



第三章 栈和队列

逻辑结构和线性表相同

运算受到了限制

根据所受限制的不同，分为栈和队列

栈的应用——括号匹配的检验

□ 正确的表达式:

➤ 括号要成对出现

➤ 表达式中括号不允许出现骑跨现象

□ 例: $a + \{2 - [b + c] * (8 * [8 + g] / [m - e] - 7) - p\}$


□ 忽略表达式中的运算对象和运算符, 只看括号:

➤ $\{ [] ([] []) \}$ ---- 成对出现 😊

➤ $\{ [()] \}$ ---- 不允许出现骑跨现象 ☹️

设计算法判断表达式中括号是否合法

从左至右读取表达式，读到运算对象和运算符，不做任何动作，直接往下继续读，读到**括号**，**要对括号**的使用是否正确进行检查


$$2+\{a*[b+d]-(w/[17-8]*[7+7])\}$$



问题

□ $\{[]([] [])\}$, $\{[]([(]))\}$

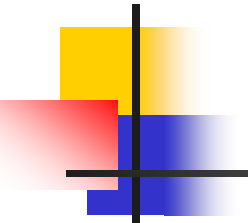
- 左右括号不配对
- 读取的右括号与栈顶保存的左括号形状不匹配

□ $\{[]([] [])\}]$ 右括号多

- 读取了右括号，但栈空，没有左括号与之配对

□ $\{[]([] [])\}$ 左括号多

- 表达式读取结束，栈中还有保留的左括号没配对
- 为了判断表达式是否读取结束，在表达式尾部加#
为表达式结束标志。
- 例： $\{[]([] [])\} \#$



1. 初始化一个空栈;

3. 读一个字符存入变量ch;

4. 若ch=='#', 转5; 否则:

4.1. 若ch为左括号, 进栈, 读下一个字符到ch; 转4;

4.2. 若ch为 ')', 若栈空, 则“括号不匹配”, 结束。

若栈顶不是 '(', 则“括号不匹配”, 结束。

否则, 退栈, 读下一个字符到ch; 转4;

4.3. 若ch为 ']', 若栈空, 则“括号不匹配”, 结束。

若栈顶不是 '[', 则“括号不匹配”, 结束。

否则, 退栈, 读下一个字符到ch; 转4;

4.4. 若ch为 '}', 若栈空, 则“括号不匹配”, 结束。

若栈顶不是 '{', 则“括号不匹配”, 结束。

否则, 退栈, 读下一个字符到ch; 转4;

4.5. 读下一个字符到ch; 转4;

5. 若栈空, 则“括号匹配”, 结束。否则“括号不匹配”, 结束。



```
void s1(){
s.top=-1;
scanf("%c",&ch);
while(ch!='#'){
if((ch=='(')||(ch=='[')||(ch=='{'))
if(s.top==MAXSIZE-1){printf("OVERFLOW");return;}
else{s.data[++s.top]=ch; scanf("%c",&ch);}
else
if(ch==')')
if((s.top==-1) ||(s.data[s.top]!='(')){printf("NoMatch");return;}
else{s.top--; scanf("%c",&ch);}
else if(ch==']')
if((s.top==-1) ||(s.data[s.top]!='[')){printf("NoMatch");return;}
else{s.top--; scanf("%c",&ch);}
else if(ch=='}')
if((s.top==-1) ||(s.data[s.top]!='{')){printf("NoMatch");return;}
else{s.top--; scanf("%c",&ch);}
else scanf("%c",&ch);}
if(s.top==-1) printf("Match"); else printf("NoMatch");return;}
```



栈的应用

- 表达式中括号是否合法
- 将由 $+$, $-$, $*$, $/$ 和单字母变量组成的普通表达式转换成逆波兰式。
- 表达式求值



栈的应用----表达式的逆波兰式

- 设计算法将由 $+$, $-$, $*$, $/$ 和单字母变量组成的普通表达式转换成逆波兰式。
- 表达式的表示形式
 - 1 前缀形式----波兰式
 - 2 中缀形式
 - 3 后缀形式----逆波兰式



前缀形式

■ $a+b$ \longrightarrow $+ab$

■ $a+b*c$ \longrightarrow $+a*bc$

■ $a+b*c-e$ \longrightarrow $-+a*bce$



后缀形式

■ $a+b$ \longrightarrow $ab+$

■ $a+b*c$ \longrightarrow $abc*+$

■ $a+b*c-e$ \longrightarrow $abc*+e-$



求逆波兰式

- $a*b+c-d/e$ #----表达式后面加“#”代表表达式结束
- 设计算法求逆波兰式--构造一个栈保存相应的运算符



求逆波兰式

- $a*b+c-d/e$ #----表达式后面加“#”代表表达式结束
- 设计算法求逆波兰式--构造一个栈保存相应的运算符

s.data[4]

s.data[3]

s.data[2]

s.data[1]

s.data[0]

s.top=-1

求逆波兰式—— a

- $\underline{a} * b + c - d / e \#$ ---- 表达式后面加“#”代表表达式结束
- 设计算法求逆波兰式 -- 构造一个栈保存相应的运算符

s.data[4]

s.data[3]

s.data[2]

s.data[1]

s.data[0]

s.top=-1

求逆波兰式—— a

- $a_b+c-d/e \#$ ——表达式后面加“#”代表表达式结束
- 设计算法求逆波兰式——构造一个栈保存相应的运算符

s.data[4]

s.data[3]

s.data[2]

s.data[1]

s.data[0]

*

← s.top=0

求逆波兰式— ab

- $a*\underline{b}+c-d/e$ #----表达式后面加“#”代表表达式结束
- 设计算法求逆波兰式--构造一个栈保存相应的运算符

s.data[4]

s.data[3]

s.data[2]

s.data[1]

s.data[0]

*

← s.top=0

求逆波兰式—— ab^*

- $a*b\text{+}c\text{-}d/e\text{\#}$ ——表达式后面加“#”代表表达式结束
- 设计算法求逆波兰式——构造一个栈保存相应的运算符

s.data[4]

s.data[3]

s.data[2]

s.data[1]

s.data[0]

*

← s.top=-1

求逆波兰式—— ab^*

- $a*b\text{+}c-d/e\text{\#}$ ——表达式后面加“#”代表表达式结束
- 设计算法求逆波兰式——构造一个栈保存相应的运算符

s.data[4]

s.data[3]

s.data[2]

s.data[1]

s.data[0]

+

← s.top=-1

求逆波兰式—— $ab*c$

- $a*b+\underline{c}-d/e$ #----表达式后面加“#”代表表达式结束
- 设计算法求逆波兰式--构造一个栈保存相应的运算符

s.data[4]

s.data[3]

s.data[2]

s.data[1]

s.data[0]

+

← s.top=0

求逆波兰式—— $ab*c$

- $a*b+c_d/e$ # --- 表达式后面加“#”代表表达式结束
- 设计算法求逆波兰式 -- 构造一个栈保存相应的运算符

s.data[4]

s.data[3]

s.data[2]

s.data[1]

s.data[0]

+

← s.top=0

求逆波兰式—— $ab*c+$

- $a*b+c_d/e$ #----表达式后面加“#”代表表达式结束
- 设计算法求逆波兰式--构造一个栈保存相应的运算符

s.data[4]

s.data[3]

s.data[2]

s.data[1]

s.data[0]

+

← s.top=-1

求逆波兰式—— $ab*c+$

- $a*b+c_d/e \#$ ——表达式后面加“#”代表表达式结束
- 设计算法求逆波兰式——构造一个栈保存相应的运算符

s.data[4]

s.data[3]

s.data[2]

s.data[1]

s.data[0]

-

← s.top=0

求逆波兰式—— $ab*c+d$

- $a*b+c-\underline{d}/e$ #----表达式后面加“#”代表表达式结束
- 设计算法求逆波兰式--构造一个栈保存相应的运算符

s.data[4]

s.data[3]

s.data[2]

s.data[1]

s.data[0]

-

← s.top=0

求逆波兰式—— $ab*c+d$

- $a*b+c-d/e$ # ---- 表达式后面加“#”代表表达式结束
- 设计算法求逆波兰式 -- 构造一个栈保存相应的运算符

s.data[4]

s.data[3]

s.data[2]

s.data[1]

s.data[0]

/
-

← s.top=1

求逆波兰式—— $ab*c+de$

- $a*b+c-d/e$ # ---- 表达式后面加“#”代表表达式结束
- 设计算法求逆波兰式 -- 构造一个栈保存相应的运算符

s.data[4]

s.data[3]

s.data[2]

s.data[1]

s.data[0]

/
-

← s.top=1

求逆波兰式—— $ab*c+de$

- $a*b+c-d/e$ #——表达式后面加“#”代表表达式结束
- 设计算法求逆波兰式——构造一个栈保存相应的运算符

s.data[4]

s.data[3]

s.data[2]

s.data[1]

s.data[0]

/
-

← s.top=1

求逆波兰式—— $ab*c+de/$

- $a*b+c-d/e$ #----表达式后面加“#”代表表达式结束
- 设计算法求逆波兰式--构造一个栈保存相应的运算符

s.data[4]

s.data[3]

s.data[2]

s.data[1]

s.data[0]

/
-

← s.top=0

求逆波兰式—— $ab*c+de/-$

- $a*b+c-d/e$ #----表达式后面加“#”代表表达式结束
- 设计算法求逆波兰式--构造一个栈保存相应的运算符

s.data[4]

s.data[3]

s.data[2]

s.data[1]

s.data[0]

/
-

← s.top=-1

1 s.top=-1;

2 读一个字符到ch;

3 若ch为结束标志‘#’,转7; 否则转4;

4 若ch为单字母, 输出, 读下一个字符ch, 转3; 否则

5 若ch为‘+’或‘-’,

5.1 若栈非空且ch的优先级比栈顶符号的优先级低,
则弹出栈顶, 转5.1; 否则转5.2,

5.2 ch入栈, 读下一个字符ch, 转3;

6 若ch为‘*’或‘/’,

6.1 若栈非空且ch的优先级比栈顶符号的优先级低, 则
弹出栈顶, 转6.1; 否则转6.2;

6.2 ch入栈, 读下一个字符ch, 转3;

7 依次弹出栈中所有符号

因为只有+, -, *, /4个运算, 且出现在当前读入的“+”或“-”前的所有运算均高于当前的+或-, 所以依次弹出栈中所有内容。

因为只有+, -, *, /4个运算, 栈顶符号为‘*’或‘/’时, 优先级高于当前读入的运算, 则弹出栈顶的运算