Topic — STACK — Unit No. 03

* Introduction to Stack :-

1. A stack is an Abstract Data Type (ADT) that serves as a collection of elements.
2. It is a linear data structure that follows a particular order in which the operations are performed.
3. Example — Pile of plates.
   Here the plates are inserted from one end and removed from the same end.
4. The order of insertion and deletion in stack may be LIFO (Last In First Out) or FILO (First In Last Out).
5. It is an ordered list of similar data type.
6. The insertion of new elements in the stack and deletion of elements from the stack is done at same end. And that end is denoted as the "top" of the stack.
   So, items will be inserted from top end and deleted from top end.

(2.5 +)

Topic

Unit No. 03

(7) Four basic operations on stack are -

1) Push :- To insert a new element in the stack, push() function is used.

2) Pop :- To remove/delete an element from stack, pop() function is used.

3) ~~is Empty :-~~

The items are removed or popped in the reversed order in which they are pushed.

If the stack becomes empty or is empty, and if we still try to delete an element from it, then an underflow condition occurs.

3) isEmpty : Returns true if the stack is empty, else false.

4) isFull :- Returns true if the stack is full, else false.

isFull() ~~cond~~ occurs when there is.

Topic

Unit No.

no space to insert an element and still we are trying to insert a new element. This condition is called overflow condition

- Top :- Returns the top element of the stack.

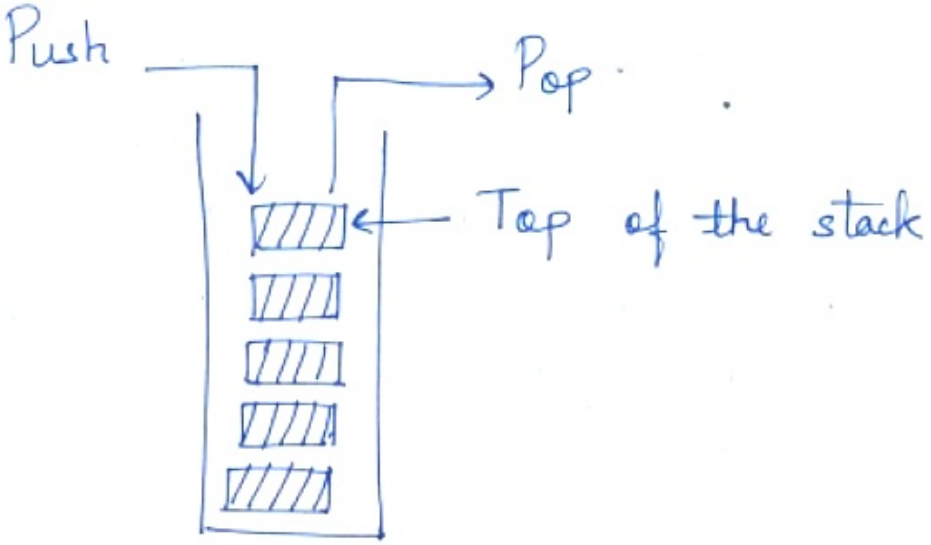

fig. Stack

* Stack as an Abstract Data Type :-

Here, stack of any particular type of elements is a finite sequence of those elements. We do the following operations on it.

P.T.O

1) Initialize the stack to be empty.
2) Determine whether the stack is empty or not.
3) Check whether the stack is full or not.
4) If the stack is not full, add/insert a new element at the top of the stack. This operation is called Push().
5) If the stack is not empty, then retrieve the element at the top.
6) If the stack is not empty, then delete the element at its top. This operation is called Pop().

* Algorithm for Push :-

```
begin
    if stack is full
        return
    end if
    else
        increment top
        stack[top] assign value
    end else
end procedure
```

Topic Unit No.

* Algorithm for Pop() :-

```
begin
  if stack is empty
     return
  end if
  else
     store value of stack[top]
     decrement top
     return value
  end else
end procedure
```

* Algorithm for ~~is e~~ isEmpty() :-

```
begin
   if top < 0
      return true
   else
      return false
end procedure
```

Topic

* Algorithm for isFull() :-

```
begin
    if top equals to MAXSIZE
        return true
    else
        return false
    end if
end procedure
```

```
void Push()
{   int x;
    if (top == size-1)
        pf("Stack Overflow");
    else
    {
        pf("Enter element to be inserted");
        ~~%d", &x);~~
        sf("%d", &x);
        top = top+1;
        a[top] = x;
    }
}
```

Topic

Unit No. 03

```
void Pop()
{
    if (top == -1)
        pf ("Stack underflow");
    else
    {
        data = a[top];
      pf("The deleted element is %d", data");
        top = top - 1;
        return data;
    }
}

void display()
{
    if (top == -1)
    pf("Stack Empty");
    else
    {
      for (i= top; i>=0; i--)
          pf("%d", a[i]);
    }
}
```

★ Time Complexity of Stack :-

- At a time, only a single element can be accessed in stack.
- While performing the Push() and Pop() operations, it takes O(1) time.

★ Applications of Stack :-

- As soon as the compiler encounters a function call, it gets pushed into the stack.
- In case of nested functions, the inner functions get executed before the outer functions. This is totally managed by stacks.

Applications :-

1. Infix to prefix conversion
2. Tower of Hanoi
3. N queens problem. (Backtracking concept).
4. String reversal.

Topic

Unit No. 03

★ Concept of Implicit & Explicit Stack :-

(1) Implicit stack is not implemented by a program developer(user). Every computer system has a built-in implicit stack.

(2) This implicit stack is used by compiler to implement recursion. Recursion is handled through an implicit stack.

(3) Each time a function is called, its return address, local variables and other parameters are allocated on the top of the stack.

When the control returns from a function, the most recent allocation of these variables is freed and the previous set of variables becomes active.

(4) For ex—

When a procedure with some arguments is invoked, the arguments are put on the stack and then the procedure is run.

Topic

Unit No.

ex- 3 + 4

Put 3 on the stack,
then 4,
then execute " + ".

ex- hd(list)

Put the value of the list on ~~stad~~ stack and then run/execute "hd".

If a procedure is defined to produce any results, then invoking the procedure will implicitly cause the results to be put on the stack when the procedure is finished.

ex-

If the numbers 3 and 4 are on the stack, and the addition procedure + runs, then the procedure after finishing will replace the top 2 items on the stack with the single result, the number 7.

So, the instruction :

3 + 4 ⇒

Topic

Unit No. 03

is equivalent to these four instructions :

3
4
+

⇒ In short, recursion involves the use of implicit stacks.

(2) Explicit Stack :-

1) Explicit stack is implemented by a programmer as an ADT.
2) Algorithms needing a stack can use this explicit stack.
3) Some applications of an explicit stack are -

- non-recursive version of a recursive algorithm.
- non-recursive tree traversal algorithm.
- conversion of an expression from infix to prefix/postfix form.

* Applications :-

1] Infix to Postfix Conversion.

- One of the applications of stack is conversion of infix expression to postfix form.

- Infix Expression :- The expression of the form <operand> <operator> <operand> is infix expression, where an operator is in-between every pair of operands.

ex- A + B

- Postfix Expression :- The expression of the form <operand> <operand> <operator> is postfix expression, where an operator is followed for every pair of operands.

ex- A B +

Topic

Unit No. 03

- Why there is a need of postfix representation of the expression?

→ The compiler scans the expression either from left to right or from right to left.

Consider: a + b * c + d

The compiler first scans the expression to evaluate the term b*c (* is at higher precedence than +), then again scans the expression to add 'a' to it.

The result is then added to 'd' after another scan.

The repeated scanning makes it very inefficient. Hence, it is better to convert the expression to postfix (or prefix) form before evaluation.

The corresponding expression in postfix form is:

a b c * + d +

^ *
/
+, -

The postfix expressions can be evaluated easily using a stack. (check pg. no. 15)

ex- Consider an expression :-

→ A + B * C / D - F + A ^ E

No.	Symbol Scanned	Stack	Postfix Expression	Description/ Reason
1.	A		A	Push A in stack ~~Print A~~
2.	+	+	A	Push + in stack
3.	B	+	AB	Print B
4.	*	+*	AB	Push * on +,
5.				coz * is of
				higher precedence
				than +
5.	C	+*	ABC	Print C
6	/	+		Pop * and
			ABC*	print
		+/	ABC*	Add / to stack
7.	D	+/	ABC*D	Print D
8.	-		ABC*D/	Pop / and print
			ABC*D/+	Pop + and "
		-	ABC*D/+	Push -
				Coz, - has

Prepared by :

Page No. 14

lowest ~~&~~ equal prece. than / ~~&~~ equal prece. to ~~than others~~ Precedence.

Topic

Unit No. 03

9.	F	−	ABC*D/+F	Print F
10.	+		ABC*D/+F−	Pop −, coz − have equal precedance as that of +.
		+	ABC*D/+F−	Push +
11.	A	+	ABC*D/+F−A	Print A
12.	^	+^	ABC*D/+F−A	Push ^
13.	E	+^	ABC*D/+F−AE	Print E
14.	Empty		ABC*D/+F−AE^	Pop ^
			ABC*D/+F−AE^+	Pop +

The postfix expression is :−

ABC * D/+F − AE^+

The expressions written in postfix (/prefix) form are evaluated faster as compared to infix notation as paranthesis (brackets) are not required in postfix. Hence, compiler finds it easier to evaluate an expression in postfix form instead of infix.

P.T.O.

Topic

(2) $(A+(B*C-(D/E\hat{}F)*G)*H)$

No	Symbol Scanned	Stack	Postfix Expression	Description
1	(	(		Push '('
2.	A	(	A	Print A
3	+	(+	A	Push +
4	(	(+(	A	Push (
5	B	(+(	AB	Print B
6.	*	(+(*	AB	Push *
7.	C	(+(*	ABC	Print C
8.	−	(+(	ABC*	Pop *, print.
		(+(−	ABC*	Push −
9.	(	(+(−(	ABC*	Push (
10	D	(+(−(	ABC*D	Print D
11.	/	(+(−(/	ABC*D	Push /
12	E	(+(−(/	ABC*DE	Print E
13	^	(+(−(/^	ABC*DE	Push ^
14	F	(+(−(/^	ABC*DEF	Print F
15	)			Pop all, till 1st closing) bracket is encountered and print.
		(+(−	ABC*DEF^/	

Topic

Unit No. 03

16	*	(+(-*	ABC*DEF^/	Push *
17	G	(+(-*	ABC*DEF^/G	Print G
18	)			Pop all till closing
				) bracket is
				encountered and
		(+	ABC*DEF^/G*-	print. Discard (
19	*	(+*	ABC*DEF^/G*-	Push *
20	H	(+*	ABC*DEF^/G*-H	Print H
21	)			Pop all till)
				closing bracket
				is encountered.
			ABC*DEF^/G*-H*+	Print it and
				discard the
				opening bracket
				(

So, the resultant postfix expression is :-

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

Unit No.

Topic

★ Algorithm for Infix to Postfix :-

① Start scanning the expression from left to right.

② If the scanned character is an operand, then print it.

③ ~~Else~~ Else

- If the precedence of the scanned operator is higher than the precedence of the operator in the stack (or stack is empty or has "("), then push the scanned operator in the stack.

- Else, pop all the operators, that have greater or equal precedence than the scanned operator. Once you pop them, push the scanned operator in stack. If we see a paranthesis while popping, then stop popping and push the scanned operator in the stack.

Topic

Unit No. 03

4. If the scanned character is an '(', push it to the stack.
5. If the scanned character is an ')', pop the stack and print it until a '(' is encountered and discard both the paranthesis.
6. Now, repeat steps 2-6 until the whole infix is scanned.
7. Print output.
8. Do the pop and ~~etc~~ print until stack is not empty.

Topic

Unit No. 03

* Algorithm to convert an infix expression into the postfix expression :-

(0) ~~Add~~ "y".

(1) Declare all the global variables,
an array of characters named as 'stack',
an integer variable 'top', initialized to -1.

(2) Function 'push (character x)'

```
set top ← top + 1
set stack [top] ← x
```

end of push ().

(3) Function 'pop ()' which returns a character.

```
return stack [top]
set top ← top - 1
```

end of pop().

(4) Function 'precedence (character c)' returns an integer

```
if c = '+' or c = '-' then
        return 1
```

Topic

Unit No.

```
else if c = '*' or c = '/' then
        return 2
else if c = '^' then
        return 3
else
        return -1
end of 'precedence()'
```

(5) Function 'main()'

```
Read a // the infix expression
Print 'the postfix expression is = "
for i = 0 to length(a) - 1 repeat
        set c ← a[i]

    if precedence(c) > 0 then
    while top ≠ -1 and
precedence(stack[top]) ≥ precedence(c) repeat
        print pop()
    end of while loop
```

```
        push c
end of if
else if c = ')' then
        set x ← pop()
while x ≠ '(' repeat
        print x
        set x ← pop()
end of while loop
end of else if
else if c = '(' then
        push (c)
end of else if
else
        print c
end of else
end of for loop
```

for i = 0 to top+1 repeat
print pop()
end of for loop
end of main() function.

Code :-

```
#include <stdio.h>
char stack[20];   // declaring global
int top = -1;     // variables.

//function to push elements into the
//stack

void push (char x)
{
    stack[++top] = x;
}

//function to pop elements
char pop()
{
   return stack[top--];
}
```

Topic

Unit No. 03

```
// function to return the value according
// to the precedence
int precedence (char c)
{
  if (c == '+' || c == '-')
      return 1;
  else if (c == '*' || c == '/')
      return 2;
  else if (c == '^')
      return 3;
  else
      return -1;
}

int main()
{
   int i;
   char a[20], c, x;
```

Topic Unit No.

```
// taking the input expression to be
// evaluated
printf("Enter the infix expression:");
  gets(a);

// conversion into postfix form
printf("The Postfix Expression is:");
for(i=0; a[i]!='\0'; i++)
{
    c= a[i];
if(precedence(c)>0)
{
  while(top!=-1 && precedence(stack[top])
    >=precedence(c))
  {
    printf("%c", pop());
  }
  push(c);
}
```

Topic

Unit No. 03

```
else if (c == ')')
{
    x = pop();
    while (x != '(')
    {
      printf("%c", x);
      x = pop();
    }
}
else if (c == '(')
    push(c);
else
    printf("%c", c);
} //end for loop
for(i= 0 ; i<= top + 1 ; i++)
    printf("%c", pop());
} //end main()
```

Topic

Unit No.

2] Evaluation of Postfix Expression :-

As Postfix Expression is without paranthesis and can be evaluated as two operands and an operator at a time, this becomes easier for the compiler and the computer to handle.

Algorithm -

1) Scan the input string from left to right.
2) For each input symbol,
a) If it is a digit (operand), then push it onto the stack.
b) If it is an operator (+, -, *, /, ^), then pop out the top two digits from the stack and apply the operator on them.
Later on, push the result on to the stack.
3) When the expression is ended, the number in the stack is the final answer.

Topic

Unit No. 03

ex—

① Infix Expression :- (6+4) * (9-3)

conversion

Postfix Expression :- 64 + 93 - *

Evaluation of above postfix expression using Stack :-

Scanned Symbol	Stack	Description	Evaluation
6	6	push (6)	-
4	6 4	push (4) on (6)	-
+		pop (4), pop (6)	Perform 6+4 = 10.
		Push (10)	
	10		
9	10 9	Push (9) on (10)	-
3	10 9 3	Push (3) on (9)	-
-	10	Pop (3), pop (9)	Perform 9 - 3 = 6
		Push (6)	
	10 6		
*		Pop (6), pop (10)	Perform (10) * (6) = 60
		Push (60)	

Topic

Unit No. 03

	60		
Endof exp		Pop (60)	Print the result as 60

(2) Infix Expression :- 4 + (5 * 6) // = 34.

Postfix Expression :- 4 5 6 * + // = 34

Scanned Symbol	Stack	Description	Evaluation
4	4	Push (4)	-
5	4 5	Push (5)	-
6	4 5 6	Push (6)	-
*		Pop (6), Pop (5)	Perform 5 * 6 = 30
		Push (30)	Push (30) into stack
	4 30		
+		Pop (30), pop (4)	Perform 4 + 30 = 34
		Push (34)	Push (34) into stack
End	34	Pop (34)	Final result is 34.

Topic

Unit No. 03

★ Pseudo Code to implement Postfix Expression Evaluation using Stack :-

```
#include <stdio.h>
#include <ctype.h>

#define max 30 // size of stack
#define postfixsize 30 // size of postfix
                       // expression.

int Stack[max];
int top = -1;

void push (int item)
{
    if(top >= max-1)
    {
        printf (" Stack Overflow ");
        return;
    }
    else
    {
        top = top + 1;
        stack [top] = item;
    }
}// end of push()
```

Topic

Unit No.

```
int pop()
{
    int item;
    if (top<0)
      printf (" Stack underflow ");
    else
    {
      item = Stack[top];
      top = top - 1;
      return (item);
    }
} // end of pop()

void EvaluatePostfix (char postfix[])
{
    int i, value, operand1, operand2;
    char ch;

    for(i=0; postfix[i] != ')'; i++)
    {
        ch = postfix[i];
```

```
if (isdigit(ch)) // if operand, push it.
{
    push(ch-'0');
/* ch-'0' takes the specified digit
instead of its ASCII value */
  } // end of if
  else if (ch == '+' || ch == '-' || ch == '*' ||
                                    ch = '/')
/* If operator, then pop the top 2
elements and perform the operation */
{
    operand 1 = pop();
    operand 2 = pop();

    switch (ch)
    {
        case '*':
            value = operand 2 * operand 1;
        break;
```

Topic Unit No.

```
case '/' :
    value = operand2 / operand1;
break;

case '+' :
    value = operand2 + operand1;
break;

case '-' :
    value = operand2 - operand1;
break;
} // end of switch

push(value); // push back the result
             // on stack
} // end of else if
printf("Result of Postfix Expression Evaluation
  is : %d ", pop());
} // end of EvaluatePostFix()
```

Topic

Unit No. 03

```
int main()
{
   int i;
char postfix[postfixsize];
//char array used to store postfix exp.
printf("Assume only 4 operators +, -, *, /
and single digit operands only");

printf("Enter postfix exp and press right
paranthesis ')' for end of exp :");

for (i=0; i<=postfixsize-1; i++)
  {
     scanf("%c", &postfix[i]);
   if (postfix[i]==')')
       break;
  }//endof for
   EvaluatePostfix(postfix);
   return 0;
}//end of main
```

Topic

Unit No.

Execution

⇓

Assume only 4 operators +, -, *, / and single digit operands only.

Enter postfix exp, and press right parenthesis ')' for end of exp: 4 5 6 * +)

Result of Postfix Expression Evaluation is :
34

Topic

Unit No. 03

3] Decimal to Binary Conversion :-

The conversion of decimal numbers to binary numbers is done because computer can understand only binary number system.

Representation of Decimal and Binary Numbers :-

Binary numbers :- (0, 1)
Decimal —"— :- (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

Ex- Convert 17 to Binary

2	17	1
2	8	0
2	4	0
2	2	0
	1	

Write it in reverse order.

⇒ Binary number of 17 is 10001

ex- 10

2	10	0
2	5	1
2	2	0
	1	

Write in reverse order.

10 ⇒ 1010

ex 7

2	7	1
2	3	1
	1	

Write in reverse order.

7 ⇒ 111

37

Topic

Unit No. 03

★ Algorithm for Decimal to Binary Conversion:

1. Input the decimal number.
2. Divide the number by 2 and store the remainder in stack / array.
3. Repeat step 2 until the number cannot be divided or further.
4. Print the remainder in a reverse order.

★ Code :-

```
int stack[20]; // for storing binary number.
int top = -1; // initially, stack is empty.

void push(int x)
{
    stack[++top] = x;
}

void pop()
{
    top--;
}
```

Topic Unit No.

```
void decimalToBinary(int num)
{
  if (num == 0)
    { printf("0");
      return;
    }
  while (num > 0)
  {
    push(num % 2); // push remainder
                   // on stack
    num = num/2;
  }
  while (top != -1)
  {
    printf("%d", stack[top]);
    pop();
  }
}
int main()
{
  int num = 7; // Or scan no. from user.
  decimalToBinary(num);
  return 0;
}
```