**西安电子科技大学**

**信息存储与检索 课程实验报告**

**实验名称 上机实验报告4**

经济与管理 学院 2006011 班

成 绩

姓名 马浩毓 学号 20069100105

同作者

实验日期 2023 年 5 月 22日

|  |
| --- |
| 指导教师评语：  实验内容：  实验效果：  实验报告：  考勤情况：  其他情况：  指导教师：  年 月 日 |

**一、实验目的：**

* 加深理解逆波兰、准波兰转换的基本原理。
* 熟悉检索式的逆波兰、准波兰转换过程。
* 熟练运用自己熟悉的软件工具或编程语言，实现布尔提问式的逆波兰、波兰转换方法并对运行结果进行分析。

**二、机器环境（硬件、软件配置，编程语言）**

Windows 10操作系统

Python 3.8.1 编译环境为Pycharm

**三、算法或原理：**

在布尔检索系统中，为了实现用户的检索需求，首先需要对用户输入的布尔逻辑检索式进行必要的加工和变换处理。布尔检索是以倒排文档为基础。常用的变换处理方法有逆波兰变换法、准波兰变换法、析取范式变换法等。

**3.1. 逆波兰变换**

  逆波兰变换法又称福岛方法，该方法使用最早、影响也最大。

  处理要点：先将布尔检索式（即提问式）转换成逆波兰表示形式，然后将逆波兰表示形式翻译成一组检索指令。（即两步处理）

  （后续的检索处理过程，就是执行一组检索指令）

（1）逆波兰表达式

  例如：A加B之和，再乘以C。

  中缀表示法：(A+B)\* C（其中，运算符在两个运算项的中间）

  运算规则类似于算术运算，一般是从左到右进行，遇到括号时优先括号内的运算。但是其括号无法去掉，书写并非最简洁 。

  注：不同的算符规定了不同的运算优先次序（即算符优先级）

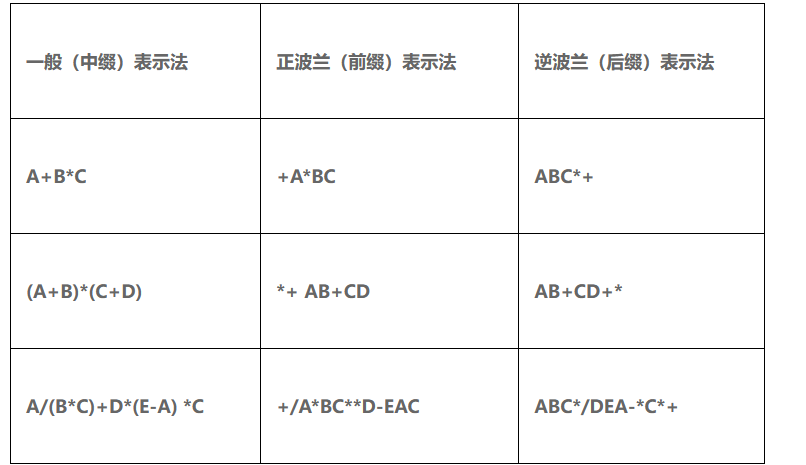
 后来，波兰的一位学者卢卡西维兹提出两种不用括号的表示法，称为波兰表示法。

  例如：对 (A+B) \* C ，有两种表示。

前缀表示法：算符在前，为 \*+ABC。又称正波兰表示法。

后缀表示法：算符在后，为 AB+C\*。又称逆波兰表示法。

三种表达式例示比较：



分析前述三种表示法，有以下共同点：

1. 运算符个数保持不变；
2. 运算项个数不变，且出现的先后次序不变。
3. 又有以下不同点：
4. 正波兰和逆波兰表示法都是无括号的表示法，结构上比中缀表示法更简洁；
5. 逆波兰表示法又比正波兰表示法运算处理更简单方便，符合常规的执行顺序。

这里，给出对表达式的逆波兰（后缀）表示法的运算规则：

从左向右扫描后缀表示形式，当遇到运算项时，就把它保存起来；当遇到运算符时，就取出其前面紧接着的两个运算项依次进行运算处理，并把其结果当作一个新的运算项保存起来；再继续向右检查及扫描其它符号，直至结束。

**（2）布尔检索式的逆波兰变换**

   为了实现提问式的逆波兰变换，首先需要在计算机内存设置以下3个工作区域 ：

  逆波兰输出区；算符栈；检索词表区。

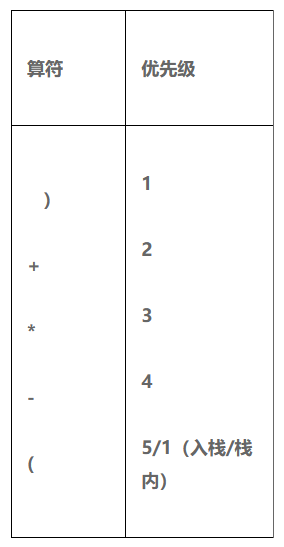
逆波兰输出区——存放经变换处理后提问式的逆波兰形式，即算项和算符形成的序列；

算符栈——一个“后进先出”的表区，主要用来暂存运算符，并重新排列运算符，以便确定运算顺序。（临时使用的堆栈）

检索词表区——将提问式中的检索词列表，并定义对应地址编号。在逆波兰输出区中，使用地址编号来代替作为运算项的检索词。

逆波兰变换过程中对运算符的处理，是通过各自的优先级来控制的。算符优先级高低分别由数值大小表示。

**算符优先级列表**



提问式示例:(A+B)\*(C+D)．（归结为两类：检索词、算符）

提问式的逆波兰变换处理规则：

  从左向右逐个扫描提问式的字符，根据不同情形予以相应的处理。

①如果是检索词（运算项），则将其置入检索词表区中，并将相应的词表地址送入逆波兰输出区中。

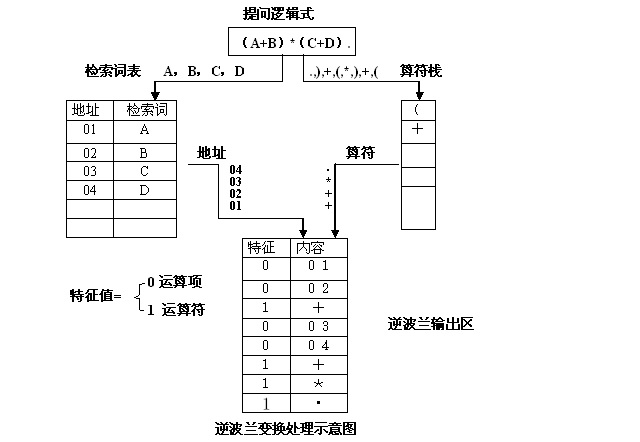
②如果是运算符+，\*，-，则将它与算符栈的栈顶算符按照优先级进行比较：若它的优先级比栈顶算符的优先级高，则把它压入算符栈内；若相等或低于栈顶算符的优先级，则退出（或弹出）栈顶算符，转送入逆波兰输出区，然后再与新的栈顶算符比较优先级，以此类推。

③如果是左括号“（”，由于其入栈优先级为5，即优先级最高，应无条件入栈。进栈后其优先级变为最低（栈内优先级），即变为1。

④如果是右括号“）”，则将与之配对的“（”之间的所有算符从算符栈中按“后进先出”次序依次弹出，移入逆波兰输出区，并清除这对括号。

⑤ 若为检索式结束标志“．”，则将留在算符栈中的算符按“后进先出”次序全部移入逆波兰输出区中，最后将“．”亦置入其中。

**注意：三个工作区域的初始状态均为空；逆波兰区的特征值：0表示运算项，1表示算符。**

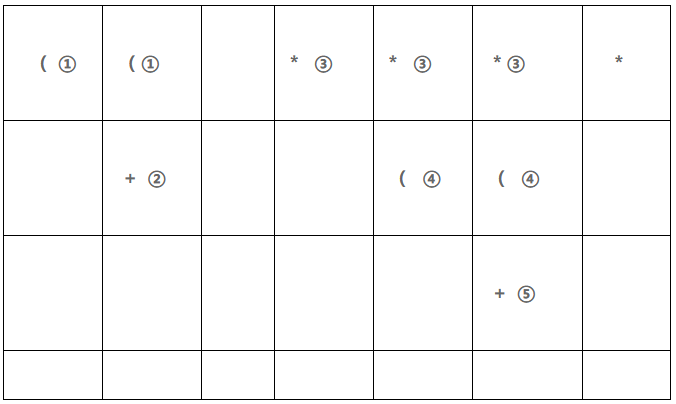


逆波兰变换处理示意图大致反映出变换过程中各个表区的状态。

下表示出提问式(A+B)\*(C+D).

变换过程中算符栈变化情况:

(其中， ①、 ②、③等表示进栈顺序，每列最上方为栈底)



**（3）检索指令表的生成**

当逆波兰变换结果（即逆波兰形式）形成之后，下一步需要生成一组检索指令。

这一过程需设置3个表区：

  检索指令表、工作区、工作区管理表。

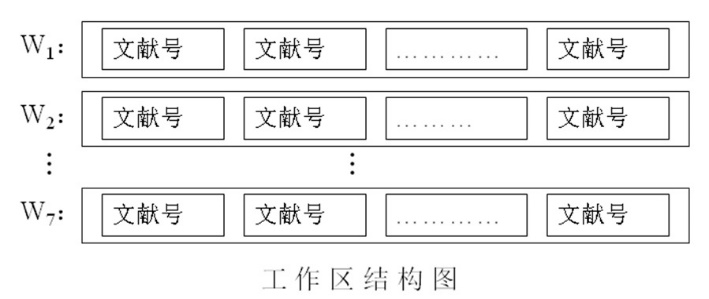
  检索指令表：放置由逆波兰输出区转换而来的一条条检索指令。

  工作区：存放包含某个检索词的文献号，和运算的结果（含中间结果、最终结果）。通常放在内存，一般设7个工作区。

  一个工作区的单元数即所存放的文献号个数（有时也称为工作区长度），其大小视检索系统的规模而定。

  工作区管理表：管理、控制工作区的使用。（结构为7行、2列）

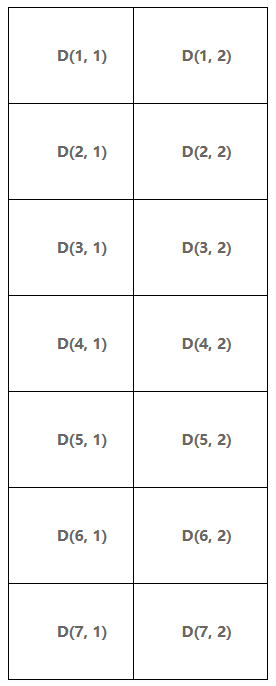
工作区的区号为1~7。设置原则：工作区单元数不小于倒排档中任一检索词之后可能出现的文献号个数（考虑最大个数，定长设置）。



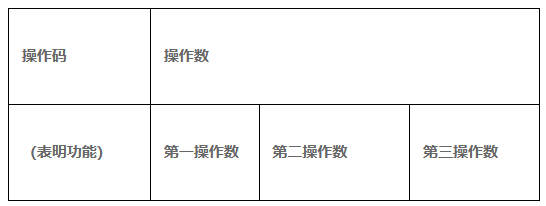
工作区管理表的解释：

  D(j，1)，即第1列。 D(j，1)=1，表示第j个工作区被占用；D(j，1)=0，表示第j个工作区未被占用（即空闲）。  其中， j取值1~7。

  D(j，2)，即第2列，存放第j个工作区的运算次序值（初值为0、加1递增，相当于一个统一的计数器）。



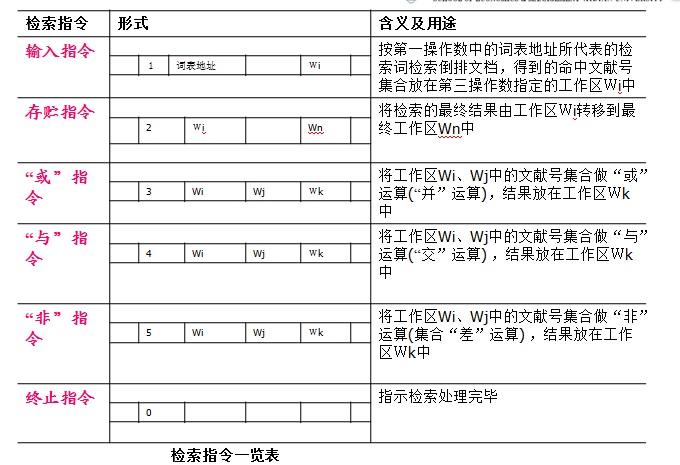
检索指令表用来存放若干条检索指令。按检索指令的功能，共有6种检索指令(操作)。每条检索指令的构成形式如下所示：



**6种检索指令:**

**输入指令、存贮指令、“或”指令、“与”指令、“非”指令、终止指令。**

**1、2、3 、4、5、0。**



下面讨论如何生成检索指令（表）。

**注意：初态时各表区清空/清0；工作区号是由小到大使用；严格区分检索指令表中的地址编号（01、02等）和工作区号码（1-7）！**

由逆波兰输出区内容生成检索指令表，遵循以下处理顺序和规则:

从逆波兰输出区的第一行开始逐行扫描，同时按以下顺序给出检索指令。

① 遇到检索词表地址(运算项)：由工作区管理表中找出可用（即空闲）的工作区(号)，将词表地址作为第一操作数，某个工作区(号)作为第三操作数，做出“输入指令”，置入检索指令表中。（置运算次序值：+1递增）

② 遇到算符：由工作区管理表中取出两个运算次序最高的(即最近被占用的)工作区号码，依其占用的先后分别作为第一、二操作数，再在工作区管理表中找出一个“未被占用”的工作区的号码作为第三操作数，按算符性质做出“或”、“与”、“非”检索指令之一，将指令下推进检索指令表。

需要注意的是：当实际执行完该运算指令后，由第三操作数所确定的工作区仍“被占用”，而由第一、二操作数所确定的工作区则要获得释放，即改为“未被占用”，这一点要及时在工作区管理表中反映出来。

③ 当扫描到结束标志“．”时，说明逆波兰输出区已扫描完毕，且工作区管理表中必定显示只有一个工作区“被占用”。这时，将此工作区的号码作为第一操作数，最终工作区的号码作为第三操作数，做出“存贮指令”，送入检索指令表中。

④ 当形成“存贮指令”之后，做出“终止指令”，下推进检索指令表。

后面我们以提问式（A+B）\*（C+D）.为例，说明将逆波兰输出区内容如何变换为检索指令表。

当生成检索指令表之后，将按检索指令表中的顺序逐条执行检索指令，即可完成布尔检索过程。

**3.2．检索处理过程**

使用表区：检索指令表、检索词表、工作区。

按照检索指令表中的指令顺序，逐条执行检索指令：

①遇输入指令，由第一操作数所示地址从检索词表内取出检索词，进而查询倒排档，取出包含该检索词的文献号集合，存放到第三操作数所指示的工作区。

②遇运算指令，将第一、二操作数对应的文献号集合（放在两个工作区）进行规定的集合运算，得到新的文献号集合并存放到第三操作数所指示的工作区。

③遇存贮指令，将文献号集合转贮到最终工作区。

**倒排文档检索举例**

|  |
| --- |
| **简化的文献记录集合**（包括6篇文献）：  001专家系统在情报检索中的应用(标引词:专家系统;智能检索系统)  002一种新的倒排档溢出处理算法(标引词:倒排档;溢出处理)  003情报检索专家系统的特点与发展(标引词:专家系统;智能检索系统)  004提问式中的位置算符(标引词:提问逻辑式;位置算符)  005提问式准波兰变换算法的研究(标引词:提问逻辑式;准波兰变换)  006智能检索系统的设计与开发(标引词:智能检索系统)  **以“#”为记录分隔符，上面6个文献记录的顺序文档（主文档）的机器存放形式如下：** |
| 001专家系统在情报检索中的应用#002一种新的倒排档溢出处理算法#003情报检索专家系统的特点与发展#004提问式中的位置算符#005提问式准波兰变换算法的研究#006智能检索系统的设计与开发# |

相应的主题词倒排档的形式如下：

倒排档  002

提问逻辑式 004，005

位置算符 004

溢出处理 002

智能检索系统 001，003，006

专家系统 001，003

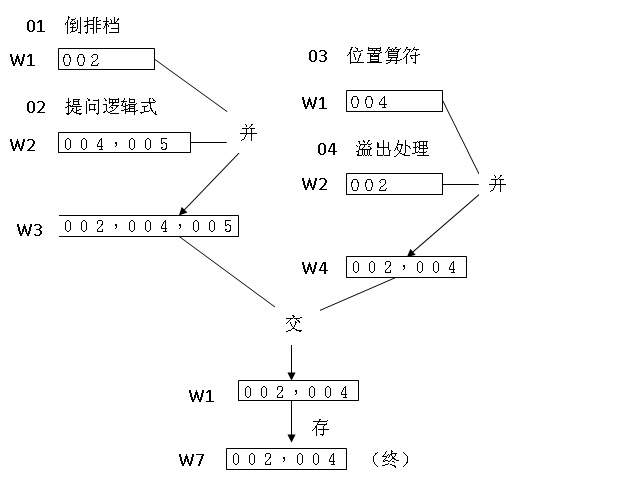
准波兰变换 005

对如下布尔检索式进行逆波兰变换：

（倒排档＋提问逻辑式）\*（位置算符＋溢出处理）.



检索处理过程示意（即执行检索指令序列）：



可以看出，对布尔检索式

（倒排档＋提问逻辑式）\*（位置算符＋溢出处理）.

的处理结果为：集合{002，004}

 即该布尔检索式对应的命中文献号是：002，004

 于是，可由命中文献号查找顺排文档（主文档）：

 002 一种新的倒排档溢出处理算法

 004 提问式中的位置算符

**3.3 准波兰变换法**

针对福岛法存在的逆波兰变换对系统内存空间要求过高的问题，我国学者提出了一种改进的方法——准波兰变换法。

思路：利用二叉树表示检索式，并对二叉树进行一定的调整，再通过二叉树的操作实现检索式的变换。

目的：检索处理所需内存工作区个数最少。

工具：采用表达式二叉树。

表达式二叉树：将一个表达式用一棵二叉树表示，其中运算项作为树的叶子结点，运算符作为树或子树的根结点。

如二个检索式： ① (A+B)\*(C+D).   ② A+B\*(C+D).

方法：将检索式表示为二叉树；调整二叉树的结构使大枝在左、小枝在右；对调整后的二叉树进行后序遍历。

分析A+B\*(C+D)对应的二叉树，其后序遍历结果：ABCD+\*+。   此时按照福岛法的变换处理过程，需要使用5个工作区。这里依据的是逆波兰表达式的运算规则，即遇到运算项必须保存下来（用工作区）。

而如果先处理大的分枝，后处理小的分枝，即调整树结构，使其后序遍历结果变成： CD+B\*A+ ，此时就只需要使用3个工作区。

通过分析对应二叉树，可以发现：

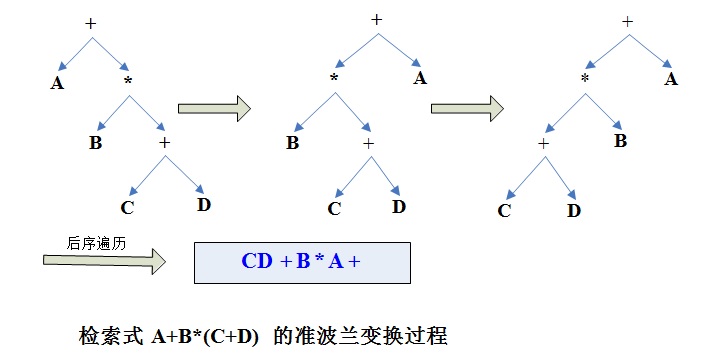
当树的两棵子树不对称时，先处理较大分支，占用工作区较少。

于是给出对检索式的准波兰变换规则:

(1)创建检索式的二叉树表示形式；

(2)比较二叉树的每一层次上的左、右子树是否对称。若不对称，则把大的分枝保留或调整到左边，小的分枝保留或调整到右边，直到全部节点的左右子树窦处理完毕。

(3)后序遍历该二叉树，结点输出序列就是检索式的准波兰形式。



准波兰变换法的分析：

（1）该方法可以将内存工作区从7个降低至5个，其算法比逆波兰变换稍微复杂一些。统计认为，提问式中的检索词个数在20个以内，都是适用的。

（2）对于已采用逆波兰变换法的检索系统，只要增加一个重组模块，即可获得理想的准波兰变换结果。

**四、程序、代码内容**（要求按程序设计规范描述）

**4.1 逆波兰转换**

def is\_operator(char):  
 # 判断字符是否是运算符  
 operators = ['+'**,** '-'**,** '\*'**,** '/']  
 return char in operators  
  
def precedence(operator):  
 # 定义运算符的优先级  
 if operator in ['+'**,** '-']:  
 return **1** elif operator in ['\*'**,** '/']:  
 return **2** return **0**def infix\_to\_rpn(expression):  
 # 将中缀表达式转换为逆波兰表达式  
 rpn\_stack = [] # 用于存储逆波兰表达式运算项的栈  
 operator\_stack = [] # 用于存储运算符的栈  
 tokens = expression.split() # 将表达式拆分成单个字符或数字的列表  
  
 for token in tokens:  
 if token.isalnum():  
 # 如果是数字或字母，则直接添加到逆波兰表达式栈  
 rpn\_stack.append(token)  
 elif is\_operator(token):  
 # 如果是运算符  
 while (operator\_stack and is\_operator(operator\_stack[-**1**]) and  
 precedence(operator\_stack[-**1**]) >= precedence(token)):  
 # 当运算符栈不为空且栈顶元素为运算符且栈顶运算符的优先级大于等于当前运算符时，将栈顶运算符弹出并添加到逆波兰表达式栈  
 rpn\_stack.append(operator\_stack.pop())  
 operator\_stack.append(token) # 将当前运算符添加到运算符栈  
 elif token == '(':  
 # 如果是左括号，则直接添加到运算符栈  
 operator\_stack.append(token)  
 elif token == ')':  
 # 如果是右括号，则将运算符栈中的运算符弹出并添加到逆波兰表达式栈，直到遇到左括号  
 while operator\_stack and operator\_stack[-**1**] != '(':  
 rpn\_stack.append(operator\_stack.pop())  
 operator\_stack.pop() # 弹出左括号  
  
 # 将剩余的运算符添加到逆波兰表达式栈  
 while operator\_stack:  
 rpn\_stack.append(operator\_stack.pop())  
  
 return ' '.join(rpn\_stack) # 将逆波兰表达式栈中的元素用空格连接成字符串并返回  
  
  
# 主程序  
search\_expression = input("请输入检索式：")  
rpn\_expression = infix\_to\_rpn(search\_expression)  
  
print("逆波兰表达式："**,** rpn\_expression)

**4.2 准波兰转换**

# 实现准波兰表达式转换  
def polish\_notation(expression):  
 #定义所需要的工作区  
 operators = ['+'**,** '-'**,** '\*'**,** '/']  
 pn\_stack = []#准波兰表达式栈  
  
 # 将逆波兰表达式中的字符串转换成列表  
 expression\_list = expression.split()  
 for i in expression\_list:  
 # 遍历逆波兰表达式，把遇到的运算符之前的运算项压入栈。  
 pn\_stack.append(i)  
 print(pn\_stack)  
 if i in operators:  
 del pn\_stack[-**1**]#遍历时将算符也遍历了进去，应先删除。  
 #将栈底的两个运算项弹出  
 a = pn\_stack.pop()  
 b = pn\_stack.pop()  
 #遇到运算符后，比较字符栈底的两项的运算项长度  
 if len(a)>len(b):  
 pn\_stack.append(a+b+i)#如果栈底运算项长度大于倒数第二项，将二者先翻转顺序再和算符拼接  
 else:  
 pn\_stack.append(b+a+i)#否则，直接拼接  
 return ' '.join(pn\_stack) # 将准波兰表达式栈中的元素用空格连接成字符串并返回  
  
  
#主程序  
search\_expression = input("请输入逆波兰表达式：")  
pn\_expression= polish\_notation(search\_expression)  
  
print(pn\_expression)

**五、结论及分析：**

本部分统一使用检索式：**A+B\*(C+D).**

**5.1 逆波兰转换**

1.首先判断输入字符是否为运算符并对运算符进行优先级定义。代码如下所示：

def is\_operator(char):  
 # 判断字符是否是运算符  
 operators = ['+'**,** '-'**,** '\*'**,** '/']  
 return char in operators  
  
def precedence(operator):  
 # 定义运算符的优先级  
 if operator in ['+'**,** '-']:  
 return **1** elif operator in ['\*'**,** '/']:  
 return **2** return **0**

这段代码定义了两个函数is\_operator和precedence用于在逆波兰转换函数中判断检索式为运算项和运算符。

2.定义所用到的工作区，代码如下:

def infix\_to\_rpn(expression):  
 # 将中缀表达式转换为逆波兰表达式  
 rpn\_stack = [] # 用于存储逆波兰表达式运算项的栈  
 operator\_stack = [] # 用于存储运算符的栈  
 tokens = expression.split() # 将表达式拆分成单个字符或数字的列表

上述代码定义了函数infix\_to\_rpn用于将中缀检索式转换为逆波兰表达式。又定义了三个list类型变量rpn\_stack，operator\_stack，tokens用于储存逆波兰表达式运算项，运算符，将表达式拆分后的单个字符或数字。

3.接下来为逆波兰转换部分，负责将中缀表达式转换为逆波兰表达式，代码如下：

for token in tokens:  
 if token.isalnum():  
 # 如果是数字或字母，则直接添加到逆波兰表达式栈  
 rpn\_stack.append(token)  
 elif is\_operator(token):  
 # 如果是运算符  
 while (operator\_stack and is\_operator(operator\_stack[-**1**]) and  
 precedence(operator\_stack[-**1**]) >= precedence(token)):  
 # 当运算符栈不为空且栈顶元素为运算符且栈顶运算符的优先级大于等于当前运算符时，将栈顶运算符弹出并添加到逆波兰表达式栈  
 rpn\_stack.append(operator\_stack.pop())  
 operator\_stack.append(token) # 将当前运算符添加到运算符栈  
 elif token == '(':  
 # 如果是左括号，则直接添加到运算符栈  
 operator\_stack.append(token)  
 elif token == ')':  
 # 如果是右括号，则将运算符栈中的运算符弹出并添加到逆波兰表达式栈，直到遇到左括号  
 while operator\_stack and operator\_stack[-**1**] != '(':  
 rpn\_stack.append(operator\_stack.pop())  
 operator\_stack.pop() # 弹出左括号

上述代码首先遍历tokens[]栈，使用isnum函数令程序在遇到字母或数字（即运算项）时直接将其压入逆波兰输出栈。

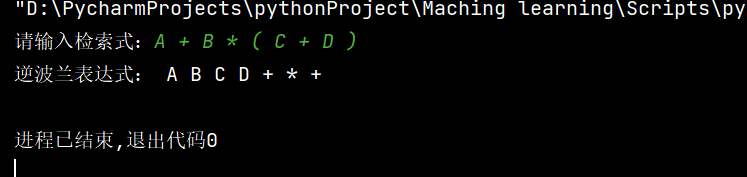
而后使用is\_operator函数判断token是否为算符，如果是运算符，当运算符栈不为空且栈顶元素为运算符且栈顶运算符的优先级大于等于当前运算符时，将栈顶运算符弹出并添加到逆波兰表达式栈。

最后是遇到括号的情况，在遇到左括号时，直接压入算符栈，遇到右括号时则将运算符栈中的运算符弹出并添加到逆波兰表达式栈，直到遇到左括号。最后弹出左括号即可。

1. 最后，将算符栈剩余的算符压入逆波兰输出栈，使用主程序启动即可。代码如下：

# 将剩余的运算符添加到逆波兰表达式栈  
 while operator\_stack:  
 rpn\_stack.append(operator\_stack.pop())  
  
 return ' '.join(rpn\_stack) # 将逆波兰表达式栈中的元素用空格连接成字符串并返回  
  
  
# 主程序  
search\_expression = input("请输入检索式：")  
rpn\_expression = infix\_to\_rpn(search\_expression)  
  
print("逆波兰表达式："**,** rpn\_expression)

1. 使用检索式运行结果如下所示：



**5.2 准波兰变换**

根据准波兰变换原理我们得知该算法是基于二叉树的变换实现的，而逆波兰表达式则可以看作一种子树根节点表示在后的二叉树，因此准波兰变换可由逆波兰表达式实现。

在代码中我们首先定义准波兰变换函数Polish\_notation,然后定义了一个运算符栈operators和一个准波兰表达式栈pn\_stack。

# 实现准波兰表达式转换  
def polish\_notation(expression):  
 #定义所需要的工作区  
 operators = ['+'**,** '-'**,** '\*'**,** '/']  
 pn\_stack = []#准波兰表达式栈

准备工作完成后，我们将逆波兰表达式分割为列表。

# 将逆波兰表达式中的字符串转换成列表  
expression\_list = expression.split()

我们遍历逆波兰表达式，遇到算符之前，运算项全部压入栈。

for i in expression\_list:  
 # 遍历逆波兰表达式，把遇到的运算符之前的运算项压入栈。  
 pn\_stack.append(i)

遇到算符之后，将栈顶的两项算符比较长度，长度长的相当于子树的深度更深，应放在树的右侧,需要反转顺序，反之则不变。最后将算符和反转后的运算项结合在一起即可。

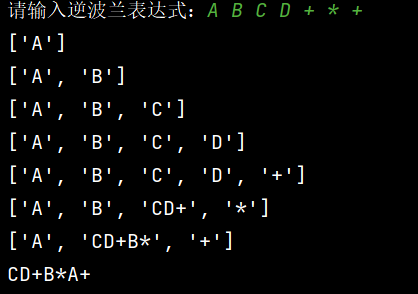
if i in operators:  
 del pn\_stack[-**1**]#遍历时将算符也遍历了进去，应先删除。  
 #将栈底的两个运算项弹出  
 a = pn\_stack.pop()  
 b = pn\_stack.pop()  
 #遇到运算符后，比较字符栈底的两项的运算项长度  
 if len(a)>len(b):  
 pn\_stack.append(a+b+i)#如果栈顶运算项长度大于第二项，将二者先翻转顺序再和算符拼接  
 else:  
 pn\_stack.append(b+a+i)#否则，直接拼接  
return ' '.join(pn\_stack) # 将准波兰表达式栈中的元素用空格连接成字符串并返回

这段代码中，a,b分别为栈顶的运算项，先将他们弹出，比较大小后拼接。

最后依旧运行主程序：

#主程序  
search\_expression = input("请输入逆波兰表达式：")  
pn\_expression= polish\_notation(search\_expression)

结果和栈内字符变化如下：



与二叉树变换相比较，结果相同：

