



5.1

栈 Stacks

郝家胜

hao@uestc.edu.cn

自动化工程学院



内容回顾

- 线性表的概念
- 线性表的抽象数据类型
- 线性表的实现

线性表的抽象数据类型

● 数学模型

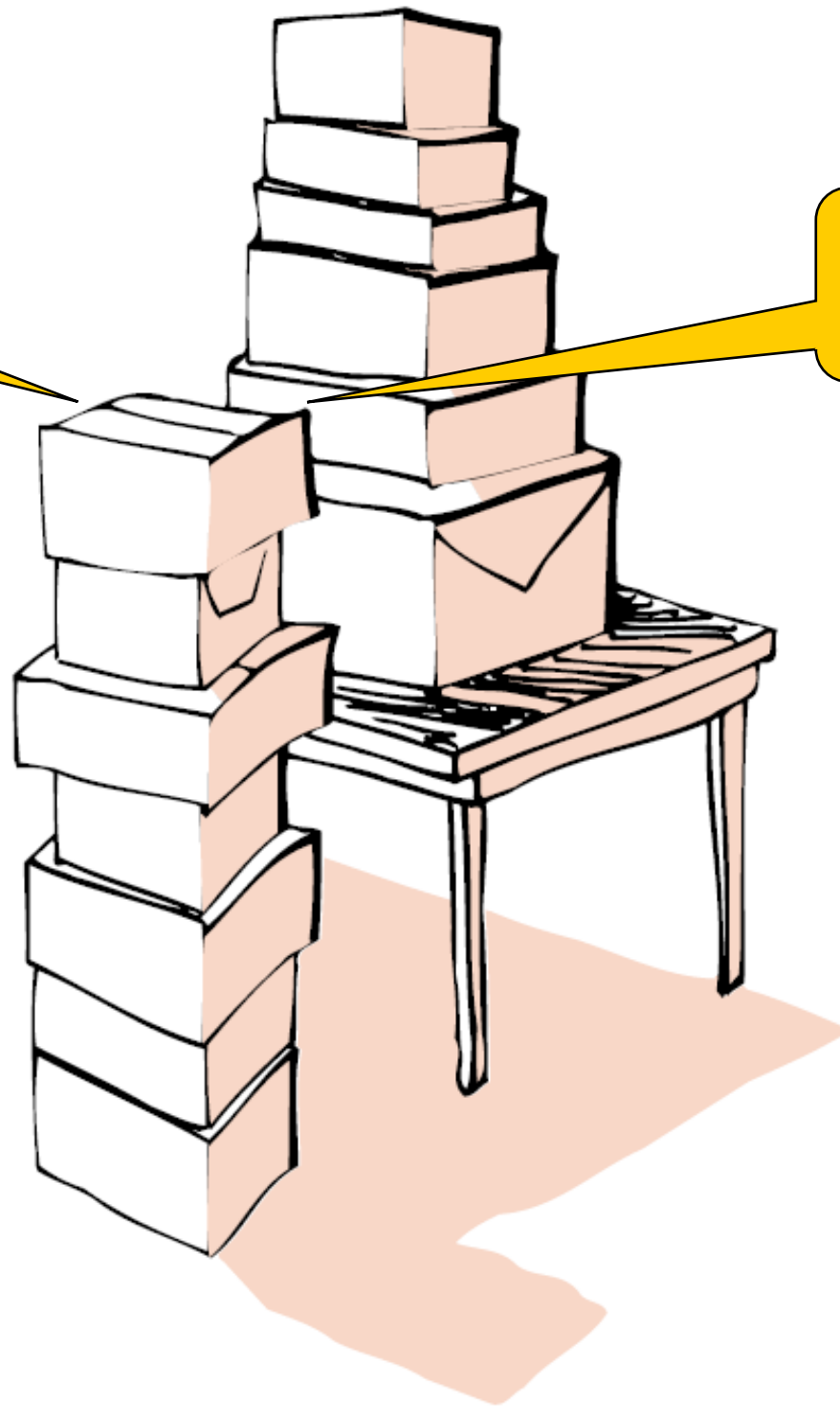
$L: (a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$

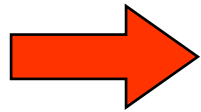
● 接口声明

- ▶ `LENGTH (L)`
- ▶ `GET (L, i)`
- ▶ `INSERT (L, i, x)`
- ▶ `DELETE (L, i)`

“取”操作必须在这端进行

“放”操作必须在同一端进行



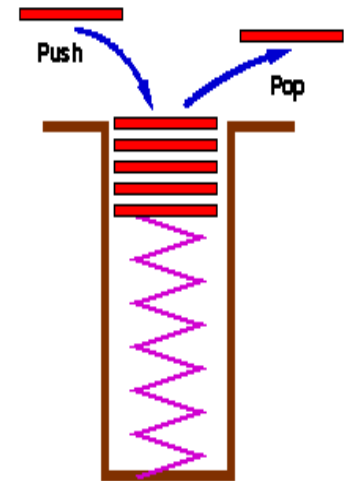


- 栈的ADT定义
- 栈的应用举例
- 栈的表示和实现
- 栈与递归的实现

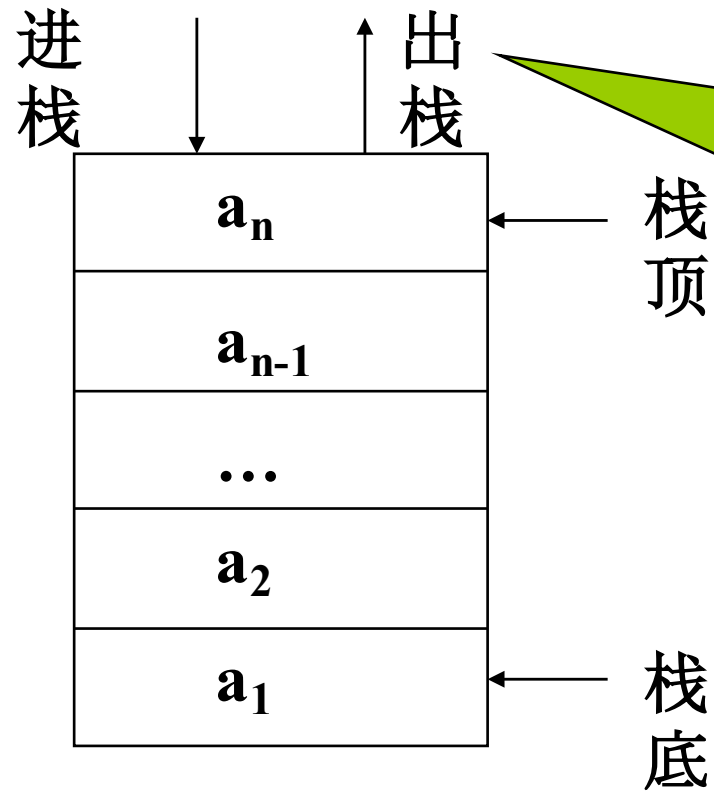
- 队列的ADT定义
- 链队列
- 循环队列

栈的概念

- 只能在**同一端**进行插入和删除的线性表；
- 允许插入和删除的一端称为**栈顶** (*top*), 另一端称为**栈底** (*bottom*)
- 特点**后进先出** (*LIFO*)



栈的示意图



栈中元素总是后进栈的元素先出来，即后进先出 (LIFO)

在一端访问的受限线性表

线性表

栈

Insert(L, **i**, x)

$1 \leq i \leq n+1$

Delete(L, **i**)

$1 \leq i \leq n$

Insert(S, **n+1**, x)

Delete(S, **n**)

栈的基本操作

- 构造栈：创建一个空栈。
- 进栈：在栈的顶部压入（插入）元素 x 。
- 出栈：若 S 不空，则弹出（删除）顶部元素。
- 取栈顶：取栈顶元素，并不改变栈中内容。
- 判空栈：若栈 S 为空返回“真”，否则返回“假”。
- 判满栈：若栈 S 为满返回“真”，否则返回“假”。

- ❖ 栈满时再有元素进栈，发生“**上溢**”，程序会中断，是一种出错状态，应尽量避免。
- ❖ 栈空时，再作退栈运算，发生“**下溢**”，下溢则可能是正常现象，常以此作为控制转移的条件。

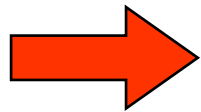
栈的抽象数据类型

● 数学模型

$S: (a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$

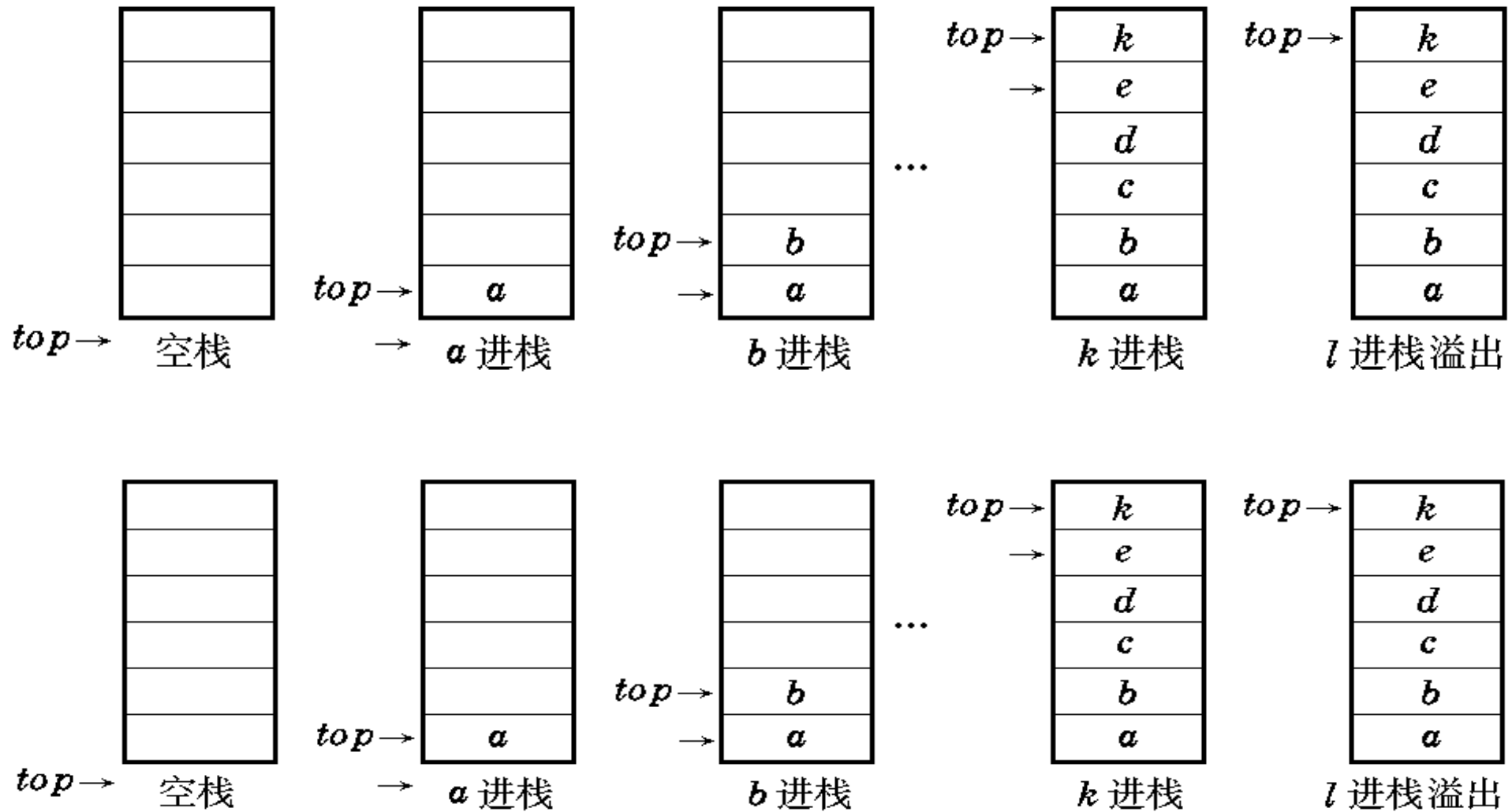
● 接口声明

- ▶ $PUSH(S, x)$: //进栈
- ▶ $POP(S)$: //出栈
- ▶ $TOP(S)$: //取栈顶元素
- ▶ $IS_EMPTY(S)$: //判栈空否
- ▶ $IS_FULL(S)$: //判栈满否

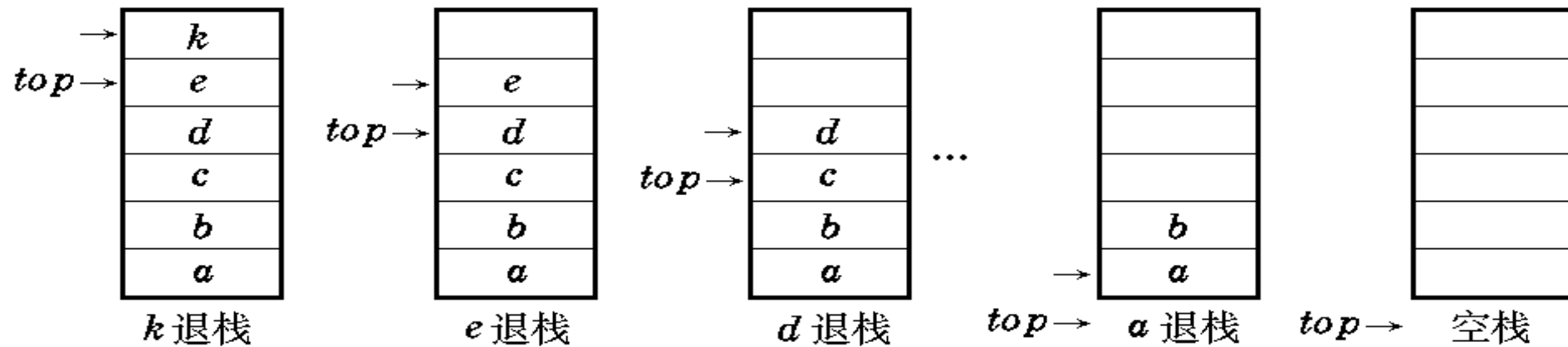
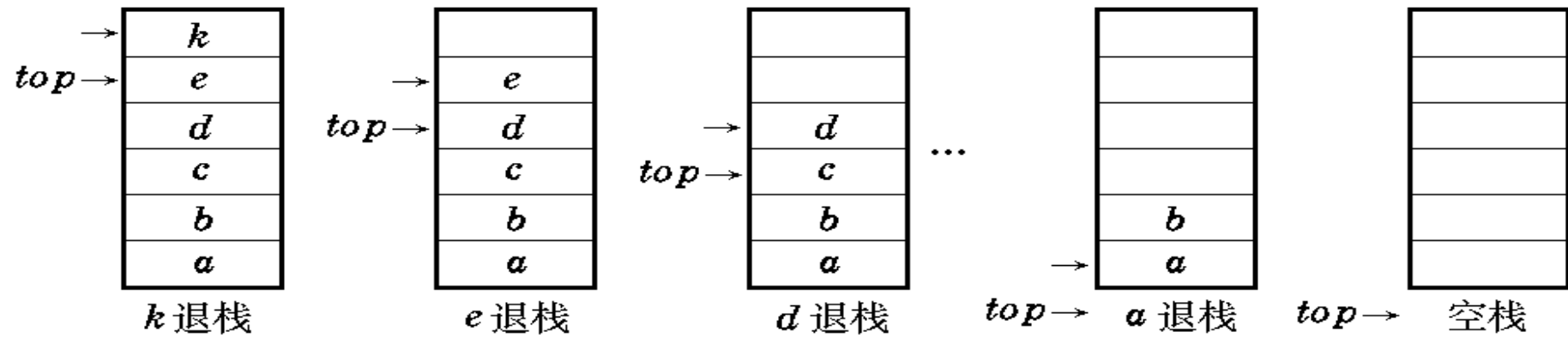


- 栈的ADT定义
- 栈的应用举例
- 栈的表示和实现
- 栈与递归的实现

- 队列的ADT定义
- 链队列
- 循环队列



进栈示例



退栈示例

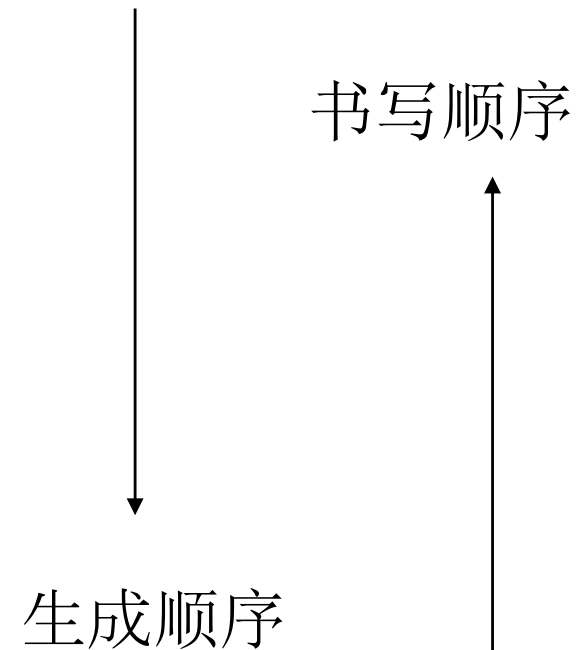
数制的转换

十进制数转换为八进制数

原理: $N = (N \text{ div } d) * d + N \text{ mod } d$

例: $(1348)_{10} = (2504)_8$

| N | N div 8 | N mod d |
|------|---------|---------|
| 1348 | 168 | 4 |
| 168 | 21 | 0 |
| 21 | 2 | 5 |
| 2 | 0 | 2 |



数值转换算法

RADIX-CONVERSION (n, d)

```
1  ▷ 将n转换为d进制表示
2   $S \leftarrow \text{CREATE\_STACK}()$ 
3  while  $n > 0$  do
4    PUSH ( $S, n \% d$ )
5     $n \leftarrow n / d$ 
6  end
7  while !IS_EMPTY( $S$ ) do
8     $n \leftarrow \text{POP}(S)$ 
9    print  $n$ 
10 end
```

使用已有的数据结构，
无需考虑实现的细节！

特点：先把一些数据
顺次记下来，之后按
照逆序处理数据。

括号匹配

考虑含有 () 和 [] 的表达式中括号的匹配。

匹配算法思想：

顺序读入表达式（忽略其他非括号字符）直至末尾。

- 如果是左括号，暂时记下来；
- 如果是右括号，它必须和最近读入的左括号匹配。如果它和最近读入的左括号匹配，则可抹去匹配的左括号，否则，匹配失败。
- 最后，如果读完表达式时匹配成功，否则，匹配失败。

算法中使用的结构特点：
先进后出，
故可以用栈实现。

例：[([)]]

[([] ())]

习题：使用栈实现括号匹配算法。

后缀表达式的计算

后缀表达式：运算符紧跟在它的两个运算数之后，又称逆波兰记法。

例如： $a b * c +$, $a b + c *$, $a b c + *$, $a b c * +$.

特点：表达式无需括号，计算方法简单。

后缀表达式的计算：

顺序从左到右读取表达式；

- 若是操作数，记下来；
- 若是运算符，则取最后的两个操作数做运算，结果记下来；

表达式读取结束时，最后记下来的结果是表达式的值。

后缀表达式的计算

上述计算过程具有“后记下来的先使用”的特点，故使用栈实现。

后缀表达式的计算：

顺序从左到右读取表达式；

- **若是操作数，则入栈；**
- **若是运算符，则取栈顶的两个操作数做运算，将结果入栈；**

表达式读取结束时，栈顶的结果是表达式的值。

中缀表达式求值

表达式的构成：操作数、运算符和界限符（后两者统称为算符）。

例： $4+2*3$

$$4 + 2*3 + (2+3) * 4$$

求值过程：

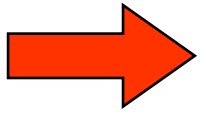
如果读入的是操作数，暂时记下来；

如果读入的是算符，则需要和前一个算符比较运算优先级；

如果当前算符优先级高，则先记下(等待它的操作数)，否则，先计算前一个算符；计算结果暂时记下。

操作数和算符均有先进后出的特点，故可用两个栈实现。

- 栈的ADT定义
- 栈的应用举例
- 栈的表示和实现
- 栈与递归的实现
- 队列的ADT定义
- 链队列
- 循环队列



顺序栈

栈的元素可以用顺序结构存储, 称为顺序栈;
也可用链式结构存储, 称为链式栈.

```
typedef struct stack_t{  
    int  *data;    // 存放栈元素的数组  
  
    int   top;     // 指向栈顶位置, 栈空时top= -1  
  
    int   depth;  // 栈的深度 (最大容量)  
} *STACK;
```

用C实现

顺序栈的构造

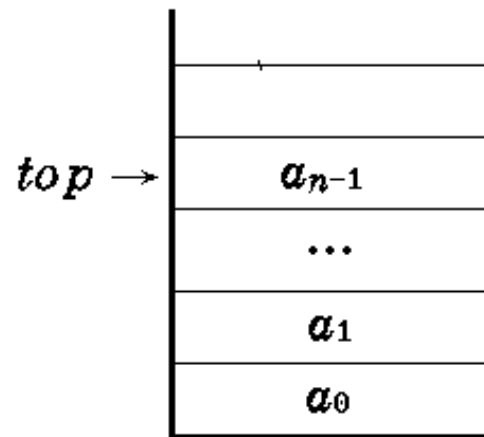
```
/* 创建一个空的顺序栈 */
STACK create_stack()
{
    STACK stack = (STACK) malloc(sizeof(struct stack_t));
    stack->data = (int*) malloc(sizeof(int) * MAX_LENGTH);
    stack->depth = MAX_LENGTH;
    stack->top = -1;

    return stack;
}

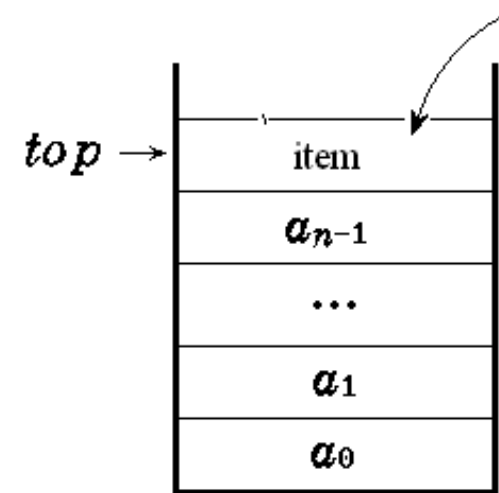
bool is_stack_empty(s)
{
    return (s->top == -1);
}

bool is_stack_full(s)
{
    return (s->top == s->depth - 1);
}
```

压栈



入栈前



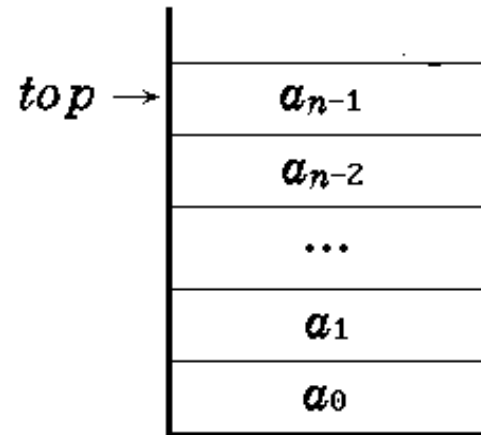
入栈后

```
int stack_push(STACK s, int x)
{
    if (is_stack_full(s)) return E_OVERFLOW;

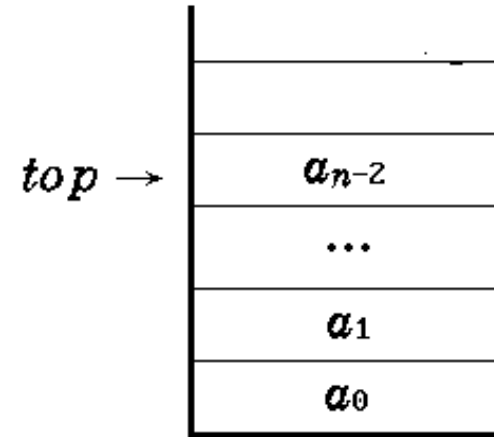
    s->top += 1;
    s->data[s->top] = x;

    return E_SUCCESS;
}
```


出栈



出栈前



出栈后

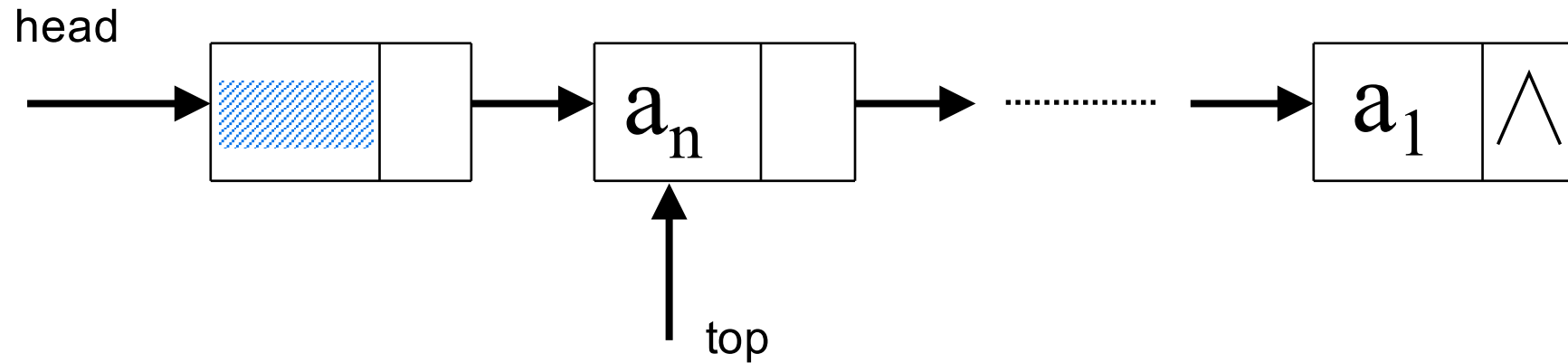
```
int stack_pop(STACK s)
{
    int x;

    if (is_stack_empty(s)) return E_UNDERFLOW;

    x = s->data[s->top];
    s->top -= 1;

    return x;
}
```

链式栈



- 链式栈无栈满问题，空间可扩充
- 链式栈的栈顶在链头
- 插入与删除仅在栈顶处执行

链式栈的类型说明

```
typedef struct node_t {  
    int data;  
    struct node_t *next;  
} *NODE;
```

```
typedef struct {  
    NODE head;  
    int length;  
} *STACK;
```

链式栈的实现

```
/* 创建一个空的链接栈 */
```

```
STACK create_stack()
```

```
{
```

```
    STACK stack ;
```

```
    stack = (STACK)malloc(sizeof(struct stack_t));
```

```
    stack->length = 0 ;
```

```
    stack->head = _create_node();
```

```
    return list;
```

```
}
```

```
bool is_stack_empty(s)
```

```
{
```

```
    return s->length == 0;
```

```
}
```

```
bool is_stack_full(s)
```

```
{
```

```
    return false;
```

```
}
```

压栈

```
int stack_push(STACK s, int x)
{
    if (is_stack_full(s)) return E_OVERFLOW;

    NODE top = s->head->next;
    NODE new_node = _create_node();
    new_node->data = x;
    new_node->next = top;
    s->head->next = new_node;

    s->length += 1;

    return E_SUCCESS;
}
```

出栈

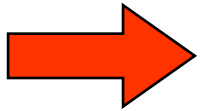
```
int stack_pop(STACK s)
{
    if (is_stack_empty(s)) return E_UNDERFLOW;

    NODE top = s->head->next;
    int x = top->data;

    s->head->next = top->next;
    s->length -= 1;
    free(top);

    return x;
}
```

- 栈的ADT定义
- 栈的应用举例
- 栈的表示和实现
- 栈与递归
- 队列的ADT定义
- 链队列
- 循环队列



递归的概念

一个函数是递归的, 如果它直接或者间接地调用了它本身.

一个问题有递归解, 如果此问题可化为较简单情况下的类似问题.

一个递归解有两部分构成:

1. 最简单情况下的解;
2. 一种将一般情况划归到较简单情况的方法.

Factorials: 一个递归定义

- 数学定义:

$$n! = n \times (n - 1) \times \cdots \times 1.$$

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n - 1)! & \text{if } n > 0. \end{cases}$$

递归基
(base case)

递归步: 将一般情况
化为较简单情况
(recursive case)

```
int factorial(int n)
```

```
/* Pre:  n is a nonnegative integer.
```

```
   Post: Return the value of the factorial of n. */
```

```
{
```

```
    if (n == 0) return 1;
```

```
    else      return n * factorial(n - 1);
```

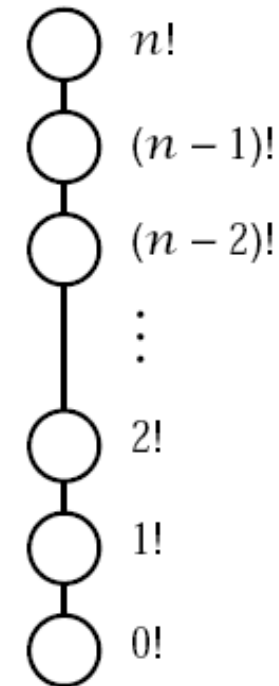
```
}
```

+ : 非常简洁, 易理解;

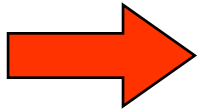
- : 需要保存一系列的计算结果.

Example:

$$\begin{aligned}
 \text{factorial}(5) &= 5 * \text{factorial}(4) \\
 &= 5 * (4 * \text{factorial}(3)) \\
 &= 5 * (4 * (3 * \text{factorial}(2))) \\
 &= 5 * (4 * (3 * (2 * \text{factorial}(1)))) \\
 &= 5 * (4 * (3 * (2 * (1 * \text{factorial}(0))))) \\
 &= 5 * (4 * (3 * (2 * (1 * 1)))) \\
 &= 5 * (4 * (3 * (2 * 1))) \\
 &= 5 * (4 * (3 * 2)) \\
 &= 5 * (4 * 6) \\
 &= 5 * 24 \\
 &= 120.
 \end{aligned}$$



- 栈
- 栈的应用举例
- 栈的实现
- 栈与递归
- 栈与函数调用
- 队列



栈与函数调用

当在一个函数的运行期间调用另一个函数时，在运行该被调用函数之前，需先完成三项任务：

- 将所有的实在参数、返回地址等信息传递给被调用函数保存；
- 为被调用函数的局部变量分配存储区；
- 将控制转移到被调用函数的入口。

从被调用函数**返回**调用函数**之前**,
应该完成下列三项任务:

- **保存**被调函数的**计算结果** ;
- **释放**被调函数的**数据区** ;
- 依照被调函数保存的返回地址将**控制转移**到调用函数。


多个函数嵌套调用的规则是：

后调用先返回！

此时的内存管理实行“**栈式管理**”

例如：

```
void main( ){    void a( ){    void b( ){  
    ...          ...          ...  
    a( );        b( );  
    ...          ...  
} //main        } // a      } // b
```



函数a的数据区
Main的数据区

函数调用的实现

- 函数调用之间的关系: 先调用的后返回;
- 调用函数时要分配数据区存放返回地址, 实在参数以及局部变量等, 称之为活动记录;
- 数据区的特点: 先分配的数据区后释放; 所以,
- 程序运行中这种数据区的分配和释放要用栈(运行栈)来实现;
- 栈顶存放的是当前运行函数的数据: 调用函数时在栈顶为它分配数据区, 返回时释放栈顶数据区.
- 递归调用的实现同理进行: 对于每个递归调用, 将相应的(不同)活动记录入栈.

小结

栈是操作受限制的线性表。

- 它具有线性结构
- 栈的插入和删除操作只能在一端进行
- 它们的存储结构可以是顺序的，也可以是链式的