# 数据结构

夏天

xiat@ruc.edu.cn

中国人民大学信息资源管理学院

# 线性表 — Linear List

1. 基本概念
2. 顺序表
3. 链表

# 线性表

- 线性表举例：
  - 英文字母表 $\mathrm{A}, \mathrm{B}, \mathrm{C}, \cdots, \mathrm{Z}$
  - 某单位近 $5$ 年的计算机数量 $(40, 60, 100, 150, 180)$
  - 某产品淘宝的销售记录.
- 特点
  - 数据元素是多样的，但具有相同特性
  - 相邻元素之间有序偶关系 $<$ 前驱, 后继 $>$

# 线性表

**线性表是 n 个数据元素的有限序列**

$$(a_1, a_2, \cdots, a_i, \cdots, a_n)$$

- 存在唯一的第一个数据元素，存在唯一的的最后一个数据元素。
- 除第一个之外，每个数据元素只有一个前驱；除最后一个之外，每个数据元素只有一个后继。
- 线性表的长度：线性表中元素的个数，常记作 length, len, n. 当空表时，len=0.

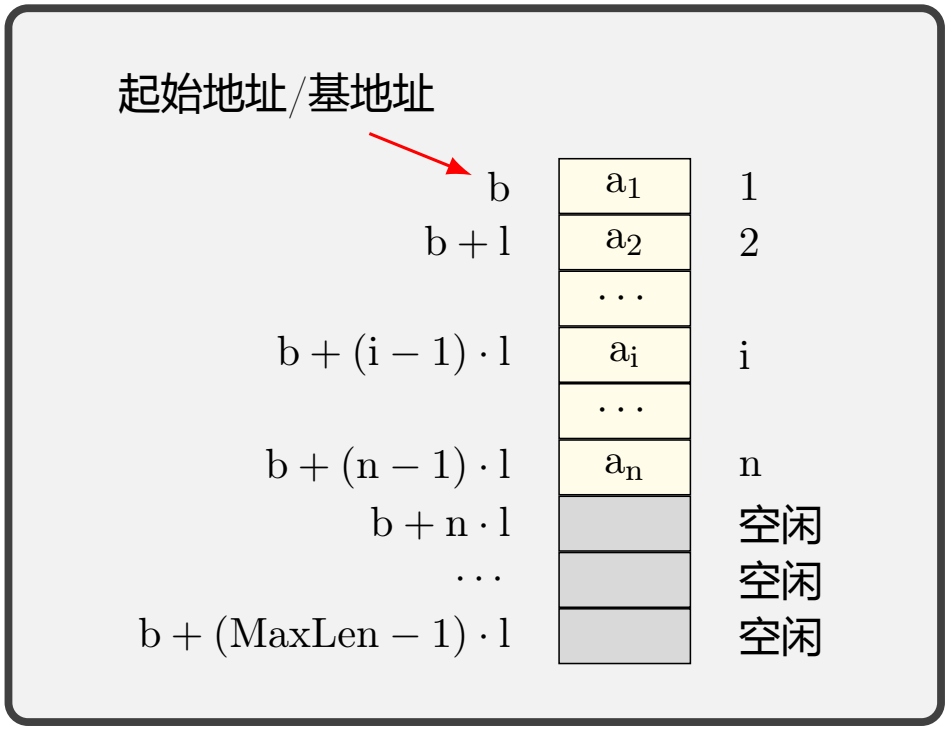
# 线性表的两种存储方式

1. 方案 1：顺序存储 — 称顺序表
2. 方案 2：链式存储 — 称链表

# 方案 1: 顺序表 (Sequence List)

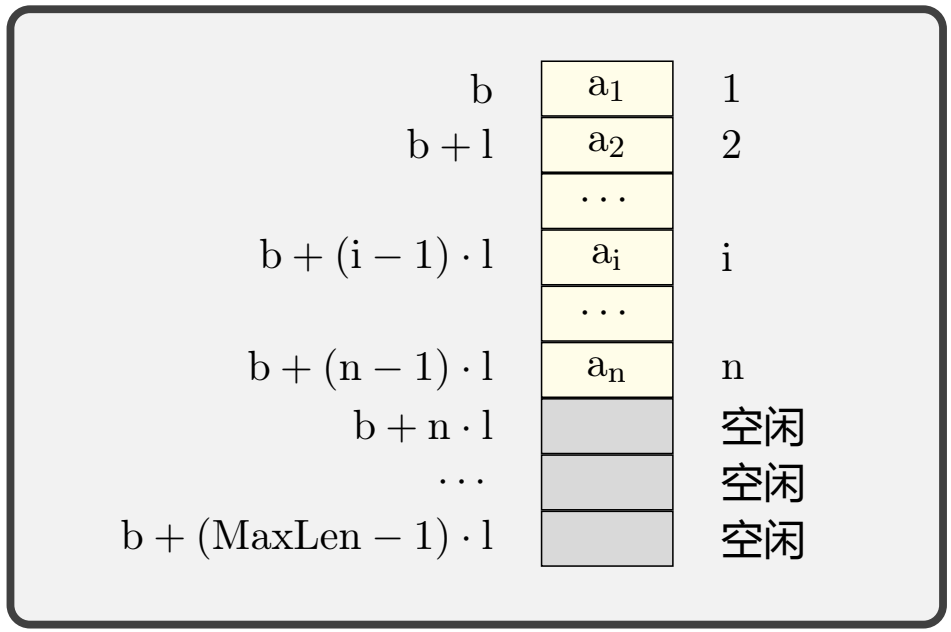
$$(a_1, a_2, \cdots, a_i, \cdots, a_n)$$

- 建一个数组，用一组地址连续的存储单元依次存储数据元素。
  - 逻辑上相邻的数据元素，其物理存储位置也相邻

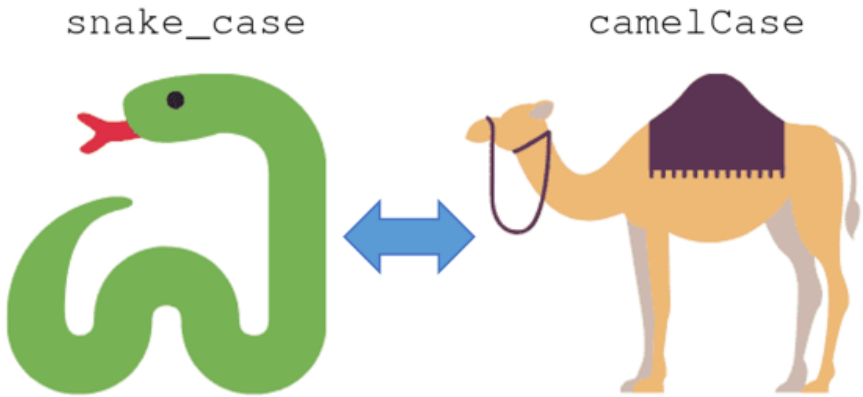


# 顺序表 (Sequence List)

- 为便于维护处理，记录 3 个变量
  - 存放表元素的数组 list；
  - 表的长度 length；
  - 存储容量 maxSize；
- 常用的基本操作
  - 判断是否空: isEmpty()
  - 求长度: length()
  - 取元素: get(i)
  - 插入操作: insert(i,x)
  - 删除操作: remove(i)
  - 查找: indexOf(x)
  - 输出: display()



# 补充：常用命名规则之驼峰与蛇形命名法




**Camel case**

```
{
    "firstName": "John",
    "lastName": "Smith",
    "email": "john.smith@example.com",
    "createdAt": "2021-02-20T07:20:01",
    "updatedAt": "2021-02-20T07:20:01",
    "deletedAt": null
}
```

**Snake case**

```
{
    "first_name": "John",
    "last_name": "Smith",
    "email": "john.smith@example.com",
    "created_at": "2021-02-20T07:20:01",
    "updated_at": "2021-02-20T07:20:01",
    "deleted_at": null
}
```

```
public class SequenceList {
    int maxSize; //最大长度
    int length; //当前长度
    Elem[] list; //对象数组

    public bool isEmpty()
    public int length()
    public Elem get(i) throws Exception
    public void insert(i,e)
    public void remove(i) throws Exception
    public int indexOf(e)
    public void display()
    public void clear()
}
```

```python
class SequenceList:
    def __init__(self, max_size, elements):
        self.max_size = max_size
        self.length = len(elements)
        self.elements = [0]*max_size
        for idx, value in enumerate(elements):
            self.elements[idx] = value

    def __len__(self):
        return self.length

    def __getitem__(self, i):
        return self.elements[i]

    def __setitem__(self, i, value):
```

# 初始化顺序表

Java/C Example

```
//初始化空表
void initList(int size) {
    list = new Elem[size];
    maxSize = size; //初始存储容量
    length = 0; //空表长度为 0
}
```

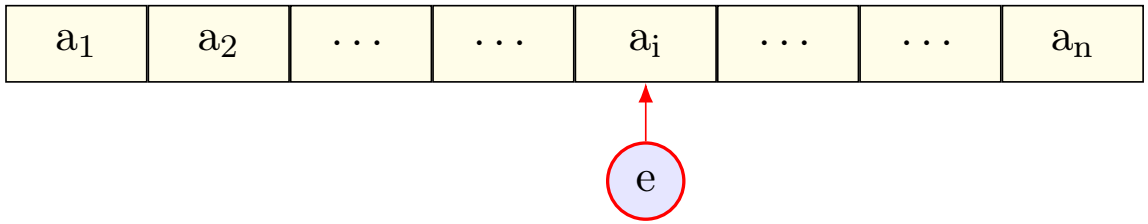
Python Example:

```
    self.elements = [0]*max_size
```

# 在 index 位置上插入元素 e

```
boolean insert(int index, Elem e) {
 if (length== maxSize)   //当前线性表已满
   {print(" 顺序表已满!"); return false;}
 if (index < 0 || index > length)   //插入位置编号不合法
  {print(" 参数错误!"); return false;}

 for (int j = length- 1; j >= index; j--)  //向后移动元素
   list[j + 1] = list[j];

 list[index] = e;   //插入元素
 length ++;
 return true;
}
```

# 插入元素的时间复杂度




- 在长为 $n$ 的线性表中插入一个元素，所需移动元素次数的平均次数为?
  - 有多少种可能的插入位置？$n+1$
  - 假设这些位置以同等概率出现，每个概率 $\frac{1}{n+1}$。每个情形下分别移动 $n, n-1, \cdots, 0$ 次。求加权和为 $n/2$。
  - 平均时间复杂度：$T(n) = O(n)$

# 删除 index 位置上的元素


```
boolean remove(int index) {
   if(index<0||index>=length)
       return false;
   if(getLength()==0)
       return false;
   for (int j=index;j<length-1; j++)   //前移
       Elem[j]=elem[j+1];
   length--;
}
```

# 删除元素的时间复杂度


- 在长度为 $n$ 的线性表中删除一个元素：
  - 共有 $n$ 个可能的位置，每个有 $1/n$ 的概率。每个情形下分别移动 $n-1, ..., 0$ 次。求加权和为 $(n-1)/2$。
  - $T(n) = O(n)$

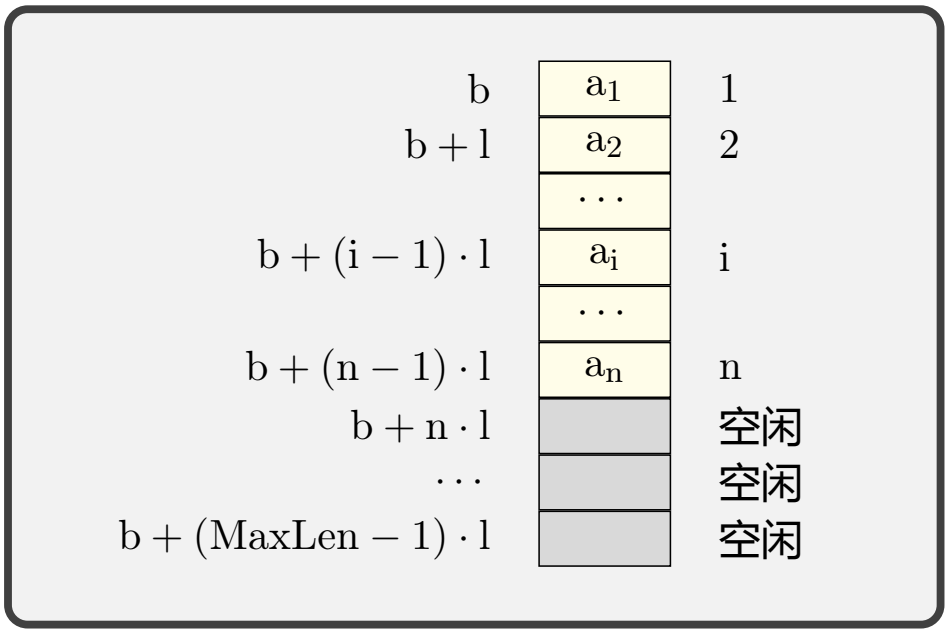
# 删除元素的时间复杂度


- 在长度为 n 的线性表中删除一个元素：
  - 共有 n 个可能的位置，每个有 $1/n$ 的概率。每个情形下分别移动 n-1, ..., 0 次。求加权和为 $(n\text{-}1)/2$。
  - $T(n) = O(n)$

**结论**

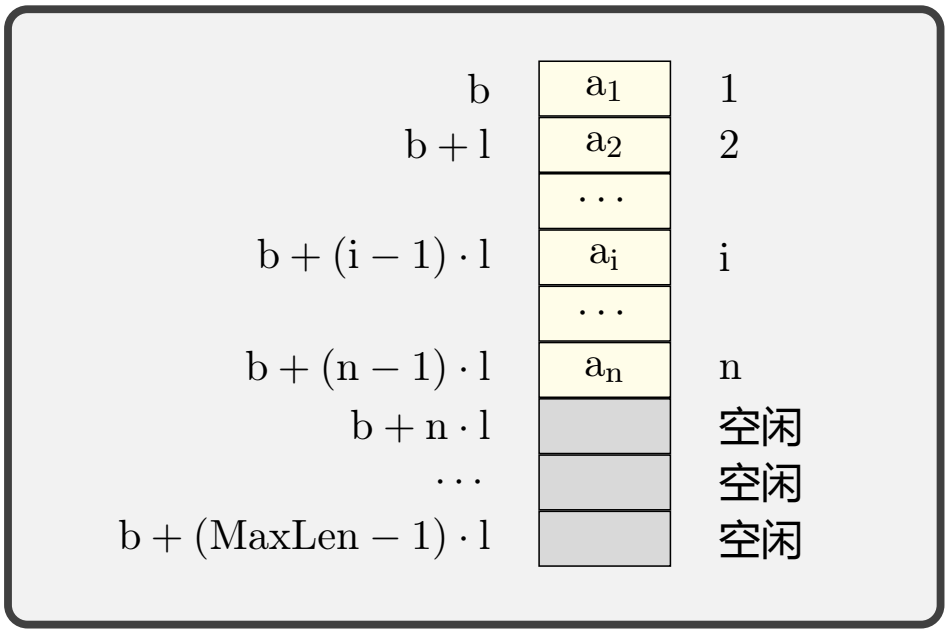
在顺序表中插入或删除一个元素时，平均移动一半元素，当 n 很大时，效率很低.

# 顺序表特点：地址连续



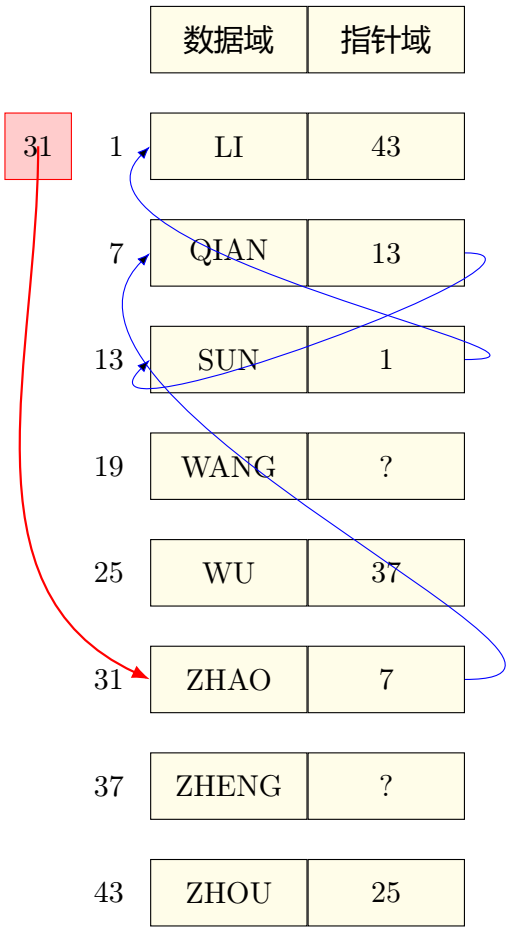

- 优点
  - 直观
  - 随机存储效率高

# 顺序表特点：地址连续



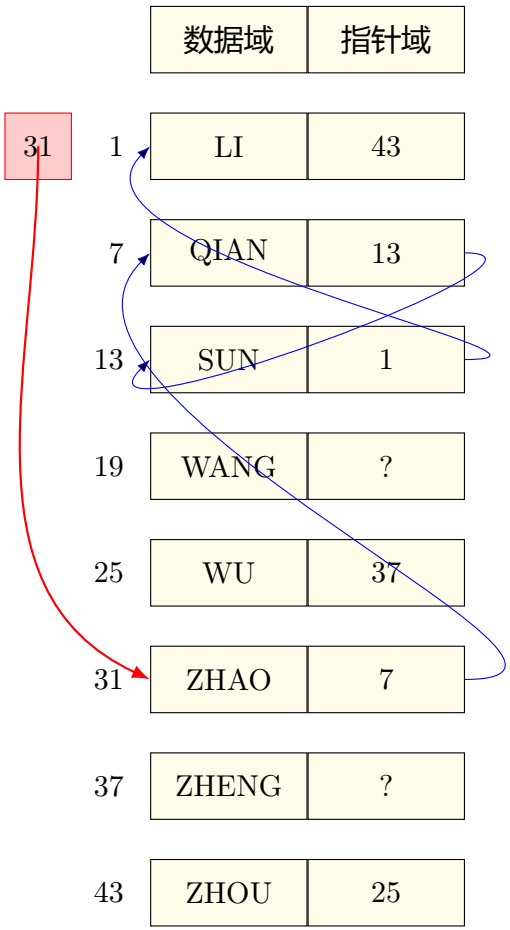

- 优点
  - 直观
  - 随机存储效率高
- 缺点
  - 移动元素代价大

# 方案 2: 链式存储---链表




用一组任意的存储单元存储线性表的数据元素，利用指针指向直接后继的存储位置

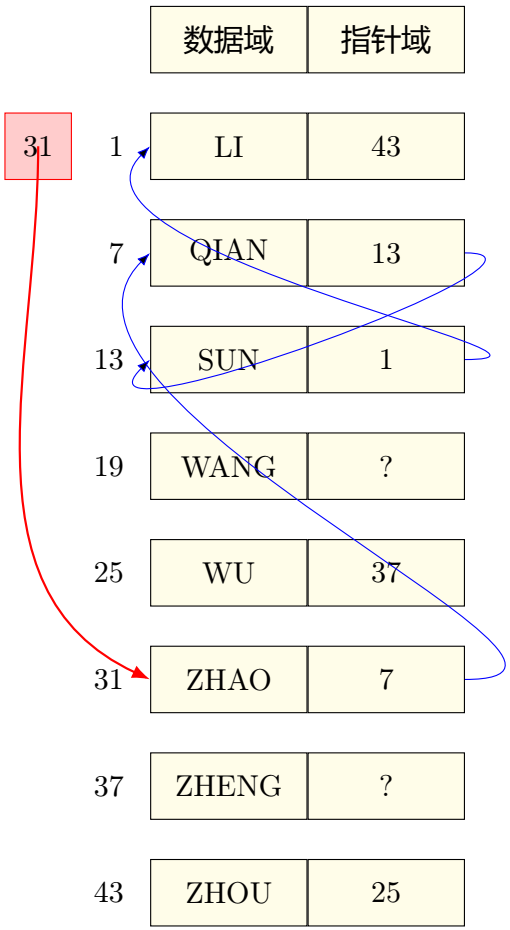
# 单链表的类型定义




- 单链表：每个结点有一个指针域
- 单链表可由头指针唯一确定，头指针指向第一个结点。
- Java Code:

```
class Node {
  Object data;
  Node next;
}
```

# 单链表的类型定义




- 单链表：每个结点有一个指针域
- 单链表可由头指针唯一确定，头指针指向第一个结点。
- Python Code:

```
class Node:
  def __init__(self, data, next=None):
    self.data = data
    self.next = next

first = Node(1)
second = Node(2)
last = Node(3)
first.next=second
second.next = last
print(first.next.next.data)
```

# 单链表上的常见操作

1. 建立单链表 create()
2. 求表长 length()
3. 查找 index(value)
4. 插入 insert(i,e)
5. 删除 remove(i)
6. 获取元素 get(index)
7. 显示 display()

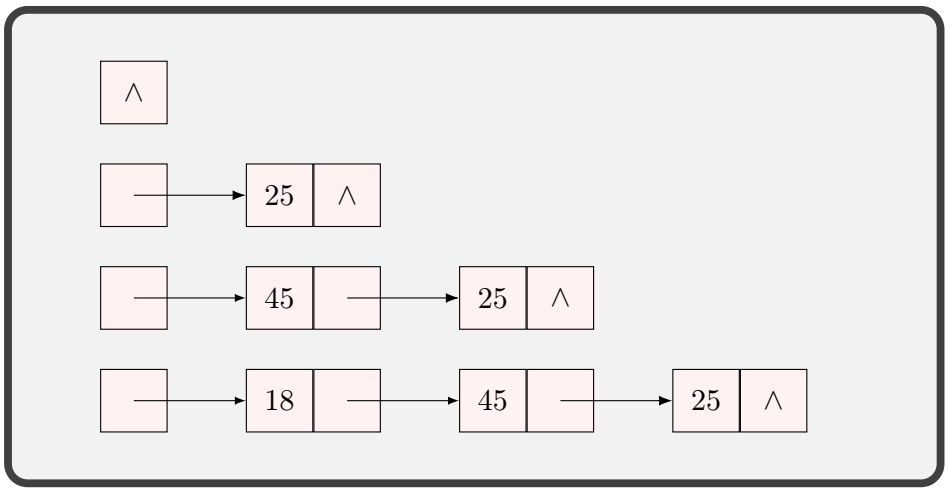
# 建立单链表

- 链表是动态管理的：链表中的每个结点占用的存储空间不是预先分配，而是运行时系统根据需求而生成的。
- 建立单链表从空表开始，每读入一个数据元素则申请一个结点，插入链表。

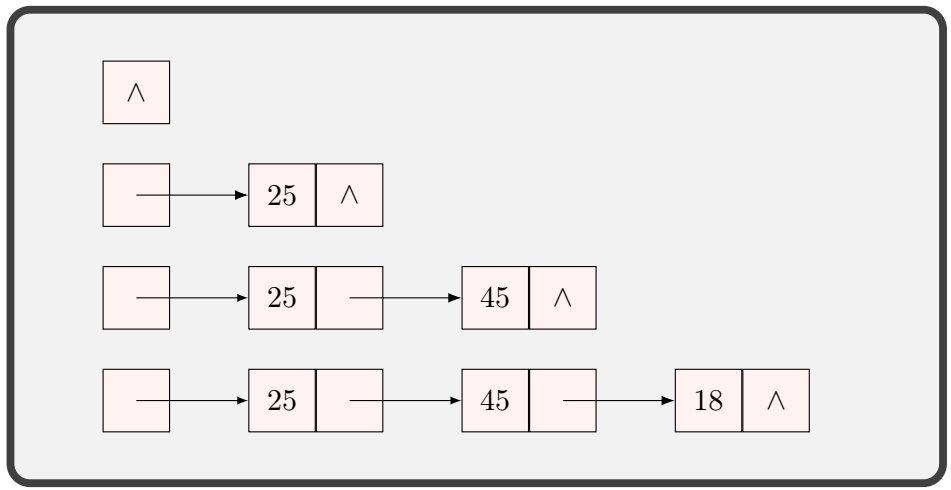
# 建立单链表：两种不同方式

如依次读入 25, 45, 18...

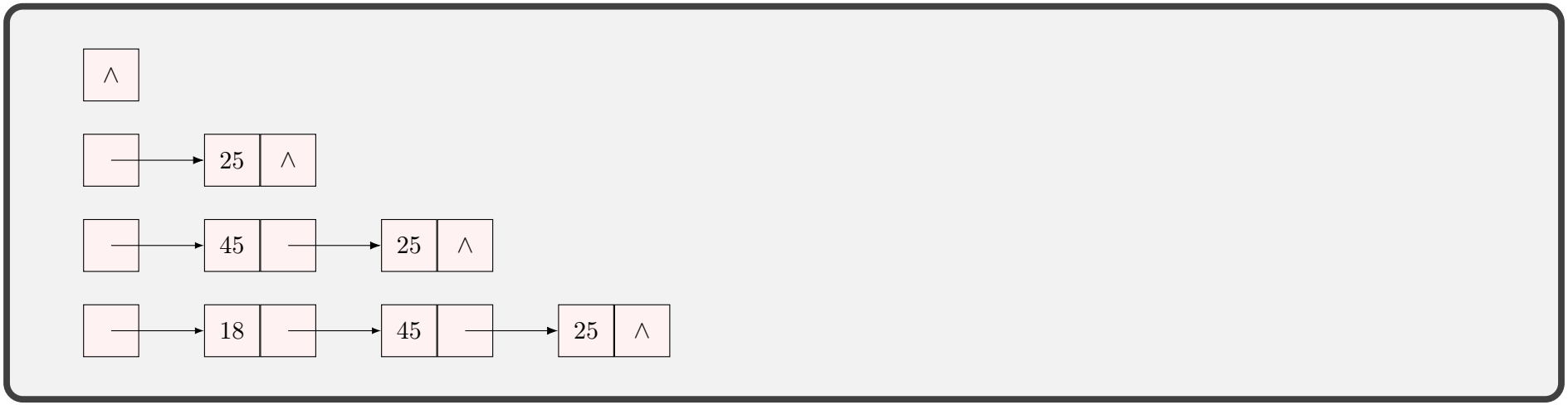
头部插入：




尾部插入：

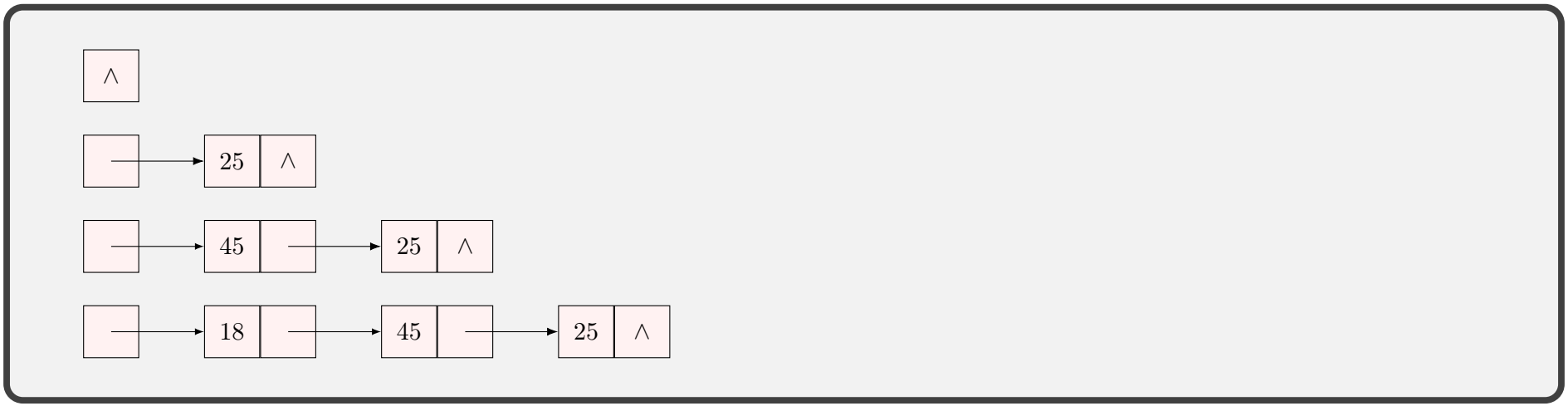


# 建立单链表：头部插入




```
Node s = new Node(val);//s 指向新结点
s.next = head;//新结点后继为当前头结点
head = s;//头指针指向新结点
```

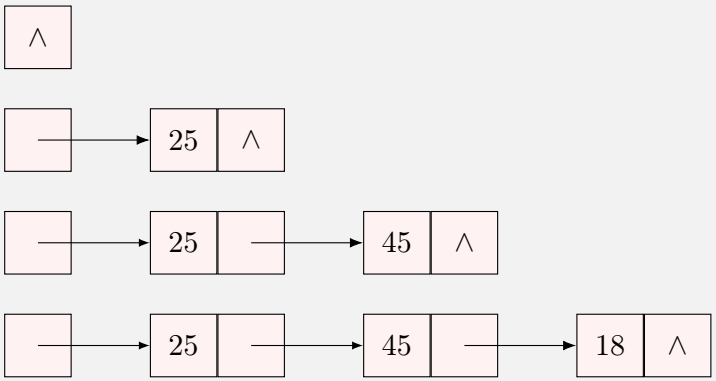
# 建立单链表：头部插入




```
Node s = new Node(val);//s 指向新结点
s.next = head;//新结点后继为当前头结点
head = s;//头指针指向新结点
```

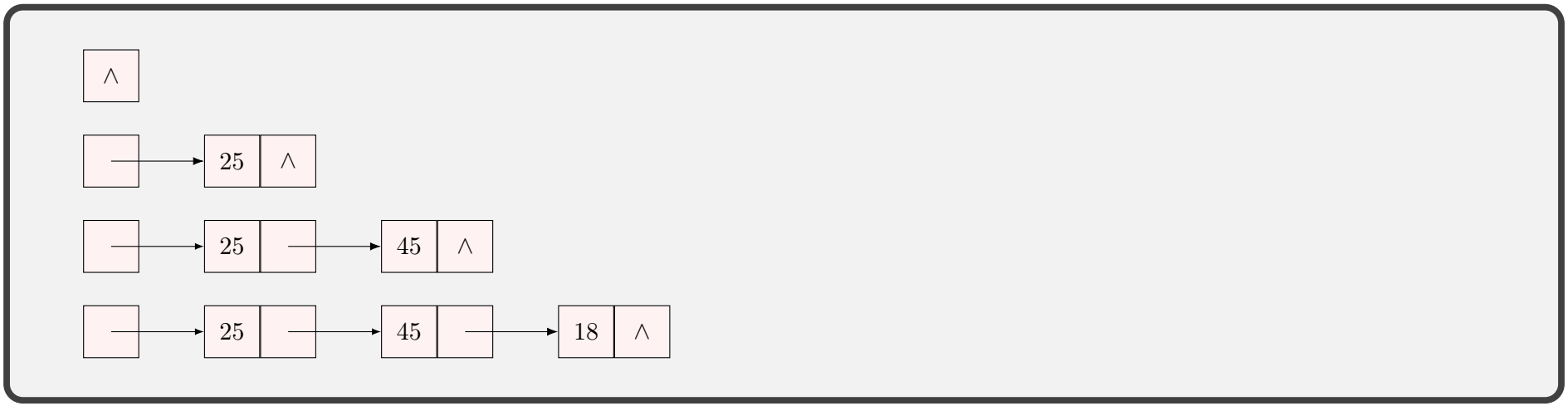
在空表时是否可行？

# 建立单链表：尾部插入




```
Node s=new Node(val);//s 指向新结点
Node r=Findlast(Head);//找到尾结点
r.next=s;//新结点成为尾结点的后继
r=s;
```

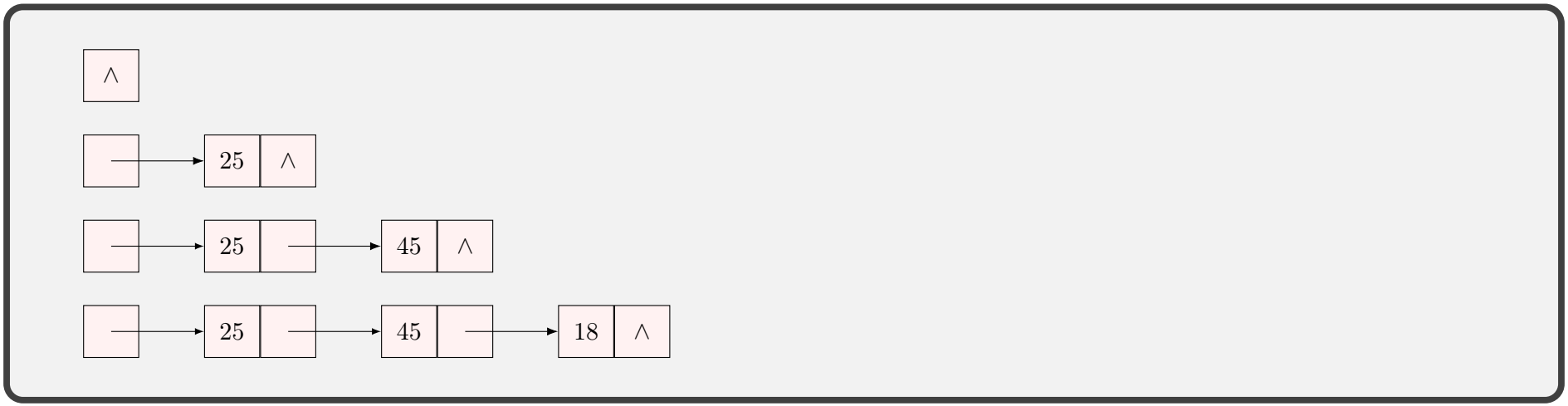
# 建立单链表：尾部插入




```
Node s=new Node(val);//s 指向新结点
Node r=Findlast(Head);//找到尾结点
r.next=s;//新结点成为尾结点的后继
r=s;
```

在空表时是否可行？

# 建立单链表：尾部插入



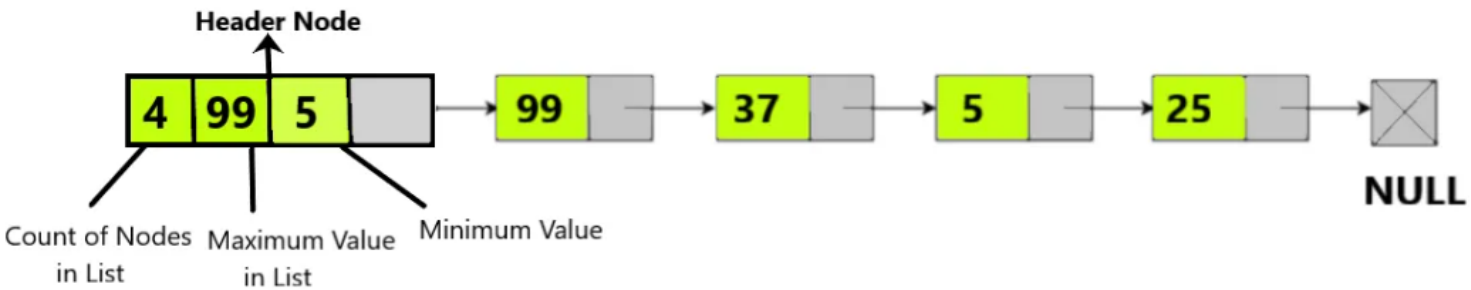

```
Node s=new Node(val);
if(!head) //空表，插入第一个结点
   head=s; //新结点作为第一个结点
else //非空表
   r.next=s;//新结点作为最后一个结点的后继

r=s; //尾指针 r 指向新的尾结点
```

# 头结点问题

- 在上面的算法中，空表和非空表的处理是不同的
  - 当链表为空，新结点作为第一个结点，地址放在链表头指针变量中（第一个结点没有前驱，其地址就是整个链表的地址）；
  - 否则，新结点地址放在其前驱的指针域。
- 上述问题在很多操作中都会遇到，为方便操作，可在链表头部加一个“头结点”。

# 头结点问题

- 头结点的类型与数据结点一致，其数据域无定义，指针域中存放的是第一个数据结点的地址，空表时为空。头结点的数据域可以为空，也可存放线性表长度等附加信息，但此结点不能计入链表长度值。
- 加入头结点完全是为了运算的方便。有了头结点，即使是空表，头指针变量 head 也不为空，空表和“非空表”的处理成为一致。

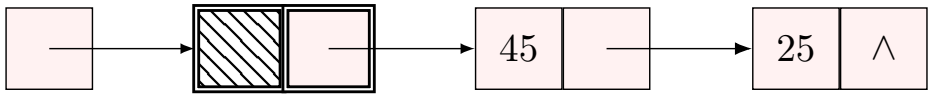


# 约瑟夫环

在罗马人占领乔塔帕特后，约瑟夫及他的 40 个战友躲到一个洞中，这些犹太人宁死也不想被敌人抓到，于是决定了一个自杀方式：

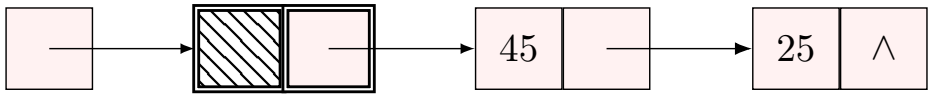
- 41 个人排成一个圆圈，由第 1 个人开始报数，每报数到第 3 人该人就必须自杀，然后再由下一个重新报数，直到所有人都自杀身亡为止。
- 约瑟夫说他和另一个人逃过了这场死亡游戏: by luck or by the hand of God.
- 请问约瑟夫在这个圆圈中的位置是？

# 求表长




```
int GetLength() {
 Node p= head.next;
 int  len=0;
 while (p){
  p=p.next;
  len++;
 }
 return  len;
}
```

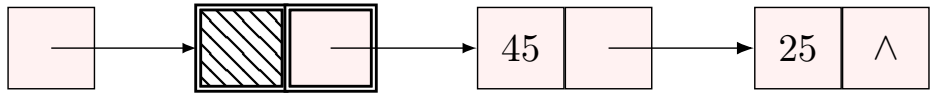
# 求表长




```
int GetLength() {
 Node p= head.next;
 int  len=0;
 while (p){
  p=p.next;
  len++;
 }
 return  len;
}
```

```
class Node:
    def __init__(self, data, next=None):
        self.data = data
        self.next = next

def get_length(hp):
    p = hp.next
    len = 0
    while(p!=None):
        p = p.next
        len = len + 1
    return len

hp = Node(None, Node(45, Node(25)))
print(get_length(hp))
```

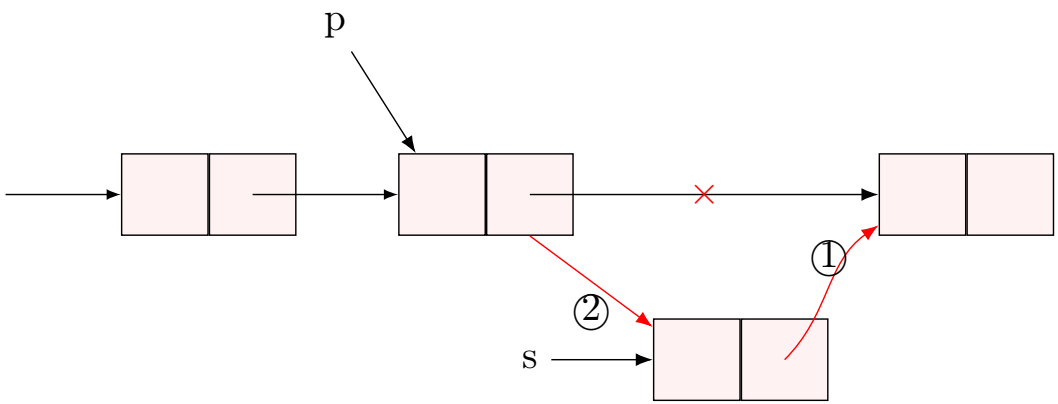
# 单链表的查找




- 按值查找 index(value): 是否存在数据元素 X？序号是？
- 算法思路："顺藤摸瓜" ——从第一个结点开始，判断当前结点的值是否等于 x，若是则返回该结点的指针，否则继续检查下一个，直到表尾。如果找不到则返回空。

```
public int index(value){  //在 L 中查找值为 x 的结点
 Node  p=Head.next;  int j=0;
 while ( p && p.data !=value){
  p=p->next; j++;
 }
 if(p) return j; else return -1;
}
```

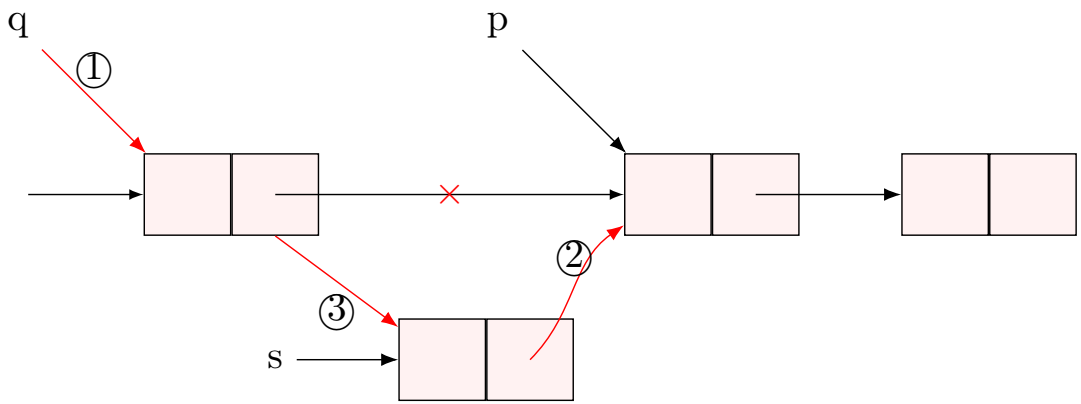
# 插入新结点：在给定结点的前后




- 新结点插入到 p 的后面

```
s.next = p.next;
p.next = s;
```

# 插入新结点：在给定结点的前后




- 新结点插入到 p 的前面

  找到 p 的前驱 q，在 q 之后插入 s。

```
q=head;
while (q.next!=p) q=q.next;
s.next=q.next;
q.next=s;
```

# 插入新结点

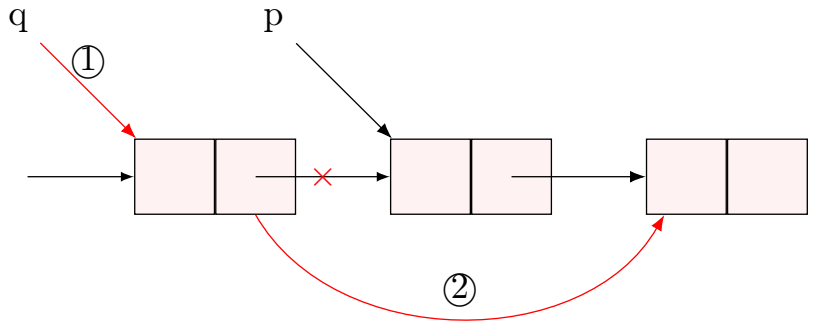
- 时间复杂度
  - 后插操作为 $O(1)$: 不受 n 的影响
  - 前插操作因为要先找到 p 的前驱，时间性能为 $O(n)$。

# 插入新结点

- 时间复杂度
  - 后插操作为 O(1): 不受 n 的影响
  - 前插操作因为要先找到 p 的前驱，时间性能为 O(n)。
- 一个小技巧：
  - 将 s 插入到 p 的后面，然后把 p.data 与 s.data 交换，这样能使得时间复杂性为 O(1)。

# 删除结点

- 删除 p 指向的结点




q.next=q.next.next;

- 首先要找到 p 的前驱结点 q，其时间复杂性为 O(n)。
- 若要删除 p 的后继结点 (假设存在)，则可以直接完成：

p.next=p.next.next;

- 该操作的时间复杂性为 O(1) 。

# 删除第 i 个结点

```
int Del(LinkList L, int i) //删除链表 L 第 i 个结点
{
 p=get(L, i-1); //查找第 i-1 个结点
 if (p==NULL) {
  printf( “第i-1个结点不存在” ); return -1;
 } else{
  if (!p->next)
   return 0; //第 i 个结点不存在
  else {
   p.next=p.next.next; //从链表中删除 i
   return 1;
  }
 }
```

# 单链表操作小结

- 在单链表上当前结点之前插入、删除一个结点，必须知道其前驱结点。
- 单链表不具有按序号随机访问的特点，只能从头指针开始一个个顺序进行。

# 约瑟夫环

TODO