SIES Graduate School of Technology (Autonomous) NAAC A+ (Affiliated to University of Mumbai)	***End Semester Examination (R-24) SH 2025***
Branch: ECS	Course: Data Structure
Year/ Semester: SE III	Course code: (ECC302)
Time: 03 hours	Marks: 80

Note: 1. All questions are compulsory.
2. Figures to right indicate full marks.
3. Assume suitable data wherever necessary.

		Marks
Q.1	Attempt any FOUR. (All questions carry equal marks)	**20**
A.	Sketch and explain one linear and one non-linear data structure.	05

Short Descriptions of various Data Structures

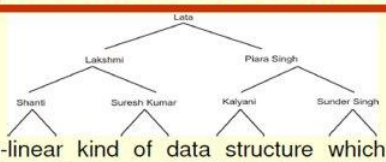
Array:

Amit	Jatin	Rajesh	Naveen	Sonia
1	2	3	4	5

- ✓ It is the linear collection of finite number of homogeneous data elements.
- ✓ If we consider an array with elements then elements of the array will be referred using index set consisting of consecutive numbers i.e. The elements of an array can be referred by using different notations:
- ✓ When array is stored into the computer's memory, its elements occupies the consecutive memory locations.
- ✓ The total number of elements in an array is known as the size of that a array and can be calculated by using a simple formula:
- ✓ Where is the lower index of the array and is the upper index of the array.
- ✓ Programming languages also support multidimensional arrays.

Short Descriptions of various Data Structures

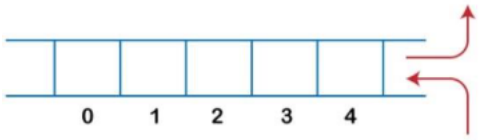
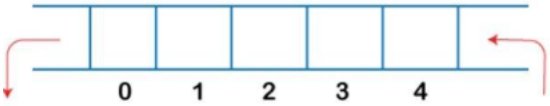
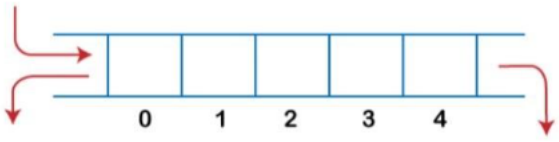
Tree:

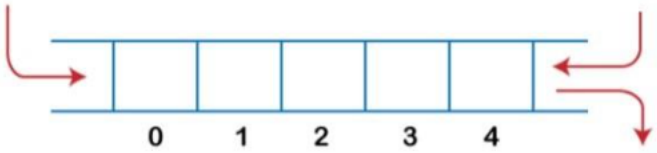


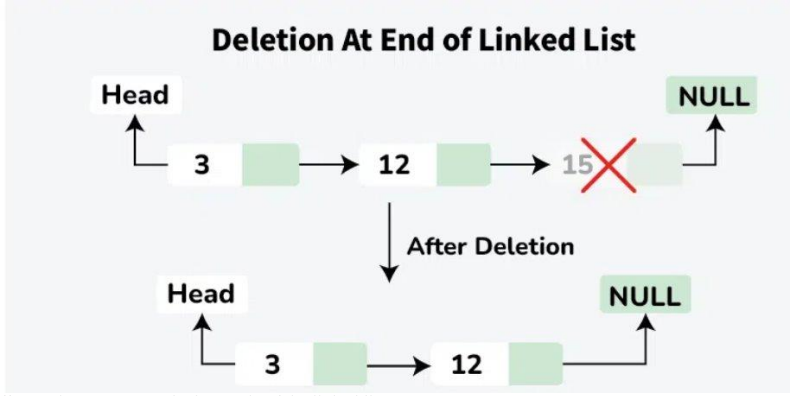
- ✓ Tree is a non-linear kind of data structure which is used to represent data elements having hierarchical relationship between them.
- ✓ Tree structure is also referred as parent child relationship. The basic difference between linear and non linear data structures is that in linear data structures, for each element there is a fixed next element but in case of non-linear data structure each element can have many different next elements.
- ✓ A very common example is the ancestor tree as shown in figure. This tree shows the ancestors of Lata. Her parents are Lakshmi and Piyara Singh. Lakshmi's parents are Shanti and Suresh Kumar. Piyara Singh's parents are Kalyani and Sunder Singh and so on.

B.	Explain how stack data structure is used in well form ness of parenthesis. Given a string **s** representing an expression containing various types of brackets: {}, (), and [], the task is to determine whether the brackets in the expression are balanced or not. A balanced expression is one where every opening bracket has a corresponding closing bracket in the correct order. *The idea is to put all the opening brackets in the stack. Whenever you hit a closing bracket, search if the top of the stack is the opening bracket of the same nature. If this holds then pop the stack and continue the iteration. In the end if the stack is empty, it means all*	05

	brackets are balanced or well-formed. Otherwise, they are not balanced.
	• *Step-by-step approach:*
	• *Declare a character stack (say temp).*
	• *Now traverse the string **s**.*
	  • *If the current character is an opening bracket ('(' or '{' or '[') then push it to stack.*
	  • *If the current character is a closing bracket (')' or '}' or ']') and the closing bracket matches with the opening bracket at the top of stack, then pop the opening bracket. Else **s** is not balanced.*
	• *After complete traversal, if some starting brackets are left in the stack then the expression is not balanced, else balanced.*	

C.	Explain the double ended queue. Explain its types • The deque represents Double Ended Queue. • In the queue, the inclusion happens from one end while the erasure happens from another end. • The end at which the **addition happens is known as the backside** while the **end at which the erasure happens is known as front end**. • Deque is a direct information structure in which the inclusion and cancellation tasks are performed from the two finishes. We can say that deque is a summed up form of the line. • Deque can be utilized both as stack and line as it permits the inclusion and cancellation procedure on the two finishes. • In deque, the inclusion and cancellation activity can be performed from one side. The stack adheres to the LIFO rule in which both the addition and erasure can be performed distinctly from one end; in this way, we reason that **deque can be considered as a stack.**  In deque, the addition can be performed toward one side, and the erasure should be possible on another end. The queue adheres to the FIFO rule in which the component is embedded toward one side and erased from another end. Hence, we reason that the **deque can likewise be considered as the queue.**  • There are two types of Queues, **Input-restricted queue,** and **output-restricted queue**. • **Information confined queue**: The info limited queue implies that a few limitations are applied to the inclusion. In info confined queue, the addition is applied to one end while the erasure is applied from both the closures.  **Yield confined queue:** The yield limited line implies that a few limitations are applied to	05

	the erasure activity. In a yield limited queue, the cancellation can be applied uniquely from one end, while the inclusion is conceivable from the two finishes.  	

D.	Illustrate and write a function to delete a node at the end of single link list
	 
	
	// Function to remove the last node of the linked list
	struct Node* removeLastNode(struct Node* head)
	{
	// If the list is empty, return NULL
	if (head == NULL) {
	return NULL;
	}
	
	// If the list has only one node, delete it and return
	// NULL
	if (head->next == NULL) {
	free(head);
	return NULL;
	}
		05

E.	Construct expression tree for given expression and find the post order traversal for the tree. a + (b * c) + d * (e + f)  a b c * + d e f + * +	05

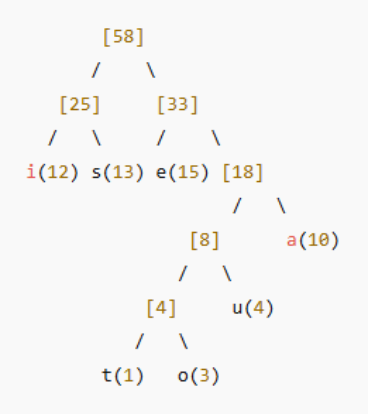
	Post order	
F.	What is the meaning of collision in hashing? Explain with an example. The hashing process generates a small number for a big key, so there is a possibility that two keys could produce the same value. The situation where the newly inserted key maps to an already occupied, and it must be handled using some collision handling technology.	05
Q.2	Attempt any FOUR. (All questions carry equal marks)	**40**
A.	Define and explain the stack data structure with suitable example. Give algorithms for Push, Pop functions. Insertion: push() ●push() is an operation that inserts elements into the stack. The following is an algorithm that describes the push() operation in a simpler way. Algorithm 1 − Checks if the stack is full. 2 − If the stack is full, produces an error and exit. 3 − If the stack is not full, increments top to point next empty space. 4 − Adds data element to the stack location, where top is pointing. 5 − Returns success. Deletion: pop() ●pop() is a data manipulation operation which removes elements from the stack. The following pseudo code describes the pop() operation in a simpler way. ●Algorithm 1 − Checks if the stack is empty. 2 − If the stack is empty, produces an error and exit. 3 − If the stack is not empty, accesses the data element at which top is pointing. 4 − Decreases the value of top by 1. 5 − Returns success.	10

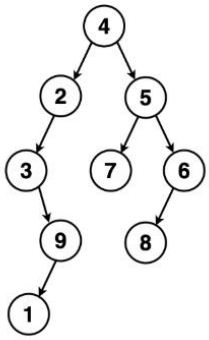
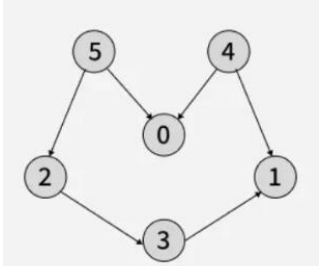
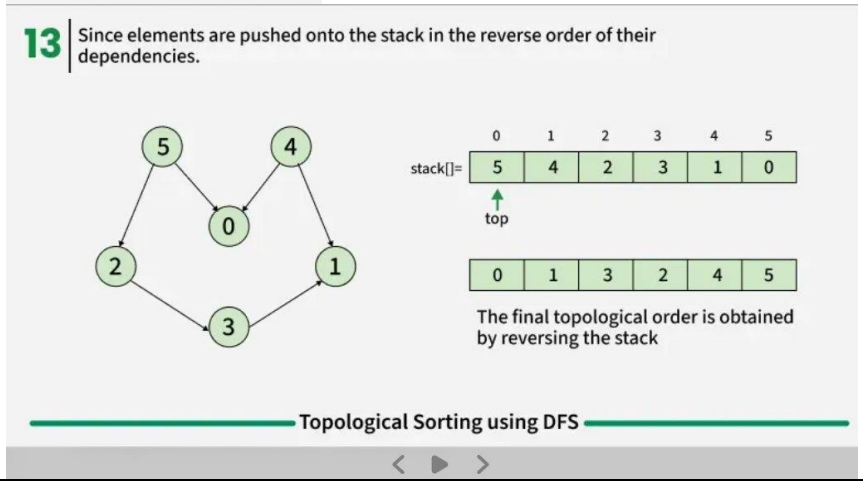
B.	Illustrate and write functions to add node in a doubly link list for all the cases.
	// Function to insert a new node at the front of doubly linked list
	struct Node *insertAtFront(struct Node *head, int new_data) {
	// Create a new node
	struct Node *new_node = createNode(new_data);
	// Make next of new node as head
	new_node->next = head;
	// Change prev of head node to new node
	if (head != NULL) {
	head->prev = new_node;
	}
	// Return the new node as the head of the doubly linked list
	return new_node;
	}
	Main()
	head = insertAtFront(head, data);
	// Function to insert a new node at the end of the doubly linked list
	struct Node* insertEnd(struct Node *head, int new_data) {
	struct Node *new_node = createNode(new_data);
	// If the linked list is empty, set the new node as the head
	if (head == NULL) {
	head = new_node;
	} else {
		10

```
        struct Node *curr = head;
        while (curr->next != NULL) {
            curr = curr->next;
        }
        // Set the next of last node to new node
        curr->next = new_node;
        // Set prev of new node to last node
        new_node->prev = curr;
    }
    return head;
}
// Function to insert a new node at a given position
struct Node* insertAtPosition(struct Node *head, int pos, int new_data) {
    // Create a new node
    struct Node *new_node = createNode(new_data);
    // Insertion at the beginning
    if (pos == 1) {
        new_node->next = head;
        // If the linked list is not empty, set the prev of head to new node
        if (head != NULL) {
            head->prev = new_node;
        }
        // Set the new node as the head of linked list
        head = new_node;
        return head;
    }
    struct Node *curr = head;
```

C. Construct huffman tree for the following — 10

Characters	a	e	i	o	u	s	t
Frequencies	10	15	12	3	4	13	1



	<table><tr><th>Character</th><th>Huffman Code</th></tr><tr><td>i</td><td>00</td></tr><tr><td>s</td><td>01</td></tr><tr><td>e</td><td>10</td></tr><tr><td>a</td><td>111</td></tr><tr><td>u</td><td>1101</td></tr><tr><td>t</td><td>11000</td></tr><tr><td>o</td><td>11001</td></tr></table>	
D.	Describe the three common binary tree traversal methods (inorder, preorder, postorder). Perform these traversals on the following binary tree and show the output: 	10
E.	Write the algorithm for topological sorting and perform the same on the graph given below  	10
F.	Describe hash function using mid square method and folding method with example. The mid-square method is a very good hashing method. It involves two steps to compute	10

the hash value-

- Square the value of the key k i.e. k2
- Extract the middle r digits as the hash value.

Formula:

$$h(K) = h(k \times k)$$

Here,

***k** is the key value.*

The value of **r** can be decided based on the size of the table.

Example:

Suppose the hash table has 100 memory locations. So r = 2 because two digits are required to map the key to the memory location.

k = 60

k x k = 60 x 60

= 3600

h(60) = 60

The hash value obtained is 60

Formula:

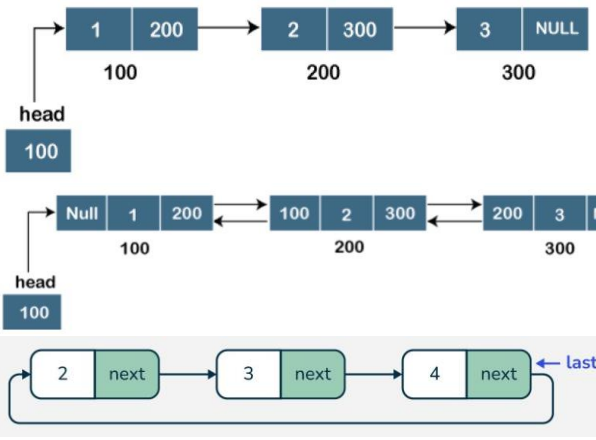
$$k = k1, k2, k3, k4, ..., kn$$

$$s = k1 + k2 + k3 + k4 + ... + kn$$

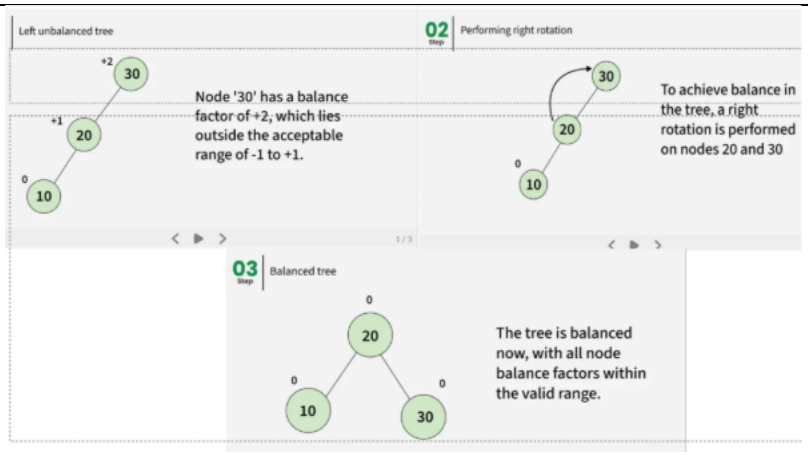
$$h(K) = s$$

Here,

s** is obtained by adding the parts of the key **k

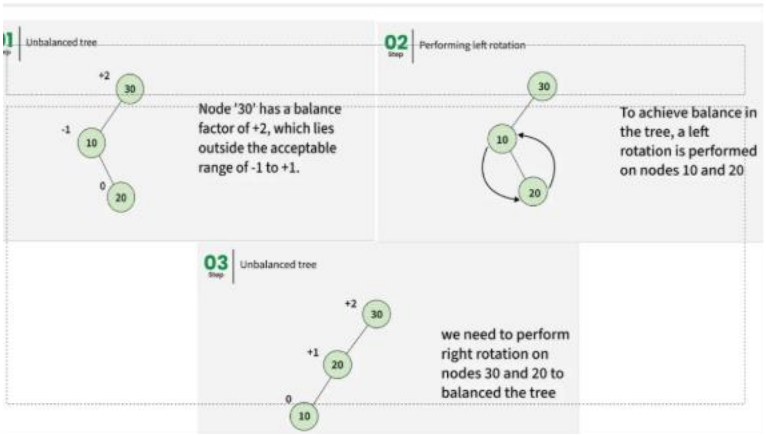
	Example: $k = 12345$ $k1 = 12, k2 = 34, k3 = 5$ $s = k1 + k2 + k3$ $= 12 + 34 + 5$ $= 51$ $h(K) = 51$	
Q.3	Attempt any FOUR	**20**
A.	Given a stack of Size 4 perform following operations in sequencePush(12), Push(25), Push (33), Pop(), Push(47), Push(51), Push(66), i)Determine the state of stack after each operation ii)Identify if any overflow or underflow conditions occur. Consider a stack of size 4. Initially, the stack is empty. After the first operation **Push(12)**, the stack becomes **[12]** with 12 at the top. The next operation **Push(25)** adds another element, making the stack **[12, 25]**. Then **Push(33)** adds one more element, resulting in **[12, 25, 33]**. When the **Pop()** operation is performed, the top element 33 is removed, leaving the stack as **[12, 25]**. The next operation **Push(47)** adds a new element to the top, so the stack becomes **[12, 25, 47]**. After that, **Push(51)** fills the last available space, giving **[12, 25, 47, 51]**, which makes the stack full. When the next operation **Push(66)** is attempted, it cannot be added because the stack has reached its maximum capacity, leading to a **Stack Overflow** condition. No **Underflow** occurs in this sequence since the stack is never empty when a pop operation is performed.	05
B.	Sketch and explain different types of link list  A **singly linked list** is the simplest type of linked list in which each node contains two parts — one for storing data and another for storing the address of the next node in the sequence. The last node points to **NULL**, indicating the end of the list. Traversal in a singly linked list is possible only in one direction, starting from the head node and moving toward the last node. It is easy to implement and uses less memory compared to other linked lists, but it	05

	does not allow backward movement, making certain operations like reverse traversal or deletion of the last node less efficient. A **doubly linked list** extends the concept of the singly linked list by adding a pointer to the **previous node** along with the next pointer. Each node in a doubly linked list therefore contains three parts — the previous pointer, data, and the next pointer. This structure allows traversal in both directions — forward and backward — providing greater flexibility in insertion and deletion operations. However, the additional pointer increases memory usage and makes implementation slightly more complex compared to a singly linked list. A **circular linked list** is a variation in which the last node is connected back to the first node, forming a circular structure. This can be implemented as either a singly or a doubly circular linked list. In a circular singly linked list, the next pointer of the last node points to the first node, while in a circular doubly linked list, both the first and last nodes are linked to each other in both directions. Circular linked lists are especially useful in applications like **round-robin scheduling**, **real-time systems**, and **continuous data processing**, where the data structure needs to be accessed in a repeating cycle.	
C.	Differentiate between array and link list <table><tr><th>Parameter</th><th>Array</th><th>Linked List</th></tr><tr><td>Size</td><td>Specified during declaration.</td><td>No need to specify; grow and shrink during execution.</td></tr><tr><td>Storage Allocation</td><td>Element location is allocated during compile time.</td><td>Element position is assigned during run time.</td></tr><tr><td>Order of the elements</td><td>Stored consecutively</td><td>Stored randomly</td></tr><tr><td>Accessing the element</td><td>Direct or randomly accessed, i.e., Specify the array index or subscript.</td><td>Sequentially accessed, i.e., Traverse starting from the first node in the list by the pointer.</td></tr><tr><td>Insertion and deletion of element</td><td>Slow relatively as shifting is required.</td><td>Easier, fast and efficient.</td></tr><tr><td>Searching</td><td>Binary search and linear search</td><td>linear search</td></tr><tr><td>Memory required</td><td>less</td><td>More</td></tr><tr><td>Memory Utilization</td><td>Ineffective</td><td>Efficient</td></tr></table>	05
D.	Explain Left-Left and Left-Right rotation in AVL tree with example **Left-Left Rotation:** • Occurs when a node is inserted into the left subtree of the left child, causing the balance factor to become **more than +1**. • **Fix:** Perform a single **right** rotation.	05

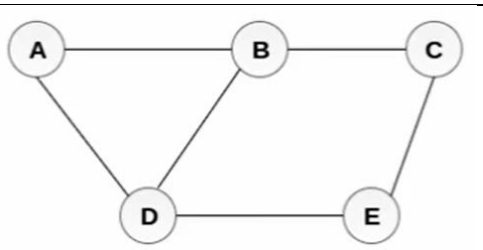


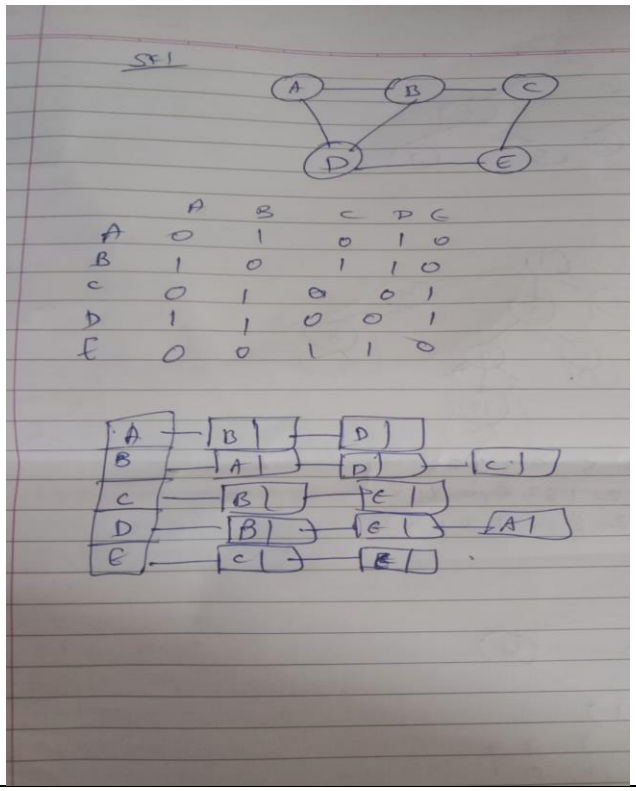
Left-Right Rotation:

- Occurs when a node is inserted into the right subtree of the left child,which disturbs the balance factor of an ancestor node, making it left-heavy.
- Fix: Perform a left rotation on the left child, followed by a rightrotation on the node.



E.	Write a short note on B trees A B Tree of order m can be defined as an m-way search tree which satisfies thefollowing properties: • All leaf nodes of a B tree are at the same level, i.e. they have the same depth(height of the tree). • The keys of each node of a B tree (in case of multiple keys), should be stored inthe ascending order. • In a B tree, all non-leaf nodes (except root node) should have atleast m/2 children. • All nodes (except root node) should have at least m/2 - 1 keys. • If the root node is a leaf node (only node in the tree), then it will have no childrenand will have at least one key. If the root node is a non-leaf node, then it will haveat least 2 children and at least one key. • A non-leaf node with n-1 key values should have (n) non NULL children. • We can see in the above diagram that all the leaf nodes are at the same leveland all non-leaf nodes have no empty sub-tree and have number of keys oneless than the number of their children.	05
F.	Illustrate the Adjacency list and adjacency matrix for the graph given below.	05


A
B
C
D
E


SF1
A
B
C
D
E
A B C D E
A 0 1 0 1 0
B 1 0 1 1 0
C 0 1 0 0 1
D 1 1 0 0 1
E 0 0 1 1 0
A
B
C
D
E
B
D
A
D
C
B
E
B
E
A1
C
E

***********All the Best***********