UNIT-3

STACK

1. **What is a stack?**

A stack is a linear data structure that follows the **Last In, First Out (LIFO)** principle. This means that the most recently added (pushed) element is the first one to be removed (popped).

2. **What are the two main operations of a stack?**
   - **Push** (inserts an element)
   - **Pop** (removes the top element)

3. **What is the time complexity of push and pop operations?**

Both **Push** and **Pop** operations take **O(1) (constant time)** because they operate only at the top of the stack.

4. **What is stack overflow?**

5. Stack overflow occurs when we try to push an element into a stack that has reached its maximum size (for an array-based stack).

Example:

If a stack is implemented using a fixed-size array (e.g., size 5), pushing a 6th element will cause **stack overflow**.

5. **What is stack underflow?**

Stack underflow happens when we try to pop an element from an empty stack. Since there are no elements to remove, it leads to an error.

6. **How can a stack be implemented?**

Using **arrays** (fixed size) or **linked lists** (dynamic size).

7. **Which is better: array-based or linked-list-based stack?**
   - Array-based stacks are faster but have a fixed size.
   - Linked-list-based stacks are flexible but use extra memory.

8. What is the top of the stack?

The **stack top** refers to the **element at the highest position** in a stack. It is the most recently added item and is the first to be removed when performing a **pop** operation.

9. **Where are stacks used in real life?**
   - Undo/Redo in editors
   - Backtracking (e.g., maze solving, recursion)
   - Expression evaluation (postfix, prefix)
   - Function call management in recursion

10. **How does a stack handle function calls?**

The **call stack** stores function calls and returns them in **LIFO** order

11. What is the top of the stack?

The **top** element is the last inserted element, which will be removed first when popping.

Example:

Stack: [10, 20, 30]

Top element = 30 (last pushed element)

12.what are the Advantages of Stack ?

1. **Follows LIFO (Last In, First Out) Order**
    - Ensures the most recently added data is accessed first, which is useful in function calls, undo operations, and expression evaluation.
2. **Efficient Memory Management**
    - Stack memory allocation is **automatic** and **fast**, making function calls and local variable storage more efficient than heap allocation.
3. **Simplifies Function Call Management**
    - The **call stack** keeps track of function calls and local variables, ensuring smooth execution of recursive and nested functions.
4. **Backtracking and Undo Operations**
    - Stacks are used in algorithms that require **backtracking** (e.g., solving mazes, Depth First Search (DFS), undo/redo features in text editors).
5. **Expression Evaluation and Parsing**
    - Stacks simplify evaluating expressions in **postfix notation** and **parsing syntax** in compilers.
6. **Efficient Operations (O(1) Complexity)**
    - Push and pop operations take **constant time (O(1))**, making them very fast.
7. **Less Memory Wastage** (Compared to Queues)
    - Since elements are removed from the top, there is **no need for shifting** like in queues.
8. **Easy to Implement**
    - Stacks can be implemented using **arrays** or **linked lists** with minimal effort.

13.what are the Disadvantages of Stack?

1. **Limited Size (Fixed in Array Implementation)**
    - If implemented using an **array**, the stack has a **fixed size**, which can lead to **stack overflow** if too many elements are pushed.
2. **Stack Overflow and Underflow**
    - **Stack Overflow** occurs when pushing an element into a full stack.
    - **Stack Underflow** happens when popping an element from an empty stack.

3. **No Random Access**
   - Unlike arrays, stacks **do not allow direct access** to elements in the middle; only the **top element** can be accessed.
4. **Extra Memory in Linked List Implementation**
   - If implemented using a **linked list**, extra memory is needed for pointers, making it less space-efficient than an array-based stack.
5. **Limited Flexibility for Large Data Handling**
   - Stacks are **not ideal for large dynamic data storage** since removing an element from the middle is not possible without modifying the structure.
6. **Complex Debugging and Error Handling**
   - Since the stack only allows access to the top element, debugging or retrieving past elements can be difficult.

14.what are the Real-Time Applications of Stack

1. **Function Calls (Call Stack)**
   - When a function is called, it is pushed onto the **call stack**. Once the function finishes execution, it is popped off.
   - Used in **recursion**, where multiple function calls are stacked until the base case is reached.
2. **Undo/Redo in Text Editors**
   - Every action (typing, deleting, formatting) is pushed onto a stack.
   - Undo: Pops the last action and restores the previous state.
   - Redo: Reapplies the last undone action.
3. **Backtracking Algorithms**
   - Used in **maze solving, puzzle solving, and DFS (Depth First Search)**.
   - If a wrong path is chosen, the algorithm **backtracks** by popping from the stack.
4. **Expression Evaluation and Syntax Parsing**
   - Used in **compilers and calculators** for:
     - Converting infix expressions to postfix/prefix.
     - Evaluating mathematical expressions.
5. **Browser Forward and Back Buttons**
   - Every visited page is **pushed** onto a stack.
   - Clicking "Back" **pops** the current page and returns to the previous page.
   - Clicking "Forward" pushes the popped page back.
6. **Undo Operations in Graphics Software**
   - Drawing or editing actions are stored in a stack for easy reversal.

7. **Memory Management in OS (Stack vs Heap)**
    - The stack is used for **local variables and function calls**, while the heap is for **dynamic memory allocation**.
8. **Balancing Parentheses in Expressions**
    - Used in **checking balanced brackets ({[()]})** by pushing opening brackets and popping when a matching closing bracket is found.
9. **Compiler Syntax Checking**
    - Stacks help check **correct nesting of loops, function calls, and expressions** in programming languages.
10. **Reversing a String or Data**

- Characters are pushed onto a stack and then popped out in reverse order.

15. What are the different types of stack implementations? Compare them

Stacks can be implemented in two main ways:

1. Array-Based Stack

- Uses a fixed-size array to store elements.
- The top of the stack is updated with each push or pop operation.

ADV:

- Faster access (O(1) time complexity for push/pop).
- Simple to implement.

DIS:

- Fixed size, leading to stack overflow if full.
- Memory may be wasted if the stack is not fully used.

2. Linked List-Based Stack

- Uses a linked list where each node contains the data and a pointer to the next node.

Advantages:

- No fixed size (dynamically allocated).
- No memory wastage.

Disadvantages:

- Extra memory needed for pointers.
- Slightly slower due to pointer manipulation.

16. Explain the working of a stack with its operations and real-world analogy.

Answer:

A **stack** is a **linear data structure** that follows the **Last In, First Out (LIFO)** principle. This means the last element added (pushed) onto the stack will be the first one to be removed (popped).

Operations on Stack:

1. **Push** – Adds an element to the top of the stack.
2. **Pop** – Removes and returns the top element of the stack.
3. **Peek (Top)** – Returns the top element without removing it.
4. **isEmpty** – Checks if the stack is empty.
5. **isFull** (for array-based stacks) – Checks if the stack is full.

Real-World Analogy:

Imagine a stack of plates in a cafeteria:

- When a new plate is placed, it is added on top (**Push**).
- When someone takes a plate, they pick the top one (**Pop**).
- The plate at the bottom is only accessible after removing all others (**LIFO** principle).

17. How is a stack used in recursion? Explain with an example in C.

Answer:

A stack is used to manage function calls in **recursion**. Every function call is **pushed onto the call stack**, and when the function completes, it is **popped** off the stack.

Example: Factorial Calculation Using Recursion

```
#include <stdio.h>
// Recursive function to calculate factorial
int factorial(int n) {
```

```
  if (n == 0)
        return 1;
    return n * factorial(n - 1);
}
// Main function
int main() {
    int num = 5;
    printf("Factorial of %d is %d\n", num, factorial(num));
    return 0;
}
```

Stack Behavior for factorial(5):

```
factorial(5)
factorial(4)
factorial(3)
factorial(2)
factorial(1)
factorial(0)  # Base case, returns 1
```

Each function call is **pushed onto the stack** until factorial(0) is reached. Then, it starts **popping** the calls and computing the final result.

Output:

Factorial of 5 is 120

18. How is Stack Used in Call Logs?

- **LIFO Principle:** The most recent call is displayed first, while older calls move down the list.
- **Push Operation:** When a new call is made or received, it is **pushed** onto the call log stack.
- **Pop Operation:** When viewing or deleting call logs, the latest (top) call is **popped** first.
- **Limited Size:** If the call log reaches a maximum limit (e.g., 100 entries), the **oldest calls are removed** to make space for new ones (similar to a circular stack).

19.How Stack is Used in Viewing History?

- **LIFO Principle:** The most recent activity (page visited, file opened) is stored at the **top** of the stack.
- **Push Operation:** When a new activity occurs (e.g., opening a webpage), it is **pushed** onto the stack.
- **Pop Operation:** When the user presses "Back," the latest activity is **popped** off the stack, returning to the previous activity.
- **Forward Navigation (Redo):** Another stack is used to store activities when moving forward.

20.Real-World Use Cases of Stack in Viewing History

1. **Web Browsers:**
   - When a user visits a webpage, it is **pushed onto the stack**.
   - Clicking the **Back button** pops the current page and navigates to the previous one.
   - Clicking **Forward** can be implemented using another stack.
2. **File Explorers:**
   - When navigating through folders, each visited folder is **pushed** onto the stack.
   - Pressing **Back** pops the current folder and returns to the previous one.
3. **Mobile Applications:**
   - App screens (activities) are stacked so that when a user presses the **Back button**, the last screen is **popped** and removed.

1.Explain the properties of Stack?

Properties of Stack

A **stack** is a linear data structure that follows the **LIFO (Last In, First Out)** principle. Here are its key properties:

1. **LIFO Principle**
   - The last element inserted is the first one to be removed.
   - Example: If you push A, B, C onto a stack, C will be popped first.
2. **Operations**
   - **Push**: Adds an element to the top of the stack.
   - **Pop**: Removes the top element from the stack.
   - **Peek (Top)**: Returns the top element without removing it.
   - **isEmpty**: Checks if the stack is empty.

   - **isFull**: Checks if the stack is full (for array-based stacks).
3. **Memory Allocation**
   - Can be implemented using **arrays** (fixed size) or **linked lists** (dynamic size).
4. **Access Restriction**
   - Only the **top** element can be accessed or modified directly.
5. **Recursive Nature**
   - Stacks are used in **recursion** (function call stack).
6. **Applications**
   - **Expression evaluation** (Postfix, Prefix, Infix)
   - **Backtracking** (like in maze solving, undo operations)
   - **Function calls** (maintains activation records)
   - **Browser history and undo-redo operations**

2.Write a C program to implement a stack using an array?

```
#include <stdio.h>

#include <stdlib.h>

#define MAX 5 // Maximum size of the stack

int stack[MAX], top = -1;

// Function to push an element onto the stack

void push() {

    int value;

    if (top == MAX - 1) {

        printf("Stack Overflow! Cannot push more elements.\n");

        return;

    }

    printf("Enter the value to push: ");

    scanf("%d", &value);

    stack[++top] = value;

    printf("%d pushed into the stack.\n", value);

}
```

```
// Function to pop an element from the stack
void pop() {
    if (top == -1) {
        printf("Stack Underflow! No elements to pop.\n");
        return;
    }
    printf("%d popped from the stack.\n", stack[top--]);
}
// Function to return the top element without removing it
void peek() {
    if (top == -1) {
        printf("Stack is empty!\n");
        return;
    }
    printf("Top element is: %d\n", stack[top]);
}
// Function to display all elements in the stack
void display() {
int i;
    if (top == -1) {
        printf("Stack is empty!\n");
        return;
    }
```

```
    printf("Stack elements: ");
    for (i = top; i >= 0; i--)
        printf("%d ", stack[i]);
    printf("\n");
}

// Main function with menu-driven interface
int main() {
    int choice;
    while (1) {
        printf("\nStack Operations:\n");
        printf("1. Push\n2. Pop\n3. Peek\n4. Display\n5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1: push();
                      break;
            case 2: pop();
                     break;
            case 3: peek();
                     break;
            case 4: display();
                     break;
```

```
            case 5: exit(0);

            default: printf("Invalid choice! Please enter a valid option.\n");

        }

    }

    return 0;

}
```

Sample test data

1. Push

2. Pop

3. Peek

4. Display

5. Exit

Enter your choice: 1

Enter the value to push: 10

10 pushed into the stack.

Enter your choice: 1

Enter the value to push: 20

20 pushed into the stack.

Enter your choice: 1

Enter the value to push: 30

30 pushed into the stack.

Enter your choice: 4

Stack elements: 30 20 10

Enter your choice: 3

Top element is: 30

Enter your choice: 2

30 popped from the stack

Enter your choice: 4

Stack elements: 20 10

Enter your choice: 5

3. Explain the concept of a stack in detail with an example.

Answer:

A **stack** is a linear data structure that follows the **LIFO (Last In, First Out)** principle, meaning that the last element inserted into the stack is the first one to be removed. It is similar to a stack of plates, where the topmost plate is the first to be removed.

A stack can be implemented using:

1. **Arrays** (Static implementation)
2. **Linked Lists** (Dynamic implementation)

Operations on Stack:

1. **Push:** Adds an element to the top of the stack.
2. **Pop:** Removes the top element from the stack.
3. **Peek:** Retrieves the top element without removing it.
4. **isEmpty:** Checks whether the stack is empty.
5. **isFull:** Checks whether the stack is full (only in an array implementation).

Example:

Consider a stack with the following sequence of operations:

Push(10) → Push(20) → Push(30) → Pop() → Push(40)

Step-by-step changes in the stack:

Operation: Push(10)   Stack: [10]

Operation: Push(20) Stack: [10, 20]
Operation: Push(30) Stack: [10, 20, 30]
Operation: Pop() Stack: [10, 20] (30 is removed)
Operation: Push(40) Stack: [10, 20, 40]

Applications of Stack:

1. **Expression Evaluation:** Converting infix expressions to postfix/prefix.
2. **Undo/Redo Mechanism:** Used in text editors and applications.
3. **Recursion Handling:** Function calls use a stack to store return addresses.
4. **Backtracking:** Used in maze-solving algorithms and game development.

---

4. What are the advantages and disadvantages of using a stack?

Answer:

Advantages of Stack:

1. **Efficient for LIFO Operations:** Best suited for scenarios where the last inserted element needs to be accessed first.
2. **Memory Management:** Used in function calls and recursion (call stack).
3. **Simple and Fast:** Push and Pop operations take **O(1) time complexity**.

Disadvantages of Stack:

1. **Limited Size in Array Implementation:** Fixed size can lead to stack overflow.
2. **Difficult to Access Middle Elements:** Unlike linked lists, direct access to elements other than the top is not possible.
3. **Dynamic Memory Usage (Linked List Implementation):** Requires extra memory for pointers.

---

5. How is a stack used in recursion? Explain with an example.

Answer:

Recursion is a process where a function calls itself until a base condition is met. Internally, recursion uses a **call stack** to keep track of function calls.

How Stack is Used in Recursion:

Each function call is **pushed** onto the stack.

1. When a function completes execution, it is **popped** from the stack.
2. The stack stores return addresses and local variables of each function call.
3. Example: Factorial Using Recursion

```
#include <stdio.h>
int factorial(int n) {
    if (n == 0) return 1; // Base case
    return n * factorial(n - 1); // Recursive call
}
int main() {
    int num = 5;
    printf("Factorial of %d is %d\n", num, factorial(num));
    return 0;
}
```

Stack Representation for factorial(5)
Call: factorial(5) → Push 5
Call: factorial(4) → Push 4
Call: factorial(3) → Push 3
Call: factorial(2) → Push 2
Call: factorial(1) → Push 1
Call: factorial(0) → Push 0 (Base Case, Return 1)
Return: factorial(1) → Pop 1
Return: factorial(2) → Pop 2
Return: factorial(3) → Pop 3
Return: factorial(4) → Pop 4
Return: factorial(5) → Pop 5

Final result: factorial(5) = 5 × 4 × 3 × 2 × 1 = 120

---

6. Differentiate between stack and queue.

Answer:

Feature	Stack (LIFO)	Queue (FIFO)
Principle	Last In, First Out (LIFO)	First In, First Out (FIFO)
Insertion (Push/Enqueue)	Performed at the **top**	Performed at the **rear**
Deletion (Pop/Dequeue)	Done from the **top**	Done from the **front**

Feature	Stack (LIFO)	Queue (FIFO)
Implementation	Arrays, Linked Lists	Arrays, Linked Lists, Circular Queues
Example Use Case	Function calls, Undo operations	Scheduling processes, Printer queue

7.Explain how stacks are used in expression evaluation and conversion.

Answer:

Stacks play a crucial role in **expression evaluation** and **conversion** between different types of mathematical expressions:

Infix Expression: Operators are between operands (e.g., A + B).

- **Postfix Expression:** Operators follow operands (e.g., A B +).
- **Prefix Expression:** Operators precede operands (e.g., + A B).

Why Use Stacks?

1. **Operator Precedence Handling:** Stacks help maintain the correct order of operations.
2. **Efficient Processing:** Conversion and evaluation become easy using a stack.

---

1. Infix to Postfix Conversion Using Stack

Example: Convert A + B * C to postfix form.

Step	Symbol	Stack (Operators)	Postfix Expression
1	A	-	A
2	+	+	A
3	B	+	A B
4	*	+ *	A B
5	C	+ *	A B C
6	End	Pop stack	A B C * +

Final postfix expression: **A B C * +**

---

2. Evaluating a Postfix Expression Using Stack

Example: Evaluate **5 3 + 8 ***

Step	Symbol	Stack (Operands)
1	5	5
2	3	5 3
3	+	8 (5 + 3)
4	8	8 8
5	*	64 (8 × 8)

Final result: **64**

Infix, Prefix, and Postfix Notation Using Stack

In **mathematical expressions**, there are three common notations:

1. **Infix Notation** – Operators are placed **between operands** (e.g., A + B).
2. **Prefix Notation (Polish Notation)** – Operators are placed **before operands** (e.g., + A B).
3. **Postfix Notation (Reverse Polish Notation - RPN)** – Operators are placed **after operands** (e.g., A B +).

Conversion Between Notations

Notation	Example Expression (A + B * C)
Infix	A + (B * C)
Prefix	+ A * B C
Postfix	A B C * +

---

1. Infix to Postfix Conversion Using Stack

Algorithm

1. **Initialize an empty stack** for operators.
2. **Scan the infix expression from left to right**.
3. **If operand (A-Z or 0-9), append to output**.
4. **If an operator (+, -, *, /)**:
   - Pop operators from the stack that have **higher or equal precedence** and append them to the output.
   - Push the current operator onto the stack.
5. **If '(' (left parenthesis), push it onto the stack**.
6. **If ')' (right parenthesis), pop from the stack to the output until '(' is found**.
7. **At the end, pop all remaining operators from the stack**.

C Program: Infix to Postfix Conversion

```
#include <stdio.h>
#include <ctype.h>  // For isalnum()
#include <string.h>
#define MAX 100  // Stack size
// Stack structure
struct Stack {
   char arr[MAX];
   int top;
};
// Initialize stack
void initialize(struct Stack* stack) {
   stack->top = -1;
}
// Check if stack is empty
int isEmpty(struct Stack* stack) {
   return stack->top == -1;
}
// Push an element
void push(struct Stack* stack, char value) {
   stack->arr[++stack->top] = value;
}
// Pop an element
```

```
char pop(struct Stack* stack) {
    return stack->arr[stack->top--];
}
// Get precedence of operators
int precedence(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    return 0;  // For non-operators
}
// Convert infix to postfix
void infixToPostfix(char* infix, char* postfix) {
    struct Stack stack;
    initialize(&stack);
    int j = 0;
    for (int i = 0; infix[i] != '\0'; i++) {
        char ch = infix[i];
        // If operand, add to output
        if (isalnum(ch)) {
            postfix[j++] = ch;
        }
        // If '(', push to stack
        else if (ch == '(') {
            push(&stack, ch);
        }
        // If ')', pop until '(' is found
        else if (ch == ')') {
            while (!isEmpty(&stack) && stack.arr[stack.top] != '(') {
                postfix[j++] = pop(&stack);
            }
            pop(&stack);  // Remove '('
        }
        // If operator, handle precedence
        else {
            while (!isEmpty(&stack) && precedence(stack.arr[stack.top]) >= precedence(ch)) {
                postfix[j++] = pop(&stack);
            }
            push(&stack, ch);
        }
    }
    // Pop remaining operators
    while (!isEmpty(&stack)) {
        postfix[j++] = pop(&stack);
    }
```

```c
 postfix[j] = '\0';  // Null terminate string
}
// Main function
int main() {
    char infix[] = "A+B*C";
    char postfix[MAX];
    infixToPostfix(infix, postfix);
    printf("Postfix Expression: %s\n", postfix);
    return 0;
}
```

Output:

Postfix Expression: ABC*+

---

2. Infix to Prefix Conversion Using Stack

Algorithm

1. **Reverse the infix expression**.
2. **Convert infix to postfix (using stack)**.
3. **Reverse the result to get the prefix expression**.

C Program: Infix to Prefix Conversion

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define MAX 100
// Stack structure
struct Stack {
    char arr[MAX];
    int top;
};
// Initialize stack
void initialize(struct Stack* stack) {
    stack->top = -1;
}
// Check if stack is empty
int isEmpty(struct Stack* stack) {
    return stack->top == -1;
}
// Push an element
```

```
void push(struct Stack* stack, char value) {
    stack->arr[++stack->top] = value;
}
// Pop an element
char pop(struct Stack* stack) {
    return stack->arr[stack->top--];
}
// Get precedence of operators
int precedence(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    return 0;
}
// Reverse a string
void reverse(char* str) {
    int n = strlen(str);
    for (int i = 0; i < n / 2; i++) {
        char temp = str[i];
        str[i] = str[n - i - 1];
        str[n - i - 1] = temp;
    }
}
// Convert infix to prefix
void infixToPrefix(char* infix, char* prefix) {
    // Reverse infix expression
    reverse(infix);
    // Replace ( with ) and vice versa
    for (int i = 0; infix[i] != '\0'; i++) {
        if (infix[i] == '(') infix[i] = ')';
        else if (infix[i] == ')') infix[i] = '(';
    }
    // Convert to postfix
    char postfix[MAX];
    struct Stack stack;
    initialize(&stack);
    int j = 0;
    for (int i = 0; infix[i] != '\0'; i++) {
        char ch = infix[i];
        if (isalnum(ch)) {
            postfix[j++] = ch;
        } else if (ch == '(') {
            push(&stack, ch);
        } else if (ch == ')') {
```

```
 while (!isEmpty(&stack) && stack.arr[stack.top] != '(') {
                postfix[j++] = pop(&stack);
            }
            pop(&stack);
        } else {
            while (!isEmpty(&stack) && precedence(stack.arr[stack.top]) >= precedence(ch)) {
                postfix[j++] = pop(&stack);
            }
            push(&stack, ch);
        }
    }
    while (!isEmpty(&stack)) {
        postfix[j++] = pop(&stack);
    }
    postfix[j] = '\0';
    // Reverse postfix to get prefix
    reverse(postfix);
    strcpy(prefix, postfix);
}
// Main function
int main() {
    char infix[] = "A+B*C";
    char prefix[MAX];
    infixToPrefix(infix, prefix);
    printf("Prefix Expression: %s\n", prefix);
    return 0;
}
```

Output:

Prefix Expression: +A*BC

Summary

Notation	**Example**	**Conversion Using Stack**
Infix	(A + B) * C	Given directly
Postfix	A B + C *	Use **stack** to push/pop operators
Prefix	+ A * B C	Reverse + Convert to postfix + Reverse