**Experiment 6**

(PART B : TO BE COMPLETED BY STUDENTS)

***(Students must submit the soft copy as per following segments within two hours of the practical. The soft copy must be uploaded on the Blackboard or emailed to the concerned lab in charge faculties at the end of the practical in case the there is no Black board access available)***

|  |  |
| --- | --- |
| Roll No. C013 | Name: Ashmit Jain |
| Class : B | Batch : B1 |
| Date of Experiment: | Date of Submission: |
| Grade : | Time of Submission: |
| Date of Grading: |  |

**B.1 Software Code written by student:**

***(Paste your code completed during the 2 hours of practical in the lab here)***

**Task1:**

Write a C/C++ program to implement operations of stack using linked list.

****

****

**Task2:**

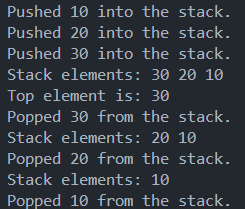
Write a C/C++ program to implement operations of queue using linked list.

****

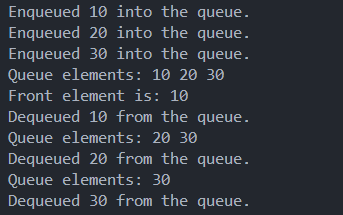
**B.2 Input and Output:**

***(Paste your program input and output in following format, If there is error then paste the specific error in the output part. In case of error with due permission of the faculty extension can be given to submit the error free code with output in due course of time. Students will be graded accordingly.)***

**Task1:**

****

**Task2:**

****

**B.3 Observations and learning [w.r.t. all tasks]:**

***Observations for the given experiment are as follows:***

**Stack (LIFO - Last In First Out) Using Linked List:**

* **Push Operation**:
  + A new node is inserted at the **top** of the stack.
  + The newly inserted node’s next pointer is set to point to the previous top node, making it the new top of the stack.
  + Time complexity: **O(1)** for each push, as insertion is done at the head of the list.
* **Pop Operation**:
  + Removes the top node from the stack, i.e., the node at the head of the linked list.
  + After removal, the next node becomes the new top.
  + Time complexity: **O(1)** for each pop, as removal happens at the head of the list.
* **Peek Operation**:
  + Directly returns the value at the top of the stack without removing it.
  + Time complexity: **O(1)**.
* **Advantages**:
  + Dynamic size: The stack can grow and shrink dynamically as nodes are added or removed, unlike arrays where size must be predefined.
  + No need to shift elements as in array-based implementations.

#### Queue (FIFO - First In First Out) Using Linked List:

* **Enqueue Operation**:
  + A new node is inserted at the **rear** of the queue.
  + The rear pointer is updated to point to this new node.
  + Time complexity: **O(1)** for each enqueue, as insertion is done at the tail of the list.
* **Dequeue Operation**:
  + The node at the **front** (head) of the queue is removed.
  + The front pointer is updated to point to the next node in the list.
  + Time complexity: **O(1)** for each dequeue, as removal happens at the head of the list.
* **Peek Operation**:
  + Returns the value of the front node without removing it.
  + Time complexity: **O(1)**.
* **Advantages**:
  + Dynamic size: Like the stack, the queue can grow and shrink dynamically based on the number of elements.
  + No need for element shifting as required in array-based implementations when dequeuing.

**Common Observations:**

* **Dynamic Nature**:
  + Both stack and queue implementations using linked lists are dynamic, allowing for flexible memory usage.
  + Unlike array-based implementations, you don’t need to allocate a fixed amount of memory upfront.
* **Memory Efficiency**:
  + Memory is allocated only when needed (during push or enqueue operations).
  + However, linked list implementations do have a small memory overhead for storing pointers (next) in addition to data.
* **No Overflow**:
  + Since memory allocation is dynamic, there’s no risk of overflow (as with arrays when the predefined capacity is reached), except when system memory is exhausted.
* **Time Complexity**:
  + Both implementations have constant time complexity, **O(1)**, for key operations (push, pop, enqueue, dequeue, peek).
  + Linked lists avoid the **O(n)** worst-case time complexity of shifting elements in array-based stacks/queues (especially for dequeuing or popping in arrays).
* **Destruction and Cleanup**:
  + Both the stack and queue implementations require careful cleanup to avoid memory leaks. The destructor is used to free up the memory allocated for nodes when the stack or queue is destroyed.
* **Performance Trade-offs**:
  + Linked list implementations might have slightly worse cache performance than array-based implementations, as linked list nodes are not stored contiguously in memory.

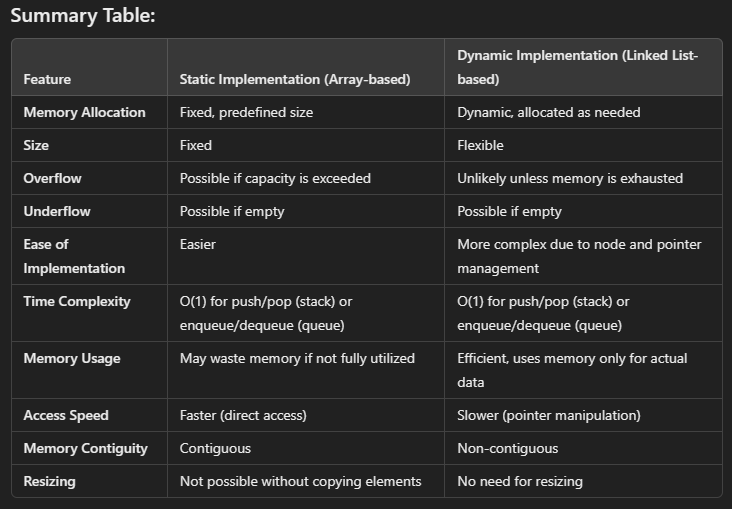
**B.4 Conclusion:**

* The linked list approach to implementing both stacks and queues is efficient in terms of dynamic size management and constant-time operations for insertion and removal.
* It provides more flexibility compared to array-based implementations, which may have fixed sizes and require costly element shifting for certain operations.

**B.5 Question of Curiosity**

***(To be answered by student based on the practical performed and learning/observations)***

**Q.1 Differentiate between static and dynamic implementation of stack and queue.**



* **Static (Array-based)**:
* This implementation is ideal for use cases where the maximum size of the stack or queue is known in advance and does not change frequently. It offers faster operations with minimal overhead but is limited in flexibility.
* **Dynamic (Linked List-based)**:
* This implementation is better suited for scenarios where the number of elements in the stack or queue is highly variable. While it may have some performance overhead due to dynamic memory management, it offers greater flexibility and memory efficiency.