Computer Science By SAMPAT LILER

These complete notes have been made for class 12th board computer science exam.

---

3. Stack

Stack Introduction

A **stack** is a linear data structure that follows the **Last In, First Out (LIFO)** principle. It means the element that is added last is removed first.

Key Features of a Stack:

- Operates like a **stack of plates**—you can only remove the top plate first.
- Uses two main operations:
  - **Push** → Add an element to the top.
  - **Pop** → Remove an element from the top.
- Used in applications such as:
  - **Function call management** (Recursive calls).
  - **Expression evaluation** (Postfix, Infix, Prefix).
  - **Undo/Redo operations** in applications.
  - **Backtracking** in algorithms (like the Knight's Tour problem).

Stack Operations

1. Push Operation (Insertion)

- Adds an element to the top of the stack.
- If the stack is full (in a fixed-size stack), it leads to a **Stack Overflow** error.
- **Algorithm:**
  1. Check if the stack is full.
  2. If not, increment the top index.
  3. Insert the new element at the new top position.

2. Pop Operation (Deletion)

- Removes the top element from the stack.
- If the stack is empty, it results in a **Stack Underflow** error.
- **Algorithm:**
  1. Check if the stack is empty.
  2. If not, remove the top element.
  3. Decrement the top index.

3. Peek Operation

- Returns the top element of the stack without removing it.

4. isEmpty Operation

- Checks whether the stack is empty.

5. isFull Operation

- Checks whether the stack is full (in case of a fixed-size array implementation).

Stack Implementation in Python

A stack can be implemented using **lists** or the **collections.deque** module.

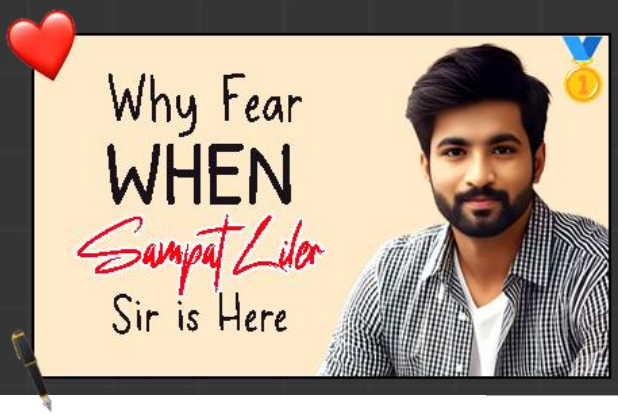
1. Using Lists

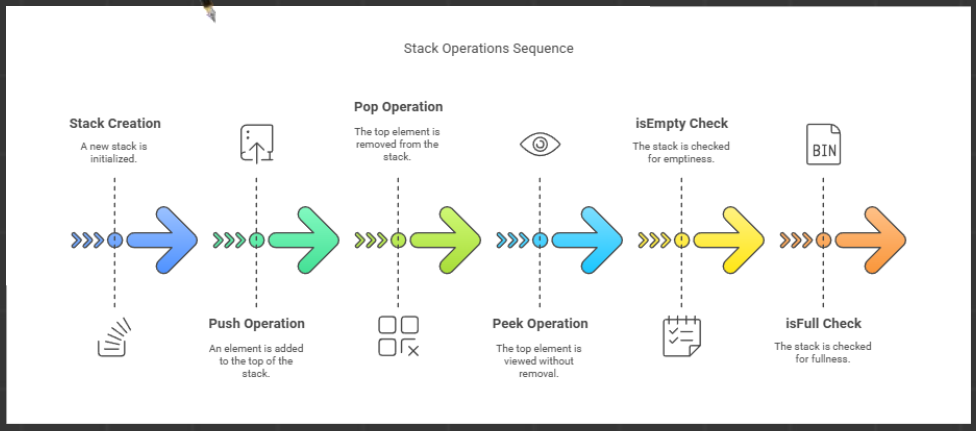
class Stack:

```
    def __init__(self):
        self.stack = []

    def push(self, item):
        self.stack.append(item)


    def pop(self):
        if not self.is_empty():
            return self.stack.pop()
        return "Stack Underflow"

    def peek(self):
        if not self.is_empty():
            return self.stack[-1]
        return None

    def is_empty(self):
        return len(self.stack) == 0

# Example Usage
s = Stack()
s.push(10)
s.push(20)
print(s.pop())  # Output: 20
print(s.peek())  # Output: 10
```

2. Using collections.deque

```
from collections import deque

stack = deque()
stack.append(10)
stack.append(20)
print(stack.pop())  # Output: 20
print(stack[-1])  # Output: 10
```





Notations for Arithmetic Expressions

An arithmetic expression can be written in three notations:

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

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

Conversion from Infix to Postfix Notation

Rules for Conversion

1. **Operands** (A, B, C, etc.) are written **as they appear**.
2. **Operators** are **rearranged** based on precedence.
    - * and / have **higher precedence** than + and -.
3. **Parentheses** dictate priority.

Algorithm

1. **Initialize an empty stack** for operators.
2. **Scan the infix expression** from left to right.
3. **If the character is an operand**, add it to the output.
4. **If the character is an operator**, push it onto the stack.
5. **If the character is '('**, push it onto the stack.
6. **If the character is ')'**, pop from the stack until '(' is found.
7. **Pop remaining operators** in the stack.

Example

Infix: (A + B) * C

Postfix: A B + C *

Python Program

```
def precedence(op):
    if op in ('+', '-'):
        return 1
    if op in ('*', '/'):
        return 2
    return 0

def infix_to_postfix(expression):
    stack = []
    result = ""

    for char in expression:
        if char.isalnum():
            result += char
        elif char == '(':
            stack.append(char)
        elif char == ')':
            while stack and stack[-1] != '(':
                result += stack.pop()
            stack.pop()
        else:
            while stack and precedence(stack[-1]) >= precedence(char):
                result += stack.pop()
            stack.append(char)

    while stack:
        result += stack.pop()
```

```
    return result

expr = "(A+B)*C"
print(infix_to_postfix(expr))  # Output: AB+C*
```

Evaluation of Postfix Expression

A postfix expression is evaluated using a **stack**:

1. **Scan from left to right.**
2. **If the element is an operand, push it to the stack.**
3. **If the element is an operator, pop two elements from the stack, apply the operation, and push the result back.**
4. **Repeat until the expression is fully scanned.**

Example

Postfix Expression: 5 3 + 8 *

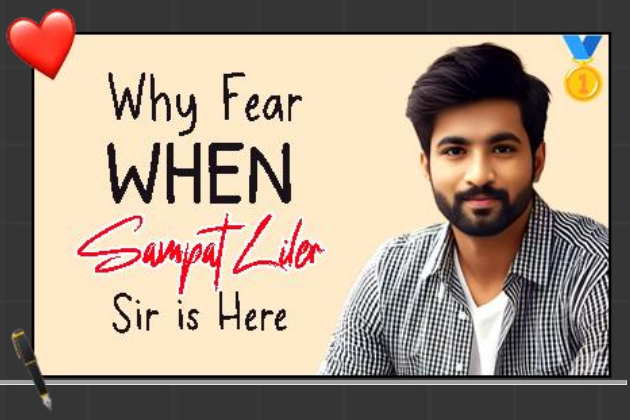
Step-by-step Execution:

Final Answer = **64**

Python Program

```
def evaluate_postfix(expression):
    stack = []
    for char in expression:
        if char.isdigit():
            stack.append(int(char))
        else:
            b = stack.pop()
            a = stack.pop()
            if char == '+':
                stack.append(a + b)
            elif char == '-':
                stack.append(a - b)
            elif char == '*':
                stack.append(a * b)
            elif char == '/':
                stack.append(a // b)  # Integer division
    return stack[0]

expr = "53+8*"
print(evaluate_postfix(expr))  # Output: 64
```

Step	Stack
5	[5]
3	[5, 3]
+	[8] (5+3)
8	[8, 8]
*	[64] (8*8)



Summary

- **Stacks** follow the **LIFO** principle and are used for function calls, undo-redo operations, expression evaluation, and more.
- **Notations (Infix, Prefix, Postfix)** determine the position of operators.
- **Postfix expressions are evaluated efficiently** using a stack.

4. Queue and Deque

Introduction to Queue

A **Queue** is a linear data structure that follows the **First In, First Out (FIFO)** principle. This means the element added first is removed first. It is similar to a line of people waiting for service, where the person who arrives first gets served first.

Real-Life Examples of Queue:

- People standing in a queue at a **bank counter**.
- Vehicles lined up at a **toll plaza**.
- Calls in a **customer service center** waiting to be answered.

Terminology:

- **Front** → The position from where elements are removed.
- **Rear** → The position where new elements are added.
- **Enqueue** → Operation to add an element at the rear.
- **Dequeue** → Operation to remove an element from the front.

Operations on Queue

1. Enqueue (Insertion)

- Adds an element to the **rear** of the queue.
- If the queue is full, an **Overflow** error occurs.

2. Dequeue (Deletion)

- Removes an element from the **front** of the queue.
- If the queue is empty, an **Underflow** error occurs.

3. Peek

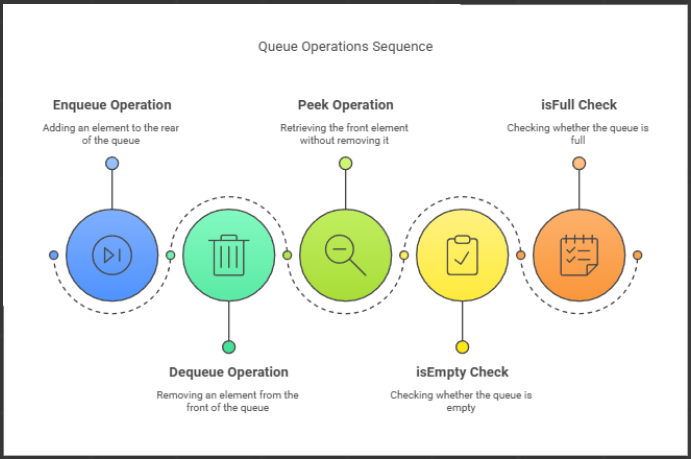
- Retrieves the **front** element **without removing it**.

4. isEmpty

- Checks whether the queue is **empty**.

5. isFull

- Checks whether the queue is **full** (for a fixed-size queue).
-



Implementation of Queue Using Python

Queues can be implemented using:

1. **Lists**
2. **collections.deque** (Recommended for better efficiency)
3. **queue.Queue** module

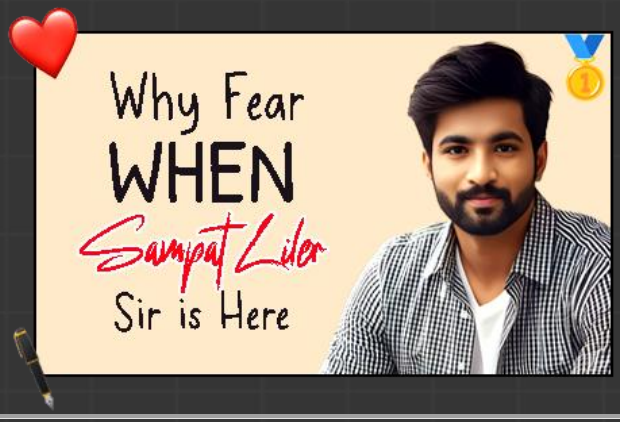
1. Using List

```
class Queue:
    def __init__(self):
        self.queue = []

    def enqueue(self, item):
        self.queue.append(item)

    def dequeue(self):
```

```
        if not self.is_empty():
            return self.queue.pop(0)  # Removes the front element
        return "Queue Underflow"

    def peek(self):
        if not self.is_empty():
            return self.queue[0]  # Returns the front element
        return None

    def is_empty(self):
        return len(self.queue) == 0

# Example Usage
q = Queue()
q.enqueue(10)
q.enqueue(20)
print(q.dequeue())  # Output: 10
print(q.peek())  # Output: 20
```

2. Using collections.deque (Recommended)

```
from collections import deque

queue = deque()
queue.append(10)  # Enqueue
queue.append(20)
print(queue.popleft())  # Dequeue: 10
print(queue[0])  # Peek: 20
```



Introduction to Deque (Double-Ended Queue)

A **Deque** (pronounced as "deck") is a **double-ended queue** where elements can be inserted and removed from **both ends**.

Features of Deque:

- Can work as **both** a stack and a queue.
- Supports insertion and deletion at **both front and rear**.
- Used in **palindrome checking, undo operations, and browser history**.

Types of Deque:

1. **Input-restricted deque** → Insertions allowed at one end, deletions from both ends.
2. **Output-restricted deque** → Deletions allowed at one end, insertions from both ends.

Operations on Deque

1. InsertFront

- Inserts an element at the **front**.

2. InsertRear

- Inserts an element at the **rear**.

3. DeleteFront

- Removes an element from the **front**.

4. DeleteRear

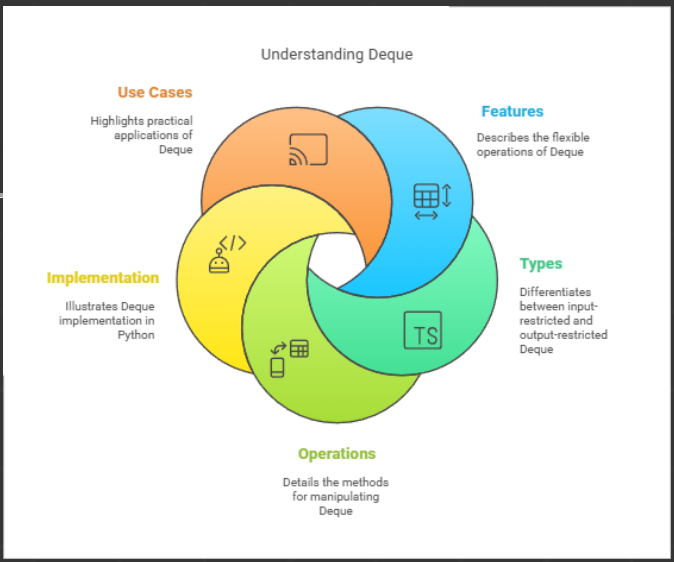
- Removes an element from the **rear**.

5. PeekFront

- Retrieves the **front** element without removing it.

6. PeekRear

- Retrieves the **rear** element without removing it.



Implementation of Deque Using Python

Using collections.deque

```
from collections import deque

class Deque:
    def __init__(self):
        self.deque = deque()

    def insert_front(self, item):
        self.deque.appendleft(item)

    def insert_rear(self, item):
        self.deque.append(item)

    def delete_front(self):
        if not self.is_empty():
            return self.deque.popleft()
        return "Deque Underflow"

    def delete_rear(self):
        if not self.is_empty():
            return self.deque.pop()
        return "Deque Underflow"

    def peek_front(self):
        if not self.is_empty():
            return self.deque[0]
        return None

    def peek_rear(self):
        if not self.is_empty():
            return self.deque[-1]
        return None

    def is_empty(self):
        return len(self.deque) == 0

# Example Usage
d = Deque()
d.insert_front(10)
d.insert_rear(20)
```

```
print(d.delete_front())  # Output: 10
print(d.delete_rear())  # Output: 20
```

Key Differences: Queue vs. Deque

Feature	Queue	Deque
Insertion	Only at rear	Both front and rear
Deletion	Only from front	Both front and rear
Flexibility	Less flexible	More flexible
Use Cases	Print jobs, CPU scheduling	Palindrome check, Undo operations

---

Example: Checking if a String is a Palindrome Using Deque

A **palindrome** is a string that reads the same forward and backward (e.g., "madam").

Algorithm:

1. Insert all characters into a deque.
2. Remove characters from **both ends** and compare them.
3. If they are all equal, the string is a **palindrome**.

Python Code:

```
from collections import deque

def is_palindrome(string):
    d = deque(string)

    while len(d) > 1:
        if d.popleft() != d.pop():
            return False
    return True

# Example Usage
print(is_palindrome("madam"))  # Output: True
print(is_palindrome("hello"))  # Output: False
```

Summary

- **Queues** follow **FIFO** (First In, First Out) principle and are used in **CPU scheduling, print jobs, and call centers**.
- **Deque** is a **double-ended queue** allowing insertion and deletion from **both ends**.
- **Deque is more flexible** and can be used for **stack-like or queue-like operations**.
- **Python provides efficient built-in methods using collections.deque** for better performance.

👉 Subscribe Youtube Channel - Anvira Education - YouTube

👉 Join Course - Https://Anviraeducation.Com/

👉 Follow Us On Facebook - Https://Www.Facebook.Com/Anviraedu

👉 Follow Us On Instagram - https://www.instagram.com/anvira_edu/

👉 Sampat Sir Instagram - https://www.instagram.com/writersampat/

👉 Join Our Telegram Channel - https://t.me/Anviraeducation20