Lab Manual

For

Data Structures Lab (ARI254)

University School of Automation & Robotics (USAR)

GGS IP University, East Delhi Campus

Surajmal Vihar, Delhi-110092

ARI 254:: Data Structures Lab	LTP:002	Credit:1

Course Outcome (CO)

CO1: Ability to understand and use various data structures for solving real world problems.

CO2: Ability to learn the basic idea of array data structure and how it is stored in computer memory, and to use arrays for programming problems. **[K2, K3, K4]**

CO3: To gain an idea of Linked List and its types, implement linked lists while performing various operation associated with various type of linked lists. **[K3, K4]**

CO4: Developed intuition for stacks, program stack ADT using arrays, linked list, and conversion *infix* to *postfix*.

CO5: Better understanding of different types of queues, and efficient approaches to implement queue using arrays and linked lists. **[K3, K4]**

CO6: Ability to implement tree ADT using Arrays, Linked list, tree traversal, and use appropriate tree data structures to solve real-world problems**.** Learning about BST, heap, and B-tree data structure and ability to implement and apply these data structure to solve various real-world problems.

CO7: Better understanding of graph properties, representations, and efficient implementation of Graph traversal methods.

CO8: Better understanding and efficient implementation of various searching and sorting techniques.

CO9: Better understanding of hashing techniques, linear probing, double hashing, and quadratic hashing to resolve the collision.

CO/ PO	PO01	PO02	PO03	PO04	PO05	PO06	PO07	PO08	PO09	PO10	PO11	PO12
CO1	3	2	1	1	3	-	-	-	2	2	1	2
CO2	3	3	3	3	3	-	-	-	3	3	3	2
CO3	3	2	1	1	3	-	-	-	2	2	1	3
CO4	3	3	3	3	3	-	-	-	3	3	3	3
CO5												
CO6												
CO7												
CO8												
CO9												
Average	**3**	2.5	2	2	3	-	-	-	2.5	2.5	2	2.5

DETAILED SYLLABUS

LAB 1: Introduction to Data Structures

LAB 2: Array Data Structure

LAB 3: Linked List-I (Singly Linked List), Linked List-II (Doubly Linked List)

LAB 4: Stack

LAB 5: Queues

LAB 6: Tree - I, Tree - II

LAB 7: Graphs

LAB 8: Searching & Sorting

LAB 9: Hashing

Text Books:

“Introduction to Algorithms” by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein

“The Art of Computer Programming” by Donald E. Knuth

“Introduction to Algorithms: A Creative Approach” by Udi Manber

“Let us C” by Yaswant Kanitker, “Pointers in C” by Yaswant Kanitker,

References:
[1] https://www.geeksforgeeks.org/data-structures/?ref=shm
[2] https://www.javatpoint.com/data-structure-tutorial

LAB Equipment

Following hardware and software are necessary to perform the experiments in Advanced Algorithms lab.

Hardware:
1. A computer that can execute C/C++ programs.

Software:
1. Windows / UNIX Operating System.
2. Compilers: Dev C++ / GCC

Programming Languages:
C / C++

Content

LABORATORY-I (INTRODUCTION TO DATA STRUCTURES)

Objective:

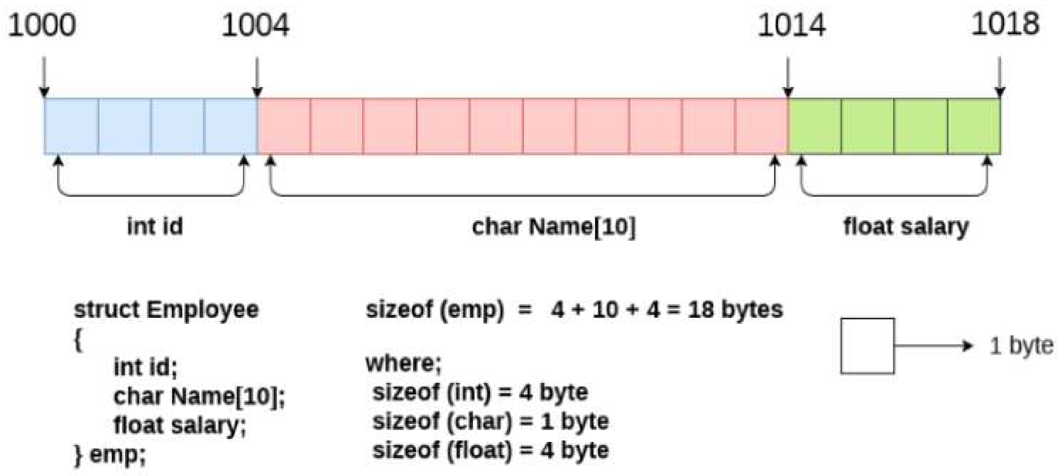
To revise the useful concept of programming prerequisite which will be useful for future lesions. Structures, union, pointers, 1D array etc. are covered in this lab.

Brief Theory:

Structures: A structure is a user defined data type in C/C++. A structure creates a data type that can be used to group items of possibly different types into a single type. key word is used to create a structure.

Syntax of Structure:

```
struct structureName{
    data_type member1;
    data_type member2;
    .
    .
    .
    Data_type memberN;
};
```



```
struct Employee
{
    int id;
    char Name[10];
    float salary;
} emp;
```

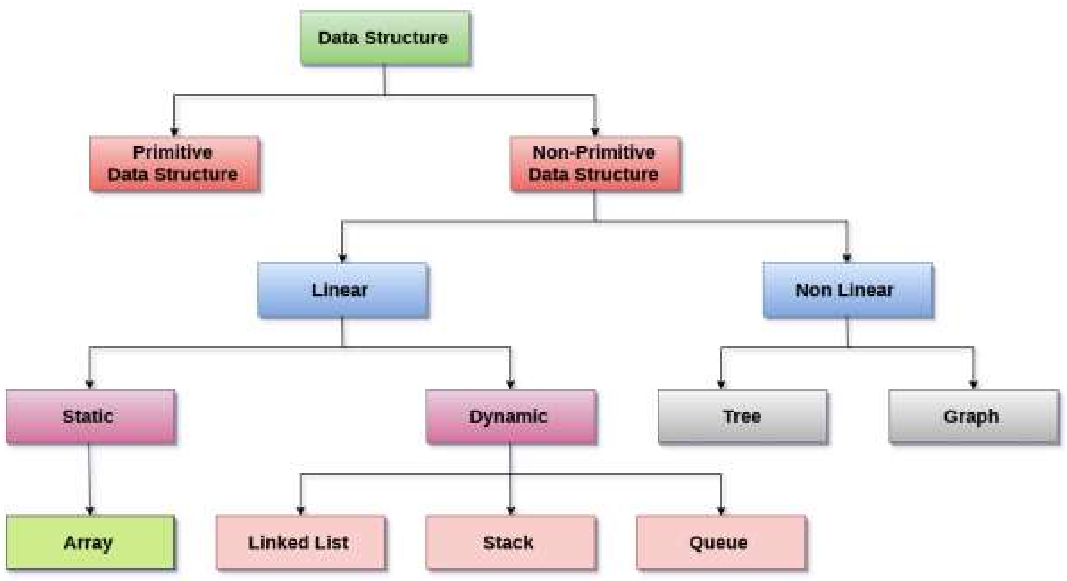
```
sizeof (emp)  =  4 + 10 + 4 = 18 bytes

where;
 sizeof (int) = 4 byte
 sizeof (char) = 1 byte
 sizeof (float) = 4 byte
```

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are widely used in almost every aspect of Computer Science i.e., Operating System, Compiler Design, Artificial intelligence, Graphics and many more.

Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way. It plays a vital role in enhancing the performance of a software or a program as the main function of the software is to store and retrieve the user's data as fast as possible.

Based on how they are stored in the memory data structures are classified as:



The main characteristics of data structures are as follows:

• **Correctness** - Data structure implementation should implement its interface correctly.

• **Time Complexity** - Running time or the execution time of operations of data structure must be as small as possible.

• **Space Complexity** - Memory usage of a data structure operation should be as little as possible.

Observations/Outcome:

1. Develop the understanding of data structure and its types.
2. Ability to use various data structures for solving unseen problems.
3. Understand the use of appropriate data structure to solve real world problems.

Problems on:

1. Basic C program using loops, structure.
2. Programs on unions, 1D array.
3. To design a small database type application using structure, pointer, array.

Summarized Course Outcome [CO1]: Ability to understand and use various data structures for solving real world problems.

LABORATORY-II (ARRAY)

Objective:

To understand one of the most basic data structures called array. This will help the students as a pre-requisite for most of the complex data structures.

Brief Theory:

Arrays are defined as the collection of similar types of data items, data items in array are stored at contiguous memory locations. It is one of the simplest data structures where each data element can be randomly accessed by using its index number.

Properties of array:

- Each element in an array is of the same data type and carries the same size.
- Elements in the array are stored at contiguous memory locations from which the first element is stored at the smallest memory location.
- Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.

Arrays are useful because:

- Sorting and searching a value in an array are easier.
- Arrays are best to process multiple values quickly and easily.
- Arrays are good for storing multiple values in a single variable **-** In computer programming, most cases require storing a large numbers of data of a similar type. To store such an amount of data, we need to define many variables. It would be very difficult to remember the names of all the variables while writing the programs. Instead of naming all the variables with a different name, it is better to define an array and store all the elements into it.

Observations/Outcomes:

1. The students will learn the basic idea of array data structure and how it is stored in computer memory.
2. The students will be able to use arrays for programming problems.

Problems on:

1. Real world scenarios involving array.
2. Implementation of multi-dimensional arrays and problems related to them like matrix multiplication.

Summarized Course Outcome [CO2]: Ability to learn the basic idea of array data structure and how it is stored in computer memory, and to use arrays for programming problems.

LABORATORY-III (LINKED LIST)

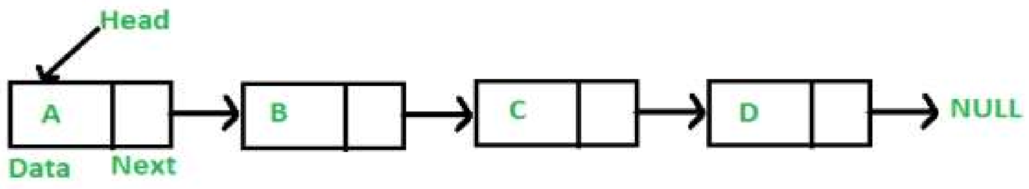
Linked List - I

Objectives:

To get familiar with the linked list data structure and gain the ability to utilize linked list data structure to develop other advanced data structure.

Brief Theory:

Linked List is a linear data structure, and it is very common data structure which consists of group of nodes in a sequence which is divided in two parts. Each node consists of its own data field and the address field. Data field is used to store the data and the address field is used to store the address of its adjacent nodes. First node of a linked list is called head, and only the address of the first node is stored to store a linked list. All other nodes can be accessed from the first node itself by sequentially exploring the adjacent node. Basically, linked list forms a chain like structure. called head Linked Lists are used to create trees and graphs. Structure is used to create a node in linked list.



Advantage of Linked List:

- Unlike array In Linked Lists, the size of the list need not be fixed in advance.
- Linked list is dynamic in nature, which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed.
- Linked lists let you insert elements at the beginning and end of the list.

Disadvantage:

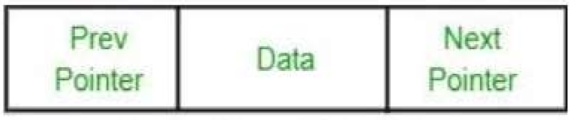
- One disadvantage of linked list is random access is not possible in case of linked list, to access any element we have to search it from the head, which lead to complexity of searching.
- Another disadvantage of linked list is memory wastage, each nodes waist some amount of memory to store the address of its adjacent nodes.

Types:

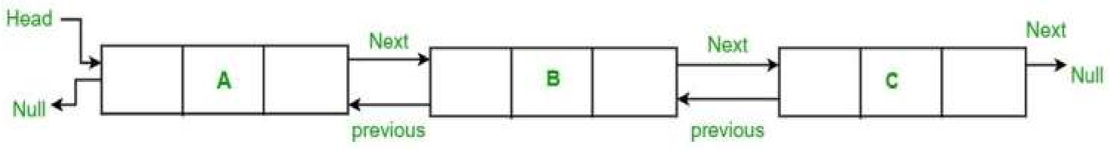
1. **Singly Linked List**: Singly linked lists contain nodes which have a data part as well as an address part i.e., next, which points to the next node in the sequence. In singly linked

list from any particular node there is no way to reach its previous node. The operations we can perform on singly linked lists are insertion, deletion, and traversal.

2. **Doubly Linked List**: In a doubly linked list, each node maintains two pointers namely and which are used to store the previous and next adjacent element respectively. The first link points to the previous node and the next link points to the next node in the sequence.



Node of double linked list



Observations/Outcomes:

1. To get an idea of Linked List and its types.
2. The students will be able to develop Linked List and its type from scratch.

Problems on:

1. Problem on implementation of the various operations on singly *Linked List*, like **create**, **display**, **delete**.
2. Program to create doubly linked list, and search for a particular key.
3. Inserting, deleting elements in linked list at various position.

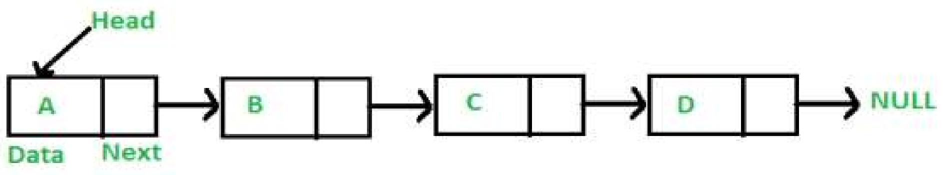
LINKED LIST-II

Objectives:

To dive deep into linked list data structures, it types, and different algorithms related to it.

Brief Theory:

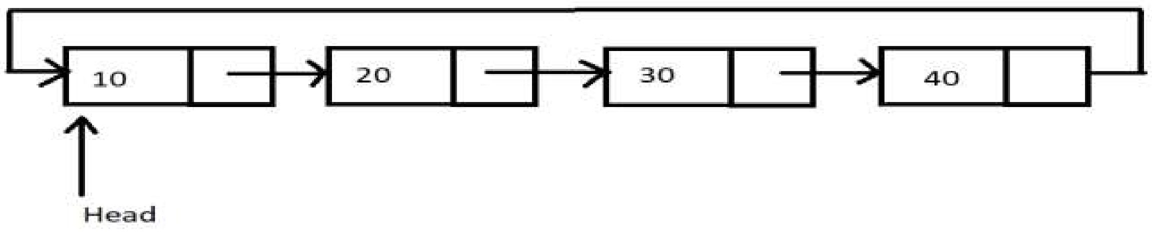
Linked List is a linear and it is very common data structure which consists of group of nodes in a sequence which is divided in two parts. Each node consists of its own data and the address of the next node and forms a chain. Linked Lists are used to create trees and graphs.



Properties:

- They are a dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed.
- Linked List reduces the access time.
- Linked lists are used to implement stacks, queues, graphs, etc.
- Linked lists let you insert elements at the beginning and end of the list.
- In Linked Lists we don't need to know the size in advance.

Circular Linked List: In the circular linked list, the last node of the list contains the address of the first node and forms a circular chain.



Observations/Outcomes:

1. Understand the concepts of *Circular Linked List*.
2. Implement a *Circular Linked List* from scratch.

Problems on:

1. Implement various types of **insertions** and **deletions** possible in *Circular Linked List*.
2. Compare the working of *Single, Doubly* and *Circular Linked List*.

Summarized Course Outcome [CO3]: To gain an idea of Linked List and its types. implement linked lists while performing various operation associated with various type of linked lists.

LABORATORY - IV (STACK)

Objective:

The aim is to get the basic idea of the Stack data structure and its implementation, along with its various operations, as well as its various use cases.

Brief Theory:

A stack is an abstract data type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack. A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last is accessed first. In stack terminology, insertion operation is called PUSH operation and removal operation is called POP operation.

A stack can be implemented by means of Array, Structure, Pointer and Linked-List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays which makes it a fixed size stack implementation.

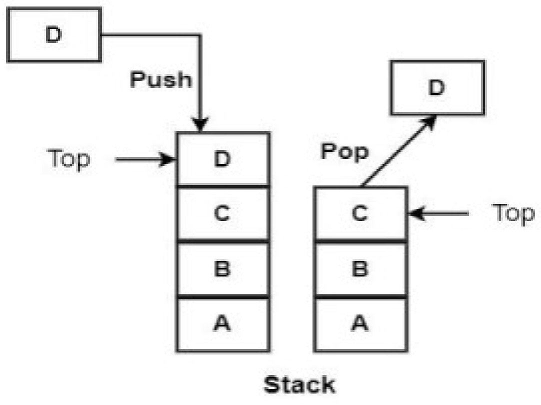
Basic Operations:

- : Pushing (storing) an element on the stack.
- **:** Removing (accessing) an element from the stack.

To use a stack efficiently we need to check status of stack as well. For the same purpose, the following functionality is added to stacks:

- **:** Get the top data element of the stack, without removing it.
- **:** Check if stack is full.
- **:** Check whether the stack is empty and return true or false.

Below given diagram tries to depict push and pop operation of stack:



Observations/Outcome:

1. Developed the intuition of stack.
2. Learnt the basic operations of stack i.e., and etc.
3. Learnt to use stack for unseen programming problems.

Problems on:

1. Programs to implement stack ADT using array.
2. Program to implement stack ADT using linked list.
3. Conversion of *infix* to *postfix* using stack.

Summarized Course Outcome [CO4]: Developed intuition for stacks, program stack ADT using arrays, linked list, and conversion *infix* to *postfix.*

LABORATORY-V (QUEUE)

Objectives:

The aim is to build proper intuition of queue and be able to implement different types of Queues.

Brief Theory:

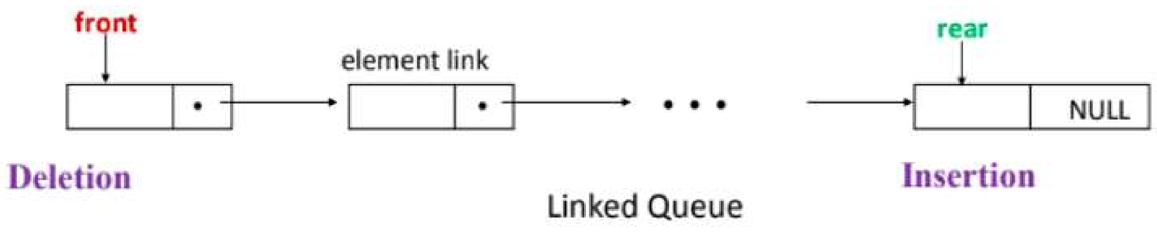
A queue is a linear data structure. It's an ordered list that follows the FIFO principle (First In - First Out). A queue is a structure that has some insertion and deletion constraints. In the case of Queue, insertion is done from one end, which is referred to as the rear end. The deletion is carried out by another end, which is referred to as a front end. The technical terminology for insertion and deletion in Queue are and respectively. It has two pointers in its structure: a pointer and a pointer, element will be added using rear pointer and will be deleted using front pointer.

Operations in Queue:

1. ***Create Q:*** It creates an empty queue, Q.
2. ***Enqueue Q, item:*** Adds the element item to the rear of a queue and returns the new queue. The complexity of enqueue operation is O(1)
3. ***Dequeue Q:*** Returns the older element of the queue.

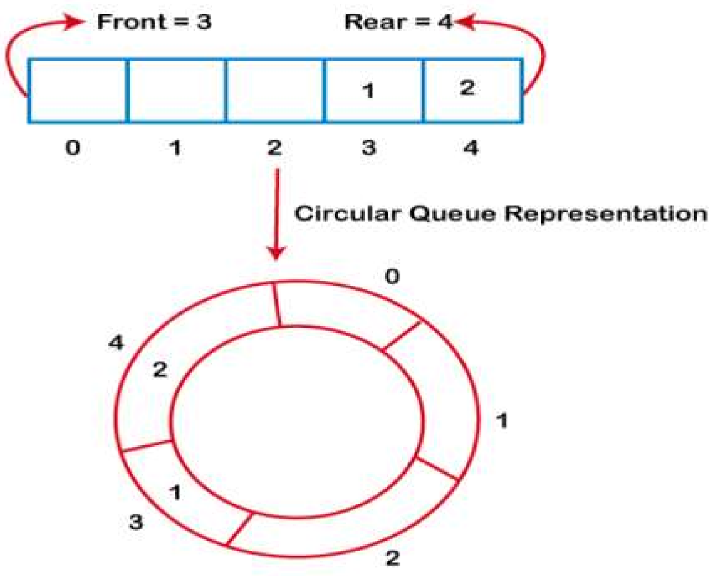
Queues can be implemented in two ways;

1. Using array data structure
2. Using linked list data structure.



Circular Queue: There is a limitation in the array implementation of Queue. If the rear reaches to the end position of the Queue, then there might be possibility that some vacant spaces are left in the beginning which cannot be utilized. So, to overcome those limitations, the concept of the circular queue was introduced.

A circular queue is similar to a linear queue as it is also based on the FIFO (First in First Out) principle except that the last position is connected to the first position in a circular queue that forms a circle. It is also known as a **Ring Buffer**. A modular function is used to implement *Enqueue* and *Dequeue* operations in circular queue.



Priority Queue:

A priority queue is a collection of elements such that each element has been assigned a priority. The order in which elements are processed comes from the following rules:

1. An element of higher priority is processed before any element of lower priority.

2. Two elements with the same priority are processed according to the order in which they were added to the queue.

These priority queue can be implemented using arrays and linked lists.

Observations/Outcome:

1. Better understanding of different types of queues.
2. To understand efficient approaches to implement queue using arrays and linked lists.

Problems on:

1. Creation of Queue using array and then implementation of *Enqueue* and *Dequeue* operations.
2. Creation of Queue using linked list and then implementation of *Enqueue* and *Dequeue* operations.
3. Problems on circular queue of array implementation.

Summarized Course Outcome [CO5]: Better understanding of different types of queues, and efficient approaches to implement queue using arrays and linked lists.

LABORATORY-VI (TREE)

TREE-I

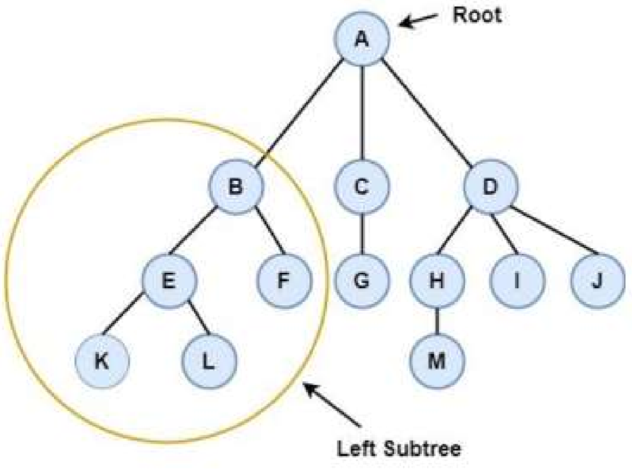
Objectives:

The aim is to learn about the abstract data type **tree** and make students able to implement tree to solve several real-life problems.

Brief Theory:

Tree is a non-linear data structure in which data are stored in hierarchical manner. Tree can be defined as a finite set of one or more nodes such that:

1. There is a specially designated node called the root.
2. The remaining nodes are portioned into $n \geq 0$ disjoint sets T_1, T_2, T_3, … T_n where each of these sets is a tree. T_1, T_2, T_3, … T_n are called the subtrees of the root. For example, in the below tree A is root node, and node within the orange circle is a subtree of A. Similarly, {C, G} and {D, H, I, J, M} are two other subtree of root A.



Tree Terminologies:

Edge: In a tree data structure, the connecting link between any two nodes is called as edge. In a tree with 'N' number of nodes there will be a maximum of 'N-1' number of edges.

Parent Node: If N is immediate predecessor of a node X, then N is called the parent of X. In the above tree A is the parent node of B and C.

Sibling: Nodes which belong to same Parent are called as siblings. E and F are siblings in the above tree.

Internal Node: In tree data structure node which has at least one child is called as internal Node. A, B, C, D, E, F, G, H, I, J are the internal node in the above tree.

Level: The number of edges present in a path from a node to the root is called the level of that node. Level of root is zero, if a node is at level l, its children are at level $l + 1$. For example, level of node H is 2.

Height: In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as height of that particular Node. Height of H is 1.

Tree Traversal: Visiting every node of the tree exactly once is known as traversal. Any hierarchical data structure like a tree can be traversed in different ways. Tree can be traversed in three ways:

[1] **In order traversal** in which for every node first left subtree will be visited then root and then right subtree will be visited,

[2] **Pre order traversal** in which for every node first root then left subtree then and then right subtree will be visited,

[3] **Post Order traversal** in which first left subtree, then right subtree and then root will be visited.

Binary Tree: Binary tree is a special kind of tree in which every node can have at most two children. Binary tree can be represented either using array or linked list. For array representation if a root node is stored at array index, then its left child will be stored at index and the right child will be stored at index. For linked list representation along with data fields every node will contain two pointers, one will be linked with left child, and another will be linked with right child.

Observations/Outcome:

1. Learn to implement tree ADT using array.
2. Ability to implement tree ADT using linked list.
3. Learn to implement various tree traversal methods.
4. Develop the ability to use appropriate tree data structure to solve real world problem.

Problems on:

1. Tree representation.
2. Binary tree implementation.
3. Implementation of different tree traversal algorithms.
4. Inserting new elements, deleting existing elements, and updating elements in tree.

<u>TREE-II</u>

Objective:

To understand the concept of binary search tree (BST), heap and height balancing tree like AVL and B-tree and develop the ability of using those data structure to solve real-world problems.

Brief Theory:

Binary search tree: Binary search tree is a binary tree with some restriction. In binary search tree value of root node will be larger than any node in the left subtree and will be smaller than any node in the right subtree. Because of this restriction searching time of an element will be reduced to *O(log N)* .

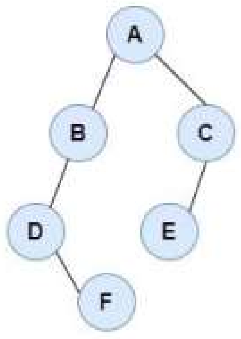
Almost Complete Binary Tree: A almost complete binary tree is actually a binary tree with some restriction in which all the levels are completely filled except possibly the last level and the last level has all keys as left as possible.

Heap: A Heap is a special Tree-based data structure in which the tree is a complete binary tree. Heap is of two types, i.e., Max-Heap and Min-Heap.

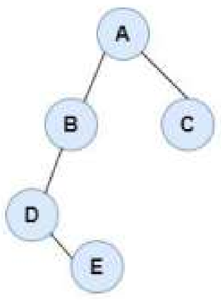
Max-Heap: In a Max-Heap the key present at the root node must be maximum among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree.

Min-Heap: In a Min-Heap the key present at the root node must be minimum among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree.

Height Balancing Tree: Height balancing tree means the difference of the height of left subtree and the height of right subtree of any node is not more than 1. While inserting element in a height balancing tree for any node if the height difference of left and right subtree exceed 1 then appropriate action will immediately be taken to balance the height. The advantage of height balancing tree is that in worst case also searching for an element will take *O(log N)* time. AVL tree B-Tree is the example of height balancing tree.



Height balanced tree



Not height balanced tree

Observations/Outcomes:

1. The students will learn about BST, heap, and B-tree data structure.
2. The students will be able to implement and apply these data structure to solve various computational problems.

Problems on:

1. Creation of Max-Heap and Min-Heap.
2. Insertion and deletion of elements in heap.

3. Problems on binary search tree.
4. Problems on B-Tree creation.
5. Problems on adding element in B-Tree.
6. Problems on removing elements from B-Tree.

Summarized Course Outcome [CO6]:

- Ability to implement tree ADT using Arrays, Linked list, tree traversal, and use appropriate tree data structures to solve real-world problems**.**
- Learning about BST, heap, and B-tree data structure and ability to implement and apply these data structure to solve various real-world problems.

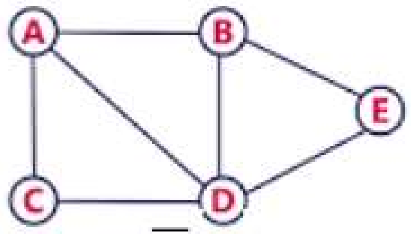
LABORATORY-VII (GRAPHS)

Objectives:

To understand the representation, implementation, and traversal of graph data structure.

Brief Theory:

A Graph **G(V, E)** can be represented by set of vertices denoted by **V** and edges **E**. A graph is a collection of set of vertices connected by edges. Vertices of graph is called node, and these nodes are having data field which stores the data. Difference between tree and graph is tree is acyclic whereas graph may contain cycle. Many real-world problems can be represented using graph, and those problem can be solved using various graph algorithms. For example, let's say we want to find the shortest distance between two cities, we can represent every city as a node of a graph and the edge as the distance between two cities. After this representation we can apply shortest path finding algorithm of graph to determine the shortest path between two cities.



$V = \{A, B, C, D, E\}$

$E = \{(A, B), (B, E), (A, D), (A, C), (C, D), (D, E), (B, D)\}$

Types of Graphs:

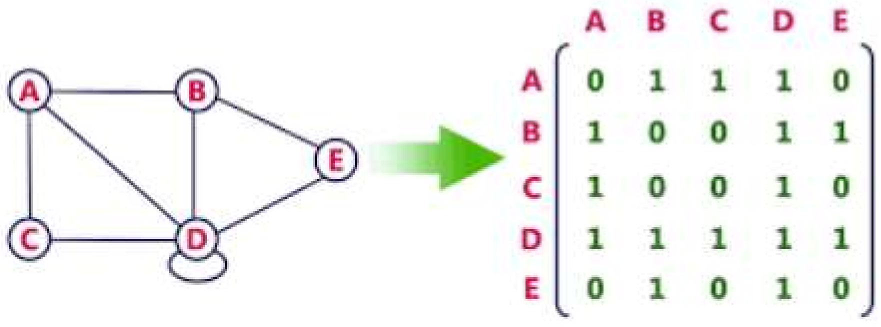
1. **Directed graph**: In directed graph, edges have a specific direction. Edge (a, b) and Edge (b, a) have different meaning in case of directed graph.
2. **Undirected**: In undirected graph, edges don't represent a specific direction. Edge (a, b) and edge (b, a) is equivalent in case of undirected graph.
3. **Weighted Graphs:** Weights are associated with the edges connecting two vertices. For example, if the two vertices are two cities, then edge between them may represent the distance between those two cities. If the nodes represent the two workstations in a network, then edge between two nodes may represent the cost to reach a workstation from other one.

4. **Unweighted Graphs**: Weights are not associated with the edges connecting two vertices. If there one undirected edge is present between two nodes that simply means that these two nodes are related.

Representation of Graphs: These graphs can be stored in two ways:

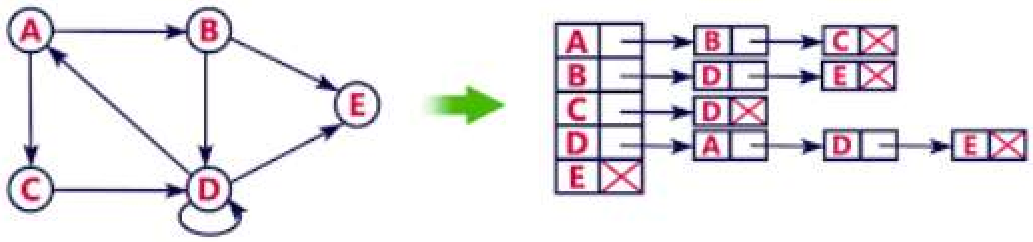
1. Sequential or Adjacency matrix Representation
2. Linked list or Adjacency List Representation

Adjacency matrix Representation: The graph G can be represented in the form of matrix representation and the weights associated with the edge from one node (node i) to the other (node j) forms the entries (Aij) of the matrix A.



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 1 & 0 \\ B & 1 & 0 & 0 & 1 & 1 \\ C & 1 & 0 & 0 & 1 & 0 \\ D & 1 & 1 & 1 & 1 & 1 \\ E & 0 & 1 & 0 & 1 & 0 \end{array}$$

Adjacency list Representation: An adjacency list is used in the linked representation to store the Graph in the computer's memory. It is efficient in terms of storage as we only have to store the values for edges.



Graph Traversal: The graphs can be traversed either by Breadth first search (BFS) or Depth first search (DFS). In BFS we traverse from the root node to the other nodes in a breadth first manner and uses queue to go to its next node. Depth first search (DFS) traverses the whole graph depth-wise and uses stack to get to the next nodes.

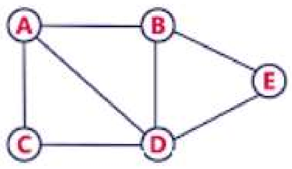
Depth First Search: *A, B, E, D, C*

Breadth First Search: *A, B, C, D, E*

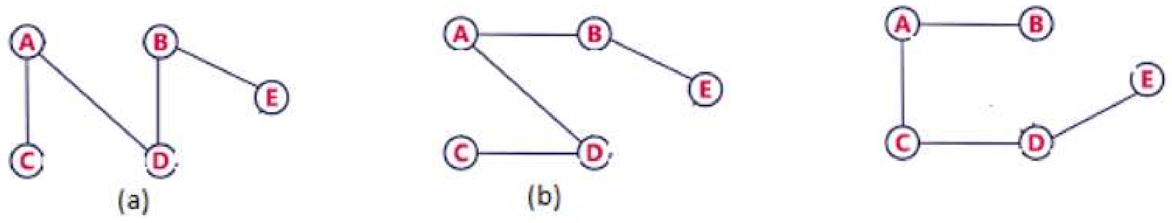
Spanning Tree:

A spanning tree of a graph is the subset if a graph which covers all the vertices with minimum number of possible edges. Spanning tree is used to find the minimum path connecting all nodes in a graph.

A spanning tree does not have cycles and it cannot be disconnected.



Graph G



Spanning trees of graph G

Observations/Outcomes:

1. Understand the properties of graph data structures.
2. Learning the different types of graph representations.
3. Implementation of DFS and BFS traversal.

Problems on:

1. Implement the adjacency list representation of a graph with m vertices and n edges. Also check whether a path of a given length exist between any two pair of vertices or not.
2. Find the minimum spanning tree of an undirected Graph using greedy approach.

Summarized Course Outcome [CO7]: Better understanding of graph properties, representations, and efficient implementation of Graph traversal methods.

LABORATORY-VIII (SEARCHING AND SORTING)

Objective:

- To search an element from a list of elements.
- To perform sorting using different sorting techniques.

Brief Theory:

Searching is the method to retrieve an element or data from any data structure. The searching techniques are broadly classified into:

Linear Search: In Linear search we try to retrieve an element by checking the array elements in one-by-one fashion. It takes time to perform the searching operation as it requires the whole array of size to be traversed for checking the elements.

Binary Search: This searching technique requires the array to be sorted before performing the search operation. The sorted array is given as input, which is divided into two subarrays of equal size. The element is first searched with the middle element of the array, if the element is larger than the middle element, then it will lie in the right sub array else search in the left subarray. This step is performed repeatedly until the item is found.

The time complexity for Binary search is $O(\log n)$, where **n** is the size of the array.

Sorting: Sorting refers to arranging the given data in a specified order. It requires the element to follow a particular order e.g., Roll numbers of students are arranged in ordered way. There are two different categories in sorting. They are:

- **Internal Sorting:** When all data is placed in memory, then sorting is called internal sorting.
- **External Sorting:** When all data that needs to be sorted cannot be placed in memory at a time, the sorting is called External Sorting. External Sorting is used for massive amount of data.

There are various types of sorting techniques available:

Bubble Sort: It is a simple comparison-based sorting technique. In this sorting technique each element is compared with its adjacent element and the elements are swapped if it's not in order. The time complexity bubble sort is $O(n^2)$ where n is the size of the array. It is not suitable for large data size.

Insertion Sort: It is a comparison-based sorting technique where one of the sub arrays is always sorted. For an element to be placed into the sorted subarray, an appropriate place needs to be found to insert the element in the sorted sub array. The worst-case time complexity for an Insertion sort is $O(n^2)$. The number of comparisons in the best case is $O(n)$ and in worst case is $O(n^2)$.

Selection Sort: Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty, and the unsorted part is the entire list. The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right. This algorithm is not suitable for large data sets as its average and worst-case complexities are $O(n^2)$ of where **n** is the number of items.

Merge Sort: It follows the Divide and Conquer approach for sorting an array. In merge sort, an array is first divided into two halves. The two sub arrays are then sorted and merged to form the complete sorted array. The sorting of subarrays continues till the number of elements in the sub array reduces to 1.

Quick sort: It follows Divide and Conquer approach to sort the array. Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made, and another array holds values greater than the pivot value. Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are $O(n^2)$, respectively and the best-case complexity is "$O(n \log n)$".

Heap Sort: A heap is a complete binary tree, and the binary tree is a tree in which the node can have the utmost two children. Heapsort is a popular and efficient sorting algorithm. Heap sort processes the elements by creating the min-heap or max-heap using the elements of the given array. Min-heap or max-heap represents the ordering of array in which the root element represents the minimum or maximum element of the array. The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list. Heapsort is the in-place sorting algorithm. The time complexity of heap sort is $O(n \log n)$.

Observations/Outcome:

1. To understand the different searching techniques.
2. To understand the working of different types of sorting techniques.

Problems on:

1. Write a program to sort the elements of the given array in descending order using the logic of algorithm. Always select the median element of the array as pivot.
2. Implement Binary search for an array & check whether the given element is present in the array.
3. Modify the merge procedure of merge sort to perform the merging of 3 sorted arrays into a single array.
4. Write a function which sorts the numbers in array using the heapsort algorithm.

Summarized Course Outcome [CO8]: Better understanding and efficient implementation of various searching and sorting techniques.

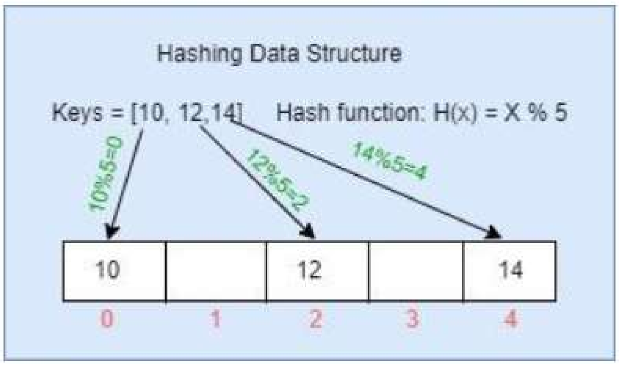
LABORATORY-IX (HASHING)

Objective:

To learn about hashing techniques.

Brief Theory:

Hashing is done for the faster access of elements. In hashing a given key is converted to a smaller value by the help of hash function, and then the given key is stored at the index. Basically, the goal of the hash function is determining the hash value of a particular key, and then that hash value will be used as index to store that particular key. Generally, the hash function are simple modular functions, which can determine the hash value at hence the element can be stored and access in time using hashing. Below picture is depicting a hashing scheme.



Collision: If the hash value calculated by hah function is same for more than one key then those keys are indexed at same slot, this is known as collision. That collision needs to be handled properly; few collision handling techniques are:

1. Linear Probing.
2. Double Hashing.
3. Quadratic Hashing.

In the hashing scheme shown in the above picture if key value 15 needs to be inserted now, then that will lead to collision because 15 will also map to index 0.

Observations/Outcome:

1. Student will learn how to stores data efficiently so that those data can be accessed in very efficient manner.

Problems on:

1. Hashing Techniques.
2. Linear probing, double hashing, and quadratic hashing to resolve the collision.

Summarized Course Outcome [CO9]: Better understanding of hashing techniques, linear probing, double hashing, and quadratic hashing to resolve the collision.