Stack

Presentation topic

Date: 14th Ashad, Monday

SN.	Presentation Topic	Roll No.
1.	**Asymptotic Notations**	**101-109**
2.	**Linear Data Structure**	**110-115**
3.	**Dynamic Memory Allocation**	**116-121**
4.	**Pointer and its Application**	**122-127**
5.	**Array Vs Structure**	**128-132**
6.	**Stack**	**133-136**

Stack

- ***A stack is an ordered collection of items into which new items may be inserted and***
- ***from which items may be deleted at one end, called the top of the stack***

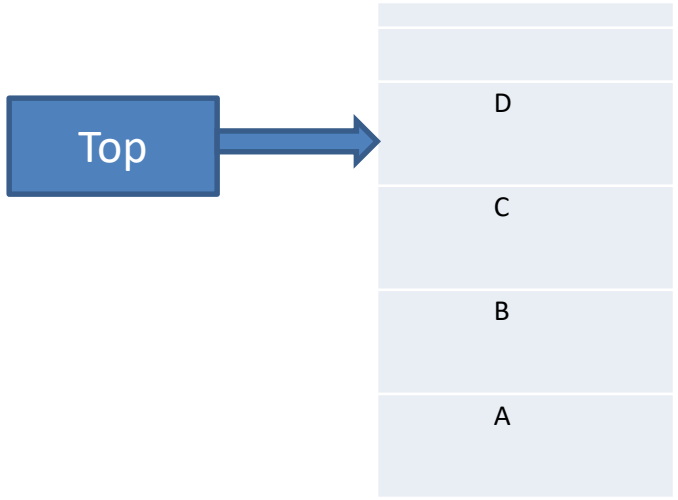
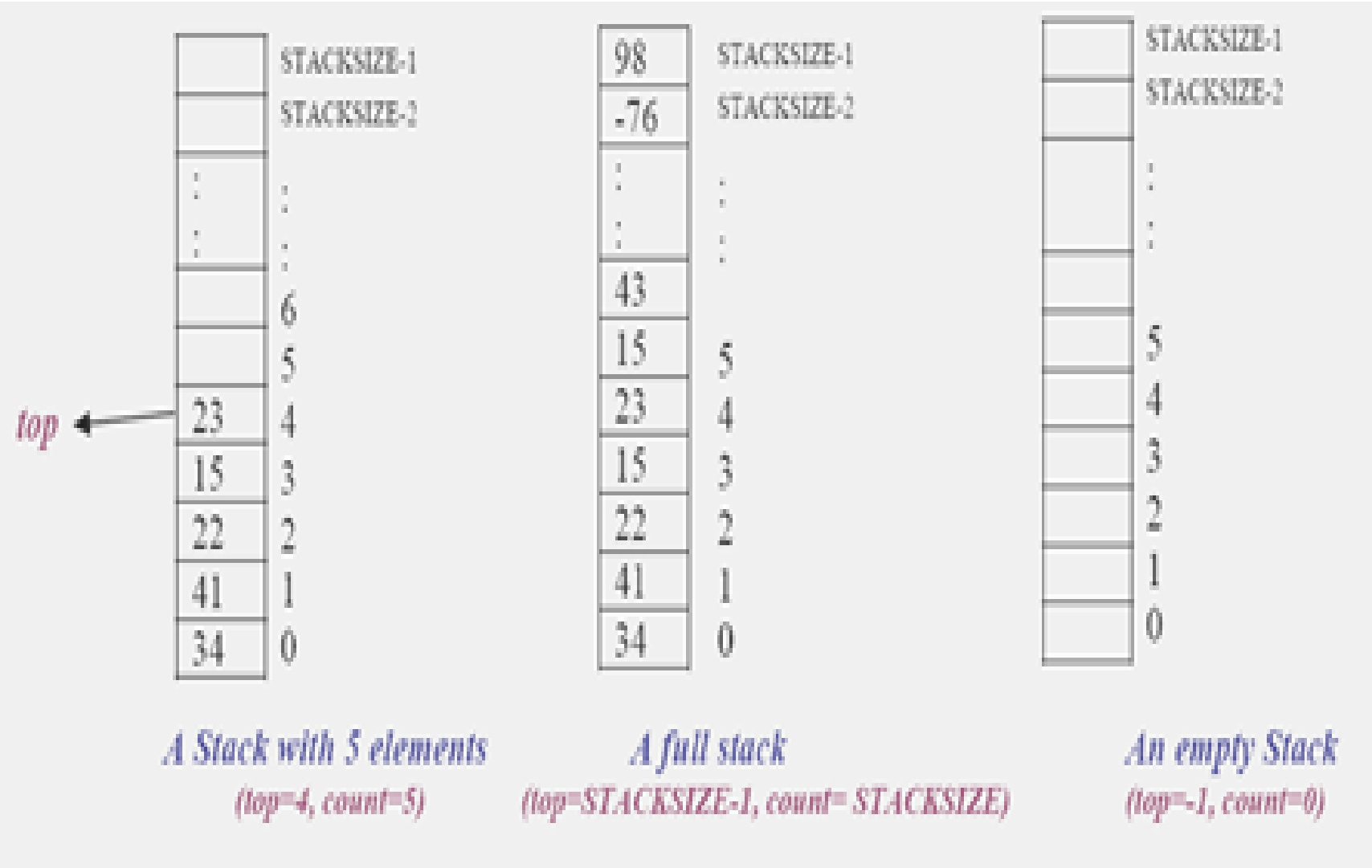
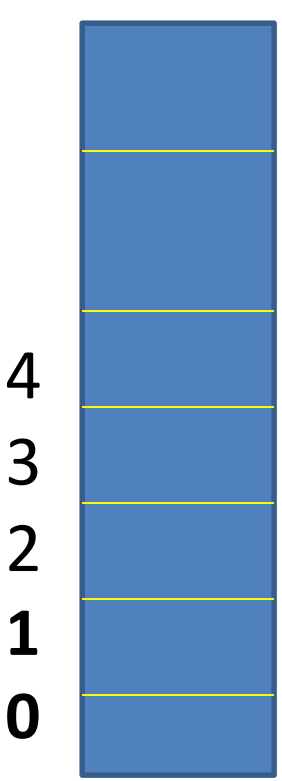


fig. stack containing data items

Cont…

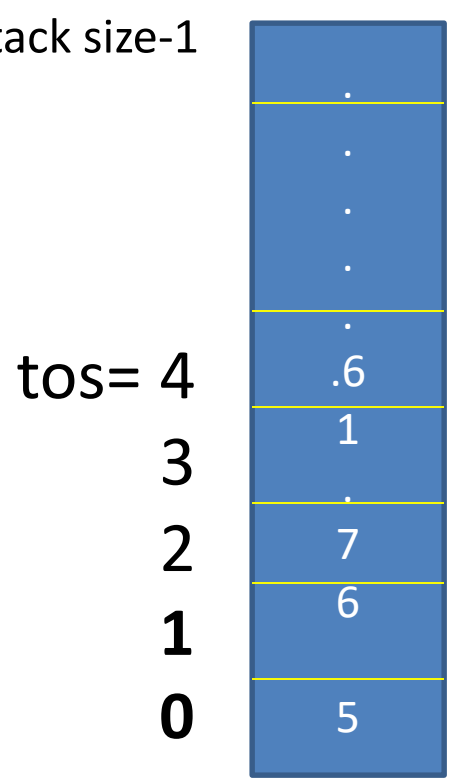

STACKSIZE-1
STACKSIZE-2
top
23
15
22
41
34
6
5
4
3
2
1
0
A Stack with 5 elements
(top=4, count=5)
98
-76
43
15
23
15
22
41
34
A full stack
(top=STACKSIZE-1, count= STACKSIZE)
An empty Stack
(top=-1, count=0)



tos = -1, count = 0

Stack is empty

stack size-1

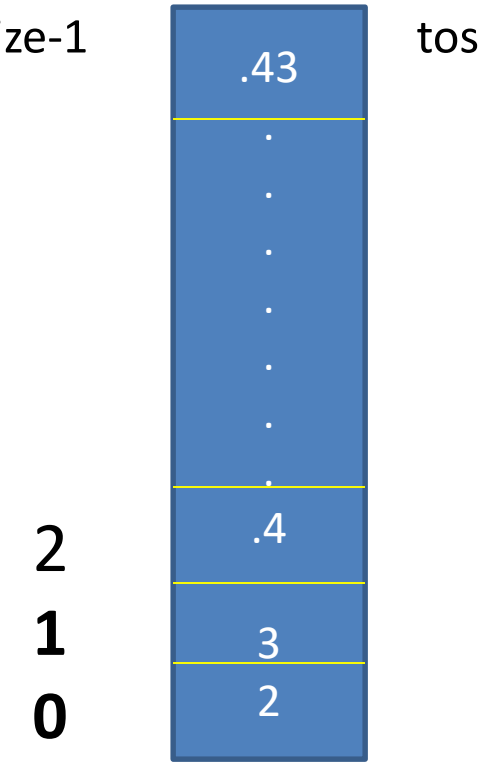


tos= 4 count =5

Stack with 5 elelments

stack size-1

tos



tos=size-1 , cout=stack size

Full of stack

Applications of Stack:

Stack is used directly and indirectly in the following fields:

- To *evaluate the expressions (postfix, prefix*)
- To keep the p*age-visited history in a Web browser*
- To perform the u*ndo sequence in a text editor*
- Used in *recursion*
- To pass the parameters between the f*unctions in a C program*
- Can be used as an a*uxiliary data structure for implementing algorithms*
- Can be used as a *component of other data structures*

Stack Operations

- ***PUSH*** operation
- ***POP*** operation

The **PUSH** and the **POP** operations are the ***basic or primitive*** operations on a stack. Some others operations are:

✓ ***CreateEmptyStack*** operation:

This operation is used to create an empty stack.

✓ ***IsFull*** operation:

The isfull operation is used to check whether the stack is full or not (i.e. stack overflow)

✓ ***IsEmpty*** operation:

The isempty operation is used to check whether the stack is empty or not. (i. e. stack underflow)

✓ ***Top operations:***

This operation returns the current item at the top of the stack, it doesn't remove it

The Stack ADT

- A **stack** of elements of type ***T*** is a finite sequence of elements of ***T*** together with the operations
- **CreateEmptyStack(S):**
  - Create or make stack ***S*** be an empty stack
- **Push(S, x)**
  - Insert ***x*** at one end of the stack, called its **top**
- **Top(S)**
  - If stack ***S*** is not empty; then retrieve the element at its **top**
- **Pop(S)**
  - If stack ***S*** is not empty; then delete the element at its **top**
- **IsFull(S)**
  - Determine if ***S*** is full or not. Return **true** if ***S*** is full stack; return **false** otherwise
- **IsEmpty(S)**
  - Determine if ***S*** is empty or not. Return **true** if ***S*** is an empty stack; return **false** otherwise.

Implementation of Stack

Stack can be implemented in two ways:

- 1. Array Implementation of stack
  (or static implementation)
- 2. Linked list implementation of stack
  (or dynamic)

Array Implementation of stack

- In this implementation top is an integer value (an index of an array) that indicates the top position of a stack.
- Each time data is added or removed, top is incremented or decremented accordingly, to keep track of current top of the stack.
- By convention, in C implementation the empty stack is indicated by setting the value of top to -1 (top=-1).

Eg.

```
#define MAX 10
sruct stack
{
        int items[MAX];     //Declaring an array to store items
        int top;                    //Top of a stack
};
 typedef struct stack st;  // struct stack st;
```

Creating Empty stack

- The value of top=-1
- indicates the empty stack in C implementation. */*Function to create an empty stack*/*

```
void create_empty_stack(st *s)   //(struct stack *s)
{
        s->top=-1;
}
```

- **Stack Empty or Underflow:**

?

```
int isempty(st *s)
{
        if(s->top==-1) return 1;
        else
        return 0;
}
```

- **Stack Full or Overflow:**

?

Algorithm for PUSH and POP operations on Stack

Algorithm for PUSH (inserting an item into the stack) operation:

This algorithm adds or inserts an item at the top of the stack

1. [Check for stack overflow?]

if top=MAXSIZE-1 then

print "Stack Overflow" and Exit

else

Set top=top+1 [Increase top by 1]

Set Stack[top]:= item [Inserts item in new top position]

2. Exit

Algorithm to pop data item form stack

The PUSH and POP functions

Push

```
void push()
{
    int item;
    if(top == MAXSIZE - 1)          //Checking stack overflow
        printf("\n The Stack Is Full");
    else
    {
        printf("Enter the element to be inserted");
        scanf("%d",  &item);            //reading an item
        top= top+1;                     //increase top by 1
        stack[top] = item;              //storing the item at the top of the stack
    }
}
```

Push

```
void push(st *s, int element)
{
    if(isfull(s))  /* Checking Overflow condition */
    printf("\n \n The stack is overflow: Stack Full!!\n");
    else
    /* First increase top by 1 and store element at top position*/

    s->items[++(s->top)]=element;
}
```

POP function

Stack implementation in array

//implementation of stack using array

```
#include<process.h>
#include<conio.h>
#include<stdio.h>
#define max 10
void push();
void pop();
void display();
struct stack
{
 int tos;
 int item[max];
};
struct stack s;
```

```
void main()
{
 int choice;
 s.tos = -1;      //stack is empty
 printf("\n 1. push");
 printf("\n 2. pop");
 printf("\n 3. display");
 printf("\n 4. exit");
 do
 {
  printf("\n enter your choice:");
  scanf("%d", &choice);

  switch(choice)
  {
   case 1:
   {
    push();
    break;
   }
   case 2:
   {
    pop();
    break;
   }

   case 3:
   {
    display();
    break;
   }
   case 4:
   exit(1);
  }
 }while(choice!=5);
getch();
}
```

```
//push operation
void push()
{
 int data;
if(s.tos == max-1)  //check stack is full or not
 {
  printf("\n sorry stack is full");
 }
else
  printf("\n enter a value:");
  scanf("%d", &data);
  s.tos=s.tos+1;
  s.item[s.tos]=data;
}
//pop operation
void pop()
{
 //int data;

if(s.tos== -1)
  printf("\n stack is empty");
else
  printf("\n the popped value is :%d", s.item[s.tos]);
  s.tos--;
}

//display data item in stack
void display()
{
 int i;
if(s.tos == -1) // to check stack is empty or not
  printf("\n sorry stack is an empty");
else
  printf("\n the data item is :");
  for(i=s.tos; i>=0; i--)
  {
   printf("%d\t", s.item[i]);
  }
}
```

Using Pointer

```
//array implementation stack
using pointer
#include<process.h>
#include<conio.h>
#include<stdio.h>
#include<stdlib.h>
#define max 10
void push();
void pop();
void display();
struct stack
{
 int tos;
 int item[max];
};
struct stack *s;

void main()
{
 int choice;
 s=(struct
stack*)malloc(sizeof(struct
stack));   //memory allocation
 s->tos = -1;      //stack is empty
 printf("\n 1. push");
 printf("\n 2. pop");
 printf("\n 3. display");
 printf("\n 4. exit");
```

```
do
{
 printf("\n enter your choice:");
 scanf("%d", &choice);

 switch(choice)
 {
  case 1:
  {
   push();
   break;
  }

  case 2:
  {
   pop();
   break;
  }

  case 3:
  {
   display();
   break;
  }
  case 4:
  exit(1);
  }
 }while(choice!=5);
getch();
}
```

```
//push operation
void push()
{
 int data;
 //check stack is full or not
if(s->tos == max-1)
 {
  printf("\n sorry stack is full");
 }
else
  printf("\n enter a value:");
  scanf("%d", &data);
  s->tos=s->tos+1;
  s->item[s->tos]=data;
}

//pop operation
void pop()
{
 //int data;

if(s->tos== -1)
  printf("\n stack is empty");
else
  printf("\n the popped value is :%d", s->item[s-
>tos]);
  s->tos--;
}

//display data item in stack
void display()
{
 int i;
if(s->tos == -1) // to check empty or not
  printf("\n sorry stack is an empty");
else
  printf("\n the data item is :");
  for(i=s->tos; i>=0; i--)
  {
   printf("%d\t", s->item[i]);
  }
}
```

Infix, Prefix and Postfix Notation

- One of the applications of the stack is to evaluate the expression.
- We can represent the expression following three types of notation:

➢ Infix

➢ Prefix

➢ Postfix

Infix expression

It is an ordinary mathematical notation of expression where operator is written in *between the operands*.

Example: A+B.

Here '***+***' is an ***operator*** and ***A*** and ***B*** are called ***operands***

Prefix notation

In prefix notation the operator precedes the two operands.

That is the *operator is written before the operands*.

It is also called **polish notation**.

Example: +AB

Postfix notation

In postfix notation the operators are written after the operands so it is called the postfix notation (post mean after).

In this notation the operator follows the two operands.

Example: AB+

Examples.

- A + B (Infix)
- + AB (Prefix)
- AB + (Postfix)

Cont..

- Both prefix and postfix are parenthesis free expressions.

Eg.

- (A + B) * C Infix form
- *+ A B C Prefix form
- A B + C * Postfix form

Converting an Infix Expression to Postfix

- First convert the sub-expression to postfix that is to be evaluated first and repeat this process

Example

- A + (B * C) *parenthesis for emphasis*
- A + (BC*) *convert the multiplication*
- A (BC*) + *convert the addition*
- ABC*+ *postfix form*

Converting an Infix Expression to Postfix

- **First convert the sub-expression to postfix that is to be evaluated first and repeat this process**.
- You substitute intermediate postfix sub-expression by any variable whenever necessary that makes it easy to convert.
- Remember, to convert an infix expression to its postfix equivalent,
- we first convert the innermost parenthesis to postfix, resulting as a new operand
- In this fashion parenthesis can be successively eliminated until the entire expression is converted
- The last pair of parenthesis to be opened within a group of parenthesis encloses the first expression within the group to be transformed
- This last in, first-out behavior suggests the use of a stack

Precedence rule

While converting infix to postfix you have to consider the **precedence rule**

1. Exponentiation (the expression A$B is A raised to the B power, so that 3$2=9)
2. Multiplication/Division
3. Addition/Subtraction

- When un-parenthesized operators of the same precedence are scanned, the order is assumed to be left to right except in the case of exponentiation, where the order is assumed to be from right to left.

Eg.

- A+B+C means (A+B)+C
- ABC means A$(B$C)

- By using parenthesis we can override the default precedence.
- Consider an example that illustrate the converting of infix to postfix expression,
- A + (B* C).
- Use the following **rule** to convert it in postfix:

1. Parenthesis for emphasis
2. Convert the multiplication
3. Convert the addition
4. Post-fix form

Illustration

A + (B * C)	Infix form
A + (B * C)	Parenthesis for emphasis
A + (BC*)	Convert the multiplication
A (BC*) +	Convert the addition
ABC*+	Post-fix form

Examples

- (A + B) * ((C - D) + E) / F Infix form
- (AB+) * ((C – D) + E) / F
- (AB+) * ((CD-) + E) / F
- (AB+) * (CD-E+) / F
- (AB+CD-E+*) / F
- AB+CD-E+*F/ Postfix form

Home work

Infix	*Postfix*
A + B	AB +
A + B – C	AB + C -
(A + B) * (C – D)	AB + CD - *
A $ B * C – D + E / F / (G + H)	AB $ C * D – EF / GH + / +
((A + B) * C – (D - E)) $ (F + G)	AB + C * DE - - FG + $
A – B / (C * D $ E)	ABCDE $ * / -

Algorithm to convert infix to postfix notation

Let here two stacks opstack and poststack are used and otos & ptos represents the opstack top and poststack top respectively.

otos -> opstack top

Ptos -> poststack top

1.Scan one character at a time of an infix expression from left to right
2.opstack = the empty stack
3.Repeat till there is data in infix expression

3.1 if scanned character is ‘(‘ then push it to opstack

3.2 if scanned character is operand then push it to poststack

3.3 if scanned character is operator then

if(opstack!=-1)

while(precedence (opstack[otos])>precedence(scan character)) then pop and push it into poststack

otherwise

push into opstack

3.4 if scanned character is ‘)’ then

pop and push into poststack until ‘(‘ is not found and ignore both symbols

4.pop and push into poststack until opstack is not empty.
5.Return

Opstack =operator stack

Eg. converting infix to post fix

A+B-C

Step	scan line/ip symbol	Poststack	opstack
1	A	A	-----
2	+	A	+
3	B	AB	+
4	-	AB+	-
5	C	AB+C	-
6	---	AB+C-	

A+B*C/D-E *Convert into post fix*

A+B*C/D-E

Step	Scan line/ip symbol	Postfix	opstack
1	A	A	----------
2	+	A	+
3	B	AB	+
4	*	AB	+*
5	C	ABC	+*
6	/	ABC*	+/
7	D	ABC*D	+/
8	-	ABC*D/+	-
9	E	ABC*D/+E	-
	---------	ABC*D/+E-	

Eg.

- A+B*(C+D)
- (A*B-(C-D))/(E+F)
- (A+B)*C+D/(E+F*G)-H
- (A+B)-C*D/(E-F/G)

Trace of Conversion Algorithm

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

Scan character	Poststack	opstack
(	……	(
(	……	((
A	A	((
-	A	((-
(	A	((-(
B	AB	((-(
+	AB	((-(+
C	ABC	((-(+
)	ABC+	((-
)	ABC+-	(
*	ABC+-	(*
D	ABC+-D	(*
)	ABC+-D*	…….
$	ABC+-D*	$
(	ABC+-D*	$(
E	ABC+-D*E	$(
+	ABC+-D*E	$(+
F	ABC+-D*EF	$(+
)	ABC+-D*EF+	$
……	ABC+-D*EF+$ (postfix)	…….

Converting an Infix expression to Prefix expression

- The precedence rule for converting from an expression from infix to prefix are identical.
- **A+B-C is -+ABC**

  A+B-C (infix)

  = (+AB)-C

  = -+ABC (prefix)

example:

A $ B * C – D + E / F / (G + H) infix form

A $ B * C – D + E / F / (G + H) infix

=A $ B * C – D + E / F /(+GH)

=$AB* C – D + E / F /(+GH)

=*$ABC-D+E/F/(+GH)

=*$ABC-D+(/EF)/(+GH)

=*$ABC-D+//EF+GH

= (-*$ABCD) + (//EF+GH)

=+-*$ABCD//EF+GH which is in prefix form.

Algorithm infix to prefix

1.Scan one character at a time of an infix expression from right to left. 2.opstack = the empty stack

3.Repeat till there is data in infix expression

3.1 if scanned character is ‘)’ then push it to (operator stack) opstack

3.2 if scanned character is operand then push it to

prestack (Prefix stack)

3.3 if scanned character is operator then

if(opstack!=-1)

while(precedence (opstack[otos])>precedence(scan character)) then pop and push it into prestack

otherwise

push into opstack

3.4 if scanned character is ‘(’ then

pop and push into prestack until ‘)’ is not found and ignore both symbols

4.pop and push into prestack until opstack is not empty.

5.Pop and print from prefix stack to get prefix.

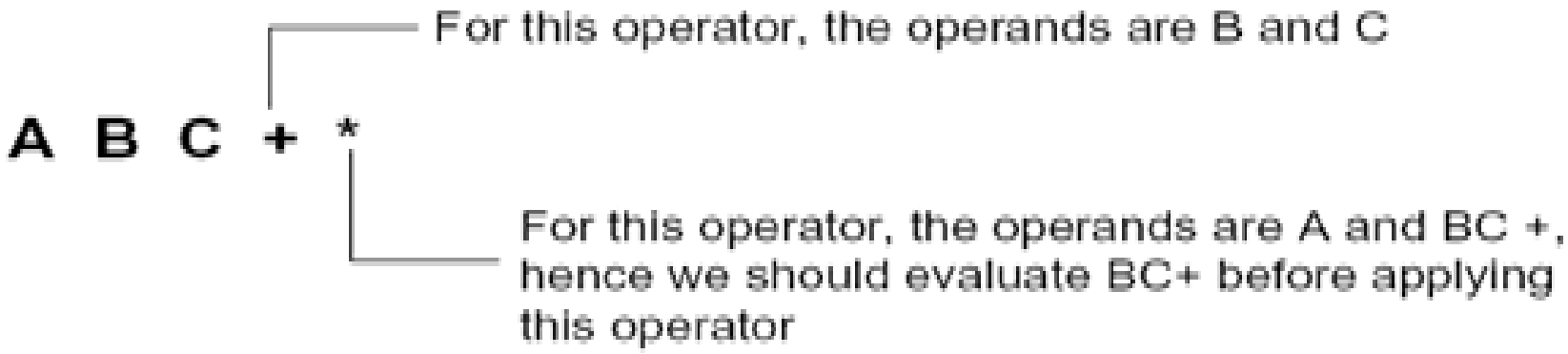
Opstack =operator stack

A $ B * C – D + E / F / (G + H)

SN.	Scan line	Prestack	opstack
1	)	----	)
2	H	H	)
3	+	H	)+
4	G	HG	)+
5	(	HG+	------
6	/	HG+	/
7	F	HG+F	/
8	/	HG+F	//
9	E	HG+FE	//
10	+	HG+FE//	+
11	D	HG+FE//D	+
12	-	HG+FE//D	+-
13	C	HG+FE//DC	+-
14	*	HG+FE//DC	+-*
15	B	HG+FE//DCB	+-*
16	$	HG+FE//DCB	+-*$
17	A	HG+FE//DCBA$*-+	

Evaluating the Postfix expression

Each operator in a postfix expression refers to the previous two operands in the expression



procedure:

- Each time we read an operand we push it onto a stack.
- When we reach an operator, its operands will be the top two elements on the stack.
- We can then pop these two elements
- perform the indicated operation on them and push the result on the stack so that it will be available for use as an operand of the next operator.

Example

3 4 5 * +

=3 20 +

=23 (answer)

- **Evaluating the given postfix expression:**

6 2 3 + - 3 8 2 / + * 2 $ 3 + = 52

=6 5 - 3 8 2 / + * 2 $ 3 +

=1 3 8 2 / + * 2 $ 3 +

=1 3 4 + * 2 $ 3 +

=1 7 * 2 $ 3 +

=7 2 $ 3 +

=49 3 +

= 52

Algorithm to evaluate the postfix expression

1. Scan one character at a time from left to right of given postfix expression
    1.1 if scanned symbol is operand then read its corresponding value and push it into vstack (value stack)
    1.2 if scanned symbol is operator then
        – pop and place into op2
        – pop and place into op1
        – compute result according to given operator and push result into vstack
2. pop and display which is required value of the given postfix expression
3. return

Trace of Evaluation

- **ABC+*CBA-+***

  - 123+*321-+*

Scanned character	Value	Op2	Op1	Result	vstack

Scanned character	Value	Op2	Op1	Result	vstack
A	1	……	…….	…….	1
B	2	……	……	…..	1 2
C	3	……	……	…..	1 2 3
+	…….	3	2	5	1 5
*	……	5	1	5	5
C	3	…….	……	……	5 3
B	2	…	…….		5 3 2
A	1	…….	…….	……	5 3 2 1
-	…….	1	2	1	5 3 1
+	…….	1	3	4	5 4
*	…….	4	5	20	20

Evaluating the Prefix Expression

To evaluate the prefix expression we use two stacks and some time it is called two stack algorithms.

One stack is used to store operators and another is used to store the operands.

Consider an example for this

+ 5 *3 2 prefix expression

= +5 6 =11

- **Illustration**: Evaluate the given prefix expression
- / + 5 3 – 4 2
- prefix equivalent to (5+3)/(4-2) infix notation = / 8 – 4 2
- = / 8 2 = 4

Program to evaluate postfix expression

```
/*program for evaluating postfix expression*/
#include<stdio.h>
#include<conio.h>
 #include<math.h>
#include<string.h>
void push(int);
int pop();
int vstack[100];
int tos=-1;
void main()
{
        int i,res,l,op1,op2,value[100];
        char postfix[100],ch;
        clrscr();
        printf("Enter a valid postfix\n");
        gets(postfix);
        l=strlen(postfix);
```

```
for(i=0;i<=l-1;i++)
        {
         if(isalpha(postfix[i]))
          {
                printf("Enter value of %c",
postfix[i]);
                scanf("%d", &value[i]);
                push(value[i]);
          }
        else
        {
ch=postfix[i];
op2=pop();
op1=pop();
 switch(ch)
{
        case '+':
        push(op1+op2);
        break;
        case'-':
        push(op1-op2);
        break;
        case'*':
        push(op1*op2);
        break;
        case'/':
        push(op1/op2);
        break;
        case'$':
        push(pow(op1,op2));
        break;
        case'%':
        push(op1%op2);
        break;
} }
}
printf("The reault is:"); res=pop(); printf("%d", res);
 getch();
}
```

```
/********insertion function**********/
void push(int val)
{
        vstack[++tos]=val;
}
/*********deletion function************/
int pop()
{
        int n; n=vstack[tos--];
        return(n);
}
```

/*program to convert infix to postfix expression*/

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<string.h>
#include<ctype.h>
int precedency(char);
void main()
{
int i,otos=-1, ptos=-1, l,  l1;
char infix[100],poststack[100],opstack[100];
printf("Enter a valid infix\n");
gets(infix);
l=strlen(infix);
l1=l;
for(i=0;i<=l-1;i++)
{
if(infix[i]=='(')
{
        opstack[++otos]=infix[i];
        l1++;
}
else if(isalpha(infix[i]))
        {
        poststack[++ptos]=infix[i];
        }
```

```
else if (infix[i]==')')
{
l1++;
while(opstack[otos]!='(')
{
poststack[++ptos]=opstack[otos];
  otos--;
}
otos--;
}
else       //operators
{
if(precedency(opstack[otos])>precedency(infix[i]))
{
poststack[++ptos]=opstack[otos--];
            opstack[++otos]=infix[i];
}
opstack[++otos]=infix[i];
}

}
while(otos!=-1)
{
   poststack[++ptos]=opstack[otos];
   otos--;
}
/**for displaying**/
for(i=0;i<l1;i++)
 {
            printf("%c",poststack[i]);
  }
getch();
}
```

```
/*precedency function*/
int precedency(char ch)
{
switch(ch)
{
case '$':
return(4);
// break;
case'*':
case'/':
return(3); // break;
case'+':
case'-':
return(2); // break;
default:
return(1);
}
}
```