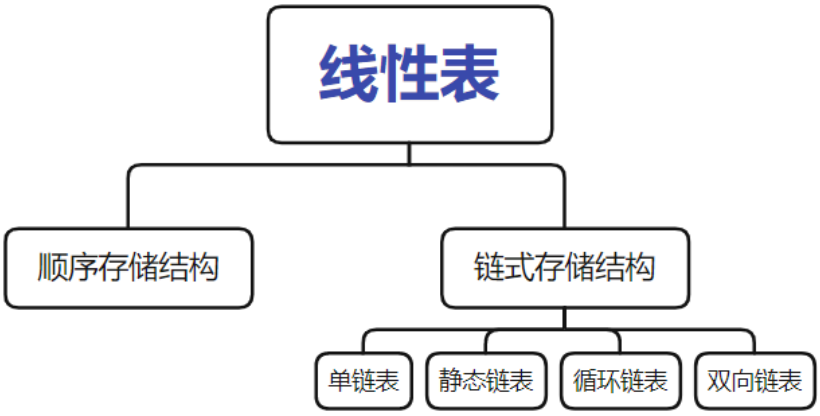
线性表



线性表（List）：由 $n(n \geq 0)$ 数据 $a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n$ 元素组成的有限序列；若元素存在多个，则第一个元素无前驱，最后一个元素无后继，其他每个元素都有且只有一个前驱和后继；

- a_{i-1} 是 a_i 的**直接前驱元素**，a_{i+1} 是 a_i 的**直接后继元素**；
- **线性表的长度**：元素个数 $n(n \geq 0)$；当 $n = 0$ 时，称为**空表**；
- i 为数据元素 a_i 在线性表中**位序**；
- 同一线性表中的元素必须具有相同的特性，数据元素之间的关系是线性关系；

```
ADT 线性表（List）
Data
  线性表的数据对象集合为 {a1,a2,...,an}，每个元素的类型均为 DataType。其中，除第一个元素为，每个元素有且只有一个直接前驱元素，除了最后一个元素为，每一个元素有且只有一个直接后继元素。数据元素之间的关系是一对一的关系。
Operation
  InitList(*L);        //初始化操作，建立一个空的线性表
    ListEmpty(L);      //判断线性表是否为空
    ClearList(*L);     //清空
    GetElem(L,i,*e);   //索引，将线性表 L 中的第 i 个位置元素返回给 e
    LocateElem(L,e);   //查询，将线性表 L 中查找与给定值 e 相等的元素，如果查找成功，返回该元素在表中序号表示成功；否则，返回 0 表示失败
    ListInsert(*L,i,e);   //插入，在线性表 L 中的第 i 个位置插入新元素 e
    ListDelete(*L,i,*e);  //删除，删除线性表 L 中的第 i 个位置元素，并用 e 返回其值
    ListLength(L);        //长度，返回线性表 L 的元素个数
endADT
```

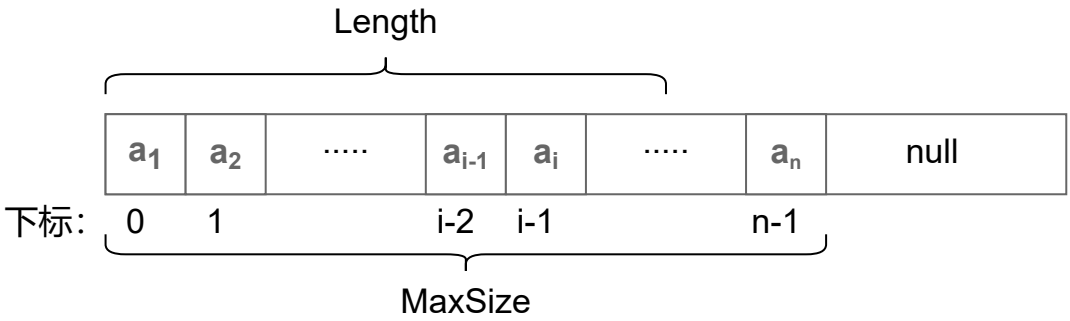
线性表的顺序存储结构

指的是用一段**地址连续**的存储单元依次存储线性表的数据元素；利用数据元素的存储位置表示线性表中相邻数据元素之间的前后关系，即线性表的逻辑结构与存储结构一致；

线性表的 3 个基本属性：

- **最大存储容量**：数组长度 MAXSIZE；
- **当前长度**：Length；
- **起始位置**：数组 data 的存储位置就是存储空间的存储位置；其**地址计算方式**为：假设占用的是 c 个存储单元，LOC 表示获得存储位置的函数

$$LOC(a_i) = LOC(a_1) + (i-1) * c \tag{1}$$



插入元素

- 插入位置不合理，抛出异常；
- 如果线性表长度大于等于数组长度，则抛出异常或动态增加容量；
- 遍历：从最后一个元素开始便利到第 i 个位置，分别将他们都往后移动一个位置；
- 将要插入元素填入位置 i 处；
- 表长加 1；

删除元素

- 如果删除位置不合理，抛出异常；
- 去除删除元素；
- 遍历：从删除元素开始遍历到最后一个元素，分别将它们都向前移动一个位置；
- 表长减 1；

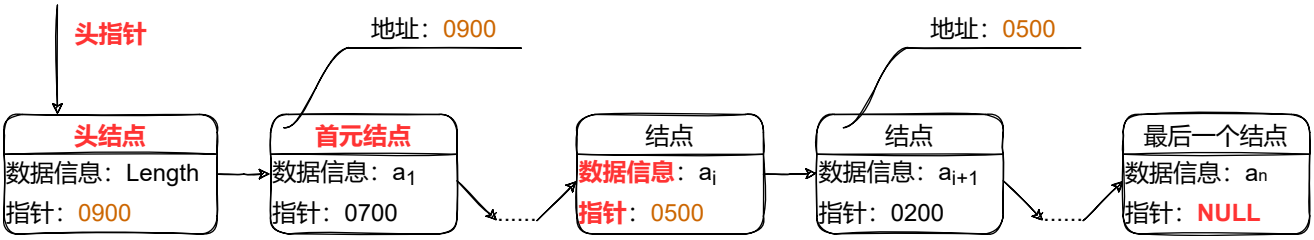
线性表的链式存储结构

用一组**任意**的存储单元存储线性表的数据元素，即数据元素可以存在内存未被占用的任意位置。

链表：n 个结点链结在一起；

结点（Node）：数据元素 a_i 的指针域和数据域组成的存储映像；

- **数据域**：存储数据元素信息的域；
- **指针域**：存储直接后继位置的域；



> **首元结点**：第一个数据元素的结点；
>
> **头指针**（必要）：指向第一个结点（包括头结点）的存储位置；
>
> - 头指针具有标志作用，所以常用头指针冠以链表的名字；
> - 无论链表是否为空，头指针均不为空；
>
> **头结点**（非必要）：首元结点前附设一个结点。头结点是为了使首元结点的操作和其他结点一致定义的，如在首元结点前插入结点和删除首元结点。
>
> - 头结点的数据域：不存储任何信息/存储线性表的长度。
> - 头结点的指针域：存储指向首元结点的指针。
> - 若线性表为空表，则头结点的指针域为空；

以下操作以单链表为例：重要操作：

```
p=L;          //p 指向头结点；
s=L->next;    //s 指向首元结点；
p=p->next;    //p 指向下一结点；
```

> 注：
>
> - **删除时最重要的是用指针保存下一个结点的地址**。
> - 对结点进行操作，注意考虑指向下一结点的指针是否有效；
> - 遍历链表结点时，循环结束的条件是当前结点的指针域是否为 nullptr。

链表初始化

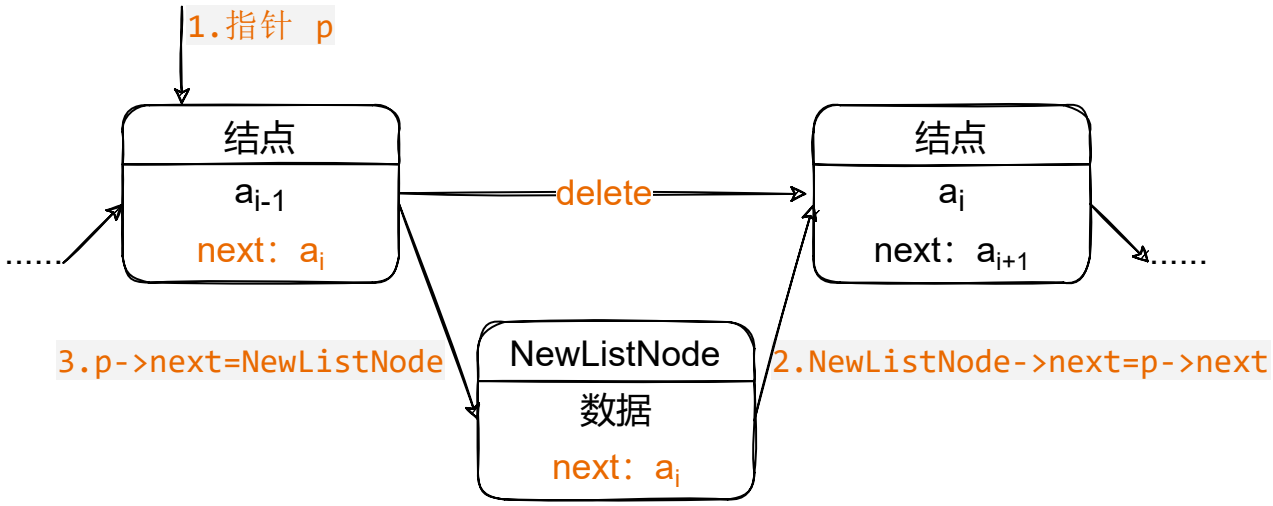
1. 生成新结点作为头结点；
2. 将头结点指针域置空；
3. 用头指针 `headptr` 指向头结点；

获取第 i 个元素：从首元结点开始遍历，计数器 `j` 累计，直到 `j==i`；若到链表指针为空，则说明第 `i` 个结点不存在；

查找元素 elem：从首元结点开始遍历比较结点数据与 `elem`；

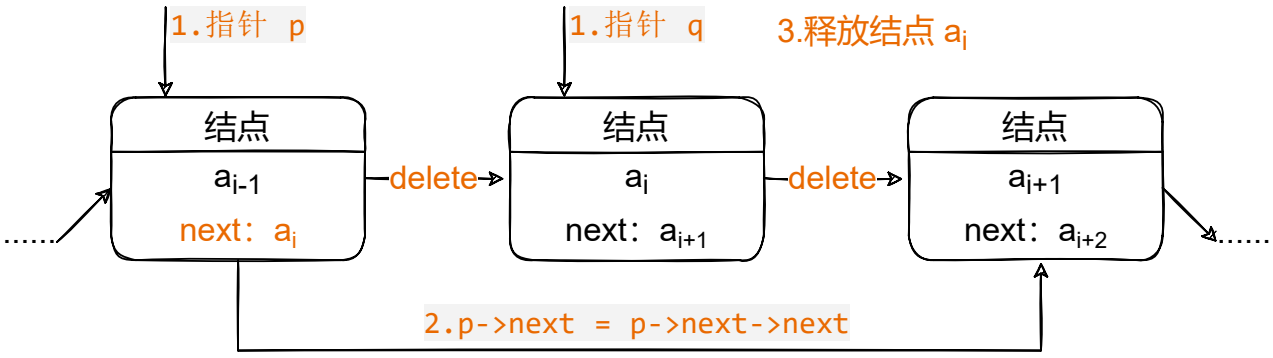
在第 i 个结点前插入元素：

1. 遍历查找 a_{i-1}，令指针 `p` 指向该节点；
2. 新结点 `NewListNode` 先连后继 a_i；
3. 再连前驱 a_{i-1}；



删除第 i 个元素：

1. 遍历查找 a_{i-1}，令指针 `p` 指向该节点，指针 `q` 指向 a_i；
2. 令结点 a_{i-1} 指向 a_{i+1}，`p->next = p->next->next;`
3. 释放结点 a_i 的空间；



链表的创建：创建单链表的过程就是一个动态、生成结点的过程。

1. 链表初始化；
2. 循环：将新结点 `NewListNode` 插入到尾结点之后；

链表的清空：

1. 令指针 `p` 指向首元结点 `p=headptr->next;`
2. 循环判断 `p` 是否为空：
    1. 令指针 `q` 指向下一个结点；
    2. 释放指针 `p` 所指的结点；
    3. 将 `q` 赋值给 `p` ；
3. 将头结点的指针置空 `headptr->next=null;`

小结

链表的分类

- **单链表**：链表中每个结点中只包含一个指针域；
- **循环链表**(circular linked list)：将单链表中令尾结点的指针指向头结点。
- **双向链表**(double linked List)：在单链表的每个结点中，再设置一个指向其前驱结点的指针。
- **静态链表**：为了给没有指针的高级语言设计的一种实现单链表能力的方法，**一般用不上**；

说明：

- 循环列表的作用：循环链表解决了一个很麻烦的问题，如何从当中一个结点出发，访问到链表的全部结点。
- 双向链表比单链表多了可以反向遍历查找，其代价是插入/删除时需要改变两个指针变量；
- **尾指针**的作用：在单链表中，访问首元结点的时间为 $O(1)$ （有头结点 ）；访问最后一个结点的时间为 $O(n)$ （需要遍历链表所有元素）。通过尾结点，查找终端结点是 $O(1)$ ，而首元结点，其实就是 `rear->next->next` ，其时间复杂也为 $O(1)$ 。

顺序表和链表的比较

随机存取法：在访问线性表时，可以快速地计算出任何一个数据元素的存储地址。因此可以粗略地认为，访问每个元素所花时间相等 $O(1)$，这种存取元素的方法被称为随机存取法；

顺序存取法：访问只能通过头指针进入链表，通过每个结点的指针域依次向后顺序扫描其余结点，这种存储方式称为顺序存取法；

顺序表

- 优点：
  - 可以快速的读取表中任一位置的元素；
  - 逻辑结构与物理结构一致（无须为表示表中元素之间的逻辑关系而增加额外的存储空间）；
- 缺点：
  - 在插入、删除某一元素时，需要移动大量元素；
  - 存储空间分配不灵活，数据元素的个数不能自由扩充；

链式存：

- 优点：
  - 结点空间可以动态申请和释放；
  - 数据元素的逻辑次序靠结点的指针来指示，插入和删除时不需要移动数据元素。
- 缺点：
  - 存储密度小，每个结点的指针域需额外占用存储空间。当每个结点的数据域所占字节不多时，指针域所占存储空间的比重显得很大。
  - 链式存储结构是非随机存取结构。对任一结点的操作都要从头指针依指针链查找到该结点，这增加了算法的复杂度。

单链表的查找、插入、删除算法时间效率分析

比较项目 \ 存储结构		顺序表	链表
空间	存储空间	预先分配，会导致空间闲置或溢出现象	动态分配，不会出现存储空间闲置或溢出现象
	存储密度	不用为表示结点间的逻辑关系而增加额外的存储开销，存储密度等于1	需要借助指针来体现元素间的逻辑关系，存储密度小于1
时间	存取元素	随机存取，按位置访问元素的时间复杂度为O(1)	顺序存取，按位置访问元素时间复杂度为O(n)
	插入、删除	平均移动约表中一半元素，时间复杂度为O(n)	不需移动元素，确定插入、删除位置后，时间复杂度为O(1)
适用情况		① 表长变化不大，且能事先确定变化的范围 ② 很少进行插入或删除操作，经常按元素位置序号访问数据元素	① 长度变化较大 ② 频繁进行插入或删除操作

> 说明：插入删除结点时，因线性链表不需要移动元素，只要修改指针，一般情况下时间复杂度为$O(1)$。但是，如果要在单链表中进行前插或删除操作，由于要从头查找前驱结点，所耗时间复杂度为$O(n)$

code

> 顺序存储结构代码.c
>
> 顺序存储结构代码.cpp
>
> 线性表的链式存储代码.c
>
> 线性表的链式存储代码.cpp