Stack and Queues

Stacks

- Introduction
- ADT of Stack
- Operations on Stack
- Array Implementation of Stack
- Applications of Stack-Well form-ness of Parenthesis
- Infix to Postfix Conversion and Postfix Evaluation, Recursion

Stack and Queues

Queues

Introduction

ADT of Queue

Operations on Queue

Array Implementation of Queue

Types of Queue-Circular Queue

Priority Queue

Introduction of Double Ended Queue

Applications of Queue

Stack

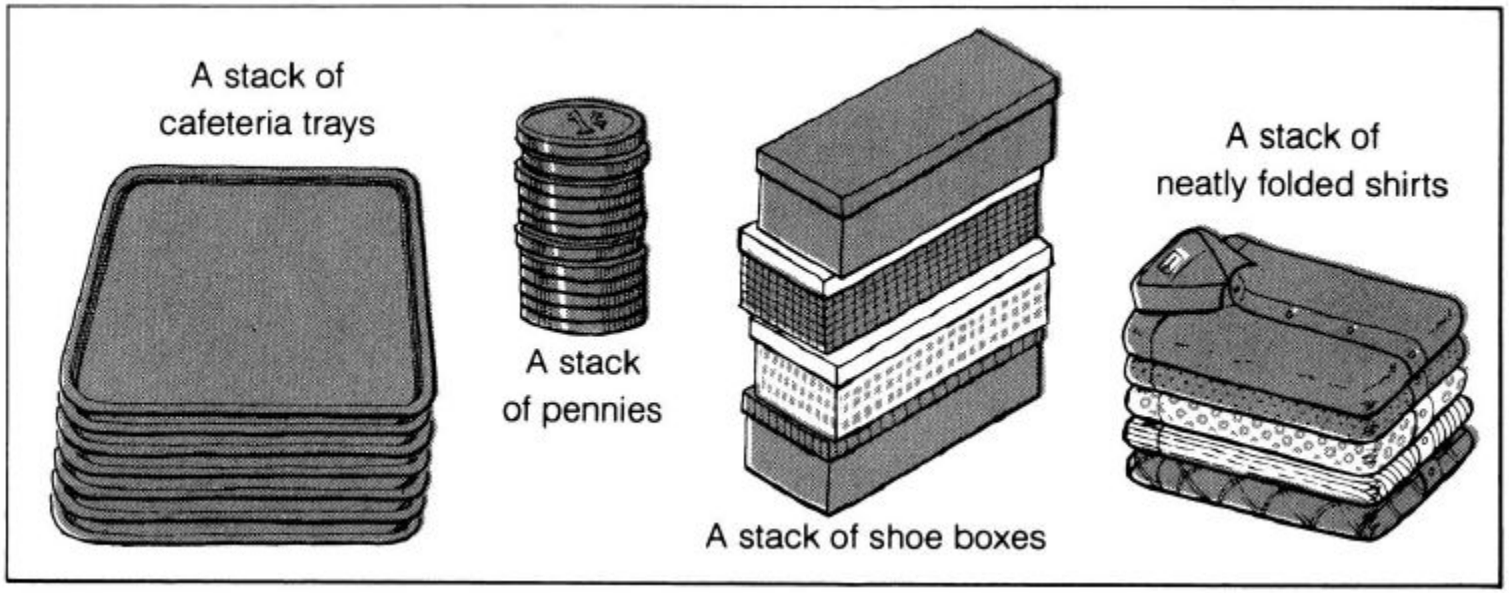
- Stores a set of elements in a particular order
- Stack principle: LAST IN FIRST OUT
- = LIFO
- It means: the last element inserted is the first one to be removed

- Example

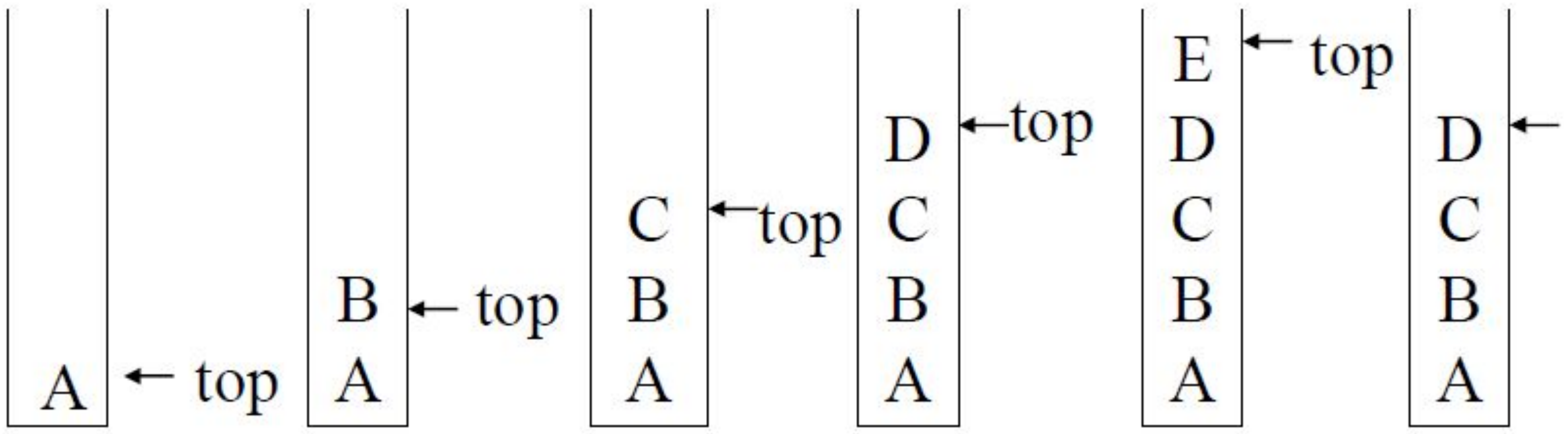
- Which is the first element to pick up?

Stack Definition

- It is an ordered group of homogeneous items of elements.
- Elements are added to and removed from the top of the stack (the most recently added items are at the top of the stack).
- The last element to be added is the first to be removed (**LIFO**: Last In, First Out).



Last In First Out



What is the difference?

Stack & Queue vs. Array

- Arrays are data storage structures while stacks and queues are specialized DS and used as programmer’s tools.
- Stack –a container that allows push and pop
- Queue -a container that allows enqueue and dequeue
- No concern on implementation details.
- In an array any item can be accessed, while in these data structures access is restricted.
- They are more abstract than arrays.

Stack Applications

- *Applications*
  - Real life
  - Pile of books
  - Plate trays
- More applications related to computer science
  - Program execution stack (read more from your text)
  - Evaluating expressions

Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Saving local variables when one function calls another, and this one calls another, and so on.
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Basic Stack Operations

- Initialize the Stack.
- Pop an item off the top of the stack (delete an item)
- Push an item onto the top of the stack (insert an item)
- Is the Stack empty?
- Is the Stack full?
- Clear the Stack
- Determine Stack

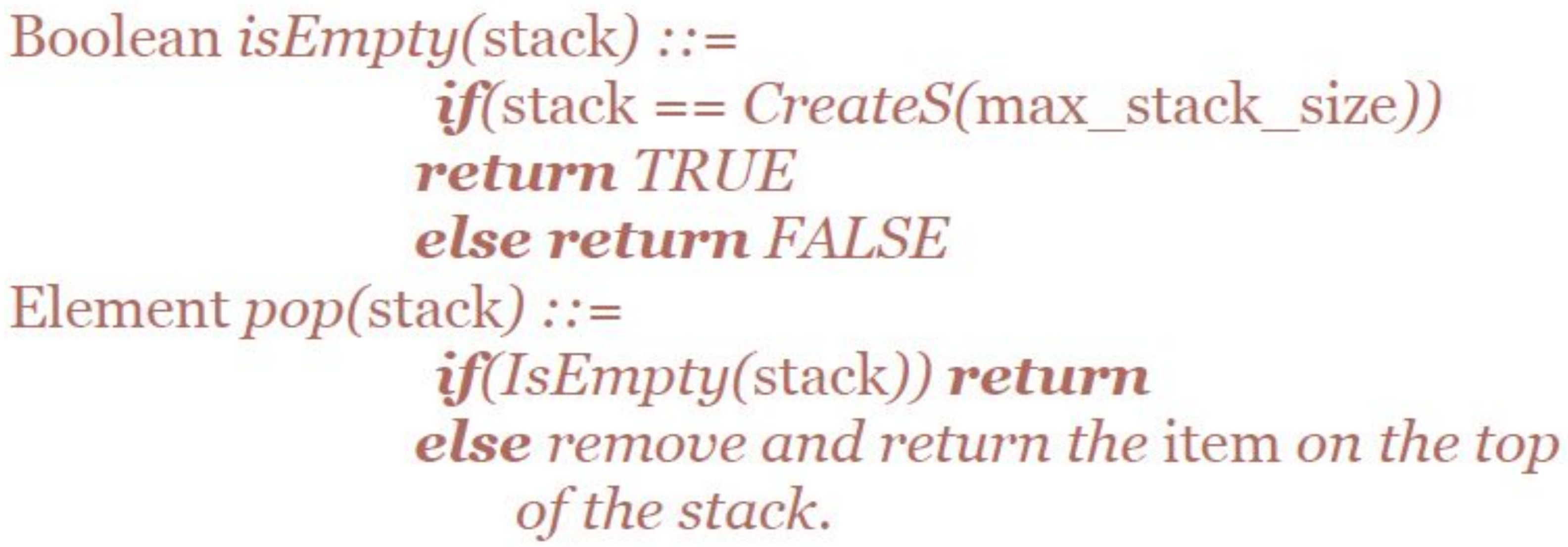
Stack ADT

objects: *a finite ordered list with zero or more elements.*

methods:

```
 for all stack ∈ Stack, item ∈ element, max_stack_size
  ∈ positive integer
 Stack createS(max_stack_size) ::=
         create an empty stack whose maximum size is
         max_stack_size
 Boolean isFull(stack, max_stack_size) ::=
         if (number of elements in stack == max_stack_size)
          return TRUE
         else return FALSE
 Stack push(stack, item) ::=
         if (IsFull(stack)) stack_full
         else insert item into top of stack and return
```

Stack *ADT*



```
Boolean isEmpty(stack) ::=
                if(stack == CreateS(max_stack_size))
                return TRUE
                else return FALSE
Element pop(stack) ::=
                if(IsEmpty(stack)) return
                else remove and return the item on the top
                    of the stack.
```

Run-time Stack

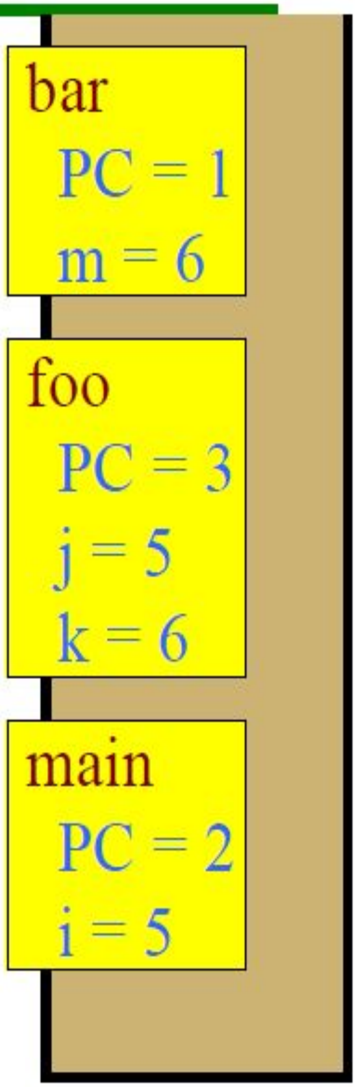
- The C++ run-time system keeps track of the chain of active functions with a stack
- When a function is called, the run-time system pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a function returns, its frame is popped from the stack and control is passed to the method on top of the stack

```
main() {
  int i;

 i = 5;
  foo(i);
  }

foo(int j) {
  int k;
  k = j+1;
  bar(k);
  }

bar(int m) {
  …
  }
```



Array Implementation of the Stacks

- The stacks can be implemented by the use of ***arrays*** and ***linked lists***.
- One way to implement the stack is to have a data structure where a variable called top keeps the location of the elements in the stack (array)
- An array is used to store the elements in the stack

Array-based Stack Implementation

- Allocate an array of some size (pre-defined)
  - Maximum N elements in stack
- Bottom stack element stored at element 0
- last index in the array is the *top*
- Increment *top* when one element is pushed, decrement after pop

Stack Stack Implementation: CreateS, isEmpty, isFull

Stack *createS(max_stack_size)* *::=*

```
 #define MAX_STACK_SIZE 100  /* maximum stack size */
typedef struct {
data_type ST[MAX_STACK_SIZE];
 int top;
 } stack;
 stack s;
```

Stack initialization

```
void initialize_stack(stack *sp)
{
sp->top = -1;
}
```

Stack empty

```
int is_stack_empty
(int top)
{
if(top ==-1)
return 1;
Else
return 0;
}
```

Stack full

```
int is_stack_full
(int top)
{
if(top ==MAX_STACK_SIZE-1)
return 1;
Else
return 0;
}
```

Push

```
void push(stack *sp, element item)
{
if (stack is not full)
 {
sp->top = sp->top + 1;
sp->ST[sp->top] = item;
}
Else
display stack is full;
}
```

Pop

```
element pop(stack *sp)
{
if (stack is not empty)
{
item = sp->ST[sp->top];
sp->top = sp->top –1;
return item;
}
Else
return an error key
}
```

Linked List Implementation of Stack:

- We can represent a stack as a linked list. In a stack push and pop operations are performed at one end called top.

- We can perform similar operations at one end of list using *top* pointer. The linked stack looks as shown in figure

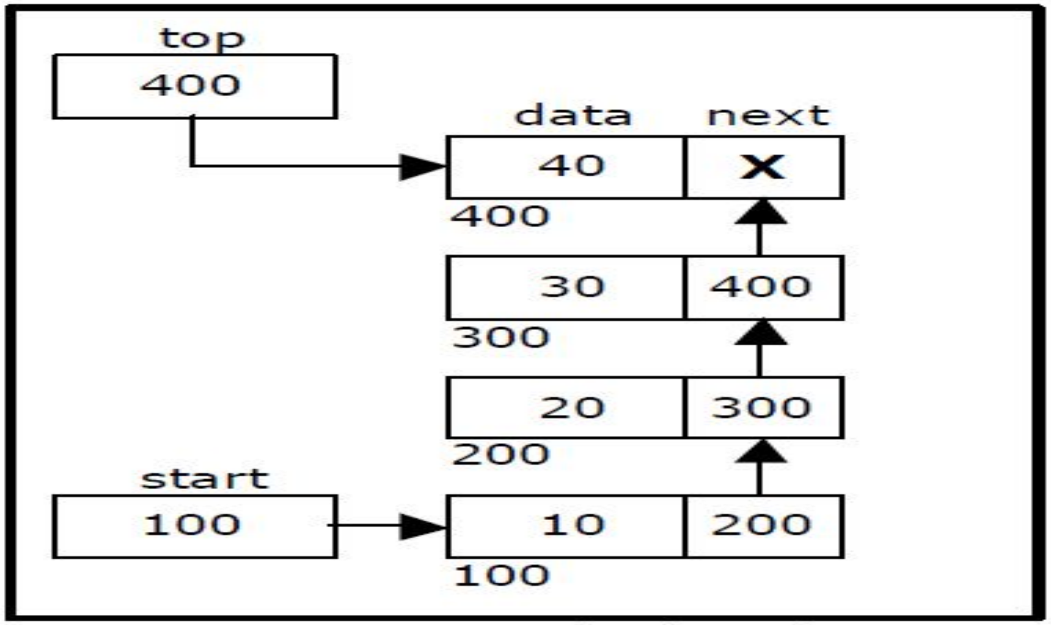


Figure 4.3. Linked stack representation

Linked List Implementation of Stack:

```
typedef struct STACK {
int data ;
struct STACK *down;
} stack;
stack *top;
Initialize top = NULL
```

Linked List Implementation of Stack:*Push*

```
 void push(int X)
{
stack *node;
node = (stack *) malloc (sizeof (stack));
if (node == NULL)
{
printf(“ Stack memory is full\n”);
exit(1);
}
Else
{
node->data = X;
node->down = top;
top = node;
} }
```

Linked List Implementation of Stack:Pop

```
 int pop()
{
int X;
stack *temp;
if (top == NULL)
{
printf(“The stack is empty\n”);
exit(1);
}
Else
{
X = top->data;
temp = top;
top = top->down;
free(temp);return X;}Pop
```

Algebraic Expressions:

- An algebraic expression is a legal combination of operators and operands. Operand is the quantity on which a mathematical operation is performed.

- Operand may be a variable like x, y, z or a constant like 5, 4, 6 etc. Operator is a symbol which signifies a mathematical or logical operation between the operands. Examples of familiar operators include +, -, *, /, ^ etc.

- An algebraic expression can be represented using three different notations. They are infix, postfix and prefix notations:

Infix to Postfix Conversion

- **Infix:** It is the form of an arithmetic expression in which we fix (place) the arithmetic operator in between the two operands.
- Example: (A + B) * (C - D)

- **Prefix:** It is the form of an arithmetic notation in which we fix (place) the arithmetic operator before (pre) its two operands. The prefix notation is called as polish notation (due to the polish mathematician Jan Lukasiewicz in the year 1920).
- Example: * + A B – C D

Infix to Postfix Conversion

Postfix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator after (post) its two operands. The postfix notation is called as *suffix notation* and is also referred to *reverse polish notation*

Example: A B + C D - *

The three important features of postfix expression are:

1. The operands maintain the same order as in the equivalent infix expression.
2. The parentheses are not needed to designate the expression unambiguously.
3. While evaluating the postfix expression the priority of the operators is no longer relevant.

Infix to Postfix Conversion

- We consider five binary operations: +, -, *, / and $ or ↑ (exponentiation). For these binary operations, the following in the order of precedence (highest to lowest):

OPERATOR	PRECEDENCE	VALUE
Exponentiation ($ or ↑ or ^)	Highest	3
*, /	Next highest	2
+, -	Lowest	1

Converting expressions using Stack:

Let us convert the expressions from one type to another. These can be done as follows:

- 1. Infix to postfix
- 2. Infix to prefix
- 3. Postfix to infix
- 4. Postfix to prefix
- 5. Prefix to infix
- 6. Prefix to postfix

Conversion from infix to postfix:

Procedure to convert from infix expression to postfix expression is as follows:

1. Scan the infix expression from left to right.

2. a) If the scanned symbol is left parenthesis, push it onto the stack.

   b) If the scanned symbol is an operand, then place directly in the postfix expression (output).

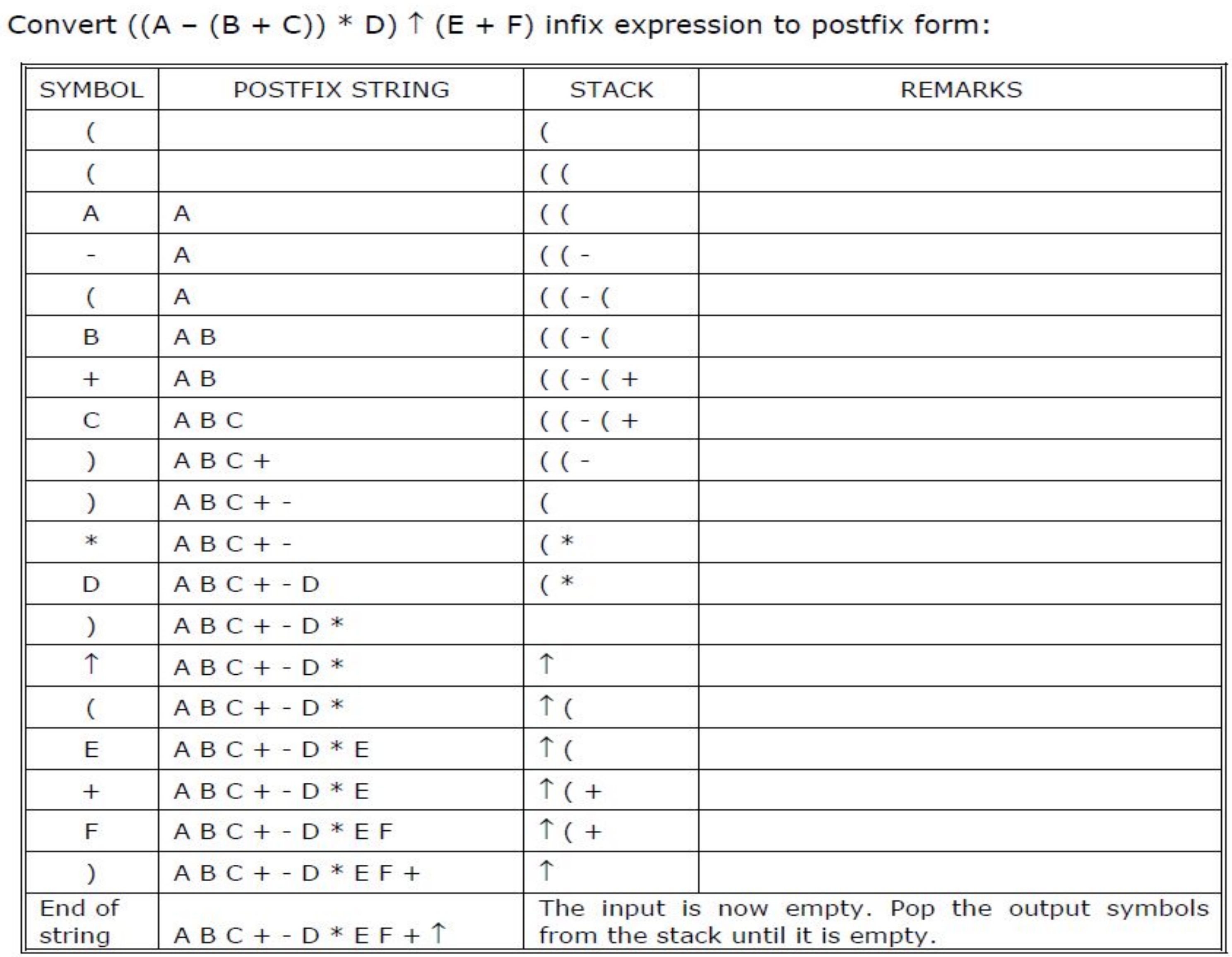
   c) If the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.

Conversion from infix to postfix:

d) If the scanned symbol is an operator, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (*or greater than or equal*) to the precedence of the scanned operator and push the scanned operator onto the stack otherwise, push the scanned operator onto the stack.

$$((A - (B + C)) * D) \uparrow (E + F)$$

Conversion from infix to postfix:



Convert ((A – (B + C)) * D) ↑ (E + F) infix expression to postfix form:

SYMBOL	POSTFIX STRING	STACK	REMARKS
(		(	
(		((	
A	A	((	
-	A	((-	
(	A	((- (	
B	A B	((- (	
+	A B	((- (+	
C	A B C	((- (+	
)	A B C +	((-	
)	A B C + -	(	
*	A B C + -	(*	
D	A B C + - D	(*	
)	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 +	↑	
End of string	A B C + - D * E F + ↑	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 2:

Convert a + b * c + (d * e + f) * g the infix expression into postfix form.

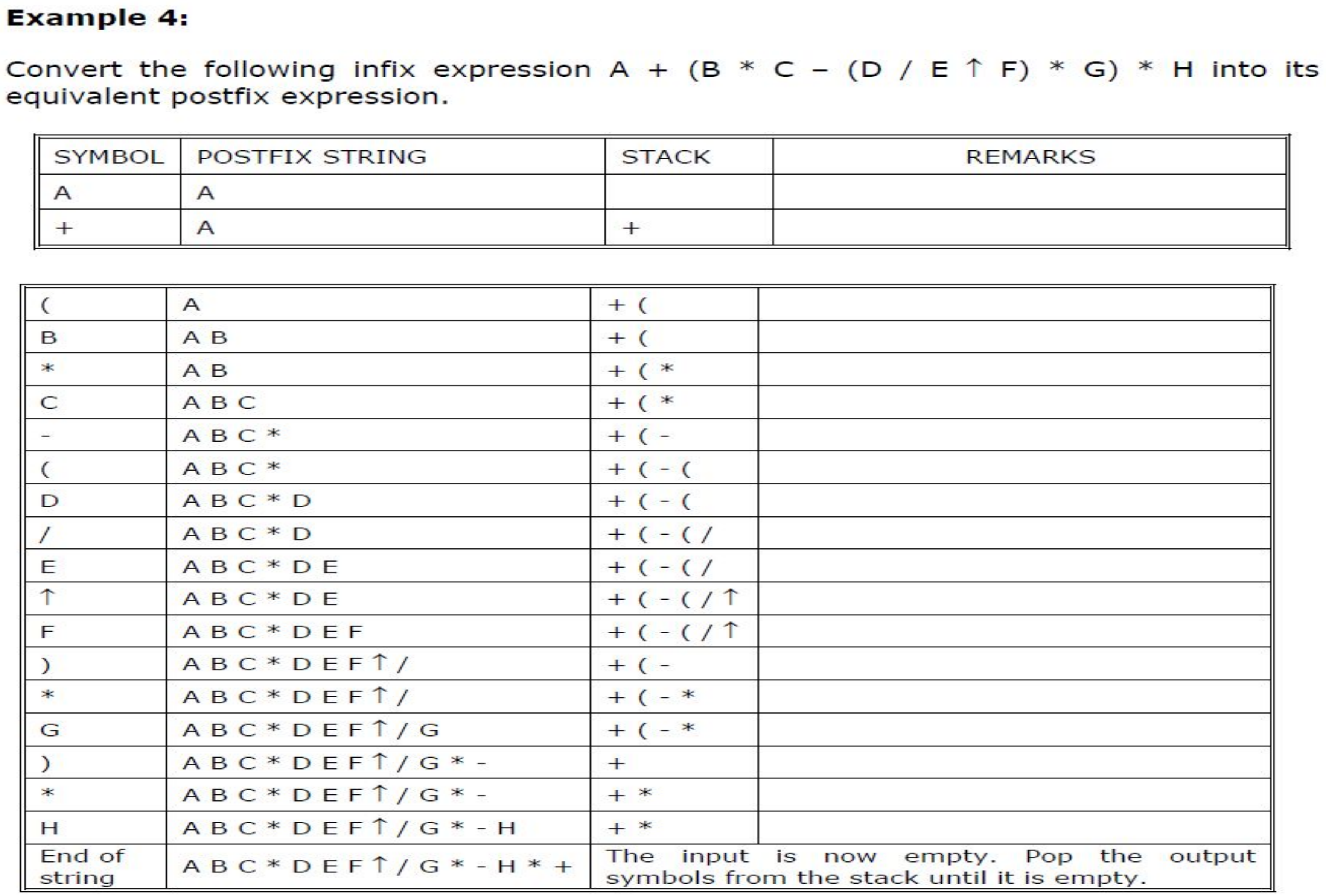
SYMBOL	POSTFIX STRING	STACK	REMARKS
a	a		
+	a	+	
b	a b	+	
*	a b	+ *	
c	a b c	+ *	
+	a b c * +	+	
(	a b c * +	+ (	
d	a b c * + d	+ (	
*	a b c * + d	+ (*	
e	a b c * + d e	+ (*	
+	a b c * + d e *	+ (+	
f	a b c * + d e * f	+ (+	
)	a b c * + d e * f +	+	
*	a b c * + d e * f +	+ *	
g	a b c * + d e * f + g	+ *	
End of string	a b c * + d e * f + g * +	The input is now empty. Pop the output symbols from the stack until it is empty.	

Conversion from infix to postfix:

Example 3:

Convert the following infix expression A + B * C – D / E * H into its equivalent postfix expression.

SYMBOL	POSTFIX STRING	STACK	REMARKS
A	A		
+	A	+	
B	A B	+	
*	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 /	- *	
H	A B C * + D E / H	- *	
End of string	A B C * + D E / H * -	The input is now empty. Pop the output symbols from the stack until it is empty.	



Example 4:

Convert the following infix expression A + (B * C – (D / E ↑ F) * G) * H into its equivalent postfix expression.

SYMBOL	POSTFIX STRING	STACK	REMARKS
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	+ *	
End of string	A B C * D E F ↑ / G * - H * +	The input is now empty. Pop the output symbols from the stack until it is empty.	

Program to convert an infix to postfix expression:

```
# include <string.h>

char postfix[50];
char infix[50];
char opstack[50];                  /*  operator stack  */
int i, j, top = 0;

int lesspriority(char op, char op_at_stack)
{
        int k;
        int pv1;                   /*  priority value of  op  */
        int pv2;                   /*  priority value of op_at_stack  */
        char operators[] = {'+', '-', '*', '/', '%', '^', '(' };
        int priority_value[] = {0,0,1,1,2,3,4};
        if( op_at_stack == '(' )
                return 0;
        for(k = 0; k < 6; k ++)
        {
                if(op == operators[k])
                        pv1 = priority_value[k];
        }
        for(k = 0; k < 6; k ++)
        {
                if(op_at_stack == operators[k])
                        pv2 = priority_value[k];
        }
        if(pv1 < pv2)
                return 1;
        else
                return 0;
}
```

Program to convert an infix to postfix expression:

```
void push(char op)        /* op - operator  */
{
        if(top == 0)
        {
                opstack[top] = op;
                top++;
        }
        else
        {
                if(op != '(' )
                {
                        while(lesspriority(op, opstack[top-1]) == 1 && top > 0)
                        {
                                        postfix[j] = opstack[--top];
                                        j++;
                        }
                }
                opstack[top] = op;      /* pushing onto stack  */
                top++;
        }
}
```

```
/* before pushing the operator
'op' into the stack check priority
of op  with top of opstack if less
then pop the operator from stack
then push into postfix string else
push op onto stack itself  */
```

```
pop()
{
        while(opstack[--top] != '(' )              /*  pop until '(' comes   */
        {
                postfix[j] = opstack[top];
                j++;
        }
}
```

Program to convert an infix to postfix expression:

```
void main()
{
        char ch;
        clrscr();
        printf("\n Enter Infix Expression  : ");
        gets(infix);
        while( (ch=infix[i++]) != '\0')
        {
                switch(ch)
                {
                        case ' ' : break;
                        case '(' :
                        case '+' :
                        case '-' :
                        case '*' :
                        case '/' :
                        case '^' :
                        case '%' :
                                push(ch);                   /* check priority and push  */
                                break;
                        case ')' :
                                pop();
                                break;
                        default :
                                postfix[j] = ch;
                                j++;
                }
        }
        while(top >= 0)
        {
                postfix[j] =  opstack[--top];
                j++;
```

Program to convert an infix to postfix expression:

```
}
postfix[j] = '\0';
printf("\n Infix Expression : %s ", infix);
printf("\n Postfix Expression : %s ", postfix);
getch();
}
```

Stack

Stack

Stack