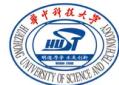


# Stack

- 有一定操作限制的线性表
- 只在一端进行插入和删除
  - Push 插入，入栈
  - Pop
  - Last in first out (LIFO)

## 栈的抽象数据类型描述



4

类型名称: 堆栈 (Stack)

数据对象集: 一个有0个或多个元素的有穷线性表。

操作集: 长度为MaxSize的堆栈  $S \in \text{Stack}$ , 堆栈元素  $\text{item} \in \text{ElementType}$

1、Stack CreateStack( int MaxSize ): 生成空堆栈, 其最大长度为MaxSize;

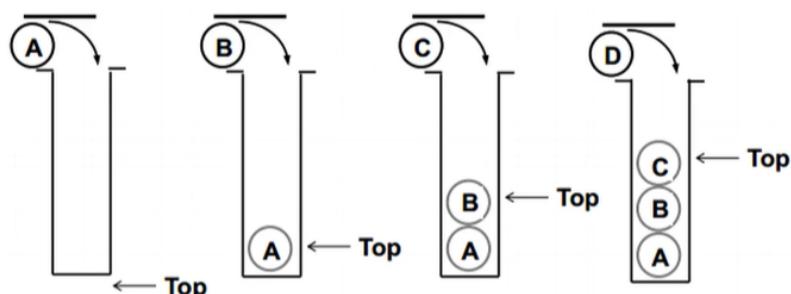
2、int IsFull( Stack S, int MaxSize ): 判断堆栈S是否已满;

3、~~void Push( Stack S, ElementType item )~~: 将元素item压入堆栈;

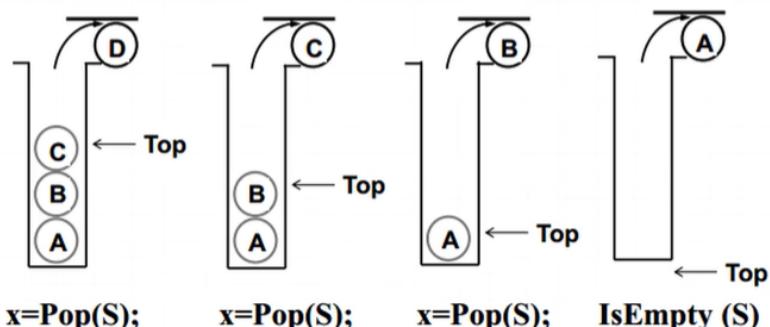
4、int IsEmpty ( Stack S ): 判断堆栈S是否为空;

5、~~ElementType Pop( Stack S )~~: 删除并返回栈顶元素;

## Push & Pop



$\text{CreatStack}(); \text{ Push}(S, A); \text{ Push}(S, B); \text{ Push}(S, C);$



$x = \text{Pop}(S);$

$x = \text{Pop}(S);$

$x = \text{Pop}(S);$

$\text{IsEmpty}(S)$

# 栈的顺序存储实现

## 栈的顺序存储实现



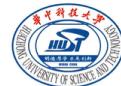
栈的顺序存储结构通常由一个一维数组和一个记录栈顶元素位置的变量组成。

```
#define MaxSize <储存数据元素的最大个数>
typedef struct SNode *Stack;
struct SNode{
    ElementType Data[MaxSize];
    int Top;
};
```

注意：这里的Data[]为一个数组

## 思考题：用一个数组实现两个栈

### 讨论



【例】 请用一个数组实现两个堆栈，要求最大地利用数组空间，使数组只要有空间入栈操作就可以成功。

【分析】 一种比较聪明的方法是使这两个栈分别从数组的两头开始向中间生长；当两个栈的栈顶指针相遇时，表示两个栈都满了。

```
#define MaxSize <存储数据元素的最大个数>
struct DStack {
    ElementType Data[MaxSize];
    int Top1; /* 堆栈 1 的栈顶指针 */
    int Top2; /* 堆栈 2 的栈顶指针 */
} S;
```

S.Top1 = -1;  
S.Top2 = MaxSize;

满栈条件：Top1 == Top2

# 栈的链式存储

- 在表头进行插入/删除

# Create Stack



## CreatStack

13

```
typedef struct SNode *Stack;
struct SNode{
    ElementType Data;
    struct SNode *Next;
};
```

- (1) 堆栈初始化（建立空栈）  
(2) 判断堆栈S是否为空

```
Stack CreateStack()
{ /* 构建一个堆栈的头结点，返回指针 */
    Stack S;
    S = (Stack) malloc(sizeof(struct SNode));
    S->Next = NULL;
    return S;
}

int IsEmpty(Stack S)
{ /* 判断堆栈S是否为空，若为空函数返回整数1，否则返回0 */
    return (S->Next == NULL);
}
```

Huazhong University of Science and Technology

- 每次Push相当于插入，记得申请新空间

```
void Push( ElementType item, Stack S)
{ /* 将元素item压入堆栈S */
    struct SNode *TmpCell;
    TmpCell = (struct SNode *) malloc(sizeof(struct SNode));
    TmpCell->Element = item;
    TmpCell->Next = S->Next;
    S->Next = TmpCell;
}
```

```
ElementType Pop(Stack S)
{ /* 删除并返回堆栈S的栈顶元素 */
    struct SNode *FirstCell;
    ElementType TopElem;
    if( IsEmpty( S ) ) {
        printf("堆栈空");
        return NULL;
    } else {
        FirstCell = S->Next;
        S->Next = FirstCell->Next;
        TopElem = FirstCell->Element;
        free(FirstCell);
        return TopElem;
    }
}
```

栈的应用

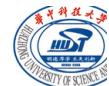
```

void LineEdit() {
    //利用字符栈 S,从终端接收一行并传送至调用过程的数据区。
    InitStack(S);           //构造空栈 S
    ch = getchar();          //从终端接收第一个字符
    while (ch != EOF) { //EOF 为全文结束符
        while (ch != EOF && ch != '\n') {
            switch (ch) {
                case '#': Pop(S, c);      break; //仅当栈非空时退栈
                case '@': ClearStack(S);  break; //重置 S 为空栈
                default : Push(S, ch);    break; //有效字符进栈,未考虑栈满情形
            }
            ch = getchar(); //从终端接收下一个字符
        }
        将从栈底到栈顶的栈内字符传送至调用过程的数据区;
        ClearStack(S); //重置 S 为空栈
        if (ch != EOF) ch = getchar();
    }
    DestroyStack(S);
} // LineEdit

```

## 表达式求值

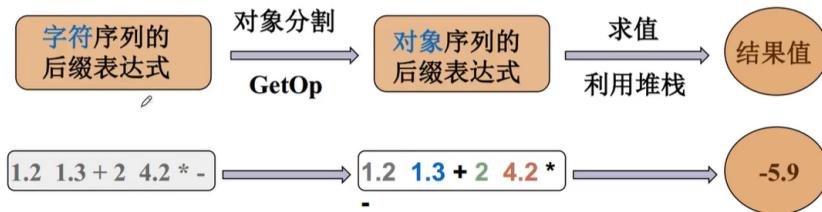
### 栈应用：表达式求值



18

- 栈实现后缀表达式求值的基本过程：

- 1. 运算数：**入栈；
- 2. 运算符：**从堆栈中弹出适当数量的运算数，计算并结果入栈；
- 3. 最后，**堆栈顶上的元素就是表达式的结果值。



- 中缀表达式 --> 后缀表达式：

- 栈

➤ 从头到尾读取中缀表达式的每个对象，对不同对象按不同的情况处理。

- ① **运算数：**直接输出；
- ② **左括号：**压入堆栈；
- ③ **右括号：**将栈顶的运算符弹出并输出，直到遇到左括号（出栈，不输出）；
- ④ **运算符：**
  - 若优先级大于栈顶运算符时，则把它压栈；
  - 若优先级小于等于栈顶运算符时，将栈顶运算符弹出并输出；再比较新的栈顶运算符，直到该运算符大于栈顶运算符优先级为止，然后将该运算符压栈；
- ⑤ 若各对象处理完毕，则把堆栈中存留的运算符一并输出。

例子：

示例：( 2\* (9+6/3-5) +4)



步骤	待处理表达式	堆栈状态 (底↔顶)	输出状态
1	2* (9+6/3-5) +4		
2	* (9+6/3-5) +4		2
3	(9+6/3-5) +4	*	2
4	9+6/3-5) +4	* (	2
5	+6/3-5) +4	* (	2 9
6	6/3-5) +4	* ( +	2 9
7	/3-5) +4	* ( +	2 9 6
8	3-5) +4	* ( + /	2 9 6
9	-5) +4	* ( + /	2 9 6 3
10	5) +4	* ( -	2 9 6 3 / +
11	) +4	* ( -	2 9 6 3 / + 5
12	+4	*	2 9 6 3 / + 5 -
13	4	+	2 9 6 3 / + 5 - *
14		+	2 9 6 3 / + 5 - * 4
15			2 9 6 3 / + 5 - * 4 +

## Hanoi塔时间复杂度分析

基本操作为移动一个“饼”

移动次数的递推公式：

$$M(n) = 2 * M(n - 1) + 1$$

推知：

$$T(n) = 2 * T(n - 1) + 1$$

化简：

$$T(n) = 2^n - 1$$

得到：

$$TimeComplexity = O(2^n)$$

## 递归和栈

# 递归和栈

□ 递归进行是有条件的。一般常把判断语句加在递归语句以前。

- 递归的最底层应该有返回值，以供上层递归的调用。否则会死循环。
- 参量的初始化应该在递归以前。
- 递归调用需要利用堆栈。

每次调用要把本次调用的参数和局部变量保存在栈顶。  
 每次从下一层调用返回到上一层调用时，从栈顶恢复本层调用的参数和局部变量的值。

Huazhong University of Science and Technology

Page 57, Fig. 3.7

## 迷宫问题

### 迷宫问题 - 算法设计

