BHARATI VIDYAPEETH DEEMED UNIVERSITY
COLLEGE OF ENGINEERING, PUNE - 43

DEPARTMENT OF COMPUTER ENGINEERING

Lab Manual

Data Structures and Algorithmic Thinking

B.Tech Computer Sem III (2020 Course)

VISION OF THE INSTITUTE

"To be World Class Institute for Social Transformation through Dynamic Education"

MISSION OF THE INSTITUTE

- To provide quality technical education with advanced equipment, qualified faculty members, infrastructure to meet needs of profession and society.
- To provide an environment conductive to innovation, creativity, research and entrepreneurial leadership.
- To practice and promote professional ethics, transparency and accountability for social community, economic and environmental conditions.

VISION OF THE DEPARTMENT

"To pursue and excel in the endeavor for creating globally recognized computer engineers through quality education."

Mission of the Department

- To impart engineering knowledge and skills confirming to a dynamic curriculum.
- To develop professional, entrepreneurial & research competencies encompassing continuous intellectual growth.
- To produce qualified graduates exhibiting societal and ethical responsibilities in working environment

PROGRAM EDUCATIONAL OBJECTIVES(PEOs):

1. Demonstrate technical and professional competencies by applying engineering fundamentals, computing principles and technologies.
2. Learn, Practice, and grow as skilled professionals/ entrepreneur/researchers adapting to the evolving computing landscape.
3. Demonstrate professional attitude, ethics, understanding of social context and interpersonal skills leading to a successful career.

ROGRAM SPECIFIC OUTCOMES(PSO)s:

PSO 1: To design, develop and implement computer programs on hardware towards solving problems.
PSO 2: To employ expertise and ethical practise through continuing intellectual growth and adapting to the working environment.

PROGRAMME OUTCOMES(POs):

Upon completion of the course the graduate engineers will be able to:

1. Apply the knowledge of mathematics, science, engineering fundamentals, and computing for the solution of complex engineering problems.
2. Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using computer engineering foundations, principles, and technologies.
3. Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for public health and safety, and cultural, societal, and environmental considerations.
4. Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.
6. Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal, and cultural issues, and the consequent responsibilities relevant to the professional engineering practice.
7. Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and the need for sustainable development.
8. Apply ethical principles while committed to professional responsibilities and norms of the engineering practice.
9. Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings
10. Communicate effectively on complex engineering activities with the engineering community and with the society at large, such as being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions. 11. Apply the engineering and management principles to one's work, as a member and leader in a team.
12. Recognise the need for and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Course Name: - Data Structures and Algorithmic Thinking

Hours Per Week: 3 Hrs.

Course Outcomes:

1. Demonstrate the use of ADTs,
2. Develop code to illustrate sorting and searching algorithms.
3. Comprehend the real time problem.
4. Practise and apply Iterative Thinking
5. Practise and apply Recursive Thinking
6. Apply algorithms and data structures in various real-life software problems

HOW OUTCOMES ARE ASSESSED?

Course Outcome	Assignment Number	Level	Proficiency evaluated by
Demonstrate the use of ADTs,	**1,2,3,4,5,7,9**	**3,3,3,3,3,3,3**	Performing Practical and reporting results
Develop code to illustrate sorting and searching algorithms.	**6**	**3**	Problem definition &Performing Practical and reporting results
Comprehend the real time problem.	**6**	**2**	Performing experiments and reporting results
Practice and Apply Iterative Thinking	**8,9,10**	**3,3,3**	Performing experiments and reporting results
Practice and Apply Recursive Thinking.	**9,10**	**3,3**	Performing experiments and reporting results

Apply algorithms and data structures in various real-life software problems	**8,10**	**3,3**	Performing experiments and reporting results

CO-PO mapping

CO Statements	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12	PSO 1	PSO 2
CO1	3		3					2		3		2	3	
CO2	2	3	3		3			2		3		2	3	
CO3	2		3		3			2		3		2		3
CO4	2	3	3	3	3			2		3		2	2	3
CO5	3	3	3	3	3			2		3		2	1	
CO6	1	3		1	1			2		3		2		2

Guidelines for Student's Lab Journal

- The laboratory assignments are to be submitted by student in the form of journal. The Journal consists of prologue, Certificate, table of contents, and handwritten write-up of each assignment (Title, Objectives, Problem Statement, Outcomes, software & Hardware requirements, Date of Completion, Assessment grade/marks and assessor's sign, Theory-Concept, algorithms, printouts of the code written using coding standards, sample test cases etc.
- Practical Examination will be based on the term work submitted by the student in the form of journal
- Candidate is expected to know the theory involved in the experiment
- The practical examination should be conducted if the journal of the candidate is completed in all respects and certified by concerned faculty and head of the department
- All the assignment mentioned in the syllabus must be conducted

List Of Assignments

1. Create a graph and perform Depth First Search and Breadth First Search.
2. Create a record containing the cities and distances between two cities. Find the optimal path between any two cities. The resulting path will contain all the cities taken into consideration
3. Write a program to search a number in a binary search tree.
4. Write a program to display a mirror image of a binary tree.
5. Write a program to perform the operations of insertion and deletion on the Heap.

6. a) Write a menu driver program to compute the factorial and display a Fibonacci series for the number entered by the user. (Use recursion)

   b) Write a program to solve the tower of Hanoi problem.

7. Apply backtracking to solve the 4 Queens problem.
8. Write a program to sort the given list of elements using Heap sort.
9. Apply the following sorting techniques on the set of Integers.
   i. Quick Sort
   ii. Merge Sort
10. Write a program to apply search using hashing techniques on the student database of B.Tech Computer Engineering.

1. Create a graph and perform Depth First Search and Breadth First Search

AIM: The aim of this lab is to understand the concepts of graph theory and implement Breadth First Search (BFS) and Depth First Search (DFS) algorithms to traverse a graph.

OBJECTIVES:

1. To understand the concepts of graphs, vertices, and edges.
2. To learn how to represent a graph using a dictionary.
3. To implement BFS and DFS algorithms to traverse a graph.
4. To analyze the time and space complexity of BFS and DFS algorithms.

THEORY:

1. Introduction to Graph Theory: In graph theory, a graph is a non-linear data structure that consists of a set of vertices and a set of edges that connect these vertices. Graphs are used to represent many real-world scenarios, such as social networks, transportation networks, and computer networks.
2. Representation of Graph: A graph can be represented mathematically as G = (V, E), where V is the set of vertices and E is the set of edges. An edge is a connection between two vertices, and it can be either directed or undirected. In a directed graph, the edges have a direction, whereas in an undirected graph, the edges have no direction.
3. Breadth First Search (BFS): Breadth First Search (BFS) is a popular algorithm used to traverse a graph. BFS starts at the root node and explores all the neighbors at the current depth level before moving to the next level. This means that it visits all the vertices at a given distance from the starting vertex before moving on to the vertices at the next distance. BFS is typically implemented using a queue data structure.
4. Depth First Search (DFS): Depth First Search (DFS) is another popular algorithm used to traverse a graph. DFS explores as far as possible along each branch before backtracking. This means that it visits all the vertices in a branch of the graph before moving on to another

branch. DFS is typically implemented using a stack data structure or recursion.

5. Applications of BFS and DFS: BFS and DFS algorithms are used to visit all the vertices in a graph and can be used to solve many graph problems, such as finding the shortest path between two vertices or determining whether a graph is connected.
6. Time and Space Complexity: The time complexity of BFS and DFS algorithms is O(V+E), where V is the number of vertices and E is the number of edges in the graph. The space complexity of BFS and DFS algorithms is O(V), where V is the number of vertices in the graph.

ALGORITHM:

1. Creating a Graph: To create a graph, we can use a dictionary to represent the graph, where the keys are the vertices and the values are the edges. We can define a Graph class that has two methods: add_vertex and add_edge.

add_vertex: This method takes a vertex as an argument and adds it to the graph as a key with an empty list as the value.

add_edge: This method takes two vertices as arguments and adds a connection between them by appending one to the other's list.

2. Implementing BFS: BFS can be implemented using a queue data structure. The steps to implement BFS are as follows:

bfs: This function takes a starting vertex as an argument.

- Initialize a queue and enqueue the starting vertex.
- Initialize a visited set and add the starting vertex to it.
- While the queue is not empty, dequeue the first vertex and print it.
- For each of the vertex's neighbors, if it hasn't been visited yet, add it to the queue and visited set.

3. Implementing DFS: DFS can be implemented using a stack data structure or recursion. The steps to implement DFS are as follows:

dfs: This function takes a starting vertex and a visited set as arguments.

- Add the starting vertex to the visited set and print it.

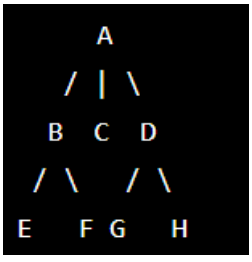
- For each of the vertex's neighbors, if it hasn't been visited yet, recursively call dfs on that neighbor with the visited set.

Time complexity:

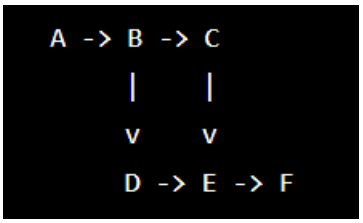
The time complexity of BFS and DFS is O(V + E), where V is the number of vertices and E is the number of edges in the graph. This is because both algorithms visit each vertex and edge once. However, the space complexity of BFS is higher than that of DFS, as BFS needs to keep track of all the vertices at a given distance from the starting vertex.

TASK:

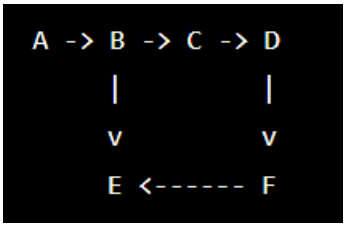
1. Create a graph with 5 nodes and the following edges:
   - Node 1 is connected to nodes 2, 3, and 4
   - Node 2 is connected to nodes 3 and 5
   - Node 3 is connected to node 5
   - Node 4 is connected to node 5
   - Node 5 has no outgoing edges
2. Perform a Breadth First Search on the graph starting from node 1.
3. Perform a Depth First Search on the graph starting from node 1.
4. Consider the following undirected graph. Perform a BFS starting from vertex A. Write down the order in which vertices are visited.



5. Consider the following directed graph. Perform a DFS starting from vertex A. Write down the order in which vertices are visited.



6. Consider the following directed graph. Perform a DFS starting from vertex C. Write down the order in which vertices are visited.



Practice Assignment Questions:

1. Implement the Graph class and add the vertices A, B, C, D, E, and F with the following connections: A-B, A-D, B-C, C-E, D-E, and E-F.
2. Visualize the graph using any suitable tool.
3. Implement the BFS algorithm starting from vertex A.
4. Implement the DFS algorithm starting from vertex A.
5. What is the time complexity of BFS and DFS? Why?

2.Create a record containing the cities and distances between two cities. Find the optimal path between any two cities. The resulting path will contain all the cities taken into consideration

AIM: The aim of this lab is to understand how to create a record containing the cities and distances between two cities, and to implement an algorithm to find the optimal path between any two cities.

OBJECTIVES:

- To learn about graphs and how they can be used to represent cities and distances.
- To learn how to create a record of cities and distances.
- To implement the Dijkstra's algorithm to find the optimal path between any two cities.
- To understand the time and space complexity of the Dijkstra's algorithm.

THEORY:

1. Representation of Cities and Distances: We can represent the cities and distances between them as a weighted graph, where the vertices represent the cities and the edges represent the distances between them. The weight of each edge represents the distance between two cities.
2. Adjacency Matrix: An adjacency matrix is a square matrix used to represent a graph. Each element in the matrix represents an edge in the graph. If there is an edge between vertices i and j, then the value of the element at row i and column j is 1. Otherwise, it is 0. If the graph is weighted, the value of the element at row i and column j is the weight of the edge between vertices i and j.
3. Dijkstra's Algorithm: Dijkstra's algorithm is a popular algorithm used to find the shortest path between two vertices in a weighted graph. It works by assigning a tentative distance to every vertex and then iteratively selecting the vertex with the

smallest tentative distance and updating its neighbors. This process is repeated until the destination vertex is reached. Dijkstra's algorithm is typically implemented using a priority queue data structure.

4. Time and Space Complexity: The time complexity of Dijkstra's algorithm is O(E log V), where E is the number of edges and V is the number of vertices. This is because each edge is processed only once and each vertex is added to the priority queue once. The priority queue takes O(log V) time to insert and delete elements. The space complexity of Dijkstra's algorithm is O(V), where V is the number of vertices. This is because we need to store the distances of each vertex from the source vertex.

ALGORITHM:

Creating a Record of Cities and Distances:

1. Create a list of all the cities that need to be considered.
2. Create a matrix to store the distances between all pairs of cities.
3. For each pair of cities, enter the distance between them into the matrix.

Finding the Optimal Path between Two Cities:

1. Initialize a priority queue Q and a distance array dist.
2. Set the distance of the source vertex to 0 and the distance of all other vertices to infinity.
3. Insert all vertices into the priority queue Q.
4. While Q is not empty, remove the vertex with the smallest distance from Q.
5. For each neighbor of the current vertex, calculate the distance from the source vertex through the current vertex.
6. If the calculated distance is less than the tentative distance stored in the distance array, update the distance array.
7. Repeat steps 4-6 until the destination vertex is reached.

DIJKSTRA'S Algorithm:

Input: A weighted graph G = (V, E) and a source vertex s.

Output: A shortest path from s to every other vertex in G.

1. Initialize a priority queue Q and a distance array dist.
2. Set the distance of the source vertex s to 0 and the distance of all other vertices to infinity.
3. Insert all vertices into the priority queue Q.
4. While Q is not empty, remove the vertex u with the smallest distance from Q.
5. For each neighbor v of u, calculate the distance from s to v through u.
6. If the calculated distance is less than the tentative distance stored in dist[v], update dist[v].
7. If dist[v] was updated, update the priority of v in Q to the new value of dist[v].
8. Repeat steps 4-7 until Q is empty.

At the end of the algorithm, the distance array dist will contain the shortest distance from the source vertex s to every other vertex in the graph, and the shortest path from s to any other vertex can be reconstructed by following the edges with the smallest weights from s to the destination vertex.

TASK:

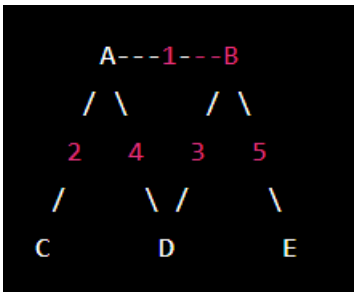
1. Create a record of the following cities and their distances:
   - City A is 0 km away from City B, 3 km away from City C, and 5 km away from City D
   - City B is 3 km away from City A, 2 km away from City C, and 7 km away from City D
   - City C is 5 km away from City A, 2 km away from City B, and 4 km away from City D
   - City D is 7 km away from City B, 4 km away from City C, and 0 km away from City E
   - City E is 10 km away from City D
2. Find the optimal path from City A to City E

3. Suppose you have a graph with the following adjacency matrix:

	A	B	C	D	E
A	0	4	0	2	0
B	4	0	3	0	0
C	0	3	0	0	5
D	2	0	0	0	1
E	0	0	5	1	0

Starting at vertex A, use Dijkstra's algorithm to find the shortest paths to all other vertices.

4. Consider the following graph, starting at vertex A, find the shortest path to all other vertices.



5. Suppose you have a weighted directed graph with the following adjacency matrix:

	A	B	C	D
A	0	10	0	5
B	0	0	1	2
C	0	3	0	7
D	0	0	0	0

Starting at vertex A, use Dijkstra's algorithm to find the shortest paths to all other vertices.

Practice Assignment Questions:

1. What is a graph?
2. How can we represent cities and distances using a graph?
3. What is Dijkstra's algorithm?
4. What is the time complexity of Dijkstra's algorithm?
5. How can we find the optimal path between any two cities using Dijkstra's algorithm?
6. What data structure is typically used to implement Dijkstra's algorithm?
7. What is the space complexity of Dijkstra's algorithm?
8. How can we create a record of cities and distances?
9. Can Dijkstra's algorithm be used to find the shortest path in an unweighted graph? Why or why not?
10. How can we improve the performance of Dijkstra's algorithm on large graphs?

3.Write a program to find a number in a binary tree

AIM: The aim of this lab is to write a program to find a number in a binary search tree.

OBJECTIVES:

- Understand the concept of binary search tree.
- Learn how to search for a number in a binary search tree.
- Implement the program to find a number in a binary search tree using recursion.

THEORY:

1. A binary search tree is a data structure in which each node has at most two children, referred to as the left and right subtrees. The left subtree contains nodes with values less than the node's value, while the right subtree contains nodes with values greater than the node's value. This property allows for efficient searching of elements within the binary search tree.
2. To search for an element in a binary search tree, we begin by comparing the value of the element we are searching for with the value of the current node. If the values match, we have found the element and the search is successful. If the value we are searching for is less than the current node's value, we move to the left subtree and repeat the comparison. If the value we are searching for is greater than the current node's value, we move to the right subtree and repeat the comparison.
3. The search algorithm is implemented recursively until the element is found or it is determined that the element does not exist in the tree. In order to search for an element in a binary search tree, we must begin at the root node of the tree. If the root node is null, then the tree is empty and the search is unsuccessful. If the value of the root node matches the value we are searching for, then the search is successful. If the value of the root node is less than the value we are searching for, we recursively search the right subtree. If the value of the root node is greater than the value we are searching for, we recursively search the left subtree.

4. The time complexity of searching for an element in a binary search tree is O(log n) in the average case, where n is the number of nodes in the tree. In the worst case, the time complexity can be O(n) if the binary search tree is unbalanced and degenerates into a linked list.

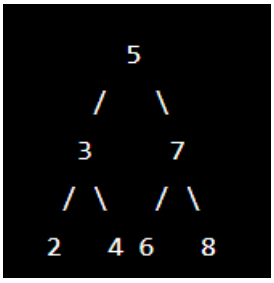
ALGORITHM:

The algorithm to find a number in a binary search tree is as follows:

1. Start at the root node of the tree.
2. If the root node is null, the search is unsuccessful.
3. If the root node's value is equal to the number being searched, the search is successful.
4. If the root node's value is less than the number being searched, search the right subtree of the root node recursively.
5. If the root node's value is greater than the number being searched, search the left subtree of the root node recursively.

TASK:

1. Create a binary search tree with the following numbers: 8, 3, 10, 1, 6, 14, 4, 7, and 13.
2. Search for the number 6 in the binary search tree.
3. Given the following binary search tree, find whether the number 8 is present or not with steps



PRACTICE ASSIGNMENT QUESTIONS

1. What is a binary search tree?
2. What is the property of a binary search tree that makes searching for a number efficient?
3. Describe the algorithm to find a number in a binary search tree.
4. Implement a program to find a number in a binary search tree using recursion.

4. Write a program to display a mirror image of a binary search tree

AIM: To write a program to find the mirror image of a given binary search tree.

OBJECTIVES:

- To understand the concept of a binary search tree and its mirror image.
- To learn how to traverse a binary search tree recursively.
- To implement the algorithm for finding the mirror image of a binary search tree.
- To test the program with different inputs and verify the output.

THEORY:

1. A binary search tree is a data structure in which each node has at most two children, referred to as the left and right subtrees. The left subtree contains nodes with values less than the node's value, while the right subtree contains nodes with values greater than the node's value. The mirror image of a binary search tree is a binary search tree in which the left and right subtrees of each node are swapped.

2. To find the mirror image of a binary search tree, we need to traverse the tree and swap the left and right subtrees of each node recursively. We start at the root node of the tree and swap its left and right subtrees. Then we recursively swap the left and right subtrees of each child node until we have swapped all the nodes in the tree.

3. The time complexity of finding the mirror image of a binary search tree is O(n), where n is the number of nodes in the tree.

ALGORITHM:

1. Define a function **mirror()** that takes the root node of a binary search tree as input.
2. If the root node is null, return null.
3. Create a new node **mirrorNode** with the same value as the root node.
4. Set the left child of **mirrorNode** to the result of calling **mirror()** with the right child of the root node as input.
5. Set the right child of **mirrorNode** to the result of calling **mirror()** with the left child of the root node as input.
6. Return **mirrorNode**.

TASK:

1. Create a binary search tree with the following numbers: 8, 3, 10, 1, 6, 14, 4, 7, and 13.
2. Find the mirror image of the binary search tree.

PRACTICE ASSIGNMENT QUESTIONS:

1. Write a program to create a binary search tree from a given list of integers and print its mirror image.
2. Write a program to create a binary search tree from a given list of characters and print its mirror image.
3. Write a program to create a binary search tree from a given list of strings and print its mirror image.
4. Write a program to create a binary search tree from user input and print its mirror image.
5. Write a program to compare the given binary search tree and its mirror image.

5. Write a program to perform the operations of insertion and deletion on Heap

AIM: To write a program to perform the operation of insertion and deletion on a heap.

OBJECTIVES:

- To understand the concept of a heap data structure.
- To learn how to insert and delete elements in a heap.
- To implement the algorithms for insertion and deletion on a heap.
- To test the program with different inputs and verify the output.

THEORY:

1. HEAP: A Heap is a binary tree with some additional properties that make it useful for a wide range of applications. In a binary heap, the elements are stored in an array, where each element has a specific index.
2. Min Heap and Max Heap: A heap can be of two types - Min Heap and Max Heap. In a Min Heap, the smallest element is always stored at the root node, while in a Max Heap, the largest element is always stored at the root node.
3. Heap Operations: A Heap supports two basic operations - Insertion and Deletion. The insertion operation adds an element to the heap while maintaining the heap property, while the deletion operation removes an element from the heap while maintaining the heap property.
4. Heapify: Heapify is the process of converting a binary tree into a heap. This process involves arranging the nodes of the tree in a specific order, such that the heap property is satisfied.
5. Insertion: Insertion is the process of adding a new element to the heap. The new

element is always added to the last position in the heap, and then it is compared with its parent. If the new element is smaller than its parent (in the case of a Min Heap) or greater than its parent (in the case of a Max Heap), then they are swapped. This process continues until the heap property is satisfied.

6. Deletion: Deletion is the process of removing the root node from the heap. After removing the root node, the last element in the heap is moved to the root position. Then the new root node is compared with its children, and if necessary, they are swapped. This process continues until the heap property is satisfied.

7. Time Complexity: The time complexity of Insertion and Deletion in a Heap is O(log n), where n is the number of elements in the heap.
8. Space Complexity: The space complexity of a Heap is O(n), where n is the number of elements in the heap.

ALGORITHM:

Insertion

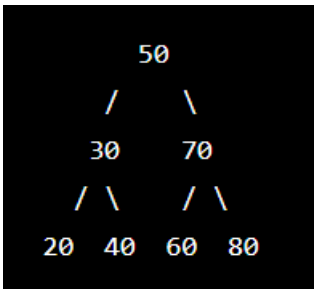
1. Define a function **insert()** that takes the heap array and the value to be inserted as input.
2. Add the value to the end of the array.
3. Calculate the index of the parent of the newly added value by dividing its index by 2.
4. While the parent index is greater than or equal to 1 and the value at the parent index is less than the value at the newly added index:
    - Swap the value at the parent index with the value at the newly added index.
    - Update the newly added index to be the parent index.
    - Recalculate the parent index.
5. Return the updated heap array.

Deletion

1. Define a function **delete()** that takes the heap array as input.
2. Swap the root element (i.e., the first element in the array) with the last element in the array.
3. Remove the last element from the array.
4. Set the current index to 1 (i.e., the root element index).
5. While the current index has at least one child and the value at the current index is less than the value of either of its children:
    - If the left child index is out of bounds or the right child value is greater than the left child value:
        - Swap the value at the current index with the value at the right child index.
        - Update the current index to be the right child index.
    - Otherwise:
        - Swap the value at the current index with the value at the left child index.
        - Update the current index to be the left child index.
6. Return the updated heap array.

TASK:

1. Create an empty max heap.
2. Insert the following numbers into the heap: 6, 9, 5, 8, 7, 1, 3, 2, and 4.
3. Delete the maximum element from the heap.
4. Given the following binary tree perform the following operations



a) Insert the element 55 into the heap.

b) Delete the maximum element from the heap.

c) Insert the element 90 into the heap.

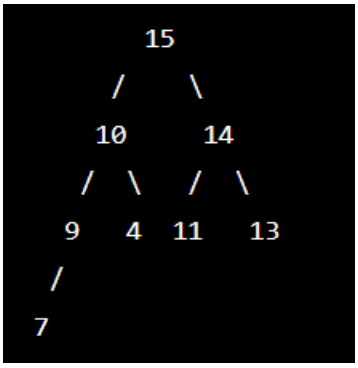
d) Delete the maximum element from the heap.

Write down the resulting heap after each operation.

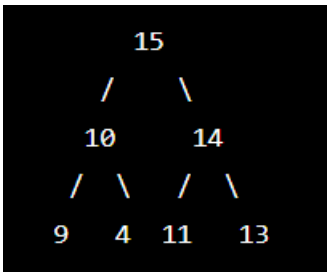
5. Draw the binary heap that would be created from the following array of integers.

```
[15, 8, 20, 7, 10, 16, 25, 12, 18]
```

6. Given the following binary heap, perform a deletion operation on it and show the resulting heap.



7. Given the following binary heap, perform an insertion operation on it for the value 6 and show the resulting heap.



PRACTICE ASSIGNMENT QUESTIONS:

1. Write a program to create a max heap from a given list of integers and print it.
2. Write a program to create a min heap from a given list of integers and print it.
3. Write a program to insert an element in a max heap and print the updated heap.
4. Write a program to delete the root element from a max heap and print the updated heap.

5. Write a program to insert an element in a min heap and print the updated heap.

6. Write a menu driver program to compute the factorial and display a Fibonacci series for the number entered by the user using recursion and write a program to solve the tower of Hanoi problem

AIM - The aim of this lab is to write a program to perform the following tasks: a) Compute the factorial and display a Fibonacci series for the number entered by the user using recursion. b) Solve the tower of Hanoi problem.

OBJECTIVES:

After completing this lab, you will be able to:

- Understand the concept of recursion.
- Write a menu driver program to compute the factorial and display a Fibonacci series.
- Write a program to solve the tower of Hanoi problem.

THEORY:

1. **Recursion -** Recursion is a technique in which a function calls itself repeatedly until a base condition is met. In a recursive function, the function calls itself with a different input value. The base condition is the condition that stops the recursion.
2. **Factorial -** Factorial is the product of all positive integers from 1 to n, where n is the number entered by the user. The factorial of a number is denoted by n!. For example, the factorial of 5 is 5! = 5 x 4 x 3 x 2 x 1 = 120.
3. **Fibonacci Series -** Fibonacci series is a series of numbers in which each number is the sum of the two preceding numbers, starting from 0 and 1. For example, the first ten numbers in the Fibonacci series are: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34.
4. **Tower of Hanoi -** The Tower of Hanoi is a mathematical puzzle consisting of three rods and a number of disks of different sizes, which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the

top, thus making a conical shape. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
- No disk may be placed on top of a smaller disk.

ALGORITHM

Factorial and Fibonacci Series

1. If the user enters 0 or 1, display 1 as the factorial and 0 as the Fibonacci series.
2. If the user enters a number greater than 1, display the following menu: a. To compute factorial, press 1 b. To display Fibonacci series, press 2 c. To exit, press 3
3. If the user presses 1, call the factorial function and display the result.
4. If the user presses 2, call the Fibonacci function and display the series.
5. If the user presses 3, exit the program.

Factorial Function

1. If n is 0 or 1, return 1.
2. Otherwise, return n * factorial(n-1).

Fibonacci Function

1. If n is 0, return 0.
2. If n is 1, return 1.
3. Otherwise, return fibonacci(n-1) + fibonacci(n-2).

Tower of Hanoi

1. Start

2. Define function hanoi(n, src, dest, aux)
3. If n == 1, print move disk 1 from src to dest and return
4. Call hanoi(n-1, src, aux, dest)
5. Print move disk n from src to dest
6. Call hanoi(n-1, aux, dest, src)
7. Define main function
8. Read input for number of disks n
9. Call hanoi(n, 'A', 'C', 'B')
10. End

TASK:

1. Ask the user to enter a number.
2. Compute and display the factorial of the number entered.
3. Display the Fibonacci series for the number entered.

1. Ask the user to enter the number of disks.
2. Solve the Tower of Hanoi problem for the number of disks entered.

Other tasks:

1. Write a recursive function to find the greatest common divisor (GCD) of two numbers.
2. Write a recursive function to calculate the sum of the digits of a given integer.
3. Write a recursive function to check if a given string is a palindrome.
4. Write a recursive function to calculate the power of a given number.
5. Write a recursive function to generate all permutations of a given string.

PRACTICE ASSIGNMENT QUESTIONS

1. Write a recursive function to find the factorial of a given positive integer n.
2. Write a recursive function to find the nth number in the Fibonacci series.
3. Write a menu-driven program that allows the user to choose whether to compute the factorial or the Fibonacci series for a given input value. Implement the program using recursion.
4. Write a non-recursive program to find the nth number in the Fibonacci series.
5. Implement the tower of Hanoi problem using a non-recursive algorithm.
6. Write a recursive function to print the steps required to solve the tower of Hanoi problem for a given number of disks.
7. Modify the program to solve the tower of Hanoi problem to output the minimum number of moves required to solve the problem for a given number of disks.
8. Write a program to calculate the sum of the first n numbers in the Fibonacci series using recursion.
9. Write a program to calculate the sum of the first n numbers in the Fibonacci series without using recursion.
10. Modify the program to calculate the sum of the first n even numbers in the Fibonacci series.

7. Apply Backtracking to solve the 4 Queens problem

AIM: The aim of this lab is to learn about the backtracking technique and its application to solve the 4 Queens problem.

OBJECTIVES:

- Understand the backtracking technique.
- Implement the backtracking algorithm to solve the 4 Queens problem.
- Test and analyze the performance of the algorithm.

THEORY:

1. Backtracking is a general algorithmic technique that is often used to solve optimization problems. It is an algorithmic approach that attempts to find all possible solutions to a problem by constructing a tree of choices and exploring all possible paths down the tree until a solution is found. If a path reaches a point where no further progress can be made, the algorithm "backs up" to the previous choice point and explores a different path. Backtracking is often used to solve problems where the solution space is very large and it is not feasible to explore all possible solutions. Instead, backtracking explores only those parts of the solution space that are likely to contain a solution.
2. The 4 Queens problem is a classic example of a problem that can be solved using backtracking. In this problem, we are asked to place 4 queens on a 4x4 chessboard in such a way that no two queens threaten each other. In other words, no two queens can be placed in the same row, column, or diagonal.
3. To solve this problem using backtracking, we start by placing the first queen in the first row of the chessboard. We then attempt to place the second queen in the second row, the third queen in the third row, and the fourth queen in the fourth row. At each step, we check whether the placement of the queen is consistent with the rules of the game. If it is not, we backtrack to the previous step and try a different placement. If all possible

placements have been tried and none are successful, we backtrack again to the previous step and try a different placement.

4. To determine whether a placement of the queens is consistent with the rules of the game, we need to check whether any two queens are in the same row, column, or diagonal. This can be done by examining the positions of the queens that have already been placed and comparing them to the position of the queen that is being placed.
5. is a powerful algorithmic technique that can be used to solve a wide range of problems. However, it can be computationally expensive, particularly for problems with a large search space. In practice, it is often used in conjunction with other algorithmic techniques, such as pruning or heuristics, to speed up the search and reduce the computational cost.

ALGORITHM:

1. Initialize a 4x4 chessboard with all entries as 0, which indicates that no queen is placed in any cell of the board.
2. Start with the leftmost column of the board.
3. Place a queen in the first row of the current column.
4. Check if the queen conflicts with any previously placed queens. If not, move to the next column and repeat steps 3 and 4.
5. If all columns have been filled, print the solution.
6. If a conflict occurs in any column, move the queen in the current column to the next row and repeat step 4.
7. If the queen cannot be placed in any row of the current column, backtrack to the previous column and move the queen in that column to the next row.
8. Repeat steps 4 to 7 until a solution is found or all possible configurations have been tried.

In this algorithm, backtracking is used to find all possible solutions for the 4 Queens problem by trying different configurations of the chessboard and undoing any changes that lead to conflicts. The algorithm essentially tries to place a queen in each column of the board, one at a time, and

checks if it conflicts with any of the previously placed queens. If a conflict occurs, the algorithm backtracks to the previous column and tries a different row for the queen in that column. This process continues until a solution is found or all possible configurations have been tried.

TASK:

1. Solve the 4 Queens problem using backtracking.
2. Use backtracking to generate all possible combinations of N elements from a given set.
3. Given a maze represented as a 2D array, apply backtracking to find a path from the starting point to the end point.

PRACTICE ASSIGNMENT QUESTIONS:

1. Write a non-recursive function to solve the 4 Queens problem.
2. Implement the backtracking algorithm to solve the 5 Queens problem.
3. Write a program to count the total number of solutions for the 4 Queens problem.
4. Modify the program to print all solutions to the 4 Queens problem.
5. Modify the program to solve the N Queens problem for any given value of N.
6. Write a program to visualize the solutions to the 4 Queens problem on a 4x4 chessboard.
7. Modify the program to visualize the solutions to the N Queens problem for any given value of N.
8. Compare the performance of the recursive and non-recursive solutions to the 4 Queens problem.

8. Write a program to sort a given list of elements using Heap sort

AIM: To write a program to sort a given list of elements using Heap sort.

OBJECTIVES:

- To understand the concept of Heap sort.
- To implement Heap sort algorithm using an array.
- To analyze the time complexity and space complexity of the algorithm.
- To compare the efficiency of Heap sort with other sorting algorithms.

THEORY:

Heap sort is a sorting algorithm that works by first building a heap from the given array, then repeatedly extracting the maximum element from the heap and placing it at the end of the sorted array. This process is repeated until the heap is empty.

Heap is a binary tree data structure where the key of each node is greater than or equal to the keys of its children. This is called the heap property. In a max heap, the key of the parent node is greater than or equal to the keys of its children, while in a min heap, the key of the parent node is less than or equal to the keys of its children.

The heap sort algorithm can be divided into two phases: buildHeap and heapSort.

1. buildHeap: The buildHeap function takes an array as input and builds a max heap from it. The algorithm starts with the first non-leaf node and applies the heapify procedure to each node. The heapify procedure ensures that the subtree rooted at the given node satisfies the heap property.

2. heapSort: The heapSort function repeatedly extracts the maximum element from the heap and places it at the end of the sorted array. This process is repeated until the heap is empty. To extract the maximum element, the root node is removed and replaced with the last element in the heap. The heap is then restored by applying the heapify procedure to the root node.

The time complexity of heap sort is O(n log n) in both the worst and average cases. The space complexity is O(1) as the algorithm sorts the input array in place.

ALGORITHM:

```
heapSort(arr, n):
    # Build max heap
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    # Extract elements from heap
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]   # swap
        heapify(arr, i, 0)

def heapify(arr, n, i):
    largest = i  # Initialize largest as root
    left = 2 * i + 1  # left child
    right = 2 * i + 2  # right child

    # See if left child of root exists and is greater than root
    if left < n and arr[left] > arr[largest]:
        largest = left

    # See if right child of root exists and is greater than root
    if right < n and arr[right] > arr[largest]:
        largest = right
```

```
# Change root, if needed
if largest != i:
    arr[i], arr[largest] = arr[largest], arr[i]   # swap

    # Heapify the root.
    heapify(arr, n, largest)
```

In the above algorithm, **arr** is the input array to be sorted and **n** is the length of the array. The **heapify()** function is used to maintain the heap property at a given node.

TASK:

1. Create a list with the following elements: 5, 3, 8, 4, 2, 7, 1, 6, and 9.
2. Sort the list using Heap sort.

PRACTICE ASSIGNMENT QUESTIONS:

1. Write a program to implement Heap sort in C++.
2. Compare the time complexity of Heap sort with other sorting algorithms like Bubble sort, Insertion sort, and Quick sort for a list of 10000 elements.
3. Implement Heap sort algorithm using a priority queue and compare its efficiency with the array implementation.
4. Explain the working of Heap sort with an example.
5. Write a program to sort a list of strings using Heap sort.
6. What is the worst-case time complexity of Heap sort?

9. Merge Sort and Quick Sort

AIM: To implement Quick Sort and Merge Sort algorithms to sort a given list of elements.

OBJECTIVES:

- To understand the working of Quick Sort and Merge Sort algorithms
- To implement Quick Sort and Merge Sort algorithms
- To analyze the time complexity of Quick Sort and Merge Sort algorithms
- To compare the performance of Quick Sort and Merge Sort algorithms on various inputs

THEORY:

Sorting is a fundamental operation in computer science, and various sorting algorithms are available to sort a list of elements. Two popular sorting algorithms are Quick Sort and Merge Sort.

Quick Sort: Quick Sort is a divide-and-conquer algorithm. It works by partitioning an array into two sub-arrays, and then recursively sorting the sub-arrays. The steps to perform Quick Sort are:

- Choose a pivot element from the array
- Partition the array around the pivot element such that all elements smaller than the pivot element are on the left of the pivot element, and all elements greater than the pivot element are on the right of the pivot element
- Recursively apply Quick Sort on the sub-array to the left of the pivot element and the sub-array to the right of the pivot element.

The choice of the pivot element affects the performance of Quick Sort. Ideally, the pivot element should be the median element in the array. However, finding the median element is not easy, and in practice, the first or the last element in the array is often chosen as the pivot element.

Merge Sort: Merge Sort is also a divide-and-conquer algorithm. It works by recursively dividing an array into two halves, sorting the two halves independently, and then merging the sorted

halves. The steps to perform Merge Sort are:

- Divide the array into two halves
- Recursively apply Merge Sort on the left half and the right half of the array
- Merge the sorted left half and the sorted right half.

The merging step in Merge Sort is the most critical step, and it can be implemented in different ways. The most common way is to use an auxiliary array to store the merged result.

ALGORITHM:

Quick Sort:

```
QuickSort(array A, low, high) {
    if (low < high) {
        pivotIndex = Partition(A, low, high) // partition the array around a pivot
        QuickSort(A, low, pivotIndex - 1) // sort the left sub-array
        QuickSort(A, pivotIndex + 1, high) // sort the right sub-array
    }
}

Partition(array A, low, high) {
    pivotValue = A[high] // choose the last element as the pivot
    i = low - 1 // initialize index of smaller element
    for j = low to high - 1 {
        if (A[j] <= pivotValue) {
            i = i + 1 // increment index of smaller element
            Swap(A[i], A[j]) // swap A[i] and A[j]
        }
    }
    Swap(A[i+1], A[high]) // swap A[i+1] and A[high], put the pivot element in its
    return i + 1 // return the index of the pivot element
}
```

Merge Sort:

```
MergeSort(array A, start, end) {
    if (start < end) {
        middle = (start + end) / 2
        MergeSort(A, start, middle) // sort left half of array
        MergeSort(A, middle+1, end) // sort right half of array
        Merge(A, start, middle, end) // merge sorted halves
    }
}

Merge(array A, start, middle, end) {
    // create temporary arrays to hold left and right halves
    leftArray = A[start:middle]
    rightArray = A[middle+1:end]

    // set initial indices for left and right arrays
    leftIndex = 0
    rightIndex = 0
    resultIndex = start

    // merge left and right arrays into sorted result array
    while (leftIndex < length(leftArray) and rightIndex < length(rightArray)) {
```

```
        if (leftArray[leftIndex] <= rightArray[rightIndex]) {
            A[resultIndex] = leftArray[leftIndex]
            leftIndex = leftIndex + 1
        } else {
            A[resultIndex] = rightArray[rightIndex]
            rightIndex = rightIndex + 1
        }
        resultIndex = resultIndex + 1
    }

    // copy remaining elements from left or right array
    while (leftIndex < length(leftArray)) {
        A[resultIndex] = leftArray[leftIndex]
        leftIndex = leftIndex + 1
        resultIndex = resultIndex + 1
    }
    while (rightIndex < length(rightArray)) {
        A[resultIndex] = rightArray[rightIndex]
        rightIndex = rightIndex + 1
        resultIndex = resultIndex + 1
    }
}
```

TASK:

Merge Sort:

1. Given the following list of numbers: [7, 4, 2, 8, 5], show the steps of the Merge Sort algorithm to sort the list.
2. Write a C program to implement Merge Sort to sort a list of numbers.
3. Explain the time complexity of the Merge Sort algorithm.
4. What is the space complexity of the Merge Sort algorithm?
5. Given the following list of strings: ["cat", "dog", "bird", "fish"], show the steps of the Merge Sort algorithm to sort the list.

Quick Sort:

1. Given the following list of numbers: [7, 4, 2, 8, 5], show the steps of the Quick Sort algorithm to sort the list.
2. Write a C program to implement Quick Sort to sort a list of numbers.
3. Explain the time complexity of the Quick Sort algorithm.

4. What is the space complexity of the Quick Sort algorithm?
5. Given the following list of strings: ["cat", "dog", "bird", "fish"], show the steps of the Quick Sort algorithm to sort the list.

Combined Questions:

1. Consider the following list of numbers: [6, 3, 8, 2, 5, 1, 4, 7]. Show the step-by-step process of sorting this list using Merge Sort. Draw the state of the list after each merge operation.
2. Show the step-by-step process of sorting the following list of numbers using Quick Sort: [5, 1, 9, 3, 7, 6, 8, 2, 4]. Draw the state of the list after each partition operation.
3. Consider the following list of numbers: [4, 7, 2, 8, 1, 3, 6, 5]. Which sorting algorithm would be more efficient in sorting this list, Merge Sort or Quick Sort? Justify your answer with a brief explanation.

PRACTICE QUESTIONS:

1. Explain the working of Merge Sort with the help of an example.
2. What is the time complexity of Quick Sort in the worst case scenario? Explain.
3. Write a program to sort the given array using Quick Sort.
4. How does Merge Sort ensure stability while sorting elements?
5. Compare the time complexity of Merge Sort and Quick Sort for sorting a large array.
6. Can we use Quick Sort for sorting a linked list? Why or why not?
7. Write a program to sort the given list of elements using Merge Sort.
8. What is the basic idea behind Quick Sort? Explain with an example.
9. How can we improve the performance of Merge Sort in terms of space complexity?
10. In what situations would you prefer to use Merge Sort over Quick Sort, and vice versa?

10. Write a program to apply search using hashing techniques on the student database of B.Tech Computer Engineering.

AIM: To apply search using hashing techniques on the student database of B.Tech Computer Engineering.

OBJECTIVES:

- To understand the concept of hashing.
- To implement hash function to store and retrieve data.
- To apply collision resolution techniques.
- To implement search operations using hashing.

THEORY:

1. Hashing is a technique that is used to store and retrieve data in a faster and more efficient way. In this technique, data is stored in an array and the data is accessed using a key value. A hash function is used to map the key value to an index in the array where the data is stored.
2. The hash function takes the key as an input and generates an index in the array where the data will be stored. A good hash function should be able to distribute the data evenly across the array, minimizing the chances of collisions.
3. Collision is a situation where two different keys are mapped to the same index in the array. There are different techniques used to resolve collisions, including linear probing, quadratic probing, and chaining.
4. In linear probing, when a collision occurs, the program will search for the next available index in the array and store the data there. In quadratic probing, the program will use a quadratic function to find the next available index. In chaining, each index in the array is a linked list, and when a collision occurs, the data is added to the linked list at that index.

ALGORITHM:

1. Initialize a hash table with a fixed size.
2. Define a hash function that maps the keys to an index in the hash table.
3. When a new record is inserted, apply the hash function to the key to get the index in the hash table.
4. If the index is empty, insert the record at that index.
5. If the index is occupied, apply the collision resolution technique.
6. When searching for a record, apply the hash function to the key to get the index in the hash table.
7. If the record is not found at that index, apply the collision resolution technique.
8. Repeat steps 6 and 7 until the record is found or until the entire hash table has been searched.

TASK:

1. Given the following dataset of student names and their corresponding roll numbers, create a hash table using linear probing:

Name	**Roll Number**
John Smith	1001
Jane Doe	2002
David Lee	3003
Sarah Wang	4004
Bob Chen	5005

2. Given a hash table with the following entries and their corresponding indices, perform a search operation for the roll number "3003" using linear probing:

Index	**Entry**
0	
1	John Smith
2	Sarah Wang
3	David Lee
4	
5	Bob Chen
6	
7	
8	Jane Doe
9	

3. Why is hashing considered a faster search algorithm compared to linear search or binary search? Explain with an example.

PRACTICE ASSIGNMENT QUESTIONS:

1. Write a program to implement linear probing to resolve collisions in a hash table.
2. Implement a hash function that uses the last two digits of the student's roll number as the key.
3. Write a program to search for a student record using chaining to resolve collisions in a hash table.
4. Design a hash function that can handle large data sets and reduces the chances of collisions.