BCS304
Third Semester B.E./B.Tech Degree Examination, Dec. 2025 / Jan. 2026
DATA STRUCTURES AND APPLICATIONS

CBCS SCHEME

USN 1 C R 2 4 I S 0 1 4

BCS304

Third Semester B.E./B.Tech. Degree Examination, Dec.2025/Jan.2026

Data Structures and Applications

Time: 3 hrs. Max. Marks: 100

Note: 1. Answer any FIVE full questions, choosing ONE full question from each module.
2. M : Marks , L: Bloom's level , C: Course outcomes.

			M	L	C
		Module – 1			
Q.1	a.	Define Data Structure. Explain with neat diagram different types of data structure with examples. What are the primitive operations that can be performed?	10	L2	CO1
	b.	Define structure and union? Explain how they are different from each other, with suitable example	5	L2	CO1
	c.	What do you mean by pattern matching? Outline Kruth, pattern matching algorithm.	5	L2	CO1
		OR			
Q.2	a.	Define stack. Give the implementation of push ( ) POP ( ) and Display ( ) functions by considering its empty and full conditions.	7	L2	CO1
	b.	Write an algorithm to evaluate a postfix expression and apply the same for the given postfix expression 6, 2, 13, -4, 2, ×, +.	7	L3	CO1
	c.	Write the postfix form of the following using stack, (i) $A*(B*C+D*E)+F$ (ii) $(A+(B*C)/(D-E))$	6	L3	CO1
		Module – 2			
Q.3	a.	What are the disadvantages of ordinary queue? Discuss the implementation of circular queue.	8	L2	CO2
	b.	Write a note on multiple stacks and priority queue	6	L2	CO2
	c.	Define Queue. Discuss how to represent Queue using dynamic arrays.	6	L2	CO2
		OR			
Q.4	a.	What are Linked list? Explain the different types of Linked List with neat diagram.	4	L2	CO2
	b.	Give the structure definition for Singly Linked List (SSL). Write a C function to, (i) Insert an element at the end of SSL. (ii) Delete at node at the end of SSL.	8	L3	CO2

	c.	Write a C function to add two polynomials show the Linked List representation of below two polynomials $p(x)=3x^{10}+2x^2+1$ $q(x)=8x^{10}+5x^5+3x^2+2$	8	L3	CO2
		Module – 3			
Q.5	a.	Write a C-function for the following operation on doubly Linked List (DLL): (i) Addition of a DLL node (ii) Concatenation of two DLL	8	L3	CO3
	b.	Write a C-function for the following operations on circular Linked List (i) Inserting at the front of a List. (ii) Find the number of nodes in circular list.	8	L3	CO3
	c.	Represent the given Sparse matrix using linked list representation $A=\begin{bmatrix}2&0&0&0\\4&0&0&3\\0&0&0&0\\7&0&1&1\\0&0&6&0\end{bmatrix}$	4	L3	CO3
		OR			
Q.6	a.	Explain the different types binary tree representation with example	8	L3	CO3
	b.	Define Threaded Binary tree. Discuss in threaded binary tree.	4	L3	CO3
	c.	Discuss Inorder, preorder, postorder and level order traversal with suitable recursive function for each.	8	L2	CO3
		Module – 4			
Q.7	a.	Write a function to perform the following operations on Binary Search Tree (BST): (i) Inserting on element into BST (ii) Recursive search of a BST	8	L3	CO4
	b.	Discuss selection Trees with suitable example	8	L2	CO4
	c.	Explain transforming a forest into a binary tree with an example.	4	L2	CO4
		OR			
Q.8	a.	Define graph. Show the adjacency matrix and adjacency List representation of the graph given below  Fig. Q8 (a)	6	L3	CO4

	b.	Define the following Terminologies with examples (i) Vertex /node) (ii) Self loop (iii) Weighted graph (iv) Parallel edges	7	L1	CO4
	c.	Explain in detail elementary graph operations	7	L1	CO4
		Module – 5			
Q.9	a.	What is collision? What are the methods to resolve collision? Explain linear probing with example	8	L2	CO5
	b.	Explain in details about static and dynamic hashing	6	L2	CO5
	c.	Discuss Lefties Trees with an example	6	L2	CO5
		OR			
Q.10	a.	Explain different types of HASH functions with example	6	L2	CO5
	b.	Discuss different types of rotations with suitable examples	6	L3	CO5
	c.	Define Red-Black Tree. Splay tree. Discuss the method to insert an element into Red-Black tree	8	L2	CO5

*****

1 a) Define Data Structure Explain with, neat diagram different types of data structure with examples. What are the primitive operations that can be performed? (10 Marks)

Definition of Data Structure (2 Marks)

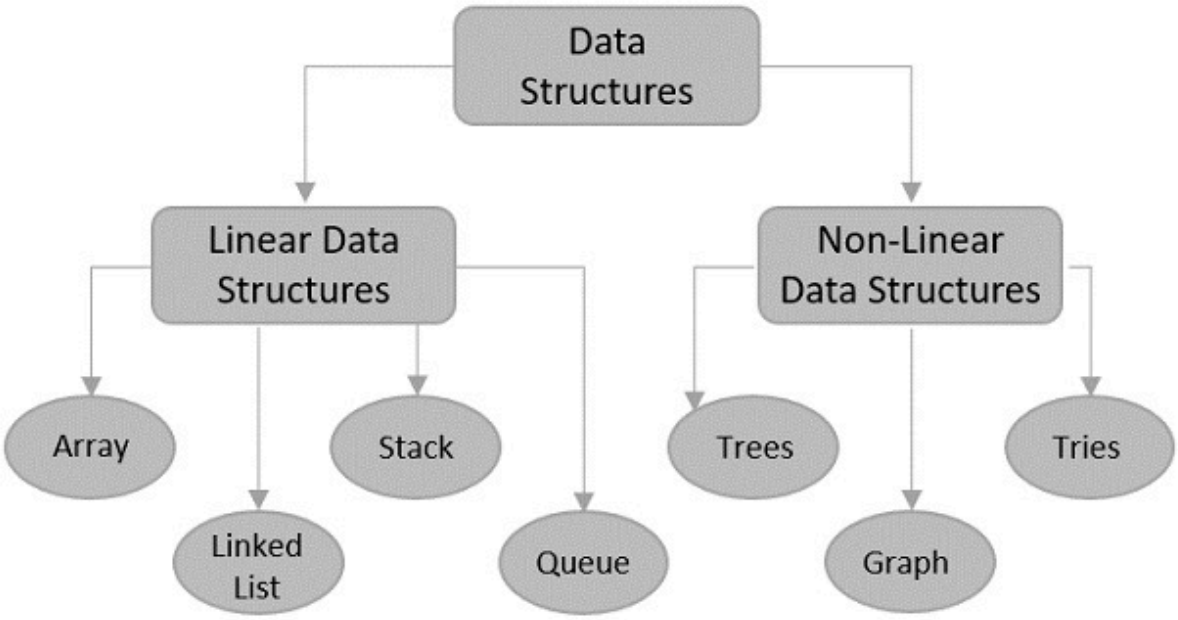
A **Data Structure** is a systematic way of organizing, storing, and managing data in memory so that it can be accessed and modified efficiently.

It enables:

- Efficient data access
- Better memory utilization
- Improved performance of algorithms

Types of Data Structures (with Neat Diagram & Examples) (5 Marks)

Data Structures are broadly classified into:



A) Primitive Data Structures

These are basic data types directly supported by the programming language.

Examples:

- `int`
- `float`
- `char`
- `double`

✔ Store single values
✔ Fixed memory size

B) Non-Primitive Data Structures

These are complex data structures derived from primitive types.

1. Linear Data Structures

Elements are arranged sequentially.

Examples:

- Array
- Linked List
- Stack
- Queue

✔ Traversal is linear
✔ Each element has a unique predecessor and successor (except first and last)

2. Non-Linear Data Structures

Elements are arranged hierarchically or interconnected.

Examples:

- Tree
- Graph

✔ One-to-many or many-to-many relationships
✔ Used to represent hierarchical or network data

Primitive Operations on Data Structures (3 Marks)

Primitive operations are basic operations that can be performed on any data structure.

1. **Traversal**
    - Visiting each element exactly once.
2. **Insertion**
    - Adding a new element at a specified position.
3. **Deletion**
    - Removing an element from the data structure.
4. **Searching**
    - Finding the location of an element.
5. **Sorting**
    - Arranging elements in ascending or descending order.
6. **Merging**
    - Combining two data structures into one.

1 b) Define structure and union? Explain how they are different from each other, with suitable example (5 Marks)

Definition of Structure (1 Mark)

A **Structure** in C is a user-defined data type that groups variables of different data types under a single name.
Each member gets **separate memory allocation**.

Syntax Example:

```
struct Student {
    int roll;
    float marks;
    char grade;
};
```

Definition of Union (1 Mark)

A **Union** in C is a user-defined data type similar to structure, but **all members share the same memory location**.

Syntax Example:

```
union Student {
    int roll;
    float marks;
    char grade;
};
```

Difference Between Structure and Union (2 Marks)

Feature	**Structure**	**Union**
Memory Allocation	Separate memory for each member	Shared memory among all members
Size	Sum of sizes of all members	Size of largest member
Simultaneous Access	All members can store values at the same time	Only one member can store value at a time
Memory Efficiency	Less efficient	More efficient

Suitable Example with Explanation (1 Mark)

```
#include <stdio.h>

struct Test1 {
    int a;
    float b;
};

union Test2 {
    int a;
    float b;
};
```

If:

- `int` = 4 bytes

- `float` = 4 bytes

Then:

- Size of `struct Test1` = 4 + 4 = **8 bytes**
- Size of `union Test2` = **4 bytes** (largest member)

✔ In structure → both a and b can hold values simultaneously.
✔ In union → assigning value to b overwrites value of a.

1 c) What do you mean by pattern matching? Outline Kruth, pattern matching algorithm. (5 Marks)

Definition of Pattern Matching (1 Mark)

Pattern Matching is the process of finding the occurrence(s) of a given **pattern (substring)** within a larger **text string**.

Applications:

- Text editors (Find option)
- Compiler design
- DNA sequence analysis
- Search engines

Knuth Pattern Matching Algorithm (KMP Algorithm) – Outline (4 Marks)

The **Knuth–Morris–Pratt (KMP)** algorithm is an efficient string matching algorithm developed by:

- Donald Knuth
- Vaughan Pratt
- James H. Morris

It improves over the naive string matching approach by **avoiding unnecessary comparisons**.

Key Idea

Instead of rechecking previously matched characters, KMP uses a preprocessing table called:

LPS (Longest Proper Prefix which is also Suffix)

This helps in shifting the pattern intelligently.

Steps of KMP Algorithm

Step 1: Preprocessing Phase

- Construct the **LPS array** for the pattern.
- LPS[i] stores the length of the longest proper prefix which is also suffix for pattern[0…i].

Step 2: Searching Phase

- Compare characters of text and pattern.
- If characters match → move forward.
- If mismatch occurs → use LPS value to shift the pattern.
- No need to recompare matched characters.

Simple Example

Text: ABABDABACDABABCABAB
Pattern: ABABCABAB

KMP uses LPS table to skip unnecessary comparisons and finds the pattern efficiently.

2 a) Define stack. Give the implementation of push () POP () and Display () functions by considering its empty and full conditions. (7 Marks)

Definition of Stack (2 Marks)

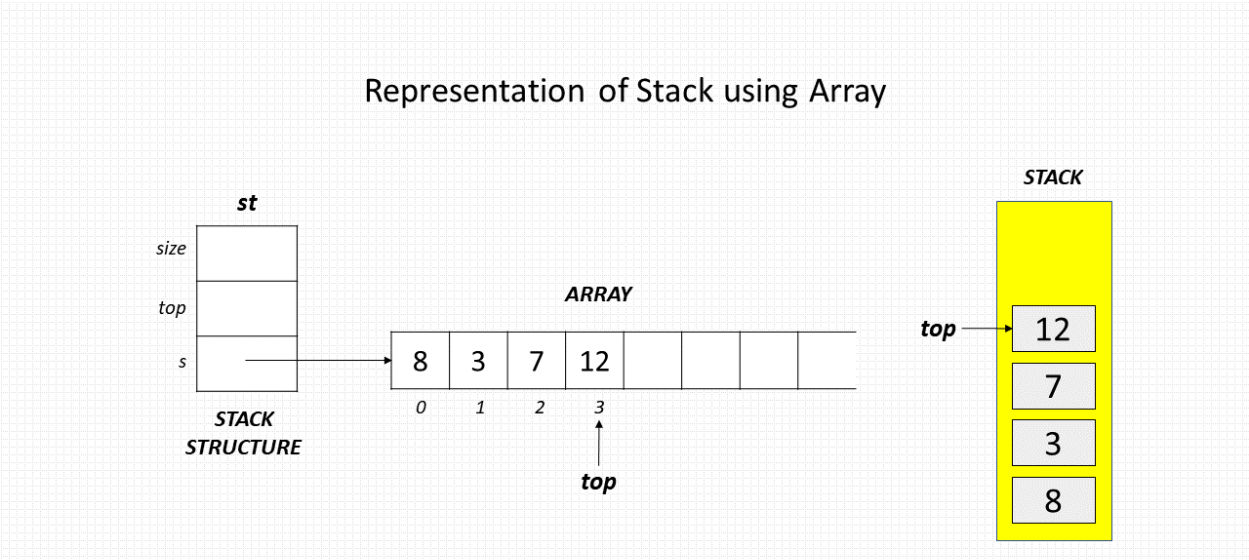
A **Stack** is a linear data structure in which insertion and deletion are performed only at **one end**, called the **TOP**.

Stack follows the principle:

LIFO (Last In First Out)

Example: Stack of plates – the last plate placed is the first one removed.

Stack Representation (Array Implementation) (1 Mark)



- Stack is implemented using an **array**
- `top` variable keeps track of the topmost element
- Initially, `top = -1`

Stack Operations Implementation (4 Marks)

Assumptions:

- `MAX = 5`
- Stack is implemented using array

Declaration

```
#include <stdio.h>
#define MAX 5

int stack[MAX];
int top = -1;
```

PUSH Operation (Insertion)

```
void push(int value)
{
    if (top == MAX - 1)
    {
        printf("Stack Overflow\n");
    }
    else
    {
        top++;
        stack[top] = value;
        printf("Inserted: %d\n", value);
    }
}
```

✔ **Overflow Condition:** `top == MAX - 1`

POP Operation (Deletion)

```
void pop()
{
    if (top == -1)
    {
        printf("Stack Underflow\n");
    }
    else
    {
        printf("Deleted: %d\n", stack[top]);
        top--;
    }
}
```

✔ **Underflow Condition:** `top == -1`

DISPLAY Operation

```
void display()
{
    if (top == -1)
    {
        printf("Stack is Empty\n");
    }
    else
```

```
    {
        for (int i = top; i >= 0; i--)
        {
            printf("%d\n", stack[i]);
        }
    }
}
```

✔ Displays elements from **top to bottom**

2 b) Write an algorithm to evaluate a postfix expression and apply the same for the given postfix expression 6, 2, 13, -4, 2, x, + (7 Marks)

Algorithm to Evaluate Postfix Expression (4 Marks)

Postfix expression is evaluated using a **Stack**.

Algorithm: Evaluate_Postfix(expression)

1. Create an empty stack S.
2. Scan the postfix expression from **left to right**.
3. For each symbol:
    - If it is an **operand**, push it onto the stack.
    - If it is an **operator**:
        - Pop operand2 from stack.
        - Pop operand1 from stack.
        - Compute:
          `result = operand1 operator operand2`
        - Push the result back onto the stack.
4. After scanning the entire expression:
    - The final value in the stack is the result.
5. Stop.

Evaluation of Given Expression (3 Marks)

Symbol	Operation	Stack
6	Push	6
2	Push	6, 2
+	6 + 2 = 8	8
13	Push	8, 13
-4	Push	8, 13, -4
2	Push	8, 13, -4, 2
×	-4 × 2 = -8	8, 13, -8
+	13 + (-8) = 5	8, 5
+	8 + 5 = 13	13

2 c) Write the postfix form of the following using stack,
(i) A*(B*C+D*E)+F
(ii) (A+(B*C)/(D-E)) (6 Marks)

(i) A*(B*C* + *D*E) + F (3 Marks)

Expression:

`A*(B*C + D*E) + F`

Scan	Stack	Postfix
A		A
*	*	A
(	* (	A
B	* (	AB
*	* (*	AB
C	* (*	ABC
+	* (+	ABC*
D	* (+	ABC*D

*	* (+ *	ABC*D
E	* (+ *	ABC*DE
)	*	ABC*DE*+
+	+	ABC*DE*+*
F	+	ABC*DE*+*F
End		ABC*DE*+*F+

Postfix Form (i):

ABC∗DE∗+∗F+

(ii) (A + (B*C)/(D - E)) (3 Marks)

Expression:

`(A + (B*C)/(D - E))`

Scan	**Stack**	**Postfix**
(	(	
A	(	A
+	(+	A
(	(+ (	A
B	(+ (	AB
*	(+ (*	AB
C	(+ (*	ABC
)	(+	ABC*
/	(+ /	ABC*
(	(+ / (	ABC*
D	(+ / (	ABC*D
-	(+ / (-	ABC*D
E	(+ / (-	ABC*DE
)	(+ /	ABC*DE-

)		ABC*DE-/+
End		ABC*DE-/+

Postfix Form (ii):

ABC∗DE−/+\

3 a) What are the disadvantages of ordinary queue? Discuss the implementation of circular queue. (8 Marks)

Disadvantages of Ordinary Queue (Linear Queue) (3 Marks)

A **Linear Queue** follows **FIFO (First In First Out)** principle.
It is usually implemented using an **array with two pointers: FRONT and REAR**.

Main Disadvantages:

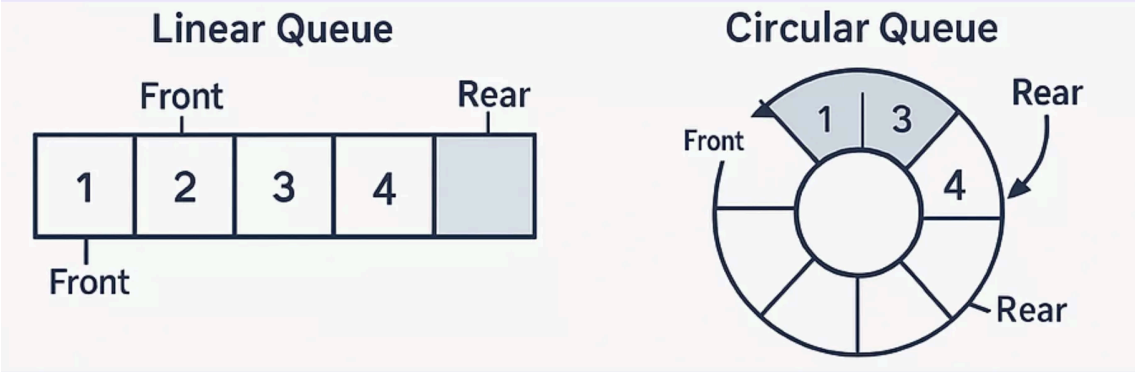
1. **Wastage of Memory**
    - After deletion, unused spaces at the beginning cannot be reused.
    - Even if there is free space, insertion is not possible when `REAR = MAX - 1`.
2. **False Overflow**
    - Condition: `REAR == MAX - 1`
    - Even though there are empty spaces at the front, queue is treated as full.
3. **Inefficient Memory Utilization**
    - Memory is not optimally used.
    - Requires shifting elements (costly operation) to reuse space.

Circular Queue (Concept) (2 Marks)

A **Circular Queue** overcomes the limitation of linear queue by treating the array as circular.

The last position is connected to the first position.

- After reaching the end, `REAR` moves to index `0`
- Efficient memory utilization



Implementation of Circular Queue (3 Marks)

Declaration

```
#define MAX 5
int queue[MAX];
int front = -1, rear = -1;
```

Insertion (Enqueue)

Full Condition:

```
(front == 0 && rear == MAX-1)
OR
(front == rear + 1)
```

```
void enqueue(int value)
{
    if ((front == 0 && rear == MAX-1) || (front == rear + 1))
        printf("Queue is Full\n");

    else
    {
        if (front == -1)
            front = rear = 0;
        else if (rear == MAX-1)
            rear = 0;
        else
            rear++;

        queue[rear] = value;
```

```
    }
}
```

Deletion (Dequeue)

Empty Condition:

```
front == -1
```

```
void dequeue()
{
    if (front == -1)
        printf("Queue is Empty\n");

    else
    {
        printf("Deleted: %d\n", queue[front]);

        if (front == rear)
            front = rear = -1;
        else if (front == MAX-1)
            front = 0;
        else
            front++;
    }
}
```

3 b) Write a note on multiple stacks and priority queue (6 Marks)

Multiple Stacks (3 Marks)

Definition

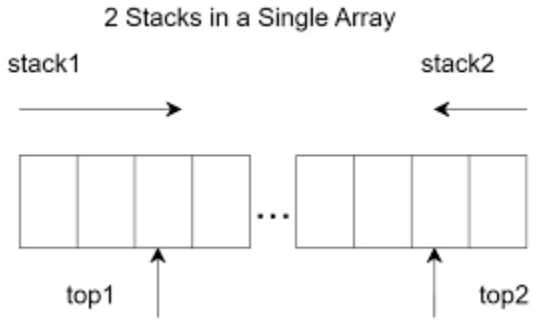
Multiple stacks refer to the implementation of **two or more stacks using a single array**.

Instead of allocating separate arrays for each stack, memory is shared efficiently.

Types of Multiple Stack Implementation

(i) Two Stacks in One Array

- Stack1 grows from **left to right**
- Stack2 grows from **right to left**
- Both share the same array



Conditions:
Overflow Condition:

```
top1 + 1 == top2
```

- **Advantages:**
    - Better memory utilization
    - No wastage of unused space

(ii) More than Two Stacks

Implemented using:

- Linked list approach
- Dynamic memory allocation
- Auxiliary arrays (for next index tracking)

Priority Queue (3 Marks)

Definition

A **Priority Queue** is a special type of queue in which each element is associated with a **priority value**.

- Element with **higher priority** is served first.
- If priorities are equal → follows FIFO.

Types of Priority Queue

1. **Ascending Priority Queue**
    - Smaller value → Higher priority
    - Deletion removes smallest element
2. **Descending Priority Queue**
    - Larger value → Higher priority
    - Deletion removes largest element

Implementation Methods

- Using Array
- Using Linked List
- Using Heap (Efficient – O(log n))

Applications

- CPU scheduling
- Dijkstra's Algorithm
- Printer task scheduling
- Event-driven simulation

3 c) Define Queue. Discuss how to represent Queue using dynamic arrays. (6 Marks)

Definition of Queue (2 Marks)

A **Queue** is a linear data structure that follows the principle:

FIFO (First In First Out)

- Insertion is done at the **REAR**
- Deletion is done at the **FRONT**

Example: Ticket counter line – first person entering is the first person served.

Representation of Queue Using Dynamic Arrays (4 Marks)

Unlike static arrays (fixed size), **dynamic arrays** are allocated at runtime using memory allocation functions such as `malloc()` in C.

This allows flexible memory usage.

Structure Representation

```
#include <stdio.h>
#include <stdlib.h>

struct Queue {
    int *arr;
    int front, rear;
    int capacity;
};
```

Dynamic Memory Allocation

```
struct Queue* createQueue(int size)
{
    struct Queue* q = (struct Queue*)malloc(sizeof(struct Queue));
    q->capacity = size;
    q->front = 0;
    q->rear = -1;
    q->arr = (int*)malloc(size * sizeof(int));
    return q;
}
```

Enqueue Operation

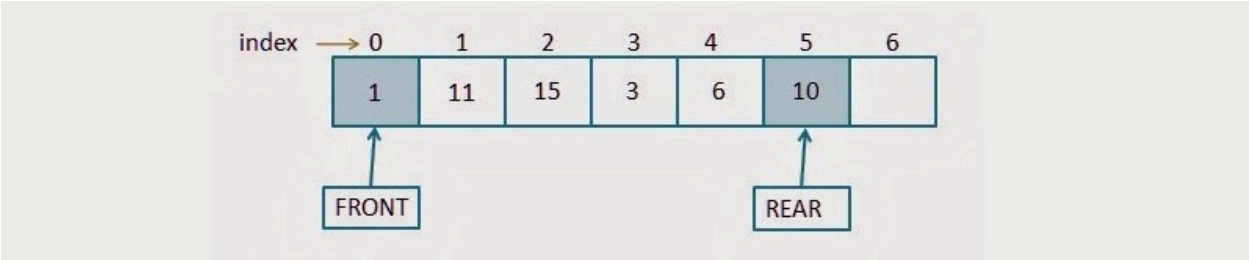
```
void enqueue(struct Queue* q, int value)
{
    if (q->rear == q->capacity - 1)
        printf("Queue Overflow\n");
    else
```

```
    {
        q->rear++;
        q->arr[q->rear] = value;
    }
}
```

Dequeue Operation

```
void dequeue(struct Queue* q)
{
    if (q->front > q->rear)
        printf("Queue Underflow\n");
    else
    {
        printf("Deleted: %d\n", q->arr[q->front]);
        q->front++;
    }
}
```

Queue Representation Diagram



4 a) What are Linked list? Explain the different types of Linked List with neat diagram (4 Marks)

Definition of Linked List (1 Mark)

A **Linked List** is a linear data structure in which elements (called **nodes**) are stored in **non-contiguous memory locations** and connected using **pointers (links)**.

Each node contains:

- **Data**
- **Link (Pointer) to next node**

Unlike arrays, linked lists do not require contiguous memory.

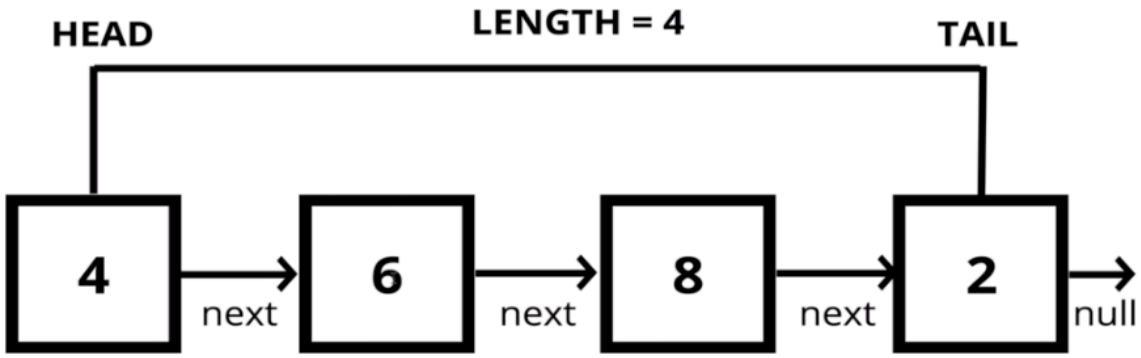
Types of Linked Lists (3 Marks)

Singly Linked List (SLL)

Each node contains:

- Data
- Pointer to next node

Last node points to **NULL**.



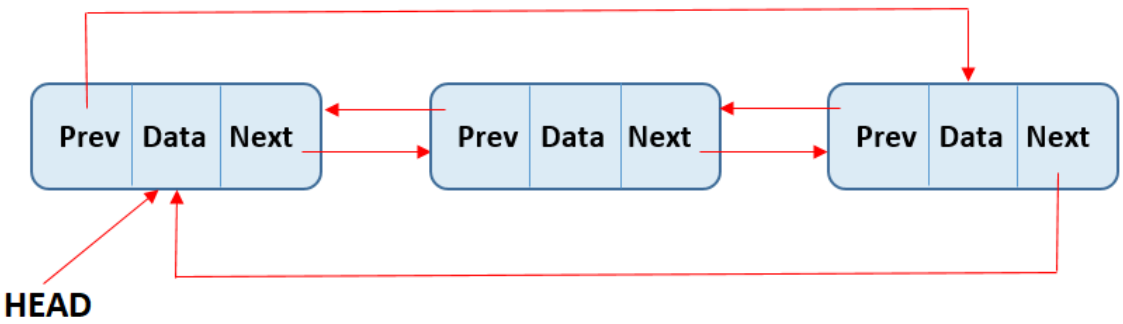
Features:

- Traversal in one direction only
- Simple implementation

Doubly Linked List (DLL)

Each node contains:

- Data
- Pointer to next node
- Pointer to previous node

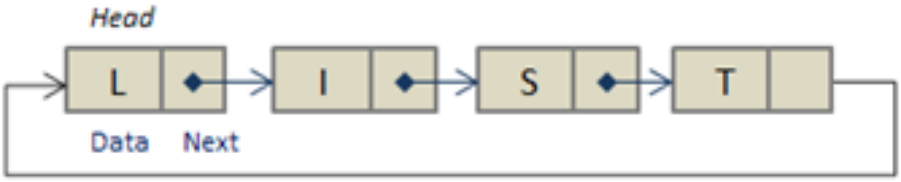


Features:

- Traversal in both forward and backward directions
- Requires extra memory for previous pointer

Circular Linked List (CLL)

Last node points back to the **first node** instead of NULL.



Features:

- No NULL at end
- Useful in round-robin scheduling

4 b) Give the structure definition for Singly Linked List (SSL). Write a C function to, (8 Marks)
(i) Insert an element at the end of SSL.
(ii) Delete at node at the end of SSL.

Structure Definition of Singly Linked List (2 Marks)

In a **Singly Linked List (SLL)**, each node contains:

- Data
- Pointer to the next node

Structure Definition

```
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int data;
    struct node *next;
};

struct node *head = NULL;
```

Insert an Element at the End of SLL (3 Marks)

Algorithm:

1. Create a new node.
2. Assign data to the node.
3. If list is empty → new node becomes head.
4. Otherwise → traverse till last node.
5. Set last node's `next` to new node.

C Function

```
void insert_end(int value)
{
    struct node *newnode, *temp;

    newnode = (struct node*)malloc(sizeof(struct node));
    newnode->data = value;
    newnode->next = NULL;
```

```
    if (head == NULL)
    {
        head = newnode;
    }
    else
    {
        temp = head;
        while (temp->next != NULL)
        {
            temp = temp->next;
        }
        temp->next = newnode;
    }
}
```

Delete a Node at the End of SLL (3 Marks)

Algorithm:

1. If list is empty → display message.
2. If only one node → free it and set head to NULL.
3. Otherwise → traverse to second last node.
4. Free the last node.
5. Set second last node's `next = NULL`.

C Function

```
void delete_end()
{
    struct node *temp, *prev;

    if (head == NULL)
    {
        printf("List is empty\n");
    }
    else if (head->next == NULL)
    {
```

```
        free(head);
        head = NULL;
    }
    else
    {
        temp = head;
        while (temp->next != NULL)
        {
            prev = temp;
            temp = temp->next;
        }
        prev->next = NULL;
        free(temp);
    }
}
```

4 c) Write a C function to add two polynomials show the Linked List representation of below two polynomials (8 Marks)

p(x)=3x^14 +2x^2+1

q(x)=8x^14+5x^5+3x^2+2

Linked List Representation of Polynomial (2 Marks)

Each node of the linked list contains:

- **Coefficient (coeff)**
- **Exponent (exp)**
- **Pointer to next node**

Structure Definition

```
struct node
{
    int coeff;
    int exp;
    struct node *next;
};
```

C Function to Add Two Polynomials (4 Marks)

Logic:

1. Compare exponents of both lists.
2. If exponents are equal → add coefficients.
3. If exponent of first > second → copy first node.
4. Else → copy second node.
5. Continue until both lists are exhausted.

C Function

```
struct node* addPoly(struct node* p, struct node* q)
{
    struct node *result = NULL, *temp = NULL, *newnode;

    while (p != NULL && q != NULL)
    {
        newnode = (struct node*)malloc(sizeof(struct node));
        newnode->next = NULL;

        if (p->exp == q->exp)
        {
            newnode->coeff = p->coeff + q->coeff;
            newnode->exp = p->exp;
            p = p->next;
            q = q->next;
        }
        else if (p->exp > q->exp)
        {
            newnode->coeff = p->coeff;
            newnode->exp = p->exp;
            p = p->next;
        }
        else
        {
            newnode->coeff = q->coeff;
            newnode->exp = q->exp;
            q = q->next;
        }

        if (result == NULL)
        {
```

```
            result = newnode;
            temp = newnode;
        }
        else
        {
            temp->next = newnode;
            temp = newnode;
        }
    }

    return result;
}
```

Resultant Polynomial (2 Marks)

Now add the given polynomials:

p(x)=3x^14+2x^2+1
q(x)=8x^14+5x^5+3x^2+2
Combine Like Terms:

- 3x^{14} + 8x^{14} = 11x^{14}
- 2x^2 + 3x^2 = 5x^2
- 1+2=3
- 5x^5

Final Polynomial:

11x^{14} + 5x^5 + 5x^2 + 3

Linked List Representation:

`Head → (11,14) → (5,5) → (5,2) → (3,0) → NULL`

5 a) Write a C-function for the following operation on doubly Linked List (DLL) (8 Marks)
(i) Addition of a DLL nodes
(ii) Concatenation of two DLL

```
#include <stdio.h>
#include <stdlib.h>

struct node
{
   int data;
   struct node *prev;
```

```
    struct node *next;
};
typedef struct node NODE;
```

(i) Addition of a DLL nodes (4 Marks)

```
int sumDLL(NODE *head)
{
    NODE *temp = head;
    int sum = 0;

    while(temp != NULL)
    {
        sum = sum + temp->data;
        temp = temp->next;
    }

    return sum;
}
```

(ii) Concatenation of two DLL(4 Marks)

```
NODE* concatenateDLL(NODE *head1, NODE *head2)
{
    NODE *temp;

     if(head1 == NULL)
        return head2;

      if(head2 == NULL)
        return head1;

     temp = head1;
    while(temp->next != NULL)
        temp = temp->next;

    temp->next = head2;
    head2->prev = temp;

    return head1;
}
```

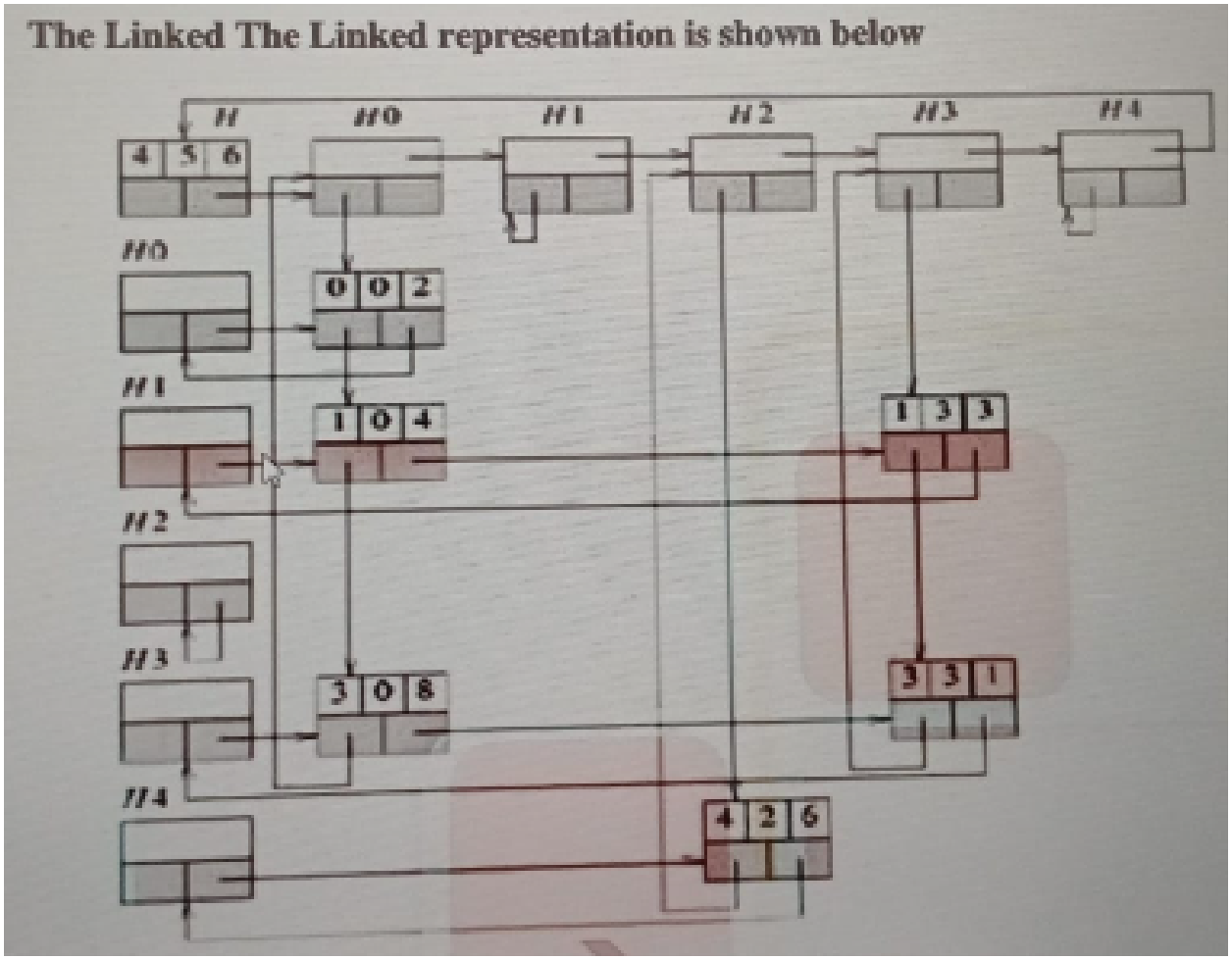
5 **b)** Write a C-function for the following operations on circular Linked List (8 Marks)
(i) Inserting at the front of a List **(4 Marks)**
(ii) Find the number of nodes in circular list **(4 Marks)**

```
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int data;
    struct node *next;
};
```

```
typedef struct node NODE;

NODE* insertFront(NODE *head, int item)
{
   NODE *newnode, *temp;

   newnode = (NODE*)malloc(sizeof(NODE));
   newnode->data = item;

   if(head == NULL)
   {
      newnode->next = newnode;
      head = newnode;
      return head;
   }

   temp = head;
   while(temp->next != head)
      temp = temp->next;

   newnode->next = head;
   temp->next = newnode;
   head = newnode;

   return head;
}

int countNodes(NODE *head)
{
   int count = 0;
   NODE *temp;

   if(head == NULL)
      return 0;

   temp = head;

   do
   {
      count++;
      temp = temp->next;
   }
   while(temp != head);

   return count;
}
```

5 **c)** Represent the given Sparse matrix using linked list representation (4 Marks)

$$A = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 7 & 0 & 1 & 1 \\ 0 & 0 & 6 & 0 \end{bmatrix}$$



6 a) Explain the different types binary tree representation with example. (8 Marks)

A Binary Tree can be represented in memory mainly in two ways:

1. Sequential Representation (Array Representation) **(4 Marks)**
2. Linked Representation (Pointer Representation) **(4 Marks)**

Sequential Representation (Array Representation)

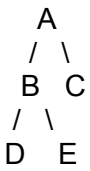
In sequential representation, the nodes of the binary tree are stored in an array. The root is stored at index 1, the left child of a node at index *i* is stored at position 2i, the right child at 2i+1, and the parent at i/2.

This method is simple and does not require pointers, but it wastes memory if the tree is not complete, so it is mainly suitable for complete binary trees.

Linked Representation (Pointer Representation)

In linked representation, each node contains three parts: data, a pointer to the left child, and a pointer to the right child.

This method uses dynamic memory and efficiently represents any type of binary tree without memory wastage, but it requires extra memory for pointers and is slightly more complex to implement.

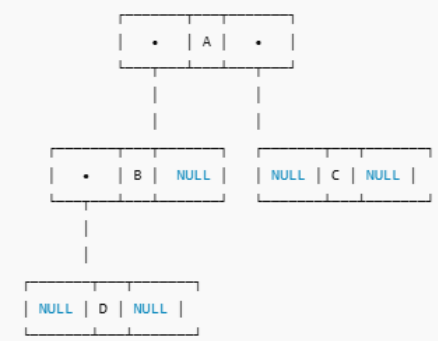


For the Given Graph:

Array Representation:

Index	**1**	**2**	**3**	**4**	**5**
Node	**A**	**B**	**C**	**D**	**E**

Linked Representation:



6 b) Define Threaded Binary tree. Discuss in threaded binary tree. (4 Marks)

A Threaded Binary Tree is a binary tree where null pointers in leaf nodes are replaced with "threads" to allow in-order traversal without the use of recursion or a stack.

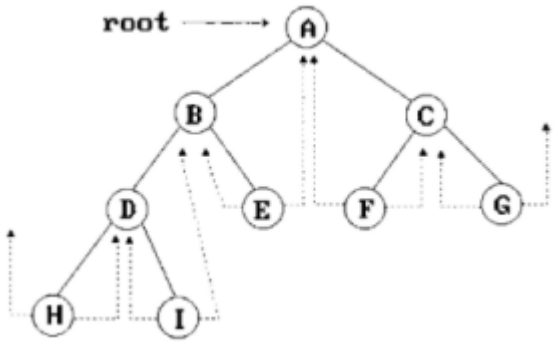
In a threaded binary tree, each node has an additional pointer called a thread, which points to either its in-order predecessor or successor. This allows us to efficiently traverse the tree without using recursion or a stack.

ALGORITHM: (4 Marks)

(1) If ptr → left_child is null, replace ptr → left_child with a pointer to the node that would be visited before ptr in an inorder traversal. That is, we replace the null link with a pointer to the inorder predecessor of ptr.

(2) If ptr → right_child is null, replace ptr → right_child with a pointer to the node that would be visited after ptr in an inorder traversal. That is, we replace the null link with a pointer to the inorder successor of ptr.

Threaded Binary tree for the given elements are: A, B, C, D, E, F, G, H, I.



6 c) Discuss inorder, preorder, postorder and level order traversal with suitable recursive function for each. (8 Marks)

INORDER: (2 Marks)

```
void inorder(struct Node *root)
{
    if (root == NULL)
        return;

    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}
```

PREORDER: (2 Marks)

```
void preorder(struct Node *root)
{
    if (root == NULL)
        return;

    printf("%d ", root->data);
    preorder(root->left);
    preorder(root->right);
}
```

POSTORDER: (2 Marks)

```
void postorder(struct Node *root)
{
    if (root == NULL)
        return;
```

```
    postorder(root->left);
    postorder(root->right);
    printf("%d ", root->data);
}
```

LEVEL ORDER: (2 Marks)

```
struct Node* queue[100];
int front = -1, rear = -1;

void enqueue(struct Node *temp)
{
    if (rear == 99)
        return;

    if (front == -1)
        front = 0;

    queue[++rear] = temp;
}

struct Node* dequeue()
{
    if (front == -1 || front > rear)
        return NULL;

    return queue[front++];
}

void levelorder(struct Node *root)
{
    struct Node *temp;

    if (root == NULL)
        return;

    enqueue(root);

    while (front <= rear)
    {
        temp = dequeue();
        printf("%d ", temp->data);

        if (temp->left != NULL)
            enqueue(temp->left);

        if (temp->right != NULL)
            enqueue(temp->right);
    }
}
```

7 **a)** Write a function to perform the following operations on Binary Search Tree (BST) (8 Marks)
(i) Inserting an element into BST
(ii) Recursive search of a BST

```
struct Node
{
   int data;
   struct Node *left;
   struct Node *right;
};
```

(i) Inserting an element into BST (4 Marks)

```
struct Node* insert(struct Node *root, int key)
{
   struct Node *newnode;

   if (root == NULL)
   {
      newnode = (struct Node*)malloc(sizeof(struct Node));
      newnode->data = key;
      newnode->left = NULL;
      newnode->right = NULL;
      return newnode;
   }

   if (key < root->data)
      root->left = insert(root->left, key);
   else if (key > root->data)
      root->right = insert(root->right, key);

   return root;
}
```

(ii) Recursive search of a BST (4 Marks)

```
struct Node* search(struct Node *root, int key)
{
   if (root == NULL || root->data == key)
      return root;

   if (key < root->data)
      return search(root->left, key);
   else
      return search(root->right, key);
}
```

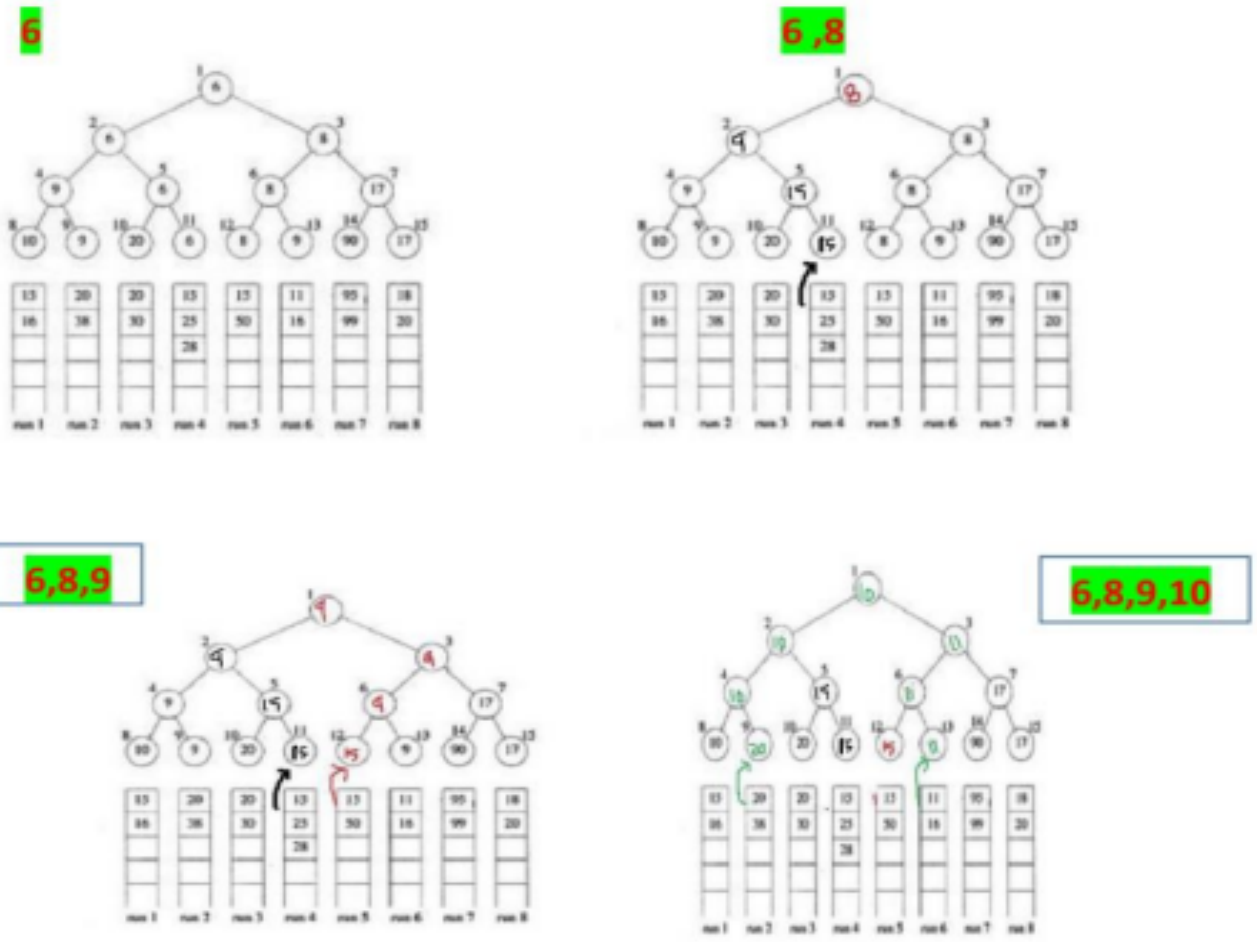
7 b) Discuss selection Trees with suitable example. (8 Marks)

SELECTION TREE (4 Marks)
Example (4 Marks)

This is also called as a tournament tree. This is such a tree data structure using which the winner (or loser) of a knock out tournament can be selected. There are two types of selection trees namely: winner tree and loser tree.

WINNER TREE

This is a complete binary tree in which each node represents the smaller of its two children. Thus, the root node represents the smallest node in the tree.



7 c) Explain transforming a forest into a binary tree with an example. (4 Marks)

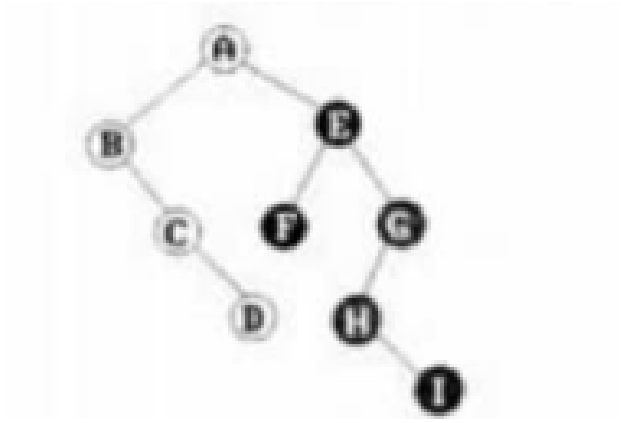
A forest is a collection of disjoint trees. It can be converted into a binary tree using the Left-Child Right-Sibling (LCRS) representation.

Rules:

1. The first child of a node → becomes its left child in the binary tree.
2. The next sibling of a node → becomes its right child in the binary tree.

3. Apply this to all nodes.
4. If the forest has multiple trees, connect the roots as right siblings.

Converting below forest into Binary Tree:



8 a) Define graph. Show the adjacency matrix and adjacency list representation of the graph given below. (6 Marks)

A Graph is a non-linear data structure consisting of a set of vertices (nodes) and a set of edges that connect pairs of vertices.

Mathematically, a graph is represented as:

$$G=(V,E)$$

Where:

- **V** = Set of vertices
- **E** = Set of edges connecting the vertices

Adjacency Matrix (3 Marks)

```
      1  2  3  4
1  →  0  1  1  1
2  →  1  0  1  0
3  →  1  1  0  1
4  →  1  0  1  0
```

Adjacency List (3 Marks)

```
1 → 2 → 3 → 4
2 → 1 → 3
3 → 1 → 2 → 4
4 → 1 → 3
```

8 b) Define the following terminologies with example: (7 Marks)
i) vertex
ii) self loop
iii) weighted graph
iv) parallel edges

i)Vertex: (2 Marks)

A vertex (plural: vertices) is a fundamental unit of a graph. It represents an object, point, or entity in a graph. Example: In a social network graph, each person can be a vertex. If we have people A, B, and C, the vertices are:

```
Vertices = {A, B, C}
```

ii) Self Loop:(1 Mark)

A self-loop is an edge that connects a vertex to itself.

Edge: A → A

iii)Weighted Graph: (2 Marks)

A weighted graph has edges with values (weights) showing cost, distance, or time. Example: In a city map, roads have distances:

City1 —10km→ City2, City2 —15km→ City3

iv) Parallel Edges: (2 Marks)

Parallel edges are two or more edges connecting the same pair of vertices.

Edge1: City1 → City2

Edge2: City1 → City2

8 c) Explain in detail Elementary Graph Operations. (7 Marks)

Elementary graph operations are:

BFS(Breadth first search) **(3 Marks)**
DFS (Depth First Search) **(3 Marks)**
Spanning trees **(1 Mark)**

i) BFS(Breadth first search)

BFS is a traversal technique where we visit all neighbors first before going to the next level.

Working Principle:

Start from source vertex.
Visit all adjacent vertices.
Then visit neighbors of those vertices.
Uses Queue (FIFO).

ii) DFS (Depth First Search)

DFS is a traversal technique where we go as deep as possible in one direction before backtracking.

Working Principle:

Start from a source vertex.
Visit one of its unvisited neighbors.
Continue moving deeper.
If no unvisited neighbor is left, backtrack.
Uses Stack (or Recursion).

A — B — D

| |

C E

BFS for the Given Graph: A → B → C → D → E
DFS for the Given Graph: A → B → D → E → C

iii) Spanning Trees

A spanning tree is a subgraph of a connected graph that includes all vertices with no cycles and exactly V − 1 edges. It connects all nodes using the minimum number of edges.

9 a) What is collision? What are the methods to resolve collision? Explain linear probing with example (8 Marks)

What is Collision? (2 Marks)

In **Hashing**, a **collision** occurs when two or more keys are mapped to the **same hash table index** using a hash function.

Since the hash table size is limited, different keys may produce the same hash value.

Example:

If

h(key)=keymod 10

Then:

- h(25) = 5
- h(15) = 5

Both keys map to index 5 → **Collision occurs**

Methods to Resolve Collision (3 Marks)

There are two main techniques:

(i) Open Addressing

All elements are stored inside the hash table itself.

Methods:

- **Linear Probing**
- Quadratic Probing
- Double Hashing

(ii) Separate Chaining

- Each table index contains a linked list.
- Colliding elements are stored in that list.

3️⃣ Linear Probing (3 Marks)

Definition

Linear Probing is a collision resolution technique in which we search for the next available slot sequentially.

If a collision occurs at index $h(k)$, then we check:

$(h(k)+1) \bmod m$

If still full:

$(h(k)+2) \bmod m$

Continue until empty slot is found.

Formula

$hi(k)=(h(k)+i) \bmod m$

Where:

- m = table size
- i = 0,1,2,3,…

Example

Let:

- Hash function:
  $h(k)=k \bmod 10h(k) = k \mod 10h(k)=k \bmod 10$
- Table size = 10
- Insert keys: **23, 43, 13**

Step 1: Insert 23

h(23) = 3
Place at index 3

Step 2: Insert 43

h(43) = 3 → Collision
Check (3+1)=4 → Empty
Place at index 4

Step 3: Insert 13

h(13) = 3 → Collision
Index 4 → Occupied
Check index 5 → Empty
Place at index 5

Advantage

- Simple implementation

Disadvantage

- Causes **Primary Clustering**
  (Consecutive occupied slots form clusters)

9 b) Explain in details about static and dynamic hashing (6 Marks)

Static Hashing (3 Marks)

Definition

Static Hashing is a hashing technique in which the **hash table size is fixed** at the time of creation and does not change during program execution.

- Number of buckets remains constant.
- Hash function remains unchanged.

Working

Let:

$h(k)=k \bmod m$

Where:

- k = key
- m = fixed table size

Keys are inserted based on the hash function.
If collision occurs → resolved using:

- Linear probing
- Quadratic probing
- Separate chaining

Example

Table size = 5
Hash function:

$h(k) = k \bmod 5$

Insert keys: 12, 22, 32

All map to index 2 → Collision handled using probing/chaining.

Advantages

- Simple implementation
- Easy to understand

Disadvantages

- Table overflow when size exceeds limit
- Poor performance when load factor increases
- Requires rehashing if size must increase

Dynamic Hashing (3 Marks)

Definition

Dynamic Hashing is a hashing technique in which the **hash table size grows or shrinks dynamically** based on the number of records.

It avoids overflow and improves performance.

Types of Dynamic Hashing

1. **Extendible Hashing**
2. **Linear Hashing**

Extendible Hashing (Concept)

- Uses a directory of pointers to buckets.
- Hash value is represented in binary.
- When bucket overflows → bucket splits.
- Directory size may double.

Linear Hashing (Concept)

- Buckets split gradually.

- No directory required.
- Table size increases step-by-step.

Advantages

- Efficient memory utilization
- Reduces overflow
- Maintains good performance even with large data

Disadvantages

- More complex implementation
- Requires additional memory for directory (in extendible hashing)

9 c) Discuss Leftist Trees with an example. (6 Marks)

Definition of Leftist Tree (2 Marks)

A **Leftist Tree** (also called **Leftist Heap**) is a special type of **binary heap** used to efficiently perform **merge (union)** operations.

It satisfies two properties:

(i) Heap Property

- In a **Min Leftist Tree**, the parent node is smaller than its children.
- (Similar to Min Heap)

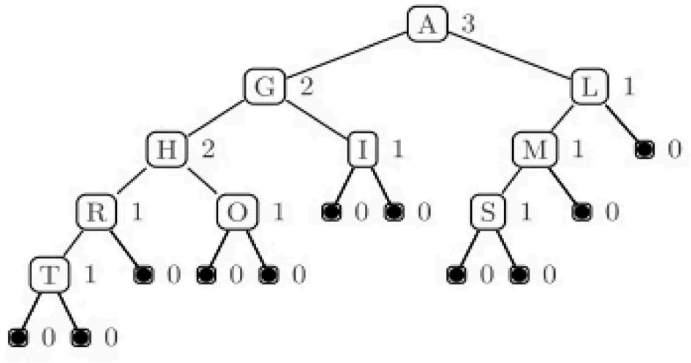
(ii) Leftist Property

For every node:

NPL(left)≥NPL(right)Where:

NPL (Null Path Length) = length of shortest path to a node with no two children. This ensures the tree is **left heavy**.

Structure of Leftist Tree (1 Mark)



Example of Leftist Tree (2 Marks)

Consider inserting elements:

10, 20, 30, 5, 15

After applying heap and leftist properties:

```
        5
        / \
      10   15
     /  \
   20   30
```

- Root = 5 (minimum element)
- Left subtree is heavier than right
- NPL(left) ≥ NPL(right)

Operations in Leftist Tree (1 Mark)

Main operations:

1. **Merge (Union)** – Most important operation
2. Insert (by merging single node tree)
3. DeleteMin (remove root and merge subtrees)

10 a) Explain different types of HASH functions with example (6 Marks)

Introduction to Hash Function (1 Mark)

A **Hash Function** is a function that transforms a given key into a valid **index (address)** in a hash table.

h(k)=Index in Hash Table

A good hash function should:

- Be simple to compute
- Distribute keys uniformly
- Minimize collisions

Types of Hash Functions (5 Marks)

1.Division (Remainder) Method

Formula:

$h(k)=k \bmod m$

Where:

- k = key
- m = table size (preferably prime)

Example:

Let m = 10

For key 27:

$h(27)=27 \bmod 10$

✔ **Simple and widely used**

✖ **May cause clustering if m is poorly chosen**

2. Multiplication Method

Formula:

$h(k)=\lfloor m(kA \bmod 1) \rfloor$

Where:

- A = constant (0 < A < 1)
- m = table size

Example:

Let:

- k = 50
- m = 10
- A = 0.618

50×0.618=30.9

Fractional part = 0.9

h(50)=10×0.9=9

✔ Works well for any table size

✖ Slightly complex

3. Mid-Square Method

Method:

1. Square the key.
2. Extract middle digits.

Example:

Key = 24

24^2 = 576

Middle digit = 7

So:

h(24)=7

✔ Good distribution

✖ Extra computation required

4. Folding Method

🔹 **Method:**

Divide key into parts and add them.

Example:

Key = 123456

Split: 123 + 456 = 579

If m = 100:

h(123456)=79

✔ **Useful for long keys**

✖ **May still produce collisions**

5. Digit Extraction Method

Method:

Select specific digits from the key.

Example:

Key = 987654

Select 2nd and 4th digits → 8 and 6

h(987654)=86

✔ **Simple**

✖ **Depends on digit pattern**

10 b) Discuss different types of rotations with suitable examples (6 Marks)

Out of syllabus

10 c) Define Red-Black Tree, Splay tree. Discuss the method to insert an element into Red-Black tree (8 Marks)

Out of syllabus