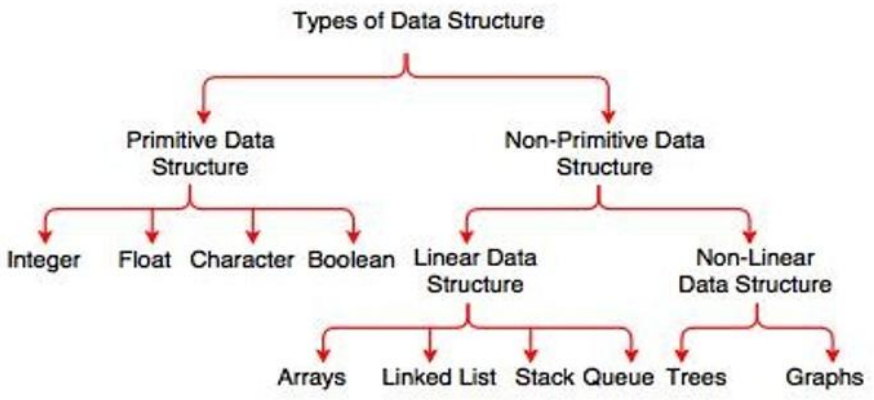
Q. NO.	SUB Q.	STEPWISE SOLUTION	MARKS	TOT MARKS
Q1	**A**	**Attempt following (2 marks)**		
	a	**Enlist the types of Data Structure.**  **Types of Data Structure :** 1. Array 2. Link-List 3. Stack 4. Queue 5. Trees 6. Graphs		
	B	**Attempt any ONE (4 marks)**		
	a)	**Explain Time Complexity and Space Complexity.** **<u>Time Complexity :</u>** 1) Time complexity of an algorithm quantifies (states that) the amount of time taken by an algorithm to run as a function of the length of the input. 2) Every algorithm requires some amount of computer time to execute its instruction to perform the task. This computer time required is called time complexity.	**02**	

3) The time complexity is mainly calculated by counting the number of steps to finish the execution.
4) The time complexity of algorithms is most commonly expressed using the big O notation.
5) There are three asymptotic notations that are used to represent the time complexity of an algorithm. They are:

a) Θ Notation (theta)
b) Big O Notation
c) Ω Notation

- The time required by an algorithm falls under three types

1) **Worst Case:** In which we analyse the performance of an algorithm for the input, for which the algorithm takes long time or space.
2) **Average Case:** In which we analyse the performance of an algorithm for the input, for which the algorithm takes time or space that lies between best and worst case.
3) **Best Case:** In which we analyse the performance of an algorithm for the input, for which the algorithm takes less time or space.

Space Complexity :

1) Space complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input.
2) There are two main parts of space complexity one is fixed part. And another is variable part.
3) A fixed part that is a space required to store certain data and variables.(i.e. simple variables and constants, program size etc.), that are not dependent of the size of the problem.
4) A variable part is a space required by variables, whose size is totally dependent on the size of the problem.
5) For example, recursion stack space, dynamic .memory allocation etc

02

	b)	**Write about i) Abstract Data Type ii) Operations on Data Structures** 1. **Abstract Data Type :** An Abstract Data Type (ADT) is a conceptual model that defines a data structure by its behavior (operations), rather than by its implementation. It specifies what operations can be performed on the data and how they should behave, without dictating how the operations are implemented internally. Example:- 1) Stack 2) Queue 3) Graph 4) Tree 5) Array 6) Linked List 2. **Operations on Data Structure :** Data structures organize and store data efficiently. Different operations are performed on them based on the type of data structure and its intended use. 1) Insert – Add an element. 2) Delete – Remove an element. 3) Search – Find an element. 4) Traverse – Visit all elements in the ADT. 5) Sort – Arrange elements in a specific order. 6) Merge – Combine two ADTs into one. 7) Size – Get the number of elements.	**02** **02**	

Q2	A	Attempt Any THREE (12 marks)		
	a)	**Define following. i. Linear Search ii. Binary Search**		

Linear Search : Linear search is a simple searching algorithm used in data structures. It works by sequentially checking each element of a list or array until the desired element is found or the list ends. **02**

Example:-
Search for the number 5 in the array [2, 4, 1, 7, 5, 3].

Array: [2, 4, 1, 7, 5, 3]
Target: 5

Step 1: [2, _, _, _, _, _] → No match
Step 2: [_, 4, _, _, _, _] → No match
Step 3: [_, _, 1, _, _, _] → No match
Step 4: [_, _, _, 7, _, _] → No match
Step 5: [_, _, _, _, 5, _] → Match found at index 4 **02**

Binary Search : Binary search is an efficient searching algorithm used to find an element in a sorted array or list. It repeatedly divides the search interval in half and compares the target element with the middle element. If the target element is smaller, it narrows the search to the left half; if larger, it searches in the right half. This process continues until the target element is found or the search space is reduced to zero.

Example:-

Index	0	1	2	3	4	5	6
Value	2	4	6	8	9	12	15

Step 1:

1. Middle element = index 3 (value = 8).
2. 9 > 8, search the right half: [9, 12, 15].

Step 2:

1. New middle element = index 4 (value = 9).
2. 9 == 9, element found at index 4.

b) **Write algorithm for bubble sort.**

Bubble Sort is a simple comparison-based sorting algorithm. It works by repeatedly "bubbling" the largest unsorted element to the end of the list through adjacent comparisons and swaps.

Steps:

1) Compare two adjacent elements.
2) Swap them if they are in the wrong order (i.e., if the first is greater than the second).
3) Repeat this process for all elements, "bubbling" the largest unsorted element to the end.
4) After each full pass through the list, the next largest element is in its correct position.
5) Repeat this for the remaining unsorted portion of the list until the entire list is sorted.

Algorithm:

Step1: Repeat steps 2 for I=0 toN-1
Step2: repeat for J=0 to N-I
Step3: If A[J]>a[J+1],then
SWAP A[J] and A[J+1]
[End OF Inner Loop]
[End OF Outer Loop]
Step 4:EXIT

Where,

I: Outer loop index for passes (0 to N-1).

J: Inner loop index for comparisons (0 to N-I).

A: Array being sorted.

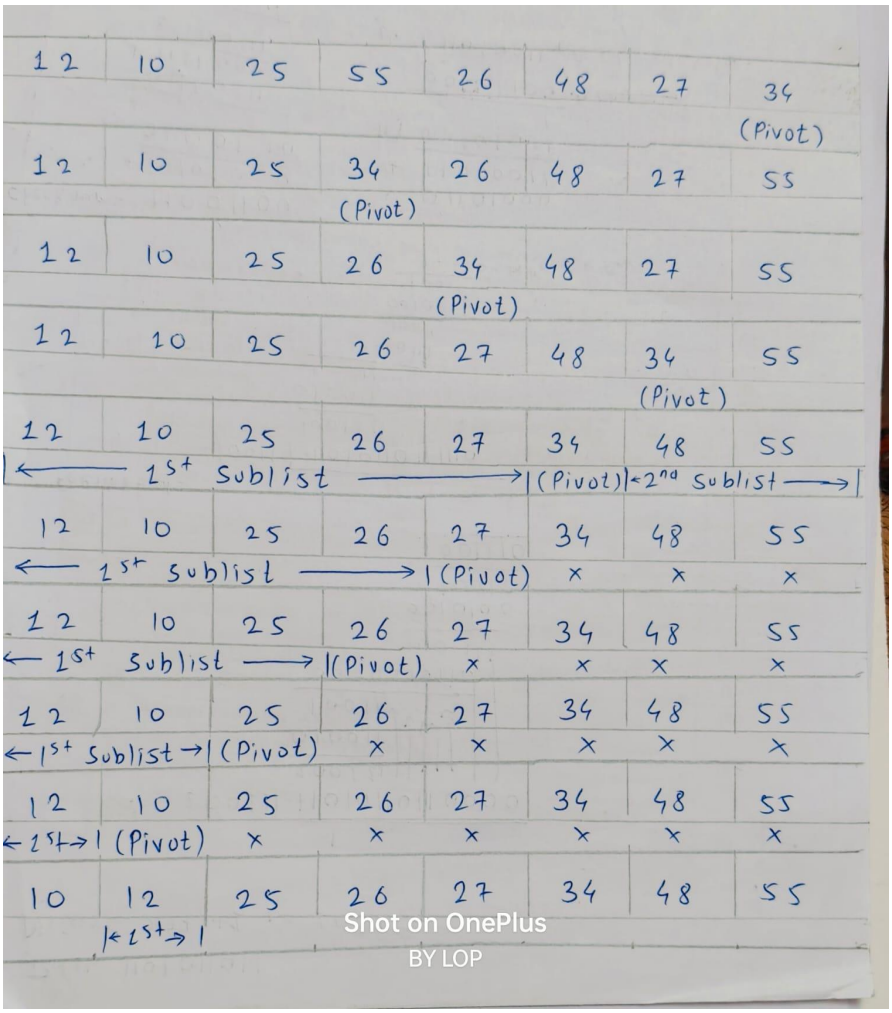
SWAP: Operation to exchange A[J] and A[J+1] if A[J] > A[J+1].

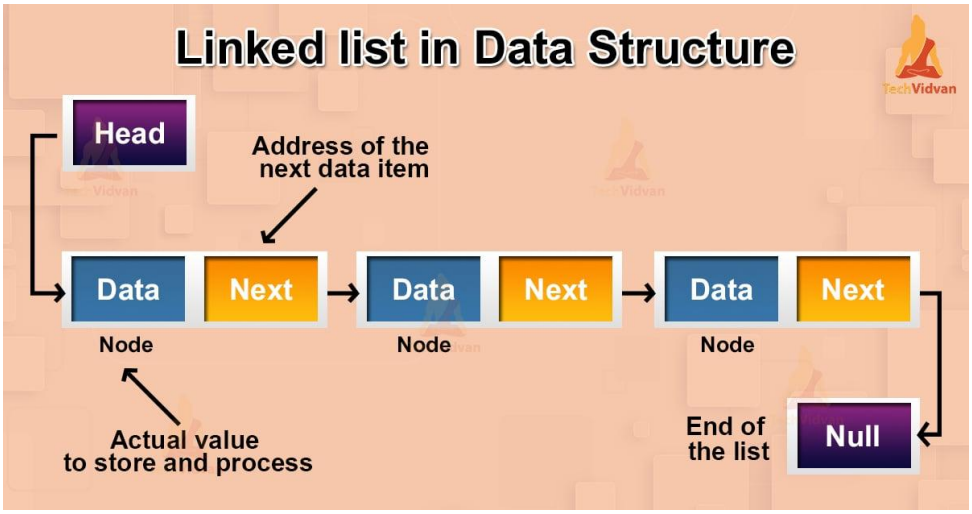
EXIT: End of sorting process.

c) **Sort list of numbers using Selection Sort:**
12 10 25 55 26 48 27 34

Selection Sort is a simple sorting algorithm that works by repeatedly finding the minimum (or maximum) element from the unsorted portion of the list and moving it to the beginning (or end).

No. of Passes								
Pass 1:-	10 (MIN)	12 (LOC)	25	55	26	48	27	34
Pass 2:-	10	12 (MIN) (LOC)	25	55	26	48	27	34
Pass 3:-	10	12	25 (MIN) (loc)	55	26	48	27	34
Pass 4:-	10	12	25	26 (MIN)	55 (LOC)	48	27	34
Pass 5:-	10	12	25	26	27 (MIN)	48	55 (LOC)	34
Pass 6:-	10	12	25	26	27	34 (MIN)	55	48 (LOC)
Pass 7:-	10	12	25	26	27	34	48 (MIN)	55 (LOC)
Sorted Array:-	10	12	25	26	27	34	48	55

	d)	**Apply Quick sort to sort list of numbers: 12 10 25 55 26 48 27 34** **(Note: select pivot element as last element of the list)** Quick Sort is a highly efficient divide-and-conquer sorting algorithm. It selects a pivot element from the array, then partitions the array such that elements smaller than the pivot are placed on its left and larger ones on its right. The process is recursively applied to the left and right sub-arrays.  		
Q3	**A**	**Attempt any THREE (12 marks)**		
	a)	**Define the term:** **i. Node ii. Information field iii. Next pointer iv. Null pointer**		



(i) Node : In data structures (DS), a node is a fundamental unit or building block of many complex structures, such as linked lists, trees, and graphs.

(ii) Information Field : The term "information field" in data structures typically refers to the portion of a data structure that holds the actual data or value. This is contrasted with other parts of the data structure that might hold pointers, references, or metadata.

(iii) Next Pointer : In data structures, a next pointer is a pointer that points to the next node in a sequence, typically in linked lists. It is an essential component in various types of linked lists (like singly linked lists, doubly linked lists, and circular linked lists) to maintain a reference to the next element in the sequence.

(iv) Null Pointer : In data structures, a null pointer is a special type of pointer that does not point to any valid memory location or object. It is typically used to signify that the pointer is not currently holding any valid data or memory address.

	b)	**Write an algorithm to insert an element at the beginning of Linked List** Inserting an element at the beginning of a linked list involves adding a new node before the current head of the list. This operation updates the list's head to point to the new node. **Steps:** 1. **Create a New Node**: Allocate memory for a new node and set its value. 2. **Point New Node to Current Head**: Set the new node's next pointer to the current head of the list. 3. **Update Head:** Change the head of the list to the new node. This makes the new node the first element in the list. **Algorithm:** Step 1: Create a new node, NEWNODE. Step 2: IF NEWNODE = NULL, then • Write OVERFLOW • Go to Step 7 • [END OF IF] Step 3: Set NEWNODE->DATA = VALUE • [Assign the data (VALUE) to the new node] Step 4: IF START = NULL, then • Set NEWNODE->NEXT = NULL • Set START = NEWNODE • Go to Step 6 • [END OF IF] Step 5: Set NEWNODE->NEXT = START • Set START = NEWNODE Step 6: Write Node inserted at the beginning. Step 7: EXIT. Where,		

1) **NEWNODE:** A pointer to the newly created node. It holds the new data and points to the next node in the list.
2) **VALUE:** The data you want to insert into the new node.
3) **START:** A pointer to the head of the linked list. It keeps track of the first node in the list.
4) **IF:** A conditional statement that executes code based on whether a condition is true.
5) **OVERFLOW:** An error message indicating that memory allocation for the new node failed (e.g., if NEWNODE is NULL).
6) **SET:** An operation to assign a value to a variable or pointer.
7) **NULL:** A special value indicating that a pointer does not point to any node (i.e., the end of the list).
8) **NEXT:** A pointer in a node that points to the next node in the linked list.
9) **WRITE:** An operation to output a message or result.
10) **EXIT:** Terminates the algorithm or program.

c) **Write an algorithm to delete an element at the end of Linked List**

Deleting the last node in a linked list involves navigating through the list, identifying the last node, and removing it by updating the second-to-last node's next pointer to NULL.

Steps:-

1) **Check if List is Empty:** If the list is empty (i.e., head == NULL), there is nothing to delete.
2) **Check for Single Node:** If the list contains only one node (head->next == NULL), delete the head and update head to NULL.
3) **Traverse the List:** Iterate through the list to find the second-to-last node.
4) **Update Pointers:** Set the second-to-last node's next pointer to NULL, effectively removing the last node.
5) **Delete the Last Node:** Free or deallocate the memory of the last node if required.

Algorithm:

Step1: IF START = NULL, THEN
Write UNDERFLOW (indicating that the list is empty)
Go to Step 8
[END OF IF]

Step2: IF START->NEXT = NULL, THEN
Set TEMP = START
Set START = NULL
Free TEMP
Go to Step 8
[END OF IF]
Step3: Set PTR = START

Step4: Repeat Steps 5 and 6 while PTR->NEXT->NEXT != NULL
[END OF LOOP]

Step5: Set PTR = PTR->NEXT

Step6: [END OF LOOP]

Step7: Set TEMP = PTR->NEXT
Set PTR->NEXT = NULL
Free TEMP

Step8: EXIT

Where,

1) **START**: Pointer to the first node in the linked list (i.e., the head of the list).
2) **PTR**: Pointer used to traverse the linked list, starting from `START`.

		3) **TEMP:** Temporary pointer used to store and delete the node that needs to be removed. 4) **NULL:** Represents the end of the list or an empty list. 5) **NEXT:** Refers to the pointer in each node that links to the next node. 6) **UNDERFLOW:** Indicates an empty list condition when there are no nodes to delete. 7) **Free:** Frees the memory of the node being deleted to avoid memory leaks. 8) **EXIT:** Terminates the algorithm.		
	d)	**Develop an algorithm to delete the node having last occurrence of the key (a character). Accept key from the user.** To delete the node with the last occurrence of a given key in a linked list, you need to traverse the entire list to locate the last occurrence of the key, then remove that node. The key is provided by the user, and the algorithm ensures only the last instance of the key is deleted. **Steps:** 1. **Traverse the List**: Go through the entire linked list while keeping track of nodes containing the key. 2. **Identify the Last Occurrence**: Record the node (or the node before it) where the last occurrence of the key appears. 3. **Delete the Node**: Update pointers to skip the node containing the last occurrence of the key. 4. **Handle Edge Cases**: Consider cases where the key is not found, or the list has only one node.		

Algorithm:

Step 1: Accept the key (KEY) from the user.

Step 2: IF START = NULL, THEN
    Write "List is empty"
    EXIT
[END OF IF]

Step 3: Initialize PTR = START, LAST_OCCURRENCE = NULL, and LAST_OCCURRENCE_PREV = NULL

Step 4: Traverse the list using PTR:
    WHILE PTR != NULL, DO
      IF PTR->DATA = KEY, THEN
        Set LAST_OCCURRENCE_PREV = LAST_OCCURRENCE
        Set LAST_OCCURRENCE = PTR
      [END OF IF]
      Set PTR = PTR->NEXT
    [END OF WHILE]

Step 5: IF LAST_OCCURRENCE = NULL, THEN
    Write "Key not found in the list"
    EXIT
[END OF IF]

Step 6: IF LAST_OCCURRENCE_PREV = NULL, THEN
    Set START = START->NEXT
ELSE
    Set LAST_OCCURRENCE_PREV->NEXT = LAST_OCCURRENCE->NEXT
[END OF IF]

Step 7: Free LAST_OCCURRENCE

Step 8: EXIT

		Where, 1) **START**: Pointer to the first node in the list. 2) **PTR**: Pointer to traverse the list node by node. 3) **LAST_OCCURRENCE**: Tracks the node containing the last occurrence of the key. 4) **LAST_OCCURRENCE_PREV**: Stores the previous node of the last occurrence, used to update the pointer to delete the node. 5) **KEY**: Character input by the user, used to find the node containing the last occurrence of the key.		