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.

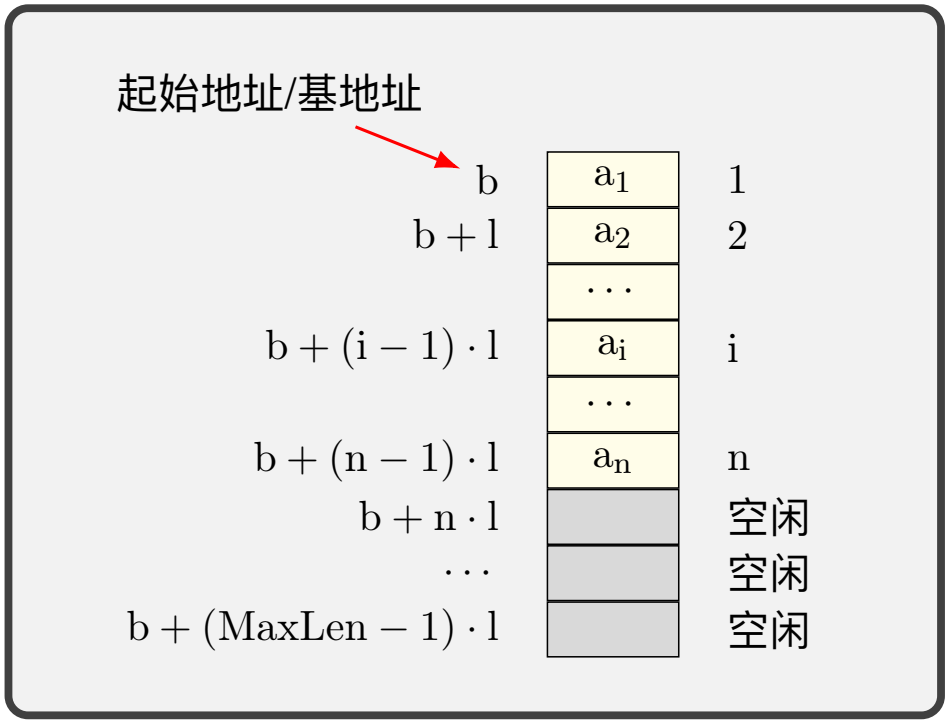
线性表的两种存储方式

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)

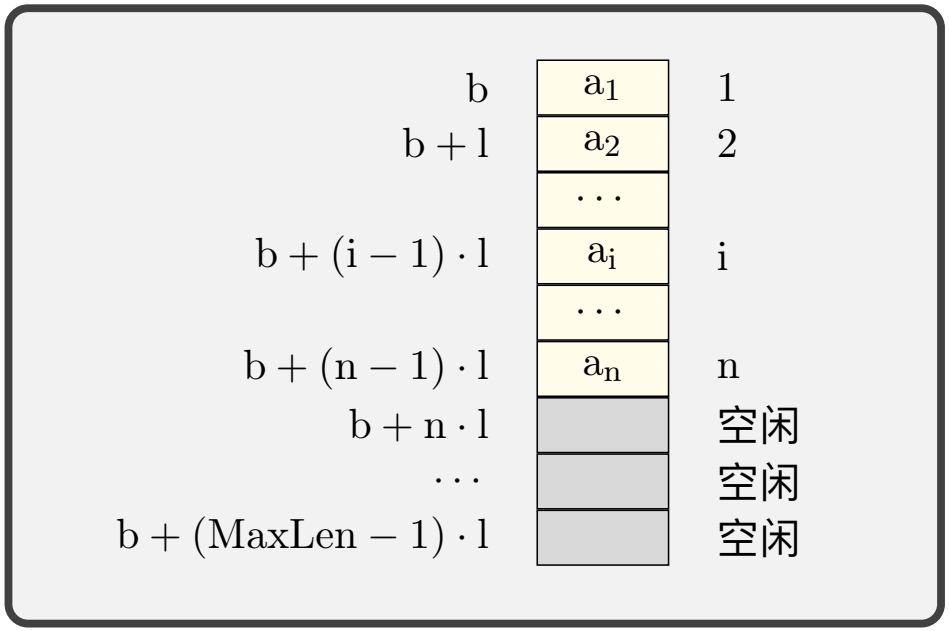
$$(a_1, a_2, \cdots, a_i, \cdots, a_n)$$

- 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

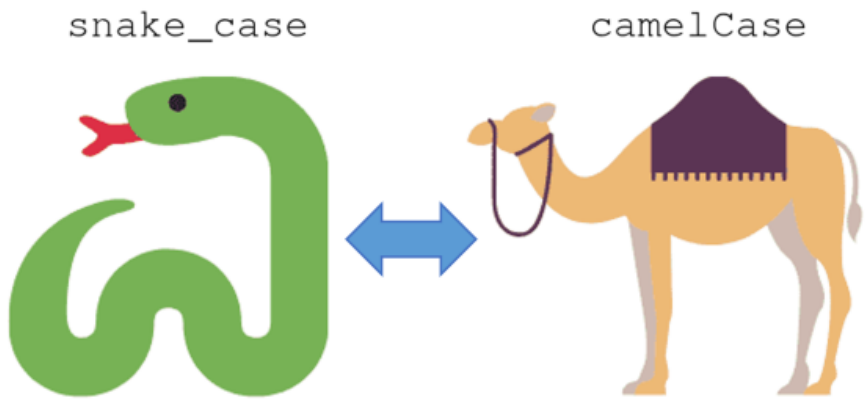


顺序表 (Sequence List)

- 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()
}
```

```
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
}
```

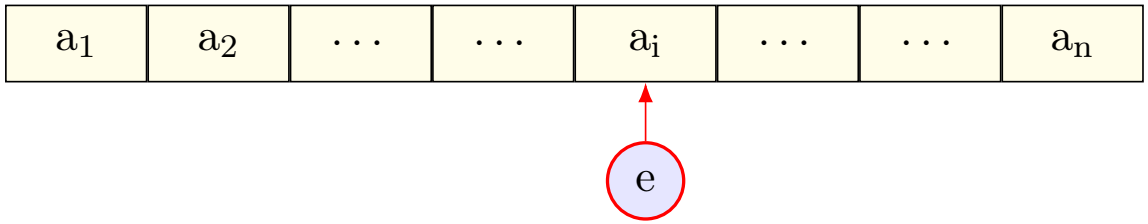
Python Example:

```
    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
     {print(" 参数错误!");  return false;}

   for (int j = length- 1; j >= index; j--)     //向后移动
      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。
  - $\mathrm{T(n)} = \mathrm{O(n)}$

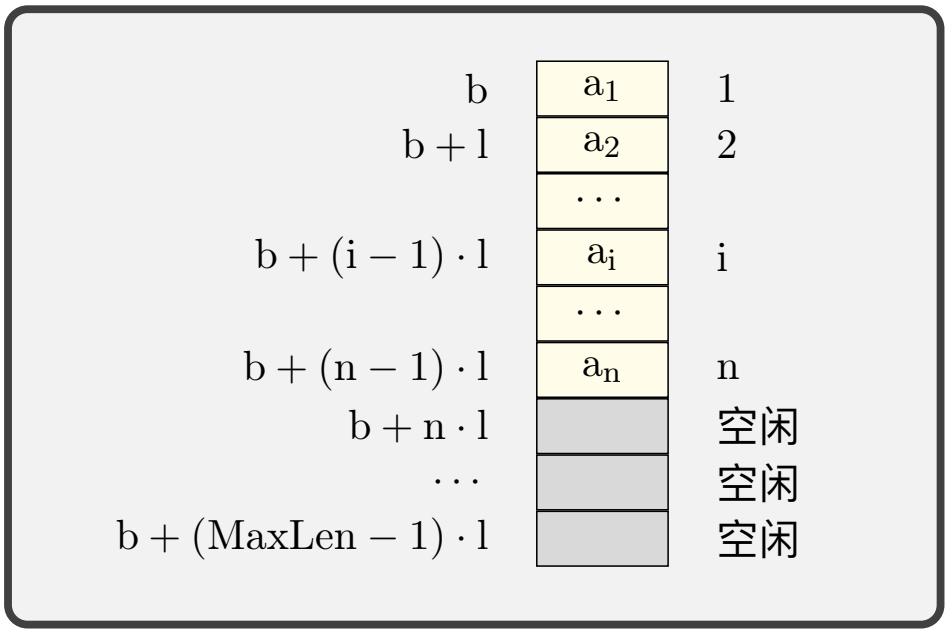
删除元素的时间复杂度

- 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-1)/2。
  - $\mathrm{T(n)} = \mathrm{O(n)}$

结论

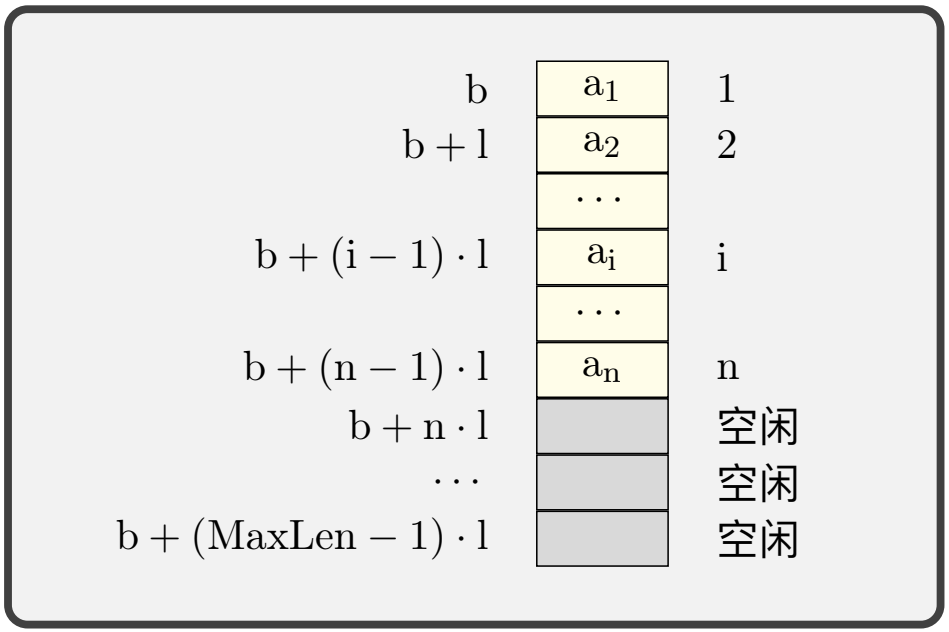
在顺序表中插入或删除一个元素时，平均移动一半元素，当 n 很大时，效率很低.

顺序表特点：地址连续



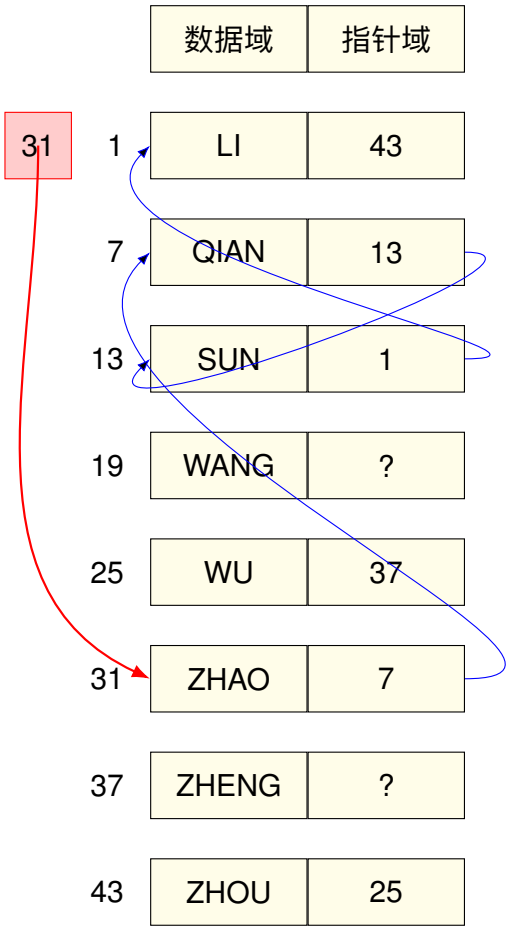
- 优点
  - 直观
  - 随机存储效率高

顺序表特点：地址连续



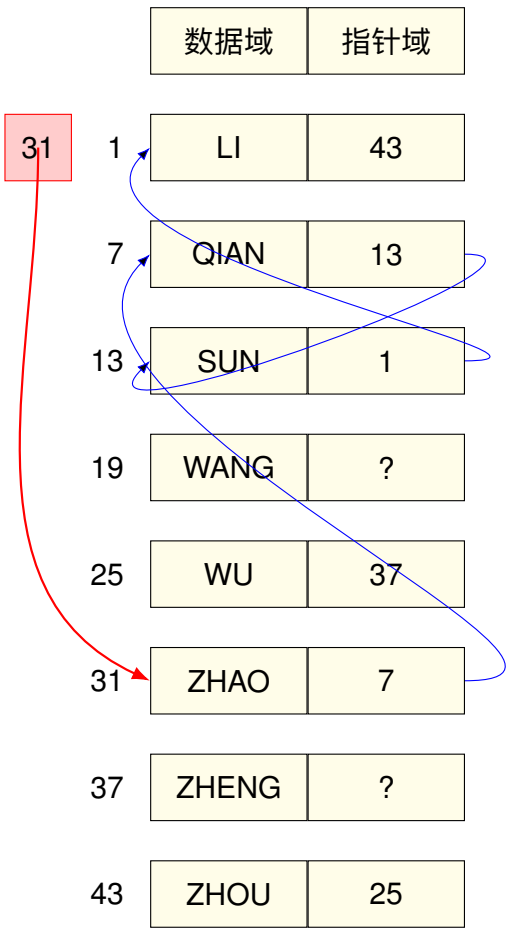
- 优点
  - 直观
  - 随机存储效率高
- 缺点
  - 移动元素代价大

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

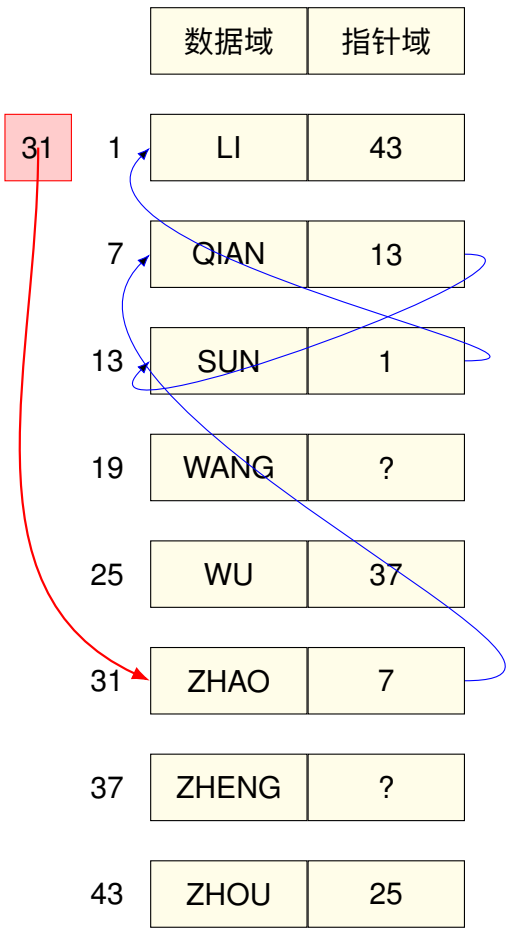
单链表的类型定义



- 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=No
    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()

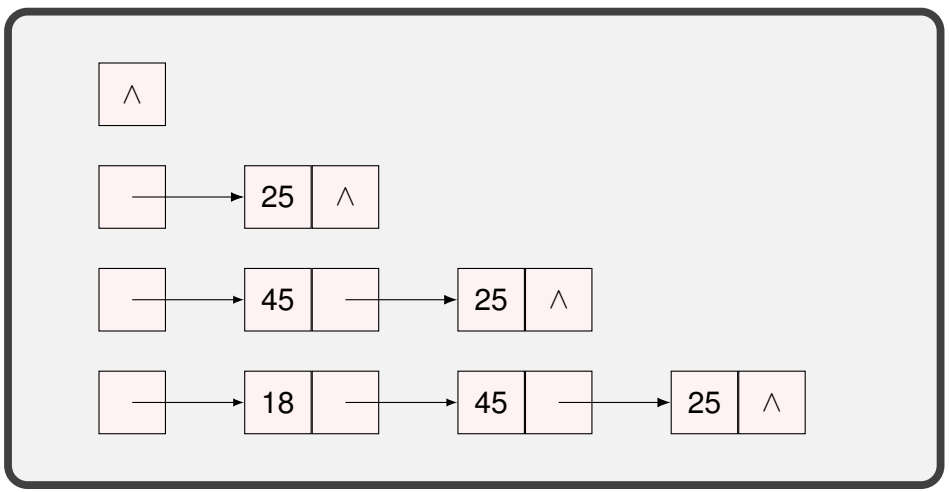
建立单链表

- 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。

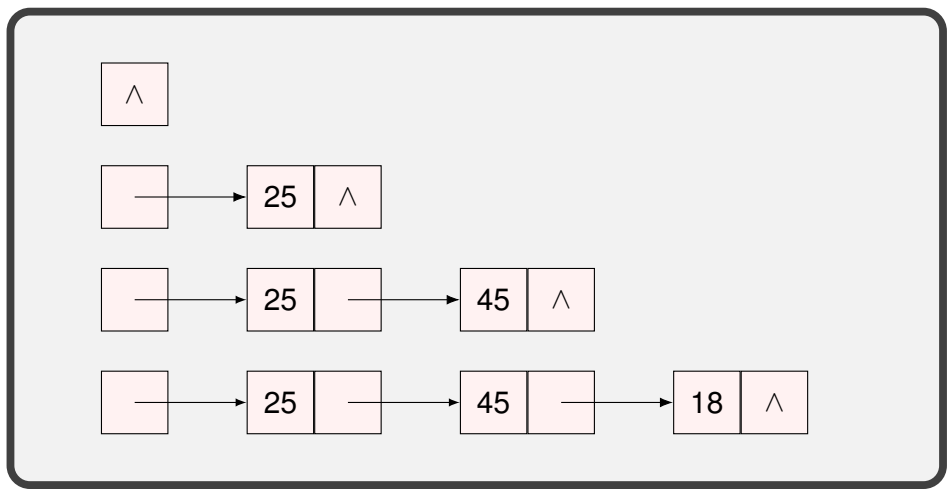
建立单链表：两种不同方式

如依次读入 25, 45, 18...

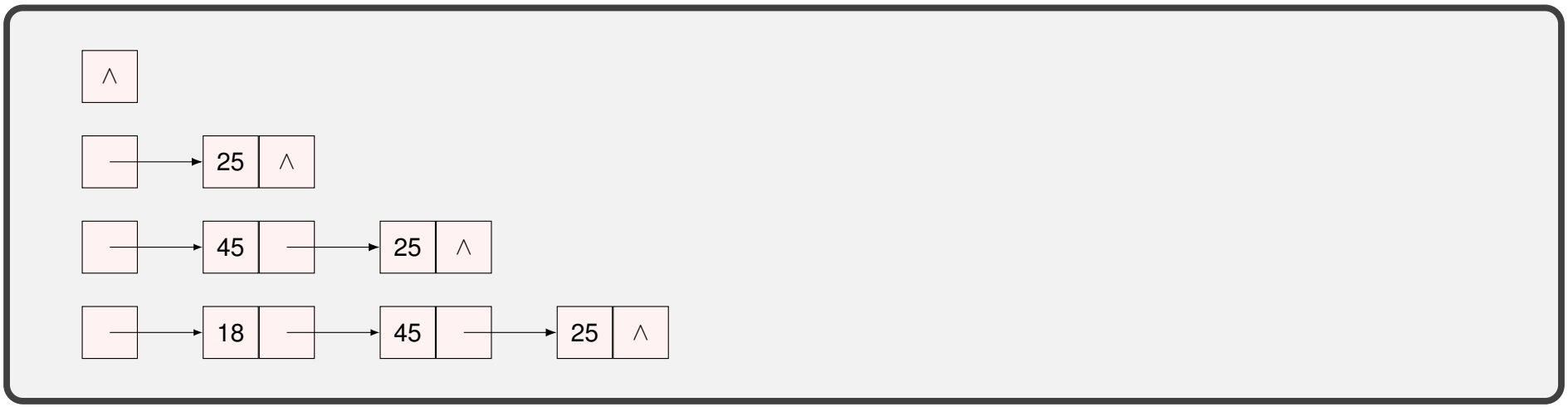
头部插入：



尾部插入：

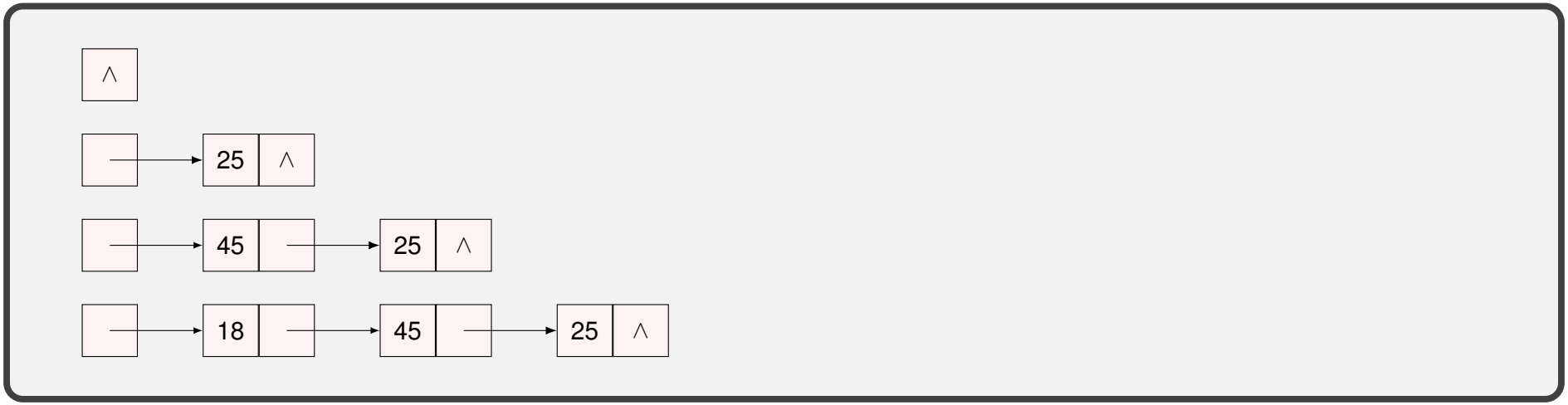


建立单链表：头部插入




```
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
```

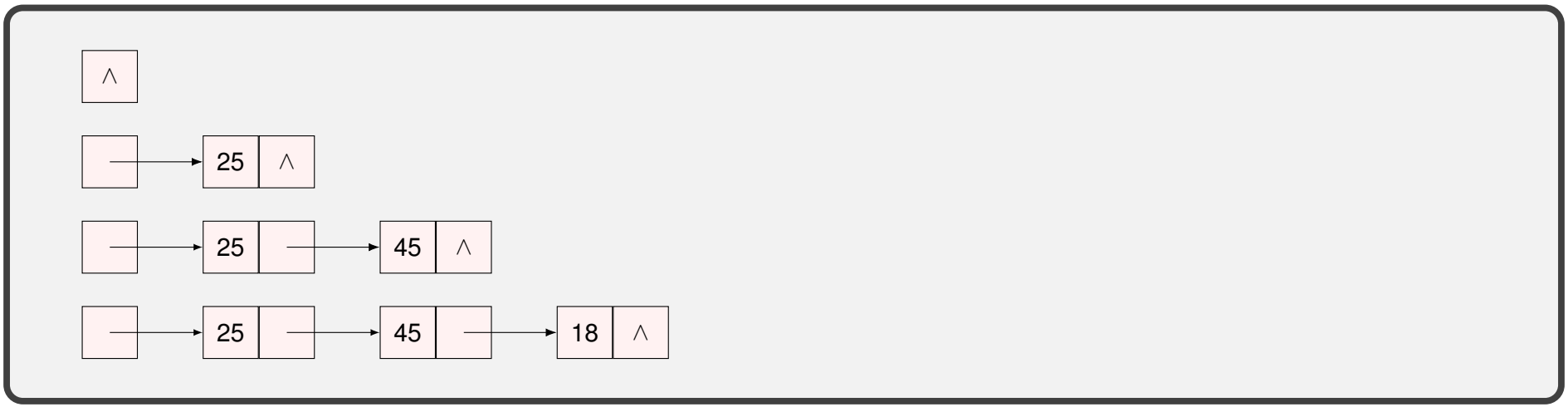
建立单链表：头部插入




```
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
```

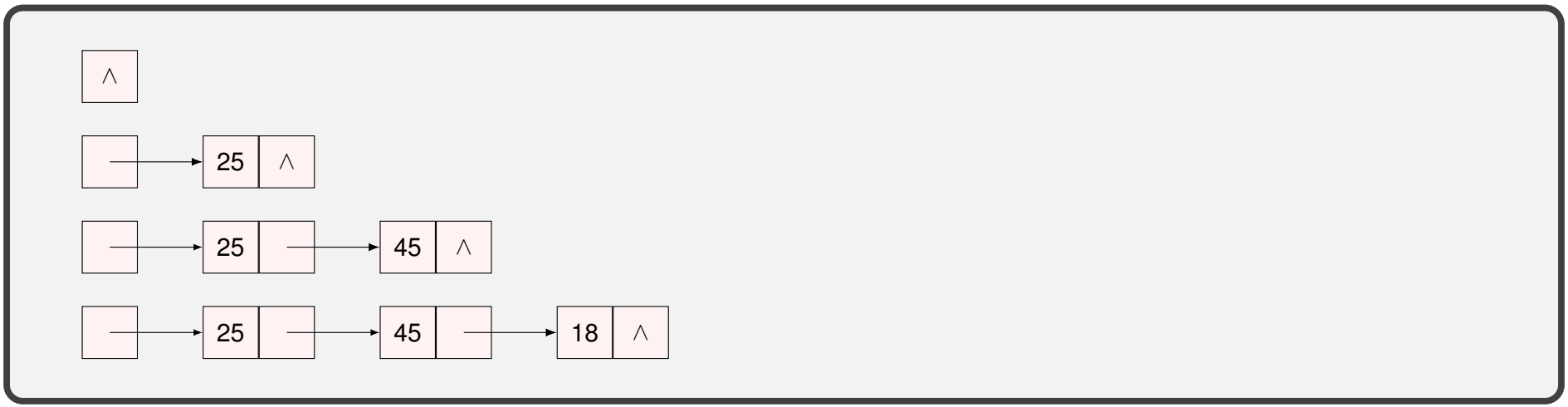
在空表时是否可行?

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;
```

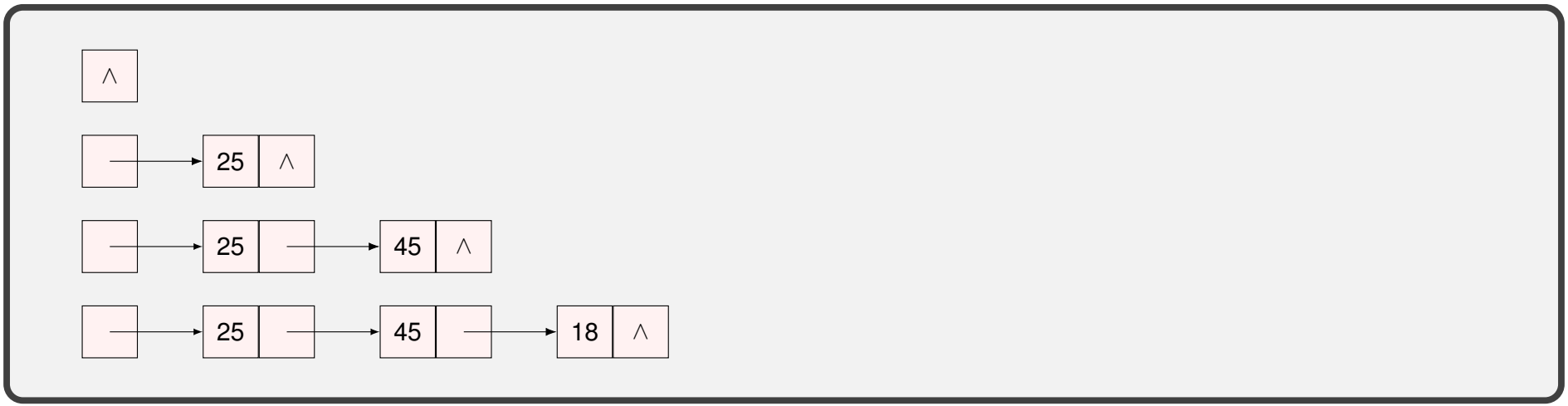
建立单链表：尾部插入




```
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



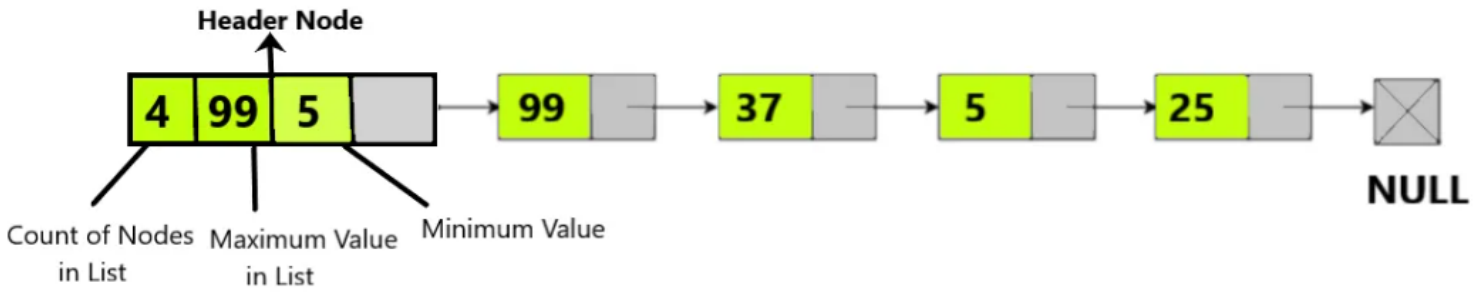

```
Node s=new Node(val);
if(!head)   // 空表，插入第一个结点
    head=s; //新结点作为第一个结点
else // 非空表
    r.next=s;//新结点作为最后一个结点的后继

r=s; // 尾指针 r 指向新的尾结点
```

头结点问题

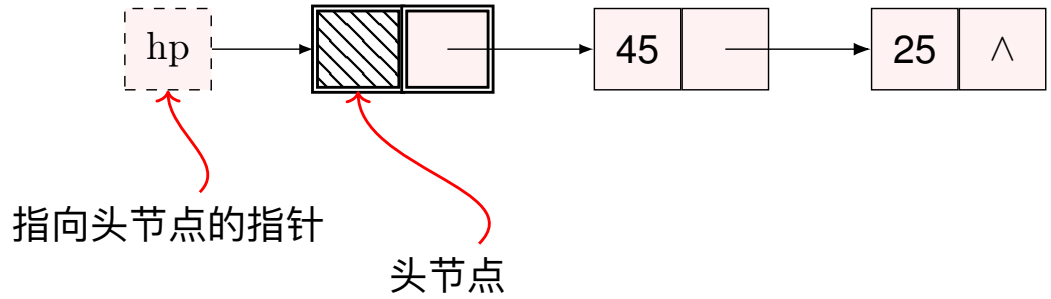
- 在上面的算法中，空表和非空表的处理是不同的
  - 当链表为空，新结点作为第一个结点，地址放在链表头指针变量中（第一个结点没有前驱，其地址就是整个链表的地址）；
  - 否则，新结点地址放在其前驱的指针域。
- 上述问题在很多操作中都会遇到，为方便操作，可在链表头部加一个“头结点”。

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（或记作 h，或者 hp: head pointer）也不为空，空表和“非空表”的处理成为一致。
- 注意区分：头节点和指向头节点的指针

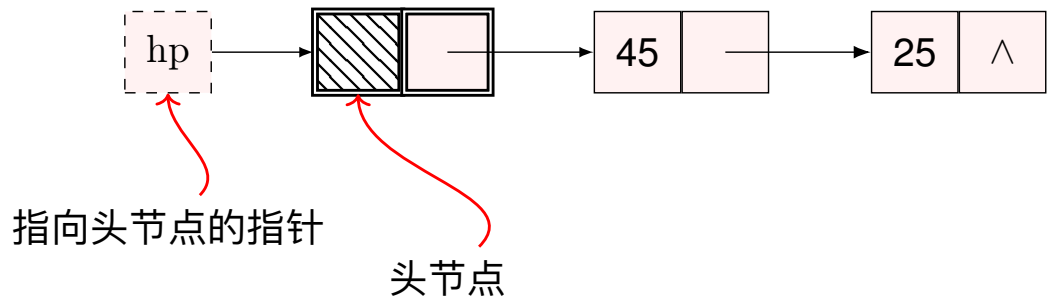


求表长




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

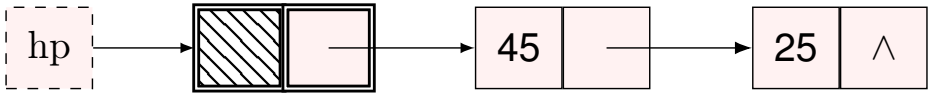
求表长




```
int GetLength() {
  Node p= hp.next; // hp: head pointer
  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
```

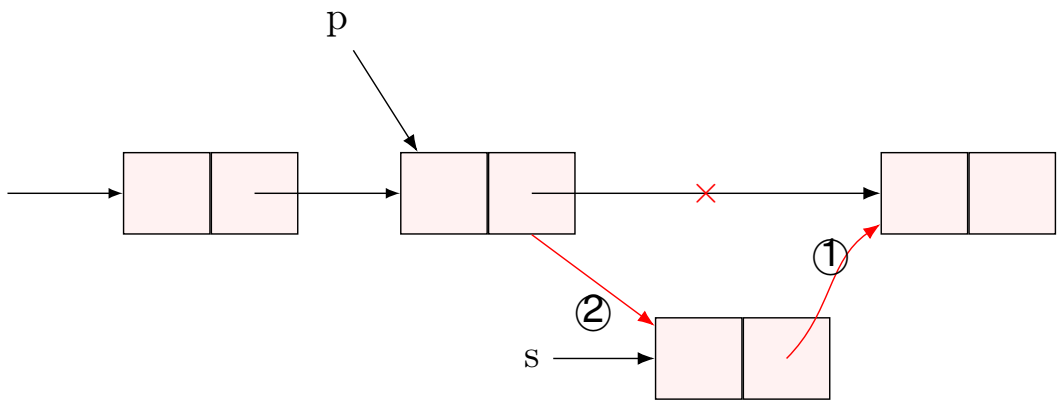
单链表的查找



- 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=hp.next;  int j=0;
  while ( p && p.data !=value){
    p=p->next; j++;
  }
  if(p) return j; else return -1;
```

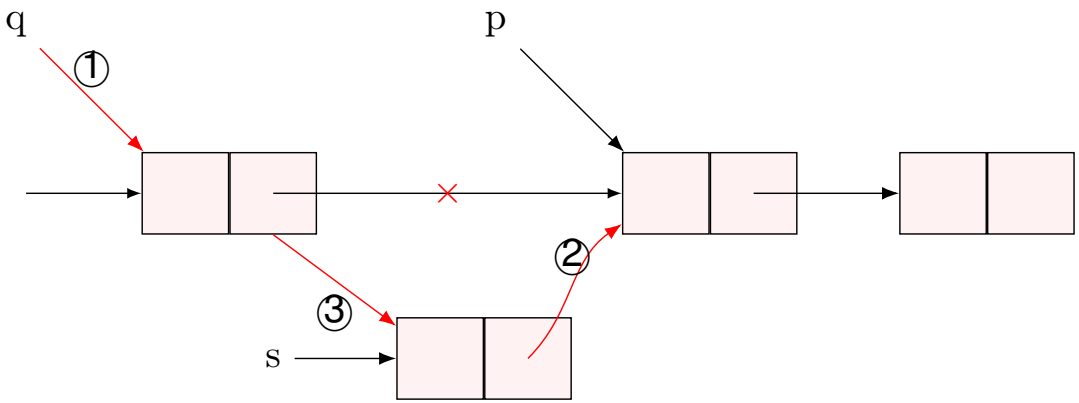
插入新结点：在给定结点的前后



- 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

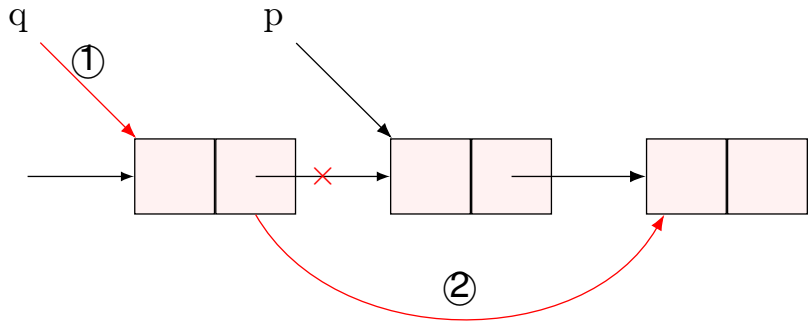
- 时间复杂度
  - 后插操作为 $\mathrm{O}(1)$: 不受 n 的影响
  - 前插操作因为要先找到 p 的前驱，时间性能为 $\mathrm{O}(\mathrm{n})$。

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 $\mathrm{O(n)}$。
- 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 $\mathrm{O}(1)$ 。

删除第 i 个结点

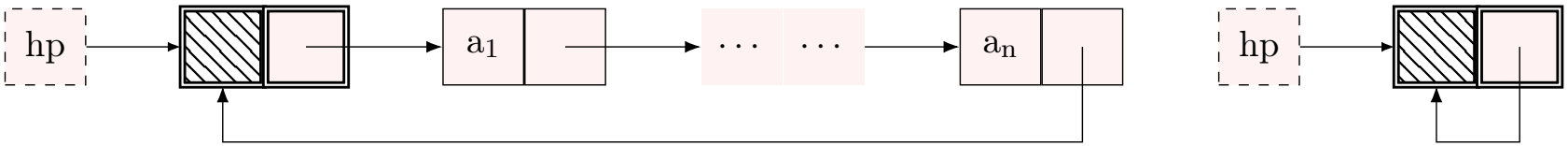
```
int  Del(LinkList  L, int i)  //删除链表 L 第 i 个结
{
  p=get(L, i-1);  //查找第 i-1 个结点
  if (p==NULL) {
    printf(" 第 i-1 个结点不存在");    return -1;
  } else{
    if (!p->next)
      return 0; //第 i 个结点不存在
    else {
      p.next=p.next.next; //从链表中删除 i
      return 1;
    }
  }
```

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。

单循环链表

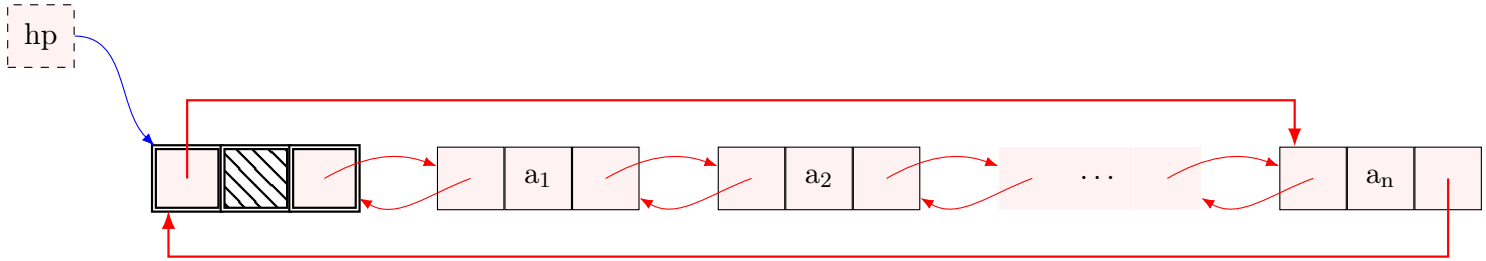
- 链表头尾结点相连



(a) 非空表　　(b) 空表

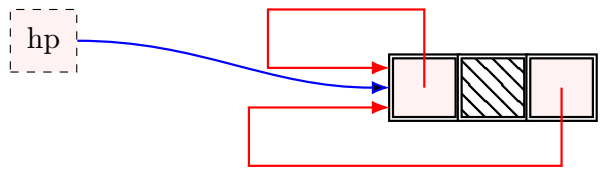
- 结点的定义有什么变化?
- 链表操作有什么变化? 比如:
  - 原来用头结点的 next 是否为 NULL (None) 判断空表，现在呢?
  - 原来用结点的 next 是否为 NULL (None) 判断尾结点，现在呢?

双向链表



(a) 有多个节点时的双向链表

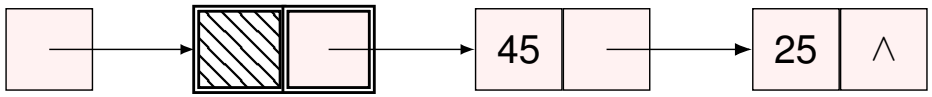
- 为什么需要双向链表?
- 在单链表中查找后继的时间性能是 $O(1)$，查找前驱的时间性能是 $O(n)$，如果也希望找前驱的时间性能达到 $O(1)$，则需要在空间上付出代价——每个结点再加一个指向前驱的指针域。
- 结点的定义和链表的操作有什么变化?



(b) 只有头节点的情况

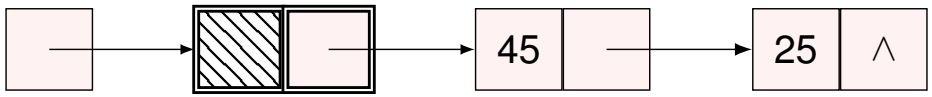
```
class Node {
  Object data;
  Node prior, next;
}
```

链表特点



- 优点
  - 插入删除方便
  - 动态分配

链表特点



- 优点
  - 插入删除方便
  - 动态分配
- 缺点
  - 随机访问效率低

顺序表 VS 链表

顺序表优点

- 直观
- 随机访问效率高
- 没有为表达结点间的逻辑关系增加的额外开销

顺序表缺点

- 插入删除时效率低
- 需要预先分配足够大的存储空间

链表优点

- 插入删除方便
- 动态分配

链表缺点

- 为表达结点间的逻辑关系增加“指针域”
- 随机访问效率低

选取恰当的存储结构 I

1. 基于存储的考虑
   - 使用顺序表，在程序执行之前要明确规定它的存储规模（对 MAXSIZE 要有合适的设定），过大造成浪费，过小造成溢出。可见对线性表的长度或存储规模难以估计时，不宜采用顺序表。
   - 链表不用事先估计存储规模，但链表的存储密度较低。存储密度是指一个结点中数据元素所占的存储单元和整个结点所占的存储单元之比。显然链式存储结构的存储密度是小于 1 的。

选取恰当的存储结构 II

2. 基于运算的考虑
   - 在顺序表中按序号访问 a_i 的时间性能时 $O(1)$，而链表中按序号访问的时间性能 $O(n)$，所以如果经常做的运算是按序号访问数据元素，显然顺序表优于链表；
   - 在顺序表中做插入、删除时平均移动表中一半的元素，如果数据元素的信息量较大且表较长，不可忽视这一点；
   - 在链表中作插入、删除，虽然也要找插入位置，但操作主要是比较操作，从这个角度考虑优于顺序表。

选取恰当的存储结构 III

3. 其它
   - 顺序表容易实现，任何高级语言中都有数组类型，相对来讲简单直观，也是用户考虑的一个因素。

总之，两种存储结构各有长短，选择那一种由实际问题中的主要因素决定。通常“较稳定”的线性表选择顺序存储，而频繁做插入删除的话宜选择链式存储。

本堂小结

1. 线性表的概念和两种存储方式.
2. 顺序线性表的特征、基本操作.
3. 链表的特征、基本操作.
4. 循环链表和双向链表
5. 顺序表与链表大 PK
6. 顺序表和链表适用的场合

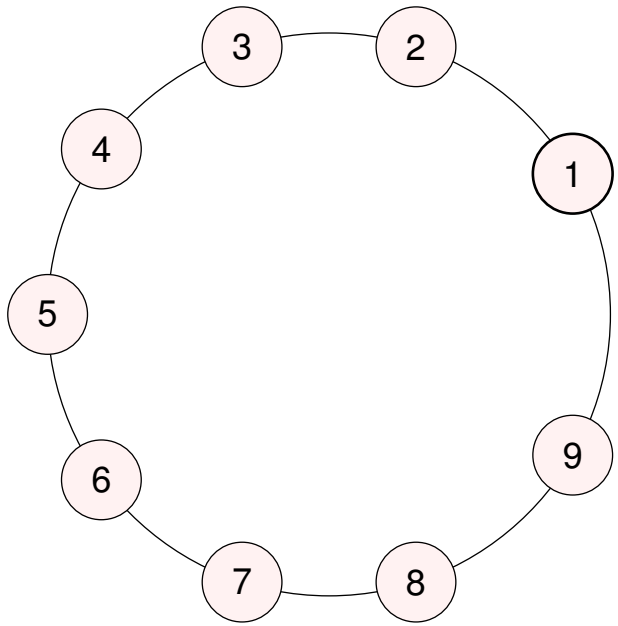
约瑟夫环

在罗马人占领乔塔帕特后，约瑟夫（Joseph）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?

约瑟夫环

- 已知 n 个人（以编号 $1, 2, 3 \cdots n$ 分别表示）围坐在一张圆桌周围。从编号为 k 的人开始报数，数到 m 的那个人出列；他的下一个人又从 1 开始报数，数到 m 的那个人又出列；依此规律重复下去，直到圆桌周围的人全部出列。
- 例如：n = 9, k = 1, m = 5；出局顺序如下



约瑟夫环动画演示

见 PPT 动画

栈和队列

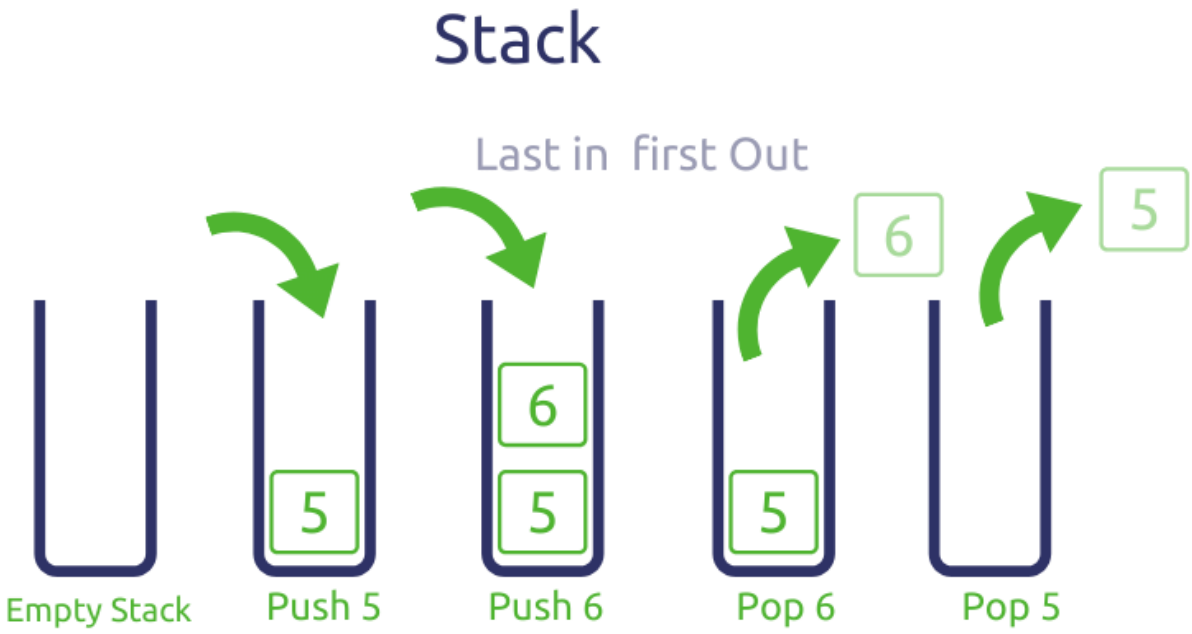
栈和队列：两种特殊的线性表

栈/Stack

- 限制仅在表的一端进行插入和删除。
- 通常称插入、删除的一端为栈顶 (Top)，另一端为栈底 (Bottom)
- LIFO：Last In First Out

队/Queue

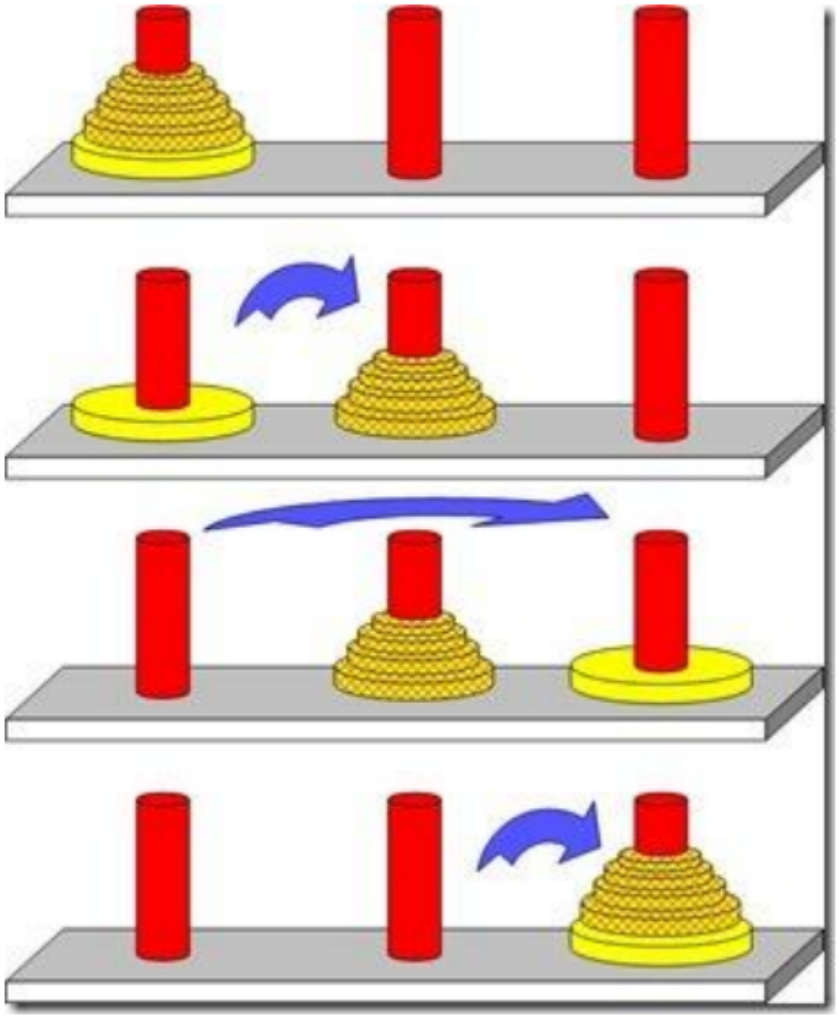
- 限制仅在表的一端进行插入、在另一端进行删除。
- 允许插入的一端称队尾 (rear)，允许删除的一段称为队头 (front)。
- FIFO：First In First Out



- 操作系统中的栈
  - 由编译器自动分配释放，存放函数的参数值，局部变量的值等。栈使用的是一级缓存，被调用时处于存储空间中，调用完毕立即释放。
- 数据结构中的栈
  - 一种后进先出的数据结构

为什么设计栈、研究栈?

- 栈的一个重要应用是在程序设计中实现递归，从而使许多实际问题大大简化。

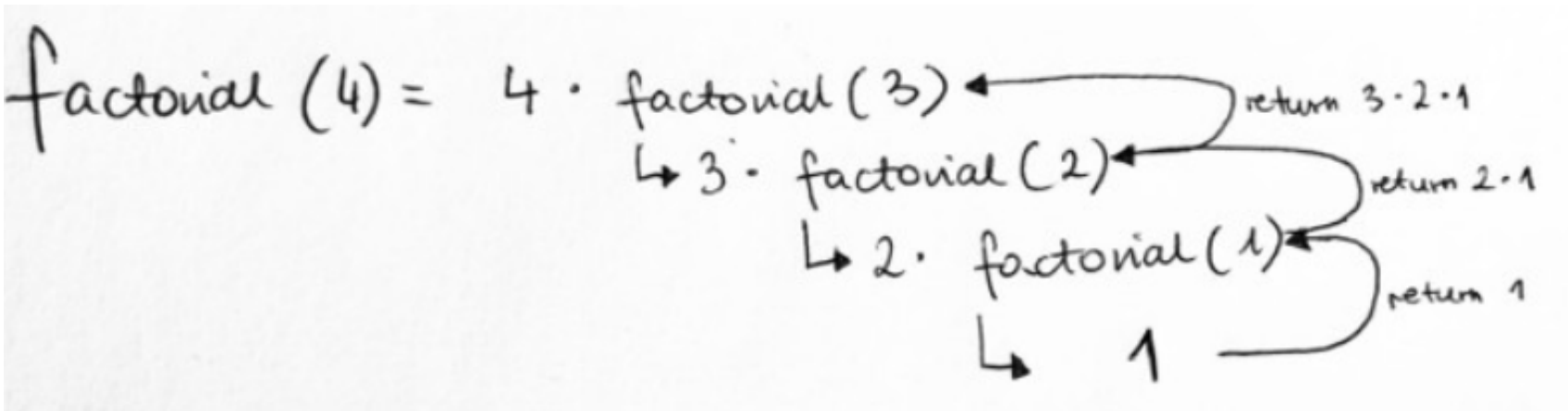


- 上帝创造世界的时候做了三根金刚石柱子，在一根柱子上从下往上按大小顺序摞着 64 片黄金圆盘。上帝命令婆罗门把圆盘从下面开始按大小顺序重新摆放在另一根柱子上，并且规定一次只能移动一个圆盘，在小圆盘上不能放大圆盘。
- 有预言说，这件事完成时宇宙会在一瞬间闪电式毁灭。也有人相信婆罗门至今还在一刻不停地搬动着圆盘。
- 18,446,744,073,709,551,615 次搬动才能挪完 64 片金盘!

举例：计算 n 的阶乘

```
int factorial (int n) {
  int  f ;
  if (n==1) f=1;
  else f=n*fact (n-1) ;
  return f;
}
```

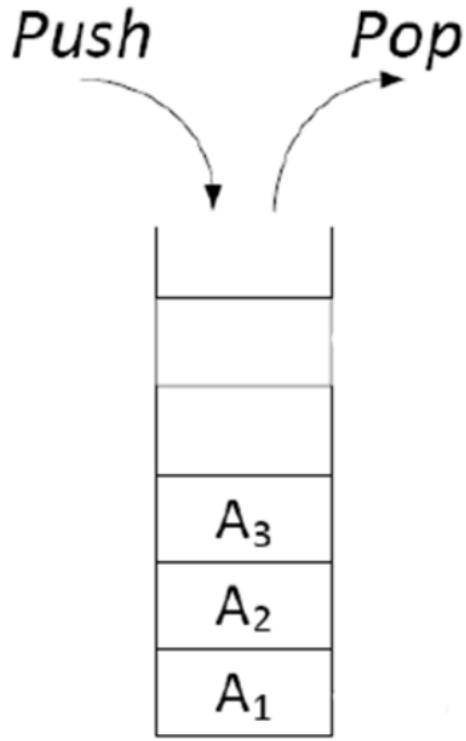
1. 将调用函数的现场 (各寄存器的值，中断时的程序地址等) 入栈，转入被调函数；
2. 执行被调函数，如又调用其它函数，则执行上述步骤；
3. 被调函数执行完，取栈顶的值，恢复调用函数时的现场，根据现场中的指令地址，恢复调用函数在中断处继续执行。



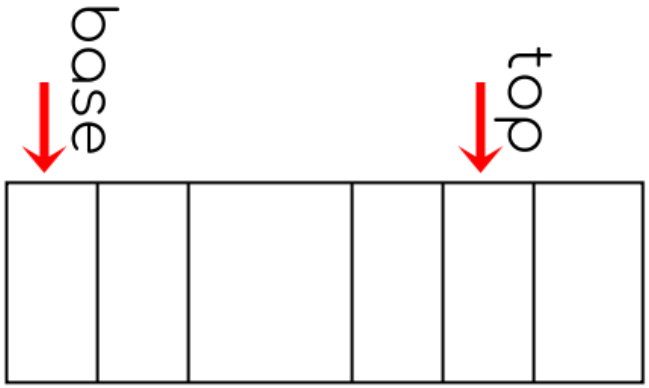
栈的存储表示方法

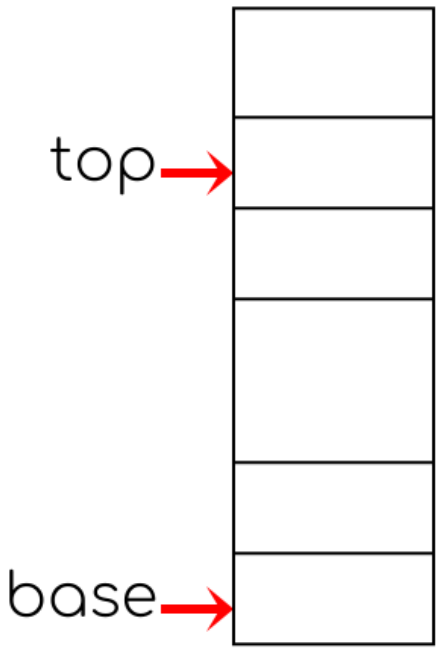
请考虑其常用操作。试想选用顺序存储还是链式存储?

- 特点：后进先出
  - 1、经常性的在栈顶插入新元素，以及取栈顶元素；
  - 2、无须访问非栈顶元素。



1. 顺序栈



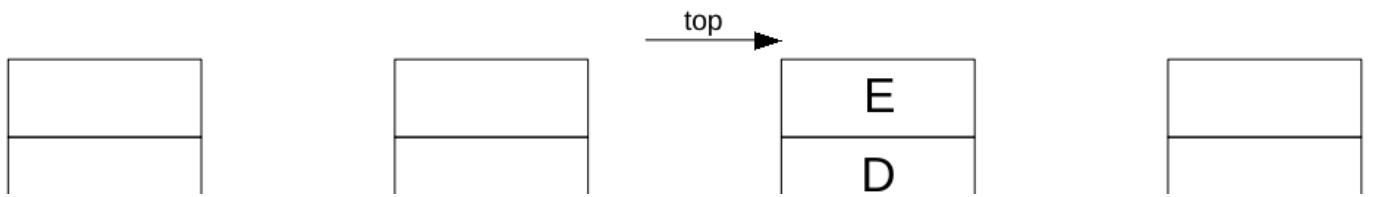


1. 顺序栈中元素用地址连续的存储单元依次存放；
2. 栈底位置固定不变；
3. 栈顶位置 top 随着进栈和出栈的操作而变化。

顺序栈类型定义：

```
class stack<Elem>{
  Object[] data;
  int top;
  int maxSize;
} SeqStack;
```

- 动态分配
  - 先为栈分配一个初始容量，在栈的空间不够使用时再逐段扩大。
- 指针 base
  - 始终指向栈底位置，如果 base 为 NULL 表示栈不存在；
- 指针 top
  - 初值指向栈底，即 top = =base，表示栈空；(不唯一)
  - 每当插入新的栈顶元素，指针 top++；
  - 每当删除栈顶元素，指针 top--；
  - 非空栈中的栈顶指针始终在栈顶元素的下一个位置上。



对于顺序栈，入栈需要判栈是否满，是则需要中止、或重新分配空间,否则出现空间溢出。

```
public Boolean push (Elem e) {
  if ( top==maxSize) {
    print(" 栈已满");
    return false;
  }
  data[top++] = e;
  return true;
}
```

对于顺序栈，入栈需要判栈是否满，是则需要中止、或重新分配空间,否则出现空间溢出。

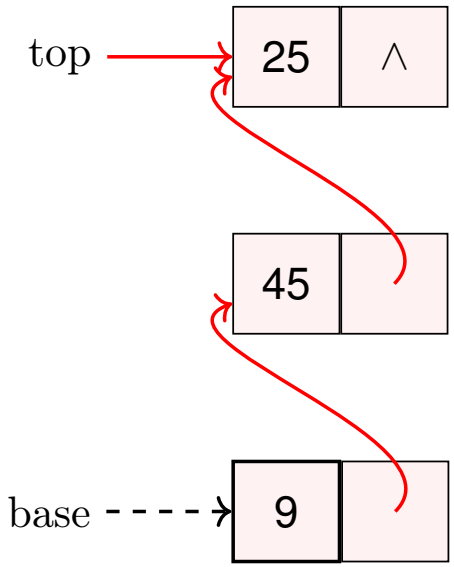
```
public Boolean push (Elem
  if ( top==maxSize) {
    print(" 栈已满");
    return false;
  }
  data[top++] = e;
  return true;
}
```

出栈首先要判断栈是否为空；否则栈空时进行操作将出现下溢错误。

```
public Elem pop() {
  if(top==base) {
    return null;
  } else {
    top = top-1
    return data[top];
  }
}
```

2. 链栈

- 链栈有无栈满溢出问题?
- 指针如何指?
  - 从栈顶依次向后指，因为操作主要是在栈顶插入、删除，经常需要根据栈顶元素找次顶元素。
- 是否要加头结点?
  - No. 因为头部插入不会出现处理不一致的问题.



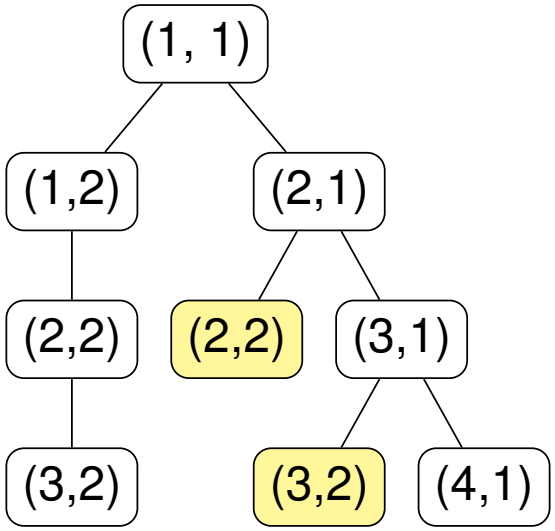
写出链栈的类型定义:

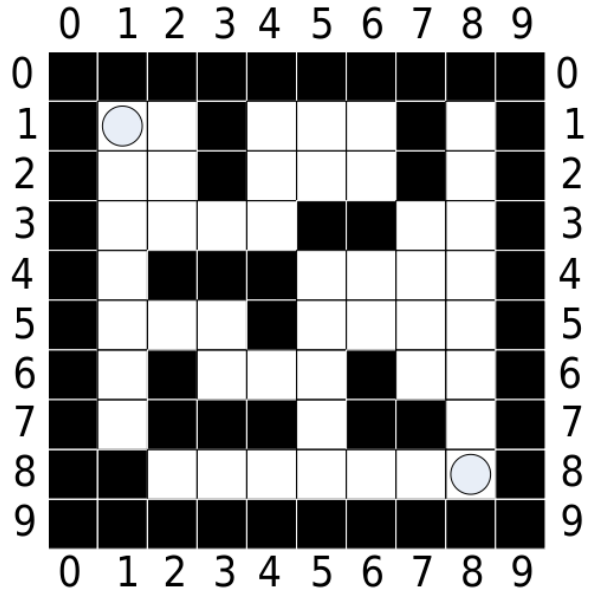
```
class Node{
  Elem data;
  Node next;
}

Node  top ; // 链栈
```

请尝试写出入栈出栈的算法

```
 public push (Elem e) {
    //是否还要判断栈是否满？
    top=new Node(e, top);
    return OK;
}

public pop( ){
   //是否还要判断栈是否空？
   e=top;
   top=top.next;
   e.next=null;
   return e.data;
}
```

栈的应用 — 迷宫求解





- 处理思路：
  - 对可通行路径的空间的深度优先搜索
  - 如果当前位置是出口，找到结果
  - 否则，保存当前位置，进入到下一个可以走的位置，探索路径
  - 如果上一步没有成功，继续探索下一个可以走的位置。
  - 如果四个方向都没有成功，说明这条路

队/Queue

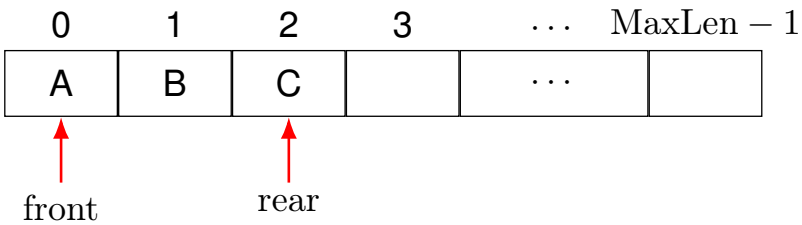
队/Queue

- 限制仅在表的一端进行插入、在另一端进行删除。
- 允许插入的一端称队尾 (rear)，允许删除的一段称为队头 (front)。
- FIFO：First In First Out

- 比如生活中排队购物、操作系统中的作业排队等。

队的存储表示方法

- 队也有两种存储表示方法: 顺序队、链队
- 顺序队：利用地址连续的存储单元依次存放数据元素。



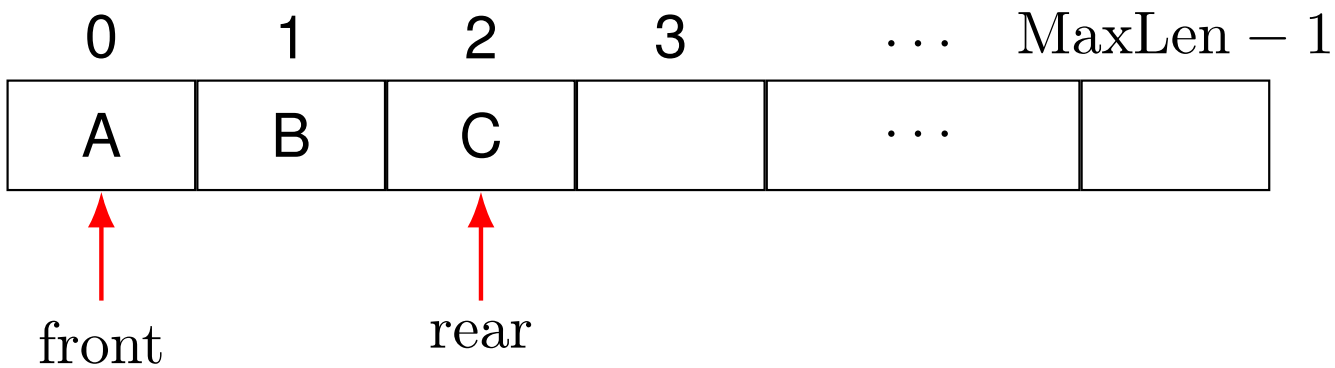
请写出顺序队的类型定义：

```
class SeqQueue{
  ElemType  data[ ]
  int maxsize;
  int rear; //队尾,
}
```

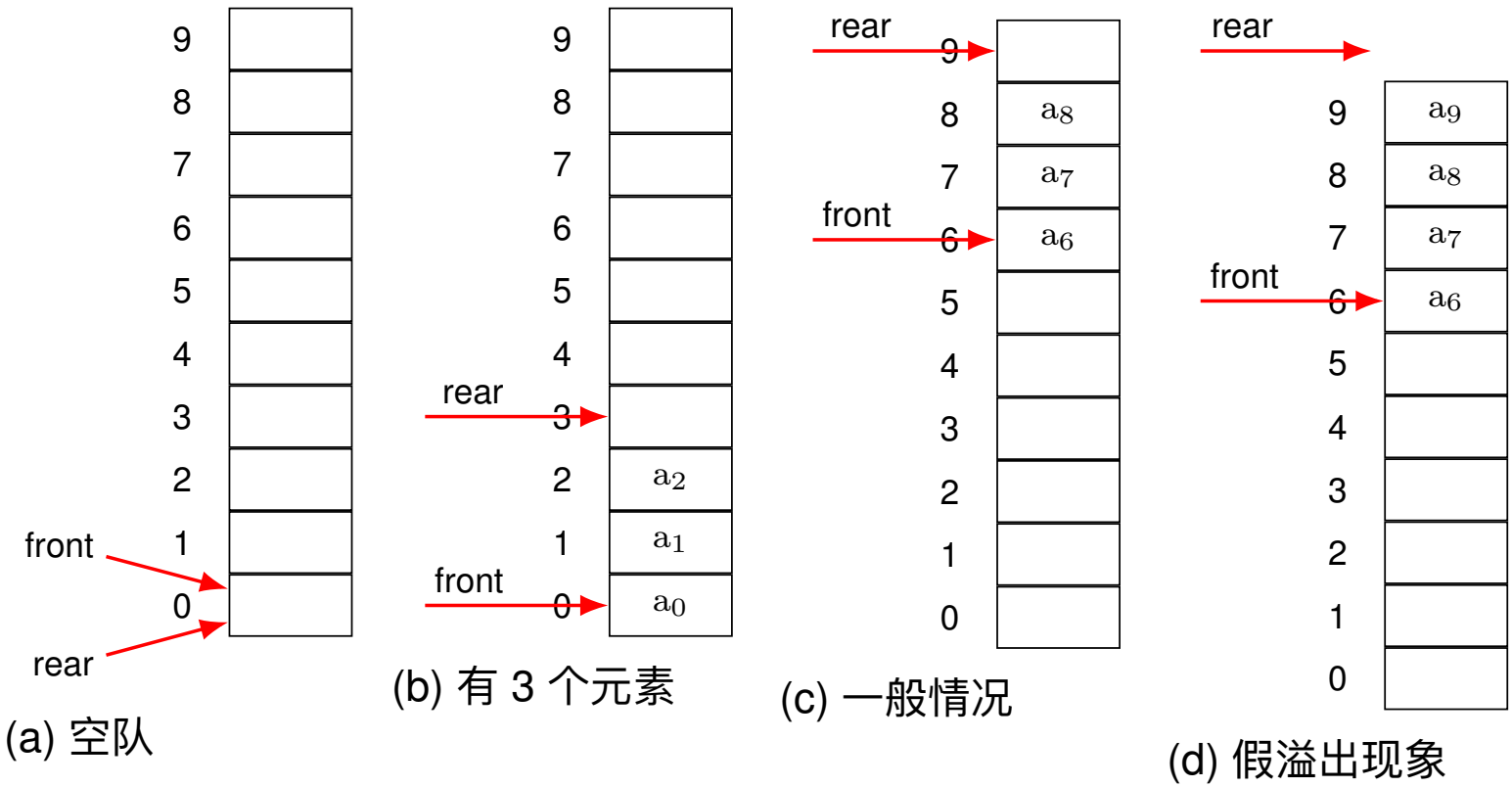
可否??

顺序队的基本操作

- 入队：如有空间，元素 x 入队后队尾指针加 1
  - data[rear++]=x;
- 出队：如有元素，队头指针加 1，表明队头元素出队。
  - x=sq.data[sq.front++];

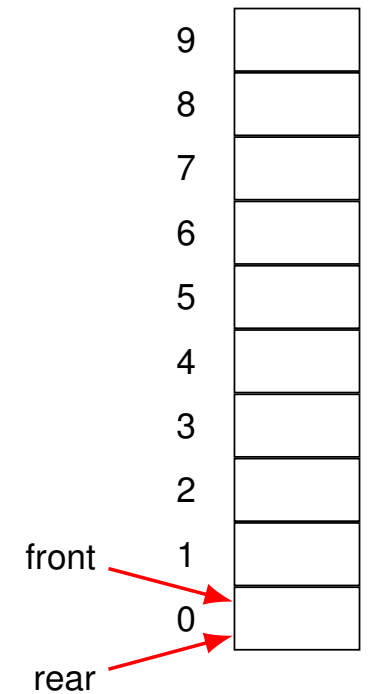


- 设 maxsize=10。请你写出如下状态或执行某操作后的顺序队元素。
  1. 空队；
  2. a0, a1, a2 依次入队；
  3. a3, a4, a5, a6, a7, a8 依次入队， a0, a1, a2, a3, a4, a5 依次出队；
  4. a9, a10 入队。

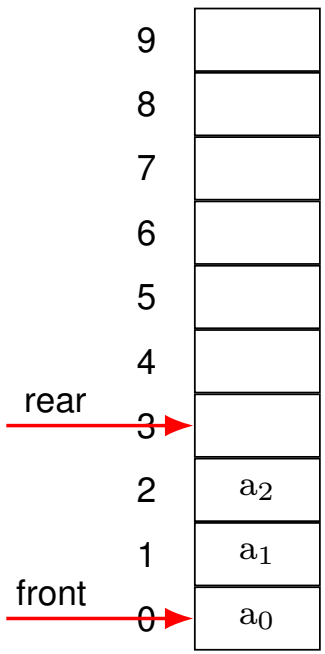


(a) 空队　(b) 有 3 个元素　(c) 一般情况　(d) 假溢出现象

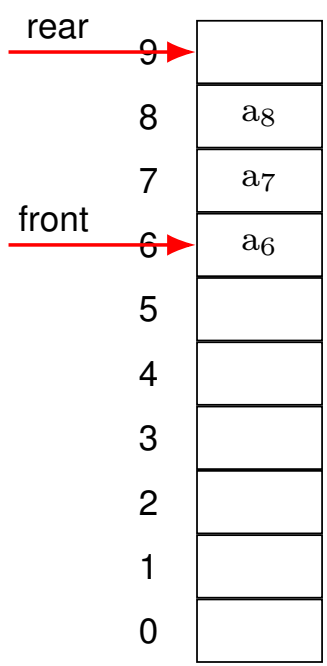
- 设 maxsize=10。请你写出如下状态或执行某操作后的顺序队元素。
  1. 空队；
  2. a0, a1, a2 依次入队；
  3. a3, a4, a5, a6, a7, a8 依次入队，a0, a1, a2, a3, a4, a5 依次出队；
  4. a9, a10 入队。



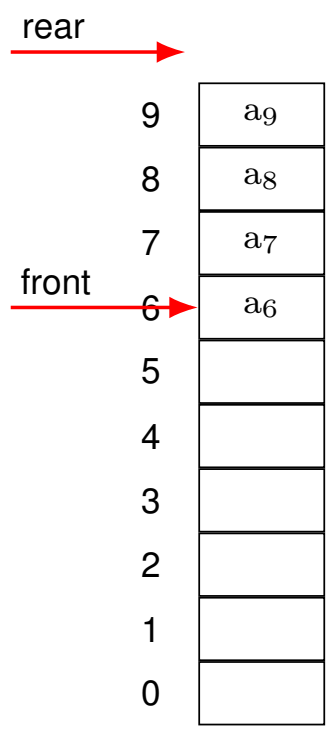
(a) 空队



(b) 有 3 个元素



(c) 一般情况



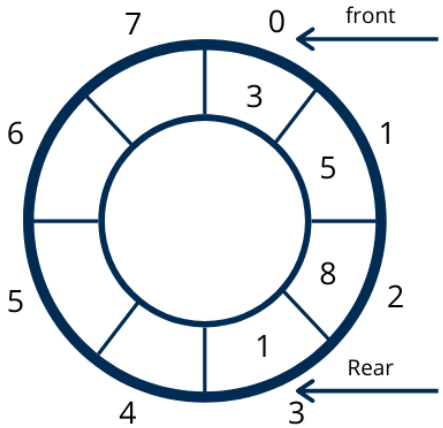
(d) 假溢出现象

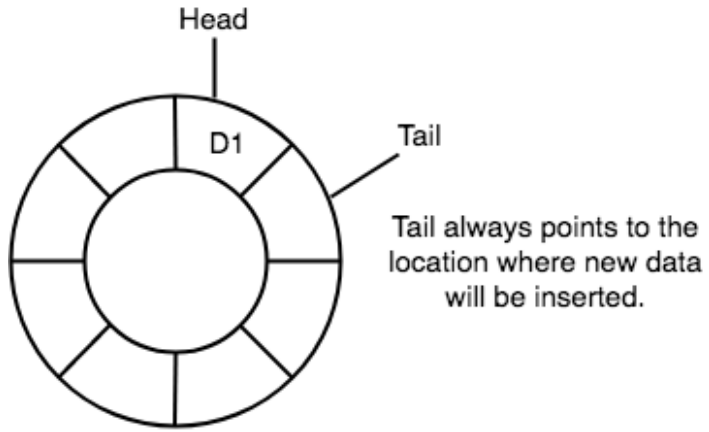
- 入队出队会使整个队列整体后移
- “假溢出”：队尾指针已经移到了最后，不能再执行入队操作，但此时队中并未真的“满员”。如何解决这个问题?

- Linear Queue 入队出队会使整个队列整体后移
- “假溢出”: 队尾指针已经移到了最后, 不能再执行入队操作，但此时队中并未真的“满员”。如何解决这个问题?
  1. 平移元素;
  2. 循环队列 (Circular Queue)：将顺序队列的存储区假想为环状空间。在假溢出时，将新元素插入到第一个位置上，这样做，虽物理上队尾在队首之前，但逻辑上队首仍然在前。入列和出列仍按“先进先出”的原则进行。
     - 入队时的队尾指针操作：
       rear=(rear+1) % maxsize;
     - 出队时的队头指针操作：
       front=(front+1) % maxsize;

循环队列 (Circular Queue)[1]

- head/front points to the first (oldest) used element — the next element to be read
- tail/rear points to the first (oldest) unused element — the next element to be written
- rear: 多数情况下指向尾部元素，tail: 多数情况下指向尾部的下一个元素。教材中的 rear 等同于 tail



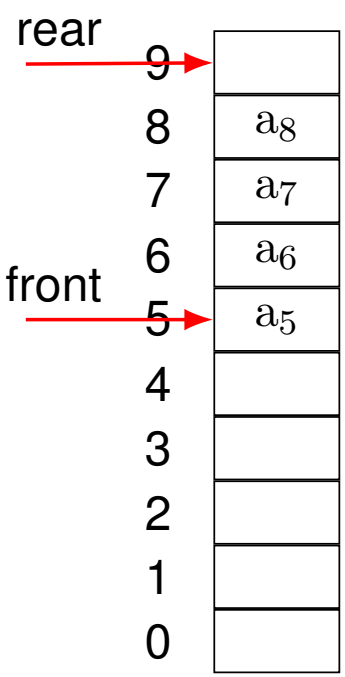


[1]延伸阅读：https://en.wikipedia.org/wiki/Circular_buffer

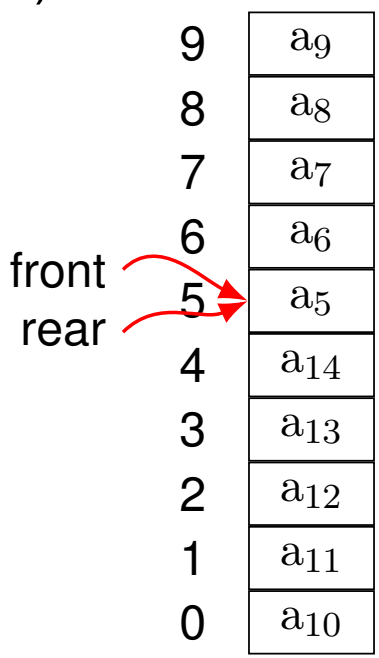
- 设 maxsize=10。队列状态如 (a) 所示。请写出分别执行如下操作后的 front 和 rear 值：
  - 情况 1: a9, a10, a11, a12, a13, a14 依次入队
  - 情况 2: a5, a6, a7, a8 依次出队

- 设 maxsize=10。队列状态如 (a) 所示。请写出分别执行如下操作后的 front 和 rear 值：
  - 情况 1: a9, a10, a11, a12, a13, a14 依次入队
  - 情况 2: a5, a6, a7, a8 依次出队

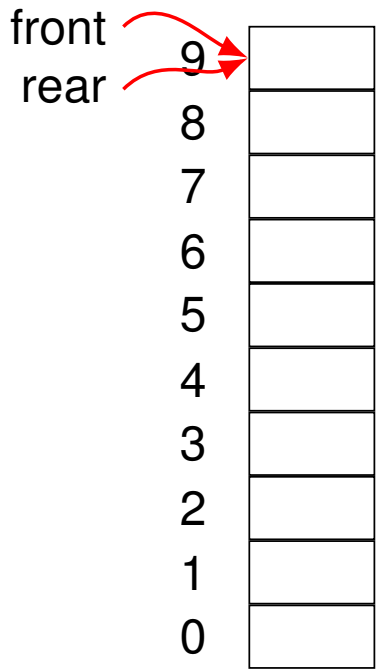
(a) 有 4 个元素



(b) 情况 1: 队满



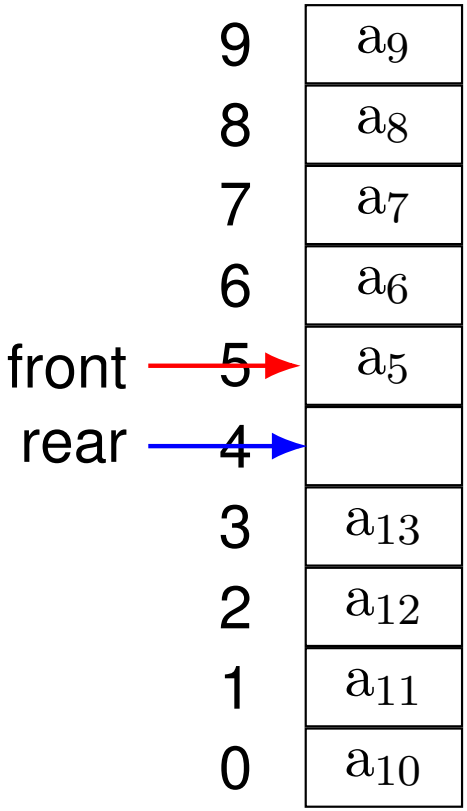
(c) 情况 2: 队空



当 front=rear, 队满还是队空？如何判断队满？如何判断队空？

循环队的队满判别

- 方法一
  - 附设一个存储队中元素个数的变量如 num，当 num=0 时队空，当 num=maxsize 时为队满。
- 方法二
  - 少用一个元素空间，把右图所示情况视为队满，此时的状态是队尾指针加 1 就会从后面赶上队头指针：(rear+1) % maxsize==front，从而和空队区别开来。



循环队的基本操作

```
class SeqQueue  {
  ElemType data[];
  int maxsize;
  int front, rear;   /* 队头队尾指针 */
  int num;           /* 队中元素的个数 */
}

// 入队
int InQueue (ElemType  x){
  if  (num==maxsize) {
    printf(" 队满");  return 0;
  } else {
    data[rear]=x;
    rear=(rear+1) % maxsize;
    num++;
    return 1;     /* 入队完成 */
  }
}
```

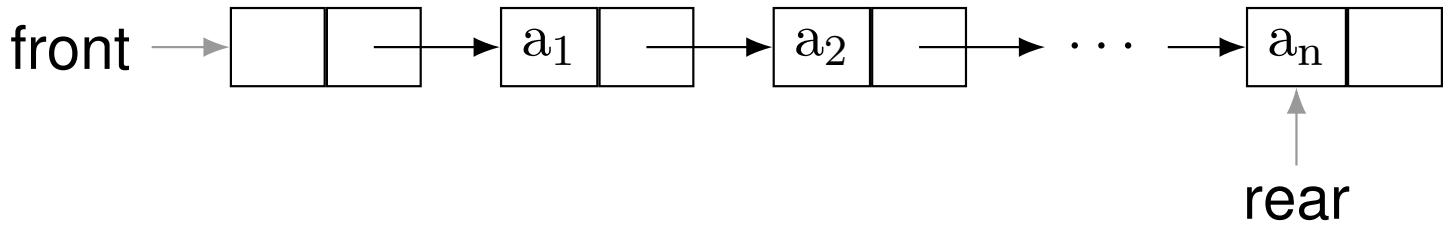
循环队的基本操作

```
//出队
int OutQueue (ElemType  x) {
  if  (num==0) {
    printf(" 队空");
    return -1;
  } else {
    x=data[front];  /* 读出队头元素 */
    front=(front+1) % maxsize;
    num--;
    return 1;     /* 出队完成 */
  }
}
```

Python:

```
class SeqQueue:
    def __init__(self, max_size=10):
        self.max_size = max_size + 1 # 实际多放一个空间，区分对空、队满
        self.front = 0
        self.rear = 0
        self.elements = [None]*max_size

    def enqueue(self, element) -> bool:
        if (self.rear + 1) % self.max_size == self.front:
            print('Queue is Full!')
            return False
        else:
            self.elements[self.rear] = element
            self.rear = (self.rear + 1) % self.max_size
            return True

    def dequeue(self):
        if self.front == self.rear:
            print('Queue is Empty!')
```

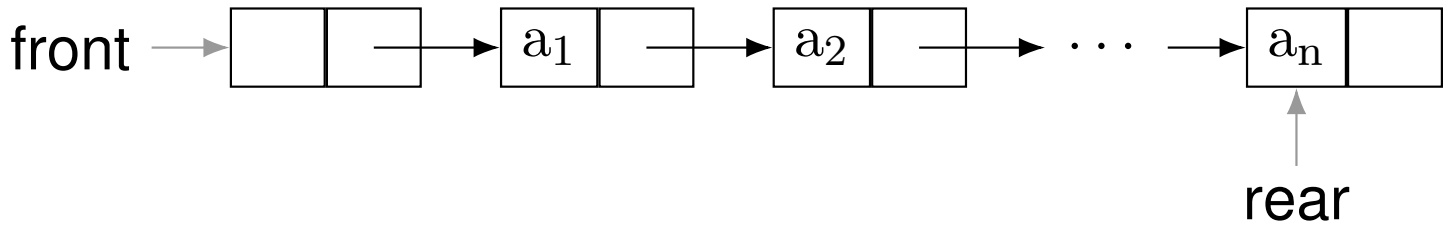
链队



- 带头结点吗?
- 尾指针 rear 指向队尾元素，提高插入操作效率
- 请写出链队的类型定义

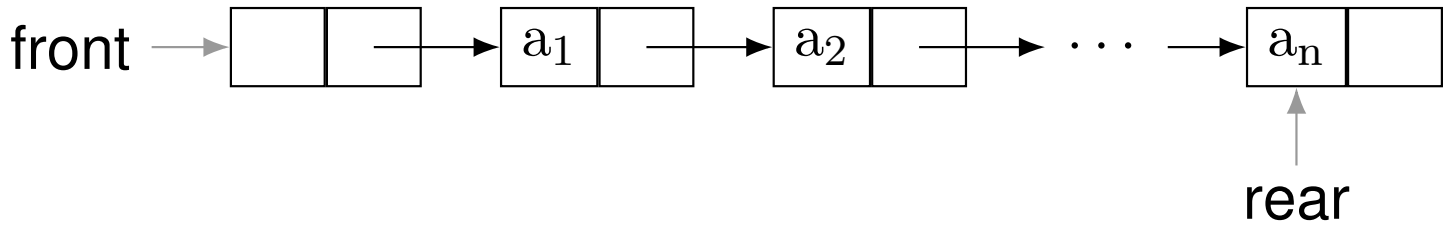
```
class QNode{
  ElemType  data;
  QNode next;
}
class LQueue{
```

链队的基本操作 — 入队




```
void InQueue(ElemType  x) {
   rear.next=new QNode(x,null);
   rear=rear.next;
}
```

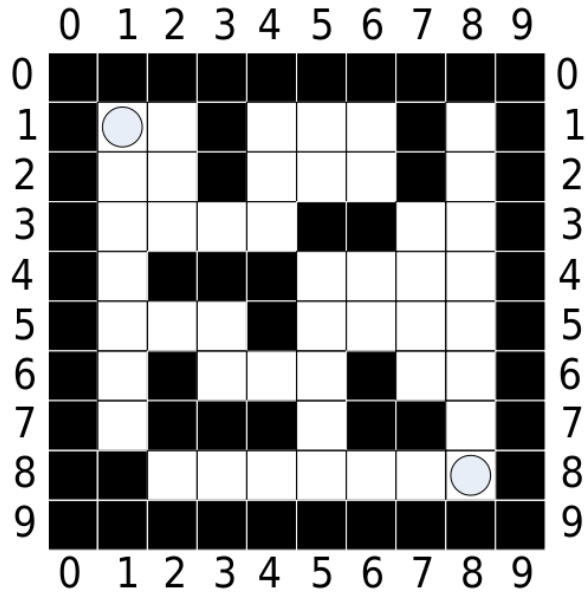
链队的基本操作 — 出队



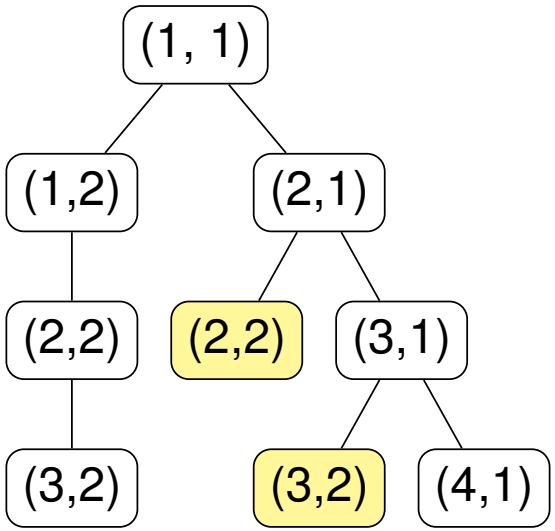

```
ElemType OutQueue() {
  QNode p;
  if (front==rear) {
    printf (" 队空");
    return null;
  } else {
    p=front.next;      //把队的第一个结点取出
    front.next=p.next;    //把队头直接指向原第二个结点
    e=p.data;          /* 队头元素放 e 中 */
    if (front.next==NULL)
```

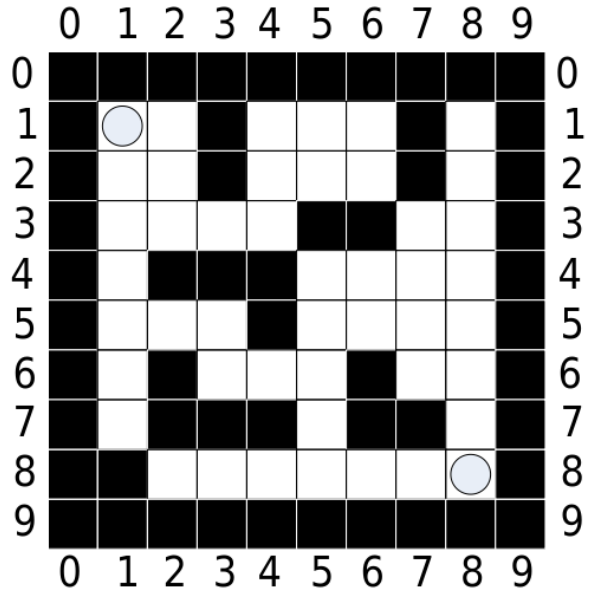
队列应用 — 迷宫寻路

- 从迷宫入口点（1,1）出发，向四周搜索，记下所有一步能到达的坐标点；然后依次再从这些点出发，再记下所有一步能到达的坐标点，…，依此类推，直到到达迷宫的出口点 (8,8) 为止，然后从出口点沿搜索路径回溯直至入口。这样就找到了一条迷宫的最短路径，否则迷宫无路径。
- 与基于栈的迷宫求解的异同?



解法图示

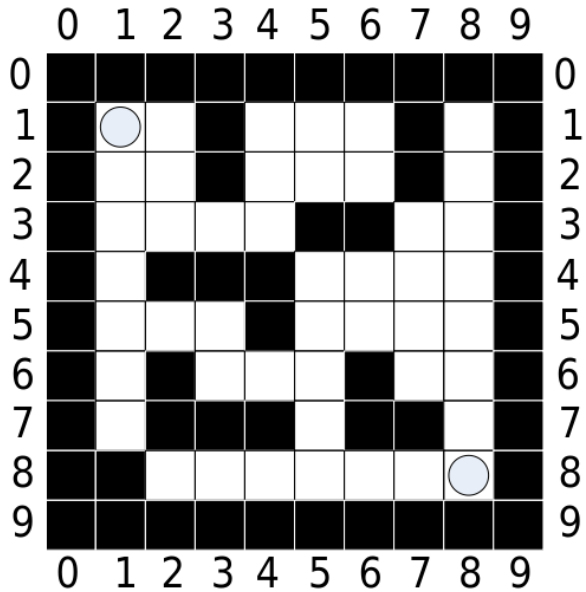




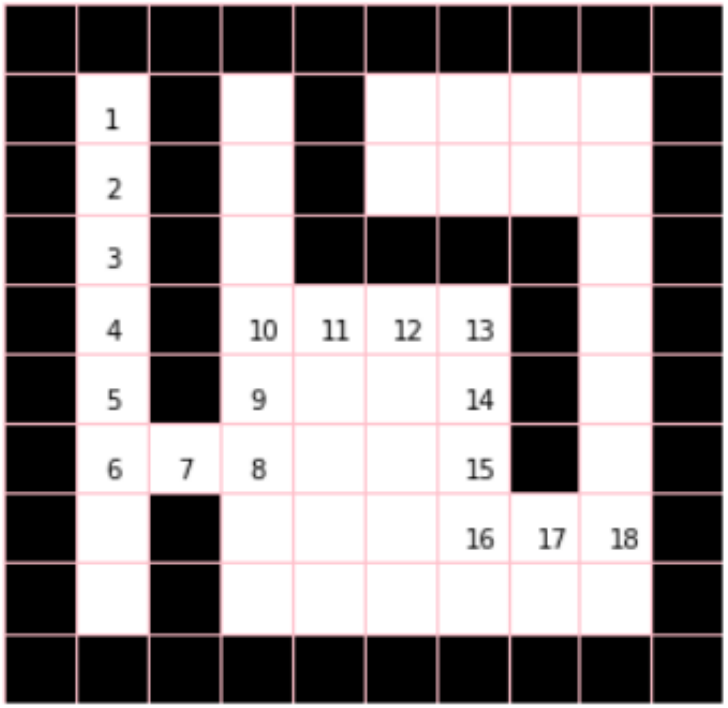
- 上述处理的实质：
  - 对可通行路径的空间的广度优先搜索
- 如何向前摸索?
- 到达出口后，如何回溯?

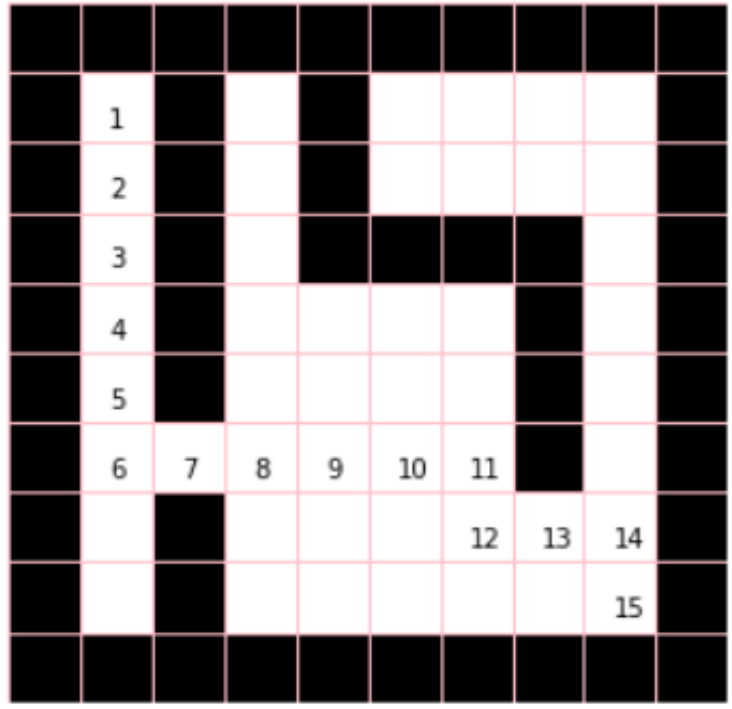
解法图示

- 如何向出口摸索：搜索路径的存储 1
  - 在搜索过程中必须记下每一个可到达的坐标点，以便从这些点出发继续向四周搜索，使用“FIFO”的队列保存已到达的点即可.
- 如何向入口回溯：搜索路径的存储 2
  - 为了能够从出口点沿搜索路径回溯直至入口，对于每一点，记下坐标点的同时，还要记下到达该点的前驱点或者从前驱点来的方向（8 个方向之一）。



夏老师实现结果的对比





实现代码见 ipynb/stack.ipynb 和 ipynb/queue.ipynb