**20181025**

# 第 1 章　绪论

## 🕮　1.1　计算

计算模型=计算机=信息处理工具
所谓算法，即特定计算模型下，旨在解决特定问题的指令序列。

算法特点：

1) 输入：待处理的信息(问题)
2) 输出：经处理的信息(答案)
3) 正确性：的确可以解决指定的问题
4) 确定性：任一算法都可以描述为一个由基本操作组成的序列
5) 可行性：每一基本操作都可实现，且在常数时间内完成
6) 有穷性：对于任何输入，经有穷次基本操作，都可以得到输出

……

有穷性举例：

$$Hailstone(n) = \begin{cases} \{1\}, & n \le 1 \\ \{n\} \cup Hailstone\left(\frac{n}{2}\right), & n\text{偶数} \\ \{n\} \cup Hailstone(3n+1), & n\text{奇数} \end{cases}$$

Hailstone Sequence：至今也没有证明对于任意 $n$，其序列是有穷或无穷。

什么是好算法：

1) 正确：符合语法，能够编译、链接
2) 健壮：能判别不合法的输入并做适当处理，而不致非正常退出
3) 可读：结构化代码，变量准确命名、注释…
4) 效率：速度尽可能快；存储空间尽可能少

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

Data Structure and Algorithms 简称 DSA，不同 DSA 性能有好坏优劣之别。
1) 引入理想、同一、分层次的尺度；
2) 运用该尺度，以测量 DSA 的性能。

## 🕮　1.2　计算模型

### ❁　1.2.1　问题规模

如果考察 $T_A(P)$ = 算法 A 求解问题实例 P 的计算成本，意义不大，因为可能的实例太多，如何归纳概括？使用划分等价类。
问题实例的规模，往往是决定计算成本的主要因素。

通常：规模接近，计算成本也接近；规模扩大，计算成本亦上升。

### ❁　1.2.2　最坏情况

令$T_A(n)$ = 算法 A 求解某一问题规模为$n$的实例，所需的计算成本。

讨论特定算法 A 时，简记为$\mathrm{T}(n)$。
然而这样的定义仍有问题。因为对于同一问题，等规模的不同实例，计算成本不尽相同，甚至可能有实质性差别。

如平面上$n$个点，找到所形成三角形面积最小的三个点。以蛮力算法为例，最坏情况需要枚举所以$C_n^3$种组合，但运气好的话，一上来就找到。

因此，稳妥起见，取$\mathrm{T}(n) = \max\{\mathrm{T}(\mathrm{P}) \mid |\mathrm{P}| = n\}$
也就是在规模同为$n$的所有实例中，只关注最坏(成本最高)情况。

### 1.2.3 理想模型

同一个问题通常有多种算法，如何判断它们的优劣呢?
实验统计是最直接的方法，但不足够准确反映算法真正效率。
1) 不同算法，可能更适应于不同**规模/类型**的输入
2) 同一算法，可能由不同**程序员**、不同**编程语言**、经不同**编译器**实现；可能实现并运行于不同的**体系结构**、**操作系统**...

为了给出**客观**的评价，必须抽象出一个**理想**的模型，不依赖于上述各种因素，从而直接准确地描述、测量、评价算法。

### 1.2.4 图灵机模型

图灵机 TM (Turing Machine)
(1) Tape (纸带)：依次均匀地划分为单元格，各注有某一字符，默认为'#';
(2) Alphabet (字母表)：字符的种类有限
(3) Head (读写头)：总是对准某一单元格，并可读取和改写其中的字符，每经过一个节拍，可转向左侧或右侧的邻格
(4) State (状态表)：TM 总是处于有限种状态的某一种，每经过一个节拍，可(按照规则)转向另一种状态;

转换函数(Transition Function)：(q, c; d, L/R, p)
若当前状态为 q 且字符为 c，则将当前字符改写为 d；转向左侧/右侧的邻格，转入 p 状态；一旦转入特定的状态'h'，则停机。


**实例**：TM : Increase
功能：将二进制非负整数加一
算法：全'1'的后缀翻转为全'0'，原最低位的'0'或'#'翻转为'1'。

### 1.2.5 RAM 模型

RAM (Random Access Machine)与图灵机类似，都假设无限空间。
寄存器顺序编号，总数没有限制：R[0], R[1], R[2], R[3], …

每一基本操作只需常数时间：

| | |
|---|---|
| R[i] <- c | R[i] <- R[j] |
| R[i] <- R[R[j]] | R[R[i]] <- R[j] |
| R[i] <- R[j] + R[k] | R[i] <- R[j] - R[k] |
| IF R[i] = 0 GOTO l | IF R[i] > 0 GOTO l |
| GOTO l | STOP |

与 TM 模型一样，RAM 模型也是一般计算工具的简化与抽象，独立于具体平台，对算法的效率做出可信的比较与评判。

因为在这些模型中做了转换：
算法的运行时间 $\propto$ 算法需要执行的**基本操作次数**
$T(n)=$ 算法为求解规模为$n$的问题所需执行基本操作次数

**实例**：RAM : Floor
功能：向下取整的除法
对于$c \geq 0, d > 0$：

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

算法：反复从 R[0]=1+c 中减去 R[1]=d，统计在下溢之前，所做减法次数$x$。

```
0    R[3] <- 1                r3 = 1
1    R[0] <- R[0] + R[3]      r0++
2    R[0] <- R[0] - R[1]      r0 -= r1
3    R[2] <- R[2] + R[3]      r2++
4    IF R[0] > 0 GOTO 2       if r0 > 0 goto 2
5    R[0] <- R[2] - R[3]      else r2-- and r0 = r2
6    STOP                     return r0
```

执行此过程，可以记录为一张表，表的行数就是执行基本指令的总次数。

## 🕮 1.3 大 O 记号

随着问题规模增长，计算成本如何增长?
这里更关心足够大的问题，注重考察成本的增长趋势。
渐进分析(asymptotic analysis)：当$n \gg 2$后，对于规模为 n 的问题，计算成本：
需执行的基本操作次数：$T(n)=?$
需占用的存储单元数：$S(n)=?$

### 1.3.1 大 O 记号 (big-O notation)

$T(n) = O(f(n))$ $if$ $\exists c > 0$，当 n≫2 后，有$T(n) < c \cdot f(n)$

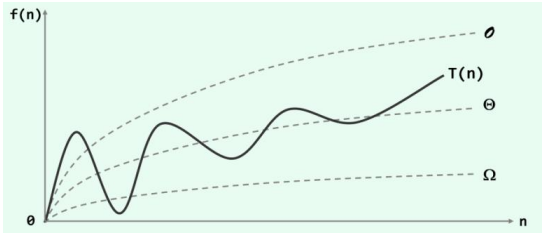
如$T(n) = \sqrt{5n \cdot [3n \cdot (n+2)+4]+6} < \sqrt{5n \cdot [6n^2+4]+6} < \sqrt{35n^3+6}$

$< 6 \cdot n^{1.5} = O(n^{1.5})$

与$T(n)$相比，$f(n)$更为简洁，但依然能反映增长趋势
1) 常系数可以忽略：$O(f(n)) = O(c \cdot f(n))$
2) 低次项可以忽略：$O(n^3+n^2) = O(n^3)$
考虑问题要**长远**($n$足够大)、**主流**(常系数、低次项等非主流可以忽略)。

big-O 是T($n$)的上界(upper bound)，可以认为 big-O 是对T($n$)的悲观估计。

其他符号：
1) $\mathrm{T}(n)=\Omega(f(n))$：$\exists c>0$，当 n≫2 后，有$\mathrm{T}(n)>c\cdot f(n)$
也就是$\Omega$是T($n$)的下界。
2) $\mathrm{T}(n)=\Theta(f(n))$：$\exists c_1>c_2>0$，当 n≫2 后，有$c_1\cdot f(n)>\mathrm{T}(n)>c_2\cdot f(n)$
T($n$)能被同一函数经过$c_1,c_2$放大倍后从上下两方界定。
认为$\Theta$构成T($n$)的确界。




### ❁ 1.3.2 高效解

$O(1)$：常数(constant function)

$$1=2018=2018\times 2018=O(1)$$

这类算法的效率最高。
不含转向(循环、调用、递归等)必顺序执行，即是$O(1)$

$O(\log n)$：对数
不写底数，因为常底数无所谓。
这类算法非常高效，复杂度无限接近于常数$O(1)$

### ❁ 1.3.3 有效解

$O(n^k)$：多项式 (polynomial function)

$$a_k n^k+a_{k-1}n^{k-1}+\cdots+a_1 n+a_0=O(n^k),a_k>0$$

$O(n)$：线性 (linear function)
很多编程习题主要在$O(n)$~$O(n^2)$之间。

这类算法的效率通常认为已令人满意，然而这个标准是不是太低了?
凡多项式复杂度，都认为可解，而不是难解。

### ❁ 1.3.4 难解

$O(2^n)$：指数 (exponential function)

$$\forall c>1,n^c=O(2^n)$$
$$n^{1000}=O(1.0000001^n)=O(2^n)$$
$$1.0000001^n=\Omega(n^{1000})$$

这类算法的计算成本增长极快，通常被认为不可忍受。
从$O(n^c)$到$O(2^n)$是从有效算法到无效算法的分水岭。
很多问题的$O(2^n)$算法往往显而易见，但设计出$O(n^c)$算法却很难。

**例**：S 包含$n$个正整数，$\sum S = 2m$。S 是否存在子集 T，满足$\sum T = m$?
直觉算法：逐一枚举 S 的每一子集，统计其中元素总和
然而需要迭代轮数：

$$\sum_{i=0}^{n} C_n^i = (1+1)^n = 2^n$$

该算法时间复杂度为$O(2^n)$，不好。

`2-Subset is NP-complete`：就目前的计算模型而言，不存在可在多项式时间内回答此问题的算法。也就是上面的直觉算法已属最优。

## 🕮 1.4 算法分析

### ❁ 1.4.1 算法分析主要任务

两个主要任务 = 正确性(不变性×单调性) + 复杂度

C++等高级语言的基本指令，均等效于常数条 RAM 基本指令；在渐近意义下，二者大体相当。
1) 分支转向：goto // 算法灵魂；出于结构化考虑被隐藏了
2) 循环迭代：for, while, ... // 本质上是: if + goto
3) 调用 + 递归 // 本质上也是 goto

复杂度分析主要方法：
1) 迭代：级数求和
2) 递归：递归跟踪 + 递归方程
3) 猜测 + 验证

### ❁ 1.4.2 级数

① 算术级数：与末项平方同阶

$$\mathrm{T}(n) = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} = \mathrm{O}(n^2)$$

② 幂方级数：比幂次高出一阶

$$\mathrm{T}_d(n) = 1 + 2^d + \cdots + n^d = \sum_{k=0}^{n} k^d \approx \int_0^n x^{d+1} dx = \frac{1}{d+1} n^{d+1} = O(n^{d+1})$$

③ 几何级数($a > 1$)：与末项同阶

$$\mathrm{T}_a(n) = a^0 + a^1 + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1} = O(a^n)$$

④ 收敛级数

$$\frac{1}{1\times 2} + \frac{1}{2\times 3} + \frac{1}{3\times 4} + \cdots \frac{1}{(n-1)n} = 1 - \frac{1}{n} = O(1)$$

$$1 + \frac{1}{2^2} + \cdots + \frac{1}{n^2} = O(1)$$

⑤ 级数未必收敛，但长度有限
调和级数与对数级数：

$$h(n) = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \Theta(\log n)$$

$$\log 1 + \log 2 + \cdots + \log n = \log(n!) = \Theta(n \log n)$$

### ❀ 1.4.3 实例

① 取非极端元素

给定整数子集 S，$|S| = n \geq 3$，找出元素$a \in S, a \neq \max(S)\ and\ \ a \neq \min(S)$

算法：1) 从 S 中任取 3 个元素$\{x, y, z\}$
2) 确定并排除其中最大、最小值
3) 输出剩下的元素

无论输入规模$n$多大，上述算法需要执行时间不变：

$$T(n) = c = O(1) = \Omega(1) = \Theta(1)$$

② 起泡排序

给定$n$个整数，按非降序排列。

有序序列，任意一对相邻元素顺序；无序序列，总有一对相邻元素逆序

扫描交换：依次比较每一对相邻元素，如有必要，交换之。若一趟扫描都没有交换，排序完成；否则，进行下一趟扫描交换。

冒泡排序 Python 实现：

```python
def bubble_sort(arr):
    for i in range(len(arr)-1, 0, -1):
        exchange = False
        for j in range(i):  # 一趟扫描交换
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                exchange = True
        if not exchange:  # 如果一趟扫描没有一次交换，说明有序，直接结束
            return
```

最坏时间复杂度为$O(n^2)$。

### ❀ 1.4.4 算法的正确性证明

不变性：经过 k 轮扫描交换后，最大的 k 个元素必然就位

单调性：经过 k 轮扫描交换后，问题的规模缩减至 n - k

正确性：经至多 n 趟扫描后，算法必然终止，且能给出正确解答

### 1.4.5 封底估算 (Back-Of-The-Envelope Calculation)

时间量的概念：

1 天 $= 24\text{hr} \times 60\text{min} \times 60\text{sec} \approx 25 \times 4000 = 10^5\text{sec}$

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

为祖国健康工作五十年 $= 1.6 \times 10^9\text{sec}$

三生三世 $= 300\text{yr} = 10^{10} sec$

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

**例**：考察对全国人口普查数据的排序$n = 10^9$

可从**硬件**机器上做改进和**算法**上做改进：

| 算法 \ 硬件 | 普通 PC 1GHz $10^9$flops | 天河 1A $10^{15}$flops |
|---|---|---|
| bubble sort $10^{18}$ | $10^9$sec/30yr | $10^3$sec/20min |
| merge sort $30 \times 10^9$ | 30sec | $3 \times 10^{-5}$sec/0.03ms |

归并排序比冒泡排序效率提高大于$3 \times 10^7$；天河 1A 比普通 PC 效率提高$10^6$。

普通 PC 使用归并排序比天河 1A 使用冒泡排序快得多。

可见，算法改进的威力十分巨大。

**20181026**

## 1.5 迭代与递归

Someone said: "To iterate is human, to recurse, divine." 迭代乃人工，递归方神通

然而递归的效率不如迭代。

**例**：数组求和 (迭代)

问题：计算任意 $n$ 个整数之和

实现：逐一取出每个元素，累加之

```
int my_sum(int arr[], int n){
    int s = 0; // O(1)
    for (int i = 0; i < n; i++){ // O(n)
        s += arr[i]; // O(1)
    }
    return s; // O(1)
}
```

无论 arr 内容如何，都有：

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

空间复杂度通常认为是除去输入本身所占空间外用于计算所需要的额外空间。

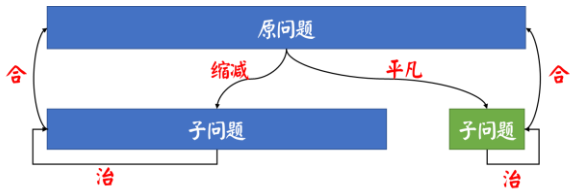
因此此处空间复杂度为$O(1)$。

### 1.5.1 减而治之(decrease and conquer)

为求解一个大规模问题，可以将其划分为两个子问题：

其中一个是退化的平凡情况，另一规模缩减，但形式与原问题一样。

分别求解子问题，由子问题的解得到原问题的解。



**例**：数组求和 (线性递归)

```
int sum(int arr[], int n){
    return (n < 1) ? 0 : sum(arr, n-1) + arr[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，计入对应子实例)，其总和即算法执行时间。
递归调用，从前往后，值返回从后往前：

$$\text{main}() \to \text{sum}(\text{arr}, \text{n}) \to \text{sum}(\text{arr}, \text{n}-1) \to \cdots \to \text{sum}(\text{arr}, 1) \to \text{sum}(\text{arr}, 0)$$

本例单个递归只需O(1)时间，$\text{T}(n) = \text{O}(1) \times (n+1) = \text{O}(n)$

递归跟踪直观形象，但仅适用于简明的递归模式。对于复杂的递归模式，需要使用递推方程(recurrence)，其特点是间接抽象。

从递推角度看，为求解 sum(arr, n)需：
1) 递归求解规模为 n-1 的问题 sum(arr, n-1) // T(n-1)
2) 累加上 arr[n-1] // O(1)
3) 递归基：sum(arr, 0) // O(1)
递推方程：$\text{T}(n) = \text{T}(n-1) + \text{O}(1)$ // recurrence
递归基：$\text{T}(0) = \text{O}(1)$ // base

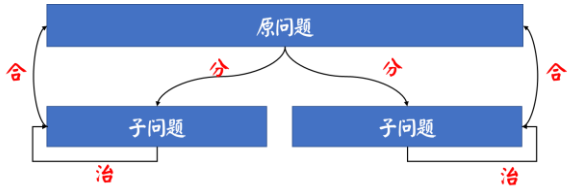
$$\text{T}(n) = \text{T}(n-1) + \text{O}(1) = \text{T}(n-2) + 2 \times \text{O}(1) = \cdots$$
$$= \text{T}(0) + n \times \text{O}(1) = (n+1)\text{O}(1) = \text{O}(n)$$

**例**：数组倒置

```
def reverse(arr, low, high): # 递归
    if low < high:  # 问题规模的奇偶性不变，需要两个递归基
        arr[low], arr[high] = arr[high], arr[low]
        reverse(arr, low+1, high-1)

def reverse1(arr, low, high): # 迭代
    while low < high:
        arr[low], arr[high] = arr[high], arr[low]
        low+=1
        high-=1
```

### 1.5.2 分而治之(divide and conquer)

为求解一个大规模问题，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。




**例**：数组求和(二分递归)

```
def my_sum(arr, low, high):  # 区间范围 arr[low, high]
    if low == high:
```

```
        return arr[low]
    mid = (low + high) // 2
    return my_sum(arr, low, mid) + my_sum(arr, mid+1, high)
```

递归跟踪：$T(n)$ =各层递归实例所需时间之和

$= O(1) \times (2^0 + 2^1 + \cdots + 2^{\log n}) = O(1) \times (2^{\log n+1} - 1) = O(n)$

递推方程：从递推角度看，为求解 sum(arr, low, high)需：

1) 递归求解 sum(arr, low, mid)和 sum(arr, mid+1, high) // 2T(n/2)
2) 将子问题的解累加 // O(1)
3) 递归基：sum(arr, low, low) // O(1)

递推关系：$T(n) = 2 \times T(n/2) + O(1)$

递归基：$T(1) = O(1)$

$T(n) = k \times T(1) + (2^k - 1)O(1), k = \log_2 n$

故$T(n) = (\log n + (n-1))O(1) = O(n)$

**例**：Max2

从数组区间 arr[low, high)找出最大两个整数 arr[a]和 arr[b]

元素比较的次数要求尽可能少

```
def max2(arr, low, high):
    a, b = low, low
    # 扫描 arr[low, high)找出最大值 arr[a]
    for i in range(low+1, high):
        if arr[i] > arr[a]:
            a = i
    # 扫描 arr[low, a)
    for i in range(low+1, a):
        if arr[i] > arr[b]:
            b = i
    # 再扫描 arr(a, high), 找出次大值 arr[b]
    for i in range(a+1, high):
        if arr[i] > arr[b]:
            b = i
    return a, b

if __name__ == '__main__':
    arr = [randint(-10, 99) for x in range(10)]
    print(arr)
    print(max2(arr, 0, len(arr)))
```

结果：

```
[13, 99, 53, 97, -8, 83, 57, 58, 26, -4]
(1, 3)
```

无论如何，比较次数总是$\Theta(2n-3)$(最好最坏情况一样)

```
def max2_2(arr, low, high):
    a, b = low, low+1
    # 起始 arr[a]为前两个较大者,arr[b]为较小者
    if arr[b] > arr[a]:
        a, b = b, a
    for i in range(low+2, high):
        # 若发现元素比 arr[b]先给 b, 再与 arr[a]比较
        if arr[i] > arr[b]:
```

```
            b = i
            if (arr[b] > arr[a]):
                a, b = b, a
    return a, b

if __name__ == '__main__':
    arr = [randint(-10, 99) for x in range(10)]
    print(arr)
    a, b = max2_2(arr, 0, len(arr))
    print(f'最大:arr[{a}]={arr[a]}, 次大:arr[{b}]={arr[b]}')
```

结果：

```
[50, 69, 36, 95, 20, 25, 1, 46, 61, 54]
最大:arr[3]=95, 次大:arr[1]=69
```

最好情况(循环一次 if)：$1+(n-2)\times 1=n-1$

最坏情况(循环两次 if)：$1+(n-2)\times 2=2n-3$

最坏情况并没有实质改进。

二分递归：

```
def max2_3(arr, low, high):
    a, b = (low, low+1) if arr[low] > arr[low+1] else (low+1, low)
    if low+2 == high:  # T(2) = 1
        return a, b
    if low+3 == high:  # T(3) <= 3
        if arr[low+2] > arr[a]:
            a, b = low+2, a
        elif arr[low+2] > arr[b]:
            b = low+2
        return a, b

    mid = (low+high)//2
    la, lb = max2_3(arr, low, mid)  # 左边最大两个
    ra, rb = max2_3(arr, mid, high)  # 右边最大两个
    if arr[la] > arr[ra]:  # 左边最大>右边最大
        return la, lb if arr[lb] > arr[ra] else ra
    return ra, rb if arr[rb] > arr[la] else la
```

$\mathrm{T}(n)=2\times\mathrm{T}(n/2)+2\le 5n/3-2$

**20181027**

## 🕮 1.6 动态规划

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

从某种意义上说，动态规划是：用递归找出算法的本质，给出初步解后，再将其有效地转为迭代的形式。

**例**：fib()

$\mathrm{fib}(n)=\mathrm{fib}(n-1)+\mathrm{fib}(n-2):\{0,1,1,2,3,5,8,\dots\}$

① 递归

```
def fib(n): # 递归
    return n if n < 2 else fib(n-1)+fib(n-2)
```

使用 Python 运行，在 n=30 左右开始变得肉眼可见的慢...

复杂度：$\mathrm{T}(0)=\mathrm{T}(1)=1;\ \mathrm{T}(n)=T(n-1)+\mathrm{T}(n-2)+1, n>1$

令$\mathrm{S}(n)=[\mathrm{T}(n)+1]/2$

则$S(0)=1=\text{fib}(1), S(1)=1=\text{fib}(2)$

$2S(n)-1=2S(n-1)-1+2S(n-2)-1+1$

即：$S(n)=S(n-1)+S(n-2)=\text{fib}(n+1)$

所以：$T(n)=2S(n)-1=2\text{fib}(n+1)-1=O(\text{fib}(n+1))$

$$\text{fib}(n)\approx\Phi^n,\Phi=\frac{1+\sqrt{5}}{2}\approx 1.61803$$

∴ $T(n)=O(\Phi^n)=O(2^n)$

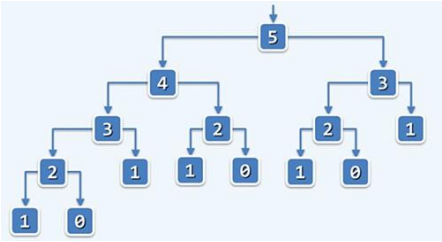
封底估算：$\Phi^{36}\approx 2^{25},\Phi^{43}\approx 2^{30}\approx 10^9\text{flo}\approx 1\text{sec}$

$\Phi^5\approx 10,\Phi^{67}\approx 10^{14}\text{flo}=10^5\text{sec}\approx 1\text{day}$

$\Phi^{92}\approx 10^{19}\text{flo}=10^{10}\text{sec}\approx 10^5\text{day}\approx 3\text{century}$

这说明该算法不好，严格说它甚至不是一个算法。

递归版 fib()低效的根源：各递归实例均被大量重复地调用。




② 迭代

1) 记忆(memoization)

将已计算的实例结果制表(全局数组)备查。

2) 动态规划(dynamic programming)

颠倒计算方向：由自顶而下递归，变为自底而上迭代

```
def fib_2(n):  # 迭代
    a, b = 0, 1
    while n > 0:
        a, b = b, a+b  # a, b 交替滚动向前走
        n -= 1
    return a
```

复杂度$T(n)=O(n)$，且只需$O(1)$空间。

**例**：最长公共子序列 LCS

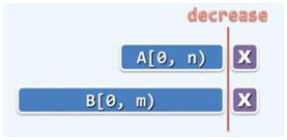
子序列(subsequence)：由序列中若干字符，按原相对次序构成

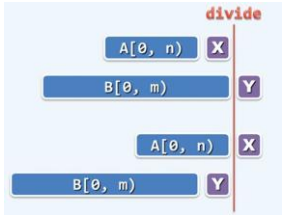
最长公共子序列(Longest Common Subsequence)：两个序列公共子序列中的最长者。(1) 可能有多个：如"education"和"advantage"的 LCS 可以是"data"或"dana"；(2) 可能有歧义：如"didactical"和"advantage"的 LCS 是"data"，但其中"d"是来自"didactical"第 1 个"d"还是第 2 个"d"?


简单版本为：只需计算 LCS 的长度

① 递归

对于序列 A[0, n]和 B[0, m]，LCS(A, B)有 3 种情况：
1) n = -1 或 m = -1，某个序列为空序列，则结果为空序列"" //递归基
2) 若 A[n]=B[m]='X'，则取 LCS(A[0, n), B[0, m))+'X' //减而治之
3) 若 A[n]≠B[m]，则在 LCS(A[0, n], B[0, m))与 LCS(A[0, n), B[0, m])取更长者




```python
def lcs(a, b):
    def lcs_rec(a, b, n, m):
        if n < 0 or m < 0:
            return 0
        if a[n] == b[m]:
            return lcs_rec(a, b, n-1, m-1)+1 # 最后字符相等，减而治之
        return max(lcs_rec(a, b, n, m-1), lcs_rec(a, b, n-1, m)) # 分而治之
    return lcs_rec(a, b, len(a)-1, len(b)-1)

if __name__ == '__main__':
    a = 'educational'
    b = 'advantage'
    print(lcs(a, b))  # 4
```

单调性：无论如何，每经一次比较，原问题的规模必可减小
具体地，作为输入的两个序列，至少其一的长度缩短一个单位

最好情况(只出现减而治之不出现分而治之)，只需$O(n+m)$时间
但问题在于，一旦有分而治之的情况出现，原问题将分解为两个子问题
更糟的是，它们的规模之和是原来的两倍，而且在随后进一步导出的子问题，也有类似情况，造成大量的雷同。

最坏情况：LCS(A[0, a], B[0, b])出现次数为：

$$C_{n+m-a-b}^{n-a} = C_{n+m-a-b}^{m-b}$$

特别地，LCS(A[0], B[0])的次数可多达：

$$C_{n+m}^{n} = C_{n+m}^{m}$$

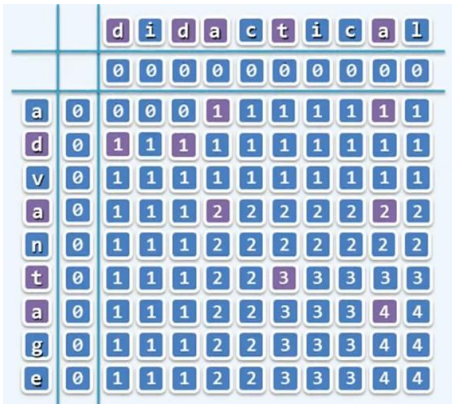
当$n = m$时，复杂度为$O(2^n)$。

② 迭代
与 fib()类似，LCS 的递归也有大量重复的递归实例(子问题)
最坏情况，共计出现$O(2^n)$个。

各子问题，分别对应于 A 和 B 的某个前缀组合，因此总数不过是$n \times m$种。
采用动态规划的策略，只需$O(n \times m)$时间即可计算出所有子问题。

1) 将所有子问题(假想地)列成一张表
2) 颠倒计算方向，从 LCS(A[0], B[0])出发，依次计算出所有项。

$$C[i][j]=\begin{cases}0 & i=0 \text{ or } j=0\\ C[i-1][j-1]+1 & i,j>0, A[i]=B[j]\\ \max\{C[i][j-1],C[i-1][j]\} & i,j>0, A[i]\neq B[j]\end{cases}$$




```python
def lcs_2(a, b):
    n, m = len(a), len(b)
    # 子问题假想表格(n+1)(m+1), 初始全部填 0
    c = [[0 for j in range(m+1)] for i in range(n+1)]
    # 从上到下从左往右遍历表格
    for i in range(1, n+1):
        for j in range(1, m+1):
            if a[i-1] == b[j-1]:  # 减而治之,左上角+1
                c[i][j] = c[i-1][j-1]+1
            # 分而治之,左边或上面较大者
            else:
                c[i][j] = max(c[i][j-1], c[i-1][j])
    # 表格右下角即为最终结果
    return c[n][m]

if __name__ == '__main__':
    a = 'hoshizora rin'
    b = 'nishikino maki'
    print(lcs(a, b))  # 6
    print(lcs_2(a, b))  # 6
```

**总结**：利用递归设计出可行且正确的解；在此基础上使用动态规划消除重复计算，提高效率。

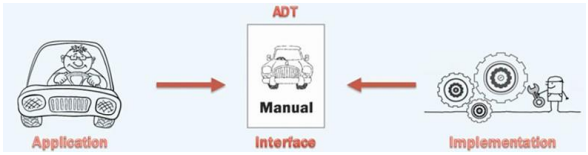
**20181030**

## 第 2 章　向量

### 🕮　2.1　接口与实现

抽象数据类型(Abstract Date Type)：数据模型+定义在该模型上的一组操作

数据结构(Data Structure)：基于某种特定语言，实现 ADT 的一整套算法




用户只关心提供的功能；实现者负责具体内部实现，说明书就是实现者与用户之间达成的协议、规范。

最基本的线性结构统称为序列(sequence)，根据数据项的逻辑次序与物理存储地址对应关系不同，分为向量(vector)和列表(list)。

C/C++语言，数组 A[]中的元素与[0, $n$)的编号一一对应。第 $n$ 号元素虽然不一定存在，此处虚拟地放在后面，作为哨兵。


反之，每个元素均可由编号唯一指代，并可直接访问：

A[$i$]的物理地址= $A + i \times s$, s为单个元素所占空间

所以也称为线性数组(linear array)。

向量(vector)是 C++等高级语言中数组这种数据结构的推广和泛化，由一组元素按线性次序封装而成：

(1) 各元素与[0, n)内的秩(rank, 就是索引)一一对应 //循秩访问(call-by-rank)
(2) 元素的**类型**不限于基本类型
(3) 操作、管理、维护更加简化、同一、安全
(4) 更为方便地参与复杂数据结构的定制与实现

按照 ADT 的规范，向量结构必须提供一系列的操作接口：

| **操作** | **功能** |
|---|---|
| size() | 元素总数 |
| get(r) | 获取 r 号元素 |
| put(r, e) | r 号元素替换为 e |
| insert(r, e) | r 号位插入 e，后继元素后移 |
| remove(r) | 删除并返回 r 号元素，后继元素前移 |
| disordered() | 判断是否有序(非降序)，返回逆序对数目 |
| sort() | 排序(非降序) |
| find(e) | 查找目标元素 e，返回 rank，不存在返回-1 |
| search(e) | 查找目标元素 e，返回不大于 e 且 rank 最大的元素(有序向量) |
| deduplicate() | 删除重复元素 |
| uniquify() | 删除重复元素(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 |

设计接口时，暂时不关系具体实现方法，只关心操作语义。

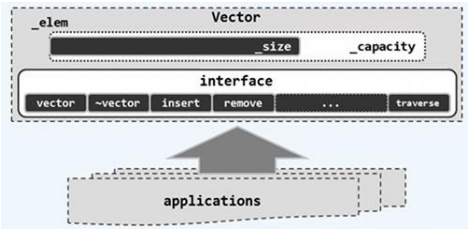
① Vector 模板类：

```
typedef int Rank; // 秩
#define DEFAULT_CAPACITY 3 // 默认初始容量(实际应用可设更大)
template <typename T> class Vector{ // 向量模板类
    private:
        Rank _size; // 元素个数
        int _capacity; // 容量
        T* _elem; // 数据区
    protected:
        // 内部函数
    public:
        // 构造函数
        // 析构函数
```

```
    // 只读接口
    // 可写接口
    // 遍历接口
};
```

定义一个元素类型为 T 的 Vector 模板类(//泛型?)。

Vector 类被封装，对用户提供一些接口。用户通过接口使用，而不需要了解内部实现：




② 构造函数和析构函数：

```
    public:
        // 构造函数
        Vector(int c=DEFAULT_CAPACITY){
            _elem = new T[_capacity=c]; // 申请长度为 c 类型为 T 的一段连续空间
            _size = 0; // 元素个数为 0
        }
        Vector(T const * A, Rank low, Rank high){ // 数组区间复制
            copyFrom(A, low, high); // 需要实现
        }
        Vector(Vector<T> const& V, Rank low, Rank high){ // 向量区间复制
            copyFrom(V._elem, low, high);
        }
        Vector(Vector<T> const& V){ // 向量整体复制
            copyFrom(V._elem, 0, V._size);
        }
        // 析构函数
        ~Vector(){ // 释放内部空间
            delete [] _elem;
        }
```

③ copyFrom()函数

//写在类外部，内部需要声明，该函数不是外部接口，可设为 protected

```
template <typename T> // 元素类型(T 为基本类型或已重载赋值操作符=)
void Vector<T>::copyFrom(T const * A, Rank low, Rank high){
    _elem = new T[_capacity=2*(high-low)]; // 分配 2 倍空间
    _size = 0;
    while (low < high){ // 将 A[low, high)元素逐一复制到_elem[0, high-low)
        _elem[_size++] = A[low++];
    }
}
```

## 🕮 2.2 可扩充向量

### ❀ 2.2.1 静态空间管理

开辟内部数据_elem[]并使用一段地址连续的物理空间。

_capacity：总容量；_size：当前元素个数

若采用静态空间管理策略，容量_capacity 固定，则有明显不足：
1) 上溢(overflow)：_elem[]不足以存放所有元素，尽管系统有足够空间
2) 下溢(underflow)：_elem[]中的元素寥寥无几，造成空间浪费
更糟的是，一般很难准确预测空间的需求量。

装填因子(load factor)：λ=_size / _capacity，是衡量空间利用率的重要指标。
也就是需要保证向量的装填因子既不超过 1，也不太接近于 0。

### ❀ 2.2.2 动态空间管理

**蝉的哲学**：身体每经过一段时间，以致无法为外壳容纳，即蜕去原先的外壳，代之以一个新的外壳。
同理，对于向量，在即将发生上溢时，适当地扩大内部数组的容量，使之足以容纳新的元素，即可扩充向量(extendable vector)。

扩容算法实现：expand()函数

```
template <typename T>
void Vector<T>::expand(){ // 向量空间不足时扩容
    if (_size < _capacity)
        return; // 未满不必扩容
    _capacity = max(_capacity, DEFAULT_CAPACITY); // 不低于最小容量
    T* oldElem = _elem; // 备份
    _elem = new T[_capacity <<= 1]; // 容量加倍
    for (int i = 0; i < _size; i++){ // 复制原来内容
        _elem[i] = oldElem[i];
    }
    delete [] oldElem; // 释放原来空间
}
```

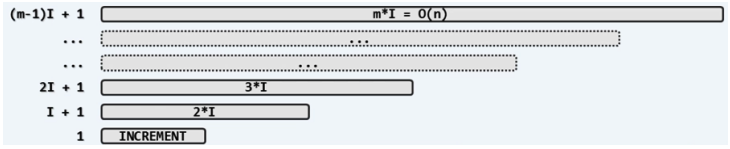
扩容后数据区地址由操作系统分配，与原来没有直接关系。此时，若直接引用数组，往往会导致共同指向原数组的其它指针失效，成为野指针(wild pointer)；而封装为向量后，可继准确地引用各元素，从而有效避免了野指针的风险。

### ❀ 2.2.3 扩容策略

① 容量递增策略

```
    T* oldElem = _elem;
    _elem = new T[_capacity += INCREMENT]; // 追加固定大小的容量
```

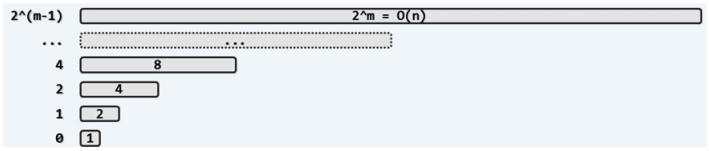
最坏情况在初始容量为 0 的空向量中，连续插入$n = m \times I$ (>>2)个元素：
在第 1、I + 1、2I + 1、3I + 1...次加入时都需要扩容。
即使不计申请空间操作，各次扩容过程复制原向量的时间成本依次为：
0、I、2I、3I...($m$-1)I　//算术级数
总耗时$= I \times (m-1)/2 = O(n^2)$，每次扩容的分摊成本为$O(n)$。




② 容量加倍策略

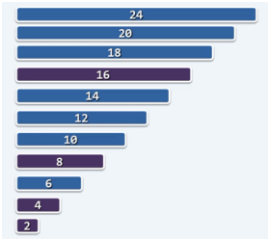
```
    T* oldElem = _elem; // 备份
    _elem = new T[_capacity <<= 1]; // 容量加倍
```

最坏情况：在初始容量为 1 的满向量中，连续插入$n = 2^m$ (>>2)个元素：
在第 1、2、4、8、16...次插入时都需要扩容。
各次扩容过程复制原向量的时间成本依次为：
1、2、4、8、16... $2^m$ //几何级数
总耗时$= O(n)$，每次扩容的分摊成本$O(1)$。




两种策略对比：

| | 递增策略 | 倍增策略 |
|---|---|---|
| 累计扩容时间 | $O(n^2)$ | $O(n)$ |
| 分摊扩容时间 | $O(n)$ | $O(1)$ |
| 装填因子 | $\approx 100\%$ | $> 50\%$ |




倍增策略在空间效率上做出适当牺牲，换取时间方面的巨大收益。

### ⚽ 2.2.4 分摊分析

平均复杂度(average/expected complexity)
根据数据结构各种操作出现概率的分布，将对应的成本加权平均。
各种可能的操作作为**独立事件**分别考察，割裂了操作之间的相关性和连贯性，往往不能准确评判数据结构和算法的真实性能。

分摊复杂度(amortized complexity)
对数据结构**连续**地实施**足够多**次操作，所需的总体成本分摊至单次操作。
从**实际可行**的角度，对一系列操作做整体考量，更加真实刻画了可能出现的操作序列，可以更精准地评判数据结构和算法的真实性能。

// 动态缩容算法实现：shrink()

**20181031**

## 🕮 2.3 无序向量

### ⚽ 2.3.1 元素访问

通过 get(r)和 put(r, e)接口可以读写向量元素；但就便捷性而言，远不如数组元素访问形式：A[r]。为了沿用下标访问，需要重载下标操作符[]：

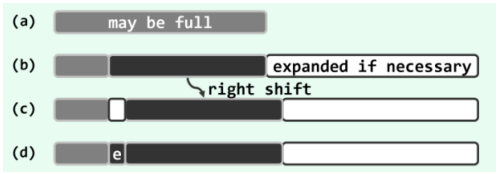
```
template <typename T>
T &Vector<T>::operator[](Rank r) const { //重载下标操作符,返回引用:T&,可作为左值
    // assert 0 <= r < _size; // 此处对于意外和错误简单处理,实际应用需严格
    return _elem[r];
}
```

main()函数测试：

```
int main(int argc, char const *argv[]){
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    Vector<int> myvector(arr, 0, 5); // 选取前 5 个
```

```
    myvector.show(); // [1, 2, 3, 4, 5]
    int a = myvector[1]; // 右值
    cout << "a = " << a << endl; //a = 2
    myvector[2] = 33; // 左值,因为返回引用,所以可以修改元素的值
    myvector.show();  //[1, 2, 33, 4, 5]
    return 0;
}
```

### ❀ 2.3.2 插入



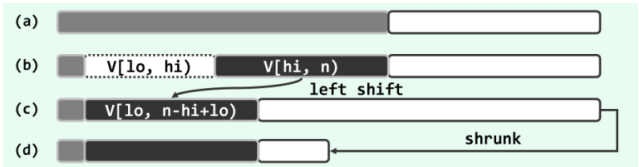

```
template <typename T>
Rank Vector<T>::insert(Rank r, T const &e) // r 处插入元素 e, 0<=r<=_size
{
    expand(); // 如果满了就扩容
    for (int i = _size; i > r; i--) // 自后向前,r 后的元素后移一个单元
        _elem[i] = _elem[i - 1];
    _elem[r] = e; // r 处插入 e
    _size++;      // 更新元素个数
    return r;     // 返回秩
}
```

main()函数测试：

```
int main(int argc, char const *argv[]){
    // ...接着上面继续测试
    myvector.insert(2, 99); // 2 号位插入 99
    myvector.show(); // [1, 2, 99, 33, 4, 5]
    cout << "size = " << myvector.size() << endl; // size = 6
    return 0;
}
```

### ❀ 2.3.3 删除

① 区间删除




```
template <typename T> //删除区间[low, high)
int Vector::remove(Rank low, Rank high){
    if (low == high)
        return 0;
    while (high < _size) // [high,_size)元素向前向后,往前移动 high-low 位
        _elem[low++] = _elem[high++];
    _size = low;       // 循环结束时 low 就是新的元素个数
    shrink();          // 如有必要则缩容
    return high - low; // 返回删除元素个数
}
```

动态缩容 shrink()：//一般不是必须的

```cpp
template <typename T>
void Vector<T>::shrink(){ //装填因子过小时压缩所占空间
    if (_capacity < DEFAULT_CAPACITY << 1)
        return; //不会收缩到 DEFAULT_CAPACITY 以下
    if (_size << 2 > _capacity)
        return;                    //以 25%为界,装填因子>25%不缩容
    T *oldElem = _elem;             //备份
    _elem = new T[_capacity >>= 1]; //容量减半
    for (int i = 0; i < _size; i++)
        _elem[i] = oldElem[i]; //复制原来内容
    delete[] oldElem;          //释放原空间
}
```

测试：

```cpp
int main(int argc, char const *argv[]){
    // ...接着上面继续测试
    myvector.remove(2, 4);                      // 删除[2, 4)号元素
    myvector.show();                            // [1, 2, 4, 5]
    cout << "size = " << myvector.size() << endl; // size = 4
    return 0;
}
```

### ② 单元素删除

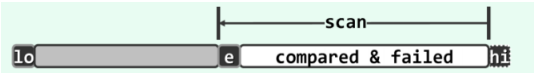
可以视为区间删除的特例。删除秩为 r 的元素，相当于区间删除[r, r+1)。

```cpp
template <typename T> //删除秩为 r 的元素
T Vector<T>::remove(Rank r)
{
    T e = _elem[r];   // 记录删除元素
    remove(r, r + 1); // 调用区间删除
    return e;         // 返回被删除元素
}
```

所以说为什么不反过来，基于单元素删除，通过反复调用，实现区间删除呢?
单元素删除耗时主要是后继元素往前移操作，正比于删除元素后缀的长度$= n - r = O(n)$；循环次数为区间宽度$= high - low = O(n)$。
基于单元素删除的区间删除复杂度为$O(n^2)$；而之前的区间删除复杂度为$O(n)$。

## 2.3.4 查找

不是所有类型都支持判等和比较。假设：
无序向量：T 为可判等的基本类型，或已重载操作符==或!=
有序向量：T 为可比较的基本类型，或已重载操作符<或>




```cpp
template <typename T> // 无序向量的逆序查找:返回元素 e 的最后位置,不存在返回 low-1
Rank Vector<T>::find(T const &e, Rank low, Rank high) const{
    while ((--high >= low) && (e != _elem[high])) // 从后向前
        ;
    return high; // 返回 high 是元素的秩(索引),如果元素不存在则 high=low-1
}
```

是否查找成功，交由上层调用者判断。
该查找算法，最好情况：一上来就命中$O(1)$；最坏情况：遍历完$O(n)$。
具体复杂度与输入相关，此类算法称为输入敏感(input-sensitive)的算法。

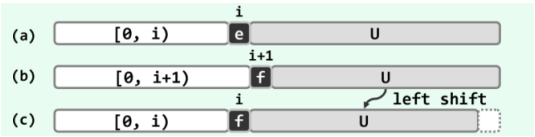
### ❁ 2.3.5 唯一化

去除向量中重复元素。

```
template <typename T> // 删除无序向量中重复元素
int Vector<T>::deduplicate()
{
    int oldSize = _size; // 记录原来元素个数
    Rank i = 1;          // 从_elem[1]开始查重
    while (i < _size)    // 从前向后考察每个元素
    {
        if (find(_elem[i], 0, i) < 0) // 该元素在前面有没有出现(至多一个)
            i++;                      // 没有重复,指针向后移
        else
            remove(i); // 有重复,删除该位置元素
    }
    return oldSize - _size; // 返回被删除元素个数
}
```

**不变性**：在循环中，当前元素_elem[i]的前缀_elem[0, i)中各元素彼此互异

初始 i=1 时，只有唯一前驱_elem[0]，自然成立。一般情况：




如图(a)，假设_elem[i]前缀元素互异。经过此步迭代有两种结果：

(1) e 与前缀_elem[0, i)任何元素不相同，如图(b)。i++后，新的前缀_elem[0, i)同样满足不变性，其规模增加一个单位。

(2) 前缀含有与 e 相同元素，如图(c)，由前面满足的不变性可知，相同的元素最多只有一个。删除 e 后，前缀_elem[0, i)依然保持不变性。

**单调性**：随着反复迭代：

(1) 当前元素前缀的长度单调非降，迟早增至_size

(2) 当前元素后缀的长度单调下降，迟早减至 0

故算法必然终止，且最多迭代$O(n)$轮。

**复杂度**：

find()对于前缀查找，remove()对于后缀元素往前移，故每轮迭代 find()和 remove()累计消耗$O(n)$时间，总体位为$O(n^2)$

进一步优化：

(1) 仿照有序向量的 uniquify()高效版思路，元素移动次数降至$O(n)$，但比较次数仍是$O(n^2)$，而且稳定性将被破坏。

(2) 先对需要删除的元素做标记，最后统一删除。稳定性保持，但查找长度更长，导致更多的比对操作。

(3) 先 sort()变为有序向量，再 uniquify()：简明实现最优$O(n \log n)$。

### ❁ 2.3.6 遍历

统一对各元素分别实施 visit 操作(事先约定的操作)。

如何指定 visit 操作?如何将其传递到向量内部?
① 利用函数指针机制，只读或局部性修改

```
template <typename T>
void Vector<T>::traverse(void (*visit)(T &)) //函数指针
{
    for (int i = 0; i < _size; i++)
        visit(_elem[i]);
}
```

② 利用函数对象机制，可全局性修改

```
template <typename T>
template <typename VST>
void Vector<T>::traverse(VST &visit) //函数对象
{
    for (int i = 0; i < _size; i++)
        visit(_elem[i]);
}
```

后者的功能更强，适用范围更广。

**实例**：将向量所有元素加一
首先需要实现一个可使单个 T 类型元素加一的类：

```
template <typename T> // 假设 T 可直接递增或已重载操作符++
struct Increase       // 简化起见使用 struct
{                    // 函数对象,通过重载操作符"()"实现,行为上类似于函数
    virtual void operator()(T &e) { e++; }
};
```

定义一个 increase()函数：

```
template <typename T>
void increase(Vector<T> &v){
    v.traverse(Increase<T>());
}
```

// 运行报错...
no known conversion for argument 1 from ‘Increase<int>’ to ‘Increase<int>&’

**20181101**

♏ 凛ちゃん、お誕生日おめでとう。

にゃにゃにゃにゃー

## 🕮 2.4 有序向量 (sorted vector)

### ❀ 2.4.1 有序性

有序序列任意一对相邻元素都顺序；无序序列总存在一对相邻元素逆序。
因此，相邻逆序对的数目，可以度量向量的**逆序**程度。

判断向量是否有序(非降序)的函数 disordered()：

```
template <typename T>
int Vector<T>::disordered() const{
    int cnt = 0; //计数器
    for (int i = 1; i < _size; i++){
        if (_elem[i - 1] > _elem[i]) // 逆序计数器+1
            cnt++;
    }
    // 返回相邻逆序对总数,如果只是判断是否有序,首次遇到逆序对即可结束
    return cnt;
}
```

测试：

```
    // 接着上面继续测试...
    myvector.show(); // [1, 2, 4, 5]
    cout << "逆序对 = " << myvector.disordered() << endl; // 逆序对 = 0
    myvector.insert(0, 99);
    myvector.show(); // [99, 1, 2, 4, 5]
    cout << "逆序对 = " << myvector.disordered() << endl; // 逆序对 = 1
```

无序向量经预处理可转换为有序向量，相关算法大多可以优化。

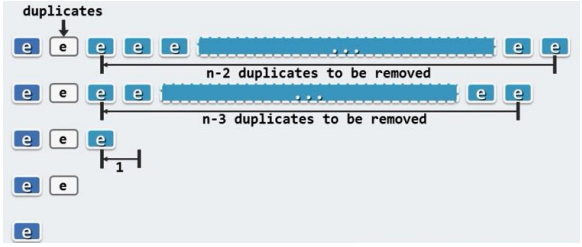
### ❀ 2.4.2 唯一化

#### ① 低效算法

在有序向量中，重复的元素必然相互紧邻构成一个区间，因此每个区间只需保留一个元素即可。扫描一遍向量，比较相邻元素，如果相同则删除后一个元素。

```
template <typename T>
int Vector<T>::uniquify(){
    int oldSize = _size; // 记录原来元素个数
    for (int i = 0; i < _size - 1;){ // 从前往后逐一比对相邻元素
        if (_elem[i] == _elem[i + 1])
            remove(i + 1); // 若相等,删除后者
        else
            i++;
    }
    // 返回删除元素个数;_size 的变化由 remove()完成
    return oldSize - _size;
}
```

复杂度：



循环次数$n-1$，最坏情况(所有元素相同)每次都要调用 remove()
耗时$O(n-1)\sim O(1)$；累计$O(n^2)$。与无序向量的 deduplicate()相比，尽管省去 find()，但复杂度相同，非常低效。

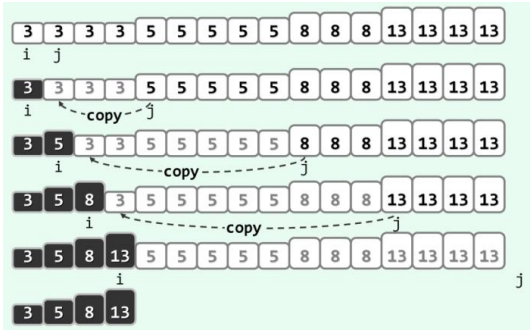
### ② 高效算法

低效算法的根源在于，同一元素可作为被删除元素的后继多次前移。
如果能以重复区间为单位，成批删除相同元素，性能必将改进。

如图_elem[i]后面元素_elem[i+1, j-1)都与_elem[i]相同，而_elem[j]不同。只需将_elem[j]移到_elem[i+1]位，就相当于删除了重复元素。


```
template <typename T>
int Vector<T>::uniquify(){
    Rank i = 0, j = 0;
    while (++j < _size){ // 逐一扫描直至末尾
        if (_elem[i] != _elem[j])
            _elem[++i] = _elem[j]; // 与 i 位第一个不同元素,移到 i+1 处
    }
    _size = ++i; // 直接截断尾部多余元素
    shrink();
    return j - i; // 删除元素个数
}
```

如图是某个有序向量去重的实例：




其中并没有做显式的删除操作，而是将不同元素复制到前面；循环结束截断尾部多余元素(重新设置_size)，从而达到删除多余元素的目的。

总共$n-1$次迭代，每次常数时间，累计$O(n)$时间。

## 2.4.3 查找

### ① 统一接口

```
template <typename T> //  查找算法统一接口, 0<=low<high<=_size
Rank Vector<T>::search(T const &e, Rank low, Rank high) const
{ // 50%概率随机选取: 二分查找 or 斐波那契查找
    return (rand() % 2) ? binSearch(_elem, e, low, high)
                        : fibSearch(_elem, e, low, high);
}
```

如何处理特殊情况：如目标元素不存在，或存在多个?

② 语义约定

应该便于有序向量自身的维护，如 v.insert(v.search(e)+1, e)
即使查找失败，也应该给出新元素适当的插入位置；
若允许重复元素，则每一组也需按其插入的次序排列。

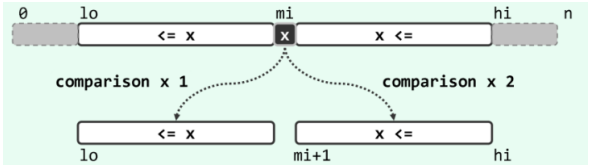
**约定**：在有序向量区间[low, high)，确定不大于 e 的最后一个元素
若$-\infty < e < v[low]$，返回$low - 1$ (左侧哨兵)
若$v[high - 1] < e < +\infty$，返回$high - 1$ (末元素=右侧哨兵左邻)

这样，如果特别小的元素返回 low-1，插入在 low 处；
特别大的元素返回 high-1，插入在 high 处；
相同元素的话返回区间最后位置，插入在其后一个位置。
所以，新元素插入在查找返回值+1 的位置，而且相同元素，后插入的在后面。
这种语言约定非常合理。

### 2.4.4 二分查找 版本 A (binary search version A)

原理：减而治之
以任一元素 x = arr[mid]为界(轴点)，都可以将待查区间分为三部分：
$arr[low, mid) \le arr[mid] \le arr(mid, high)$
只需将目标元素 e 与 x 逐一比较，即可分为三种情况：
1) $e < x$：e 若存在，必在左侧子区间 arr[low, mid)，递归深入
2) $e > x$：e 若存在，必在右侧子区间 arr(mid, high)，递归深入
3) e==x：命中目标，直接返回 //若有多个，返回哪个?

二分(折半)策略：轴点 mid 总是取中点，于是每经过至多两次比较，或能命中，或将问题规模缩减一半。




```
template <typename T> // 有序向量区间[low, high)二分查找
static Rank binSearch(T *A, T const &e, Rank low, Rank high){
    while (low < high){
        Rank mid = (low + high) >> 1; // 轴点为中点
        if (e < A[mid])
            high = mid;      // 前半段[low, mid)查找
        else if (e > A[mid]) // 建议写成:A[mid] < e
            low = mid + 1;   // 后半段(mid, high)查找
        else
            return mid;
    }
    return -1; // 查找失败，返回-1 是不够的
}
```

线性递归：经过至多两次比较可将问题规模 $n$ 减半，即：$T(n) = T(n/2) + O(1) = O(\log n)$，大大优于顺序查找$O(n)$。

为了更为精细地评估查找算法的性能，可以考查元素大小的比较次数，即查找长度(search length)。

通常，需分别针对成功与失败查找，从最好、最坏、平均等角度评估。
如版本 A 的成功、失败的平均查找长度大致为$O(1.5 \cdot \log n)$。

尽管二分查找版本 A 最坏也能保证$O(\log n)$的渐进时间复杂度，但其常系数 1.5 仍有改进余地。因为其每一步迭代为了确定左、右子区间，需要做 1 或 2 次元素比较，所以不同情况对应查找长度不均衡。

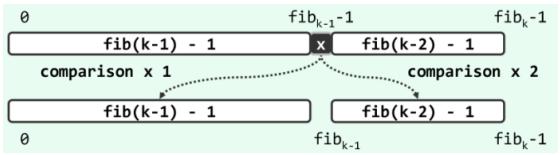
**20181103**

### ⊛ 2.4.4 Fibonacci 查找

二分查找版本 A 的效率仍有改进余地，因为其中转向左、右分支前的关键码**比较次数**不等(向左+1、向右+2)，而**递归深度**却相同。

若能通过递归深度的不均衡，对转向成本的不均衡进行补偿，平均查找长度应能进一步缩短。如[low, mid)区间加长，(mid, high)区间缩短。

因此子向量切割点 mid 不必居中，不妨按黄金分割比确定 mid。
设向量长度$n = \text{fib(k)} - 1$，则可取$mid = \text{fib(k} - 1) - 1$，于是前、后子向量的长度分别为$\text{fib(k} - 1) - 1$、$\text{fib(k} - 2) - 1$：




```
template <typename T> // 有序向量区间[low, high)fibonacci 查找
static Rank fibSearch(T *A, T const &e, Rank low, Rank high){
    cout << "fibonacci search..." << endl;
    Fib fib(high - low); // 用 O(logn)时间创建 Fib 数列,不小于 n 的最小 fib
    while (low < high){
        while (high - low < fib.get())
            fib.prev(); // 向前顺序查找,分摊 O(1)
        Rank mid = low + fib.get() - 1; // 按黄金比例切分
        if (e < A[mid])
            high = mid; // 深入前半段[low, mid)
        else if (A[mid] < e)
            low = mid + 1; // 深入后半段(mid, high)
        else
            return mid;
    }
    return -1;
}
```

fibonacci 序列类：

```
class Fib{ //fibonacci 序列类
  private:
    int f, g; //f=fib(k-1), g=fib(k),均为 int,很快就会溢出
  public:
```

```
    Fib(int n){ //初始化为不小于 n 的最小 fib 项
        f = 1;
        g = 0;
        while (g < n)
            next();
    } //fib(-1), fib(0), O(logn)时间
    int get(){ //获取当前 fib 项, O(1)时间
        return g;
    }
    int next(){    //转至下一 fib 项, O(1)时间
        g += f;    // 下一个 fib
        f = g - f; // 原来的 g
        return g;
    }
    int prev(){    //转至上一 fib 项, O(1)时间
        f = g - f; // 前两个 fib
        g -= f;    // 原来的 f, 前一个 fib
        return g;
    }
};
```

main()函数测试：

```
    time_t t = time(0); // #include <ctime>
    srand((unsigned)t); // 以当前时间戳为 rand()函数生成随机数种子
    int arr[] = {2, 5, 6, 8, 11, 14, 23, 34, 56, 67, 85};
    int length = sizeof(arr) / sizeof(int);
    Vector<int> myvector(arr, 0, length);
    myvector.show();
    int index = myvector.search(5, 0, length);
    cout << "index of 5: " << index << endl;
```

结果：

```
[2, 5, 6, 8, 11, 14, 23, 34, 56, 67, 85]
fibonacci search...
index of 5: 1
```

Fibonacci 查找的平均查找长度(average search length, AVL)，在常系数意义下优于二分查找。

通用策略：对于任何A[0, n)，总选取A[λn]作为轴点，$0 \le \lambda < 1$

如：二分查找对应于$\lambda = 0.5$，Fibonacci 查找对应于$\lambda = \varphi = 0.6180339\cdots$

在[0, 1)内λ取何值才能达到最优？设平均查找长度为$\alpha(\lambda) \cdot \log_2 n$，何时$\alpha(\lambda)$最小？


递推式：

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

整理后：

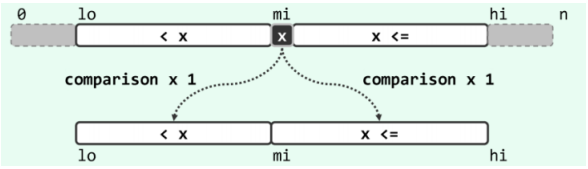
$$\frac{-\ln 2}{\alpha(\lambda)} = \frac{\lambda \ln \lambda + (1 - \lambda) \ln (1 - \lambda)}{2 - \lambda}$$

当$\lambda = \varphi$时，$\alpha(\lambda) = 1.440420\cdots$达到最小。相比于二分查找的 1.5，效率略有提高，但不改变算法的框架的话，提升已经是极限了。

### ֍ 2.4.4 二分查找 版本 B (改进版)

二分查找中左右分支转向代价不平衡的问题，也可直接解决。比如每次迭代仅做 1 次关键码比较，这样的话，所有分支只有 2 个方向，而非 3 个。

轴点 mid 同样取中点，查找每深入一层，问题规模也缩减一半。
1) e < x：e 若存在，必在左侧子区间 arr[low, mid)，递归深入
2) e > x：e 若存在，必在右侧子区间 arr[mid, high)，递归深入
只有当 high - low=1 时，才判断元素是否命中。




```
template <typename T> // 有序向量区间[low, high)二分查找改进版
static Rank binSearch(T *A, T const &e, Rank low, Rank high){
    while (1 < high - low){ // 有效查找区间宽度缩减到 1 时,退出循环
        Rank mid = (low + high) >> 1; // 轴点为中点
        if (e < A[mid])
            high = mid; // 前半段[low, mid)查找
        else
            low = mid; // 后半段[mid, high)查找
    }
    // 循环结束,high=low+1,查找区间只有一个元素 A[low]
    return (e == A[low]) ? low : -1;
}
```

相对于版本 A，该版本最好情况变坏(版本 A 最好一发命中)，最坏情况变好；各种情况的查找长度更接近，整体性能更趋于稳定。

### ֍ 2.4.5 二分查找 版本 C (终极版)

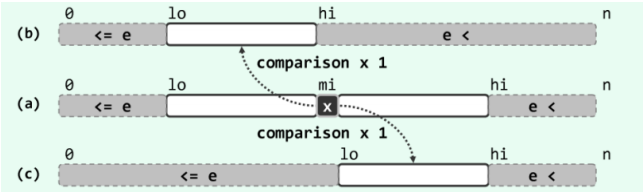
以上二分查找和 Fibonacci 查找算法均未严格实现 search()接口的语义约定，即返回不大于 e 的最后一个元素。

只有实现这一约定，才能支持相关算法，如：v.insert(1 + v.search(e), e)
1) 当有多个命中时，返回最后面的(秩最大)；
2) 失败时，返回小于 e 的最大者(含哨兵 low-1)

```
template <typename T> // 有序向量区间[low, high)二分查找终极版
static Rank binSearch(T *A, T const &e, Rank low, Rank high){
    while (low < high){ // 不变性: A[0, low) <=e <A[high, n)
        Rank mid = (low + high) >> 1; // 轴点为中点
        if (e < A[mid])
            high = mid; // 前半段[low, mid)查找
        else
            low = mid + 1; // 后半段(mid, high)查找
    }
    // 循环结束,low=high,A[low]为大于 e 的最小元素
    return low - 1; //low-1 是不大于 e 的元素最大秩
}
```

**正确性**
在循环中，存在**不变性**：A[0, low) <= e < A[high, n)

初始时，$\text{low}=0, \text{high}=\text{n}, \text{A}[0,\text{low})=\text{A}[\text{high},\text{n})=\emptyset$，自然成立
数学归纳：假设不变性一直保持到如图(a)，根据 e<A[mid]?分为：
1) e<A[mid]：如图(b)，令 high=mid，右侧区域 $\text{A}[\text{high},\text{n})$向左扩展，该区间元素都不小于 A[mid]，自然大于 e；
2) A[mid]<=e：如图(c)，令 low=mid+1，左侧区间 A[0,low)向右扩展，该区间元素都不大于 A[mid]，自然也不大于 e。




也就是经过一次迭代，无论哪种情况，不变性依然保持。
**单调性**：显而易见，循环结束时 low=high，搜索区间被压缩成宽度为 0 的区间(分界)。该分界左侧不大于 e，右侧严格大于 e。
最后返回分界左侧元素的秩，即 low-1。

至于判断搜索元素 e 是否在向量中，可以将所返回的秩(假如 res)，再判断 A[res]与 e 是否相等即可。

### ☢ 2.4.6 插值查找 (interpolation search)

假设已知有序向量中各元素随机分布的规律，如均匀独立的随机分布。
于是[low, high)各元素大致按照线性趋势增长：

$$\frac{\text{mid}-\text{low}}{\text{high}-\text{low}}=\frac{\text{e}-\text{A}[\text{low}]}{\text{A}[\text{high}]-\text{A}[\text{low}]}$$

因此，通过猜测轴点 mid，可以极大地提高收敛速度

$$\text{mid}=\text{low}+(\text{high}-\text{low})\cdot\frac{\text{e}-\text{A}[\text{low}]}{\text{A}[\text{high}]-\text{A}[\text{low}]}$$

例如：在英文词典中，binary 大致位于 2/26 处；search 大致位于 19/26 处。

**性能**：
1) 最坏情况$\text{O}(\text{high}-\text{low})=\text{O}(n)$，即不满足均匀独立分布。
2) 最好情况：一发命中。
3) 一般平均的情况，每经一次比较 n 缩至$\sqrt{n}$。

待查区间宽度缩减趋势：$n, n^{\frac{1}{2}}, n^{\frac{1}{2^2}}, \cdots n^{\frac{1}{2^k}}, \cdots, 2$

经多少次比较后，有$n^{\frac{1}{2^k}}<2$?

$(1/2)^k \log n<1 \Rightarrow -k\log 2+\log\log n<0 \Rightarrow k>\log\log n$
所以，平均时间复杂度为$\text{O}(\log\log n)$

**字宽折半**

$n$的二进制位宽为$\log n$，每次开根，对应二进制位宽相当于变为原来一半：

$$\log n, \frac{1}{2}\log n, \frac{1}{2^2}\log n \ldots$$

插值查找相当于对$n$的**二进制位宽**做二分查找，而二分查找迭代次数与初始值成对数关系，因此可得插值查找的迭代次数是$\log\log n$。

**对比**

然而从$O(\log n)$到$O(\log\log n)$优势不明显。除非查找区间宽度极大，或者比较操作成本极高。

例如$n = 2^{32} = 4G, \log_2 n = 32, \log_2\log_2 n = 5$，从 32 到 5 改进不是非常明显。

但是插值查找具有明显的缺陷：

1) 易受小扰动的干扰和蒙骗；
2) 需要引入乘法和除法运算，成本更高。

实际可行的方法：首先通过插值查找，将查找范围缩小到一定的范围，然后再进行二分查找。

大规模：插值查找；中规模：二分查找；小规模：顺序查找

**20181106**

## 🕮 2.5 起泡排序

排序的统一接口：

```
template <typename T>
void Vector<T>::sort(Rank low, Rank high){ // 区间[low, high)
    switch (rand() % 5){ // 视具体情况灵活选取或扩充
        case 1: bubbleSort(low, high); break;
        case 2: selectionSort(low, high); break;
        case 3: mergeSort(low, high); break;
        case 4: heapSort(low, high); break;
        default: quickSort(low, high); break;
    }
}// 在此统一接口下,具体算法不同实现,见后续
```

起泡排序(Bubble Sort, 见 P6)：每一趟扫描交换，都记录是否存在逆序元素；如果一趟扫描元素做过交换，那么表示存在逆序对。

```
template <typename T>
void Vector<T>::bubbleSort(Rank low, Rank high){
    while (!bubble(low, hi--)); // 逐趟扫描交换,直至全部有序
}

template <typename T>
void Vector<T>::swap(T &a, T &b){
    T tmp = a;
    a = b;
    b = tmp;
}

template <typename T>
bool Vector<T>::bubble(Rank low, Rank high){ // 一趟扫描交换
    bool sorted = true; //有序标志
```

```
    for (Rank i = low + 1; i < high; i++){
        if (_elem[i - 1] > _elem[i]){ // 存在逆序
            sorted = false;
            swap(_elem[i - 1], _elem[i]);
        }
    }
    return sorted;
}
```

如果向量中，前缀无序(如 r 个元素)，后缀有序，时间复杂度为$O(nr)$。
当$r = \sqrt{n}$时，时间复杂度为$O(n^{1.5})$
如果能记录下最后一个逆序对的位置 last，下次扫描范围设为[low, high)，那么时间复杂度变为$O(n)$ (设$r = \sqrt{n}$)。

改进：

```
template <typename T>
void Vector<T>::bubbleSort(Rank low, Rank high){
    while (low < (high = bubble(low, high))); // 逐趟扫描交换,直至全部有序
}

template <typename T>
Rank Vector<T>::bubble(Rank low, Rank high){ // 一趟扫描交换
    Rank last = low; //最右侧的逆序对初始化为[low-1, low]
    for (Rank i = low + 1; i < high; i++){
        if (_elem[i - 1] > _elem[i]){ // 存在逆序
            last = i; // 更新最右侧逆序对的位置
            swap(_elem[i - 1], _elem[i]);
        }
    }
    return last; // 逻辑标志 sorted 改为秩 last
}
```

**起泡排序综合评价**
上面的改进只是对于某些情况；起泡排序效率最好$O(n)$，最坏$O(n^2)$。

输入含重复元素时，算法的稳定性(stability)是更为细致的要求。
即：重复元素在输入、输出序列中的相对次序，是否保持不变
例如：输入：3, $2_a$, 1, 4, $2_b$
输出：1, $2_a$, $2_b$, 3, 4 // stable
1, $2_b$, $2_a$, 3, 4 // unstable

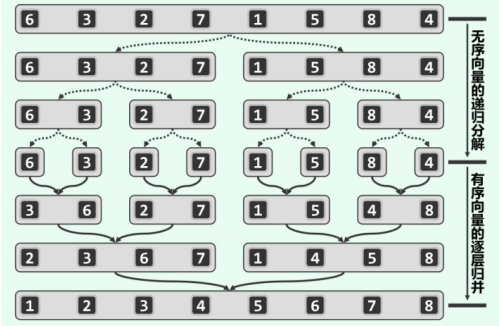
起泡排序是**稳定**的排序算法。任意元素 a 和 b 相对位置发生变化，只可能：经过与其他元素的交换，二者互相接近直至相邻；在接下来一轮扫描交换，二者因逆序而交换位置。如果二者相等(如 $2_a$ 和 $2_b$)，它们是不会交换位置的，也就是稳定的。如果将 if 判断的`>`换成`>=`，就变为不稳定的。

## 🕮 2.6 归并排序

基于比较式算法(comparison-based algorithm)简称 CBA 式算法，其最坏情况需要$\Omega(n \log n)$的时间(下界)。

归并排序原理：分治策略
1) 序列一分为二 // O(1)
2) 子序列递归排序 //2×T(n/2)
3) 合并有序子序列 //O(n)
时间复杂度：$T(n) = 2T(n/2) + O(n) = O(n \log n)$

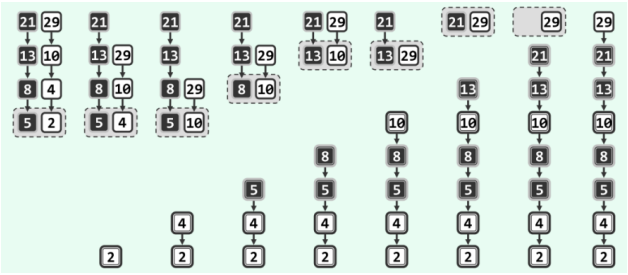
示例：




基本实现：

```
template <typename T>
void Vector<T>::sort(){ // 无参数外部排序接口,调用归并排序(为了测试)
    mergeSort(0, _size);
}

template <typename T>
void Vector<T>::mergeSort(Rank low, Rank high){ //[low, high)
    if (high - low < 2)
        return; // 单个元素自然有序
    Rank mid = (low + high) >> 1; //一分为二
    mergeSort(low, mid);
    mergeSort(mid, high);  // 对两部分归并排序
    merge(low, mid, high); // 归并
}
```

上面 3 步核心任务是归并两个子序列，也就是上面 merge()函数。
两两归并也称二路归并(2-way merge)。将两个有序序列合并为一个有序序列，即 A[low, high) = A[low ,mid) + A[mid, high)



```
template <typename T>
void Vector<T>::merge(Rank low, Rank mid, Rank high){
    T *A = _elem + low; //合并后的向量 A[0, high-low)=_elem[low, high)
    int lsize = mid - low;
    T *B = new T[lsize]; //前子向量 B[0, lsize)=_elem[low, mid)
    for (Rank i = 0; i < lsize; i++)
```

```
        B[i] = A[i]; //复制前子向量 B
    int rsize = high - mid;
    T *C = _elem + mid; // 后子向量 C[0, rsize)=_elem[mid, high)
    for (Rank i = 0, lp = 0, rp = 0; (lp < lsize) || (rp < rsize);){
        // C 已全部归并,或 C 不小于 B, B 接至 A 末尾
        if ((lp < lsize) && (rsize <= rp || (B[lp] <= C[rp])))
            A[i++] = B[lp++];
        // B 已全部归并,或 B 大于 C, C 接至 A 末尾
        if ((rp < rsize) && (lsize <= lp || (C[rp] < B[lp])))
            A[i++] = C[rp++];
    }
    delete[] B; //释放临时空间 B
}
```

上面 B 是复制出来的临时空间，C 本来就在_elem 上。当 B 提前归并完，后面 C 复制到 A 完全是多余的。

部分精简：

```
    for (Rank i = 0, lp = 0, rp = 0; lp < lsize;){
        // B 大于 C, C 接至 A 末尾; B 已全部归并直接退出循环,因为 A 末尾与 C 末尾一样
        if ((rp < rsize) && (C[rp] < B[lp]))
            A[i++] = C[rp++];
        // C 已全部归并,或 C 不小于 B, B 接至 A 末尾
        if (rsize <= rp || (B[lp] <= C[rp]))
            A[i++] = B[lp++];
    } // 交换两句次序,删除冗余逻辑
```

main()测试：

```
    int arr[] = {45, 34, 12, 78, 98, 57, 31, 65, 23};
    int length = sizeof(arr) / sizeof(int);
    Vector<int> myvector(arr, 0, length);
    myvector.show(); // [45, 34, 12, 78, 98, 57, 31, 65, 23]
    myvector.sort();
    myvector.show(); // [12, 23, 31, 34, 45, 57, 65, 78, 98]
```

merge()总体迭代只需线性时间$O(n)$。归并排序最坏时间复杂度为$O(n\log n)$。
**注意**：待归并子序列不必等长。这一算法及结论也适用于链表。

上面的虽然简洁，但是感觉不易理解。容易理解版本：

```
template <typename T>
void Vector<T>::merge(Rank low, Rank mid, Rank high){
    T *tmp = new T[high - low]; // 存放排序元素的临时数组
    Rank i = 0, lp = low, rp = mid;
    while (lp < mid && rp < high){ // 将两部分较小值放入临时数组
        if (_elem[lp] <= _elem[rp])
            tmp[i++] = _elem[lp++];
        else
            tmp[i++] = _elem[rp++];
    }
    // 一边全部归并,将另一边放入
    while (lp < mid)
        tmp[i++] = _elem[lp++];
    while (rp < high)
        tmp[i++] = _elem[rp++];
```

```
    // 临时数组元素倒回
    for (Rank j = low; j < high; j++)
        _elem[j] = tmp[j - low];
    delete[] tmp; // 释放临时空间
}
```

**20181107**

# 第 3 章　列表

## 🕮　3.1　接口与实现

### ❀　3.1.1　从静态到动态

根据是否修改数据结构，所有操作大致分为两类：
1) 静态：仅读取，数据结构的内容及组成一般不变：get()、search()...
2) 动态：需写入，数据结构局部或整体将改变：insert()、remove()...

对应地，数据元素的存储与组织方式也可分为：
1) *静态存储*：数据空间整体创建或销毁
数据元素的物理存储次序与其逻辑次序**严格一致**，可支持高效的静态操作。
如向量，元素的物理地址与其逻辑次序线性对应。静态操作如 get()操作只需$O(1)$时间；而动态操作如 insert()和 remove()就力不从心，需要$O(n)$时间。
2) *动态存储*：为各数据元素动态地分配和回收物理地址
逻辑上相邻的元素记录彼此的物理地址，在逻辑上形成一个整体；可支持高效的动态操作。

### ❀　3.1.2　从向量到列表

*列表*(list)是采用动态存储策略的典型结构，其中的元素称为*结点*(node)。各结点通过指针或引用彼此连接，在逻辑上构成一个线性序列。

相邻结点彼此互称为*前驱*(predecessor)和*后继*(successor)，前驱或后继若存在，必然唯一。没有前驱的结点称为*首结点*(first/front node)；没有后继的结点称为*末结点*(last/rear node)。

### ❀　3.1.3　从秩到位置

向量支持*循秩访问*(call-by-rank)：根据元素的秩，可在$O(1)$时间内直接确定其物理地址：V[i]的物理地址 $= V + i \times s$, s 为单个元素占用空间
比喻：某个街道 V，各住户的地址间距离均为 s，则对于门牌号为 i 的住户，地理位置$= V + i \times s$

列表同属于线性序列，也可以通过秩定位结点；但是列表的循秩访问成本过高，已不合时宜。
因此，应改用*循位置访问*(call-by-position)方式：利用结点之间的相互引用，找到特定的结点。
比喻：找到朋友 A 的亲戚 B 的同事 C 的老师 D ... 的同学 Z

### ❀　3.1.4　列表结点：ADT 接口

作为列表的基本元素，列表结点首先需要独立地封装实现。

结点的基本操作接口：

| 操作 | 功能 |
|---|---|
| prev | 前驱结点的位置 |
| next | 后继结点的位置 |
| data | 当前结点的数据对象 |
| insertPrev(e) | 插入前驱结点，存入被引用对象 e，返回新结点位置 |
| insertNext(e) | 插入后继结点，存入被引用对象 e，返回新结点位置 |

列表结点模板类：

```
#define Position(T) ListNode<T> * // 列表结点的位置
typedef int Rank;

template <typename T> //简洁起见,完全开放而不过度封装
struct ListNode{ //列表结点模板类(双向链表形式实现)
    T data;
    Position(T) prev; // 前驱
    Position(T) next; // 后继
    ListNode() {} // 针对 header 和 trailer 的构造
    ListNode(T e, Position(T) p = nullptr, Position(T) n = nullptr){ // 默认构造器
        this->data = e;
        this->prev = p;
        this->next = n;
    }
    Position(T) insertPrev(T const &e); //插入作为前驱
    Position(T) insertNext(T const &e); // 插入作为后继
};
```

### ❀ 3.1.5 列表：ADT 接口

列表模板类

```
template <typename T>
class List{
  private:
    int _size;
    Position(T) header; // 头部尾部哨兵
    Position(T) trailer;
  protected: // 内部函数
  public: // 构造、析构、只读接口、可写接口、遍历接口，与向量类似
};
```

头结点(header)、首结点(first node)、末结点(last node)、尾结点(trailer)的秩可理解为-1、0、n-1、n。header 和 trailer 作为哨兵结点(sentinel node)，对外不可见。


默认构造函数：

```
    List() { init(); } //默认构造函数
```

内部函数 init()：

```
template <typename T>
void List<T>::init(){ // 初始化，创建列表统一调用
    header = new ListNode<T>;
```

```
    trailer = new ListNode<T>;
    header->next = trailer;
    header->prev = nullptr;
    trailer->prev = header;
    trailer->next = nullptr;
    _size = 0;
}
```

## 🕮 3.2 无序列表

### ❀ 3.2.1 秩到位置

通过重载下标操作符，循秩访问元素

```
template <typename T> // 重载[]通过秩直接访问列表结点,效率低,慎用!
T &List<T>::operator[](Rank r) const{ //assert: 0 <= r < _size
    Position(T) p = first();
    while (0 < r--) // 从首结点出发第 r 个结点
        p = p->next;
    return p->data;
}
```

平均、最坏时间复杂度O($n$)

### ❀ 3.2.2 查找

在结点 p(可能是 trailer)的 n 个前驱中，找到等于 e 的最后者。

```
template <typename T> // 0 <= n <= rank(p) < _size
Position(T) List<T>::find(T const &e, int n, Position(T) p) const{ //顺序查找 O(n)
    while (0 < n--) //从右向左逐个将 p 的前驱与 e 比较
        if (e == (p = p->prev)->data)
            return p; //直至命中或范围越界
    return nullptr;      //越出左边界,查找失败
}
```

对于整个列表查找元素 e，是上面的一个特例：

```
Position(T) find(T const &e) const { return find(e, _size, trailer); }
```

当然还可以重载 find(e, p, n)函数，表示在结点 p 的 n 个后继中查找 e。

### ❀ 3.2.3 插入

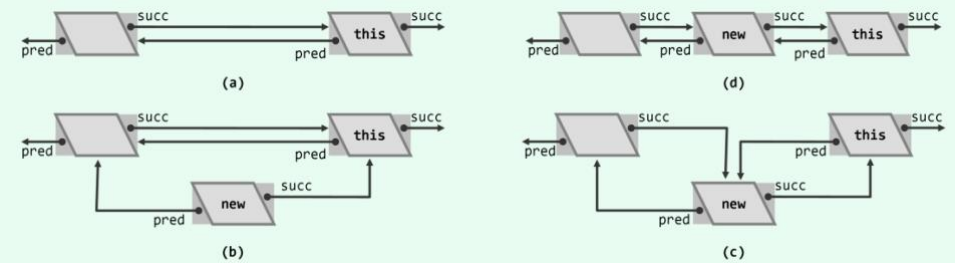
#### ① 前插入

```
template <typename T>
Position(T) List<T>::insertBefore(Position(T) p, T const &e){
    _size++;
    return p->insertPrev(e); //e 作为 p 的前驱插入
}
```

ListNode 实现的插入算法：

```
template <typename T> //前插入算法(后插入算法完全对称)
Position(T) ListNode<T>::insertPrev(T const &e){ //O(1)
    Position(T) x = new ListNode(e, this->prev, this); //创建新结点
    this->prev->next = x;
    this->prev = x; // 建立链接
    return x;       // 返回新结点位置
}
```

插入的示意图：




② 基于复制的构造

```
template <typename T>
void List<T>::copyNodes(Position(T) p, int n){ //O(n)
    init(); //初始化,创建头尾哨兵
    while (n--){
        append(p->data); // 作为末结点插入
        p = p->next;
    }
}
```

其中尾插法 append()相当于在 trailer 前面插入一个结点：

```
    Position(T) append(T const &e) { return insertBefore(trailer, e); }
```

对应构造函数：

```
    List(List<T> const &L){ //整体复制列表 L
        copyNodes(L.first(), L.size());
    }
    List(List<T> const &L, Rank r, int n){ //复制列表 L 中自第 r 项起的 n 项
        copyNodes(L[r], n);
    }
    List(Position(T) p, int n){ //复制列表中自位置 p 起的 n 项
        copyNodes(p, n);
    }
```

③ 后插入

```
template <typename T> // 后插入算法
Position(T) ListNode<T>::insertNext(T const &e){
    Position(T) x = new ListNode(e, this, this->next); //创建新结点
    this->next->prev = x;
    this->next = x;
    return x;
}

template <typename T>
Position(T) List<T>::insertAfter(Position(T) p, T const &e){
    _size++;
    return p->insertNext(e); //e 作为 p 的后继插入
}
```

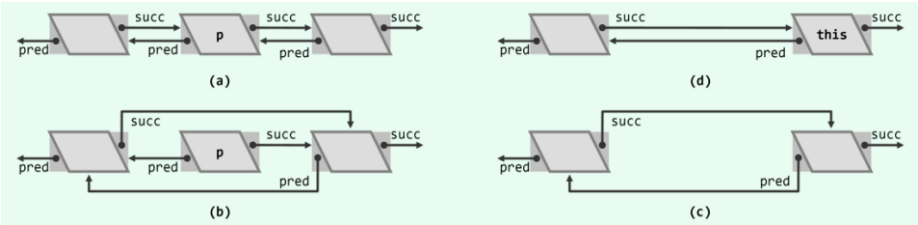
对应头插法 prepend()：

```
    Position(T) prepend(T const &e) { return insertAfter(header, e); }
```

### 3.2.4 删除

```
template <typename T>
T List<T>::remove(Position(T) p){ //O(1)
    T e = p->data; //备份删除结点的数据
    p->prev->next = p->next;
    p->next->prev = p->prev;
    delete p; // 释放空间
    _size--; // 规模减少
    return e;
}
```

示意图：




删除首结点 pop()：

```
    T pop() { return remove(first()); } //删除首结点
```

### 3.2.5 析构

```
    ~List(){//析构函数
        clear();       // 清空列表
        delete header; // 释放头尾哨兵
        delete trailer;
    };
```

内部 clear()实现：

```
template <typename T>
int List<T>::clear(){//清空列表,O(n)
    int oldSize = _size;
    while (0 < _size)
        pop();//反复删除首结点
    return oldSize;
}
```

### 3.2.6 唯一化

```
template <typename T>
int List<T>::deduplicate(){
    if (_size < 2) //一个结点自然不重复
        return 0;
    int oldSize = _size; //记录原规模
    Rank r = 1;
    for (Position(T) p = first()->next; p != trailer; p = p->next){
        Position(T) q = find(p->data, r, p); //在 p 的 r 个真前驱寻找相同者
        q ? remove(q) : r++;
    }//循环过程中,p 的所有前驱互不相同
    return oldSize - _size;
}//复杂度 O(n^2)
```

### 3.2.7 测试

打印列表：

```
template <typename T>
void List<T>::show() const{
    if (empty()){
        cout << "[]" << endl;
        return;
    }
    cout << "[";
    for (Position(T) p = first();; p = p->next){
        if (p->next == trailer){
            cout << p->data << "]" << endl;
            return;
        }
        cout << p->data << ", ";
    }
}
```

main()函数：

```
    List<int> ll;
    int arr[] = {3, 4, 5, 1, 2, 4, 5, 6, 1};
    int length = sizeof(arr) / sizeof(int);
    for (int i = 0; i < length; i++){
        if (i % 3 == 1)
            ll.append(arr[i]); // 测试尾插和 insertBefore()
        else
            ll.prepend(arr[i]); //测试头插和 insertAfter()
    }
    ll.show(); // [1, 5, 4, 1, 5, 3, 4, 2, 6]
    cout << "size: " << ll.size() << endl; // size: 9
    ll.deduplicate(); //测试去重
    ll.show(); // [1, 5, 3, 4, 2, 6]
    cout << "size: " << ll.size() << endl; // size: 6
    ll.insertAfter(ll.find(3), 23); // 测试后插和查找
    ll.show(); // [1, 5, 3, 23, 4, 2, 6]
    cout << "first:" << ll.pop() << endl; // first:1; 测试删除
    ll.show(); // [5, 3, 23, 4, 2, 6]
    ll.clear(); // 测试清空列表
    ll.show(); // []
    cout << "size: " << ll.size() << endl; // size: 0
```

## 3.3 有序列表

和有序向量一样，有序列表能使列表的一些操作变得更加高效。

### 3.3.1 唯一化

```
template <typename T>
int List<T>::uniquify(){ //有序列表删除重复元素,只需遍历列表一次,O(n)
    if (_size < 2) //一个结点自然不重复
        return 0;
    int oldSize = _size;
    Position(T) p = first();
    for (Position(T) q = p->next; q != trailer; q = p->next)
    { //p 为各区段起点,q 为 p 的后继
        if (p->data != q->data)
            p = q; //若不等,转向下一个区段
        else
```

```
            remove(q); //若相同,删除后者
    }
    return oldSize - _size;
}
```

### 3.3.2 查找

```
template <typename T> // 在有序列表结点 p 的 n 个真前驱中找到不大于 e 的最后者
Position(T) List<T>::search(T const &e, int n, Position(T) p) const
{ //对于 p 的最近 n 个前驱,从右向左逐个比较
    while (0 <= n--){
        if (((p = p->prev)->data) <= e)
            break;
    }
    return p; //直至命中,数值越界或范围越界,返回查找终止的位置
}
```

与无序列表一样，有序列表查找，最好$O(1)$；最坏、平均$O(n)$。
究其原因，列表是**动态存储策略**，结点的物理地址与逻辑次序毫无关系，无法像有序向量那样使用减而治之策略。

## 3.4 选择排序

选择排序(selection sort)将序列分为无序前缀和有序后缀，前缀每个元素不大于后缀最小元素。每次只需从前缀选出最大者，作为最小元素移至后缀最前，即可使有序部分不断扩大。
当然也可以分为有序前缀(小)和无序后缀(大)。

**实例：**

| 迭代次数 | 前缀无序子序列 | 后缀有序子序列 |
|---|---|---|
| 0 | 5 2 7 4 6 3 1 | ^ |
| 1 | 5 2 4 6 3 1 | 7 |
| 2 | 5 2 4 3 1 | 6 7 |
| 3 | 2 4 3 1 | 5 6 7 |
| 4 | 2 3 1 | 4 5 6 7 |
| 5 | 2 1 | 3 4 5 6 7 |
| 6 | 1 | 2 3 4 5 6 7 |
| 7 | ^ | 1 2 3 4 5 6 7 |

**实现：**

```
template <typename T> //起始于 p 的连续 n 个结点做选择排序
void List<T>::selectionSort(Position(T) p, int n){
    Position(T) head = p->prev; // 待排序区间(head,tail)
    Position(T) tail = p;
    for (int i = 0; i < n; i++) // 寻找到正确 tail 位置,可能是尾哨兵
        tail = tail->next;
    while (1 < n){
        Position(T) pmax = selectMax(head->next, n); // 找到最大者,待实现
        insertBefore(tail, remove(pmax)); //将其删除并移至有序区间最前
        tail = tail->prev; // 有序区间范围扩大
        n--; // 待排区间减小
    }
}
```

上面使用了 remove()和 insertBefore()，都用到了动态空间分配 new 和 delete，虽然可认为是常数复杂度，但实际时间消耗，大致是基本操作的 100 倍。
当数据较小时，可以考虑直接交换两个结点的 data。

selectMax()实现：

```
template <typename T> // 从起始于 p 的 n 个结点查找最大者
Position(T) List<T>::selectMax(Position(T) p, int n){
    Position(T) pmax = p;
    for (Position(T) cur = p->next; 1 < n; n--, cur = cur->next){
        if (!lt(cur->data, pmax->data)) //当前元素>=pmax 则更新位置
        // if (cur->data>= pmax->data) // 有的类型不能直接比较,自己写个呗
            pmax = cur; //多个相同元素选取靠后的移动,为了稳定性
    }
    return pmax;
}
```

**性能：**
共迭代 n 次，第 k 次迭代中：
selectMax()为$\Theta(n-k)$
remove()和 insertBefore()为$O(1)$
总体复杂度为$\Theta(n^2)$(最好和最坏都是)。

虽然如此，选择排序元素的移动远远少于起泡排序 //实际更为费时
$\Theta(n^2)$主要来源于元素比较操作 //成本相对更低

向量的选择排序：

```
template <typename T>
void Vector<T>::selectionSort(Rank low, Rank high){
    for (Rank i = low; i < high - 1; i++){
        Rank min_index = i; //记录最小元素的秩
        for (Rank j = i + 1; j < high; j++){
            if (_elem[j] < _elem[min_index])
                min_index = j;
        }
        swap(_elem[i], _elem[min_index]); //将最小元素交换到有序区间最后
    }
}
```

**20181109**

## 🕮 3.5 插入排序

插入排序(insertion sort)，思路为：始终将整个序列看作两部分：
有序的前缀 L[0, r)，无序的后缀 L[r, n)；
通过迭代，反复地将后缀无序首元素插入到前缀适合位置。

初始时：r = 0，空列表肯定有序
迭代：e = L[r]，在有序前缀中确定合适的位置插入 e，得到有序的 L[0, r]
插入排序的**不变性**：
随着 r 的递增，前缀 L[0, r)始终有序，直到 r = n，L 整体有序。

实例：

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

实现：

```
template <typename T> //起始于 p 的连续 n 个结点做插入排序
void List<T>::insertionSort(Position(T) p, int n){ //valid(p) && rank(p)+n<=size
    for (int r = 0; r < n; r++){
        // 从 p 的 r 个前驱寻找合适位置(不大于 p->data 的最后位置),后插入,r 即有序前缀长度
        insertAfter(search(p->data, r, p), p->data);
        p = p->next; //转向下一个结点,无序的首结点
        remove(p->prev); // 插入是复制元素再插入,故前一个结点已无用
    } // n 次迭代,每次 O(r+1)
} // O(1)辅助空间,属于就地算法
```

**性能**

最好：完全有序，每次迭代只需 1 次比较，0 次交换，累计$O(n)$时间。
最坏：完全逆序，第$k$次迭代需要$O(k)$次比较，1 次交换，累计$O(n^2)$时间。

**平均性能**

假定：各元素取值遵守均匀、独立分布；L[r]插入完成时，有序前缀 L[0, r]中哪个元素是此前的 L[r]？实际 $r+1$ 个元素均有可能，概率都是 $1/(r+1)$
因此插入 L[r]花费时间的数学期望是：

$$\frac{0+1+2+\cdots+r}{r+1}+1=\frac{r}{2}+1$$

于是，总体时间的数学期望(平均复杂度)是：$O(n^2)$

逆序对(inversion)
前面某个元素比后面某个元素大，则它们构成一个逆序对。
$n$个元素，任何两个元素都可能构成逆序对，$C_n^2=O(n^2)$。
如：4, 3, 2, 1 中有(4, 3), (4, 2), (3, 2), (4, 1), (3, 1), (2, 1)共 6 个逆序对。

假设每个逆序对都记到后一个元素的账上。对于某个结点 p，其逆序对数记为$i(p)$
整个序列所有逆序对数为$\sum_p i(p)=I$。

对于插入排序，如将某个结点 p 插入到前面有序前缀适当位置，那么该位置后面和 p 之间的元素和 p 都构成逆序对，并都记在 p 的账上，也就是 p 需要比较$i(p)$

次才能找到适当位置。所以，整个插入排序比较次数就是逆序对总数 I。
而元素插入总共 n 次，即插入排序需要时间$O(I+n)$

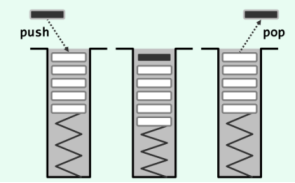
当完全有序时，逆序对总数$I=0$，插入排序需要时间$O(n)$
当完全逆序时，逆序对总数$I=C_n^2=O(n^2)$，插入排序需要时间$O(n^2)$

因为输入不同，插入排序算法效率相差很大，所以插入排序也是输入敏感(input-sensitive)的算法。

# 第 4 章　栈与队列

## 🕮　4.1　栈接口与实现

栈(stack)是存放数据对象的一种特殊容器，其中元素按照线性的逻辑次序排列。栈中可操作的一段称为栈顶(stack top)，另一无法直接操作的盲端称为栈底(stack bottom)。栈的特点：后进先出(last-in-first-out, LIFO)。




栈 ADT 操作接口

| | |
|---|---|
| size() | 栈的规模 |
| empty() | 判断栈是否为空 |
| push(e) | 将 e 插入栈顶 |
| pop() | 删除并返回栈顶元素 |
| top() | 获取栈顶元素 |

栈可以视为序列的特例，可以基于向量和列表实现栈结构。向量末端插入和删除操作耗时$O(1)$，而首端耗时$O(n)$，故基于向量实现的栈的栈顶应该设为向量的末端。而列表则相反，栈顶应该设为首结点。

// C++或 Java 可以使 Stack 继承 Vector 或 List，然而感觉 C++太坑了...改用其他语言了

```
class Stack():  # 栈
    def __init__(self):
        self.__elem = []
        self.__size = 0  # 该属性可有可无，可由 len(self.__elem)代替

    def empty(self):  # 判空
        return self.__size <= 0

    def size(self):  # 获取元素个数
        return self.__size

    def push(self, e):  # 压栈
        self.__elem.append(e)
        self.__size += 1

    def pop(self):  # 出栈
        if self.empty():
```

```
            return
        self.__size -= 1
        return self.__elem.pop()

    def top(self):  # 获取栈顶元素
        if self.empty():
            return
        return self.__elem[-1]

    def __len__(self):  # len()
        return len(self.__elem)
```

## 🕮 4.2 栈的应用

### ❁ 4.2.1 逆序输出

**进制转换：**

```
def convert(n, base):  # 十进制 n 转换到 base 进制
    digit = '0123456789ABCDEF'
    assert 1 < base <= len(digit)
    s = Stack()
    while n > 0:
        s.push(digit[n % base])  # 将余数(对应位数)入栈
        n //= base  # n 变为 n 除以 base 的商
    ret = ''
    while not s.empty():
        ret += s.pop()
    return ret

if __name__ == "__main__":
    print(f'123 的 16 进制: {convert(123, 16)}')  # 123 的 16 进制: 7B
    print(f'123 的 5 进制: {convert(123, 5)}')  # 123 的 5 进制: 443
```

### ❁ 4.2.2 递归嵌套

具有自相似性的问题可递归描述，但分支位置和嵌套深度不固定。

**括号匹配**

顺序扫描表达式，用栈记录已扫描部分

反复迭代：凡是遇到(，则将(入栈；遇到)，则将(弹出

若扫描结束，栈为空，则括号匹配；若栈提前变空或非空，则括号不匹配。

```
def paren(expression):  # 括号匹配
    s = Stack()
    for i in expression:  # 逐一检查字符
        if i == '(':  # 遇到左括号入栈
            s.push(i)
        elif i == ')':  # 遇到右括号,则弹出左括号
            if not s.empty():
                s.pop()
            else:  # 遇到右括号,且栈为空,不匹配
                return False
    return s.empty()  # 最终,栈为空则匹配

if __name__ == "__main__":
    expr = '((1+2)*3+4/(5-6))*7-(8-9)/10'
    print('匹配' if paren(expr) else '不匹配')
```

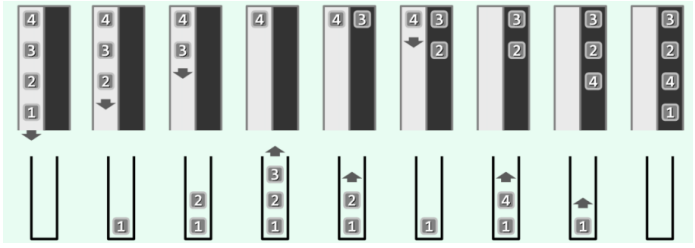
如果只是一种括号，可以简单使用一个计数器就能实现匹配功能。
但是以上思路和算法，可推广到多种括号的情况。此时就不能使用多个计数器，如"[ ( ] )"。甚至，只要约定括号通用格式，不必固定括号的类型和数目。
如 HTML 标签： | 、<body>|</body>...

### ⊛ 4.2.3 栈混洗

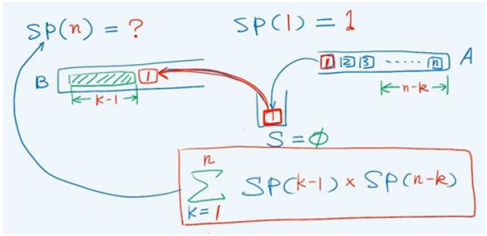
考查三个栈 A、B 和 S。其中，B 和 S 初始为空；A 含有$n$个元素，自顶而下构成输入序列：A =< $a_1, a_2, \cdots, a_n$]。此处约定尖括号为栈顶、方括号为栈底。

若只允许通过 S.push(A.pop())弹出 A 的栈顶元素并随即压入栈 S 中，或通过 B.push(S.pop())弹出 S 的栈顶元素并随即压入栈 B 中，经一系列操作后，A 中的元素都转入 B 中，B 中元素自底而上构成序列记作：B = $[a_{k1}, a_{k2}, \cdots, a_{kn}>$
则该序列称作原输入序列的一个栈混洗(stack permutation)。

如图，将<1, 2, 3, 4]变为[3, 2, 4, 1>



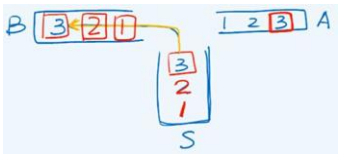
长度为$n$的输入序列，可能有多种栈混洗，总数记为SP($n$)，显然< $n!$




$$\mathrm{SP}(n) = \sum_{k=1}^{n} \mathrm{SP}(k-1) \times \mathrm{SP}(n-k) = catalan(n) = \frac{(2n)!}{(n+1)!\,n!}$$

**甄别**

输入序列<1, 2, 3, ..., n]的任意排列[$p_1, p_2, p_3, \cdots p_n$ >是否为栈混洗?
简单情况：<1, 2, 3]的栈混洗有 5 种，全排列有 6 种
其中[3, 1, 2>不是栈混洗，因为 3 是 A 中最后出栈，而最先到 B，意味着 3 在 S 弹出时 1 和 2 还在 S 中，且 1 是先出 A 栈，必在 S 的栈底，而 2 在 1 上面；此时只可能先弹出 2，再弹出 1，即结果是[3, 2, 1>，而不存在[3, 1, 2>。




对于任意三个元素，能否按某种相对次序出现在混洗中，与其他元素无关：
$\forall\ 1 \le a < b < c \le n$, [$\cdots, c, \cdots, a, \cdots, b, \cdots$ >必定不是栈混洗(出现大-小-中)，不妨称之为禁形(forbidden pattern)。

Knuth 在"the art of computer programming"中指出：
A permutation is a stack permutation iff it does NOT involve the permutation 312
是栈混洗的充要条件是：排列中不存在诸如 312 的禁形。

$[p_1, p_2, p_3, \cdots p_n >$是<1, 2, 3, ..., n]的栈混洗的充要条件：
当且仅当$i < j$，不含模式$[\cdots, j+1, \cdots, i, \cdots, j, \cdots >$
如此得到一个$O(n^2)$的甄别算法，但不令人满意。

借助栈 A、B 和 S，模拟混洗过程，只需$O(n)$的时间：
每次 S.pop()之前，检查 S 是否已空；或需弹出的元素在 S 中，却非顶元素

```
def stack_permutation(A, B):
    # A 是如<1, 2, 3, ..., n]的输入序列,B 是待甄别的序列
    assert len(A) == len(B)
    n = len(A)
    S = Stack()
    ia = 0  # 序列 A 的下标
    for i in B:
        while S.empty() or i != S.top():  # 只要 B 当前元素未出现在 S 栈顶
            if ia >= n:  # 说明 B 当前元素在栈 S 中间,弹不出来...
                return False
            S.push(A[ia])  # 反复从 A 中取出顶元素,压入 S
            ia += 1
        S.pop()  # S 栈顶出现相同元素，弹出，B 移到下一个元素
    return True  # 感觉能出循环肯定 S 为空,B 是栈混洗
```

每次栈混洗，都对应于栈 S 的$n$次 push()和$n$次 pop()操作构成的序列。
如将<1, 2, 3, 4]变为[3, 2, 4, 1>，栈 S 的操作为：
push(1), push(2), push(3), pop(3), pop(2), push(4), pop(4), pop(1)
将 push()换成(，pop()换成)，上面操作变为：((())())对应于一系列匹配的括号
$n$对括号构成的合法表达式对应于对$n$个元素进行栈混洗的一个合法过程。

**20181110**

### ⊛ 4.2.4 延迟缓冲

在一些应用问题中，输入可分解为多个单元并通过迭代依次扫描处理，但过程的各步计算往往滞后于扫描的进度，需要待到必要的信息已完整到一定程度后，才能作出判断并实施计算。此时栈结构可以扫演数据缓冲区的角色。

**中缀表达式求值**

给定语法正确的算术表达式 S，计算与之对应的数值。

如 UNIX/Linux Shell 使用 echo；DOS Shell (Windows 命令行)使用 set /a；PostScript 交互窗口使用逆波兰表达式：

```
$ echo $(( 0 + (1 + 23) / 4 * 5 * 67 - 8 + 9 ))
\> set /a (!0 ^<^< (1 - 2 + 3 * 4)) - 5 * (6 ^| 7) / (8 ^^ 9)
GS> 0 1 23 add 4 div 5 mul 67 mul add 8 sub 9 add =
```

Excel：= cos(0) + 1 - (2 - POWER((FACT(3) - 4), 5)) * 67 - 8 + 9
Word：Ctrl+F9 输入公式，右键更新域
2013 = NOT(0) + 12 + 34 * 56 + 7 + 89

calc：系统自带计算器

选取表达式中可以优先计算的运算符，计算值替换子表达式，运算符数量减一，直至整个表达式没有运算符，剩余元素就是结果，这是典型的减而治之策略。
但是对于较长的表达式，不知道要先计算那个，此时就可使用栈保存已经扫描过的子表达式，其中包含已经求解和未求解的部分。

求值算法 = 栈 + 线性扫描

### ⊛ 4.2.5 逆波兰表达式

逆波兰表达式 RPN (Reverse Polish Notation)，即后缀表达式(postfix)。
由运算符(operator)和操作数(operand)组成的表达式中，不使用括号(parenthesis-free)，即可表示带优先级的运算关系。
中缀表达式求值算法比较混乱，逻辑过于复杂，且不适于调试。

如：`0! + 123 + 4 * (5 * 6! + 7! / 8) / 9` 的 RPN 为：

```
0 ! 123 + 4 5 6 ! * 7 ! 8 / + * 9 / +
```

尽管 RPN 不易阅读，但 RPN 将运算符的优先级关系转换成了运算符在序列中出现的**次序**，即谁先出现谁就先计算。

算法：扫描 RPN 序列，遇到操作数就入栈；遇到运算符，就从栈中弹出元素，需要几个元素就弹出几个，计算结果入栈；直至栈中只有一个元素。

简单实例：

| 操作数栈 | 表达式 | 说明 |
|---|---|---|
| | 1 2 3 * + 4 - | 初始 |
| [ 1, 2, 3 > | 1 2 3 * + 4 - | 1, 2, 3 入栈 |
| [ 1, 6 > | 1 2 3 * + 4 - | 计算 2*3 = 6 入栈 |
| [ 7 > | 1 2 3 * + 4 - | 计算 1+6 = 7 入栈 |
| [ 7, 4 > | 1 2 3 * + 4 - | 4 入栈 |
| [ 3 > | 1 2 3 * + 4 - | 计算 7-4 = 3 入栈 |

RPN 算法简洁高效，将中缀表达式转为 RPN 在效率上是非常值得的。

**infix 到 postfix：**

**① 手工转换**

例如，中缀表达式为：`(0 ! + 1) ^ (2 * 3 ! + 4 - 5 )`
假设：事先未约定运算符之间的优先级关系。
1) 用括号显式地表示优先级

```
{([0 !] + 1) ^ ([(2 * [3 !]) + 4] - 5)}
```

2) 将运算符移到对应右括号后面

```
{([0  ] ! 1) + ([(2   [3] !) * 4] +  5) - } ^
```

3) 抹去所有括号

```
0   ! 1  +    2   3  !   * 4  +  5  -   ^
```

4) 稍微整理

```
0 ! 1 + 2 3 ! * 4 + 5 - ^
```

RPN 转换后，运算符的次序可能改变，但 RPN 中的操作数次序和在中缀表达式中的次序保持一致。

### ② 自动转换

运算符优先级表

```
operators = {'+': 0, '-': 1, '*': 2, '/': 3, '^': 4, '!': 5, '(': 6, ')': 7,
'\0': 8}
priority = [  # 优先级表
    # +     -     *     /     ^     !     (     )     \0  当前运算符
    ['>', '>', '<', '<', '<', '<', '<', '>', '>'],  # +  --
    ['>', '>', '<', '<', '<', '<', '<', '>', '>'],  # -  |
    ['>', '>', '>', '>', '<', '<', '<', '>', '>'],  # *  栈
    ['>', '>', '>', '>', '<', '<', '<', '>', '>'],  # /  顶
    ['>', '>', '>', '>', '>', '<', '<', '>', '>'],  # ^  运
    ['>', '>', '>', '>', '>', '>', ' ', '>', '>'],  # !  算
    ['<', '<', '<', '<', '<', '<', '<', '=', ' '],  # (  符
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],  # )  |
    ['<', '<', '<', '<', '<', '<', '<', ' ', '='],  # \0 --
]
```

求值算法：

```
def evaluate(s):  # 计算中缀表达式 s 的值
    s += '\0'  # 末尾添加'\0'尾哨兵
    nums, optr = Stack(), Stack()  # 存放操作数和运算符的两个栈
    rpn = ''  # 逆波兰表达式
    optr.push('\0')  # 尾哨兵'\0'首先入栈
    i = 0  # 表达式当前下标
    while not optr.empty():
        if is_digit(s[i]):
            i, n = read_num(s, i)  # 读入数字可能多位
            nums.push(n)  # 操作数入栈
            rpn += f'{n} '  # 并添加到 rpn 末尾
        elif s[i] == ' ':  # 忽略空格
            i += 1
        else:
            pri = order_between(optr.top(), s[i])
            if pri == '<':  # 栈顶运算符优先级低
                optr.push(s[i])  # 推迟计算,当前运算符入栈
                i += 1
            elif pri == '>':  # 栈顶运算符优先级高,计算
                op = optr.pop()
                rpn += f'{op} '  # 栈顶运算符出栈,接至 RPN 末尾
                if op == '!':  # 一元运算符
                    nums.push(fac(nums.pop()))
                else:  # 二元运算符
                    n2 = nums.pop()
                    n1 = nums.pop()
                    nums.push(cal(n1, op, n2))  # 计算结果入栈
            elif pri == '=':  # 优先级相等(当前运算符是右括号或'\0')
                optr.pop()  # 相当于删除右括号
                i += 1
    return rpn, nums.pop()  # 返回逆波兰表达式和计算值
```

其他函数：

```
def is_digit(n):  # 判断单个字符是不是整数
    return n in '0123456789'

def read_num(s, index):  # 只考虑正整数
    ret = 0
    while is_digit(s[index]):  # 设置了尾哨兵，不必判断是否出界
        ret = ret*10+int(s[index])
        index += 1
    return index, ret

def order_between(a, b):  # 根据优先级表获取两个运算符优先级
    return priority[operators[a]][operators[b]]

def fac(n):  # 阶乘
    s = 1
    while n > 0:
        s *= n
        n -= 1
    return s

def cal(n1, op, n2):  # 二元运算符
    if op == '+':
        return n1+n2
    elif op == '-':
        return n1-n2
    elif op == '*':
        return n1*n2
    elif op == '/':
        return n1/n2
    elif op == '^':
        return n1**n2
```

测试：

```
if __name__ == "__main__":
    infix = '0! + 123 + 4 * (5 * 6! + 7! / 8) / 9'
    rpn, ret = evaluate(infix)
    print(f'{rpn} = {ret}')  # 0 ! 123 + 4 5 6 ! * 7 ! 8 / + * 9 / +  = 2004.0
```

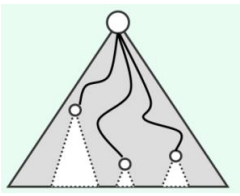
## 🕮 4.3 试探回溯法

### ❀ 4.3.1 试探与回溯

以经典的旅行商问题 (traveling salesman problem, TSP)为例，目标是计算出由给定的$n$个城市构成的一个序列，使得按此序列对这些城市的环游成本(比如机票价格)最低。尽管此类问题描述并不复杂，但由于所有元素(比如城市)的每一排列都是一个候选解，往往构成一个极大的搜索空间。

通常，其搜索空间的规模与全排列总数大体相当，即$n! = O(n^n)$。若采用蛮力策略，逐一生成可能的解并检查是否合理，效率实在太低。

根据候选解的某些局部特征，以候选解子集为单位批量排除。如图所示，通常搜索空间呈树状结构，而被排除的候选解往往隶属于同一分支，故这一技巧也被形象地称为剪枝(pruning)。



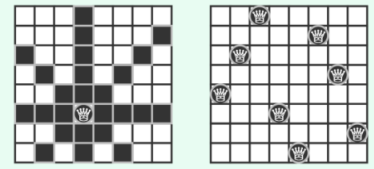
对应算法：
从零开始，尝试逐步增加候选解的长度。更准确地，这一过程是在成批地考查具有特定前缀的所有候选解。这种从长度上逐步向目标解靠近的尝试，称作试探(probing)。作为解的局部特征，特征前缀在试探的过程中一旦被发现与目标解不合，则收缩到此前一步的长度，然后继续试探下一可能的组合。
特征前缀长度缩减的操作，称作回溯(backtracking)，其效果等同于剪枝。
只要目标解的确存在，就迟早会被发现，而且只要剪枝所依据的特征设计得当，计算的效率就会大大提高。

**20181111**

### ⊛ 4.3.2 八皇后

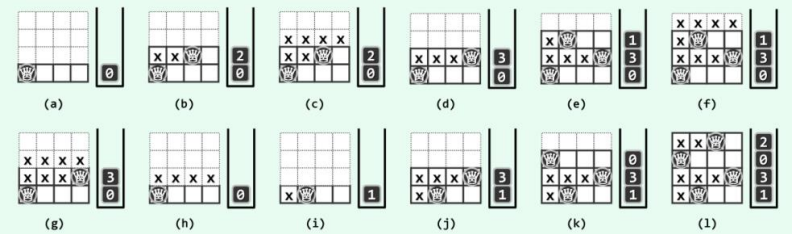
八皇后问题是一个古老而著名的问题，是回溯算法的典型案例。该问题是国际西洋棋棋手马克斯·贝瑟尔于 1848 年提出：在 8×8 格的国际象棋上摆放八个皇后，使其不能互相攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上，问有多少种摆法。
强如高斯，没学过编程，手动计算也只得出 76 种方案；然而实际答案为 92 种。

如图左边是皇后的攻击范围，右边是八皇后问题的某个解。



该问题也可以进一步推广到$n(n \geq 4)$皇后问题。

如图是求四皇后问题第一个解的过程：




**思路**：基于试探回溯策略，实现通用的$n$皇后算法。
不妨首先将各皇后分配至每一行。然后从空棋盘开始，逐个尝试着将她们放置到无冲突的某列。每放置好一个皇后，才继续试探下一个。若当前皇后在任何列都会造成冲突，则后续的试探都是没用的，此时应该回溯到上一皇后。

版本 1：非递归

```python
def conflict(lst, cur_row, cur_col):
    # cur_row 为当前行数,也是前面的总行数
    for row in range(cur_row):
        # 列数相等,或列数之差对于行数之差,即对角线,为冲突
        if abs(cur_col-lst[row]) in [0, cur_row-row]:
            return True
    return False
```

```python
def queen(n):  # n 皇后问题,利用栈,非递归
    solution = []  # list 模拟栈结构
    row, col = 0, 0
    while row > 0 or col < n:  # 这个条件怎么确定?
        if row >= n or col >= n:  # 如果出界,回溯到上一行,继续试探下一列
            row, col = row-1, solution.pop()+1
        else:  # 试探下一行,一直冲突就不断下一列,寻找可以摆放的列
            while col < n and conflict(solution, row, col):
                col += 1
            if col < n:  # 出了循环,col 可能还会越界
                solution.append(col)  # 存在就放入
                row, col = row+1, 0  # 转入下一行第一列开始试探
                if row == n:  # 如果摆放到最后一行, 将结果 yield
                    yield [x for x in solution]  # list 是可变对象,需要深拷贝
```

测试：

```python
def pretty_print(A):
    # 打印棋盘
    print(A)
    for i in A:
        lst = ['■'] * len(A)
        lst[i] = '□'
        print(''.join(lst))
    print('-'*20)

if __name__ == "__main__":
    n = 8
    ret = list(queen(n))
    print(f'{n}皇后问题总共{len(ret)}种方法\n 随机挑选一种方法:')
    pretty_print(random.choice(ret))
```

结果：

```
8 皇后问题总共 92 种方法
随机挑选一种方法:
[2, 5, 7, 0, 4, 6, 1, 3]
■■□■■■■■
■■■■■□■■
■■■■■■■□
□■■■■■■■
■■■■□■■■
■■■■■■□■
■□■■■■■■
■■■□■■■■
```

版本 2：递归+生成器

```python
def queen2(n, pre=()):  # 生成器+递归
    for col in range(n):
        if not conflict(pre, len(pre), col):
            if len(pre) == n-1:
                yield (col,)
            else:  # 后面的 tuple 层层向上传递, 组成新的 tuple, 直到最开始调用
                for ret in queen2(n, pre+(col,)):
                    yield (col,)+ret
```

版本 3：递归

```python
queen3_lst = []  # 全局 list

def queen3(A, row=0):  # 从第 0 行开始
    if row == len(A):
        queen3_lst.append([x for x in A])
```

```
        return
    for col in range(len(A)):
        if not conflict(A, row, col):  # 每列逐一尝试,如果不冲突则
            A[row] = col  # 放入并
            queen3(A, row+1)  # 递归下一行
```

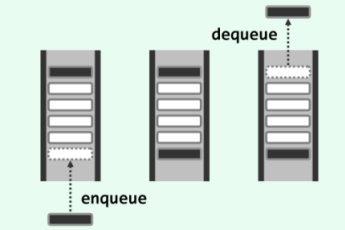
### 4.3.3 迷宫寻径

路径规划是人工智能的基本问题之一，要求依照约定的行进规则，在具有特定几何结构的空间区域内，找到从起点到终点的一条通路。
该问题的一个简化版本：空间区域限定为由 n×n 个方格组成的迷宫，除了四周的围墙，还有分布其中的若干障碍物；只能水平或垂直移动。
目标：在任意指定的起点与终点之间，找出一条通路(如果的确存在)。

## 4.4 队列接口与实现

队列(queue)也是存放数据对象的一种容器，其中的数据对象也按线性的逻辑次序排列。队列也是受限的序列：只有一端能进，而另一端只能出。

允许取出元素的一端称为队头(front)，允许插入元素的另一端称为队尾(rear)。
元素的插入和删除分别称为入队(enqueue)和出队(dequeue)。




不难看出，队列的特点是：先进先出(first-in-first-out, FIFO)
队列 ADT 操作接口

| 操作 | 功能 |
|---|---|
| size() | 获取队列规模 |
| empty() | 判断队列是否为空 |
| enqueue(e) | 将 e 插入队尾 |
| dequeue() | 删除并返回队首元素 |
| front() | 获取队首元素 |
| rear() | 获取队尾元素 |

**模板类**
队列属于序列的特例，也可以直接基于向量或列表派生。
如果基于单链表，队首应位于其首结点。

# 第 5 章 二叉树

## 5.1 树

此前的 Vector 和 List 都是线性结构(linear structure)，但它们无法同时兼顾静态操作和动态操作。树可以理解为 List 的 List，或二维 List，因此树称为半线性结构(semi-linear structure)。树常用于表示层次关系，如：文件系统、HTML DOM...

从图论角度看，树是特殊的图(连通无环图)，也是由一组顶点(vertex)和连接其间的若干边(edge)组成。$T = (V, E)$，结点数$|V| = n$，边数$|E| = e$
但计算机中的树，还要指定某一特定结点$r \in V$作为根(root)，T 则称为有根树(rooted tree)。从程序实现的角度，更多地称为顶点为结点(node)。

子树(subtree)
孩子(child)
父亲(parent)
兄弟(sibling)
(出)度(degree)：有几个孩子

可以证明树的边总数：$e = \sum_{r \in V} degree(r) = n - 1 = \Theta(n)$
//每个非根结点对应一条向上的边，而根结点没有，即边数e =顶点数$n - 1$
在衡量相关复杂度时，边数 e 可以用$n$作为参照。

若指定$T_i$为T的第$i$棵子树，$r_i$是$r$的第$i$个孩子，则称T为有序树(ordered tree)

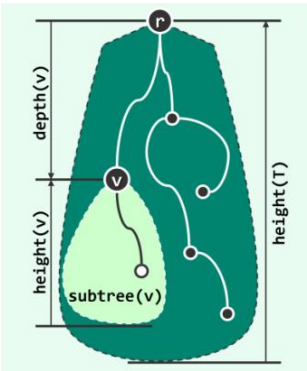
若 V 中 k+1 个结点，通过 E 中的 k 条边依次相连，则构成一条路径(path)。
$$\pi = \{(v_0, v_1), (v_1, v_2), \cdots, (v_{k-1}, v_k)\}$$
路径长度：$|\pi| = k =$ 边数
若$v_k = v_0$，则形成环路(cycle/loop)。

图中任何结点之间都有路径，则称为连通图(connected)。
不含环路，称为无环图(acyclic)。
如果边太少，则无法连通；若边太多，则容易成环。

树是在**无环**与**连通**之间达到平衡的一种图。无环表示边不能太多；连通表示边不能太少。因此树也叫：无环连通图、极小连通图、极大无环图。
所以，树中任何一结点与根之间存在唯一通路：path(v, r) = path(v)
该路径长度可以作为结点 v 的一个重要指标。

不存在歧义时，路径、结点、子树可以相互指代：
$$path(v) \sim v \sim subtree(v)$$

v 的深度(depth)：$depth(v) = |path(v)|$
path(v)上的结点，都是 v 的祖先(ancestor)，v 是它们的后代(descendent)。
如果除去自身，称为真(proper)祖先/后代。
v 的祖先若存在，一定唯一；v 的后代若存在，则未必。




根结点是所有结点的公共祖先，深度为 0。
叶子(leaf)：没有后代的结点。
所有叶子结点深度最大者，称为(子)树的高度(height)。
$$height(v) = height(subtree(v))$$

特别地，单个结点的树高度为 0；空树高度取作-1。
对于任何结点 v，depth(v) + height(v) ≤ height(T)

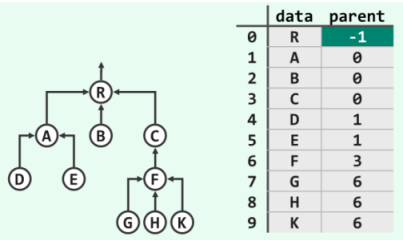
## 🕮 5.2 树的表示

### ❁ 5.2.1 接口

| 操作 | 功能 |
|---|---|
| root() | 根结点 |
| parent() | 父结点 |
| firstchild() | 长子 |
| nextsibling | 兄弟 |
| insert(i, e) | 将 e 作为第 i 个孩子插入 |
| remove(i) | 删除第 i 个孩子(及其后代) |
| traverse() | 遍历 |

### ❁ 5.2.2 父结点

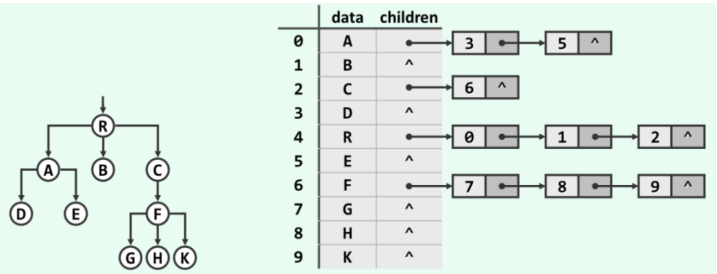
除根结点为，任何结点有且仅有一个父结点。
构思：将结点组织为序列(Vector 或 List)，各结点分别记录：




data：本身信息；
parent：父结点的秩或位置
根结点指定一个假想的父结点-1 或 NULL。
空间性能：$O(n)$
时间性能：
☺ parent()：$O(1)$
☹ root()：$O(n)$或$O(1)$(将根固定位置)
☹ firstChild()和 nextSibling()：$O(n)$
所以改进焦点集中在向下寻找孩子结点上。

### ❁ 5.2.3 孩子结点

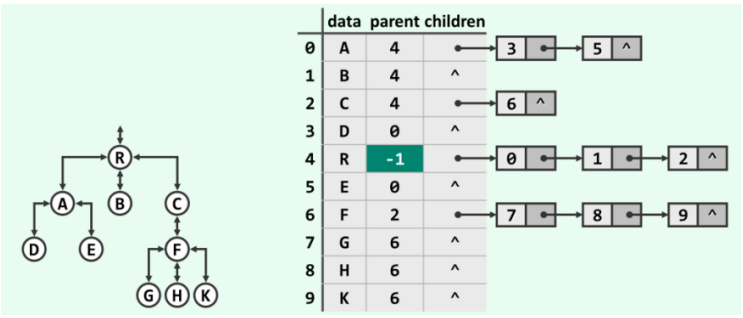
将每个结点的孩子组织成一个 Vector 或 List：




对于有 r 个孩子的结点，查找孩子所需时间为$O(r)$；但是对于查找父结点，效率又变得很低。

### 5.2.4 父结点+孩子结点

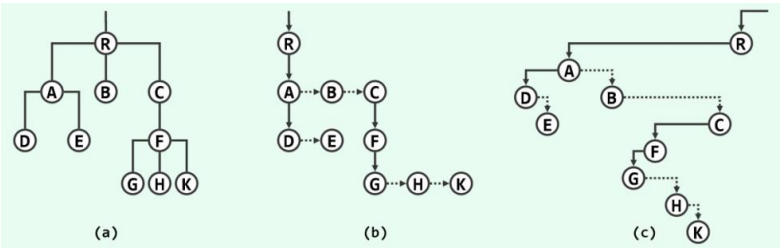
结合上面两种方法的优势，既记录父结点，也维护一个孩子的序列：




然而每个结点 children 所指向的序列的长度可能相差很大。

### 5.2.5 长子+兄弟

每个结点均设有两个引用：纵：firstChild()；横：nextSibling()

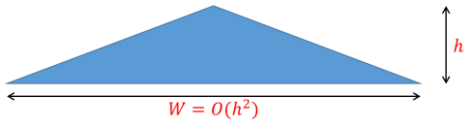



然后，一棵树就变成了二叉树...

## 5.3 二叉树概述

二叉树(binary tree)：每个结点度数都不超过 2 的树。
同一结点的孩子和子树，以左、右区分：左孩子、右孩子；左子树、右子树
其中隐含了有序性，也称有序二叉树(ordered binary tree)。

深度为 k 的结点，至多有$2^k$个。
含有 n 个结点，高度为 h 的二叉树：$h < n < 2^{h+1}$
1) $n = h + 1$：退化成一条单链(斜树)
2) $n = 2^{h+1} - 1$：满二叉树(full binary tree)



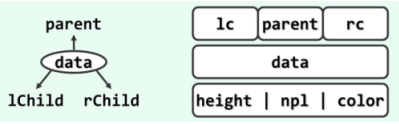

不含有度为 1 的结点的二叉树，称为真二叉树(proper binary tree)。

**描述多叉树**

二叉树的多叉树的特例，但若多叉树**有根**且**有序**时，都可以通过二叉树描述。
将多叉树的长子-兄弟表示法的结构顺时针旋转 45°，firstChild()变为 leftChild()，nextSibling()变为 rightChild()，就变成二叉树。
所以，研究树主要是研究二叉树。

## 🕮 5.4 二叉树实现

二叉树的基本组成的二叉树结点(binary tree node)，边对应于结点之间的引用。




```python
class TreeNode():  # 二叉树结点
    def __init__(self, val, parent=None, left=None, right=None):
        self.val = val
        self.parent = parent
        self.left = left
        self.right = right
        self.height = 0  # 高度

    def insert_as_left_child(self, e):  # 作为左孩子插入
        self.left = TreeNode(e, self)
        return self.left

    def insert_as_right_child(self, e):  # 作为右孩子插入
        self.right = TreeNode(e, self)
        return self.right

    def size(self):  # 后代总数,子树规模, O(n)
        s = 1  # 包括本身
        # 左右子树递归
        s += self.left.size() if self.left is not None else 0
        s += self.right.size() if self.right is not None else 0
        return s

    @staticmethod
    def stature(p):  # 获取结点 p 子树的高度
        if p is None:
            return -1
        assert isinstance(p, TreeNode)
        return p.height
```

二叉树

```python
class BinaryTree():  # 二叉树
    def __init__(self, root=None):
        self.__root = root
        self.__size = 0

    @staticmethod
    def update_height(x):  # 更新结点 x 的高度,具体规则因树不同而异
        if x is None:
            return -1
        assert isinstance(x, TreeNode)
        # x 的高度为左右子树高度较大者+1, O(1)
        x.height = 1 + max(TreeNode.stature(x.left),
                           TreeNode.stature(x.right))
        return x.height

    @staticmethod
    def update_height_above(x):  # 更新结点 x 及祖先的高度
        if x is None:
```

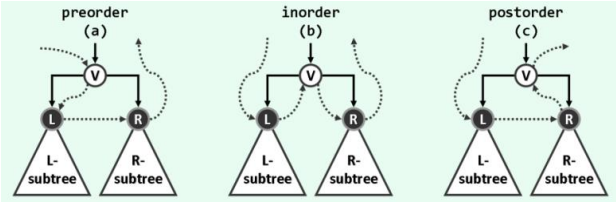
```
            return
        assert isinstance(x, TreeNode)
        while x is not None:  # 可优化,一旦高度未变,即可终止
            BinaryTree.update_height(x)
            x = x.parent  # O(depth(x))

    def size(self):  # 规模
        return self.__size

    def empty(self):  # 判空
        return self.__root is None

    def root(self):  # 树根
        return self.__root

    def insert_as_right_child(self, x, e):
        self.__size += 1  # 假设 x 存在并没有右孩子
        x.insert_as_right_child(e)
        BinaryTree.update_height_above(x)  # 祖先结点高度可以增加
        return x.right
```

**20181112**

## 🕮 5.5 二叉树的遍历

按照某种次序访问树中各结点，每个结点被访问恰好一次。
以 V 为根的子树 T，其左子树为 L，右子树为 R：$T = V \cup L \cup R$

1) 先序遍历(preorder traverse)：V | L | R
2) 中序遍历(inorder traverse)：L | V | R
3) 后序遍历(postorder traverse)：L | R | V
4) 层次(广度)优先遍历(level-order/breadth first traverse)：自上而下，先左后右




### ❁ 5.5.1 先序遍历

递归版本较为简洁：

```
    def preorder_traverse(self, visit):  # 先序遍历
        assert callable(visit)
        self.__preorder_traverse(self.__root, visit)

    def __preorder_traverse(self, x, visit):
        if x is None:
            return
        visit(x.val)  # 访问根结点元素
        self.__preorder_traverse(x.left, visit)  # 访问左子树
        self.__preorder_traverse(x.right, visit)  # 访问右子树
```

复杂度：$T(n) = O(1) + T(a) + T(n - a - 1) = O(n)$
改变 visit()函数的位置，就能实现中序遍历和后序遍历的递归版本。
递归版本也可改写为迭代版本，且都具有渐进的线性复杂度。但递归版本遍历算

法时间、空间复杂度的常系数比迭代版本更大。

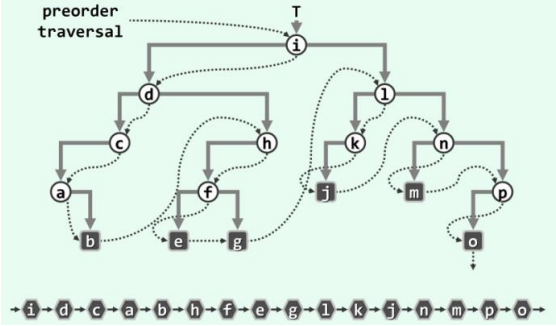
版本 1：

```
    def __preorder_i1(self, x, visit): # 没必要定义为实例方法啊...
        s = Stack()  # 辅助栈
        if x is not None:
            s.push(x)  # 根结点入栈
        while not s.empty():
            p = s.pop()  # 弹出并访问当前结点
            visit(p.val)
            if p.right is not None:  # 右孩子先入后出
                s.push(p.right)
            if p.left is not None:  # 左孩子后入先出
                s.push(p.left)
```

但很遗憾，该策略不容易推广到中序遍历和后序遍历。

版本 2：

思路：




先访问根结点，再访问其左孩子，左孩子的左孩子...直到 NULL；转移到上一个结点的右孩子...

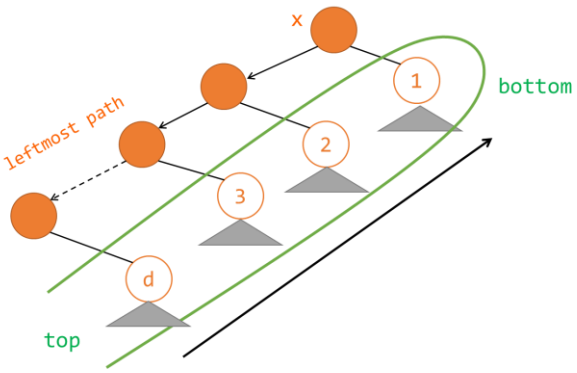
不妨将根结点沿着左分支一直下行的路线称为左侧通路(leftmost path)。

先序遍历分为：1) 自顶而下的访问左侧通路上的结点；2) 自底而上一次遍历每个右子树。

```
    def preorder_traverse(self, visit):  # 先序遍历
        assert callable(visit)
        BinaryTree.__preorder_i2(self.__root, visit)

    @staticmethod
    def __visit_along_left(x, vist, stack):
        while x is not None:
            vist(x.val)  # 访问左侧通路各个结点
            stack.push(x.right)  # 右孩子入栈
            x = x.left  # 左侧下行

    @staticmethod
    def __preorder_i2(x, visit):
        s = Stack()  # 辅助栈
        while True:
            # 访问子树 x 的左侧通路,右子树入栈缓冲
            BinaryTree.__visit_along_left(x, visit, s)
            if s.empty():
                break
```

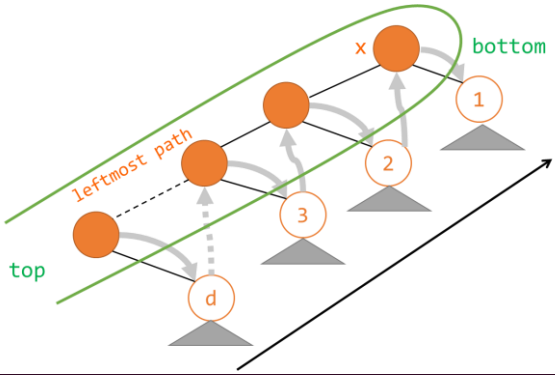
```
            x = s.pop()  # 弹出下一子树的根

if __name__ == "__main__":
    arr = [x for x in range(1, 16)]
    tree = BinaryTree()
    tree.build_tree(arr)
    lst = []
    tree.preorder_traverse(lambda x: lst.append(x))
    print(lst)  # [1, 2, 4, 8, 9, 5, 10, 11, 3, 6, 12, 13, 7, 14, 15]
```

先序遍历示意图：




### 5.5.2 中序遍历 (迭代)

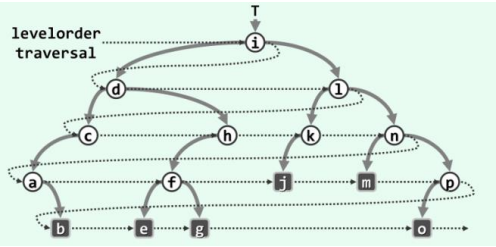
从根出发沿左侧通路下行，直到最深结点，它就是全局最先被访问的结点。




```
    @staticmethod
    def __go_along_left(x, stack):
        while x is not None:
            stack.push(x)  # 反复将左侧通路结点入栈
            x = x.left

    @staticmethod  # 为什么感觉作为 TreeNode 的实例方法比较好?
    def __inorder_i1(x, visit):  # 中序遍历迭代版本 1
        s = Stack()  # 辅助栈
        while True:
            BinaryTree.__go_along_left(x, s)
            if s.empty():
                break
            x = s.pop()  # 取出最左侧的结点
            visit(x.val)  # 访问之
            x = x.right  # 转向右子树

    def inorder_traverse(self, visit):  # 中序遍历
        assert callable(visit)
        BinaryTree.__inorder_i1(self.__root, visit)
```

```
if __name__ == "__main__":
    arr = list('idlchkna#f#j#mp#beg####o#')
    tree = BinaryTree()
    tree.build_tree(arr)
    lst = []
    tree.inorder_traverse(lambda x: lst.append(x))
    print('→'.join(lst))  # a→b→c→d→e→f→g→h→i→j→k→l→m→n→o→p
```

### 5.5.3 层次遍历




```
class TreeNode():  # 二叉树结点
    # ...为了和前面统一,应该作为 BinaryTree 类静态方法...还是算了...
    def traverse_level(self, visit):  # 以该结点为根的层次遍历
        queue = Queue()  # 辅助队列
        queue.enqueue(self)  # 首先根结点入队
        while not queue.empty():
            x = queue.dequeue()
            visit(x.val)
            if x.left is not None:  # 左孩子入队
                queue.enqueue(x.left)
            if x.right is not None:  # 右孩子入队
                queue.enqueue(x.right)

class BinaryTree():  # 二叉树
    # ...
    def level_order_traverse(self, visit):  # 层次遍历
        if self.empty():
            return
        assert callable(visit)
        self.__root.traverse_level(visit)  # 调用根结点的层次遍历

if __name__ == "__main__":
    arr = list('idlchkna#f#j#mp#beg####o#')
    tree = BinaryTree()
    tree.build_tree(arr)
    lst = []
    tree.level_order_traverse(lambda x: lst.append(x))
    print('→'.join(lst))  # i→d→l→c→h→k→n→a→f→j→m→p→b→e→g→o
```

### 5.5.4 重构

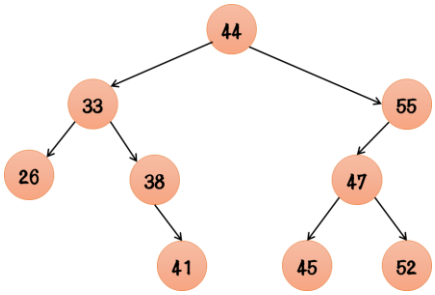
任何一棵二叉树都有前序、中序、后序遍历三个序列。
倒过来，根据先序|后序+中序就能唯一确定一棵二叉树的拓扑结构。

根据先序|后序遍历，可以确定根结点：v | LR 或 LR | v
根据根结点在中序遍历的位置，可以将序列分为 L | v | R 三部分；
根据 L 和 R 的长度，可以将先序或后序划分为 v | L | R 或 L | R | v
对于 L 和 R 可以递归使用同样的方法。

**例**：根据先序遍历和中序遍历构建二叉树

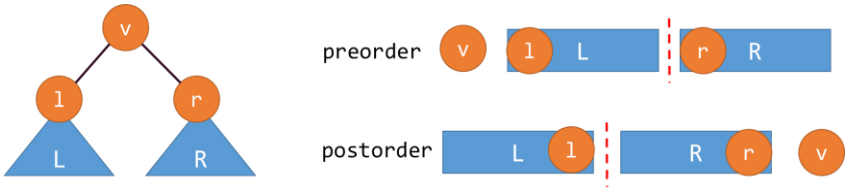
```python
from hikari import BinaryTree, TreeNode

def build_binary_tree(preoder, inorder):
    # 根据先序遍历和中序遍历构建二叉树,返回根结点
    assert set(preoder) == set(inorder)
    n = len(preoder)
    return __build(preoder, 0, n, inorder, 0, n)

def __build(preorder, plow, phigh, inorder, ilow, ihigh, parent=None):
    assert phigh-plow == ihigh-ilow
    if phigh-plow <= 0:  # 递归基, 序列长度为 0
        return
    root_val = preorder[plow]  # 先序遍历首个为根结点
    root = TreeNode(root_val, parent)
    ipos = ilow
    while ipos < ihigh:  # 寻找中序遍历根结点位置
        if inorder[ipos] == root_val:
            break
        ipos += 1
    len_left = ipos-ilow  # 左子树元素个数
    # 左子树: preorder[plow+1, plow+1+len_left) + inorder[ilow, ipos)
    root.left = __build(preorder, plow+1, plow+1+len_left,
                        inorder, ilow, ipos, root)
    # 右子树: preorder[plow+1+len_left, phigh) + inorder[ipos, ihigh)
    root.right = __build(preorder, plow+1+len_left, phigh,
                         inorder, ipos+1, ihigh, root)
    return root

if __name__ == "__main__":
    preorder_lst = [44, 33, 26, 38, 41, 55, 47, 45, 52]
    inorder_lst = [26, 33, 38, 41, 44, 45, 47, 52, 55]
    root = build_binary_tree(preorder_lst, inorder_lst)
    tree = BinaryTree(root=root)
    lst = []
    tree.postorder_traverse(lambda x: lst.append(x))
    print(lst)  # [26, 41, 38, 33, 45, 52, 47, 55, 44]
```

测试实例：




如果只知道先序和后序遍历序列，无法唯一确定一棵二叉树。因为左右子树序列 LR 不知道在哪处切分。

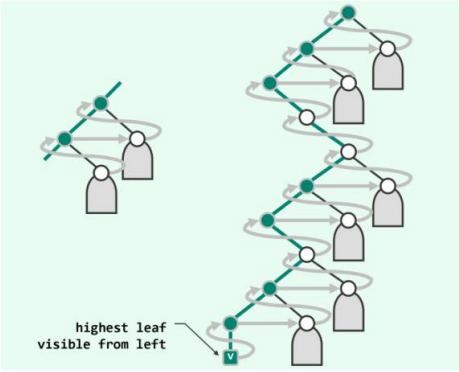
对于真二叉树，可以根据先序+后序确定唯一一棵二叉树。
如下图，对于真二叉树每个结点 v，其孩子 l、r 同时存在或不存在。
根据先序遍历的 l 找到在后序遍历中的位置，然后将后序切分；同理，也可将前序遍历切分。




**20181113**

### ❁ 5.5.5 后序遍历 (迭代)

二叉树从左侧水平向右看，未被遮挡的最高叶结点 v 是最高左侧可见叶结点(HLVFL)，即后序遍历首先访问的结点。该结点可能是左孩子或右孩子。




考查 v 与根结点唯一通路，自底而上沿着该通路：

1) 访问该结点；
2) 遍历以其右兄弟(若存在)为根的子树；
3) 向上回溯至其父结点，并转入下一片段。

```
    @staticmethod
    def __go_to_HLVFL(stack):
        # 在以栈顶结点为根的子树中，找到最高左侧可见叶结点
        x = stack.top()
        while x is not None:  # 沿途结点依次入栈(尽可能向左)
            if x.left is not None:
                if x.right is not None:  #  左右孩子入栈,如果有右孩子,先入(后出)
                    stack.push(x.right)
                stack.push(x.left)
            else:
                stack.push(x.right)  # 没有左孩子
            x = stack.top()  # 转到下一个结点
        stack.pop()  # 删除空结点

    @staticmethod
    def __postorder_iter(x, visit):  # 后序遍历迭代版本
         s = Stack()
        if x is not None:
            s.push(x)  # 根结点入栈
        while not s.empty():
            # 若栈顶不是当前结点父结点,必是其右兄弟,需要在以
            # 右兄弟为根的子树中找到 HLVFL(深入其中)
            if s.top() != x.parent:
                BinaryTree.__go_to_HLVFL(s)
            x = s.pop()
            visit(x.val)
```

测试：

```
if __name__ == "__main__":
    tree = BinaryTree()
    tree.build_tree('KiJ#h##bG#aeF##CD##')
    print('→'.join(tree.inorder_list()))  # i→b→a→h→C→e→D→G→F→K→J
    print('→'.join(tree.postorder_list()))  # a→b→C→D→e→F→G→h→i→J→K
```

但很多面试题的 TreeNode 的没有 parent 属性的，所以需要记录刚才访问的元素，并查看栈顶结点是不是其父结点，若不是，则仍需深入访问。

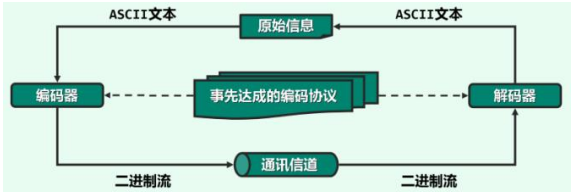
```
    @staticmethod
    def __postorder_i2(x, visit):
        if x is None:
            return
        pre = None  # 记录上一次访问结点
        s = Stack()
        s.push(x)
        while not s.empty():
            x = s.top()
            # 如果栈顶结点是叶结点或孩子结点刚被访问过，则：
            if x.is_leaf() or (pre is not None and (x.has_child_x(pre))):
                visit(x.val)  # 访问之
                pre = s.pop()  # 弹出并更新记录
            else:  # 左右孩子入栈，如果有右孩子，则先入（后出）
                if x.has_right_child():  # TreeNode 中添加的简单方法
                    s.push(x.right)
                if x.has_left_child():
                    s.push(x.left)
```

对于任何结点 x，如果存在孩子，按照右孩子、左孩子依次入栈，保证出栈时：左-右-根的顺序。当结点 x 不存在左右孩子或 x 的左右孩子都被访问过时，可以访问 x。

## 📖 5.6 编码树

### ❁ 5.6.1 二进制编码

在加载到信道上之前，信息被转换为二进制形式的过程称作*编码*(encoding)；反之，经信道抵达目标后由二进制编码恢复原始信息的过程称作*解码*(decoding)。




#### ① 生成编码表

原始信息的基本组成单位是字符，来自于某一特定的有限集合 Σ，也称为*字符集*(alphabet)。而以二进制形式承载的信息，都可表示为来自编码表$\Gamma = \{0,1\}$的某一特定二进制串。因此每一编码表都是从字符集 Σ 到编码表 Γ 的一个单射，编码是对信息文本中各字符逐个实施这一映射的过程，而解码则是逆向映射的过程。

编码表一旦制定，信息的发送方和接收方就建立了一个约定。下面是某个二进制编码表：

| 字符 | A | E | G | M | S |
|---|---|---|---|---|---|
| 编码 | 00 | 01 | 10 | 110 | 111 |

② 二进制编码

编码就是给定任意文本，通过查编码表逐一将字符翻译为二进制编码。

如文本"MESSAGE"，根据上表给出的编码方案得到二进制编码串：

"11001111111001001"

③ 二进制解码

编码器生成二进制流经信道送达后，接收方按照事先约定的编码表，依次扫描各比特位，经逐一匹配翻译出各字符，最终恢复成原文本。

| 二进制编码 | 当前匹配字符 | 解出原文 |
|---|---|---|
| 11001111111001001 | M | M |
| 01111111001001 | E | ME |
| 111111001001 | S | MES |
| 111001001 | S | MESS |
| 001001 | A | MESSA |
| 1001 | G | MESSAG |
| 01 | E | MESSAGE |

④ 解码歧义

如果编码表为：{"M": "11", "S": "111"}，对于二进制编码串"111111"，可以解码为："MMM"或"SS"。

解码过程出现歧义或错误的根本原因是编码表制定不当。因为解码算法采用：按顺序对信息比特流做子串匹配的策略。为了消除歧义，任何两个字符对应二进制编码串，相互不能是对方前缀。而此处"11"是"111"的前缀。

⑤ 前缀无歧义编码(prefix-free code)

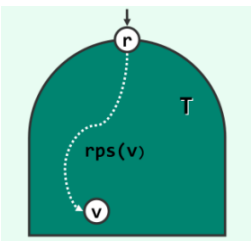
简称 PFC 编码：各字符的编码串互不为前缀。整个解码过程，对信息比特流的扫描不必回溯。

### 5.6.2 二叉编码树

① 根通路与结点编码

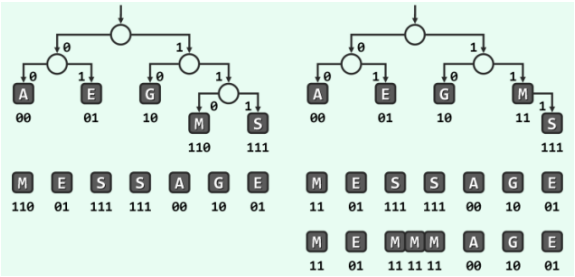
从根结点出发，向左对应 0，向右对应 1。如图，从根结点到每个结点的唯一通路，可以为各结点 v 赋予一个互异的二进制串，称为根通路串(root path string)，记为 rps(v)。长度|rps(v)| = depth(v)就是 v 的深度。

字符 x 对应编码串为 rps(v(x))，简记为 rps(x)。



② **PFC 编码树**

如图是字符集 Σ = {'A', 'E', 'G', 'M', 'S'}某两种编码方案：




右图存在解码歧义的根源在于，编码树中字符'M'是'S'的父亲。

反之，只要所有字符都对应叶结点，自然没有歧义，这是 PFC 编码的策略。

③ 基于 PFC 编码树的解码

对编码串"1100111111100101001"，从前向后扫描，同时在树中相应移动。

起始从树根出发，根据比特位向左或右深入一层，直至抵达叶结点。如扫描"110"后抵达叶结点'M'，输出字符'M'并重新回到根结点，继续扫描剩余部分。

实际上，解码过程甚至可以在二进制编码串接收过程中实时进行，而不必等到所有比特位都到达之后才开始，因此这类算法属于在线(online)算法。

## 🕮 5.7 Huffman 编码