**Lab-5: 19CSE212 Data Structure and Algorithms**

**Implementation of Stack and Queue Operations using Linked List**

1. **Quick Sort**

**Pseudocode:**

**Quick\_Sort(A, low, high)**

**{**

**if low < high**

**{**

**loc = Partition (A, low, high)**

**Quick\_Sort(A, low, loc-1)**

**Quick\_Sort(A, loc+1, high)**

**}**

**}**

**Partition (A, low, high)**

**{**

**Pivot := A[low]**

**i := low**

**j := high**

**while i < j**

**{**

**While (A[i] <= pivot)**

**{**

**i++**

**}**

**While (A[i] > pivot)**

**{**

**j++**

**}**

**If (i < j)**

**Swap (A[i], A[j])**

**}**

**while i > j**

**{**

**Swap (A[j], A[low])**

**return j**

**}**

**}**

**Lab Assignments**

1. **Analyse the recursive Quick Sort algorithm to compute its time and space complexities.**
2. **Implement the recursive algorithm to sort a list of n numbers in non-decreasing order.**
3. **Design and implement the recursive quick sort algorithm to sort a list of numbers in non-increasing order.**
4. **Implementation of Stack Operations using Linked List**

**Python Code:**

**//\* Node class** defines the structure of each node in the **linked list**. It has two instance variables: **next** and **data**, which represent the item being added to the stack, and a pointer to the next node in the stack. If a **Node object** is created without any arguments, the default value of data is **None**. **\*//**

1. **class** Node:
2. **def** \_\_init\_\_(self, data = None):
3. self. data = data
4. self. next = None

**//\* Stack class** represents the stack. It has only one instance variable, i.e., the **top**, which is a pointer to the top node of the stack. If the stack is **empty**, then assign top to **None \*//**

1. **class** Stack:
2. **def** \_\_init\_\_(self):
3. self.top = None

**//\* Push method** adds a new node to the top of the stack. It creates a new Node object with the given data and sets its next pointer to the current top of the stack. The top pointer is then updated to include the new node **\*//**

1. **def** push(self, data):
2. new\_node = Node(data)
3. new\_node.next = self.top
4. self.top = new\_node

**//\* Pop method** removes and returns the top item from the stack. It first checks if the stack is **empty** (i.e. if the top is **None**). If it's not empty, it saves the data of the current top node, updates the top to point to the next node in the stack, and returns the saved data.**\*//**

1. **def** pop(self):
2. **if** self.is\_empty():
3. **return** None
4. data = self.top.data
5. self.top = self.top.next
6. **return** data

**//\* Peek method** returns the data of the top item in the stack. It begins by making sure the stack is **empty**. If it's not empty, it returns the data attribute of the top node..**\*//**

1. **def** peek(self):
2. **if** self.is\_empty():
3. **return** None
4. **return** self.top.data
5. **def** is\_empty(self):
6. **return** self.top **is** None

**The advantage of Linked List Implementation of Stack over Array Implementation**

* The size of the stack can be **dynamic**, meaning that it can grow or shrink as items are added or removed. Unlike an array implementation of a stack, a **linked list implementation does not require a fixed size**.

**Time Complexity**

* The **push, pop, peek**, and **is\_empty** methods for a linked list implementation of a stack have a time complexity of **O(1),** because these operations only involve modifying the head node of the linked list, which takes constant time.

1. **Implementation of Queue Operations using Linked List**

**3.(a): Linear Queue**

**Operations: 1) Enqueue**

**2) Dequeue**

**Python Code:**

**//\* A linked list (LL) node stores a queue entry** **\*//**

**class** Node:

**def** \_\_init\_\_(self, data):

        self.data **=** data

        self.next **=** None

**// \*** The **Queue class** represents a queue. The queue’s **front** stores the front node of the Linked List, and the **rear** stores the last node of the Linked List **\*\\**

**class** Queue:

**def** \_\_init\_\_(self):

        self.front **=** self.rear **=** None

**def** isEmpty(self):

**return** self.front **==** None

**// \*** The **EnQueue Method** inserts an item into the queue. **\*//**

**def** EnQueue(self, item):

        temp **=** Node(item)

**if** self.rear **==** None:

            self.front **=** self.rear **=** temp

**return**

        self.rear.next **=** temp

        self.rear **=** temp

**// \*** The **DeQueue Method** deletes an item from the queue. **\*//**

**def** DeQueue(self):

**if** self.isEmpty():

**return**

        temp **=** self.front

        self.front **=** temp.next

**if**(self.front **==** None):

            self.rear **=** None

**//\*** **The Driver Code** **\*//**

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    q **=** Queue()

    q.EnQueue(item\_1)

    q.EnQueue(item\_2)

    q.DeQueue()

    q.DeQueue()

    q.EnQueue(item\_3)

    q.EnQueue(item\_4)

    q.EnQueue(item\_5)

    q.DeQueue()

**print**("Queue Front : " **+** str(q.front.data **if** q.front !**=** None **else** **-**1))

    print("Queue Rear : " **+** str(q.rear.data **if** q.rear !**=** None **else** **-**1))

**Lab Assignments**

1. **Implement the enqueue, and dequeue operations on a circular queue using a linked list.**
2. **Suppose you have a queue D containing the numbers (1, 3, 5, 7,2, 4, 6, 8), in this order. Suppose further that you have an initially empty queue Q. Give a code fragment that uses D and Q and results in D storing the elements in the order (1, 2, 3, 5, 4, 6, 7, 8). Elements are added left to right.**
3. **Suppose you have a stack S containing n elements and a queue Q that is initially empty. Describe how you can use Q to scan S to see if it contains a certain element x, with the additional constraint that your algorithm must return the elements to S in their original order. You may only use S, Q, and a constant number of other primitive variables.**
4. **Design an algorithm to implement the stack ADT using a single queue as an instance variable, and only constant additional local memory within the method bodies. Analyze the running time of the push(), pop(), and top() methods for your algorithm. Implement the algorithm to showcase the operations.**
5. **Show how to use a stack S and a queue Q to generate all possible subsets of an n-element set T non-recursively.**