
MODULE - III

Stacks and Queues

Stacks and Queues

Module Description

This module introduces two closely-related data types for manipulating large collections of objects: stack and the queue. Each of them is basically defined by two simple operations: insert or add a new item, and remove an item. When we add a data item we have a clear intention. However, when we remove an item, we should decide which one to choose. For example, the rule used in case of queue is to always remove the item that has been in the queue for longest time. This policy is known as first-in-first-out or FIFO. And the rule used in case of stack is that we always remove the element that has been in the stack for least amount of time. This policy is known as last-in first-out or LIFO.

Chapter 3.1

Stacks

Chapter 3.2

Queue

Chapter Table of Contents

Chapter 3.1

Stacks

Aim.....	139
Instructional Objectives.....	139
Learning Outcomes.....	139
3.1.1 Introduction to Stack.....	140
(i) Definition of a Stack.....	141
(ii) Array Representation of Stack.....	142
Self-assessment Questions.....	144
3.1.2 Operations on Stack.....	144
Self-assessment Questions.....	149
3.1.3 Polish Notations.....	149
(i) Infix Notation	150
(ii) Prefix Notation	151
(iii) Postfix Notation	152
Self-assessment Questions.....	155
3.1.4 Conversion of Arithmetic Expression from Infix to Postfix	155
Self-assessment Questions.....	159
3.1.5 Applications of Stack	160
(i) Balancing Symbol	160
(ii) Recursion.....	161
(iii) Evaluation of Postfix Expression.....	163
(iv) String Reversal	163
Self-assessment Questions.....	165
Summary	166
Terminal Questions.....	167
Answer Keys.....	168
Activity.....	169
Case Study	170
Bibliography.....	171
e-References	171
External Resources	171
Video Links	171



Aim

To educate and equip the students with skills and technologies of Stacks



Instructional Objectives

After completing this chapter, you should be able to:

- Outline the basic features of stack
- Describe the array representation of stack
- Explain the Polish Notations with example
- Discuss the evaluation of postfix expression using stack
- Explain the steps to convert Infix expression to postfix expression and vice versa
- Outline the applications of stacks



Learning Outcomes

At the end of this chapter, you are expected to:

- Explain operations on stack
- Convert given expressions of infix to prefix and postfix expression
- Explain string recursion applications of stack
- Compute given postfix expression using stack
- Convert to prefix expression for any given infix expression

3.1.1 Introduction to Stack

In this chapter we will introduce a data structure for representing stack as a limited access data structure. Stack data structure is used for manipulating arbitrarily large collections of data. The stack is a data structure which represents objects maintained in a particular order. In this chapter we also explain how to operate a stack data structure. This chapter demonstrates the operations for creating a stack, adding elements to a stack, deleting an element from a stack etc. Some problems have solutions that require the data associated to be arranged or organized as linear list of data elements in which operations are permitted to take place at only one end of the list. The best and the simplest examples are set of books kept one on top of another, set of playing cards, pancake, arranging laundry, stacked plates one above another, etc. Here, we group things together by placing one thing on top of another and then we have to remove things from top to bottom one at a time. The below figure 3.1.1 shows a set of books represented as a stack.



Figure 3.1.1: Picture Representing a Stack

It is interesting that something that is so simple is a critical part of nearly every program that is written. The nested function calls in a running program, conversion of an infix form of an expression to an equivalent postfix or prefix, computing factorial of a number, and so on can be accurately formulated using this simple technique. In all the above cases, it is clear that the one which most recently entered into the list is the first one to be operated. Solution to these types of problems is based on the principle Last-In-First-Out (LIFO) or First-In-Last-Out. A logical structure, which organizes the data and performs operations in LIFO or FILO principle, is termed as a Stack.

(i) Definition of a Stack

Stack is an ordered list of similar data items in which operations such as insertion and deletion are permitted to be done only at one end called top of the stack. It is a linear data structure in which operations can be performed on data objects on principle of Last-In-First-Out or First-In-Last-Out.

More formally, a stack can be defined as an abstract data type with domain of data objects and a set of functions that can be performed on data objects guided by list of axioms.

Some of the important functions used while doing operations of stacks are listed below:

1. **Create-Stack()** - Used for allocating memory
2. **Iseempty(S)** - Used for checking if stack is empty or not; it returns a Boolean
3. **Isfull()** - Used for checking if stack is full; this also return Boolean
4. **Push(S,e)** - Use to add an element on top of stack
5. **Pop(s)** - Used to remove an element from the top of the stack
6. **Top(S)** - Used to display an element in the stack

Also, some axioms are needed to be known while we do operations on stacks. Following is a list of axioms which a programmer must know:

- **Iseempty(Create-Stack()):** Always returns true value
- **Isfull(Create-Stack()):** Always returns false values
- **Iseempty(Push(S, e)):** Always returns false value
- **Isfull(Pop(S)):** Always returns false values
- **Top(push(S, e)):** The element e is displayed
- **Pop(Push(S, e)):** The element e will be removed from stack

The detailed explanation and algorithm for implementing above operations will be covered in forthcoming sections. The figure 3.1.2 demonstrates push() and pop() operations performed on stack.

As shown in the figure, initially stack contains element 1. To push element 2, stack pointer is incremented and then element 2 is pushed. So now stack contains 2 elements *i.e.*, 1 and 2.

In the second step, we push element 3 onto the stack. Thus this element will be placed on top of 2 as the top of stack is pointing one location above 2. Similarly elements 4, 5 and 6 are pushed onto the stack. After pushing element 6 the stack contains total 6 elements.

In the second part of the figure, pop instructions are executed. The first element we can read out is 6 as it is on the top of the stack. And the stack pointer is decremented. Next time if we execute a pop instruction, element 5 will be removed and so on until last element 1 is removed. Thus,

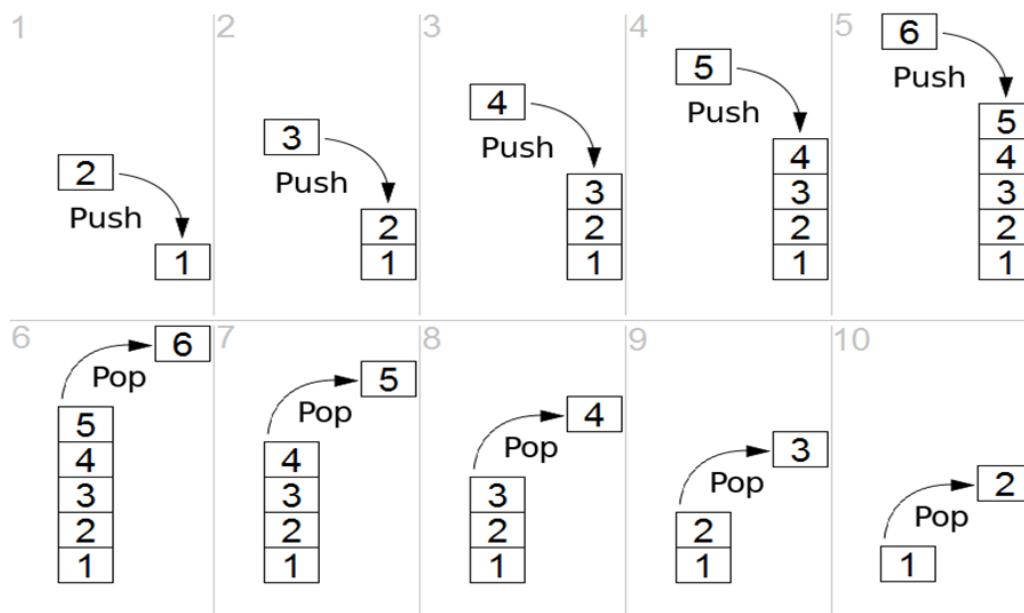


Figure 3.1.2: Push() and Pop() Operations

(ii) Array Representation of Stack

As we know a stack is a data structure designed to store collection of data where the data can be added and removed from only one end. We can implement this stack using a simple linear array. As array is a collection of similar kinds of elements. We can create a stack using a one dimensional array very easily.

For example, we can declare an array named `stack []` to store all the data elements of a stack. Normally, elements in a linear array can be accessed in any random way by using array name and its index.

But a stack is operating from only one end. Thus when a stack is implemented as an array, we should allow insertion and deletion of elements from only one end of the array.

Thus, a variable named “top” will keep track of the position of the topmost element in that stack. This variable is also called as stack pointer.

Initially the value of “top” is assigned to -1, as the stack is empty. When we push an element onto the stack, we need to increment the stack pointer by one and then insert the element at a position where stack pointer is pointing. For every push operation we have to check if stack pointer has reached the maximum size of the array stack [].

Similarly, when we perform a pop operation, the stack pointer should be decremented by 1. We should also check for a condition to see whether the stack array is empty or no.

Figure 3.1.3 demonstrates the array representation of stack

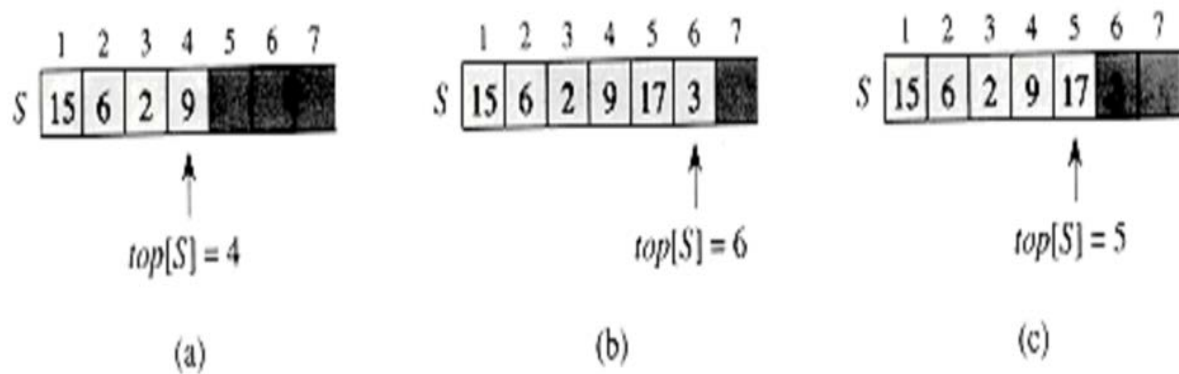


Figure 3.1.3: Array Representation of Stack

As shown in the above figure, an array named S with size 7 is declared which acts as a stack. Thus we can store total of 7 elements in this stack.

In part (a) of the figure, after adding elements 15, 6, 2, and 9, the stack pointer is pointing location 4.

In part (b), we have pushed two more elements 17 and 3 making stack pointer to have value 6.

In part (c), it shows how a pop operation is carried out on that array causing last element 3 to be popped out. Thus the stack pointer is decremented by 1 after pop operation.



Self-assessment Questions

- 1) A stack works on the principle of _____
 - a) First in first out (FIFO)
 - b) Last in last out (LILO)
 - c) First in last out (FILO)
 - d) Cyclical data structures

- 2) The difference between linear array and stack is that any elements can randomly be accessed.
 - a) True
 - b) False

- 3) Top[s] returns _____
 - a) Stack bottom
 - b) Stack top
 - c) Stack mid
 - d) Any random element

3.1.2 Operations on Stack

The operations discussed in the topics above are explained in detail in this section. Basically, stack operations may include, initializing a stack, using it storing data based on different applications and again de-initializing it. Apart from these basic things, stack is used for carrying out following two operations:

1. **Push()** – storing data item in to the stack
2. **Pop()** – Deleting a data item from the stack

Consider the operation of pushing a data on to the stack. In order to use stack most efficiently, we need to aware about the status of the stack. For this purpose, following functions are importing.

1. **stacktop()** – This function is used for displaying the topmost element in the stack
2. **isFull()** – This function is used to check if the stack is already full
3. **isEmpty()** – This function is used to check if stack is empty.

Throughout, we must maintain a pointer to the most recent pushed data on the stack. This pointer always represents the top of the stack and hence is named as top.

Before we proceed to implement push () operation, we must first learn the procedure for these support functions.

Algorithm for top() function

```
begin procedure stacktop
return stack[top]
end procedure
```

Implementation in C programming

```
int stacktop()
{
    return stack[top];
}
```

Algorithm for isFull() function

```
begin procedure isfull
iftop equals to MAXSIZE
returntrue
else
returnfalse
endif
end procedure
```

Implementation in C programming

```
bool isfull()
{
    if(top == MAXSIZE)
        returntrue;
    else
        returnfalse;
}
```

Algorithm for isempty() function

```
begin procedure isempty

iftop less than 1
returntrue
else
returnfalse
endif

end procedure
```

Implementation in C programming

```
bool isempty()
{
    if(top == -1)
        return true;
    else
        return false;
}
```

Now, to get back to push operation, we must first understand the process how push() function works. Following steps are involved:

- **Step 1:** Check if stack is full.
- **Step 2:** If stack is full then display an error and exit.
- **Step 3:** If stack is not full, increment top to point to next empty space.
- **Step 4:** Add the element on to the stack, where top is pointing.
- **Step 5:** Return

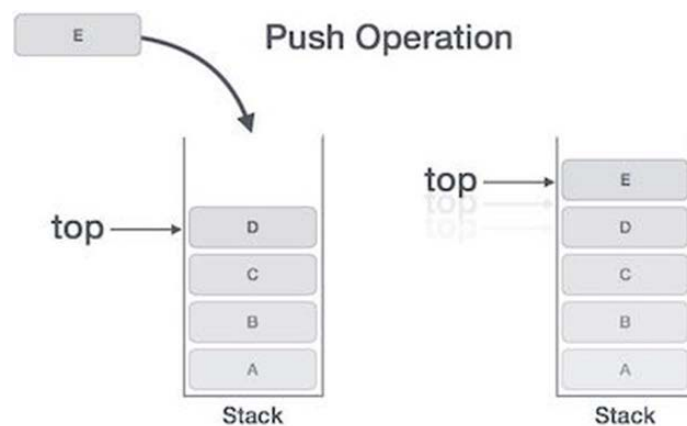


Figure 3.1.4: Push Operation on Stack

Note: If linked list is used for stack implementation, then memory space needs to be allocation in step 3.

Following is the algorithm for push operation

```
begin procedure push: stack, data
    if stack is full
```

```
return null
endif

top ← top + 1

stack[top] ← data

end procedure
```

And the corresponding C program function is also shown below

```
void push(int data)
{
    if(!isFull())
    {
        top = top + 1;
        stack[top] = data;
    } else
    {
        printf("Could not insert data, Stack is full.\n");
    }
}
```

Now, we move on to pop operation. Accessing the data element while removing it from the stack is called pop operation. Following are the steps involved in process of popping out an element from the stack.

- **Step 1** – Check if stack is empty.
- **Step 2** – If stack is empty, produce error and exit.
- **Step 3** – If stack is not empty, access the data element at which top is pointing.
- **Step 4** – Decrease the value of top by 1.
- **Step 5** – return

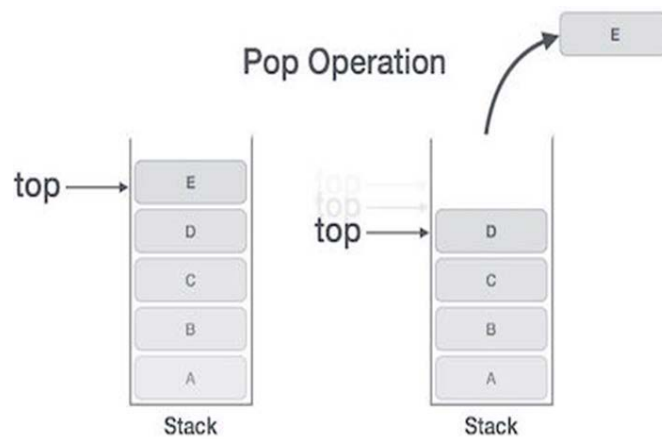


Figure 3.1.5: Pop Operation on Stack

Algorithm for implementation of pop operation

```
begin procedure pop: stack
if stack is empty
returnnull
endif
data ← stack[top]
top ← top -1
return data
end procedure
```

Corresponding C program function

```
int pop(int data)
{
    if(!isempty())
    {
        data = stack[top];
        top = top -1;
        return data;
    }else
    {
        printf("Could not retrieve data, Stack is empty.\n");
    }
}
```




Self-assessment Questions

4) Match the following

1	stacktop()	A	Used for checking if stack is empty
2	Istfull()	B	Used for displaying topmost element
3	Isempty()	C	Used for checking if stack is full

5) What push(x) does to stack

- a) Removes x from stack
- b) Add x to topmost element
- c) Add x to all the elements
- d) Add x to top of stack

6) What pop() does to stack

- a) Removes x from stack
- b) Add x to topmost element
- c) Add x to all the elements
- d) Add x to top of stack

3.1.3 Polish Notations

First we need to understand Arithmetic Expressions. An arithmetic expression is an expression which when evaluated, results in a numeric value. The method of writing arithmetic expression is known as a Notation. Same Arithmetic Expression can be written in different ways without changing the essence or meaning of that expression.

Consider an expression:

$$(5-6)*7$$

It can be written in its *infix form* as “*(-5 6)7”. In this case all the arithmetic operators are binary in nature, thus bracketing is not necessary. The above expression can be also written as

$$* - 5 6 7$$

Consider an expression “1+2”, which adds the values 1 and 2. Its prefix notation, the operators precedes the operands, thus it will be “+ 1 2”.

The product calculation depends upon the availability of two operands *i.e.*, 5-6 and 7.

Normally, the innermost expressions are evaluated first. But, in case of prefix notation operators are written ahead of operands.

Thus infix notation with parenthesis will look like

$$5 - (6 * 7)$$

Or without parenthesis it will be

$$5 - 6 * 7$$

It would change the semantics or meaning of the expression because of precedence rule.

Similarly, **Polish notation** of

$$5 - (6 * 7)$$

Will be

$$- 5 * 6 7$$

Polish notation

Polish notation, also called Polish prefix notation or prefix notation is a symbolic logic invented by Polish mathematician Jan Lukasiewicz. It is a form of notation for logic, arithmetic, and algebra. In prefix notations, the operators are placed to the left of their operands. If the operator's parity is fixed, the result is a syntax lacking parentheses or other brackets that can still be parsed without any problem. The term Polish notations also include Polish postfix notation, or Reverse Polish notation, in which the operators are placed after the operands.

(i) Infix Notation

As already discussed in the previous section, infix notation is the most common and simplest notation in which an operator is placed between two operands. This notation is also known as general form of arithmetic expression. **For example**, if arithmetic expression for adding two operands can be written in infix form as

$$A + B$$

In this **example** A and B are two operands and + is the operator.

Another **example** of infix expression is

$$A + B * C + (E - G)$$

These expressions follow a normal arithmetic precedence rule. **For example**, to evaluate the above expression, the first precedence is given to multiplication. So the product of B and C will be calculated first. Second precedence will be given to parenthesis. Therefore the result of E – G will be calculated and then A, result of product of B and C, and subtraction result of E and G will be added together.

(ii) Prefix Notation

This is also called as a polish method. When using this method, operator precedes operands i.e. instruction precedes data. Here the order of operations and operands determines the result, making parenthesis unnecessary. Taking the **example**, consider infix expression 3 (4 + 5). This could be expressed as

$$* 3 + 4 5$$

This is in contrast with the traditional algebraic methodology for performing mathematical operations, order of operation. In the expression 3(4+5), we first work inside the parentheses to add four plus five and then multiply the result by three.



Did you know?

In the olden days of the calculator, the end-user would write down the results of every step when using the algebraic Order of Operations. Not only did this slow things down, it provided an opportunity for the end-user to make errors and sometimes defeated the purpose of using a calculating machine. In the 1960's, engineers at Hewlett-Packard decided that it would be easier for end-users to learn Jan Lukasiewicz' logic system than to try and use the Order of Operations on a calculator. They modified Jan Lukasiewicz's system for a calculator keyboard by placing the instructions (operators) after the data. In homage to Jan Lukasiewicz' Polish logic system, the engineers at Hewlett-Packard called their modification reverse Polish notation (RPN).

(iii) Postfix Notation

Just opposite to prefix notation is postfix notation. Here operands precedes operator or operator is placed after operands and hence it is called postfix notation. It is also called as reverse polish expression. The infix expression $A+B$ can be written in postfix as $AB+$.

Below are some of the *examples* of expressions represented in all three notations.

Infix	Prefix	Postfix
$A+B$	$+AB$	$AB+$
$A+B*C$	$+A*BC$	$ABC*+$
$(A+B)*(C-D)$	$*+AB-CD$	$AB+CD-*$

Algorithm for evaluation of postfix expression

Consider a string of postfix arithmetic expression of operands and operators. Below given below should be followed for evaluation of a postfix expression:

- **Step 1:** Scan the string from left to right.
- Skip all the operands and values.
- If an operator is found, perform the operation on preceding two operands.
- Now replace these (2 operands and an operator) with one operand *i.e.*, the result of operation.
- Continue the process until single value remains, which is the result of the expression.

Algorithm

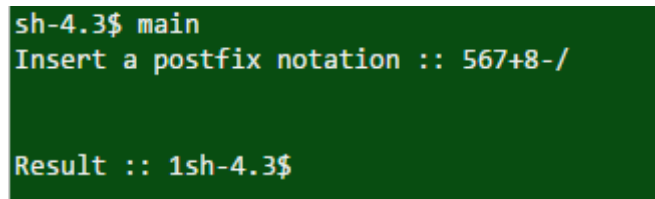
1. Add the right parentheses")" at the end of P.
 2. Scan P from left to right and repeat step 3 and 4 for each element of P until the sentinel ")" is encountered.
 3. If an operand is encountered, put it on STACK.
 4. If an operator is encountered, then:
 - a. Remove the two top elements from the STACK
 - b. Evaluates these two operators using that operator.
 - c. Place the result of (b) back on STACK.
- [End of If structure]
- [End of step 2 loop]
5. Set VALUE equal to the top element of STACK.
 6. EXIT.

Program:

```
#include<string.h>
#include<stdlib.h>
#define MAX 50
int stack[MAX];
char post[MAX];
int top=-1;
void pushstack(int tmp);
void calculator(char c);
void main()
{
    int i;
    printf("Insert a postfix notation :: ");
    gets(post);
    for(i=0;i<strlen(post);i++)
    {
        if(post[i]>='0' && post[i]<='9')
        {
            pushstack(i);
        }
        if(post[i]=='+' || post[i]=='-' || post[i]=='*' ||
        post[i]=='/' || post[i]=='^')
        {
            calculator(post[i]);
        }
    }
    printf("\n\nResult :: %d",stack[top]);
}
void pushstack(int tmp)
{
    top++;
```

```
    stack[top]=(int)(post[tmp]-48);
}
void calculator(char c)
{
    int a,b,ans;
    a=stack[top];
    stack[top]='\0';
    top--;
    b=stack[top];
    stack[top]='\0';
    top--;
    switch(c)
    {
        case '+':
            ans=b+a;
            break;
        case '-':
            ans=b-a;
            break;
        case '*':
            ans=b*a;
            break;
        case '/':
            ans=b/a;
            break;
        case '^':
            ans=b^a;
            break;
        default:
            ans=0;
    }
    top++;
    stack[top]=ans;
}
```

Output:



```
sh-4.3$ main
Insert a postfix notation :: 567+8-/

Result :: 1sh-4.3$
```



- ### 3.1.4 Conversion of Arithmetic Expression from Infix to Postfix

The order of precedence is

- Below given algorithm transforms any infix expression X into its equivalent postfix expression Y. We use stack data structure to store the operators and parenthesis.

1. Read token from Left to Right in a given infix expression X and Postfix expression Y is generated.

-
2. Input infix Expression may have following tokens:
 - a) Any Alphabet from A-Z or a-Z
 - b) Any Number from 0-9
 - c) Any Operator
 - d) Opening And Closing Braces (,)
 3. If token read is Alphabet:
 - a) Print that Alphabet as Output
 4. If token read is Digit:
 - a) Print that Digit as Output
 5. If token read is Opening Bracket “(” :
 - a) Push opening bracket ‘(’ Onto the Stack
 - b) If any Operator appears before ‘)’ then Push it onto Stack.
 - c) If Corresponding ‘)’ bracket appears then Start pop elements from Stack till ‘(’ is popped out.
 6. If token read is Operator:
 - a) Check if there is any Operator already present in Stack.
 - b) If Stack is empty, Push Operator onto the Stack.
 - c) If operator is present, check if Priority of Incoming Operator is greater than Priority of Topmost Stack Operator.
 - d) If Priority of incoming Operator is Greater, push Incoming Operator Onto Stack.
 - e) Else Pop Operator from Stack, repeat Step 6.
-

Example of converting an expression from infix to postfix

Infix expression: $A * B + C$

The order in which the operators appear is not reversed. When the '+' is read, it has lower precedence than the '*', so the '*' must be printed first.

We will show this in a table with three columns. The first will show the symbol currently being read. The second will show what is on the stack and the third will show the current contents of the postfix string. The stack will be written from left to right with the 'bottom' of the stack to the left.

Step	Current Symbol	Stack	Postfix expression
1.	A		A
2.	*	*	A
3.	B	*	AB
4.	+	+	AB*
5.	C	+	AB*C
6.			AB*C+

Step 1: The first input token is an alphabet "A". Thus it is printed as output character of postfix notation.

Step 2: Next token in the infix expression is an operator "*". Thus it is pushed onto the top of the stack if the stack is empty.

Step 3: The third token in infix expression is an alphabet "B", hence it is printed as an output character of postfix notation.

Step 4: Fourth input token is again an operator "+". But the operator on the top of the stack i.e. "*" has higher precedence as compared to operator "+". Thus the operator "*" is popped out from top of stack and printed as output of postfix notation. Push the operator "+" on to the top of the stack now.

Step 5: Next input character is an alphabet "C". Thus, printed as an output character of postfix notation.

Step 6: now it is the end of the infix expression, thus we need to pop out all the operators from the stack one by one and printed as postfix notation. Thus operator "+" is printed as the last character of the postfix notation.

Thus the Postfix expression is AB*C+

Program:

```
#include<stdio.h>
#include<ctype.h>
char stack[20];
int top = -1;
void push(char x)
{
    stack[++top] = x;
}

char pop()
{
    if(top == -1)
        return -1;
    else
        return stack[top--];
}

int priority(char x)
{
    if(x == '(')
        return 0;
    if(x == '+' || x == '-')
        return 1;
    if(x == '*' || x == '/')
        return 2;
}

int main()
{
    char exp[20];
    char *e, x;
    printf("Enter the expression :: ");
    scanf("%s",exp);
    e = exp;
    while(*e != '\0')
    {
        if(isalnum(*e))
            printf("%c",*e);
        else if(*e == '(')
            push(*e);
        else if(*e == ')')
        {
            while((x = pop()) != '(')
                printf("%c", x);
        }
        else
        {

```

```

        while(priority(stack[top]) >= priority(*e))
            printf("%c",pop());
        push(*e);
    }
    e++;
}
while(top != -1)
{
    printf("%c",pop());
}
return 0;
}

```

Output:

```

sh-4.3$ gcc -o main *.c
sh-4.3$ main
Enter the expression :: a+b*c
abc*+sh-4.3$

```



Self-assessment Questions

- 10) As per the algorithm to convert infix to postfix expression, we must ignore parenthesis present in infix expression
 - a) True
 - b) False
- 11) While converting an infix expression to postfix expression, if an operator is encountered, the operators are _____.
 - a) Pushed on to the stack
 - b) Popped out of stack
 - c) Left without doing anything
 - d) Checked for precedence level
- 12) When the string scanning ends, next operation is _____.
 - a) Popping out all operators from stack and adding them to postfix string
 - b) Exit and print result
 - c) Push all the operands on to the stack
 - d) Do nothing

3.1.5 Applications of Stack

Stacks have many useful applications in computer science. Stack form a base for many of the compilers for programming languages and sometimes is also core part of low lever programming languages like MATLAB and other assembly level languages. Some of the basic and most frequently used applications are described in the section below.

(i) Balancing Symbol

We always do syntax mistakes while typing programs. The compilers duty is to check the programs for all the syntax errors. Most of the times, we make mistakes in typing brackets or parenthesis or any operators. Lack of any one symbol may cause multiple errors in the program. Thus real error remains unidentified.

Hence a stack can be used to check if the expressions in the programs are balanced. Thus, every right bracket, parenthesis or braces must end with corresponding left counterparts.

For example, the sequence `[]` is correct, however `[()]` is invalid. As of now, consider a problem just check for balancing of parentheses, brackets, and braces and ignore other characters. A Stack can be used to balance symbols in a program. Following are the steps to do the same:

1. Create an empty stack `s[]`.
2. Scan the program file character by character till the end of file.
3. Upon identifying any symbol (parenthesis, brace, bracket etc.), push it on to the stack.
4. If stack is empty and scanned character is close bracket, brace, parenthesis etc., print an error message.
5. Else pop element from stack
6. If popped element is not corresponding open symbol, print an error message.
7. If stack is not empty at the end of file, print an error message.

This is clearly linear and actually makes only one pass through the input. It is thus on-line and quite fast.

(ii) Recursion

Recursion is considered to be the most powerful tools in a programming language. But sometimes Recursion is also considered as the most tricky and threatening concept to a lot of programmers. This is because of the uncertainty of conditions specified by user.

In short Something Referring to itself is called as a Recursive Definition

Recursion can be defined as defining anything in terms of itself. It can be also defined as repeating items in a self-similar way.

In programming, if one function calls itself to accomplish some task then it is said to be a recursive function. Recursion concept is used in solving those problems where iterative multiple executions are involved.

Thus, to make any function execute repeatedly until we obtain the desired output, we can make use of Recursion.

Example of Recursion:

The best example in mathematics is the factorial function.

$$n! = 1.2.3.....(n-1).n$$

If $n=6$, then factorial of 6 is calculated as

$$6! = 6(5)(4)(3)(2)(1) = 720$$

Consider we are calculating the factorial of any given using a simple. If we have to calculate factorial of 6 then what remains is the calculation of 5!

In general we can say

$$n! = n(n-1)! \quad (i.e., 6! = 6(5!))$$

It means we need to execute same factorial code again and again which is nothing but Recursion.

Thus the Recursive definition for factorial is:

$$f(n) = \begin{cases} 1 & \text{if } n=0 \\ n * f(n-1) & \text{otherwise} \end{cases}$$

The above Recursive function says that the factorial of any number $n=0$ is 1, else the factorial of any other number n is defined to be the product of that number n and the factorial of one less than that number .

Any recursive definitions will have some properties. They are:

1. There are one or more base cases for which recursions are not needed.
2. All cycles of recursion stops at one of the base cases.

We should make sure that each recursion always occurs on a smaller version of the original problem.

In C Programming a recursive factorial function will look like:

```
int factorial(int n)
{
    if (n==0)                //Base Case
        return 1;
    else
        return n*factorial (n-1);    //Recursive Case
}
```

The above program is for calculating factorial of any number n . First when we call this factorial function, it checks for the base case. It checks if value of n equals 0. If n equals 0, then by definition it returns 1.

Otherwise it means that the base case is not yet been satisfied. Hence it returns the product of n and factorial of $n-1$.

Thus it calls the factorial function once again to find factorial of $n-1$. Thus forming recursive calls until base case is met.

(iii) Evaluation of Postfix Expression

```
1. Push "(" onto STACK, and add " )" to the end of Q.
2. Scan Q from left to right and repeat step 3 to 6 for each element of Q until the
   STACK is empty.
3. If an operand is encountered, add it to P.
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 to P each operator (on the top of
      STACK) which has the same precedence as or higher precedence than that
      operator.
   b. Add that operator to STACK.
   [End of if structure]
6. If a right parenthesis is encountered, then:
   a. Repeatedly pop from the STACK and add to P each operator until a left
      parenthesis is encountered.
   b. Remove the left parenthesis. [Do not add the left parenthesis to P].
   [End of if structure]
   [End of step 2 loop].
7. EXIT.
```

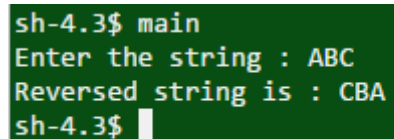
(iv) String Reversal

Since stack is LIFO data structure, it becomes of obvious use in application where there is requirement of reversing a string or for checking if a string is a palindrome or not. The simplest way to reverse a string is, scan a string from left to right and push every character on to the stack until we reach the end of the string. Once we reach the end of the string, start popping out elements from the stack and create a new string of popped elements. Repeat the process of popping from stack until stack becomes empty.

```
/*Program of reversing a string using stack */
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#define MAX 20
int top = -1;
char stack[MAX];
char pop();
void push(char);
int main()
{
    char str[20];
    unsigned int i;
    printf("Enter the string : ");
    gets(str);
    /*Push characters of the string str on the stack */
    for(i=0;i<strlen(str);i++)
        push(str[i]);
```

```
    /*Pop characters from the stack and store in string str */
    for(i=0;i<strlen(str);i++)
        str[i]=pop();
    printf("Reversed string is : ");
    puts(str);
    return 0;
}/*End of main()*/
void push(char item)
{
    if(top == (MAX-1))
    {
        printf("Stack Overflow\n");
        return;
    }
    stack[++top] =item;
}/*End of push()*/
char pop()
{
    if(top == -1)
    {
        printf("Stack Underflow\n");
        exit(1);
    }
    return stack[top--];
}/*End of pop()*/
```

Output:



```
sh-4.3$ main
Enter the string : ABC
Reversed string is : CBA
sh-4.3$
```




- 165



Summary

- Stacks are Last-In-First-Out (LIFO) data structures in which the most recent element inserted in the stack is the first one to be removed.
- The stack can be implemented using an array by creating a stack pointer variable for keeping track of top position.
- Push () and pop () are the primary operations possible on stacks for insertion and deletion of elements along with some support functions.
- Polish notation which is also called as prefix notation or simply prefix notation is a form of notation for logic, arithmetic, and algebra.
- Infix notation is the most common and simplest notation in which an operator is placed between two operands.
- In prefix notation, operator precedes operands and in postfix operands precedes the operator.
- Stacks can be used for evaluating a postfix expression, also in Recursion, String reversal etc.



Terminal Questions

1. Explain stack and its basic operations.
2. Explain the algorithm for push and pop operations.
3. Write a C program for converting infix expression to postfix expression.
4. Explain applications of stack in brief.



Answer Keys

Self-assessment Questions	
Question No.	Answer
1	c
2	b
3	b
4	1 -b,2-c,3-a
5	b
6	a
7	Operator
8	a
9	c
10	a
11	a
12	a
13	a
14	b
15	c



Activity

Activity Type: Offline

Duration: 15 Minutes

Description:

Divide the students into 4 groups.

Below are 4 infix expression, assign an expression to each group.

Each group should convert the given expression to postfix and prefix expression using stack.

- a) $3+4*5/6$
- b) $6 * (77 + 8 * 15) + 20$
- c) $(300+23)*(43-21)/(84+7)$
- d) $(4+8)*(6-5)/((3-2)*(2+2))$

Case Study

Stack based memory allocation

Stacks in computing architectures are regions of memory where data is added or removed in a last-in-first-out (LIFO) manner.

In most modern computer systems, each thread has a reserved region of memory referred to as its stack. When a function executes, it may add some of its state data to the top of the stack; when the function exits it is responsible for removing that data from the stack. At a minimum, a thread's stack is used to store the location of function calls in order to allow return statements to return to the correct location, but programmers may further choose to explicitly use the stack. If a region of memory lies on the thread's stack, that memory is said to have been allocated on the stack.

Because the data is added and removed in a last-in-first-out manner, stack-based memory allocation is very simple and typically faster than heap-based memory allocation (also known as dynamic memory allocation). Another feature is that memory on the stack is automatically, and very efficiently, reclaimed when the function exits, which can be convenient for the programmer if the data is no longer required. If however, the data needs to be kept in some form, then it must be copied from the stack before the function exits. Therefore, stack based allocation is suitable for temporary data or data which is no longer required after the creating function exits.

A thread's assigned stack size can be as small as only a few bytes on some small CPU's. Allocating more memory on the stack than is available can result in a crash due to stack overflow.

Some processor families, such as the x86, have special instructions for manipulating the stack of the currently executing thread. Other processor families, including PowerPC and MIPS, do not have explicit stack support, but instead rely on convention and delegate stack management to the operating system's application binary interface (ABI).

Questions:

1. Explain how stack based memory allocation worked.
2. What are the advantages of stack based memory allocation?

Bibliography



e-Reference

- bowdoin.edu, (2016). Computer Science 210: Data Structures. Retrieved on 19 April 2016, from <http://www.bowdoin.edu/~ltoma/teaching/cs210/fall10/Slides/StacksAndQueues.pdf>



External Resources

- Kruse, R. (2006). *Data Structures and program designing using 'C'* (2nd ed.). Pearson Education.
- Srivastava, S. K., & Srivastava, D. (2004). *Data Structures Through C in Depth* (2nd ed.). BPB Publications.
- Weiss, M. A. (2001). *Data Structures and Algorithm Analysis in C* (2nd ed.). Pearson Education.



Video Links

Topic	Link
Introduction and definition of stacks	https://www.youtube.com/watch?v=FNZ5o9S9prU
Recursion	https://www.youtube.com/watch?v=k0bb7UYy0pY
Evaluation of postfix expression using stack	https://www.youtube.com/watch?v=_EP4gpG-4kQ



Notes:

