Stacks

Introduction to Stack : What is it ?

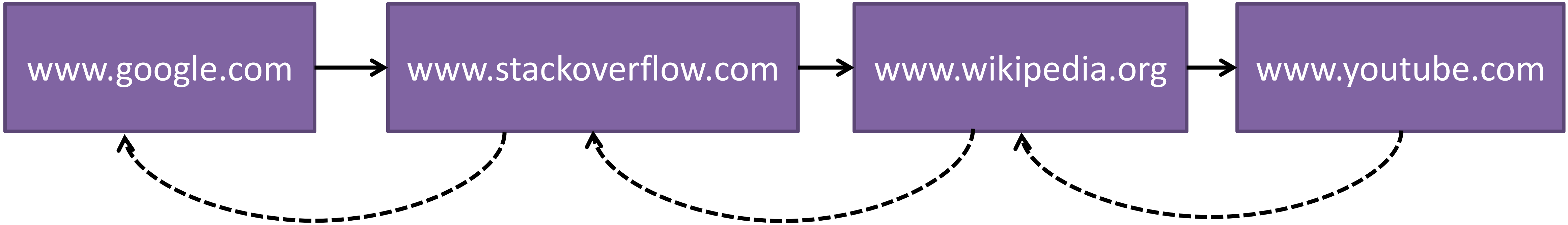
- Linear data structure
- Follows the *****LIFO (Last In First Out)***** principle.
  - The element inserted last is the first one to be removed.
- Think of it like a stack of plates:-
  - You can only add a plate on top
  - You can only remove the plate from the top
  - The last plate placed is the first one to be removed

Stack : Key characteristics

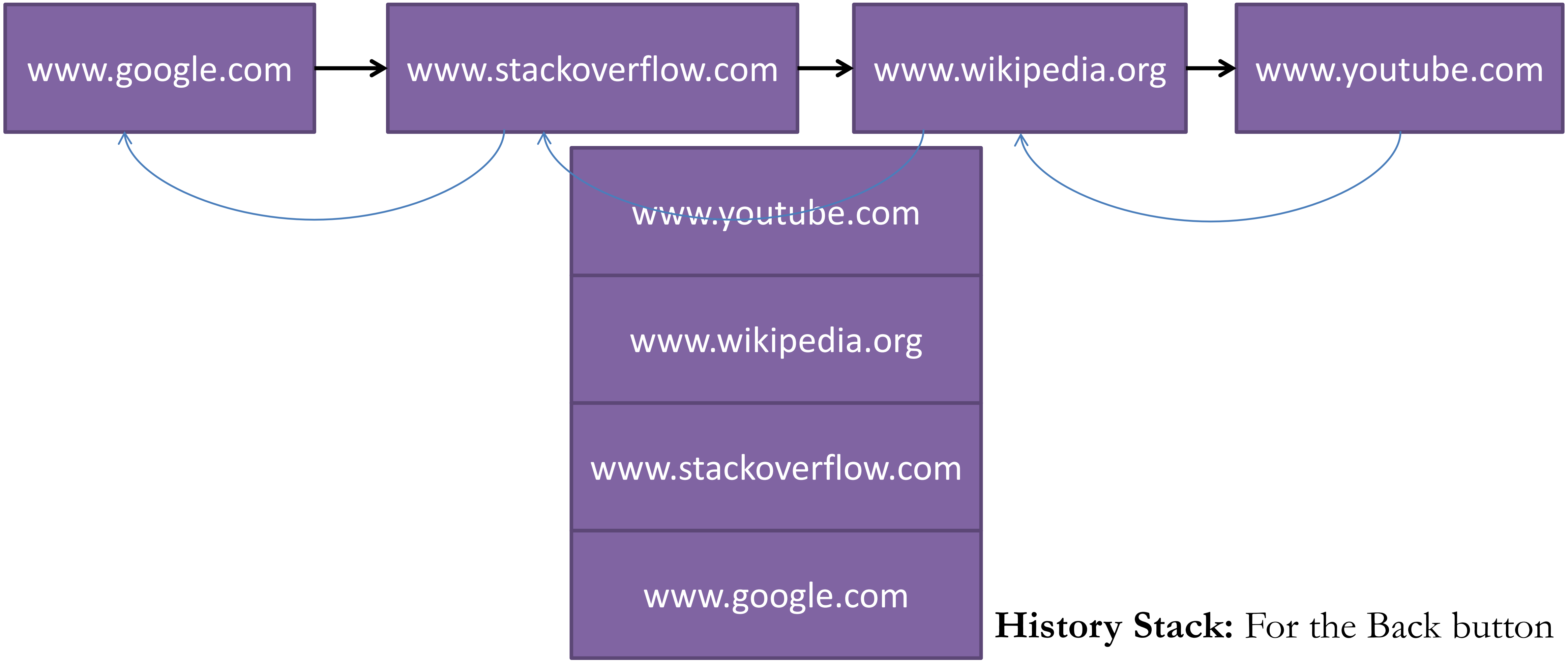
- LIFO ordering : Last in, First out
- Limited Access : Can only access the top element
- Dynamic Size : Can grow and shrink as needed
- Uses a ‘top’ pointer to track the last element

Stack : Real life examples

- Browser History Back Button



Stack : Real life examples



History Stack: For the Back button

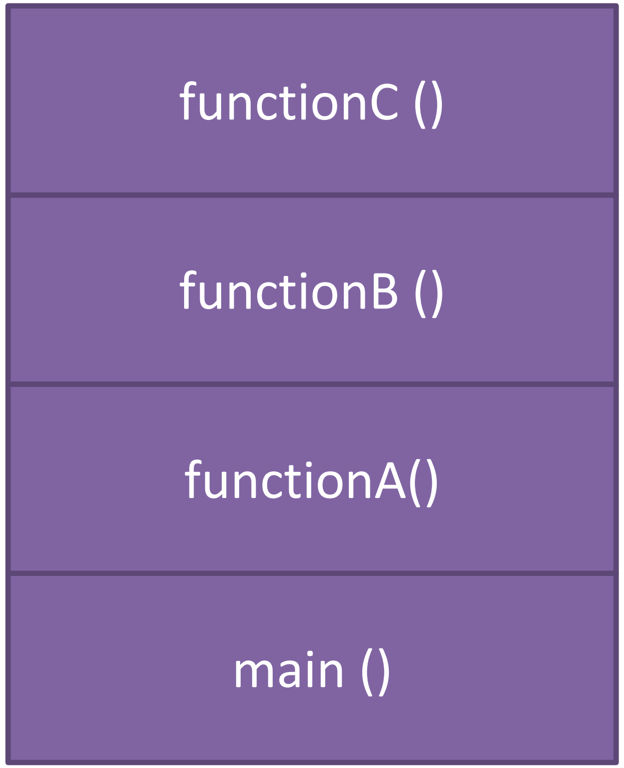
Stack : Real life examples

- Browser History Back Button| When you browse websites:
  - Each new page you visit is ****pushed**** onto the stack
  - Clicking "Back" button ****pops**** the current page
  - You go back to the previous page (second from top)
- Undo/Redo Functionality| Text editors use stacks for undo operations:
  - Each action (typing, deleting) is pushed onto the undo stack
  - Pressing Ctrl+Z pops the last action and reverses it
  - Redo uses another stack to store undone actions

Stack : Real life examples

Call Stack in Programming | When functions call other functions:

```
main() calls function A
  → function A calls function B
    → function B calls function C
      → function C completes (pops from stack)
    → function B completes (pops from stack)
  → function A completes (pops from stack)
→ back to main()
```



Stack : Real life examples

- Pile of Books
  - You can only add a book on **top**
  - You can only take the **top** book
  - The last book placed is the first one you'll pick
- Stack of Plates in the Cafeteria
  - Clean plates are stacked one on **top** of another
  - Customers take the **top** plate
  - Staff adds new plates on **top**
  - Last plate added = First plate taken

Stack : Primary operations

1. **push(element)**: Add element to the top
2. **pop()**: Remove and return the top element
3. **peek() / top()**: View the top element without removing
4. **isEmpty()**: Check if stack is empty
5. **isFull()**: Check if stack is full (for fixed-size stacks)
6. **size()**: Return number of elements

Stack implementation using Array

Initial State (Empty Stack)

```
maxSize = 5
top = -1 (indicates empty stack)

Index:  [0]  [1]  [2]  [3]  [4]
Array:  [ ]  [ ]  [ ]  [ ]  [ ]
         ↑
        top = -1
```

Operation 1: push(10)

```
top = 0

Index:  [0]  [1]  [2]  [3]  [4]
Array:  [10] [ ]  [ ]  [ ]  [ ]
         ↑
        top
```

Operation 2: push(20)

```
top = 1

Index:  [0]  [1]  [2]  [3]  [4]
Array:  [10] [20] [ ]  [ ]  [ ]
              ↑
             top
```

Operation 3: push(30)

```
top = 2

Index:  [0]  [1]  [2]  [3]  [4]
Array:  [10] [20] [30] [ ]  [ ]
                   ↑
                  top
```

Stack implementation using Array

Operation 4: push(40)

```
top = 3

Index:  [0]  [1]  [2]  [3]  [4]
Array:  [10] [20] [30] [40] [ ]
                         ↑
                        top
```

Operation 5: push(50) - Stack Full

```
top = 4

Index:  [0]  [1]  [2]  [3]  [4]
Array:  [10] [20] [30] [40] [50]
                              ↑
                             top
Status: FULL (top == maxSize - 1)
```

Operation 6: push(60) - Stack Overflow

```
Index:  [0]  [1]  [2]  [3]  [4]
Array:  [10] [20] [30] [40] [50]
                              ↑
                           top = 4

❌ Cannot push! Stack is FULL
   isFull() returns true
```

Stack implementation using Array

Operation 7: pop() - Returns 50

```
Before:
Index:  [0]  [1]  [2]  [3]  [4]
Array:  [10] [20] [30] [40] [50]
                             ↑
                          top = 4

After:
Index:  [0]  [1]  [2]  [3]  [4]
Array:  [10] [20] [30] [40] [ ]
                        ↑
                     top = 3
Returned: 50
```

Operation 8: pop() - Returns 40

```
Index:  [0]  [1]  [2]  [3]  [4]
Array:  [10] [20] [30] [ ]  [ ]
                   ↑
                top = 2
Returned: 40
```

Operation 9: peek() - Returns 30 (without removal)

```
Index:  [0]  [1]  [2]  [3]  [4]
Array:  [10] [20] [30] [ ]  [ ]
                   ↑
                top = 2
Returned: 30 (top not changed)
```

Stack implementation using Array

isEmpty() Check

```
Empty Stack (top == -1):
Index:  [0]  [1]  [2]  [3]  [4]
Array:  [ ]  [ ]  [ ]  [ ]  [ ]
         ↑
      top = -1
isEmpty() → true ✓

Non-Empty Stack (top >= 0):
Index:  [0]  [1]  [2]  [3]  [4]
Array:  [10] [20] [30] [ ]  [ ]
                    ↑
                top = 2
isEmpty() → false ×
```

isFull() Check

```
Full Stack (top == maxSize - 1):
Index:  [0]  [1]  [2]  [3]  [4]
Array:  [10] [20] [30] [40] [50]
                              ↑
                          top = 4
isFull() → true ✓

Not Full (top < maxSize - 1):
Index:  [0]  [1]  [2]  [3]  [4]
Array:  [10] [20] [30] [ ]  [ ]
                    ↑
                top = 2
isFull() → false ×
```

Stack using Array : Key properties

```
Stack Properties
---------------------------------------
• LIFO: Last In First Out
• top = -1: Empty stack
• top = maxSize - 1: Full stack
• Push: array[++top] = element
• Pop: return array[top--]
• Peek: return array[top]
• isEmpty: top == -1
• isFull: top == maxSize - 1
```

Stack Using Array: Implementation

```java
class StackUsingArray {
    private int maxSize;
    private int[] stackArray;
    private int top;

    // Constructor
    public StackUsingArray(int size) {
        this.maxSize = size;
        this.stackArray = new int[maxSize];
        this.top = -1; // Stack is empty
    }

    // Push operation
    public void push(int value) {
        if (isFull()) {
            System.out.println("Stack Overflow! Cannot push " + value);
            return;
        }
        stackArray[++top] = value;
        System.out.println("Pushed: " + value);
    }

    // Pop operation
    public int pop() {
        if (isEmpty()) {
            System.out.println("Stack Underflow! Cannot pop");
            return -1;
        }
        int value = stackArray[top--];
        System.out.println("Popped: " + value);
        return value;
    }
```

```java
// Peek operation
    public int peek() {
        if (isEmpty()) {
            System.out.println("Stack is empty!");
            return -1;
        }
        return stackArray[top];
    }

    // Check if empty
    public boolean isEmpty() {
        return (top == -1);
    }

    // Check if full
    public boolean isFull() {
        return (top == maxSize - 1);
    }

    // Get size
    public int size() {
        return top + 1;
    }

    // Display stack
    public void display() {
        if (isEmpty()) {
            System.out.println("Stack is empty!");
            return;
        }
        System.out.print("Stack: ");
        for (int i = 0; i <= top; i++) {
            System.out.print(stackArray[i] + " ");
        }
        System.out.println("<- Top: " + stackArray[top]);
    }
}
```

Stack using Array : Time complexity

Operation	Time
push()	O(1)
pop()	O(1)
peek()	O(1)
isEmpty()	O(1)
isFull()	O(1)

The in-built Stack class in Java

- Java provides a built-in Stack class in java.util

```java
import java.util.Stack;

public class Demo {
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();

        stack.push(10);
        stack.push(20);
        stack.push(30);

        System.out.println(stack.pop());  // 30
        System.out.println(stack.peek()); // 20
    }
}
```

Why **Not** use java.util.Stack ?

- Even though it works, **Stack is considered a legacy class**.
  - It extends Vector (outdates, automatically sync)
- Stack class inherits many from Vector that don’t make sense for Stack
- Not recommended in modern Java (Oracle documentation also suggests)
- Replacement – Deque (Double Ended Queue)

Deque – The double ended Queue

- Supports LIFO behavior
- Two main implementations
  - ArrayDeque
  - LinkedList
- Faster than Stack
- No unnecessary synchronization
- Pure stack behavior

```java
import java.util.Deque;

public class Demo {
    public static void main(String[] args) {
        Deque<Integer> stack = new ArrayDeque<>();

        stack.push(10);
        stack.push(20);
        stack.push(30);

        System.out.println(stack.pop());  // 30
        System.out.println(stack.peek()); // 20
    }
}
```

Infix to Postfix Conversion

- Imagine you're a computer trying to evaluate this expression: 3 + 5 * 2
  - You read from left to right.
  - You see 3, then +, then 5. Should you add them?
  - But how would you know? You haven't seen the * yet!
- This is exactly why computers need postfix notation
- In postfix, operators come AFTER their operands, removing all ambiguity

  Postfix : 3 5 2 * +
- How to convert an infix expression to postfix ?

Infix to Postfix : Examples

- Basic Precedence
  - **Expression:** A + B * C
  - **Result:** ABC*+
- The Power of Parentheses
  - **Expression: (A + B) * C**
  - **Result:** AB+C*
- Complex Expression
  - **Expression: A + B * C - D / E ^ F**
  - **Result:** ABC*+DEF^/-

Infix to Postfix conversion
Why Stack is the perfect tool ?

Expression: A + B * C | **Result:** ABC*+

- Think of the stack as a ***waiting room for operators***
  - **Operators must wait** for their operands to be processed
  - **Higher priority operators** should get processed first (they "cut in line")
  - **Parentheses** create temporary "sub-rooms" where inner operations must finish first
  - Stack naturally handles **LIFO (Last In First Out)** - perfect for this hierarchy!

Infix to Postfix conversion : Algorithm

The Rules (Remember: "**SCOPE**")

- **S**can from left to right
- **C**opy operands directly to output
- **O**perators go to stack (but follow precedence rules!)
- **P**arentheses: ‘ **(** ' goes to stack, ‘ **)** ' pops until '('
- **E**mpty stack at the end

- Precedence Hierarchy

Level 3: ^ (Exponentiation - Highest)
Level 2: *, /
Level 1: +, - (Lowest)

- Golden rule for stack operations when we see an operator
  - **Pop** all operators from stack that have **equal or higher precedence**
  - **Then push** the current operator

Infix to Postfix *SCOPE* in Action

Expression: A + B * C

Step	Symbol	Stack	Output	Explanation
1	A	[ ]	A	Operand → directly to output
2	+	[+]	A	Operator → push to stack
3	B	[+]	AB	Operand → directly to output
4	*	[+, *]	AB	* has higher precedence than +, so push
5	C	[+, *]	ABC	Operand → directly to output
6	END	[ ]	ABC*+	Pop all: * first, then +

Result: ABC*+

Notice how * stayed on top of + in the stack. That's because * has higher precedence. When we empty the stack at the end, * comes out first (LIFO), appearing before + in the output!

Infix to Postfix *SCOPE* in Action

Expression: (A + B) * C

Step	Symbol	Stack	Output	Explanation
1	(	[(]		Opening parenthesis → push to stack
2	A	[(]	A	Operand → directly to output
3	+	[(, +]	A	Operator → push to stack
4	B	[(, +]	AB	Operand → directly to output
5	)	[ ]	AB+	Pop until '(' → + comes out, discard '('
6	*	[*]	AB+	Operator → push to stack
7	C	[*]	AB+C	Operand → directly to output
8	END	[ ]	AB+C*	Pop remaining: *

Result: AB+C*

Parentheses create a "barrier" in the stack. When we hit ')', we pop everything until we find '('. This ensures the expression inside parentheses is evaluated first, regardless of precedence

Infix to Postfix *SCOPE* in Action

Expression: A + B * C - D / E ^ F | **Result:** ABC*+DEF^/-

Step	Symbol	Stack	Output	Explanation
1	A	[ ]	A	Operand → output
2	+	[+]	A	Stack empty → push
3	B	[+]	AB	Operand → output
4	*	[+, *]	AB	* higher than + → push
5	C	[+, *]	ABC	Operand → output
6	-	[-]	ABC*+	- same as +, pop *, pop +, push -
7	D	[-]	ABC*+D	Operand → output
8	/	[-, /]	ABC*+D	/ higher than - → push
9	E	[-, /]	ABC*+DE	Operand → output
10	^	[-, /, ^]	ABC*+DE	^ highest → push
11	F	[-, /, ^]	ABC*+DEF	Operand → output
12	END	[ ]	ABC*+DEF^/-	Pop all: ^, then /, then -

When we encounter '-' (step 6), we pop both * and + because they have higher or equal precedence

Time & Space Complexity

- **Time Complexity**
  - O(n) where n is the length of infix expression
  - Single pass through the expression
  - Each character pushed/popped at most once
- **Space Complexity**
  - O(n) Stack can contain at most n operators
  - Output string also has n characters

Real-World Applications

- Some Programming language Compilers and Interpreters
- Virtual Machines(e.g. JVM) use postfix-style evaluation
- Mathematical expression libraries(exprtk (C++), muParser(C++),tinyexpr(C) , mathjs(JS) use infix to postfix and evaluate using stack

Infix to Postfix : The Big picture

- A * B + C / D → AB*CD/+
- (A + B) * (C - D) → AB+CD-*
- A ^ B ^ C → ABC^^ (right associative: A^(B^C))
- ((A + B) * C - D) / E → AB+C*D-E/

- Stack(LIFO) handles precedence and nesting
- Operands go out, operators wait based on precedence
- Calculators, compilers and expression evaluators uses this.

Infix to Postfix : Common mistakes to avoid

- **Forgetting to pop operators at the end** → Incomplete postfix
- **Not handling precedence correctly** → Wrong evaluation order
- **Forgetting to remove '(' when seeing ')'** → Stack overflow
- **Treating spaces as characters** → Malformed output

Using Stack for Postfix expression evaluation

3 + 5 * 2 → 3 5 2 * +

The Two Rule System:

1. **If you see a NUMBER** → Push it onto the stack
2. **If you see an OPERATOR** → Pop two numbers, apply operator, push result back

Postfix expression evaluation

- The Problem with Infix Evaluation, 3 + 5 * 2
  - ✕ Look ahead to check precedence
  - ✕ Track parentheses
  - ✕ Make multiple passes
  - ✕ Complex logic with lots of conditions
- The Beauty of Postfix Evaluation, 3 5 2 * +
  - ☑ Scan left to right (single pass!)
  - ☑ No precedence checking needed
  - ☑ No parentheses to handle
  - ☑ Simple stack operations

Example 1: Postfix evaluation

- **Equivalent Infix:** (5 + 3) * 2 | **Postfix Expression:** 5 3 + 2 *

Step	Symbol	Action	Stack	Explanation
1	5	Push	[5]	Number → push to stack
2	3	Push	[5, 3]	Number → push to stack
3	+	Pop, Operate, Push	[8]	Pop 3 and 5, calculate 5+3=8, push 8
4	2	Push	[8, 2]	Number → push to stack
5	*	Pop, Operate, Push	[16]	Pop 2 and 8, calculate 8*2=16, push 16
END	-	Result	[16]	Final answer: **16**

Notice how the addition happened FIRST (steps 1-3), then multiplication (steps 4-5)

Example 2: Postfix evaluation

- **Equivalent Infix:** 8 / 2 – 3 | **Postfix Expression:** 8 2 / 3 -

Step	Symbol	Action	Stack	Explanation
1	8	Push	[8]	Number → push to stack
2	2	Push	[8, 2]	Number → push to stack
3	/	Pop, Operate, Push	[4]	Pop 2 and 8, calculate 8/2=4, push 4
4	3	Push	[4, 3]	Number → push to stack
5	-	Pop, Operate, Push	[1]	Pop 3 and 4, calculate 4-3=1, push 1
END	-	Result	[1]	Final answer: **1**

```
Stack: [8, 2]
Pop: second = 2, first = 8
Calculate: 8 / 2 = 4  (NOT 2 / 8!)
```

Stack: [... bottom, first, second] , Pop twice: second = pop(), first = pop()
Result: **first** OPERATOR **second**

Example 3: Postfix evaluation

- **Equivalent Infix:** 5 * (6 + 2) / 4 | **Postfix Expression:** 5 6 2 + * 4 /

Step	Symbol	Action	Stack	Calculation	Explanation
1	5	Push	[5]	-	Push operand
2	6	Push	[5, 6]	-	Push operand
3	2	Push	[5, 6, 2]	-	Push operand
4	+	Pop, Operate, Push	[5, 8]	6+2=8	Pop 2,6 → add → push 8
5	*	Pop, Operate, Push	[40]	5*8=40	Pop 8,5 → multiply → push 40
6	4	Push	[40, 4]	-	Push operand
7	/	Pop, Operate, Push	[10]	40/4=10	Pop 4,40 → divide → push 10
END	-	Result	[10]	-	Final answer: **10**

See how 6+2 was evaluated first (step 4), creating the parenthesized sub-expression result, then that result (8) was used in the multiplication!

Example 4: Postfix evaluation

- **Equivalent Infix:** 2 ^ (3 ^ 4) = 2^81 | **Postfix Expression:** 2 3 4 ^ ^/

Step	Symbol	Action	Stack	Calculation
1	2	Push	[2]	-
2	3	Push	[2, 3]	-
3	4	Push	[2, 3, 4]	-
4	^	Pop, Operate, Push	[2, 81]	3^4 = 81
5	^	Pop, Operate, Push	[2417851639229258349412352]	2^81 = huge!

Postfix 2 3 4 ^ ^ automatically handles right associativity. The rightmost operation (3^4) happens first!

Note: This example produces 2^81, which has 22 digits. The double data type can only handle ~15-17 significant digits accurately. For exact results with large exponents, use BigInteger instead.

Time and Space Complexity

- **Time Complexity: O(n)**
  - Single pass through the expression
  - Each number is pushed once
  - Each number is popped once
  - Each operator does constant time work
  - Total: O(n) where n = number of tokens
- **Space Complexity: O(n)**
  - Stack can contain at most n/2 numbers (when all numbers come first)
  - Worst case: Expression like "1 2 3 4 5 + + + +"

JVM(Stack-based VM) Example

int result = (a + b) * c;

```
iload_1      // Load a
iload_2      // Load b
iadd         // Add (stack: [a+b])
iload_3      // Load c
imul         // Multiply (stack: [(a+b)*c])
istore_4     // Store result
```

Infix to Prefix Conversion Quick Reference

- Also referred to as Polish Notation
- Operators come BEFORE their operands
- Infix: A + B , Prefix: + A B , Postfix: A B +
- Prefix is the "mirror" of postfix!
- Memory Trick:
  - **PRE**fix: Operator comes **PRE**viously (before)
  - **POST**fix: Operator comes **POST**eriorly (after)

Infix, Prefix & Postfix examples

- The Pattern:

Infix	Prefix	Postfix
A + B	+ A B	A B +
A + B * C	+ A * B C	A B C * +
(A + B) * C	* + A B C	A B + C *
A * B + C	+ * A B C	A B * C +

Notice: Prefix and Postfix are **mirrors** in operator placement!

Evaluation of Prefix Expressions Quick Reference

Algorithm: Scan RIGHT to LEFT

Prefix: + * 2 3 4

Stack (***scanning right to left***):

4 → [4]

3 → [4, 3]

2 → [4, 3, 2]

* → [4, 6] (pop 2,3 → 2*3=6 → push 6)

+ → [10] (pop 6,4 → 6+4=10 → push 10)

Result: 10

Infix to Prefix Expression Quick Reference

The "Reverse-Reverse " Shortcut

Steps:

- Reverse the infix expression (swap (with))
- Convert to postfix using standard algorithm
- Reverse the result

Example

Original Infix: A + B * C

- Step 1 (Reverse): C * B + A [swap (and)]
- Step 2 (Postfix): C B * A +
- Step 3 (Reverse): + A * B C

This is PREFIX!

Some examples of where Prefix is used

- Lisp and Functional languages
- Some Vintage Calculators
- Abstract Syntax Trees(AST)

```
Expression: (a + b) * c

AST (prefix-like):
   *
  / \
 +   c
/ \
a   b

Pre-order traversal: * + a b c  (Prefix!)
```

Queues

Introduction to Queue : What is it ?

- Linear data structure
- Follows the *****FIFO (First In First Out)***** principle.
  - The element inserted first is the first one to be removed.
- Think of it like a line in a coffee shop:-
  - The first person to join the line is the first person to be served
  - New people join at the back (**rear**) of the line
  - People leave from the **front** of the line

Queue : Real life examples

- **Customer Service Hotlines** - When you call customer support, you hear "You are caller number 7 in the queue." The first caller gets helped first!
- **Printer Queue** - Sent multiple documents to print? They print in the order you sent them. First document in, first document out
- **Operating System Task Scheduling** - Your computer's CPU handles tasks in a queue. When you open multiple apps, the OS manages them using various queue strategies.

Queue : Real life examples

- **Messaging Systems** - Apps like WhatsApp use queues to deliver messages in the correct order, ensuring your conversation makes sense
- **Breadth-First Search (BFS) -** In graph algorithms, queues help us explore nodes level by level. We'll learn this later.
- **Video Streaming Buffers -** Ever notice that loading circle on YouTube? That's a queue filling up with video chunks before playing them in order

Stock Trading

- **Real Scenario:** A stock trading platform processes buy and sell orders.
- The Context:
  - **Requirement:** Must execute orders in EXACT sequence received (legal requirement)
  - **Critical System:** Financial regulations, audit trails, real money
- Order received at 09:30:00.001 → Buy 1000 shares at ₹500
- Order received at 09:30:00.002 → Sell 1000 shares at ₹498
- If order 2 executes before order 1: - Wrong sequence - Legal violation - Financial loss - Regulatory penalty

Stock Trading Example: Solution

- Use a **Queue data structure** (First-In-First-Out):
  - Order comes in → Goes to back of queue
  - Order executes → Taken from front of queue
  - Guaranteed sequence preservation

Stock Trade : Queue

```
TIME: 09:30:00.001 - Order arrives
┌────────────────────────────────────────────┐
│ QUEUE (FIFO - First In, First Out)         │
├────────────────────────────────────────────┤
│ FRONT → [Buy 1000 shares @ ₹500] ← BACK    │
└────────────────────────────────────────────┘
     ↓ Process (Execute Trade)
     ✓ Executed at 09:30:00.002

TIME: 09:30:00.003 - New order arrives
┌────────────────────────────────────────────┐
│ QUEUE                                      │
├────────────────────────────────────────────┤
│ FRONT → [Sell 500 shares @ ₹498] ← BACK    │
└────────────────────────────────────────────┘
     ↓ Process
     ✓ Executed at 09:30:00.004

TIME: 09:30:00.005 - Multiple orders arrive quickly
┌──────────────────────────────────────────────────────────────────────┐
│ QUEUE                                                                │
├──────────────────────────────────────────────────────────────────────┤
│ FRONT → [Buy 2000 @ ₹502] → [Sell 1500 @ ₹501] → [Buy 800 @ ₹503] ← BACK │
└──────────────────────────────────────────────────────────────────────┘
     ↓ Process in ORDER

Step 1: Execute [Buy 2000 @ ₹502]  at 09:30:00.006
Step 2: Execute [Sell 1500 @ ₹501] at 09:30:00.007
Step 3: Execute [Buy 800 @ ₹503]   at 09:30:00.008
```

```
TIME: 09:30:00.010 - High traffic period (100,000 orders/second)
┌──────────────────────────────────────────────────────────────────────────┐
│ QUEUE (Growing but maintained in ORDER)                                  │
├──────────────────────────────────────────────────────────────────────────┤
│ FRONT → [Order 1] → [Order 2] → [Order 3] → ... → [Order 50,000] ← BACK  │
└──────────────────────────────────────────────────────────────────────────┘
     ↓ Process continuously from FRONT

✓ Order received at 09:30:00.001 ALWAYS executes before 09:30:00.002
✓ FIFO guarantees legal compliance (order sequence preserved)
✓ No order "jumps the line"
```

Queue – Basic Operations

Operation	Real life Analogy	Description
Enqueue	Person joins the line	Add an element to the rear
Dequeue	Person gets served and leaves	Remove an element from the front
Peek/Front	Check who's next in line	View the front element without removing
isEmpty	Is anyone waiting?	Check if queue is empty
isFull	Is the waiting room at capacity?	Check if queue is full
Size	How many people in line?	Get the number of elements

Linear Queue : Using Array

- Initial State(Empty Queue)

```
maxSize = 5
front = 0
rear = -1

Index:  [0]  [1]  [2]  [3]  [4]
Array:  [ ]  [ ]  [ ]  [ ]  [ ]
         ↑
       front = 0
       rear = -1 (before first position)

isEmpty: rear < front → (-1 < 0) → true
```

Linear Queue : Enqueue

- Operation 1 : enqueue(10)

```
rear = 0 (rear incremented to 0)
front = 0

Index:  [0]  [1]  [2]  [3]  [4]
Array:  [10] [ ]  [ ]  [ ]  [ ]
         ↑
       front
       rear
```

- Operation 2 : enqueue(20)

```
rear = 1
front = 0

Index:  [0]  [1]  [2]  [3]  [4]
Array:  [10] [20] [ ]  [ ]  [ ]
         ↑    ↑
       front rear
```

Linear Queue : Enqueue

- Operation 3 : enqueue(30)

```
rear = 2
front = 0

Index:  [0]  [1]  [2]  [3]  [4]
Array:  [10] [20] [30] [ ]  [ ]
         ↑         ↑
       front      rear
```

- Operation 4 : enqueue(40)

```
rear = 3
front = 0

Index:  [0]  [1]  [2]  [3]  [4]
Array:  [10] [20] [30] [40] [ ]
         ↑              ↑
       front           rear
```

Linear Queue : Enqueue

- Operation 5 : enqueue(50) : Full

```
rear = 4
front = 0

Index:  [0]  [1]  [2]  [3]  [4]
Array:  [10] [20] [30] [40] [50]
         ↑                   ↑
       front                rear

Status: FULL (rear == maxSize - 1)
```

- Operation 6 : enqueue(60) : Overflow

```
Index:  [0]  [1]  [2]  [3]  [4]
Array:  [10] [20] [30] [40] [50]
         ↑                   ↑
       front              rear = 4

❌ Cannot enqueue! Queue is FULL
   isFull() returns true (rear == maxSize - 1)
```

Linear Queue : Dequeue

- Operation 7 : dequeue( ) – Returns 10

```
Before:
Index:  [0]  [1]  [2]  [3]  [4]
Array:  [10] [20] [30] [40] [50]
         ↑                   ↑
       front                rear

After:
Index:  [0]  [1]  [2]  [3]  [4]
Array:  [ ]  [20] [30] [40] [50]
              ↑              ↑
            front           rear

Returned: 10
front = 1
```

- Operation 8 : dequeue( ) – Returns 20

```
Index:  [0]  [1]  [2]  [3]  [4]
Array:  [ ]  [ ]  [30] [40] [50]
                   ↑         ↑
                 front      rear

Returned: 20
front = 2
```

Linear Queue : Dequeue

- Operation 9 : dequeue( ) – Returns 30

```
Index:  [0]  [1]  [2]  [3]  [4]
Array:  [ ]  [ ]  [ ]  [40] [50]
                        ↑    ↑
                     front  rear

Returned: 30
front = 3
```

- Operation 10 : dequeue( ) – Returns 40

```
Index:  [0]  [1]  [2]  [3]  [4]
Array:  [ ]  [ ]  [ ]  [ ]  [50]
                             ↑
                           front
                            rear

Returned: 40
front = 4
```

Linear Queue : Dequeue

- Operation 11 : dequeue( ) – Returns 50 (Queue becomes empty)

```
Before:
Index:  [0]  [1]  [2]  [3]  [4]
Array:  [ ]  [ ]  [ ]  [ ]  [50]
                             ↑
                           front
                            rear

After:
Index:  [0]  [1]  [2]  [3]  [4]
Array:  [ ]  [ ]  [ ]  [ ]  [ ]
                             ↑
                             rear
                           front = 5

Returned: 50
Queue is now EMPTY (front > rear)
front = 5, rear = 4
```

- Operation 12 : dequeue( ) on Empty queue - Underflow

```
Index:  [0]  [1]  [2]  [3]  [4]
Array:  [ ]  [ ]  [ ]  [ ]  [ ]

front = 5, rear = 4 (or can reset to front=0, rear=-1)

❌ Cannot dequeue! Queue is EMPTY
   isEmpty() returns true (front > rear)
   Queue Underflow Error!
```

Linear Queue : peek( )/front( )

```
Index:   [0]   [1]   [2]   [3]   [4]
Array:   [ ]   [ ]   [30]  [40]  [50]
                      ↑           ↑
                    front        rear

peek() returns: 30 (element at array[front])
front and rear remain unchanged
No elements removed
```

Linear Queue : isEmpty( )

```
Empty Queue (rear < front):
Index:  [0]  [1]  [2]  [3]  [4]
Array:  [ ]  [ ]  [ ]  [ ]  [ ]
         ↑
    front = 0
    rear = -1
isEmpty() → true ✓ (-1 < 0)

Another Empty State (after dequeues):
front = 5, rear = 4
isEmpty() → true ✓ (4 < 5)

Non-Empty Queue (rear >= front):
Index:  [0]  [1]  [2]  [3]  [4]
Array:  [ ]  [ ]  [30] [40] [50]
                   ↑         ↑
                 front      rear
isEmpty() → false ✗ (rear >= front)
```

Linear Queue : isFull( )

```
Full Queue (rear == maxSize - 1):
Index:  [0]  [1]  [2]  [3]  [4]
Array:  [10] [20] [30] [40] [50]
         ↑                   ↑
       front              rear = 4
isFull() → true ✓

Not Full (rear < maxSize - 1):
Index:  [0]  [1]  [2]  [3]  [4]
Array:  [10] [20] [30] [ ]  [ ]
         ↑         ↑
       front    rear = 2
isFull() → false ✗
```

Linear Queue : Key Properties Summary

```
Linear Queue Properties

• FIFO: First In First Out
• Initial: front = 0, rear = -1
• Enqueue: array[++rear] = element
• Dequeue: return array[front++]
• Peek: return array[front]
• isEmpty: rear < front
• isFull: rear == maxSize - 1

✓ NO special case for first element
  Just increment rear for every enqueue

⚠LIMITATION: Space wastage after dequeues
   Even if front indices are empty, cannot
   enqueue when rear reaches end
```

Linear Queue : Time Complexity

Operation	Time
enqueue()	O(1)
dequeue()	O(1)
peek()	O(1)
isEmpty()	O(1)
isFull()	O(1)

Problem with Simple Linear Queue

```
Initial Queue (capacity = 5):
[10] [20] [30] [40] [50]
 ↑                   ↑
front               rear

After 2 dequeues:
[  ] [  ] [30] [40] [50]
           ↑         ↑
          front     rear
```

- Problem: We can't add new elements even though the first two positions are empty! 😱

- The array has wasted space at the beginning, but our rear pointer has reached the end. This is inefficient!

Solution : Circular Queue

- A **Circular Queue** treats the array as if it were circular
- Think of it like a roundabout:

```
[0] [1] [2]
[7]     [3]
[6] [5] [4]
```

- When rear reaches index 7, the next position is index 0!

- Key Formulas - The magic lies in these formulas using the ***modulo operator***

rear = (rear + 1) % capacity
front = (front + 1) % capacity

Circular Queue : Using Array

- Initial State (Empty Circular Queue)

```
maxSize = 5
front = 0
rear = -1

Index:  [0]  [1]  [2]  [3]  [4]
Array:  [ ]  [ ]  [ ]  [ ]  [ ]
         ↑
       front = 0
       rear = -1 (before first position)

Circular View:  ⭕  All empty
```

Circular Queue : Using Array

- Operation 1: Enqueue 10

```
rear = (rear + 1) % 5 = (-1 + 1) % 5 = 0
front = 0

Index:  [0]  [1]  [2]  [3]  [4]
Array:  [10] [ ]  [ ]  [ ]  [ ]
         ↑
       front
       rear
```

- Operation 2 : Enqueue 20

```
rear = (0 + 1) % 5 = 1
front = 0

Index:   [0]  [1]  [2]  [3]  [4]
Array:   [10] [20] [ ]  [ ]  [ ]
          ↑    ↑
        front rear
```

Circular Queue : Using Array

- Operation 3: Enqueue 30

```
rear = (1 + 1) % 5 = 2
front = 0

Index:  [0]  [1]  [2]  [3]  [4]
Array:  [10] [20] [30] [ ]  [ ]
         ↑         ↑
       front      rear
```

- Operation 4 : Enqueue 40

```
rear = (2 + 1) % 5 = 3
front = 0

Index:  [0]  [1]  [2]  [3]  [4]
Array:  [10] [20] [30] [40] [ ]
         ↑              ↑
       front           rear
```

Circular Queue : Using Array

- Operation 5: Enqueue 50 (Queue full)

```
rear = (3 + 1) % 5 = 4
front = 0

Index:  [0]  [1]  [2]  [3]  [4]
Array:  [10] [20] [30] [40] [50]
         ↑                   ↑
       front               rear
```

- What is the condition?
  - (rear+1)% maxsize == front ?
- This condition can be true in another situation
  - Can you think of it ?

Circular Queue : isFull( )

- Checking isFull( )

```
Index:  [0]  [1]  [2]  [3]  [4]
Array:  [10] [20] [30] [40] [50]
         ↑                   ↑
      front=0             rear=4
      count = 5

Check: count == maxSize?
       5 == 5? YES!

Status: FULL ✓
Cannot enqueue more elements
*Use a variable count-
    enqueue - increment
    dequeue - decreament
```

Circular Queue : Using Array

- Operation 6: dequeue( ) – Returns 10

```
Array:  [10] [20] [30] [40] [50]
         ↑                   ↑
       front                rear
       count = 5

After:
Index:  [0]  [1]  [2]  [3]  [4]
Array:  [ ]  [20] [30] [40] [50]
              ↑              ↑
           front=1         rear=4
           count = 4

Returned: 10
front = (0 + 1) % 5 = 1
count decremented to 4
```

- Operation 7: dequeue( ) – Returns 20

```
Array:  [ ]  [ ]  [30] [40] [50]
                front=2    rear=4
                count = 3

Returned: 20
front = (1 + 1) % 5 = 2
count decremented to 3
```

Circular Queue : Reusing space

- Unlike linear queue, no space is wasted!

```
Current State:
Index:  [0]  [1]  [2]  [3]  [4]
Array:  [ ]  [ ]  [30] [40] [50]
         ↑         ↑         ↑
      Available front=2   rear=4
                  count = 3

✓ Now we can enqueue at positions [0] and [1]!
  Check isFull: count == maxSize?
              3 == 5? NO → Not full, can enqueue!
```

- Operation 7: enqueue(60) – Wraps around !

```
rear = (4 + 1) % 5 = 0  ← Wraps to beginning!
front = 2
count = 4

Index:   [0]  [1]  [2]  [3]  [4]
Array:   [60] [ ]  [30] [40] [50]
          ↑          ↑
       rear=0    front=2
       count = 4

✓ Successfully used index 0 again!
  This is the CIRCULAR property!
```

Circular Queue : Reusing space

- Operation 9: enqueue(70) – Queue full again !

```
rear = (0 + 1) % 5 = 1
front = 2
count = 5

Index:  [0]  [1]  [2]  [3]  [4]
Array:  [60] [70] [30] [40] [50]
              ↑    ↑
           rear=1 front=2
           count = 5

Queue wraps around:
Front → [2][3][4][0][1] ← Rear
        30  40  50  60  70

Status: FULL (count == maxSize) ✓
```

- Operation 10 : dequeue( ) – Returns 30

```
Index:  [0]  [1]  [2]  [3]  [4]
Array:  [60] [70] [ ]  [40] [50]
              ↑         ↑
           rear=1     front=3
                      count = 4

Returned: 30
front = (2 + 1) % 5 = 3
count decremented to 4
```

Circular Queue : Visualization

```
Step 1: Initial sequential fill
┌────┬────┬────┬────┬────┐
│ 10 │ 20 │ 30 │ 40 │ 50 │
└────┴────┴────┴────┴────┘
  ↑                   ↑
front=0             rear=4

Step 2: After dequeues (front moves)
┌────┬────┬────┬────┬────┐
│    │    │ 30 │ 40 │ 50 │
└────┴────┴────┴────┴────┘
  ↑         ↑         ↑
Available front=2   rear=4
```

```
Step 3: Enqueue wraps to beginning
┌────┬────┬────┬────┬────┐
│ 60 │ 70 │ 30 │ 40 │ 50 │
└────┴────┴────┴────┴────┘
       ↑    ↑
    rear=1 front=2

Circular Connection: [4] → [0]
   rear wraps around using % operator!
```

Circular Queue : isEmpty ( )

```
Empty Queue (count == 0):
Index:  [0]  [1]  [2]  [3]  [4]
Array:  [ ]  [ ]  [ ]  [ ]  [ ]
         ↑
    front = 0
    rear = -1
    count = 0

isEmpty() → true ✓
```

```
Non-Empty Queue (count > 0):
Index:  [0]  [1]  [2]  [3]  [4]
Array:  [60] [70] [ ]  [40] [50]
                        ↑
                     front=3
                     rear=1
                     count = 4

isEmpty() → false ✗
```

Circular Queue : isFull ( )

```
Full Queue (count == maxSize):
Index:   [0]  [1]  [2]  [3]  [4]
Array:   [60] [70] [80] [40] [50]
                    ↑    ↑
                rear=2 front=3
                count = 5

Check: count == maxSize?
       5 == 5? YES → FULL ✓
```

```
Not Full (count < maxSize):
Index:   [0]  [1]  [2]  [3]  [4]
Array:   [60] [ ] [ ]  [40] [50]
          ↑              ↑
       rear=0         front=3
       count = 3

Check: count == maxSize?
       3 == 5? NO → NOT FULL ✗
```

Circular Queue : Key Properties

```
┌─────────────────────────────────────────────────────┐
│  Circular Queue Formulas (Using Count)              │
├─────────────────────────────────────────────────────┤
│  Initial State:                                     │
│    front = 0, rear = -1, count = 0                  │
│                                                     │
│  enqueue(element):                                  │
│    rear = (rear + 1) % maxSize;                     │
│    array[rear] = element;                           │
│    count++;                                         │
│                                                     │
│  dequeue():                                         │
│    element = array[front];                          │
│    front = (front + 1) % maxSize;                   │
│    count--;                                         │
│    return element;                                  │
│                                                     │
│  isEmpty():  count == 0                             │
│  isFull():   count == maxSize                       │
│  peek():     array[front]                           │
│  size():     count                                  │
│                                                     │
│  ✓ Uses ALL maxSize positions (no waste!)           │
│  ✓ No ambiguity between full and empty              │
│  ✓ Simple and consistent logic                      │
└─────────────────────────────────────────────────────┘
```

Advantages of Circular Queue

- **Efficient Space Utilization**: No wasted space in the array
- **Fast Operations**: All operations remain O(1)
- **Memory Friendly**: Reuses freed positions
- **Perfect for Fixed-Size Buffers**: Used in network buffers, IO buffers, etc.

Circular Queue
Real World Applications

- CPU Scheduling (Round Robin)
- Memory Management
- Traffic Systems
- Printer Queue Management
- Call Center Systems
- Buffering (Keyboard, Network)

Queues : Time Complexity

Operation	Simple Queue	Circular Queue
Enqueue	O(1)	O(1)
Dequeue	O(1)	O(1)
Peek	O(1)	O(1)
isEmpty	O(1)	O(1)
isFull	O(1)	O(1)

Queues : Key Takeaways

- Queues follow FIFO principle: First In, First Out
- Basic operations: enqueue (add), dequeue (remove), peek, isEmpty, isFull
- Simple array queues waste space after dequeues
- Circular queues solve this by wrapping around using modulo operator
- The magic formula: (index + 1) % capacity creates the circular behaviour
- All queue operations are O(1) - super fast!

Queues : Good to know

- **Priority Queues**: Where elements have priorities, not just FIFO order
- **Deque (Double-Ended Queue)**: Add/remove from both ends
- **Queue using Linked Lists**: Dynamic size with no fixed capacity
- **Applications in Algorithms**: BFS, CPU scheduling, handling requests

Queue : Final Words

- Queues are everywhere in computer science!
- From managing tasks in your operating system to handling network packets to implementing algorithms, queues are fundamental.
- "In a queue, patience is rewarded. First come, first served!"