Unit 4 Stack and Queues

A stack is a collection of objects that are inserted and removed according to the last-in, first-out (LIFO) principle. A user may insert objects into a stack at any time, but may only access or remove the most recently inserted object that remains (at the so-called "top" of the stack). The name "stack" is derived from the metaphor of a stack of plates in a spring-loaded, cafeteria plate dispenser.

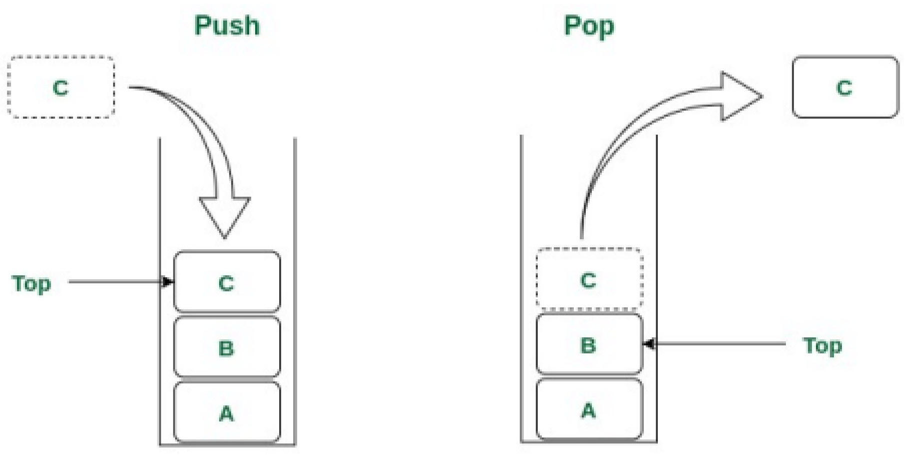
Stack:

A stack is a linear data structure in which all the insertion and deletion of data / values are done at one end only (known as top).It follows LIFO(Last In First Out) property. Insertion / Deletion in stack can only be done from top. Insertion in stack is also known as a PUSH operation. Deletion from stack is also known as POP operation in stack.

It behaves like a stack of plates, where the last plate added is the first one to be removed. Think of it this way:

Pushing an element onto the stack is like adding a new plate on top.

Popping an element removes the top plate from the stack.



The functions associated with stack are:

- isempty() – Returns whether the stack is empty – Time Complexity: O(1)
- size() – Returns the size of the stack – Time Complexity: O(1)
- top() / peek() – Returns a reference to the topmost element of the stack – Time Complexity: O(1)
- push(a) – Inserts the element 'a' at the top of the stack – Time Complexity: O(1)
- pop() – Deletes the topmost element of the stack, an error occurs if the stack is empty. – Time Complexity: O(1)

Applications of Stack:

- Internet Web browsers store the addresses of recently visited sites in a stack. Each time a user visits a new site, that site's address is "pushed" onto the stack of addresses. The browser then allows the user to "pop" back to previously visited sites using the "back" button.
- Text editors usually provide an "undo" mechanism that cancels recent editing operations and reverts to former states of a document. This undo operation can be accomplished by keeping text changes in a stack

• Expression Evaluation: It is used to evaluate prefix, postfix and infix expressions.
• Expression Conversion: It can be used to convert one form of expression(prefix,postfix or infix) to one another.
• Syntax Parsing: Many compilers use a stack for parsing the syntax of expressions.
• Backtracking: It can be used for back traversal of steps in a problem solution.
• Parenthesis Checking: Stack is used to check the proper opening and closing of parenthesis.
• String Reversal: It can be used to reverse a string.
• Function Call: Stack is used to keep information about the active functions or subroutines.

Python Implementation :
```
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)
        print(f"{item} pushed onto the stack.")

    def pop(self):
        if not self.is_empty():
            popped_item = self.items.pop()
            print(f"Popped item: {popped_item}")
            return popped_item
        else:
            print("Stack is empty. Cannot pop.")
            return None

    def peek(self):
        if not self.is_empty():
            print(f"Top element: {self.items[-1]}")
            return self.items[-1]
        else:
            print("Stack is empty. Nothing to peek.")
            return None

    def display(self):
        if not self.is_empty():
            print("Stack elements:", self.items)
        else:
            print("Stack is empty.")

    def get_size(self):
        print(f"Current stack size: {len(self.items)}")
        return len(self.items)

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


def menu():
    stack = Stack()
    while True:
        print("\nMenu Options:")
        print("1. Push: Push a new element onto the stack.")
        print("2. Pop: Remove the top element from the stack.")
        print("3. Peek: View the top element without removing it.")
        print("4. Display: Display all stack elements.")
        print("5. Get Size: Display the current size of the stack.")
        print("6. Exit: Exit the program.")

        try:
            choice = int(input("Enter your choice: "))
            if choice == 1:
                element = input("Enter the element to push: ")
                stack.push(element)
            elif choice == 2:
                stack.pop()
            elif choice == 3:
                stack.peek()
            elif choice == 4:
                stack.display()
            elif choice == 5:
                stack.get_size()
            elif choice == 6:
                print("Exiting the program. Goodbye!")
                break
            else:
                print("Invalid choice. Please select a valid option.")
        except ValueError:
            print("Invalid input. Please enter a number.")


# Run the program
if __name__ == "__main__":
    menu()
```

Stack Using Linked List :

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None



class Stack:
    def __init__(self):
        self.top = None
        self.size = 0  # Initialize size of the stack

    def is_empty(self):
        return self.size == 0

    def push(self, data):
        new_node = Node(data)
        new_node.next = self.top
        self.top = new_node
        self.size += 1  # Increment size
        print(f"Pushed {data} onto the stack.")

    def pop(self):
        if self.is_empty():
            print("Stack Underflow! Cannot pop from an empty stack.")
            return None
        popped_data = self.top.data
        self.top = self.top.next
        self.size -= 1  # Decrement size
        print(f"Popped {popped_data} from the stack.")
        return popped_data

    def peek(self):
        if self.is_empty():
            print("Stack is empty. Nothing to peek.")
            return None
        return self.top.data

    def display(self):
        if self.is_empty():
            print("Stack is empty.")
            return
        current = self.top
        print("Stack elements:")
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")

    def get_size(self):
        return self.size  # Return the size of the stack


def menu():
    stack = Stack()
    while True:
        print("\nMenu:")
        print("1. Push")
        print("2. Pop")
        print("3. Peek")
        print("4. Display")
        print("5. Get Size")
        print("6. Exit")
        choice = int(input("Enter your choice: "))

        if choice == 1:
            data = int(input("Enter the data to push: "))
            stack.push(data)
        elif choice == 2:
            stack.pop()
        elif choice == 3:
            top_element = stack.peek()
            if top_element is not None:
                print(f"Top element is: {top_element}")
        elif choice == 4:
            stack.display()
        elif choice == 5:
            print(f"Size of stack: {stack.get_size()}")
        elif choice == 6:
            print("Exiting program. Goodbye!")
            break
        else:
            print("Invalid choice. Please try again.")


# Run the menu-driven program
if __name__ == "__main__":
    menu()
```

Application of Stack :

1) Matching parenthesis {[()]} , Check whether parenthesis are matching

```
class Stack:
    def __init__(self):
        self.items = []

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

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

    def pop(self):
        if self.is_empty():
            return None
        return self.items.pop()

    def peek(self):
        if self.is_empty():
            return None
        return self.items[-1]


def is_matching_parentheses(expression):
    stack = Stack()
    # Define matching pairs
    matching_pairs = {')': '(', '}': '{', ']': '['}

    for char in expression:
        if char in "({[":
            stack.push(char)  # Push opening brackets onto the stack
        elif char in ")}]":
            # If stack is empty or top of stack does not match, return False
            if stack.is_empty() or stack.pop() != matching_pairs[char]:
                return False

    # If stack is not empty, some opening brackets are unmatched
    return stack.is_empty()


# Example usage
expression = input("Enter an expression to check for matching parentheses: ")
if is_matching_parentheses(expression):
    print("The parentheses in the expression are balanced.")
else:
    print("The parentheses in the expression are NOT balanced.")
```

2) Infix to postfix expression;

Steps to Convert Infix to Postfix

1. Scan the infix expression **from left to right**.
2. If the scanned character is an operand, put it in the postfix expression.
3. Otherwise, do the following
   - If the precedence of the current scanned operator is higher than the precedence of the operator on top of the stack, or if the stack is empty, or if the stack contains a '(', then push the current operator onto the stack.
   - Else, pop all operators from the stack that have precedence higher than or equal to that of the current operator. After that push the current operator onto the stack.
4. If the scanned character is a '**(**', push it to the stack.
5. If the scanned character is a '**)**', pop the stack and output it until a '**(**' is encountered, and discard both the parenthesis.
6. Repeat steps **2-5** until the infix expression is scanned.
7. Once the scanning is over, Pop the stack and add the operators in the postfix expression until it is not empty.
8. Finally, print the postfix expression.

Operator Precedence and Associativity

Operator	Precedence	Associativity
^	High (3)	Right
* /	Medium (2)	Left
+ -	Low (1)	Left

Example: Convert `A + B * (C ^ D - E) ^ (F + G * H) - I` to Postfix

Step-by-Step Process:

Step	Action/Character	Stack	Postfix Expression
1	A	-	A
2	+	+	A
3	B	+	A B
4	*	+ *	A B
5	(	+ * (	A B
6	C	+ * (	A B C
7	^	+ * (^	A B C
8	D	+ * (^	A B C D
9	-	+ * (-	A B C D ^

Step	**Action/Character**	**Stack**	**Postfix Expression**
10	E	+ * (-	A B C D ^ E
11	)	+ *	A B C D ^ E -
12	^	+ ^	A B C D ^ E -
13	(	+ ^ (	A B C D ^ E -
14	F	+ ^ (	A B C D ^ E - F
15	+	+ ^ (+	A B C D ^ E - F
16	G	+ ^ (+	A B C D ^ E - F G
17	*	+ ^ (+ *	A B C D ^ E - F G
18	H	+ ^ (+ *	A B C D ^ E - F G H
19	)	+ ^	A B C D ^ E - F G H * +
20	-	-	A B C D ^ E - F G H * + ^
21	I	-	A B C D ^ E - F G H * + ^ I
22	End of Expression	-	A B C D ^ E - F G H * + ^ I -

```
class Stack:
    def __init__(self):
        self.items = []

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

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

    def pop(self):
        if self.is_empty():
            return None
        return self.items.pop()

    def peek(self):
        if self.is_empty():
            return None
        return self.items[-1]


def precedence(op):
    """Return the precedence of an operator."""
    if op == '^':
        return 3
    if op in '*/':
        return 2
    if op in '+-':
        return 1
    return 0


def is_operator(c):
    """Check if the character is an operator."""
    return c in '+-*/^'


def infix_to_postfix(expression):
    stack = Stack()
    postfix = []

    for char in expression:
        if char.isalnum():  # Operand (letters or numbers)
            postfix.append(char)
        elif char == '(':  # Opening parenthesis
            stack.push(char)
        elif char == ')':  # Closing parenthesis
            # Pop until matching '(' is found
            while not stack.is_empty() and stack.peek() != '(':
                postfix.append(stack.pop())
            stack.pop()  # Remove the '(' from the stack
        elif is_operator(char):
            # Pop operators of higher or equal precedence
            while (not stack.is_empty() and
                   precedence(char) <= precedence(stack.peek()) and
                   char != '^'):  # '^' is right-associative
                postfix.append(stack.pop())
            stack.push(char)

    # Pop remaining operators in the stack
    while not stack.is_empty():
        postfix.append(stack.pop())

    return ''.join(postfix)
```

```
# Example usage
expression = input("Enter an infix expression: ")
postfix_expression = infix_to_postfix(expression)
print(f"Postfix expression: {postfix_expression}")
```

3) Evaluate postfix expression:

Algorithm

1. **Initialize**:
   - o Create an empty stack to hold operands.
2. **Scan the Expression**:
   - o Traverse the postfix expression from left to right.
3. **Process Each Character**:
   - o **Operand**:
     - ▪ Push operands (numbers) onto the stack.
   - o **Operator**:
     - ▪ Pop the top two elements from the stack.
     - ▪ Perform the operation: **second popped operand** `operator` **first popped operand**.
     - ▪ Push the result back onto the stack.
4. **Final Result**:
   - o After scanning the entire expression, the result of the evaluation will be the only element left in the stack.

Example

Evaluate Postfix Expression:

```
text
Copy code
23*54*+9-
```

Step-by-Step Process

Step	Character	Stack	Action/Result
1	`2`	`[2]`	Push `2`.
2	`3`	`[2, 3]`	Push `3`.
3	`*`	`[6]`	Pop `3` and `2`, compute `2*3=6`. Push `6`.
4	`5`	`[6, 5]`	Push `5`.
5	`4`	`[6, 5, 4]`	Push `4`.
6	`*`	`[6, 20]`	Pop `4` and `5`, compute `5*4=20`. Push `20`.
7	`+`	`[26]`	Pop `20` and `6`, compute `6+20=26`. Push `26`.

Step	Character	Stack	Action/Result
8	`9`	`[26, 9]`	Push `9`.
9	`-`	`[17]`	Pop `9` and `26`, compute `26-9=17`. Push `17`.

Final Result

The final result of the postfix expression `23*54*+9-` is:

```
class Stack:
    def __init__(self):
        self.items = []

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

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

    def pop(self):
        if self.is_empty():
            return None
        return self.items.pop()


def evaluate_postfix(expression):
    stack = Stack()

    for char in expression:
        if char.isdigit():  # If operand, push to stack
            stack.push(int(char))
        elif char in "+-*/^":  # If operator, pop two operands and apply operation
            b = stack.pop()
            a = stack.pop()
            if char == '+':
                result = a + b
            elif char == '-':
                result = a - b
            elif char == '*':
                result = a * b
            elif char == '/':
                result = a / b  # Perform floating-point division
            elif char == '^':
                result = a ** b  # Exponentiation
            stack.push(result)  # Push result back to the stack
        else:
            raise ValueError(f"Invalid character encountered: {char}")

    # The final result is the last item in the stack
    return stack.pop()


# Example usage
postfix_expression = input("Enter a postfix expression: ")
try:
    result = evaluate_postfix(postfix_expression)
    print(f"The result of the postfix expression is: {result}")
except Exception as e:
    print(f"Error: {e}")
```

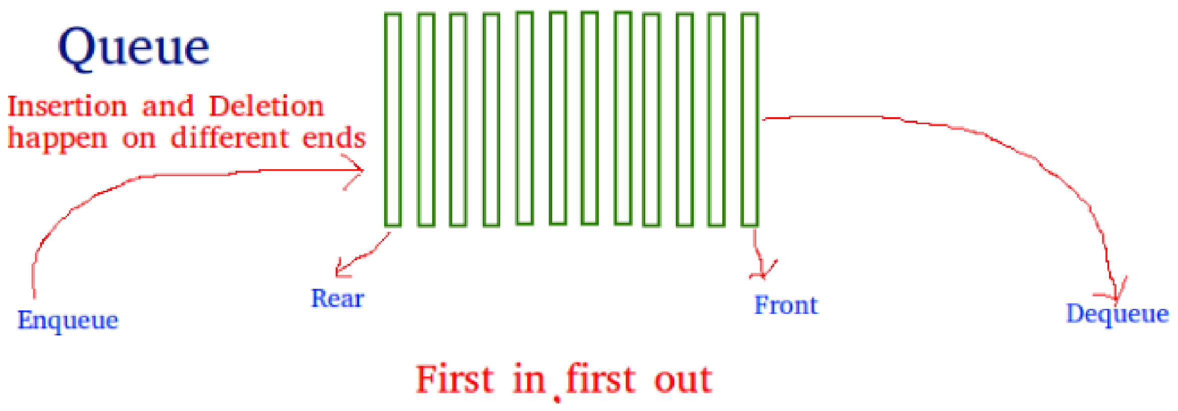
To reverse each word in a sentence using a stack in Python

```
def reverse_words_with_stack(sentence):
    words = sentence.split()  # Split the sentence into words
    reversed_sentence = []    # List to hold reversed words

    for word in words:
        stack = []            # Stack to reverse each word
        for char in word:     # Push each character of the word onto the stack
            stack.append(char)

        reversed_word = ""    # Pop characters to form the reversed word
        while stack:
            reversed_word += stack.pop()

        reversed_sentence.append(reversed_word)  # Add the reversed word to
the list

    return " ".join(reversed_sentence)  # Combine the reversed words into a
sentence


# Test the function
sentence = "Hello World from Python"
result = reverse_words_with_stack(sentence)
print("Original Sentence:", sentence)
print("Reversed Sentence:", result)
```

Queue Data Structure

A **queue** is an important data structure in **programming**. It follows the FIFO (First In, First Out) method and is open at both ends. Data insertion is done at one end, the rear end or the tail of the queue, while deletion is done at the other end, the front end or the head of the queue.

FIFO Principle in Queue:
FIFO Principle states that the first element added to the Queue will be the first one to be removed or processed. So, Queue is like a line of people waiting to purchase tickets, where the first person in line is the first person served. (i.e. First Come First Serve).



Basic Terminologies of Queue

- **Front:** Position of the entry in a queue ready to be served, that is, the first entry that will be removed from the queue, is called the **front** of the queue. It is also referred as the **head** of the queue.
- **Rear:** Position of the last entry in the queue, that is, the one most recently added, is called the **rear** of the queue. It is also referred as the **tail** of the queue.
- **Size:** Size refers to the **current** number of elements in the queue.
- **Capacity:** Capacity refers to the **maximum** number of elements the queue can hold.

Applications of Simple Queue:

Resource Allocation: Simple Queues are is useful for resource allocation in operating systems that manage resource requests such as CPU, memory, and I/O devices.
Batch Processing: Queues accommodate batch jobs, for instance, tasks like data processing or rendering images, which are queued up for sequential execution.
Message Buffering: It helps to buffer the message in communication systems to ensure a smooth data flow among processes.

Applications of Circular Queue

- **CPU Scheduling**: Used in operating systems to manage processes.
- **Data Buffering**: Frequently used in data streaming and buffering applications.
- **Simulation Systems**: Helpful in simulating real-world systems and processes, e.g., traffic flow control.

Applications of Priority Queue;

- **Data Compression**: In methods like Huffman coding, priority queues organize characters based on how often they appear, which helps reduce file sizes.
- **Event Simulation**: Priority queues manage events in simulations, making sure that events happening sooner are processed before later ones.
- **Top-k Elements**: They can keep track of the top-k items coming from a data stream, allowing rapid access to the most important ones.

Deque Applications:

- **Undo/Redo Functions**: Deques are useful in applications for keeping track of actions so users can go back or forward through their history.
- **Store Web Browser History**: In a web browser, deques can be used to easily store the visited pages to add new pages and remove old ones.
- **Graph Traversal**: In algorithms like Breadth-First Search (BFS), deques help efficiently manage which nodes to explore next.
- **Palindrome Checking**: Deques can check if a word or phrase reads the same backwards as forward by comparing letters from both ends.

Basic Queue Operations in Queue Data Structure

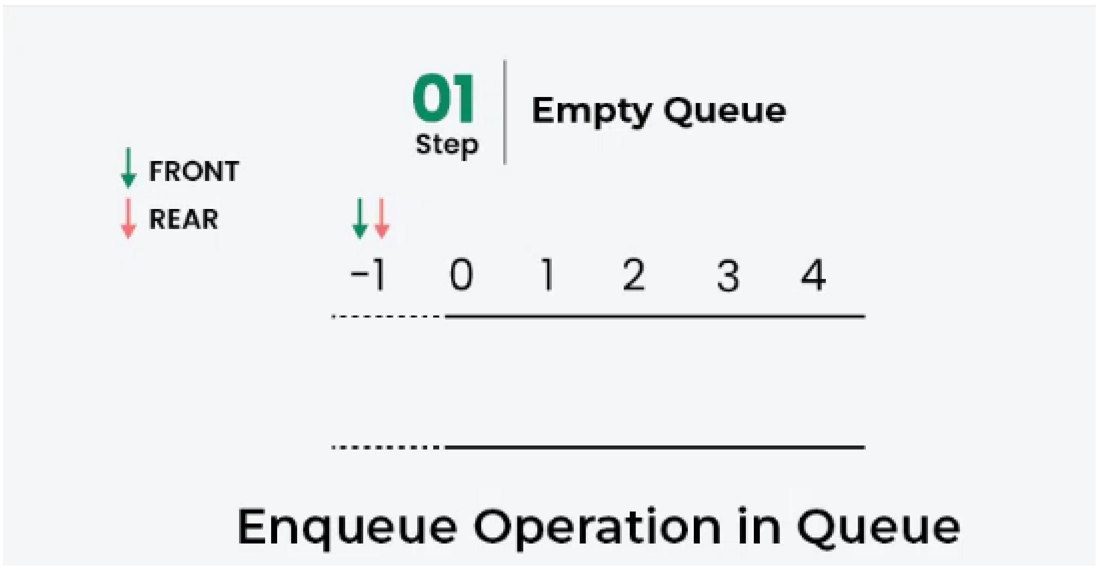
Below are the basic queue operations in data structure:

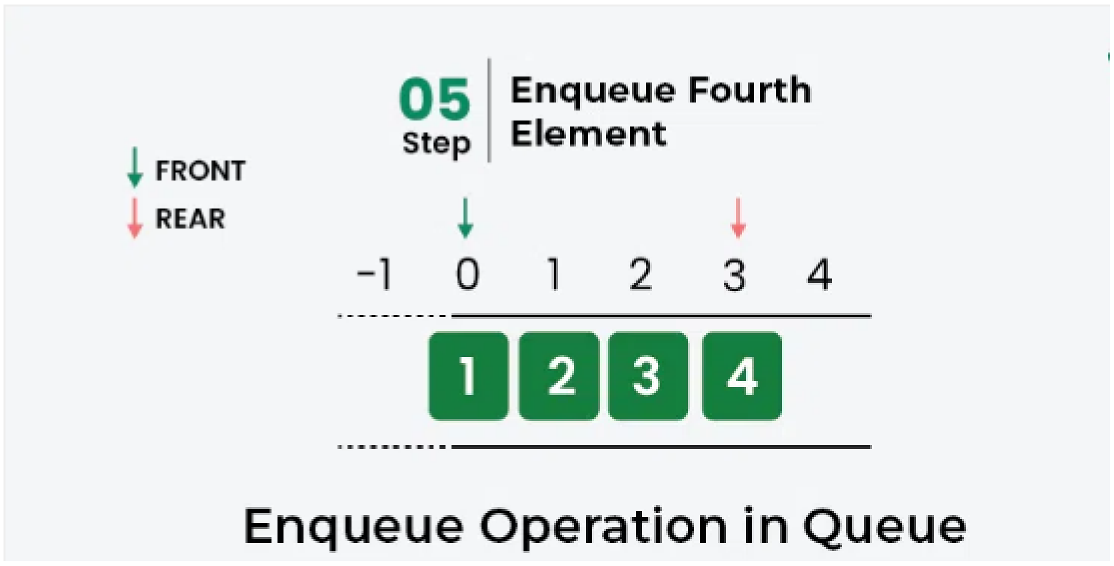
Operation	Description
enqueue()	Process of adding or storing an element to the end of the queue
dequeue()	Process of removing or accessing an element from the front of the queue
peek()	Used to get the element at the front of the queue without removing it
initialize()	Creates an empty queue
isfull()	Checks if the queue is full
isempty()	Check if the queue is empty

Operations on Queue

1. Enqueue:

Enqueue operation **adds (or stores) an element to the end of the queue.**
Steps:
1. Check if the **queue is full.** If so, return an **overflow** error and exit.
2. If the queue is **not full**, increment the **rear** pointer to the next available position.
3. Insert the element at the rear.

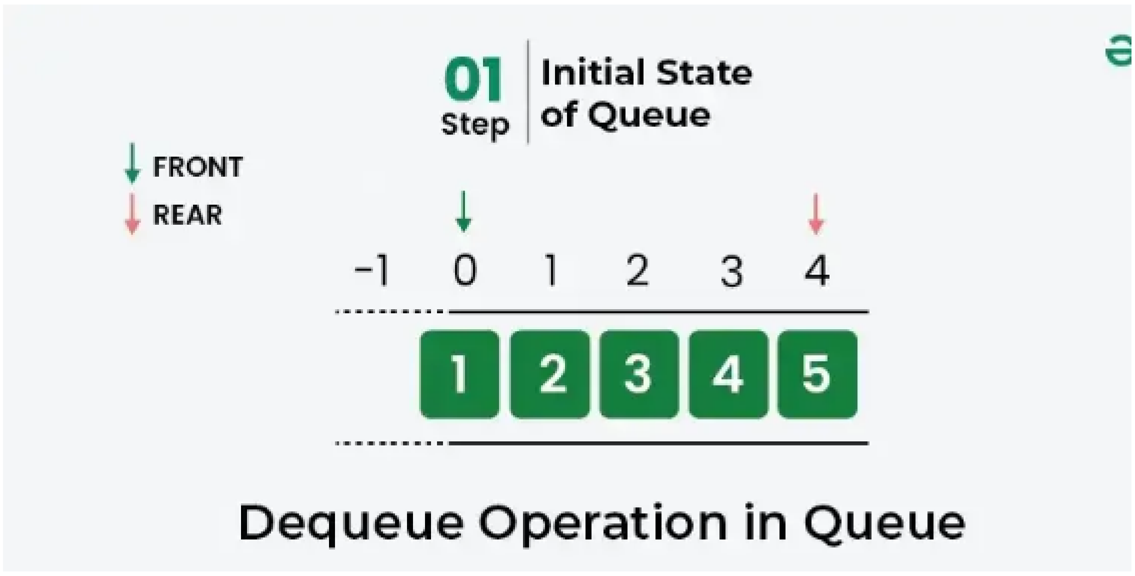


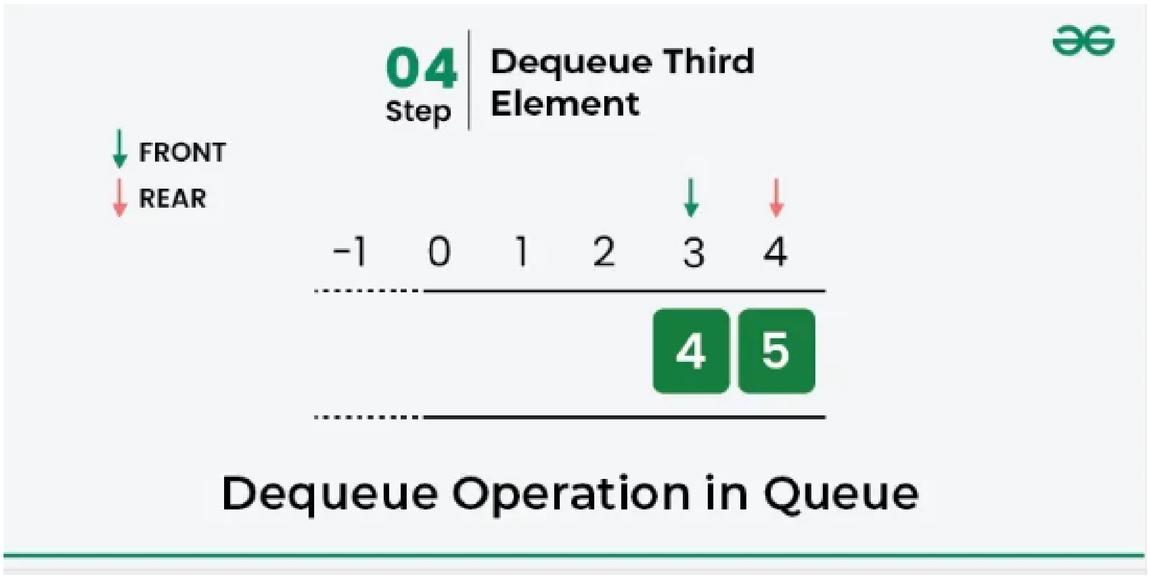


2. Dequeue:

Dequeue operation removes the element at the front of the queue. The following steps are taken to perform the dequeue operation:

1. Check if the **queue is empty.** If so, return an **underflow** error.
2. Remove the element at the **front.**
3. **Increment** the **front** pointer to the next element.





3. Peek or Front Operation:

This operation returns the element at the front end without removing it.

4. Size Operation:

This operation returns the numbers of elements present in the queue.

5. isEmpty Operation:

This operation returns a boolean value that indicates whether the queue is empty or not.

6. isFull Operation:

This operation returns a boolean value that indicates whether the queue is full or not.

Python Implementation of Queue

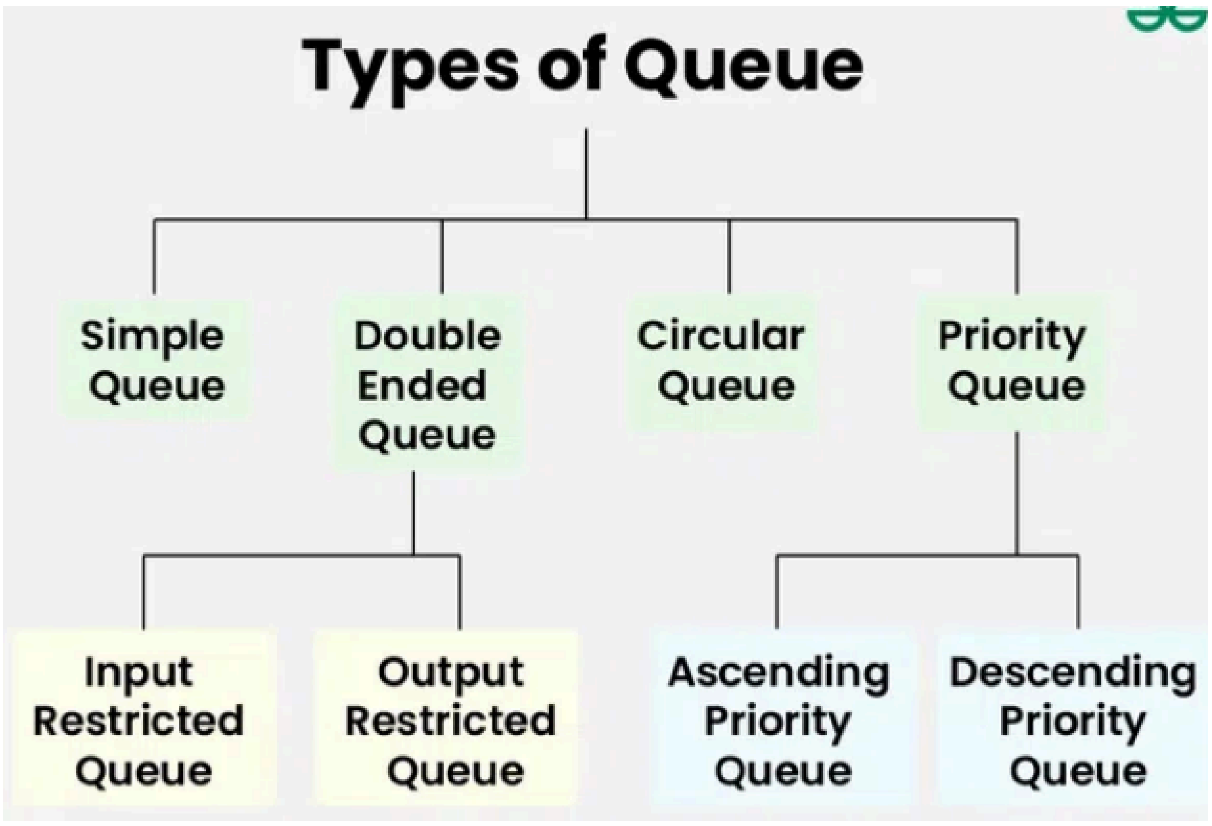
```
class SimpleQueue:
    def __init__(self):
        self.queue = []

    def enqueue(self, item):
        self.queue.append(item)
        print(f"{item} added to the queue.")

    def dequeue(self):
        if self.is_empty():
            print("Queue is empty. Cannot dequeue.")
        else:
            item = self.queue.pop(0)
            print(f"{item} removed from the queue.")

    def size(self):
        return len(self.queue)

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

    def print_queue(self):
        if self.is_empty():
            print("Queue is empty.")
        else:
            print("Queue contents:", self.queue)

    def peek(self):
        if self.is_empty():
            print("Queue is empty. No front element.")
        else:
            print("Front element:", self.queue[0])

# Menu-driven program
def main():
    queue = SimpleQueue()

    while True:
        print("\nQueue Operations:")
        print("1. Enqueue")
        print("2. Dequeue")
        print("3. Size of Queue")
        print("4. Check if Queue is Empty")
        print("5. Print Queue")
        print("6. Peek at Front Element")
        print("7. Exit")

        choice = input("Enter your choice (1-7): ")

        if choice == '1':
            item = input("Enter the item to enqueue: ")
            queue.enqueue(item)
        elif choice == '2':
            queue.dequeue()
        elif choice == '3':
            print("Size of queue:", queue.size())
        elif choice == '4':
            if queue.is_empty():
                print("Queue is empty.")
            else:
                print("Queue is not empty.")
        elif choice == '5':
            queue.print_queue()
        elif choice == '6':
            queue.peek()
        elif choice == '7':
            print("Exiting the program. Goodbye!")
            break
        else:
            print("Invalid choice. Please try again.")

if __name__ == "__main__":
    main()
```

Queue using Linked list

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedListQueue:
    def __init__(self):
        self.front = None
        self.rear = None

    def enqueue(self, item):
```

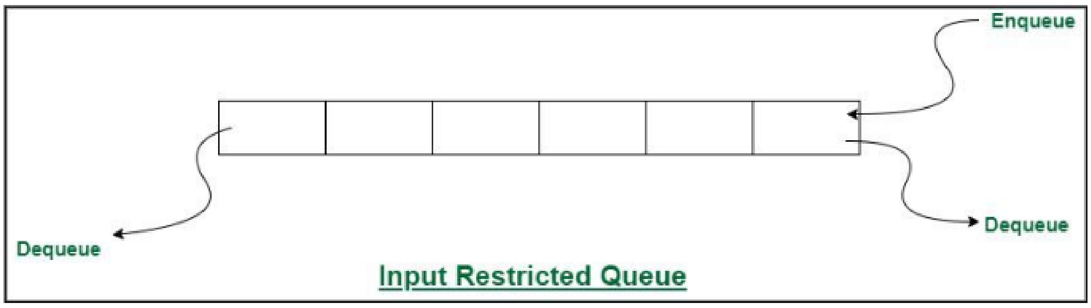
```
    new_node = Node(item)
    if self.rear is None:
        self.front = self.rear = new_node
    else:
        self.rear.next = new_node
        self.rear = new_node
    print(f"{item} added to the queue.")

def dequeue(self):
    if self.is_empty():
        print("Queue is empty. Cannot dequeue.")
    else:
        item = self.front.data
        self.front = self.front.next
        if self.front is None:
            self.rear = None
        print(f"{item} removed from the queue.")

def size(self):
    count = 0
    current = self.front
    while current:
        count += 1
        current = current.next
    return count

def is_empty(self):
    return self.front is None

def print_queue(self):
    if self.is_empty():
            print("Queue is empty.")
        else:
            current = self.front
            print("Queue contents:", end=" ")
            while current:
                print(current.data, end=" ")
                current = current.next
            print()

    def peek(self):
        if self.is_empty():
            print("Queue is empty. No front element.")
        else:
            print("Front element:", self.front.data)

# Menu-driven program
def main():
    queue = LinkedListQueue()

    while True:
        print("\nQueue Operations:")
        print("1. Enqueue")
        print("2. Dequeue")
        print("3. Size of Queue")
        print("4. Check if Queue is Empty")
        print("5. Print Queue")
        print("6. Peek at Front Element")
        print("7. Exit")

        choice = input("Enter your choice (1-7): ")
        if choice == '1':
            item = input("Enter the item to enqueue: ")
            queue.enqueue(item)
        elif choice == '2':
            queue.dequeue()
        elif choice == '3':
            print("Size of queue:", queue.size())
        elif choice == '4':
            if queue.is_empty():
                print("Queue is empty.")
            else:
                print("Queue is not empty.")
        elif choice == '5':
            queue.print_queue()
        elif choice == '6':
            queue.peek()
        elif choice == '7':
            print("Exiting the program. Goodbye!")
            break
        else:
            print("Invalid choice. Please try again.")

if __name__ == "__main__":
    main()
```

Types of Queues



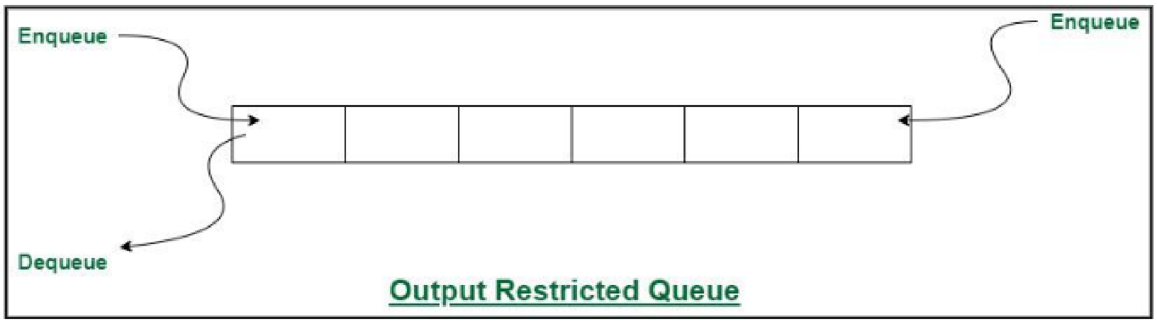
Queue data structure can be classified into 4 types:

1. **Simple Queue:** Simple Queue simply follows **FIFO** Structure. We can only insert the element at the back and remove the element from the front of the queue.
2. **Double-Ended Queue (Deque)**: In a double-ended queue the insertion and deletion operations, both can be performed from both ends. They are of two types:
   - **Input Restricted Queue:** This is a simple queue. In this type of queue, the input can be taken from only one end but deletion can be done from any of the ends.



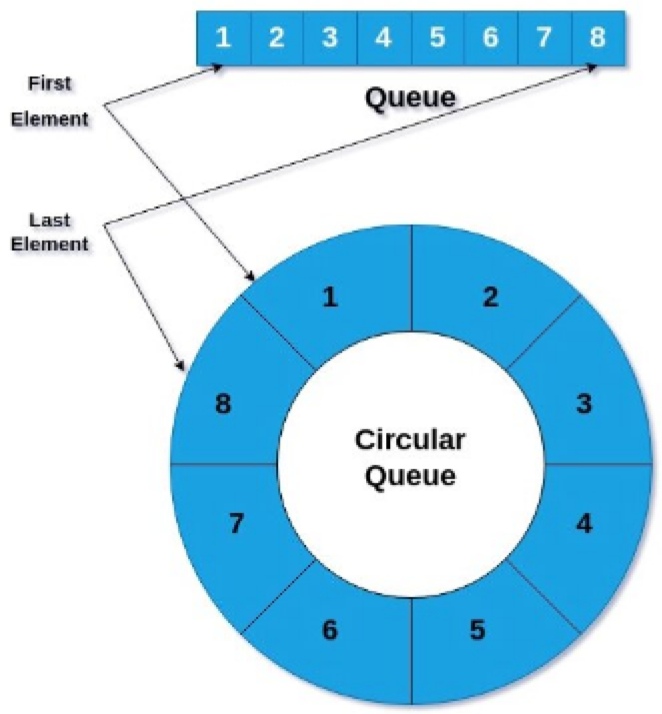
Input Restricted Queue

- **Output Restricted Queue:** This is also a simple queue. In this type of queue, the input can be taken from both ends but deletion can be done from only one end.



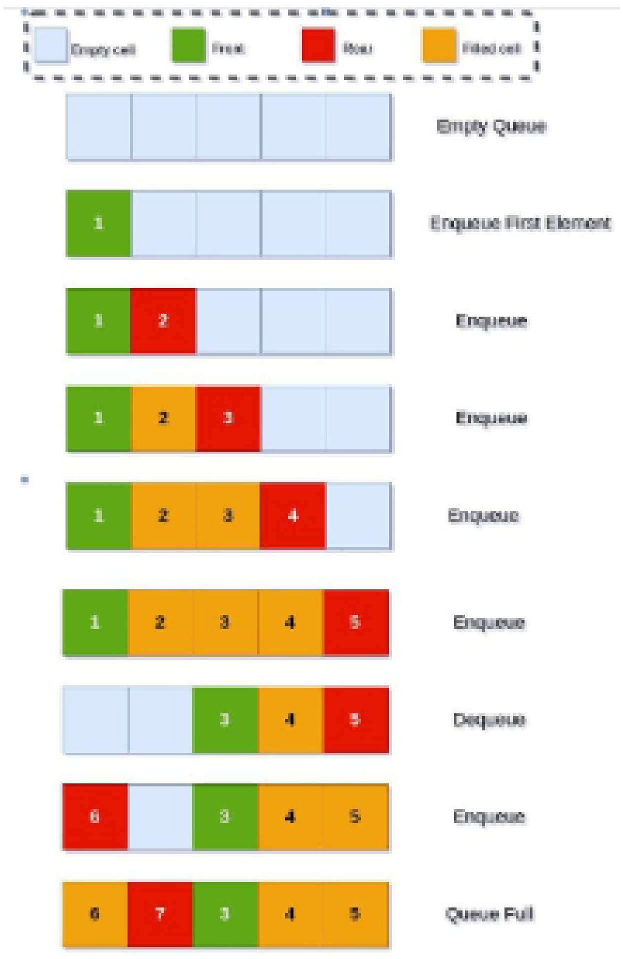
Output Restricted Queue

3. **Circular Queue:** This is a special type of queue where the last position is connected back to the first position. Here also the operations are performed in FIFO order.



Circular Queue Operations

Circular queues have several key operations that allow for the efficient management of elements:



1. Enqueue (Adding an element)

Add an element to the rear of the circular queue.

Steps:

- Check if the queue is full. If (rear + 1) % size == front, the queue is full.
- If the queue is not full, update the rear pointer to (rear + 1) % size.
- Insert the new element at the rear position.
- If the queue was initially empty (i.e., front was -1), set front to 0.

Example in Python:

```
def enqueue(self, item):
    if self.is_full():
        print("Queue is full!")
        return
    if self.front == -1:
        self.front = 0
    self.rear = (self.rear + 1) % self.capacity
    self.queue[self.rear] = item
```

2. Dequeue (Removing an element)

Remove an element from the front of the circular queue.

Steps:

- Check if the queue is empty. If front == -1, the queue is empty.
- If the queue is not empty, remove the element at the front position.
- Update the front pointer to (front + 1) % size.
- If the queue becomes empty after the dequeue operation (i.e., front equals rear), reset both front and rear to -1.

Example in Python:

```
def dequeue(self):
    if self.is_empty():
        print("Queue is empty!")
        return None
    item = self.queue[self.front]
    if self.front == self.rear:
        self.front = -1
        self.rear = -1
    else:
        self.front = (self.front + 1) % self.capacity
    return item
```

3. Peek/Front (Viewing the front element)

View the front element without removing it from the circular queue.

Steps:

- Check if the queue is empty. If front == -1, the queue is empty.
- If the queue is not empty, return the element at the front position.

Example in Python:

```
def peek(self):
    if self.is_empty():
        print("Queue is empty!")
        return None
    return self.queue[self.front]
```

4. isEmpty (Checking if the queue is empty)

Check if the circular queue is empty.

Steps:

The queue is empty if front == -1.

Example in Python:

```
def is_empty(self):
    return self.front == -1
```

5. isFull (Checking if the queue is full)

Check if the circular queue is full.

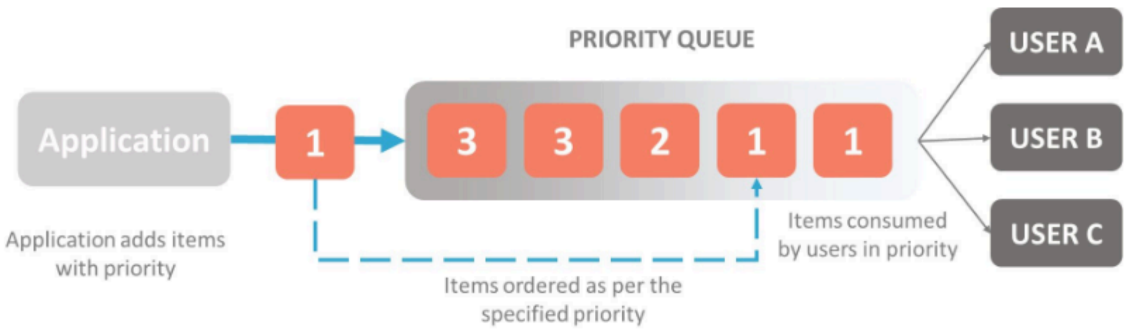
Steps:

The queue is full if (rear + 1) % size == front.

Example in Python:

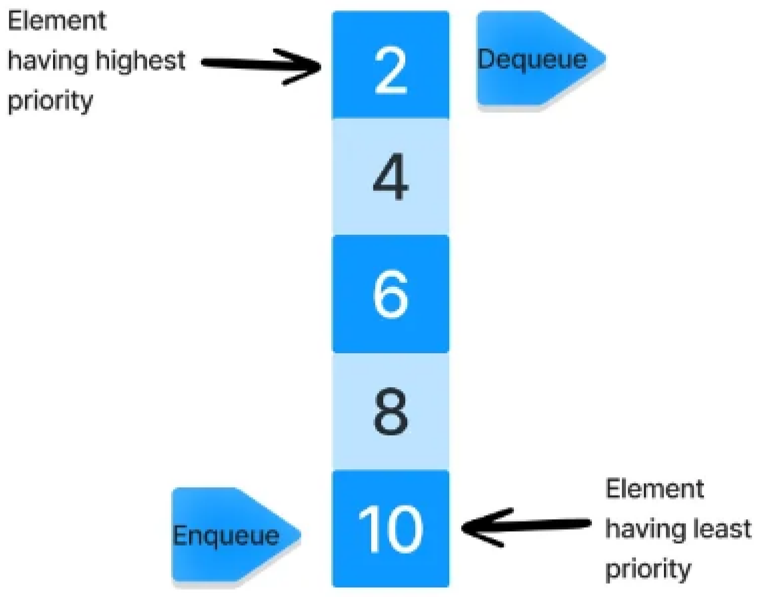
```
def is_full(self):
    return (self.rear + 1) % self.capacity == self.fron
```

4. **Priority Queue**: A priority queue is a special queue where the elements are accessed based on the priority assigned to them.

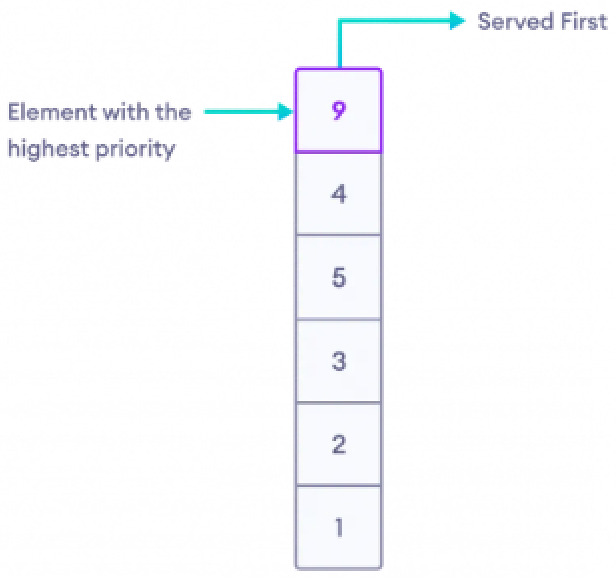

PRIORITY QUEUE
Application
1
3
3
2
1
1
USER A
USER B
USER C
Application adds items with priority
Items consumed by users in priority
Items ordered as per the specified priority

5. They are of two types:

- **Ascending Priority Queue:** In Ascending Priority Queue, the elements are arranged in increasing order of their priority values. Element with smallest priority value is popped first.



- **Descending Priority Queue:** In Descending Priority Queue, the elements are arranged in decreasing order of their priority values. Element with largest priority is popped first.


Served First
Element with the
highest priority
9
4
5
3
2
1