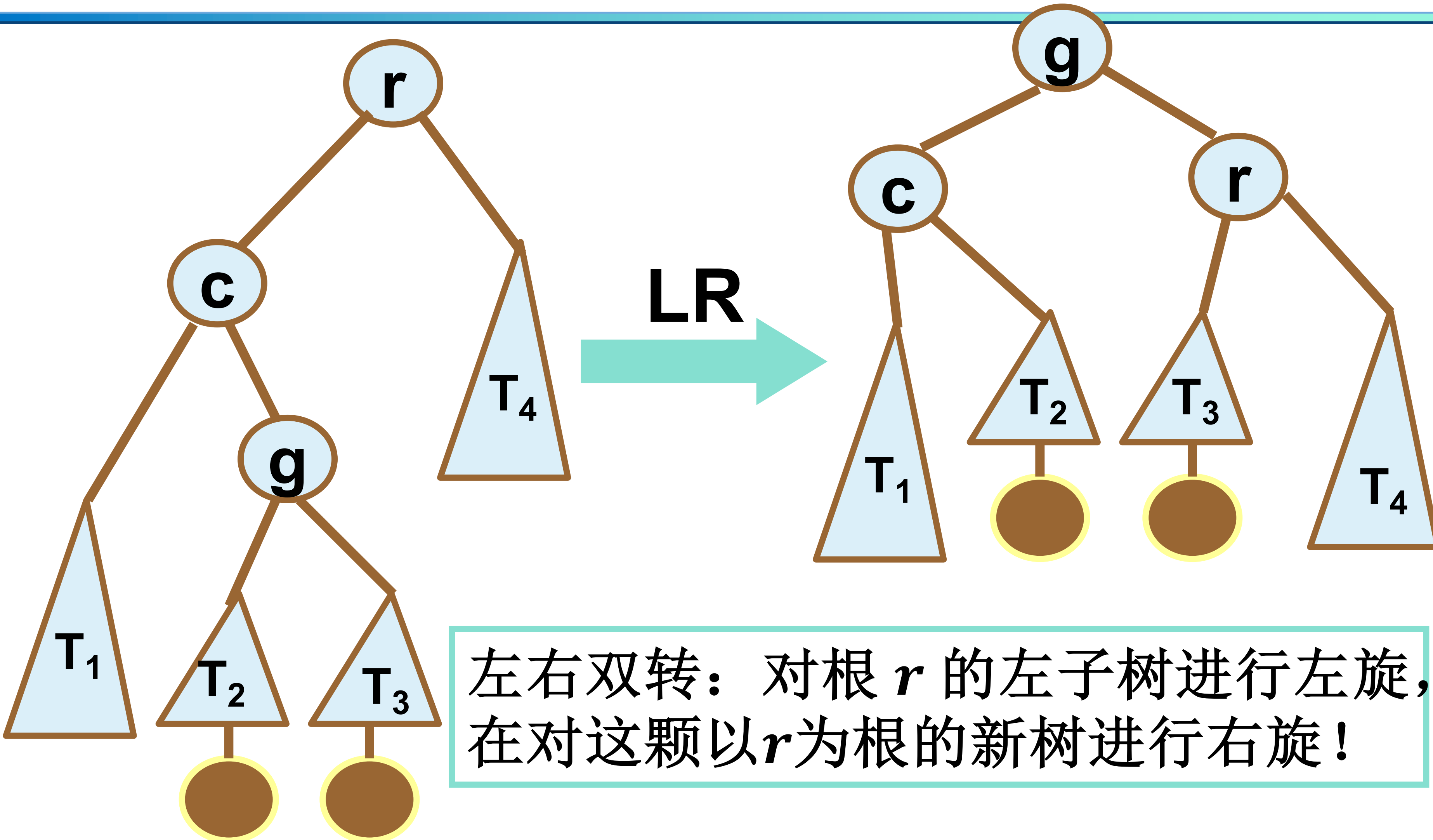


平衡查找树——AVL树

左右双转：对根r的右子树进行右旋，在对这颗以r为根的新树进行左旋！

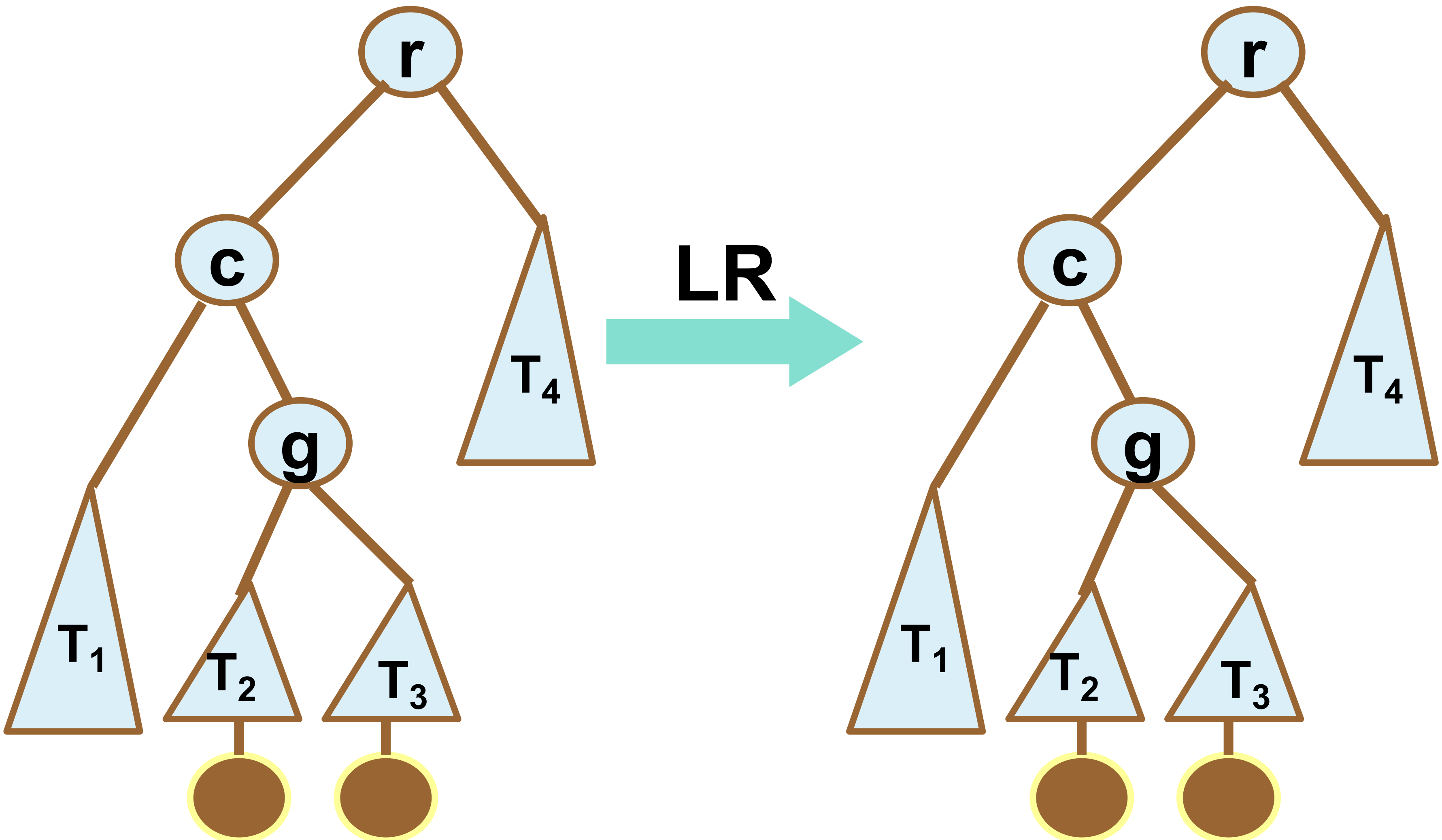


平衡查找树——AVL树

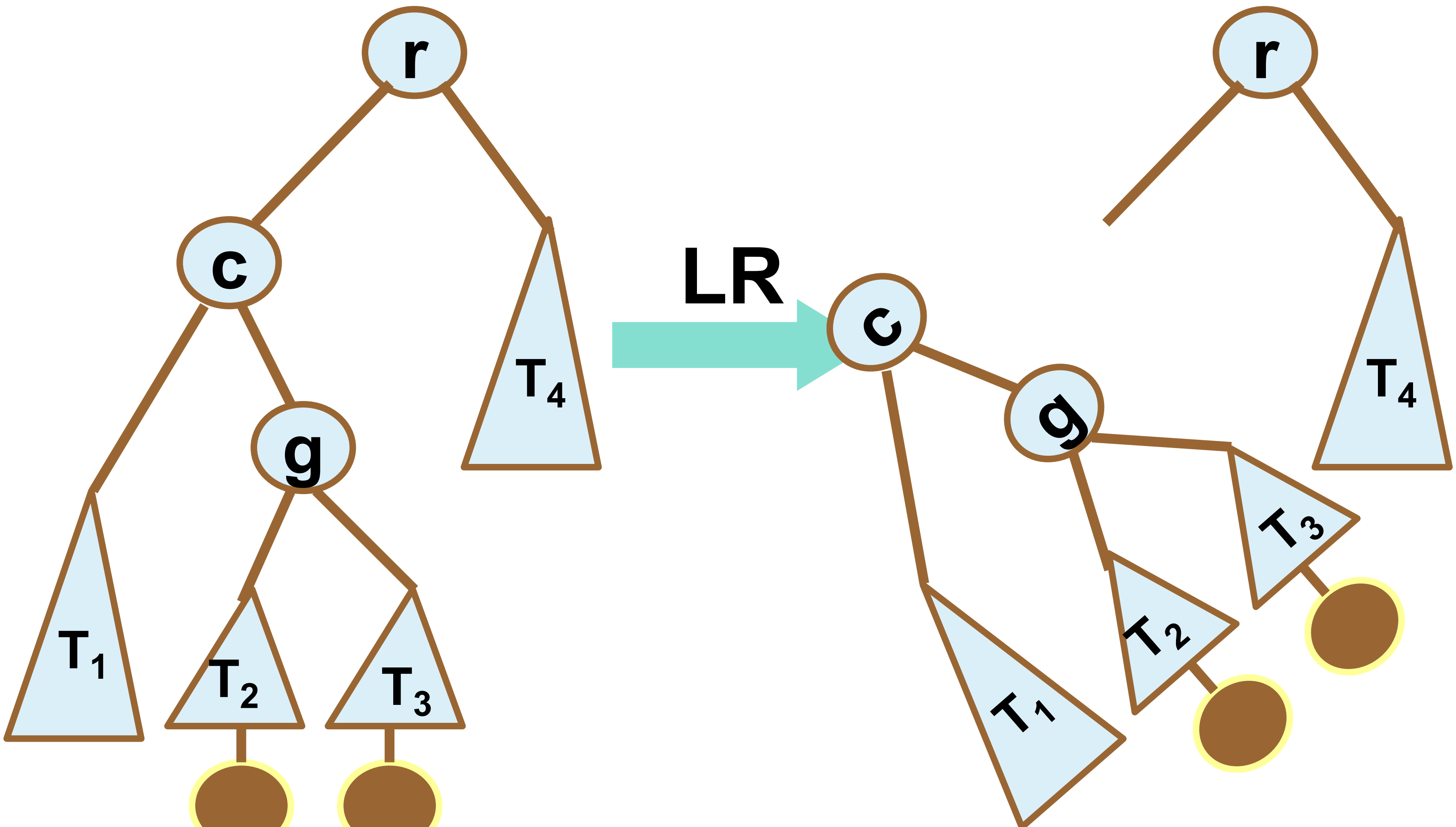


左右双转：对根 r 的左子树进行左旋，在对这颗以 r 为根的新树进行右旋！

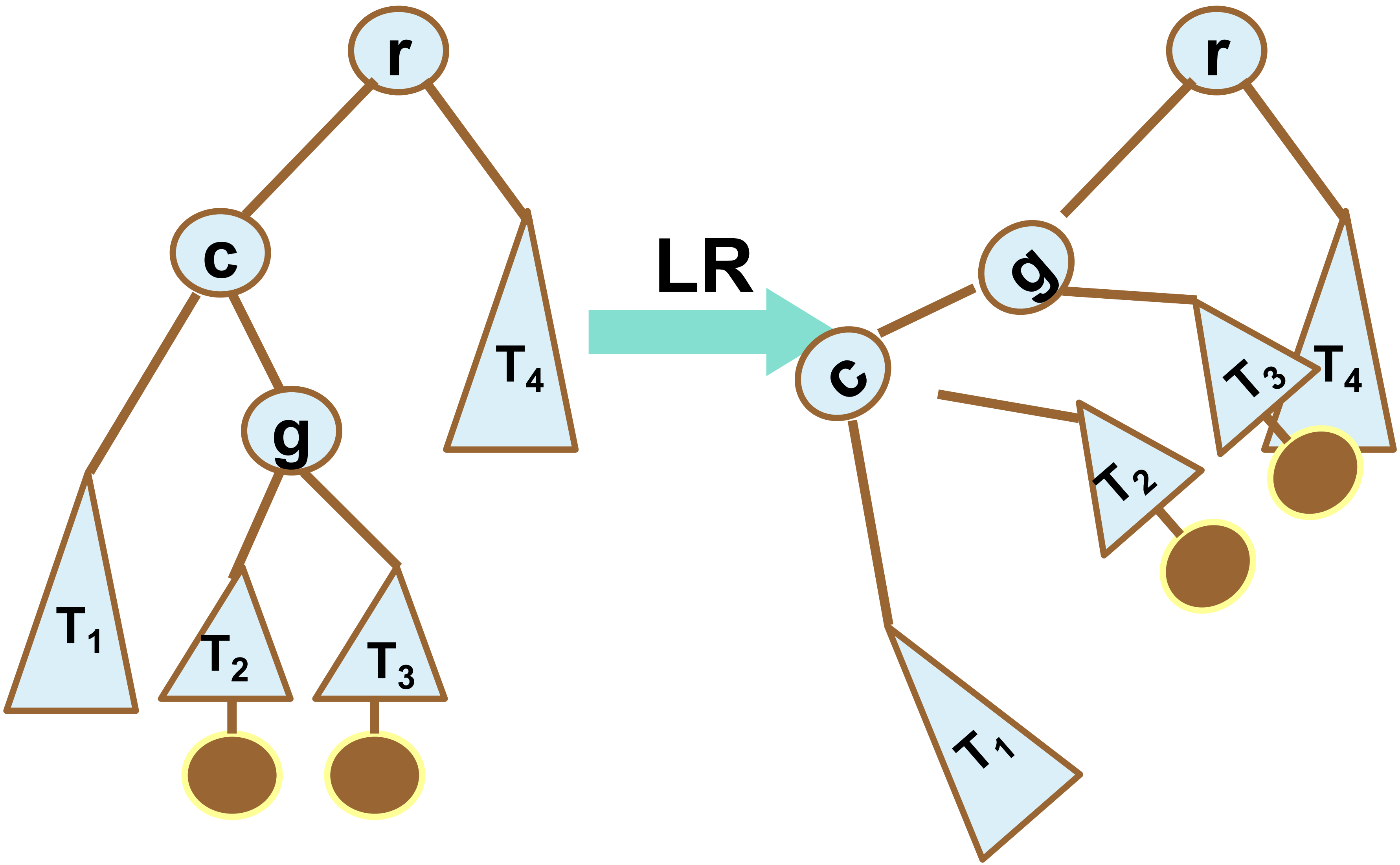
平衡查找树——AVL树



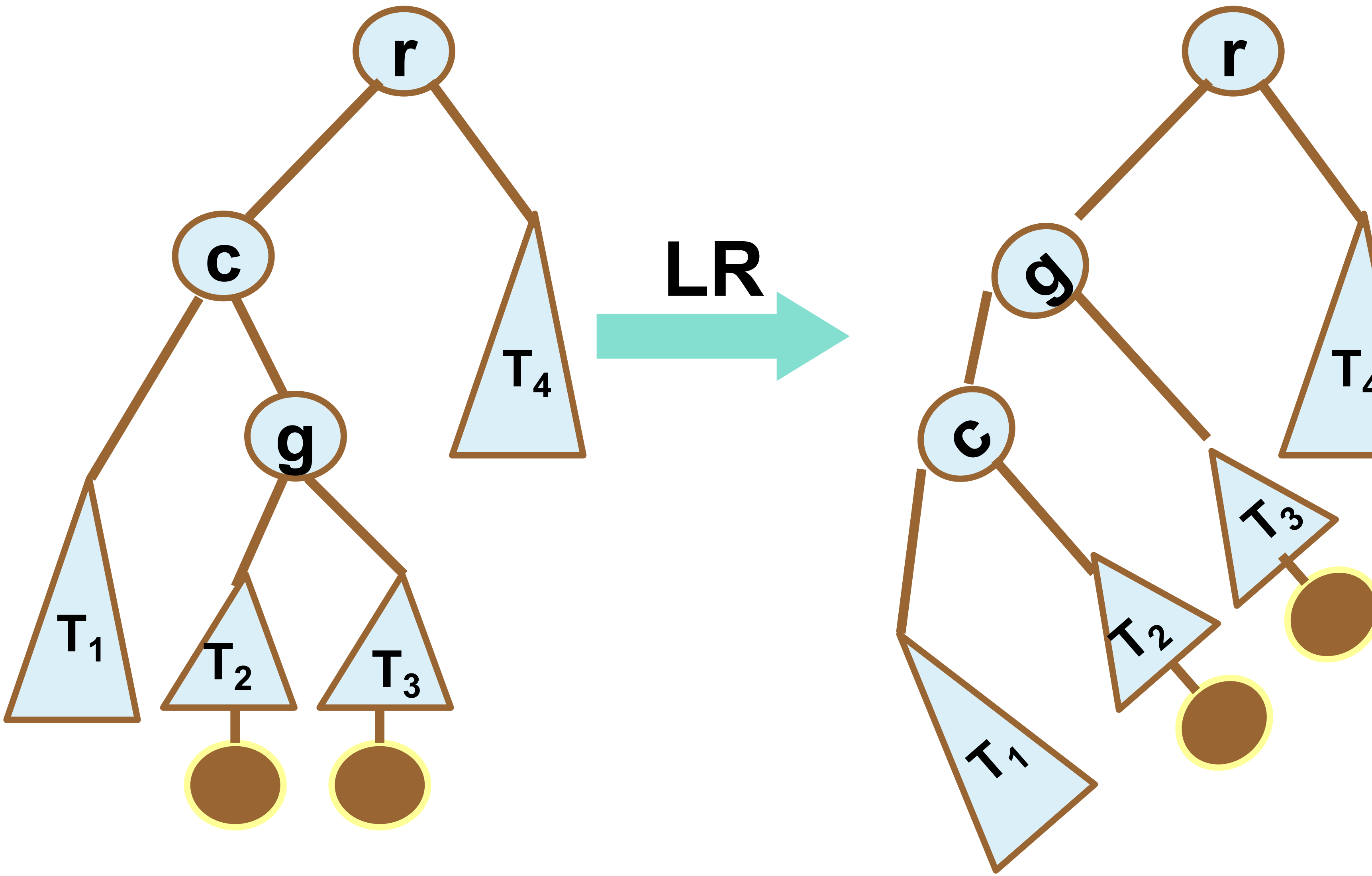
平衡查找树——AVL树



平衡查找树——AVL树

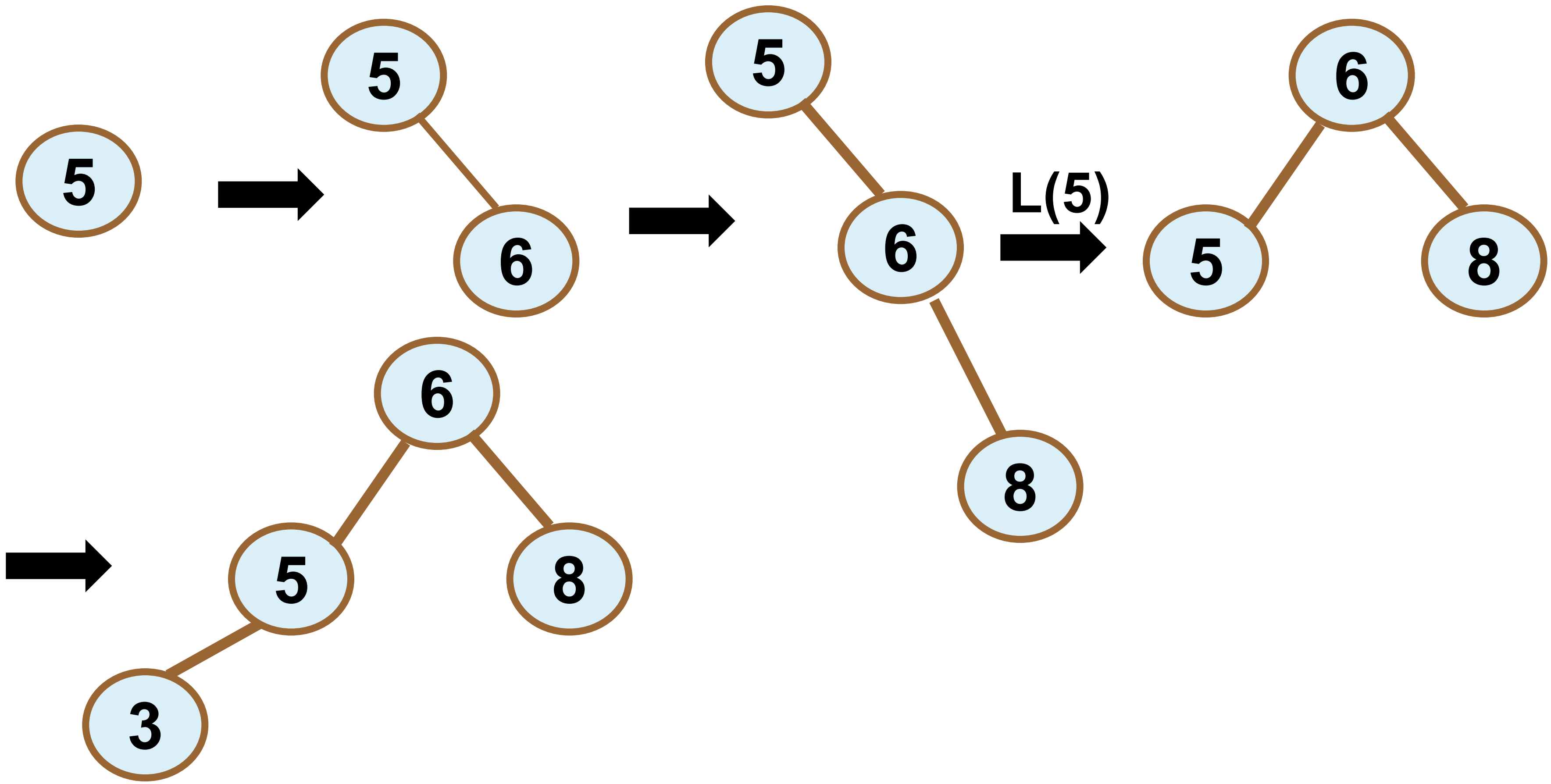


平衡查找树——AVL树



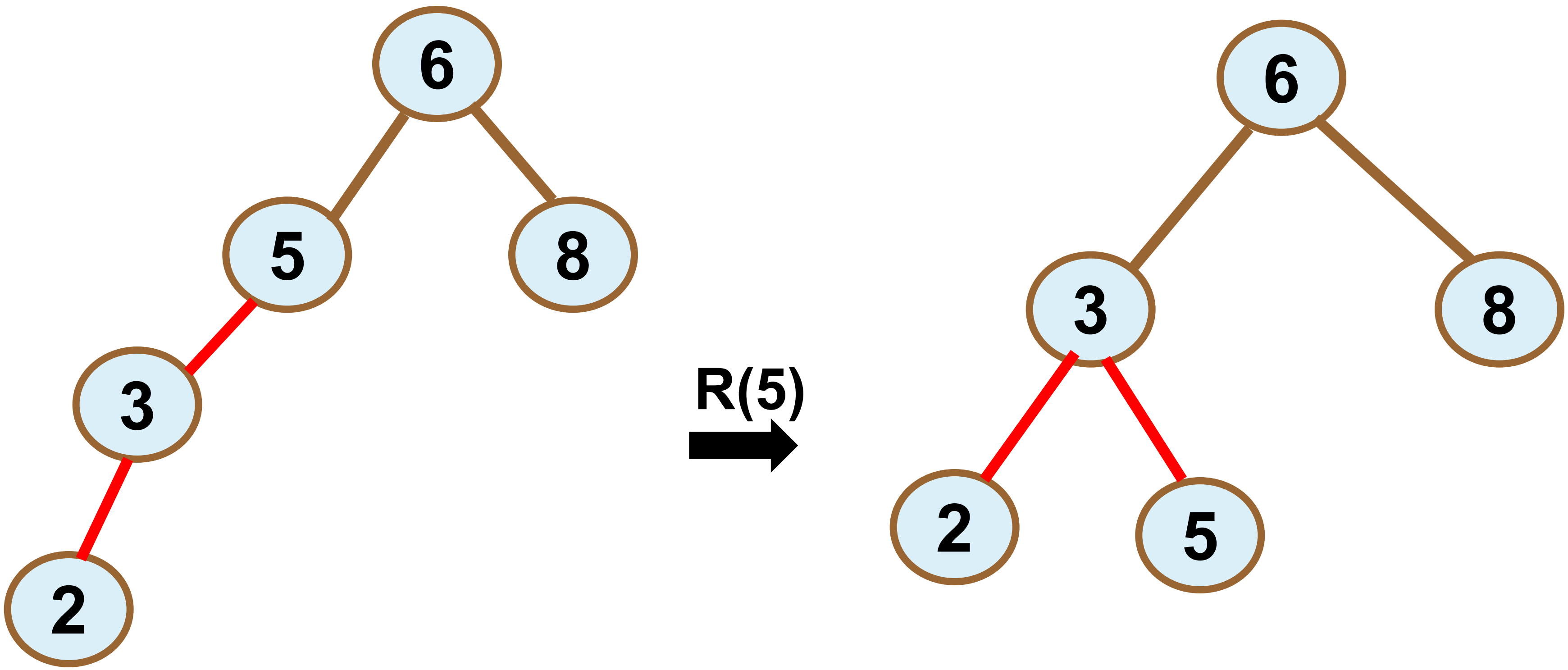
平衡查找树——AVL建树

- **依次插入节点：5，6，8，3，2，4，7**



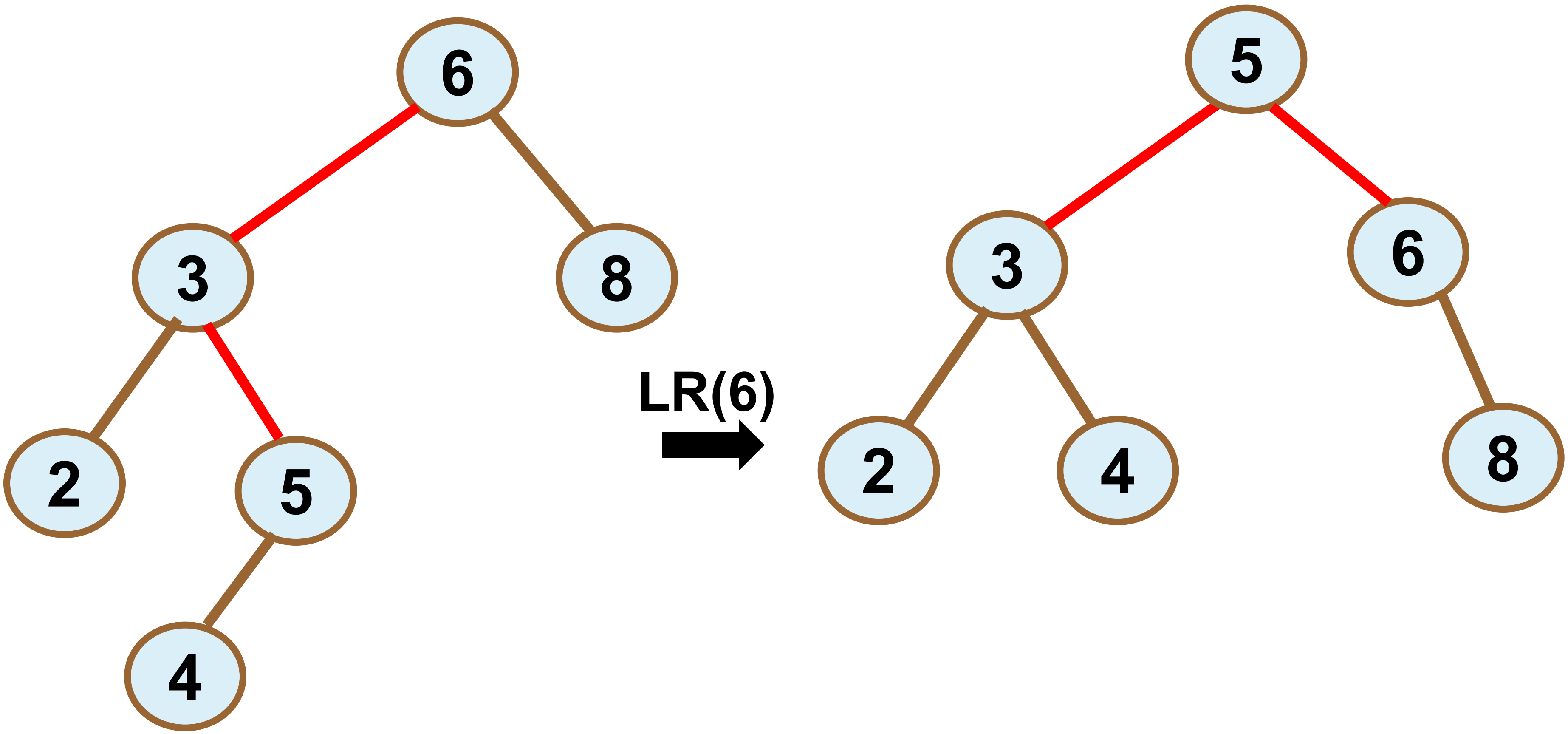
平衡查找树——AVL建树

- **依次插入节点：5，6，8，3，2，4，7**



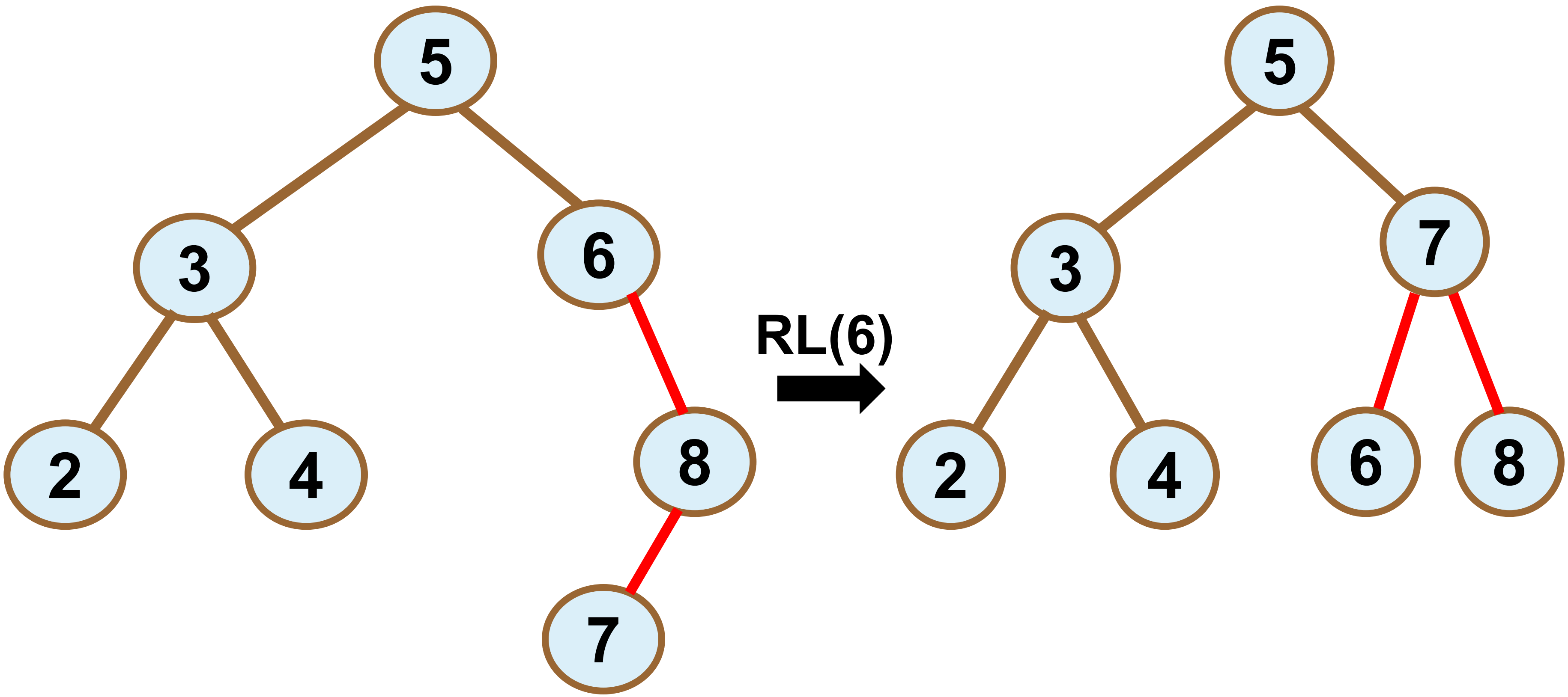
平衡查找树——AVL建树

- 依次插入节点：5，6，8，3，2，4，7



平衡查找树——AVL建树

- **依次插入节点：5，6，8，3，2，4，7**



平衡查找树——AVL树

- **请记住，如果有若干个节点的平衡因子为±2，先找出最靠近新插入的叶子的不平衡节点，然后再旋转以该节点为根的树。**
- **AVL树的效率：关键是树的高度**

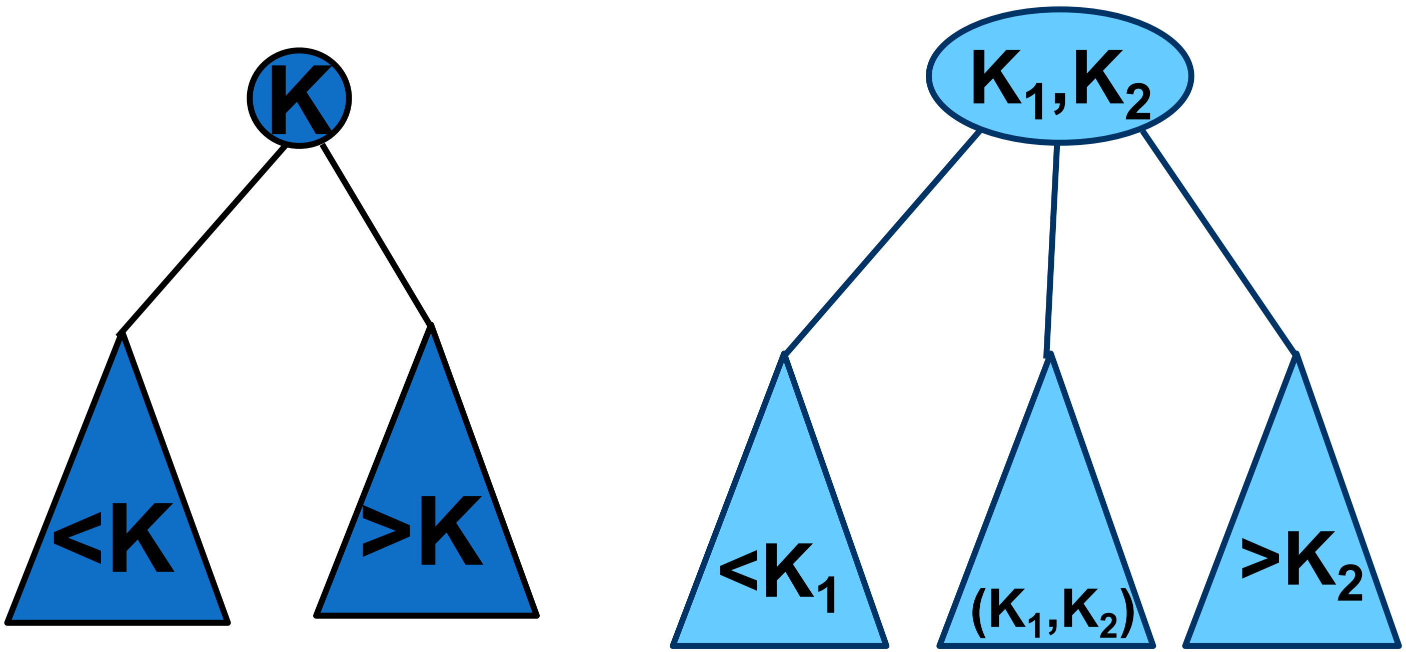
$$\lfloor \log_2 n \rfloor \le h < 1.4405 \log_2 (n+2) - 1.3277$$

- 大量实验表明，除非n比较小，否则这个高度大概是 $1.01\log_2 n + 0.1$，因此和用折半查找有序数组基本相同。

平衡查找树——2-3树

- **2-3树是一种可以包含两种类型节点的树：2节点和3节点。**
  - 一个2节点只包含一个键K和两个子女
    - 左子女作为一棵所有键都小于K的子树的根
    - 右子女作为一棵所有键都大于K的子树的根
  - 一个3节点包含两个有序的键K_1和K_2（$K_1<K_2$）并且有3个子女。
    - 最左边的子女作为键值小于K_1的子树的根
    - 中间的子女作为键值位于K_1和K_2之间的子树的根
    - 最右边的子女作为键值大于K_2的子树的根

平衡查找树——2-3树



2-3树中所有节点必须在同一层，
也就是说，2-3树总是高度平衡的：
每个叶子从树的根到叶子的路径长度相同

平衡查找树——2-3树

- **在2-3树中查找给定键K:**
  - 从根开始，如果根是一个2节点，就把它当作一个二叉查找树操作；
  - 如果根是一个3节点，在不超过两次比较之后就能确定是停止查找还是在根的3棵子树的哪一棵中继续查找

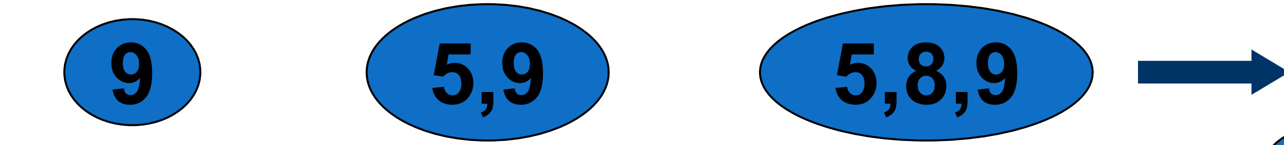
平衡查找树——2-3树

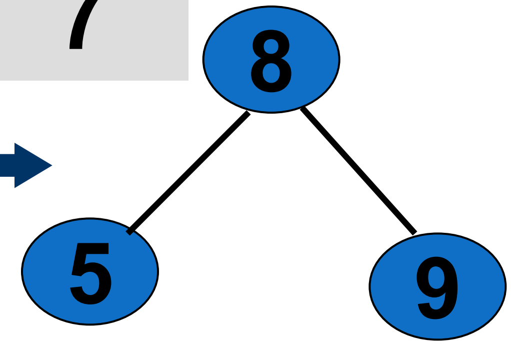
北京理工大学

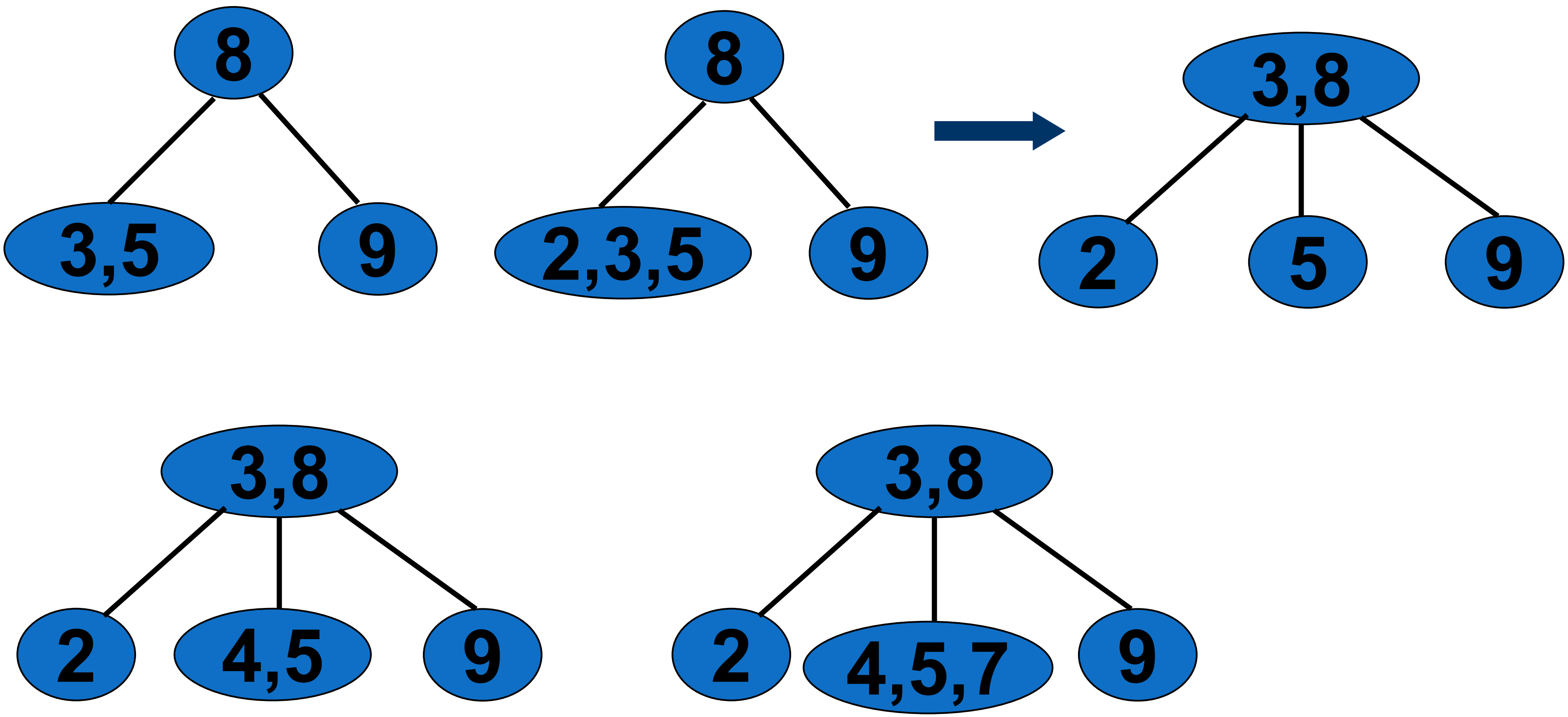
➢ **2-3树的插入新键：**

- 首先除非空树，否则总把一个新键K插入到一个叶子里，通过查找K来确定一个合适的插入位置
- 如果找到的叶子是一个2节点，根据K是小于还是大于节点中原来的键，把K作为第一个键或是第二个键进行插入
- 如果叶子是一个3节点，把叶子分裂成两个节点：三个键（原来的两个和一个新键）中最小的键放在第一个叶子里，最大的键放在第二个叶子里，同时中间的键提升到原来叶子的父母中去

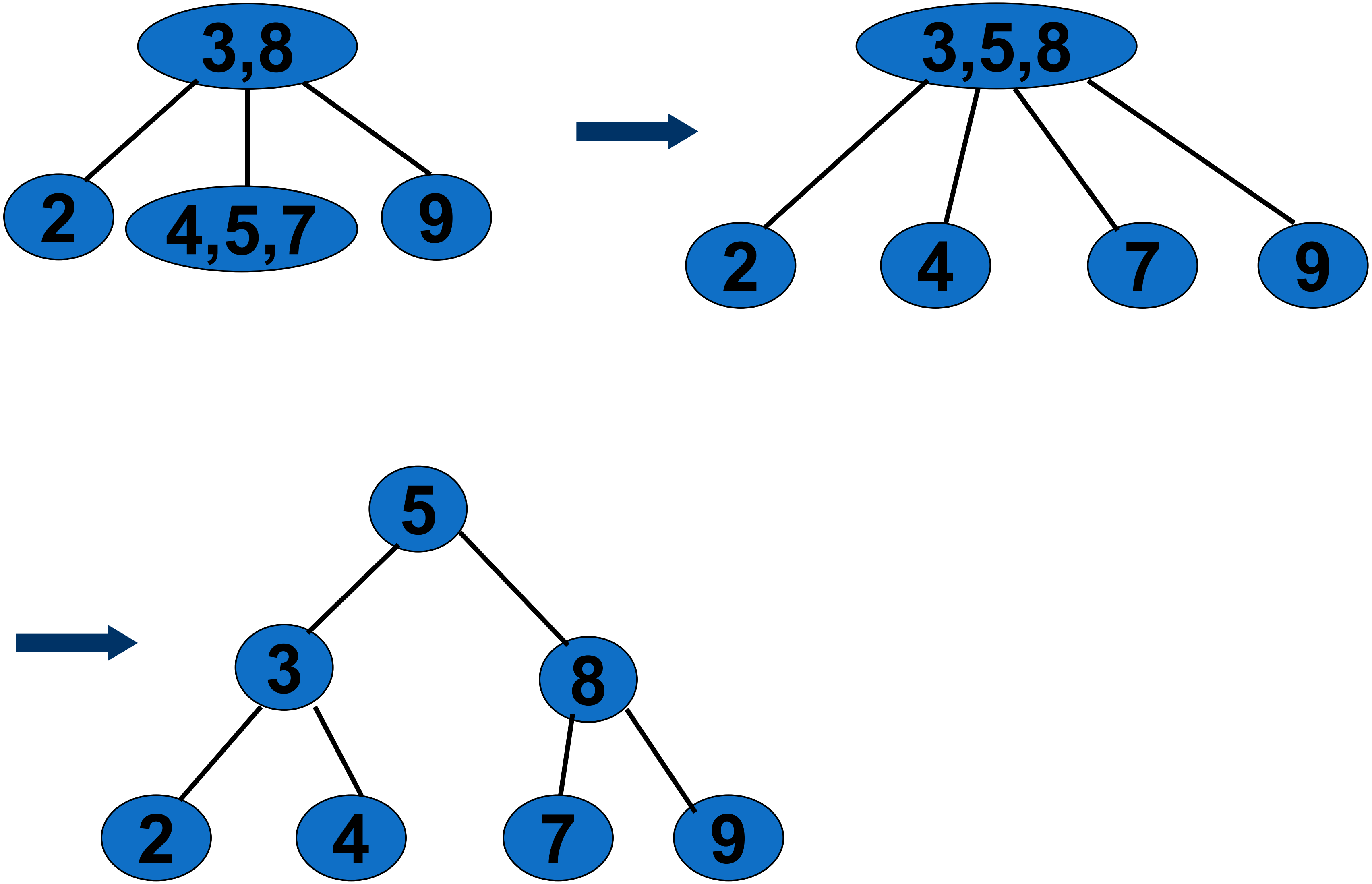
9， 5， 8， 3， 2， 4， 7







9， 5， 8， 3， 2， 4， 7



2-3树的效率：

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

变治法

- 6.1 预排序(Presorting)
- 6.2 高斯消去法(Gaussian Elimination)
- 6.3 平衡查找树(Balanced Search Trees)
- 6.4 堆和堆排序(Heaps and Heapsort)
- 6.5 霍纳法则和二进制幂(Horner's Rule)
- 6.6 问题简化(Problem Reduction)

堆和堆排序

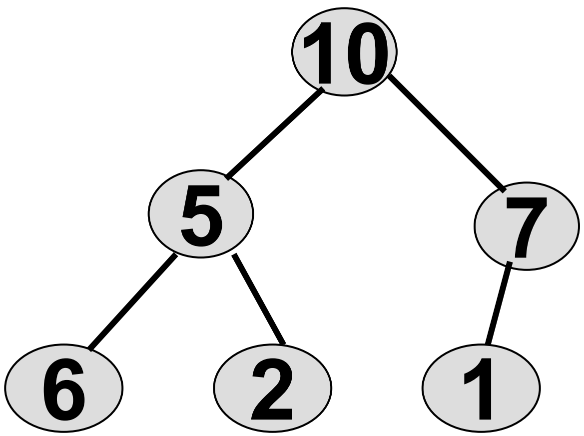
- **堆是一种灵巧的、部分有序的数据结构，尤其适合用于实现优先队列，因为他对于：**
  - 找出一个具有最高优先级的元素（最大值）
  - 删除一个具有最高优先级的元素
  - 添加一个元素到集合中

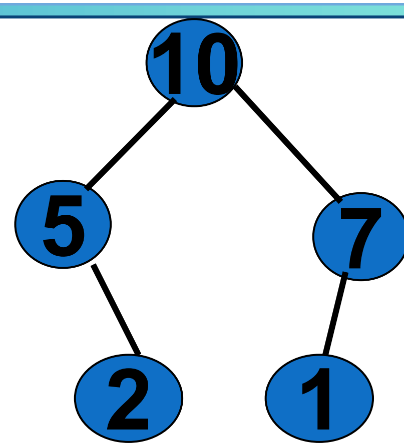
都能高效的实现。

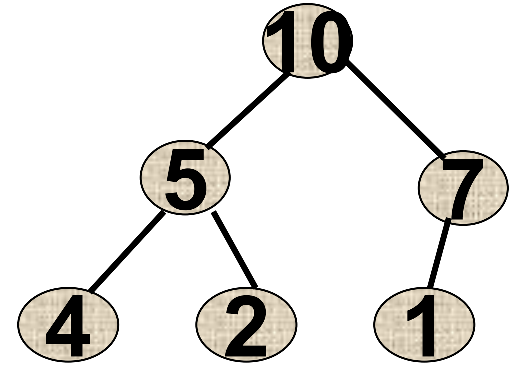
堆和堆排序

- 定义：一棵二叉树，树的节点中包含键（每个节点一个键），并且满足下列条件：
  - 这棵二叉树是完全二叉树
  - 每个节点的键都要大于等于子女
- 性质
  - n个节点的堆，高度为$\lfloor \log_2 n \rfloor$
  - 树的根总是包含堆的最大元素
  - 堆的一个节点以及该节点的子孙也是一个堆
  - 可以用数组实现堆，即从上到下，从左到右存储堆的元素

堆和堆排序







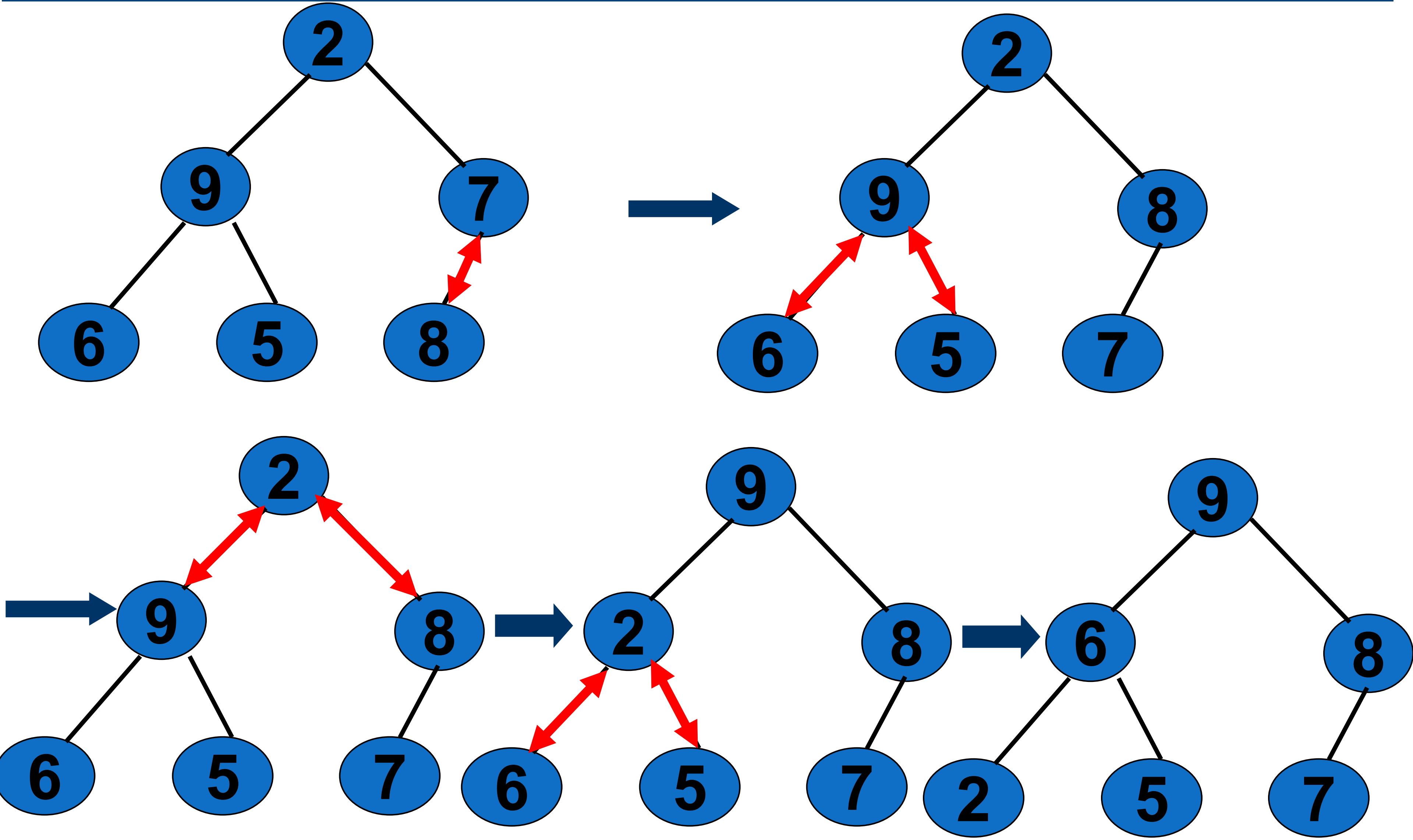
×　　×　　√

下标	0	1	2	3	4	5	6
值		10	5	7	4	2	1

堆和堆排序——构造堆

- 自底向上堆构造
  - 初始化一个包含n个节点的完全二叉树，按照给定顺序放置键
  - 从最后的双亲节点开始，到根为止，检查这些节点是否满足要求，如果不满足，将父节点与最大孩子节点交换
- 自顶向下堆构造
  - 将一个节点K附加在当前堆的最后一个叶子后面
  - 把K和它的双亲节点比较，K小于等于双亲则算法停止，否则交换之。继续与上层双亲节点比较

堆和堆排序——构造堆



自底向上堆构造

北京理工大学

```
算法 HeapBottomUp(H[1…n])
{//用自底向上算法构造一个堆
    for(i=n/2;i<=1;i--)
  {
      k=i; v=H[k];  heap=false;
      while(!heap && 2*k<=n)
      {
          j= 2*k;
          if (j<n)     //存在两个孩子
            if (H[j]<H[j+1])
              j++; //取其中较大的一个孩子
          if (v>=H[j])//用较大的孩子进行比较
            heap= true;
          else { H[k] = H[j];  k=j;}//将孩子的值给k节点
      }
       H[k] = v;
    }
}
```

自底向上堆构造

最差情况下的算法效率：

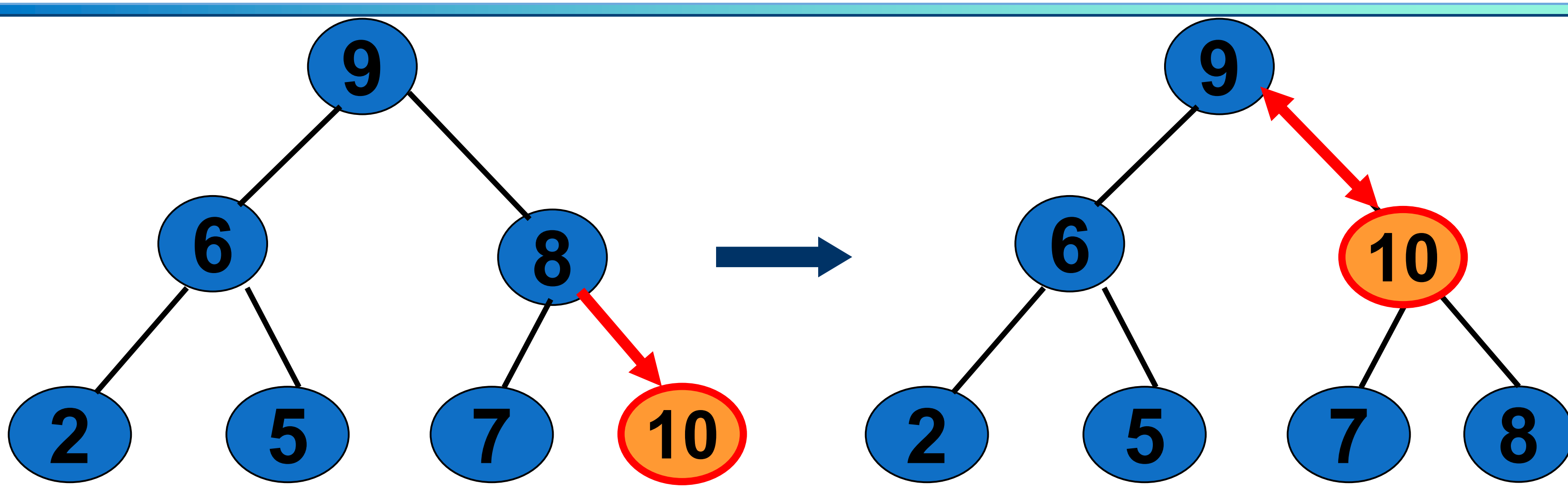
- 设n=2^k-1，即满树，每一层节点最多且高度h=k-1
- 最坏情况下每个位于第i层的键都移动到叶子层h中。因为移动到下一层要进行两次比较，所以位于第i层的键总共需要2(h-i)次比较：

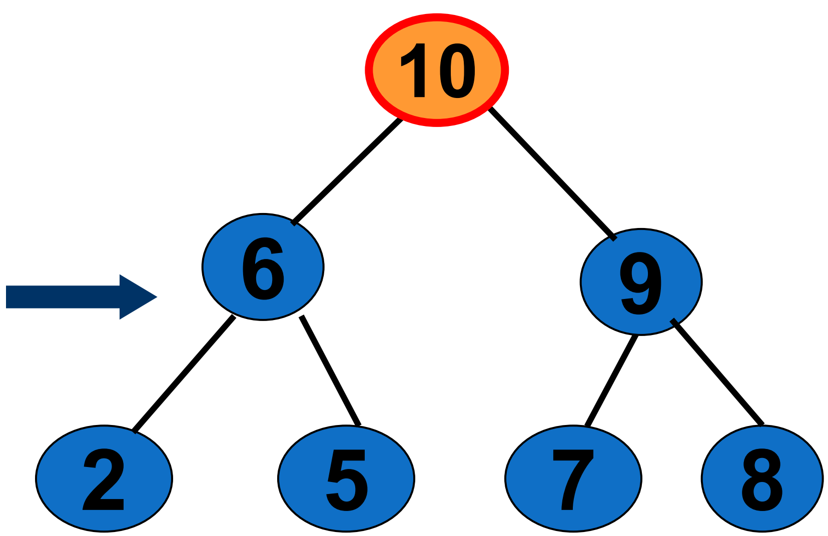
$$C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{\text{第i层}} 2(h-i) = \sum_{i=0}^{h-1} 2(h-1)2^i$$
$$= 2(n - \log_2(n+1))$$

自顶向下堆构造

- **另一种算法(效率较低)将新的键连续插入预先构造好的堆，来构造一个新堆，即：自顶向下堆构造(top-down heap construction)**
  - 首先，把包含键K的新节点负载当前对最后一个叶子后面。
  - 其次，按照下面的方法把K筛选到它的适当位置。

自顶向下堆构造



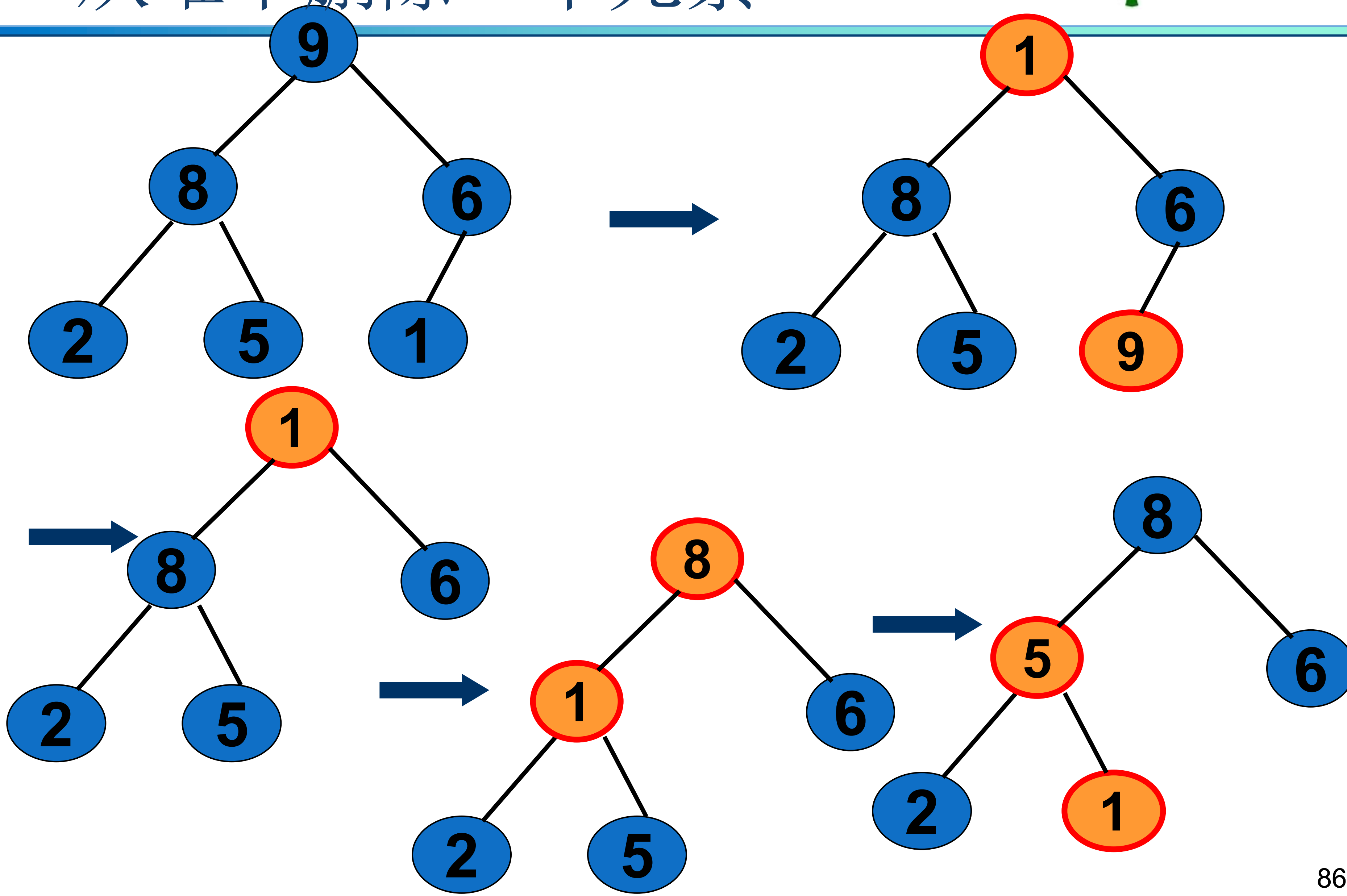


插入操作所需键值比较次数≤ $\log_2 n$，所以插入效率属于 $O(logn)$

从堆中删除一个元素

- **考虑一种最重要的情况：删除根键**
  - 根键和堆最后一个键K交换
  - 堆规模减1（删除了一个元素）
  - 树的“堆化”：K沿树向下筛选，验证沿途节点是否满足堆的要求，不满足就将其与较大孩子节点交换，直到满足为止

从堆中删除一个元素



堆删除的效率

- 删除的效率取决于交换和堆的规模减1后，树的”堆化“所需的键值比较次数。
- 因为是沿树向下比较，所以它所需键值比较次数不可能超过堆的高度的两倍。
- 所以删除效率 $\leq 2\log_2 n$, 属于 $O(logn)$

堆排序

J. W. J. Williams发明的堆排序算法分两步走：

- 构造堆：为一个给定的数组构造一个堆
- 删除最大键：对剩下的堆应用$n-1$次根删除操作

效率：

- 构造堆：$O(n)$
- 删除最大键：$\le 2\sum_{i=1}^{n-1}\log_2 i = 2n\log_2 n$

$\therefore$总效率为

$$O(n)+O(2n\log_2 n)=O(n\log n)$$

变治法

- 6.1 预排序(Presorting)
- 6.2 高斯消去法(Gaussian Elimination)
- 6.3 平衡查找树(Balanced Search Trees)
- 6.4 堆和堆排序(Heaps and Heapsort)
- 6.5 霍纳法则和二进制幂(Horner's Rule)
- 6.6 问题简化(Problem Reduction)

霍纳法则

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

$$\Downarrow$$

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

霍纳法则

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

霍纳法则

$$p(x) = x(x(x(2x - 1) + 3) + 1) - 5$$

$$p(x) = 2x^4 - x^3 + 3x^2 + x - 5$$

系数	**x = 3**
2	2
-1	**3**×2+(-1)=5
3	**3**×5+3=18
1	**3**×18+1=55
-5	**3**×55－5=160

霍纳法则

算法 Horner(p[0…n],x)

{ //用霍纳法则求多项式在给定点的值

//输入:n次多项式的系数组$p[0,\ldots,n]$，及数字x

//输出:多项式在x的值

$p = p[n]$;

for(i=n-1;i >=0;i--)

$p = x * p + p[i]$;

return p;

}

$$M(n) = A(n)$$

$$= \sum_{i=0}^{n-1} 1 = n$$

霍纳法则

- 霍纳法则还有一些有用的副产品：该算法在计算$p(x)$在某一点x_0上的值时所产生的中间数字，恰好可以作为$p(x)$除以$x-x_0$的商的系数，而算法的最后结果，除了等于$p(x)$以外，还等于这个除法的余数。
- 这种被称为**综合除法**。

$p(x)=2x^4-x^3+3x^2+x-5$	
系数	$x=3$
2	2
-1	**3**×2+(-1)=5
3	**3**×5+3=18
1	**3**×18+1=55
-5	**3**×55－5=160

$$\frac{2x^4-x^3+3x^2+x-5}{x-3}$$
$$=2x^3+5x^2+18x+55\text{余}160$$

二进制幂

- **运用基于“改变表现”的思想，使用指数n的二进制表示计算a^n：**
  - 从左到右处理二进制串(运用霍纳法则表示指数)
  - 从右到左处理二进制串(不用霍纳法则)

二进制幂

- 运用基于“改变表现”的思想，使用指数n的二进制表示计算a^n：
  - 从左到右处理二进制串
  - 从右到左处理二进制串
- 设$n = b_I \ldots bi \ldots b_0$是在二进制系统中，表示一个正整数$n$的比特串。所以可以把$n$通过下面多项式表示：

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

$$p(x) = b_l x^l + \cdots + b_i x^i + \cdots + b_0$$

二进制幂

北京理工大学

- 用霍纳法则计算二进制多项式p(x=2)

//当n≥1，第一个数字总是1

```
p=1;
for(i=l-1;i>=0;i--）
      p=2*p+bi;
```

- 相对于$a^n = a^{p(2)}$

$a^p = a^1$;

for(i=l-1;i>=0;i--）

$$a^p = a^{2*p+bi}$$

$$a^{2p+b_i} = a^{2p} \times a^{b_i} = \begin{cases} (a^p)^2, & \mathrm{b_i} = 0 \\ (a^p)^2 \times a, & \mathrm{b_i} = 1 \end{cases}$$

二进制幂

- 用霍纳法则计算二进制多项式p(x=2)

//当n≥1，第一个数字总是1

```
p=1;
for(i=l-1;i>=0;i--)
    p=2*p+b_i;
```

- 相对于$a^n=a^{p(2)}$

$a^p=a^1$;

for(i=l-1;i>=0;i--)

$a^p=a^{2^*p+b_i}$

```
for (i=l-1; i>=0; i--) {
    a^p *= a^p;
    if (b_i) a^p *=a;
}
```

二进制幂

```
算法 LRBExp(a,b(n))
{//从左到右二进制幂算法计算a^n
  product = a;
  for(i=l-1;i>=0;i--){
      product×=product;
      if(b_i==1)   product ×= a;
  }
  return product;
}
```

$$(b-1) \le M(n) \le 2(b-1)$$

$$b-1 = \lfloor \log_2 n \rfloor$$

北京理工大学

从右到左二进制幂

$$\mathrm{a}^{n} = a^{b_l 2^l + \cdots + b_i 2^i + \cdots + b_0}$$

$$= a^{b_l 2^l} \cdots a^{b_i 2^i} \cdots a^{b_0}$$

因此可以用各项：

$$a^{b_i 2^i} = \begin{cases} \mathrm{a}^{2^i}, & \text{如果b}_i = 1 \\ 1, & \text{如果b}_\mathrm{i} = 0 \end{cases}$$

的积来计算a^n

二进制幂

```
算法 RLBExp(a,b(n))
{//从右到左二进制幂算法计算aⁿ
   term=a;  //初始化aⁿ
   if(b0==1)   product = a;
   else  product = 1;
   for(i=1;i<l;i++) {
         term*=term;
         if (bi==1)    product*=term;
   }
   return     product;
}
```

变治法

北京理工大学

- 6.1 预排序(Presorting)
- 6.2 高斯消去法(Gaussian Elimination)
- 6.3 平衡查找树(Balanced Search Trees)
- 6.4 堆和堆排序(Heaps and Heapsort)
- 6.5 霍纳法则和二进制幂(Horner's Rule)
- 6.6 问题简化(Problem Reduction)

问题简化

- 问题化简是一种重要的解题策略
- 问题化简思想在计算机科学理论中扮演了一个中心角色，但难度在于如何找到一个可以化简手头问题的目标问题
- 举例说明
  - 求最小公倍数
  - 计算图中的路径数量
  - 优化问题的化简
  - 线性规划
  - 简化为图问题

1、求最小公倍数

北京理工大学

- lcm(m,n)：记作两个正整数m和n的最小公倍数，如lcm(24,60)=120:
  - 蛮力法：求出m和n的质因数，把所有公共质因数的积乘以m的不在n中的质因数，再乘以n的不在m中的质因数，例如

$$24=2\times2\times2\times3$$

$$60=2\times2\times3\times5$$

$$\mathrm{lcm}(24,60)=(2\times2\times3)\times2\times5=120$$

不仅缺乏效率，而且需要一个连续的质数列表

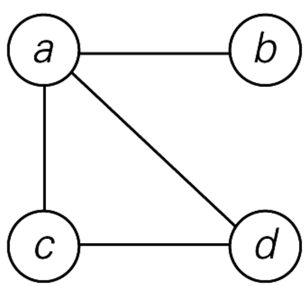
1、求最小公倍数

- 通过问题化简，发现lcm和gcd的积刚好把m和n的质因数都包含了一次，故有：

$$lcm(m,n)=\frac{m\times n}{\gcd(m,n)}$$

2、计算图中的路径数量

- 计算图中两个顶点之间的路径数量问题
  - **从图(无向图或有向图)中的第i个顶点到第j个顶点之间，长度为k>0的不同路径的数量等于A^k的第(i,j)个元素，其中，A是该图的邻接矩阵。**



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

FIGURE 6.16 A graph, its adjacency matrix A, and its square A^2. The elements of A and A^2 indicate the number of paths of lengths 1 and 2, respectively.

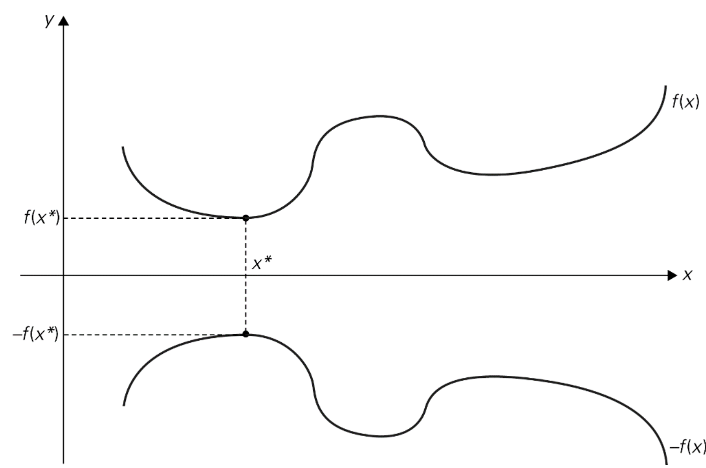
3、优化问题的化简

北京理工大学

- 最大化问题：求某些函数的最大值问题
- 最小化问题：求某些函数的最小值问题
- 现在，我们必须求某个函数的最小值，假设我们知道一个求函数最大值的算法，我们如何利用后者呢？

$$\min(f(x)) = -\max(-f(x))$$

- **即为求函数的最小值，我们可以先求它的负函数的最大值**



3、优化问题的化简

- 最小化和最大化问题之间的关系是非常具有一般性的：它对于定义在任何定义域D上的函数都有效。即，我们可以对满足额外约束条件的多个变量的函数应用这个公式。
- 求函数极值点的标准微积分过程实际上就是问题化简：
  - 求出函数的导数$f'(x)$
  - 对方程$f'(x)=0$求解

4、线性规划

- 许多决策最优化的问题都可以化简为“线性规划”问题的一个实例
- 线性规划问题是一个多变量线性函数的最优化问题
  - 这些变量所要满足的一些约束是以线性等式或者线性不等式的形式出现的
- 线性规划具有足够的灵活度，可以对种类广泛的重要应用来建模
  - 飞机航班工作人员排班、交通和通信网络规划、石油勘探和提纯，以及工业生产优化
- 线性规划被认为是应用数学史上最重要成就之一

4、线性规划

➤ **例1 假定有一个大学基金需要进行一亿美元的投资。**

- 这笔钱必须分成三类投资：股票，债券和现金。
- 基金经理们对他们的股票，债券和现金的预期年收益分别是10%，7%和3%。
- 因为股票比债券的风险更高，要求投资在股票上的资金不能超过债券投资的三分之一。
- 此外，现金投资至少应相当于股票和债券投资总额的25%。

➤ **基金经理如何投资才能使收益最大化？**

4、线性规划

使$0.10x+0.07y+0.03z$最大化

约束条件是：

1、$x+y+z=100$

2、$x \leqslant y/3$

3、$z \geqslant 0.25(x+y)$

4、$x \geqslant 0, y \geqslant 0, z \geqslant 0$

•单纯形法:只能成功处理不把变量值限定在整数中的线性规划问题

•算法的最差效率属于指数级的，但在典型输入时的性能非常好

使$c_1x_1+\ldots+c_nx_n$最大化（或最小化）

约束条件是：$a_{i1}x_1+\ldots+a_{in}x_n \le b_i,\ i=1,\ldots,m$

$x_1 \ge 0,\ldots,\ x_n \ge 0$

4、线性规划

- 例2 背包问题：给定一个承重为W的背包和n个重量为$w_1,\cdots,w_n$、价值为$v_1,\cdots,v_n$的物品，求这些物品中最有价值的一个子集，并且能够装到背包中。

使 $\sum_{j=1}^{n} v_j x_j$ 最大化（或最小化）

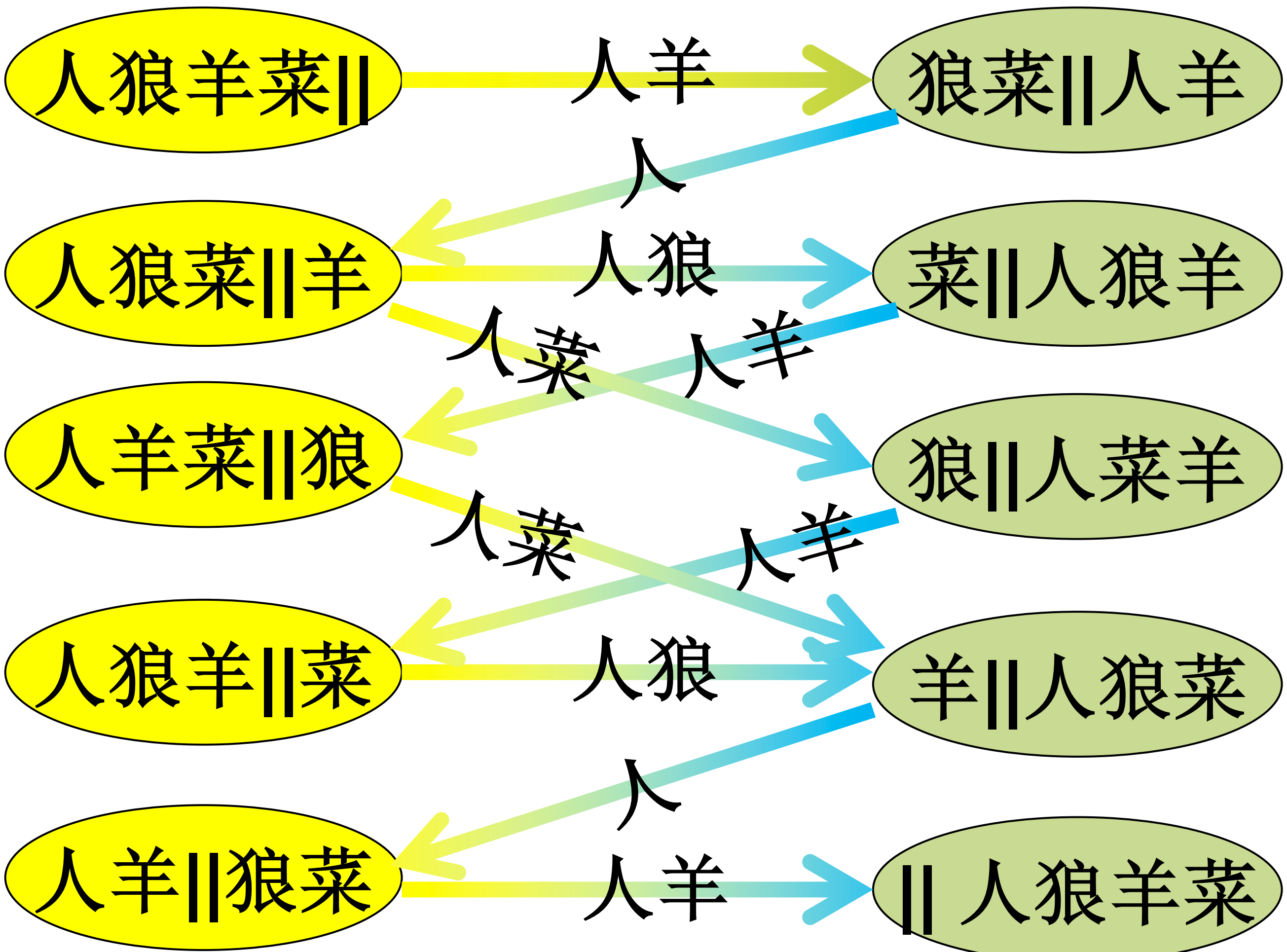
约束条件是：$\sum_{j=1}^{n} w_j x_j \le W$

$0 \le x_j \le 1,\ j=1,\ldots,n$

5、简化为图问题

- 可以把许多问题化简为图问题，然后再求解。
  - 状态空间图：把问题化简为一个求初始状态顶点到目标状态顶点之间路径的问题。
- 例 一个农夫在河边带了一只狼、一只羊和一筐白菜。他需要把这三样东西带到河的对岸。然而这艘船只能容下农夫本人和另外一样东西。如果农夫不在场，狼就会吃掉羊，羊也会吃掉白菜。请为农夫解决这个问题，或者证明它无解。

5、简化为图问题


人狼羊菜||
人羊
狼菜||人羊
人
人狼菜||羊
人狼
菜||人狼羊
人菜
人羊
人羊菜||狼
狼||人菜羊
人菜
人羊
人狼羊||菜
人狼
羊||人狼菜
人
人羊||狼菜
人羊
|| 人狼羊菜