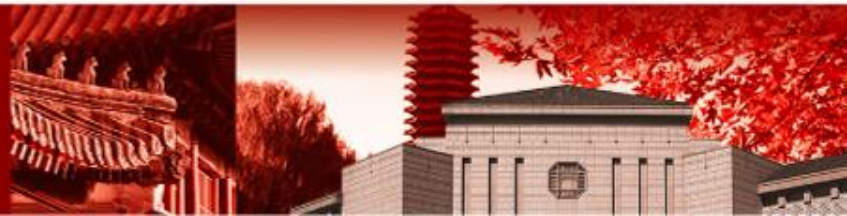
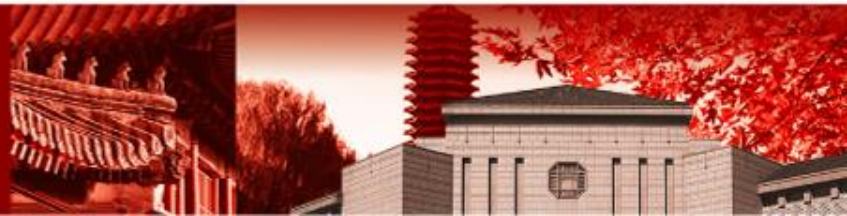
数据结构与算法B

04 – 栈

目录



4.1 栈的定义与操作

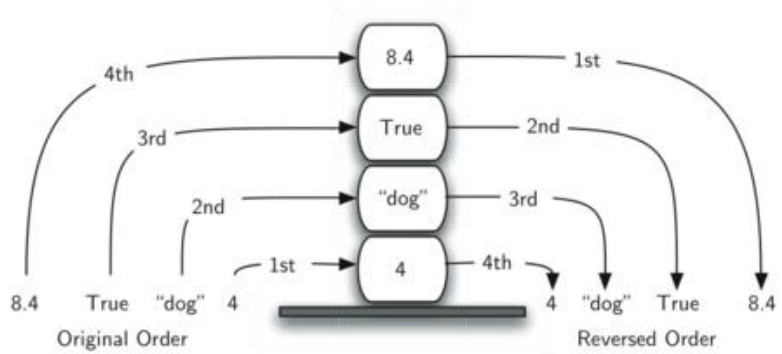


栈的定义

- 栈(Stack)是一种有次序的数据项集合，在栈中，数据项的加入和移除都仅发生在同一端
  - 这一端通常叫做栈“顶top”，另一端就叫做栈“底base”
- 这种次序通常称为“后进先出LIFO”：Last in First out
  - 这是一种基于数据项保存时间的次序
  - 时间越短的离栈顶越近，会最先被删除
  - 时间越长的离栈底越近，留在栈中的时间越长
- 日常生活中有很多栈的应用
  - 盘子、托盘、书堆等等

栈的特性：反转次序

- 我们观察一个由混合的python原生数据对象形成的栈
  - 进栈和出栈的次序正好相反

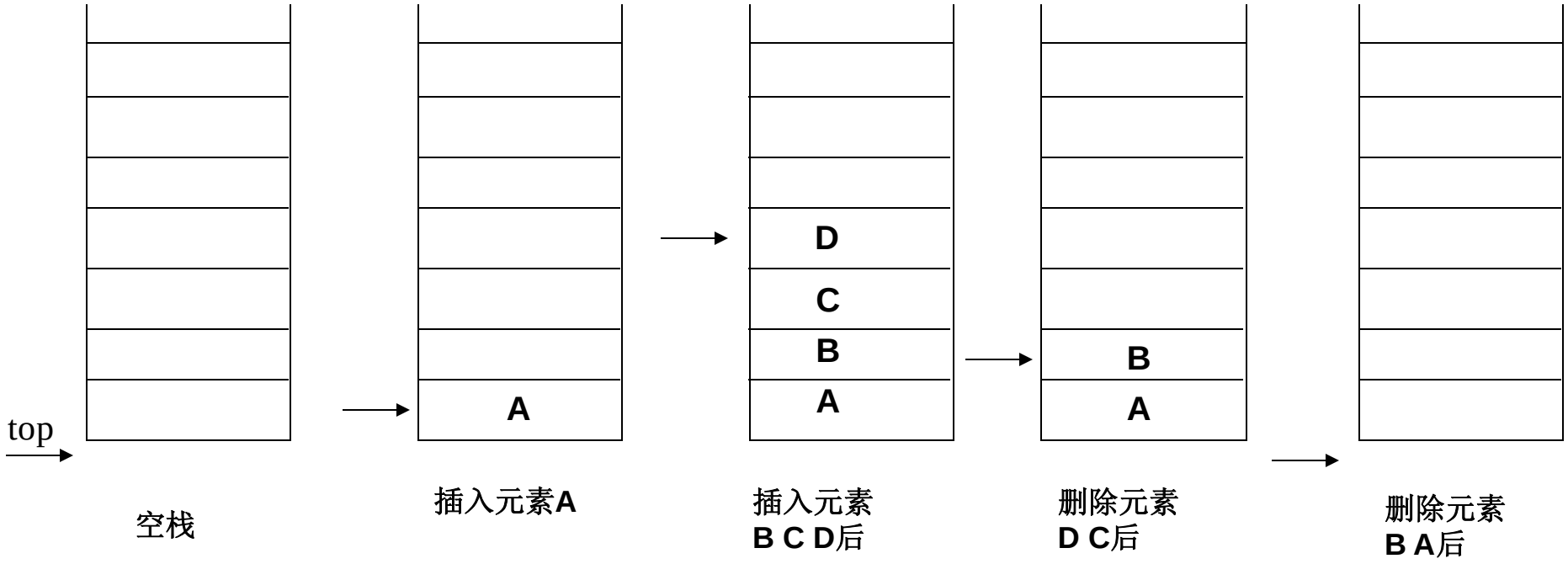


- 这种访问次序反转的特性，我们在某些计算机操作上碰到过
  - 浏览器的“后退back”按钮，最先back的是最近访问的网页
  - Word的“Undo”按钮，最先撤销的是最近的操作

栈的抽象数据类型

- 抽象数据类型“栈”是一个有次序的数据集，每个数据项仅从“栈顶”一端加入到数据集中、从数据集中移除，栈具有后进先出LIFO的特性
- 栈是一种受限的线性结构，即仅允许在一端操作
- 抽象数据类型“栈”定义为如下的操作
  - Stack()：创建一个空栈，其中不包含任何数据项
  - push(item)：将item数据项加入栈顶，无返回值
  - pop()：将栈顶数据项移除，返回栈顶的数据项，栈被修改
  - peek()：“窥视”栈顶数据项，返回栈顶的数据项但不移除，栈不被修改
  - isEmpty()：返回栈是否为空栈
  - size()：返回栈中有多少个数据项

栈的插入、删除



栈的实现方式

- 在清楚地定义了抽象数据类型Stack之后，我们看看如何用Python来实现它
- Python的面向对象机制，可以用来实现用户自定义类型
  - 将ADT Stack实现为Python的一个Class
  - 将ADT Stack的操作实现为Class的方法
  - 由于Stack是一个数据集，所以可以采用Python的原生数据集来实现，我们选用最常用的数据集List来实现
- 一个细节：Stack的两端设置，栈顶和栈底
  - 可以将List的任意一端（index=0或者-1）设置为栈顶
  - 通常，我们选用List的末端（index=-1）作为栈顶
    - 这样栈的操作就可以通过对list的append和pop来实现

用Python实现ADT Stack

```
class Stack:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[len(self.items)-1]

    def size(self):
        return len(self.items)
```

创建新栈

检查栈是否为空

向栈添加元素item

利用list的pop方法，删除并返回表尾元素

返回栈顶的数据项但不移除

获取栈中元素数量

ADT Stack的另一个实现

- 如果我们把List的另一端（首端index=0）作为Stack的栈顶，同样也可以实现Stack（下左，右为栈顶设定末端的实现）
- 不同的实现方案保持了ADT接口的稳定性
  - 但性能有所不同，栈顶首端的版本（左），其push/pop的复杂度为O(n)，而栈顶尾端的实现（右），其push/pop的复杂度为O(1)

```
class Stack:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def push(self, item):
        self.items.insert(0,item)

    def pop(self):
        return self.items.pop(0)

    def peek(self):
        return self.items[0]

    def size(self):
        return len(self.items)
```

```
class Stack:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[len(self.items)-1]

    def size(self):
        return len(self.items)
```

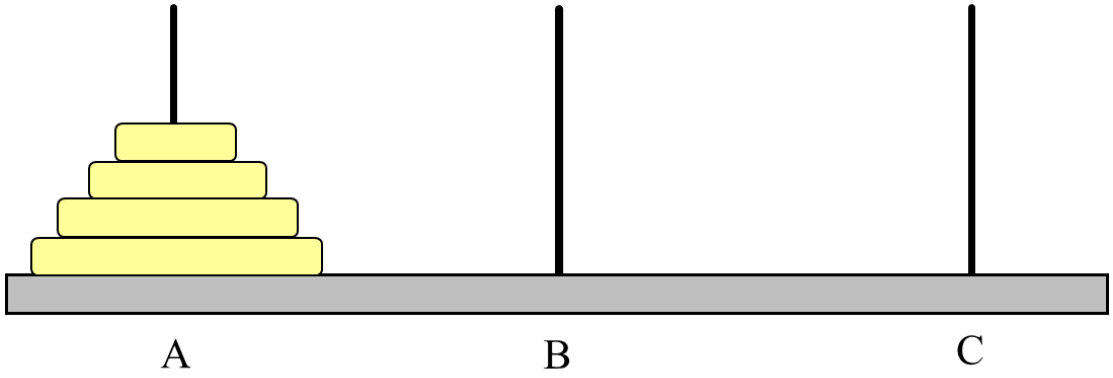
4.2 栈与递归

什么是递归(Recursion)

- 递归是一种解决问题的方法，其精髓是将问题分解为规模更小的相同问题，持续分解，直到问题规模小到可以用非常简单直接的方式来解决。
- 递归的问题分解方式非常独特，其算法方面的明显特征就是：在算法流程中调用自身。
- 递归为我们提供了一种对复杂问题的优雅解决方案，精妙的递归算法常会出奇简单，令人赞叹。

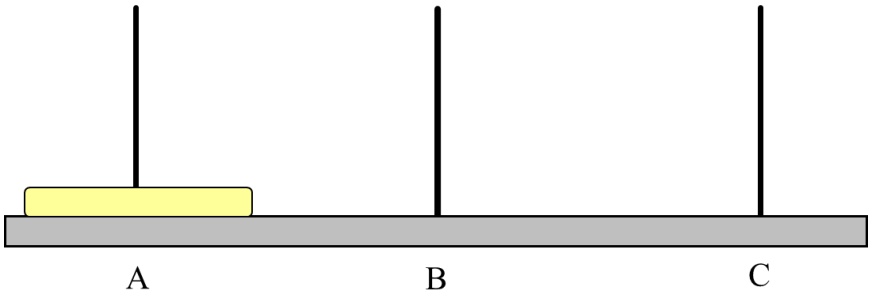
初识递归：汉诺塔

- 汉诺塔：在一个平板上立有三根柱子，分别记为A, B, C。
  - 开始时，柱子 A 上依次叠放着从大到小 n 个圆盘
  - 游戏的目标就是将 A 上的 n 个圆盘全部转移到 C 上
  - 要求每次只能移动某根柱子最上层一个圆盘，且只能将小的圆盘叠放在大的圆盘之上。
- 输出完整的操作序列

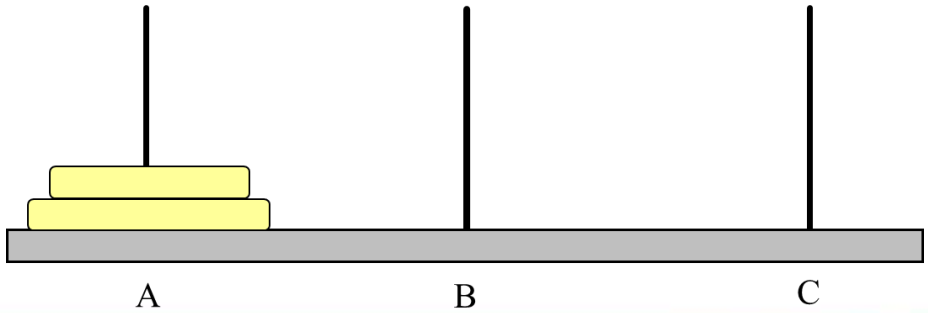


初识递归：汉诺塔

- 先来看简单的情形：n = 1
  - 直接将其移动到C上即可：A – C

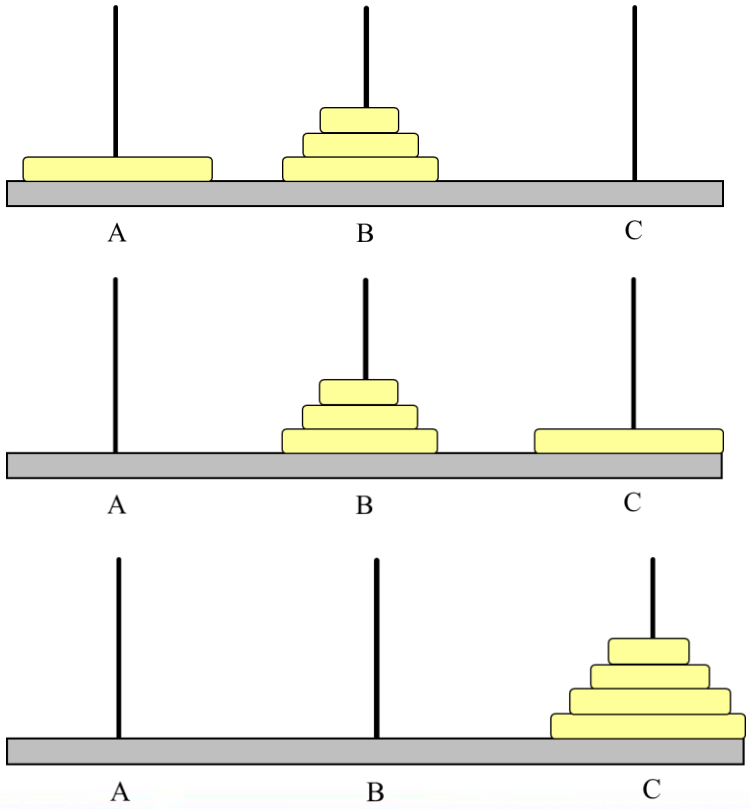


- n = 2：
  - 借助B中转，A—B , A—C. B—C



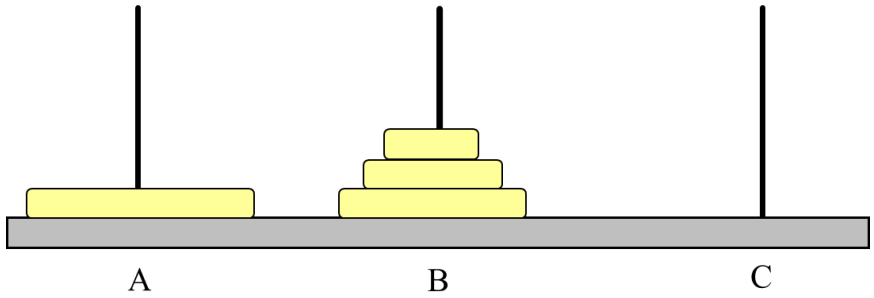
初识递归：汉诺塔

- 对于更较大的n，实际上我们也可以把解决问题的过程分为几步：
  - 将n-1个小的圆盘移动到空闲的柱子上
  - 将最大的圆盘移动到目标柱子上
  - 将n-1个小的圆盘移动到目标柱子上



初识递归：汉诺塔

- 尽管还不知道，如何能够将n-1个小的圆盘从一个柱子上移动到另一个柱子上，但实际上问题已经被递归地解决了：
  - 移动n-1个圆盘，是一个与原问题形式相同，但规模减小的问题
    - 尽管一个柱子上存在着第n个最大的圆盘，但这丝毫不会影响对n-1个较小圆盘的操作。因此，这个问题可以称与原问题形式相同
  - 因此还可以使用“三步走”的思想：先移动n-2个圆盘，再移动最大的圆盘，再把n-2个圆盘移到最大的圆盘上。
  - 如此递归，直到n=1（边界情况），就可以直接完成



初识递归：汉诺塔

- 上面的递归算法变成程序

```
def hanoi(n, start, by, to):
    if n == 1:
        print("%s -- %s" % (start, to))
        return
    else:
        hanoi(n-1, start, to, by)
        print("%s -- %s" % (start, to))
        hanoi(n-1, by, start, to)
```

- 上面程序的要点：
  - 1. 问题分解为更小规模的相同问题，并表现为“调用自身”
  - 2. 对“最小规模”问题的解决：简单直接

递归“三定律”

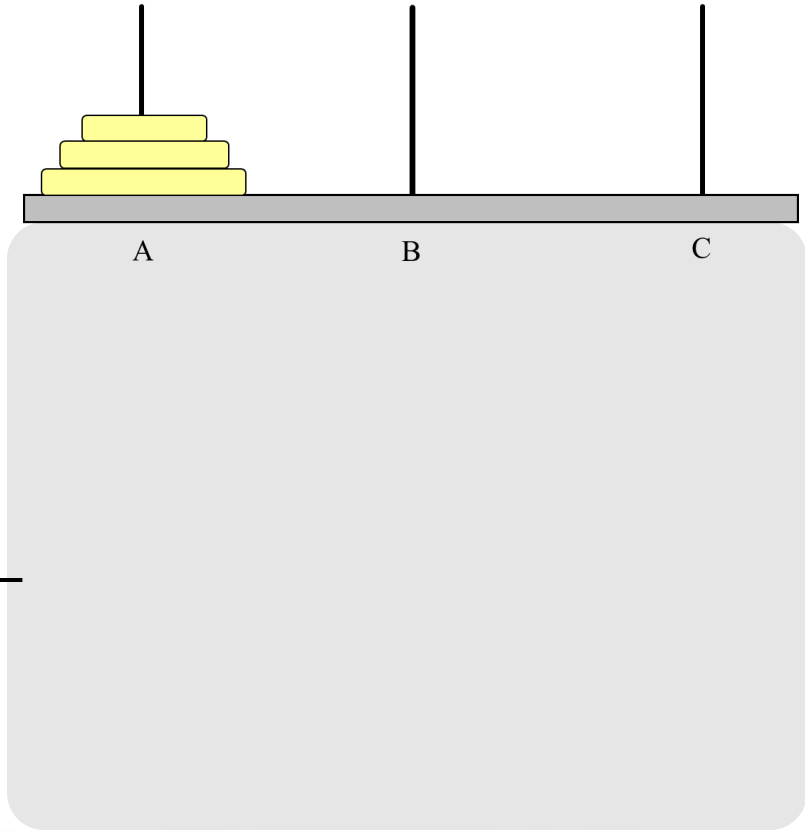
- 1. 递归算法必须有一个基本结束条件
  - 递归出口：最小规模问题的直接解决
- 2. 递归算法必须向基本结束条件演进
  - 减小问题规模
- 3. 递归算法必须调用自身
  - 关键：将问题分解成了规模更小的相同问题，通过解决更小规模的问题来解决原问题

递归程序是如何执行的

- 递归函数调用和返回过程的链条就是用栈来管理的。
- 例：Hanoi(3, ‘A’, ‘B’, ‘C’)

```
def hanoi(n, start, by, to):
    if n == 1:
        print(“%s -- %s” % (start, to))
        return
    else:
        hanoi(n-1, start, to, by)
        print(“%s -- %s” % (start, to))
        hanoi(n-1, by, start, to)
```

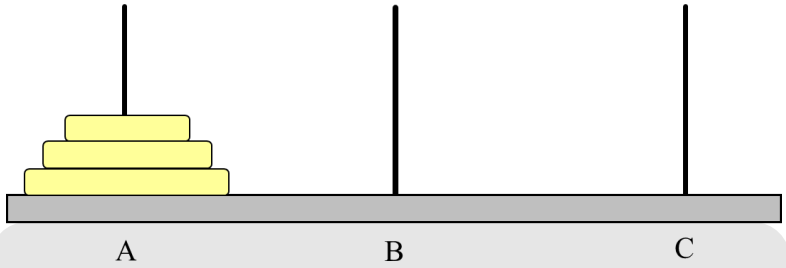
运行时内存（栈）



递归程序是如何执行的

- 递归函数调用和返回过程的链条就是用栈来管理的。
- 例：Hanoi(3, ‘A’, ‘B’, ‘C’)

```
def hanoi(n, start, by, to):
    if n == 1:
        print(“%s -- %s” % (start, to))
        return
    else:
        hanoi(n-1, start, to, by)
        print(“%s -- %s” % (start, to))
        hanoi(n-1, by, start, to)
```



运行时内存（栈）

hanoi(3, ‘A’, ‘B’, ‘C’)

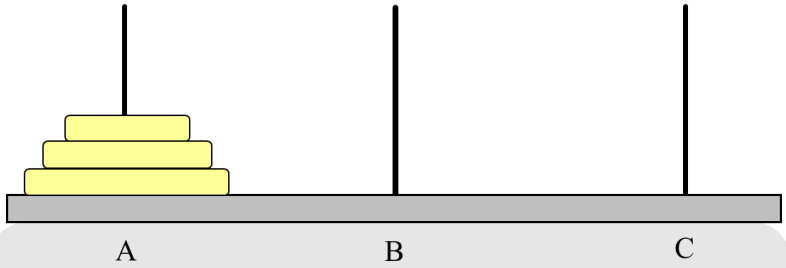
函数栈帧（Frame）

存储局部变量、返回值、内存基址等

递归程序是如何执行的

- 递归函数调用和返回过程的链条就是用栈来管理的。
- 例：Hanoi(3, ‘A’, ‘B’, ‘C’)

```
def hanoi(n, start, by, to):
    if n == 1:
        print(“%s -- %s” % (start, to))
        return
    else:
        hanoi(n-1, start, to, by)
        print(“%s -- %s” % (start, to))
        hanoi(n-1, by, start, to)
```



运行时内存（栈）

hanoi(2, ‘A’, ‘C’, ‘B’)

存储局部变量、返回值、内存基址等

函数栈帧（Frame）

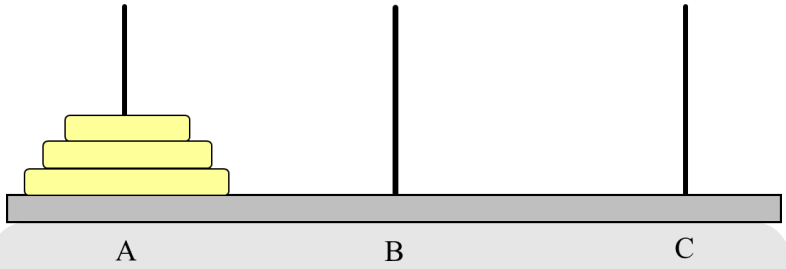
hanoi(3, ‘A’, ‘B’, ‘C’)

存储局部变量、返回值、内存基址等

递归程序是如何执行的

- 递归函数调用和返回过程的链条就是用栈来管理的。
- 例：Hanoi(3, ‘A’, ‘B’, ‘C’)

```
def hanoi(n, start, by, to):
    if n == 1:
        print("%s -- %s" % (start, to))
        return
    else:
        hanoi(n-1, start, to, by)
        print("%s -- %s" % (start, to))
        hanoi(n-1, by, start, to)
```



运行时内存（栈）

函数栈帧（Frame）

hanoi(1, ‘A’, ‘B’, ‘C’)
存储局部变量、返回值、内存基址等

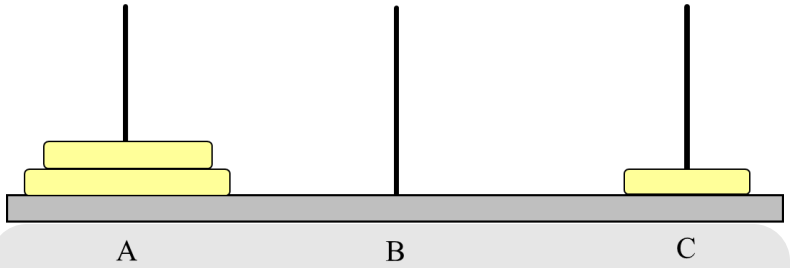
hanoi(2, ‘A’, ‘C’, ‘B’)
存储局部变量、返回值、内存基址等

hanoi(3, ‘A’, ‘B’, ‘C’)
存储局部变量、返回值、内存基址等

递归程序是如何执行的

- 递归函数调用和返回过程的链条就是用栈来管理的。
- 例：Hanoi(3, ‘A’, ‘B’, ‘C’)

```
def hanoi(n, start, by, to):
    if n == 1:
        print(“%s -- %s” % (start, to))
        return
    else:
        hanoi(n-1, start, to, by)
        print(“%s -- %s” % (start, to))
        hanoi(n-1, by, start, to)
```



运行时内存（栈）

函数栈帧（Frame）

hanoi(1, ‘A’, ‘B’, ‘C’)
存储局部变量、返回值、内存基址等

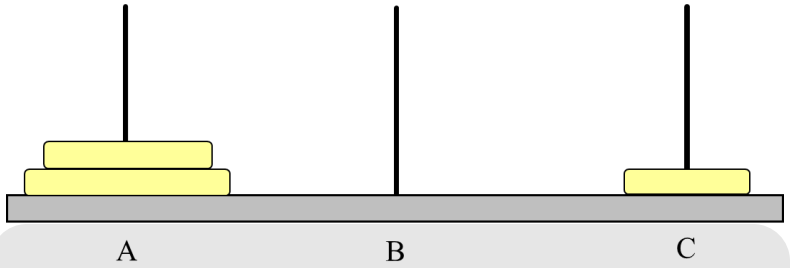
hanoi(2, ‘A’, ‘C’, ‘B’)
存储局部变量、返回值、内存基址等

hanoi(3, ‘A’, ‘B’, ‘C’)
存储局部变量、返回值、内存基址等

递归程序是如何执行的

- 递归函数调用和返回过程的链条就是用栈来管理的。
- 例：Hanoi(3, ‘A’, ‘B’, ‘C’)

```
def hanoi(n, start, by, to):
    if n == 1:
        print(“%s -- %s” % (start, to))
        return
    else:
        hanoi(n-1, start, to, by)
        print(“%s -- %s” % (start, to))
        hanoi(n-1, by, start, to)
```



运行时内存（栈）

函数栈帧（Frame）

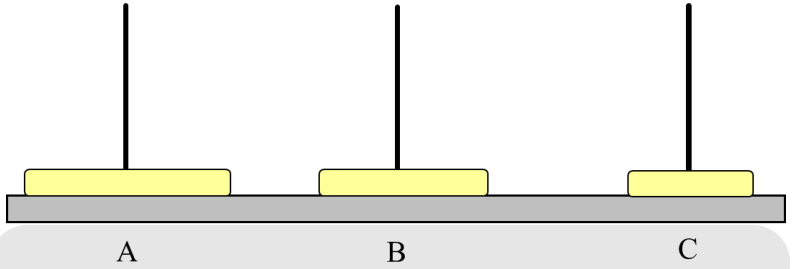
hanoi(2, ‘A’, ‘C’, ‘B’)
存储局部变量、返回值、内存基址等

hanoi(3, ‘A’, ‘B’, ‘C’)
存储局部变量、返回值、内存基址等

递归程序是如何执行的

- 递归函数调用和返回过程的链条就是用栈来管理的。
- 例：Hanoi(3, ‘A’, ‘B’, ‘C’)

```
def hanoi(n, start, by, to):
    if n == 1:
        print(“%s -- %s” % (start, to))
        return
    else:
        hanoi(n-1, start, to, by)
        print(“%s -- %s” % (start, to))
        hanoi(n-1, by, start, to)
```



运行时内存（栈）

hanoi(2, ‘A’, ‘C’, ‘B’)

存储局部变量、返回值、内存基址等

函数栈帧（Frame）

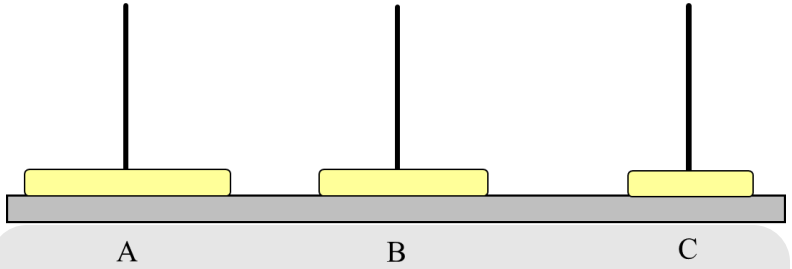
hanoi(3, ‘A’, ‘B’, ‘C’)

存储局部变量、返回值、内存基址等

递归程序是如何执行的

- 递归函数调用和返回过程的链条就是用栈来管理的。
- 例：Hanoi(3, ‘A’, ‘B’, ‘C’)

```
def hanoi(n, start, by, to):
    if n == 1:
        print(“%s -- %s” % (start, to))
        return
    else:
        hanoi(n-1, start, to, by)
        print(“%s -- %s” % (start, to))
        hanoi(n-1, by, start, to)
```



运行时内存（栈）

函数栈帧（Frame）

hanoi(1, ‘C’, ‘A’, ‘B’)
存储局部变量、返回值、内存基址等

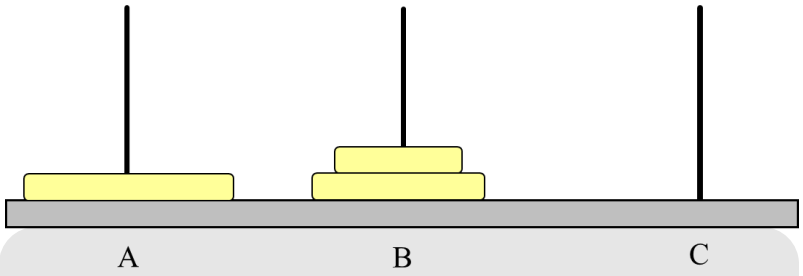
hanoi(2, ‘A’, ‘C’, ‘B’)
存储局部变量、返回值、内存基址等

hanoi(3, ‘A’, ‘B’, ‘C’)
存储局部变量、返回值、内存基址等

递归程序是如何执行的

- 递归函数调用和返回过程的链条就是用栈来管理的。
- 例：Hanoi(3, ‘A’, ‘B’, ‘C’)

```
def hanoi(n, start, by, to):
    if n == 1:
        print(“%s -- %s” % (start, to))
        return
    else:
        hanoi(n-1, start, to, by)
        print(“%s -- %s” % (start, to))
        hanoi(n-1, by, start, to)
```



运行时内存（栈）

函数栈帧（Frame）

hanoi(1, ‘C’, ‘A’, ‘B’)
存储局部变量、返回值、内存基址等

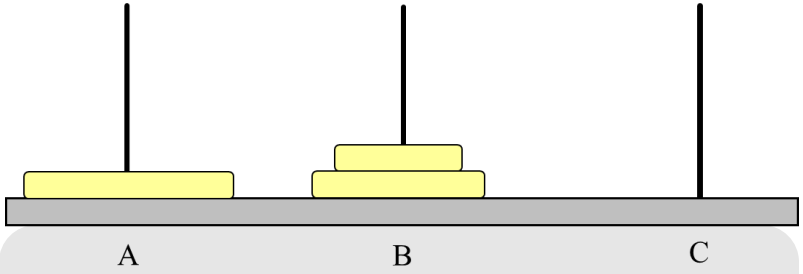
hanoi(2, ‘A’, ‘C’, ‘B’)
存储局部变量、返回值、内存基址等

hanoi(3, ‘A’, ‘B’, ‘C’)
存储局部变量、返回值、内存基址等

递归程序是如何执行的

- 递归函数调用和返回过程的链条就是用栈来管理的。
- 例：Hanoi(3, ‘A’, ‘B’, ‘C’)

```
def hanoi(n, start, by, to):
    if n == 1:
        print(“%s -- %s” % (start, to))
        return
    else:
        hanoi(n-1, start, to, by)
        print(“%s -- %s” % (start, to))
        hanoi(n-1, by, start, to)
```



运行时内存（栈）

函数栈帧（Frame）

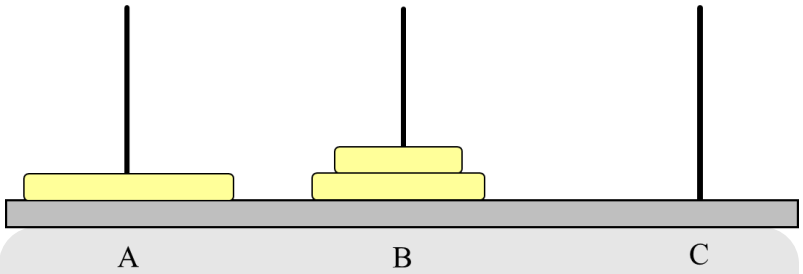
hanoi(2, ‘A’, ‘C’, ‘B’)
存储局部变量、返回值、内存基址等

hanoi(3, ‘A’, ‘B’, ‘C’)
存储局部变量、返回值、内存基址等

递归程序是如何执行的

- 递归函数调用和返回过程的链条就是用栈来管理的。
- 例：Hanoi(3, ‘A’, ‘B’, ‘C’)

```
def hanoi(n, start, by, to):
    if n == 1:
        print(“%s -- %s” % (start, to))
        return
    else:
        hanoi(n-1, start, to, by)
        print(“%s -- %s” % (start, to))
        hanoi(n-1, by, start, to)
```



运行时内存（栈）

hanoi(3, ‘A’, ‘B’, ‘C’)

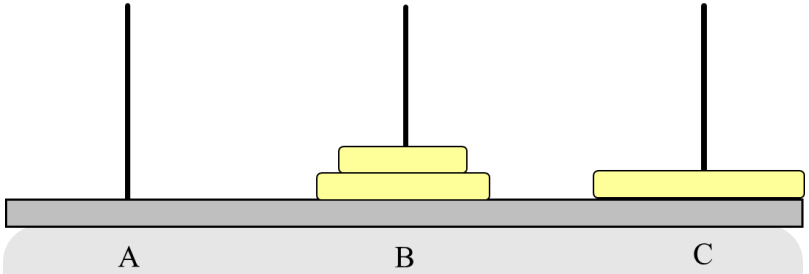
函数栈帧（Frame）

存储局部变量、返回值、内存基址等

递归程序是如何执行的

- 递归函数调用和返回过程的链条就是用栈来管理的。
- 例：Hanoi(3, ‘A’, ‘B’, ‘C’)

```
def hanoi(n, start, by, to):
    if n == 1:
        print(“%s -- %s” % (start, to))
        return
    else:
        hanoi(n-1, start, to, by)
        print(“%s -- %s” % (start, to))
        hanoi(n-1, by, start, to)
```



运行时内存（栈）

函数栈帧（Frame）

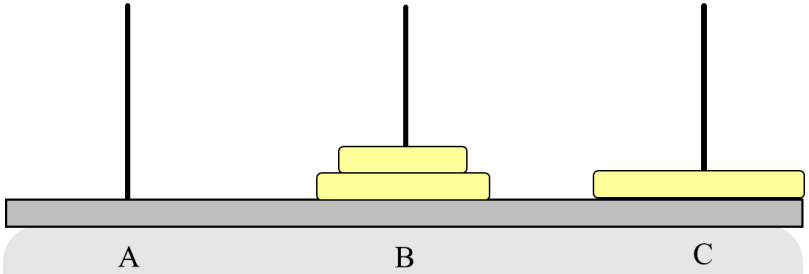
hanoi(3, ‘A’, ‘B’, ‘C’)

存储局部变量、返回值、内存基址等

递归程序是如何执行的

- 递归函数调用和返回过程的链条就是用栈来管理的。
- 例：Hanoi(3, ‘A’, ‘B’, ‘C’)

```
def hanoi(n, start, by, to):
    if n == 1:
        print(“%s -- %s” % (start, to))
        return
    else:
        hanoi(n-1, start, to, by)
        print(“%s -- %s” % (start, to))
        hanoi(n-1, by, start, to)
```



运行时内存（栈）

函数栈帧（Frame）

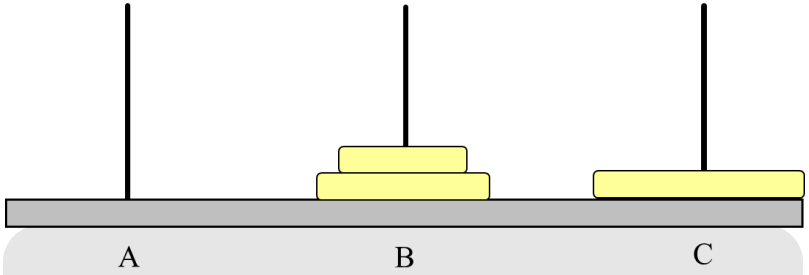
hanoi(2, ‘B’, ‘A’, ‘C’)
存储局部变量、返回值、内存基址等

hanoi(3, ‘A’, ‘B’, ‘C’)
存储局部变量、返回值、内存基址等

递归程序是如何执行的

- 递归函数调用和返回过程的链条就是用栈来管理的。
- 例：Hanoi(3, ‘A’, ‘B’, ‘C’)

```
def hanoi(n, start, by, to):
    if n == 1:
        print(“%s -- %s” % (start, to))
        return
    else:
        hanoi(n-1, start, to, by)
        print(“%s -- %s” % (start, to))
        hanoi(n-1, by, start, to)
```



运行时内存（栈）

函数栈帧（Frame）

hanoi(1, ‘B’, ‘C’, ‘A’)
存储局部变量、返回值、内存基址等

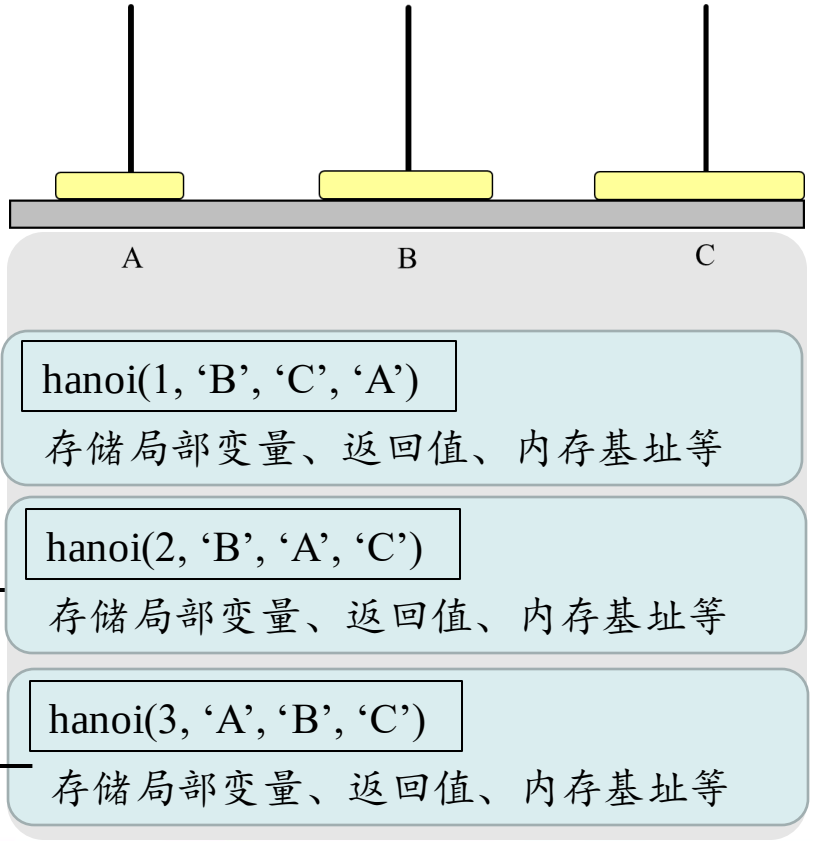
hanoi(2, ‘B’, ‘A’, ‘C’)
存储局部变量、返回值、内存基址等

hanoi(3, ‘A’, ‘B’, ‘C’)
存储局部变量、返回值、内存基址等

递归程序是如何执行的

- 递归函数调用和返回过程的链条就是用栈来管理的。
- 例：Hanoi(3, ‘A’, ‘B’, ‘C’)

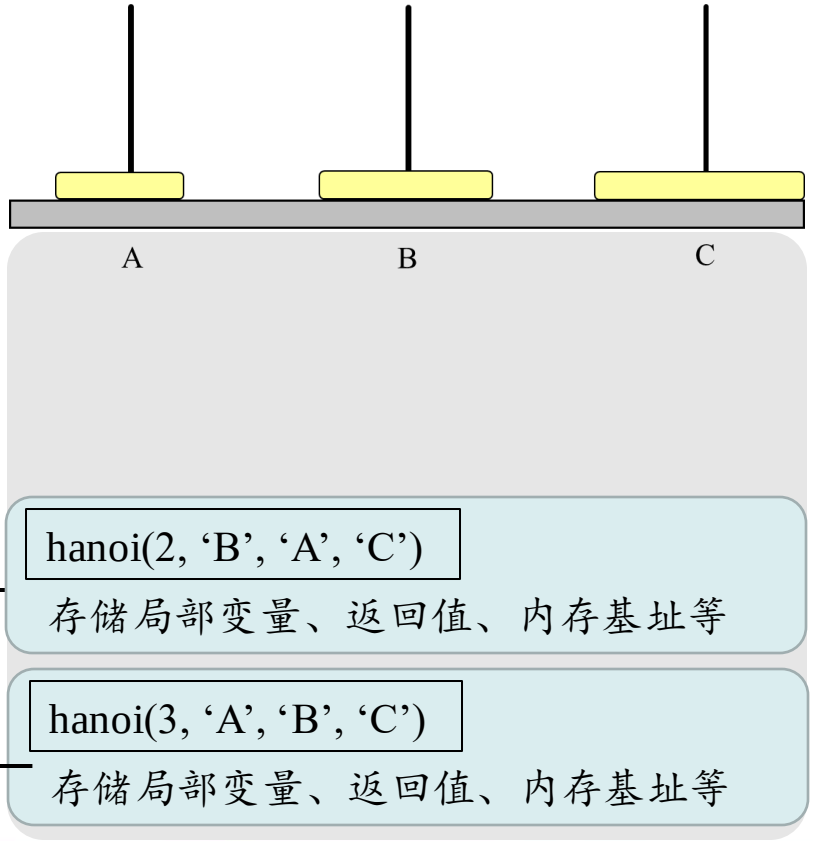
```
def hanoi(n, start, by, to):
    if n == 1:
        print(“%s -- %s” % (start, to))
        return
    else:
        hanoi(n-1, start, to, by)
        print(“%s -- %s” % (start, to))
        hanoi(n-1, by, start, to)
```



递归程序是如何执行的

- 递归函数调用和返回过程的链条就是用栈来管理的。
- 例：Hanoi(3, ‘A’, ‘B’, ‘C’)

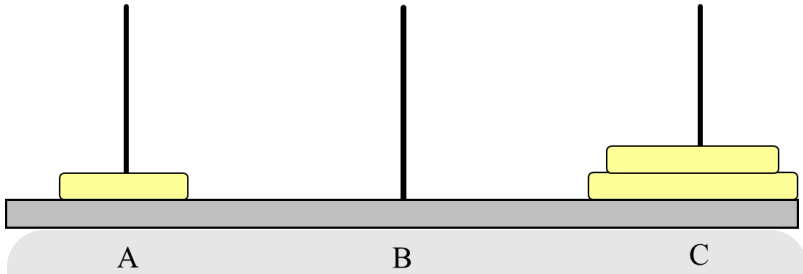
```
def hanoi(n, start, by, to):
    if n == 1:
        print(“%s -- %s” % (start, to))
        return
    else:
        hanoi(n-1, start, to, by)
        print(“%s -- %s” % (start, to))
        hanoi(n-1, by, start, to)
```



递归程序是如何执行的

- 递归函数调用和返回过程的链条就是用栈来管理的。
- 例：Hanoi(3, ‘A’, ‘B’, ‘C’)

```
def hanoi(n, start, by, to):
    if n == 1:
        print(“%s -- %s” % (start, to))
        return
    else:
        hanoi(n-1, start, to, by)
        print(“%s -- %s” % (start, to))
        hanoi(n-1, by, start, to)
```



运行时内存（栈）

函数栈帧（Frame）

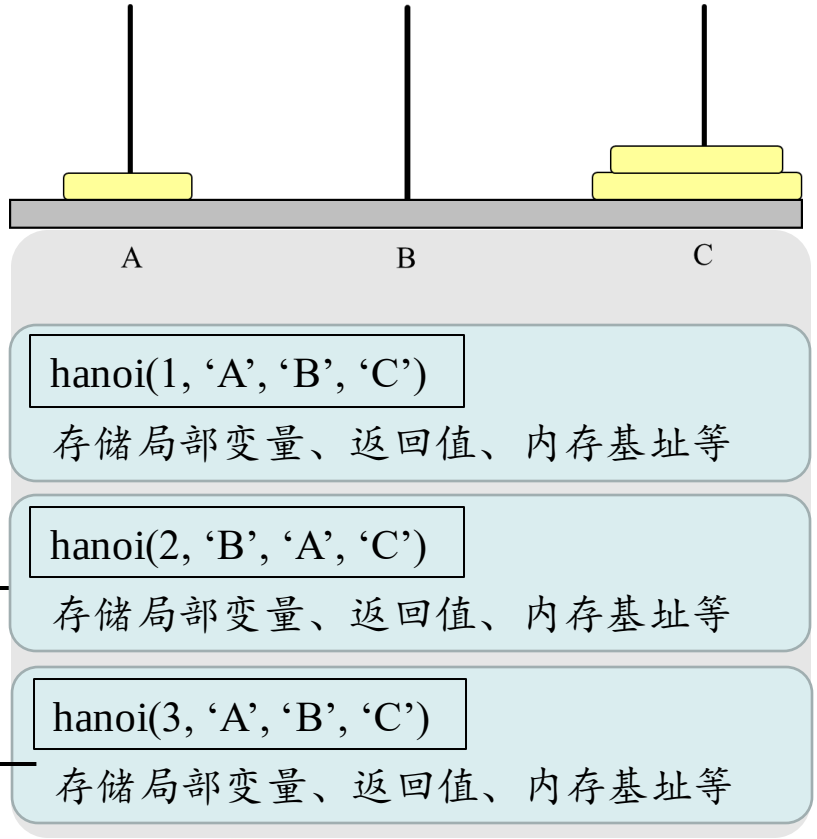
hanoi(2, ‘B’, ‘A’, ‘C’)
存储局部变量、返回值、内存基址等

hanoi(3, ‘A’, ‘B’, ‘C’)
存储局部变量、返回值、内存基址等

递归程序是如何执行的

- 递归函数调用和返回过程的链条就是用栈来管理的。
- 例：Hanoi(3, ‘A’, ‘B’, ‘C’)

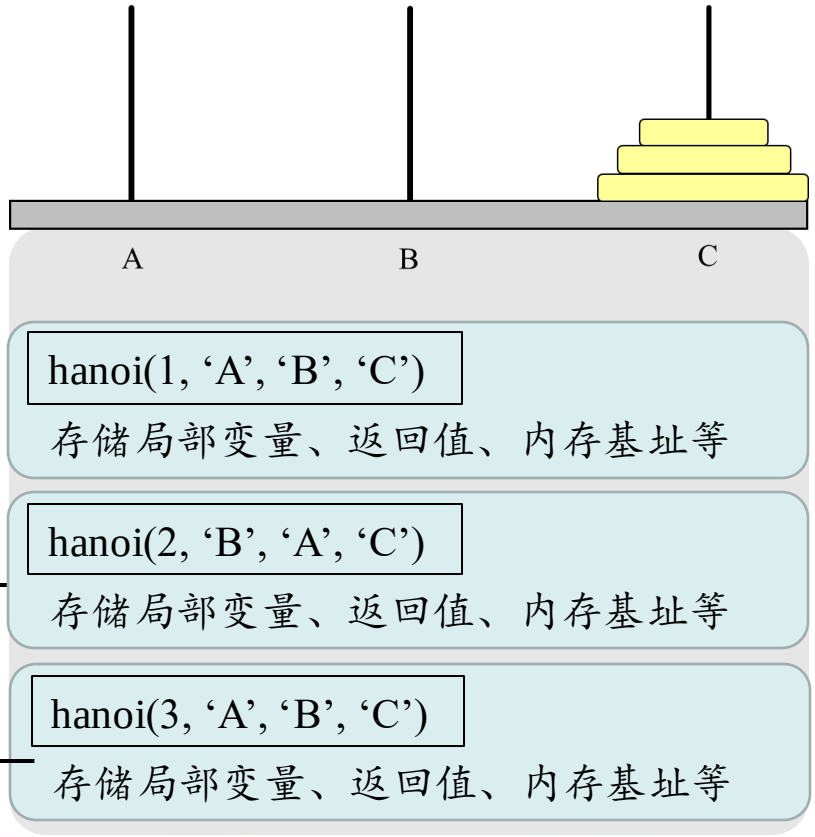
```
def hanoi(n, start, by, to):
    if n == 1:
        print("%s -- %s" % (start, to))
        return
    else:
        hanoi(n-1, start, to, by)
        print("%s -- %s" % (start, to))
        hanoi(n-1, by, start, to)
```



递归程序是如何执行的

- 递归函数调用和返回过程的链条就是用栈来管理的。
- 例：Hanoi(3, ‘A’, ‘B’, ‘C’)

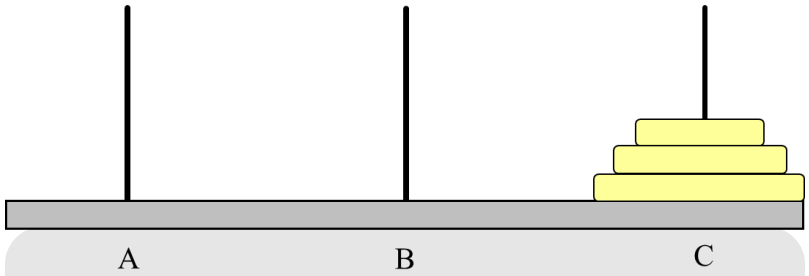
```
def hanoi(n, start, by, to):
    if n == 1:
        print("%s -- %s" % (start, to))
        return
    else:
        hanoi(n-1, start, to, by)
        print("%s -- %s" % (start, to))
        hanoi(n-1, by, start, to)
```



递归程序是如何执行的

- 递归函数调用和返回过程的链条就是用栈来管理的。
- 例：Hanoi(3, ‘A’, ‘B’, ‘C’)

```
def hanoi(n, start, by, to):
    if n == 1:
        print(“%s -- %s” % (start, to))
        return
    else:
        hanoi(n-1, start, to, by)
        print(“%s -- %s” % (start, to))
        hanoi(n-1, by, start, to)
```



运行时内存（栈）

hanoi(2, ‘B’, ‘A’, ‘C’)
存储局部变量、返回值、内存基址等

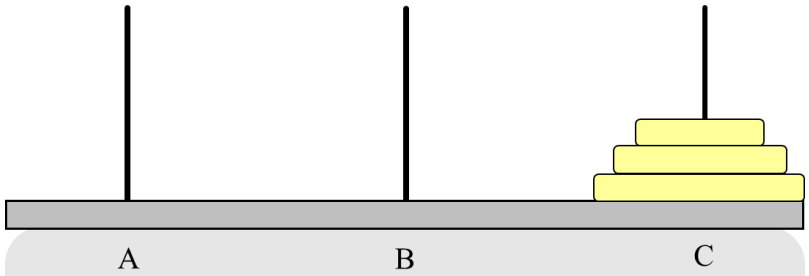
函数栈帧（Frame）

hanoi(3, ‘A’, ‘B’, ‘C’)
存储局部变量、返回值、内存基址等

递归程序是如何执行的

- 递归函数调用和返回过程的链条就是用栈来管理的。
- 例：Hanoi(3, ‘A’, ‘B’, ‘C’)

```
def hanoi(n, start, by, to):
    if n == 1:
        print(“%s -- %s” % (start, to))
        return
    else:
        hanoi(n-1, start, to, by)
        print(“%s -- %s” % (start, to))
        hanoi(n-1, by, start, to)
```



运行时内存（栈）

函数栈帧（Frame）

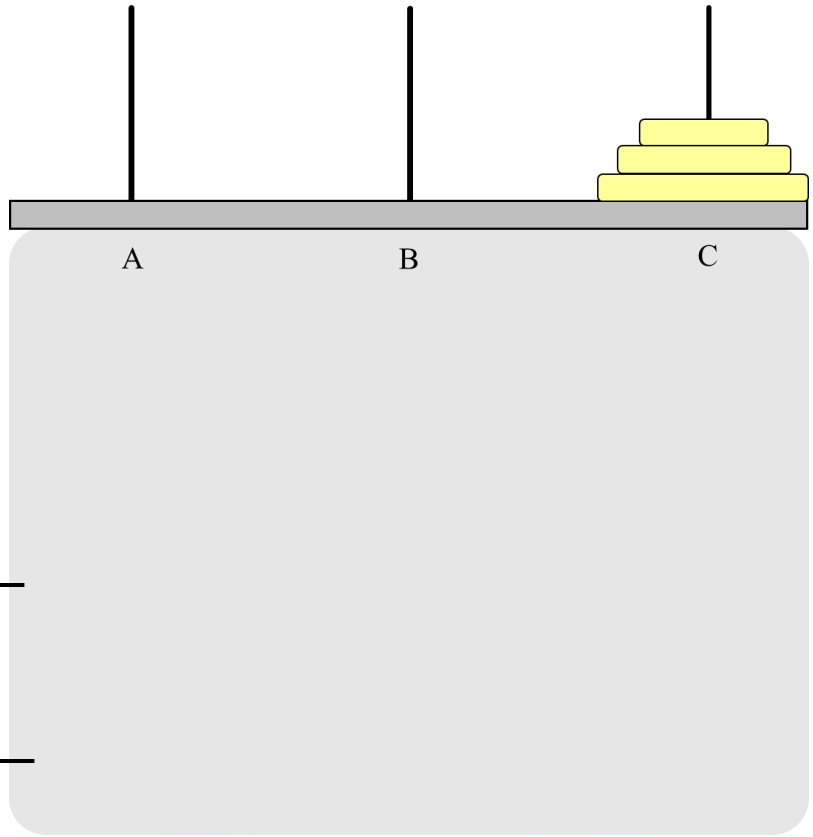
hanoi(3, ‘A’, ‘B’, ‘C’)

存储局部变量、返回值、内存基址等

递归程序是如何执行的

- 递归函数调用和返回过程的链条就是用栈来管理的。
- 例：Hanoi(3, ‘A’, ‘B’, ‘C’)

```
def hanoi(n, start, by, to):
    if n == 1:
        print(“%s -- %s” % (start, to))
        return
    else:
        hanoi(n-1, start, to, by)
        print(“%s -- %s” % (start, to))
        hanoi(n-1, by, start, to)
```



基于栈的非递归程序（补充内容）

- 程序在执行时，使用栈来管理函数的调用与返回
  - 递归函数只是一个特例
- 如果我们在程序里直接使用栈来模拟这一过程，就可以将任意递归程序转变为非递归程序

基于栈的非递归程序（补充内容）

```
def non_recursive_Hanoi(n, start, by, to):
    stack = Stack()
    stack.push((n, start, by, to, 0))

    while not stack.empty():
        n, start, by, to, flag = stack.peek()
        if n == 1:
            print("%s -- %s" % (start, to))
            stack.pop()
            continue
        if flag == 0: # 第一次递归之前
            stack.pop()
            stack.push((n, start, by, to, 1))
            stack.push((n-1, start, to, by, 0))
        if flag == 1:   # 第二次递归之前
            stack.pop()
            stack.push((n, start, by, to, 2))
            print("%s -- %s" % (start, to))
            stack.push((n-1, by, start, to, 0))
        if flag == 2:   # 第二次递归之后
            stack.pop()
```

初次调用函数，压入栈。
除参数外，还存储一个标记，
起到内存基址的定位作用

获取栈顶元素，不弹出

pop栈顶元素，表示一次函数
调用结束，栈帧弹出

改变当前栈顶元素的标记，相
当于改变当前函数内的语句执
行位置。再进行递归调用

递归调用时，将标记设为0

栈溢出：Stack Overflow

- 递归失败的情形：栈溢出
  - 没有设置正确的递归终止条件，或者终止条件无法达到
  - 算法不收敛，问题子规模经递归没有减小
  - 问题规模过大或算法效率过低，在达到终止条件之前已经耗尽内存
- 总之，递归调用的深度过深时，程序耗尽了可用的栈空间，就会导致栈溢出错误。
  - 栈溢出错误（Stack Overflow）其实是最容易发现和改正的错误之一，通常是递归设置错误引起的死循环；这寓意了Stack Overflow社区就像调试栈溢出错误一样简洁高效。

    --来自 Stack Overflow(一个著名的IT技术问答网站）的解答

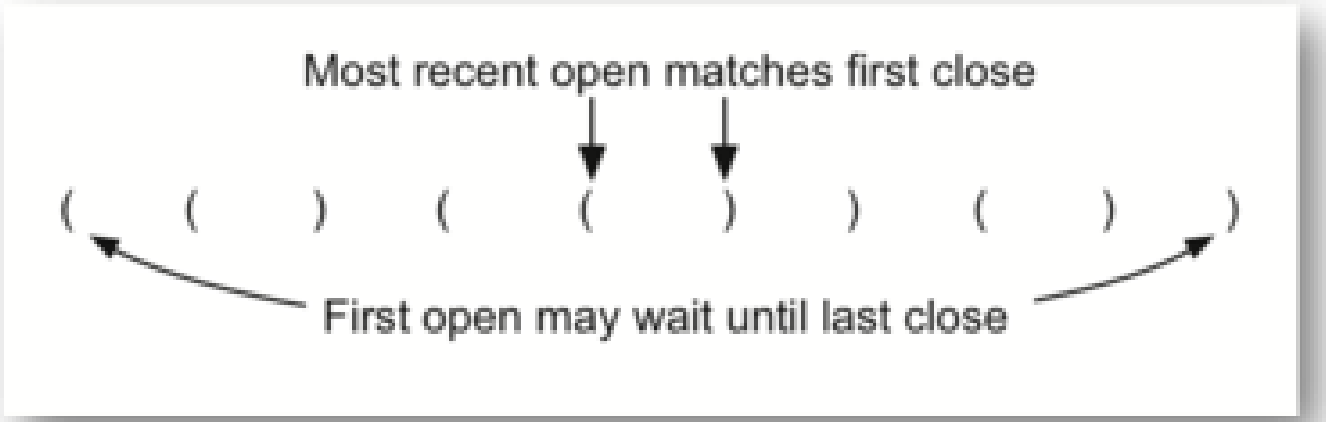
4.3 栈的应用

括号匹配

- 我们都写过这样的表达式：(5+6)*(7+8)/(4+3)
  - 这里的括号是用来指定表达式项的计算优先级
- 有些函数式语言，如Lisp，在函数定义的时候会用到大量的括号
  - 比如：(defun square(n)
    (* n n))
  - 这个语句定义了一个计算平方值的函数square，返回参数n的平方值。
- 当然，括号的使用必须遵循一定的“平衡”规则
  - 首先，每个开括号要恰好对应一个闭括号；
  - 其次，每对开闭括号要正确的嵌套
  - 正确的括号：()()()()，((()))，()(()())
  - 错误的括号：((((((()，()))，()()(()

括号匹配

- 对括号是否正确匹配的识别，是很多语言编译器的基础算法
- 下面分析如何构造这个括号匹配识别的算法
  - 从左到右扫描，最近打开的左括号，应该匹配最先遇到的右括号
  - 这样，第一个左括号（最早打开），就应该匹配最后一个右括号（最后遇到）
  - 这种次序反转的识别，正好符合栈的特性！



括号匹配

```
from pythonds.basic.stack import Stack

def parChecker(symbolString):
    s = Stack()
    balanced = True
    index = 0
    while index < len(symbolString) and balanced:
        symbol = symbolString[index]
        if symbol == "(":
            s.push(symbol)
        else:
            if s.isEmpty():
                balanced = False
            else:
                s.pop()

        index = index + 1

    if balanced and s.isEmpty():
        return True
    else:
        return False

print(parChecker('((()))'))
print(parChecker('(()'))
```

左括号，入栈

遇到无法匹配的右括号，表示序列不合法

右括号，出栈

结束后仍有未匹配的左括号，表示序列不合法

更多种括号的匹配

- 在实际的应用里，我们会碰到更多种括号，如python中列表所用的方括号“[]”，字典所用的花括号“{}”，元组和表达式所用的圆括号“()”
- 不同括号混合使用时，就要分别判断各自的开闭匹配情况
- 下面这些是匹配的
  - {{([][])}()}
  - [[{{(())}}]]
  - [][][]({})
- 下面这些是不匹配的
  - ([)]　　(((()]))
  - [{()]

更多种括号的匹配

- 需要改进的地方
  - 碰到各种左括号仍然入栈
  - 碰到各种右括号的时候需要判断栈顶的左括号是否跟右括号属于同一种类

```
def parChecker(symbolString):
    s = Stack()
    balanced = True
    index = 0
    while index < len(symbolString) and balanced:
        symbol = symbolString[index]
        if symbol == "(":
            s.push(symbol)
        else:
            if s.isEmpty():
                balanced = False
            else:
                s.pop()

        index = index + 1

    if balanced and s.isEmpty():
        return True
    else:
        return False
```

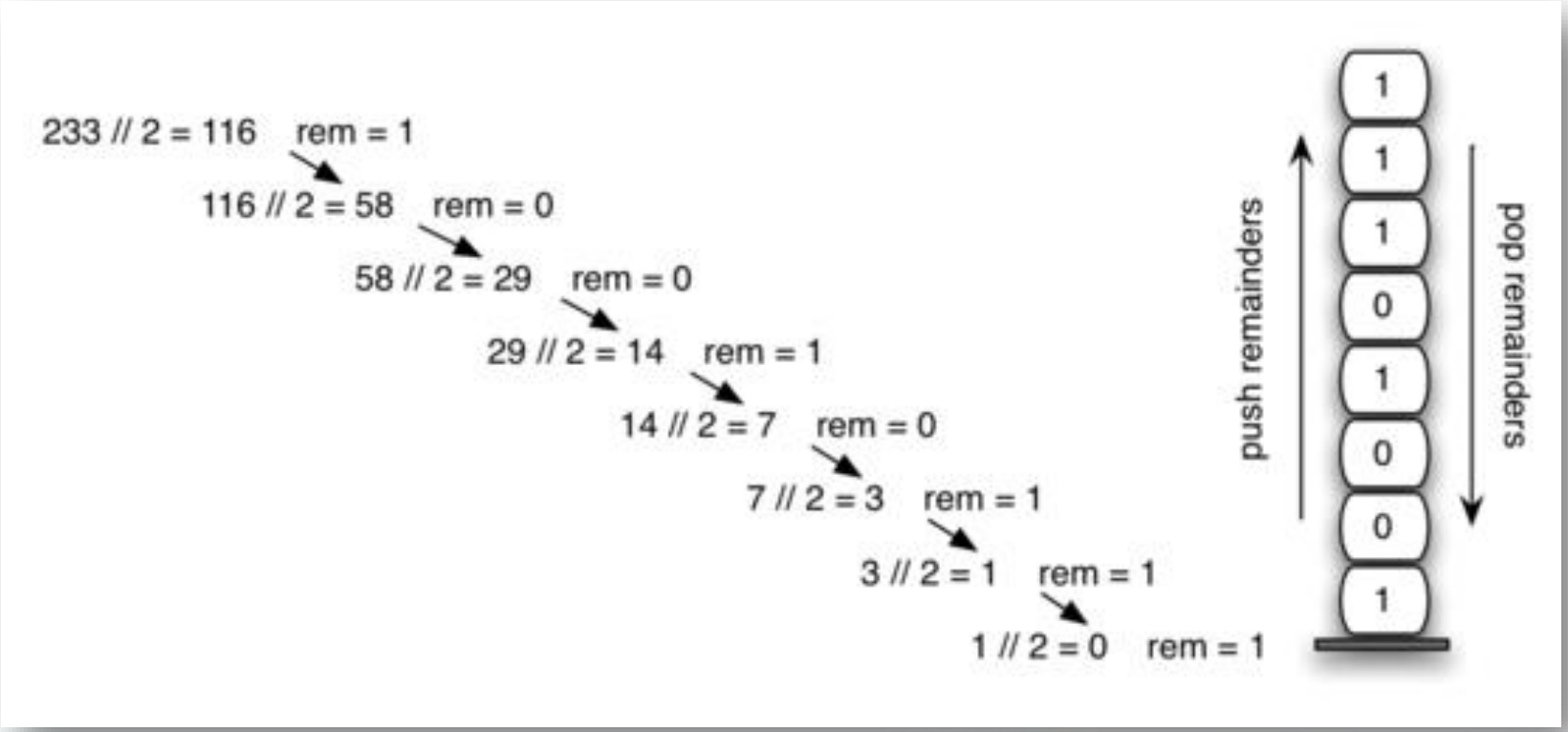
```
from pythonds.basic.stack import Stack

def parChecker(symbolString):
    s = Stack()
    balanced = True
    index = 0
    while index < len(symbolString) and balanced:
        symbol = symbolString[index]
        if symbol in "([{":
            s.push(symbol)
        else:
            if s.isEmpty():
                balanced = False
            else:
                top = s.pop()
                if not matches(top,symbol):
                    balanced = False
        index = index + 1
    if balanced and s.isEmpty():
        return True
    else:
        return False

def matches(open,close):
    opens = "([{"
    closers = ")]}"
    return opens.index(open) == closers.index(close)


print(parChecker('{{([][])}()}'))
print(parChecker('[{()]'))
```

十进制转换为二进制

- 二进制是计算机中最基本的概念，作为组成计算机最基本部件的逻辑门电路，其输入和输出均仅为两种状态：0和1
- 但十进制是人类传统文化中最基本的数值概念，如果没有进制之间的转换，人们跟计算机的交互会相当的困难
- 整数是最通用的数据类型，我们经常需要将整数在二进制和十进制之间转换
  - 如：$(233)_{10}$的对应二进制数为$(11101001)_2$，具体是这样：
  - $(233)_{10}=2\times10^2+3\times10^1+3\times10^0$
  - $(11101001)_2=1\times2^7+1\times2^6+1\times2^5+0\times2^4+1\times2^3+0\times2^2+0\times2^1+1\times2^0$
- 十进制转换为二进制，采用的是“除以2”的算法
  - 将整数不断除以2，每次得到的余数就是由低到高的二进制位

十进制转换为二进制

- “除以2”的过程，得到的余数是从低到高的次序，而输出则是从高到低，所以需要一个栈来反转次序。



十进制转换为二进制

```
from pythonds.basic.stack import Stack

def divideBy2(decNumber):
    remstack = Stack()

    while decNumber > 0:
        rem = decNumber % 2
        remstack.push(rem)
        decNumber = decNumber // 2

    binString = ""
    while not remstack.isEmpty():
        binString = binString + str(remstack.pop())

    return binString

print(divideBy2(42))
```

求余数

地板除

扩展到更多进制转换

- 十进制转换为二进制的算法，很容易可以扩展为转换到任意进制，只需要将“除以2”算法改为“除以N”算法就可以
- 计算机中另外两种常用的进制是八进制和十六进制
  - $(233)_{10}$等于$(351)_8$和$(E9)_{16}$
  - $(351)_8=3\times82+5\times81+1\times80$
  - $(E9)_{16}=14\times161+9\times160$
- 主要的问题是如何表示八进制及十六进制
  - 二进制有两个不同数字0、1
  - 十进制有10个不同数字0、1、2、3、4、5、6、7、8、9
  - 八进制可用8个不同数字0、1、2、3、4、5、6、7
  - 十六进制的16个数字则是0、1、2、3、4、5、6、7、8、9、A、B、C、D、E、F

十进制转换为十六以下任意进制

```
from pythonds.basic.stack import Stack

def baseConverter(decNumber,base):
    digits = "0123456789ABCDEF"

    remstack = Stack()

    while decNumber > 0:
        rem = decNumber % base
        remstack.push(rem)
        decNumber = decNumber // base

    newString = ""
    while not remstack.isEmpty():
        newString = newString + digits[remstack.pop()]

    return newString

print(baseConverter(25,2))
print(baseConverter(25,16))
```

前缀、中缀和后缀表达式

- 基础的表达式，例如：B*C，很容易知道这是B乘以C
- 这种操作符（operator）介于操作数（operand）中间的表示法，称为中缀表示法
- 但有时候中缀表示法会引起混淆，如“A+B*C”，是A+B然后再乘以C，还是B*C然后再去加A？
  - 人们引入了操作符“优先级”的概念来消除混淆，规定高优先级的操作符先计算，相同优先级的操作符从左到右依次计算
    - 这样A+B*C就没有疑义是A加上B与C的乘积
  - 同时引入了括号来表示强制优先级，括号的优先级最高，而且在嵌套的括号中，内层的优先级更高
    - 这样(A+B)*C就是A与B的和再乘以C

前缀、中缀和后缀表达式

- 虽然人们已经习惯了这种表示法，但计算机处理最好是能明确规定所有的计算顺序，这样无需处理复杂的优先规则
- 引入全括号表达式：在所有的表达式项两边都加上括号
  –A+B*C+D，应表示为((A+(B*C))+D)
- 可否将表达式中操作符的位置稍移动一下？

前缀、中缀和后缀表达式

- 例如中缀表达式A+B，将操作符移到前面，变为“+AB”，或者将操作符移到最后，变为“AB+”
- 我们就得到了表达式的另外两种表示法：前缀表达式(波兰表达式)和后缀表达式(逆波兰表达式)
- 这样A+B*C将变为前缀的“+A*BC”，后缀的“ABC*+”，为了帮助理解，子表达式加了下划线

Infix Expression	Prefix Expression	Postfix Expression
A + B	+ A B	A B +
A + B * C	+ A * B C	A B C * +

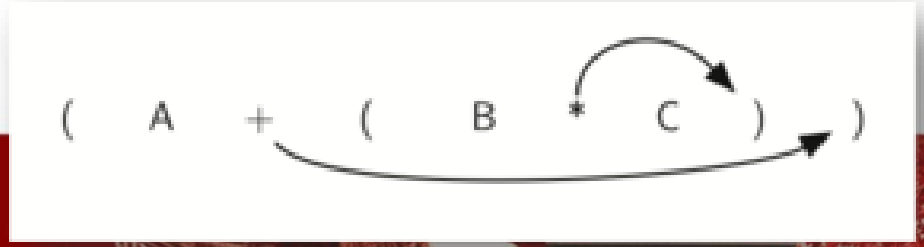
前缀、中缀和后缀表达式

- 再来看中缀表达式“(A+B)*C”，按照转换的规则，前缀表达式是“*+ABC”，而后缀表达式是“AB+C*”
- 神奇的事情发生了，在中缀表达式里必须的括号，在前缀和后缀表达式中消失了？
- 在**前缀和后缀表达式**中，**操作符的次序完全决定了运算的次序**，不再有混淆
- 下面看更多的例子

Infix Expression	Prefix Expression	Postfix Expression
A + B * C + D	+ + A * B C D	A B C * + D +
(A + B) * (C + D)	* + A B + C D	A B + C D + *
A * B + C * D	+ * A B * C D	A B * C D * +
A + B + C + D	+ + + A B C D	A B + C + D +

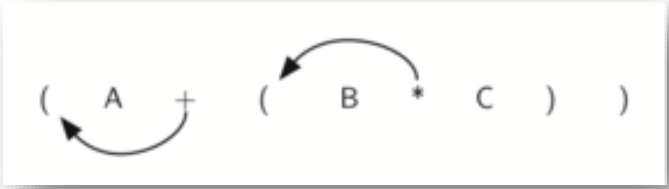
中缀表达式转换为前缀和后缀形式

- 我们将几个中缀表达式手动成了前缀和后缀的形式
- 如何设计一个算法，实现中缀到前缀和后缀表达式的转换？
- 先来考虑全括号的中缀表达式：
  - 对于A+B*C，写成全括号形式：(A+(B*C))，显式表达了计算次序
- 注意到：每一对括号，都包含了一组完整的操作符和操作数
  - 将每一对括号内的唯一的操作符，移至最后
  - 然后，删除所有括号得到的就是后缀表达式
  - 类似地，如果移到最前，得到的就是前缀表达式
  - 进一步再把更多的操作符移动到相应的右括号处替代之，再删去左括号，那么整个表达式就完成了到后缀表达式的转换

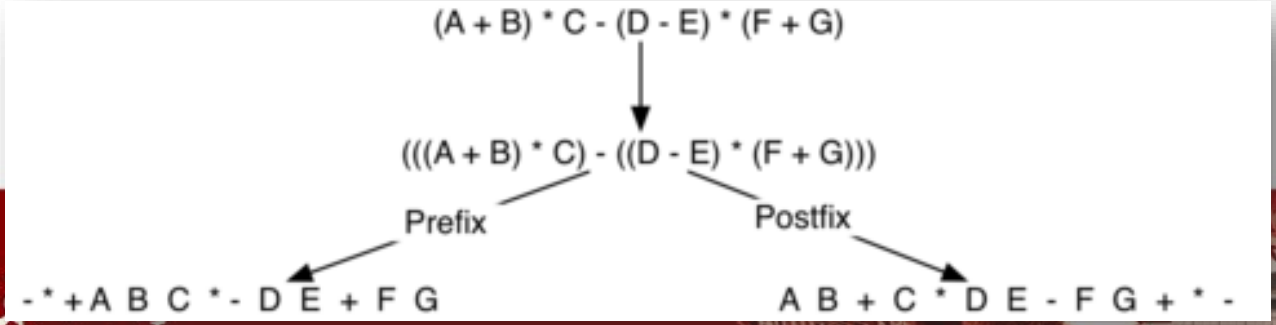


中缀表达式转换为前缀和后缀形式

- 同样的，如果我们把操作符移动到左括号的位置替代之，然后删掉所有的右括号，也就得到了前缀表达式



- 所以，中缀表达式转换成前缀或者后缀，只需要两个步骤
  - 将中缀表达式转换为全括号形式
  - 将所有的操作符移动到子表达式所在的左括号（前缀）或者右括号（后缀）处，替代之，再删除所有的括号



通用的中缀转后缀算法

- 考虑：中缀表达式A+B*C，其对应的后缀表达式是ABC*+
  - 操作数ABC的顺序没有改变。
  - 操作符的出现顺序，在后缀表达式中反转了，由于*的优先级比+高，所以后缀表达式中操作符的出现顺序与运算次序一致。
- 在中缀表达式转换为后缀形式的处理过程中，由于操作符比操作数要晚输出，所以在扫描到对应的第二个操作数之前，需要把操作符先保存起来，而这些暂存的操作符，由于优先级的规则，还有可能要反转次序输出。
  - 在A+B*C中，+虽然先出现，但优先级比后面这个*要低，所以它要等*处理完后，才能再处理。
- 这种反转特性，使得我们考虑用栈来保存暂时未处理的操作符

通用的中缀转后缀算法

- 再看看(A+B)*C，对应的后缀形式是AB+C*。
  - 这一次，+的输出比*要早，主要是因为括号使得+的优先级提升，高于括号之外的*
- 回顾“全括号”技术，后缀表达式中操作符应该出现在左括号对应的右括号位置
- 所以遇到左括号，要标记下，其后出现的操作符优先级提升了，一旦扫描到对应的右括号，就可以马上输出这个操作符

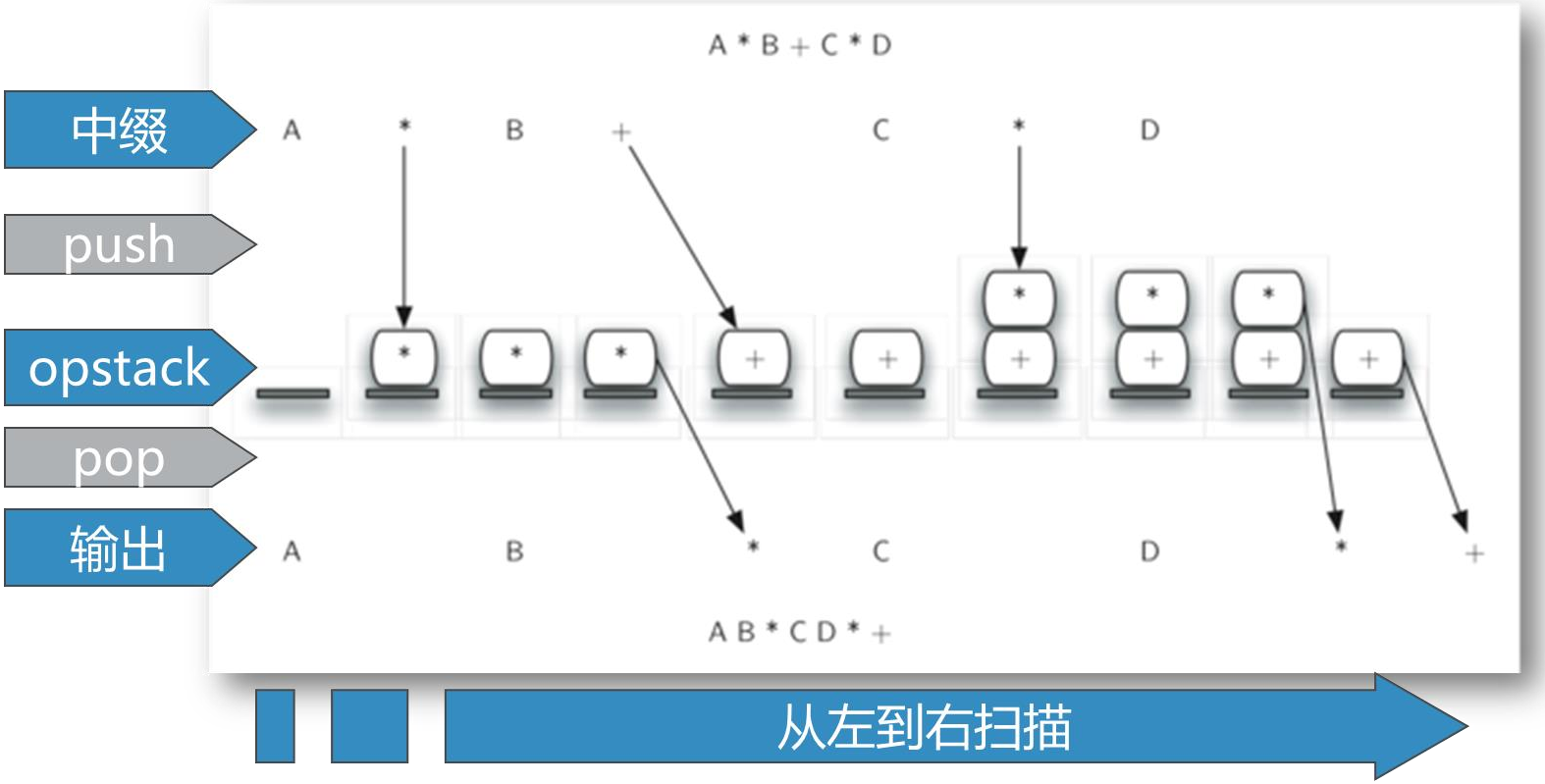
通用的中缀转后缀算法

- 总结下，在从左到右扫描逐个字符扫描中缀表达式的过程中，采用一个栈来暂存未处理的操作符，
- 这样，栈顶的操作符就是最近暂存进去的，当遇到一个新的操作符，就需要跟栈顶的操作符比较下优先级，再行处理。
- 后面的算法描述中，约定中缀表达式是由空格隔开的一系列单词（token）构成，操作符单词包括*/+-()，而操作数单词则是单字母标识符A、B、C等。

通用的中缀转后缀算法：流程

- 创建空栈opstack用于暂存操作符，空表用于保存后缀表达式
- 用split方法，将中缀表达式转换为单词（token）的列表
- 从左到右扫描中缀表达式单词列表
  - 如果单词是一个操作数，则直接添加到后缀表达式列表的末尾
  - 如果单词是一个左括号“(”，则压入opstack栈顶
  - 如果单词是一个右括号“)”，则反复弹出opstack栈顶的操作符，加入到输出列表末尾，直到碰到左括号
  - 如果单词是一个操作符“*/+-”，则压入opstack栈顶。但在压入之前，要比较其与栈顶操作符的优先级，如果栈顶的高于或等于它，就要反复弹出栈顶操作符，加入到输出列表末尾，直到栈顶的操作符优先级低于它
  - 中缀表达式单词列表扫描结束后，把opstack栈中的所有剩余操作符依次弹出，添加到输出列表末尾
- 把输出列表再用join方法合并成后缀表达式字符串，算法结束。

通用的中缀转后缀算法：实例



通用的中缀转后缀算法：流程

```
from pythonds.basic.stack import Stack

def infixToPostfix(infixexpr):
    prec = {}
    prec["*"] = 3
    prec["/"] = 3
    prec["+"] = 2
    prec["-"] = 2
    prec["("] = 1
    opStack = Stack()
    postfixList = []
    tokenList = infixexpr.split()

    for token in tokenList:
        if token in "ABCDEFGHIJKLMNOPQRSTUVWXYZ" or token in "0123456789":
            postfixList.append(token)
        elif token == '(':
            opStack.push(token)
        elif token == ')':
            topToken = opStack.pop()
            while topToken != '(':
                postfixList.append(topToken)
                topToken = opStack.pop()
        else:
            while (not opStack.isEmpty()) and \
               (prec[opStack.peek()] >= prec[token]):
                  postfixList.append(opStack.pop())
            opStack.push(token)

    while not opStack.isEmpty():
        postfixList.append(opStack.pop())
    return " ".join(postfixList)
```

记录操作符优先级

解析表达式到单词列表

操作数

(

)

操作符

操作符

合成后缀表达式字符串

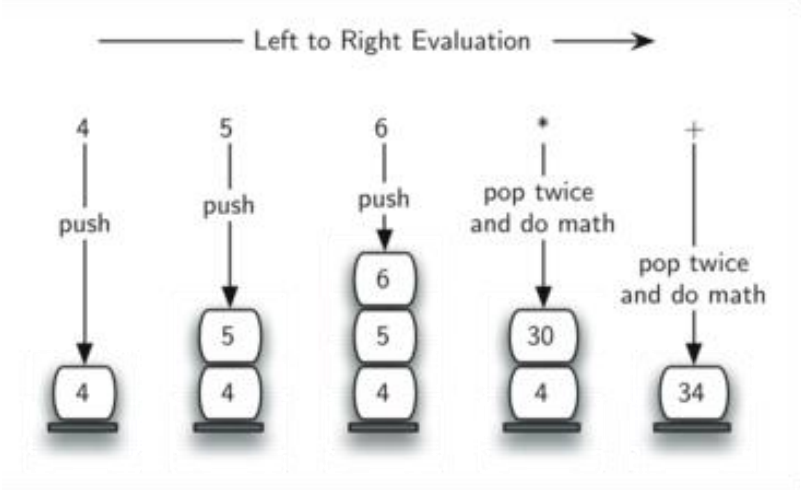
后缀表达式求值

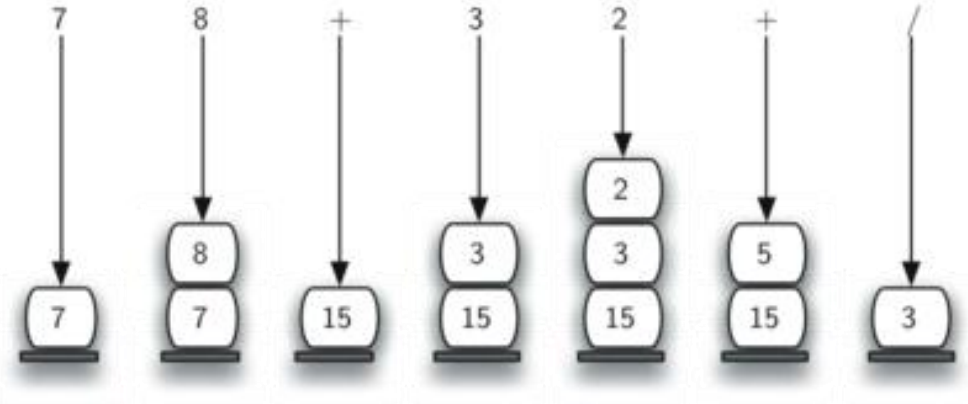
- 作为栈结构的结束，我们来讨论“后缀表达式求值”问题
- 跟中缀转换为后缀问题不同，
- 在对后缀表达式从左到右扫描的过程中，
- 由于操作符在操作数的后面，
- 所以要暂存操作数，在碰到操作符的时候，再将暂存的两个操作数进行实际的计算
  - 仍然是栈的特性：操作符只作用于离它最近的两个操作数

后缀表达式求值

- 如“4 5 6 * +”，我们先扫描到4、5两个操作数，但还不知道对这两个操作数能做什么计算，需要继续扫描后面的符号才能知道
- 继续扫描，又碰到操作数6，所以还不能知道如何计算，继续暂存入栈，直到“*”，现在知道是栈顶两个操作数5、6做乘法，我们弹出两个操作数，计算得到结果30
  - 需要注意：先弹出的是右操作数，后弹出的是左操作数，这个对于-/很重要！
- 为了继续后续的计算，需要把这个中间结果30压入栈顶
- 当所有操作符都处理完毕，栈中只留下1个操作数，就是表达式的值

后缀表达式求值


Left to Right Evaluation
4
push
5
push
6
push
*
pop twice
and do math
+
pop twice
and do math
4
5
4
6
5
4
30
4
34


7
8
+
3
2
+
/
7
8
7
15
3
15
2
3
15
5
15
3

后缀表达式求值

- 创建空栈operandStack用于暂存操作数
- 将后缀表达式用split方法解析为单词（stoken）的列表
- 从左到右扫描单词列表
  - 如果单词是一个操作数，将单词转换为整数int，压入operandStack栈顶
  - 如果单词是一个操作符（*/+-），就开始求值，从栈顶弹出2个操作数，先弹出的是右操作数，后弹出的是左操作数，计算后将值重新压入栈顶
- 单词列表扫描结束后，表达式的值就在栈顶
- 弹出栈顶的值，返回。

后缀表达式求值

```
def postfixEval(postfixExpr):
    operandStack = Stack()
    tokenList = postfixExpr.split()

    for token in tokenList:
        if token in "0123456789":
            operandStack.push(int(token))
        else:
            operand2 = operandStack.pop()
            operand1 = operandStack.pop()
            result = doMath(token,operand1,operand2)
            operandStack.push(result)
    return operandStack.pop()

def doMath(op, op1, op2):
    if op == "*":
        return op1 * op2
    elif op == "/":
        return op1 / op2
    elif op == "+":
        return op1 + op2
    else:
        return op1 - op2
```

操作数

操作符