STACKS

29	Data Structures	Stacks: Array-Based	- Stack ADT, LIFO principle - Push, pop, peek, isEmpty, isFull - Array-based implementation, overflow handling
30	Data Structures	Stacks: Linked List	- Linked list stack implementation - Dynamic sizing advantages - Prefix/postfix evaluation (shunting-yard)
31	Data Structures	Stack Applications	- Balanced parentheses validation - Expression evaluation (infix to postfix) - Function call stack simulation

1. Stack ADT (Abstract Data Type)

Definition

A **stack** is a linear data structure that follows the **LIFO (Last In First Out)** principle.
All insertion and deletion operations are performed from **one end only**, called the **TOP**.

Why it is called ADT

- ADT defines **what operations** can be performed
- It does **not** define how they are implemented
- Implementation can be using **array** or **linked list**

Key Characteristics

- Linear data structure
- Access is restricted
- Only top element is accessible

2. LIFO Principle (Last In First Out)

Meaning

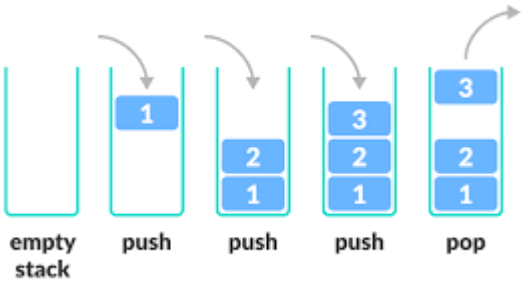
The element inserted **last** is removed **first**.

Real-Life Example

- Stack of plates
- Undo / Redo operation

- Browser back button

Diagram



3. Stack Operations

Basic Stack Operations

Operation	Description
push	Insert element
pop	Remove top element
peek	View top element
isEmpty	Check if stack is empty
isFull	Check if stack is full

Industry Uses of Stack

---

1) Function Call Management (Call Stack)

Industry use:

- All programming languages (C, C++, Java, Python)

How it is used:

- Each function call is pushed onto stack
- Stores:
    - Local variables
    - Function parameters
    - Return address

- When function ends, it is popped

Used in:

- Compilers
- Operating Systems
- Backend applications

---

2)Undo / Redo Operations

Industry use:

- Text editors (VS Code, MS Word)
- Design tools (Photoshop, Figma)

How it works:

- User actions are pushed onto stack
- Undo → pop last action
- Redo → push again

Used in:

- IDEs
- Graphic design software

---

3) Expression Evaluation

Industry use:

- Compilers & Interpreters

Examples:

- Infix → Postfix conversion
- Postfix expression evaluation

Used in:

- Calculator apps
- Programming language compilers

---

4) Syntax Checking & Parsing

Industry use:

- Code editors & compilers

How stack helps:

- Checks balanced parentheses { } ( ) [ ]
- Detects syntax errors

Used in:

- IDEs
- Code analyzers

---

5) Browser Navigation

Industry use:

- Web browsers (Chrome, Firefox)

How it works:

- Visited pages pushed onto stack
- Back button → pop page
- Forward button → separate stack

Used in:

- Web applications
- Mobile browsers

---

6) Memory Management

Industry use:

- Operating Systems

How stack is used:

- Stack memory allocation for local variables
- Automatic memory cleanup after function return

Used in:

- System programming
- Embedded systems

---

7) Recursion Handling

Industry use:

- Algorithms & problem-solving

How stack helps:

- Each recursive call stored on stack
- Prevents loss of execution state

Used in:

- Tree traversals
- Divide & conquer algorithms

---

8) Backtracking Algorithms

Industry use:

- Artificial Intelligence
- Game development

Examples:

- Maze solving
- Sudoku solver

Used in:

- AI engines
- Game logic

---

9) Thread Execution & Context Switching

Industry use:

- Operating Systems

How stack helps:

- Stores execution context of threads
- Helps in task switching

Used in:

- Multitasking systems

---

10) Data Reversal

Industry use:

- Applications & utilities

Examples:

- Reversing strings
- Reversing data streams

Used in:

- Text processing tools

4. Array-Based Stack Implementation

Representation

int stack[MAX];

int top = -1;

- top = -1 → stack is empty
- Fixed size stack

5. PUSH Operation

Purpose

Inserts an element into the stack.

Algorithm

1. Check if top == MAX - 1
2. If true → Stack Overflow
3. Else increment top
4. Insert element at stack[top]

```
void push(int x) {
  if (top == MAX - 1) {
    cout << "Stack Overflow";
    return;
  }
  top++;
  stack[top] = x;
}
```

Time Complexity

- **O(1)**

6. POP Operation

Purpose

Removes the top element from the stack.

Algorithm

1. Check if top == -1
2. If true → Stack Underflow
3. Else decrement top

Code

```
void pop() {
  if (top == -1) {
    cout << "Stack Underflow";
    return;
  }
  top--;
}
```

Time Complexity

- **O(1)**

---

7. PEEK Operation

Purpose

Returns the top element without removing it.

Code

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

---

8. isEmpty Operation

Purpose

Checks whether stack is empty.

Condition

top == -1

Code

```
bool isEmpty() {
  return top == -1;
}
```

---

9. isFull Operation

Purpose

Checks whether stack is full.

Condition

top == MAX - 1

Code

```
bool isFull() {
  return top == MAX - 1;
```

}

---

10. Stack Overflow & Underflow

Stack Overflow

Occurs when **push operation** is performed on a **full stack**.

Stack Underflow

Occurs when **pop operation** is performed on an **empty stack**.

Why Overflow Happens in Array Stack

- Fixed size memory
- Cannot grow dynamically

---

11. Advantages of Array-Based Stack

- Simple implementation
- Fast operations
- No extra memory for pointers

---

12. Disadvantages of Array-Based Stack

- Fixed size
- Overflow problem
- Memory wastage

- # Interview Important Points
-
  - Stack follows **LIFO** principle
  - Access only through **TOP**
  - Push & pop are **O(1)** operations
  - Array stack has **fixed size**
  - Overflow occurs when stack is full

SYLLABUS

- Linked list stack implementation
- Dynamic sizing advantages

- Prefix / Postfix evaluation (Shunting-Yard concept)

---

1. Stack Using Linked List

Definition

A **stack using linked list** is an implementation of stack where elements are stored in **nodes**, and each node points to the next node.
Insertion and deletion are performed from **one end only**, called the **TOP**.

Why Linked List for Stack

- No fixed size
- Stack grows and shrinks dynamically
- Better memory utilization

Node Structure

```
struct Node {
  int data;
  Node* next;
};
Node* top = NULL;
```

- top = NULL → stack is empty

---

2. PUSH Operation (Linked List Stack)

Steps

1. Create a new node
2. Store data in new node
3. Set newNode->next = top
4. Update top = newNode

Code

```
void push(int x) {
   Node* newNode = new Node();
   newNode->data = x;
   newNode->next = top;
   top = newNode;
}
```

Time Complexity

- **O(1)**

---

3. POP Operation (Linked List Stack)

Steps

1. Check if stack is empty
2. Store top node in temp
3. Move top to next node
4. Delete temp node

Code

```
void pop() {
   if (top == NULL) {
      cout << "Stack Underflow";
      return;
   }
   Node* temp = top;
   top = top->next;
   delete temp;
}
```

Time Complexity

- **O(1)**

---

4. Advantages of Dynamic Sizing

Dynamic Size Meaning

Stack size increases or decreases **at runtime** based on need.

Advantages

- No overflow (until memory is exhausted)
- No memory wastage
- Efficient memory usage
- Suitable for real-world applications

"Linked list stack provides dynamic memory allocation, unlike array-based stack."

---

5. Linked List Stack vs Array Stack

Feature	Array Stack	Linked List Stack
Size	Fixed	Dynamic
Overflow	Possible	Rare
Memory	Wastage	Efficient
Speed	Faster	Slightly slower

---

6. Prefix & Postfix Expressions

Infix Expression

Operator between operands

A + B

Postfix Expression

Operator after operands

AB+

Prefix Expression

Operator before operands

+AB

---

7. Why Prefix/Postfix Are Important

- No parentheses required
- Faster evaluation by computer
- Used in compilers

---

8. Shunting-Yard Algorithm (Concept)

Purpose

Converts **infix expression** into **postfix or prefix** using a stack.

Developed by **Edsger Dijkstra**.

---

9. Operator Precedence

Operator	Precedence
^	Highest
* /	Medium
+ -	Lowest

---

10. Infix to Postfix Conversion Algorithm

Steps

1. Scan infix expression left to right
2. Operand → add to output
3. '(' → push to stack
4. ')' → pop until '('
5. Operator:
    - Pop operators with higher or equal precedence
    - Push current operator

Example

Infix: A + B * C

Postfix: ABC*+

---

11. Postfix Expression Evaluation

Algorithm

1. Scan postfix expression
2. Operand → push to stack
3. Operator → pop two operands
4. Apply operator
5. Push result back

Example

Postfix: 23*54*+9-

Result: 17

---

12. Interview Important Points

✓ Linked list stack is dynamically sized
✓ Push & pop are O(1)
✓ No overflow issue in linked list stack
✓ Shunting-yard converts infix to postfix
✓ Postfix evaluation uses stack

SYLLABUS

- Balanced parentheses validation
- Expression evaluation (infix to postfix)
- Function call stack simulation

STACKS

1. Balanced Parentheses Validation

Problem Statement

Check whether the given expression has **balanced brackets**.

Valid Examples

() ✓

([]) ✓

{[()]} ✓

Invalid Examples

(] ✗

([)] ✗

((✗

Why Stack Is Used

- Stack follows **LIFO**
- Closing bracket must match **most recent opening bracket**

Algorithm

1. Create an empty stack
2. Scan expression from left to right
3. If opening bracket → push into stack
4. If closing bracket:
    - Stack empty → invalid
    - Pop and check matching
5. After scanning:
    - Stack empty → balanced
    - Else → unbalanced

Code (C++)

```
bool isBalanced(string exp) {
   stack<char> s;


   for(char ch : exp) {
      if(ch == '(' || ch == '{' || ch == '[')
         s.push(ch);
      else {
         if(s.empty()) return false;
         char top = s.top();
         s.pop();
         if((ch == ')' && top != '(') ||
            (ch == '}' && top != '{') ||
            (ch == ']' && top != '['))
            return false;
      }
   }
   return s.empty();
}
```

Time Complexity

- **O(n)**

---

2. Expression Evaluation (Infix to Postfix)

Expression Types

Type	Example
Infix	A + B

STACKS

Type	Example
Postfix	AB+
Prefix	+AB

Why Convert Infix to Postfix

- No brackets needed
- Easier for computers to evaluate
- Used in compilers

Operator Precedence

^ (highest)

* /

+ -

Algorithm (Infix → Postfix)

1. Scan infix expression
2. Operand → add to postfix
3. '(' → push to stack
4. ')' → pop until '('
5. Operator:
    - Pop higher/equal precedence operators
    - Push current operator

Example

Infix: A + B * C

Postfix: ABC*+

Postfix Evaluation Steps

1. Scan postfix expression
2. Operand → push to stack
3. Operator → pop two operands
4. Perform operation
5. Push result back

Industry Use:

- Compilers
- Calculator applications

---

3. Function Call Stack Simulation

What Is Call Stack

A **call stack** is a stack data structure used to manage **function calls** in a program.

What Each Stack Frame Stores

- Function parameters
- Local variables
- Return address

Example Code

```
void fun1() {
   cout << "Hello";
}


void fun2() {
   fun1();
   cout << "Hi";
```

```
}


int main() {
   fun2();
}
```

Execution Order

main()

└ fun2()

└ fun1()

- Function calls → push onto stack
- Function returns → pop from stack

Why Call Stack Is Important

- Supports recursion
- Maintains execution order
- Manages memory automatically

“Every function call creates a stack frame in the call stack.”

Advantages of Using Stack in Applications

- Efficient memory management
- Simple and fast operations
- Essential for program execution