LAB MANUAL OF

DATA STRUCTURES LAB

ETCS -255

Maharaja Agrasen Institute of Technology, PSP area,
Sector – 22, Rohini, New Delhi – 110085

(Affiliated to Guru Gobind Singh Indraprastha University, New Delhi)

1. Introduction to the Lab

Lab Objective

The essential goals of doing data structures lab is to acquire skills and knowledge in imperative programming. Data structure is structuring and organizing data in efficient manner so that it can be accessed and modified easily. It determines the logical linkages between data elements and affect the physical processing of data. All software programs use data structures of some kind.

Course Outcomes

At the end of the course, a student will be able to:

ETCS255.1: Create arrays and apply various operations on the array created.
ETCS 255.2: Make use of link list to implement the concept of dynamic memory allocation and apply various operations on link list.
ETCS 255.3: Make use of link list/ array to create stack and queue with contiguous and non-contiguous data storage mechanism.
ETCS 255.4: Construct a binary tree and apply various operations on it by utilizing the concepts of basic data structures.
ETCS 255.5: Make use of appropriate data structures to create graph and apply various traversal techniques on it.
ETCS 255.6: Apply various sorting techniques on 1-D array.

2. LAB REQUIREMENTS

Hardware Detail

Intel i3/C2D Processor/2 GB RAM/500GB HDD/MB/Lan Card/ Key Board/ Mouse/CD Drive/15" Color Monitor/ UPS	24 Nos
LaserJet Printer	1 No

Software Detail

Cent OS/Fedora Linux

Code Blocks

3. LIST OF EXPERIMENTS (As prescribed by G.G.S.I.P.U)

DATA STRUCTURES LAB

Paper Code: ETCS-255 **Paper: Data Structures Lab**

List of Experiments

1. Perform Linear Search and Binary Search on an array.
Description of programs:
a. Read an array of type integer.
b. Input element from user for searching.
c. Search the element by passing the array to a function and then returning the position of the element from the function else return -1 if the element is not found.
d. Display the position where the element has been found.

2. Implement sparse matrix using array.
Description of program:
a. Read a 2D array from the user.
b. Store it in the sparse matrix form, use array of structures.
c. Print the final array.

3. Create a linked list with nodes having information about a student and perform I. Insert a new node at specified position.
II. Delete of a node with the roll number of student specified. III. Reversal of that linked list.

4. Create doubly linked list with nodes having information about an employee and perform Insertion at front of doubly linked list and perform deletion at end of that doubly linked list.

5. Create a circular linked list having information about a college and perform Insertion at front and perform Deletion at end.

6. Create a stack and perform Pop, Push, Traverse operations on the stack using Linear Linked list.

7. Create a Linear Queue using Linked List and implement different operations such as Insert, Delete, and Display the queue elements.

8. Create a Binary Tree (Display using Graphics) perform Tree traversals (Preorder, Postorder, Inorder) using the concept of recursion.

9. Implement insertion, deletion and display (inorder, preorder and postorder) on binary search tree with the information in the tree about the details of a automobile (type, company, year of make).

10. To implement Insertion sort, Merge sort, Quick sort, Bubble sort, Bucket sort, Radix sort, Shell sort, Selection sort, Heap sort and Exchange sort using array as a data structure.

NOTE:- At least 8 Experiments out of the list must be done in the semester.

4. LIST OF EXPERIMENTS (Beyond the syllabus)

DATA STRUCTURES LAB

Paper Code: ETCS-255 **Paper: Data Structures Lab**

List of Experiments

1. To implement concept of Pointers.
2. Create a circular Queue and implement different operations such as Insert, Delete, and Display the queue elements.
3. Write a program to implement priority queue.
4. WAP to Implement Depth-First-Search in a graph.
5. WAP to Implement Breadth-First-Search in graph.

5. FORMAT OF THE LAB RECORD TO BE PREPARED BY THE STUDENTS

The front page of the lab record prepared by the students should have a cover page as displayed below.

DATA STRUCTURES

ETCS-255

Font should be (Size 20", italics bold, Times New Roman)

Faculty name

Student name

Roll No.:

Semester:

Font should be (12", Times Roman)

Maharaja Agrasen Institute of Technology, PSP Area,

Sector – 22, Rohini, New Delhi – 110085

Font should be (18", Times Roman)

Index

Exp. no	Experiment Name	Date of performance	Date of checking	Marks	Signature

MARKING SCHEME FOR THE PRACTICAL EXAMS

There will be two practical exams in each semester.

i. Internal Practical Exam
ii. External Practical Exam

INTERNAL PRACTICAL EXAM

It is taken by the respective faculty of the batch.

MARKING SCHEME FOR THIS EXAM IS:

Total Marks: 40

Division of 10 marks per practical is as follows:

	Rubrics for : Laboratory (General)			
Sr No.	Experiment Component (LAC)	Max. Marks	Grading Rubrics	
			2 marks	1 mark
1	Practical Performance	2	Completeness of practical, exhibits proficiency in using different types of inputs.	Incomplete practical, unformatted, lacks comments, Demonstrates no proficiency.
2	Output and Validation	2	Output is free of errors and output is obtained. Demonstrates excellent understanding of the concepts relevant to the experiment.	Output contains few logical errors and/or no output is obtained. Demonstrates partial understanding of the concepts relevant to the experiment.
3	Attendance and Viva Questions Answered	4	1.Four marks for answering more than 75% questions. 2. Three marks for answering more than 60% questions. 2. Two marks for answering more than 50% questions. 3. One mark for answering less than 50% questions.	
4	Timely Submission of Lab Record	2	On time submission	Late submission

Each experiment will be evaluated out of 10 marks. At the end of the semester average of 8 best performed practical will be considered as marks out of 40.

EXTERNAL PRACTICAL EXAM

It is taken by the concerned lecturer of the batch and by an external examiner. In this exam student needs to perform the experiment allotted at the time of the examination, a sheet will be given to the student in which some details asked by the examiner needs to be written and at the last viva will be taken by the external examiner.

MARKING SCHEME FOR THIS EXAM IS:

Total Marks:	60

Division of 60 marks is as follows

1.	Sheet filled by the student:	20
2.	Viva Voice:	15
3.	Experiment performance:	15
4.	File submitted:	10

NOTE:

- Internal marks + External marks = Total marks given to the students
  (40 marks) (60 marks) (100 marks)

- Experiments given to perform can be from any section of the lab.

6. INSTRUCTIONS FOR EACH LAB EXPERIMENT

Experiment 1

Aim: Perform Linear Search and Binary Search on an array. Description of programs:
a. Read an array of type integer.
b. Input element from user for searching.
c. Search the element by passing the array to a function and then returning the position of the element from the function else return -1 if the element is not found.
d. Display the position where the element has been found.

Theory:
The search operation can be done in the following two ways:

Linear search:

It's a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

Pseudo code for Linear search:

```
procedure  linear_search (list, value)
  for each item in the list
    if match item == value
      return the item's location
    end if
  end for
end procedure
```

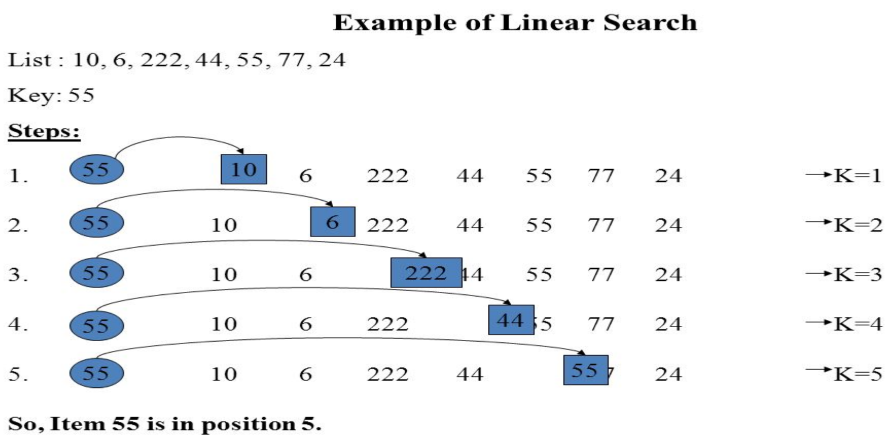
Binary search:

This search algorithm works on the principle of divide and conquers. For this algorithm, the data collection should be in the sorted form. Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the sub array reduces to zero.

pseudo code for Binary search:

```
BinarySearch(A[0..N-1], value)
 {
     low = 0
     high = N - 1
     while (low <= high) {
         // invariants: value > A[i] for all i < low
                        value < A[i] for all i > high
         mid = (low + high) / 2
         if (A[mid] > value)
             high = mid - 1
         else if (A[mid] < value)
             low = mid + 1
         else
             return mid
     }
     return not_found // value would be inserted at index "low"
  }
```

Example:



Example of Linear Search

List : 10, 6, 222, 44, 55, 77, 24

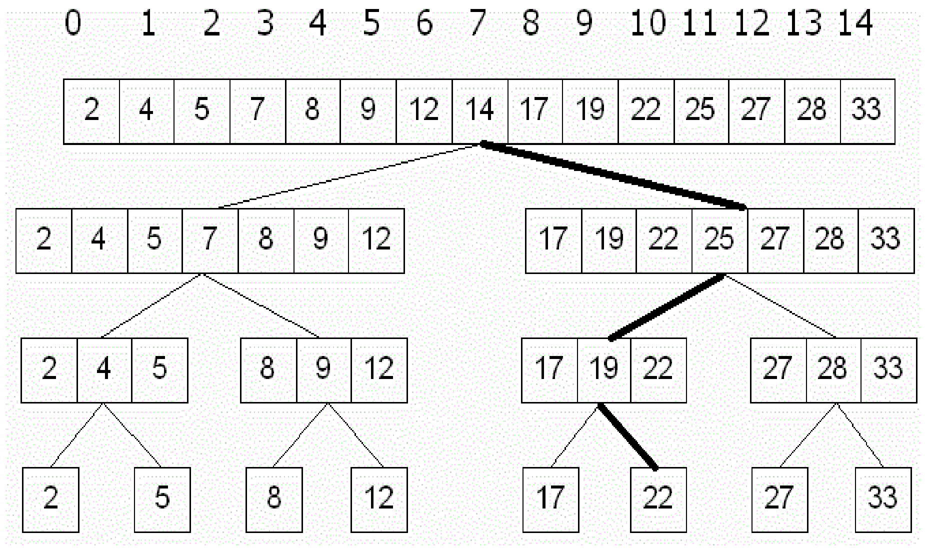
Key: 55

Steps:

1. 55 10 6 222 44 55 77 24 →K=1
2. 55 10 6 222 44 55 77 24 →K=2
3. 55 10 6 222 44 55 77 24 →K=3
4. 55 10 6 222 44 55 77 24 →K=4
5. 55 10 6 222 44 55 77 24 →K=5

So, Item 55 is in position 5.

Example of Binary search:



<u>Performance instructions:</u>

A menu driven program has to be created, where the user will enter the choice of the search operation to be performed and the set of data on which search operation has to be done.

The position of the element entered by the user, to be searched, is displayed as an output of the program.

Sample Input:

Enter the array:……………………….

Array eneterd is: 2,5,10,67,89

Enter the choice of search technique:

Press 1. For linear search and 2 for binary search
1.
Enter the number you want to search:
5

Sample Output: the required element is at position 2.

Viva Questions:

Q1. The sequential search, also known as ____________________

Q2. What is the primary requirement for implementation of binary search in an array?

Q3. Is binary search applicable on array and linked list both?

Q4. What is the principle of working of Binary search?

Q5. Which searching algorithm is efficient one?

Q6. Under what circumstances, binary search cannot be applied to a list of elements?

Experiment 2

Aim: Implement sparse matrix using array.

Description of program:

a. Read a 2D array from the user.
b. Store it in the sparse matrix form.
c. Print the final array.

Theory:

A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have 0 value, then it is called a sparse matrix.

Advantage of Sparse Matrix:

- Storage: There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
- Computing time: Computing time can be saved by logically designing a data structure traversing only non-zero elements

Pseudo code:

```
int main()
{
   // Assume 4x5 sparse matrix
   int sparseMatrix[4][5] =
   {
      {0 , 0 , 3 , 0 , 4 },
      {0 , 0 , 5 , 7 , 0 },
      {0 , 0 , 0 , 0 , 0 },
      {0 , 2 , 6 , 0 , 0 }
   };

   int size = 0;
   for (int i = 0; i < 4; i++)
      for (int j = 0; j < 5; j++)
         if (sparseMatrix[i][j] != 0)
            size++;

   // number of columns in compactMatrix (size) must be
   // equal to number of non - zero elements in
   // sparseMatrix
   int compactMatrix[3][size];

   // Making of new matrix
   int k = 0;
   for (int i = 0; i < 4; i++)
      for (int j = 0; j < 5; j++)
         if (sparseMatrix[i][j] != 0)
         {
            compactMatrix[0][k] = i;
            compactMatrix[1][k] = j;
```

```
                compactMatrix[2][k] = sparseMatrix[i][j];
                k++;
            }

    for (int i=0; i<3; i++)
    {
        for (int j=0; j<size; j++)
            printf("%d ", compactMatrix[i][j]);

        printf("\n");
    }
    return 0;
}
```

Sample input:

```
    0 , 0 , 3 , 0 , 4
    0 , 0 , 5 , 7 , 0
    0 , 0 , 0 , 0 , 0
    0 , 2 , 6 , 0 , 0
```

Sample Output:

```
     0 0 1 1 3 3

     2 4 2 3 1 2

     3 4 5 7 2 6
```

Viva - Questions:

Q1. What is sparse matrix?

Q2. What are the different ways of representing sparse matrix in memory?

Q3. What is the advantage of representing only non-zero values in sparse matrix?

Q4. Which data structure is used to implement a matrix?

Q5. What are various types of representation of matrix?

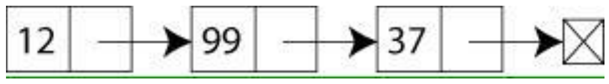
Q6. How much space does a tria-diagonal sparse matrix take when it is stored in an array.

Experiment 3

Aim: Create a linked list with nodes having information about a student and perform

I. Insert a new node at specified position.
II. Delete of a node with the roll number of student specified.
III. Reversal of that linked list.

Theory:
Linked list consists of a sequence of nodes, each containing data fields and one or two references ("links") pointing to the next and/or previous nodes. A linked list is a self-referential data type because it contains a pointer or link to another data of the same type. Linked lists permit insertion and removal of nodes at any point in the list in constant time, but do not allow random access.
The simplest kind of linked list is a singly-linked list, which has one link per node. This link points to the next node in the list, or to a null value or empty list if it is the final node.



Pseudo code:

```
// A linked list node
struct Node
{
  int data;                      //contains data about the student for example roll no
  struct Node *next;             //points towards the next node of linked lsit
};
//creation of linked list
{
    /* 1. allocate node */
    struct Node* new_node = new (Node);

    /* 2. put in the data  */
    new_node->data  = new_data;

    /* 3. Make next of new node as Null, as the given new node is last node of the created linked
lsit */
    new_node->next = Null;

    /* 4. move the head to point to the new node */
head=start
while (head->next!=NULL)
    {
      head=head->next;
    }

            Head->next=new_node;
}

// insertion at a specified position

/* Given a node prev_node, insert a new node after the given
   prev_node */
{

    /* 1. allocate newnode */
    struct Node* new_node =new(node);
      /* 2. put in the data  */
    new_node->data  = new_data;
      /*3. check if the given prev_node is NULL */
    if (prev_node == NULL)
    {
      new_node->next=start;
            Start=newnode
    }

    /* 4. Make next of new node as next of prev_node */
    new_node->next = prev_node->next;

    /* 5. move the next of prev_node as new_node */
    prev_node->next = new_node;
}
```

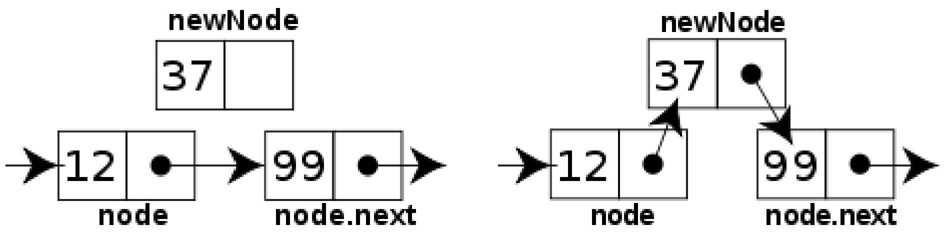
//**Deletion at specified position**

```
{
        /* traverse the linked list till the required roll number is fetched*/
        head=start
        While(head!=null && head->data!= data)
        {
                Prev_node=head;
                head=head->data;
        }
        /*delete the given node*/
        Prev_node->next=head->next;
}
```

//**reversal of linked list**

```
{
    struct Node* prev   = NULL;
    struct Node* current = *head_ref;
    struct Node* next;
    while (current != NULL)
    {
        next  = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
}
```

Example:



Performance instructions:

A menu driven program has to be created, where the user will enter the choice of the operation to be performed. A user can insert, delete, display or reverse the linked list.

The updated linked list according to user's choice will be displayed as output.

Sample Input :

Enter the operation you want to perform: 1. Insert 2. Delete 3. Reverse 4. Display 5. Exit

Operation: 1

Enter the value you want to add: 67

Do you want to continue: 1. Yes   2 no

…………………………………

Sample Output:

The elements of linked list is: 23->56->67->89

Viva-questions:

Q1. What type of memory allocation is referred for Linked lists?

Q2. Describe what is Node in link list? Name the types of Linked Lists?

Q3. What is the difference between Linear Array and Linked List?

Q4. Mention the steps to insert data at the starting of a singly linked list?

Q5. For which header list, you will found the last node contains the null pointer?

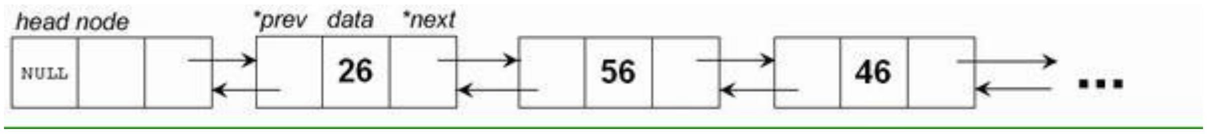
Q6.When we should prefer linked list over arrays?

Experiment-4

Aim: Create doubly linked list with nodes having information about an employee and perform Insertion at front of doubly linked list and perform deletion at end of that doubly linked list.

Theory:

A doubly linked list is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains two fields, called links, that are references to the previous and to the next node in the sequence of nodes. The previous and next links of beginning and ending nodes, respectively, point to some kind of terminator, a null value, to facilitate traversal of the list. The two node links allow traversal of the list in either direction. While adding or removing a node in a doubly linked list requires changing more links than the same operations on a singly linked list.



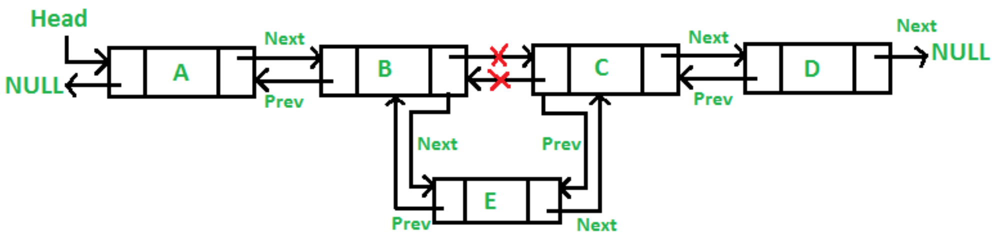
<u>**Pseudo code**</u>:

```
// A linked list node
struct Node
{
  int data;                  //contains information of employee
  struct Node *next;    //points to next node of linked list
struct Node *prev;      //points to previous node of linked list
};
//insertion at beginning

insertAtFront(data)
{
    if head is null
        head = new Node(data);
        // Tail and head cannot point at the same node
        tail = null;
    else:
        if tail is null:
            tail = new Node(data);
            // Update references
            // Head --> Tail
            // Head <-- Tail
            head.setNext(tail)
            tail.setPrev(head)
        else:
            Node<T> prevHead = head;
            Node<T> newHead = new Node(data);
            // Update references
            // newHead.next --> prevHead
            // prevHead.prev <-- newHead
            newHead.setNext(prevHead)
            prevHead.setPrev(newHead)
            head = newHead
        end if else;
    end if else;
    increment size
}
```

//deletion at the end

```
removeFromBack()
{
    if head is not null:
        if tail is not null:
            // Two or more elements exist
            newTail = tail.getPrev()
            tail = null
            newTail.setNext(null)
            tail = newTail
        else:
            // Tail is null but head is not null
            // means that this is the last element
            head = null
        end if else;
        decrement size
    end if;
}
```

Example:



<u>Performance instructions:</u>

A menu driven program has to be created, where the user will enter the choice of the operation to be performed. A user can insert at the beginning, delete the last node, and display the doubly linked list. The updated linked list according to user's choice will be displayed as output.

Sample Input :

Enter the operation you want to perform: 1. Insert 2. Delete 3. Display 4. Exit

Operation: 1

Enter the value you want to add: 67

Do you want to continue: 1. Yes 2 no

…………………………………

Sample Output:

67-78-89

Viva-Questions:

Q1. What is the advantage of using doubly linked list?

Q2. How many pointers will be affected if a node gets deleted from the middle position?

Q3. Is there any Null pointer in doubly linked list?

Q4. What is the difference between singly and doubly linked lists?

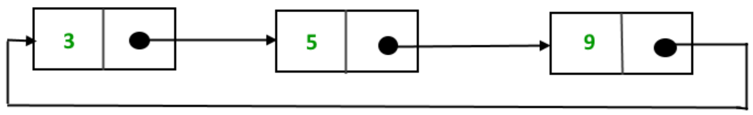
Q5. In what condition does the start pointer value gets changed?

Experiment-5

Aim: Create a circular linked list having information about a college and perform Insertion at front and perform Deletion at end.

Theory:

In a circularly-linked list, the first and final nodes are linked together. This can be done for both singly and doubly linked lists. To traverse a circular linked list, we can begin at any node and follow the list in either direction until we return to the original node. Viewed another way, circularly-linked lists can be seen as having no beginning or end



Pseudo-code:

// insert node in an empty List

```
struct Node *addToEmpty(struct Node *last, int data)

{
    // This function is only for empty list
    if (last != NULL)
      return last;

    // Creating a node dynamically.
    struct Node *last =
          (struct Node*)malloc(sizeof(struct Node));

    // Assigning the data.
    last -> data = data;

    // Note : list was empty. We link single node
    // to itself.
    last -> next = last;

    return last
}
```

//insertion at the front position

```
struct Node *addBegin(struct Node *last, int data)
{
  if (last == NULL)
     return addToEmpty(last, data);

  // Creating a node dynamically.
  struct Node *temp
        = (struct Node *)malloc(sizeof(struct Node));

  // Assigning the data.
  temp -> data = data;

  // Adjusting the links.
  temp -> next = last -> next;
  last -> next = temp;

  return last;
}
```

//delete a node

```
void deleteNode(struct Node *head, int key)
{
    if (head == NULL)
        return;
     // Find the required node
    struct Node *curr = head, *prev;
    while (curr->data != key)
    {
        if (curr->next == head)
        {
            printf("\nGiven node is not found"
                   " in the list!!!");
            break;
        }
         prev = curr;
        curr = curr -> next;
    }

    // Check if node is only node
    if (curr->next == head)
    {
        head = NULL;
        free(curr);
        return;
    }

    // If more than one node, check if
    // it is first node
    if (curr == head)
    {
        prev = head;
        while (prev -> next != head)
            prev = prev -> next;
        head = curr->next;
        prev->next = head;
        free(curr);
    }

    // check if node is last node
    else if (curr -> next == head)
    {
        prev->next = head;
        free(curr);
    }
    else
    {
        prev->next = curr->next;
        free(curr);
    }
}
```

Performance instructions:

A menu driven program has to be created, where the user will enter the choice of the operation to be performed. A user can insert at the beginning, delete a node, and display the circular linked list.

The updated linked list according to user's choice will be displayed as output.

Sample Input :

```
// stored linked list is 67-89-90-34-56

Enter the operation you want to perform: 1. Insert   2. Delete   3. Display  4. Exit

Operation: 2

Do you want to continue: 1. Yes     2 no

…………………………………
```

Sample Output:

```
67-89-90-34
```

Viva-Questions:

Q1. Is there any Null pointer in circular linked list?

Q2. What is circular linked list?

Q3.whethear a circular linked list is single way list or two way list?

Q4.Which pointer signifies the starting of circular linked list?

Q5. Describe the steps to delete the starting node of linked list.

Experiment-6

Aim: Create a stack and perform Pop, Push, Traverse operations on the stack using Linear Linked list.

Theory: A stack is a data structure based on the principle of Last In First Out *(LIFO)*. Mainly the following three basic operations are performed in the stack:

- Push: Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- Pop: Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.

Algorithm:

push(value) - Inserting an element into the Stack

```
Step 1: Create a newNode with given value.
Step 2: Check whether stack is Empty (top == NULL)
Step 3: If it is Empty, then set newNode → next = NULL.
Step 4: If it is Not Empty, then set newNode → next = top.
Step 5: Finally, set top = newNode.
```

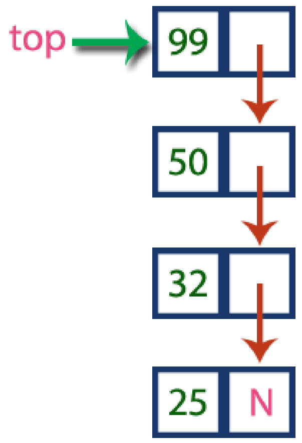
pop() - Deleting an Element from a Stack

```
Step 1: Check whether stack is Empty (top == NULL).
Step 2: If it is Empty, then display "Stack is Empty!!! Deletion is not possible!!!" and terminate the function
Step 3: If it is Not Empty, then define a Node pointer 'temp' and set it to 'top'.
Step 4: Then set 'top = top → next'.
Step 7: Finally, delete 'temp' (free(temp)).
```

display() - Displaying stack of elements

```
Step 1: Check whether stack is Empty (top == NULL).
Step 2: If it is Empty, then display 'Stack is Empty!!!' and terminate the function.
Step 3: If it is Not Empty, then define a Node pointer 'temp' and initialize with top.
Step 4: Display 'temp → data --->' and move it to the next node. Repeat the same until temp reaches to the first node in the stack (temp → next != NULL).
Step 4: Finally! Display 'temp → data ---> NULL'.
```

Example:



In above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32,50 and 99.

Performance instructions:

A menu driven program has to be created, where the user will enter the choice of the operation to be performed. A user can push an element, pop an element from the stack or display the elements of the stack.

The stack elements will be displayed in last in first out order.

Sample Input :

Enter the operation you want to perform: 1. Push   2. Pop   3. Display 4. Exit

Operation: 1

Enter the element : 56

Do you want to continue: 1. Yes   2 no

…………………………………

Sample Output:

//elements are entered in order 78,89,90,45

Stack elements are: 56,45,90,89,78

Viva Questions:

Q1. What is the principle of working of stack?

Q2. Give any application of stack.

Q3. What are the various operations that can be applied over stack?

Q4. Which type of data structure is used to implement stack?

Q5. Which type of memory allocation does stack uses?

Experiment-7

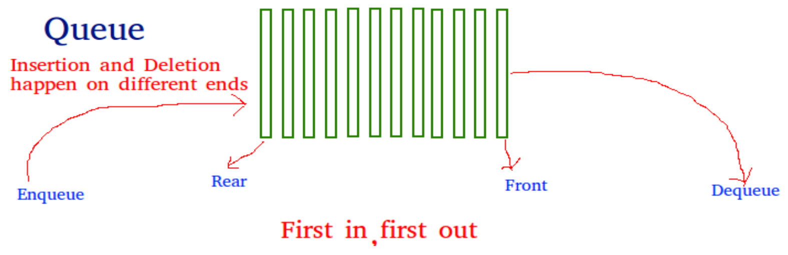
Aim: Create a Linear Queue using Linked List and implement different operations such as Insert, Delete, and Display the queue elements.

Theory:

Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

Operations on Queue : Mainly the following four basic operations are performed on queue:
Enqueue: Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.
Dequeue: Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.

Front: Get the front item from queue.
Rear: Get the last item from queue.



Algorithm:

enQueue(value) - Inserting an element into the Queue

```
    Step 1: Create a newNode with given value and set 'newNode → next' to NULL.
    Step 2: Check whether queue is Empty (rear == NULL)
    Step 3: If it is Empty then, set front = newNode and rear = newNode.
    Step 4: If it is Not Empty then, set rear → next = newNode and rear = newNode.
```

deQueue() - Deleting an Element from Queue

```
    Step 1: Check whether queue is Empty (front == NULL).
    Step 2: If it is Empty, then display "Queue is Empty!!! Deletion is not possible!!!" and
    terminate from the function
    Step 3: If it is Not Empty then, define a Node pointer 'temp' and set it to 'front'.
    Step 4: Then set 'front = front → next' and delete 'temp' (free(temp)).
```

display() - Displaying the elements of Queue

```
    Step 1: Check whether queue is Empty (front == NULL).
    Step 2: If it is Empty then, display 'Queue is Empty!!!' and terminate the function.
    Step 3: If it is Not Empty then, define a Node pointer 'temp' and initialize with front.
```

Step 4: Display 'temp → data --->' and move it to the next node. Repeat the same until 'temp' reaches to 'rear' (temp → next != NULL).
Step 4: Finally! Display 'temp → data ---> NULL'.

Example:

```
queue
-------------
| a | b | c |
-------------
  ^       ^
  |       |
front    rear

----Now, Enter(queue, 'd')...

queue
-----------------
| a | b | c | d |
-----------------
  ^           ^
  |           |
front        rear

----Now, ch = Delete(queue)...

queue          ch
-------------  -----
| b | c | d |  | a |
-------------  -----
  ^       ^
  |       |
front    rear
```

<u>Performance instructions:</u>

A menu driven program has to be created, where the user will enter the choice of the operation to be performed. A user can add an element, delete an element from the queue or display the elements of the queue.

The queue elements will be displayed in first in first out order.

Viva-Questions:

Q1. Give the working principle of Queue.

Q2. Give any application of Queue.

Q3. What are the various operations that can be applied over Queue?

Q4. Which type of data structure is used to implement Queue?

Q5. Which type of memory allocation does Queue uses?

Experiment-8

Aim: Create a Binary Tree perform Tree traversals (Preorder, Postorder, Inorder) using the concept of recursion.

Theory:
A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child. Tree traversal is the process of visiting each node in a tree data structure. Tree traversal, also called walking the tree, provides for sequential processing of each node in what is, by nature, a non-sequential data structure. Such traversals are classified by the order in which the nodes are visited. There are three different ways of traversal such as pre-order, in-order and post-order traversal.

Preorder traversal: root, left child, right child

In-order traversal: left child, root, right child

Post-order traversal: left child, right child, root

Algorithm:

pre-order traversal

```
sub PreOrder(TreeNode)
  Output(TreeNode.value)
  If LeftPointer(TreeNode) != NULL Then
    PreOrder(TreeNode.LeftNode)
  If RightPointer(TreeNode) != NULL Then
    PreOrder(TreeNode.RightNode)
end sub
```

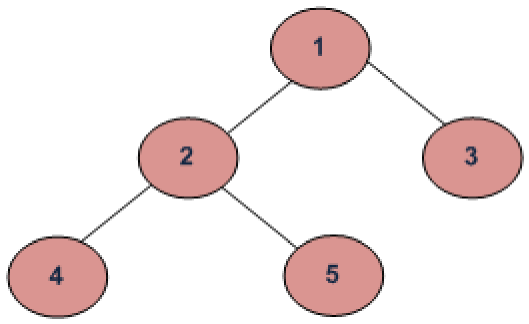
in-order traversal

```
sub InOrder(TreeNode)
  If LeftPointer(TreeNode) != NULL Then
    InOrder(TreeNode.LeftNode)
  Output(TreeNode.value)
  If RightPointer(TreeNode) != NULL Then
    InOrder(TreeNode.RightNode)
end sub
```

post-order traversal

```
sub PostOrder(TreeNode)
  If LeftPointer(TreeNode) != NULL Then
    PostOrder(TreeNode.LeftNode)
  If RightPointer(TreeNode) != NULL Then
    PostOrder(TreeNode.RightNode)
  Output(TreeNode.value)
end sub
```

Example:



(a) Inorder (Left, Root, Right) : 4 2 5 1 3

(b) Preorder (Root, Left, Right) : 1 2 4 5 3

(c) Postorder (Left, Right, Root) : 4 5 2 3 1

<u>**Performance instructions:**</u>

A binary tree is created. The user will enter the choice of the traversal procedure to be performed.

The elements of binary tree will be displayed. The order of elements will be according to the traversal technique chose by the user.

Viva –Questions:

Q1. How many child nodes there can be for any parent node in a binary tree?

Q2. How can you traverse a binary tree?

Q3. Does the last node of post-order traversal and first node of pre-order traversal of a binary tree always same?

Q4. How can you represent a tree in memory?

Q5. What is the order of nodes in in-order traversal?

Q6. Can a tree be called a graph also?

Experiment-9

Aim: Implement insertion, deletion and display (inorder, preorder and postorder) on binary search tree with the information in the tree about the details of a automobile (type, company, year of make).

Theory:

Binary Search Tree, is a node-based binary tree data structure which has the following properties:

The left subtree of a node contains only nodes with keys lesser than the node's key.

The right subtree of a node contains only nodes with keys greater than the node's key.

The left and right subtree each must also be a binary search tree.

Algorithm:

insertion in BST

Step 1: Create a newNode with given value and set its left and right to NULL.

Step 2: Check whether tree is Empty.

Step 3: If the tree is Empty, then set set root to newNode.

Step 4: If the tree is Not Empty, then check whether value of newNode is smaller or larger than the node (here it is root node).

Step 5: If newNode is smaller than or equal to the node, then move to its left child. If newNode is larger than the node, then move to its right child.

Step 6: Repeat the above step until we reach to a leaf node (e.i., reach to NULL).

Step 7: After reaching a leaf node, then isert the newNode as left child if newNode is smaller or equal to that leaf else insert it as right child.

Deletion Operation in BST

Deleting a node from Binary search tree has following three cases...

Case 1: Deleting a Leaf node (A node with no children)

Case 2: Deleting a node with one child

Case 3: Deleting a node with two children

Case 1: Deleting a leaf node

We use the following steps to delete a leaf node from BST...

Step 1: Find the node to be deleted using search operation

Step 2: Delete the node using free function (If it is a leaf) and terminate the function.

Case 2: Deleting a node with one child

We use the following steps to delete a node with one child from BST...

Step 1: Find the node to be deleted using search operation

Step 2: If it has only one child, then create a link between its parent and child nodes.

Step 3: Delete the node using free function and terminate the function.

We use the following steps to delete a node with two children from BST...

Step 1: Find the node to be deleted using search operation

Step 2: If it has two children, then find the largest node in its left subtree (OR) the smallest node in its right subtree.

Step 3: Swap both deleting node and node which found in above step.

Step 4: Then, check whether deleting node came to case 1 or case 2 else goto steps 2

Step 5: If it comes to case 1, then delete using case 1 logic.

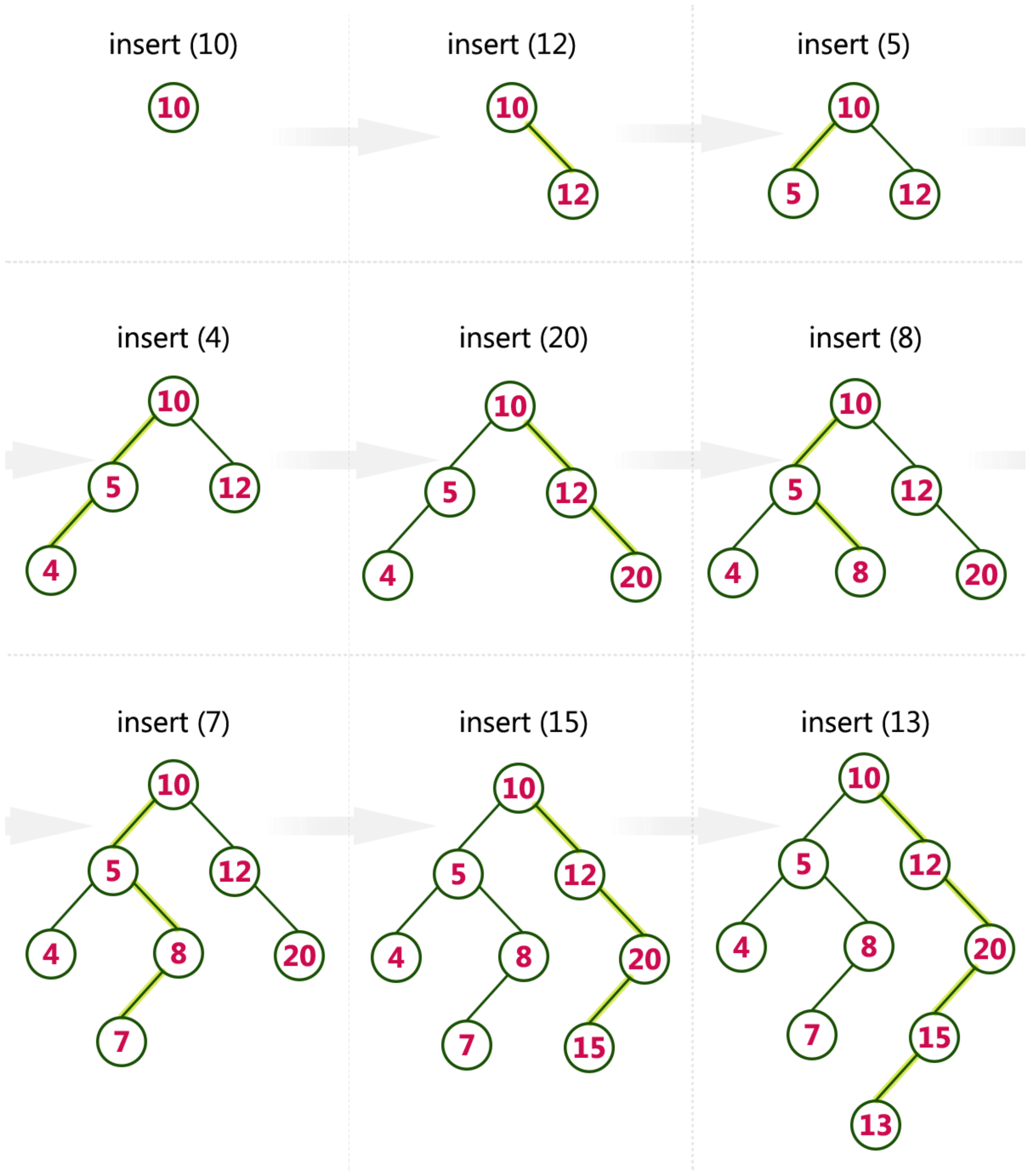
Step 6: If it comes to case 2, then delete using case 2 logic.

Step 7: Repeat the same process until node is deleted from the tree.

Example

Construct a Binary Search Tree by inserting the following sequence of numbers...

10,12,5,4,20,8,7,15 and 13



Viva-Questions:

Q1. What is BST?

Q2. Give any application of BST.

Q3. Wheather a binary search tree is balanced or not?

Q4. Which traversal will generate a ascending order list of nodes in BST?

Q5. How will you decide the root of the BST?

Experiment 10

Aim: To implement Insertion sort, Merge sort, Quick sort, Bubble sort, Bucket sort, Radix sort, Shell sort, Selection sort, Heap sort and Exchange sort using array as a data structure.

a Insertion sort:

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time.

Pseudocode:

```
i ← 1
while i < length(A)
    j ← i
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j ← j - 1
    end while
    i ← i + 1
end while
```

Example:

3 7 4 9 5 2 6 1

3 7 4 9 5 2 6 1

3 **7** 4 9 5 2 6 1

3 **4** 7 9 5 2 6 1

3 4 7 **9** 5 2 6 1

3 4 **5** 7 9 2 6 1

2 3 4 5 7 9 6 1

2 3 4 5 **6** 7 9 1

1 2 3 4 5 6 7 9

Viva Questions:

Q1. Consider a situation where swap operation is very costly. Which of the following sorting algorithms should be preferred so that the number of swap operations are minimized in general?

Q2. Which sorting algorithm will take least time when all elements of input array are identical? Consider typical implementations of sorting algorithms.

Q3. Consider an array of elements arr[5]= {5,4,3,2,1} , what are the steps of insertions done while doing insertion sort in the array.

Q4. Sort the given list using insertion sort

56.89.78.23.12.4.79.4.,36,29

B Merge sort

Conceptually, a merge sort works as follows:

1. Divide the unsorted list into *n* subsists, each containing 1 element (a list of 1 element is considered sorted).
2. Repeatedly merge subsists to produce new sorted subsists until there is only 1 subsist remaining. This will be the sorted list.

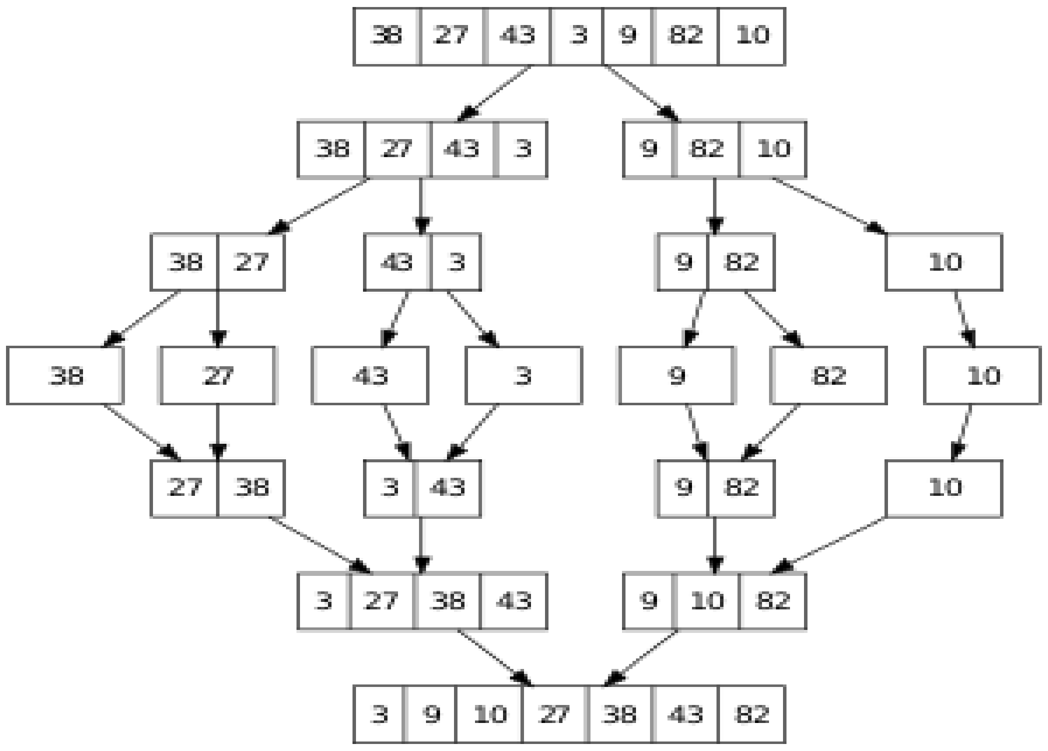
PSEUDOCODE:

Let A be an input array in which p is its lower bound and q is its upper bound. Function MERGE-SORT is called recursively, and it also call function MERGE which merges two unsorted arrays into a single sorted array.

```
MERGE-SORT (A, p, r)
  1. If p < r
  2. q = [(p + q) /2]
  3. MERGE-SORT (A, p, q)
  4. MERGE-SORT (A, q+1, r)
  5. MERGE (A, p, q, r)
MERGE (A, p, q, r)
  1. $n_1$ = q – p +1
  2. $n_2$ = r – q
  3. let L [1... $n_1$ + 1] and L [1... $n_2$ + 1] be new arrays
  4. for i=1 to $n_1$
  5. L [i] = A [ p + i -1]
  6. for j=1 to $n_2$
  7. R [j] =A [ q + j]
  8. L [$n_1$ + 1]= $\infty$
  9. R [$n_2$ + 1]= $\infty$
  10. i = 1
  11. j = 1
  12. for k = p to r
  13. if L [i] $\leq$ R [ j]
  14. A [k] = L [i]
  15. i = i + 1
  16. else A [k] = R [ j]
  17. j = j + 1
```

Sample Example:

A recursive merge sort algorithm used to sort an input array of 7 integer values. It is recursively dividing array into two subarrays and applying merge function to sort them.



Viva - Questions:

1. Why is time complexity of merge sort O(nlogn)

2. Is it possible to do merge sort in place?

3. What is the total number of passes in merge sort of n numbers?

4. Given two sorted lists of size m, n then what are the number of comparisons needed in the worst case by merge sort?

5. What is the output of merge sort after the 2nd pass given the following sequence of numbers: 3,41,52,26,38,57,9,49

6. Give any application of merge sort?

c. Quick sort

The basic steps in this sort are:

Divide: The array *A* [*p*… *r*] is partitioned (rearranged) into two nonempty subarrays *A* [*p*... *q*] and *A* [*q* + 1... *r*] such that each element of *A* [*p* . . *q*] is less than or equal to each element of *A* [*q* + 1... *r*]. The index *q* is computed as part of this partitioning procedure.

Conquer: The two subarrays *A* [*p*… *q*] and *A* [*q* + 1… *r*] are sorted by recursive calls to quicksort.

Combine: Since the subarrays are sorted in place, no work is needed to combine them: the entire array *A* [*p*.. *r*] is now sorted.

Pseudocode:

The following procedure implements quicksort. It takes A as input array which has p as its lower bound and r as its upper bound.

```
QUICKSORT (A,p,r)
        1 if p<r
        2     then q←PARTITION (A,p,r)
        3          QUICKSORT (A,p,q)
        4          QUICKSORT (A,q + 1,r)
```

To sort an entire array *A*, the initial call is QUICKSORT (*A*, 1, *length*[*A*]).

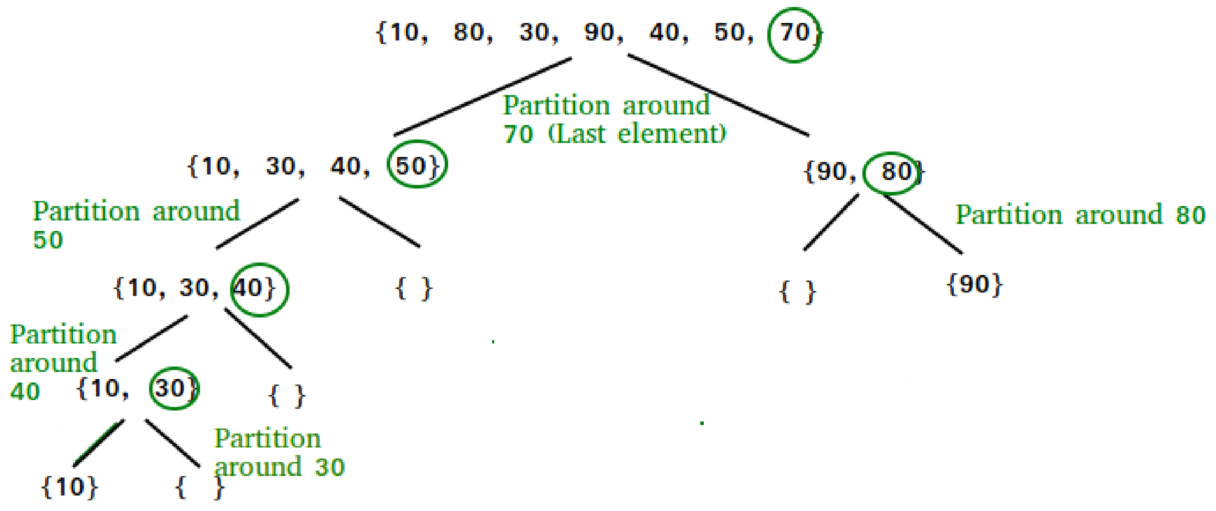
Partitioning the array:

The key to the algorithm is the PARTITION procedure, which rearranges the sub array *A* [*p*.. *r*] in place.

```
PARTITION (A,p,r)
        1  x←A[p]
        2  i←p - 1
        3 j←r + 1
        4 while TRUE
        5     do repeat j←j - 1
        6          until A[j] ≤x
        7        repeat i←i + 1
        8          until A[i] ≥x
        9        if i<j
        10           then exchange A[i] ↔A[j]
        11           else return j
```

Sample Example:

Consider an array of 7 elements and let last element is chosen as pivot. The following figure shows the sample execution of the partition procedure used in the quick sort.



It has partitioned the array in such a way that elements less than equal to pivot comes on its left side and elements greater than pivot are on its right side. This procedure is recursively called in the subarrays.

Viva- Questions:

1. What value of *q* does PARTITION return when all elements in the array A [*p* . . . *r*] have the same value?

2. What is the working principle of Quick sort?

3. How would you modify QUICKSORT to sort in non increasing order?

4. Why Quick sort is considered as best sorting?

D Bubble sort

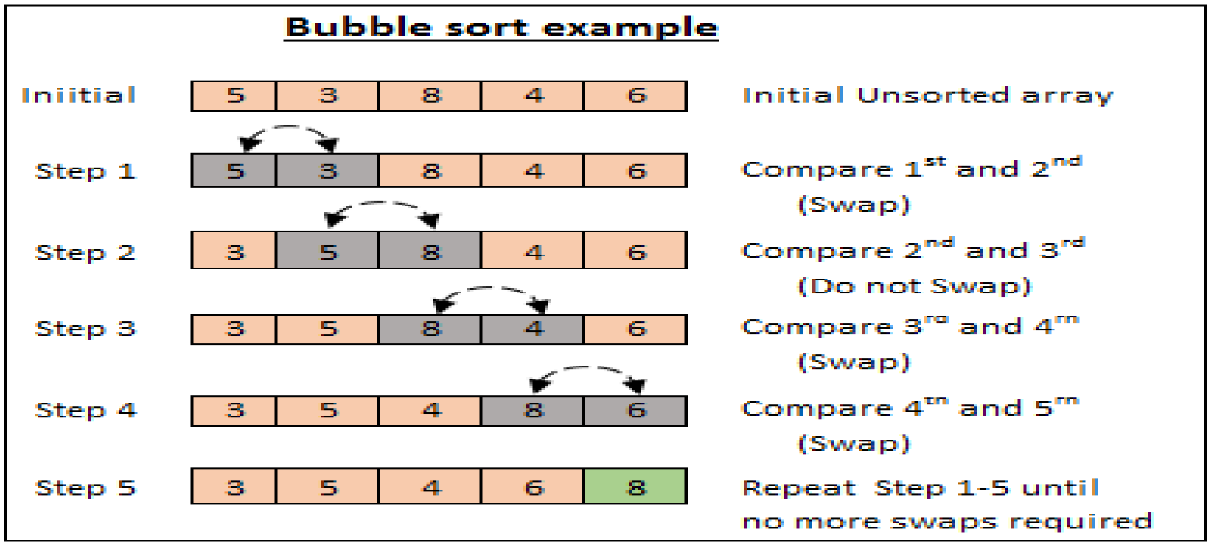
This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared, and the elements are swapped if they are not in order.

Pseudo code:

We assume list is an array of n elements. We further assume that swap function swaps the values of the given array elements.

```
begin BubbleSort (list)
  for all elements of list
    if list[i] > list[i+1]
      swap (list[i], list [i+1])
    end if
  end for
    return list
  end BubbleSort
```

Sample Example:



Bubble sort example
Iniitial
5 3 8 4 6
Initial Unsorted array
Step 1
5 3 8 4 6
Compare 1st and 2nd (Swap)
Step 2
3 5 8 4 6
Compare 2nd and 3rd (Do not Swap)
Step 3
3 5 8 4 6
Compare 3rd and 4th (Swap)
Step 4
3 5 4 8 6
Compare 4th and 5th (Swap)
Step 5
3 5 4 6 8
Repeat Step 1-5 until no more swaps required

Viva- Questions:

1. What is the best-case efficiency of bubble sort?

2. Is bubble sorting a stable sort?

3. How much time will bubble sort take if all the elements are same?

4. What would happen if bubble sort didn’t keep track of the number of swaps made on each pass through the list?

5. List main properties to be considered in bubble sort?

E Bucket sort

Bucket sort, is a sorting algorithm that works by distributing the elements of an array into a number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm. It is a distribution sort, a generalization of pigeonhole sort, and is a cousin of radix sort in the most-to-least significant digit flavour. Bucket sort can be implemented with comparisons and therefore can also be considered a comparison sort algorithm.

It works as follows:

1. Set up an array of initially empty "buckets".
2. Scatter: Go over the original array, putting each object in its bucket.
3. Sort each non-empty bucket.
4. Gather: Visit the buckets in order and put all elements back into the original array.

Pseudo code:

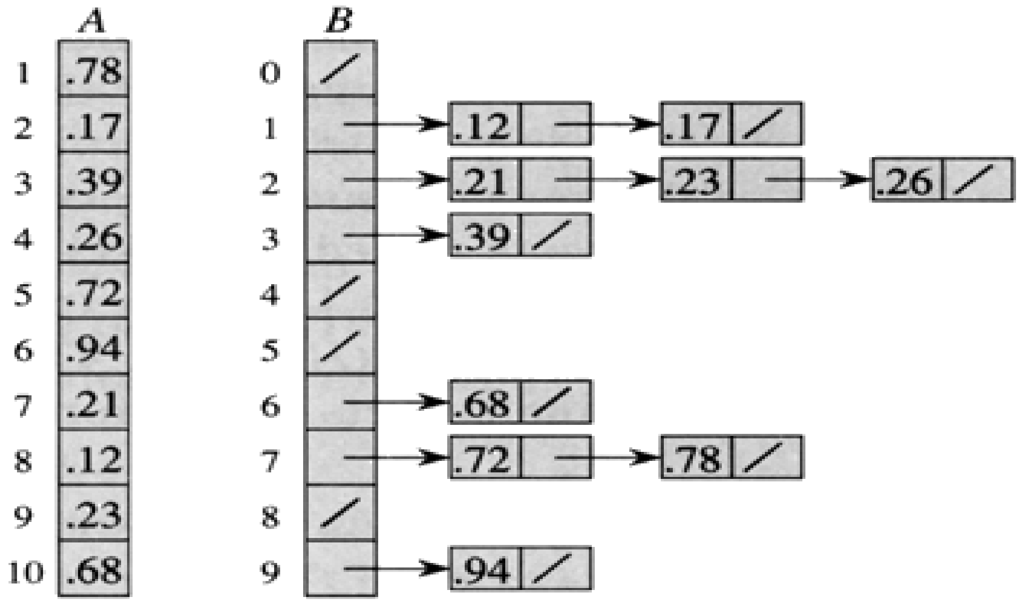
The code assumes that input is an n-element array A and each element in A satisfies $0 \le A[i] \le 1$. We also need an auxiliary array B [0 . . *n* -1] for linked-lists (buckets).

BUCKET SORT (A)

```
1. n ← length [A]
2. For i =1 to n do
3. Insert A[i]into list B [A[i]/b] where b is the bucket size
4. For i =0 to n-1do
5. Sort list B with Insertion sort
6. Concatenate the lists B [0], B [1] ... B [n-1] together in order.
```

Sample Example:

Given input array A [1...10]. The array B [0...9] of sorted lists or buckets after line 5. Bucket i holds values in the interval [*i*/10, (*i* +1)/10]. The sorted output consists of a concatenation in order of the lists first B [0] then B [1] then B[2] ... and the last one is B[9].



Viva - Questions:

1. Is the bucket sort in place sort? Why or why not?
2. Is bucket sort a stable sort?
3. Why bucket sort is good for large size arrays?
4. State any disadvantage of using this sort.
5. Can this sort be used for sorting negative numbers?

F Radix sort

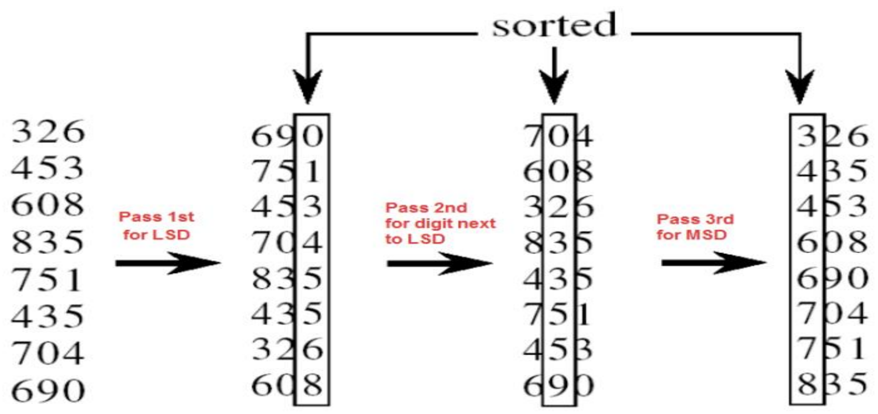
Radix Sort is a non-comparative sorting algorithm. It is one of the most efficient and fastest linear sorting algorithms. In radix sort, we first sort the elements based on last digit (least significant digit). Then the result is again sorted by second digit, continue this process for all digits until we reach most significant digit. We use counting sort to sort elements of every digit.

Let A be an array and d be number of digits.

Pseudo code:

```
Radix-Sort(A, d)
//It works same as counting sort for d number of passes.
//Each key in A [1...n] is a d-digit integer.
// (Digits are numbered 1 to d from right to left.)
    for j = 1 to d do
        //A[]-- Initial Array to Sort
        intcount[10] = {0};
        //Store the count of "keys" in count[]
        //key- it is number at digit place j
        for i = 0 to n do
         count[key of(A[i]) in pass j]++

        fork = 1 to 10 do
         count[k] = count[k] + count[k-1]

        //Build the resulting array by checking
        //new position of A[i] from count[k]
        for i = n-1 down to 0 do
         result[ count[key of(A[i])] ] = A[j]
         count[key of(A[i])]--

        //Now main array A[] contains sorted numbers
        //according to current digit place
        for i=0 to n do
          A[i] = result[i]

    end for(j)
end func
```

Sample Example:



In the above example:
For 1st pass: We sort the array on basis of least significant digit (1s place) using counting sort. Notice that 435 is below 835, because 435 occurred below 835 in the original list.
For 2nd pass: We sort the array on basis of next digit (10s place) using counting sort. Notice that here 608 is below 704, because 608 occurred below 704 in the previous list, and similarly for (835, 435) and (751, 453).
For 3rd pass: We sort the array on basis of most significant digit (100s place) using counting sort. Notice that here 435 is below 453, because 435 occurred below 453 in the previous list, and similarly for (608, 690) and (704, 751).

Viva- Questions:

1. Is Radix Sort preferable to Comparison based sorting algorithms like Quick-Sort?

2. Is this sort stable?

3. How much additional space is taken by this sort?

4. State any disadvantage of using this sort.

G Shell sort

It is a generalized version of insertion sort. It is an in–place comparison sort. It is also known as diminishing increment sort. This algorithm uses insertion sort on the large interval of elements to sort. Then the interval of sorting keeps on decreasing in a sequence until the interval reaches 1. These intervals are known as gap sequence.

Increment Sequences:

1. Shell's original sequence: N/2 , N/4 , …, 1 (repeatedly divide by 2);
2. Hibbard's increments: 1, 3, 7, …, $2^k – 1$;
3. Knuth's increments: 1, 4, 13, …, $(3^k – 1) / 2$;
4. Sedge wick's increments: 1, 5, 19, 41, 109…

Here interval is calculated based on Knuth's formula as −

Knuth's Formula

h = h * 3 + 1, where ->h is interval with initial value 1

Pseudo code:

```
Procedure shellSort ()
A: array of items
  /* calculate interval*/
  while interval < A. length /3 do:
    interval = interval * 3 + 1
  end while

  while interval > 0 do:
    for outer = interval; outer < A. length; outer ++ do:
    /* select value to be inserted */
ValueToInsert = A[outer]
    inner = outer;
      /*shift element towards right*/
      while inner > interval -1 &&A [inner - interval] >= valueToInsert do:
        A[inner] = A [inner - interval]
        inner = inner - interval
      end while
    /* insert the number at hole position */
    A[inner] = valueToInsert
    end for
  /* calculate interval*/
  interval = (interval -1) /3;
  end while
end procedure
```

Sample Example:

An example run of Shell sort with gaps 5, 3 and 1 is shown below.

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}
Input data	62	83	18	53	07	17	95	86	47	69	25	28
After 5-sorting	17	28	18	47	07	25	83	86	53	69	62	95
After 3-sorting	17	07	18	47	28	25	69	62	53	83	86	95
After 1-sorting	07	17	18	25	28	47	53	62	69	83	86	95

The first pass, 5-sorting, performs insertion sort on five separate sub arrays (a_1, a_6, a_{11}), (a_2, a_7, a_{12}), (a_3, a_8), (a_4, a_9), (a_5, a_{10}). For instance, it changes the sub array (a_1, a_6, a_{11}) from (62, 17, 25) to (17, 25, 62). The next pass, 3-sorting, performs insertion sort on the three sub arrays (a_1, a_4, a_7, a_{10}), (a_2, a_5, a_8, a_{11}), (a_3, a_6, a_9, a_{12}). The last pass, 1-sorting, is an ordinary insertion sort of the entire array (a_1... a_{12}).

Viva Questions:

1. State any application of shell sort?

2. Given the following list of numbers: [5,16,20,12,3,8,9,17,19,7]. What is the content of the list after all swapping is complete for a gap size of 3?

3. Is this sort stable and does it takes extra additional space.

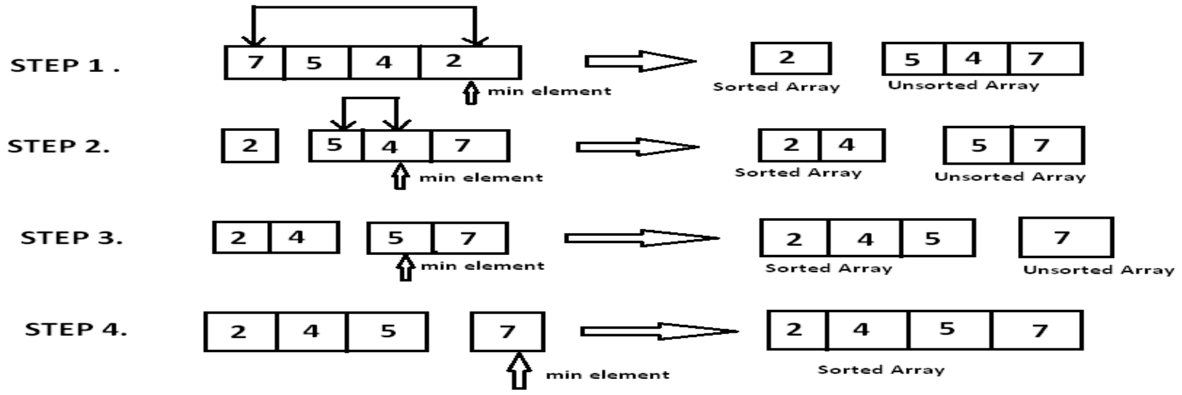
H Selection sort

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.

Pseudo code:

```
Procedure selection sort
list: array of items
  n: size of list
   for i = 1 to n - 1
   /* set current element as minimum*/
      min = i
      /* check the element to be minimum */
      for j = i+1 to n
    if list[j] < list [min] then
min = j;
         end if
end for
      /* swap the minimum element with the current element*/
      if indexMin != i  then
         swap list[min] and list[i]
      end if
   end for
end procedure:
```

Sample Example:



Viva Questions:

1. What is the average case performance of Selection Sort?

2. For each i from 1 to n-1, how many comparisons are there in this sort?

3. If all the input elements are identical then how it will affect the time taken by this sort?

4. What is the output of selection sort after the 2nd iteration given the following sequence of numbers: 16 3 46 9 28 14

5. What is straight selection sort?

I Heap sort

This algorithm is divided into two basic parts:

- Creating a Heap of the unsorted list.
- Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.

What is a heap?

Heap is a special tree-based data structure, which satisfies the following special heap properties:

1. **Shape Property:** Heap data structure is always a Complete Binary Tree, which means all levels of the tree are fully filled.
2. **Heap Property:** All nodes are either *[greater than or equal to]* or *[less than or equal to]* each of its children. If the parent nodes are greater than their children, heap is called a **Max-Heap**, and if the parent nodes are smaller than their child nodes, heap is called **Min-Heap**.

Initially on receiving an unsorted list, the first step in heap sort is to create a Heap data structure (Max-Heap or Min-Heap). Once heap is built, the first element of the Heap is either largest or smallest (depending upon Max-Heap or Min-Heap), so we put the first element of the heap in our array. Then we again make heap using the remaining elements, to again pick the first element of the heap and put it into the array. We keep on doing the same repeatedly until we have the complete sorted list in our array.

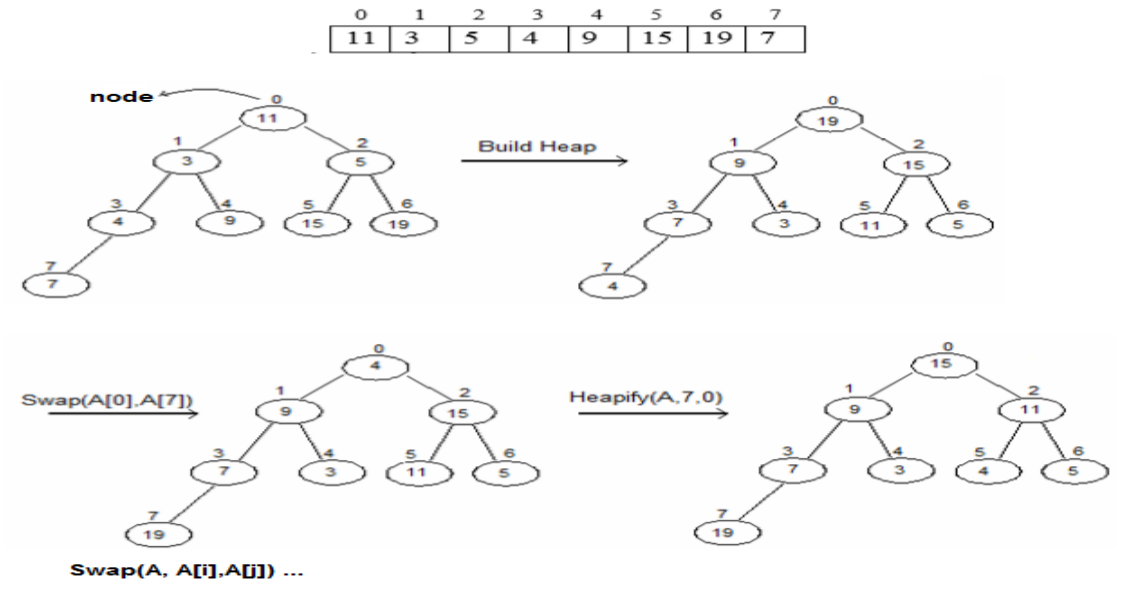
Pseudo code:

```
Heapsort (A) {
  BuildHeap(A)
  for i <- length (A) downto 2 {
    exchange A [1] <-> A[i]
    heapsize <- heapsize -1
    Heapify(A, 1)
}
```

```
BuildHeap (A) {
  heapsize <- length (A)
  for i <- floor (length/2) downto 1
Heapify (A, i)
}
```

```
Heapify (A, i) {
  le <- left (i)
  ri <- right(i)
  if (le<=heapsize) and (A[le]>A[i])
    largest <- le
  else
    largest <- i
  if (ri<=heapsize) and (A[ri]>A[largest])
    largest <- ri
  if (largest != i) {
    exchange A[i] <-> A[largest]
    Heapify(A, largest)
  }
}
```

Sample Example:



Viva Questions:

1. What are the minimum and maximum numbers of elements in a heap of height *h*?

2. Where in a heap might the smallest element reside?

3. Is an array that is in reverse sorted order a heap?

4. Does heap sort uses extra space for storage?

5. What is the effect of calling HEAPIFY *(A, i*) when the element A[*i*] is larger than its children?

Experiments Beyond the syllabus

Experiment 1

Aim: To implement concept of Pointers.

Theory:

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is –

type *var-name;

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

There are a few important operations, which we will do with the help of pointers very frequently. (a) We define a pointer variable

(b) assign the address of a variable to a pointer

(c) access the value at the address available in the pointer variable.

This is done by using unary operator * that returns the value of the variable located at the address specified by its operand.

Pseudo code:

```
int main ()
{

        int  var = 20;   /* actual variable declaration */
        int  *ip;        /* pointer variable declaration */

        ip = &var;  /* store address of var in pointer variable*/

        printf("Address of var variable: %x\n", &var  );

        /* address stored in pointer variable */
        printf("Address stored in ip variable: %x\n", ip );

        /* access the value using the pointer */
        printf("Value of *ip variable: %d\n", *ip );

        return 0;
}
```

Sample Output:

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

Viva Questions:

Q1. What is indirection?

Q2. How many levels of pointers can you have?

Q3. What is a null pointer?

Q4. What is a void pointer?

Q5. Can you add pointers together? Why would you?

Q6.How would you define a pointer to a structure?

Experiment 2

Aim: Create a circular Queue and implement different operations such as Insert, Delete, and Display the queue elements.

Theory:

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.

Operations on Circular Queue:

Front: Get the front item from queue.
Rear: Get the last item from queue.
enQueue(value): enter an element in queue.
deQueue(): delete an element from the queue.

Pseudo code:

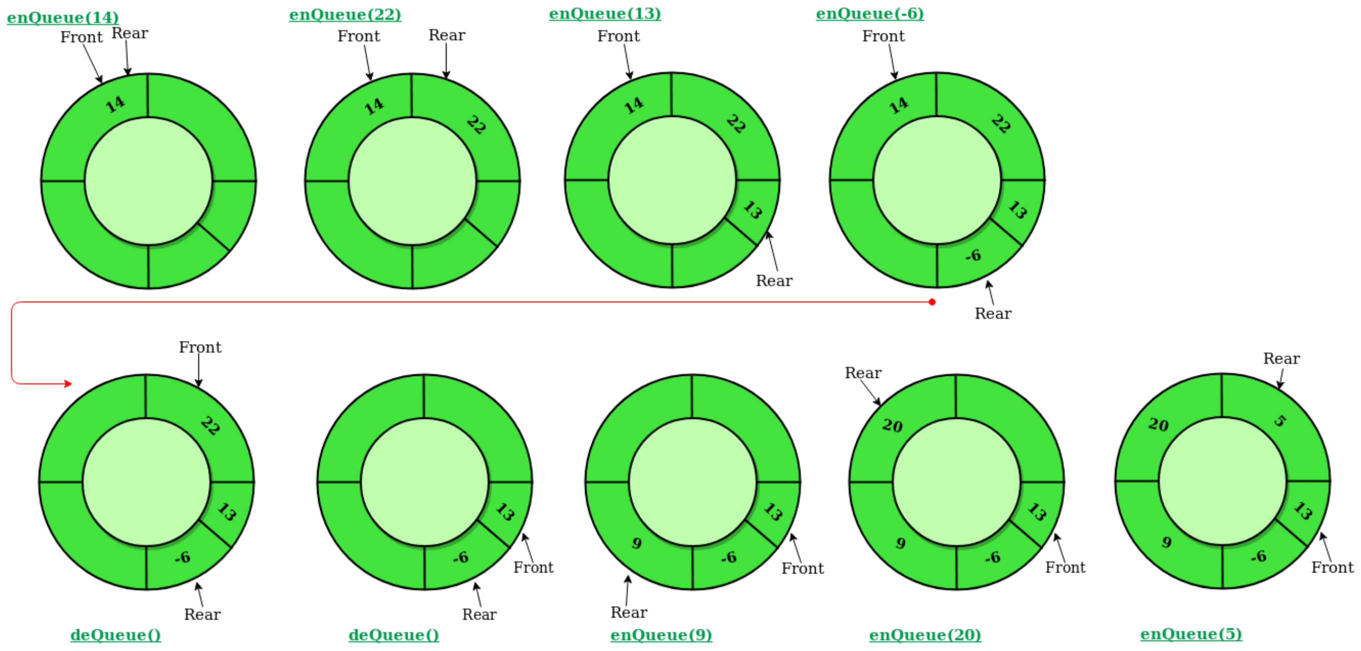
//**enQueue(value)** This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.

```
Steps:
    1.  Check whether queue is Full –
     Check ((rear == SIZE-1 && front == 0) || (rear == front-1)).
    If it is full then display Queue is full.
    2.  If queue is not full then, check if (rear == SIZE – 1 && front != 0)
    3    if it is true then set rear=0
    4    insert element.
```

//**deQueue()** This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position.

```
Steps:
    1Check whether queue is Empty means check (front==-1).
    2If it is empty then display Queue is empty. If queue is not empty then step 3
    3Check if (front==rear)
        a.  if it is true then set front=rear= -1
         else check if (front==size-1)
         b. if it is true
        then set front=0
        and return the element.
```

Example:


enQueue(14)
Front
Rear
14
enQueue(22)
Front
Rear
14
22
enQueue(13)
Front
14
22
13
Rear
enQueue(-6)
Front
14
22
13
-6
Rear
Front
22
13
-6
Rear
deQueue()
13
-6
Front
Rear
deQueue()
13
9
-6
Front
Rear
enQueue(9)
Rear
20
13
9
-6
Front
enQueue(20)
Rear
20
5
13
9
-6
Front
enQueue(5)

Viva Questions:

Q1. What are the different variations in a queue?

Q2. What is Circular Queue?

Q3 What is the use of Circular Queue?

Q4 Explain the condition of overflow in a circular queue ?

Q5 What are application of Queue ?

Q6 What are various ways of implementing queue?

Experiment 3

Aim: Write a program to implement priority queue.

Theory:
Priority Queue is an extension of queue with following properties.

1) Every item has a priority associated with it.
2) An element with high priority is dequeued before an element with low priority.
3) If two elements have the same priority, they are served according to their order in the queue.

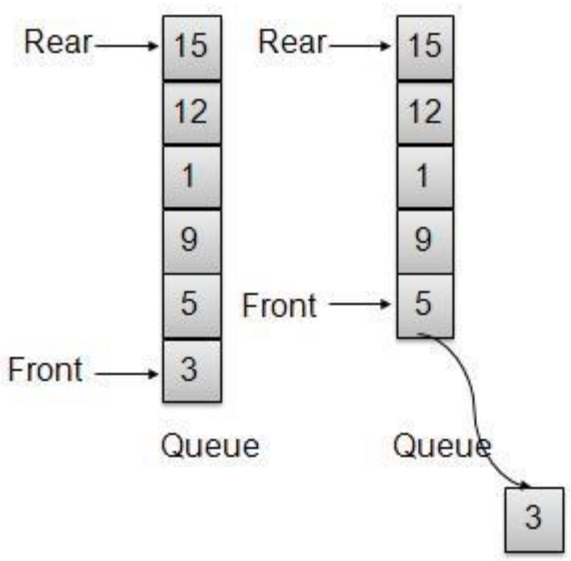
A typical priority queue supports following operations.

insert(item, priority): Inserts an item with given priority.
getHighestPriority(): Returns the highest priority item.
deleteHighestPriority(): Removes the highest priority item.

Pseudo code:
//enqueue
```
void insert(int data){
   int i = 0;

   if(!isFull()){
      // if queue is empty, insert the data

      if(itemCount == 0){
         intArray[itemCount++] = data;
      }else{
         // start from the right end of the queue
         for(i = itemCount - 1; i >= 0; i-- ){
            // if data is larger, shift existing item to right end
            if(data > intArray[i]){
               intArray[i+1] = intArray[i];
            }else{
               break;
            }
         }
         // insert the data
         intArray[i+1] = data;
         itemCount++;
      }
   }
}
```

//dequeue
```
int removeData(){
   return intArray[--itemCount];
}
```

Example:



One Item removed from front

Viva Questions:

Q1. What is priority Queue?

Q2. What are the various ways of implementing priority queue?

Q3. What are the various operations that can be implemented on priority queue?

Q4. Give some applications of priority queue.

Q5. Define a structure that will represent a priority queue node.

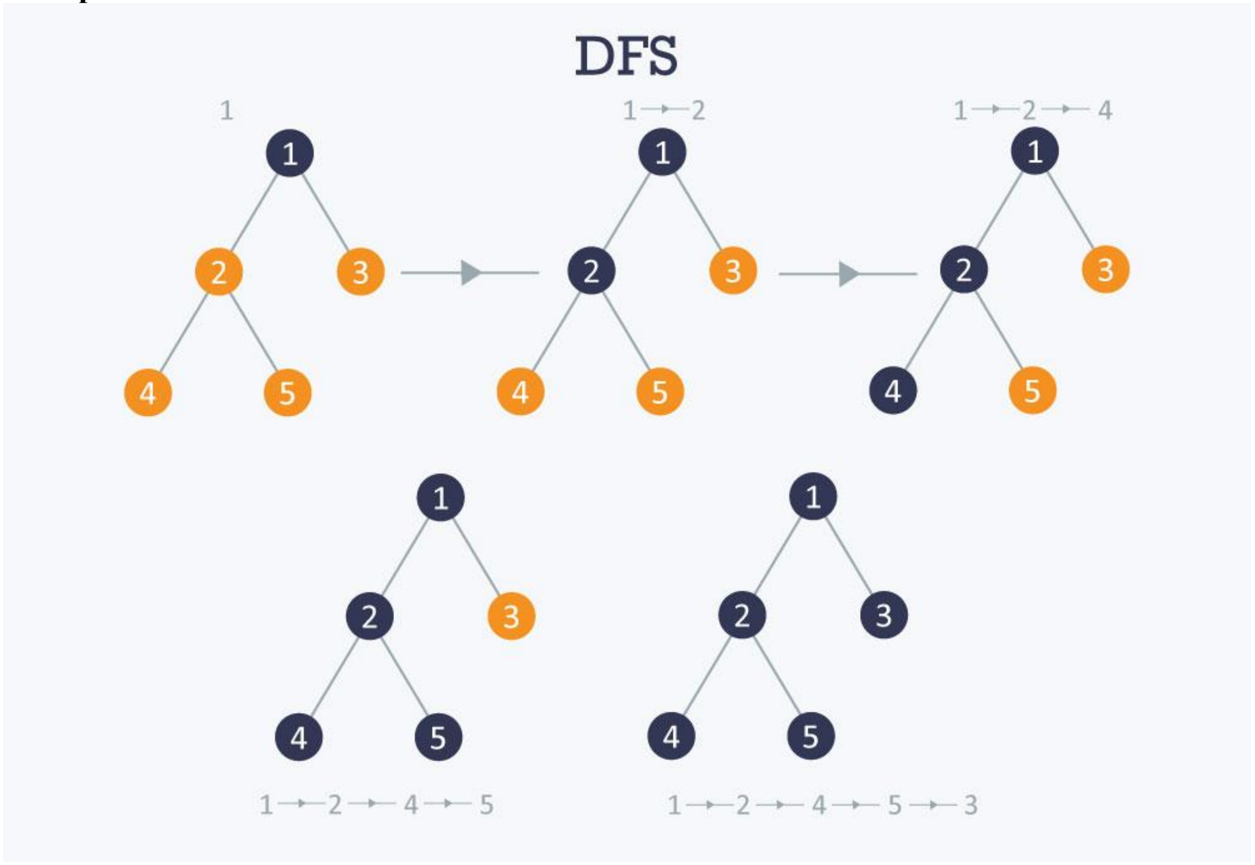
Experiment 4

Aim: WAP to Implement Depth-First-Search in a graph.

Theory:
Graph traversal means visiting every vertex and edge exactly once in a well-defined order.
The DFS algorithm is a recursive algorithm that uses the idea of backtracking. Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected. This recursive nature of DFS can be implemented using stacks.

Pseudo code:
```
DFS-recursive(G, s):
        mark s as visited
        for all neighbours w of s in Graph G:
            if w is not visited:
                DFS-recursive(G, w)
```

Example:



Viva Questions:

Q1. What is a graph?

Q2. What is graph traversal?

Q3. What are various methods of graph traversals?

Q4. Which data structure is used for implementing DFS?

Q5.Can a node repeats itself in graph traversal?

Experiment 5

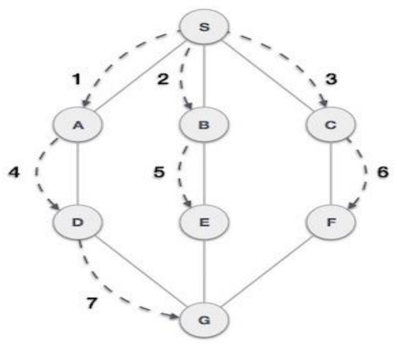
Aim: WAP to Implement Breadth-First-Search in graph.

Theory:
BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

Pseudo code:

```
BFS (G, s)                  //Where G is the graph and s is the source node
{
 let Q be queue.
     Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are marked.
     mark s as visited.
     while ( Q is not empty)
          //Removing that vertex from queue,whose neighbour will be visited now
          v  =  Q.dequeue( )
         //processing all the neighbours of v
         for all neighbours w of v in Graph G
              if w is not visited
                       Q.enqueue( w )            //Stores w in Q to further visit its neighbour
                       mark w as visited.
}
```

Example:



Viva Questions:

Q1. What is adjacency matrix?

Q2. What is adjacency list?

Q3. What is BFS?

Q4. Which data structure is used for implementing BFS?

Q5. What is cyclic graph?

7. Viva Questions

1. What is a data structure?
2. What is linear data structure?
3. What are the ways of representing linear structure?
4. What is a non-linear data structure?
5. What are various operations performed on linear structure? What is a square matrix?
6. What is a sparse matrix?
7. What is a triangular matrix?
8. What is a tridiagonal matrix?
9. What is row major ordering?
10. What is column major ordering?
11. What is a linked list?
12. What is a null pointer?

13. What is a free pool or free storage list or list of available space? 14. What
is garbage collection?
15. What is overflow?
16. What is underflow? What is a header node?
17. What is a header linked list?
18. What is a header node?
19. What is a grounded linked list?
20. What is circular header list?
21. What is a two-way list?
22. What is a stack?
23. What is a queue?
24. What is infix notation?
25. What is polish notation?
26. What is reverse polish notation?
27. What is recursive function?
28. W hat is a priority queue?
29. Define a deque.
30. Define Tree,Binary tree,Binary search tree.
31. What are various ways of tree traversal?
32. What is an AVL tree?

33. What are similar trees, and when the trees are called copies of each other? 34. What is
searching?
34. What is linear search?
35. What is binary search?

36. Why binary search cannot be applied on a linked list? 38. What
is a connected graph?
37. What is depth-first traversal?
38. What is breadth-first traversal?

39. Why is the algorithm for finding shortest distances called greedy? 42. What
are advantages of selection sort over other algorithms? 43. What are disadvantages
of insertion sort?
40. Define the term divide and conquer.
41. What is a pivot?