
计算思维通识教育

Computational Thinking

# 第3章 数据间的逻辑关系

主讲人：曹轶臻

联系方式：caoyizhen@cuc.edu.cn

计算思维是一种选择合适的方式去陈述一个问题或对问题的相关方面建模，使其易于处理的思维方法

CONTENTS



计算机与网络空间安全学院
School of Computer and Cyber Sciences

计算思维通识教育
Computational Thinking

# 01

# 认识数据间的逻辑关系

数据间的逻辑关系 1-1

什么是数据结构 1-2

数据结构与算法 1-3

数据结构的分类 1-4

计算机与网络空间安全学院
School of Computer and Cyber Sciences

# 1-1 数据间的逻辑关系



- 一台晚会的节目顺序安排/火车的多个车厢 — **链表**
- 糖果分配器/手枪弹夹/只有一个出入口的直梯 — **堆栈**
- 扶梯/在食堂打饭时 — **队列**
- 分专业、分班时 — **树**
- 微博的关注/在朋友圈点赞评论时 — **图**

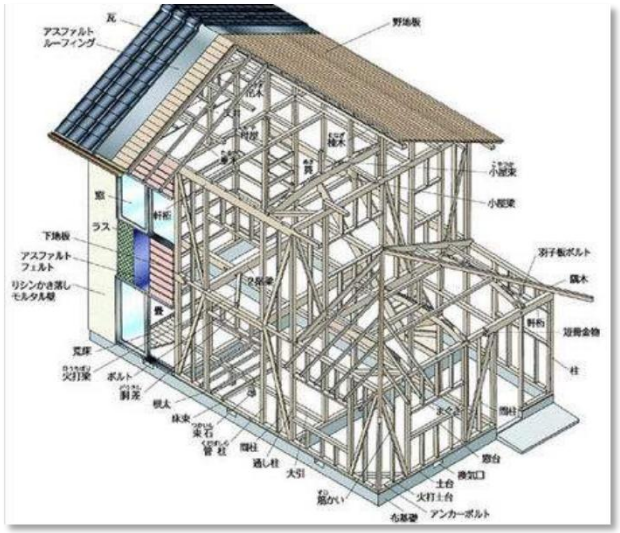
如何在计算机里存储这些数据?

什么样的存储和表示方式更高效?

# 1-2 什么是数据结构



当初人们试图建造计算机的主要原因之一是用来存储和管理一些数字化的信息和数据，这也是最初数据结构概念的来源。



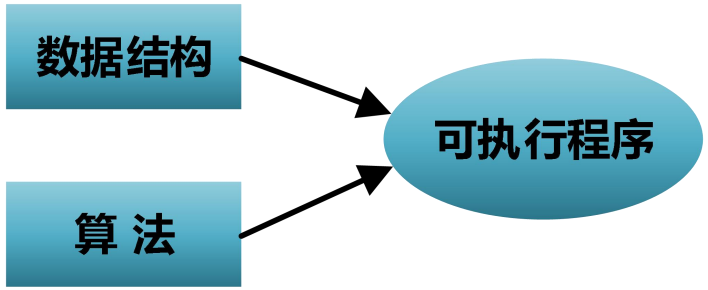
数据结构（Data Structure）是数据的表示法，用于存储和组织数据。

编写程序就像盖房子一样，要先规划出房子的结构图。

# 1-3 数据结构+算法



使用适合的数据结构，再通过程序设计语言所提供的数据类型、引用方法以及相应的操作就可以实现对应的**算法**

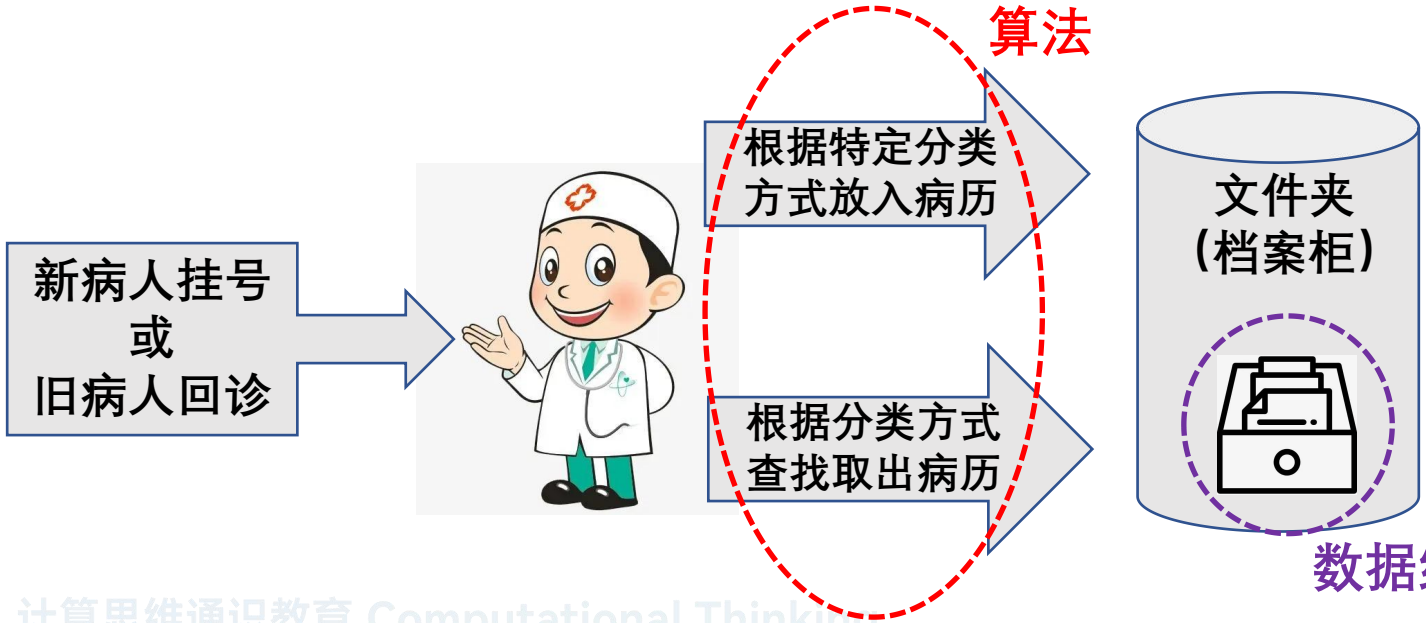



- 一个程序能否快速而有效地完成预定的任务取决于是否选对了数据结构，
- 而程序是否能清楚而正确地把问题解决则取决于算法。

因此，可以认为：

“数据结构+算法=有效率的可执行程序”

以医院为例，将病历表准备好，新病人填写好个人信息，管理人员就可以按照某种次序（例如姓氏、年龄或电话号码）将**病历表**加以分类，然后存档。



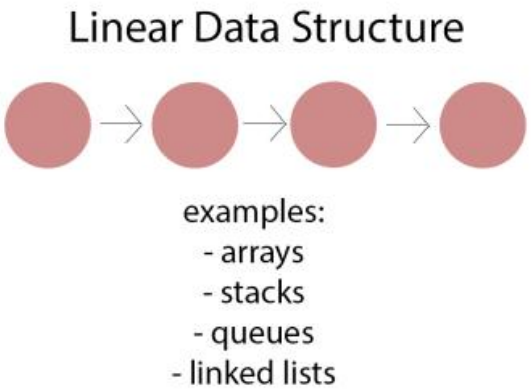
# 1-4 数据结构分类

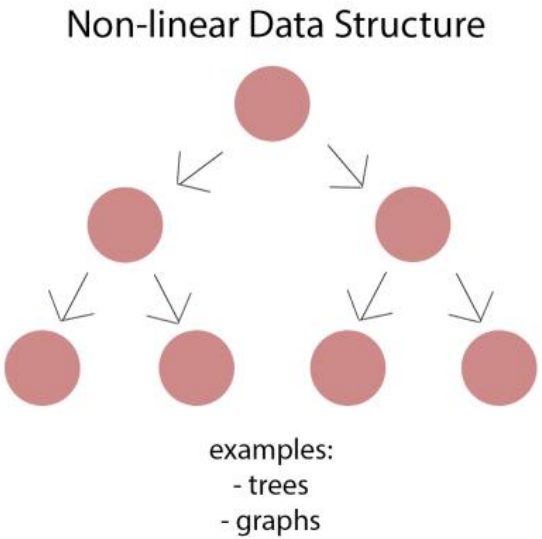


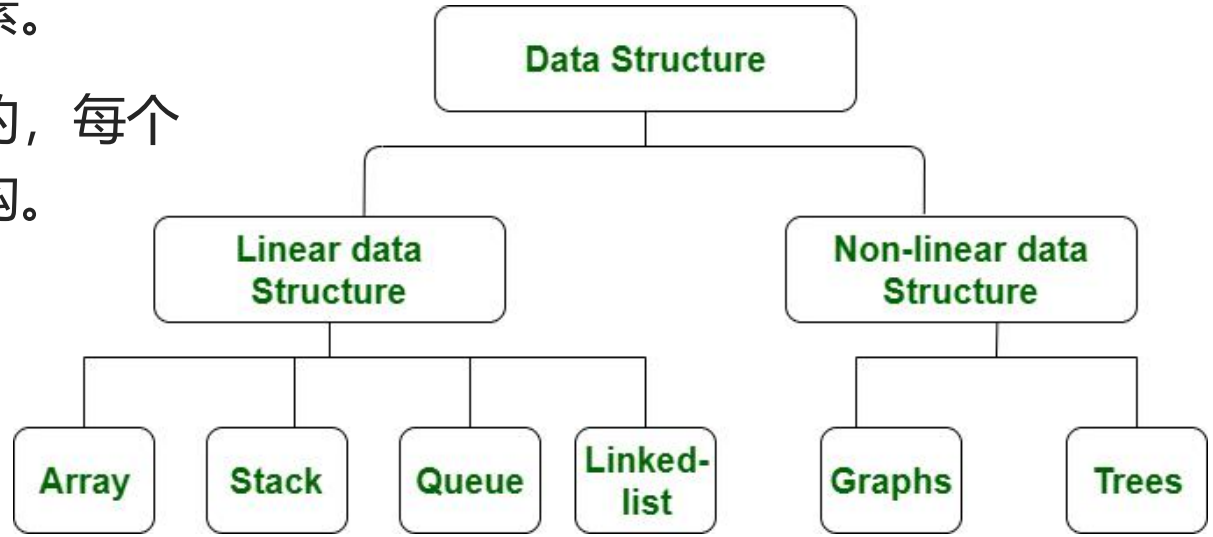
按照数据间的逻辑关系，数据结构可分为

线性数据结构，是一种数据项按顺序或线性排序的结构，并且每个成员都附加到其前一个和下一个相邻元素。

非线性数据结构，是一种数据项不是按顺序排列的，每个成员可以通过多条路径连接到其他成员的数据结构。



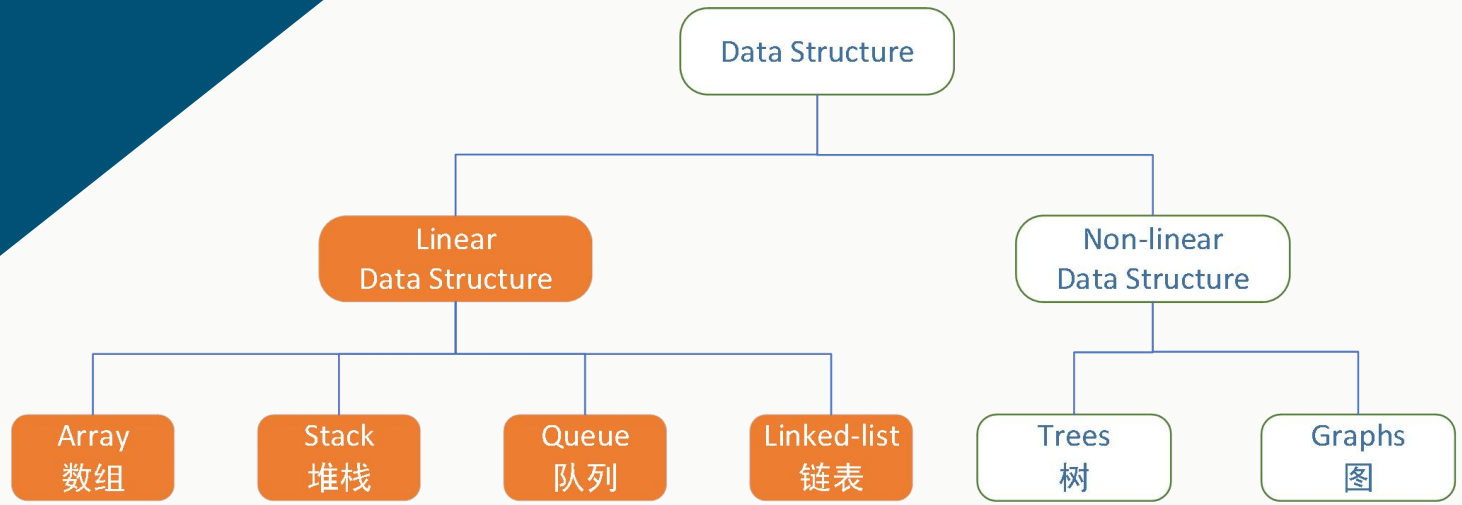




不同种类的数据结构适用于不同种类的程序应用，选择适当的数据结构是让算法发挥最大性能、程序效率更高的主要因素。

# 02

# 线性数据结构

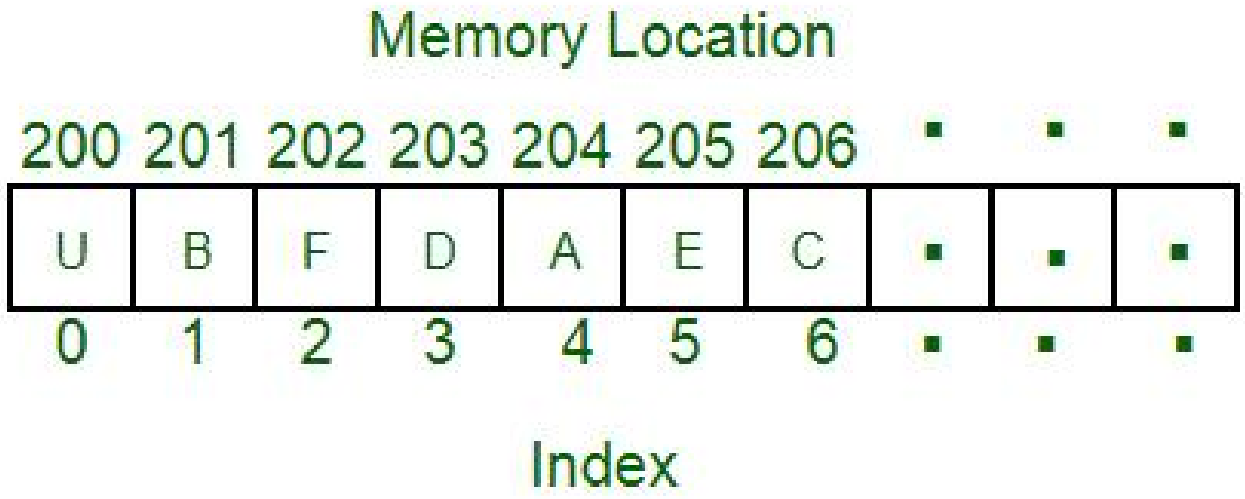


# 2-1 数组



数组（Array）是存储在连续内存位置的数据的集合。

将多个相同类型的数据存储在一起，这使得计算每个元素的位置变得更容易，只需将偏移量添加到数组第一个元素的内存位置。




Python中提供了相似但更灵活的基础类型：**List**

列表同样可以是使用索引，且不要求数据元素必须是同一类型。

每个元素都可以通过它们在数组中的索引来唯一标识，例如 letter[6] ='C'

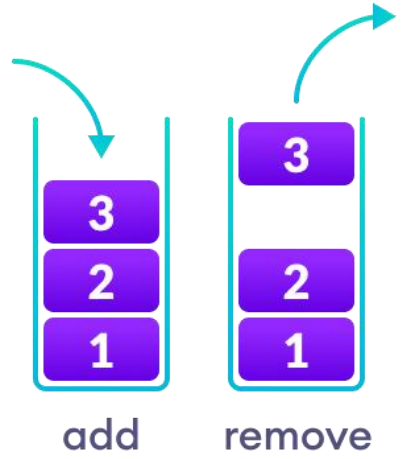
# 2-2 堆栈 Stack



堆栈（Stack）是一组相同数据类型的线性组合，

所有的操作均在堆栈顶端进行，

具有“后进先出”（LIFO，Last In First Out）的特性。




糖果堆栈具有哪些操作?

- 从最上面取出一颗糖
- 往最上面放入一颗糖
- 看看最上面的糖（但不吃）
- 一共有几颗糖
- 是不是全部吃完了

# 2-2 堆栈 Stack

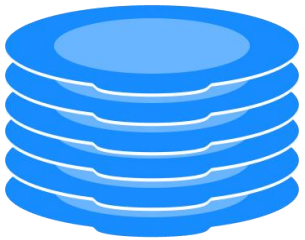


糖果堆栈具有哪些操作?

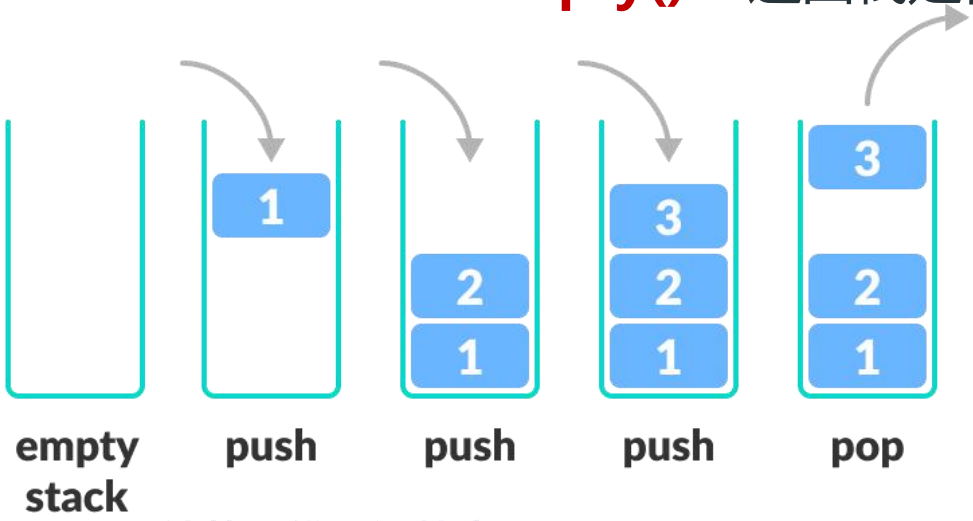
- 从最上面取出一颗糖
- 往最上面放入一颗糖
- 看看最上面的糖（但不吃）
- 一共有几颗糖
- 是不是全部吃完了

堆栈的操作:

- **pop()** – 弹出栈顶的数据元素（删除该元素）
- **push(a)** – 将数据元素a添加到栈的顶端
- **peek()** – 返回栈顶的元素，但不弹出
- **size()** – 返回栈的大小（数据元素的个数）
- **empty()** – 返回栈是否为空



stack




我们可以用任何编程语言（如 C、C++、Java、Python 或 C#）实现堆栈，但规范几乎相同。

# 2-2 堆栈 Stack



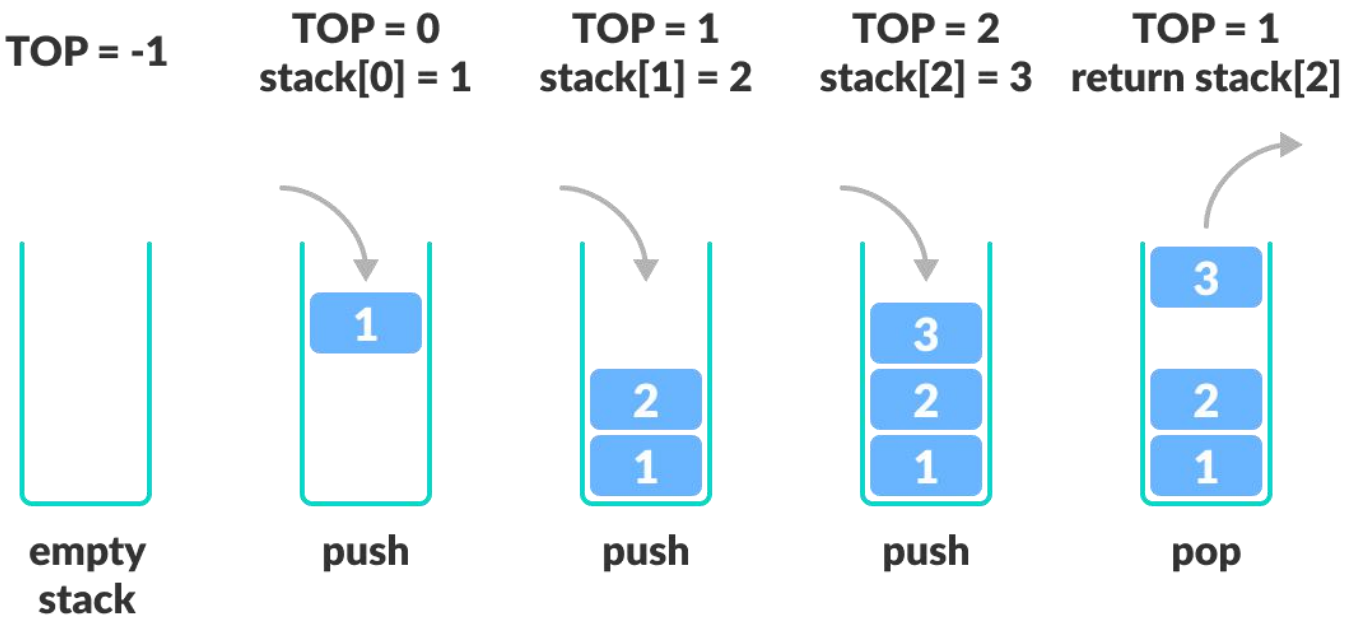
堆栈的工作原理

Why stack?

堆栈的push 和 pop 操作只需要恒定的时间，时间复杂度为O(1)，即不会随着规模的增大而增大。

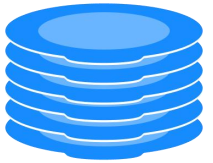
1. 一个称为 TOP 的指针用于跟踪堆栈中的顶部元素。

2. 在初始化堆栈时，我们将其值设置为-1，以便我们可以通过比较 TOP == -1 来检查堆栈是否为空。




3. 在 push 一个元素时，我们增加 TOP 的值并将新元素放置在 TOP 指向的位置。

4. 在pop弹出一个元素时，我们返回 TOP 指向的元素并减少它的值。在弹出之前，我们检查堆栈是否已经为空。



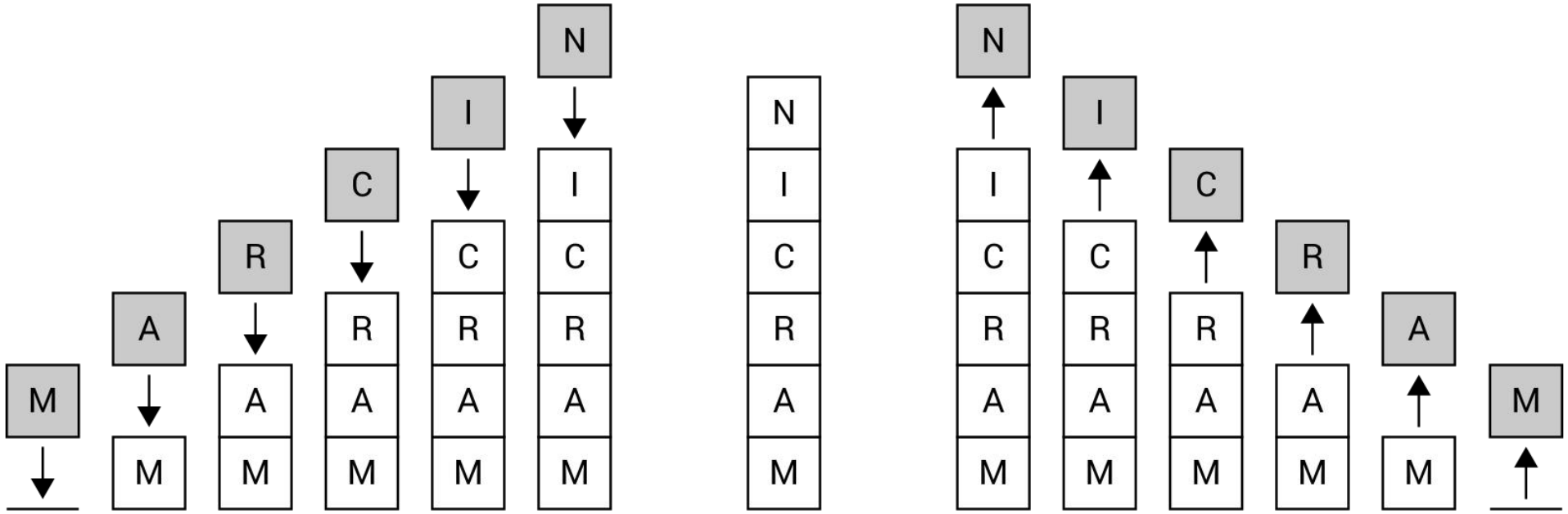
# 2-2 堆栈 Stack



栈虽然是一种实现起来很简单的数据结构，但它的功能非常强大。

堆栈最常见的用途举例：

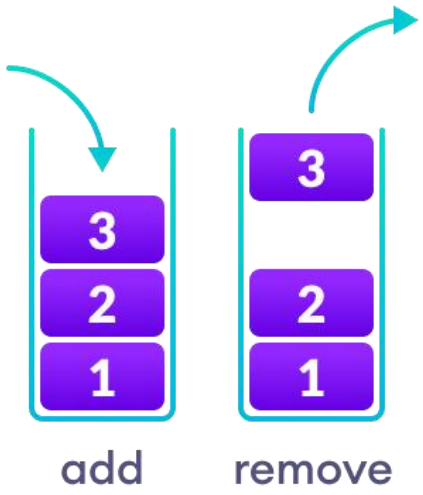
1. 反转一个单词



# 2-2 堆栈 Stack

堆栈最常见的用途举例：

1. 反转一个单词
2. 在编译器中计算后缀形式表达式的值




Infix Expression (5 + 3) * (8 - 2)

Postfix Expression 5 3 + 8 2 - *

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

| Reading Symbol | Stack Operations | Evaluated Part of Expression |
|---|---|---|
| Initially | Stack is Empty | Nothing |
| 5 | push(5) | Nothing |
| 3 | push(3) | Nothing |
| + | value1 = pop()<br>value2 = pop()<br>result = value2 + value1<br>push(result) | value1 = pop(); // 3<br>value2 = pop(); // 5<br>result = 5 + 3; // 8<br>Push( 8 )<br>(5 + 3) |
| 8 | push(8) | (5 + 3) |
| 2 | push(2) | (5 + 3) |
| - | value1 = pop()<br>value2 = pop()<br>result = value2 - value1<br>push(result) | value1 = pop(); // 2<br>value2 = pop(); // 8<br>result = 8 - 2; // 6<br>Push( 6 )<br>(8 - 2)<br>(5 + 3) , (8 - 2) |
| * | value1 = pop()<br>value2 = pop()<br>result = value2 * value1<br>push(result) | value1 = pop(); // 6<br>value2 = pop(); // 8<br>result = 8 * 6; // 48<br>Push( 48 )<br>(6 * 8)<br>(5 + 3) * (8 - 2) |
| $ End of Expression | result = pop() | Display (result)<br>48<br>As final result |

Infix Expression (5 + 3) * (8 - 2) = 48

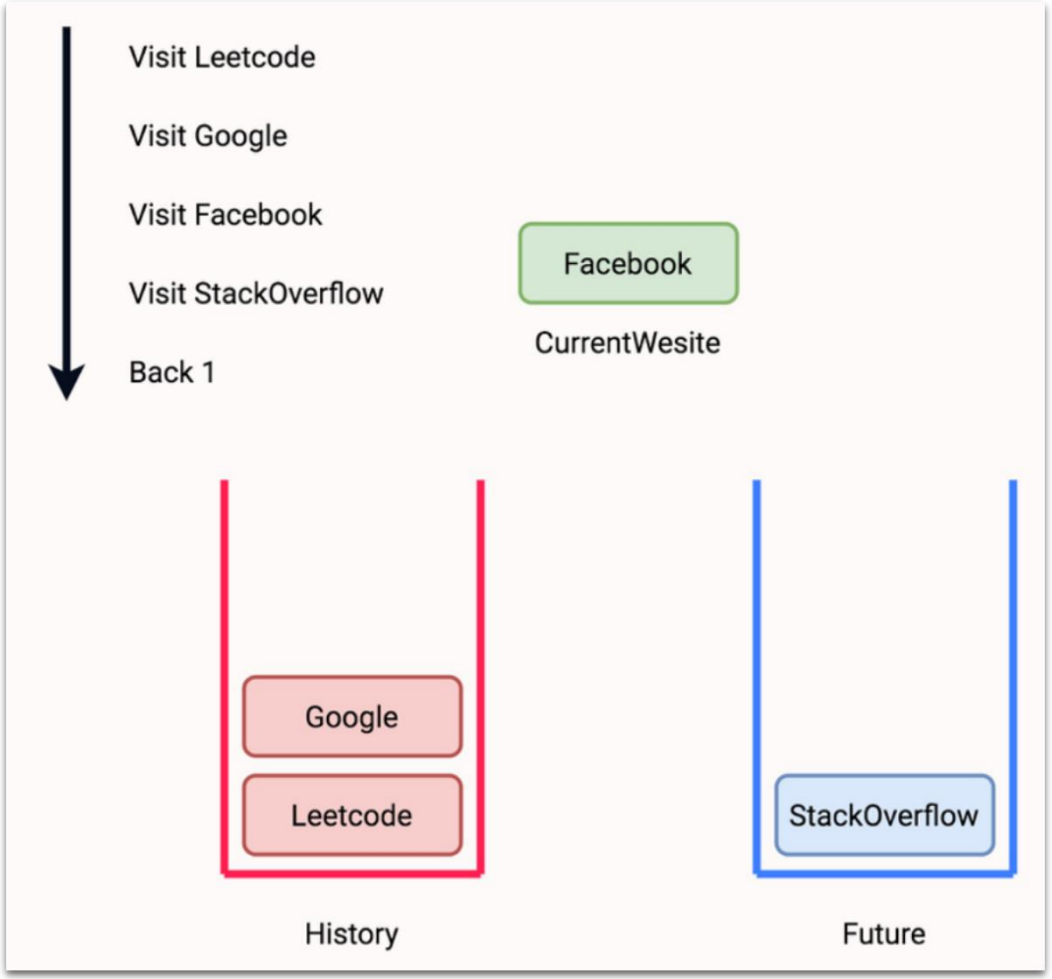
Postfix Expression 5 3 + 8 2 - * value is 48

# 2-2 堆栈 Stack



堆栈最常见的用途举例：

1. 反转一个单词
2. 在编译器中计算后缀形式表达式的值
3. 浏览器的后退按钮将之前访问过的所有 URL 保存在堆栈中。每次访问新页面时，它都会添加到堆栈顶部。当按下后退按钮时，当前 URL 将从堆栈中删除，并访问前一个 URL。




图源：https://leetcode.com/problems/design-browser-history/discuss/674486/two-stacks-pretty-code

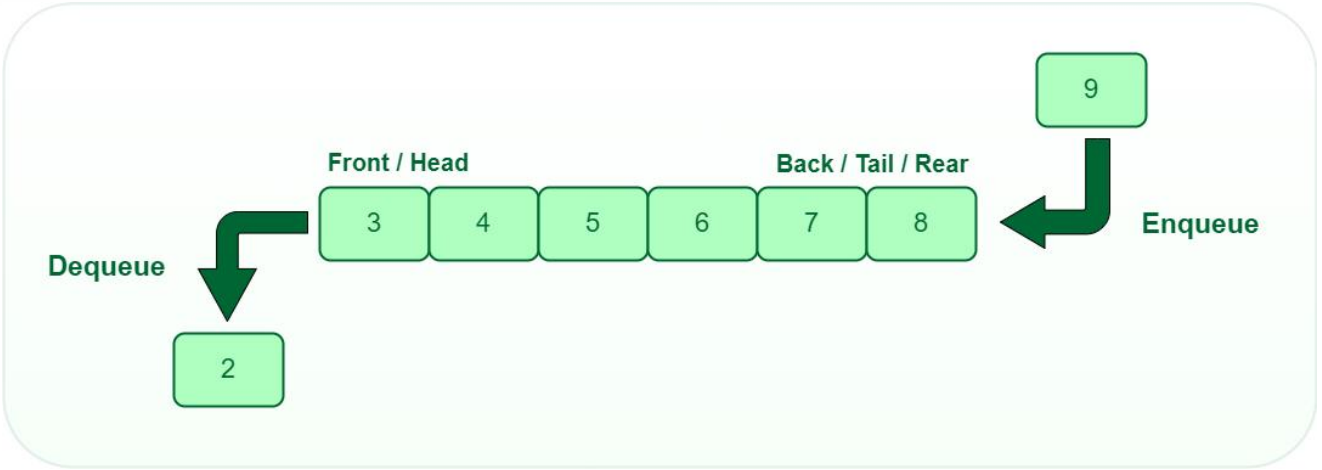
# 2-3 队列 Queue



队列是编程中非常有用的数据结构。

队列遵循先进先出 (FIFO，First In First Out) 规则。

这类似于电影售票队列，第一个进入队列的人就是第一个拿到票的人。





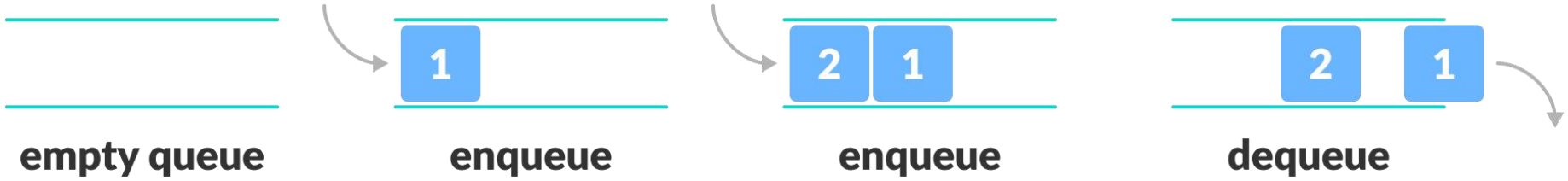
将数据元素放入队列称为入队（enqueue）

从队列中删除元素称为出队（dequeue）

我们可以用任何编程语言（如 C、C++、Java、Python 或 C#）实现队列，但规范几乎相同。

# 2-3 队列 Queue






队列的基本操作：

- **enqueue(a)** – 添加元素到队列尾部
- **dequeue()** – 将队列前端的元素删除并返回它的值
- **peek()** – 返回队列前端元素的值，但不删除它
- **size()** – 返回队列的大小（数据元素的个数）
- **empty()** – 返回队列是否为空

# 2-3 队列 Queue



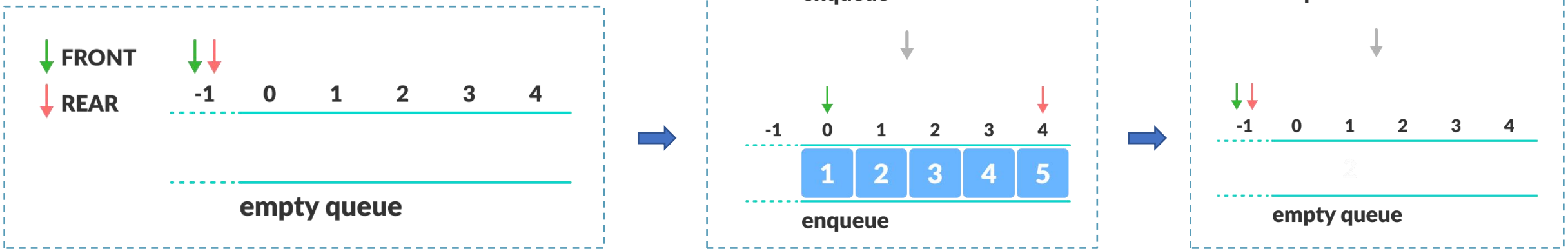
## 队列的工作原理

两个指针 FRONT 和 REAR

FRONT 指向队列的第一个元素

REAR 指向队列的最后一个元素

最初，将 FRONT 和 REAR 的值设置为 -1

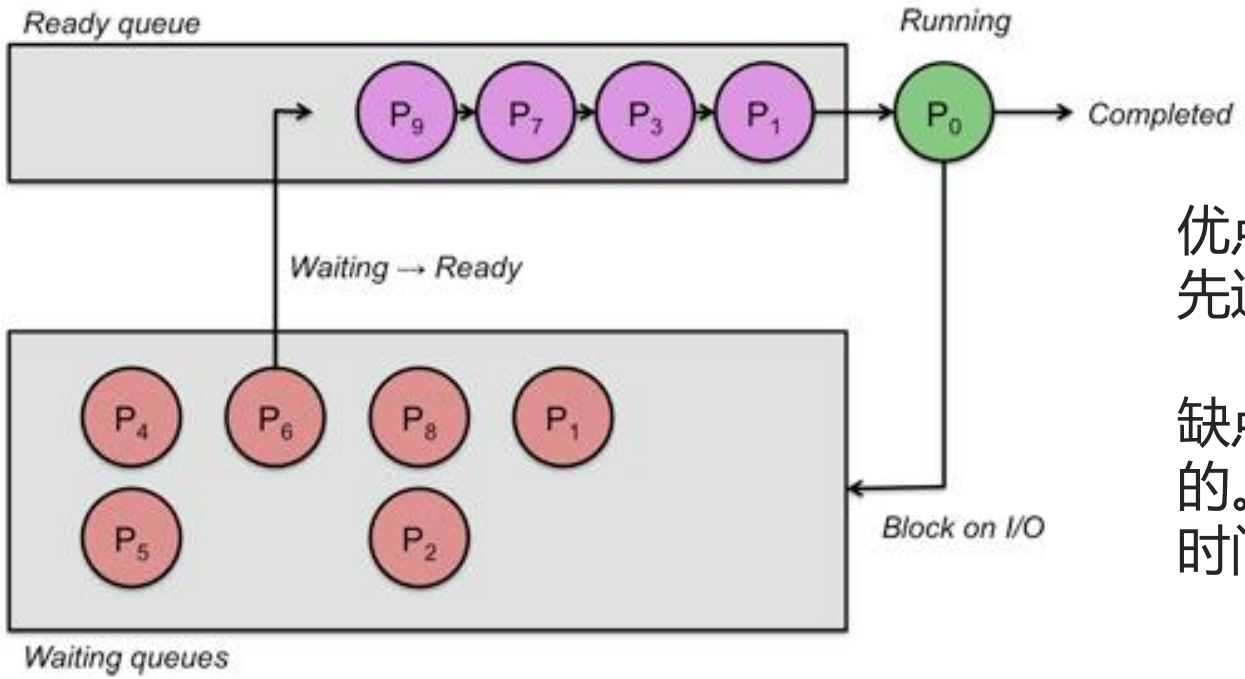


# 2-3 队列 Queue



## 队列的应用

当一个资源在多个消费者之间共享时，例如：CPU 调度、磁盘调度。




优点：FIFO调度实现简单、公平（排队的第一个先运行）。

缺点：先来先服务调度的最大缺点是它不是抢占式的。因此，它不适合交互式作业。另一个缺点是长时间运行的进程会延迟它后面的所有作业。

最简单的CPU调度方式：先来先服务（First-Come, First-Served）

# 2-3 队列 Queue

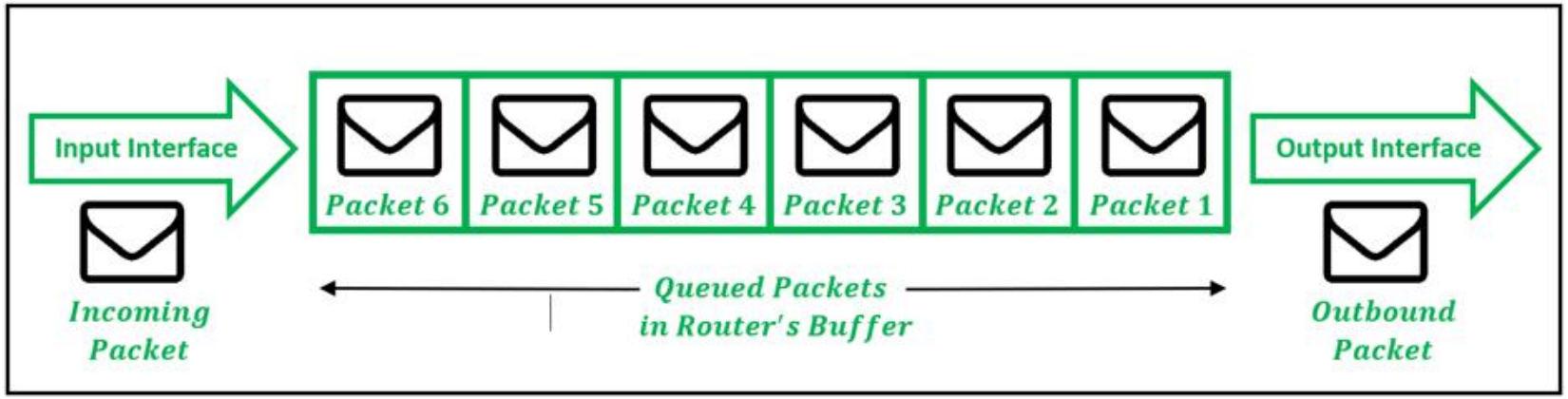


## 队列的应用

异步传输数据时（数据不一定以与发送相同的速率接收），例如：网络设备队列、IO缓冲等。

路由器有多个输入和输出接口，分别接收和发送数据包。

这些接口连接着不同速率的网络，使用队列来缓存一个个待处理的数据包。




大多数路由器遵循的默认排队方案是 FIFO。FIFO队列中的所有数据包都按照它们到达路由器的顺序进行服务。当内存达到饱和时，尝试进入路由器的新数据包将被丢弃（尾部丢弃）。

# 2-4 链表 Linked List



链表（Linked List）是包含一系列**连接节点**的线性数据结构。

**特性**：其各个数据项在计算机内存中的位置是不连续且随机（Random）存放的，其优点是数据的插入或删除都相当方便。

当有新数据加入链表后，就向系统申请一块内存空间，而在数据被删除后，就把这块内存空间还给系统，在链表中添加和删除数据都不需要移动大量的数据。

设计数据结构时较为麻烦，另外在查找数据时，也无法像静态数据（如数组）那样可以随机读取数据，必须按序查找到该数据为止。

在日常生活中有许多链表抽象概念的运用，例如可以把链表想象成火车，有多少人就挂多少节车厢，当假日人多、需要较多车厢时就多挂些车厢，平日里人少时就把车厢的数量减少，这种做法非常有弹性。

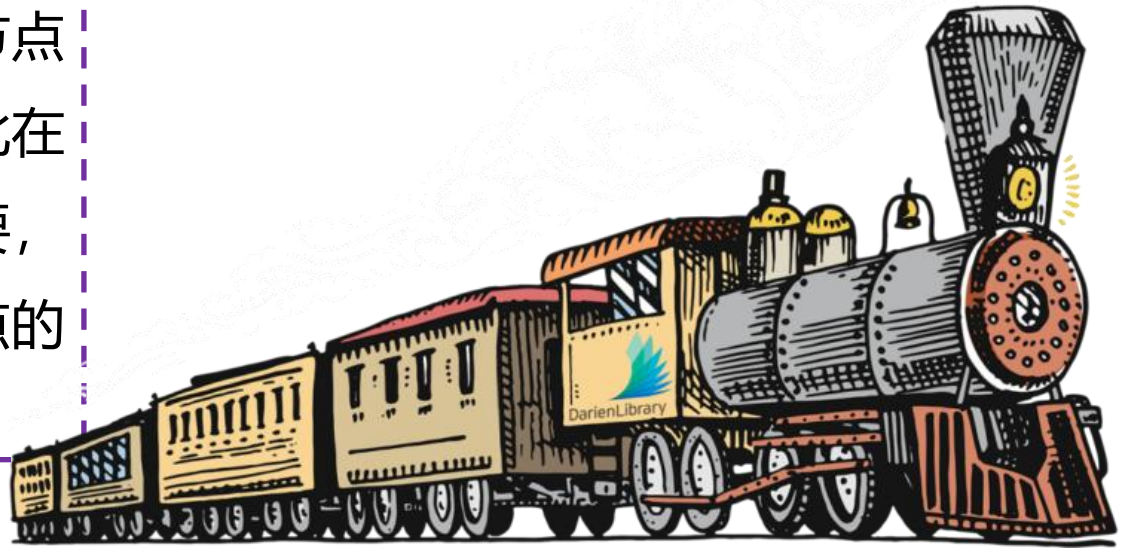
# 2-4 链表 Linked List



**单向链表** 我们最常使用的是“单向链表”（Single Linked List）。一个单向链表节点基本上由两个元素组成，即数据字段和指针，其中的指针指向链表中下一个节点在内存中的地址。


第一个节点是“头指针（Head）”，最后一个节点的指针设为NULL，表示它是“链表尾”，不指向任何地方。

由于单向链表中所有节点都知道节点本身的下一个节点在哪里，但是对于前一个节点却没有办法知道，因此在单向链表的各种操作中，“**头指针**”就显得相当重要，只要存在头指针就可以遍历整个链表，进行链表节点的加入和删除等操作。



# 2-4 链表 Linked List



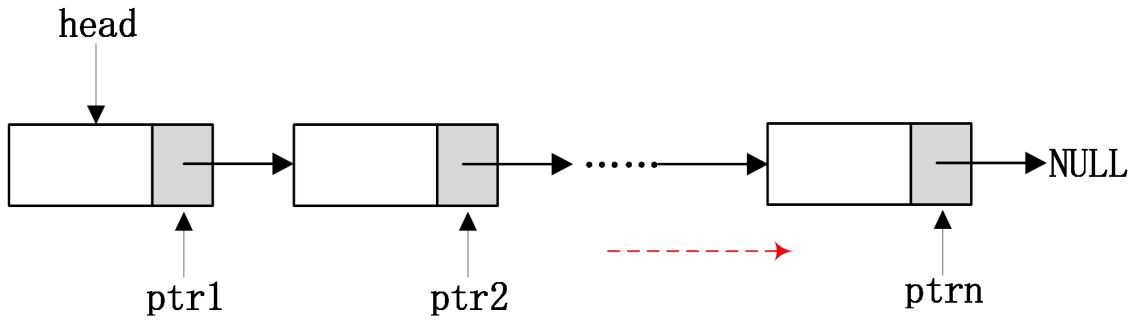
## 单向链表的遍历


### 什么是遍历?

- 遍历就是把单链表中的各个节点挨个拿出来，就叫遍历。
- 遍历的要点：一是不能遗漏、二是不能重复、追求效率。

### 遍历的方法?

从头指针+头节点开始，顺着链表挂接指针依次访问链表的各个节点，取出这个节点的数据，然后再往下一个节点，直到最后一个节点，结束返回。




### 单链表的特点:

- Head+头节点为整个链表的起始，最后一个节点内部的指针值为NULL。
- 从起始到结尾中间由各个节点内部的指针来挂接。
- 由起始到结尾的路径有且只有一条。

单链表的这些特点就决定了它的遍历算法。

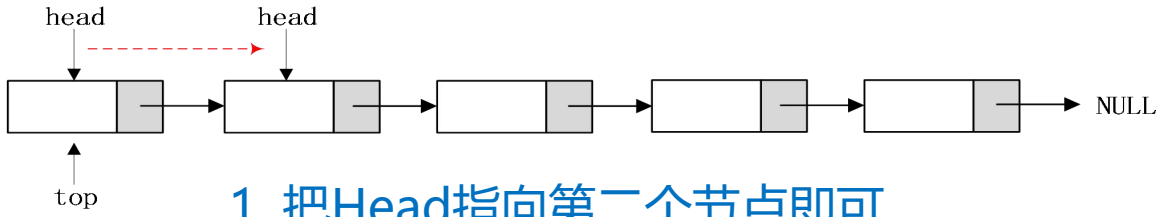
# 2-4 链表 Linked List



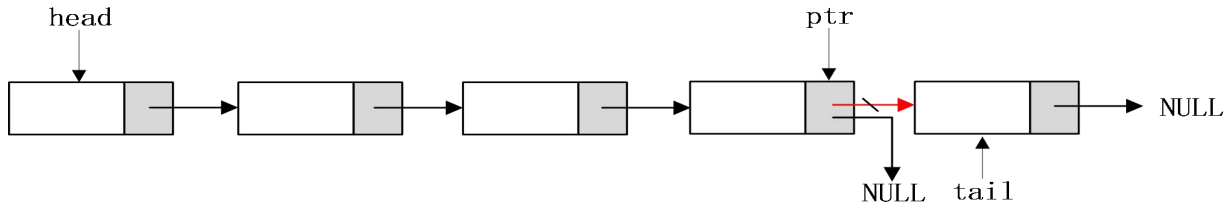
## 单向链表节点的删除算法

在单向链表类型的数据结构中，如果要在链表中删除一个节点，根据所删除节点的位置会有三种不同的情况：

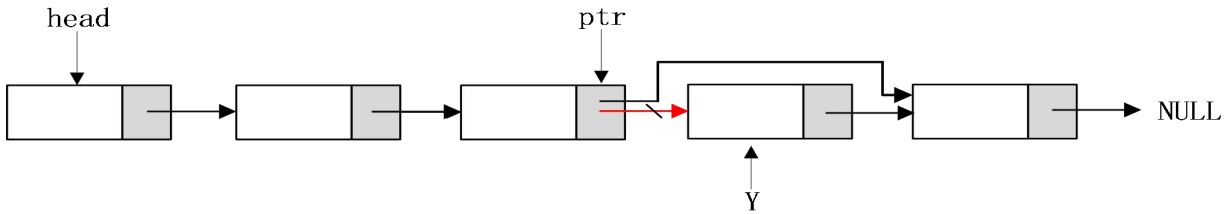
1. 删除链表的第一个节点
2. 删除链表的最后一个节点
3. 删除链表的中间的节点




1. 把Head指向第二个节点即可




2. 将指向最后一个节点的指针ptr直接指向NULL（空节点）即可




3. 将删除节点的前一个节点的指针指向要删除节点的下一个节点即可

# 2-4 链表 Linked List

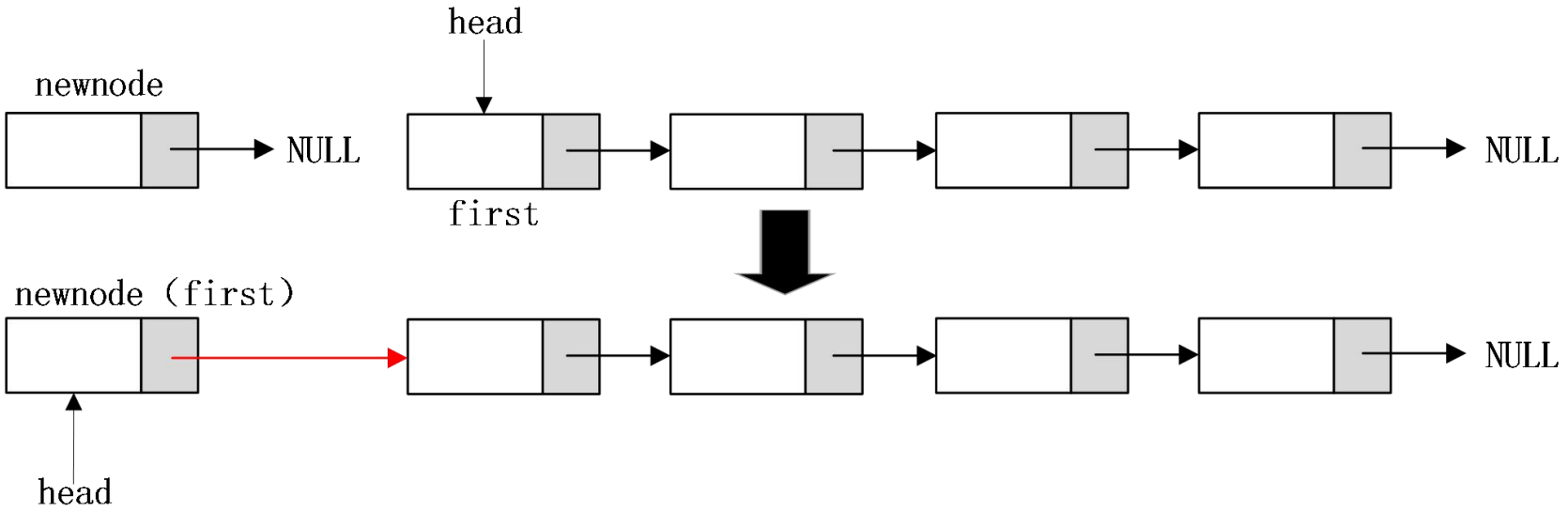


## 单向链表节点的添加算法

在单向链表中添加新节点，有三种情况：

1. 加到第一个节点之前
2. 加到最后一个节点之后
3. 加到此链表中间任一位置

1. 只需把新节点的指针指向链表原来的第一个节点，再把链表头指针指向新节点即可



# 2-4 链表 Linked List

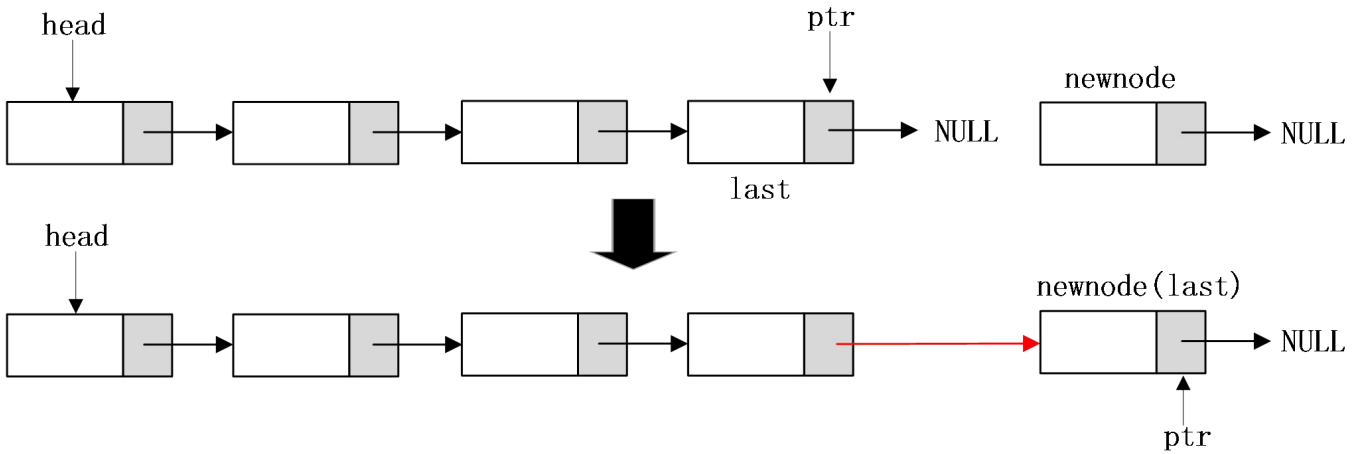


## 单向链表节点的添加算法

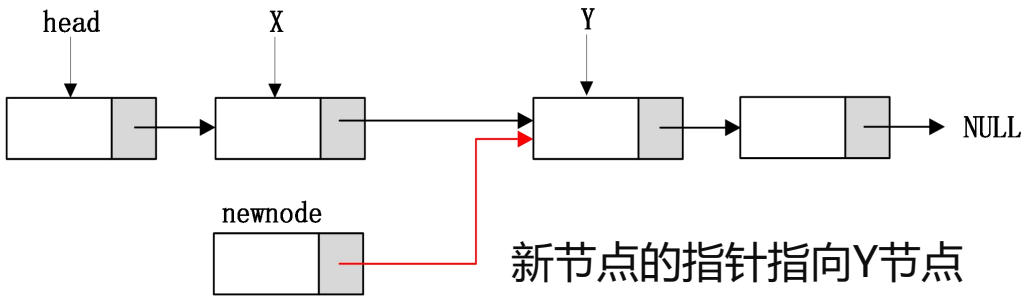
在单向链表中添加新节点，有三种情况：

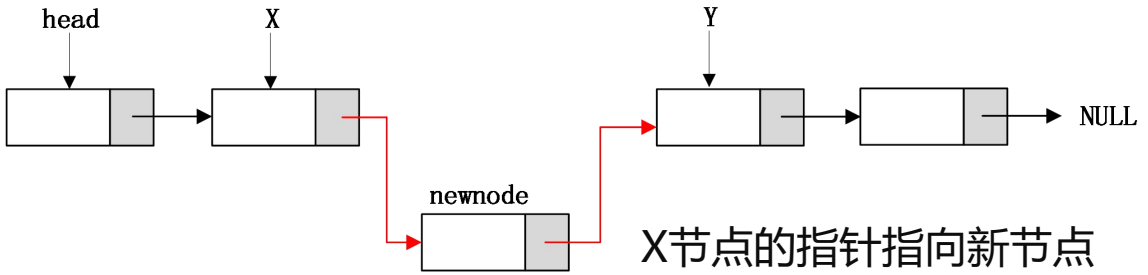
1. 加到第一个节点之前
2. 加到最后一个节点之后
3. 加到此链表中间任一位置

2. 把最后一个节点的指针指向新节点，新节点的指针指向NULL




3. 假设要插入的节点在X与Y之间，只要将X节点的指针指向新节点，新节点的指针指向Y节点



# 本次课程结束，请继续加油！

Computational Thinking

计算机与网络空间安全学院
School of Computer and Cyber Sciences