UNIT-I

Applications of Stack -Implementations of Towers of Hanoi, Parenthesis checker, conversions from infix to prefix and infix to postfix.

Trees: Basic terminology, Binary Tree, Complete Binary Tree, Full Binary Tree, Representation of Trees- Using Arrays and Linked lists (advantages and disadvantages), Implementation of tree traversal.

****

Prerequisites:

A data structure can be defined as follows...

Data structure is a method of organizing a large amount of data more efficiently so that any operation on that data becomes easy.

OR

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used

Advantages of Data structures:

Efficiency

If the choice of a data structure for implementing a particular ADT is proper, it makes the program very efficient in terms of time and space

Reusability

The data structure provides reusability means that multiple client programs can use the data structure.

Abstraction

The data structure specified by an ADT also provides the level of abstraction. The client cannot see the internal working of the data structure, so it does not have to worry about the implementation part. The client can only see the interface

Types of Data Structures

There are two types of data structures:

i) Primitive data structure

ii) Non-primitive data structure

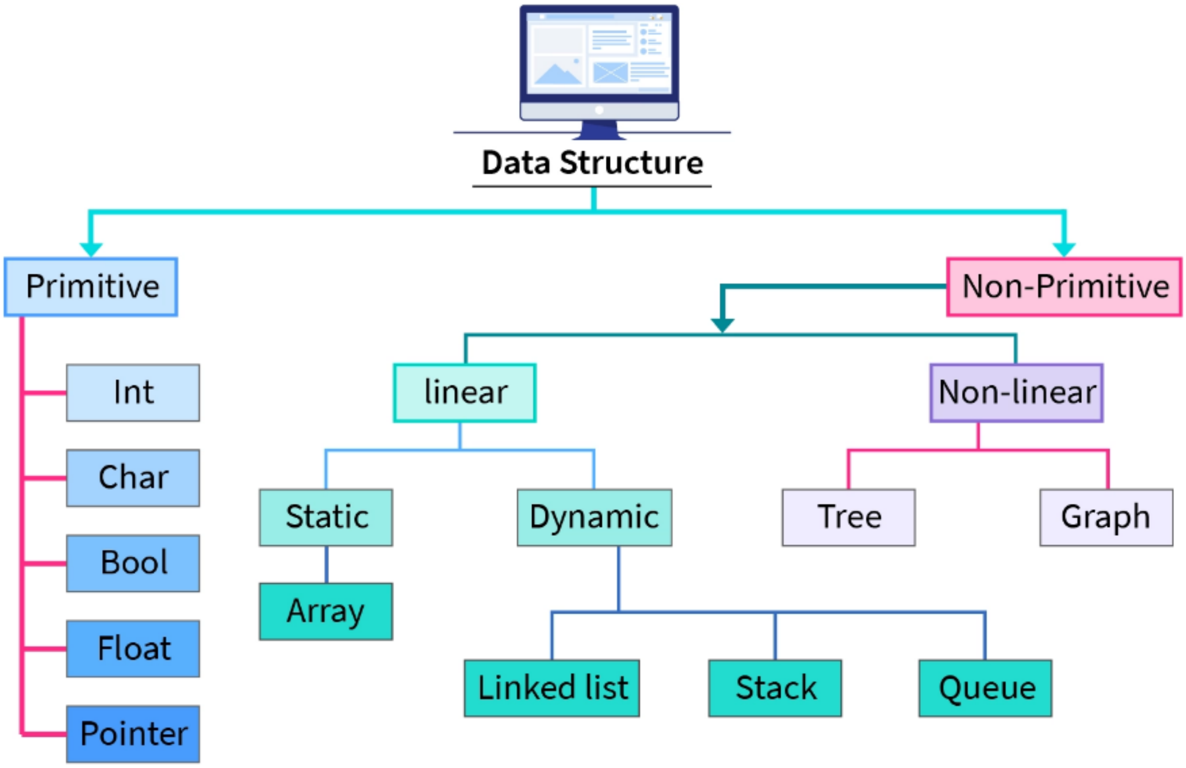
Primitive Data structure: The primitive data structures are primitive data types. The int, char, float, double, and pointer are the primitive data structures that can hold a single value.

Non-Primitive Data structure: The non-primitive data structure is divided into two types:

- Linear data structure
- Non-linear data structure

Linear Data Structure: The arrangement of data in a sequential manner is known asa linear data structure. The data structures used for this purpose are Arrays, Linked list, Stacks, and Queues. In these data structures, one element is connected to onlyone another element in a linear form.

Non-Linear Data Structure: When one element is connected to the 'n' number of elements known as a non-linear data structure. The best example is trees and graphs. In this case, the elements are arranged in a random manner.



Static Representation of Data structures:

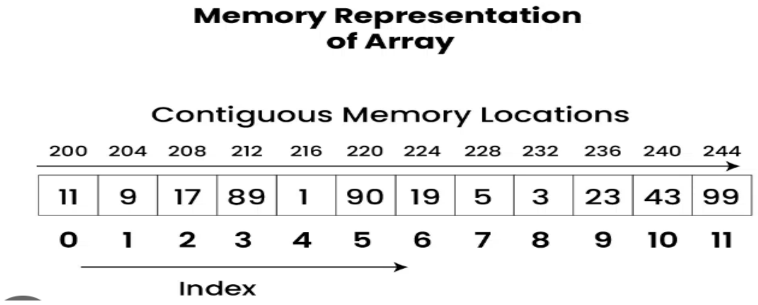
Static data structure:

- In Static data structure the size of the structure is fixed. It is a type of data structure where the size is allocated at the compile time.
- Therefore, the maximum size is fixed. The content of the data structure can be modified but without changing the memory space allocated to it.

  Example: Array.

Array : An array is a collection of elements identified by index or key values.

- The size of the array is specified at the time of creation and remains constant.
- For example, an array of integers can be declared as int arr[8];



Static Vs Dynamic Data Structures (Comparisons)

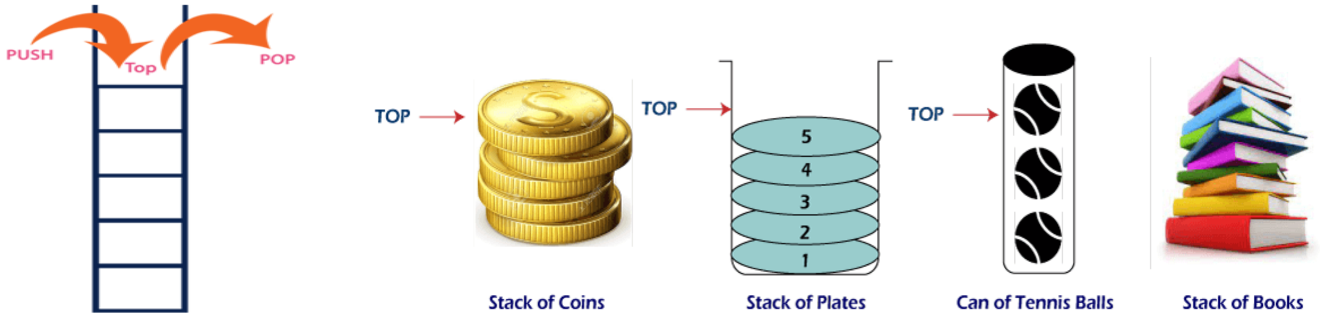
S. No	Static Memory Allocation	Dynamic Memory Allocation
1	When the allocation of memory performs at the compile time, then it is known as static memory.	When the memory allocation is done at the execution or run time, it is called dynamic memory allocation.
2	The memory is allocated at the compile time.	The memory is allocated at the runtime.
3	In static memory allocation, the memory cannot be changed while executing a program.	In dynamic memory allocation, while executing a program, the memory can be changed.
4	Static memory allocation is preferred in an array.	Dynamic memory allocation is preferred in the linked list.
5	It saves running time as it is fast.	It is slower than static memory allocation.
6	Static memory allocation allots memory from the stack.	Dynamic memory allocation allots memory from the heap.
7	Once the memory is allotted, it will remain from the beginning to end of the program.	Here, the memory can be alloted at any time in the program.
8	Static memory allocation is less efficient as compared to Dynamic memory allocation.	Dynamic memory allocation is more efficient as compared to the Static memory allocation.
9	This memory allocation is simple.	This memory allocation is complicated.

STACKS

Stack Definition:

“A stack is a linear data structure. In which, insertions and deletions are takes place at one end, called the top. Stack maintains a variable called top, which keeps track of the top most elements in the stack”.

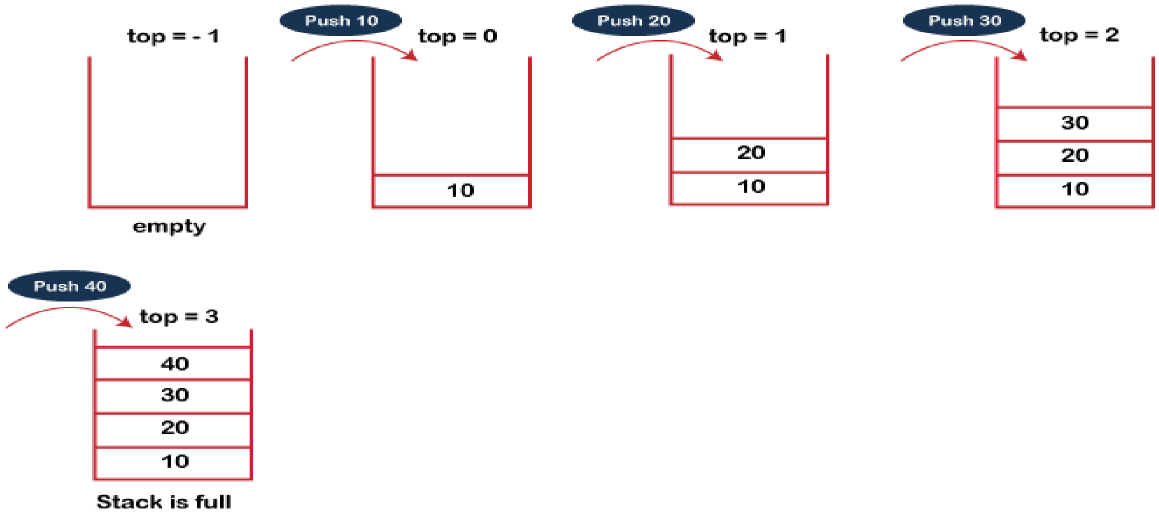
“Stack is a recursive data structure having pointer to its top element. Stacks are sometimes called as **Last-In-First-Out (LIFO)** lists i.e. the element which is inserted first in the stack, will be deleted last from the stack”



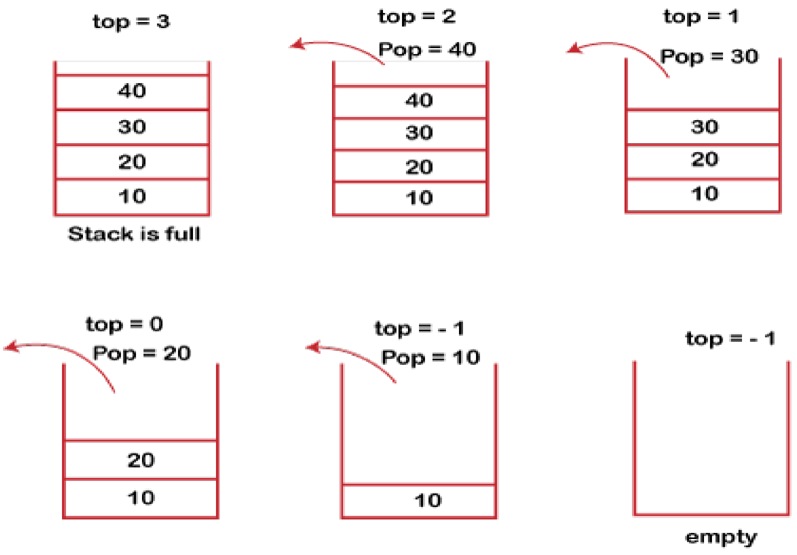
Standard Stack Operations:

- **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.
- **display( ):** It prints all the elements available in the stack

PUSH operation



POP operation:



Top and its value:

Top position	Status of stack
-1	Empty
0	Only one element in the stack
N-1	Stack is full
N	Overflow

Stack implementation using Arrays:

Push operation: Adding an element into the top of the stack is referred to as push operation.

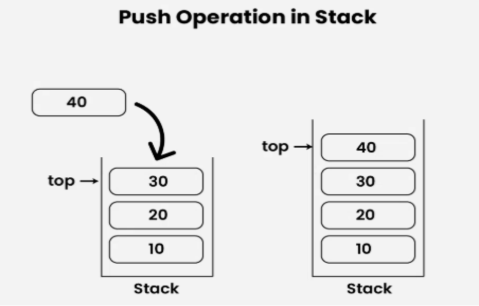
Push operation involves following two steps.

1. Increment the variable Top so that it can now refer to the next memory location.
2. Add element at the position of incremented top. This is referred to as adding new element atthe top of the stack.

Stack is overflow when we try to insert an element into a completely filled stacktherefore, our main function must always avoid stack overflow condition.

Algorithm:

```
1. begin
2. if top = n then stack full
3. top = top + 1
4. stack (top) : = item;
5. end
```

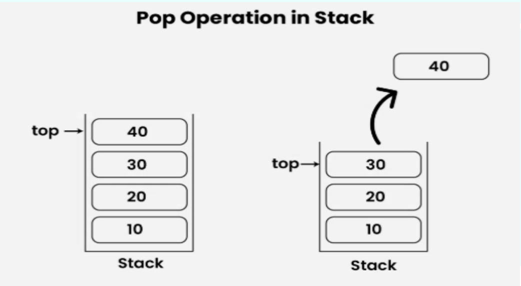


Time Complexity: O(1)

Pop operation: Deletion of an element from the top of the stack is called pop operation. The value of the variable top will be incremented by 1 whenever an item is deleted from the stack. The top most element of the stack is stored in an another variable and then the top is decremented by 1. Theunderflow condition occurs when we try to delete an element from an already empty stack.

Algorithm:

```
1. begin
2. if top = 0 then stack empty;
3. item := stack(top);
4. top = top - 1;
5. end;
```

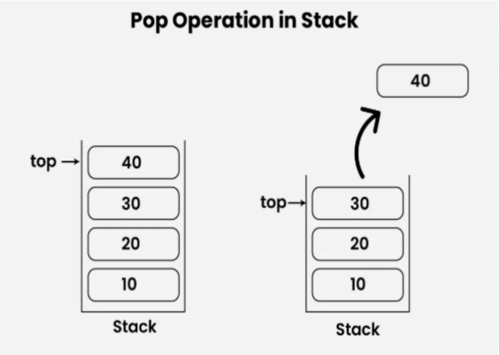


Time Complexity: O(1)

Peek operation: Peek operation involves returning the element which is present at the top of the stack without deleting it. Underflow condition can occur if we try to return the top element in an alreadyempty stack.

Algorithm:

```
PEEK (STACK, TOP)
1. Begin
2. if top = -1 then stack empty
3. item = stack[top]
4. return item
5. End
```



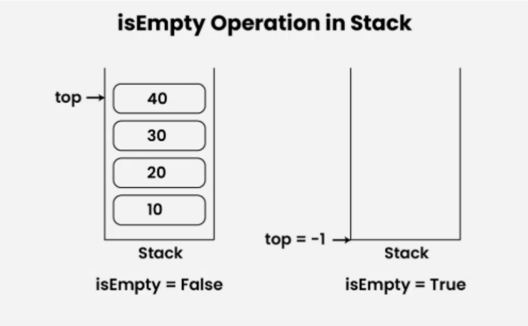
Time complexity: O(n)

isEmpty Operation

Returns true if the stack is empty, else false.

Algorithm for isEmpty Operation:

- Check for the value of **top** in stack.
- If **(top == -1)**, then the stack is **empty** so return **true** .
- Otherwise, the stack is not empty so return **false** .

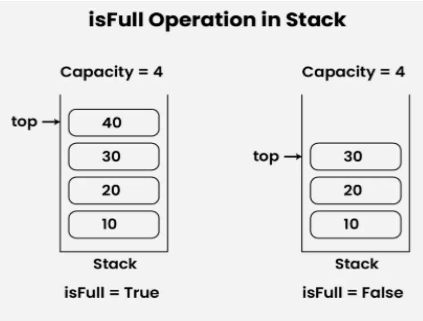


isFull Operation

Returns true if the stack is full, else false.

Algorithm for isFull Operation:

- Check for the value of **top** in stack.
- If **(top == capacity-1),** then the stack is **full** so return **true.**
- Otherwise, the stack is not full so return **false.**



Applications of stacks:

1. **Expression Evaluations & Conversions:** A stack is a very effective data structure for evaluating arithmetic expressions in programming languages. An arithmetic expression consists of operands and operators.
2. **Backtracking(Recursion):** Backtracking is another application of Stack. It is a recursive algorithm that is used for solving the optimization problem.
3. **Parsing (Delimiter Checking) :** The common application of Stack is delimiter checking, i.e., parsing that involves analyzing a source program syntactically. It is also called parenthesis checking.
4. **Function calls:** Stacks are used to keep track of the return addresses of function calls, allowing the program to return to the correct location after a function has finished executing.
5. **Editors:** Undo and Redo functions in any text editor.
6. **Tree Traversals:** Stacks are useful for Depth First Search Traversal method.
7. **Browsers:** Stacks are useful for function calls, storing the activation records. The history of a web browser is stored in the form of a stack.

Expression Types:

Based on the operator position, expressions are divided into THREE types. They are as follows...

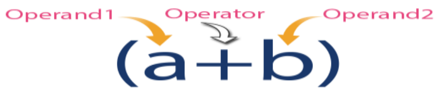
1. **Infix Expression**
2. **Postfix Expression**
3. **Prefix Expression**

Infix Expression:

- In infix expression, operator is used in between the operands.
- The general structure of an Infix expression is as follows...

Operand1 Operator Operand2

Example:

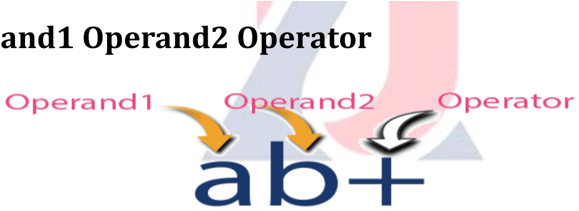


Postfix Expression:

- In postfix expression, operator is used after operands. We can say that "**Operator follows the Operands**".
- The general structure of Postfix expression is as follows...

Operand1 Operand2 Operator

Example:

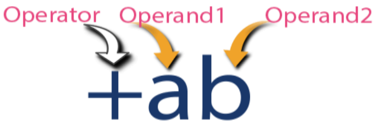


Prefix Expression:

- In prefix expression, operator is used before operands. We can say that "**Operands follows the Operator**".
- The general structure of Prefix expression is as follows...

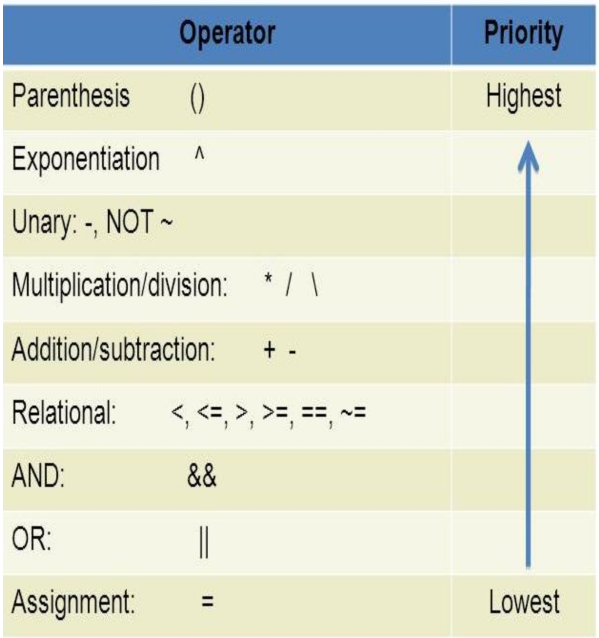
Operator Operand1 Operand2

Example:



❖ Every expression can be represented using all the above three different types of expressions. And we can convert an expression from one form to another form like Infix to Postfix, Infix to Prefix, Prefix to Postfix and vice versa.

Order of Operations (Precedence) from left to right.



Operator	Priority
Parenthesis ()	Highest
Exponentiation ^	↑
Unary: -, NOT ~	
Multiplication/division: * / \	
Addition/subtraction: + -	
Relational: <, <=, >, >=, ==, ~=	
AND: &&	
OR: ||	
Assignment: =	Lowest

^	**Exponentiation**	**right to left**

Evaluation of Infix Expressions:

Example:

i) 4 + 6 * 2 = > 4 + (6 * 2)

= (4 + 12)

= 16

ii) 2 + 3 * 4 – 5 => 2 + (3 * 4) - 5

= (2 + 12) – 5

= (14 – 5)

= 9

iii) 2 * 6 /2 – 3 + 7 => (2 * 6) / 2 – 3 + 7

= (12 / 2) – 3 + 7

= (6 – 3) + 7

= (3 + 7)

= 10

iv) 2 * 3 + 5 * 4 – 9 => (2 * 3) + 5 * 4 – 9

= (6) + (5*4) – 9

= (6) + (20) – 9

= (6 + 20) – 9

= (26) – 9

= (26 – 9)

= 17

Representation of infix to Prefix Expression (Polish notation) using stack

Example

A + B * C – D => A + (* B C) – D
⇨ (+ A * B C) – D
⇨ (- + A * B C D)
⇨ - + A * B C D

Algorithm:

1. First, reverse the infix expression given in the problem. While reversing each '('will become ')' and each ')' becomes '('.
2. Scan the expression from **left to right**. Whenever the operands arrive, print them.
3. If the operator arrives and the stack is found to be empty, then simply push the operator into the stack.
4. If the operator is '(', then push it into the stack. If the operator is ')', then pop all the operators from the stack till it finds (opening bracket in the stack.
5. If the incoming operator has **higher precedence and same precedence with the TOP** of the stack, push the incoming operator into the stack.
6. If the incoming operator has **lower precedence** than the TOP of the stack, pop, and print the top of the stack. Test the incoming operator against the top of the stack .
7. When we reach the end of the expression, pop, and print all the operators from the top of the stack.
8. If the top of the stack is '(', push the operator on the stack.
9. At the end, reverse the output

Example:01- A + B * C – D

- If we are converting the expression from infix to prefix, we need first to reverse the expression.
- The Reverse expression would be: D-C*B+A

Input expression	Stack	Prefix expression
D		D
-	-	D
C	-	DC
*	-*	DC
B	-*	DCB
+	-+	DCB*
A	-+	DCB*A
		DCB*A+-

The above expression, i.e., **DCB*A+-,** is not a final expression. We need to reverse this expression to obtain the prefix expression.

Now infix expression is -+A*BCD

Example:02- A + (B * C)

- If we are converting the expression from infix to prefix, we need first to reverse the expression.
- The Reverse expression would be: (C * B) + A

Input expression	Stack	Prefix expression
(	(	
C	(	C
*	(*	C
B	(*	CB
)		CB*
+	+	CB*
A	+	CB*
		CB*A+

The above expression, i.e., **CB*A+,** is not a final expression. We need to reverse this expression to obtain the prefix expression.

Now infix expression is +A*BC

```
// C Program to Convert Infix to Prefix notation
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define MAX 100

char stack[MAX];
int top = -1;
void push(char c) {
   if (top < MAX - 1) {
       stack[++top] = c;
   }
}

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

```
char peek() {
    if (top == -1) return -1;
    return stack[top];
}

int precedence(char c) {
    switch (c) {
        case '^': return 3;
        case '*':
        case '/': return 2;
        case '+':
        case '-': return 1;
        default: return -1;
    }
}
void reverse(char *exp) {
    int len = strlen(exp);
    for (int i = 0; i < len/2; i++) {
        char temp = exp[i];
        exp[i] = exp[len - i -1];
        exp[len - i -1] = temp;
    }
}
void infixToPostfix(char* infix, char* postfix) {
    int i, k = 0;
    for (i = 0; infix[i]; i++) {
        char c = infix[i];
        if (isalnum(c)) {
            postfix[k++] = c;
        } else if (c == '(') {
            push(c);
        } else if (c == ')') {
            while (top != -1 && peek() != '(') {
                postfix[k++] = pop();
            }
            pop(); // Remove '('
        } else {
            while (top != -1 && precedence(peek()) >= precedence(c)) {
                postfix[k++] = pop();
            }
            push(c);
        }
    }
    while (top != -1) {
        postfix[k++] = pop();
    }
    postfix[k] = '\0';
}

void infixToPrefix(char* infix, char* prefix) {
    char revInfix[MAX], revPostfix[MAX];
    strcpy(revInfix, infix);
    reverse(revInfix);

    // Swap '(' and ')'
    for (int i = 0; revInfix[i]; i++) {
        if (revInfix[i] == '(') revInfix[i] = ')';
        else if (revInfix[i] == ')') revInfix[i] = '(';
    }

    // Convert reversed infix to postfix
    top = -1; // Reset stack top
    infixToPostfix(revInfix, revPostfix);

    // Reverse postfix to get prefix
    reverse(revPostfix);
    strcpy(prefix, revPostfix);
}

int main() {
    char infix[MAX], prefix[MAX];
    printf("Enter infix expression: ");
    scanf("%s", infix);
    infixToPrefix(infix, prefix);
    printf("Prefix expression: %s\n", prefix);
    return 0;
}
```

OUTPUT:

```
Infix Expression: a*(b+c/d)-e/f
Prefix Expression: -*a+b/cd/ef
```

Representation of infix to Postfix Expression (Reverse Polish notation) using stack:

Example

A + B * C – D => A + (B C *) – D

⇨ (A B C * +) – D

⇨ (A B C * + D -)

⇨ A B C * + D –

Algorithm:

1. Read all the symbols one by one from **left to right** in the given Infix Expression.
2. If the reading symbol is **operand**, then directly print it to the result (Output).
3. If the reading symbol is left parenthesis **'('**, then **Push it on to the Stack.**
4. If the reading symbol is right parenthesis **')'**, **pop the stack and print** the operators until left parenthesis is found.
5. if the reading symbol is the operator and the Stack is empty or contains the '(', ')' symbol, push the operator into the Stack.
6. If incoming symbol has **higher precedence than the top of the stack**, push it on the stack
7. If incoming symbol has **lower precedence than the top of the stack**, pop and print the top. Then test the incoming operator against the new top of the stack.
8. If incoming operator has **equal precedence with the top of the stack**, use associatively rule
   - ✓ If associatively **L to R** then pop and prints the top of the stack and then push the incoming operator.
   - ✓ If associatively **R to L** then push the incoming operator
9. At the end of the expression, pop and print all operators of stack.

NOTE:

- **+,- operators has same precedence**
- ***,/ operators has same precedence**

Example: 01- A + B * C – D

Input expression	Stack	Postfix expression
A		A
+	+	A
B	+	AB
*	+*	AB
C	+*	ABC
-	-	ABC*+
D	-	ABC*+D
		ABC*+D-

Now Postfix expression is ABC*+D-

Example: 02- (A + B) * (C - D)

Input expression	Stack	Postfix expression
(	(	
A	(	A
+	(+	A
B	(+	AB
)		AB+
*	*	AB+
(	*(	AB+
C	*(	AB+C
-	*(-	AB+C
D	*(-	AB+CD
)	*	AB+CD-
		AB+CD-*

Program to Convert Infix to Postfix notation

```
// C Program to Convert Infix to Postfix notation
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include <ctype.h>
#include <string.h>

char stack[100];
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;
return 0;
}
int main()
{
char exp[100];
char *e, x;
printf("Enter the Infix expression : "); scanf("%s",exp);
printf("\n");
printf("Postfix express is : ");
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:

```
Enter the Infix expression : A+B*C-D
Postfix express is : A B C * + D -
```

Evaluation of Postfix Expression (Reverse Polish notation) using Stack:

Algorithm:

Step-1: Create an empty STACK and start scanning the Postfix expression from Left to Right.

Step-2: if the element is an Operand, PUSH it into the STACK.

Step-3: if the element is an **Operator**, POP twice and get top most two elements A and B; Calculate **B Operator A** and PUSH the result back into the STACK.

Step-4: When the expression is ended, the value in the STACK is the final answer.

Example:01

i) 4 + 6 * 2 = > 4 + (6 2 *)
= (4 6 2 * +)
= 4 6 2 * +

Empty	Push(4)	Push(6)	Push(2)	Pop( )->A=2 Pop( )->B=6 B*A = 6*2=12	Push(12)	Pop( ) ->A=12 Pop( ) ->B=4 B+A = 4+12=16	Push(16)
			2				
		6	6		12		
	4	4	4	4	4		16
Stack							

Therefore, the final answer of postfix notation 4 6 2 * + is **16**

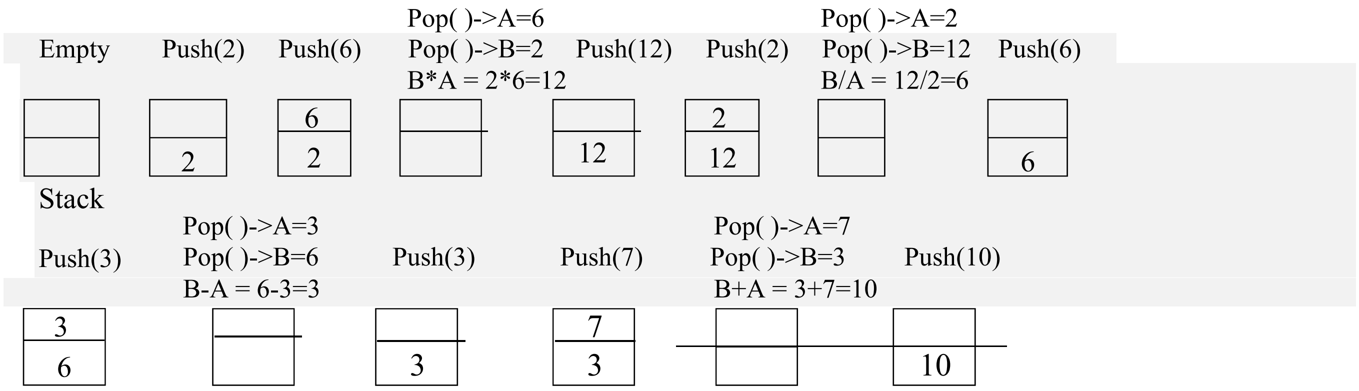
ii) 2 + 3 * 4 – 5 => 2 + (3 4 *) - 5
= (2 3 4 * +) – 5
= (2 3 4 5 * + -)
= 2 3 4 * + 5 –

Empty	Push(2)	Push(3)	Push(4)	Pop( )->A=4 Pop( )->B=3 B*A = 4*3=12	Push(12)	Pop( )->A=12 Pop( )->B=2 B+A = 2+12=14	Push(14)	Push(5)
			4					
		3	3		12			5
	2	2	2	2	2		14	14
Stack								

Pop( )->A=5 Pop( )->B=14 B-A = 14-5=9	Push(9)
	9

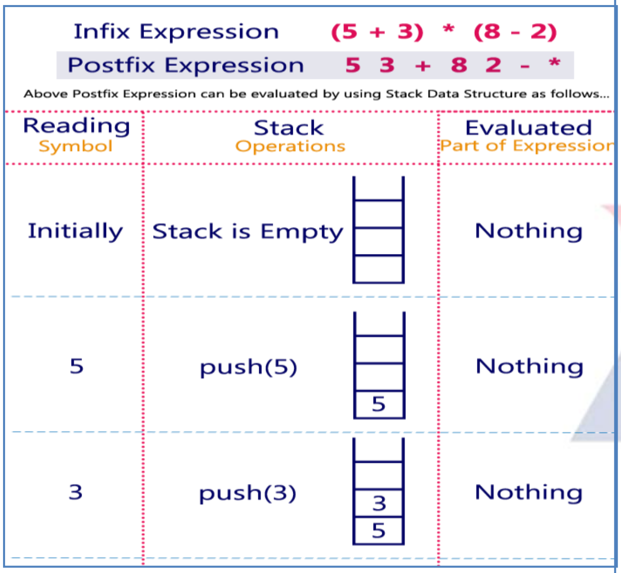
Therefore, the final answer of postfix notation 2 3 4 5 * + - is **9**

iii) 2 * 6 /2 – 3 + 7 => (2 6 *) / 2 – 3 + 7
= (2 6 * 2 /) – 3 + 7
= (2 6 * 2 / 3 -) + 7
= (2 6 * 2 / 3 - 7 +)
= 2 6 * 2 / 3 – 7 +



Therefore, the final answer of postfix notation 2 6 * 2 / 3 – 7 + is **10**

Example-02- **(5+3)*(8-2)**

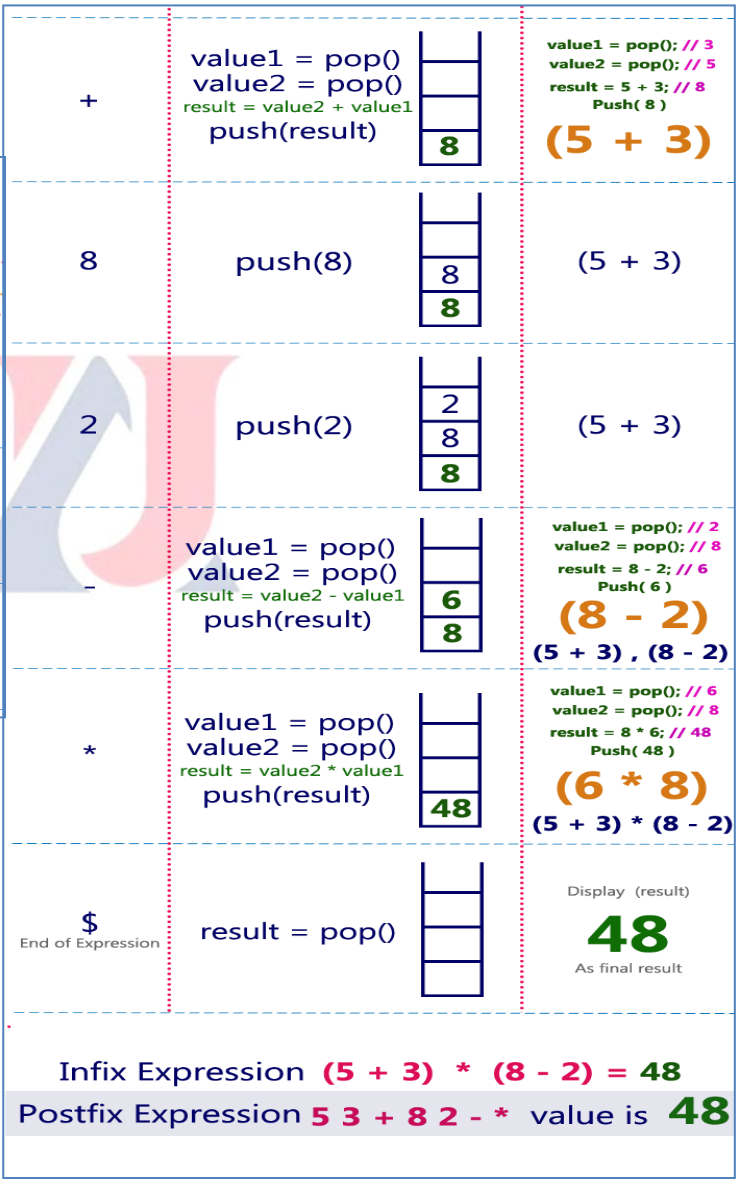


Infix Expression (5 + 3) * (8 - 2)

Postfix Expression 5 3 + 8 2 - *

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations	Evaluated Part of Expression
Initially	Stack is Empty	Nothing
5	push(5) 5	Nothing
3	push(3) 3 5	Nothing



+	value1 = pop()
	value2 = pop()
	result = value2 + value1
	push(result)
	8
		value1 = pop(); // 3
		value2 = pop(); // 5
		result = 5 + 3; // 8
		Push(8)
		(5 + 3)

8	push(8) 8 8	(5 + 3)
2	push(2) 2 8 8	(5 + 3)

-	value1 = pop()
	value2 = pop()
	result = value2 - value1
	push(result)
	6
	8
		value1 = pop(); // 2
		value2 = pop(); // 8
		result = 8 - 2; // 6
		Push(6)
		(8 - 2)
		(5 + 3) , (8 - 2)

*	value1 = pop()
	value2 = pop()
	result = value2 * value1
	push(result)
	48
		value1 = pop(); // 6
		value2 = pop(); // 8
		result = 8 * 6; // 48
		Push(48)
		(6 * 8)
		(5 + 3) * (8 - 2)

$ End of Expression	result = pop()	Display (result) 48 As final result

Infix Expression (5 + 3) * (8 - 2) = 48

Postfix Expression 5 3 + 8 2 - * value is 48

Program to evaluate Postfix notation

```
                    // c program to evaluate postfix notation
#include<stdio.h>
#include<ctype.h>
 #include<stdlib.h>
int stack[20]; int top=-1;
void push(int x)
{
stack[++top]=x;
}
int pop()
{
return stack[top--];
}
void main()
{
char exp[20];
char *e;
int n1,n2,n3,num;
printf("\nEnter the expression :");
scanf("%s",exp);
e=exp;
while(*e != '\0')
{
        if(isdigit(*e))
        {
                num = *e-48;
                 push(num);
        }
        else
        {
                n1=pop();
                n2=pop();
                switch(*e)
                {
                case '+':
                {
                n3=n1+n2; break;
                }
                case '-':
                {
                n3=n2-n1; break;
                }
                case '*':
                {
                n3=n1*n2; break;
                }
                case '/':
                {
                n3=n2/n1; break;
        } }
        push(n3);
}
 e++;
}
printf("\nThe result of expression %s = %d \n ",exp, pop()); getch( ); }
```

Towers of Hanoi

- Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted.
- These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one.
- There are other variations of the puzzle where the number of disks increase, but the tower count remains the same.

Rules:

The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are

- ✓ Only one disk can be moved at a time.
- ✓ Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
- ✓ No disk may be placed on top of a smaller disk.

Tower of Hanoi using Recursion:

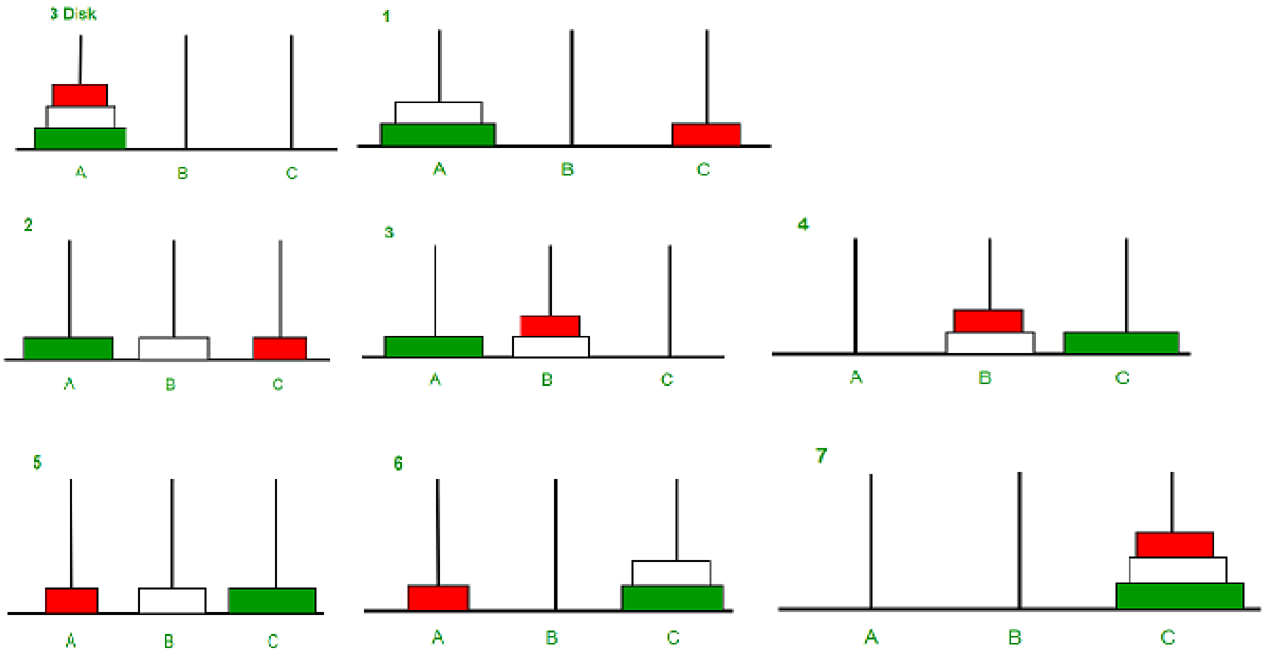
The idea is to use the helper node to reach the destination using recursion. Below is the pattern for this problem:

Example: Input: N = 3,

Step 1 – Move N-1 disks from A to B, using C.

Step 2 – Move Last (Nth) disk from A to C.

Step 3 – Move N-1 disks from B to C, using A



A recursive algorithm

```
START
Procedure Hanoi(disk, source, dest, aux)
  IF disk == 1, THEN
    move disk from source to dest
  ELSE
    Hanoi(disk - 1, source, aux, dest)     // Step 1
    move disk from source to dest          // Step 2
    Hanoi(disk - 1, aux, dest, source)     // Step 3
  END IF
END Procedure
STOP
```

Tower of Hanoi puzzle with n disks can be solved in minimum 2^n-1 steps. This presentation shows that a puzzle with 3 disks has taken $2^3 - 1 = 7$ steps.

Following are the implementations of this approach

```
#include <stdio.h>
// Recursive function to solve Tower of Hanoi
void TH(int n, char source, char auxiliary, char destination) {
  if (n == 1) {
                printf("Move disk 1 from %c to %c\n", source, destination);
    return;
  }
  // Move n-1 disks from source to auxiliary
  TH(n - 1, source, destination, auxiliary);
  // Move the nth disk from source to destination
  printf("Move disk %d from %c to %c\n", n, source, destination);
  // Move n-1 disks from auxiliary to destination
  TH(n - 1, auxiliary, source, destination);
}
int main() {
  int n;
  printf("Enter number of disks: ");
  scanf("%d", &n);
  // Call the function with Source = A, Auxiliary = B, Destination = C
   TH(n, 'A', 'B', 'C');
  return 0;
}
```

Output:

```
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
```

Question: Recursive Implementation of Tower of Hanoi (4 Disks)

You are given the classic **Tower of Hanoi** problem with **4 disks** and **3 pegs(rings)** labeled A, B, and C. The objective is to move all 4 disks from **peg A (source)** to **peg C (target)** using **peg B (auxiliary)**, following these rules:

1. Only one disk can be moved at a time.
2. A disk can only be placed on an empty peg or on top of a larger disk.
3. You must use recursion to solve the problem.

Task:

Write a recursive C function named tower_of_hanoi that prints each move required to solve the problem.

References:

- https://dotnettutorials.net/lesson/tower-of-hanoi-using-recursion-in-c/
- https://understanding-recursion.readthedocs.io/en/latest/17%20Hanoi.html

Parenthesis checker

- The parentheses are used to represent the mathematical representation.
- The balanced parenthesis means that when the opening parenthesis is equal to the closing parenthesis, then it is a balanced parenthesis.

The parenthesis is represented by the brackets shown below:

( ) , { }, [ ]

Where, (→ Opening bracket and

) → Closing bracket

Example 1: 2 * ((4/2) + 5)

The above expression has two opening and two closing brackets which means that the above expression is a balanced parenthesis.

Example 2: 2 * ((4/2) + 5

The above expression has two opening brackets and one closing bracket, which means that both opening and closing brackets are not equal; therefore, the above expression is unbalanced.

Algorithm to check balanced parenthesis:

Now, we will check the balanced parenthesis by using a variable. The variable is used to determine the balance factor. Let's consider the variable 'x'. The algorithm to check the balanced parenthesis is given below:

Step 1: Set x equal to 0.

Step 2: Scan the expression from left to right.

For each opening bracket "(", increment x by 1.

For each closing bracket ")", decrement x by 1.

This step will continue scanning until x<0.

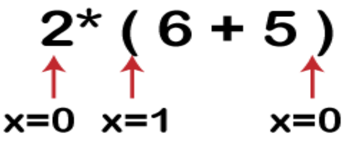
Step 3: If x is equal to 0, then "Expression is balanced."

Else

"Expression is unbalanced."

Let's understand the above algorithm through an example.

Suppose expression is 2 * (6 + 5)



Solution:

- First, the x variable is initialized by 0.
- The scanning starts from the variable '2', when it encounters '(' then the 'x' variable gets incremented by 1 and when the x reaches to the last symbol of the expression, i.e., ')' then the 'x' variable gets decremented by 1 and it's final value becomes 0.
- We have learnt in the above algorithm that if x is equal to 0 means the expression is balanced; therefore, the above expression is a balanced expression.

Implementation of Parenthesis checker

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_SIZE 100

// Global variables for stack and top
char stack[MAX_SIZE];
int top = -1;

// Function to push a character onto the stack
void push(char data) {
   if (top == MAX_SIZE - 1) {
      printf("Overflow stack!\n");
      return;
   }
   top++;
   stack[top] = data;
}

// Function to pop a character from the stack
char pop() {
   if (top == -1) {
      printf("Empty stack!\n");
      return ' ';
   }
   char data = stack[top];
   top--;
   return data;
}
```

```
// Function to check if two characters form a matching pair of parentheses
int is_matching_pair(char char1, char char2) {
    if (char1 == '(' && char2 == ')') {
        return 1;
    } else if (char1 == '[' && char2 == ']') {
        return 1;
    } else if (char1 == '{' && char2 == '}') {
        return 1;
    } else {
        return 0;
    }
}
// Function to check if the expression is balanced
int isBalanced(char* text) {
    int i;
    for (i = 0; i < strlen(text); i++) {
        if (text[i] == '(' || text[i] == '[' || text[i] == '{') {
            push(text[i]);
        } else if (text[i] == ')' || text[i] == ']' || text[i] == '}') {
            if (top == -1) {
                return 0; // If no opening bracket is present
            } else if (!is_matching_pair(pop(), text[i])) {
                return 0; // If closing bracket doesn't match the last opening bracket
            }
        }
    }
    if (top == -1) {
        return 1; // If the stack is empty, the expression is balanced
    } else {
        return 0; // If the stack is not empty, the expression is not balanced
    }
}

// Main function
int main() {
    char text[MAX_SIZE];
    printf("Input an expression in parentheses: ");
    scanf("%s", text);

    // Check if the expression is balanced or not
    if (isBalanced(text)) {
        printf("The expression is balanced.\n");
    } else {
        printf("The expression is not balanced.\n");
    }
    return 0;
}
```

OUTPUT:

```
Input an expression in parentheses: (([{}])
The expression is not balanced.

Input an expression in parentheses: (([{}]))
The expression is balanced.

Input an expression in parentheses: (A[B{C+D}])
The expression is balanced.
```

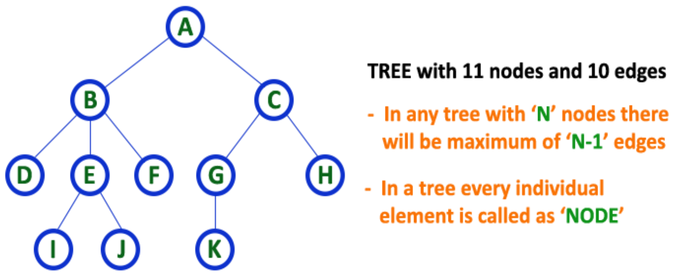
TREES

Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.

A tree data structure can also be defined as follows...

A **tree** is a finite set of **nodes** together with a finite set of directed **edges** (links/branches) that define parent-child (*HIERARCHICAL*) relationships. **Tree** isa non-linear data structure

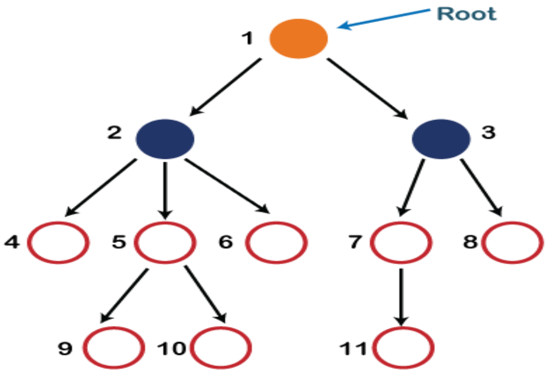
Example:



TREE with 11 nodes and 10 edges

- In any tree with 'N' nodes there will be maximum of 'N-1' edges

- In a tree every individual element is called as 'NODE'



A tree satisfies the following properties:

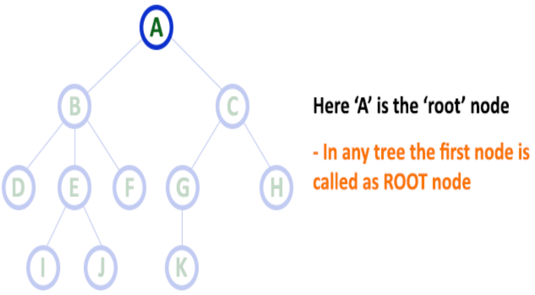
- It has one designated node, called the root that has no parent.
- Every node, except the root, has exactly one parent.
- A node may have zero or more children.
- There is a unique directed path from the root to each node.

Terminology:

In a tree data structure, we use the following terminolo

1. **Root: Root:**

- In a tree data structure, the first node is called as **Root Node.**
- Every tree must have a root node. We can say that the root node is the **origin** of the tree data structure.

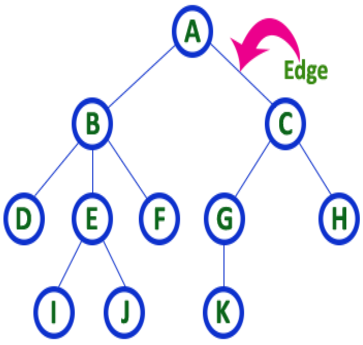


Here 'A' is the 'root' node

- In any tree the first node is called as ROOT node

2. **Edge:**

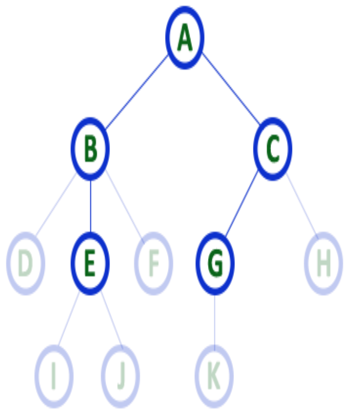
- In a tree data structure, the connecting link between any two nodes is called as **EDGE.**
- In a tree with '**N**' number of nodes there will be a maximum of '**N-1**' number of edges.



- In any tree, 'Edge' is a connecting link between two nodes.

3. Parent:

- In a tree data structure, the node which is a predecessor of any node is called as **PARENT NODE**.
- In simple words, the node which has a branch from it to any other node is called a parent node.
- Parent node can also be defined as **"The node which has child / children"**

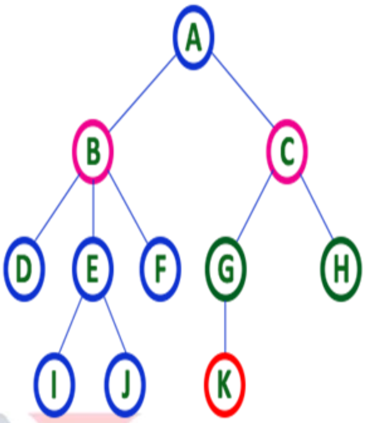


Here A, B, C, E & G are Parent nodes

- In any tree the node which has child / children is called 'Parent'
- A node which is predecessor of any other node is called 'Parent'

4. Child:

- In a tree data structure, the node which is descendant of any node is called as **CHILD Node**.
- In simple words, the node which has a link from its parent node is called as child node.
- In a tree, any parent node can have any number of child nodes.
- In a tree, all the nodes except root are child nodes.



Here B & C are Children of A

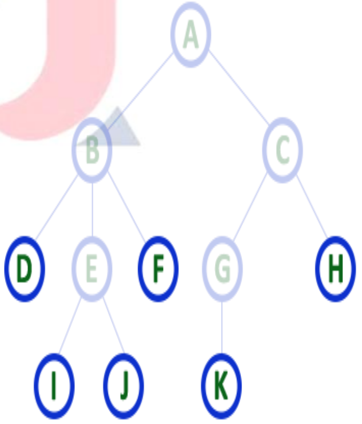
Here G & H are Children of C

Here K is Child of G

- descendant of any node is called as CHILD Node

5. Leaf:

- In a tree data structure, the node which does not have a child is called as **LEAF Node**.
- In simple words, a leaf is a node with no child.
- In a tree data structure, the leaf nodes are also called as **External Nodes**.
- External node is also a node with no child.
- In a tree, <u>leaf node is also called as **'Terminal'** node.</u>

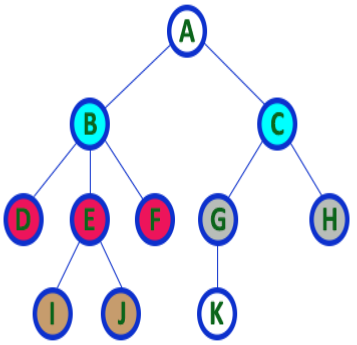


Here D, I, J, F, K & H are Leaf nodes

- In any tree the node which does not have children is called 'Leaf'
- A node without successors is called a 'leaf' node

6. Siblings:

- In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**.
- In simple words, the nodes with the same parent are called Sibling nodes.



Here B & C are Siblings

Here D E & F are Siblings

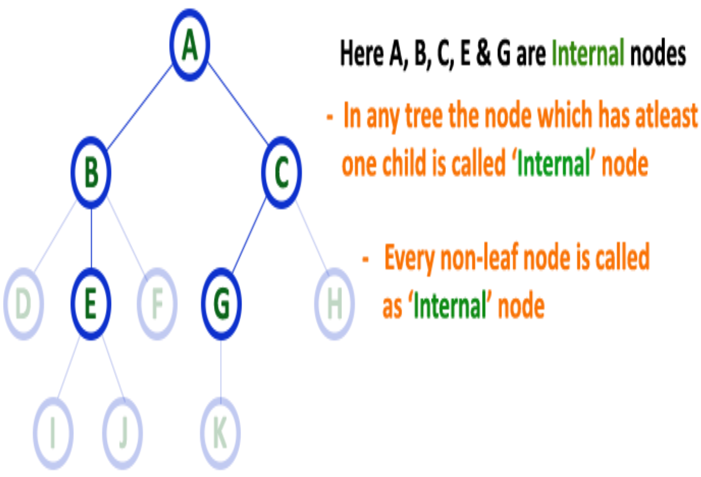
Here G & H are Siblings

Here I & J are Siblings

- In any tree the nodes which has same Parent are called 'Siblings'
- The children of a Parent are called 'Siblings'

7 Internal Nodes:

- In a tree data structure, the node which has at least one child is called as **INTERNAL Node**.
- In simple words, an internal node is a node with at least one child.
- In a tree data structure, nodes other than leaf nodes are called as **Internal Nodes.**
- **The root node is also said to be Internal Node** if the tree has more than one node. Internal nodes are also called as **'Non-Terminal'** nodes.



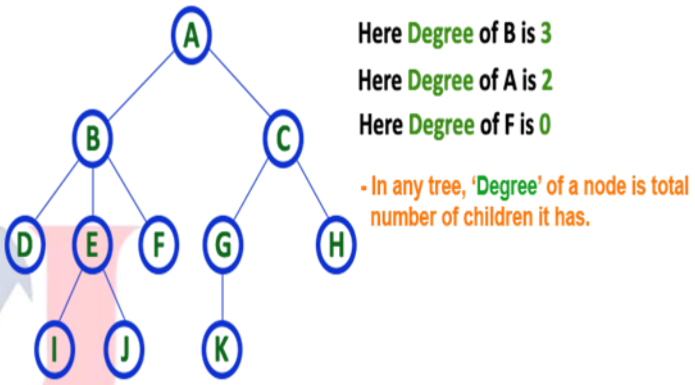
Here A, B, C, E & G are Internal nodes

- In any tree the node which has atleast one child is called 'Internal' node

- Every non-leaf node is called as 'Internal' node

8. Degree:

- In a tree data structure, the total number of children of a node is called as **DEGREE** of that Node.
- In simple words, the Degree of a node is total number of children it has.
- The highest degree of a node among all the nodes in a tree is called as '**Degree of Tree**'.



Here Degree of B is 3

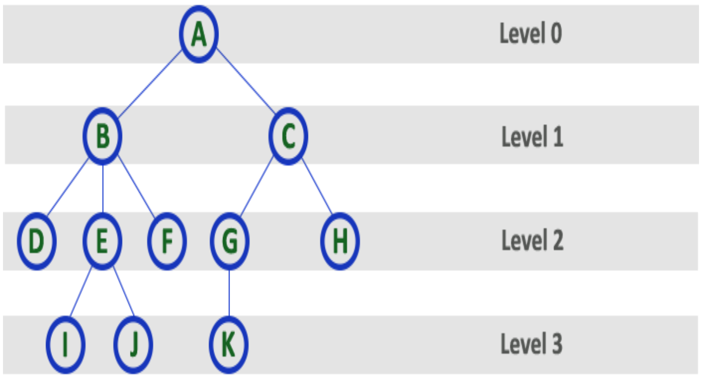
Here Degree of A is 2

Here Degree of F is 0

- In any tree, 'Degree' of a node is total number of children it has.

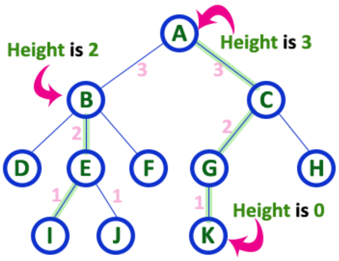
9. Level:

- In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on...
- In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



10. Height:

- In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as **HEIGHT** of that Node.



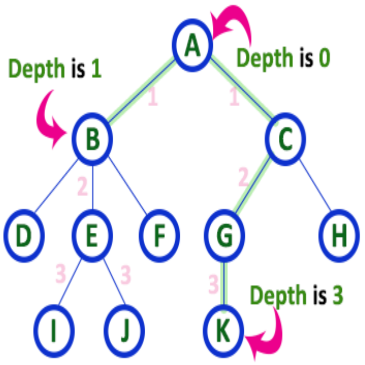
Here Height of tree is 3

- In any tree, 'Height of Node' is total number of Edges from leaf to that node in longest path.

- In any tree, 'Height of Tree' is the height of the root node.

11. Depth:

- In a tree data structure, the total number of edges **from root node** to a particular node is called as **DEPTH** of that Node.
- In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, **depth of the root node is '0'.**

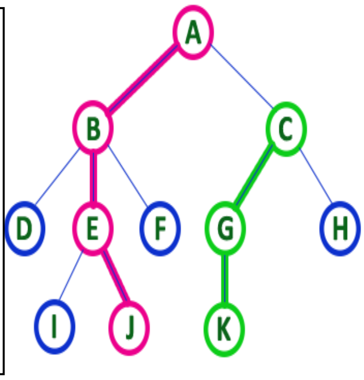


Here Depth of tree is 3

- In any tree, 'Depth of Node' is total number of Edges from root to that node.
- In any tree, 'Depth of Tree' is total number of edges from root to leaf in the longest path.

12. Path:

- In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as **PATH** between those two Nodes.
- **Length of a Path** is total number of nodes in that path. In below example **the path A - B - E - J has length 4**.



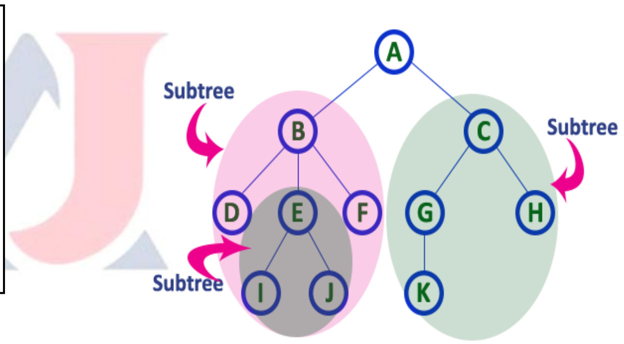
- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

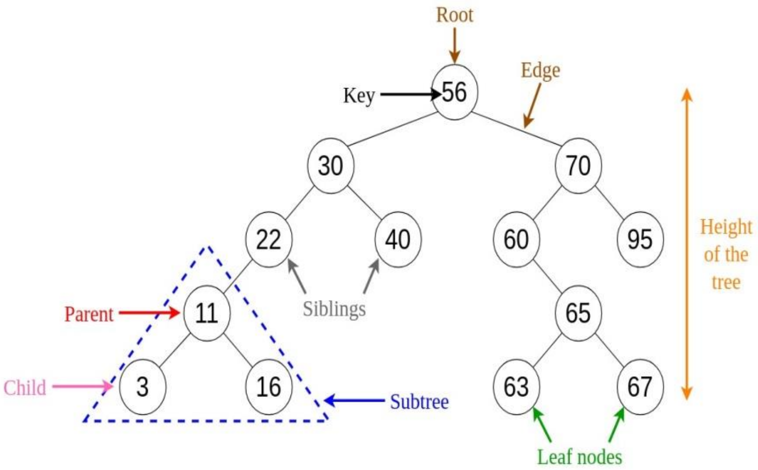
Here, 'Path' between A & J is
A - B - E - J

Here, 'Path' between C & K is
C - G - K

13. Sub Tree:

In a tree data structure, each child from a node forms a sub tree recursively. Every child node will form a sub tree on its parent node.





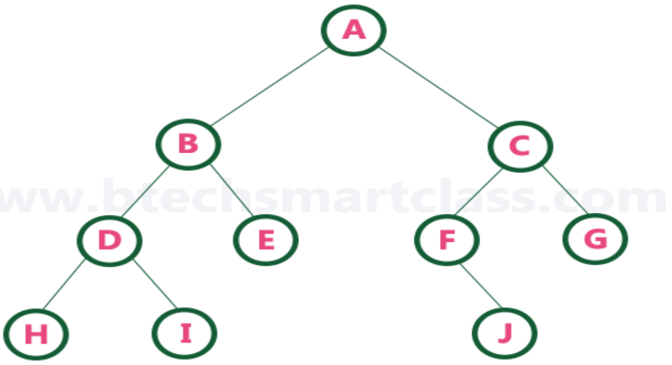
BINARY TREE

In a normal tree, every node can have any number of children. A binary tree is a special type of tree data structure in which every node can have a **maximum of 2 children**. One is known as a left child and the other is known as right child.

A tree in which every node can have a maximum of two children is called Binary Tree.

In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.

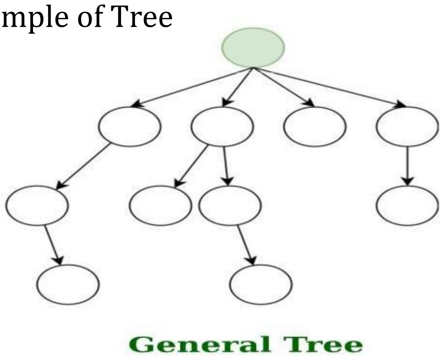
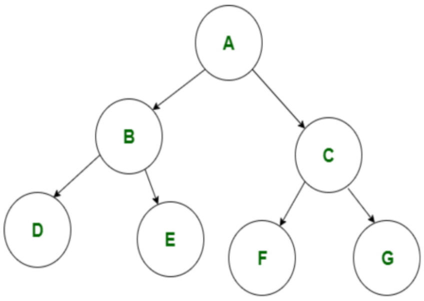
Example:



Binary Tree Properties:

- ✓ A binary tree with n elements, n > 0, has exactly ***N*-1 edges**.
- ✓ A binary tree of height *H*, *H* >= 0, has at least *h* and at most 2^{h+1}-1 elementsor nodes in it.
- ✓ The height of a binary tree that contains *N* elements, *N* >= 0, is at least(*log2*(*n*+1)) and at most *n.*

Difference between tree and binary tree

Tree	Binary Tree
Tree never be empty	Binary tree may be empty
A node may have any no of	A node may have at most nodes / children's.
An example of Tree  	An example of Binary Tree 

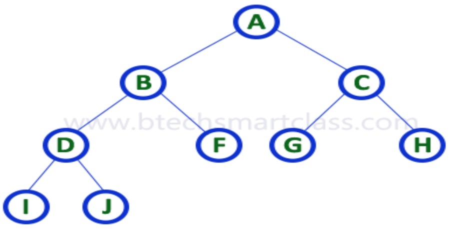
There are different types of binary trees and they are

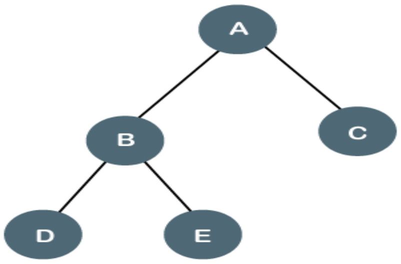
1. Full Binary (Strictly Binary) Tree:

A strictly Binary Tree can be defined as follows...

A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree

Strictly binary tree is also called as **Full Binary Tree** or **Proper Binary Tree** or **2-Tree**



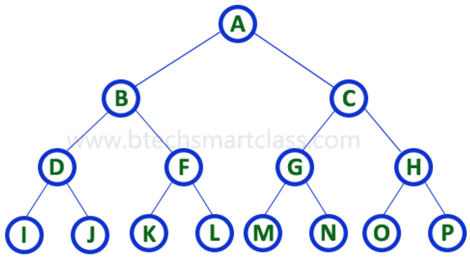


2. Complete Binary Tree:

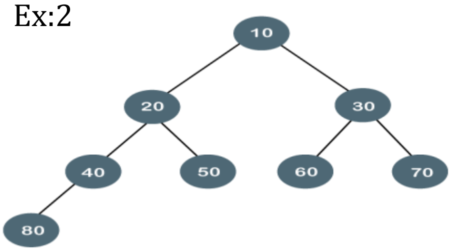
- In complete binary tree all the nodes must have exactly two children (**except the last level**) and at every level of complete binary tree there must be **2^{level}** number of nodes.
- For example at level 2 there must be $2^2 = 4$ nodes and at level 3 there must be $2^3 = 8$ nodes.

A complete binary tree is a binary tree in which every level is completely filled except possibly the last level. In the unfilled level, the nodes are attached starting from the left-most position

Ex:1



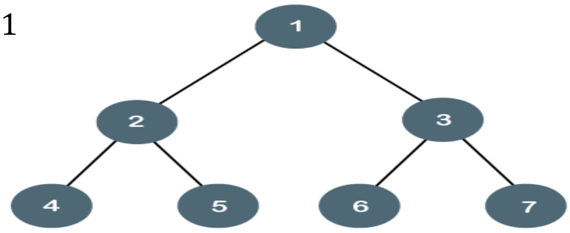
Ex:2



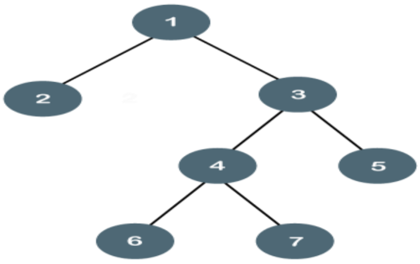
3. Perfect Binary Tree:

A perfect binary tree is a binary tree in which all interior nodes have two children and all leaves have the same depth or same level.

Ex:1



Ex:2

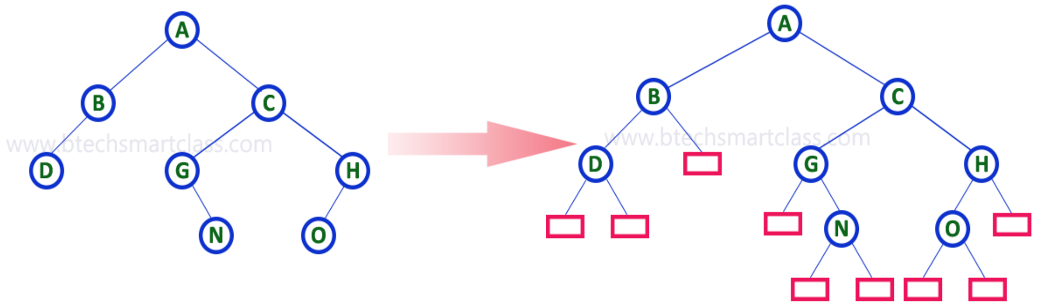


Perfect binary tree **Not a perfect binary tree**

4. Extended Binary Tree:

A binary tree can be converted into Full Binary tree by adding dummy nodes to existing nodes wherever required.

The full binary tree obtained by adding dummy nodes to a binary tree is called as Extended Binary Tree.



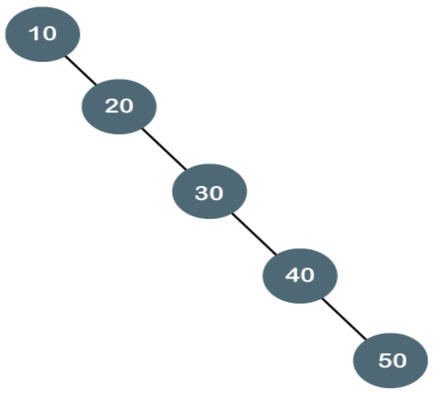
In above figure, a normal binary tree is converted into full binary tree by adding dummy nodes (In pink colour).

5. Degenerate Binary Tree:

The degenerate binary tree is a tree in which all the internal nodes have only one children.

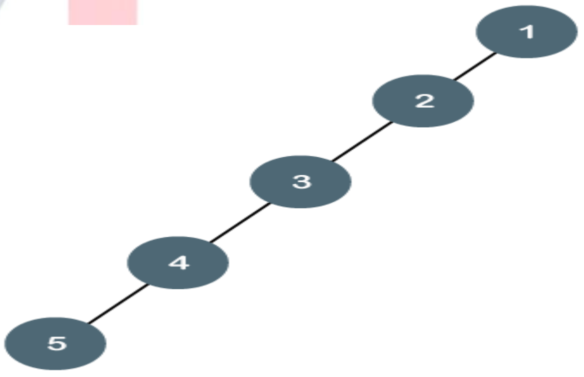
Let's understand the Degenerate binary tree through examples.

Ex:-1



right-skewed

Ex:-2



left-skewed

- ➢ The above **Ex-1** tree is a degenerate binary tree because all the nodes have only one child. It is also known as a **right-skewed** tree as all the nodes have a right child only.
- ➢ The above **Ex-2** tree is also a degenerate binary tree because all the nodes have only one child. It is also known as a **left-skewed** tree as all the nodes have a left child only.

Binary Tree Representations

A binary tree data structure is represented using two methods. Those methods are as follows...

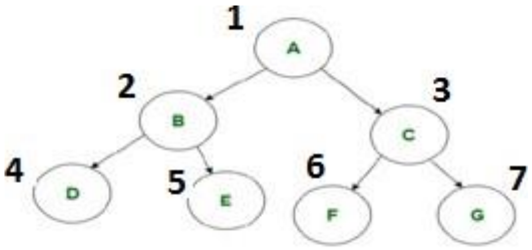
a) Array Representation

b) Linked List Representation

Representation of Binary Trees using Arrays

- The sequential representation uses an array for the storage of tree elements (nodes).
- The number of nodes a binary tree has defines the size of the array being used.
- The ROOT node of the binary tree lies at the array's first index.
- The index at which a particular node is stored will define the indices at which the right and left children of the node will be stored. An empty tree has NULL or ZERO as its first index

Example-1:



Binary Tree

Height of the given Binary Tree,**h = 2**

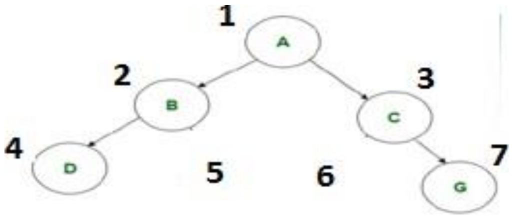
Total number of nodes of given binary tree = $\mathbf{2^{h+1} - 1}$
$= 2^{2+1} - 1$
$= 2^3 - 1$
$= 8-1 = 7$

Therefore the size of Array = **7** (i.e., Maximum number of nodes)

1	2	3	4	5	6	7
A	B	C	D	E	F	G

Array representation

Example-2:



Binary Tree

Height of the given Binary Tree,**h = 2**

Total number of nodes of given binary tree = $\mathbf{2^{h+1}- 1}$
$= 2^{2+1} - 1$
$= 2^3 - 1$
$= 8-1 = 7$

Therefore the size of Array = 7 (i.e., Maximum number of nodes)

1	2	3	4	5	6	7
A	B	C	D	----	-----	G

Advantages:

- **Static data storage**: Direct access to any node is possible due its static memory allocation.
- **Random Access**: To find the parent, left child or right child of any particular node is fast because of random access.

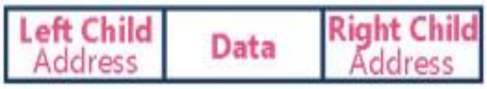
Disadvantages:

- **Memory Wastage**: wastage of memory due to its memory allocation for missing elements.
- **Implementation**: It depends on height of the tree and as well as total nodes of the tree.
- **Insertion and Deletion operation**: The insertion and deletion of any node in the tree will be costlier (difficult) as other nodes have to be adjusted at appropriate positions.

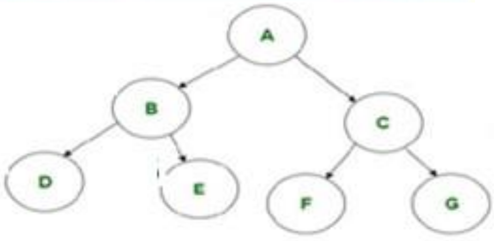
Representation of Binary Trees using Linked List

- Binary trees in linked representation are stored in the memory as linked lists. These lists have nodes that aren't stored at adjacent or neighboring memory locations and are linked to each other through the parent-child relationship associated with trees.
- We use a double linked list to represent a binary tree.
- In a double linked list, everynode consists of **three fields**;
  - ✓ First field for storing left child address,
  - ✓ Second for storing actual data and
  - ✓ Third for storing right child address.

In this linked list representation, a node has the following structure:

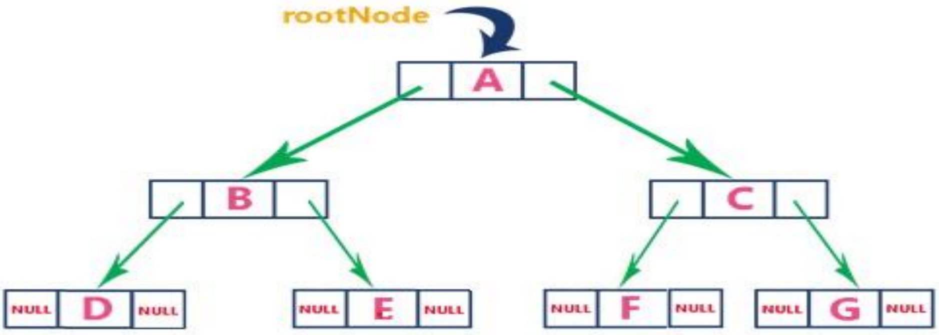


Example:1

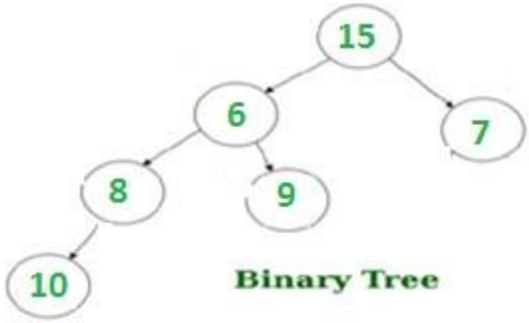


Binary Tree

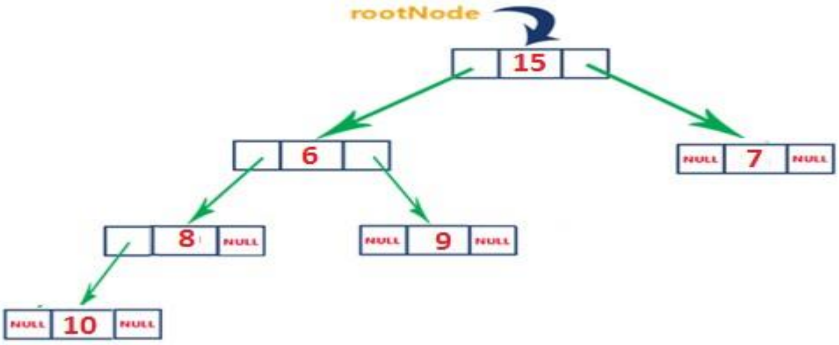
The above example of the binary tree represented using Linked list representation is shown as follows...



Example-2:



The above example of the binary tree represented using Linked list representation is shown as follows...



Advantages:

- **Dynamic data storage**: it can grow and shrink at runtime by allocating and deallocating memory.
- **No Memory Wastage**: Efficient memory utilization can be achieved since the size of the linked list increase or decrease at runtime.
- **Insertion and Deletion operation**: Insertion and deletion operations are quite easier in the linked list. There is no need to shift elements after the insertion or deletion of an element only the address present in the next pointer needs to be update.

Disadvantages:

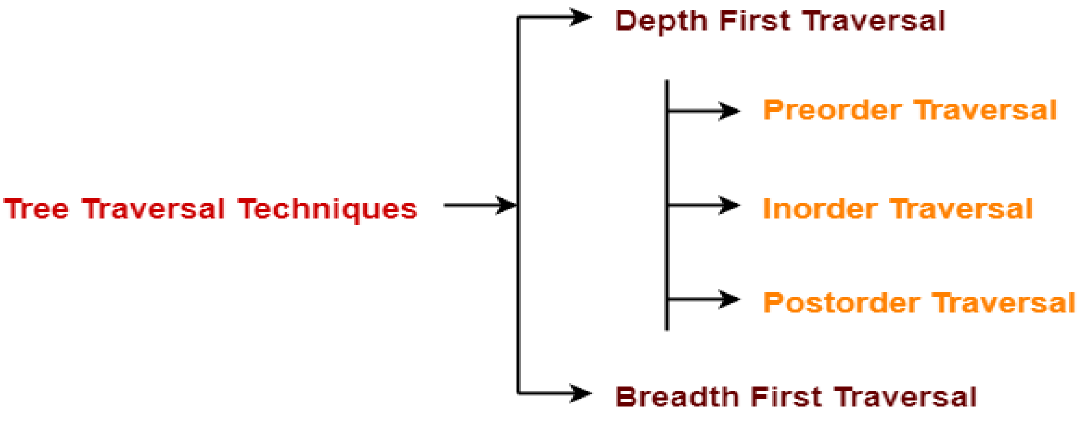
- **Memory Usage**: Extra memory is required in the **linked list** as compared to an array. Because in a linked list, **a pointer is also required** to store the address of the next element and it require extra memory for itself.
- **Traversal:** Direct or random accessing to an element is not possible in a linked list due to its dynamic memory allocation. That means extra memory is used for pointers with every element

Binary Tree Traversal

When we wanted to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree, displaying order of nodes depends on the traversal method.

Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.

There are three types of binary tree traversals.



1. In-order Traversal: (left Child - root - right Child)

In this method, the left sub-tree (node) is visited first, and then the ROOT node is visited and later to visited the right sub-tree (node). If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order. We should always remember that every node may represent a sub-tree itself.

Left Sub-tree (node)→ ROOT node → Right Sub-tree (node)

Algorithm:

Until all nodes are traversed (visited)

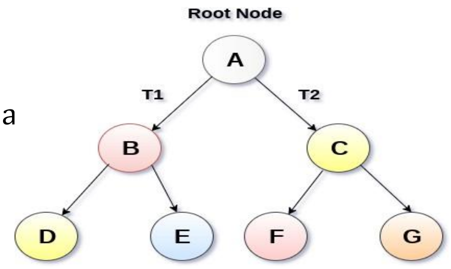
Step-1: Recursively traverse Left sub-tree // inorder(root->left)

Step-2: Visit ROOT node // display(root->data)

Step-3: Recursively traverse Left sub-tree //inorder(root->right)

Example-1:

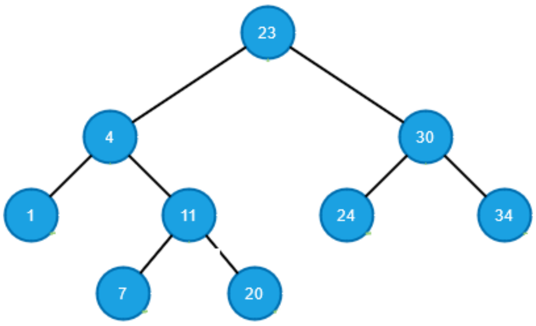
- ✓ We start from A, and following in-order traversa
- ✓ We move to its left sub-tree first; that is B.
- ✓ B is also traversed in-order.
- ✓ The process goes on until all nodes are visited.



The In-order traversal of the above binary tree is

D → B → E → A → F → C → G

Example-2:



- ✓ We start from 23, and following in-order traversal.
- ✓ We move to its left sub-tree first; that is 4.
  - ✓ **4 is also traversed in-order.**
- ✓ The process goes on until all nodes are visited.

The In-order traversal of the above binary tree is

1 → 4 → 7 → 11 → 20 → 23 → 24 → 30 → 34

2. Pre - Order Traversal (root – left Child – right Child):

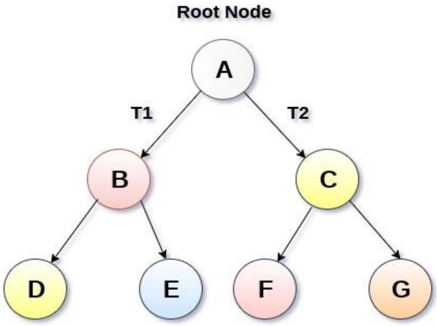
In this method, the ROOT node is visited first, and then the leftsub-tree (node) is visited and later to visited the right sub-tree (node).

ROOT node →Left Sub-tree (node)→ Right Sub-tree (node)

Algorithm:

```
Until all nodes are traversed (visited)
Step-1: Visit ROOT node // display(root->data)
Step-2: Recursively traverse Left sub-tree // inorder(root->left)
Step-3: Recursively traverse Right sub-tree //inorder(root->right)
```

Example-1:

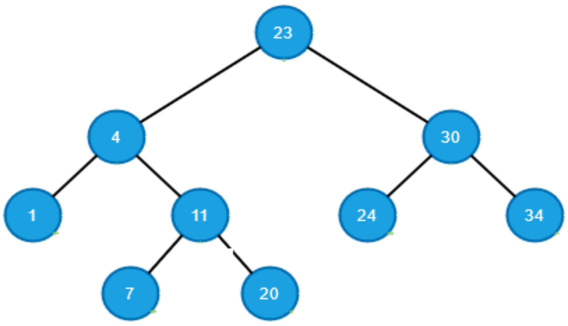


- → We start from A, and following pre-order traversal.
- → We move to its left sub-tree first; that is B.
- → **B is also traversed pre-order.**
- → The process goes on until all nodes are visited.

The pre-order traversal of the above binary tree is

A→ B → D→ E→ C→ F→ G

Example-2:



- → We start from 23, and following pre-order traversal
- → We move to its left sub-tree first; that is 4.
- → 4 is also traversed pre-order.
- → The process goes on until all nodes are visited

The pre-order traversal of the above binary tree is

23→ 4 → 1→ 11 → 7→ 20→ 30→24→ 34

3. Post - Order Traversal (left Child – right Child - root):

In this method, the ROOT node is visited first, and then the leftsub-tree(node) is visited and later to visited the right sub-tree(node).

Left Sub-tree (node)→ Right Sub-tree (node)→ ROOT node

Algorithm:

Until all nodes are traversed (visited)

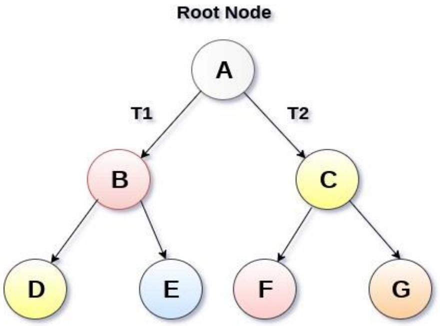
Step-1: Recursively traverse Left sub-tree // inorder(root->left)

Step-2: Recursively traverse Right sub-tree //inorder(root->right)

Step-3: Visit ROOT node // display(root->data)

Example-1:

- ➔ We start from A, and following post-order traversal.
- ➔ We move to its left sub-tree first; that is B.
- ➔ **B is also traversed post-order.**
- ➔ The process goes on until all nodes are visited.

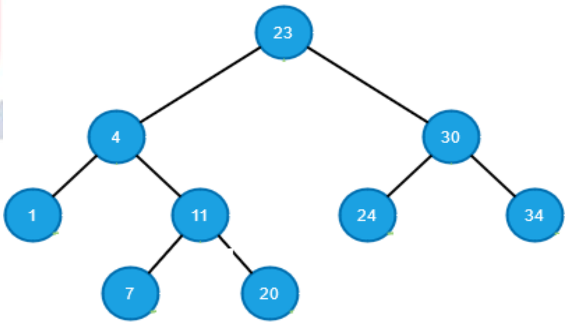


The post-order traversal of the above binary tree is

D→ E→ B→ F→ G→ C→ A

Example-2:

- ➔We start from 23, and following post-order traversal.
- ➔ We move to its left sub-tree first; that is 4.
- ➔ **4 is also traversed post-order.**
- ➔ The process goes on until all nodes are visited.

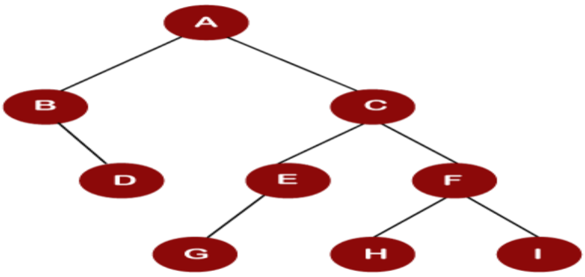


The post-order traversal of the above binary tree is

1→ 7→ 20→ 11 →4→ 24→ 34→30→ 23

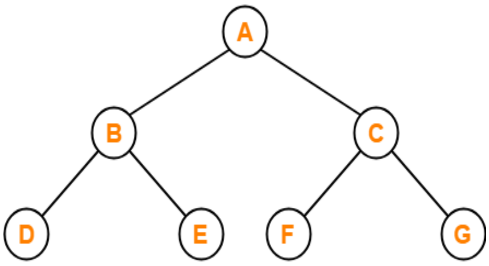
Practice Examples:

Example-3: Consider the following binary tree



- ➢ **In - Order Traversal : B, D, A, G, E, C, H, F, I.**
- ➢ **Pre - Order Traversal: A, B, D, C, E, G, F, H, I**
- ➢ **Post - Order Traversal: D, B, G, E, H, I, F, C, A**

Example-4: Consider the following binary tree



1. **In - Order Traversal :D,B,E,A,F,C,G**
2. **Pre - Order Traversal : A,B,D,E,C,F,G**
3. **Post - Order Traversal : D,E,B,F,G,C,A**

Construct a Binary Tree using In-order and Post-order Tree traversals

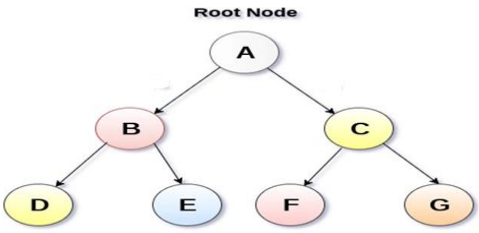
Algorithm:

1. Find the root node using the post-order traversal.
2. Find the left sub trees and the right sub trees using in-order traversal by finding the index of the root node of respective sub trees.
3. Once the root node is found, we can recourse down on the left and right sub trees, i.e., left sub array, and right sub array split at respective root index to repeat the same process until we find at most a single element in either sub-array.

Example-1: To Construct a binary tree using the following In-order and Post-order traversal of the binary tree:

In-order: D→ B→ E→ A →F→ C→ G

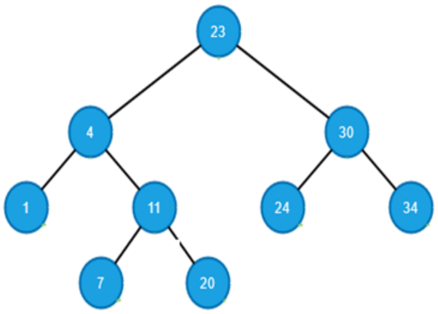
Post-order: D→ E→ B→ F →G→ C→ A



Example-2: To Construct a binary tree using the following In-order and Post-order traversals of the binary tree:

In-order: 1→ 4→ 7→ 11 →20→ 23→ 24→ 30→ 34

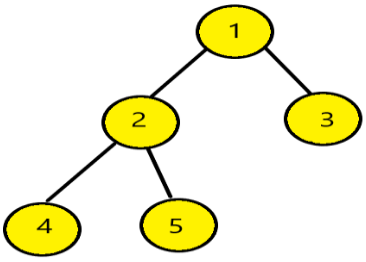
Post-order: 1→ 7→ 20→ 11 →4→ 24→ 34→ 30→ 23



Example:3

Inorder Traversal: [4, 2, 5, 1, 3]

Postorder Traversal: [4, 5, 2, 3, 1]



Construct a Binary Tree using In-order and Pre-order Tree traversals

Algorithm:

1. Find the root node using the pre-order traversal.
2. Find the left sub trees and the right sub trees using in-order traversal by finding the index of the root node of respective sub trees.
3. Once the root node is found, we can recurse down on the left and right sub trees, i.e., left sub array, and right sub array split at respective root index to repeat the same process until we find at most a single element in either sub-array

Example-1: Construct binary tree for given preorder and inorder traversals.

Inorder: 2-3-4-5-6-8-10
Preorder: 5-3-2-4-8-6-10

- As we know that preorder visits the root node first then the first value always represents the root of the tree. From above sequence 5 is the root of the tree.

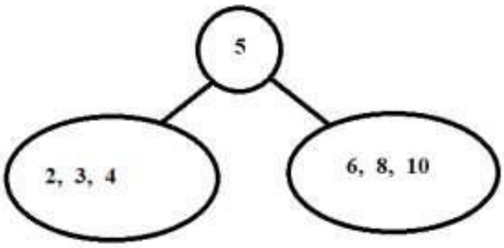
Preorder

5 -3-2-4-8-6-10

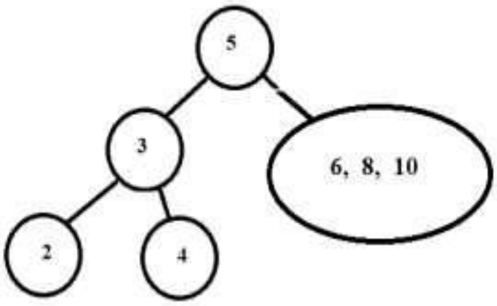
- From above inorder traversal, we know that a node's left subtree is traversed before it followed by its right subtree. Therefore, all values to the left of 5 in inorder belong to its left subtree and all values to the right belong to its right subtree.

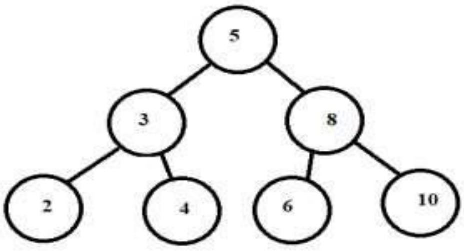
Inorder

2-3-4 ← **5** → 6-8-10



✓ Now for the left sub tree do the same as above.
✓ Preorder traversal of left sub tree is 3 -2-4.
So 3 becomes the root.
✓ Inorder traversal divided further into 2 ← **3** → 4
✓
✓ Now for the right subtree do the same as above.
✓ Preorder traversal of the right subtree is 8 -6-10.
So 8 becomes the root.
✓ Inorder traversal divided further into 6 ← **8** → 10

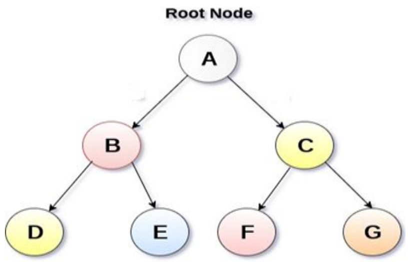




So, in this way we constructed the original tree from given preorder and inorder travers

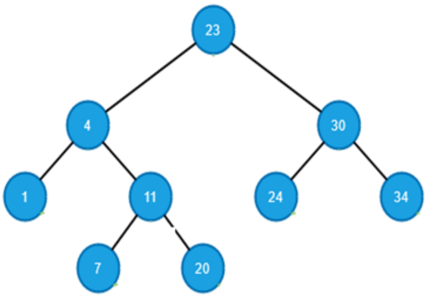
Example:2: Construct binary tree for given postorder and inorder traversals.

In-order: D→ B→ E→ A →F→ C→ G
Pre-order: A→ B→ D→ E →C→ F→ G



Example:3 Construct binary tree for given postorder and inorder traversals.

In-order: 1→ 4→ 7→ 11 →20→ 23→ 24→ 30→ 34
Pre-order: 23→ 4→ 1→ 11 →7→ 20→ 30→ 24→ 34



Implementation of Binary tree traversals

```
#include <stdio.h>
#include <stdlib.h>
// Define the structure for a node
struct Node {
   int data;
   struct Node* left;
   struct Node* right;
};

// Function to create a new node
struct Node* createNode(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->left = NULL;
   newNode->right = NULL;
   return newNode;
}
// Recursive function to insert nodes from user input
struct Node* insertNode() {
   int data;
   printf("Enter node value (-1 for no node): ");
   scanf("%d", &data);

   if (data == -1)
      return NULL;

   struct Node* root = createNode(data);

   printf("Enter left child of %d\n", data);
   root->left = insertNode();

   printf("Enter right child of %d\n", data);
   root->right = insertNode();

   return root;
}

// Inorder traversal (Left, Root, Right)
void inorder(struct Node* root) {
   if (root != NULL) {
      inorder(root->left);
      printf("%d ", root->data);
      inorder(root->right);
   }
}
```

```
// Preorder traversal (Root, Left, Right)
void preorder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}

// Postorder traversal (Left, Right, Root)
void postorder(struct Node* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->data);
    }
}
// Main function
int main() {
    printf("Create binary tree:\n");
    struct Node* root = insertNode();

    printf("\nInorder traversal: ");
    inorder(root);

    printf("\nPreorder traversal: ");
    preorder(root);

    printf("\nPostorder traversal: ");
    postorder(root);

    return 0;
}
```

Input format

"First you must give the root element, followed by left and right element of each node in level order. If a left or right element is not present, enter -1."

For Example consider above Example-3:

INPUT:

```
Create binary tree:

Enter node value (-1 for no node): 23

Enter left child of 23

Enter node value (-1 for no node): 4

Enter left child of 4

Enter node value (-1 for no node): 1
Enter left child of 1
Enter node value (-1 for no node): -1
Enter right child of 1
Enter node value (-1 for no node): -1
Enter right child of 4
Enter node value (-1 for no node): 11
Enter left child of 11
Enter node value (-1 for no node): 7
Enter left child of 7
Enter node value (-1 for no node): -1
Enter right child of 7
Enter node value (-1 for no node): -1
Enter right child of 11
Enter node value (-1 for no node): 20
Enter left child of 20
Enter node value (-1 for no node): -1
Enter right child of 20
Enter node value (-1 for no node): -1
Enter right child of 23
Enter node value (-1 for no node): 30
Enter left child of 30
Enter node value (-1 for no node): 24
Enter left child of 24
Enter node value (-1 for no node): -1
Enter right child of 24
Enter node value (-1 for no node): -1
Enter right child of 30
Enter node value (-1 for no node): 34
Enter left child of 34
Enter node value (-1 for no node): -1
Enter right child of 34
Enter node value (-1 for no node): -1
```

OUTPUT:

```
Inorder traversal: 1 4 7 11 20 23 24 30 34
Preorder traversal: 23 4 1 11 7 20 30 24 34
Postorder traversal: 1 7 20 11 4 24 34 30 23
```

SHORT ANSWER QUESTIONS

1. What role does a stack play in the implementation of the Towers of Hanoi problem?
2. Why is a stack used in a parenthesis checker program?
3. In infix to postfix conversion, when do you pop operators from the stack?
4. What is the key difference between infix and prefix expressions regarding operator placement?
5. How does a stack assist in converting infix expressions to prefix?
6. What is a binary tree? Define a complete binary tree.
7. What is a full binary tree?
8. What is the difference between a full and a complete binary tree
9. How is a binary tree represented using arrays?
10. What are the advantages of array representation of binary trees?
11. What are the disadvantages of array representation of binary trees?
12. How is a binary tree represented using linked lists?
13. What are the advantages of linked list representation in trees?
14. What is tree traversal and why is it needed?
15. Describe inorder traversal of a binary tree.

LONG ANSWER QUESTIONS

1. A recursive puzzle is to be solved where disks need to be moved from a source rod to a destination rod, following the Towers of Hanoi rules.
   a) Explain how the recursive solution to the Towers of Hanoi problem works in C?
   b) Describe the role of the stack in each recursive call.
2. A program must verify whether mathematical or logical expressions have balanced and properly nested parentheses before evaluation
   a) Discuss how stack operations help check for balanced parentheses in an expression.
   b) Write a C program to implement this logic and explain how the stack is updated with each character in the expression.
3. You are working on a symbolic mathematics tool that takes user input in infix form and converts it to prefix for faster computation by the backend parser.
   a) Explain how stacks can be used to convert an infix expression to prefix in C.
   b) Describe the precedence and associativity rules applied during conversion.
   c) Provide and explain a C implementation of the algorithm.

4. A command-line calculator is being developed in C, and you are responsible for implementing the conversion of expressions from infix to postfix notation for evaluation.
    a) Explain the stack-based algorithm for converting an infix expression to postfix using C.
    b) Include a complete program and explain how operators and operands are processed and output is generated.
5. A data structure must be created to store hierarchical data using binary trees, along with the ability to traverse it in different orders.
    a) Describe how to represent a binary tree using structures and pointers in C.
    b) Explain how preorder, inorder, and postorder traversals work and write C functions to implement each traversal.