线性表 (Linear List)

线性表是一种抽象的数据结构，它定义了一组数据的逻辑关系。它的特点是：

1. **有穷性**：数据元素的个数是有限的。
2. **有序性**：数据元素之间存在着一对一的线性关系，除了第一个和最后一个元素外，每个元素都有且只有一个前驱和后继。
3. **同质性**：所有数据元素都具有相同的数据类型。

线性表可以采用不同的存储结构来实现，主要有两种：

1. **顺序存储结构 (Sequential Storage Structure)**：也称为顺序表。
2. **链式存储结构 (Linked Storage Structure)**：也称为链表(如单链表、双向链表、循环链表等）。

所以，**顺序表是线性表的一种具体的实现方式**。

顺序表 (Sequential List)

顺序表是指将线性表中的数据元素按照其逻辑顺序，**依次存储到一块连续的物理存储空间中**。

它的主要特点是：

- **物理存储上的连续性**：相邻的元素在内存中也是相邻的。
- **支持随机访问**：可以通过下标(索引）直接访问任何一个元素，时间复杂度为 $O(1)$。
- **插入和删除操作可能涉及大量元素的移动**：在表的中间位置进行插入或删除操作时，为了保持存储的连续性，可能需要移动其后的所有元素，时间复杂度为 $O(n)$。
- **存储空间预分配或动态扩展**：通常需要预先分配一块足够大的存储空间，或者在空间不足时进行动态扩容(这通常意味着需要重新分配更大的内存空间，然后将所有元素复制过去）。

特例法 n=1 yyds

特性	链式结构(链表)	顺序结构(顺序表/数组)
存储方式	非连续存储，通过指针链接	连续存储
随机访问	不支持 ($O(n)$)	支持 ($O(1)$)
插入/删除	灵活高效 ($O(1)$，已知位置或前驱)	中间位置低效 ($O(n)$)
存储密度	低(有指针开销)	高(无每个元素的指针开销)
空间利用	动态分配，无碎片浪费	可能有预留空间浪费，但数据紧凑
缓存友好性	较差	较好
内存要求	无需大块连续内存	需要大块连续内存

因此，我们不能简单地说链式结构优于顺序结构。正确的说法应该是：

- **如果需要频繁地在中间位置进行插入和删除操作，并且对随机访问要求不高，链式结构(如链表）是更好的选择。**
- **如果需要频繁地进行随机访问(通过索引），并且对存储密度和缓存性能有较高要求，顺序结构(如数组）是更好的选择。**

1. 牺牲一个存储单元的方式 (常用)

这种方法下，队头指针 `front` 指向队列的第一个元素，队尾指针 `rear` 指向队列最后一个元素的**下一个空闲位置**。队空时 `front == rear`，队满时 `(rear + 1) % size == front`。

长度计算公式：

$$\text{长度} = (\text{rear} - \text{front} + \text{size})\%\text{size}$$

解释：

- `rear - front`：如果 `rear >= front`(即队列没有“跨过”数组的末尾），那么 `rear - front` 就是元素的数量。
- `rear - front + size`：如果 `rear < front`(即队列“跨过”了数组的末尾，队尾在队头之前），那么 `rear - front` 会得到一个负数。加上 `size`(数组总容量) 可以将其转换为正数。
- `% size`：最后对 `size` 取模，是为了处理两种情况的统一。
  - 当 `rear >= front` 时，`rear - front` 可能是 `size - 1`(队满），`% size` 得到 `size - 1`。
  - 当 `rear < front` 时，`rear - front + size` 得到的是正确长度，`% size` 确保结果在 `[0, size-1]` 范围内。
  - 队空时 `front == rear`，公式结果为 `(0 + size) % size = 0`，正确。

循环队列如何解决“假溢出”：

循环队列的核心思想是：**将数组看作一个环形结构，当队尾指针到达数组末尾时，它可以“绕回”到数组的开头，继续使用前面的空闲空间。**

为了实现这种“循环”效果，我们通常使用**取模运算 (%)**：

- **队头指针 (front)**：指向队列的第一个元素。
- **队尾指针 (rear)**：指向队列的最后一个元素的**下一个位置**（通常这样定义，方便入队操作）。
- **队列大小 (size)**：数组的总容量。

操作原理：

1. **初始化**：
   - `front = -1`（或 `0`）
   - `rear = -1`（或 `0`）
   - 为了区分队空和队满，通常会牺牲一个存储单元，即队列最多只能存储 `size - 1` 个元素。或者有其他判断方式。
2. **入队 (Enqueue)**：
   - 首先判断队列是否已满。
   - 如果未满，`rear = (rear + 1) % size`。
   - 将新元素存入 `array[rear]`。
   - 如果队列原来是空的，则 `front = 0`。
3. **出队 (Dequeue)**：
   - 首先判断队列是否为空。
   - 如果非空，取出 `array[front]` 的元素。
   - `front = (front + 1) % size`。
   - 如果出队后队列变为空（即 `front == rear`），则将 `front` 和 `rear` 重置为 `-1`（或初始状态）。

“串”在数据结构中，特指**字符串 (String)**。

它是一种特殊的线性表，其特殊性体现在：

1. **数据元素是字符：** 串中的每一个数据元素都是一个**字符**（例如：'a', 'B', '!', '7', '中' 等）。这是它与普通线性表最大的区别。普通的线性表的数据元素可以是任何类型（整数、浮点数、自定义对象等）。
2. **操作的特殊性：** 串的操作通常不是简单的增删改查单个元素，而是针对**子串**进行操作，例如：
   - **查找子串：** 判断一个串是否包含另一个串。
   - **提取子串：** 从一个串中截取一部分形成新串。
   - **连接（连接）：** 将两个或多个串拼接在一起。
   - **替换子串：** 将串中的某个子串替换为另一个串。
   - **比较：** 比较两个串的大小（通常按字典序）。
   - **模式匹配：** 查找某个特定模式的子串在主串中的位置（如 KMP 算法）。

从线性表的角度理解串：

- **线性性：** 串中的字符是按照一定的顺序排列的，每个字符除了第一个和最后一个，都有唯一的前驱和后继。这符合线性表的定义。
- **同质性：** 串中的所有元素都是字符类型，这也符合线性表的同质性要求。
- **有穷性：** 串的长度是有限的，包含的字符数量是有限的。

总结来说，串就是由零个或多个字符组成的有限序列。

- **零个字符的串称为空串。**
- **只包含一个空格字符的串称为空格串。**

希尔排序正是为了解决插入排序的这个缺点而诞生的。它的核心思想是：

1. **分组进行插入排序：** 先将整个待排序序列分割成若干个子序列，对这些子序列分别进行插入排序。
2. **逐步减小增量：** 子序列的构成不是连续的，而是跳跃式的。它通过选择一个“增量”（或“步长” `gap`），将相距 `gap` 个位置的元素看作一个子序列。
3. **最终进行直接插入排序：** 随着排序的进行，增量会逐渐减小，直到增量为 1。当增量为 1 时，希尔排序就变成了标准的插入排序。

五大排序

算法名称	时间复杂度 (平均/最差)	空间复杂度 (平均)	稳定性	优点	缺点	适合场景
冒泡排序	$O(n^2)$	$O(1)$	稳定	简单易懂	效率低，比较和交换次数多	教学、小规模数据、基本有序数据
选择排序	$O(n^2)$	$O(1)$	不稳定	简单，交换次数少	效率低，比较次数多，与初始顺序无关	教学、小规模数据、对交换次数有要求
插入排序	$O(n^2)$	$O(1)$	稳定	简单，对小规模和基本有序数据高效，在线算法	效率低 (无序大规模数据)，移动次数多	小规模数据、基本有序数据、在线排序
快速排序	$O(n \log n)$	$O(\log n)$	不稳定	效率高 (平均)，原地排序 (经典)	最坏情况 $O(n^2)$，不稳定，递归开销	通用排序、大规模数据
归并排序	$O(n \log n)$	$O(n)$	稳定	效率高且稳定，适合外部排序	需要额外 $O(n)$ 空间，实现相对复杂	稳定性要求高、大数据集、外部排序、链表排序

冒泡 交换实现

选择 找最小换

插入 扑克牌

快排：均O(nlogn) 分治，找元素，将数组做大于/小于此元素一分为二，递归

归并 O(nlogn) 分划分，再用双指针合并两个有序列表

经过 k 趟（次）冒泡排序，最后 k 个元素就一定会被排好序

寻找最小： 在未排序的子数组中找到最小元素（或最大元素，取决于升序还是降序）。

交换位置： 将找到的最小元素与未排序子数组的第一个元素进行交换

对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。

基于上述定义，完全二叉树具有以下重要性质：

1. **节点数量与层数的关系：**
   - 如果一棵完全二叉树的深度为 k，那么它的节点总数 n 满足：$2^{k-1} \leq n < 2^k$。
   - 特别地，如果它是满二叉树，则 $n = 2^k - 1$。
2. **用数组存储的优势：**
   - 由于完全二叉树的节点编号是连续的，所以**最适合用一维数组（或列表）来存储**。
   - 如果将根节点存储在数组索引 1 (或 0) 的位置：
     - 若根节点在索引 1 处(更常见于算法描述，方便计算)：
       - 对于任意节点 i ($1 \leq i \leq n$):
         - 其左孩子节点索引为 $2i$。
         - 其右孩子节点索引为 $2i + 1$。
         - 其父节点索引为 $\lfloor i/2 \rfloor$。
     - 若根节点在索引 0 处：
       - 对于任意节点 i ($0 \leq i < n$):
         - 其左孩子节点索引为 $2i + 1$。
         - 其右孩子节点索引为 $2i + 2$。
         - 其父节点索引为 $\lfloor (i - 1)/2 \rfloor$ (当 $i > 0$ 时)。
   - 这种数组存储方式不仅节省了存储节点的指针空间，还使得查找父子节点关系非常高效($O(1)$)。
3. **叶子节点和非叶子节点的数量关系：**
   - 如果一棵完全二叉树有 n 个节点，那么叶子节点的数量为 $\lceil n/2 \rceil$ (或 $n_0 = \lfloor n/2 \rfloor + 1$)。
   - 非叶子节点的数量为 $\lfloor n/2 \rfloor$ (或 $n_1 = \lfloor n/2 \rfloor$)。
   - 这是因为所有非叶子节点都至少有一个孩子，而所有叶子节点都没有孩子。
4. **度为1的节点（只有一个孩子）最多只有一个：**
5. **拥有 n 个节点的完全二叉树的高度/深度：**
   - 一棵有 n 个节点的完全二叉树的深度(或高度，从根到最远叶子节点的最长路径上的边数+1，或节点数）为 $\lfloor \log_2 n \rfloor + 1$。
   - 例如，1个节点深度1，3个节点深度2，7个节点深度3。
   - 对于 n 个节点的完全二叉树，其高度是唯一的。

(边数+1)

根处为1

shell希尔排序 gap序列对插入排序的改进，分组进行插入排序

```
数据结构
├── 逻辑结构
│   ├── 集合结构
│   ├── 线性结构：线性表、栈、队列、串
│   ├── 非线性结构：数组、树、堆、图
│   └── 树形结构
└── 物理结构 (存储结构)
    ├── 顺序存储
    ├── 链式存储
    ├── 索引存储 (Indexing)：B树、B+树等，适合有序数据和范围查询
    └── 散列存储 (Hashing)：在单一键值查找上表现优异
```

逻辑结构与存储结构的关系
分离但相互关联： 逻辑结构是抽象的，存储结构是具体的。一个逻辑结构可以有多种存储结构来实现，而一个存储结构也可以用于实现多种逻辑结构。
存储结构是逻辑结构的载体： 数据的逻辑关系必须通过存储结构来体现。例如，线性逻辑关系可以通过顺序存储（数组）或链式存储（链表）来实现。
选择存储结构的依据： 选择哪种存储结构来实现特定的逻辑结构，取决于具体的应用需求，包括：
操作的频率： 哪些操作（查找、插入、删除、遍历等）最频繁？
数据量大小： 数据是小规模还是大规模？
空间利用率： 是否对内存占用有严格要求？
访问模式： 需要随机访问还是顺序访问？

常见排序算法的稳定性：

- **稳定排序算法：**
  - 冒泡排序 (Bubble Sort)
  - 插入排序 (Insertion Sort)
  - 归并排序 (Merge Sort)
  - 计数排序 (Counting Sort)
  - 桶排序 (Bucket Sort)
  - 基数排序 (Radix Sort)
- **不稳定排序算法：**
  - 选择排序 (Selection Sort)
  - 快速排序 (Quick Sort)
  - 堆排序 (Heap Sort)
  - 希尔排序 (Shell Sort)

已知总节点数n，求叶子节点数　①-满二叉树(每层都满)　$N=2^k-1$ (k为深度

叶子节点 2^{k-1}

②完全二叉树 $\lceil \frac{n}{2} \rceil$ ：前k-1层满，最后一层可能未满(集中在左边)

③真二叉树　无单子节点的节点　满足 $N_{叶}=N_{非叶}+1$　N_{0} 比 $\lceil \frac{17}{2} \rceil$

不能通过前后序确定二叉树　前中/中后✓

完全二叉树适合顺序存储非完全适合链式

(内存小缓存快有指针)

无向图 边权不同，最小生成树唯一

空间

{深搜 平均 $\log V$

广搜 最快✓ (求不对称 O(权值))

- **通用关系 (任意树):**
  - 边数 $E = N - 1$
  - 所有节点出度之和 $ = N - 1$ (如果 N_k 指出度为 k 的节点数量)
  - $N_1 + 2N_2 + \cdots + mN_m = N - 1$
  - $N = N_0 + N_1 + N_2 + \cdots + N_m$
- **二叉树特有关系:**
  - **叶子节点数量 N_0 总是比度为 2 的节点数量 N_2 多 1。即 $N_0 = N_2 + 1$。**
  - 这个关系式在分析二叉树问题时非常有用。

建立一个n个结点的二叉堆 $O(n)$

堆排序 $O(n\log n)$

二叉搜索树的删除：左子树最右侧代替 / 右子树最左侧代替

单子节点→直接继承