Basic Data Structures

Unit 3 : Linear Data Structures I

Prepared by :
Prof. Khushbu Chauhan
Computer Engg. Dept.

MPSTME, NMIMS

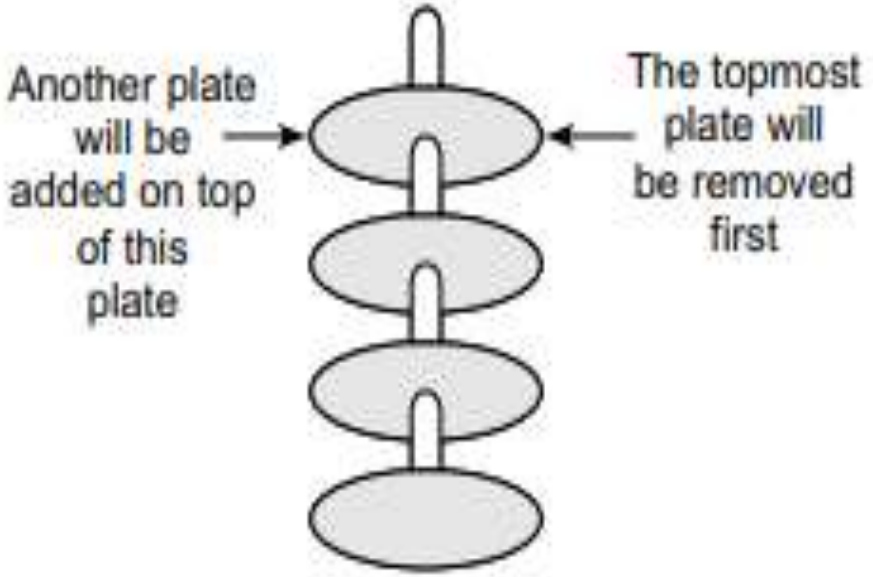
SVKM'S NMIMS
Deemed to be UNIVERSITY

Outlines

- Introduction to Stacks
- Array representation of Stacks
- Operations of a Stack
- Applications of a Stack
- Conversion and Evaluation of Arithmetic Expression
- Recursion

Introduction to Stacks

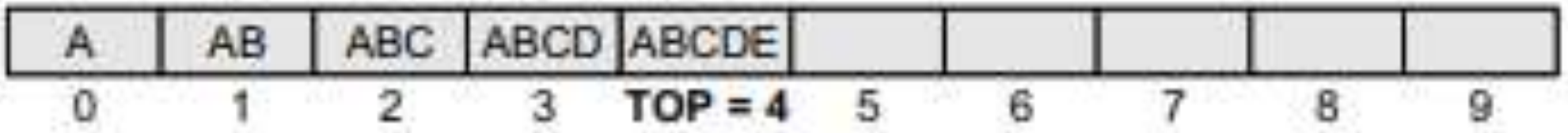
- Stack is a linear data structure which stores its elements in an ordered manner.
- The elements in a stack are added and removed only from one end, which is called the TOP. Hence, a stack is called a LIFO (Last-In-First-Out) data structure, as the element that was inserted last is the first one to be taken out.



- The system stack ensures a proper execution order of functions. Therefore, stacks are frequently used in situations where the order of processing is very important, especially when the processing needs to be postponed until other conditions are fulfilled.
- Stacks can be implemented using either arrays or linked lists.

Array representation of Stacks

- In the computer's memory, stacks can be represented as a linear array. Every stack has a variable called TOP associated with it, which is used to store the address of the topmost element of the stack. It is the position where the element will be added to or deleted from.
- There is another variable called MAX, which is used to store the maximum number of elements that the stack can hold. If TOP = NULL, then it indicates that the stack is empty and if TOP = MAX–1, then the stack is full.



- The stack shows that TOP = 4, so insertions and deletions will be done at this position. In the above stack, five more elements can still be stored.

Operations of a Stack

- A stack supports three basic operations: push, pop, and peek
- **push():** When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- **pop():** When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
- **isEmpty():** It determines whether the stack is empty or not.

- **isFull():** It determines whether the stack is full or not.
- **peek():** It returns the element at the given position.
- **count():** It returns the total number of elements available in a stack.
- **change():** It changes the element at the given position.
- **display():** It prints all the elements available in the stack.

PUSH Operation

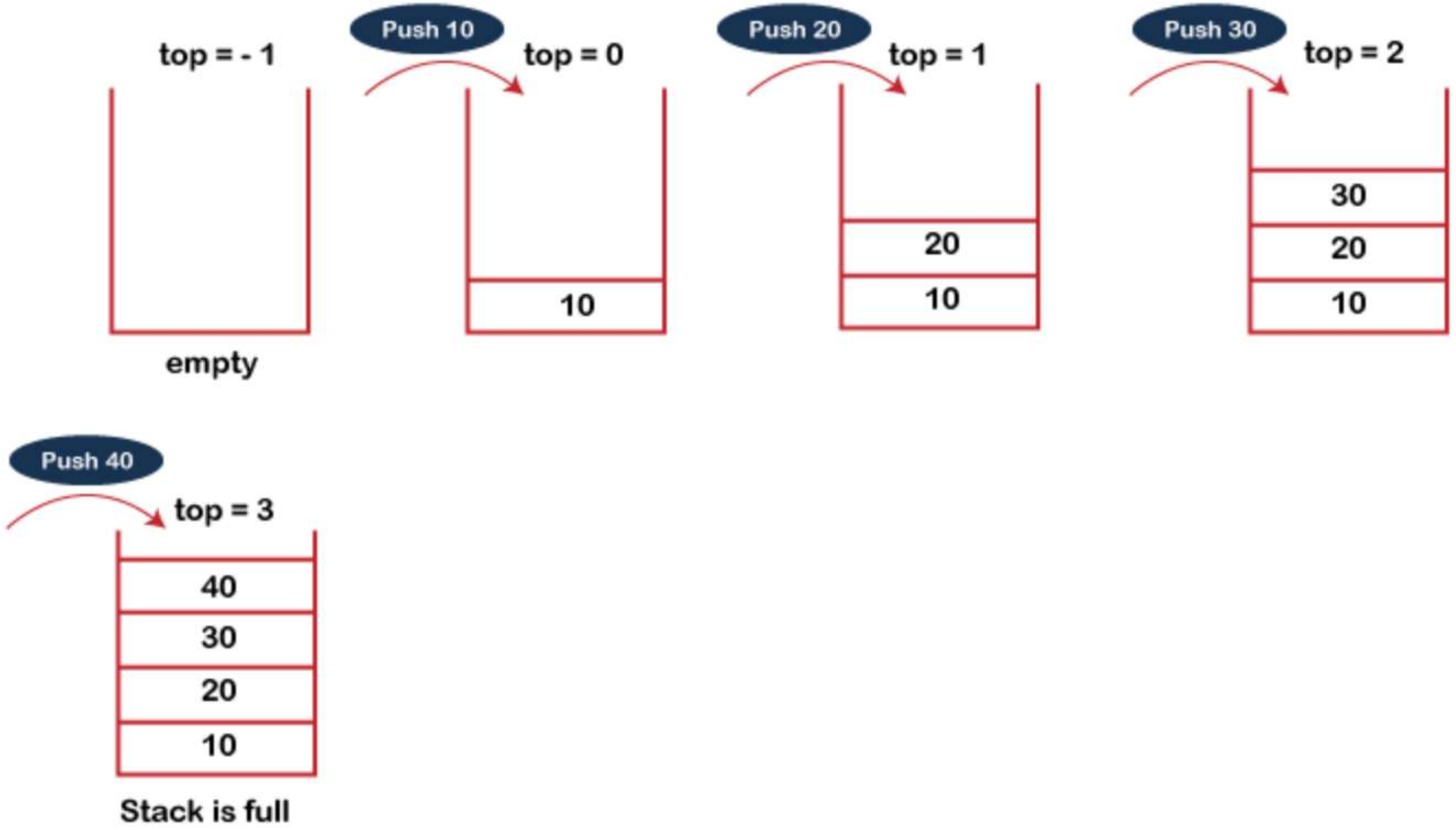
- The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack. However, before inserting the value, we must first check if TOP=MAX–1, because if that is the case, then the stack is full and no more insertions can be done. If an attempt is made to insert a value in a stack that is already full, an OVERFLOW message is printed.

Algorithm of PUSH operation:

- Step 1: IF TOP = MAX-1 PRINT OVERFLOW [END OF IF]
- Step 2: SET TOP = TOP+1
- Step 3: SET STACK[TOP] = VALUE
- Step 4: END
- In Step 1, first check for the OVERFLOW condition. In Step 2, TOP is incremented so that it points to the next location in the array. In Step 3, the value is stored in the stack at the location pointed by TOP.

```
void push (int val,int n) //n is size of the stack
{
        if (top == n-1 )
        printf("\n Overflow");
        else
        {
                top = top +1;
                stack[top] = val;
        }
}
```

- To insert an element with value 6, we first check if TOP=MAX–1. If the condition is false, then we increment the value of TOP and store the new element at the position given by stack[TOP].


top = - 1
Push 10
top = 0
10
Push 20
top = 1
20
10
Push 30
top = 2
30
20
10
empty
Push 40
top = 3
40
30
20
10
Stack is full

POP Operation

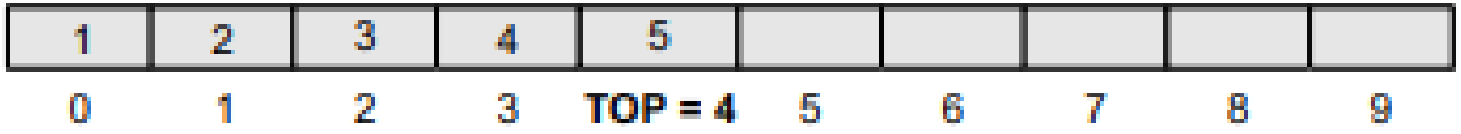
- The pop operation is used to delete the topmost element from the stack. However, before deleting the value, we must first check if TOP=NULL because if that is the case, then it means the stack is empty and no more deletions can be done. If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed.

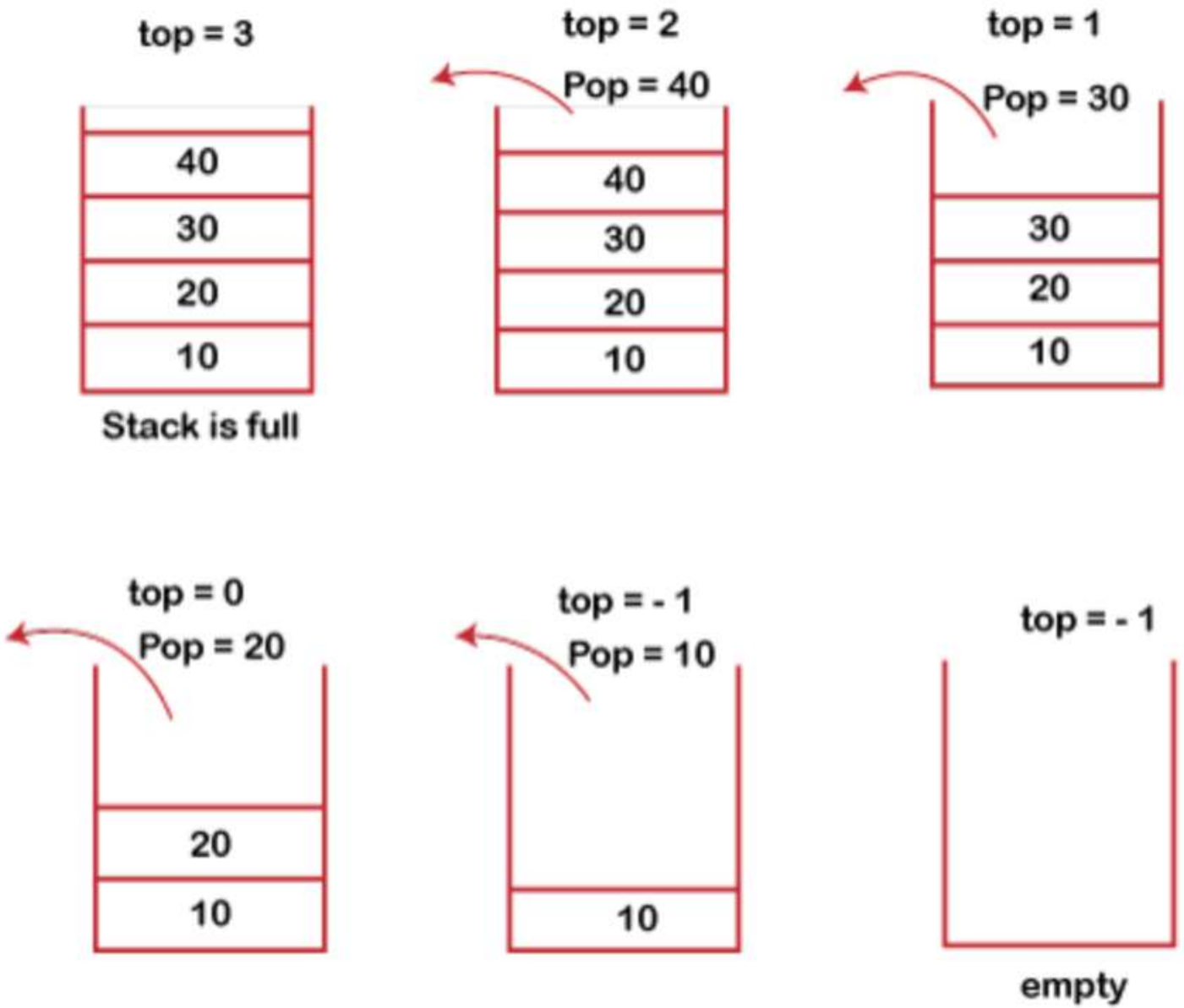
Algorithm of POP operation:

- Step 1: IF TOP = NULL PRINT UNDERFLOW [END OF IF]
- Step 2: SET VAL = STACK[TOP]
- Step 3: SET TOP = TOP-1
- Step 4: END
- In Step 1, we first check for the UNDERFLOW condition. In Step 2, the value of the location in the stack pointed by TOP is stored in VAL. In Step 3, TOP is decremented.

```
int pop ()
{
        if(top == -1)
         {
                printf("Underflow");
        return 0;
         }
         else
         {
                return stack[top - - ];
         }
}
```

- To delete the topmost element, we first check if TOP=NULL. If the condition is false, then we decrement the value pointed by TOP. Thus, the updated stack becomes as shows.



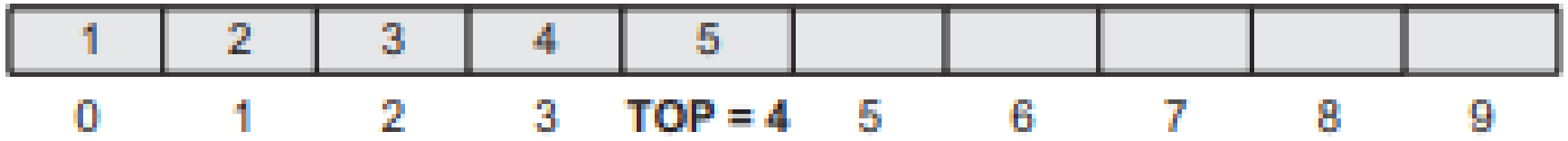

top = 3
40
30
20
10
Stack is full
top = 2
Pop = 40
40
30
20
10
top = 1
Pop = 30
30
20
10
top = 0
Pop = 20
20
10
top = - 1
Pop = 10
10
top = - 1
empty

PEEK Operation

- Peek is an operation that returns the value of the topmost element of the stack without deleting it from the stack.
- The Peek operation first checks if the stack is empty, i.e., if TOP = NULL, then an appropriate message is printed, else the value is returned.

Algorithm of Peek operation:

- Step 1: IF TOP = NULL PRINT STACK IS EMPTY Go to Step 3
- Step 2: RETURN STACK[TOP]
- Step 3: END



```
int peek()
{
        if (top == -1)
        {
                printf("Underflow");
                return 0;
         }
        else
        {
                return stack [top];
        }
}
```

Applications of stack

- Reversing a list
- Parentheses checker
- Conversion of an infix expression into a postfix expression
- Evaluation of a postfix expression
- Conversion of an infix expression into a prefix expression
- Evaluation of a prefix expression
- Recursion
- Tower of Hanoi

Reversing a list

- A list of numbers can be reversed by reading each number from an array starting from the first index and pushing it on a stack. Once all the numbers have been read, the numbers can be popped one at a time and then stored in the array starting from the first index.

Parentheses checker

- Stacks can be used to check the validity of parentheses in any algebraic expression. For example, an algebraic expression is valid if for every open bracket there is a corresponding closing bracket. For example, the expression (A+B} is invalid but an expression {A + (B – C)} is valid.

Evaluation of Arithmetic Expressions

- **Polish Notations**:
- Infix, postfix, and prefix notations are three different but equivalent notations of writing algebraic expressions. But before learning about prefix and postfix notations, let us first see what an infix notation is. We all are familiar with the infix notation of writing algebraic expressions.
- While writing an arithmetic expression using infix notation, the operator is placed in between the operands.

For example, A+B;

- here, plus operator is placed between the two operands A and B. Although it is easy for us to write expressions using infix notation, computers find it difficult to parse as the computer needs a lot of information to evaluate the expression. Information is needed about operator precedence and associativity rules, and brackets which override these rules. So, computers work more efficiently with expressions written using prefix and postfix notations

Need of Prefix and Postfix Notations

- Prefix notations are needed when we require operators before the operands while postfix notations are needed when we require operators after the operands.
- Prefix notations are used in many programming languages like LISP.
- Prefix notations and Prefix notations can be evaluated faster than the infix notation.
- Postfix notations can be used in intermediate code generation in compiler design.
- Prefix and Postfix notations are easier to parse for a machine.
- With prefix and postfix notation there is never any question like operator precedence.
- There is no issue of left-right associativity.

Postfix Notation

- Postfix notation was developed by Jan Łukasiewicz who was a Polish logician, mathematician, and philosopher. His aim was to develop a parenthesis-free prefix notation (also known as Polish notation) and a postfix notation, which is better known as Reverse Polish Notation or RPN.
- In postfix notation, as the name suggests, the operator is placed after the operands. For example, if an expression is written as A+B in infix notation, the same expression can be written as AB+ in postfix notation. The order of evaluation of a postfix expression is always from left to right. Even brackets cannot alter the order of evaluation

- The expression (A + B) * C can be written as:

  [AB+]*C or AB+C* in the postfix notation

Example :Convert the following infix expressions into postfix expressions.

(a) (A–B) * (C+D)

(b) (A + B) / (C + D) – (D * E)

Conversion of an infix expression into a postfix expression

- Let I be an algebraic expression written in infix notation. I may contain parentheses, operands, and operators. For simplicity of the algorithm we will use only +, –, *, /, % operators.
- The precedence of these operators can be given as follows:

  Higher priority *, /, %

  Lower priority +, –
- The order of evaluation of these operators can be changed by making use of parentheses.

- The algorithm uses a stack to temporarily hold operators. The postfix expression is obtained from left-to-right using the operands from the infix expression and the operators which are removed from the stack. The first step in this algorithm is to push a left parenthesis on the stack and to add a corresponding right parenthesis at the end of the infix expression. The algorithm is repeated until the stack is empty.
- The algorithm accepts an infix expression that may contain operators, operands, and parentheses.

Algorithm

Step 1: Add) to the end of the infix expression

Step 2: Push (on to the stack

Step 3: Repeat until each character in the infix notation is scanned

IF a (is encountered, push it on the stack

IF an operand (whether a digit or a character) is encountered, add it postfix expression.

IF a) is encountered, then

a. Repeatedly pop from stack and add it to the postfix expression until a (is encountered.

b. Discard the (. That is, remove the (from stack and do not add it to the postfix expression

IF an operator is encountered, then

a. Repeatedly pop from stack and add each operator (popped from the stack) to the postfix expression which has the same precedence or a higher precedence than

b. Push the operator to the stack

[END OF IF]

Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty

Step 5: EXIT

Example

Convert the following infix expression into postfix expression using the algorithm.

A – (B / C + (D % E * F) / G)* H

Infix Character Scanned	Stack	Postfix Expression
	(	
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
)		A B C / D E F * % G / + H * -

Example

Convert the following infix expression into postfix expression using the algorithm.

(L – K / A) * (C / B – A)

Evaluation of an infix expression into a postfix expression

- Using stacks, any postfix expression can be evaluated very easily. Every character of the postfix expression is scanned from left to right. If the character encountered is an operand, it is pushed on to the stack. However, if an operator is encountered, then the top two values are popped from the stack and the operator is applied on these values. The result is then pushed on to the stack.

```
Step 1: Add a ")" at the end of the
        postfix expression
Step 2: Scan every character of the
        postfix expression and repeat
        Steps 3 and 4 until ")"is encountered
Step 3: IF an operand is encountered,
        push it on the stack
        IF an operator O is encountered, then
        a. Pop the top two elements from the
           stack as A and B as A and B
        b. Evaluate B O A, where A is the
           topmost element and B
           is the element below A.
        c. Push the result of evaluation
           on the stack
        [END OF IF]
Step 4: SET RESULT equal to the topmost element
        of the stack
Step 5: EXIT
```

Example

Evaluate the following infix expression into postfix expression using the algorithm.

Infix : 9 – ((3 * 4) + 8) / 4

Postfix: 9 3 4 * 8 + 4 / -

Evaluation: 9 – (12 + 8) / 4

= 9 – 20 / 4

= 9 – 5

= 4

Character Scanned	Stack
9	9
3	9, 3
4	9, 3, 4
*	9, 12
8	9, 12, 8
+	9, 20
4	9, 20, 4
/	9, 5
-	4

Example

Evaluation of a postfix expression using a stack.

(a) 6 5 3 + 9 * +

(b) 5 6 2 + * 12 4 / -

(c) AB + CD / * GH * +

A=2, B=4, C= 6, D=3, G=8, H=7

Conversion of an infix expression into a prefix expression

- In prefix notation, as the name suggests, the operator is placed before the operands.
- Convert the following infix expressions into prefix expressions:

(a) (A–B) * (C+D)

(b) (A + B) / (C + D) – (E * F)

(c) (L – K / A) * (C / B – A)

(d) (A + B * C / D – E + F / G / (H + I))

Algorithm

```
Step 1: Scan each character in the infix
        expression. For this, repeat Steps
        2-8 until the end of infix expression
Step 2: Push the operator into the operator stack,
        operand into the operand stack, and
        ignore all the left parentheses until
        a right parenthesis is encountered
Step 3: Pop operand 2 from operand stack
Step 4: Pop operand 1 from operand stack
Step 5: Pop operator from operator stack
Step 6: Concatenate operator and operand 1
Step 7: Concatenate result with operand 2
Step 8: Push result into the operand stack
Step 9: END
```

Evaluation of an infix expression into a prefix expression

- + – 2 7 * 8 / 4 12
- + – 2 7 * 8 3
- + – 2 7 24
- + 5 24
- 29

Character scanned	Operand stack
12	12
4	12, 4
/	3
8	3, 8
*	24
7	24, 7
2	24, 7, 2
-	24, 5
+	29

Examples

- Evaluation of a prefix expression using a stack.

(a) - + / 16 + 2 3 * 10 4 * 16 3

(b) 1 2 + 3 4 * 6 / - 6 1 - 3 3 + / +

Advantages

- Postfix notation has fewer overheads of parenthesis. i.e., it takes less time for parsing.
- Postfix expressions can be evaluated easily as compared to other notations.

Recursion

- A recursive function is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself. Since a recursive function repeatedly calls itself, it makes use of the system stack to temporarily store the return address and local variables of the calling function. Every recursive solution has two major cases.

- Base case, in which the problem is simple enough to be solved directly without making any further calls to the same function.
- Recursive case, in which first the problem at hand is divided into simpler sub-parts. Second the function calls itself but with sub-parts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts.

Types of Recursion

- Direct Recursion- A function is said to be directly recursive if it explicitly calls itself. The function Func() calls itself for all positive values of n, so it is said to be a directly recursive function.
- Indirect Recursion- A function is said to be indirectly recursive if it contains a call to another function which ultimately calls it.
- Tail Recursion- A recursive function is said to be tail recursive if no operations are pending to be performed when the recursive function returns to its caller. when the called function returns, the returned value is immediately returned from the calling function.

SVKM'S NMIMS
Deemed to be UNIVERSITY

Recursive functions

- Factorial of a number
- Fibonacci Series
- Greatest Common Divisor
- Tower of Hanoi

Factorial of a number

```cpp
#include <iostream>
using namespace std;

// Define a function to calculate factorial
// recursively
int factorial(int n)
{
    // Base case - If n is 0 or 1, return 1
    if (n == 0 || n == 1) {
        return 1;
    }
    // Recursive case - Return n multiplied by
    // factorial of (n-1)

    return n * factorial(n - 1);
}

int main()
{
    int num = 5;
    // printing the factorial
    cout << "Factorial of " << num << " is "
         << factorial(num) << endl;

    return 0;
}
```

```
/tmp/tBuA0AuUNb.o
Factorial of 5 is 120


=== Code Execution Successful ===
```

Fibonacci Series

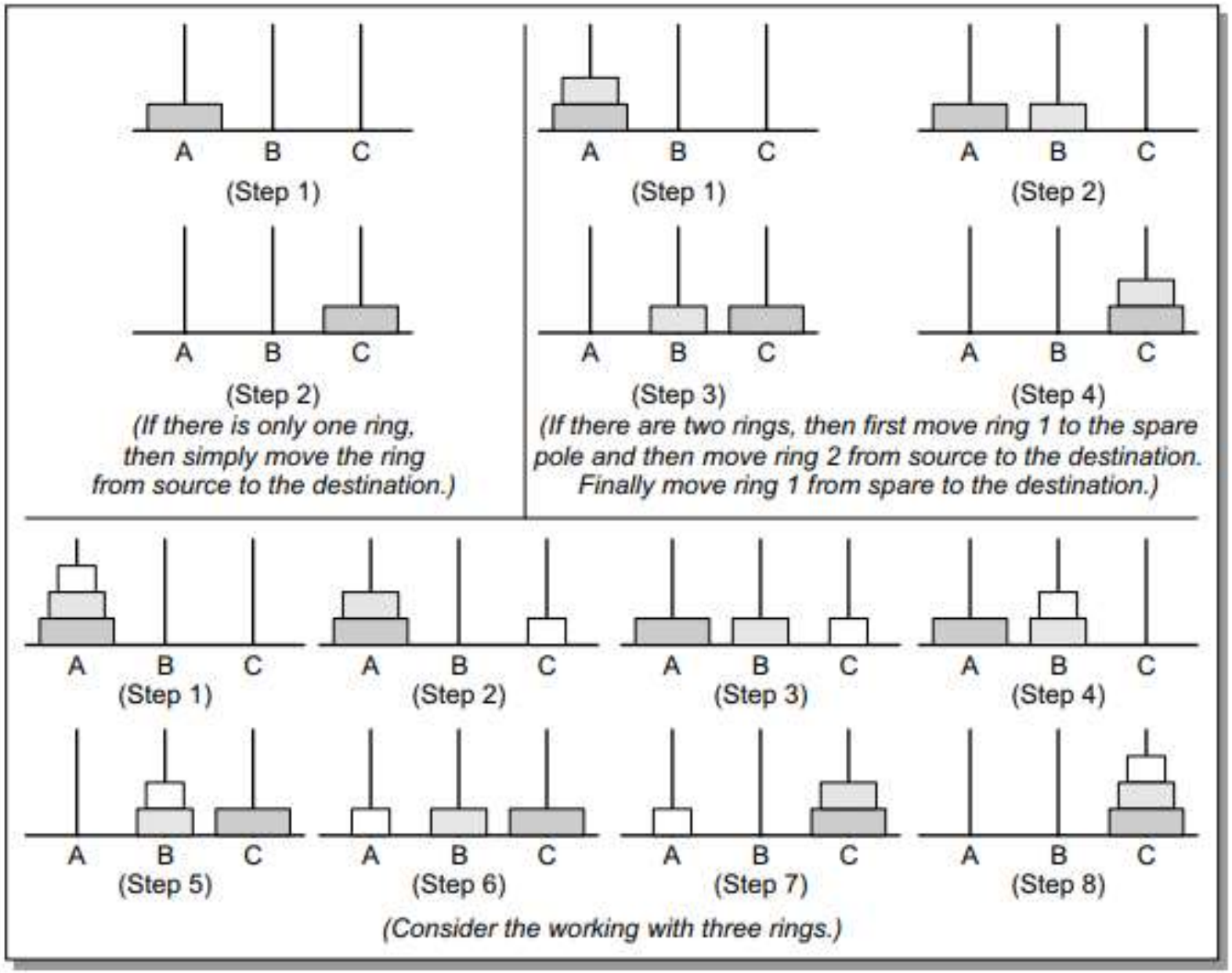
```cpp
#include <iostream>
using namespace std;
int fib(int x) {
   if((x==1)||(x==0)) {
      return(x);
   }else {
      return(fib(x-1)+fib(x-2));
   }
}
int main() {
   int x , i=0;
   cout << "Enter the number of terms of series : ";
   cin >> x;
   cout << "\nFibonnaci Series : ";
   while(i < x) {
      cout << " " << fib(i);
      i++;
   }
   return 0;
}
```

```
/tmp/s91LGOopEo.o
Enter the number of terms of series : 10

Fibonnaci Series :  0 1 1 2 3 5 8 13 21 34

=== Code Execution Successful ===
```

Greatest Common Divisor

```cpp
#include <iostream>
using namespace std;

int hcf(int n1, int n2);

int main()
{
   int n1, n2;

   cout << "Enter two positive integers: ";
   cin >> n1 >> n2;

   cout << "H.C.F of " << n1 << " & " <<  n2 << " is: " << hcf(n1, n2
       );

   return 0;
}

int hcf(int n1, int n2)
{
    if (n2 != 0)
       return hcf(n2, n1 % n2);
    else
       return n1;
}
```

```
/tmp/DgbRfBlj4a.o
Enter two positive integers: 5 25
H.C.F of 5 & 25 is: 5

=== Code Execution Successful ===
```


A B C
(Step 1)
A B C
(Step 2)
(If there is only one ring, then simply move the ring from source to the destination.)
A B C
(Step 1)
A B C
(Step 2)
A B C
(Step 3)
A B C
(Step 4)
(If there are two rings, then first move ring 1 to the spare pole and then move ring 2 from source to the destination. Finally move ring 1 from spare to the destination.)
A B C
(Step 1)
A B C
(Step 2)
A B C
(Step 3)
A B C
(Step 4)
A B C
(Step 5)
A B C
(Step 6)
A B C
(Step 7)
A B C
(Step 8)
(Consider the working with three rings.)

The advantages of using a recursive program

- Recursive solutions often tend to be shorter and simpler than non-recursive ones.
- Code is clearer and easier to use.
- Recursion works similar to the original formula to solve a problem.
- Recursion follows a divide and conquer technique to solve problems.
- In some (limited) instances, recursion may be more efficient

The drawbacks of using a recursive program

- Recursion is implemented using system stack. If the stack space on the system is limited, recursion to a deeper level will be difficult to implement.
- Aborting a recursive program in midstream can be a very slow process.
- Using a recursive function takes more memory and time to execute as compared to its nonrecursive counterpart.
- It is difficult to find bugs, particularly while using global variables.

DISCUSSION…

SVKM'S NMIMS
Deemed to be UNIVERSITY

THANK YOU