DATA STRUCTURES

LECTURE-8

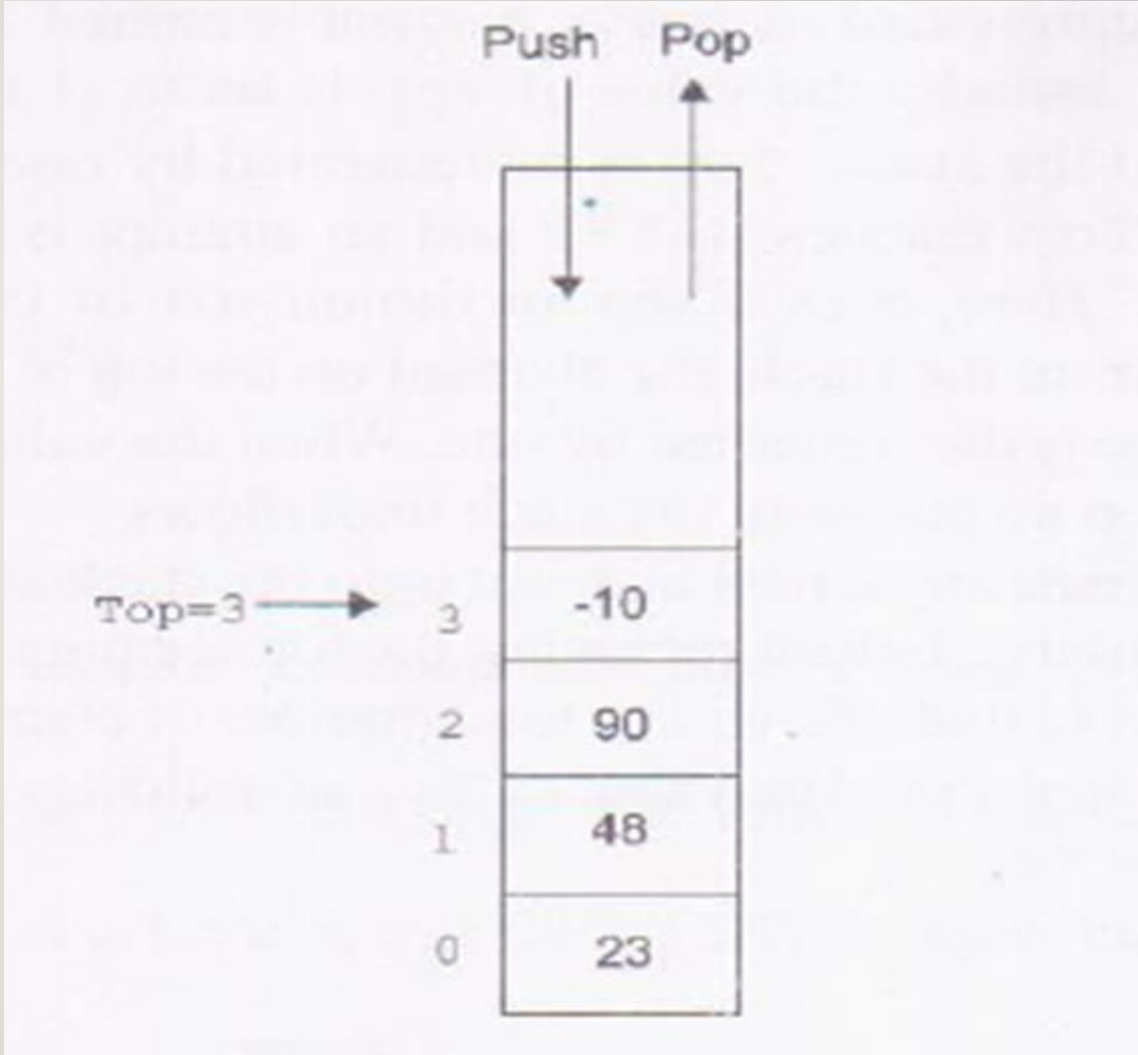
STACK

Dr. Sumitra Kisan

Stack

- A stack is a linear data structure in which an element can be inserted or deleted only at one end of the list.
- A stack works on the principle of last in first out and is also known as a Last-In-First-Out (LIFO) list.
- A bunch of books is one of the common examples of stack. A new book to be added to the bunch is placed at the top and a book to be removed is also taken off from the top.
- Therefore, in order to take out the book at the bottom, all the books above it need to be removed from the bunch.

- A stack is a linear data structure in which an element can be added or removed only at one end called the top of the stack.
- In stack terminology, the insert. and delete operations are known as **push** and **pop** operations, respectively.



- The two basic operations that can be performed on a stack are:

  Push: to insert an element onto a stack.
  Pop: to access and remove the top element of the stack.

- Before inserting a new element onto the stack, it is necessary to test the condition of overflow.

- ***Overflow*** occurs when the stack is full and there is no space for a new element and an attempt is made to push a new element.

- If the stack is not full, push operation can be performed successfully.

- Similarly, before removing the top element from the stack, it is necessary to check the condition of underflow.

- ***Underflow*** occurs when the stack is empty and an attempt is made to pop an element.

- If the stack is not empty, pop operation can be performed successfully.

MEMORY REPRESENTATION OF STACKS

- A stack can be represented in memory either as an array or as a singly linked list.
- In both the cases, insertion and deletion of elements is allowed at one end only.
- Insertion and deletion in the middle of the array or the linked list is not allowed.
- An array representation of a stack is static but linked list representation is dynamic in nature.

To implement stack as an array in C language, the following structure named stack needs to be defined.

```
struct stack
        {
                int item[MAX];
                int Top;
        };
```

- *When stacks are represented as arrays, a variable named **Top** is used to point to the top element of the stack.*
- *Initially the value of **Top** is set to **-1** to indicate an empty stack.*
- *To push an element onto the stack, **Top** is incremented by one and the element is pushed at that position.*
- *When **Top** reaches **MAX-1** and an attempt is made to push a new element, ' then stack overflows. Here, **MAX** is the maximum size of the stack.*
- *To pop (or remove) an element from the stack, the element on the top of the stack is assigned to a local variable and then Top is decremented by one. When the value of Top is equal to -1 and an attempt is made to pop an element, the stack underflows.*
- *The total number of elements in a stack at a given point of time can be calculated from the value of Top as follows: number of elements = **Top + 1***

PUSH Operation

push (Stack, Top)

1. If Top = MAX-1

        print “Overflow: stack is full” and go to step 5

   End if

2. Read item

3. Set Top =Top +1

4. Set Stack[Top] = item

5. End

Pop Operation

Pop (Stack, Top)

1. If Top = -1

    print “Underflow: stack is full” and go to step 5

    End if

2. Set item = Stack[Top]

3. Set Top =Top -1

4. Print “ the popped element is: ”, item

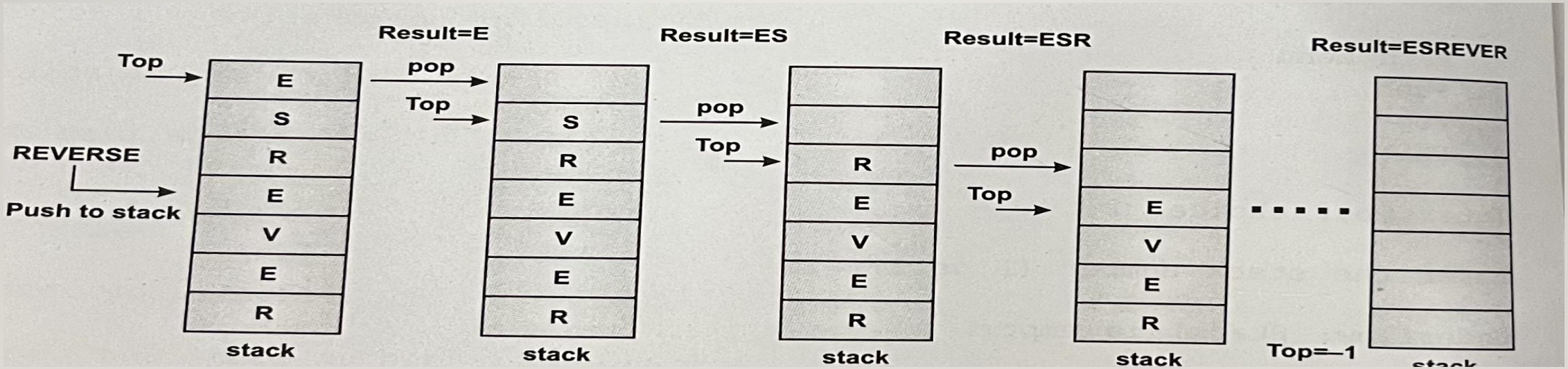
5. End

APPLICATIONS OF STACKS

- Reversing strings
- Checking whether the arithmetic expression is properly parenthesized
- Converting infix notation to postfix and prefix notations evaluating postfix expressions
- Implementing recursion and function calls

Reversing Strings

- To reverse a string, the characters of the string are pushed onto the stack one by one as the string is read from left to right.
- Once all the characters of the string are pushed onto the stack, they are popped one by one. Since the character last pushed in comes out first, subsequent pop operations result in reversal of the string.
- For example, to reverse the string "REVERSE", the string is read from left to right and its characters are pushed onto a stack, starting from the letter R, then E, V, E and so on.



```
reversal(s, str)

1.  Set i = 0
2.  While(i < length_of_str)
        Push str[i] onto the stack
        Set i = i + 1
    End While
3.  Set i = 0
4.  While(i < length_of_str)
        Pop the top element of the stack and store it in str[i]
        Set i = i + 1
    End While
5.  Print "The reversed string is: ", str
6.  End
```

Polish Expression, Reverse Polish Expression and their Compilation

- The general way of writing arithmetic expressions is known as the infix notation where the binary operator is placed between two operands on which it operates.
- For example, the expressions a+b and (a-c) *d, ((a+b) * (d/f) -f) are in infix notation.
- The order of evaluation in these expressions depends on the parentheses and the precedence of operators.
- It is difficult to evaluate an expression in infix notation.
- Thus, the arithmetic expressions in the infix notation are converted to another notation which can be easily evaluated by a computer system to produce correct result.
- The notation in which an operator occurs before its operands is known as the prefix notation (polish notation). For example, **+ab and *-acd are in prefix notation.**
- On the other hand, the notation in which an operator occurs after its operands is known as the postfix notation (reverse polish or suffix notation). For example, **ab+ and ac-d* are in postfix notation**.

Conversion of Infix to Postfix Notation

The steps for converting the expression manually are given here.

(1) The actual order of evaluation of the expression in infix notation is determined by inserting parentheses in the expression according to the precedence and associativity of operators.

(2) The expression in the innermost parentheses is converted into postfix notation by placing the operator after the operands on which it operates.

(3) Step 2 is repeated until the entire expression is converted into a postfix notation.

Example: convert the expression a+b*c into equivalent postfix notation

(1) Since the precedence of * is higher than +. the expression b* c has to be evaluated first. Hence, the expression is written as (a+(b*c))
(2) The expression in the innermost parentheses, that is, b*c is converted into its postfix notation. Hence, it is written as bc*. The expression now becomes (a+bc*)
(3) Now the operator + has to be placed after its operands. The two operands for + operator are a and the expression bc*.

The expression now becomes (abc*+)

Conversion of Infix to Postfix Notation Using Stack

Infix Q is read from left to right and the following steps are performed:

1. Push (onto stack and add) to the end of Q
2. Scan Q from left to right and repeat steps 3 to 6 for each element of Q until stack is empty.
3. if an operand is encountered, add it to postfix expression P
4. if a left parenthesis is encountered, push it onto stack
5. if an operator is encountered then:
   i. repeatedly pop from stack and add to P each operator which has the same precedence as or higher precedence than the operator.
   ii. Add operator to stack
6. If a right parenthesis is encountered , then:
   i. repeatedly pop from stack and add to P each operator until a left parenthesis is encountered.
   ii. Remove the left parenthesis.
7. Exit

Example

Conversion of Infix to Prefix Notation

- The conversion of infix expression to prefix expression is similar to conversion of infix to postfix expression.
- The only difference is that the expression in the infix notation is scanned in the reverse order, that is, from right to left. Therefore, the stack in this case stores the operators and the closing (right) parenthesis.

Convert following expression into Prefix:

(1) (a+b)*(c-d)
(2) a+(b*c-(d/e^f)*)*h
(3) a/(b-c)*d+g
(4) (a+b)*c-(d-e)^(f+g)

Evaluation of a Postfix Expression

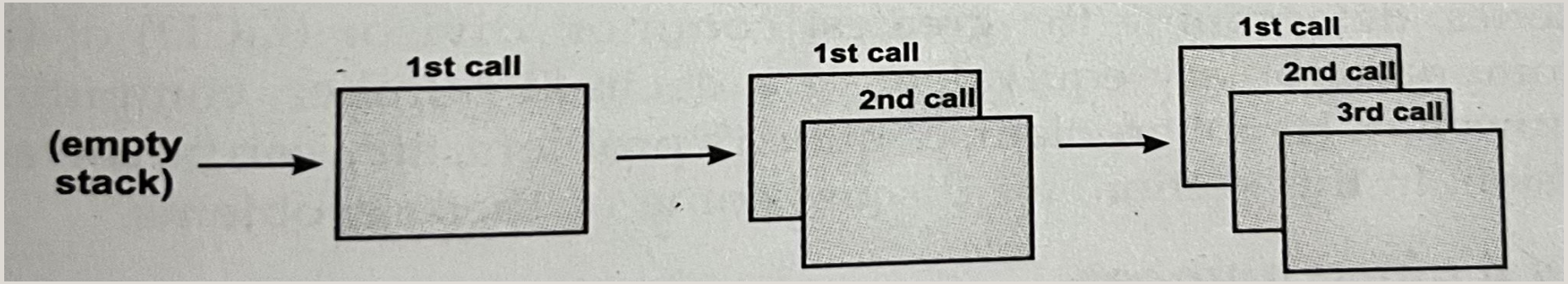
1. Add the right parenthesis at the end of expression P
2. Scan P from left to right and repeat step 3 and 4 for each element of P until the right parenthesis is encountered.
3. If an operand is encountered, put it on stack
4. If an operator is encountered then:
   i. Remove top two elements of stack, where A is the top element and B is the next to top element.
   ii. Evaluate B op A
5. Set Value equal to the top element on stack
6. Exit

Recursion

A recursive function is said to be well defined if it satisfies the following two properties:

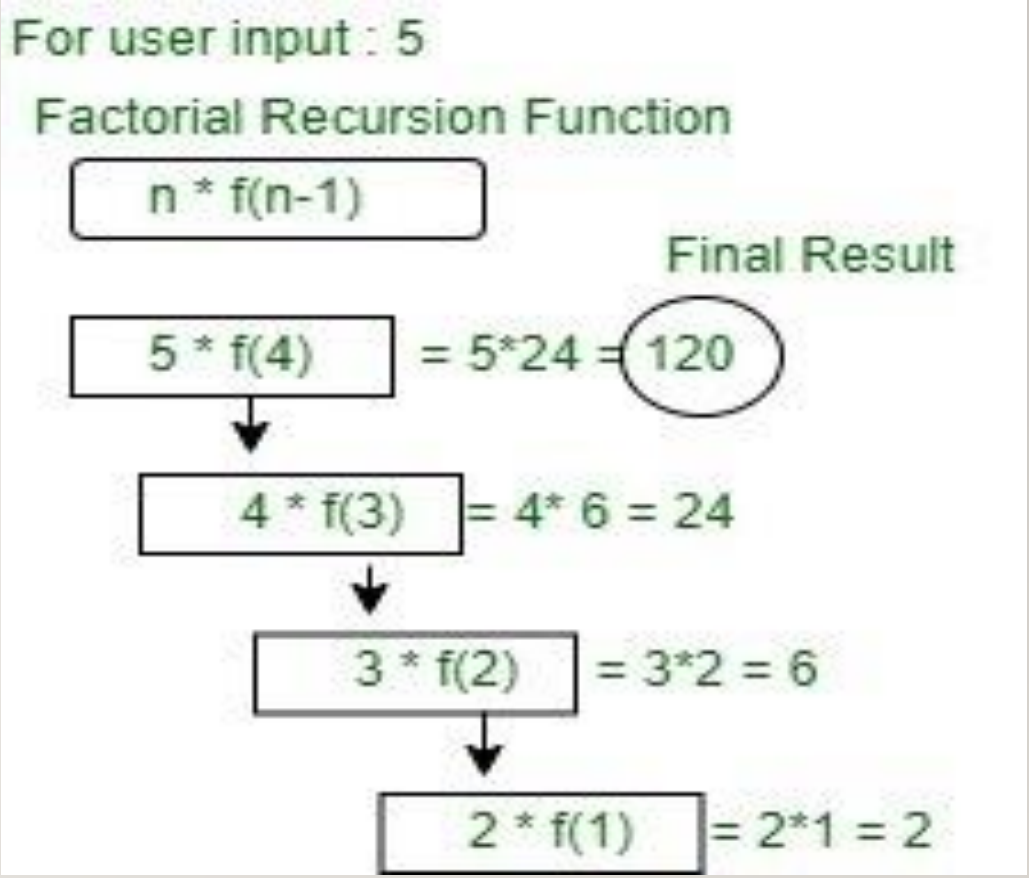
1. The arguments passed to the recursive function must have certain base values for which the function does not call itself. i.e a recursive function must include a condition or a statement to terminate the function. The statement that terminate the function is known as ***base case*** and the rest of the part is known as ***general case***.
2. Each time the function calls itself, the argument of the function must get closer to the base value.

- In each recursive call, the current values of the parameter, local variables and the return address where the control has to return from the call are required to be stored.
- A stack is maintained to store all these values.



Find the Factorial of n

```
#include <stdio.h>
  int f(int n)
{
if (n == 0 || n == 1)
        return 1;
  else
        return n * f(n - 1);
}
  int main()
{
    int n = 5;
    printf("factorial of %d is: %d", n, f(n));
    return 0;
}
```



C program to demonstrate working of recursion

```
#include <stdio.h>
void printFun(int test)
{
    if (test < 1)
        return;
    else {
          printf ("%d",test );
          printFun(test - 1);
          printf ("%d",test );
   return;
    }
}
  int main()
{
    int test = 3;
    printFun(test);
}
```