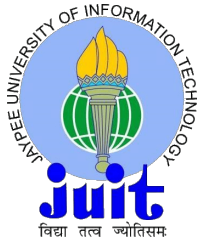


Data Structure

(Course Code: 25B11CI318)

Stack

Mr. Sandeep Kumar Patel

Assistant Professor

Jaypee University of Information Technology

Waknaghat, Solan, HP-173234

Introduction to Stack

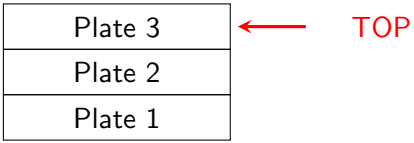
Definition

A **Stack** is a linear data structure in which insertion and deletion are performed only at one end, called the **TOP**.

- Stack follows the **LIFO** principle.
- LIFO stands for **Last In, First Out**.
- The most recently inserted element is removed first.
- Both insertion and deletion take place at the **TOP**.

Insertion $\rightarrow$ PUSH Deletion $\rightarrow$ POP

Stack: Real-Life Example

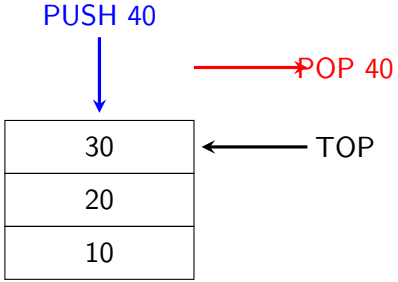


Example: Stack of Plates

The plate placed **last** is the first plate that can be removed.

Last In $\longrightarrow$ First Out

LIFO Principle



$10 \rightarrow 20 \rightarrow 30 \rightarrow 40$

40 is removed first

Abstract Data Type (ADT) of Stack

Stack ADT

A **Stack ADT** defines the data and operations of a stack without specifying how the stack is implemented.

Basic Stack Operations:

1. **Push(x)** – Insert element x at the TOP.

Abstract Data Type (ADT) of Stack

Stack ADT

A **Stack ADT** defines the data and operations of a stack without specifying how the stack is implemented.

Basic Stack Operations:

1. **Push(x)** – Insert element x at the TOP.
2. **Pop()** – Remove and return the TOP element.

Abstract Data Type (ADT) of Stack

Stack ADT

A **Stack ADT** defines the data and operations of a stack without specifying how the stack is implemented.

Basic Stack Operations:

1. **Push(x)** – Insert element x at the TOP.
2. **Pop()** – Remove and return the TOP element.
3. **Peek()** – Return the TOP element without removing it.

Abstract Data Type (ADT) of Stack

Stack ADT

A **Stack ADT** defines the data and operations of a stack without specifying how the stack is implemented.

Basic Stack Operations:

1. **Push(x)** – Insert element x at the TOP.
2. **Pop()** – Remove and return the TOP element.
3. **Peek()** – Return the TOP element without removing it.
4. **isEmpty()** – Check whether the stack is empty.

Abstract Data Type (ADT) of Stack

Stack ADT

A **Stack ADT** defines the data and operations of a stack without specifying how the stack is implemented.

Basic Stack Operations:

1. **Push(x)** – Insert element x at the TOP.
2. **Pop()** – Remove and return the TOP element.
3. **Peek()** – Return the TOP element without removing it.
4. **isEmpty()** – Check whether the stack is empty.
5. **isFull()** – Check whether the stack is full (mainly for array implementation).

Stack ADT: Formal View

Stack ADT

A stack can be represented as:

$$S = (D, O)$$

where

- D = Set of data elements
- O = Set of operations

Operations:

Push(S, x)	$\rightarrow$ Insert x at TOP
Pop(S)	$\rightarrow$ Delete TOP element
Peek(S)	$\rightarrow$ Return TOP element
isEmpty(S)	$\rightarrow$ Check whether S is empty

Stack ADT vs Implementation

Stack ADT

Defines **what** the stack does.

- Push
- Pop
- Peek
- isEmpty
- isFull

Implementation

Defines **how** the stack is stored.

- Using Array
- Using Linked List

Key Idea

ADT tells WHAT; Implementation tells HOW.

Basic Operations of Stack

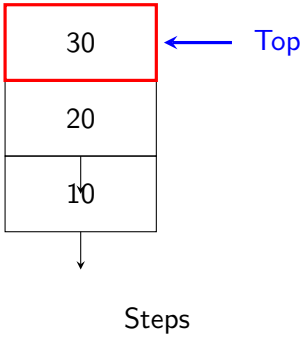
Operation	Meaning	End Used
Push(x)	Insert an element	TOP
Pop()	Delete an element	TOP
Peek()	View TOP element	TOP
isEmpty()	Check empty stack	–
isFull()	Check full stack	–

Important

All insertion and deletion operations are performed only at the **TOP** of the stack.

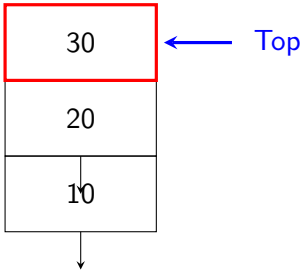
Push Operation

Push **30** onto the stack.



Push Operation

Push **30** onto the stack.

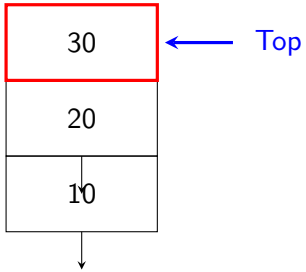


Steps

1. Create a new node.

Push Operation

Push **30** onto the stack.

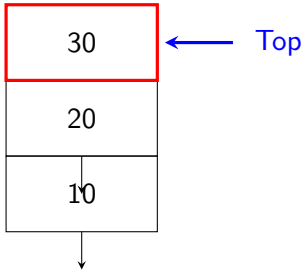


Steps

1. Create a new node.
2. Store the value in the new node.

Push Operation

Push **30** onto the stack.

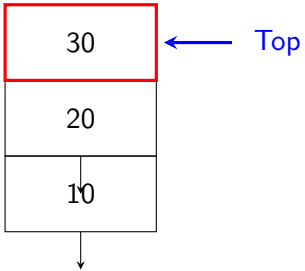


Steps

1. Create a new node.
2. Store the value in the new node.
3. New node points to the current top.

Push Operation

Push **30** onto the stack.



Steps

1. Create a new node.
2. Store the value in the new node.
3. New node points to the current `top`.
4. Make the new node the new `top`.

Push Operation

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

newNode->data = value;

newNode->next = top;

top = newNode;
```

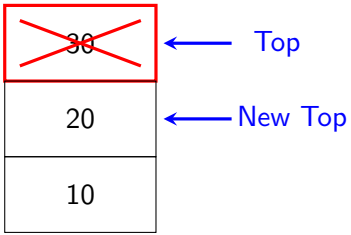
Push Operation

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

newNode->data = value;

newNode->next = top;

top = newNode;
```

Time Complexity $= O(1)$

Pop Operation

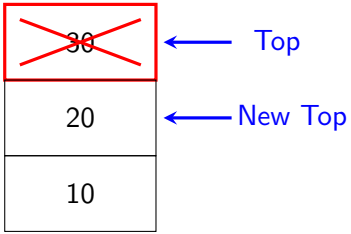
Remove the top element **30**.



Steps

Pop Operation

Remove the top element **30**.

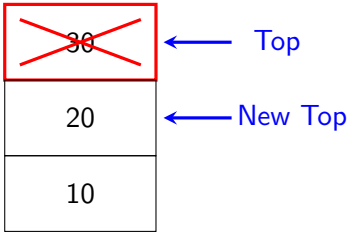


Steps

1. Check whether the stack is empty.

Pop Operation

Remove the top element **30**.

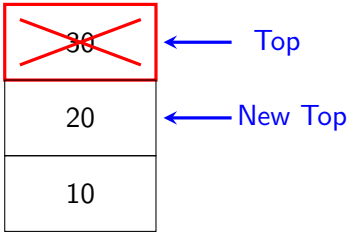


Steps

1. Check whether the stack is empty.
2. Store the top node in a temporary pointer.

Pop Operation

Remove the top element **30**.

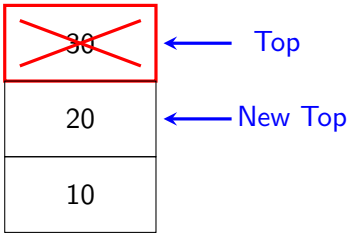


Steps

1. Check whether the stack is empty.
2. Store the top node in a temporary pointer.
3. Move `top` to the next node.

Pop Operation

Remove the top element **30**.



Steps

1. Check whether the stack is empty.
2. Store the top node in a temporary pointer.
3. Move `top` to the next node.
4. Free the old top node.

Pop Operation

```
if(top == NULL)
{
    printf("Stack Underflow");
}
else
{
    temp = top;
    top = top->next;
    free(temp);
}
```

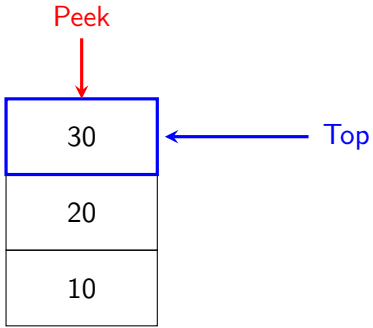
Pop Operation

```
if(top == NULL)
{
    printf("Stack Underflow");
}
else
{
    temp = top;
    top = top->next;
    free(temp);
}
```

$$\boxed{\text{Time Complexity } = O(1)}$$

Peek Operation

View the top element without removing it.



Top Element $= 30$

Peek Operation

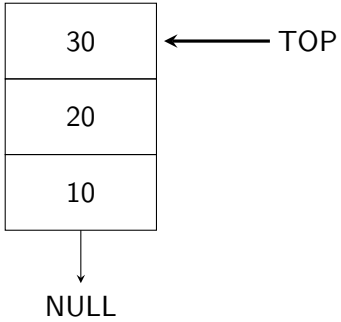
```
if(top == NULL)
{
    printf("Stack is Empty");
}
else
{
    printf("Top = %d", top->data);
}
```

Peek Operation

```
if(top == NULL)
{
    printf("Stack is Empty");
}
else
{
    printf("Top = %d", top->data);
}
```

Time Complexity $= O(1)$

Stack Using Linked List



LIFO Principle

Last In, First Out (LIFO)

Push: $O(1)$	Pop: $O(1)$	Peek: $O(1)$

Basic Operations of Stack

Operation	Meaning	End Used
Push(x)	Insert an element	TOP
Pop()	Delete an element	TOP
Peek()	View TOP element	TOP
isEmpty()	Check empty stack	–
isFull()	Check full stack	–

Important

All insertion and deletion operations are performed only at the **TOP** of the stack.

Question 1: Stack

Question:
A stack is initially empty. The following operations are performed:

$$\text{PUSH}(10), \quad \text{PUSH}(20), \quad \text{PUSH}(30), \quad \text{POP}(), \quad \text{PUSH}(40), \quad \text{POP}()$$

Question 1: Stack

Question:
A stack is initially empty. The following operations are performed:

$$\text{PUSH}(10),\quad \text{PUSH}(20),\quad \text{PUSH}(30),\quad \text{POP}(),\quad \text{PUSH}(40),\quad \text{POP}()$$

Find:

1. The element removed by each `POP()` operation.
2. The final contents of the stack.
3. The element at the TOP.

Solution: Question 1

Step-by-step execution:

Operation	Stack	TOP
PUSH(10)	10	10
PUSH(20)	10, 20	20
PUSH(30)	10, 20, 30	30
POP()	10, 20	20
PUSH(40)	10, 20, 40	40
POP()	10, 20	20

Solution: Question 1

Step-by-step execution:

Operation	Stack	TOP
PUSH(10)	10	10
PUSH(20)	10, 20	20
PUSH(30)	10, 20, 30	30
POP()	10, 20	20
PUSH(40)	10, 20, 40	40
POP()	10, 20	20

Answer:

- First `POP()` removes **30**.
- Second `POP()` removes **40**.
- Final stack: 10, 20
- TOP element: 20

Question 2: Stack

Question:
A stack contains the following elements:

$$5, \quad 10, \quad 15, \quad 20$$

where **20 is at the TOP**.
The following operations are performed:

$$\text{POP}(), \quad \text{POP}(), \quad \text{PUSH}(25), \quad \text{PUSH}(30), \quad \text{POP}()$$

Question 2: Stack

Question:
A stack contains the following elements:

$$5, \quad 10, \quad 15, \quad 20$$

where **20 is at the TOP**.
The following operations are performed:

$$\text{POP}(), \quad \text{POP}(), \quad \text{PUSH}(25), \quad \text{PUSH}(30), \quad \text{POP}()$$

Find:

1. Which elements are removed?
2. What is the final stack?
3. Which element is at the TOP?

Solution: Question 2

Step-by-step execution:

Operation	Stack	TOP
Initial	5, 10, 15, 20	20
POP()	5, 10, 15	15
POP()	5, 10	10
PUSH(25)	5, 10, 25	25
PUSH(30)	5, 10, 25, 30	30
POP()	5, 10, 25	25

Solution: Question 2

Step-by-step execution:

Operation	Stack	TOP
Initial	5, 10, 15, 20	20
POP()	5, 10, 15	15
POP()	5, 10	10
PUSH(25)	5, 10, 25	25
PUSH(30)	5, 10, 25, 30	30
POP()	5, 10, 25	25

Answer:

- Elements removed: 20, 15, 30
- Final stack: 5, 10, 25
- TOP element: 25

Applications of Stack

Why Stack?

A stack is useful whenever a problem requires processing elements in the **Last In, First Out (LIFO)** order.

Major applications of stacks:

1. Function calls and recursion
2. Expression evaluation
3. Expression conversion
4. Parentheses matching
5. Undo/Redo operations
6. Browser history
7. Backtracking
8. Depth First Search (DFS)

1: Function Calls and Recursion

When a function is called, information about the function is stored in the **call stack**. Example:

```
1 void fun1()
2 { fun3();}
3 void fun2()
4 { fun3(); }
5 void fun3()
6 { printf("Hello");}
```

1: Function Calls and Recursion

When a function is called, information about the function is stored in the **call stack**. Example:

```
1 void fun1()
2 { fun3();}
3 void fun2()
4 { fun3(); }
5 void fun3()
6 { printf("Hello");}
```

Function calls are stored as:

$$fun1 \rightarrow fun2 \rightarrow fun3$$

1: Function Calls and Recursion

When a function is called, information about the function is stored in the **call stack**. Example:

```
1 void fun1()
2 { fun3();}
3 void fun2()
4 { fun3(); }
5 void fun3()
6 { printf("Hello");}
```

Function calls are stored as:

$$\boxed{fun1 \rightarrow fun2 \rightarrow fun3}$$

When `fun3()` finishes:

$$fun3 \rightarrow fun2 \rightarrow fun1$$

Key Idea

The most recently called function finishes first. Therefore, function calls follow the **LIFO** principle.

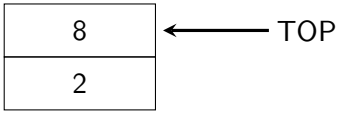
2: Expression Evaluation

Stacks are widely used to evaluate arithmetic expressions.

Example:

$$(5 + 3) \times 2$$

The stack can temporarily store:



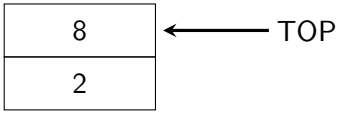
2: Expression Evaluation

Stacks are widely used to evaluate arithmetic expressions.

Example:

$$(5 + 3) \times 2$$

The stack can temporarily store:



Stacks are used in:

- Postfix expression evaluation
- Prefix expression evaluation
- Operator and operand processing

3: Expression Conversion

Stacks are used to convert expressions between different forms. **Common forms:**

- Infix
- Prefix
- Postfix

Example:

$$A + B \times C$$

3: Expression Conversion

Stacks are used to convert expressions between different forms. **Common forms:**

- Infix
- Prefix
- Postfix

Example:

$$A + B \times C$$

Infix:

$$A + B \times C$$

3: Expression Conversion

Stacks are used to convert expressions between different forms. **Common forms:**

- Infix
- Prefix
- Postfix

Example:

$$A + B \times C$$

Infix: $A + B \times C$

Postfix: $ABC \times +$

3: Expression Conversion

Stacks are used to convert expressions between different forms. **Common forms:**

- Infix
- Prefix
- Postfix

Example:

$$A + B \times C$$

Infix: $A + B \times C$

Postfix: $ABC \times +$

Prefix: $+A \times BC$

Key Idea

Operators are temporarily stored in a stack according to their precedence and associativity.

4: Parentheses Matching

Stacks can be used to check whether parentheses are balanced. Example:

$$(A + B) \times (C - D)$$

4: Parentheses Matching

Stacks can be used to check whether parentheses are balanced. Example:

$$(A + B) \times (C - D)$$

Processing:

Symbol	Stack Action
(	PUSH
A	Ignore
+	Ignore
B	Ignore
)	POP
(	PUSH
C	Ignore
-	Ignore
D	Ignore
)	POP

4: Parentheses Matching

Stacks can be used to check whether parentheses are balanced. Example:

$$(A + B) \times (C - D)$$

Processing:

Symbol	Stack Action
(	PUSH
A	Ignore
+	Ignore
B	Ignore
)	POP
(	PUSH
C	Ignore
-	Ignore
D	Ignore
)	POP

Since the stack is empty at the end: Balanced Expression

5: Undo and Redo

Stacks are used to implement **Undo** and **Redo** operations in text editors and software applications.

Suppose a user performs:

$$A \rightarrow B \rightarrow C$$

The operations are stored in an Undo stack.

5: Undo and Redo

Stacks are used to implement **Undo** and **Redo** operations in text editors and software applications.

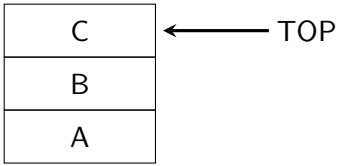
Suppose a user performs:

$$A \rightarrow B \rightarrow C$$

The operations are stored in an Undo stack.

$$\text{Undo} \Rightarrow C$$

$$\text{Undo} \Rightarrow B$$



6: Browser History

A browser can use a stack-like structure to manage previously visited pages.
Suppose the user visits:

$$\text{Google} \rightarrow \text{YouTube} \rightarrow \text{Wikipedia} \rightarrow \text{GitHub}$$

6: Browser History

A browser can use a stack-like structure to manage previously visited pages.
Suppose the user visits:

$$\text{Google} \rightarrow \text{YouTube} \rightarrow \text{Wikipedia} \rightarrow \text{GitHub}$$

Pressing **Back** removes the most recently visited page.

$$\text{Back} \Rightarrow \text{GitHub}$$

$$\text{Back} \Rightarrow \text{Wikipedia}$$

6: Browser History

A browser can use a stack-like structure to manage previously visited pages.
Suppose the user visits:

$$\text{Google} \rightarrow \text{YouTube} \rightarrow \text{Wikipedia} \rightarrow \text{GitHub}$$

Pressing **Back** removes the most recently visited page.

$$\text{Back} \Rightarrow \text{GitHub}$$

$$\text{Back} \Rightarrow \text{Wikipedia}$$

Therefore:

Last visited page is accessed first

7: Backtracking

Stack is used in problems where we need to:

- Make a choice.
- Continue with that choice.
- Return to the previous choice if it fails.

Examples:

- Maze solving
- N-Queens problem
- Sudoku solving
- Path finding

7: Backtracking

Stack is used in problems where we need to:

- Make a choice.
- Continue with that choice.
- Return to the previous choice if it fails.

Examples:

- Maze solving
- N-Queens problem
- Sudoku solving
- Path finding

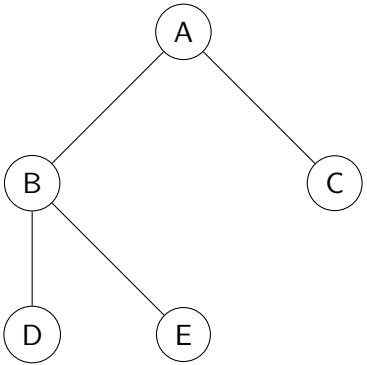
Basic Idea

Store the choices in a stack. When a dead end is reached, **pop** the most recent choice and try another option.

8: Depth First Search (DFS)

Stack is used to implement **Depth First Search (DFS)** in graphs and trees.

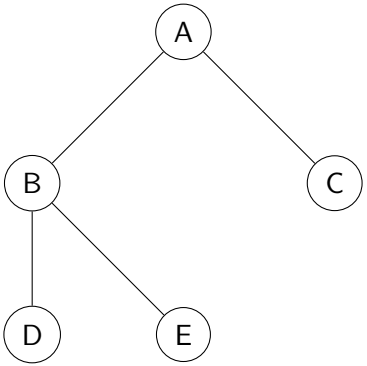
DFS explores one path as deeply as possible before backtracking.



8: Depth First Search (DFS)

Stack is used to implement **Depth First Search (DFS)** in graphs and trees.

DFS explores one path as deeply as possible before backtracking.



DFS can be implemented using:

Applications of Stack: Summary

Application	Purpose of Stack
Function Calls	Manage execution order
Recursion	Store function call information
Expression Evaluation	Store operands/operators
Expression Conversion	Manage operators
Parentheses Matching	Match opening/closing brackets
Undo/Redo	Reverse recent operations
Browser History	Access recent pages
Backtracking	Return to previous choices
DFS	Explore graph/tree paths

Remember

Whenever a problem naturally requires **Last In, First Out**, think of a **Stack**.

Thank You

Questions?