数据结构与算法B

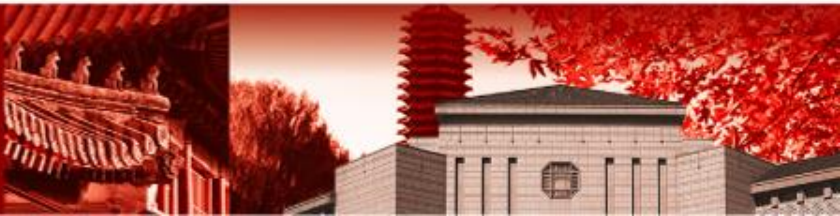
02 – 线性表

线性表

- 回忆：数据结构三要素
  - 逻辑结构：线性结构
  - 存储结构
    - 顺序存储（顺序表）
    - 链式存储（链表）
  - 数据运算：增、删、改、查

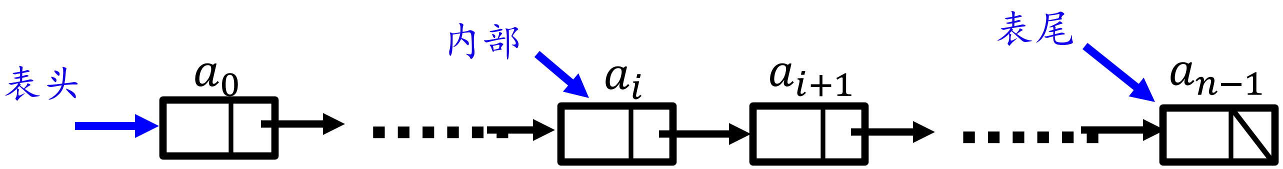
目录

2.1 线性结构



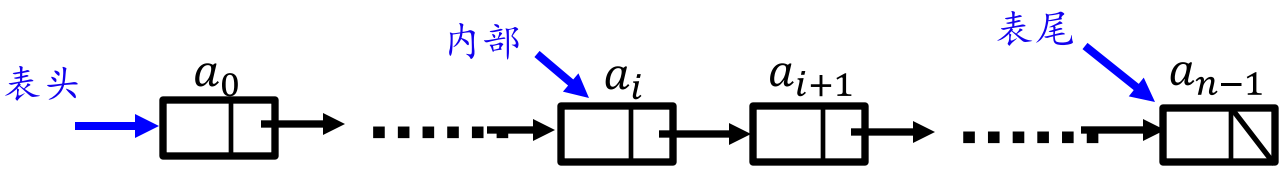
线性结构

- 线性表的逻辑结构：线性结构
- 线性结构：$\{a_0, a_1, \ldots, a_{n-1}\}$
  - **有序**的有限元素序列
  - 每个元素存在**唯一的前驱与后继**（除第一个与最后一个外）
  - 唯一的**开始节点**，它没有前驱，有一个唯一的直接后继
  - 唯一的**终止结点**，它有一个唯一的直接前驱，而没有后继
  - 其它的结点皆称为**内部结点**



线性结构

- 线性结构的特点
  - **均匀性**：虽然不同线性表的数据元素可以是各种各样的，但对于同一线性表的各数据元素必定具有相同的数据类型和长度
  - **有序性**：各数据元素在线性表中都有自己的位置，且数据元素之间的相对位置关系是线性的
- 用下标表示元素的位置，由0开始计数
- n为线性表长度，n = 0时称为空表

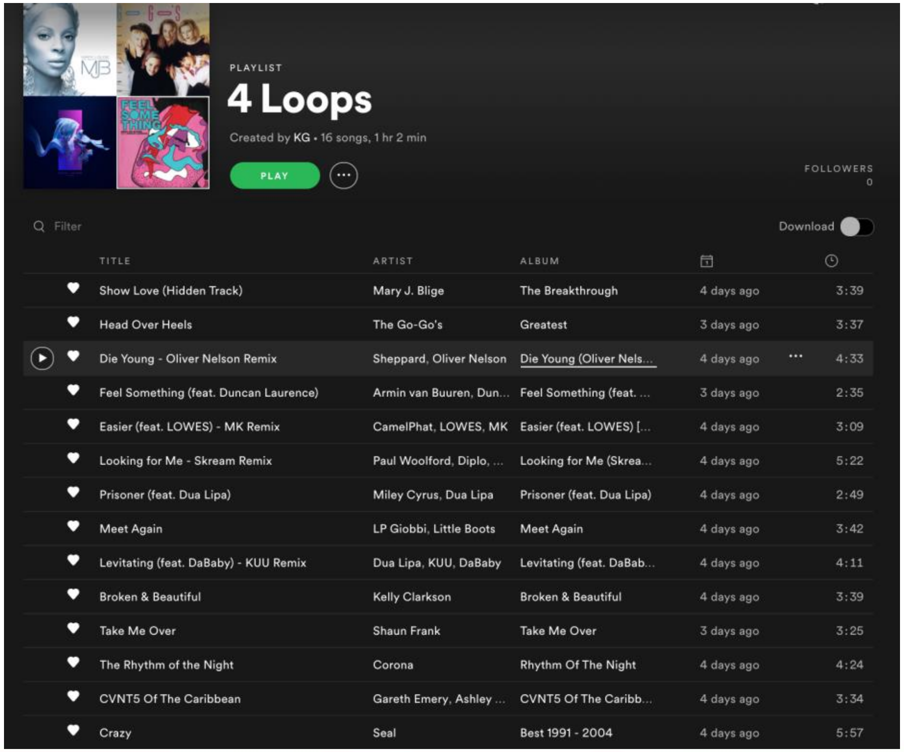


线性结构的应用

- 现实问题中，经常需要将一组同一类型的元素作为整体管理，并维护元素间的前驱与后继关系
- 例如：
  - 排队过程：队列

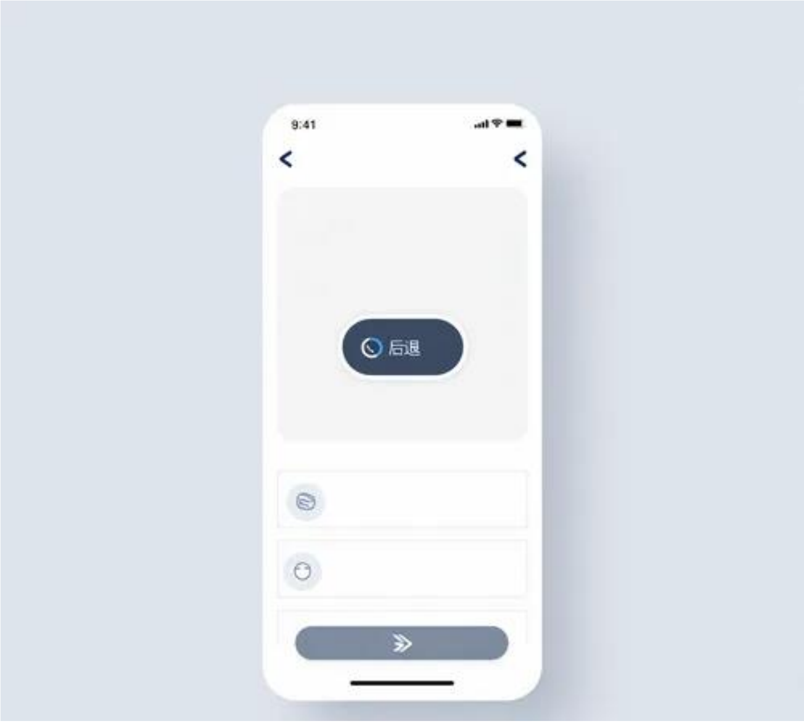
线性结构的应用

- 现实问题中，经常需要将一组同一类型的元素作为整体管理，并维护元素间的前驱与后继关系
- 例如：
  - 排队过程：队列
  - 歌曲播放列表：循环链表



线性结构的应用

- 现实问题中，经常需要将一组同一类型的元素作为整体管理，并维护元素间的前驱与后继关系
- 例如：
  - 排队过程：队列
  - 歌曲播放列表：循环链表
  - 软件界面的进入与后退：栈



线性结构的应用

- 在程序中，线性表是最基本的一种数据结构
  - 例如整数的表，字符串的表，某种复杂结构的表等
  - Python 的list 和tuple支持这类需要，可看作是线性表的实现
- 线性表还常作为更复杂的数据结构的实现基础
  - 两种基本实现：顺序表、链表
  - 在基本实现的基础上，加以扩充和限制
  - 例如：栈和队列（顺序实现与链式实现）

线性结构的运算

- 线性表的各种实现通常都需要提供一些“标准”操作
  - 创建和销毁线性表
  - 判断一个数据结构是否为空，如果数据结构的容量有限制，还需判断它是否满
  - 向结构中加入元素或从中删除元素（增删）
  - 读取和写入线性表里的元素（改查）
- 不同编程语言也可能影响需要实现的操作集合
  - 例如，Python 能自动回收不用的对象，因此不需要销毁结构的操作
- 其他基于线性表的操作：
  - 排序所有元素、按值查找元素、连接两个线性表，等等

2.2 顺序表

顺序表

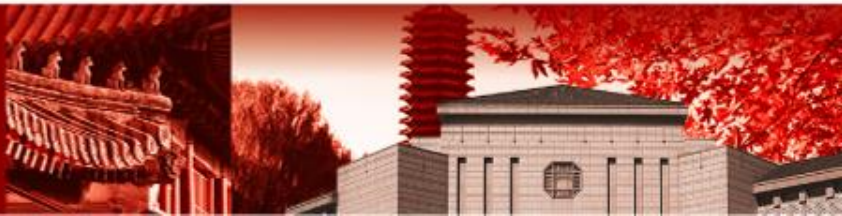
- 顺序表也称向量，是元素在内存中连续存放的线性表。
- 特点

  - 元素顺序地存储在连续存储空间中
  - 每一个元素有唯一的序号（下标）
  - 顺序表最大长度为定值
  - 读写其元素很方便，通过下标即可指定位置，其时间复杂度是O（1）
    - 只要确定了首地址，线性表中任意数据元素都可以随机存取
- 其他编程语言，如C/C++、Jave中的数组，是顺序表
- Python中的列表是顺序表

北京大學

顺序表支持的操作

序号	操作	含义	时间复杂度
1	init(n)	生成一个n个元素的顺序表	O(1)
2	init(a_0,a_1,....a_n)	生成元素为a_0,a_1,.... a_n的顺序表	O(n)
3	len()	求表中元素个数	O(1)
4	append(x)	在表的尾部添加一个元素x	O(1)
5	pop()	删除表尾元素	O(1)
6	get(i)	返回下标为i的元素	O(1)
7	set(i,x)	将下标为i的元素设置为x	O(1)
8	index(x)	查找元素x在表中的位置	O(n)
9	insert(i,x)	在下标i处插入元素x	O(n)
10	delete(i)	删除下标为i的元素	O(n)

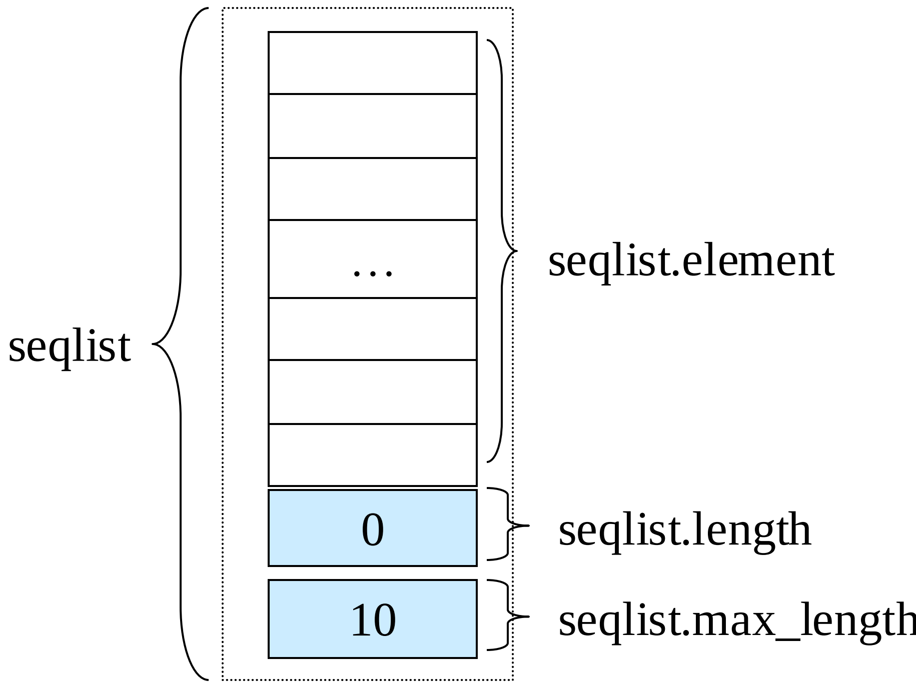


顺序表的创建

```python
class SeqList():
    def __init__(self, length: int):
        if length <= 0:
            raise ValueError(length)
        self.max_length = length
        self.length = 0
        self.element = [None] * length

seqlist = SeqList(10)
```

顺序表的创建

- 创建空表是分配一块存储，并设置记录容量
  - 元素计数值为0，右图是个容量为10的空表
  - 建立新表后应立即设置两个记录域，即seqlist.length和seqlist.max_length，保证表处于合法状态
  - 存储块一旦分配，就有了固定大小，适合创建不变表（如tuple）
  - 对于可变表，分配足以容纳所需元素的存储块，留有一些空位



访问顺序表中的元素

- 访问给定下标 i 的元素：get (i)
  - 需判断 i 值是否在表当时的合法元素范围内（$0 \le i \le n-1$）
  - 不在范围内是非法访问
  - 合法时从给定位置取得元素的值
  - 元素a_i的地址计算公式（元素编号从0 开始）：

    $$Location(a_i) = Location(a_0) + c * i$$

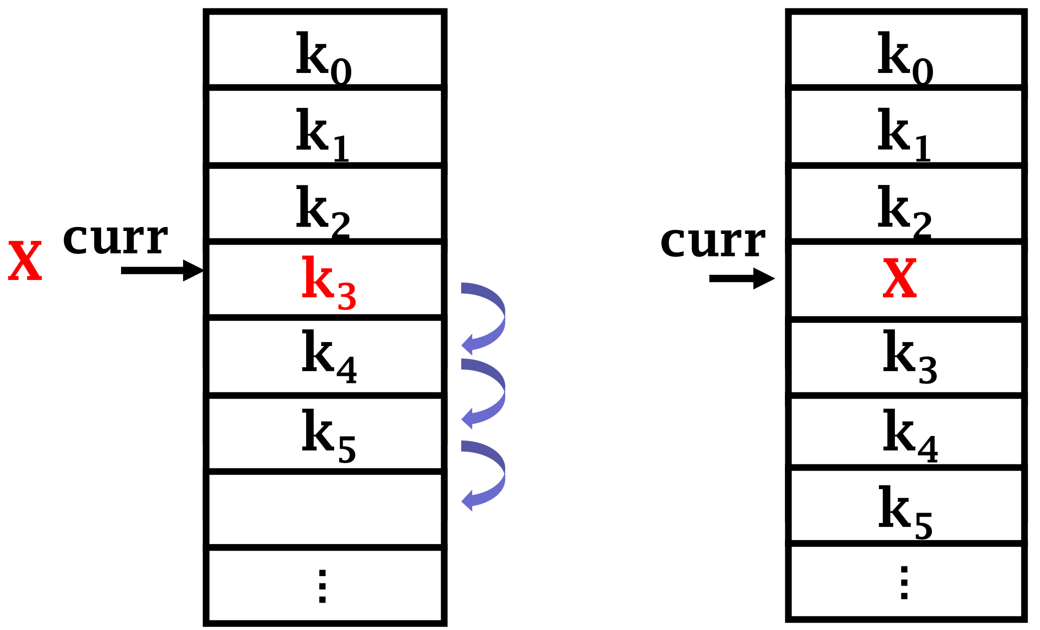
    其中a_0为表头元素，c为元素大小
  - 元素大小通常可以静态确定（如元素是整数，实数，或包含若干大小确定的元素的复杂结构）
  - 以O(1)的代价完成

顺序表中增加/删除元素（表尾操作）

- 尾端加入和删除操作的实现很简单
- 尾端加入元素（O(1) 操作）：append (x)
  - 检查表是否满，表满时操作失败
  - 把新数据存入元素存储区的第 length 个单元（length表示当前长度）
  - 将元素计数变量 length 加一
- 尾端删除元素（O(1) 操作）：pop( )
  - 简单地把元素计数变量 length 减一
- 定位的加入和删除都比较麻烦
  - 因为要保证元素在存储区前段连续存储
  - 可能需要维持原有元素的顺序

顺序表中增加元素（随机操作）

- 在任意位置加入元素（随机操作）时，需要移动已有元素，腾出要求存入元素的位置：insert(i, x)
  - 逆序地逐个后移后面元素，直至腾出指定位置后将元素放入



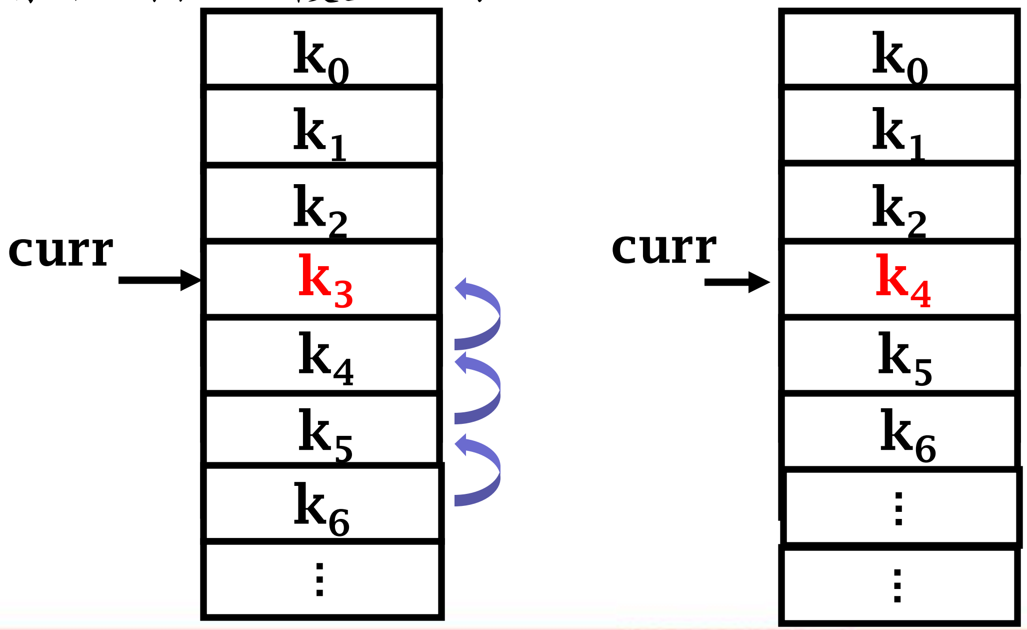
顺序表中增加元素（随机操作）

```python
def insert(self, index: int, value) -> None:
    if index < 0 or index > self.length:
        raise IndexError(index)
    if self.length == self.max_length:
        raise Exception("The sequential list is full.")

    for i in range(self.length, index, -1):
        self.element[i] = self.element[i-1]

    self.element[index] = value
    self.length += 1
```

逆序后移后面的元素
直到腾出位置
注意迭代的边界下标

顺序表中删除元素（随机操作）

- 在任意位置删除元素（随机操作）时，同样需要移动已有元素，保证所有元素连续存储: delete(i)，相当于python中list的pop(i)操作
  - 顺序地逐个前移后面元素，直到所有元素连续存储
  - 待删除元素被自然“覆盖”了



顺序表中删除元素（随机操作）

```python
def delete(self, index: int):
    if index < 0 or index >= self.length:
        raise IndexError(index)

    for i in range(index, self.length - 1):
        self.element[i] = self.element[i+1]
    self.length -= 1
```

➡ 顺序前移后面的元素，直到元素变为连续存储待删除的元素被覆盖

顺序表插入删除的复杂性分析

- 在 n 个元素的顺序表里下标 i 处：
  - 加入元素，需要移动n - i 个元素；
  - 删除元素，需要移动n - i -1个元素
- 设在位置 i 加入和删除元素的概率分别是p_i和p_i’
  - 加入操作平均移动次数 $$\sum_{i=0}^{n}(n-i)p_i$$
  - 删除操作平均移动次数 $$\sum_{i=0}^{n-1}(n-i-1)p_i'$$
- 考虑平均复杂性时要考虑实例分布，依赖于实际情况
  - 如果各种情况平均分布，平均时间复杂度是O(n)
  - 最坏情况是首端加入/删除，时间复杂度也是O(n)

实例：Python的list数据结构

- list 是一种顺序表
  - 基于下标（位置）的元素访问和更新操作，复杂性为O(1)
  - 允许任意加入元素（不会出现“表满”而无法加入新元素的情况）
  - 在不断加入元素的过程中，元素下标索引不变
- list 实现的基本约束和解决方案
  - 要求O(1) 的元素访问并维持元素的顺序，只能采用连续存储
  - 要能容纳任意多元素，必须在表满时换一块更大存储区。

实例：Python的list数据结构

- Python list 的动态扩容：

```python
import sys

my_list = []
print(f"初始容量: {sys.getsizeof(my_list)} 字节")

for i in range(20):
    my_list.append(i)
    print(f"元素数量: {len(my_list)}, 容量: {sys.getsizeof(my_list)} 字节")
```

```
初始容量: 56 字节
元素数量: 1, 容量: 88 字节
元素数量: 2, 容量: 88 字节
元素数量: 3, 容量: 88 字节
元素数量: 4, 容量: 88 字节
元素数量: 5, 容量: 120 字节
元素数量: 6, 容量: 120 字节
元素数量: 7, 容量: 120 字节
元素数量: 8, 容量: 120 字节
元素数量: 8, 容量: 120 字节
元素数量: 8, 容量: 120 字节
元素数量: 8, 容量: 120 字节
元素数量: 8, 容量: 120 字节
元素数量: 9, 容量: 184 字节
元素数量: 10, 容量: 184 字节
元素数量: 11, 容量: 184 字节
元素数量: 12, 容量: 184 字节
元素数量: 13, 容量: 184 字节
元素数量: 14, 容量: 184 字节
元素数量: 15, 容量: 184 字节
元素数量: 16, 容量: 184 字节
元素数量: 17, 容量: 248 字节
元素数量: 18, 容量: 248 字节
元素数量: 19, 容量: 248 字节
元素数量: 20, 容量: 248 字节
```

- 除了存储数据，list中还需要存储其他信息
- 在元素数量超过4，8，16时发生了扩容
  - 如何分析这一扩容策略的复杂度？

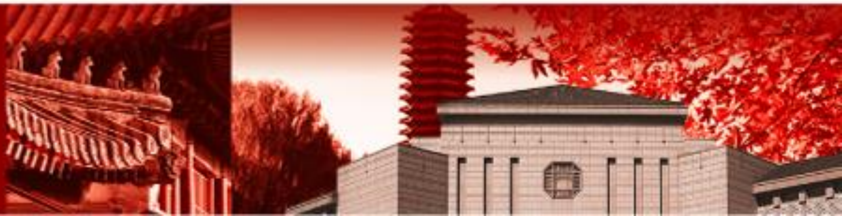
实例：Python的list数据结构

- 动态扩容机制的时间复杂度分析
  - 在容量达到 k 的幂 + 1 时进行扩容
    - 在前述示例中，容量超过4，8，16时，进行扩容（k=2）
  - 方案1：扩容时，新容量是旧容量的k倍，k是大于1的固定值
  - 执行 k^m 次append操作，需要复制的元素个数是
    - $1 + k + k^2 + \ldots + k^{m-1} = (k^m - 1)/(k - 1)$

      注释：假设第一次分配空间，容量为1
  - 因此，在元素总数为 $n = k^m$ 时，平均每次 append 操作需要复制的元素个数是

$$\frac{k^m-1}{k^m(k-1)} = \frac{n-1}{n(k-1)} < \frac{1}{k-1} = O(1)$$

  - 插入操作的平均时间复杂度是O(1)的，最坏情况（即发生扩容时）下的时间复杂度是O(n)的



实例：Python的list数据结构

- 其他共性操作的复杂度由连续表的实现方式决定
  - 元素访问和赋值，尾端加入和尾端（切片）删除是O(1) 操作
  - 一般元素加入，切片，表拼接（extend）等都是O(n) 操作。
  - pop 操作默认情况是尾端删除返回，为O(1)，指定任意位置为O(n)
  - len( ) 是O(1) 操作

2.3 链表

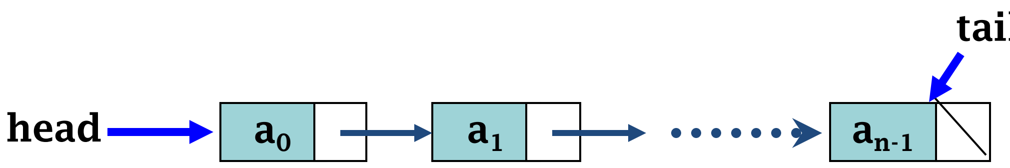
链表

- 采用链式存储结构的线性表
- 通过后继引用把一串存储结点链接成一个链
  - 每个结点存储指向后继结点的引用来表示数据元素之间的逻辑关系
  - 逻辑上相邻的元素在存储位置上不要求也相邻
  - 按照需要为表中新的元素动态地分配存储空间，动态改变长度
- 存储结点由两部分组成：
  - 数据项本身 + 指向后继结点的指针（引用）

数据	指针

单链表

- 要访问一个单链表，仅需要掌握表的首结点。从它：
  - 可以找到表的首元素
  - 还可以找到表中下一结点的位置
- 按同样方式继续下去，就可以找到表里的所有数据元素
  - 表头变量：保存着链表第一个结点的引用的变量
  - 对于空表，表头变量为None，表示第一个结点不存在
  - 根据需要还可以设置表尾变量，保存最后一个结点的引用



单链表的创建/删除

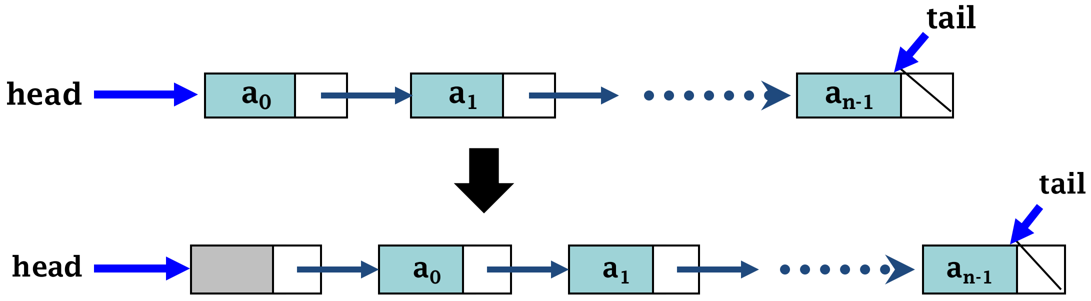
- 创建空链表：只需将表头变量设置为空
  - 在 Python 里将其设置为None
- 删除链表：丢弃表的所有结点，与具体环境有关
  - 在一些语言（如C 语言）里需要做许多事情，手动释放所有存储
  - 在 Python 里，只需简单将表头设为None，就丢掉了整个链表的所有结点。Python 的垃圾回收机制会自动回收不用的空间。

```python
class LinkNode():
    def __init__(self, node_data):
        self.data = node_data
        self.next: LinkNode = None
```

```python
class LinkList():
    def __init__(self):
        self.head: LinkNode = None
        self.length = 0
```

向单链表中插入元素

- 为了将指定元素插入到下标为 i 的位置上：
  - 定位到待插入位置的前驱结点
  - 依次更改待插入结点、前驱结点的引用
- 当然，存在一些稍微复杂的边界情况：
  - 在尾端插入时，不存在后继，将待插入结点的链接域设为None
  - 在首端插入时，不存在前驱，但需要另外设置表头变量
    - 引入一个不存放元素的空的“头结点”作为前驱，以方便应对这一情况：



向单链表中插入元素

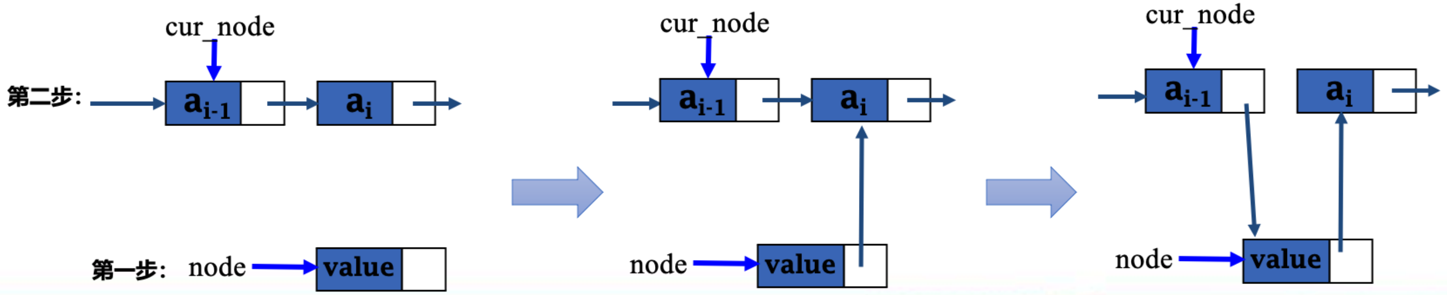
```python
def insert(self, index: int, value):
    if index < 0 or index > self.length:
        raise IndexError(index)
    node = LinkNode(value)

    cur_node, position = self.head, 0
    while position < index:
        cur_node = cur_node.next
        position += 1
    node.next = cur_node.next
    cur_node.next = node
    self.length += 1
```

➡ 创建待插入结点node

➡ 定位到下标为index的位置的前驱

➡ 修改待插入结点的指针

➡ 修改前驱结点的指针，指向待插入结点

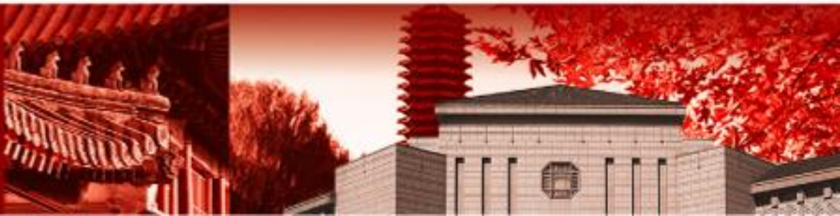


从单链表中删除元素

- 与插入操作类似，为了删除插入下标为 i 的位置的元素：
  - 定位到待插入位置的前驱结点
  - 直接修改前驱结点的指针（引用），使之指向下一个结点
  - 被删除的结点将被自动回收
- 练习：
  - 分析删除时的边界情况
  - 给出带头结点的单链表的删除操作实现

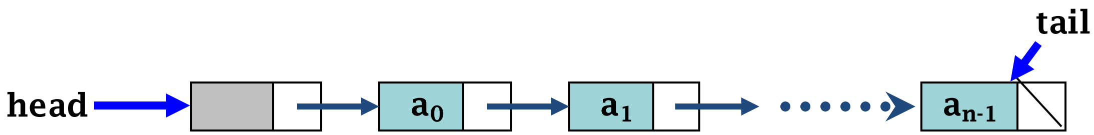
单链表操作的时间复杂性分析

- 在任意位置增加与删除元素
  - 主要代价为定位过程，平均与最坏情况都是O(n)
  - 如果指定在头部操作，则可以以O(1)代价完成
- 按下标读取或写入元素值
  - 与顺序表不同，链式存储结构下必须顺序遍历索引
  - 平均与最坏情况都是O(n)
- 将两个单链表合并
  - 容易想到：为单链表设置表尾变量，就可以快速得到表尾
  - 这种实现下合并代价为O(1)

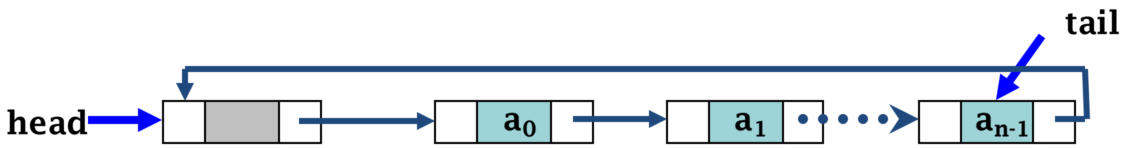


多种链式存储实现

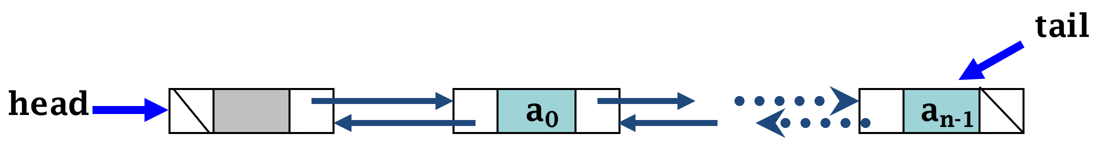
- 单链表



- 循环链表

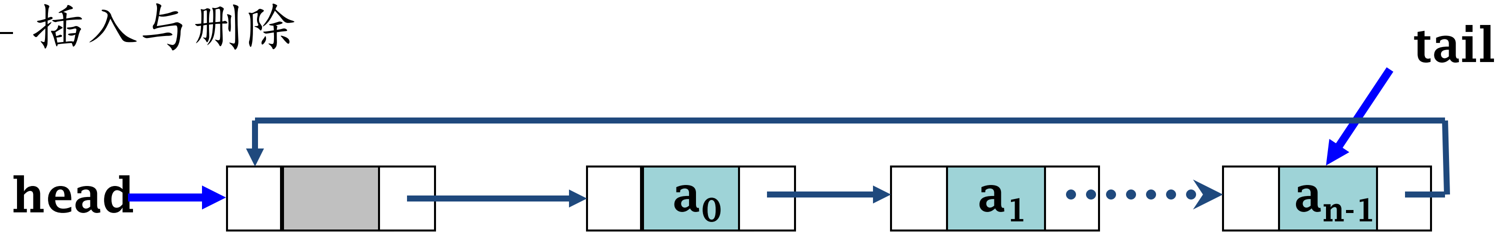


- 双链表



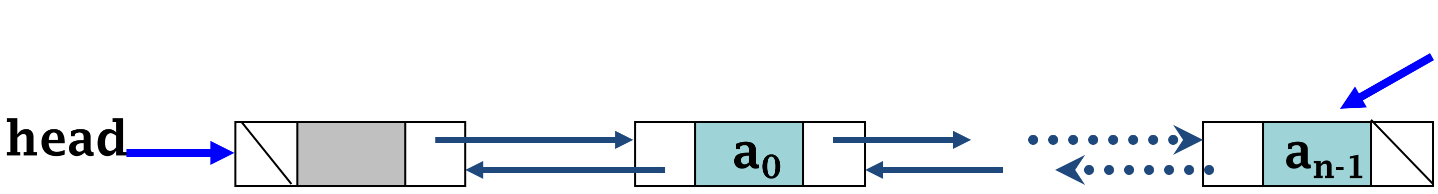
单循环链表

- 将最后一个结点的指针（引用）指向第一个结点（或头结点），构成一个循环链。
  - 从任一个结点出发，可访问表中任何一个结点元素。
  - 能够表示循环的数据关系，逻辑上是对线性结构的扩展
  - 插入与删除



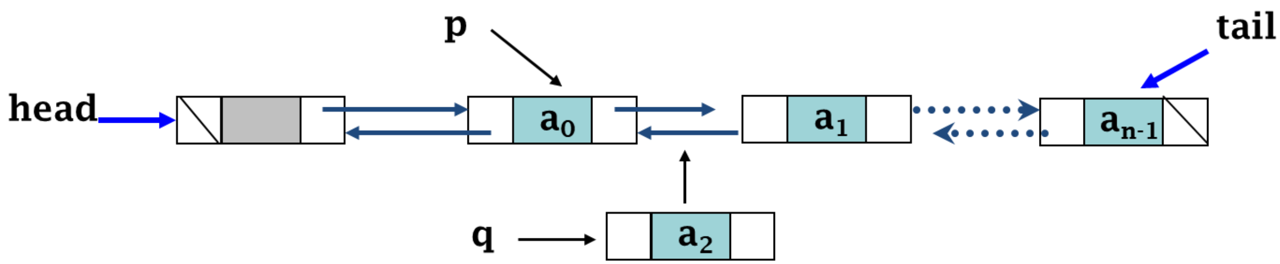
双链表

- 单链表的 next 字段仅仅指向后继结点，而无法访问前驱。
  - 为弥补单链表的不足，增加一个指向前驱的指针
  - 为链表上的遍历提供了更高的灵活性
- 思考：
  - 双链表上的插入、删除操作应该如何进行？



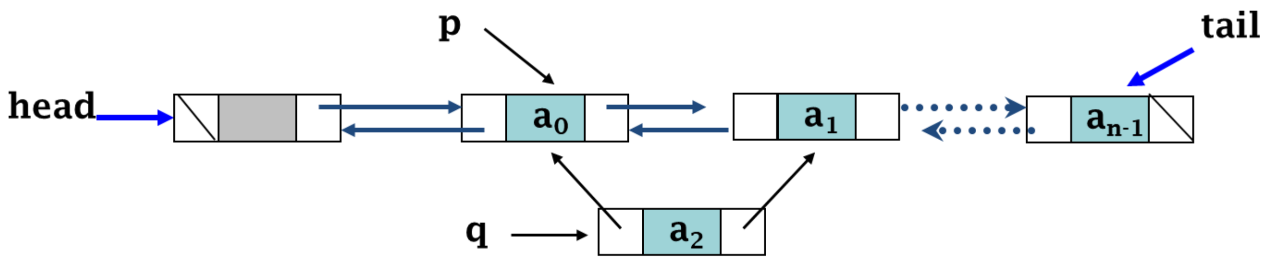
双链表的插入操作

- 插入元素q:



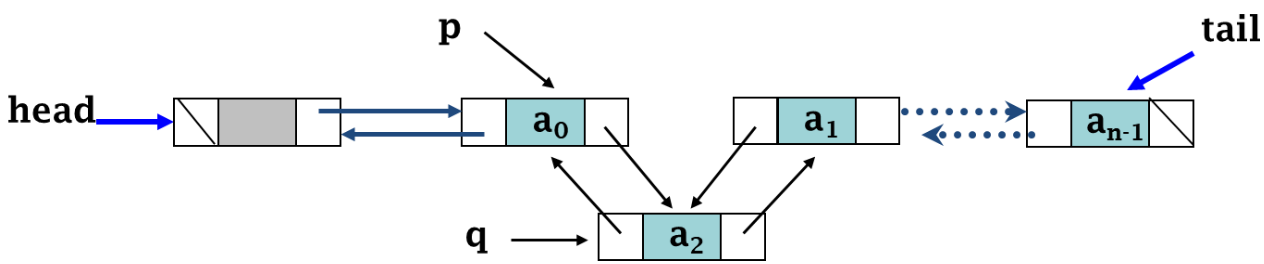
(1) **q.next=p.next**

(2) **q.prev=p**



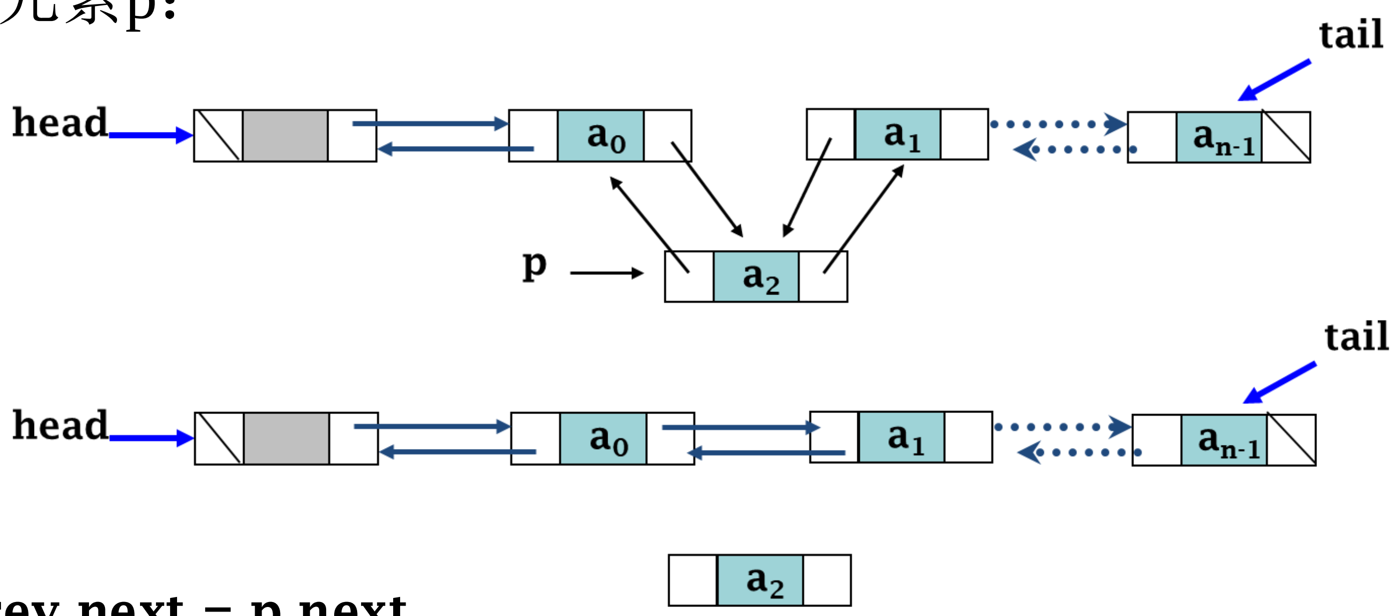
(3) **p.next=q**

(4) **q.next.prev=q**



双链表的删除操作

- 删除元素p:



(1) p.prev.next = p.next

(2) p.next.prev = p.prev

(3) p.prev = None

(4) p.next = None

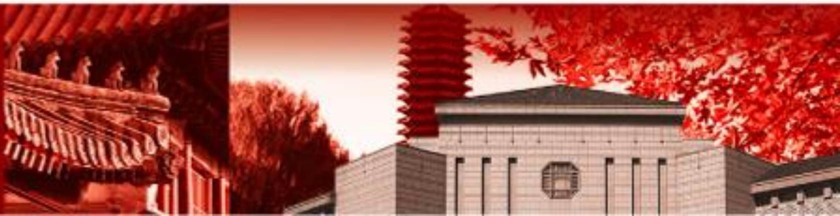
2.4 顺序表和链表的比较

顺序表和链表的比较

- 顺序表
  - 插入、删除运算时间代价 O(n)
  - 按下标的访问则O(1)完成
  - 预先申请固定长度的连续空间
  - 如果整个顺序表元素很满，则没有**结构性存储开销**

指在存储数据本身之外的额外存储开销

- 链表
  - 在给定位置上的插入、删除运算时间代价为O(1)
  - 但找第 i 个元素运算时间代价 O(n)
  - 动态地按照需要为表中新的元素分配存储空间
  - 每个元素都有结构性存储开销



顺序表和链表的比较

- 顺序表的主要优点
  - 无需使用变量引用（指针）来维护线性结构，不用花费额外开销
  - 元素的读访问非常简洁便利
- 链表的主要优点
  - 线性表的长度不受限，允许线性表的长度动态变化
  - 能够适应经常插入删除内部元素的情况
- 总结
  - 顺序表是存储静态数据的不二选择
  - 链表是存储动态变化数据的良方

顺序表和链表的存储密度

- n：线性表中当前元素的数目
- P：引用变量所占内存空间
- E：元素数据所占内存空间
- D：顺序表的最大容量
- **空间需求**
  - 顺序表的空间需求为 DE
  - 链表的空间需求为 n(P + E)
- **n 的临界值，即 n > DE / (P+E)**
  - n 越大，顺序表的空间效率就相对更高

应用场合的选择

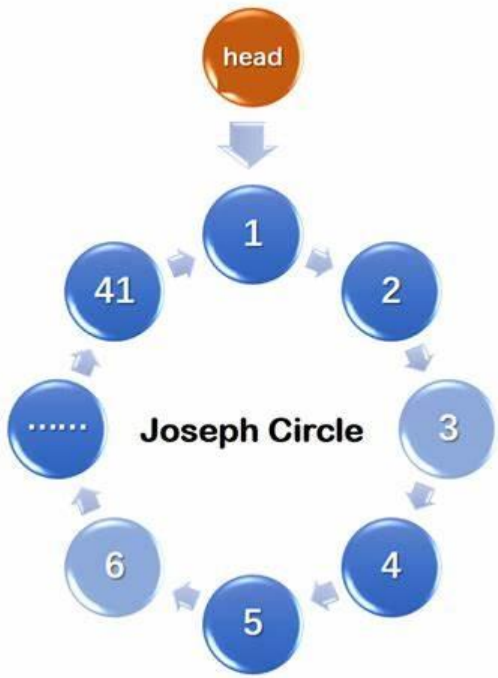
- 顺序表不适用的场合
  - 经常插入删除时，不宜使用顺序表
    - 文档的编辑、任务队列的频繁插入删除，等等
  - 线性表的最大长度也是一个重要因素
- 链表不适用的场合
  - 当读操作比插入删除操作频率大时，不应选择链表
    - 数据库记录的查询，图像的像素存储，等等
  - 当指针的存储开销，和整个结点内容所占空间相比其比例较大时，应该慎重选择

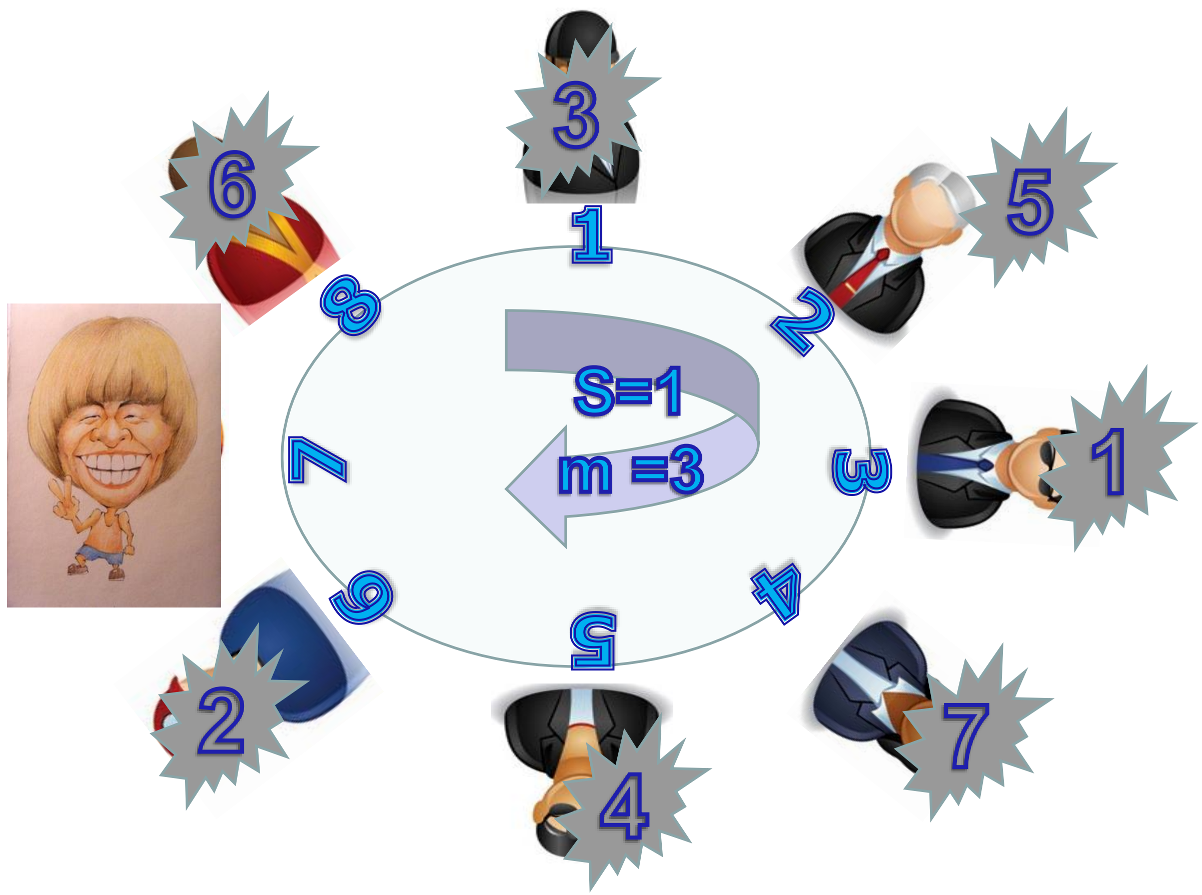
例题讲解

- 约瑟夫问题
- 一元多项式运算

约瑟夫问题

- 现有n个人围成一桌坐下，编号从1到n，从编号为s的人开始报数。
- 报数也从1开始，报到m的人离席，从离席者的下一位在座成员开始，继续从1开始报数。
- 重复该过程，直到仅剩一个成员。求出各成员的离席次序，以及最后一个在座的成员编号。

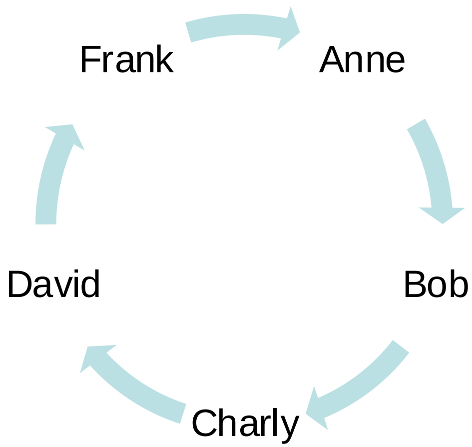




S=1
m =3
1
2
3
4
5
6
7
8
9

约瑟夫问题

- **Josephus问题描述**：对于任意给定的 n, s和m，求按出列次序得到的人员序列，以及最后一个人的序号。
  - n : 参与游戏的人数，每个人的信息
  - s : 开始的人
  - m :单次计数



顺序表方式实现

- 实现Josephus算法的步骤：
  - 1. 建立顺序表
  - 2. 维护“报数”的变量，来模拟问题流程
    - 问题中，所有人围成一个圈
    - 变量应该对全体人数取模，来模拟环结构
  - 3. 迭代删除表中元素，直至只剩余一个人

- 算法的时间复杂度分析：
  - 出列元素的删除（移动实现）为基本运算
  - 每次最多i-1个元素移动，需要n-1次
  - $(n-1)+(n-2)+\ldots\ldots+1 = n(n-1)/2 \quad \Rightarrow \quad O(n^2)$

顺序表方式实现

```
def Josephus_SeqList(n, s, m):
    seqlist = [x + 1 for x in range(n)]
    result = []
    cur_index = s - 1
    while len(seqlist) > 1:
        cur_index = (cur_index + m - 1) % len(seqlist)
        result.append(seqlist[cur_index])
        seqlist = seqlist[:cur_index] + seqlist[cur_index + 1:]
        cur_index = (cur_index) % len(seqlist)

    return seqlist[0], result
```

建立顺序表，维护未出列人员

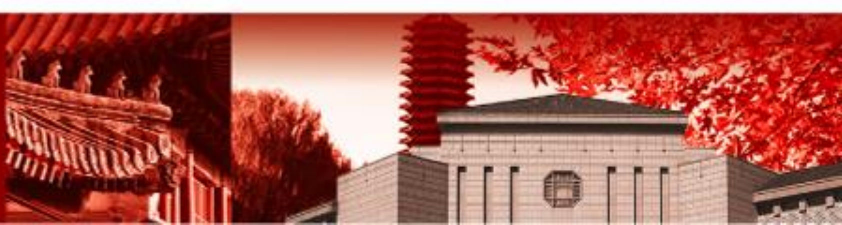
存放依次出列人员序号

维护计数变量

迭代删除顺序表中的元素

循环链表方式实现

- 实现Josephus算法的步骤：
  - 1. 建立循环链表；
  - 2. 出列算法；
    - 利用一个引用来维护当前的报数位置，初始为第s个结点
    - 迭代操作，直至剩余最后一人：
      - 沿着循环链表后移，模拟一次报数过程
      - 删除结点，并将当前报数位置后移
- 时间复杂度分析：
  - 创建链表，求第s个结点，求n个第m个应出列的元素
  - O(n) + O(s) + O(mn) = O(mn)



循环链表方式实现

```python
def josephus_LinkList(n, k):
    head = Node(1)
    prev = head
    for i in range(2, n + 1):
        prev.next = Node(i)
        prev = prev.next
    prev.next = head
    current = head
    result = []
    while current.next != current:
        for _ in range(k - 1):
            prev = current
            current = current.next
        result.append(current.data)
        prev.next = current.next
        current = prev.next
    return current.value
```

创建单循环链表

依次向链表中插入结点

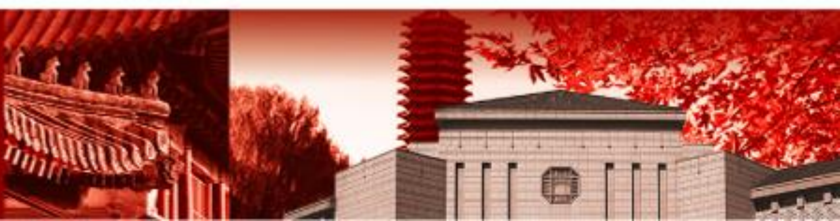
连接尾节点与头结点，形成单循环链表

存储按次序得到的人员序列

当链表中有不止一个节点时

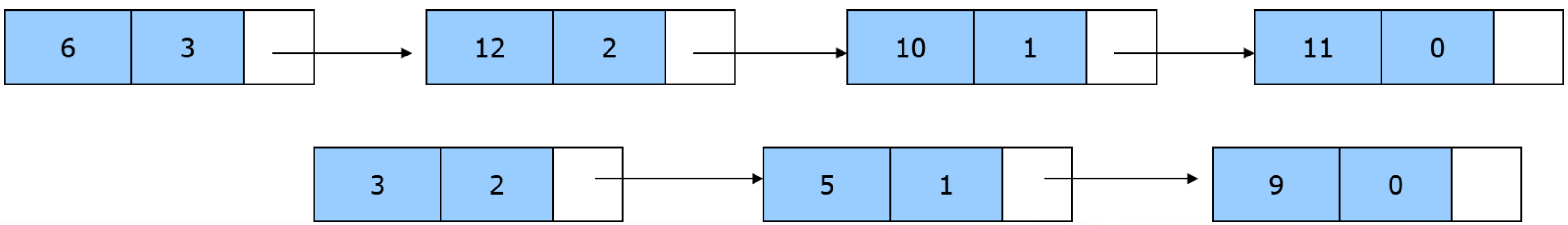
遍历至报数为k的结点

删除该current结点，调整前驱prev的指针域



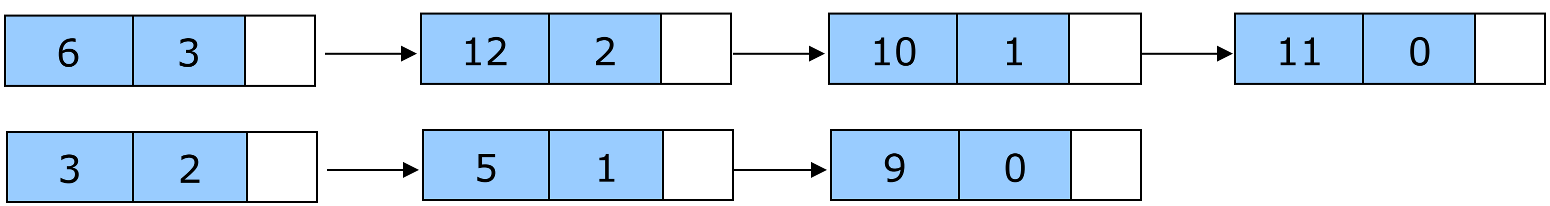
一元多项式的表示

- 数学上的一元多项式：
  – $f(x) = 6x^3+12x^2+10x+11$，$g(x) = 3x^2+5x+9$;
- 运算：$f(x)+g(x)$　$f(x)-g(x)$　$f(x)*g(x)$
- 基于链式线性表，建立一个表示多项式的线性数据结构；
  – 结点表示项，存放系数与幂次；
  – 结点之间的关系表示项之间的降幂关系。
    - 既然是降幂排列，是否还有必要存放幂次？
    - 考虑：$p(x) = 4x^{40000}+2x^{10000}+1$



一元多项式的运算

- 一元多项式加法 / 减法
  - 遍历两个多项式的链表，逐项比较指数。
  - 如果指数相同，将系数相加 / 相减，生成新的结点插入到结果多项式中。
  - 如果指数不同，将指数较大的项直接插入到结果多项式中。
  - 其中一个多项式处理结束后，检查是否有剩余结点，直接插入到结果多项式中。



一元多项式的运算

- 一元多项式乘法
  - 遍历第一个多项式的每个项，将其与第二个多项式的每个项相乘。
  - 每次相乘的结果是一个新的多项式项
    - 如果结果多项式中不存在相同幂次项，将其插入到结果多项式中。
    - 否则，直接将系数加到对应项上