



Data Structure & Algorithm Analysis

Stack

Zibin Zheng (郑子彬)

School of Data and Computer Science , SYSU

<http://www.inpluslab.com>

课程主页: <http://inpluslab.sysu.edu.cn/dsa2016/>

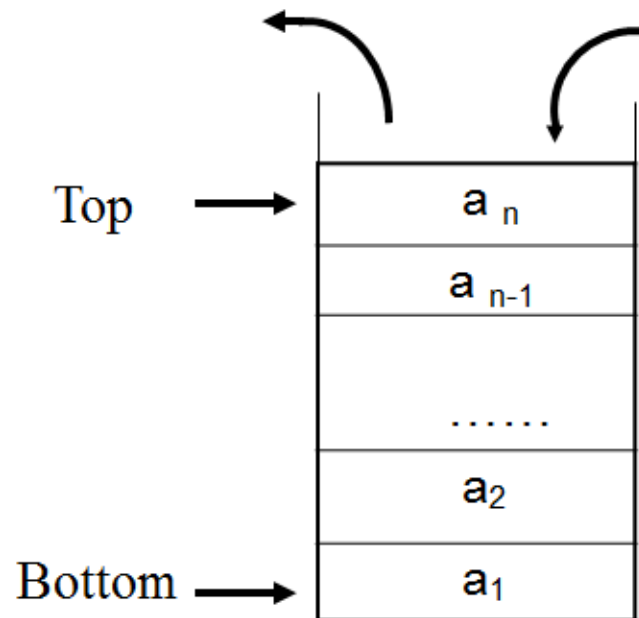
栈和队列

两种特殊的线性表——栈和队列

- 从数据结构角度看，栈和队列是操作受限的线性表，他们的逻辑结构相同。
- 从抽象数据类型角度看，栈和队列是两种重要的抽象数据类型。

Definition

- Stack (栈)
- 限定**仅在表尾**进行插入和删除操作的线性表。



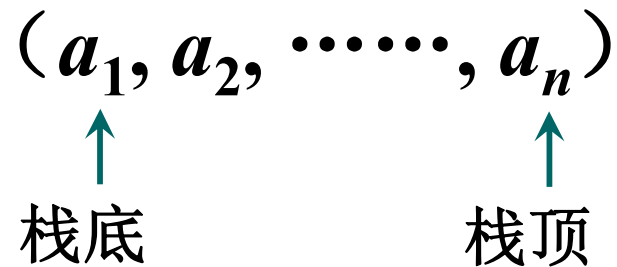
Stacks are sometimes known as:

Last In, First Out (LIFO)

Definition

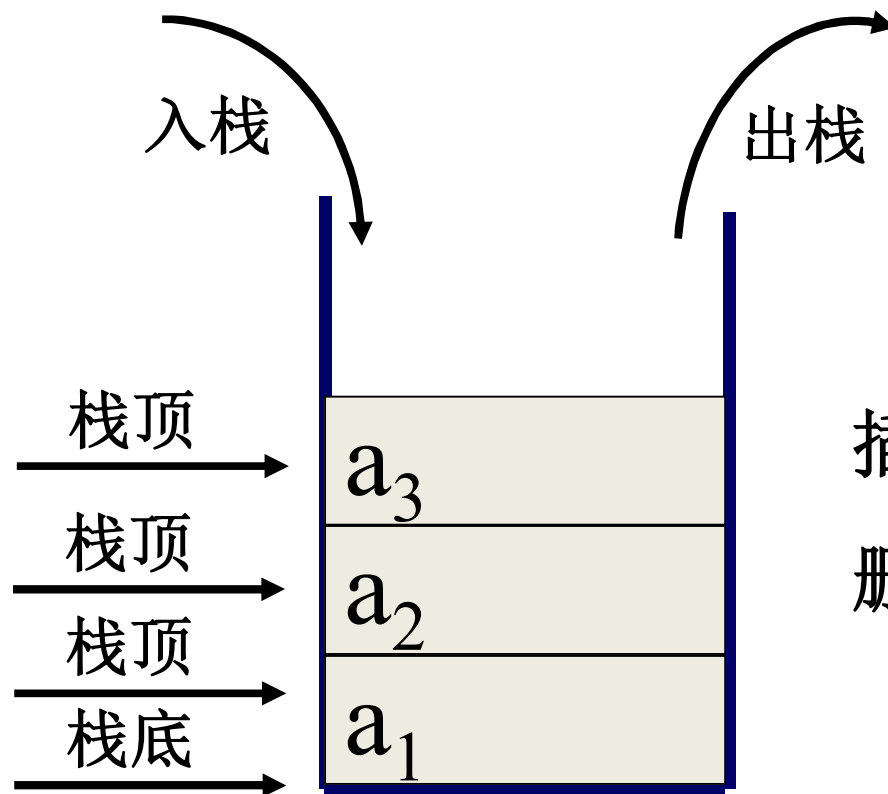
栈的逻辑结构

- **栈**：限定仅在**表的一端**进行插入和删除操作的**线性表**。
- 允许插入和删除的一端称为**栈顶**，另一端称为**栈底**。
- **空栈**：不含任何数据元素的栈。



Definition

栈的逻辑结构

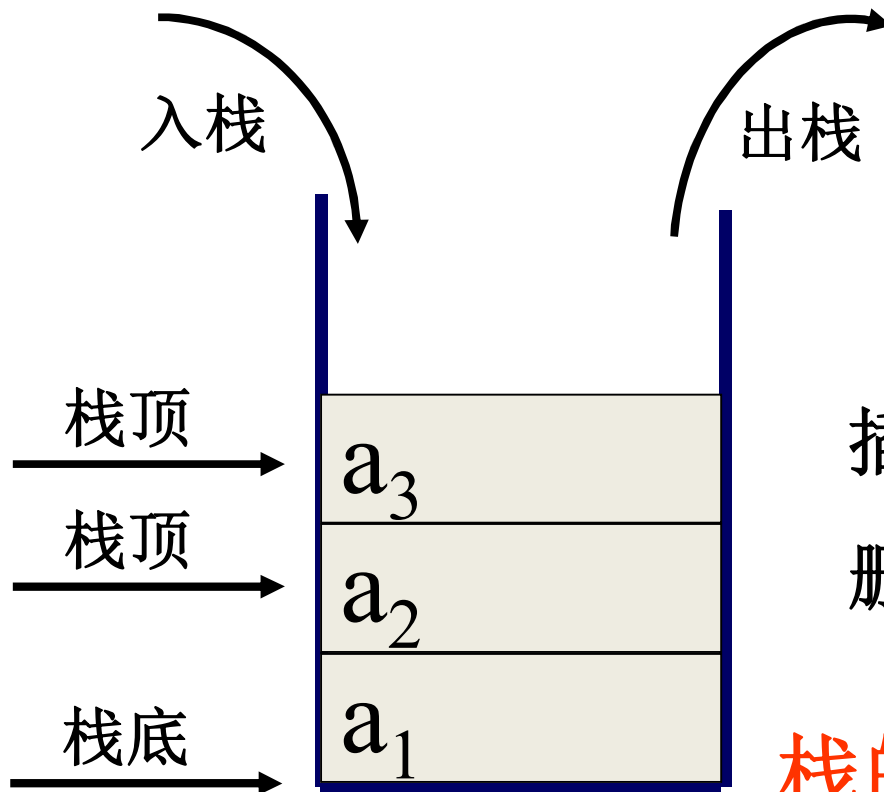


插入：入栈、进栈、压栈

删除：出栈、弹栈

Definition

栈的逻辑结构



插入：入栈、进栈、压栈

删除：出栈、弹栈

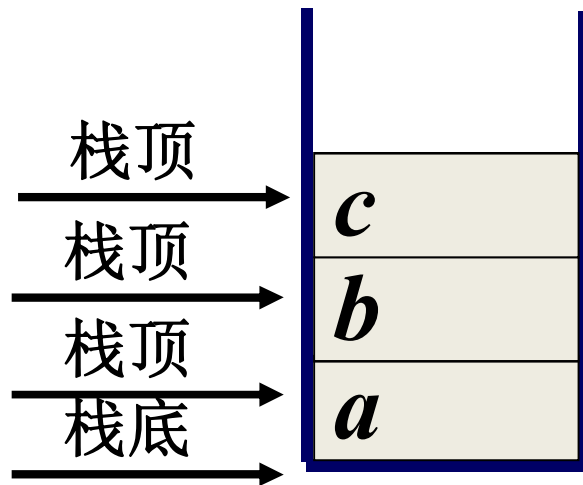
栈的操作特性：后进先出

Definition

栈的逻辑结构

例：有三个元素按 a 、 b 、 c 的次序依次进栈，且每个元素只允许进一次栈，则可能的出栈序列有多少种？

➤ 情况1:

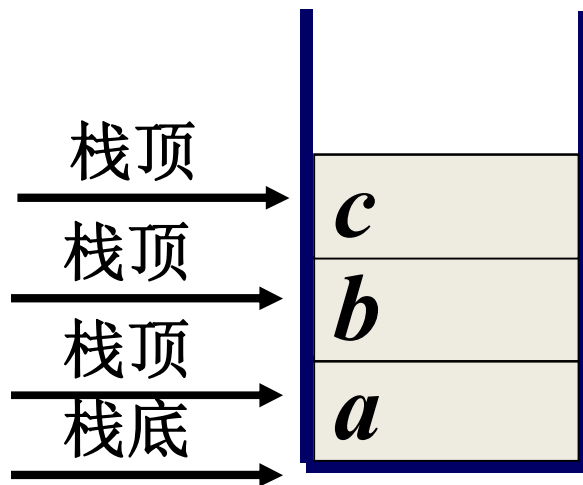


Definition

栈的逻辑结构

例：有三个元素按 a 、 b 、 c 的次序依次进栈，且每个元素只允许进一次栈，则可能的出栈序列有多少种？

➤ 情况1:



出栈序列: c

出栈序列: c 、 b

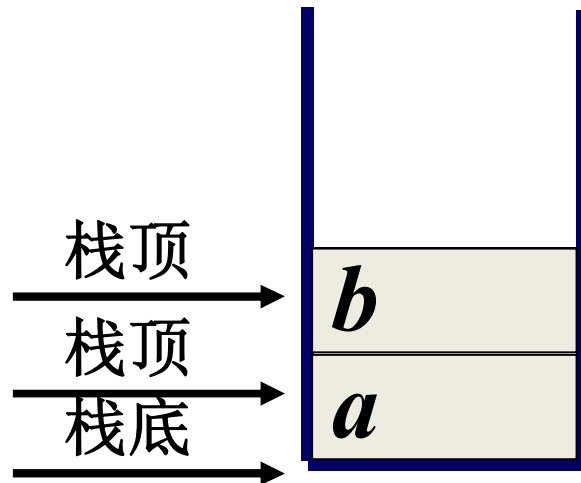
出栈序列: c 、 b 、 a

Definition

栈的逻辑结构

例：有三个元素按 a 、 b 、 c 的次序依次进栈，且每个元素只允许进一次栈，则可能的出栈序列有多少种？

➤ 情况2：



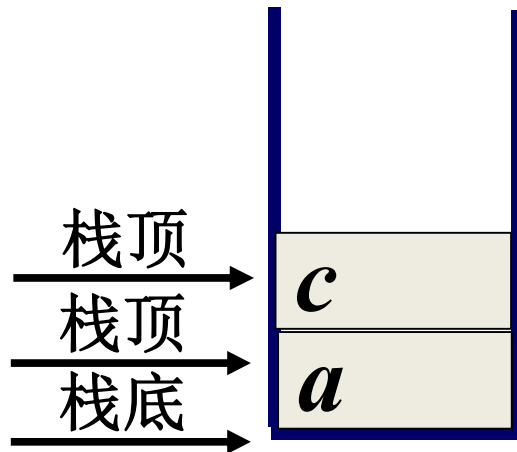
出栈序列： b

Definition

栈的逻辑结构

例：有三个元素按 a 、 b 、 c 的次序依次进栈，且每个元素只允许进一次栈，则可能的出栈序列有多少种？

➤ 情况2：



出栈序列： b

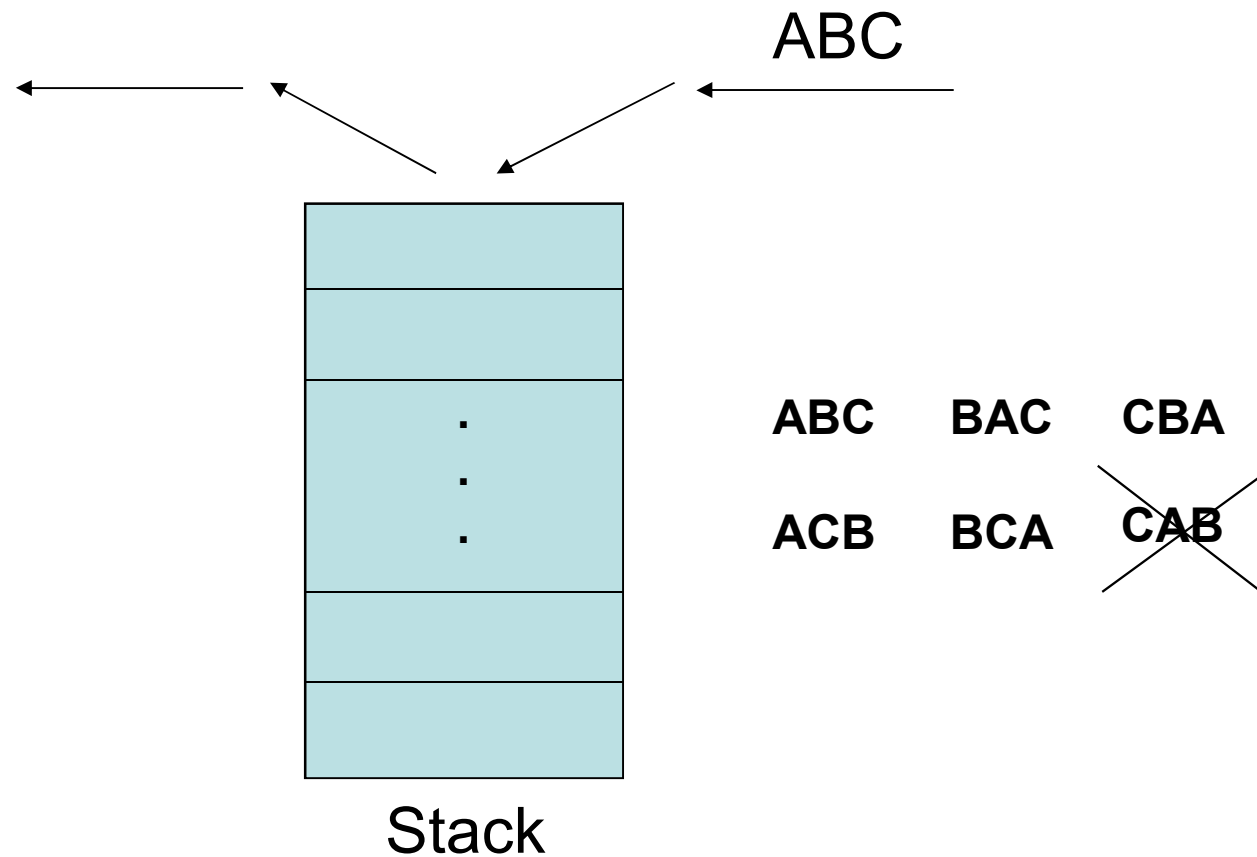
出栈序列： b 、 c

出栈序列： b 、 c 、 a

注意： 栈只是对表插入和删除操作的位置进行了限制，并没有限定插入和删除操作进行的时间。

A case

- Which sequences are impossible ?



栈的抽象数据类型定义

ADT Stack

Data

栈中元素具有相同类型及后进先出特性，
相邻元素具有前驱和后继关系

Operation

InitStack

前置条件：栈不存在

输入：无

功能：栈的初始化

输出：无

后置条件：构造一个空栈

栈的抽象数据类型定义

DestroyStack

前置条件：栈已存在

输入：无

功能：销毁栈

输出：无

后置条件：释放栈所占用的存储空间

Push

前置条件：栈已存在

输入：元素值x

功能：在栈顶插入一个元素x

输出：如果插入不成功，抛出异常

后置条件：如果插入成功，栈顶增加了一个元素

栈的抽象数据类型定义

Pop

前置条件：栈已存在

输入：无

功能：删除栈顶元素

输出：如果删除成功，返回被删元素值，否则，抛出异常

后置条件：如果删除成功，栈减少了一个元素

GetTop

前置条件：栈已存在

输入：无

功能：读取当前的栈顶元素

输出：若栈不空，返回当前的栈顶元素值

后置条件：栈不变

栈的抽象数据类型定义

Empty

前置条件：栈已存在

输入：无

功能：判断栈是否为空

输出：如果栈为空，返回1，否则，返回0

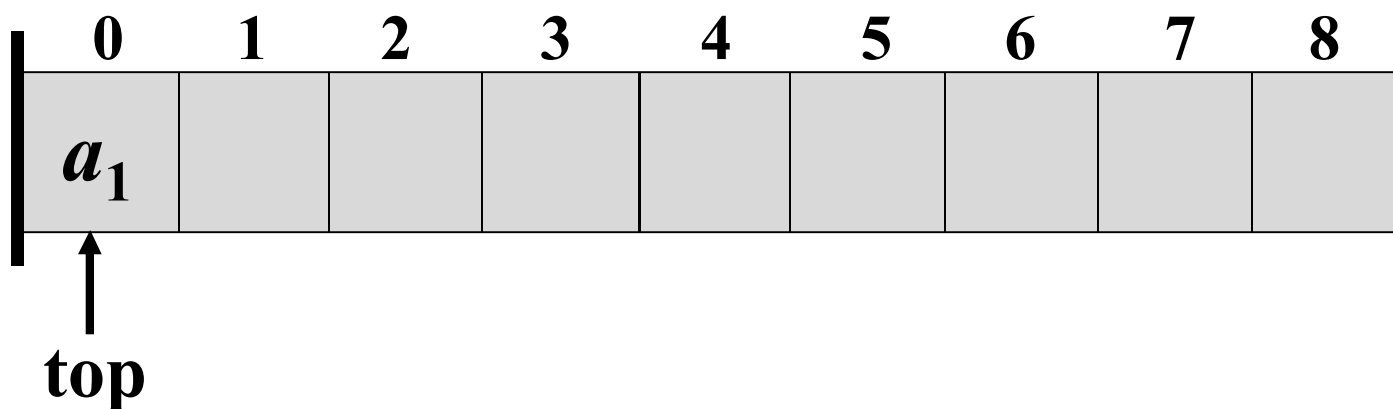
后置条件：栈不变

endADT

栈的顺序存储结构及实现

顺序栈——栈的顺序存储结构

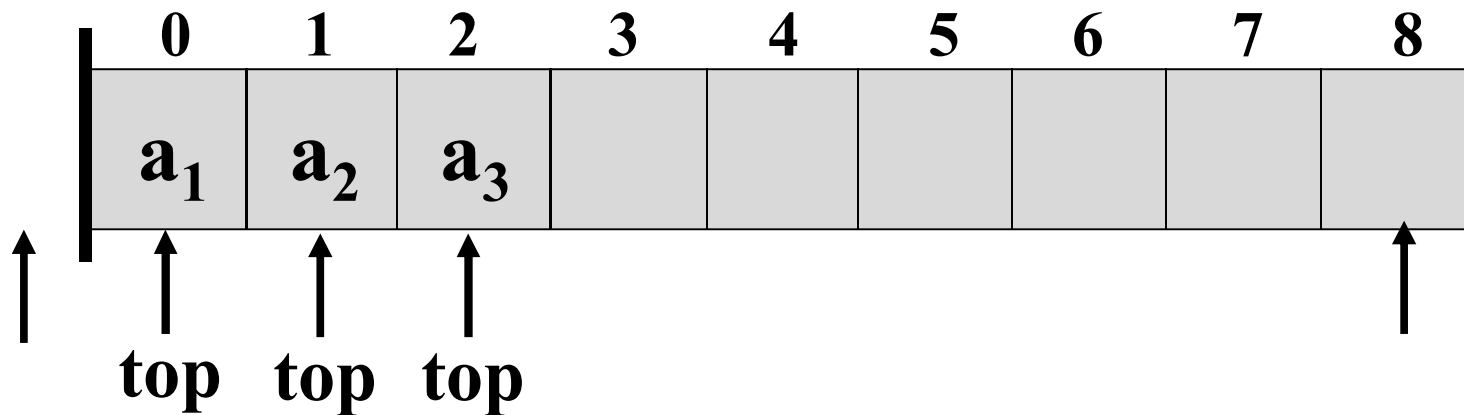
① 如何改造数组实现栈的顺序存储？



确定用数组的哪一端表示栈底。

附设指针**top**指示栈顶元素在数组中的位置。

栈的顺序存储结构及实现



进栈: $\text{top}+1$

栈空: $\text{top} = -1$

出栈: $\text{top}-1$

栈满: $\text{top} = \text{MAX_SIZE}-1$

Implementation

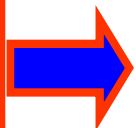
顺序栈类的声明

```
const int MAX_SIZE=100;
template <class DataType>
class seqStack
{
    public:
        seqStack ( ) ;
        ~seqStack ( );
        void Push ( DataType x );
        DataType Pop ( );
        DataType GetTop ( );
        bool Empty ( );
    private:
        DataType data[MAX_SIZE];
        int top;
}
```

入栈操作 Push

操作接口: `void Push(DataType x);`

```
template <class DataType>
void seqStack<DataType> ::Push ( DataType x)
{
    if (top == MAX_SIZE-1) throw “溢出” ;
    top++;
    data[top] = x;
}
```

 `data[++top] = x`



时间复杂度?

出栈操作 Pop

操作接口: `DataType Pop();`

```
template <class DataType>
DataType seqStack<DataType> :: Pop ( )
{
    if (top == -1) throw “栈为空” ;
    x = data[top--];
    return x;
}
```



时间复杂度?

读栈操作 GetTop

操作接口: **DataType GetTop();**

```
template <class DataType>
DataType seqStack<DataType> :: GetTop ( )
{
    if (top == -1) throw “栈为空” ;
    x = data[top];
    return x;
}
```

两栈共享空间

① 在一个程序中需要**同时**使用具有**相同**数据类型的**两个栈**，如何顺序存储这两个栈？

解决方案1:

直接解决：为每个栈开辟一个数组空间。

② 会出现什么问题？如何解决？

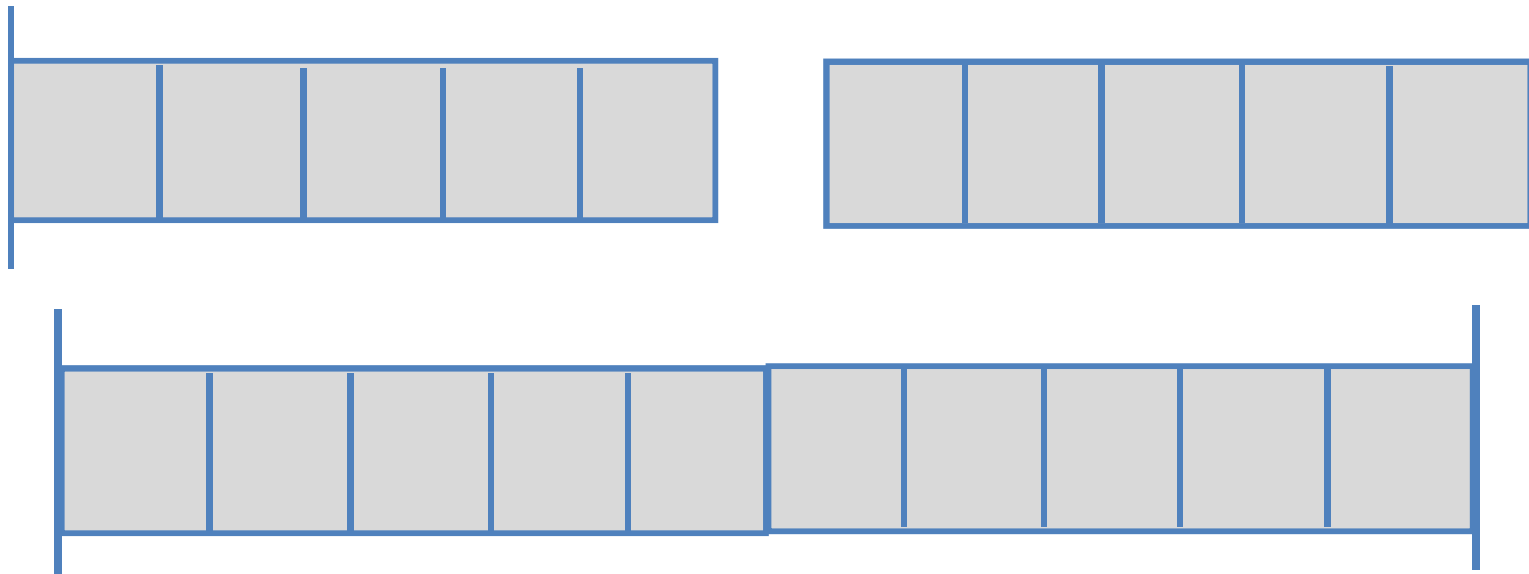
解决方案2:

顺序栈单向延伸——使用一个数组来存储两个栈

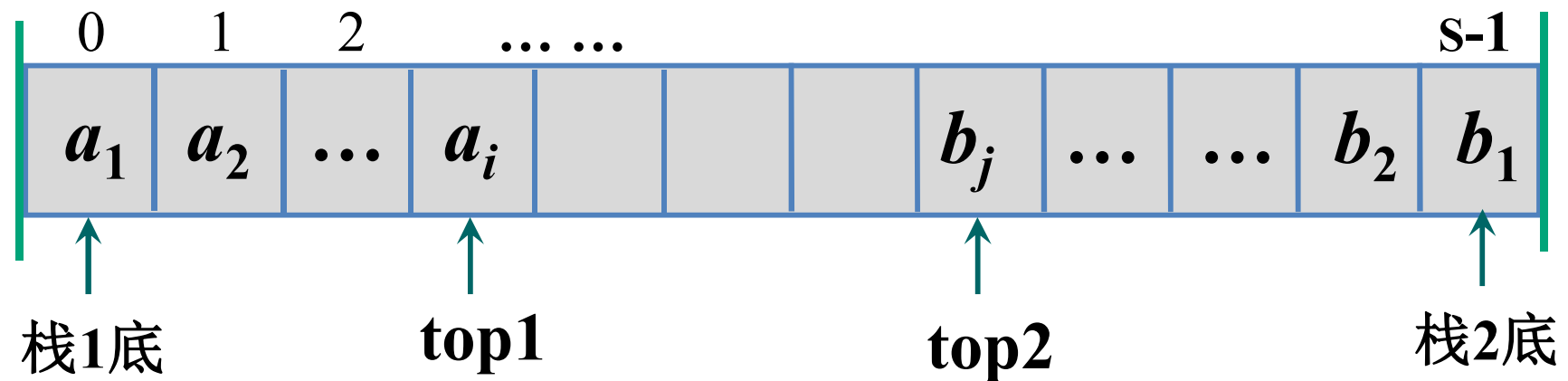
Variants of the stacks

两栈共享空间

两栈共享空间：使用一个数组来存储两个栈，让一个栈的栈底为该数组的始端，另一个栈的栈底为该数组的末端，两个栈从各自的端点向中间延伸。



Variants of the stacks



栈1的底固定在下标为0的一端；

栈2的底固定在下标为StackSize-1的一端。

$top1$ 和 $top2$ 分别为栈1和栈2的栈顶指针；

Stack_Size为整个数组空间的大小（图中用S表示）；

Variants of the stacks

两栈共享空间

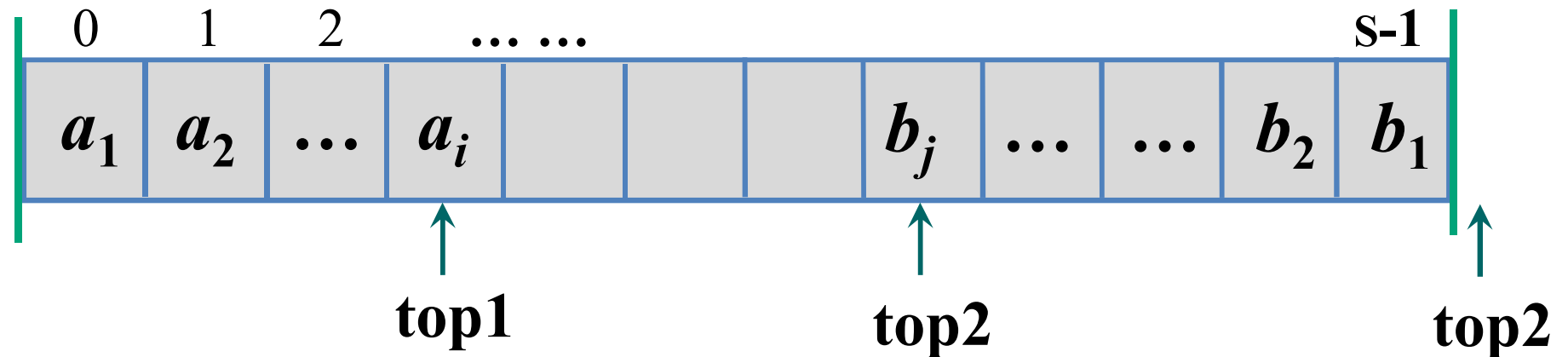


① 什么时候栈1为空?

$top1 = -1$

Variants of the stacks

两栈共享空间



① 什么时候栈1为空?

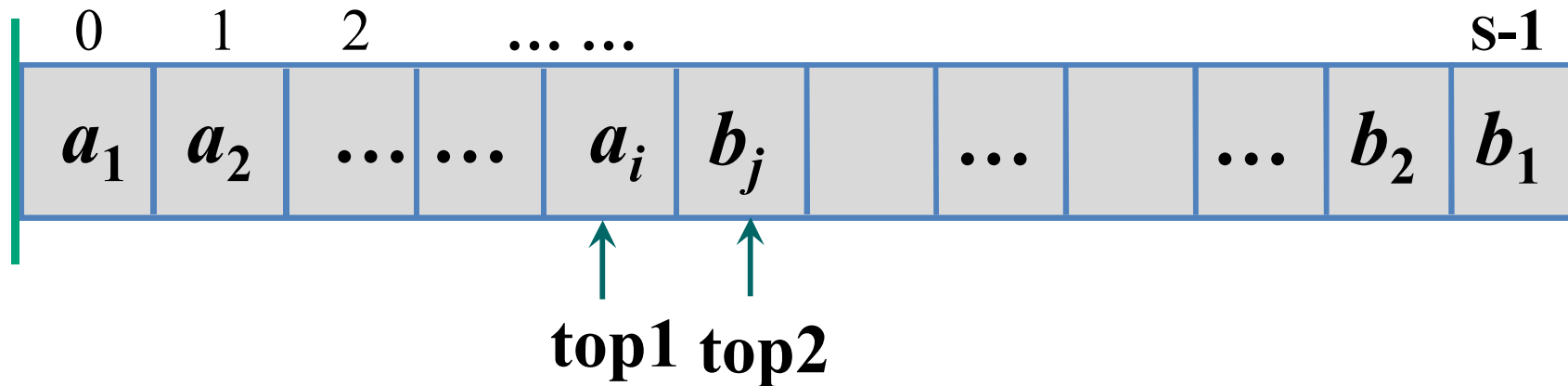
$top1 = -1$

② 什么时候栈2为空?

$top2 = \text{Stack_Size}$

Variants of the stacks

两栈共享空间



① 什么时候栈1为空?

$top1 = -1$

② 什么时候栈2为空?

$top2 = \text{Stack_Size}$

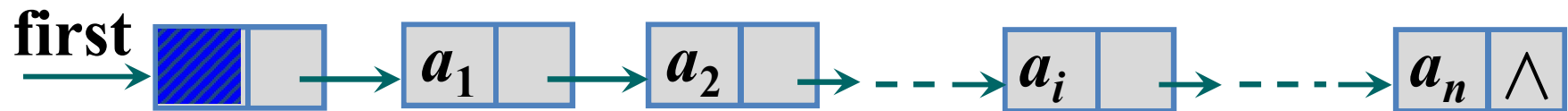
③ 什么时候栈满?

$top2 = top1 + 1$

栈的链接存储结构及实现

链栈：栈的链接存储结构

① 如何改造链表实现栈的链接存储？



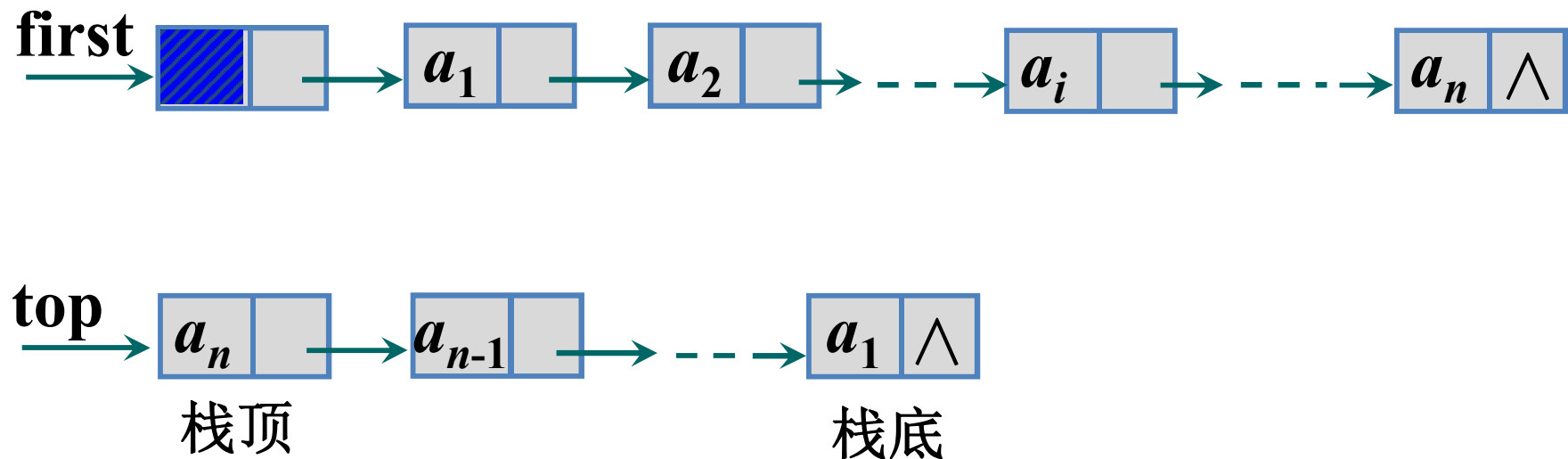
① 将哪一端作为栈顶？ 将链头作为栈顶，方便操作。

① 链栈需要加头结点吗？ 链栈不需要附设头结点。

Linked List

栈的链接存储结构及实现

链栈：栈的链接存储结构



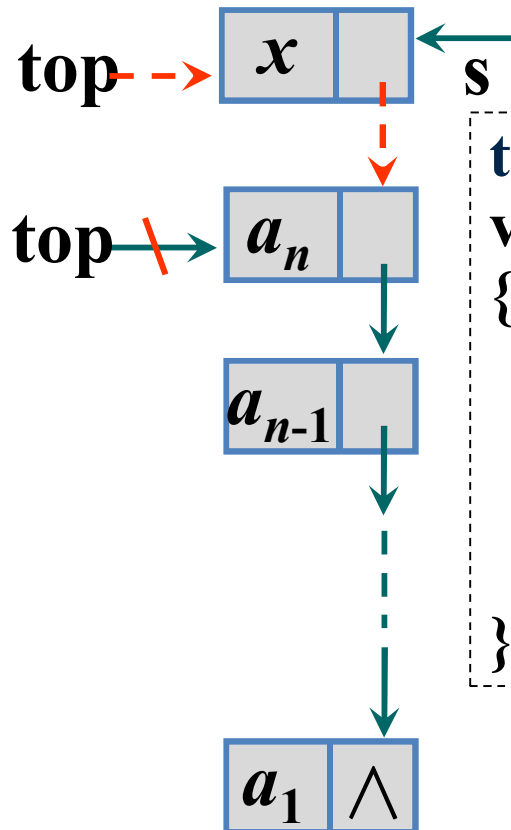
Operations

链栈的类声明

```
template <class DataType>
class LinkStack
{
    public:
        LinkStack( );
        ~LinkStack( );
        void Push(DataType x);
        DataType Pop( );
        DataType GetTop( );
        bool Empty( );
    private:
        Node<DataType> *top;
}
```

Operations

操作接口: **void Push(DataType x);**

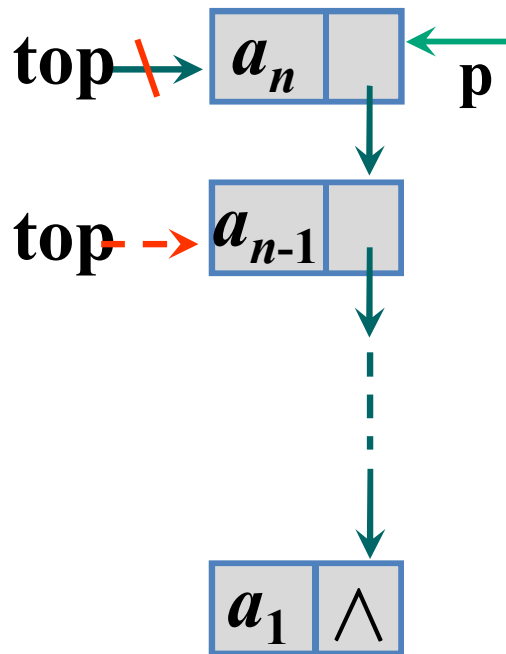


```
template <class DataType>
void LinkStack<DataType> ::Push (DataType x)
{
    s = new Node<DataType>;
    s->data = x;
    s->next = top;
    top = s;
}
```

② 为什么没有判断栈满?

Operations

操作接口: **DataType Pop();**



```
template <class DataType>
DataType LinkStack<DataType> ::Pop( )
{
    if (top == NULL)
        throw "下溢";
    x = top->data;
    p = top;
    top = top->next;
    delete p;
    return x;
}
```

① top++可以吗?

顺序栈和链式栈的比较

■ 时间效率

- 所有操作都只需常数时间
- 顺序栈和链式栈在时间效率上难分伯仲

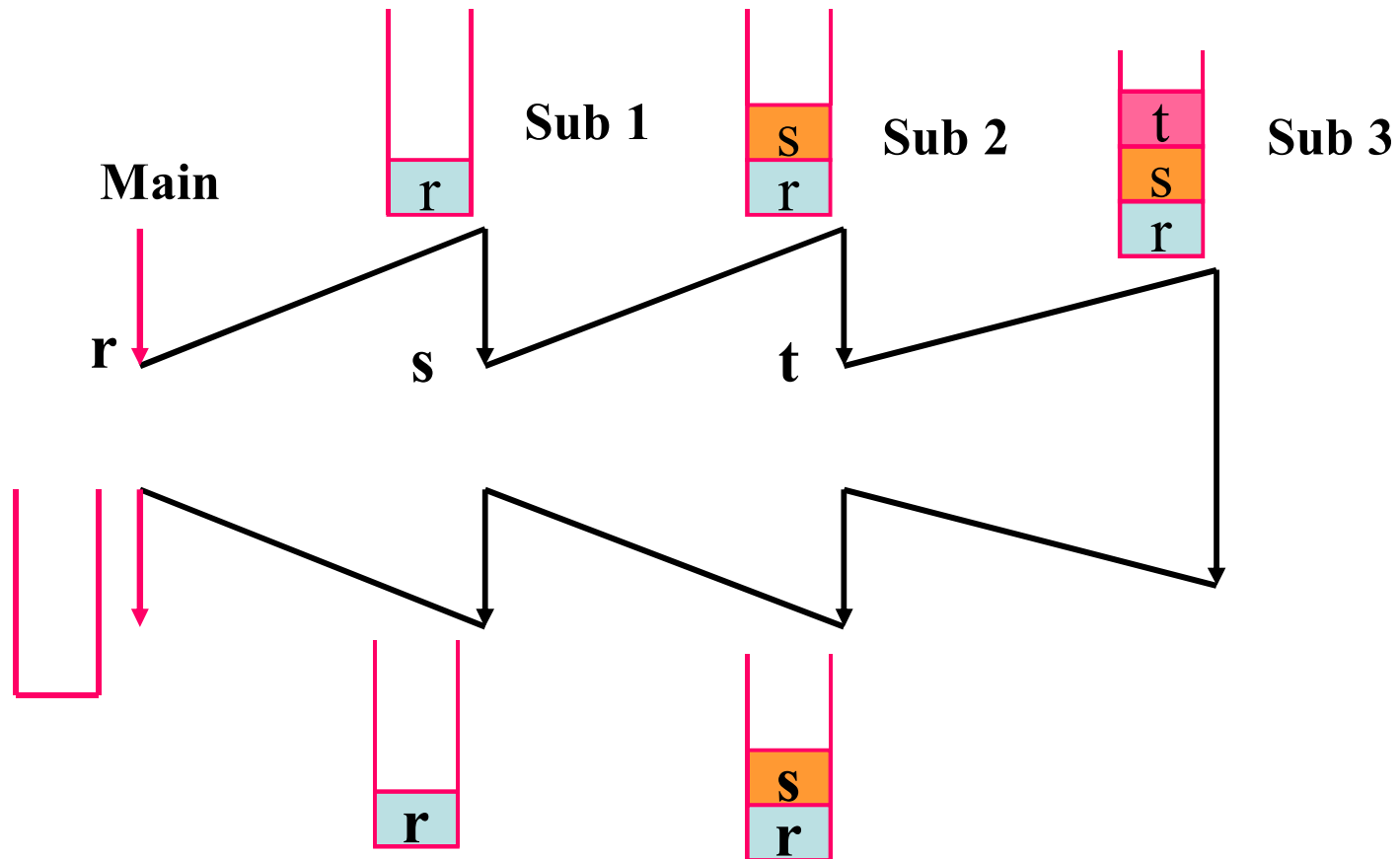
■ 空间效率

- 顺序栈须说明一个固定的长度、空间浪费
- 没有栈满的问题，只有当内存没有可用空间时才会出现栈满，但是每个元素都需要一个指针域，从而产生了结构性开销

总之，当栈的使用过程中元素个数变化较大时，用链栈是适宜的，反之，应该采用顺序栈。

Applications

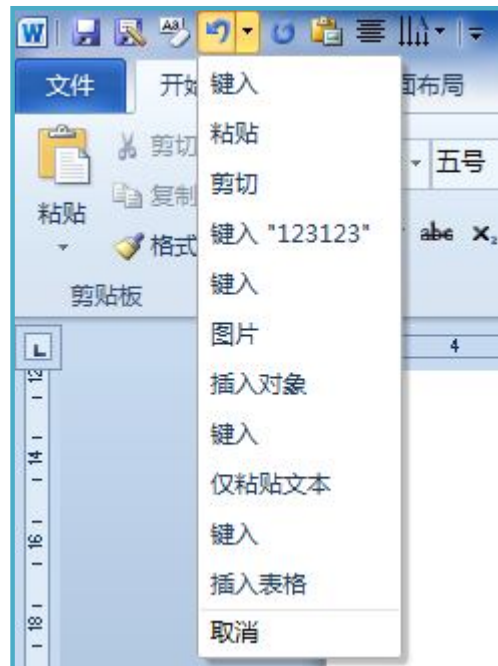
- Function call



Applications

- Historical record in the software

Office Word



Photoshop

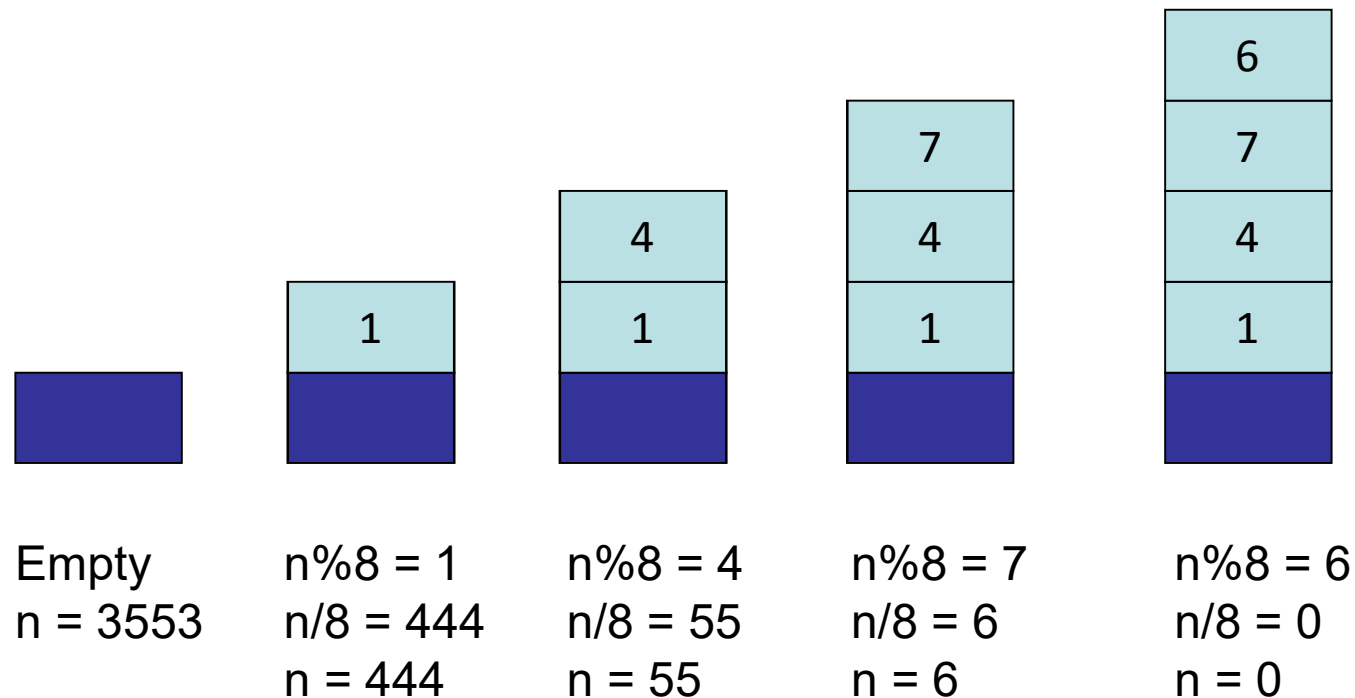


Applications

- 数制转换

- 非负十进制数转换成其它进制的数的一种简单方法

例：十进制转换成八进制 $(3553)_{10} = (6741)_8$



表达式求值算法

- 表达式都是由**操作数**（operand）、**运算符**（operator）和**界限符**（delimiter）组成的。
- 讨论简单算术表达式的求值问题——只含加、减、乘、除四则运算，所有的运算对象均为数，并以“#”为**结束符**。
- ◆ 常用算法——“**算符优先法**”
 - ◆ 算符优先法就是根据运算符和界限符的优先次序的规定来实现表达式求值的。
- ◆ 算术四则运算规则——运算符的优先次序规定：
 - 1) 先乘除，后加减；
 - 2) 从左算到右；
 - 3) 先括号内，后括号外。

算术表达式的三种表示

- 算术表达式有三种表示：

- ◆ 中缀(infix)表示

<操作数> <操作符> <操作数> , 如
 $A+B$;

- ◆ 前缀(prefix)表示——波兰式表示

<操作符> <操作数> <操作数> , 如
 $+AB$;

- ◆ 后缀(postfix)表示——逆波兰式表示

<操作数> <操作数> <操作符> , 如
 $AB+$;

中缀表达式和后缀表达式

- 中缀表达式：运算符在两个运算数中间的表达式。如：

1) $X - Y$ 2) $5 + (6 - 4 / 2) * 3$

(字母表示运算数)

- 后缀表达式：运算符紧跟在两个运算数后面的表达式。

如：

1) $XY -$ 2) $5642 / - 3 * +$

中缀到后缀

• 中缀 $5+(6-4/2)*3$

到后缀 $5642/-3*+$ 的转变

5 (6-4/2)*3 + \rightarrow 5 (6-4/2) 3* +

\rightarrow 5 6 4/2 -3*+ \rightarrow 5 6 4 2 /-3*+

后缀表达式的作用

- 去掉中缀表达式的括号
- 隐含中缀表达式的运算次序

运算符左边向右的计算，运算数则是按其右边最接近运算符的先算的次序，运算结果放回原处：

$$\begin{array}{c} 56 \underbrace{42}_{2} / - 3 * + \\ \underbrace{\quad\quad\quad}_4 \\ \underbrace{\quad\quad\quad}_{12} \\ \underbrace{\quad\quad\quad}_{17} \end{array}$$

后缀表达式的作用

计算过程

$$\begin{array}{c} 5 \ 6 \ 42 \ / \ - \ 3 \ * \ + \\ \underbrace{\hspace{1.5cm}} \\ 2 \\ \underbrace{\hspace{1.5cm}} \\ 4 \\ \underbrace{\hspace{1.5cm}} \\ 12 \\ \underbrace{\hspace{1.5cm}} \\ 17 \end{array}$$

$$\begin{array}{l} 5 \ 6 \ \underline{4 \ 2 \ /} \ - \ 3 \ * \ + \qquad 5 \ 6 \ \underline{2 \ -} \ 3 \ * \ + \\ 5 \ \underline{4 \ 3 \ *} \ + \qquad 5 \ \underline{12 \ +} \end{array}$$

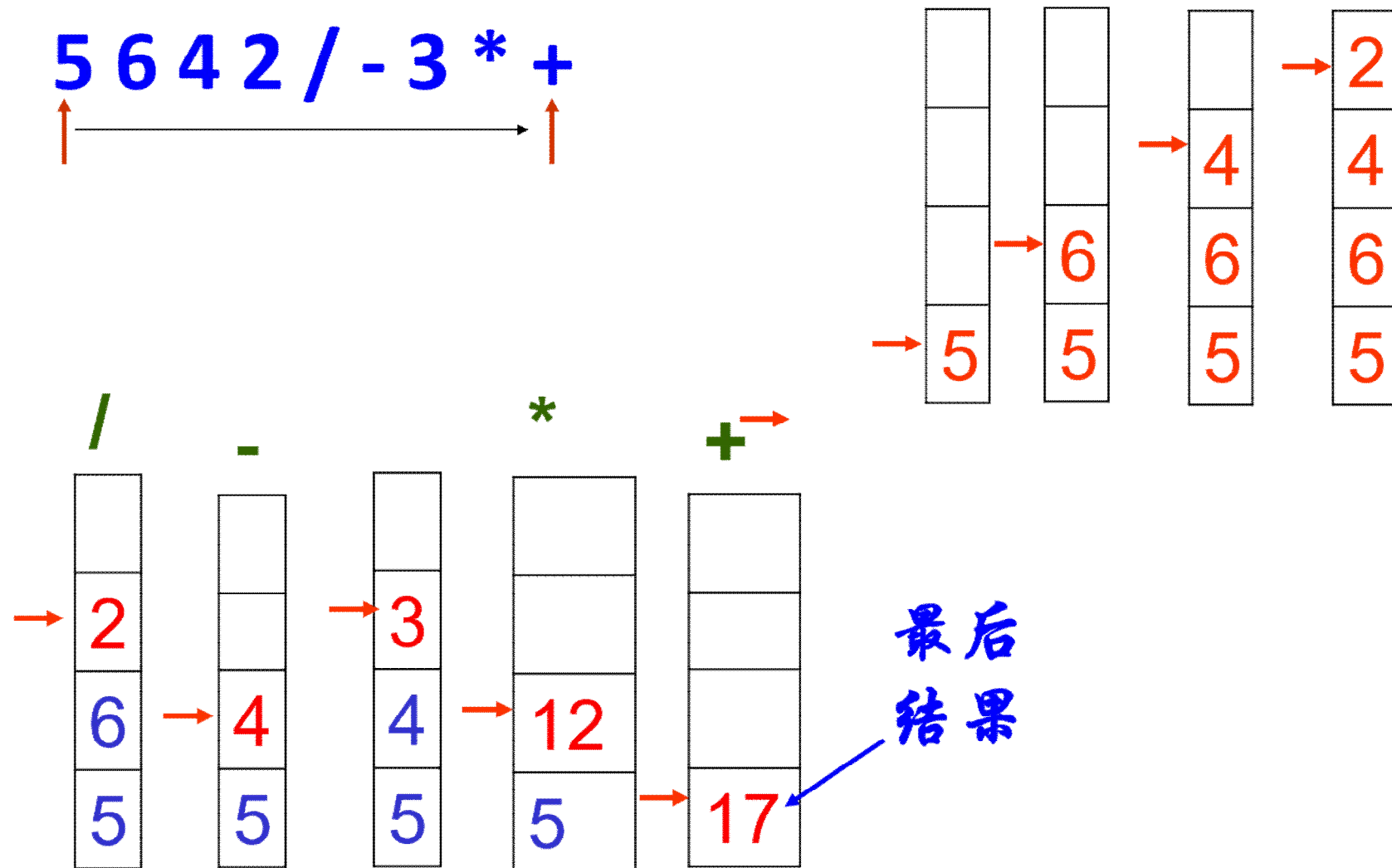
17 ← 最后结果

应用后缀表示计算表达式的值

- 从左向右顺序地扫描表达式，并用一个栈暂存扫描到的操作数或计算结果。
- 在后缀表达式的计算顺序中已隐含了加括号的优先次序，括号在后缀表达式中不出现。
- 计算例

$$\begin{array}{ccccccc} 5 & 6 & 42 & / & - & 3 & * & + \\ & & \underbrace{\hspace{1.5cm}} & & & & & \\ & & 2 & & & & & \\ & & \underbrace{\hspace{1.5cm}} & & & & & \\ & & 4 & & & & & \\ & & \underbrace{\hspace{1.5cm}} & & & & & \\ & & 12 & & & & & \\ \underbrace{\hspace{2cm}} & & \underbrace{\hspace{1cm}} & & & & & \\ & & 17 & & & & & \end{array}$$

利用栈的计算后缀表达式的过程



中缀表达式转换为后缀表达式

- 建立运算符栈,并向栈底压入# (若表达式以#结束)
- 从左向右依次读入表达式
- 如果是运算数, 则输出
- 如果是操作符则按下面操作
 - 如果栈外运算符优先级高于栈顶元素优先级,栈外运算符入栈
 - 如果栈外运算符优先级低于栈顶元素优先级,则栈顶运算符出栈输出,直至栈顶运算符优先级低于栈外运算符, 栈外运算符如栈
 - 当栈外为), 栈内运算符退至(为止
 - 当栈外为#, 栈内运算符退至#为止

算符优先关系

- 算符——运算符和界限符的统称，它们构成的集合命名为OP。
- 根据前述算术四则运算三条规则，在运算的每一步中，任意两个相继出现的算符 $op1$ 和 $op2$ 之间的优先关系至多是下面三种关系之一：

$op1 < op2$ $op1$ 的优先权低于 $op2$

$op1 = op2$ $op1$ 的优先权等于 $op2$

$op1 > op2$ $op1$ 的优先权高于 $op2$

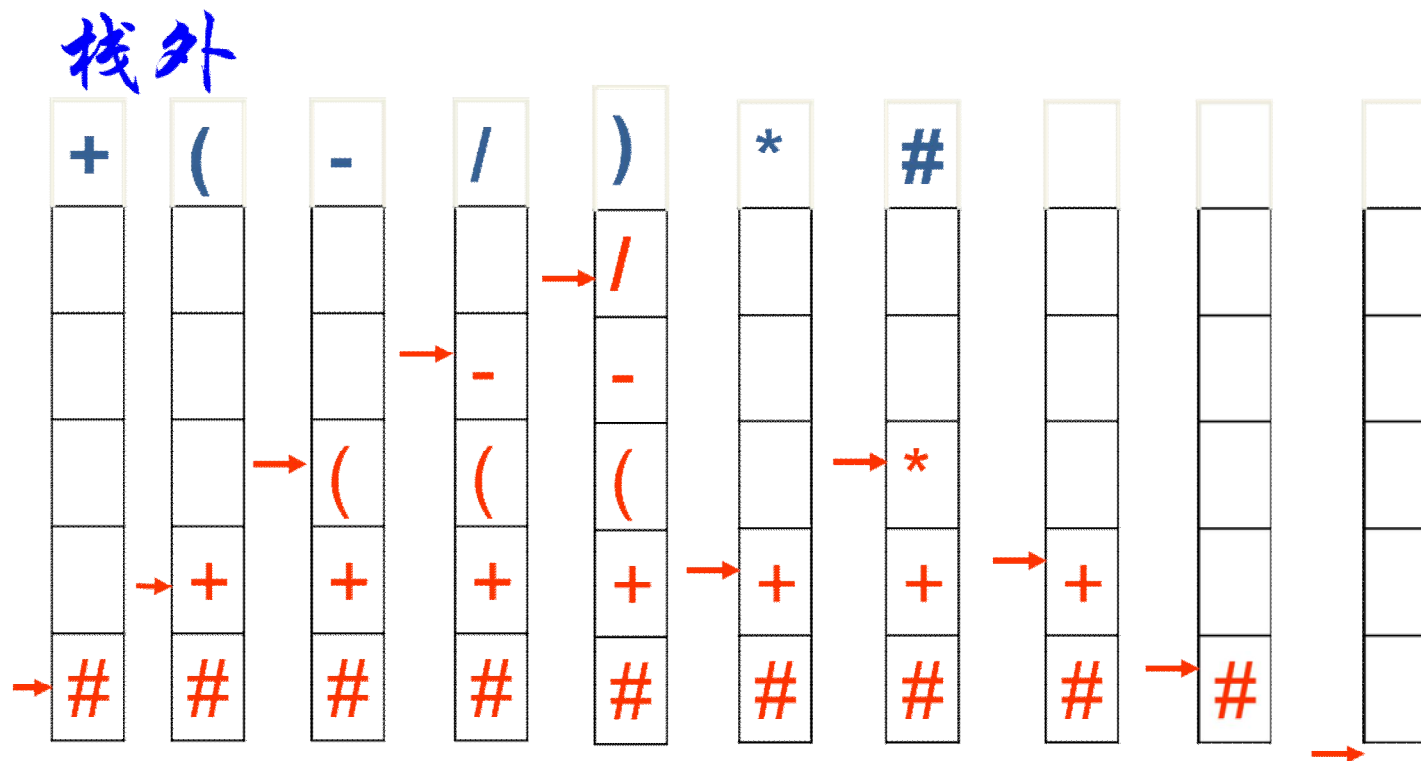
算符间的优先级关系

算符 θ_1 在算符 θ_2 前面。在算法中，相对应于 θ_1 在栈内， θ_2 在栈外

	+	-	*	/	()	#
<div> <div>θ_2</div> <div>θ_1</div> <div>+</div> </div>	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

利用栈的转换过程

$5 + (6 - 4 / 2) * 3 \#$ 5 6 4 2 / - 3 * +
↑ ————— ↑
输出



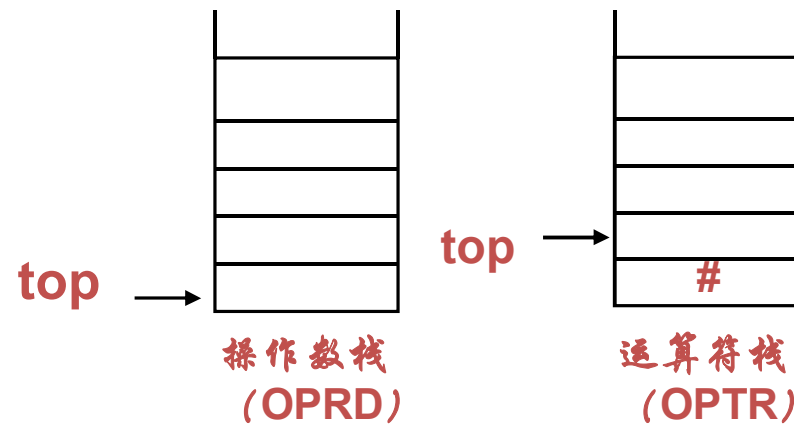
中缀表达式求值算法

将前面中缀表达式转后缀表达式，及后缀表达式计算两过程结合起来，利用所给的+、-、*、/、（、）、和#的算术运算符间的优先级的关系，可计算中缀表达式。

设置两个栈：

- (1) 操作数栈（OPRD）存放处理表达式过程中的操作数
- (2) 运算符栈（OPTR）存放处理表达式过程中的运算符

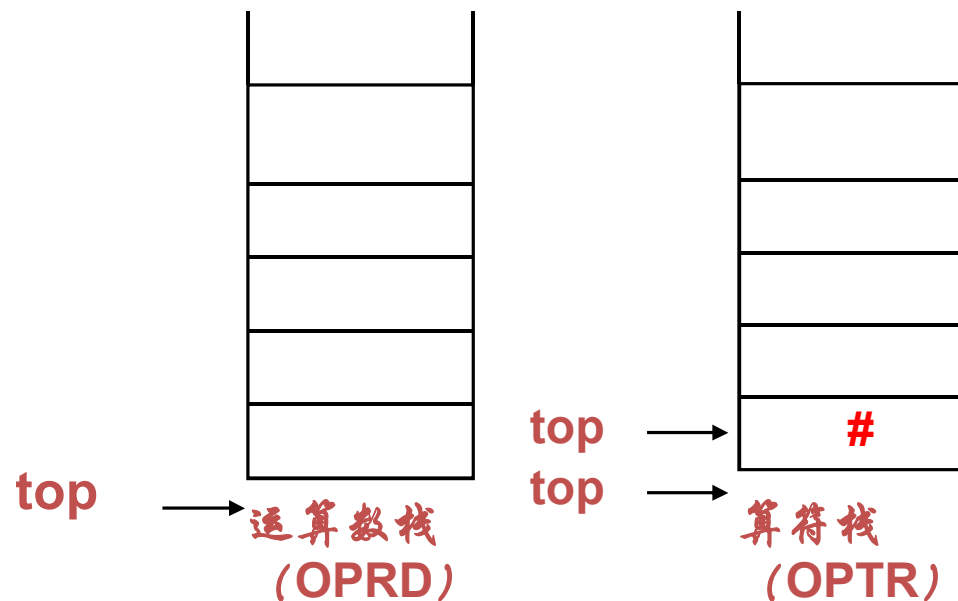
首先在运算符栈中先在栈底压入一个表达式的结束符“#”。



表达式求值算法

计算表达式: $5+(6-4/2)*3\#$ 的过程。

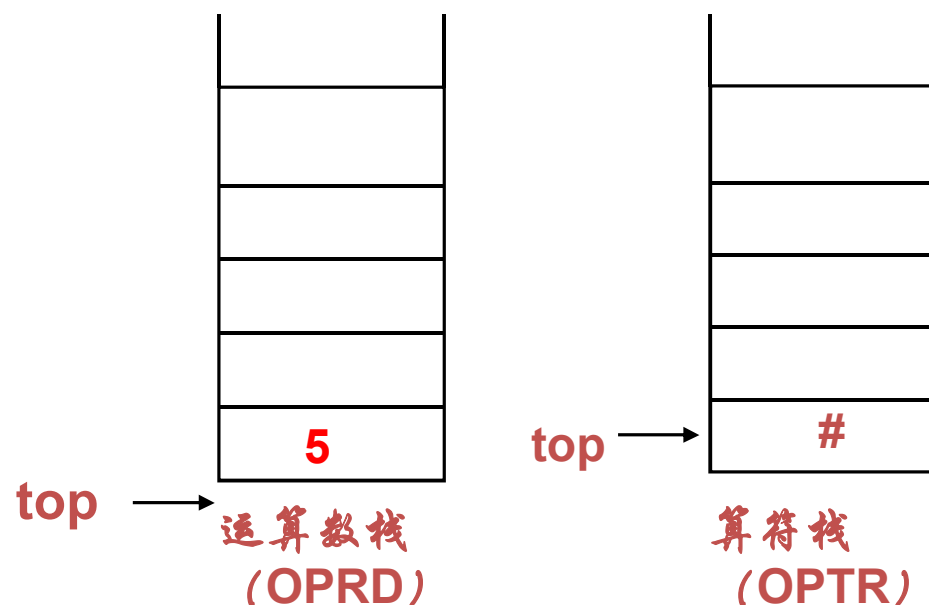
↑
◆首先在运算符栈中先在栈底压入一个表达式的结束符“#”。



表达式求值算法

计算表达式: $5+(6-4/2)*3\#$ 的过程。

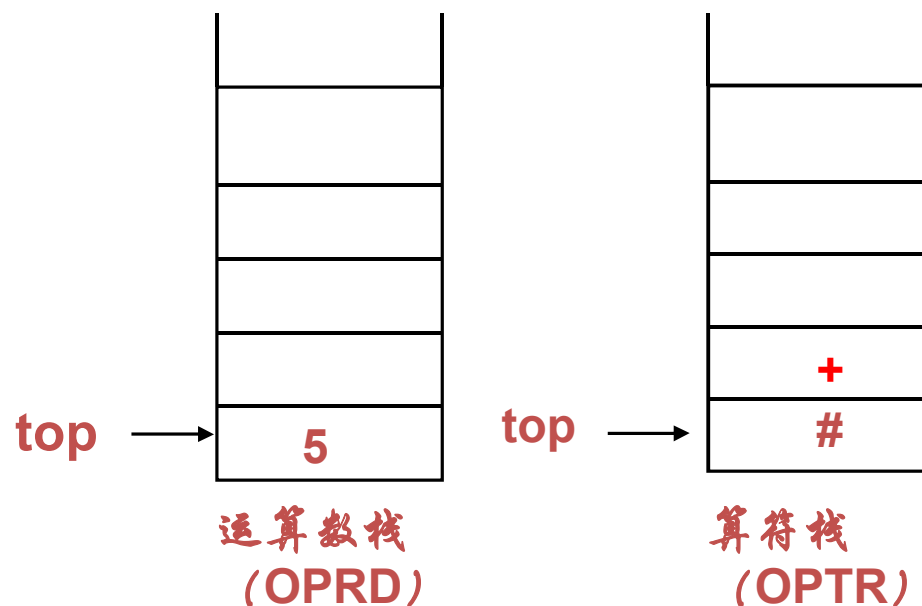
◆ 读取表达式第一个字符，是数，压入运算数栈



表达式求值算法

计算表达式: $5+(6-4/2)*3\#$ 的过程。

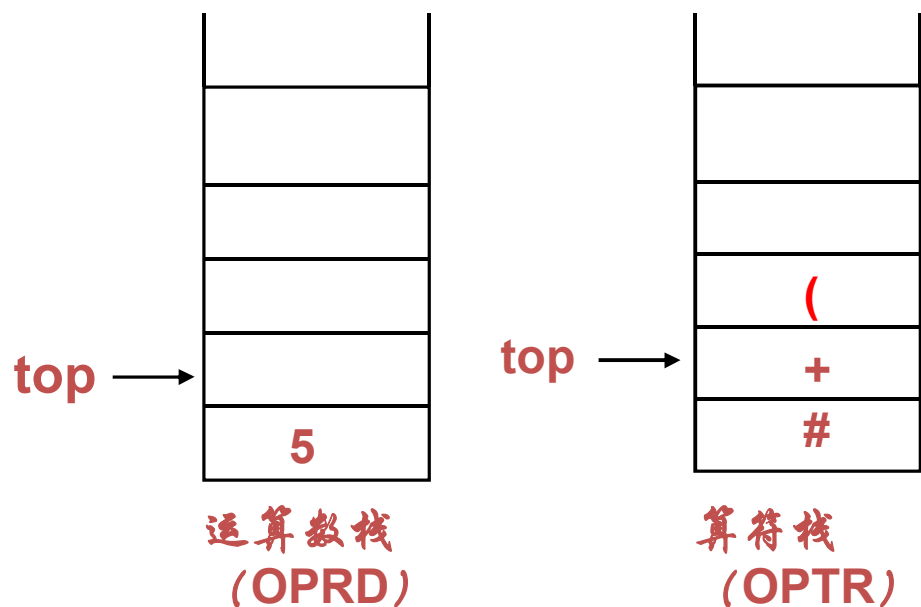
- ↑
- ◆ 读取表达式第2个字符 “+”，
 - ◆ 优先级: “+” > “#”，压入算符栈



表达式求值算法

计算表达式: $5 + (6 - 4 / 2) * 3 \#$ 的过程。

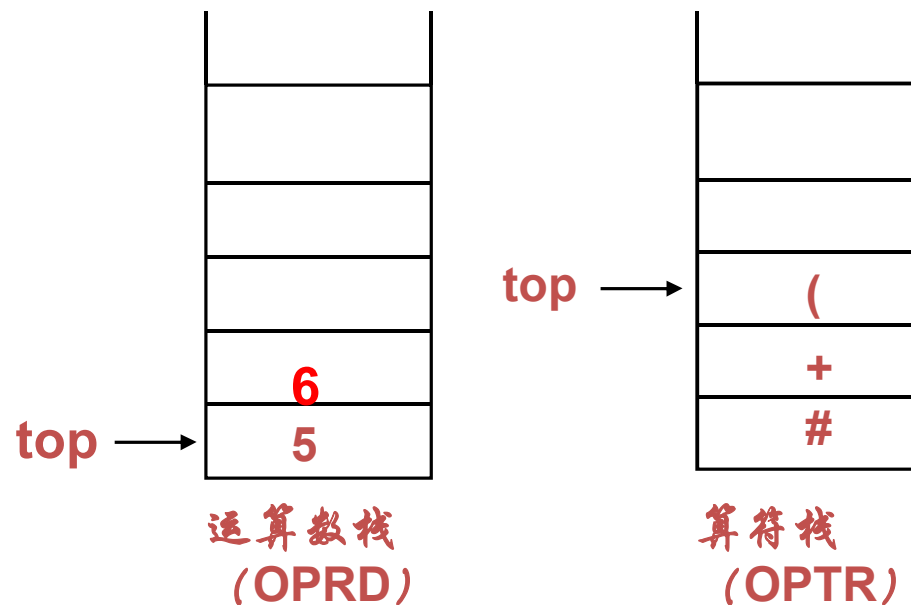
- ◆ 读取表达式第3个字符 “(”
- ◆ 优先级: 在栈外 “(” > “+”, 压入算符栈



表达式求值算法

计算表达式: $5 + (6 - 4 / 2) * 3 \#$ 的过程。

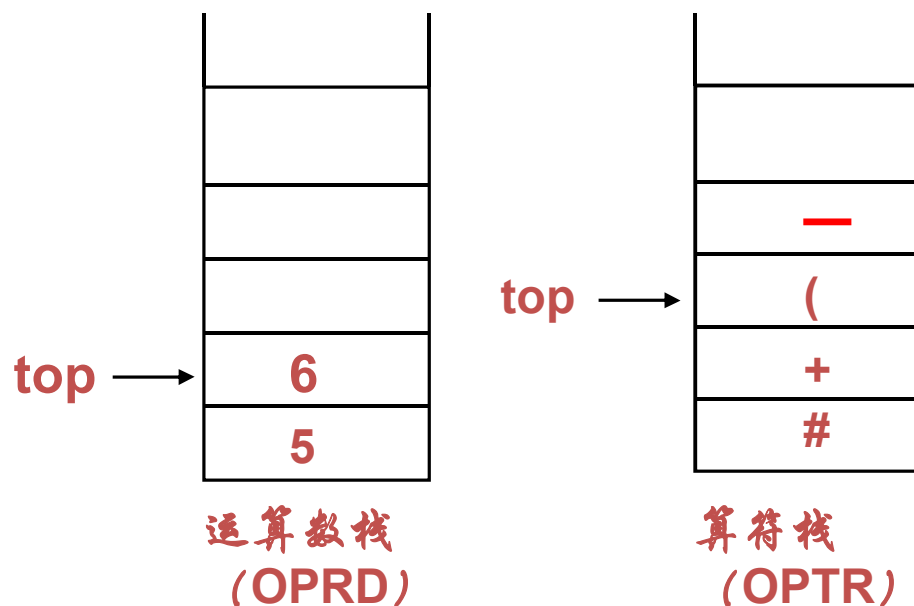
- ◆ 读取表达式第4个字符 “6”
- ◆ 压入运算数栈



表达式求值算法

计算表达式: $5 + (6 - 4 / 2) * 3 \#$ 的过程。

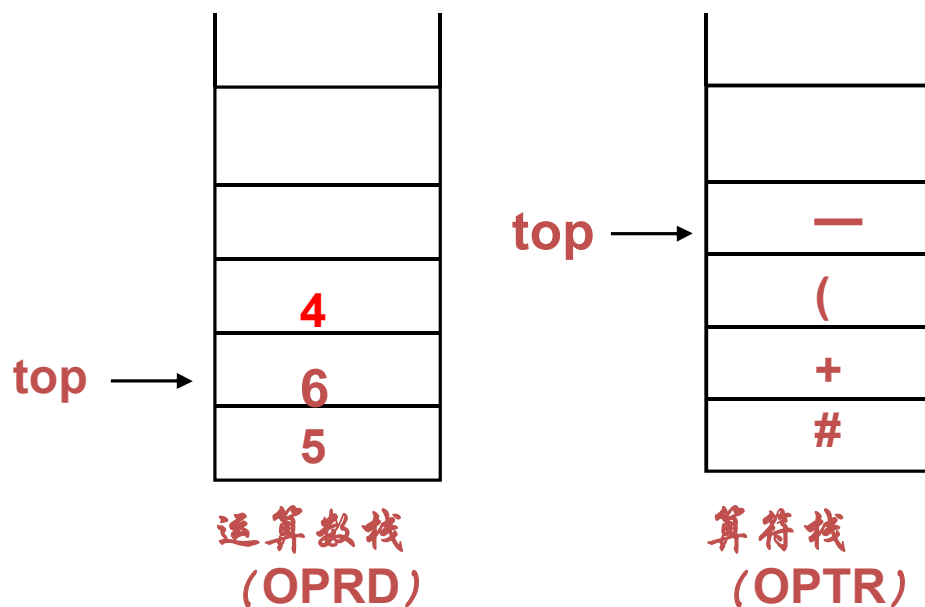
- ◆ 读取表达式第5个字符 “-”
- ◆ 优先级: 在栈外 “-” $>$ 栈内 “(”, 压入算符栈



表达式求值算法

计算表达式: $5 + (6 - 4 / 2) * 3 \#$ 的过程。

- ◆ 读取表达式第6个字符 “4”
- ◆ 压入运算数栈



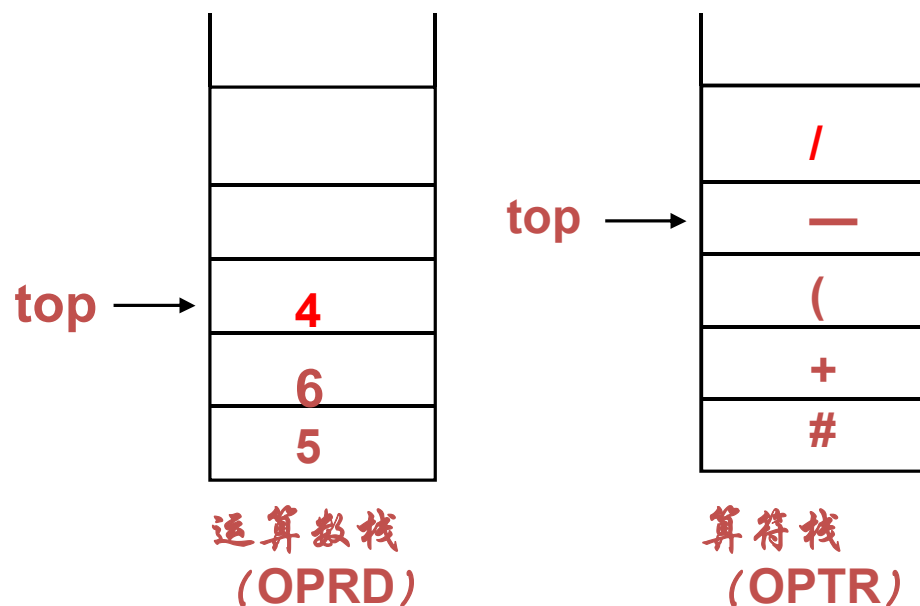
表达式求值算法

计算表达式: $5 + (6 - 4 / 2) * 3 \#$ 的过程。



◆ 读取表达式第7个字符 “/”

◆ 优先级: 在栈外 “/” > 栈内 “-”, 压入算符栈



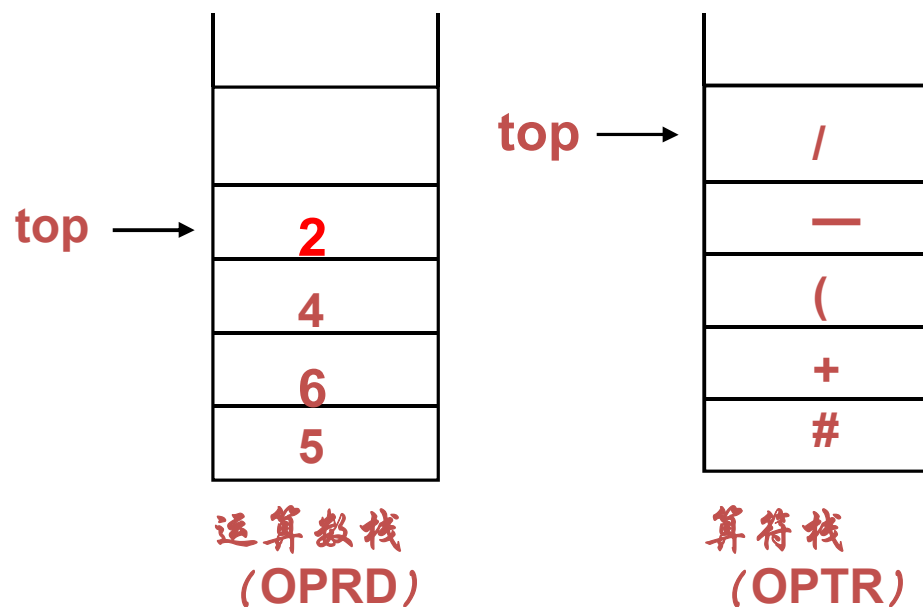
表达式求值算法

计算表达式: $5 + (6 - 4 / 2) * 3 \#$ 的过程。



◆ 读取表达式第8个字符 “2”

◆ 压入运算数栈



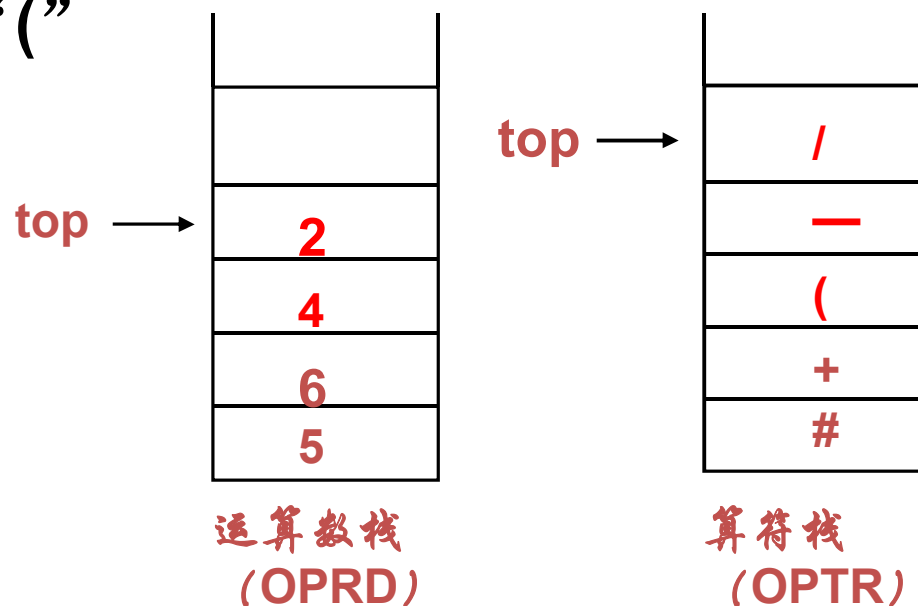
表达式求值算法

计算表达式: $5 + (6 - 4 / 2) * 3 \#$ 的过程。



◆ 读取表达式第9个字符 “)”

◆ 优先级: 在栈外 “)” < 栈内任何算法, 算符出栈计算, 直至 “(”

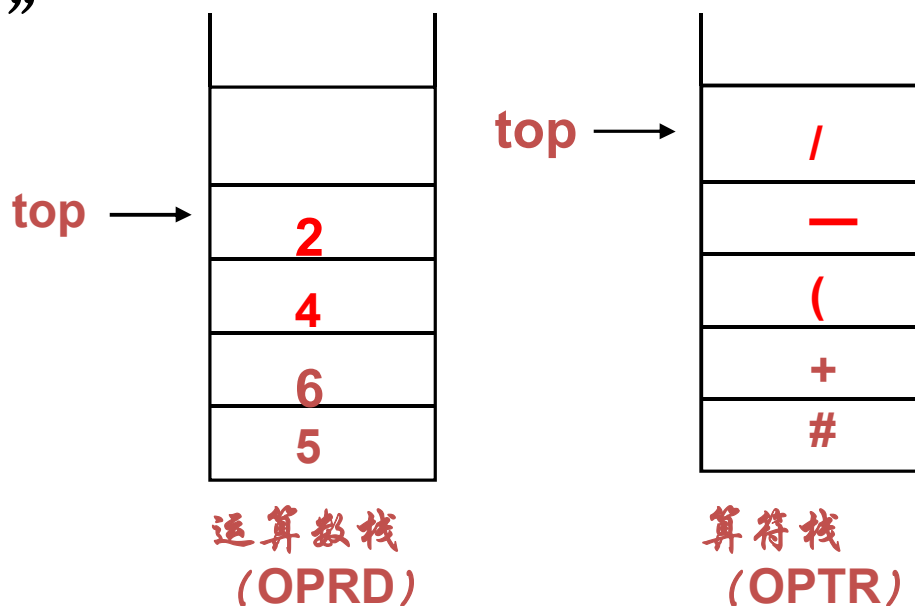


表达式求值算法

计算表达式: $5 + (6 - 4 / 2) * 3 \#$ 的过程。

◆ 读取表达式第9个字符 “)”

◆ 优先级: 在栈外 “)” < 栈内任何算法, 算符出栈计算, 直至 “(”



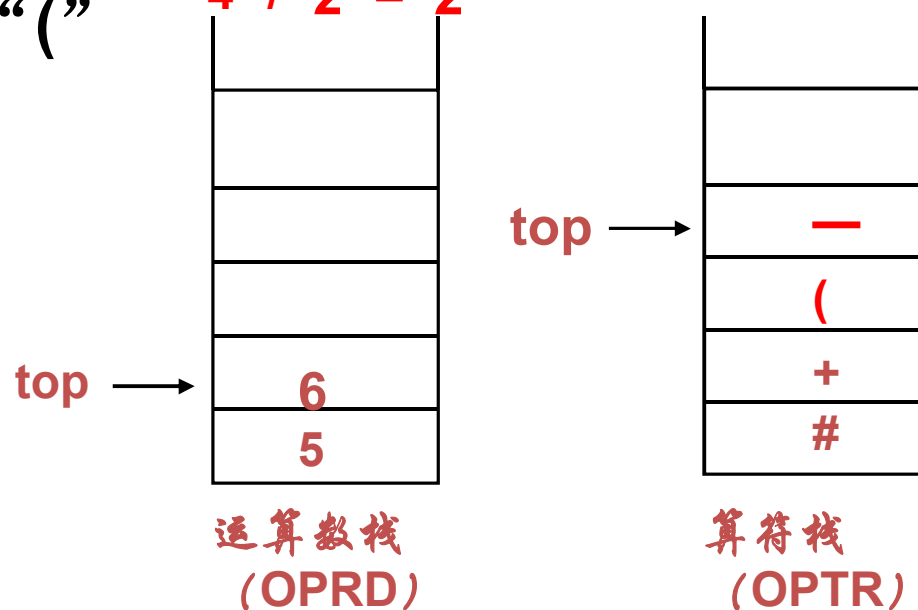
表达式求值算法

计算表达式: $5 + (6 - 4 / 2) * 3 \#$ 的过程。



◆ 读取表达式第9个字符 “)”

◆ 优先级: 在栈外 “)” < 栈内任何算法, 算符出栈计算, 直至 “(”



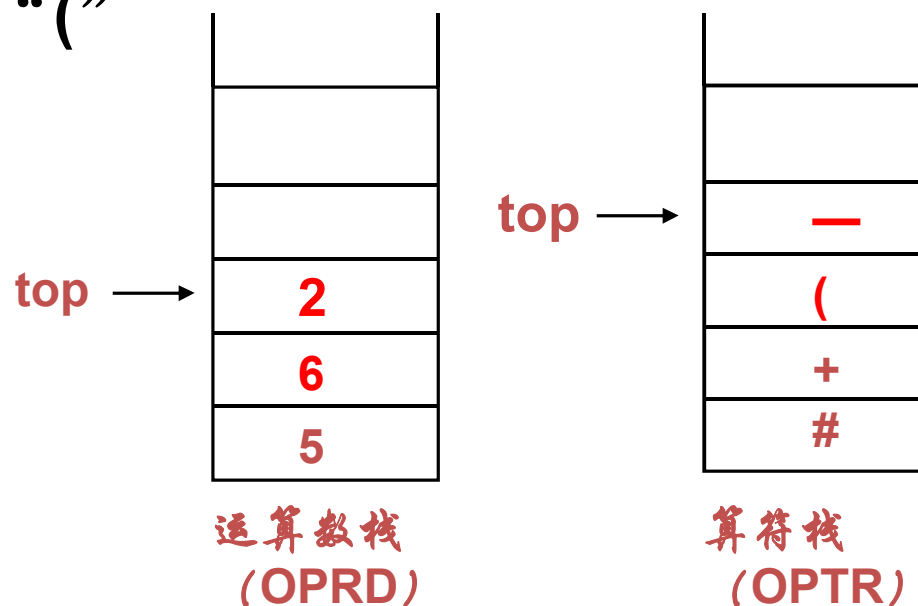
表达式求值算法

计算表达式: $5 + (6 - 4 / 2) * 3 \#$ 的过程。



◆读取表达式第9个字符 “)”

◆优先级: 在栈外 “)” < 栈内任何算法, 算符出栈计算, 直至 “(”

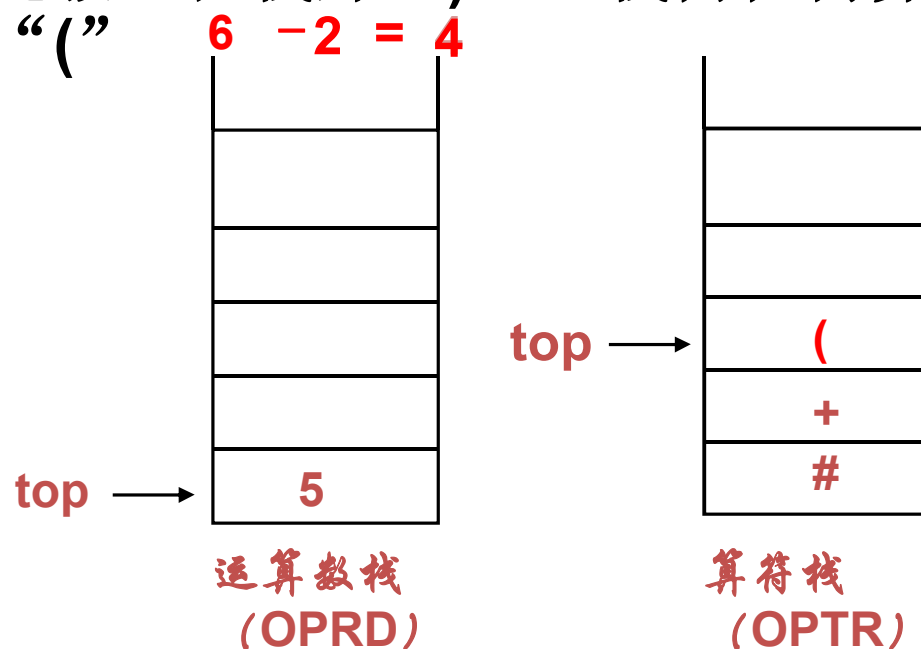


表达式求值算法

计算表达式: $5 + (6 - 4 / 2) * 3 \#$ 的过程。

◆ 读取表达式第9个字符 “)”

◆ 优先级: 在栈外 “)” < 栈内任何算法, 算符出栈计算, 直至 “(”



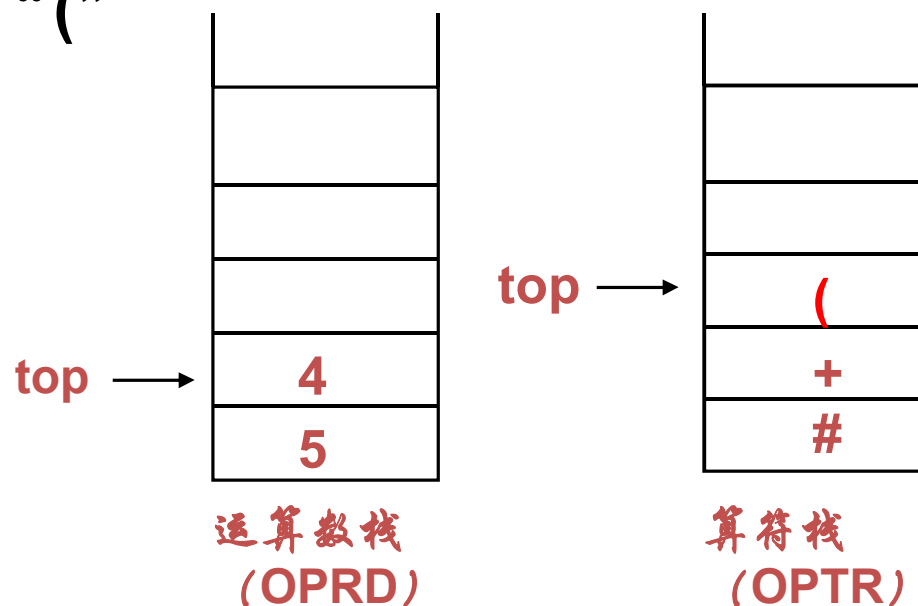
表达式求值算法

计算表达式: $5 + (6 - 4 / 2) * 3 \#$ 的过程。



◆ 读取表达式第9个字符 “)”

◆ 优先级: 在栈外 “)” < 栈内任何算法, 算符出栈计算, 直至 “(”



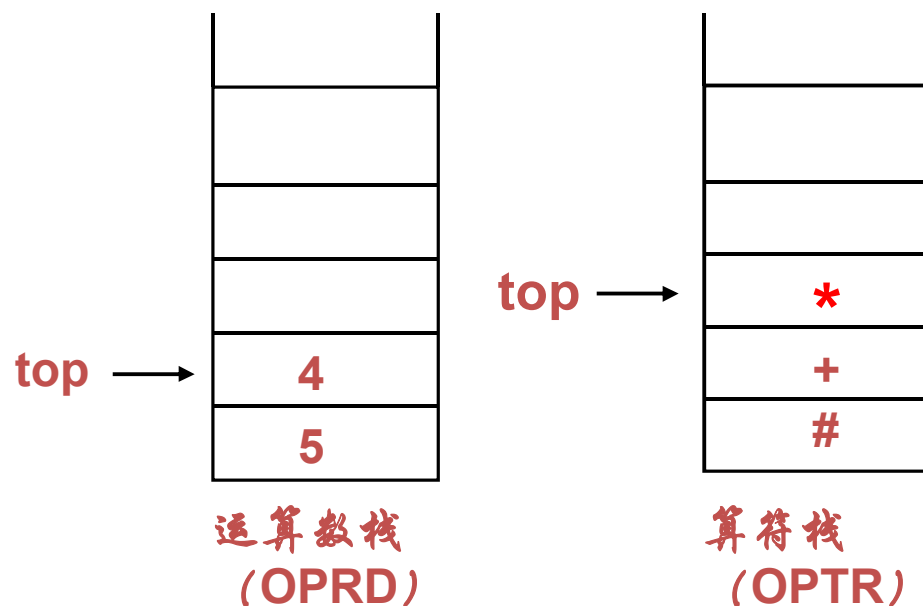
表达式求值算法

计算表达式: $5 + (6 - 4 / 2) * 3 \#$ 的过程。



◆ 读取表达式第10个字符 “*”

◆ 优先级: 在栈外 “*” > 栈内 “+”, 算符 “*” 入栈



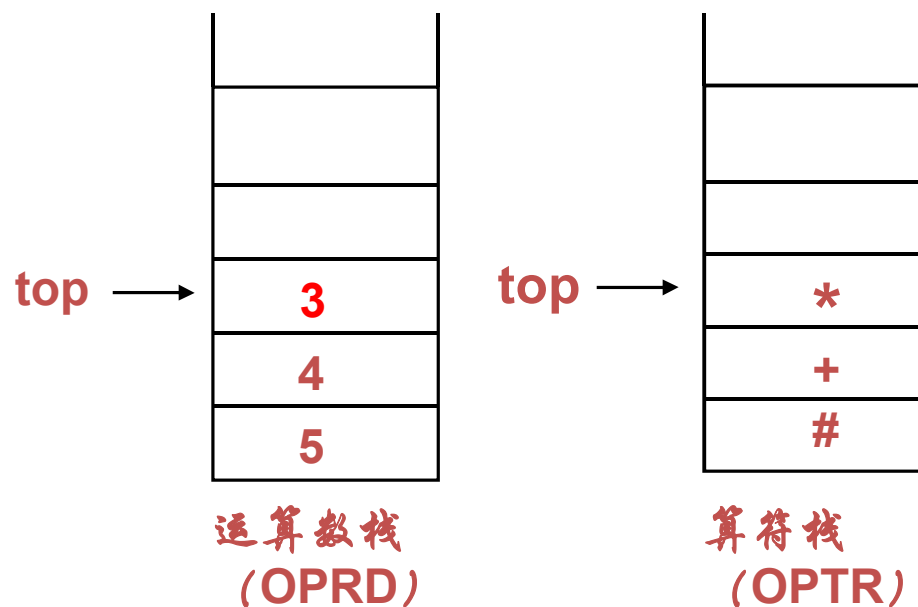
表达式求值算法

计算表达式: $5 + (6 - 4 / 2) * 3 \#$ 的过程。



◆ 读取表达式第11个字符 “3”

◆ 运算数 “3” 入数栈



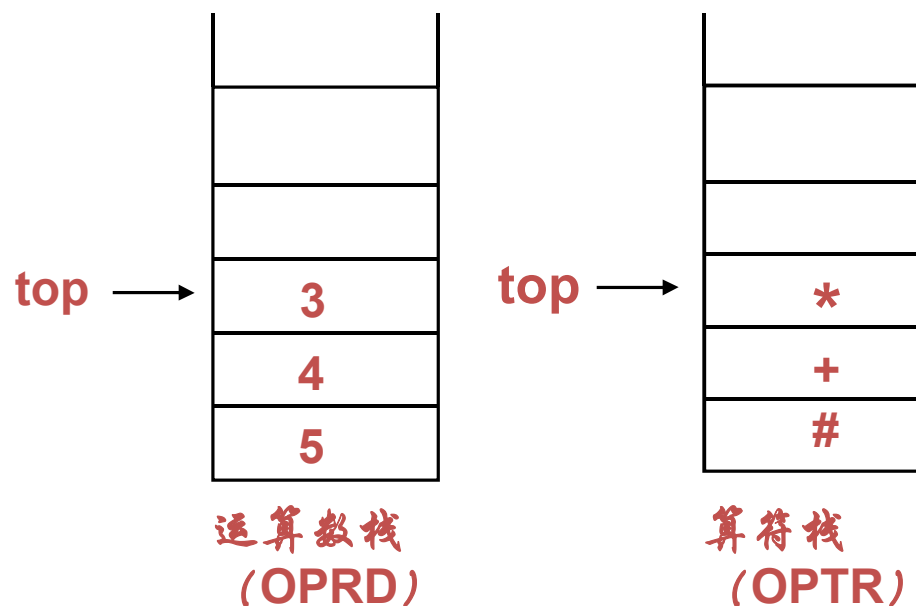
表达式求值算法

计算表达式: $5 + (6 - 4 / 2) * 3 \#$ 的过程。



◆ 读取表达式第12个字符 “#”

◆ 优先级: 在栈外 “#” < 栈内 “*”, “*” 出栈计算



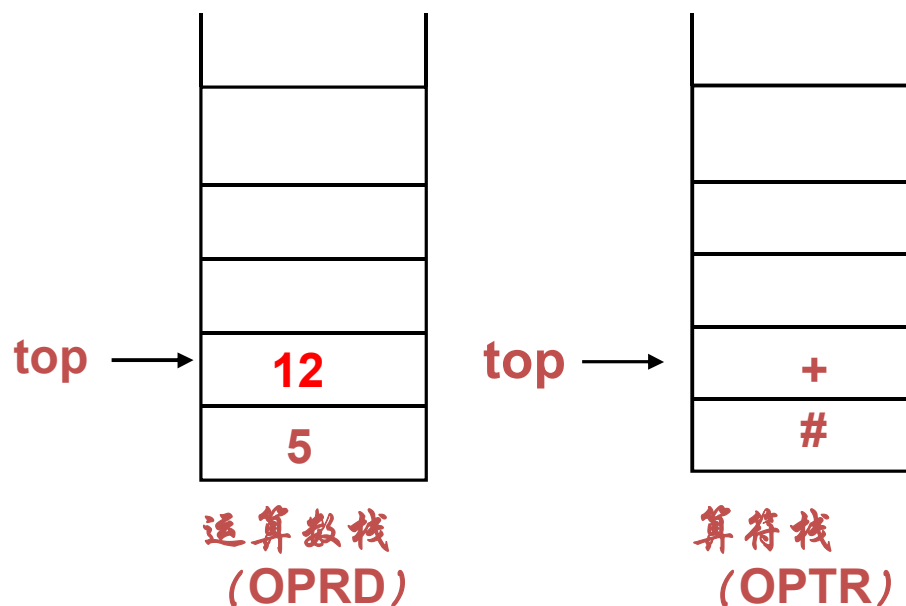
表达式求值算法

计算表达式: $5 + (6 - 4 / 2) * 3 \#$ 的过程。



◆ 读取表达式第12个字符 “#”

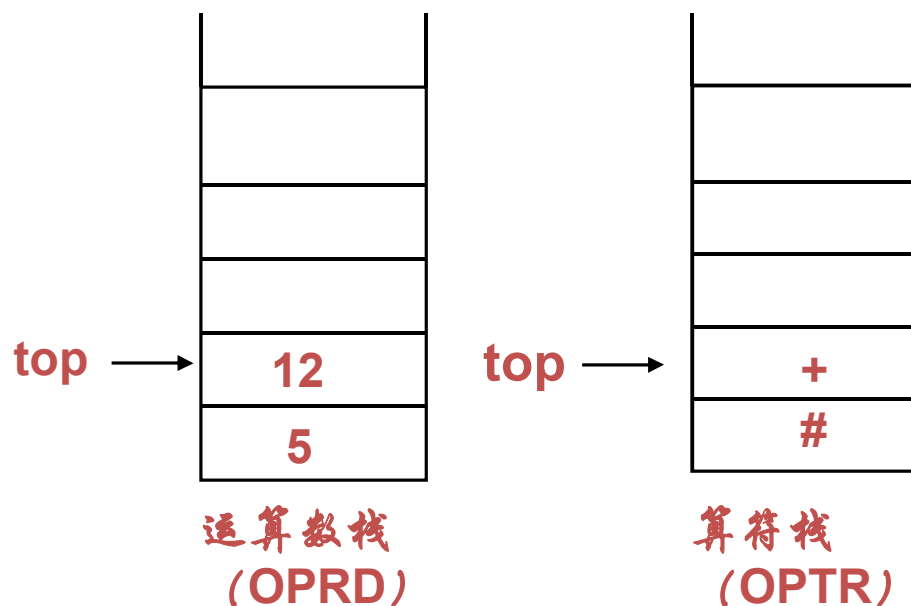
◆ 优先级: 在栈外 “#” \leq 栈内 “*”, “*” 出栈计算



表达式求值算法

计算表达式: $5 + (6 - 4 / 2) * 3 \#$ 的过程。

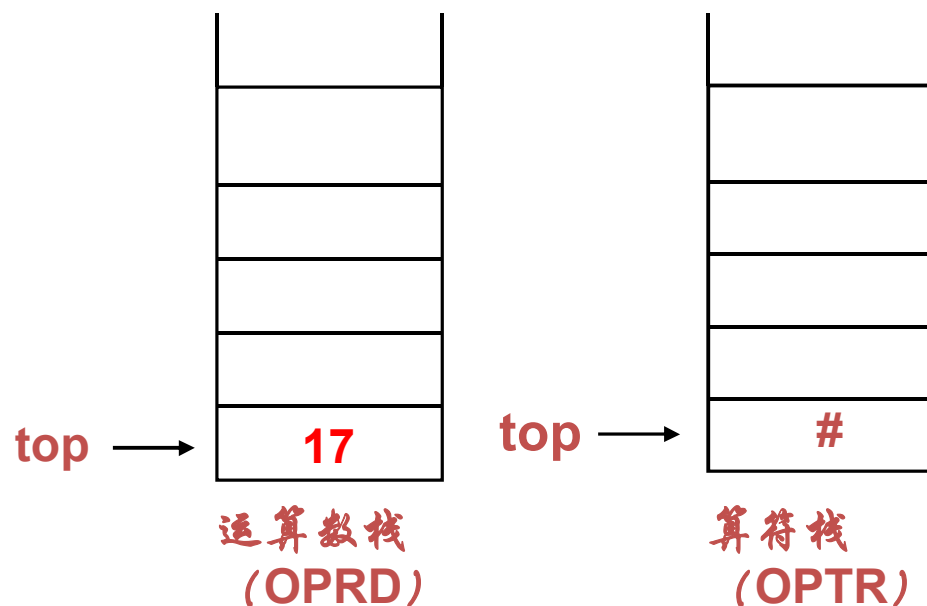
- ◆继续将表达式第12个字符“#”与**算符栈顶**字符比较
- ◆优先级: 在栈外“#” < 栈内“+”, “+”出栈计算



表达式求值算法

计算表达式: $5 + (6 - 4 / 2) * 3 \#$ 的过程。

- ◆继续将表达式第12个字符“#”与**算符栈顶**字符比较
- ◆优先级: 在栈外“#” < 栈内“+”, “+”出栈计算

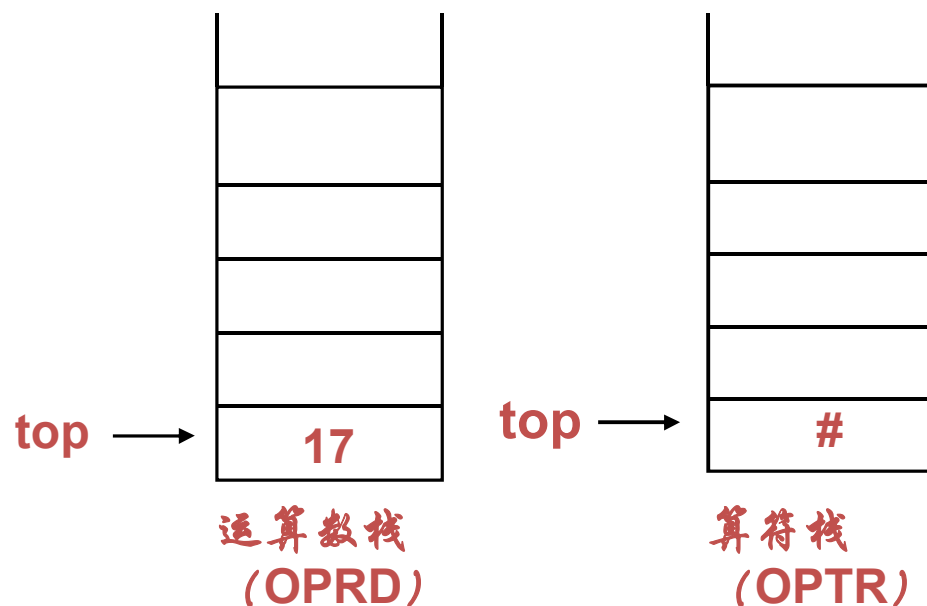


表达式求值算法

计算表达式: $5 + (6 - 4 / 2) * 3 \#$ 的过程。

◆继续将表达式第12个字符“#”与**算符栈顶**字符比较

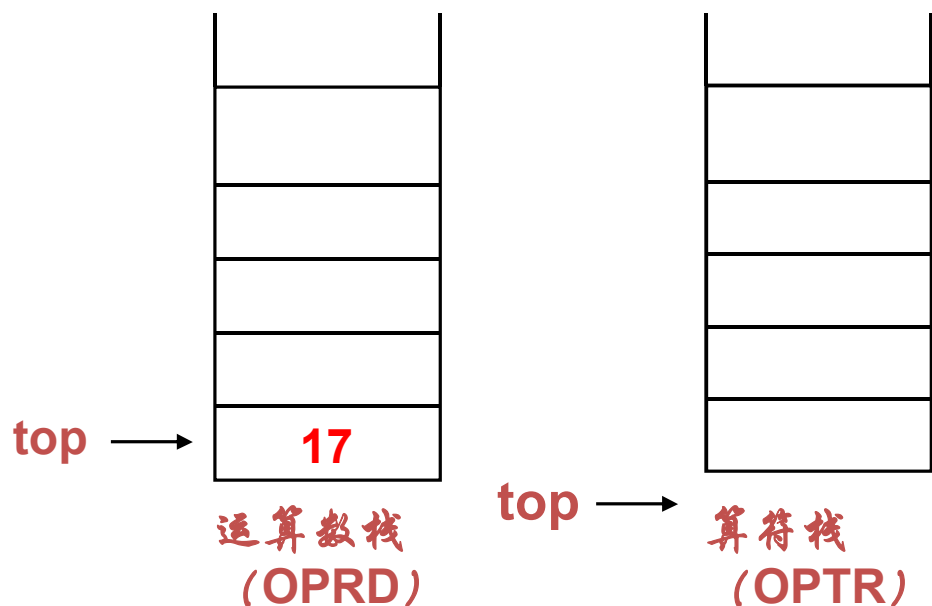
◆优先级: 在栈外“#” < 栈内“#”, “#”出栈输出结果



表达式求值算法

计算表达式: $5 + (6 - 4 / 2) * 3 \#$ 的过程。

- ◆继续将表达式第12个字符“#”与**算符栈顶**字符比较
- ◆优先级: 在栈外“#” < 栈内“#”, “#”出栈输出结果



表达式求值算法

向运算符栈的栈底压入结束符“#”，从左到右依次读出表达式中的各个符号（操作数或运算符），每读一个符号，作如下处理：

- 假如是操作数，则将其压入操作数栈，并依次读下一个符号
- 假如是运算符，则：
 - 1) 假如读出的运算符的优先级大于运算符栈栈顶运算符的优先级，则将其压入运算符栈，并依次读下一个符号
 - 2) 假如读出的是表达式结束符“#”，且运算符栈栈顶的运算符也为“#”，则表达式处理结束，最后的表达式的计算结果在操作数栈的栈顶位置
 - 3) 假如读出的是“(”，则将其压入运算符栈。
 - 4) 假如读出的是“)”，则：
 - A) 若运算符栈栈顶不是“(”，则从操作数栈连续退出两个操作数，从运算符栈中退出一个运算符，然后作相应的运算，并将运算结果压入操作数栈，然后继续执行A。注意顺序是a θ b,其中b比a先出栈)
 - B) 若运算符栈栈顶为“(”，则从运算符栈退出“(”，依次读下一个符号。
 - 5) 假如读出的运算符的优先级不大于运算符栈栈顶运算符的优先级，则从操作数栈连续退出两个操作数，从运算符栈中退出一个运算符，然后作相应的运算，并将运算结果压入操作数栈。此时读出的运算符下次重新考虑（即不读入下一个符号）

谢谢！

