PVP 23

Code: 23CS3201, 23IT3201, 23AM3201, 23DS3201

I B.Tech - II Semester – Regular / Supplementary Examinations
JUNE 2026

DATA STRUCTURES

(Common for CSE, IT, AIML, DS)

Duration: 3 hours Max. Marks: 70

Note: 1. This question paper contains two Parts A and B.
2. Part-A contains 10 short answer questions. Each Question carries 2 Marks.
3. Part-B contains 5 essay questions with an internal choice from each unit. Each Question carries 10 marks.
4. All parts of Question paper must be answered in one place.

BL – Blooms Level CO – Course Outcome

PART – A

		BL	CO
1.a)	Identify the key idea behind Insertion Sort.	L2	CO1
1.b)	Compare linear and non linear data structure.	L2	CO1
1.c)	Write an algorithm for traversing a singly linked list.	L2	CO1
1.d)	List the advantage of linked list over array.	L2	CO1
1.e)	List the applications of stack.	L2	CO1
1.f)	Explain overflow condition in stack.	L2	CO1
1.g)	List the applications of queue.	L2	CO1
1.h)	What is the condition for queue underflow?	L2	CO1
1.i)	Define collision in hashing.	L2	CO1
1.j)	Explain complete binary tree with an example.	L2	CO1

PART – B

			BL	CO	Max. Marks
UNIT-I					
2	a)	Illustrate the Binary Search technique with an example.	L2	CO2	5M
	b)	Demonstrate the time and space complexity analysis.	L2	CO1	5M
OR					
3	a)	Apply selection Sort algorithm to sort these 22,37,45,9,10,67,2 numbers in ascending order.	L3	CO2	5M
	b)	What is Abstract Data Types (ADTs)? Determine the importance of Abstract Data Types.	L2	CO1	5M
UNIT-II					
4	a)	Write algorithms to perform following operations on a circular singly linked list. i) Delete at first ii) Delete at last	L3	CO3	5M
	b)	Write algorithms to perform following operations on singly linked list. i) Insert at first ii) Insert at last iii) Insert at given position	L3	CO3	5M
OR					
5	a)	Compare Arrays and Linked List insertion and deletion operation with an example.	L4	CO4	5M

	b)	Write algorithms to perform following operations on a doubly linked list. i) Delete at first ii) Delete at last iii) Delete at given position	L3	CO3	5M
UNIT-III					
6	a)	Evaluate the given postfix expression using stack **934*8+4/-**	L3	CO3	5M
	b)	Compare stack and queue.	L4	CO4	5M
OR					
7	a)	Describe how stacks are used to check balanced parentheses. Explain with an example.	L2	CO3	5M
	b)	Write pseudocode for the array implementation of stack operations.	L3	CO3	5M
UNIT-IV					
8	a)	Discuss operations on Circular Queue for the following operations with an algorithm. i) Enqueue operation ii) Dequeue operation	L3	CO3	5M
	b)	What is Linear Queue? Compare how linear queue and circular queue utilize memory.	L4	CO4	5M
OR					
9	a)	Develop an algorithm for implementing enqueue and dequeue operations of a	L3	CO3	5M

		queue using linked list by checking necessary conditions.			
	b)	Develop algorithm for implementation of Queue using array for enqueue and dequeue operations.	L3	CO3	5M
		UNIT-V			
10	a)	Write an algorithm for Level Order tree traversal using iteration and give an example.	L3	CO3	5M
	b)	Construct hash table for the following keys 50,700,76,85,92,73,10 using separate chaining where h(k)=k mod 7.	L3	CO3	5M
		OR			
11	a)	Develop an algorithm for implementing the linear probing collision resolution technique and give an example.	L3	CO3	5M
	b)	Construct a Binary Search Tree for the given preorder traversal sequence 22,12,8,20,30,25,40.	L3	CO3	5M

I B.Tech - II Semester – Regular / Supplementary Examinations-JUNE 2026

DATA STRUCTURES

(Common for CSE, IT, AIML, DS)

PART-A

1 a) Identify the key idea behind Insertion Sort.

About Insertion sort-2M

1. b) Compare Linear and non Linear data structure.

Any two differences.
Each difference – 1 M

1. c) Write an algorithm for traversing a singly linked list.

Algorithm-2M

1. d) List the advantages of linked list over array.

Any two advantages.
Each advantage– 1 M

1.e) List the applications of stack.

Any two applications.
Each application – 1 M

1.f) Explain overflow condition in stack.

Specifying the condition- 2M

1.g) List the applications of queue.

Any two applications.
Each application – 1 M

1.h) What is the condition for queue underflow?

Specifying the condition- 2M

1.i) Define collision in hashing.

Definition of hashing- 2M

1.j) **Explain complete binary tree with an example.**

Definition-1M
Example-1M

PART-B

UNIT-I

2 a) **Illustrate the Binary Search technique with an example. (5M)**

Explanation -3M
Example -2M

b) **Demonstrate the time and space complexity analysis. (5M)**

About Time complexity – 1.5M
About Space complexity-1.5M
Example-2M

OR

3 a) **Apply Selection Sort algorithm to sort these 22,37,45,9,10,67,2 numbers in ascending order. 5M**

About Selection Sort-1M
Solving example-4M

b) **What is Abstract Data Types (ADTs)? Determine the importance of Abstract Data Types. 5M**

About ADT-2M
Importance-3M

UNIT-II

4 a) **Write algorithm to perform following operations on a circular singly linked list.**
- i) **Delete at first**
- ii) **Delete at last** **5M**

About circular singly linked list-1M
Delete at first algorithm – 2M
Delete at last algorithm – 2M

b) **Write algorithm to perform following operations on singly linked list.**
- i) **Insert at first**
- ii) **Insert at last**
- iii) **Insert at given position** **5M**

About singly linked list- 1M

About Operations-4M

OR

5 a) **Compare Arrays and Linked List insertion and deletion operation with an example. 5M**

About Array and Linked list- 1M
Insertion operation-2M
Delete Operation-2M

b) **Write algorithm to perform following operations on a doubly linked list.**
   i) **Delete at first**
   ii) **Delete at last**
   iii) **Delete at given position** **5M**

About Doubly Linked list- 1M
About Operations-4M

UNIT-III

6 a) **Evaluate the given postfix expression using stack 934*8+4/-** **5M**
Procedure – 2M
Solving Example- 3M

b) **Compare stack and queue.** **5M**
Any 5 comparisons
Each one-1M

OR

7 a) **Describe how stacks are used to check balanced parentheses. Explain with an example.**
Procedure – 3M
Example- 2M

5M

b) **Write pseudocode for the array implementation of stack operations. 5M**
About Stack-2M
Explanation of Operations- 3M

UNIT-IV

8 a) **Discuss operations on Circular Queue for the following operations with an algorithm.**
   i) **Enqueue operation**
   ii) **Dequeue operation** **5M**

About Circular Queue-1M
Each Operations-2M

b) **What is Linear Queue? Compare how linear queue and circular queue utilize memory. 5M**
About Linear Queue-1M
About Circular Queue-1M

About memory utilization-3M

OR

9 a) **Develop an algorithm for implementing enqueue and dequeue operations of a queue using linked list by checking necessary conditions. 5M**

About Queue – 1M
Each Operation-2M

b) **Develop algorithm for implementation of Queue using array for enqueuer and dequeue operations. 5M**

About Queue – 1M
Each Operation-2M

UNIT-V

10 a) **Write an algorithm for Level Order tree traversal using iteration and give an example. 5M**

About Level Order traversal algorithm – 3M
Example-2M

b) **Construct hash table for the following keys 50,700,76,85,92,73,10 using separate chaining where h(k)=kmod7 5M**

Procedure-3M
Final hash table construction-2M

OR

11 a) **Develop an algorithm for implementing the linear probing collision resolution technique and give an example. 5M**

About algorithm – 3M
Example-2M

b) **Construct a Binary Search Tree for the given preorder traversal sequence 22,12,8,20,30,25,40 5M**

About Binary Search Tree-1M
Preorder traversal-1M
Solving Problem-3M

Code: 23CS3201, 23IT3201, 23AM3201, 23DS3201 PVP 23

I B.Tech - II Semester – Regular / Supplementary Examinations-JUNE 2026

DATA STRUCTURES

(Common for CSE, IT, AIML, DS)

PART-A

1 a) Identify the key idea behind Insertion Sort. 2M

The key idea behind Insertion Sort is:
Take one element at a time and insert it into its correct position within the already sorted part of the array.

1. b) Compare Linear and non Linear data structure. 2M

Linear Data Structure	Non-Linear Data Structure
Elements are arranged linearly or sequentially, one after another.	Elements are arranged hierarchically or as a network.
Comparatively simple to implement.	Comparatively complex to implement.
Elements exist in a single level.	Elements may exist at multiple levels.
Examples: Array, Linked List, Stack, Queue	Examples: Tree, Graph

1. c) Write an algorithm for traversing a singly linked list. 2M

```
ptr=head
while(ptr!=NULL)
  print ptr->data
  ptr=ptr->next
```

1. d) List the advantages of linked list over array. 2M

A linked list is dynamic. It can grow or shrink during program execution.
Elements can be inserted or deleted without shifting other elements.
Linked lists can efficiently implement stacks, queues.
Linked lists can be easily divided or combined by changing a few links.

1.e) List the applications of stack. 2M

- Infix to postfix conversion.
- Postfix Evaluation
- Balancing Parenthesis.

1.f) Explain overflow condition in stack. 2M

The overflow condition in stack is top==SIZE-1.

1.g) List the applications of queue. 2M

The applications of queue are:

- CPU Scheduling
- Printer Spooling
- Breadth-First Search (BFS)
- Customer Service Systems

1.h) What is the condition for queue underflow? 2M

The condition for queue underflow is:

front==-1 (OR)
rear==-1 (OR)
front==-1 && rear==-1

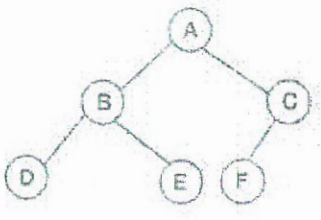
1.i) Define collision in hashing. 2M

A collision in hashing occurs when two or more different keys produce the same hash value/hash code.

1.j) Explain complete binary tree with an example. 2M

A **complete binary tree** is a binary tree in which:

- All levels are completely filled except possibly the last level.
- The nodes in the last level are filled **from left to right**.



PART-B

UNIT-I

2 a) **Illustrate the Binary Search technique with an example.** **5M**

Binary search is a fast way to find a value in a **sorted** array by repeatedly cutting the search space in half.

Compare the target to the middle element.

- If they match, you're done.
- If the target is smaller, discard the right half and search the left half.
- If the target is larger, discard the left half and search the right half.

Repeat until find it or run out of elements.

Example.

Array: [4, 9, 15, 22, 29, 37, 45, 52

Search for **45**

- **Step 1:** low = 0, high = 7, so mid = 3 → value **22**. Since 22 < 45, the target must be to the right — everything from index 0 to 3 is discarded.
- **Step 2:** low = 4, high = 7, so mid = 5 → value **37**. Still less than 45, discard again.
- **Step 3:** low = 6, high = 7, so mid = 6 → value **45**. Match found.

Each comparison eliminates half of the remaining elements, so the search takes roughly $\log_2(n)$ steps instead of scanning all n elements one by one.

For 8 elements that's just 3 comparisons; for a million elements it's only about 20. The one requirement is that the array must already be sorted — binary search doesn't work otherwise.

b) **Demonstrate the time and space complexity analysis.** **5M**

Time complexity measures the amount of **time or number of operations** an algorithm requires as the input size n increases.

Space complexity measures the amount of **extra memory** required by an algorithm as the input size increases.

Example

Consider the following algorithm:

```
Algorithm: Sum(A, n)
1. sum ← 0
2. for i ← 0 to n − 1
3.    sum ← sum + A[i]
4. return sum
```

Time Complexity

The loop executes n times.

T(n)=n

Therefore:

O(n)

So, the algorithm has **linear time complexity**.

Space Complexity

The algorithm uses only two extra variables: sum and i. The number of variables does not depend on n.

Therefore:

O(1)

So, the algorithm has **constant space complexity**.

For this algorithm, **Time Complexity = O(n)** and **Space Complexity = O(1)**.

OR

3 a) **Apply Selection Sort algorithm to sort these 22,37,45,9,10,67,2 numbers in ascending order.** **5M**

Selection sort works by repeatedly finding the minimum value in the unsorted portion of the array and swapping it to the front of that portion. Here's the trace through, step by step:

Given array:
22, 37, 45, 9, 10, 67, 2

Pass 1
Smallest element = **2**
Swap **2** with **22**:
2, 37, 45, 9, 10, 67, 22

Pass 2
Consider the remaining elements:
37, 45, 9, 10, 67, 22
Smallest element = **9**
Swap **9** with **37**:
2, 9, 45, 37, 10, 67, 22

Pass 3
Remaining elements:
45, 37, 10, 67, 22
Smallest element = **10**
Swap **10** with **45**:
2, 9, 10, 37, 45, 67, 22

Pass 4
Remaining elements:
37, 45, 67, 22
Smallest element = **22**
Swap **22** with **37**:
2, 9, 10, 22, 45, 67, 37

Pass 5
Remaining elements:
45, 67, 37
Smallest element = **37**
Swap **37** with **45**:
2, 9, 10, 22, 37, 67, 45

Pass 6
Remaining elements:
67, 45
Smallest element = **45**
Swap **45** with **67**:
2, 9, 10, 22, 37, 45, 67

Final Answer

2, 9, 10, 22, 37, 45, 67

b) **What is Abstract Data Types (ADTs)? Determine the importance of Abstract Data Types.** **5M**

An **Abstract Data Type (ADT)** is a logical or mathematical model of a data structure that defines:

- **What data is stored**
- **What operations can be performed on the data**

An ADT does **not specify how the data and operations are implemented internally**.

ADT focuses on "what to do" rather than "how to do it."

Example: Stack ADT

A stack follows the **LIFO (Last-In, First-Out)** principle.

The Stack ADT defines operations such as:

- push(x) – Insert an element
- pop() – Remove the top element
- peek() – View the top element
- isEmpty() – Check whether the stack is empty

The stack can be implemented using an **array** or a **linked list**. The user of the stack does not need to know the internal implementation.

Importance of Abstract Data Types

- **Data Abstraction** – ADTs hide unnecessary implementation details and show only essential operations.
- **Implementation Independence** – The internal implementation can be changed without affecting the user.
- **Code Reusability** – The same ADT can be reused in different programs.
- **Improved Security** – Direct access to internal data can be restricted.

UNIT-II

4 a) **Write algorithm to perform following operations on a circular singly linked list.** **5M**

i) **Delete at first**
ii) **Delete at last**

A circular singly linked list has its last node pointing back to the first node instead of NULL. The cleanest way to manage it is to keep a single pointer called last (pointing to the last node) — since last->next always gives you the first node, you don't need a separate head pointer.

Node structure:

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

i) Delete at first

```
Algorithm Delete_First(last)
1. If last == NULL
      print "List is empty"
      return
2. If last->next == last       // only one node in the list
      temp = last
      last = NULL
      free(temp)
      return
3. first = last->next          // current first node
4. last->next = first->next    // last now points to the new first
5. free(first)
6. return
```

The node right after last is always the first node. Deleting it just means re-linking last->next to skip over it — no traversal needed.

ii) Delete at last

```
Algorithm Delete_Last(last)
1. If last == NULL
      print "List is empty"
      return
2. If last->next == last       // only one node in the list
      temp = last
      last = NULL
```

```
    free(temp)
    return
3. p = last->next            // start at first node
4. While p->next != last     // walk until p is just before last
    p = p->next
5. secondLast = p
6. secondLast->next = last->next  // secondLast now points to first
7. free(last)
8. last = secondLast         // secondLast becomes the new last
9. return
```

b) Write algorithm to perform following operations on singly linked list. **5M**

- **i) Insert at first**
- **ii) Insert at last**
- **iii) Insert at given position**

A singly linked list has a head pointer to the first node, and each node's next points to the following node (or NULL if it's the last).

Node structure:

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

i) Insert at first

```
Algorithm Insert_First(head, value)
1. newNode = create a new Node
2. newNode->data = value
3. newNode->next = head
4. head = newNode
5. return head
```

The new node simply takes over as head, pointing to whatever used to be first.

ii) Insert at last

```
Algorithm Insert_Last(head, value)
1. newNode = create a new Node
2. newNode->data = value
```

```
3. newNode->next = NULL
4. If head == NULL
       head = newNode
       return head
5. p = head
6. While p->next != NULL
       p = p->next
7. p->next = newNode
8. return head
```

We must walk the whole list to reach the current last node (the one whose next is NULL), then attach the new node after it.

iii) Insert at a given position

Assume positions are 1-indexed (position 1 = insert as the new first node).

```
Algorithm Insert_At_Position(head, value, pos)
1. If pos == 1
       return Insert_First(head, value)
2. newNode = create a new Node
3. newNode->data = value
4. p = head
5. count = 1
6. While p != NULL AND count < pos - 1
       p = p->next
       count = count + 1
7. If p == NULL
       print "Position out of range"
       return head
8. newNode->next = p->next
9. p->next = newNode
10. return head
```

OR

5 a) **Compare Arrays and Linked List insertion and deletion operation with an example.** **5M**

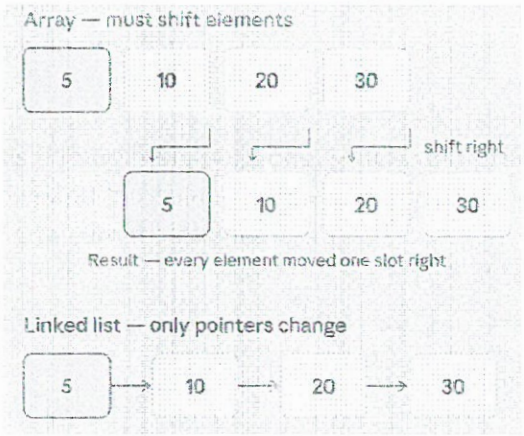
- **Array**: elements are stored in contiguous memory. Inserting or deleting in the middle requires *shifting* every subsequent element to keep them contiguous.
- **Linked list**: elements (nodes) are scattered in memory, connected via

pointers. Inserting or deleting only requires *relinking* a couple of pointers — no shifting.

Example: insert value 5 at the front of [10, 20, 30]

Array: [10, 20, 30] → must shift every element right by one slot to make room at index 0, *then* place 5 at index 0 → [5, 10, 20, 30]. That's 3 shift operations plus 1 insert.

Linked list: 10 → 20 → 30 → just create a new node with value 5, point it to the old head (10), and update head to the new node → 5 → 10 → 20 → 30. That's 1 pointer assignment, regardless of list size.



That same asymmetry shows up for deletion too — removing 10 from an array means shifting 20 and 30 left to close the gap, while removing the first node of a linked list just means moving head to point at the next node.

b) Write algorithm to perform following operations on a doubly linked list. **5M**

- **i) Delete at first**
- **ii) Delete at last**
- **iii) Delete at given position**

A doubly linked list has two pointers per node — next and prev — plus a head pointer to the first node and tail pointer to the last node for O(1) end access. Assume both head and tail are maintained.

Node structure:

```
struct Node {
  int data;
```

```
    Node* prev;
    Node* next;
};
```

i) Delete at first

```
Algorithm Delete_First(head, tail)
1. If head == NULL
     print "List is empty"
     return
2. temp = head
3. If head == tail          // only one node in the list
     head = NULL
     tail = NULL
   Else
     head = head->next
     head->prev = NULL
4. free(temp)
5. return head, tail
```

ii) Delete at last

```
Algorithm Delete_Last(head, tail)
1. If tail == NULL
     print "List is empty"
     return
2. temp = tail
3. If head == tail          // only one node in the list
     head = NULL
     tail = NULL
   Else
     tail = tail->prev
     tail->next = NULL
4. free(temp)
5. return head, tail
```

iii) Delete at a given position

Assume positions are 1-indexed (position 1 = first node).

```
Algorithm Delete_At_Position(head, tail, pos)
1. If head == NULL
     print "List is empty"
```

```
    return
2. If pos == 1
    return Delete_First(head, tail)
3. p = head
4. count = 1
5. While p != NULL AND count < pos
    p = p->next
    count = count + 1
6. If p == NULL
    print "Position out of range"
    return head, tail
7. If p == tail                    // deleting the last node
    return Delete_Last(head, tail)
8. p->prev->next = p->next
9. p->next->prev = p->prev
10. free(p)
11. return head, tail
```

UNIT-III

6 a) **Evaluate the given postfix expression using stack 934*8+4/-** **5M**

Rule

- If the symbol is an **operand**, push it onto the stack.
- If the symbol is an **operator**, pop the top two operands and perform:
  Second popped operand Operator First popped operand

Step-by-Step Evaluation

Symbol	Operation	Stack
9	Push 9	9
3	Push 3	9 3
4	Push 4	9 3 4
*	Pop Pop 3×4=12 Push 12	9 12
8	Push 8	9 12 8
+	Pop Pop 12+8=20 Push 20	9 20
4	Push 4	9, 20, 4
/	Pop Pop	9 5

	20÷4=5 Push 5	
-	Pop Pop 9−5=4 Push 4	4
End of input	Pop	

Result=4

b) Compare stack and queue. **5M**

Stack	Queue
Follows **LIFO (Last-In, First-Out)** principle.	Follows **FIFO (First-In, First-Out)** principle.
Insertion operation is called **PUSH**.	Insertion operation is called **ENQUEUE**.
Deletion operation is called **POP**.	Deletion operation is called **DEQUEUE**.
Insertion and deletion take place at the **same end**.	Insertion and deletion take place at **different ends**.
Insertion and deletion occur at the **TOP**.	Insertion occurs at the **REAR** and deletion occurs at the **FRONT**.
Uses a single pointer called **TOP**.	Uses two pointers called **FRONT** and **REAR**.
Example: **Stack of plates**.	Example: **People waiting in a line**.
Used in **recursion, expression evaluation, undo operations, and function calls**.	Used in **CPU scheduling, printer spooling, BFS, and buffering**.

OR

7) a) **Describe how stacks are used to check balanced parentheses. Explain with an example.** **5M**

A **stack** is used to check whether every opening parenthesis has a corresponding closing parenthesis in the correct order.

Algorithm

1. Scan the expression from **left to right**.
2. If an opening symbol (, {, or [is found, **PUSH** it onto the stack.
3. If a closing symbol), }, or] is found:
   - Check the top element of the stack.
   - If it is the matching opening symbol, **POP** it.
   - Otherwise, the expression is **not balanced**.
4. After scanning the entire expression:
   - If the stack is **empty**, the expression is **balanced**.
   - If the stack is **not empty**, the expression is **not balanced**.

Example: {() []}

Symbol	Action	Stack
{	Push {	{
(	Push (	{ (
)	Pop (	{
[	Push [	{ [
]	Pop [	{
}	Pop {	Empty

Since the stack is **empty at the end**, the expression is: BALANCED

b) Write pseudocode for the array implementation of stack operations. **5M**

- STACK[MAX] be an array used to store the elements.
- TOP indicate the position of the top element.
- Initially, TOP = -1.

Initialization

TOP ← -1

1. PUSH Operation

Purpose: To insert an element into the stack.

PUSH(STACK, ITEM)

1. If TOP = MAX - 1, then
   Print "STACK OVERFLOW"
   Return

2. TOP ← TOP + 1
3. STACK[TOP] ← ITEM
4. Return

2. POP Operation

Purpose: To remove the top element from the stack.

POP(STACK)

1. If TOP = -1, then
   Print "STACK UNDERFLOW"
   Return

2. ITEM ← STACK[TOP]
3. TOP ← TOP - 1
4. Return ITEM

3. PEEK Operation

Purpose: To view the top element without removing it.

PEEK(STACK)

1. If TOP = -1, then
   Print "STACK IS EMPTY"
   Return

2. Return STACK[TOP]

4. DISPLAY Operation

Purpose: To display all elements of the stack from top to bottom.

```
DISPLAY(STACK)

1. If TOP = -1, then
     Print "STACK IS EMPTY"
     Return

2. For i ← TOP down to 0, do
     Print STACK[i]

3. Return
```

UNIT-IV

8 a) **Discuss operations on Circular Queue for the following operations with an algorithm.** **5M**

i) Enqueue operation
ii) Dequeue operation

Operations on Circular Queue

A **circular queue** is a queue in which the last position is connected back to the first position. It follows the **FIFO (First-In, First-Out)** principle.

Let:

- CQ[MAX] be the circular queue array.
- FRONT point to the first element.
- REAR point to the last element.
- Initially, FRONT = REAR = -1.

i) Enqueue Operation

The **Enqueue** operation inserts a new element at the REAR end.

Overflow condition:
(REAR+1) mod MAX=FRONT

Algorithm: ENQUEUE(CQ, ITEM)

```
1. If (REAR + 1) mod MAX = FRONT then
    Print "QUEUE OVERFLOW"
    Return

2. If FRONT = -1, then
    FRONT ← 0
    REAR ← 0

3. Else
    REAR ← (REAR + 1) mod MAX

4. CQ[REAR] ← ITEM
5. Return
```

ii) Dequeue Operation

The **Dequeue** operation removes an element from the FRONT end.

Underflow condition:
FRONT=–1

Algorithm: DEQUEUE(CQ)

```
1. If FRONT = -1, then
    Print "QUEUE UNDERFLOW"
    Return

2. ITEM ← CQ[FRONT]

3. If FRONT = REAR, then
    FRONT ← -1
    REAR ← -1

4. Else
    FRONT ← (FRONT + 1) mod MAX

5. Return ITEM
```

b) What is Linear Queue? Compare how linear queue and circular queue utilize memory. **5M**

A **Linear Queue** is a linear data structure that follows the **FIFO (First-In, First-Out)** principle. The element inserted first is deleted first.

- Insertion (**Enqueue**) is performed at the **REAR**.
- Deletion (**Dequeue**) is performed at the **FRONT**.

Example

FRONT → 10 | 20 | 30 | 40 ← REAR

Here, 10 is deleted first, and new elements are inserted after 40.

Memory Utilization: Linear Queue vs Circular Queue

Feature	Linear Queue	Circular Queue
Structure	Last position is not connected to the first position.	Last position is logically connected to the first position.
Movement of REAR	REAR moves only forward.	REAR moves circularly.
Reuse of Deleted Spaces	Empty spaces created after deletion cannot be reused directly.	Empty spaces created after deletion can be reused.
Memory Utilization	Memory is not utilized efficiently.	Memory is utilized efficiently.
Wastage of Space	May cause memory wastage.	Avoids memory wastage.
Full Condition	REAR = MAX - 1	(REAR + 1) mod MAX = FRONT
Main Problem	May cause **false overflow** even when empty positions exist.	Avoids the problem of false overflow.

Example of Memory Wastage in a Linear Queue

Suppose the queue size is 5:

Initial:
FRONT → 10 | 20 | 30 | 40 | 50 ← REAR

After deleting 10 and 20:

Empty | Empty | 30 | 40 | 50
↑ ↑
FRONT REAR

Although the first two positions are empty, they cannot be reused directly in a linear queue because REAR has reached the last position. This leads to **wastage of memory** or **false overflow**.

In a circular queue, REAR can move back to the beginning and reuse those empty positions.

OR

9 a) **Develop an algorithm for implementing enqueue and dequeue operations of a queue using linked list by checking necessary conditions.** **5M**

A queue follows the **FIFO (First-In, First-Out)** principle.

In a linked-list implementation:

- FRONT points to the **first node**.
- REAR points to the **last node**.
- Insertion (**ENQUEUE**) is performed at REAR.
- Deletion (**DEQUEUE**) is performed at FRONT.

Initially:
FRONT = NULL
REAR = NULL

i) Algorithm for ENQUEUE Operation
Purpose: Insert a new element at the REAR of the queue.

Necessary condition: Check whether memory is available for a new node. If memory is unavailable, **Overflow** occurs.

Algorithm: ENQUEUE(ITEM)

1. Create a new node NEW

2. If NEW = NULL, then
   Print "QUEUE OVERFLOW"
   Return

3. NEW → DATA = ITEM
4. NEW → NEXT = NULL

5. If FRONT = NULL, then
   FRONT = NEW
   REAR = NEW

6. Else
   REAR → NEXT = NEW
   REAR = NEW

7. Return

ii) Algorithm for DEQUEUE Operation

Purpose: Delete an element from the FRONT of the queue.

Necessary condition: Check whether the queue is empty. If FRONT = NULL, **Underflow** occurs.

Algorithm: DEQUEUE()

1. If FRONT = NULL, then
   Print "QUEUE UNDERFLOW"
   Return

2. TEMP = FRONT
3. ITEM = TEMP → DATA
4. FRONT = FRONT → NEXT

5. If FRONT = NULL, then
   REAR = NULL

6. Free TEMP
7. Return ITEM

b) **Develop algorithm for implementation of Queue using array for enqueuer and dequeue operations.** **5M**

A **queue** follows the **FIFO (First-In, First-Out)** principle.

In an array implementation:

- FRONT points to the first element.
- REAR points to the last element.
- **ENQUEUE** inserts an element at REAR.
- **DEQUEUE** removes an element from FRONT.

Assume:

QUEUE[MAX]
FRONT ← -1
REAR ← -1

i) Algorithm for ENQUEUE Operation

Purpose: To insert an element into the queue.

Overflow condition:REAR=MAX−1

Algorithm: ENQUEUE(QUEUE, ITEM)

1. If REAR = MAX - 1, then
   Print "QUEUE OVERFLOW"
   Return

2. If FRONT = -1, then
   FRONT = 0

3. REAR = REAR + 1

4. QUEUE[REAR] = ITEM

5. Return

ii) Algorithm for DEQUEUE Operation

Purpose: To delete an element from the queue.

Underflow condition:

FRONT=−1

Algorithm: DEQUEUE(QUEUE)

1. If FRONT = -1 then
   Print "QUEUE UNDERFLOW"
   Return

2. ITEM = QUEUE[FRONT]

3. If FRONT = REAR, then
   FRONT = -1
   REAR = -1

4. Else
   FRONT = FRONT + 1

5. Return ITEM

UNIT-V

10 a) **Write an algorithm for Level Order tree traversal using iteration and give an example.** 5M

Level Order Traversal visits the nodes of a binary tree **level by level from left to right**. It is also called **Breadth-First Traversal**.

A **Queue** is used for iterative level order traversal because a queue follows the **FIFO (First-In, First-Out)** principle.

Algorithm: LEVEL_ORDER(ROOT)

1. If ROOT = NULL, then
   Return

2. Create an empty queue Q

3. ENQUEUE(Q, ROOT)

4. While Q is not empty, do

   a. NODE = DEQUEUE(Q)

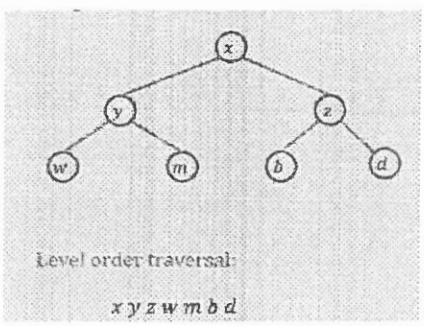
   b. Visit NODE → DATA

   c. If NODE → LEFT ≠ NULL, then
      ENQUEUE(Q, NODE → LEFT)

   d. If NODE → RIGHT ≠ NULL, then
      ENQUEUE(Q, NODE → RIGHT)

5. Stop

Example:



b) **Construct hash table for the following keys 50,700,76,85,92,73,10 using separate chaining where h(k)=kmod7** **5M**

Constructing a Hash Table Using Separate Chaining
Given
Keys:
50, 700, 76, 85, 92, 73, 10
Hash function:
h(k)=k mod 7
Since the table size is 7, the hash table indices are:
0, 1, 2, 3, 4, 5,6
Step 1: Calculate the Hash Value for Each Key

Key	Calculation k mod 7	Hash Index
50	50 mod 7=1	1
700	700 mod 7=0	0
76	76 mod 7=6	6
85	85 mod 7=1	1
92	92 mod 7=1	1
73	73 mod 7=3	3
10	10 mod 7=3	3

Keys with the same hash index are stored in a linked list using **separate chaining.**

Index	value
0	700
1	50→85->92
2	
3	73->10
4	
5	
6	76

OR

11 a) **Develop an algorithm for implementing the linear probing collision resolution technique and give an example.** 5M

Linear probing is an **open addressing** method used to resolve collisions in a hash table.

When a collision occurs, the next consecutive locations are checked one by one until an empty location is found.

The probing sequence is:

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

where:

- h(k) = original hash value
- i=0,1,2,...,m−1
- m= size of the hash table

Algorithm for Insertion Using Linear Probing

Algorithm: LINEAR_PROBING_INSERT(HT, KEY, m)

1. INDEX = KEY mod m
2. START = INDEX

3. While HT[INDEX] is not EMPTY, do

    INDEX = (INDEX + 1) mod m

    If INDEX = START, then
        Print "HASH TABLE IS FULL"
        Return

4. HT[INDEX] = KEY

5. Return

Example

Construct a hash table for the keys:

50, 700, 76, 85, 92, 73, 10

using: $h(k)=k \bmod 7$

The hash table has indices 0 to 6.

Key	Hash Calculation	Collision Resolution	Final Index
50	50 mod 7=1	Index 1 is empty	1
700	700 mod 7=0	Index 0 is empty	0
76	76 mod 7=6	Index 6 is empty	6
85	85 mod 7=1	1 occupied → check 2	2
92	92 mod 7=1	1 occupied → 2 occupied → check 3	3
73	73 mod 7=3	3 occupied → check 4	4
10	10 mod 7=3	3 occupied → 4 occupied → check 5	5

Final Hash Table

Index	Key
0	700
1	50
2	85
3	92
4	73
5	10
6	76

b) Construct a Binary Search Tree for the given preorder traversal sequence 22,12,8,20,30,25,40 **5M**

Given Preorder Sequence

22, 12, 8, 20, 30, 25, 40

BST Property

For every node in a Binary Search Tree:

Left subtree<Root<Right subtree

We insert the elements in the given preorder sequence one by one.

Construction

1. Insert 22 — It is the first element, so it becomes the **root**.

2. Insert 12 — 12 < 22, so it becomes the **left child of 22**.

3. Insert 8 — 8 < 22 and 8 < 12, so it becomes the **left child of 12**.

4. Insert 20 — 20 < 22 but 20 > 12, so it becomes the **right child of 12**.

5. Insert 30 — 30 > 22, so it becomes the **right child of 22**.

6. Insert 25 — 25 > 22 but 25 < 30, so it becomes the **left child of 30**.

7. Insert 40 — 40 > 22 and 40 > 30, so it becomes the **right child of 30**.

Final Binary Search Tree