NAME: THEURI BONFACE KARUE

REG_NO: SCT211-0573/2022

UNIT: ADTs & ALGORITHMS

UNIT_CODE:ICS 2300

ASSIGNMENT : TAKEAWAY CAT

1. Structure of a Node in a Singly Linked List [4 Marks]

A node in a singly linked list typically consists of two parts:

- **Data (Value):** This stores the information or data that the node holds.
- **Next (Pointer):** This stores the address of the next node in the list.

Example code:

```
struct Node {
   int data;      // Data part
   Node* next;    // Pointer to the next node
};
```

2. Create a Queue Using Two Stacks [6 Marks]

To create a queue using two stacks, we can use two stacks, say `stack1` and `stack2`. The basic idea is to use one stack for enqueueing (inserting elements) and another for dequeueing (removing elements).

Operations:

- **Enqueue**: Push the element onto `stack1`.
- **Dequeue**: If `stack2` is empty, pop all elements from `stack1` and push them into `stack2`. Then, pop from `stack2`.

Illustration:

Enqueue Operation (Adding to the Queue)

1. Push the element onto `stack1`.
2. `stack1` stores all the elements in the order they are added.

For example:

- Enqueue `1`: `stack1 = [1]`, `stack2 = []`
- Enqueue `2`: `stack1 = [1, 2]`, `stack2 = []`
- Enqueue `3`: `stack1 = [1, 2, 3]`, `stack2 = []`

Dequeue Operation (Removing from the Queue)

1. If `stack2` is empty:

    - Transfer all elements from `stack1` to `stack2` by popping each element from `stack1` and pushing it onto `stack2`. This reverses the order.
    - For example:
        - Before transfer: `stack1 = [1, 2, 3]`, `stack2 = []`
        - After transfer: `stack1 = []`, `stack2 = [3, 2, 1]`
2. Pop the top element from `stack2` (this is the oldest element from the queue's perspective).

If `stack2` is not empty, directly pop from `stack2`.

Code:

```
class QueueUsingStacks:
   def __init__(self):
       self.stack1 = []  # Stack 1 for enqueue
       self.stack2 = []  # Stack 2 for dequeue

   def enqueue(self, item):
       self.stack1.append(item)

   def dequeue(self):
       if not self.stack2:
           while self.stack1:
               self.stack2.append(self.stack1.pop())
       if not self.stack2:
           raise IndexError("Queue is empty")
       return self.stack2.pop()
```

3. Assign the Most Appropriate Data Structure and Explain [6 Marks]

i) Traversing in Reverse Order:

- **Most Appropriate:** Stack
- **Why:** A stack follows a Last-In-First-Out (LIFO) order, making it efficient for reversing the traversal order by pushing elements onto the stack and then popping them off.

ii) Ensuring First-In-First-Out Processing:

- **Most Appropriate:** Queue
- **Why:** A queue is inherently designed for First-In-First-Out (FIFO) operations, ensuring the first element added is the first one processed.

iii) Tracking Function Calls in Recursion:

- **Most Appropriate:** Stack
- **Why:** Function calls in recursion can be tracked using a stack, as each function call is pushed onto the stack and popped off when it returns, maintaining the correct execution order it also naturally matches the way recursive calls work (LIFO - Last In First Out), where the most recent function call needs to be completed first.

4. Write the adjacency Matrix for the following graph [4mks]

From vertex A:

- Has edge to B
- Has edge to D

From vertex B:

- Has self-loop (edge to itself)
- Has edge to C
- Has edge to D

From vertex C:

- Has edge to A
- Has edge to D

From vertex D:

- Has edge ONLY to C

Therefore the correct adjacency matrix is:

	A	B	C	D
A	0	1	0	1
B	0	1	1	1
C	1	0	0	1
D	0	0	1	0

5. Describe using an example how a node can be deleted after a given node in a linked list. [6 mks]

Example of a linked list:

Head -> A -> B -> C -> D -> E -> NULL

Here, we want to delete the node after node B (which is node C).

Steps to Delete a Node After a Given Node

1. **Identify the Node**: Start with the node after which you want to delete the next node. In this case, we have node B.
2. **Check if Next Node Exists**: Ensure that the next node (C) exists. If it doesn't, there's nothing to delete.
3. **Adjust the Links**: Change the `next` pointer of the current node (B) to skip the node to be deleted (C) and point to the node after it (D).
4. **Delete the Node**: Free the memory of the deleted node (optional in languages like Python but mandatory in C++).

Code:

```
def delete_node_after_node(previous_node):

   if previous_node is None or previous_node.next is None:
       print("Nothing to delete.")
       return  # No deletion can occur

   # Identify the node to be deleted
   node_to_delete = previous_node.next

   # Bypass the node to be deleted
   previous_node.next = node_to_delete.next

   # Optionally delete the node (Python's garbage collector handles memory cleanup)
   del node_to_delete

   print("Node deleted successfully.")
```