数据结构与算法

计算机学院

朱晨阳 副教授

zhuchenyang07@nudt.edu.cn

约瑟夫问题

- 设n个人围成一个圆圈，按一指定的方向，从第s个人开始报数，报数到m为止，报数为m的人出列，然后从下一个人开始重新报数，报数为m的人又出列，如此反复，直到所有的人都出列为止。求n个人出列顺序。
- 约瑟夫问题的C语言实现

约瑟夫问题

```
void Josephus(Vector* P, int n, int s, int m)
{   int k=1, i, s1=s, j, w;
    for(i=0; i<n; i++) { Insert(P, k, i);  k++; }
    for(j=n; j>=1; j--)
    {   s1=(s1+m-1)%j;
        if(s1==0) s1=j;
        w=GetNode(P, s1-1);
        Remove(P, s1-1);
        Insert(P, w, n-1);
    }
}
```

递归

- 能否递归求解约瑟夫问题?
- **f(n,m) = (f(n−1,m) + m - 1) % n + 1**

栈

- 栈是插入和删除操作只允许在表的同一端进行的线性表；对于顺序存储的栈，一般在表尾一端进行。因此栈是一种操作受限的线性表。
- 栈可以顺序存储和链式存储，分别称为顺序栈（第2章）和链式栈(第3章)。

进栈与出栈

- 往栈里插入一个元素称为进栈(push)，从栈里删除一个元素称为出栈(pop)。每次删除的都是最后进栈的元素，所以栈又称为后进先出(LIFO)表。允许插入和删除操作的一端称为栈顶，另一端称为栈底。不含栈元素的栈称为空栈。

栈的ADT

ADT Stack

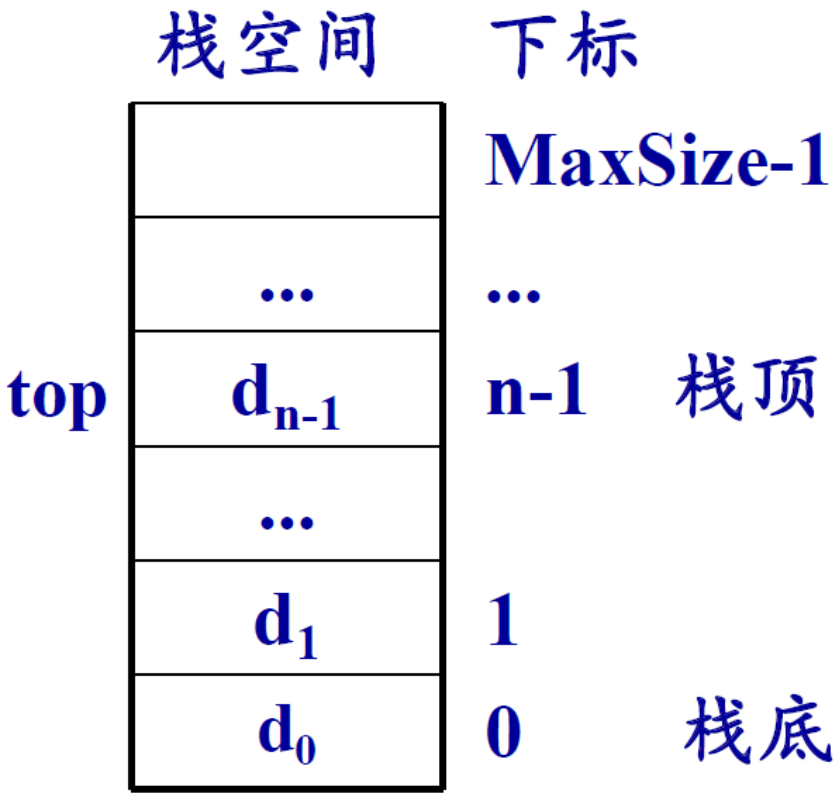
Data: D={d0,d1,d2,...,dn-1}; R={<d0,d1>,...}

Operations:

InitStack	{ }
FreeStack	释放栈所占内存
IsEmpty	true | false
IsFull	true | false
push(x)	{d0,d1,d2,...,dn-1,x}
pop	{d0,d1,d2,...,dn-2}
GetTop	dn-1
MakeEmpty	{d0,d1,d2,...,dn-1} -> {}

栈的实现

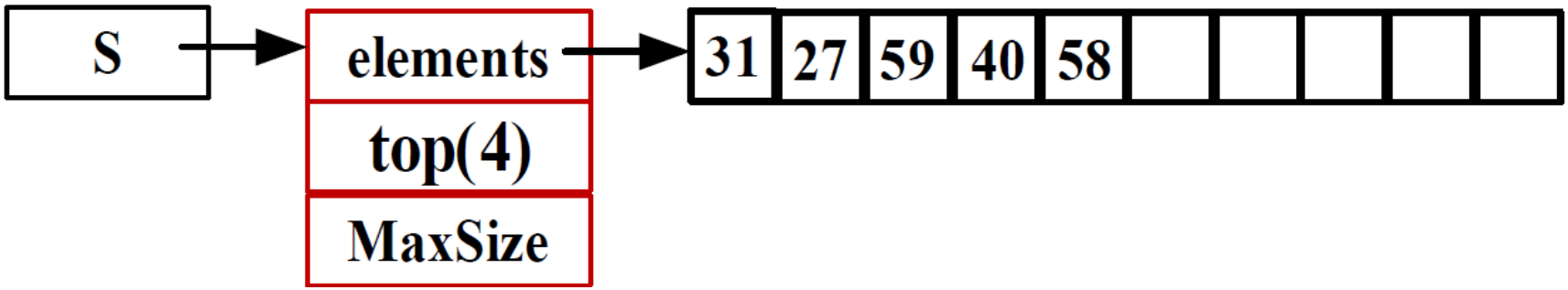
- 顺序栈用数组来存放，设该数组的最大大小为MaxSize。
- 设置一个栈顶指针top(对于顺序栈，top为栈顶元素的数组下标)。当top为-1时表示栈空；当top=MaxSize-1时，表示栈满，此时进行插入操作会“溢出”。



栈的实现

```
typedef int ElementType;
enum boolean{FALSE,TRUE};
typedef enum boolean Bool;
struct stack
{
   ElementType *elements; /*存放栈元素的数组*/
   int top;               /*栈顶指针*/
   int MaxSize;           /*栈空间的最大尺寸*/
};
```

栈的实现



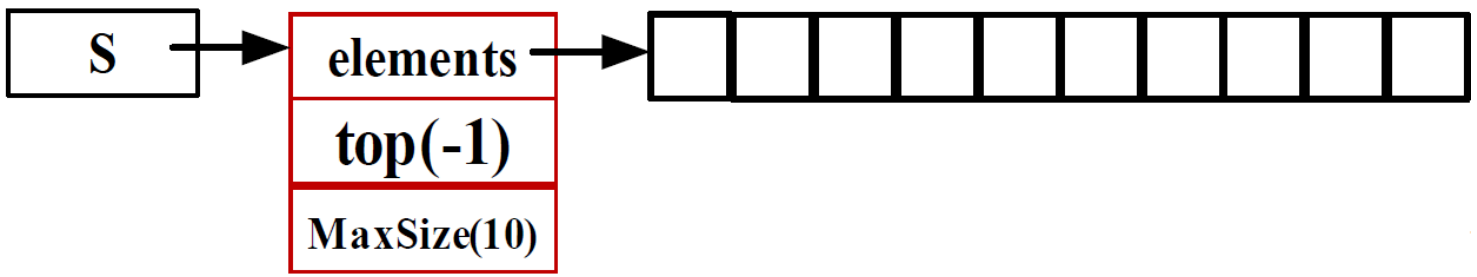
栈的实现

```
typedef struct stack Stack;
void InitStack(Stack *, int sz);/*创建栈空间，生成一个空栈*/
void FreeStack(Stack *); /*释放栈空间*/
Bool IsEmpty(Stack *S);
Bool IsFull(Stack *S);
int Push(Stack *, ElementType );
ElementType Pop(Stack *);
ElementType GetTop(Stack *);
```

栈的实现

```
void InitStack(Stack *S, int sz)
{
   S->MaxSize = sz;
   S->elements = (ElementType *)malloc(sizeof(ElementType)
 * S->MaxSize);
   S->top = -1;
}
```

InitStack(S,10)



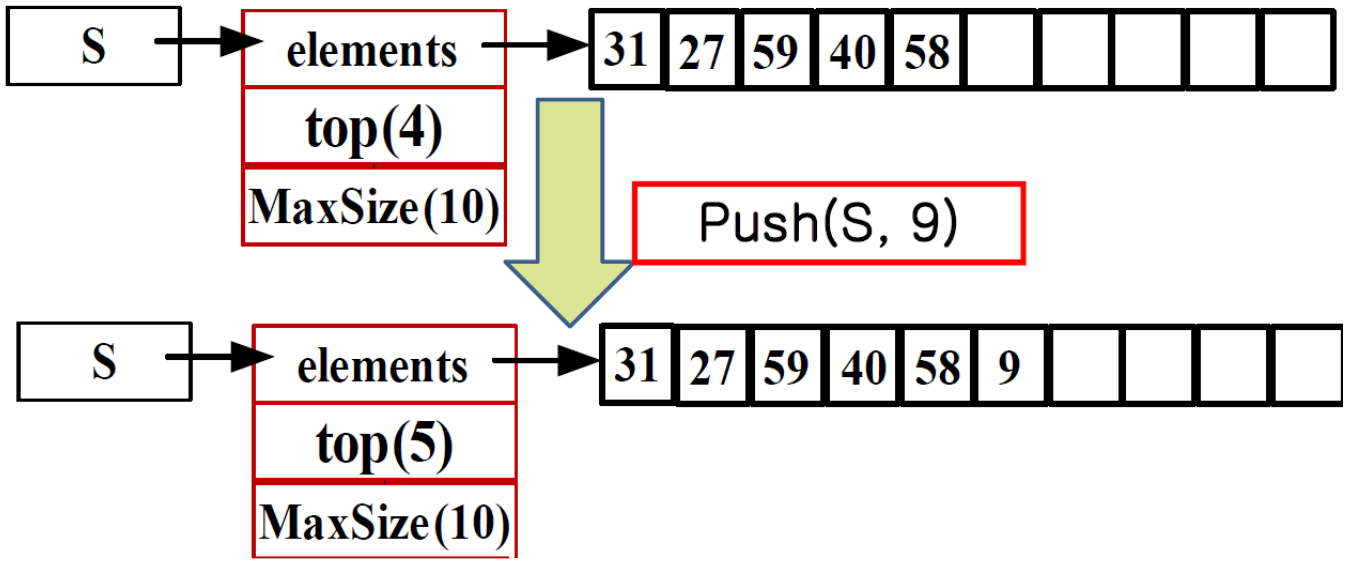
栈的实现

```
void FreeStack(Stack *S) /*释放栈空间*/
{
   free(S->elements);
}
void MakeEmpty(Stack *S) /*置栈为空栈*/
{
          S->top=-1;
}

Bool IsEmpty(Stack *S)   /*判栈是否为空*/
{
   return (Bool)(S->top = = -1);
}

Bool IsFull(Stack *S) /*判栈是否已满*/
{
   return (Bool)(S->top = = S->MaxSize-1);
}
```

栈的实现

```
int Push(Stack *S, ElementType x)
{
    if (!IsFull(S))
    {
        S->elements[++(S->top)] == x;
        return 0;
    }
    else
        return -1;
}
```

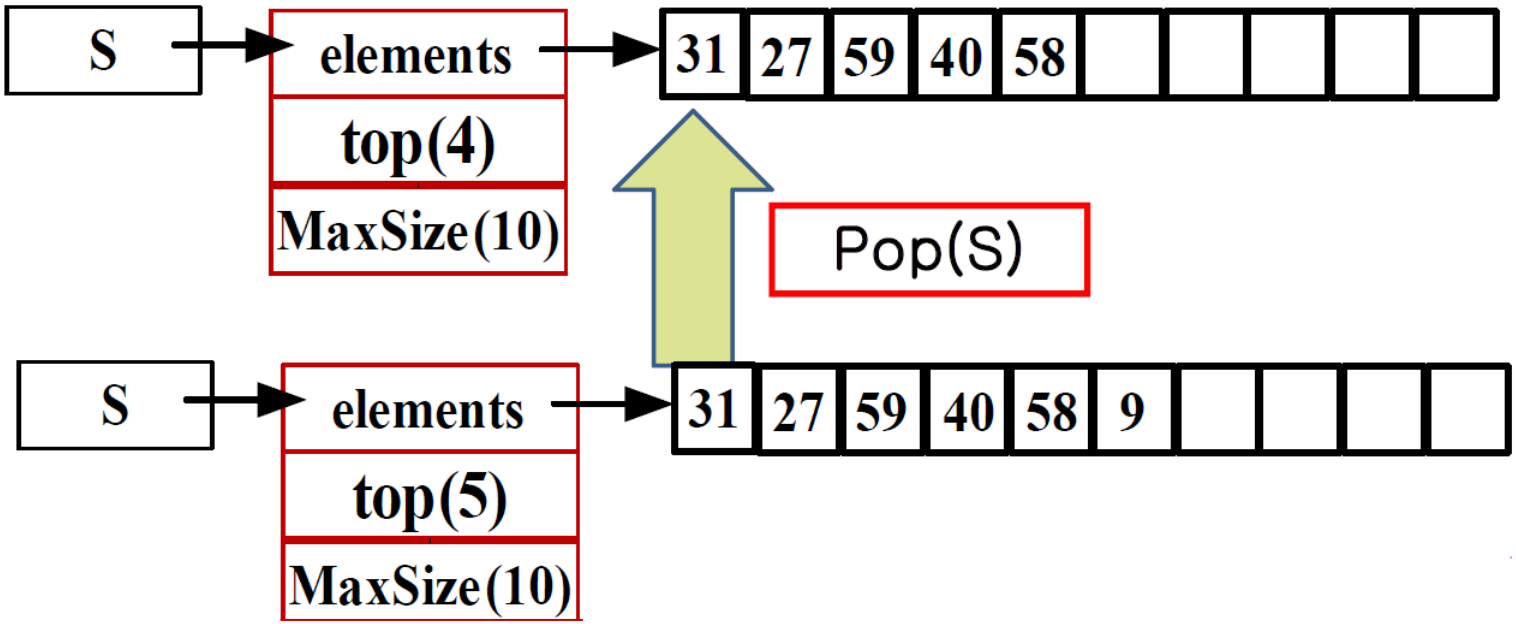
栈的实现


S
elements
top(4)
MaxSize(10)
31 27 59 40 58
Push(S, 9)
S
elements
top(5)
MaxSize(10)
31 27 59 40 58 9

栈的实现

```
ElementType Pop(Stack *S)
{
    if (!IsEmpty(S))
        return S->elements[(S->top)--];
    else
    {
        printf("stack is empty !\n");
        exit(1);
    }
}
```

栈的实现


S
elements
top(4)
MaxSize(10)
31
27
59
40
58
Pop(S)
S
elements
top(5)
MaxSize(10)
31
27
59
40
58
9

栈的实现

```
ElementType GetTop(Stack *S){
    if (!IsEmpty(S))
        return S->elements[S->top];
    else
    {
        printf("stack is empty !\n");
        exit(1);
    }
}
```

栈的应用

- 递归
  - 若一个函数、过程或者数据结构定义的内部又直接或间接地包含有定义本身的应用，则称它们是递归的。
  - 采用递归的条件：问题能分解成一个或多个规模较小但具有类似于原问题特性的子问题，子问题又可继续分解，直到子问题无须分解便可直接求解；存在一个或多个无须分解便可直接求解的最小子问题——递归终止条件。

递归

- 求和：S(n)=1+2+3+...+(n-1)+n
- 阶乘：n! = n(n-1)...1
- 递归定义：

$$S(n)=\begin{cases}1 & n=1\\ n+S(n-1) & n>1\end{cases}$$

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

递归函数

```
int Sum(int n)
{
   if (n == 1)
      return 1;
   else
      return n + Sum(n - 1);
}
```

```
int Factorial(int n)
{
   if (n == 0)
      return 1;
   else
      return n * Factorial(n - 1);
}
```

中缀表达式

- 中缀表达式：如(2 + 3) * 4
- 后缀表达式：如2 3 + 4 *
- 对于源程序中的中缀表达式，
  - 若为常量表达式，编译程序在编译过程中可立即计算出该表达式的值。
  - 若其含有变量，编译程序把表达式的计算步骤翻译成机器指令序列。

中缀表达式

- 考虑常量表达式的计算，假设：
  - 操作数只允许为个位的整常数。
  - 运算符包括二元运算符“+”、“-”、“*”、“/”、和括号“(”、“)”
  - 表达式没有语法错误。
- 计算机如何计算中缀表达式?

中缀表达式

- 表达式 8 - (3 + 5–4 / 2) * 2 / 3
  - 运算符栈 { }
  - 操作数栈 { }

扫描表达式，析取操作数或运算符，前者进操作数栈，后者进运算符栈。
关键：保持运算符栈的栈顶运算符在运算符栈中具有最高优先级

中缀表达式

当前运算符栈顶 T

当前获取的运算符 W	空	(	*或 /	+ 或-
(	?	?	?	?
)	?	?	?	?
*或 /	?	?	?	?
+或 -	?	?	?	?

中缀表达式

- 表达式 8; - (3 + 5–4 / 2) * 2 / 3
  - 运算符栈 { }
  - 操作数栈 {8 }

中缀表达式

- 表达式 8 -; (3 + 5–4 / 2) * 2 / 3
  - 运算符栈 {- }
  - 操作数栈 {8 }

中缀表达式

- 表达式 8 - (; 3 + 5–4 / 2) * 2 / 3
  - 运算符栈 {- (}
  - 操作数栈 {8 }

中缀表达式

- 表达式 8 - (3 +; 5–4 / 2) * 2 / 3
  - 运算符栈 {- (+}
  - 操作数栈 {8 3 }

中缀表达式

- 表达式 8 - (3 + 5;–4 / 2) * 2 / 3
  - 运算符栈 {- (+}
  - 操作数栈 {8 3 5}

保持运算符栈的栈顶运算符在运算符栈中具有最高优先级;通过出栈来保证这一点。

中缀表达式

- 表达式 8 - (3 + 5–;4 / 2) * 2 / 3
  - 运算符栈 {- (-}
  - 操作数栈 {8 8}

中缀表达式

- 表达式 8 - (3 + 5–4; / 2) * 2 / 3
  - 运算符栈 {- (-}
  - 操作数栈 {8 8 4}

中缀表达式

- 表达式 8 - (3 + 5–4 /; 2) * 2 / 3
  - 运算符栈 {- (- /}
  - 操作数栈 {8 8 4}

中缀表达式

- 表达式 8 - (3 + 5–4 / 2;) * 2 / 3
  - 运算符栈 {- (- /}
  - 操作数栈 {8 8 4 2}

中缀表达式

- 表达式 8 - (3 + 5–4 / 2;) * 2 / 3
  - 运算符栈 {- (- }
  - 操作数栈 {8 8 2}
- 表达式 8 - (3 + 5–4 / 2;) * 2 / 3
  - 运算符栈 {- (}
  - 操作数栈 {8 6}
- 表达式 8 - (3 + 5–4 / 2); * 2 / 3
  - 运算符栈 {- }
  - 操作数栈 {8 6}

中缀表达式

- 表达式 8 - (3 + 5–4 / 2) *; 2 / 3
  - 运算符栈 {- *}
  - 操作数栈 {8 6}

中缀表达式

- 表达式 8 - (3 + 5–4 / 2) * 2; / 3
  - 运算符栈 {- *}
  - 操作数栈 {8 6 2}

中缀表达式

- 表达式 8 - (3 + 5–4 / 2) * 2 /; 3
  - 运算符栈 {- /}
  - 操作数栈 {8 12}

中缀表达式

- 表达式 8 - (3 + 5–4 / 2) * 2 / 3;
  - 运算符栈 {- /}
  - 操作数栈 {8 12 3}
- 表达式 8 - (3 + 5–4 / 2) * 2 / 3
  - 运算符栈 {- }
  - 操作数栈 {8 4}
- 表达式 8 - (3 + 5–4 / 2) * 2 / 3
  - 运算符栈 { }
  - 操作数栈 {4}

中缀表达式

当前运算符栈顶 T

当前获取的运算符 W	空	(	*或 /	+ 或-
(	W进栈。	W进栈。	W进栈。	W进栈。
)	Error。	T出栈。	T出栈并计算...	T出栈并计算...
*或 /	W进栈。	W进栈。	T出栈并计算...	W进栈。
+或 -	W进栈。	W进栈。	T出栈并计算...	T出栈并计算...

中缀表达式

- 运算符在进栈时：进栈运算符优先级应高于栈顶运算符，否则通过出栈来满足，以确保运算符栈（+-*/）从栈顶到栈底是按优先级从高到底排列的。(为什么？)

出栈时进行计算

- 每出栈一个运算符，就出栈两个操作数进行运算，结果进入操作数栈。
- compute子过程：
  - 1 从运算符栈出栈一个运算符
  - 2 从操作数栈出栈两个操作数
  - 3 用出栈的运算符对出栈的操作数进行运算
  - 4 将运算结果进操作数栈

中缀表达式计算算法

1. 操作数栈、运算符栈清空
2. 扫描表达式，每获取一个单词w，分五种情况处理：
   - w为操作数：则w进操作数栈
   - 运算符栈空或w为“(”：w进运算符栈
   - w为“)”：
     - (1) 直到运算符栈顶为“(”，反复调用compute子过程；
     - (2) 然后将“(”出栈

中缀表达式计算算法

- w为“*”或“/”：
  - (1) 若运算符栈顶为“*”或“/”，则调用compute子过程
  - (2) w进运算符栈
- w为“+”“-”：
  - (1) 直到运算符栈顶为“(”或空，反复调用compute子过程
  - (2) w进运算符栈

3. 直到运算符栈空，反复调用compute子过程

计算实例

- 表达式：2*(3+4)-8/2
- 运算符栈：{ }
- 操作数栈：{ }