

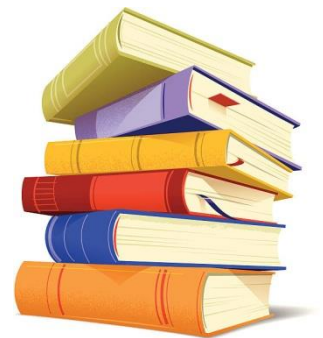
Stacks**PART-2**

Stacks: *Stacks definition, operations on stacks, Representation and evaluation of expressions using Infix, Prefix and Postfix, Algorithms for conversions and evaluations of expressions from infix to prefix and postfix using stack.*

Stack is a Linear Data Structure that follows the **LIFO (Last-In, First-Out)** principle, meaning the last element added is the first one to be removed.

Example:

Think of a stack of Books—you can only add a Book to the top, and you can only remove a Book from the top.

**Basic Operations on Stack**

A stack supports the following main operations:

Operation	Description
push(x)	Inserts an element x on top of the stack
pop()	Removes and returns the top element
peek()	Returns the top element without removing it
isEmpty()	Checks if the stack is empty
isFull()	Checks if the stack is full (for array representation)

1. Push Operation (Insert an element onto stack)**Algorithm:****PUSH(stack, TOP, MAX, item)**

1. if $TOP = MAX - 1$ then
 Print "Overflow" // Stack is full
 return
 2. else
 $TOP \leftarrow TOP + 1$
 $stack[TOP] \leftarrow item$
 Print "Item inserted"
 3. return
-

2. Pop Operation (Remove element from stack)**Algorithm:****POP(stack, TOP)**

1. if $TOP = -1$ then
 Print "Underflow" // Stack is empty
 return
 2. else
 $item \leftarrow stack[TOP]$
 $TOP \leftarrow TOP - 1$
 Print "Item deleted = ", item
 3. return item
-

3. Peek (Get top element without removing it)**Algorithm:****PEEK(stack, TOP)**

1. if $TOP = -1$ then
 Print "Stack is empty"
2. else
 Print "Top element = ", $stack[TOP]$
3. return

4. IsEmpty (Check if stack is empty)

Algorithm:

IsEmpty(TOP)

1. if $TOP = -1$ then
 return TRUE
 2. else
 return FALSE
-

5. IsFull (Check if stack is full)

Algorithm:

IsFull(TOP, MAX)

1. if $TOP = MAX - 1$ then
 return TRUE
2. else
 return FALSE

Representation of Stack

Stacks can be represented in two ways:

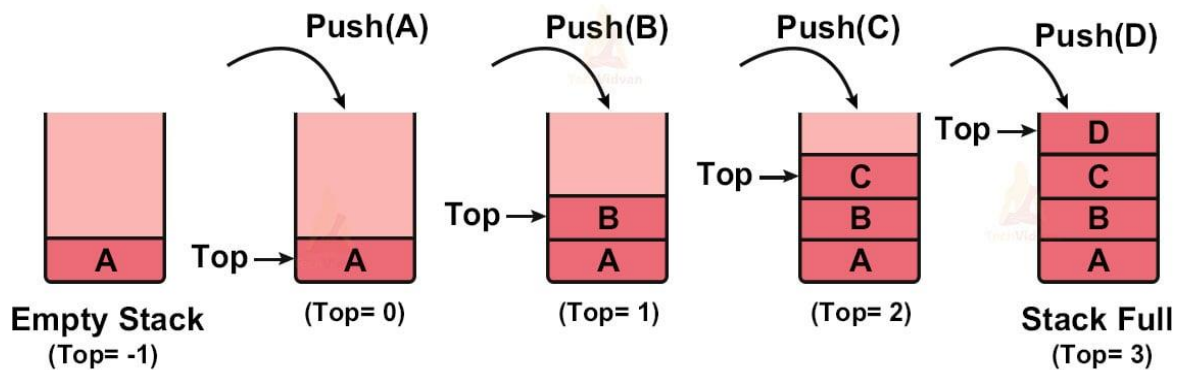
1. Array Representation

In this representation, a stack is implemented using a fixed-size array. A variable, typically called **top**, acts as an index to track the top-most element.

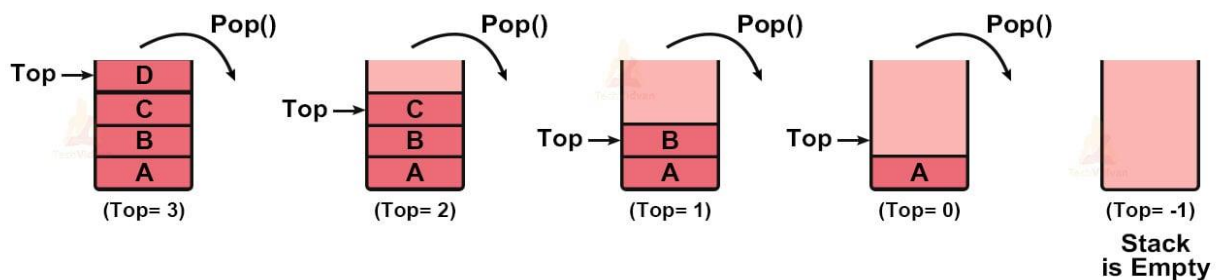
- **Initial State:** The **top** is initialized to **-1**, indicating the stack is **empty**.
- **Push Operation:** To push an element, you increment **top** and then place the new element at **stack[top]**.
- **Pop Operation:** To pop an element, you retrieve the element at **stack[top]** and then decrement **top**.

This method is efficient for fixed-size stacks but can lead to a **stack overflow** if you try to push an element onto a full stack.

Push Operation



Pop operation



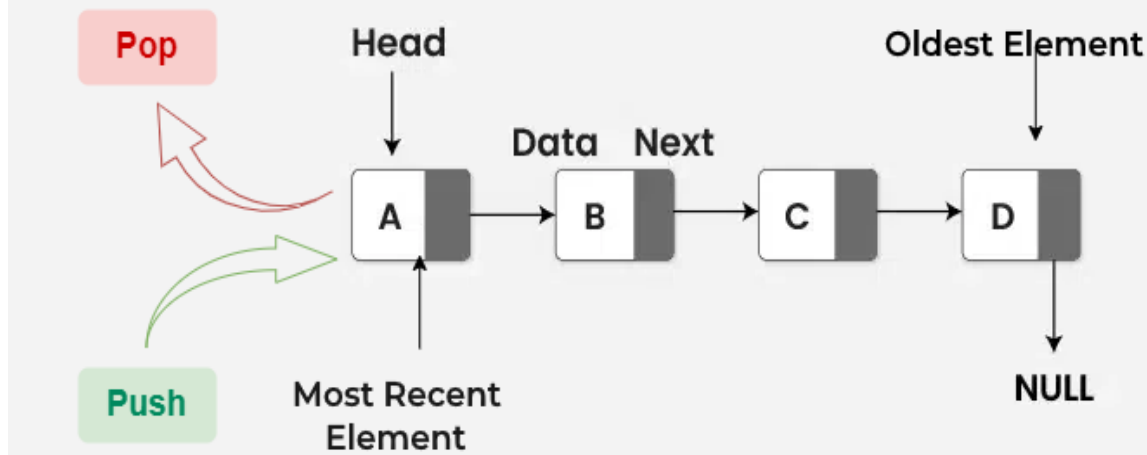
2. Linked List Representation

A stack can also be implemented using a singly linked list. In this model, the **head** of the linked list is considered the **top** of the stack.

- **Push Operation:** A new node is created and made the new head of the list. This operation involves updating a single pointer.
- **Pop Operation:** The head of the list is removed and the next node becomes the new head. This also involves updating a single pointer.

This implementation is more flexible than an array-based stack because it can grow and shrink dynamically, avoiding the stack overflow issue associated with fixed-size arrays.

Stack as Linked List



```
struct Node
```

```
{
```

```
    int data;
```

```
    struct Node *next;
```

```
};
```

```
struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
```

PUSH Operation	POP Operation
<pre>struct Node *top = NULL; newNode->data = value; newNode->next = top; top = newNode;</pre>	<pre>struct Node *temp = top; printf("%d Popped from Stack\n", top->data); top = top->next; free(temp);</pre>

Applications of Stack

1. Expression Evaluation

- **Arithmetic expression evaluation** (e.g., $2 + 3 * (4 - 1)$).
- Uses stack to handle **operands and operators** based on precedence.
- Example: Implemented in calculators.

2. Expression Conversion

- **Infix** → **Postfix/Prefix** and vice versa.
- Used in compiler design for syntax parsing.
- Example: Converting $A + B * C$ to $A B C * +$.

3. Function Call Management (Recursion)

- Each function call's **local variables return address and parameters** are stored in a **function call stack**.
- Supports **nested function calls** and recursion.

4. Undo/Redo Operations

- In text editors and graphics software.
- Actions are pushed onto an **undo stack**; popped to reverse changes.

5. Browser History

- **Back/Forward navigation** implemented using stacks.
- Going back pops from the "back stack" and pushes to the "forward stack".

6. Syntax Parsing

- Used in **compilers** to check for **balanced parentheses, brackets, and braces**.
- Also helps in parsing HTML/XML tags.

7. Depth-First Search (DFS)

- DFS in graphs is implemented using a stack (either explicit or via recursion).

8. Memory Management

- **Stack memory** stores temporary variables, return addresses, etc.
- Automatically managed with LIFO order.

9. Backtracking Algorithms

- Examples: Maze solving, Sudoku, N-Queens problem.
- The stack stores choices made so far; pop to backtrack.

Representation and Evaluation of Expressions

An **expression** in data structures is a combination of operands and operators that represents a computation.

An **operand** is a value or variable, such as A, B, or the number 5.

An **operator** is a symbol that performs an operation, such as +, -, *, or /.

Expressions are categorized into **three** types based on the relative position of the operators and operands:

There are **three common** notations for representing algebraic expressions:

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

They all represent the same expression but differ in the placement of operators relative to their operands.

1. Infix Notation

This is the most familiar notation for humans.

- **Representation:** The operator is placed **between** its operands. Parentheses and operator precedence rules are required to determine the order of operations.
- **Example:** $(A + B) * C$
- **Evaluation:** Evaluating infix expressions is complex for computers because they must account for precedence rules & Associativity.

The **BODMAS rule** is an acronym used in mathematics to define the correct order of operations for evaluating an arithmetic expression. It stands for:

- **Brackets:** Operations inside brackets (parentheses) are always performed first.
- **Orders:** This refers to powers (indices), square roots, and other exponential functions.
- **Division and Multiplication:** These are performed next, working from left to right.
- **Addition and Subtraction:** These are performed last, also working from left to right.

- Let's evaluate the expression: $10+(6\div 2)\times 3^2-5$

The final answer is **32**

2. Prefix Notation (Polish Notation)

This notation is operator-centric.

- Representation:** The operator is placed **before** its operands. It eliminates the need for parentheses and precedence rules because the order of operations is determined by the position of the operators and operands.
- Example:** + A B is the prefix equivalent of $A + B$.
- Evaluation:** Prefix expressions are typically evaluated using a stack. The expression is scanned from **right to left**.
 1. Push operands onto the stack.
 2. When an operator is encountered, pop the top two operands from the stack, apply the operation, and push the result back onto the stack.
 3. The final result is the last item on the stack.

3. Postfix Notation (Reverse Polish Notation)

This notation is operand-centric and is commonly used in calculators and compilers.

- Representation:** The operator is placed **after** its operands. Like prefix, it requires no parentheses or precedence rules.
- Example:** A B + is the postfix equivalent of $A + B$.
- Evaluation:** Postfix expressions are also evaluated using a stack, but the expression is scanned from **left to right**.
 1. Push operands onto the stack.
 2. When an operator is encountered, pop the top two operands, apply the operation, and push the result back onto the stack.
 3. The final result is the last item on the stack.

Conversion of an Infix Expression to a Postfix Expression

Converting an infix expression to a postfix expression is a classic algorithm in computer science, typically performed using a stack. The goal is to reorder the operators and operands so that the operators appear after their operands, which simplifies expression evaluation as it eliminates the need for parentheses and operator precedence rules.

Here is the algorithm for converting an infix expression to a postfix expression:

Infix to Postfix Conversion Algorithm

1. **Initialize an empty stack** for operators and an empty string or list for the postfix expression.
2. **Scan the infix expression from left to right**, character by character.
3. **Process each character:**
 - **If the character is an operand** (e.g., a letter or a number), append it directly to the postfix expression.
 - **If the character is an opening parenthesis (**, push it onto the stack.
 - **If the character is a closing parenthesis)**, pop all operators from the stack and append them to the postfix expression until an opening parenthesis (is encountered. Pop the opening parenthesis from the stack but **do not** append it to the postfix expression.
 - **If the character is an operator** (e.g., +, -, *, /, ^), follow these steps:
 - While the stack is **not empty** and the operator at the top of the stack has a **higher or equal precedence** than the current operator, pop the operator from the stack and append it to the postfix expression.
 - After the loop, push the current operator onto the stack.
4. **After scanning the entire infix expression**, pop any remaining operators from the stack and append them to the postfix expression.

The final string or list is the postfix expression.

Operator Precedence Rules

The precedence of operators is crucial for this algorithm. A common hierarchy, from highest to lowest, is:

1. $^$ (Exponentiation)
2. $*$, $/$ (Multiplication, Division)
3. $+$, $-$ (Addition, Subtraction)

Parentheses $()$ are used to override this precedence. Operators on the same level (e.g., $+$ and $-$) are typically handled with left-to-right associativity.

Example: Converting $(A + B) * C$

Let's trace the conversion of the infix expression $(A + B) * C$ using the algorithm:

Infix Character	Stack	Postfix Expression	Remarks
((Push (onto the stack.
A	(A	A is an operand, append to postfix.
+	(, +	A	+ is an operator, push onto stack.
B	(, +	A B	B is an operand, append to postfix.
)		A B +	Pop operators until (is found. Pop + and append. Discard (.
*	*	A B +	* is an operator, push onto stack.
C	*	A B + C	C is an operand, append to postfix.
(end)		A B + C *	Scan ends. Pop remaining operator * from the stack.

The final postfix expression is $A B + C *$.

Conversion of an Infix Expression to a Prefix Expression

The conversion of an infix expression to a prefix expression, also known as Polish notation, is another classic algorithm that uses a stack. Unlike infix, where operators are between operands, prefix notation places operators before their operands. This method simplifies expression evaluation by eliminating the need for parentheses and precedence rules.

The process is essentially a modified version of the infix-to-postfix algorithm. The key difference is that you process the expression in reverse and use a slightly different set of rules for handling parentheses and operators.

Infix to Prefix Conversion Algorithm

1. Reverse the Infix Expression:

- Reverse the entire infix expression.
- Swap all opening parentheses (with closing parentheses) and vice versa.

2. Convert the Modified Infix to Postfix:

- Apply the standard infix-to-postfix conversion algorithm to this modified (reversed and parenthetically swapped) expression.
- Use a stack to manage operators and a list to build the postfix result.
- **Precedence:** The operator precedence rules remain the same. The only difference is in the associativity. For operators with the same precedence (e.g., + and -), the right-to-left associativity of the original expression becomes left-to-right when reversed. However, the standard infix-to-postfix algorithm handles this naturally.

3. Reverse the Resulting Postfix Expression:

- Reverse the final postfix expression to obtain the prefix expression.

Example: Converting $(A + B) * C$

Let's walk through the conversion of the infix expression $(A + B) * C$ to prefix notation.

Step 1: Reverse the Infix Expression

Original Infix: $(A + B) * C$

1. **Reverse:** $C *) B + A ($
2. **Swap Parentheses:** $C * (B + A)$

The new expression to convert to postfix is: $C * (B + A)$

Step 2: Convert the Modified Expression to Postfix

Now, we'll apply the standard infix-to-postfix algorithm to $C * (B + A)$.

Character	Stack	Postfix Expression	Remarks
C		C	C is an operand, append it.
*	*	C	* is an operator, push it.
(*, (C	(is an opening parenthesis, push it.
B	*, (C B	B is an operand, append it.
+	*, (, +	C B	+ is an operator, push it.
A	*, (, +	C B A	A is an operand, append it.
)	*	C B A +	Pop operators until (is found. Pop +. Discard (.
(end)		C B A + *	Scan ends. Pop remaining operator *.

The resulting postfix expression is **C B A + ***

Step 3: Reverse the Resulting Postfix Expression

Finally, reverse the postfix expression $C B A + *$ to get the final prefix expression.

- $C B A + *$ reversed becomes $* + A B C$

The final prefix expression is *** + A B C**

Postfix Evaluation

Postfix evaluation is a process for solving expressions where the operators come after the operands. This method eliminates the need for parentheses and complex operator precedence rules. The algorithm uses a stack to store and manipulate operands.

Postfix Evaluation Algorithm

1. **Initialize an empty stack.**
2. **Scan the postfix expression from left to right**, character by character.
3. **Process each character:**
 - **If the character is an operand** (a number), push it onto the stack.
 - **If the character is an operator** (+, -, *, /, ^), pop the top two operands from the stack. The first operand popped is the right operand, and the second is the left. Perform the operation and push the result back onto the stack.
4. **After the scan is complete**, the final result will be the only element left on the stack. Pop this value to get the answer.

Example: Evaluating $2\ 3\ *\ 5\ +$

Let's trace the evaluation of the postfix expression $2\ 3\ *\ 5\ +$

Character	Stack	Remarks
2	2	2 is an operand, push it.
3	2, 3	3 is an operand, push it.
*	6	* is an operator. Pop 3 (right) and 2 (left). Calculate $2 * 3 = 6$. Push 6 onto the stack.
5	6, 5	5 is an operand, push it.
+	11	+ is an operator. Pop 5 (right) and 6 (left). Calculate $6 + 5 = 11$. Push 11 onto the stack.
(end)	11	The expression is fully scanned. The final result is the single element on the stack.

The result of the expression $2\ 3\ *\ 5\ +$ is **11**.

Program to implement Stack Operations using Arrays

```
#include <stdio.h>
#define SIZE 100

int stack[SIZE];
int top = -1;

// Function to add element to the stack
void push(int value)
{
    if (top == SIZE - 1)
    {
        printf("Stack Overflow! Cannot Push %d\n", value);
    } else
    {
        stack[++top] = value;
        printf("%d Pushed into Stack\n", value);
    }
}

// Function to remove element from the stack
int pop()
{
    if (top == -1)
    {
        printf("Stack Underflow! Cannot Pop\n");
        return -1;
    }
    else
    {
        int popped = stack[top--];
        printf("%d Popped from Stack\n", popped);
        return popped;
    }
}
```

// Function to return the top element

```
int peek()
{
    if (top == -1)
    {
        printf("Stack is Empty\n");
        return -1;
    }
    else
    {
        return stack[top];
    }
}
```

// Function to display all elements

```
void display()
{
    int i;
    if (top == -1)
    {
        printf("Stack is Empty\n");
    }
    else
    {
        printf("Stack Elements: ");
        for (i = top; i >= 0; i--)
        {
            printf("%d ", stack[i]);
        }
        printf("\n");
    }
}
```

```
int main()
{
    int choice, value;
```

```
while (1)
{
    printf("\n--- Stack Operations Menu ---\n");
    printf("1. Push\n2. Pop\n3. Peek\n4. Display\n5. Exit\n");
    printf("Enter your Choice: ");
    scanf("%d", &choice);

    switch (choice)
    {
        case 1:
            printf("Enter value to Push: ");
            scanf("%d", &value);
            push(value);
            break;

        case 2:
            pop();
            break;

        case 3:
            value = peek();
            if (value != -1)
                printf("Top element: %d\n", value);
            break;

        case 4:
            display();
            break;

        case 5:
            printf("Exiting Program.\n");
            return 0;

        default:
            printf("Invalid choice! Try again.\n");
    }
}
return 0;
}
```


Output:

```
F:\VJIT\2025-26\I SEM\DS\Examples\W1\StackArray.exe

--- Stack Operations Menu ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your Choice: 1
Enter value to Push: 11
11 Pushed into Stack

--- Stack Operations Menu ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your Choice: 1
Enter value to Push: 22
22 Pushed into Stack

--- Stack Operations Menu ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your Choice: 4
Stack Elements: 22 11
```

```
F:\VJIT\2025-26\I SEM\DS\Examples\W1\StackArray.exe

--- Stack Operations Menu ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your Choice: 3
Top element: 22

--- Stack Operations Menu ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your Choice: 2
22 Popped from Stack

--- Stack Operations Menu ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your Choice: 4
Stack Elements: 11
```

```
F:\VJIT\2025-26\I SEM\DS\Examples\W1\StackArray.exe

--- Stack Operations Menu ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your Choice: 5
Exiting Program.

-----
Process exited after 126.5 seconds with return value 0
Press any key to continue . . .
```

Program to implement Stack Operations using Linked List

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

// Define node structure
struct Node
{
    int data;
    struct Node *next;
};

// Top pointer to keep track of the stack top
struct Node *top = NULL;

// Push operation
void push(int value)
{
    struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
    if (!newNode)
    {
        printf("Heap overflow! Unable to Push %d\n", value);
        return;
    }
    newNode->data = value;
    newNode->next = top;
    top = newNode;
    printf("%d Pushed into Stack\n", value);
}

// Pop operation
void pop()
{
    if (top == NULL)
    {
        printf("Stack Underflow! Cannot Pop\n");
        return;
    }
}
```

```
    struct Node *temp = top;
    printf("%d Popped from Stack\n", top->data);
    top = top->next;
    free(temp);
}
```

// Peek operation

```
void peek()
{
    if (top == NULL)
    {
        printf("Stack is Empty\n");
    }
    else
    {
        printf("Top element: %d\n", top->data);
    }
}
```

// Display the stack

```
void display()
{
    struct Node *temp = top;
    printf("Top -> ");
    while (temp != NULL)
    {
        printf("[%d] -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
```

// Main menu-driven function

```
int main()
{
    int choice, value;
```

```
while (1)
{
    printf("\n--- Stack Menu ---\n");
    printf("1. Push\n2. Pop\n3. Peek\n4. Display\n5. Exit\n");
    printf("Enter your Choice: ");
    scanf("%d", &choice);

    switch (choice)
    {
        case 1:
            printf("Enter value to Push: ");
            scanf("%d", &value);
            push(value);
            break;

        case 2:
            pop();
            break;

        case 3:
            peek();
            break;

        case 4:
            display();
            break;

        case 5:
            printf("Exiting Program.\n");
            exit(0);

        default:
            printf("Invalid Choice! Try again.\n");
    }
}

return 0;
}
```

Output:

```
F:\VJIT\2025-26\I SEM\DS\Examples\W1\StackLinkedList.exe
--- Stack Menu ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your Choice: 1
Enter value to Push: 11
11 Pushed into Stack

--- Stack Menu ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your Choice: 1
Enter value to Push: 22
22 Pushed into Stack

--- Stack Menu ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your Choice: 4
Top -> [22] -> [11] -> NULL
```

```
F:\VJIT\2025-26\I SEM\DS\Examples\W1\StackLinkedList.exe
--- Stack Menu ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your Choice: 3
Top element: 22

--- Stack Menu ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your Choice: 2
22 Popped from Stack

--- Stack Menu ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your Choice: 4
Top -> [11] -> NULL
```

```
F:\VJIT\2025-26\I SEM\DS\Examples\W1\StackLinkedList.exe
--- Stack Menu ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your Choice: 5
Exiting Program.

-----
Process exited after 132.2 seconds with return value 0
Press any key to continue . . .
```

Program to Convert Infix Expression to Postfix Expression

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h> // for isalpha and isdigit

#define SIZE 100

char stack[SIZE];
int top = -1;

// Stack operations
void push(char c)
{
    if (top == SIZE - 1)
    {
        printf("Stack Overflow\n");
    } else {
        stack[++top] = c;
    }
}

char pop()
{
    if (top == -1)
    {
        return '\0';
    } else {
        return stack[top--];
    }
}

char peek()
{
    if (top == -1)
        return '\0';
    return stack[top];
}
```

```
// Function to return precedence of operators
int precedence(char op)
{
    switch (op)
    {
        case '^': return 3;
        case '*':
        case '/': return 2;
        case '+':
        case '-': return 1;
        default: return 0;
    }
}

// Function to check if the character is an operator
int isOperator(char c)
{
    return c == '+' || c == '-' || c == '*' || c == '/' || c == '^';
}

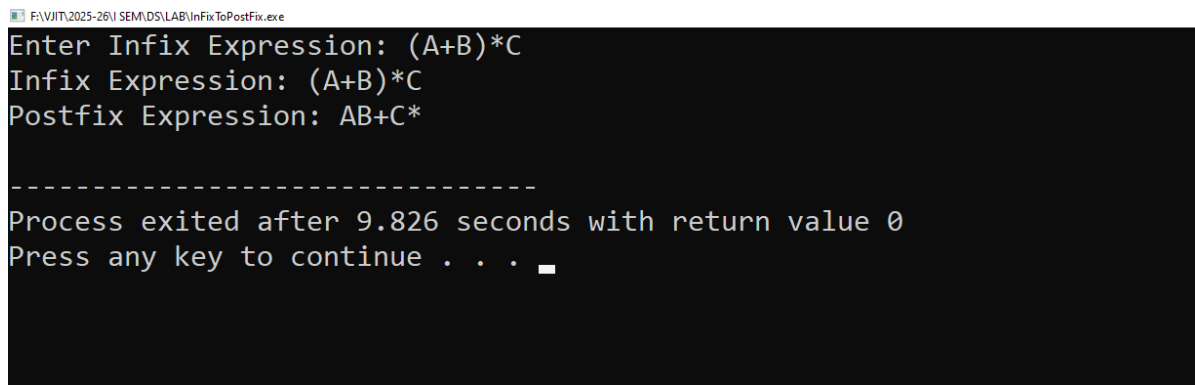
// Function to convert infix to postfix
void infixToPostfix(char* infix)
{
    char postfix[SIZE];
    int i = 0, j = 0;
    char symbol, temp;
    printf("Infix Expression: %s\n", infix);
    while ((symbol = infix[i++]) != '\0')
    {
        if (isalnum(symbol))
        {
            postfix[j++] = symbol;
        } else if (symbol == '(') {
            push(symbol);
        } else if (symbol == ')') {
            while ((temp = pop()) != '(')
            {
                postfix[j++] = temp;
            }
        }
    }
}
```

```
    } else if (isOperator(symbol)) {
        while (top != -1 && precedence(peek()) >= precedence(symbol))
        {
            postfix[j++] = pop();
        }
        push(symbol);
    }
}

while (top != -1)
{
    postfix[j++] = pop();
}

postfix[j] = '\0';
printf("Postfix Expression: %s\n", postfix);
}

// Main function
int main()
{
    char infix[SIZE];
    printf("Enter Infix Expression: ");
    scanf("%s", infix);
    infixToPostfix(infix);
    return 0;
}
```

Output:

```
F:\VJIT\2025-26\I SEM\DS\LAB\InFixToPostFix.exe
Enter Infix Expression: (A+B)*C
Infix Expression: (A+B)*C
Postfix Expression: AB+C*

-----
Process exited after 9.826 seconds with return value 0
Press any key to continue . . .
```


Program to Evaluate Postfix Expression

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h> // for isdigit
#define SIZE 100
int stack[SIZE];
int top = -1;
// Stack Operations
void push(int value)
{
    if (top == SIZE - 1)
    {
        printf("Stack Overflow\n");
    } else {
        stack[++top] = value;
    }
}
int pop()
{
    if (top == -1)
    {
        printf("Stack Underflow\n");
        return -1;
    } else {
        return stack[top--];
    }
}
// Evaluate postfix expression
int evaluatePostfix(char* expr)
{
    int i = 0;
    char ch;
    int op1, op2, result;
    while ((ch = expr[i++]) != '\0')
    {
        if (isdigit(ch))
        {
            push(ch - '0'); // Convert char to int
        } else {
```

```
        op2 = pop();
        op1 = pop();
        switch (ch)
        {
            case '+': result = op1 + op2; break;
            case '-': result = op1 - op2; break;
            case '*': result = op1 * op2; break;
            case '/': result = op1 / op2; break;
            default:
                printf("Unsupported operator: %c\n", ch);
                return -1;
        }
        push(result);
    }
}
return pop();
}
// Main function
int main()
{
    char postfix[SIZE];
    printf("Enter Postfix Expression: ");
    scanf("%s", postfix);
    int result = evaluatePostfix(postfix);
    printf("Result of Postfix Expression: %d\n", result);
    return 0;
}
```

Output:

F:\VJIT\2025-26\I SEM\DS\LAB\PostFixEval.exe

```
Enter Postfix Expression: 23*5+
Result of Postfix Expression: 11
```

```
-----
Process exited after 8.007 seconds with return value 0
Press any key to continue . . .
```