GOVERNMENT POLYTECHNIC ARVI

Model/Brief Solution & Structured Marking Scheme (PA TEST 1 – ODD 25-26)

Name of the Subject/ Course Code & Name: - DSU 313301

Name of Branch: COMPUTER ENGINEERING

Question/ Sub Q	Key Points/Key Steps/Formula/Diagram/Circuit/Step by Step Solution/Correct Answer etc	Marking Scheme	Max. Marks for Q/Sub Q
Q1.	**Attempt any FIVE**		10 Marks
Q.1. a)	***Define primitive data structure. Give 4 operations of data structure*** A **primitive data structure** is a fundamental, basic data type that is directly supported by a programming language. These structures are the built-in building blocks provided by the language to hold simple, single values. They are called "primitive" because they are not composed of other data structures; they are atomic. **Common Examples:** • **Integer (**int**):** Represents whole numbers (e.g., 5, -102, 0). • **Floating-Point (**float**,** double**):** Represents real numbers with decimal points (e.g., 3.14, -0.001). • **Character (**char**):** Represents a single symbol from a character set (e.g., 'A', '$', '3'). • **Boolean (**bool**):** Represents a logical value, typically true or false. • **Pointer/Reference:** Stores a memory address instead of a direct data value. Operations 1. Traversal 2. Insertion 3. Deletion 4. Searching 5.Sorting	02 Marks	
Q.1. b)	***Define Dynamic Memory Allocation*** **Dynamic Memory Allocation** is the process of assigning memory space for variables *during the runtime* (execution) of a program, rather than at compile time. This memory is allocated from a special area of the computer's RAM called the **heap**. This is in contrast to **static memory allocation**, where memory for variables (like global or static variables) is allocated at compile time from the **stack**.	02 Marks	

Q.1. c)	***Define the following (a) Sorting (b) Traversing*** **(a) Sorting** Definition: Sorting is the process of arranging the elements in a list or collection in a specific, logical order. The most common orders are numerical (1, 2, 3...) and alphabetical (a, b, c...), which are both forms of ascending order. **(b) Traversing** Definition: Traversing is the process of accessing and visiting each element in a data structure exactly once to perform a specific operation on it. It is one of the most fundamental operations and is a building block for many other functions like searching, printing, or calculating the sum of all elements.	02 Marks	

Q.1. d)	***Write the algorithm to Delete element in array.***
	DeleteFromArray(arr, n, pos)
	// Step 1: Check for validity
	IF n <= 0 THEN
	PRINT "Array is empty. Deletion not possible."
	RETURN
	END IF
	IF pos < 0 OR pos >= n THEN
	PRINT "Invalid position for deletion."
	RETURN
	END IF
	// Step 2: Shift elements to the left
	FOR i FROM pos TO n - 2
	arr[i] = arr[i + 1]
	END FOR
	// Optional: Set the last element to a default value (like 0 or NULL)
	// arr[n-1] = 0
	// Step 3: Decrement the size
	n = n – 1
		02 Marks	

Q.1. e)	***Enlist the various searching and sorting techniques available in data structure.*** **Searching Techniques** Searching is the process of finding the location of a target element (or key) within a data structure. **1. Linear Search** **2. Binary Search** **Sorting Techniques** Sorting is the process of arranging data in a particular order (ascending or descending). 1. Bubble Sort: 2. Selection Sort: 3. Insertion Sort: 4. Merge Sort: 5. Quick Sort:	02 Marks	
Q.1. f)	***Write an advantage of Linked List over array*** The most significant advantage of a Linked List over an array is its dynamic size and efficient memory allocation. In an Array: The size must be specified at the time of declaration. It is a static data structure. If you need to add more elements than the array can hold, you must create a new, larger array and copy all the elements over, which is a computationally expensive operation (O(n) time complexity). This can also lead to wasted memory if the array is allocated too large. In a Linked List: The size can grow or shrink dynamically at runtime. Memory is allocated for each new element (node) as and when it is needed. There is no need to pre-allocate a block of memory. This provides incredible flexibility and prevents memory wastage. This leads to two major benefits: Efficient Insertions/Deletions: Inserting or deleting a node at the beginning or middle of a linked list is a constant-time operation (O(1)) if you have a pointer to the location, as it only requires updating a few pointers. In an array, the same operation requires shifting all subsequent elements, which is an O(n) operation.	02 Marks	

	No Memory Wastage: You only use as much memory as you have nodes. There is no unused allocated memory sitting idle, as can happen with a sparsely populated array.		

Q.1. g)	***Write the steps to create a node in linked list***.
	Steps to Create a Node
	Step 1: Define the Node Structure
	Before creating nodes, you must define what a node looks like. This involves creating a structure (or a class) that contains:
	A variable to hold the data.
	A pointer variable of the same type as the structure to hold the address of the next node.
	Step 2: Allocate Memory Dynamically
	Since linked lists are dynamic data structures, memory for a new node is not allocated at compile time but during program execution (runtime).
	Use a dynamic memory allocation function (like malloc() in C or new in C++) to request a block of memory from the heap. The size of this block should be equal to the size of the node structure you defined.
	Step 3: Check if Memory Allocation was Successful
	It is crucial to check if the dynamic memory allocation was successful. If the system is out of memory, the allocation function will return a NULL pointer. Your program should handle this error to avoid a crash.
	Step 4: Assign Data to the Node
	Once memory is successfully allocated, you can assign the desired value to the data part of the new node.
	e.g., new_node->data = 10;
	Step 5: Initialize the Pointer (Next)
	The next pointer of the new node must be initialized. For a new node that is not yet connected to the list, this pointer is typically set to NULL to signify the end of the list.
	e.g., new_node->next = NULL;
	Step 6: Link the Node (Optional at Creation)
	The final step is to integrate the new node into the existing linked list. This involves updating the next pointer of an existing node to point to this new
		02 Marks	

	node. However, if you are creating the very first node (the head), you simply set the head pointer to point to this new node.		
Q.2.	**Attempt any FIVE**		20 Marks
Q.2.a)	***Explain the classification of Data structure with the help of examples*** Data structures are broadly classified into two main categories: **Primitive** and **Non-Primitive**. The non-primitive category is further divided into **Linear** and **Non-Linear** structures. **1. Primitive Data Structures** These are the fundamental data types that are built into a language and are the basic building blocks for manipulation. They can hold a single value. Definition: Basic, built-in data types supported directly by the programming language. Characteristics: They store a single value and are atomic (not composed of other data types). Examples: Integer (int): Used to store whole numbers. (e.g., 5, -102, 0) Floating-Point (float, double): Used to store real numbers with decimal points. (e.g., 3.14, -0.001) Character (char): Used to store a single character. (e.g., 'A', '$', '3') Boolean (bool): Used to store logical values, true or false. Pointer: A variable that stores the memory address of another variable. **2. Non-Primitive Data Structures** These are complex data structures that are derived from primitive data types. They are used to organize and manage large sets of data efficiently. They are also called "user-defined" data structures. A**. Linear Data Structures** Elements are arranged in a sequential order, one after the other. Each element has a unique predecessor and successor (except the first and last). Static Data Structures: Their memory size is fixed at compile time. Array: A collection of elements of the same type stored in contiguous memory locations. (e.g., int scores[10];) Dynamic Data Structures: Their size can grow or shrink at runtime. Memory is allocated as needed.	04 Marks	

	Linked List: A collection of nodes, where each node contains data and a pointer to the next node. (e.g., Singly Linked List, Doubly Linked List) Stack: A Last-In, First-Out (LIFO) structure where elements are added and removed from the same end. (e.g., Browser's "Back" button, Function call management) Queue: A First-In, First-Out (FIFO) structure where elements are added at the rear and removed from the front. (e.g., Printer job scheduling, Ticket counter line) Deque (Double-Ended Queue): A queue where insertion and deletion can occur at both ends. **B. Non-Linear Data Structures** Elements are not arranged sequentially. An element can be connected to more than one element, representing hierarchical or arbitrary relationships. Trees: A hierarchical data structure consisting of nodes connected by edges. One node is designated as the root. Binary Tree: A tree where each node has at most two children. Binary Search Tree (BST): A binary tree where the left child has a value less than the parent, and the right child has a value greater than the parent. Used for efficient searching. Heap: A complete binary tree used to implement Priority Queues. (e.g., Min-Heap, Max-Heap) Graphs: A collection of nodes (vertices) connected by edges. There is no strict parent-child relationship, unlike trees. Directed Graph (Digraph): Edges have a direction (e.g., one-way roads). Undirected Graph: Edges have no direction (e.g., social network friendships). Hash Tables: A data structure that implements an associative array abstract data type, using a hash function to map keys to values.		
Q.2.b)	***Write an algorithm and draw flow chart for Linear Search Techniques.*** **Principle:** Linear Search sequentially checks each element of the list until a match for the target value is found or the entire list has been searched. **Algorithm in Pseudocode**		

	BEGIN LINEAR_SEARCH (ARR, N, TARGET)
	Step 1: [Initialize]
	SET i = 0
	SET found = -1 // -1 indicates not found
	Step 2: [Traverse the array]
	REPEAT STEP 3 for i = 0 to N-1
	Step 3: [Check for match]
	IF ARR[i] == TARGET THEN
	SET found = i // Record the position
	PRINT "Element found at index", found
	EXIT THE LOOP // Optional: Remove to find all occurrences
	END IF
	Step 4: [Check if not found]
	IF found == -1 THEN
	PRINT "Element not found in the array."
	END IF
		04 Marks	

Q.2. c)	***Write an algorithm to sort given array using Selection Sort technique.***
	BEGIN SELECTION_SORT (ARR, N)
	// ARR is the array to be sorted
	// N is the number of elements in the array
	Step 1: [Loop through the array]
	FOR i FROM 0 TO N-2 DO
	Step 1.1: [Assume the current index is the minimum]
	SET min_index = i
	Step 1.2: [Find the index of the minimum element in the unsorted part]
	FOR j FROM i+1 TO N-1 DO
	IF ARR[j] < ARR[min_index] THEN
	SET min_index = j
	END IF
	END FOR
	Step 1.3: [Swap the found minimum element with the first element of the unsorted part]
	IF min_index != i THEN
	SWAP(ARR[i], ARR[min_index])
	END IF END FOR
		04 Marks	

Q.2. d)	***Suppose the following numbers are sorted in array A.*** ***32, 51, 27, 85, 66, 23, 13, 57*** ***Write down sorting operation by considering each pass separately using Bubble sort Technique.*** **Principle of Bubble Sort:** It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted. In each pass, the largest unsorted element "bubbles up" to its correct position at the end. Initial Array: [32, 51, 27, 85, 66, 23, 13, 57] n = 8 elements. We need n-1 = 7 passes. **Pass 1: (Largest element, 85, bubbles to the end)** Goal: Place the largest element in the correct final position (index 7). Comparisons: Compare 32 and 51: 32 < 51? Yes. No swap. [32, 51, 27, 85, 66, 23, 13, 57] Compare 51 and 27: 51 < 27? No. Swap. [32, 27, 51, 85, 66, 23, 13, 57] <-- SWAP Compare 51 and 85: 51 < 85? Yes. No swap. [32, 27, 51, 85, 66, 23, 13, 57] Compare 85 and 66: 85 < 66? No. Swap. [32, 27, 51, 66, 85, 23, 13, 57] <-- SWAP Compare 85 and 23: 85 < 23? No. Swap. [32, 27, 51, 66, 23, 85, 13, 57] <-- SWAP Compare 85 and 13: 85 < 13? No. Swap. [32, 27, 51, 66, 23, 13, 85, 57] <-- SWAP Compare 85 and 57: 85 < 57? No. Swap. [32, 27, 51, 66, 23, 13, 57, 85] <-- SWAP Array after Pass 1: [32, 27, 51, 66, 23, 13, 57, 85] (Sorted part: [85]) **Pass 2: (Second largest element, 66, bubbles to its position)** Goal: Place the next largest element in position 6. Comparisons:	04 Marks	

	32 and 27: 32 < 27? No. Swap. [27, 32, 51, 66, 23, 13, 57, 85] <-- SWAP 32 and 51: 32 < 51? Yes. No swap. 51 and 66: 51 < 66? Yes. No swap. 66 and 23: 66 < 23? No. Swap. [27, 32, 51, 23, 66, 13, 57, 85] <-- SWAP 66 and 13: 66 < 13? No. Swap. [27, 32, 51, 23, 13, 66, 57, 85] <-- SWAP 66 and 57: 66 < 57? No. Swap. [27, 32, 51, 23, 13, 57, 66, 85] <-- SWAP (No need to compare with last element) Array after Pass 2: [27, 32, 51, 23, 13, 57, 66, 85] (Sorted part: [66, 85]) **Pass 3:Comparisons:** 27 and 32: 27 < 32? Yes. No swap. 32 and 51: 32 < 51? Yes. No swap. 51 and 23: 51 < 23? No. Swap. [27, 32, 23, 51, 13, 57, 66, 85] <-- SWAP 51 and 13: 51 < 13? No. Swap. [27, 32, 23, 13, 51, 57, 66, 85] <-- SWAP 51 and 57: 51 < 57? Yes. No swap. (No need to compare last two) Array after Pass 3: [27, 32, 23, 13, 51, 57, 66, 85] (Sorted part: [51, 57, 66, 85]) **Pass 4:Comparisons:** 27 and 32: 27 < 32? Yes. No swap. 32 and 23: 32 < 23? No. Swap. [27, 23, 32, 13, 51, 57, 66, 85] <-- SWAP 32 and 13: 32 < 13? No. Swap. [27, 23, 13, 32, 51, 57, 66, 85] <-- SWAP 32 and 51: 32 < 51? Yes. No swap. (No need to compare further) Array after Pass 4: [27, 23, 13, 32, 51, 57, 66, 85]		

	(Sorted part: [32, 51, 57, 66, 85]) **Pass 5: Comparisons:** 27 and 23: 27 < 23? No. Swap. [23, 27, 13, 32, 51, 57, 66, 85] <-- SWAP 27 and 13: 27 < 13? No. Swap. [23, 13, 27, 32, 51, 57, 66, 85] <-- SWAP 27 and 32: 27 < 32? Yes. No swap. (No need to compare further) Array after Pass 5: [23, 13, 27, 32, 51, 57, 66, 85] (Sorted part: [27, 32, 51, 57, 66, 85]) **Pass 6: Comparisons:** 23 and 13: 23 < 13? No. Swap. [13, 23, 27, 32, 51, 57, 66, 85] <-- SWAP 23 and 27: 23 < 27? Yes. No swap. (No need to compare further) Array after Pass 6: [13, 23, 27, 32, 51, 57, 66, 85] The array is now sorted. However, the algorithm doesn't know this yet and will run its final pass.		

Q.2. e)	***Write an algorithm to insert element in an array.***
	BEGIN Algorithm InsertIntoArray(arr, size, capacity, element, pos)
	// Step 1: Check for space
	IF size >= capacity THEN
	PRINT "Array is full. Insertion not possible (Overflow)."
	RETURN
	END IF
	// Step 2: Check for valid position
	IF pos < 0 OR pos > size THEN
	PRINT "Invalid position for insertion."
	RETURN
	END IF
	// Step 3: Shift elements to the right
	FOR i FROM size - 1 DOWN TO pos
	arr[i + 1] = arr[i]
		04 Marks	

	END FOR
	// Step 4: Insert the new element
	arr[pos] = element
	// Step 5: Update the size
	size = size + 1
	END Algorithm		

Q.2. f)	***Describe the creation of single linked list and various operations performed on the Single Linked List.***
	A Single Linked List is a linear data structure consisting of a sequence of nodes. Each node contains:
	Data: The value or information.
	Next: A pointer/reference to the next node in the sequence.
	The list is accessed by a head pointer, which points to the first node. The last node points to NULL, signifying the end of the list.
	1. Creation of a Single Linked List
	Creating a linked list involves defining the node structure and then building the list by adding nodes.
	Step 1: Define the Node Structure
	This is typically done using a struct (in C/C++) or a class (in Java/Python) that contains the data and a pointer to a node of the same type.
	C Code:
	struct Node {
	int data; // Data part
	struct Node* next; // Pointer to the next node
	};
	Step 2: Create Nodes
	Memory for each node is allocated dynamically from the heap.
	C Code to Create a Node:
	struct Node* head = NULL; // Initialize an empty list
	// Create the first node
	struct Node* firstNode = (struct Node*)malloc(sizeof(struct Node));
	firstNode->data = 10;
	firstNode->next = NULL;
	head = firstNode; // Head points to the first node
	Step 3: Link the Nodes
		04 Marks	

Subsequent nodes are created and linked by updating the next pointers.

C Code to Create and Link a Second Node:

```
struct Node* secondNode = (struct Node*)malloc(sizeof(struct Node));
secondNode->data = 20;
secondNode->next = NULL;
firstNode->next = secondNode; // Link first node to the second
```

Operations on a Single Linked List

1. Traversal

Description: Accessing every node of the list exactly once to process it (e.g., print, count).

Algorithm:

Start from the head.

While the current pointer is not NULL:

Process the current node (e.g., print current->data).

Move to the next node: current = current->next.

2. Insertion

Description: Adding a new node to the list.

a) Insert at the Beginning

Create a new node.

Set its next pointer to the current head.

Update the head to point to the new node.

b) Insert at the End (Append)

Traverse to the last node (where next is NULL).

Create a new node.

Set the next pointer of the last node to the new node.

c) Insert after a Given Node

Given a pointer to a specific node.

Create a new node.

Set the new node's next to the given node's next.

Set the given node's next to the new node.

3. Deletion

Description: Removing a node from the list.

a) Delete the First Node

Check if the list is empty (head == NULL).

Store the head in a temporary pointer.

Update the head-to-head->next.

Free the memory of the temporary pointer.

b) Delete the Last Node

Traverse to the second last node.

Set its next pointer to NULL.

Free the memory of the last node.

c) Delete a Node with a Given Key

Find the node to be deleted and keep track of its previous node.

Set the next of the previous node to the next of the node to be deleted.

Free the memory of the node to be deleted.

4. Searching

Description: Finding the first node that contains a given value (key).

Algorithm:

Start from the head.

Traverse the list. For each node:

If current->data == key, return the node or its position.

Else, move to the next node.

Return NULL or -1 if not found.

5. Updating

Description: Changing the value of a node.

Algorithm:

Use the search operation to find the node.

Once found, change its data field to the new value.

6. Counting Nodes

Description: Finding the number of nodes in the list (length).

Algorithm:

Initialize a counter to 0.

Start from the head.

Traverse the list. For each node, increment the counter.

Return the counter.