Data Structures and Algorithm Analysis

7

Dr. Syed Asim Jalal
Department of Computer Science
University of Peshawar

Stack Implementation Using Linked List

- We can avoid the size limitation of a stack implemented with an array by using a linked list to hold the stack elements.
- We, however, need to decide where to insert elements in the list and where to remove them so that push and pop will run the fastest.
- For example, should we insert and remove at Tail or Head of the list.

- The question is at which end of the Linked List should we implement Push and Pop operation. Front or End of a Linked List.
  - We Should use singly linked list to implement Stack, as Stack do not need to traverse in any direction. So doubly linked list is not an option as it needs more memory.
  - In singly-linked list, Insertion of a node in the start takes constant time.
  - Removing element at the end would involve first traversing till the one node before the end. So removing from start is efficient compared to the End.
  - Removing an element at the start is also constant time operation.

- It makes sense to use singly List for Stack implementation and use Start of the linked list for Push and Pop operations as both adding a new node and removing a node would take constant time and less memory.

Stack Implementation using Linked List

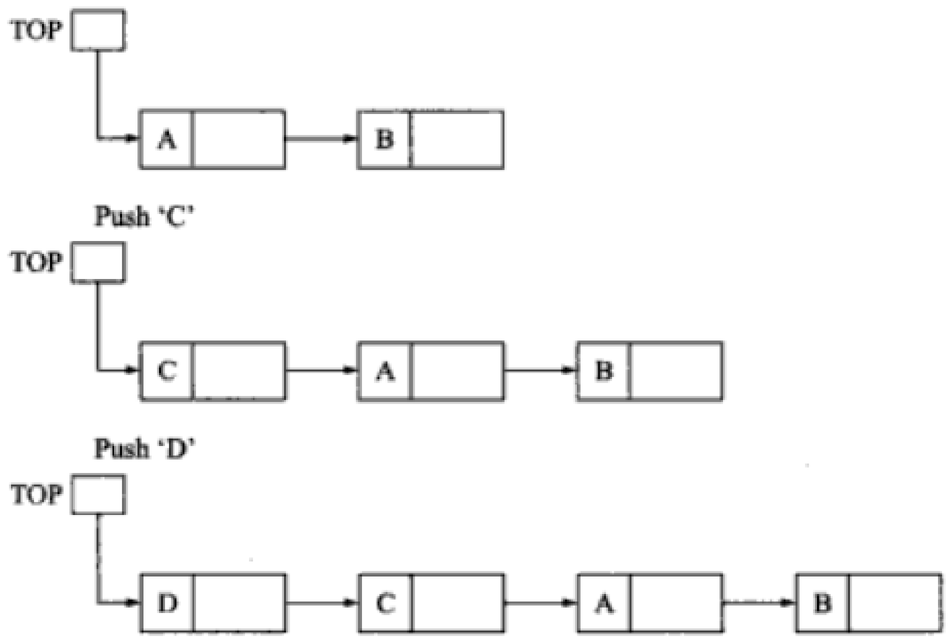


Fig. 6.5 *Pushing an Element into Stack using Linked List*

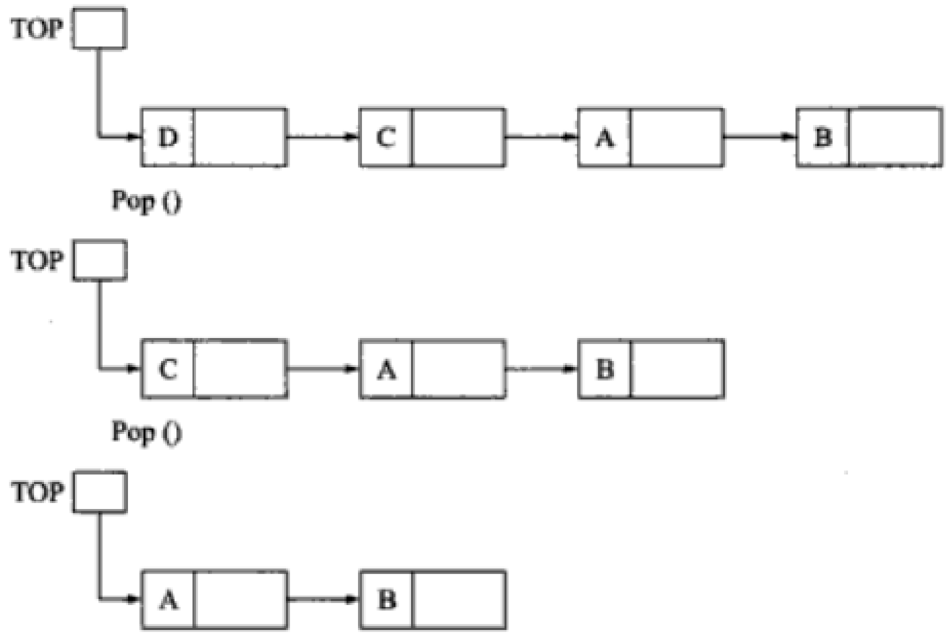


Fig. 6.6 *Popping an Element from the Stack using Linked List*

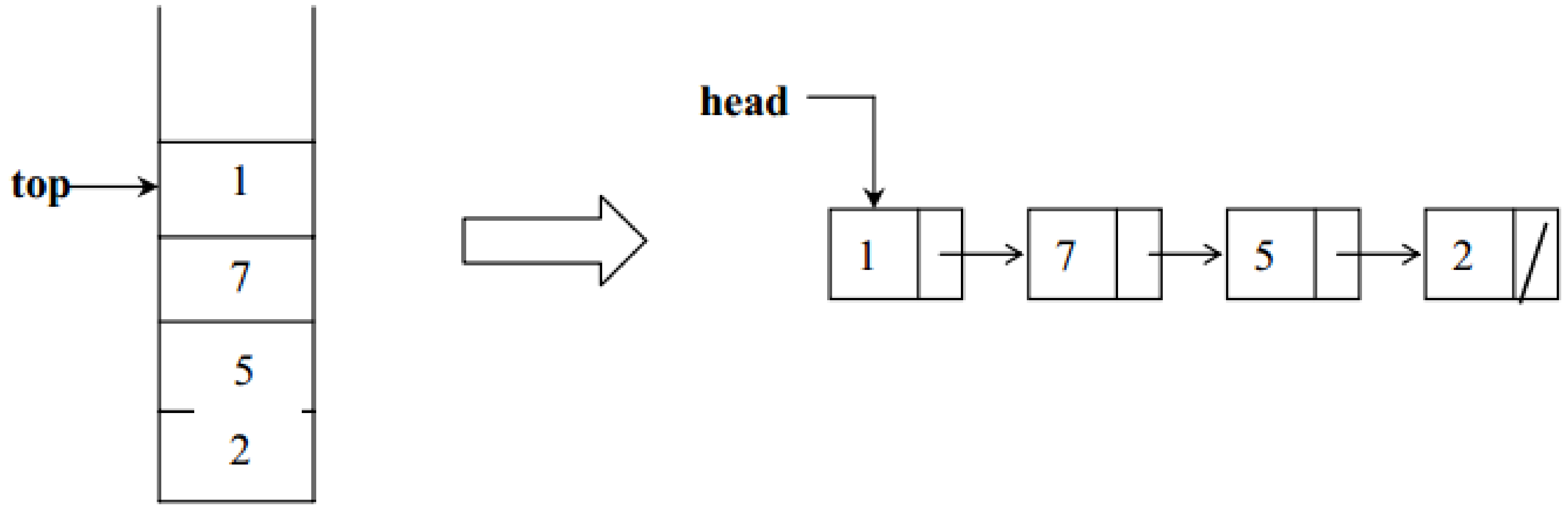


Fig 1. Stack using array (on left side) and linked list (on right side)

Stack using Linked List

- Push Operation algorithm

```
1. Input Data
2. Create a new Node pointed by nNode
3. nNode → value = Data
4. nNode → next = Head
5. Head = nNode
```

- POP Operation

```
If (Head == Null)
    Print Stack Under Flow
    Exit
Get data from the head node. d = head→data
temp= Head
Head = Head → next
Free temp
```

- Impementing IsEmpty()
- If (Head == Null)
  - Return True

Stack Implementation

Mathematical Expression Evaluation

Application of Stack:
Processing Mathematical Expressions

Application of Stack: Mathematical Expressions

- Processing Prefix, Infix and Postfix mathematical expressions
- Suppose we want to add A and B. We have three possible representations.
  - A+B: Infix notation
  - AB+: Postfix notation
  - +AB: Prefix notation
- The prefixes, Pre, In and Post refers to the position of operator

Need for Postfix and Prefix notations

- We normally use infix notations, but it has some problems. In Infix operator comes between two operands
- While evaluating infix notation we have priorities of operators. We can increase priorities by using parenthesis. Infix notations are easy for humans as they can see entire expression at once.
- While computer can read only ONE symbol at a time and cannot observe if there are parenthesis or any higher order operators in the expression **ahead**.
- Computer, therefore, cannot process Infix expressions like Humans.

- Consider A + B * C
  - We here know that multiplication should be done before addition but computer can not figure out if it only reads one character or symbol at a time
    - A + (B * C)
- We therefore came up with Prefix and Postfix notations.
- **In Prefix and Postfix notations we do not care about priorities and brackets or parenthesis.**

- Conversion to postfix

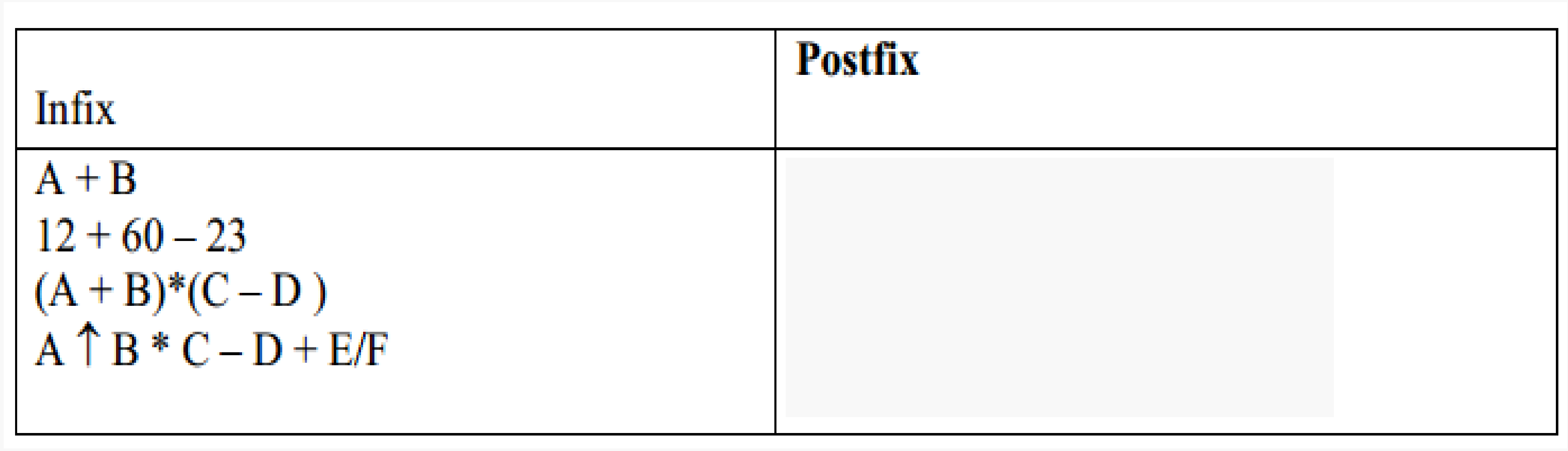
– A + (B * C)	infix form
– A + (B C *)	convert multiplication
– A (B C *) +	convert addition
– **A B C * +**	postfix form

- Conversion to postfix

– (A + B) * C	infix form
– (A B +) * C	convert addition
– (A B +) C *	convert multiplication
– **A B + C ***	postfix form

–

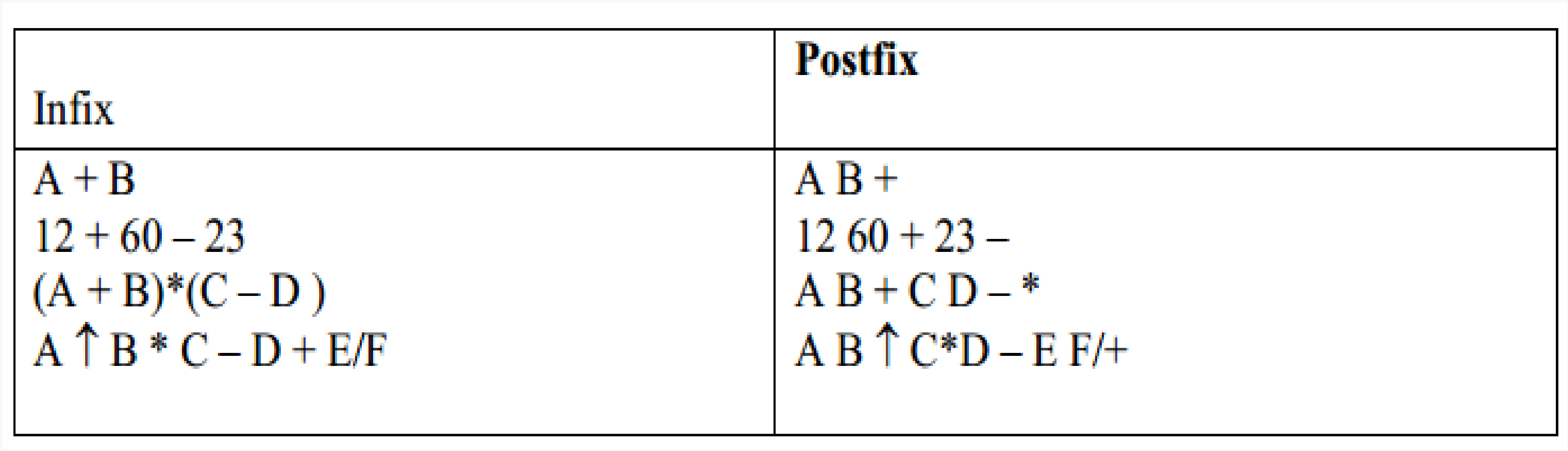
Infix to Postfix Examples



Infix	Postfix
A + B	
12 + 60 – 23	
(A + B)*(C – D)	
A ↑ B * C – D + E/F	

The up-sided arrow represent power operator, A^B

Infix to Postfix Examples



Infix	Postfix
A + B	A B +
12 + 60 – 23	12 60 + 23 –
(A + B)*(C – D)	A B + C D – *
A ↑ B * C – D + E/F	A B ↑ C*D – E F/+

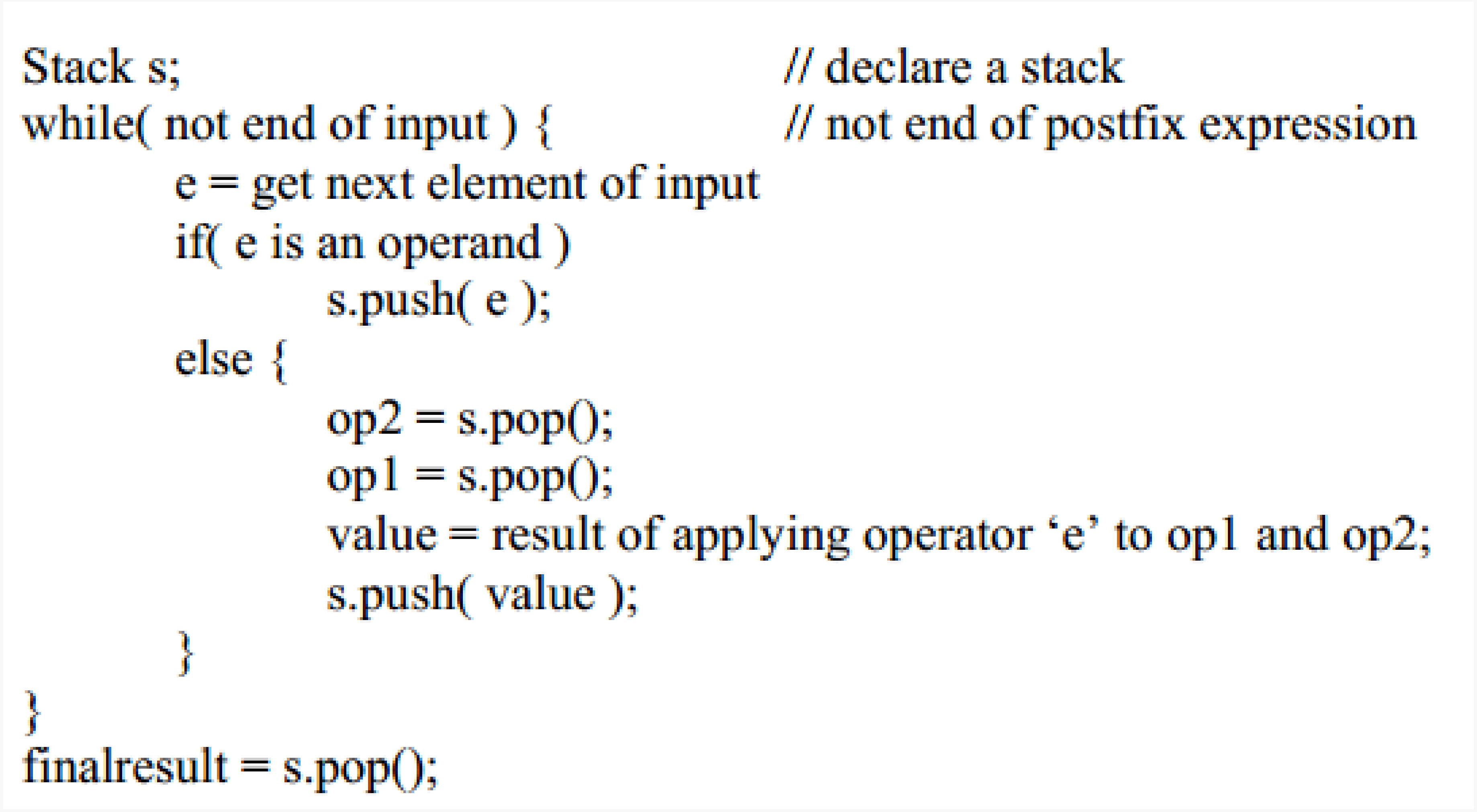
The up-sided arrow represent power operator, A^B

- Role of stack
- Stack is used both in the
  - conversion of Infix to Postfix as well as in
  - evaluating Infix notations after conversion.

Evaluating Postfix using Stack

- Each operator in a postfix expression refers to the previous two operands.
- Each time we read an operand, we PUSH it on a stack.
- When we reach an operator, we POP the two operands from the top of the stack, apply the operator and PUSH the result back on the stack.

Evaluating Postfix Algorithms



```
Stack s;                                    // declare a stack
while( not end of input ) {                 // not end of postfix expression
        e = get next element of input
        if( e is an operand )
                s.push( e );
        else {
                op2 = s.pop();
                op1 = s.pop();
                value = result of applying operator ‘e’ to op1 and op2;
                s.push( value );
        }
}
finalresult = s.pop();
```

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2	1	7	7	7,2
↑	7	2	49	49
3	7	2	49	49,3
+	49	3	52	52

Converting Infix to Postfix

Converting Infix to Postfix

- Consider the following infix expressions ‘A+B*C’ and ‘ (A+B)*C’.
- The postfix versions are ‘ABC*+’ and ‘AB+C*’.
- The order of operands in postfix is the same as the infix.
- In scanning from left to right, the operand ‘A’ can be inserted into postfix expression.
- The ‘+’ cannot be inserted until its second operand has been scanned and inserted.
- The ‘+’ has to be stored away until its proper position is found.
- When ‘B’ is seen, it is immediately inserted into the postfix expression.
- Can the ‘+’ be inserted now? In the case of ‘A+B*C’? **No because * has precedence over +**

- In case of ‘(A+B)*C’, the closing parenthesis indicates that ‘+’ must be performed first.
- **So the algorithm needs a procedure to determine precedence of operators**
  - Lets assume we have a function ‘precedence(op1,op2)’ where op1 and op2 are two operators.
  - ‘precedence(op1,op2)’
    - returns TRUE if op1 has precedence over op2,
    - otherwise FALSE is returned.

- precedence(‘*’,’+’) returns TRUE
- precedence(‘+’,’+’) returns TRUE
- precedence(‘+’,’*’) returns FALSE

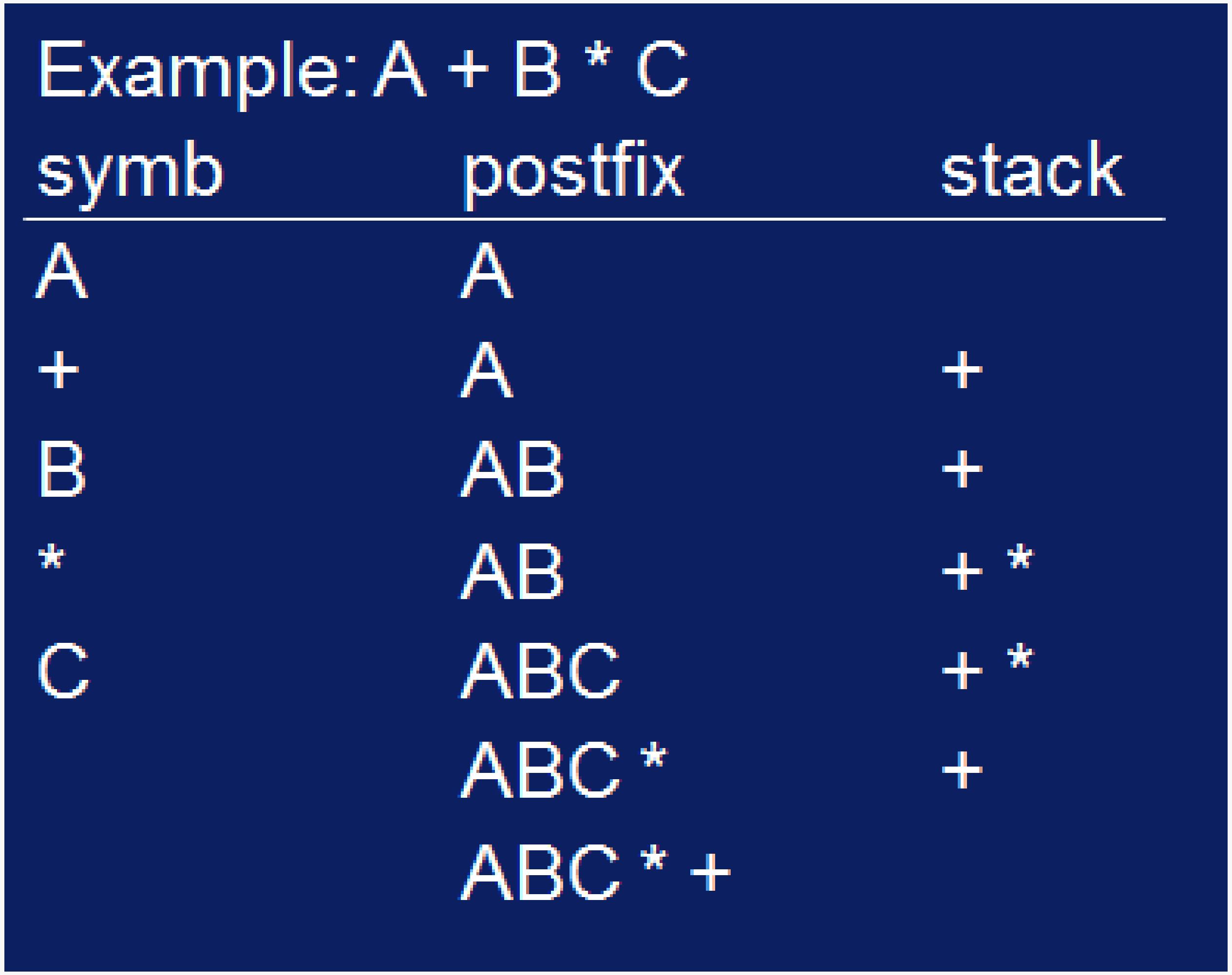
  - Based on this precedence function we will make an algorithm that converts infix expression to its postfix form.
  - First we will consider infix expression without any parenthesis.

Converting Infix to Postfix Algorithm

<u>For infix expression without parenthesis.</u>

```
Stack s;
While( not end of input )
{
    c = next input character;
    if( c is an operand )
        add c to postfix string;
    else
     {  while( !s.empty()  AND   precedence(s.top(), c)==TRUE  )
            {
                    op = s.pop();
                    add op to the postfix string;
            }
            s.push( c );
}    }
     while( !s.empty() ) {
        op = s.pop();
        add op to postfix string;
     }
```

Pop and insert all the operators in the postfix expression



Example: A + B * C

symb	postfix	stack
A	A	
+	A	+
B	AB	+
*	AB	+ *
C	ABC	+ *
	ABC *	+
	ABC * +	

Algorithm when parenthesis are involved

- We will modify the precedence function to include parenthesis as well.
- When an open parenthesis ‘(‘ is read, it must be pushed on the stack.
  - This is done by setting precedence(operator,‘(‘) to be FALSE.
- Push an operator (any one) on top of stack if the top of stack is ‘(‘.
  - This is done by making precedence(‘(‘, operator) to be FALSE

- When a ‘)’ is read, all operators up to the first ‘(‘ must be popped and placed in the postfix string.
  - To do this, precedence(operator,’)’) returns TRUE.
  - Both the ‘(‘ and the ‘)’ must be discarded:

Summary of new rules for Parenthesis rules for (and)

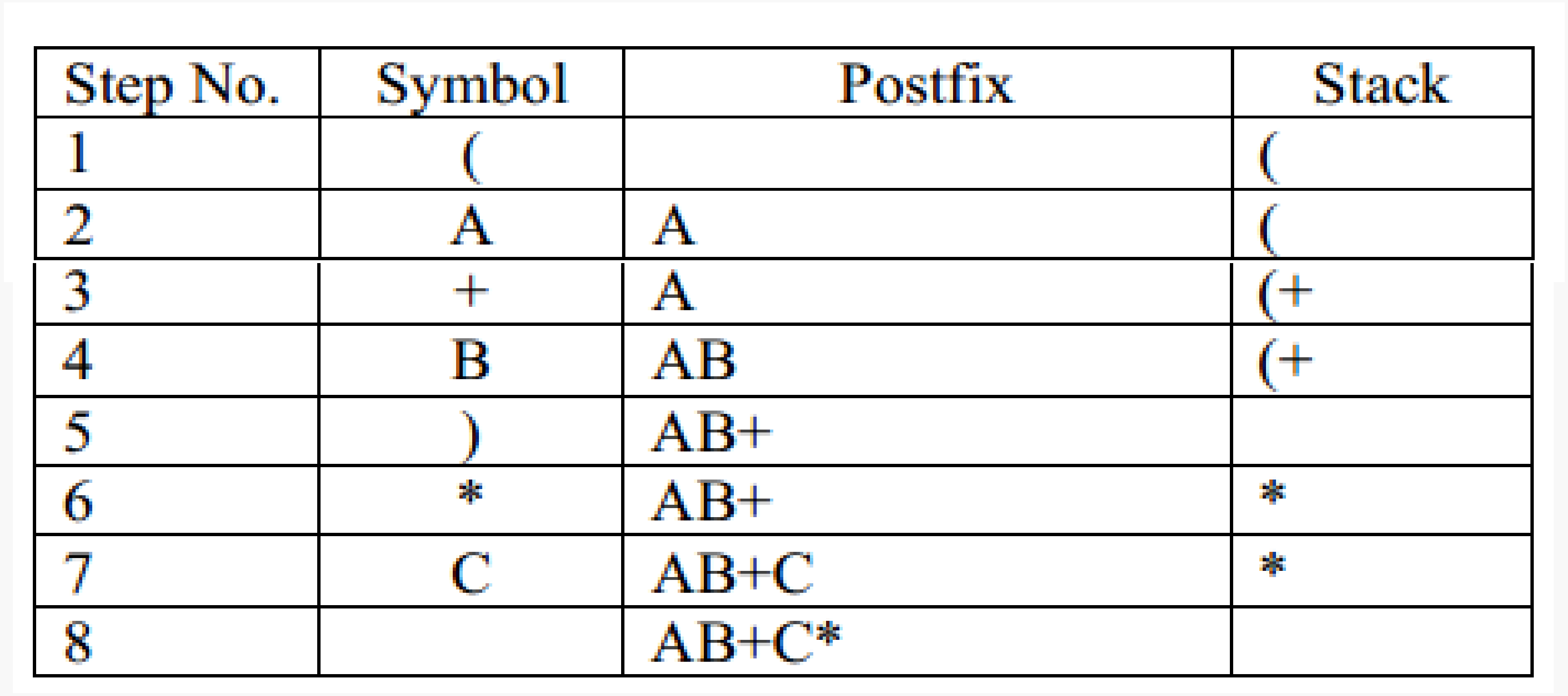
- precedence(operator,'(') = FALSE
- precedence('(', operator) = FALSE
- precedence(operator ,   ')') = TRUE

```
precedence( operator,'(' ) = FALSE
precedence( '(', operator ) = FALSE
precedence(operator,')' ) = TRUE
```

```
Stack s;
While( not end of input )
{
      c = next input character;
      if( c is an operand )
            add c to postfix string;
      else {
                 while( NOT s.empty()  AND  precedence( s.top() , c) )
                  {
                     op = s.pop();
                     add op to the postfix string;
                  }
                  if(      s.empty()  OR    c  !=  ')'  )
                           s.push( c );       //push all operators except ')'
                 else
                        s.pop();              // discard '(' from stack.
            }
}
   while(   !s.empty()   ) {          // pop all remaining operators
           op = s.pop();
           add op to postfix string;
  }
```

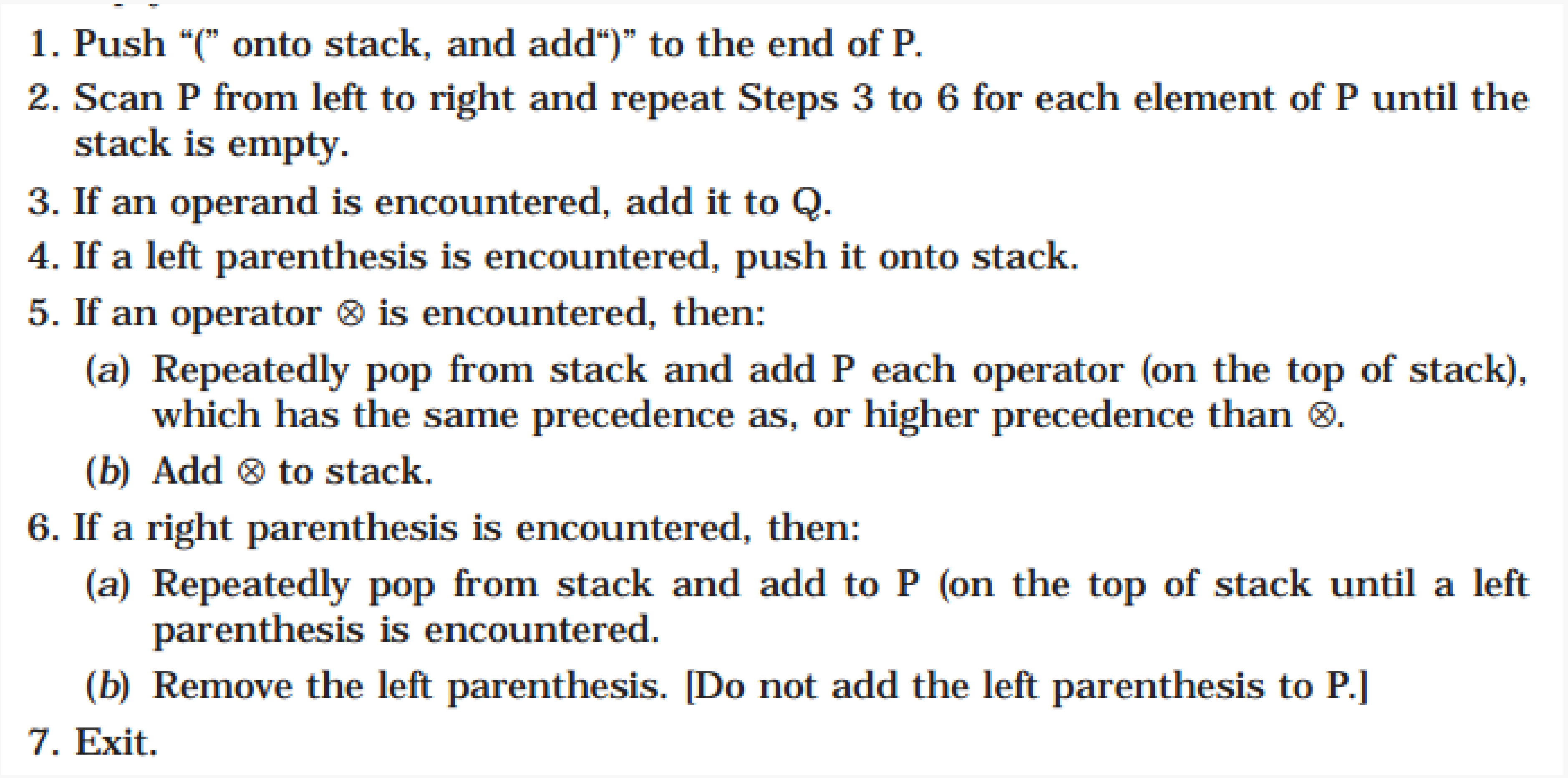
- Assumption for these algorithms is the postfix notations are correct and correctly converted from infix to postfix.

- Example: (A + B) * C



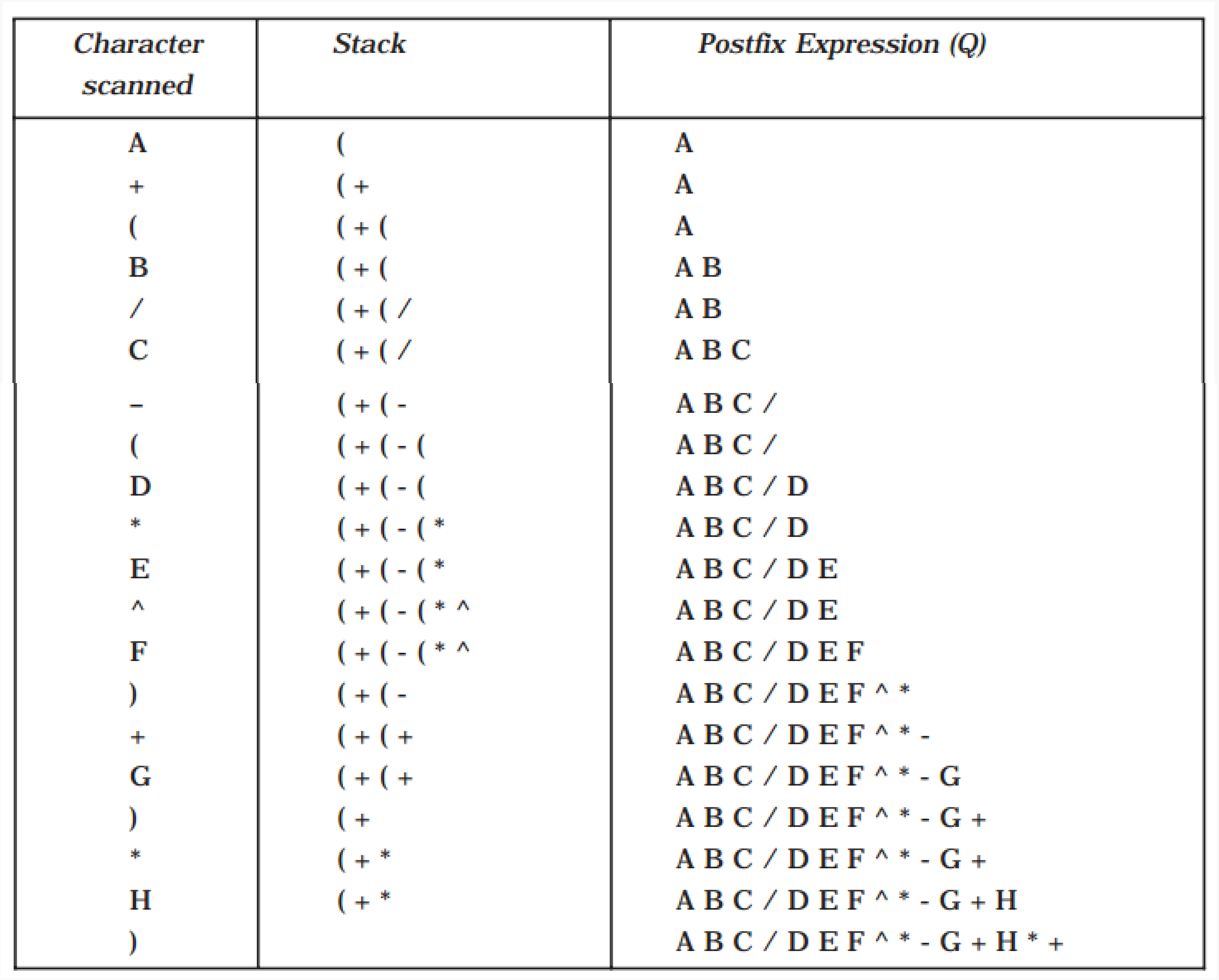
Step No.	Symbol	Postfix	Stack
1	(		(
2	A	A	(
3	+	A	(+
4	B	AB	(+
5	)	AB+	
6	*	AB+	*
7	C	AB+C	*
8		AB+C*	

Another Algorithm for self study.



1. Push "(" onto stack, and add")" to the end of P.
2. Scan P from left to right and repeat Steps 3 to 6 for each element of P until the stack is empty.
3. If an operand is encountered, add it to Q.
4. If a left parenthesis is encountered, push it onto stack.
5. If an operator ⊗ is encountered, then:
   (*a*) Repeatedly pop from stack and add P each operator (on the top of stack), which has the same precedence as, or higher precedence than ⊗.
   (*b*) Add ⊗ to stack.
6. If a right parenthesis is encountered, then:
   (*a*) Repeatedly pop from stack and add to P (on the top of stack until a left parenthesis is encountered.
   (*b*) Remove the left parenthesis. [Do not add the left parenthesis to P.]
7. Exit.

$$P = A + (B / C - (D * E \wedge F) + G) * H$$



Character scanned	*Stack*	*Postfix Expression (Q)*
A	(	A
+	(+	A
(	(+ (	A
B	(+ (	A B
/	(+ (/	A B
C	(+ (/	A B C
–	(+ (-	A B C /
(	(+ (- (	A B C /
D	(+ (- (	A B C / D
*	(+ (- (*	A B C / D
E	(+ (- (*	A B C / D E
^	(+ (- (* ^	A B C / D E
F	(+ (- (* ^	A B C / D E F
)	(+ (-	A B C / D E F ^ *
+	(+ (+	A B C / D E F ^ * -
G	(+ (+	A B C / D E F ^ * - G
)	(+	A B C / D E F ^ * - G +
*	(+ *	A B C / D E F ^ * - G +
H	(+ *	A B C / D E F ^ * - G + H
)		A B C / D E F ^ * - G + H * +