		UNIT-V			
10	a)	Let us consider a simple hash function as "key mod 11" and sequence of keys as 50, 700, 76, 85, 92, 73, 101, 45, 62, 99 with table size 11. Show how these keys will be stored, if we apply quadratic probing in case of collision.	L4	CO4	5 M
	b)	Discuss about insertion and deletion of an element in binary search tree.	L2	CO4	5 M
		OR			
11	a)	Define Binary search tree. Construct binary search tree with following keys: 55,45,65,40,60,70,66,99,2,34	L3	CO4	5 M
	b)	Assume a table has 8 slots. Using chaining, insert the following elements into the hash table. 56,66,18,72,43,65,6,17,10,5,64,16,71, and 15 are inserted in the order. Consider Hash function: h(k) = k mod m, where m=8.	L4	CO4	5 M

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

I B.Tech - II Semester – Regular Examinations - JULY 2024

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)	Define abstract data type.	L1	CO1
1.b)	Differentiate binary search and linear search.	L2	CO1
1.c)	Compare implementation of list with arrays and pointers. Which is efficient? Justify your answer in one or two lines.	L2	CO1
1.d)	How self-referential structures are useful in the implementation of linked list data structure.	L1	CO1
1.e)	List the applications of stack.	L1	CO1
1.f)	Give the overflow condition and underflow condition of stack in array implementation.	L1	CO1
1.g)	In the circular implementation of queue, what is the condition to check queue is empty or not?	L1	CO1
1.h)	List the advantages of pointer implementation of queue over array implementation.	L1	CO1
1.i)	Define binary tree.	L1	CO1
1.j)	Describe the purpose of hashing.	L2	CO1

PART – B

			BL	CO	Max. Marks
		UNIT-I			
2	a)	Apply bubble sort on the following elements: 10, 4, 12, 3, 23, 1. Show each iteration very clearly.	L3	CO2	5 M
	b)	Discuss how do we measure the complexity of an algorithm.	L2	CO1	5 M
		OR			
3	a)	Explain selection sort algorithm with suitable example.	L2	CO2	5 M
	b)	Discuss the importance of linear data structures.	L2	CO1	5 M
		UNIT-II			
4	a)	Develop pseudo code to print elements of linked list in reverse order.	L3	CO3	5 M
	b)	Discuss the following operations in circular linked list: i. Insert an element ii. Delete an element	L2	CO3	5 M
		OR			
5	a)	Compare singly linked list and doubly linked list.	L2	CO1	5 M
	b)	Explain the array implementation of list in detail.	L2	CO1	5 M

		UNIT-III			
6	a)	Develop algorithm to convert infix expression to postfix expression.	L3	CO3	5 M
	b)	Explain implementation of stack using pointers.	L2	CO1	5 M
		OR			
7	a)	Explain push() and pop() functions of stack data structure with array implementation.	L2	CO1	5 M
	b)	Describe the process of evaluating post fix expression using stack.	L2	CO3	5 M
		UNIT-IV			
8	a)	Explain about array implementation of queue.	L2	CO1	5 M
	b)	What is circular queue? What is advantage of circular queue over linear queue? Demonstrate with a scenario.	L2	CO1	5 M
		OR			
9	a)	Find the list of elements in the queue with following operations in sequence: insert(10), insert(20), delete, insert(30), insert(40), delete. Assume initially queue is empty.	L3	CO3	5 M
	b)	Discuss about pointer implementation of queue.	L2	CO1	5 M

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

PVP 23

I B.Tech - II Semester – Regular Examinations - JULY 2024

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

SHORT SCHEME

PART – A

		BL	CO
1.a)	Define abstract data type.	L1	CO1

Scheme: Definition – 2M

1.b)	Differentiate binary search and linear search.	L2	CO1

Scheme: Any two differences – 2M

1.c)	Compare implementation of list with arrays and pointers. Which is efficient? Justify your answer in one or two lines.	L2	CO1

Scheme: Any two differences - 1M

Justification – 1M

1.d)	How self-referential structures are useful in the implementation of linked list data structure.	L1	CO1

Scheme: Ant two uses/benefits of self-referential structures - 2M

1.e)	List the applications of stack.	L1	CO1

Scheme: Any Two applications – 2M

1.f)	Give the overflow condition and underflow condition of stack in array implementation.	L1	CO1

Scheme: Overflow condition – 1M

Underflow condition - 1M

1.g)	In the circular implementation of queue, what is the condition to check queue is empty or not?	L1	CO1

Scheme: Condition to check empty queue – 2M

1.h)	List the advantages of pointer implementation of queue over array implementation.	L1	CO1

Scheme: Any two advantages – 2M

1.i)	Define binary tree.	L1	CO1

Scheme: Definition – 2M

1.j)	Describe the purpose of hashing.	L2	CO1

Scheme: Purpose of Hashing – 2M

PART – B

			BL	CO	Max. Marks
		UNIT-I			
2	a)	Apply bubble sort on the following elements: 10, 4, 12, 3, 23, 1. Show each iteration very clearly.	L3	CO2	5 M
	b)	Discuss how do we measure the complexity of an algorithm.	L2	CO1	5 M

Scheme: 2 a) For showing five iterations - 5M

2 b) Measuring of Time-complexity for a sample Algorithm - 3M

Measuring of Space-complexity for a sample Algorithm – 2M

		OR			
3	a)	Explain selection sort algorithm with suitable example.	L2	CO2	5 M
	b)	Discuss the importance of linear data structures.	L2	CO1	5 M

Scheme: 3 a) Explanation of Algorithm/Procedure – 3M

Suitable example - 2M

3 b) Any Five important features of Linear Data Structures – 5M

		UNIT-II			
4	a)	Develop pseudo code to print elements of linked list in reverse order.	L3	CO3	5 M
	b)	Discuss the following operations in circular linked list: i. Insert an element ii. Delete an element	L2	CO3	5 M

Scheme: 4 a) Pseudocode – 5M

4 b) Insertion – 2M Deletion – 3M

		OR			
5	a)	Compare singly linked list and doubly linked list.	L2	CO1	5 M
	b)	Explain the array implementation of list in detail.	L2	CO1	5 M

Scheme: 5 a) Any 5 Comparisons – 5M

5 b) Sophisticated explanation of list – 5M

		UNIT-III			
6	a)	Develop algorithm to convert infix expression to postfix expression.	L3	CO3	5 M
	b)	Explain implementation of stack using pointers.	L2	CO1	5 M

Scheme: 6 a) Algorithm/Procedure – 5M

6 b) Sophisticated explanation of Stack using linked list - 5M

		OR			
7	a)	Explain push() and pop() functions of stack data structure with array implementation.	L2	CO1	5 M
	b)	Describe the process of evaluating post fix expression using stack.	L2	CO3	5 M

Scheme: 7 a) push() operation – 3M

pop() operation – 2M

7 b) Sophisticated explanation with an example – 5M

		UNIT-IV			
8	a)	Explain about array implementation of queue.	L2	CO1	5 M
	b)	What is circular queue? What is advantage of circular queue over linear queue? Demonstrate with a scenario.	L2	CO1	5 M

Scheme: 8 a) Sophisticated explanation of queue using array– 5M

8 b) Definition – 1M

Advantages – 2M

Scenario - 2M

		OR			
9	a)	Find the list of elements in the queue with following operations in sequence: insert(10), insert(20), delete, insert(30), insert(40), delete. Assume initially queue is empty.	L3	CO3	5 M
	b)	Discuss about pointer implementation of queue.	L2	CO1	5 M

Scheme: 9 a) Showing Queue contents for every operation with explanation – 5M

9 b) Sophisticated explanation of queue using linked list – 5M

UNIT-V					
10	a)	Let us consider a simple hash function as "key mod 11" and sequence of keys as 50, 700, 76, 85, 92, 73, 101, 45, 62, 99 with table size 11. Show how these keys will be stored, if we apply quadratic probing in case of collision.	L4	CO4	5 M
	b)	Discuss about insertion and deletion of an element in binary search tree.	L2	CO4	5 M

Scheme: 10 a) Construction of Hash Table with quadratic probing – 5M

10 b) Insertion operation – 2M Deletion operation – 3M

OR					
11	a)	Define Binary search tree. Construct binary search tree with following keys: 55,45,65,40,60,70,66,99,2,34	L3	CO4	5 M
	b)	Assume a table has 8 slots. Using chaining, insert the following elements into the hash table. 56,66,18,72,43,65,6,17,10,5,64,16,71, and 15 are inserted in the order. Consider Hash function: h(k) = k mod m, where m=8.	L4	CO4	5 M

Scheme: 11 a) Definition – 1M Construction of BST – 4M

11 b) Construction of Hash Table using chaining – 5M

PVP 23

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

I B.Tech - II Semester – Regular Examinations - JULY 2024

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

SCHEME OF EVALUATION

PART – A

		BL	CO
1.a)	Define abstract data type.	L1	CO1

Scheme: Definition – 2M

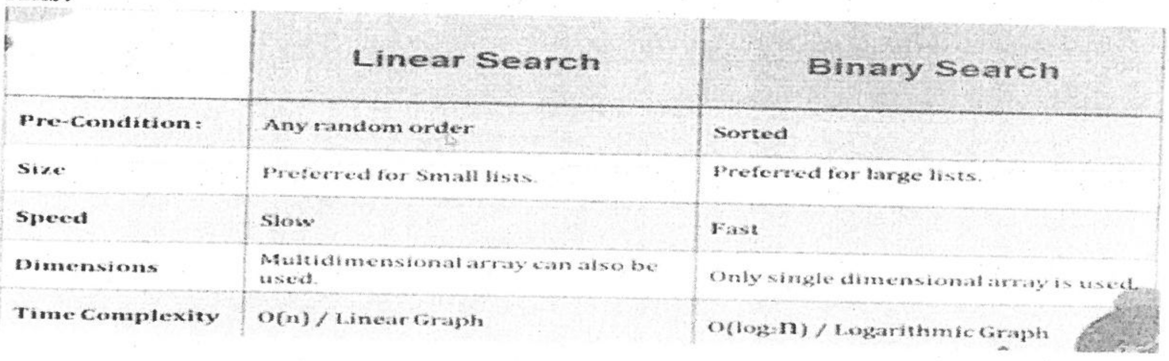
Ans: An abstract data type (ADT) is a mathematical model for data types, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations.

ADT is a collection of data together with a set of operations on that data.

ADT = Data + Operations

1.b)	Differentiate binary search and linear search.	L2	CO1

Scheme: Any two differences – 2M
Ans:



	Linear Search	Binary Search
Pre-Condition :	Any random order	Sorted
Size	Preferred for Small lists.	Preferred for large lists.
Speed	Slow	Fast
Dimensions	Multidimensional array can also be used.	Only single dimensional array is used.
Time Complexity	O(n) / Linear Graph	$O(\log_2 n)$ / Logarithmic Graph

1.c)	Compare implementation of list with arrays and pointers. Which is efficient? Justify your answer in one or two lines.	L2	CO1

Scheme: Any two differences - 1M

Justification – 1M

Ans:

Lists Implemented with Arrays

□ **Random Access:** Direct access to elements by index.

□ **Cache Efficiency:** Elements are stored contiguously, which can improve cache performance in some scenarios.

□ **Simplicity:** Relatively straightforward to implement and understand.

Lists Implemented with Pointers (Linked Lists)

□ **Dynamic Size:** Can efficiently grow and shrink without the overhead of resizing.

□ **Memory Efficiency:** Allocates memory as needed, reducing potential wasted space.

□ **Flexible Insertions and Deletions:** Allows for efficient insertions and deletions at both ends and within the list (with traversal).

Implementation of List with pointer is Better
Two Reasons:
With array implementation, Insert at beginning - O(n) but
With pointer implementation, Insert at beginning - O(1)
With array implementation, Delete at beginning - O(n) but
With pointer implementation, Delete at beginning - O(1)

1.d)	How self-referential structures are useful in the implementation of linked list data structure.	L1	CO1

Scheme: Ant two uses/benefits of self-referential structures - 2M

Ans:

Benefits of Self-Referential Structures in Linked Lists

Dynamic Size

Efficient Insertions and Deletions

Flexibility

Memory Utilization

Linked lists use memory more efficiently for data sets that change frequently, as they do not require contiguous memory allocation.

1.e)	List the applications of stack.	L1	CO1

Scheme: Any Two applications – 2M

Ans:

Expression Evaluation and Conversion

String Reversal

Balanced Parentheses

Palindrome Checking

Function Call Management

Backtracking Algorithms

Undo Mechanism in Applications

Parsing and Compilers

Browser Navigation

1.f)	Give the overflow condition and underflow condition of stack in array implementation.	L1	CO1

Scheme: Overflow condition – 1M

Underflow condition - 1M

Ans:

i) Stack Overflow: A stack overflow occurs when a program attempts to use more space on the call stack than is available.

If Stack implemented with arrays stack overflow: if(top==size-1) printf("Overflow");

ii) Stack Underflow: A stack underflow occurs when a program attempts to pop an item from an empty stack.

If Stack implemented with arrays if(top==-1) printf("Underflow"); or

If Stack implemented with Linked list if(top==NULL) printf("Underflow");

1.g)	In the circular implementation of queue, what is the condition to check queue is empty or not?	L1	CO1

Scheme: Condition to check empty queue – 2M

Ans: In a circular linked list implementation of a queue, you can check if the queue is empty using the following condition:

```
if (front == NULL) {
  // Queue is empty
}
```

In a circular Array implementation of a queue, you can check if the queue is empty using the following condition:

```
if (front==rear || front == -1 || rear == -1){
  // Queue is empty
}
```

1.h)	List the advantages of pointer implementation of queue over array implementation.	L1	CO1

Scheme: Any two advantages – 2M

Ans: Pointer implementation of queues using linked lists offers several advantages over array-based implementations.

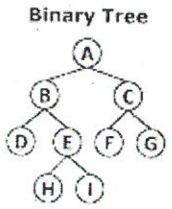
Here are some key advantages:

1. Dynamic Size
2. Efficient Memory Usage
3. No Overflow
4. Constant Time Operations
5. No Shifting of Elements
6. Simpler Code for Dynamic Operations
7. Better for Unknown or Large Sizes

1.i)	Define binary tree.	L1	CO1

Scheme: Definition – 2M

Ans: A binary tree is a hierarchical data structure in computer science where each node has at most two children, referred to as the left child and the right child.



1.j)	Describe the purpose of hashing.	L2	CO1

Scheme: Purpose of Hashing – 2M

Ans: Hashing is a technique in data structures that uses a hash function to map data of any size to a fixed-size value. The main purpose is to enable fast and efficient data access by reducing search time.

PART – B

			BL	CO	Max. Marks
UNIT-I					
2	a)	Apply bubble sort on the following elements: 10, 4, 12, 3, 23, 1. Show each iteration very clearly.	L3	CO2	5 M
	b)	Discuss how do we measure the complexity of an algorithm.	L2	CO1	5 M

2 a) For showing five iterations - 5M

Ans: Bubble sort works on the repeatedly swapping of adjacent elements until they are not in the intended order.

let's apply the bubble sort algorithm step-by-step to sort the elements: 10, 4, 12, 3, 23, 1.

Initial Array:

10,4,12,3,23,1

First Pass:

In each pass, adjacent elements are compared and swapped if they are in the wrong order.

Comparing 10 and 4:

Swap because 4 < 10

4,10,12,3,23,1

Comparing 10 and 12:

No swap because 10 < 12

4,10,12,3,23,1

Comparing 12 and 3:

Swap because 3 < 12

4,10,3,12,23,1

Comparing 12 and 23:

No swap because 12 < 23

4,10,3,12,23,1

Comparing 23 and 1:

Swap because 1 < 23

4,10,3,12,1,23

First Pass Result:

4,10,3,12,1,23

Second Pass:

Comparing 4 and 10:

No swap because 4 < 10

4,10,3,12,1,23

Comparing 10 and 3:

Swap because 3 < 10

4,3,10,12,1,23

Comparing 10 and 12:

No swap because 10 < 12

4,3,10,12,1,23

Comparing 12 and 1:

Swap because 1 < 12

4,3,10,1,12,23

Comparing 12 and 23:

No swap because 12 < 23

4,3,10,1,12,23

Second Pass Result:

4,3,10,1,12,23

Third Pass:

Comparing 4 and 3:

Swap because 3 < 4

3,4,10,1,12,23

Comparing 4 and 10:

No swap because 4 < 10

3,4,10,1,12,23

Comparing 10 and 1:

Swap because 1 < 10

3,4,1,10,12,23

Comparing 10 and 12:

No swap because 10 < 12

3,4,1,10,12,23

Comparing 12 and 23:

No swap because 12 < 23

3,4,1,10,12,23

Third Pass Result:

3,4,1,10,12,23

Fourth Pass:

Comparing 3 and 4:

No swap because 3 < 4

3,4,1,10,12,23

Comparing 4 and 1:

Swap because 1 < 4

3,1,4,10,12,23

Comparing 4 and 10:

No swap because 4 < 10

3,1,4,10,12,23

Comparing 10 and 12:

No swap because 10 < 12

3,1,4,10,12,23

Comparing 12 and 23:

No swap because 12 < 23

3,1,4,10,12,23

Fourth Pass Result:

3,1,4,10,12,23

Fifth Pass:

Comparing 3 and 1:

Swap because 1 < 3

1,3,4,10,12,23

Comparing 3 and 4:

No swap because 3 < 4

1,3,4,10,12,23

Comparing 4 and 10:

No swap because 4 < 10

1,3,4,10,12,23

Comparing 10 and 12:

No swap because 10 < 12

1,3,4,10,12,23

Comparing 12 and 23:

No swap because 12 < 23

1,3,4,10,12,23

Fifth Pass Result:

1,3,4,10,12,23

Final Sorted List:

1,3,4,10,12,23

The array is now sorted using the bubble sort algorithm after 5 passes. Each pass involves comparing adjacent elements and swapping them if necessary to gradually move larger elements towards the end of the array.

2 b) Measuring of Time-complexity for a sample Algorithm - 3M

Measuring of Space-complexity for a sample Algorithm – 2M

Ans: Complexity in algorithms refers to the amount of resources (such as time or memory) required to solve a problem or perform a task.

Time complexity, which refers to the amount of time an algorithm takes to produce a result as a function of the size of the input.

Space Complexity, which refers to the total amount of memory space used by an algorithm/program, including the space of input values for execution.

Calculate time complexity for factorial of given number.

s/e – the number of operations to be performed when we execute the given line.

freq – no.of times the given line will be executed.

Algorithm	s/e	freq	total
int fact(int n)	0	-	0
{	0	-	0
int i, f;	2	1	2
f=1;	1	1	1
for(i=1; i<=n ;i++)	2	n+1	2n+2
f=f*i;	2	n	2n
return f;	1	1	1
}	0	-	0
			4n+6

Time complexity = 4n+6 = O(n)

Space complexity:

Calculate time complexity for factorial of given number.

Code – Assume 100 Bytes

3 int variables – 3x4=12 Bytes

Total = 112 Bytes (Constant)

Space Complexity = O(1)

OR					
3	a)	Explain selection sort algorithm with suitable example.	L2	CO2	5 M
	b)	Discuss the importance of linear data structures.	L2	CO1	5 M

3 a) Explanation of Algorithm/Procedure – 3M

Suitable example - 2M

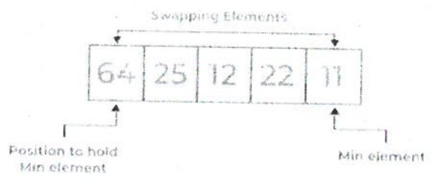
Ans: Selection sort is a simple sorting algorithm that works by repeatedly finding the minimum element from the unsorted part of the array and placing it at the beginning.

Procedure:

1. Find the minimum element in the unsorted array: Iterate through the unsorted part of the array to find the smallest element.
2. Swap the minimum element with the first element of the unsorted array: Place the minimum element at the beginning of the unsorted part.
3. Repeat: Consider the array with the newly placed element as sorted and repeat steps 1 and 2 for the remaining unsorted part until the entire array is sorted.

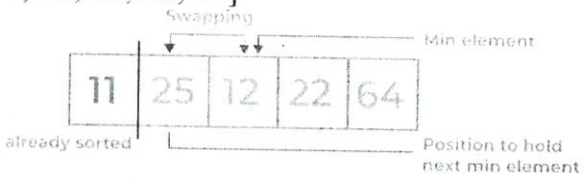
Example:

Let's say we have an array: [64, 25, 12, 22, 11]



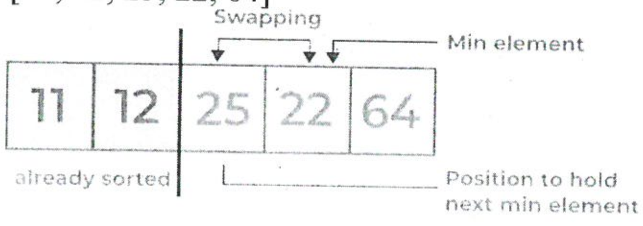
Pass 1:

- Find the minimum element in the array (11) and swap it with the first element (64).
- Array becomes: [11, 25, 12, 22, 64]



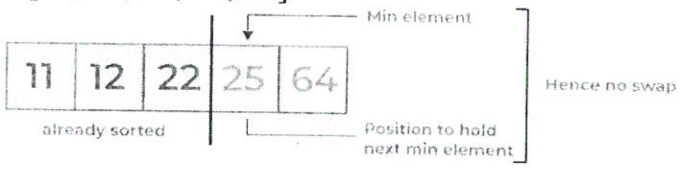
Pass 2:

- Find the minimum element in the remaining unsorted array (12) and swap it with the second element (25).
- Array becomes: [11, 12, 25, 22, 64]



Pass 3:

- Find the minimum element in the remaining unsorted array (22) and swap it with the third element (25).
- Array becomes: [11, 12, 22, 25, 64]



Pass 4:

- Find the minimum element in the remaining unsorted array (25) and swap it with the fourth element (25).
- Array becomes: [11, 12, 22, 25, 64]

11
12
22
25
64

Sorted array

The array is now sorted!

Complexity of Selection Sort:

Time Complexity: O(n^2) in all cases (best, average, and worst). This is because it always iterates through *n* elements and performs comparisons.

Space Complexity: O(1) (constant), as it requires only a few additional variables regardless of the input size.

3 b) Any Five important features of Linear Data Structures – 5M

Ans: Linear data structures play a crucial role in organizing and managing data in various applications and algorithms. They are fundamental in computer science and are characterized by their sequential and ordered nature, where each element is connected to its predecessor and successor.

1. Efficient Insertion and Deletion Operations:

Linear data structures such as arrays and linked lists offer efficient insertion and deletion operations:

Arrays: In dynamic arrays, appending elements is O(1) on average (amortized), while inserting or deleting elements from the middle requires shifting elements, which is O(n). In contrast, in linked lists, insertion and deletion operations are O(1) if the position is known.

Linked Lists: Singly linked lists offer efficient insertion and deletion at the beginning and end of the list, making them suitable for scenarios where elements need to be dynamically added or removed.

2. Ease of Access and Traversal:

Linear structures facilitate straightforward access and traversal of elements:

Arrays: Elements in arrays are accessed in constant time O(1) using their index. This makes them ideal for scenarios where direct access to elements is required.

Linked Lists: Linked lists provide sequential access to elements through traversal, which can be O(n) in the worst case, but efficient for forward and backward traversal in doubly linked lists.

3. Dynamic Size Management:

Linear structures like linked lists can grow and shrink dynamically:

Arrays: Static arrays have fixed sizes, but dynamic arrays (like in many programming languages) can resize dynamically as elements are added or removed.

Linked Lists: Linked lists can easily accommodate varying numbers of elements by allocating memory dynamically as nodes are added or freed when nodes are removed.

4. Versatility in Implementation:

Linear data structures can be adapted for various applications and algorithms:

Stacks and Queues: Implemented using arrays or linked lists, they provide specific behavior (Last-In-First-Out for stacks, First-In-First-Out for queues) crucial in algorithms like depth-first search (DFS) and breadth-first search (BFS).

5. Foundation for Advanced Data Structures:

Linear structures serve as building blocks for more complex data structures:

Trees and Graphs: Trees (like binary trees) and graphs (like adjacency lists) often use arrays or linked lists to store and manage nodes and edges, providing structured relationships between elements.

Hash Tables: Hash tables, which use arrays combined with hashing techniques, rely on efficient access and storage principles from linear structures.

UNIT-II					
4	a)	Develop pseudo code to print elements of linked list in reverse order.	L3	CO3	5 M
	b)	Discuss the following operations in circular linked list: i. Insert an element ii. Delete an element	L2	CO3	5 M

Scheme: 4 a) Pseudocode – 5M

Ans: To print the elements of a linked list in reverse order, we can use a recursive approach. The idea is to traverse the list to the end and then print the elements during the unwinding phase of the recursion.

Here is the pseudocode for printing the elements of a singly linked list in reverse order:

Pseudocode for Recursive Approach

```
Function printReverse(node):
    If node is NULL:
        Return
    End If
    // Recurse to the next node
    printReverse(node.next)
    // Print the current node's data during the unwinding phase
    Print node.data
End Function
// Driver code to call the function
Function main():
    head = Initialize the linked list with some values
    // Call the recursive function starting from the head
    printReverse(head)
End Function
```

Explanation:

1. Base case:

If the head is null, it means we've reached the end of the linked list (or it was empty to begin with). So, we simply return.

2. Recursive call:

We make a recursive call to the same function, but with the next node as the argument. This allows us to traverse the entire linked list until we reach the end.

3. Print data:

After returning from the recursive call, we print the data of the current node. Since the last recursive call returns first, this ensures the elements are printed in reverse order.

4 b) Insertion – 2M **Deletion – 3M**

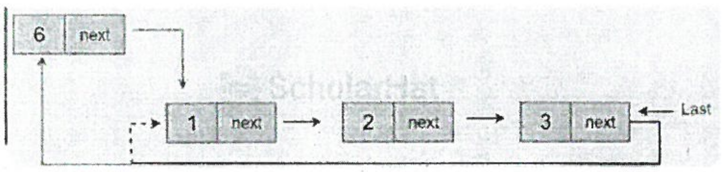
Ans: Insertion operations in a circular linked list can be performed at different positions: at the beginning, at the end, or at a specific position within the list. In a circular linked list, the last node points back to the first node, creating a circular structure. Here's a detailed explanation of the insertion operations:

1. Insertion at the Beginning

To insert a node at the beginning of a circular linked list, you need to:

1. Create a new node.

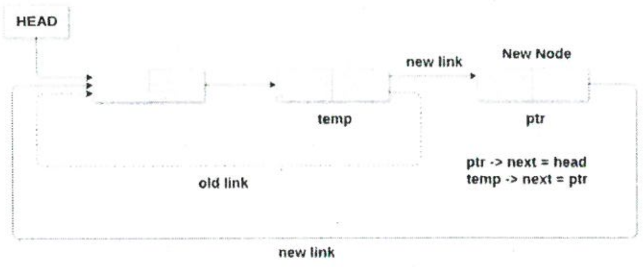
2. Update the new node to point to the current head.
3. Update the last node to point to the new node.
4. Set the head to the new node.



2. Insertion at the End

To insert a node at the end of a circular linked list, you need to:

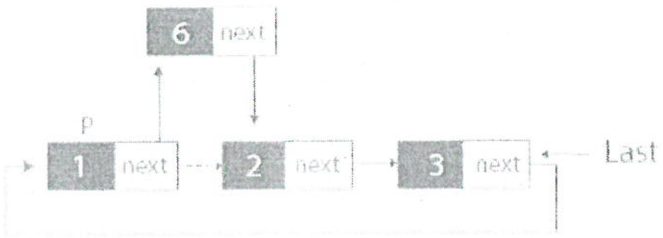
1. Create a new node.
2. Traverse to the last node.
3. Update the last node's next pointer to the new node.
4. Set the new node's next pointer to the head.



3. Insertion at a Specific Position

To insert a node at a specific position in a circular linked list, you need to:

1. Create a new node.
2. Traverse to the node just before the desired position.
3. Update the new node's next pointer to the next node in the sequence.
4. Update the previous node's next pointer to the new node.



In a circular linked list, deletion operations can be performed at various positions: at the beginning, at the end, or at a specific position.

- **Deletion at the Beginning**:

  - If the list is empty, print a message and return.

- If the list has only one node, set head to NULL.
- Otherwise, traverse to the last node, update its next pointer to point to the second node, and update head to the second node.

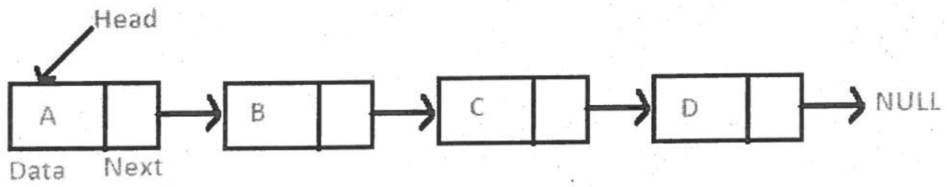
- **Deletion at the End**:
  - If the list is empty, print a message and return.
  - If the list has only one node, set head to NULL.
  - Otherwise, traverse to the second last node, update its next pointer to point to the head.

- **Deletion at a Specific Position**:
  - If the list is empty, print a message and return.
  - If the position is 1, call the deleteAtBeginning function.
  - Otherwise, traverse to the node just before the desired position, update its next pointer to skip the node to be deleted, and point to the node after the deleted node.
  - If the position is out of range, print an error message.

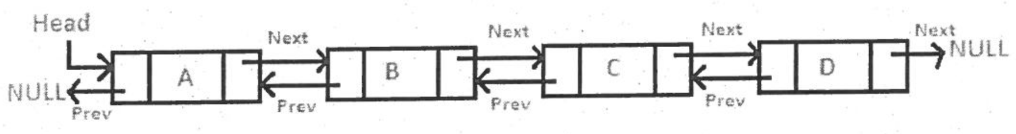
		OR			
5	a)	Compare singly linked list and doubly linked list.	L2	CO1	5 M
	b)	Explain the array implementation of list in detail.	L2	CO1	5 M

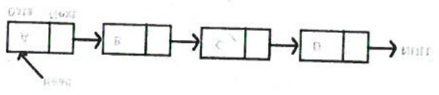
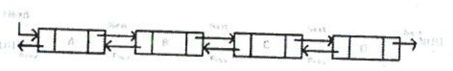
Scheme: 5 a) Any 5 Comparisons – 5M

Ans: **Introduction to Singly linked list :** A singly linked list is a set of nodes where each node has two fields 'data' and 'link'. The 'data' field stores actual piece of information and 'link' field is used to point to next node.



Introduction to Doubly linked list : A Doubly Linked List (DLL) contains an extra pointer, typically called *previous pointer*, together with next pointer and data which are there in singly linked list.



Singly linked list (SLL)	**Doubly linked list (DLL)**
SLL nodes contains 2 field -data field and next link field.	DLL nodes contains 3 fields -data field, a previous link field and a next link field.
	
In SLL, the traversal can be done using the next node link only. Thus traversal is possible in one direction only.	In DLL, the traversal can be done using the previous node link or the next node link. Thus traversal is possible in both directions (forward and backward).
The SLL occupies less memory than DLL as it has only 2 fields.	The DLL occupies more memory than SLL as it has 3 fields.
Complexity of insertion and deletion at a given position is O(n).	Complexity of insertion and deletion at a given position is O(n / 2) = O(n) because traversal can be made from start or from the end.
Complexity of deletion with a given node is O(n), because the previous node needs to be known, and traversal takes O(n).	Complexity of deletion with a given node is O(1) because the previous node can be accessed easily.
We mostly prefer to use singly linked list for the execution of stacks.	We can use a doubly linked list to execute heaps and stacks, binary trees.
When we do not need to perform any searching operation and we want to save memory, we prefer a singly linked list.	In case of better implementation, while searching, we prefer to use doubly linked list.
A singly linked list consumes less memory as compared to the doubly linked list.	The doubly linked list consumes more memory as compared to the singly linked list.
Singly linked list is less efficient. It is preferred when we need to save memory and searching is not required as pointer of single index is stored.	Doubly linked list is more efficient. When memory is not the problem and we need better performance while searching, we use doubly linked list.

5 b) Sophisticated explanation of list – 5M

Ans: **Array Implementation of List ADT**

An array-based list is an implementation of the List ADT that stores list elements in contiguous array positions. The index of the array relates to the position of the item in the list. The implementation specifies an array of a particular maximum length, and all storage is allocated before run-time.

The simplest method to implement a List ADT is to use an array that is a "linear list" or a "contiguous list" where elements are stored in contiguous array positions. The implementation specifies an array of a particular maximum length, and all storage is allocated before run-time. It is a sequence of n-elements where the items in the array are stored with the index of the array related to the position of the item in the list.

In array implementation, elements are stored in contiguous array positions (Figure 3.1). An array is a viable choice for storing list elements when the elements are sequential, because it is a commonly available data type and in general algorithm development is easy.

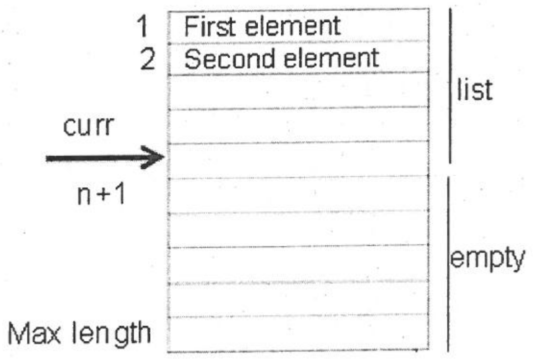


Figure 3.1 Array Implementation of List ADT

List Implementation using arrays

In order to implement lists using arrays we need an array for storing entries – **listArray[0,1,2.....m]**, a variable **curr** to keep track of the number of array elements currently assigned to the list, the number of items in the list, or current size of the list **size** and a variable to record the maximum length of the array, and therefore of the list – **maxsize as shown in Figure 3.2.**

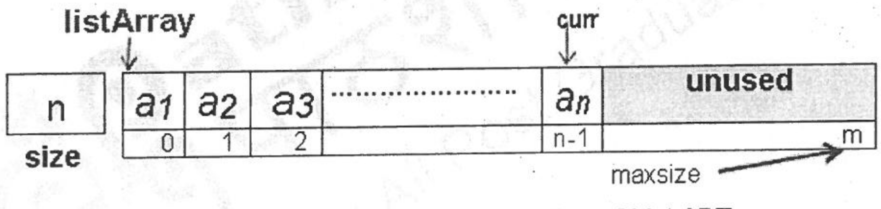


Figure 3.2Static Implementation of List ADT

Fix one end of the list at index 0 and elements will be shifted*as needed* when an element is added or removed. Therefore insert and delete operations will take O(n).

Insertion

What happens if you want to insert an item at a specified position in an existing array? The item originally at the given index must be moved right one position, and all the items after that index must be *shifted right* (Figure 3.3).

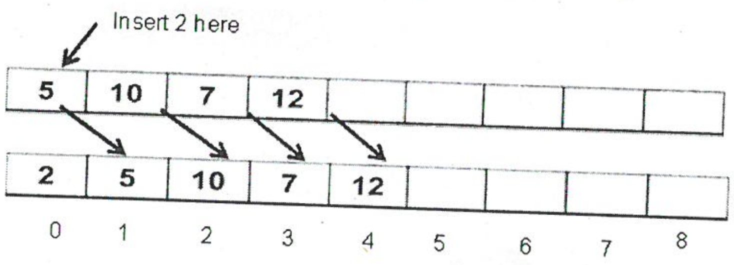


Figure 3.3 Insertion into a List

Let us consider a specific case of Insert operation. Insert (3, K, List) In this case, 3 is the index or the point of Insertion, K is the element to be inserted & List is the list in which insertion is to be done (Figure 3.4 (a) & Figure 3.4 (b)).

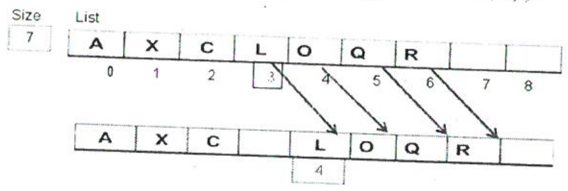


Figure 3.4 (a)An Example of Insertion into a List

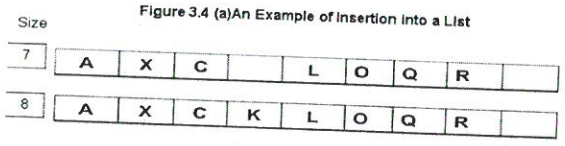


Figure 3.4 (b)An Example of Insertion into a List

Deletion from Lists

What happens if you want to remove an item from a specified position in an existing array? There are two ways in which this can be done:

– Leave *gaps* in the array, i.e. indices that contain no elements, which in practice, means that the array element has to be given a special value to indicate that it is "empty", or

– All the items after the (removed item's) index must be *shifted left* similar to what we did when we wanted to insert – only for insertion we shifted right.

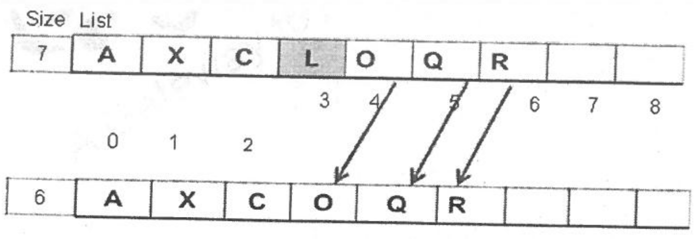


Figure 3.5 (a) Deletion from a List

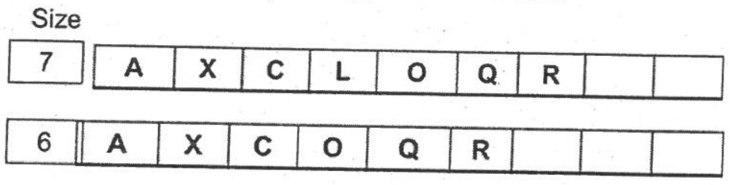


Figure 3.5 (b) Deletion from a List

UNIT-III					
6	a)	Develop algorithm to convert infix expression to postfix expression.	L3	CO3	5 M
	b)	Explain implementation of stack using pointers.	L2	CO1	5 M

Scheme: 6 a) Algorithm/Procedure – 5M

Ans:

Infix expression: *The expression of the form "a operator b" (a + b) i.e., when an operator is in-between every pair of operands.*
Postfix expression: *The expression of the form "a b operator" (ab+) i.e., When every pair of operands is followed by an operator.*

Examples:
Input: *A + B * C + D*
Output: *ABC*+D+*
Input: *((A + B) – C * (D / E)) + F*
Output: *AB+CDE/*-F+*

Algorithm to Convert Infix Expression to Postfix Expression

This algorithm utilizes a stack to manage operators and their precedence during the conversion process.

Input: Infix expression
Output: Postfix expression

1. Initialize: Create an empty stack (opstack) and Create an empty string (postfix).

2. Scan the infix expression from left to right:

- For each character:

   - If operand: Append it to the postfix string.
   - If left parenthesis: Push it onto the opstack.
   - If right parenthesis:
     - Pop operators from the opstack and append them to the postfix string until a left parenthesis is encountered.
     - Pop and discard the left parenthesis.
   - If operator:
     - While the top of the opstack has an operator with higher or equal precedence and is not a left parenthesis, pop the operator and append it to the postfix string.
     - Push the current operator onto the opstack.

3. After scanning the entire infix expression:
   - Pop all remaining operators from the opstack and append them to the postfix string.
4. Return the postfix string.

Additional Notes:

- Operator precedence is crucial. Define a function to compare the precedence of operators.
- Common operators and their precedence: ^ > * / > + -
- Use parentheses to override the default precedence.

Example:

Infix expression: a + b * c - d / e

Postfix expression: abc*+d/e-

6 b) Sophisticated explanation of Stack using linked list - 5M

Ans: Implementing a stack using pointers typically involves using a singly linked list. Each node in the linked list represents an element in the stack, and the top of the stack is represented by a pointer to the first node.

Stack Operations:

push(): Insert a new element into the stack i.e just insert a new element at the beginning of the linked list.

pop(): Return the top element of the Stack i.e simply delete the first element from the linked list.

peek(): Return the top element.

display(): Print all elements in Stack.

Here's how you can implement a stack using pointers in C:

Explanation:

- Node Structure:

Each node contains a data field to store the element and a next pointer to point to the next node in the stack.

- Top Pointer:

A pointer top is used to keep track of the topmost element of the stack. Initially, it is set to NULL to represent an empty stack.

Push Operation:

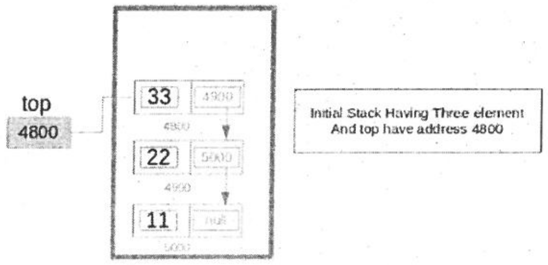
- Create a new node with the given data.
- Set the next pointer of the new node to point to the current top.
- Update the top pointer to point to the new node.

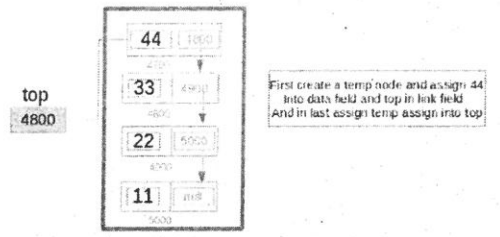
Pop Operation:

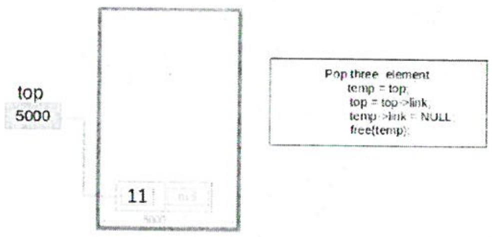
- Check if the stack is empty. If so, return an error.
- Store the top node in a temporary variable.
- Update the top pointer to point to the next node.
- Return the data from the temporary node and free its memory.

Peek Operation:

- Check if the stack is empty. If so, return an error.
- Return the data of the top node without removing it.




top
4800
44
33
4900
4800
22
5000
11
null
5000
First create a temp node and assign 44
into data field and top in link field
And in last assign temp assign into top



```
Pop three  element
temp = top;
top = top->link;
temp->link = NULL;
free(temp);
```

Advantages of using pointers:

- Dynamic memory allocation: The stack can grow or shrink dynamically as needed.
- Efficiency: Push and pop operations have a time complexity of O(1), making them very fast.

		OR			
7	a)	Explain push() and pop() functions of stack data structure with array implementation.	L2	CO1	5 M
	b)	Describe the process of evaluating post fix expression using stack.	L2	CO3	5 M

Scheme: 7 a) push() operation – 3M

pop() operation – 2M

Ans: **Stack** is a **linear data structure** which follows **LIFO** principle.

<u>Implement Stack Operations using Array:</u>

<u>Push Operation in Stack:</u>
Adds an item to the stack. If the stack is full, then it is said to be an **Overflow condition.**

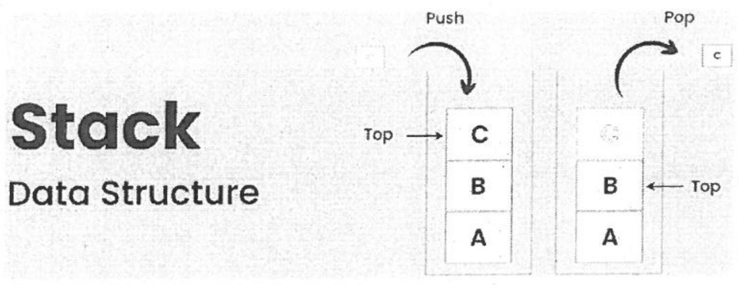
Algorithm for Push Operation:

- *Before pushing the element to the stack, we check if the stack is **full**.*
- *If the stack is full **(top == capacity-1)**, then **Stack Overflows** and we cannot insert the element to the stack.*
- *Otherwise, we increment the value of top by 1 **(top = top + 1)** and the new value is inserted at **top position**.*
- *The elements can be pushed into the stack till we reach the **capacity** of the stack.*

Pop Operation in Stack:
Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an **Underflow condition.**

Algorithm for Pop Operation:

- *Before popping the element from the stack, we check if the stack is* ***empty****.*
- *If the stack is empty (top == -1), then* ***Stack Underflows*** *and we cannot remove any element from the stack.*
- *Otherwise, we store the value at top, decrement the value of top by 1* ***(top = top – 1)*** *and return the stored top value.*



7 b) Sophisticated explanation with an example – 5M

Ans:

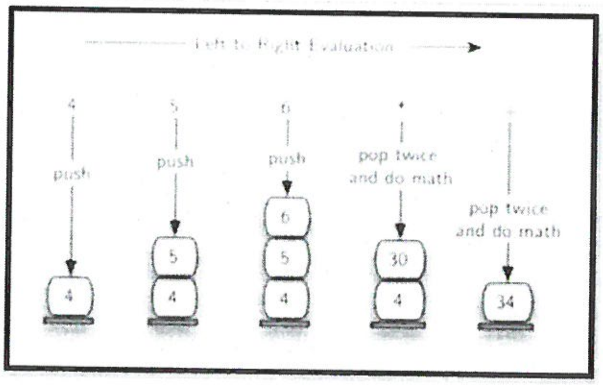
A stack data structure can be used to evaluate postfix expressions. Here's the process:

1. Create an empty stack: To store operands.
2. Iterate through the expression: Left to right.
3. Check the character: If it's an operand or operator.
4. Operands: Push the operand onto the stack.
5. Operators: Pop two elements from the stack, perform the operation, and push the result back onto the stack.
6. End of expression: The top element of the stack is the final answer.

Example for Evaluation of Postfix Expression

Let's see an example to better understand the algorithm:

Expression: 456*+



Step	Input Symbol	Operation	Stack	Calculation
1.	4	Push	4	
2.	5	Push	4,5	
3.	6	Push	4,5,6	
4.	*	Pop(2 elements) & Evaluate	4	5*6=30
5.		Push result(30)	4,30	
6.	+	Pop(2 elements) & Evaluate	Empty	4+30=34
7.		Push result(34)	34	
8.		No-more elements(pop)	Empty	34(Result)

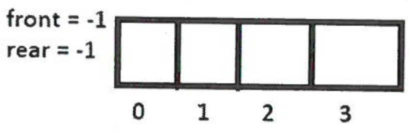
		UNIT-IV			
8	a)	Explain about array implementation of queue.	L2	CO1	5 M
	b)	What is circular queue? What is advantage of circular queue over linear queue? Demonstrate with a scenario.	L2	CO1	5 M

Scheme: 8 a) Sophisticated explanation of queue using array– 5M

Ans: A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle, where the first element added to the queue will be the first one to be removed.

(A)

Initailly the queue is empty

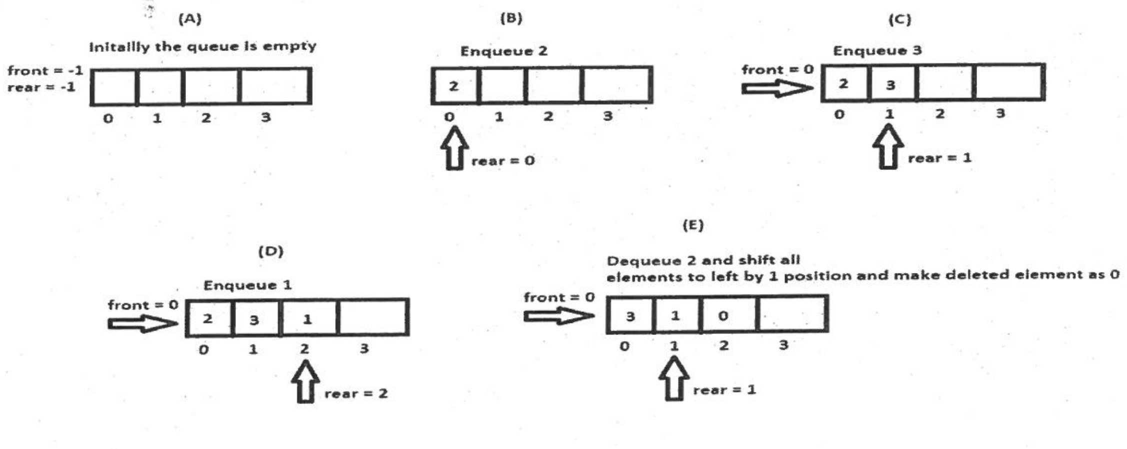


Implementing a queue using an array involves using an array to store the elements of the queue and two pointers (indices) to keep track of the front and rear of the queue. Here's a breakdown of the concept:

Basic Idea:

- Array: An array of fixed size is used to store the queue elements.
- Front Pointer: Points to the index of the first element (front) of the queue.
- Rear Pointer: Points to the index of the last element (rear) of the queue.

  Operations:
- Enqueue (Adding an element):
  - Check if the queue is full (rear = array size - 1). If full, overflow occurs.
  - Increment the rear pointer.
  - Insert the new element at the position pointed to by the rear pointer.
- Dequeue (Removing an element):
  - Check if the queue is empty (front > rear). If empty, underflow occurs.
  - Retrieve the element at the position pointed to by the front pointer.
  - Increment the front pointer.
  - Return the retrieved element.
- Peek (Accessing the front element):
  - Check if the queue is empty.
  - Return the element at the position pointed to by the front pointer without removing it.
- isEmpty():
  - Check if front > rear. If true, the queue is empty.
- isFull():
  - Check if rear = array size - 1. If true, the queue is full.



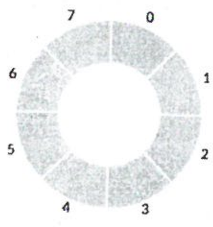
Implementing queue using array

8 b) Definition – 1M

Advantages – 2M

Scenario - 2M

Ans: A Circular Queue is an extended version of a normal queue where the last element of the queue is connected to the first element of the queue.



Circular queue representation

Advantages of Circular Queue

1. **Efficient Space Utilization**: Unlike a linear queue implemented with arrays, a circular queue can reuse the space of dequeued elements, preventing the need for shifting elements and avoiding overflow when there is still available space.
2. **Constant Time Operations**: Both enqueue and dequeue operations can be performed in O(1) time complexity.
3. **Fixed Size**: It can be implemented with a fixed size, making it predictable in terms of memory usage.

Example

Let's illustrate a circular queue with a simple example:

Initial Setup

- Size of the circular queue: 5
- The queue is empty initially.

Operations

1. **Enqueue 1, 2, 3**

Front: 0, Rear: 2

Queue: [1, 2, 3, _, _]

2. **Dequeue**

Dequeued element: 1

Front: 1, Rear: 2

Queue: [_, 2, 3, _, _]

3. **Enqueue 4, 5**

Front: 1, Rear: 4

Queue: [_, 2, 3, 4, 5]

4. **Dequeue**

Dequeued element: 2

Front: 2, Rear: 4

Queue: [_, _, 3, 4, 5]

5. **Enqueue 6**

Front: 2, Rear: 0 (wrap around)

Queue: [6, _, 3, 4, 5]

6. **Enqueue 7**

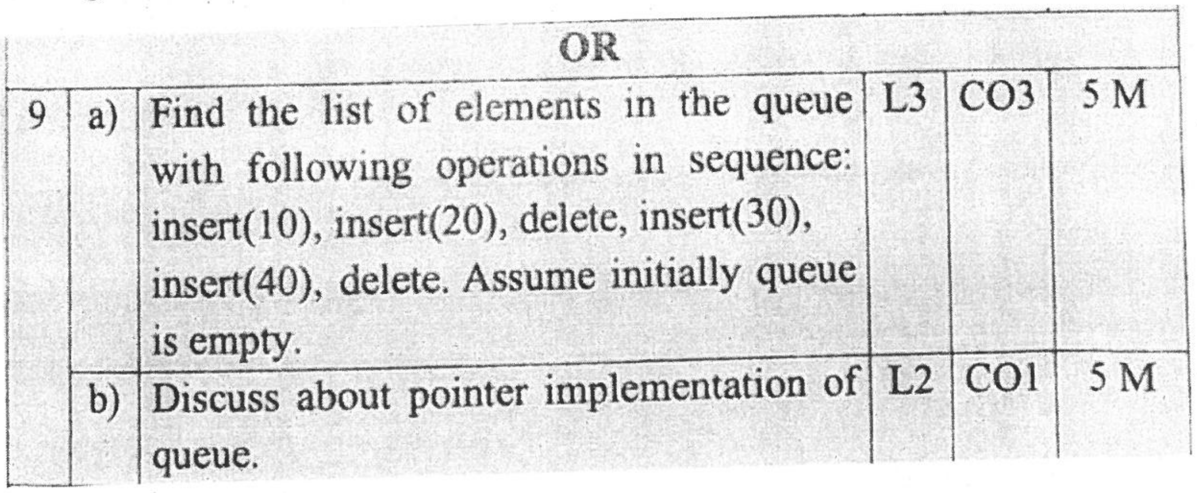
Front: 2, Rear: 1 (wrap around)

Queue: [6, 7, 3, 4, 5]

Explanation

- When the queue reaches the end of the array and there is still available space at the beginning (due to dequeued elements), the rear pointer wraps around to the beginning of the array.
- This prevents wasted space and makes full use of the array, ensuring efficient space utilization.

A circular queue is particularly useful in **scenarios** where the size of the queue is fixed, and efficient utilization of the available space is crucial. It avoids the need for shifting elements, making enqueue and dequeue operations consistently efficient.



OR					
9	a)	Find the list of elements in the queue with following operations in sequence: insert(10), insert(20), delete, insert(30), insert(40), delete. Assume initially queue is empty.	L3	CO3	5 M
	b)	Discuss about pointer implementation of queue.	L2	CO1	5 M

Scheme: 9 a) Showing Queue contents for every operation with explanation – 5M

Ans: A queue is a **linear data structure that follows the First-In-First-Out (FIFO) principle**. It operates like a line where elements are added at one end (**rear**) and removed from the other end (**front**).

Let's perform the given sequence of operations on an initially empty queue and find the list of elements after each operation.

0. Initial State

- Queue: []

1. **insert(10)**:
   - Queue: [10]
2. **insert (20)**:
   - Queue: [10, 20]
3. **delete()**:
   - Removed element: 10
   - Queue: [20]
4. **insert(30)**:
   - Queue: [20, 30]
5. **insert(40)**:
   - Queue: [20, 30, 40]
6. **delete()**:
   - Removed element: 20
   - Queue: [30, 40]

Final List of Elements in the Queue

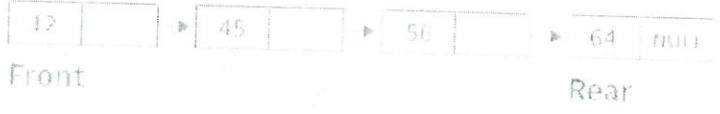
- Queue: [30, 40]

So, after performing the sequence of operations, the final list of elements in the queue is **[30, 40]**.

9 b) Sophisticated explanation of queue using linked list – 5M

Ans:

A queue is a linear data structure that follows the First In, First Out (FIFO) principle, which means that the element which is inserted first in the queue will be the first one to be removed from the queue. A good example of a queue is a queue of customers purchasing a train ticket, where the customer who comes first will be served first. A linked queue is shown here:



Operation on Linked Queue

Each node of a linked queue consists of two fields: data and next (storing address of next node). The data field of each node contains the assigned value, and the next points to the node containing the next item in the queue.
A linked queue consists of two pointers, i.e., the front pointer and the rear pointer. The front pointer stores the address of the first element of the queue, and the rear pointer stores the address of the last element of the queue.
Insertion is performed at the rear end, whereas deletion is performed at the front end of the queue. If front and rear both point to NULL, it signifies that the queue is empty.

The two main operations performed on the linked queue are:

- Insertion
- Deletion

Insertion

Insert operation or insertion on a linked queue adds an element to the end of the queue. The new element which is added becomes the last element of the queue.

Algorithm to perform Insertion on a linked queue:

1. Create a new node pointer.
   ptr = (struct node *) malloc (sizeof(struct node));

2. Now, two conditions arise, i.e., either the queue is empty, or the queue contains at least one element.

3. If the queue is empty, then the new node added will be both front and rear, and the next pointer of front and rear will point to NULL.

```
*ptr - > data = val;
if (front == NULL)
{
front = ptr;
rear = ptr;
front - > next = NULL;
 rear - > next = NULL;
}
```

4. If the queue contains at least one element, then the condition front == NULL becomes false. So, make the next pointer of rear point to new node ptr and point the rear pointer to the newly created node ptr

```
rear -> next = ptr;
rear = ptr;
```

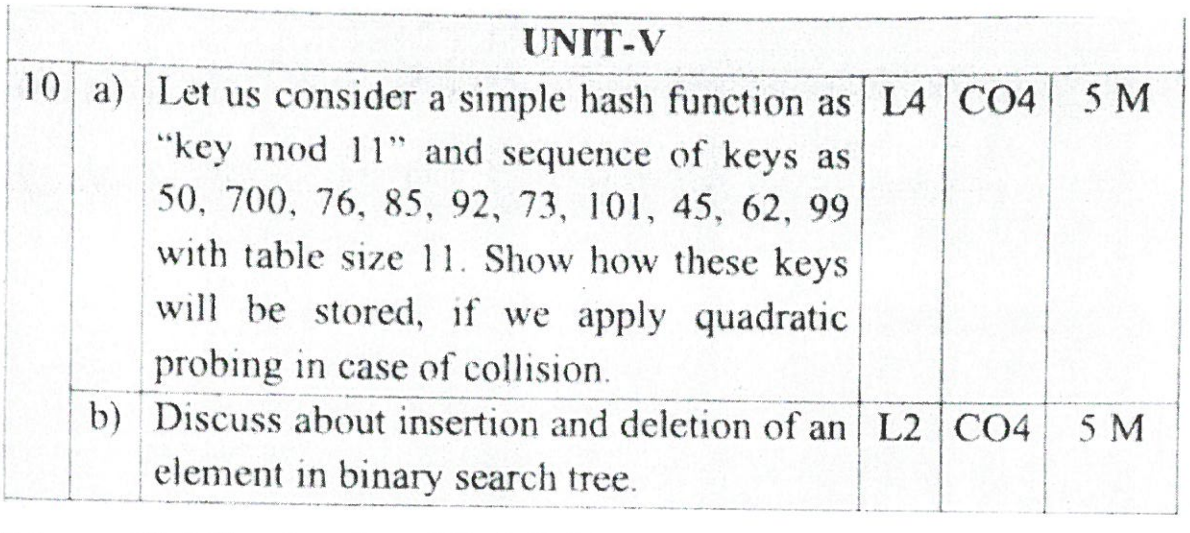
5. Hence, a new node(element) is added to the queue.

Deletion

Deletion or delete operation on a linked queue removes the element which was first inserted in the queue, i.e., always the first element of the queue is removed.
Steps to perform Deletion on a linked queue:

1. Check if the queue is empty or not.
2. If the queue is empty, i.e., front==NULL, so we just print 'underflow' on the screen and exit.
3. If the queue is not empty, delete the element at which the front pointer is pointing. For deleting a node, copy the node which is pointed by the front pointer into the pointer ptr and make the front pointer point to the front's next node and free the node pointed by the node ptr. This can be done using the following statement:

```
*ptr = front;
front = front -> next;
free(ptr);
```



UNIT-V					
10	a)	Let us consider a simple hash function as "key mod 11" and sequence of keys as 50, 700, 76, 85, 92, 73, 101, 45, 62, 99 with table size 11. Show how these keys will be stored, if we apply quadratic probing in case of collision.	L4	CO4	5 M
	b)	Discuss about insertion and deletion of an element in binary search tree.	L2	CO4	5 M

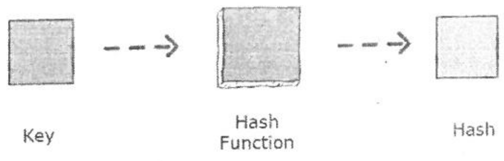
Scheme: 10 a) Construction of Hash Table with quadratic probing – 5M

Ans:

Hashing in data structures is a technique used to efficiently store and retrieve data.

Hashing is the process of converting a given key into another value. A **hash function** is used to generate the new value according to a mathematical algorithm. The result of a hash function is known as a **hash value** or simply, a **hash**.

Let us consider a simple hash function as "key mod 11" and sequence of keys as 50, 700, 76, 85, 92, 73, 101, 45, 62, 99 with table size 11. show how these keys will be stored, if we apply quadratic probing in case of collision.



Quadratic probing is a collision resolution technique in open addressing hash tables. When a collision occurs, it searches for the next available slot using a quadratic function of the form:

$$\text{hash}(\text{key},i)=(h(\text{key})+c_1\cdot i+c_2\cdot i^2) \bmod m$$

where:

- h(key) is the original hash function.
- *i* is the probing attempt (0, 1, 2, ...).
- *c1* and *c2* are constants (typically set to 1).
- *m* is the table size.

Given:

- Hash function: h(key)=key mod 11
- Table size: 11
- Sequence of keys: 50, 700, 76, 85, 92, 73, 101, 45, 62, 99

Quadratic Probing Steps

Let's insert each key step-by-step:

1. **Insert 50**:

   h(50)=50 mod 11=6

   - Slot 6 is empty. Place 50 at index 6.
   - Table: [_, _, _, _, _, _, 50, _, _, _, _]
2. **Insert 700**:

   h(700)=700 mod 11=7

   - Slot 7 is empty. Place 700 at index 7.
   - Table: [_, _, _, _, _, _, 50, 700, _, _, _]
3. **Insert 76**:

   h(76)=76 mod 11=10

   - Slot 10 is empty. Place 76 at index 10.
   - Table: [_, _, _, _, _, _, 50, 700, _, _, 76]
4. **Insert 85**:

$h(85) = 85 \bmod 11 = 8$

  - Slot 8 is empty. Place 85 at index 8.
  - Table: [_, _, _, _, _, _, 50, 700, 85, _, 76]

5. **Insert 92**:

   $h(92) = 92 \bmod 11 = 4$

   - Slot 4 is empty. Place 92 at index 4.
   - Table: [_, _, _, _, 92, _, 50, 700, 85, _, 76]

6. **Insert 73**:

   $h(73) = 73 \bmod 11 = 7$

   - Slot 7 is occupied (collision).
   - Quadratic probing: $(7+i^2) \bmod 11$
     - $i=1$: $(7+1^2) \bmod 11 = 8$(collision)
     - $i=2$: $(7+2^2) \bmod 11 = 11 \bmod 11 = 0$(empty)
   - Place 73 at index 0.
   - Table: [73, _, _, _, 92, _, 50, 700, 85, _, 76]

7. **Insert 101**:

   $h(101) = 101 \bmod 11 = 2$

   - Slot 2 is empty. Place 101 at index 2.
   - Table: [73, _, 101, _, 92, _, 50, 700, 85, _, 76]

8. **Insert 45**:

   $h(45) = 45 \bmod 11 = 1$

   - Slot 1 is empty. Place 45 at index 1.
   - Table: [73, 45, 101, _, 92, _, 50, 700, 85, _, 76]

9. **Insert 62**:

   $h(62) = 62 \bmod 11 = 7$

   - Slot 7 is occupied (collision).
   - Quadratic probing: $(7+i^2) \bmod 11$
     - $i=1$: $(7+1^2) \bmod 11 = 8$ (collision)
     - $i=2$: $(7+2^2) \bmod 11 = 11 \bmod 11 = 0$ (collision)
     - $i=3$: $(7+3^2) \bmod 11 = 16 \bmod 11 = 5$ (empty)
   - Place 62 at index 5.
   - Table: [73, 45, 101, _, 92, 62, 50, 700, 85, _, 76]

10. **Insert 99**:

    $h(99) = 99 \bmod 11 = 0$

    - Slot 0 is occupied (collision).

- Quadratic probing: $(0+i^2)\bmod 11$
    - $i=1$: $(0+1^2)\bmod 11=1$ (collision)
    - $i=2$: $(0+2^2)\bmod 11=4$ (collision)
    - $i=3$: $(0+3^2)\bmod 11=9$ (empty)
- Place 99 at index 9.
- Table: [73, 45, 101, _, 92, 62, 50, 700, 85, 99, 76]

Final Hash Table

The final hash table after inserting all the keys using quadratic probing is:

```
Index: 0   1   2   3  4   5   6   7    8   9  10
Keys: 73  45  101  _  92  62  50  700  85  99  76
```

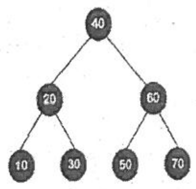
10 b) Insertion operation – 2M **Deletion operation – 3M**

Ans:

A Binary Search Tree (BST) is a hierarchical data structure in which each node has at most two children, referred to as the left child and the right child.

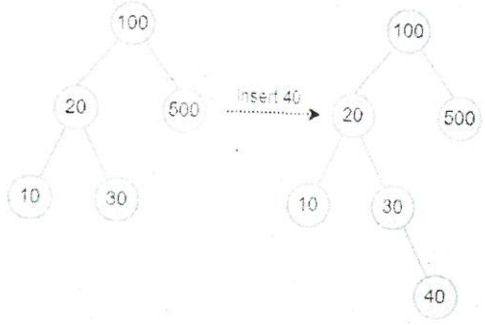
For every node in the BST:

- All nodes in its left subtree have values less than the node's value.
- All nodes in its right subtree have values greater than the node's value.
- This property holds recursively for all subtrees within the BST.



Insertion:

- Start at the root: Begin the insertion process from the root node of the BST.
- Comparison: Compare the value to be inserted with the value of the current node.
- Traversal:
    - If the value is less than the current node's value, move to the left child.
    - If the value is greater than the current node's value, move to the right child.
    - Repeat step 2 and 3 until you reach a leaf node (a node with no children).
- Insertion: Insert the new node as a child of the leaf node you reached in the previous step.

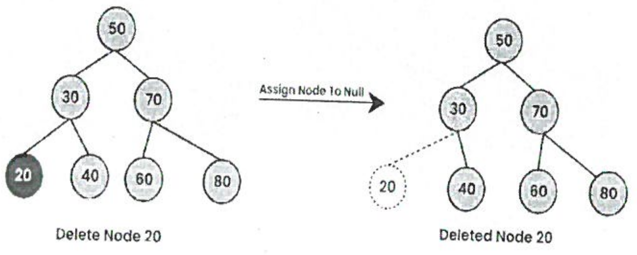


Deletion:

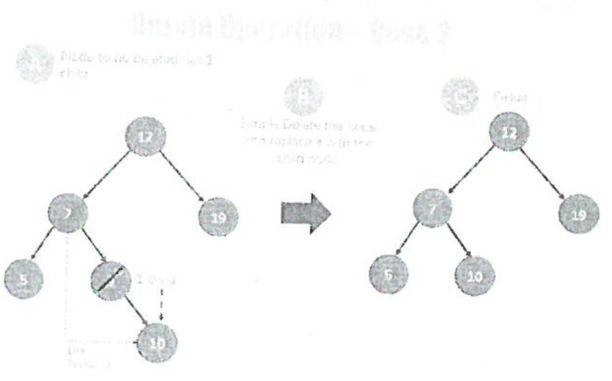
Deleting a node from a BST is slightly more complex than insertion, as it requires maintaining the BST property (left child < parent < right child). There are three cases to consider:

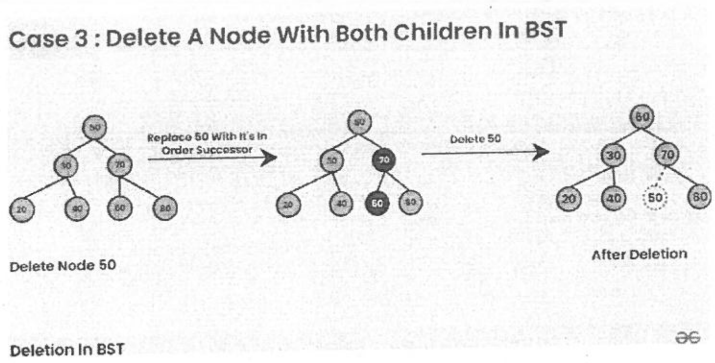
- Deleting a leaf node:
  - Simply remove the node from the tree.
- Deleting a node with one child:
  - Replace the node to be deleted with its child.
- Deleting a node with two children:
  - Find the inorder successor of the node (the smallest value in the right subtree).
  - Replace the node to be deleted with its inorder successor.
  - Delete the inorder successor (which will always be a leaf node or a node with one child).

Case 1 : Delete A Leaf Node In BST



Deletion In BST




Case 3 : Delete A Node With Both Children In BST
Replace 50 With It's In Order Successor
Delete 50
Delete Node 50
After Deletion
60
30
70
20
40
50
80
Deletion In BST

Time Complexity:

Insertion:
The time complexity of insertion in a BST is O(h), where h is the height of the tree. In the worst case, where the tree is skewed, the height can be equal to the number of nodes (n), resulting in O(n) complexity. However, in a balanced BST, the height is O(log n), leading to O(log n) insertion time.

Deletion:
Similar to insertion, the time complexity of deletion in a BST is also O(h). In a balanced BST, this becomes O(log n).

OR					
11	a)	Define Binary search tree. Construct binary search tree with following keys: 55,45,65,40,60,70,66,99,2,34	L3	CO4	5 M
	b)	Assume a table has 8 slots. Using chaining, insert the following elements into the hash table. 56,66,18,72,43,65,6,17,10,5,64,16,71, and 15 are inserted in the order. Consider Hash function: h(k) = k mod m, where m=8.	L4	CO4	5 M

Scheme: 11 a) Definition – 1M **Construction of BST – 4M**

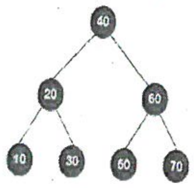
Ans:

A Binary Search Tree (BST) is a hierarchical data structure in which each node has at most two children, referred to as the left child and the right child.

For every node in the BST:

- All nodes in its left subtree have values less than the node's value.
- All nodes in its right subtree have values greater than the node's value.

- This property holds recursively for all subtrees within the BST.



To construct a Binary Search Tree (BST) with the given keys in the order: 55, 45, 65, 40, 60, 70, 66, 99, 2, 34, we follow the BST property: for each node, the left child is less than the node and the right child is greater.

Let's insert each key step-by-step:

Step-by-Step Insertion

1. **Insert 55**:
   - Root: 55
2. **Insert 45**:
   - 45 < 55: Insert 45 as the left child of 55.

```
   55
  /
45
```

3. **Insert 65**:

- 65 > 55: Insert 65 as the right child of 55.

```
   55
  /  \
45    65
```

4. **Insert 40**:

- 40 < 55: Move left to 45.
- 40 < 45: Insert 40 as the left child of 45.

```
      55
     /  \
   45    65
  /
40
```

5. **Insert 60**:

- 60 > 55: Move right to 65.
- 60 < 65: Insert 60 as the left child of 65.

```
      55
     /  \
   45    65
  /      /
40     60
```

6. **Insert 70**:

- 70 > 55: Move right to 65.
- 70 > 65: Insert 70 as the right child of 65.

```
      55
     /  \
   45    65
  /     /  \
40    60    70
```

7. **Insert 66**:

- 66 > 55: Move right to 65.
- 66 > 65: Move right to 70.
- 66 < 70: Insert 66 as the left child of 70.

```
      55
     /  \
   45    65
  /     /  \
40    60    70
           /
         66
```

8. **Insert 99**:

- 99 > 55: Move right to 65.
- 99 > 65: Move right to 70.
- 99 > 70: Insert 99 as the right child of 70.

```
      55
     /  \
   45    65
  /     /  \
40    60    70
           /  \
         66    99
```

9. **Insert 2**:

- 2 < 55: Move left to 45.
- 2 < 45: Move left to 40.
- 2 < 40: Insert 2 as the left child of 40.

```
       55
      /  \
    45    65
   /     /  \
  40   60    70
 /          /  \
2         66    99
```

10. **Insert 34**

- 34 < 55: Move left to 45.
- 34 < 45: Move left to 40.
- 34 < 40: Move left to 2.
- 34 > 2: Insert 34 as the right child of 2.

```
        55
       /  \
     45    65
     /     / \
    40   60   70
    /         / \
   2        66   99
    \
    34
```

11 b) Construction of Hash Table using chaining – 5M

Ans: Hashing in the data structure is a technique of mapping a large chunk of data into small tables using a hashing function.

Hashing is the process of converting a given key into another value. A **hash function** is used to generate the new value according to a mathematical algorithm. The result of a hash function is known as a **hash value** or simply, a **hash**.

To insert the elements 56, 66, 18, 72, 43, 65, 6, 17, 10, 5, 64, 16, 71, and 15 into a hash table using chaining with the hash function h(k)=k mod 8, we first compute the hash values for each element and then place them into the corresponding bucket.

Here's the step-by-step process:

1. **Initialize the hash table:** Since m=8, we create an array with 8 buckets (index 0 to 7).

   Hash table: [[],[],[],[],[],[],[],[]]

2. **Compute hash values and insert elements:**
   - For 56: h(56)=56 mod 8=0
     Bucket 0: [56]
   - For 66: h(66)=66 mod 8=2
     Bucket 2: [66]
   - For 18: h(18)=18 mod 8=2
     Bucket 2: [66,18]
   - For 72: h(72)=72 mod 8=0
     Bucket 0: [56,72]
   - For 43: h(43)=43 mod 8=3
     Bucket 3: [43]
   - For 65: h(65)=65 mod 8=1
     Bucket 1: [65]
   - For 6: h(6)=6 mod 8=6
     Bucket 6: [6]

   - For 17: h(17)=17 mod 8=1
     Bucket 1: [65,17]
   - For 10: h(10)=10 mod 8=2
     Bucket 2: [66,18,10]
   - For 5: h(5)=5 mod 8=5
     Bucket 5: [5]
   - For 64: h(64)=64 mod 8=0
     Bucket 0: [56,72,64]
   - For 16: h(16)=16 mod 8=0
     Bucket 0: [56,72,64,16]
   - For 71: h(71)=71 mod 8=7
     Bucket 7: [71]
   - For 15: h(15)=15 mod 8=7
     Bucket 7: [71,15]

3. **Final hash table with chaining:**

Hash table:

Index	Elements
0	[56, 72, 64, 16]
1	[65, 17]
2	[66, 18, 10]
3	[43]
4	[]
5	[5]
6	[6]
7	[71, 15]