DATA STRUCTURE MSE SOLUTIONS

1. Apply Merge Sort to this list and show the process of dividing the list into halves and then merging the sorted halves.
   [38, 27, 43, 3, 9, 82, 10]

Answer:

Initial list:

38,27,43,3,9,82,10

Step 1: Divide the list into halves

Split: 38,27,43 and 3,9,82,10

Step 2: Merge the sublists back together in sorted order

1. Merge sublists of size 1:

- Merge 27 and 43: 27,43
- Merge 3 and 9: 3,9
- Merge 82and 10: 10,82

2. Merge sublists of size 2 and 3:

- Merge 38 and 27,43: 27,38,43
- Merge 3,9 and 10,82: 3,9,10,82

3. Merge the final two halves:

- Merge 27,38,43 and 3,9,10,82: 3,9,10,27,38,43,82

Final Sorted List:

3,9,10,27,38,43,82

2. Define Linked List. Write a function to count the number of nodes in a linked list.

Answer: A Linked List is a linear data structure where each element, called a node, is stored in a separate object. Each node contains two parts:

Data: The value stored in the node.

Next: A reference (or pointer) to the next node in the sequence.

Function to count the number of nodes in a linked list

```
// Function to count the number of nodes in the linked list
int countNodes(struct Node* head) {
    int count = 0;
    struct Node* current = head; // Initialize current to head of the list
    while (current != NULL) {     // Traverse the list until current is NULL
        count++;                  // Increment the count for each node
        current = current->next;  // Move to the next node
    }
    return count;                 // Return the final count
}
```

3. List any two advantages of doubly linked list over singly linked list. Write a function to traverse a doubly linked list in both reverse order.

Answer:

Advantages of Doubly Linked List Over Singly Linked List (not limited to):

I. **Bidirectional Traversal**: In a doubly linked list, each node contains pointers to both the next and the previous nodes. This allows for traversal in both forward and backward directions, making it more versatile for certain applications

II. **Easier Deletion:** In a doubly linked list, deleting a node is easier since we have access to the previous node. This means we can directly access the previous pointer without needing to traverse from the head of the list.

III. **Backtracking Capabilities:** Because you can traverse both forwards and backwards, doubly linked lists are beneficial for applications requiring backtracking, such as navigating through browser history or implementing undo functionality in applications

IV. **Memory Efficiency in Some Cases:** Although a doubly linked list uses more memory per node due to the additional pointer, it can lead to better memory locality and cache performance in certain scenarios, especially when performing multiple sequential operations.

V. **Easier Insertion**: Inserting a node before a given node is simpler in a doubly linked list since you have direct access to the previous node. In a singly linked list, you would need to traverse from the head to find the node before the one you want to insert before.

```c
// Function to traverse the doubly linked list in forward order
void traverseForward(struct Node* node) {
    printf("Forward traversal of doubly linked list: ");
    while (node != NULL) {
        printf("%d ", node->data);
        node = node->next;
    }
    printf("\n");
}
```

```c
// Function to traverse the doubly linked list in reverse order
void traverseReverse(struct Node* node) {
    if (node == NULL) return;

    // Move to the end of the list
    while (node->next != NULL) {
        node = node->next;
    }
    // Traverse backwards and print the data
    printf("Reverse traversal of doubly linked list: ");
    while (node != NULL) {
        printf("%d ", node->data);
        node = node->prev;
    }
    printf("\n");
}
```

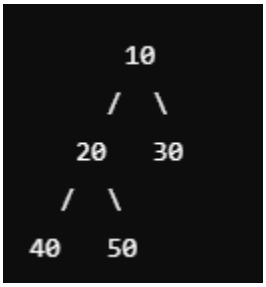
4. Define a binary tree and explain the following terminologies of tree data structure with example:

i. Root
ii. Degree of a node
iii. Depth of a Node

Answer:

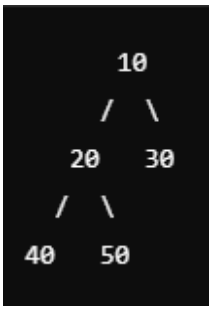
A **Binary Tree** is a hierarchical data structure in which each node has at most two children, referred to as the **left child** and **right child**. A binary tree can be empty (contain no nodes), or it can consist of a root node and two subtrees, each of which is a binary tree.

i. **Root:** The root of a tree is the topmost node in the tree. It is the only node that does not have a parent. In any tree, there is exactly one root node.



In this tree, 10 is the root node because it is the topmost node and has no parent.

ii. **Degree of a node:** The degree of a node is the number of children that a node has. In a binary tree, since each node can have at most two children, the degree of a node can be 0, 1, or 2.

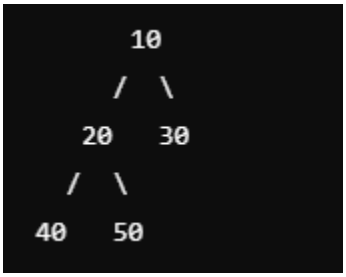


The degree of node 10 is 2 (since it has two children: 20 and 30).

The degree of node 20 is 2 (since it has two children: 40 and 50).

The degree of nodes 30, 40, and 50 is 0 (since they have no children, they are leaf nodes).

iii. **Depth of a Node:** The depth of a node is the number of edges (or levels) from the root to that node. The root node is at depth 0, and as you move down the tree, the depth increases by 1 at each level.



The **depth of the root node (10)** is 0.

The **depth of node 20** is 1 (one edge away from the root).

The **depth of nodes 40 and 50** is 2 (two edges away from the root).