DATA STRUCTURES

(R18A0584)

LABORATORY MANUAL

B.TECH II YEAR – I SEM (R18)
(2019-20)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY
(Autonomous Institution – UGC, Govt. of India)
(Recognized under 2(f) and 12 (B) of UGC ACT 1956)
(Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – 'A' Grade - ISO 9001:2015 Certified)
Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad – 500100, Telangana State, India

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Vision

- To acknowledge quality education and instill high patterns of discipline making the students technologically superior and ethically strong which involves the improvement in the quality of life in human race.

Mission

- To achieve and impart holistic technical education using the best of infrastructure, outstanding technical and teaching expertise to establish the students into competent and confident engineers.
- Evolving the center of excellence through creative and innovative teaching learning practices for promoting academic achievement to produce internationally accepted competitive and world class professionals.

PROGRAMME EDUCATIONAL OBJECTIVES (PEOs)

PEO1 – ANALYTICAL SKILLS

1. To facilitate the graduates with the ability to visualize, gather information, articulate, analyze, solve complex problems, and make decisions. These are essential to address the challenges of complex and computation intensive problems increasing their productivity.

PEO2 – TECHNICAL SKILLS

2. To facilitate the graduates with the technical skills that prepare them for immediate employment and pursue certification providing a deeper understanding of the technology in advanced areas of computer science and related fields, thus encouraging to pursue higher education and research based on their interest.

PEO3 – SOFT SKILLS

3. To facilitate the graduates with the soft skills that include fulfilling the mission, setting goals, showing self-confidence by communicating effectively, having a positive attitude, get involved in team-work, being a leader, managing their career and their life.

PEO4 – PROFESSIONAL ETHICS

To facilitate the graduates with the knowledge of professional and ethical responsibilities by paying attention to grooming, being conservative with style, following dress codes, safety codes,and adapting themselves to technological advancements.

PROGRAM SPECIFIC OUTCOMES (PSOs)

After the completion of the course, B. Tech Computer Science and Engineering, the graduates will have the following Program Specific Outcomes:

1. **Fundamentals and critical knowledge of the Computer System:-** Able to Understand the working principles of the computer System and its components , Apply the knowledge to build, asses, and analyze the software and hardware aspects of it .

2. **The comprehensive and Applicative knowledge of Software Development:** Comprehensive skills of Programming Languages, Software process models, methodologies, and able to plan, develop, test, analyze, and manage the software and hardware intensive systems in heterogeneous platforms individually or working in teams.

3. **Applications of Computing Domain & Research:** Able to use the professional, managerial, interdisciplinary skill set, and domain specific tools in development processes, identify the research gaps, and provide innovative solutions to them.

PROGRAM OUTCOMES (POs)

Engineering Graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design / development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multi disciplinary environments.
12. **Life- long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY
Maisammaguda, Dhulapally Post, Via Hakimpet, Secunderabad – 500100
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

GENERAL LABORATORY INSTRUCTIONS

1. Students are advised to come to the laboratory at least 5 minutes before (to the starting time), those who come after 5 minutes will not be allowed into the lab.
2. Plan your task properly much before to the commencement, come prepared to the lab with the synopsis / program / experiment details.
3. Student should enter into the laboratory with:
   a. Laboratory observation notes with all the details (Problem statement, Aim, Algorithm, Procedure, Program, Expected Output, etc.,) filled in for the lab session.
   b. Laboratory Record updated up to the last session experiments and other utensils (if any) needed in the lab.
   c. Proper Dress code and Identity card.
4. Sign in the laboratory login register, write the TIME-IN, and occupy the computer system allotted to you by the faculty.
5. Execute your task in the laboratory, and record the results / output in the lab observation note book, and get certified by the concerned faculty.
6. All the students should be polite and cooperative with the laboratory staff, must maintain the discipline and decency in the laboratory.
7. Computer labs are established with sophisticated and high end branded systems, which should be utilized properly.
8. Students / Faculty must keep their mobile phones in SWITCHED OFF mode during the lab sessions.Misuse of the equipment, misbehaviors with the staff and systems etc., will attract severe punishment.
9. Students must take the permission of the faculty in case of any urgency to go out ; if anybody found loitering outside the lab / class without permission during working hours will be treated seriously and punished appropriately.
10. Students should LOG OFF/ SHUT DOWN the computer system before he/she leaves the lab after completing the task (experiment) in all aspects. He/she must ensure the system / seat is kept properly.

Head of the Department **Principal**

OBJECTIVES AND OUTCOMES

Objectives:

- To make the student learn a object oriented way of solving problems.
- To make the student write ADTS for all data structures.

Outcomes:

At the end of the course the students are able to:

- For a given Search problem (Linear Search and Binary Search) student will able to implement it.
- For a given problem of Stacks, Queues and linked list student will able to implement it and analyze the same to determine the time and computation complexity.
- Student will able to write program for Selection Sort, Bubble Sort, Insertion Sort, Quick Sort, Merge Sort, Heap Sort and compare their performance in term of Space and Time complexity.

Vision:

To acknowledge quality education and instill high patterns of discipline making the students technologically superior and ethically strong which involves the improvement in the quality of life in human race.

Mission:

To achieve and impart holistic technical education using the best of infrastructure, outstanding technical and teaching expertise to establish the students into competent and confident engineers.

Evolving the center of excellence through creative and innovative teaching learning practices for promoting academic achievement to produce internationally accepted competitive and world class professionals.

RECOMMENDED SYSTEM / SOFTWARE REQUIREMENTS:

1. Intel based desktop PC of 166MHz or faster processor with at least 64 MB RAM and 100 MB free disk space.

2. turbo C++ compiler or GCC compilers

USEFUL TEXT BOOKS / REFERECES :

1. Data structures, Algorithms and Applications in C++, S.Sahni, University Press (India) Pvt.Ltd, 2nd edition, Universities Press Orient Longman Pvt. Ltd.
2. Data structures and Algorithms in C++, Michael T.Goodrich, R.Tamassia and .Mount, Wiley student edition, John Wiley and Sons.
3. Data structures using C and C++, Langsam, Augenstein and Tanenbaum, PHI.

DATA STRUCTURES Lab Manual

List of programs

S.No	Name of the program	Page no	Date	Faculty sign
1.	Write a program that uses functions to perform the following operations on singly linked list i) Creation ii) Insertion iii) Deletion iv) Traversal.			
2.	Write a program that uses functions to perform the following operations on doubly linked list i) Creation ii) Insertion iii) Deletion iv) Traversal.			
3.	Write a program that uses functions to perform the following operations on circular linked List i) Creation ii) Insertion iii) Deletion iv) Traversal.			
4.	Write a program that implement stack (its operations) using i) Arrays ii) Linked list(Pointers).			
5.	Write a program that implement Queue (its operations) using i) Arrays ii) Linked list(Pointers).			
6.	i) Write a program that implement Circular Queue using arrays. ii) Write a program that uses both recursive and non recursive functions to perform the following searching operations for a Key value in a given list of integers: a) Linear search b) Binary search.			
7.	Write a program that implements the following sorting i) Bubble sort ii) Selection sort iii)Quick sort.			
8.	Write a program that implements the following i) Insertion sort ii) Merge sort iii)Heap sort.			
9.	Write a program to implement all the functions of a dictionary (ADT) using Linked List.			
10.	Write a program to perform the following operations: a) Insert an element into a binary search tree. b) Delete an element from a binary search tree. c) Search for a key element in a binary search tree.			
11.	Write a program to implement the tree traversal methods			
12.	Write a program to perform the following operations: a) Insert an element into a AVL tree. b) Delete an element from a AVL tree. c) Search for a key element in a AVL tree.			

WEEK-1: **DATE:**

Aim: Write a program that uses functions to perform the following operations on Singly Linked List
(i)Creation (ii) Insertion (iii) Deletion (iv) Traversal.

Description:

Linked List

When we want to work with an unknown number of data values, we use a linked list data structure to organize that data. The linked list is a linear data structure that contains a sequence of elements such that each element links to its next element in the sequence. Each element in a linked list is called "Node".

Linked List can be implemented as

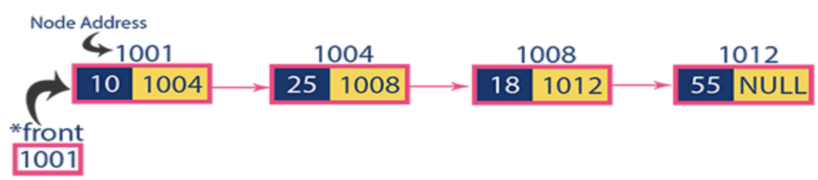
1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List

Single Linked List

Simply a list is a sequence of data, and the linked list is a sequence of data linked with each other. The formal definition of a single linked list is as follows...

In any single linked list, the individual element is called as "Node". Every "Node" contains two fields, data field, and the next field. The data field is used to store actual value of the node and next field is used to store the address of next node in the sequence. The graphical representation of a node in a single linked list is as follows...

Example



Operations on Single Linked List

The following operations are performed on a Single Linked List

1.Creation
2.Insertion
3.Deletion
4.Display

Before we implement actual operations, first we need to set up an empty list. First, perform the following steps before implementing actual operations.

1.Creation

Step 1 - Define a Node structure with two members data and next

Step 2 - Define a Node pointer 'head' and set it to NULL.

2.Insertion

In a single linked list, the insertion operation can be performed in three ways. They are as follows...

2.1 Inserting At Beginning of the list

2.2 Inserting At End of the list

2.3 Inserting At Specific location in the list

2.1 Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the single linked list...

Step 1 - Create a newNode with given value.

Step 2 - Check whether list is Empty (head == NULL)

Step 3 - If it is Empty then, set newNode→next = NULL and head = newNode.

Step 4 - If it is Not Empty then, set newNode→next = head and head = newNode.

2.2 Inserting At End of the list

We can use the following steps to insert a new node at end of the single linked list...

Step 1 - Create a newNode with given value and newNode → next as NULL.

Step 2 - Check whether list is Empty (head == NULL).

Step 3 - If it is Empty then, set head = newNode.

Step 4 - If it is Not Empty then, define a node pointer temp and initialize with head.

Step 5 - Keep moving the temp to its next node until it reaches to the last node in the list (until temp → next is equal to NULL).

Step 6 - Set temp → next = newNode.

2.3 Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the single linked list...

Step 1 - Create a newNode with given value.

Step 2 - Check whether list is Empty (head == NULL)

Step 3 - If it is Empty then, set newNode → next = NULL and head = newNode.

Step 4 - If it is Not Empty then, define a node pointer temp and initialize with head.

Step 5 - Keep moving the temp to its next node until it reaches to the node after which we want to insert the newNode (until temp1 → data is equal to location, here location is the node value after which we want to insert the newNode).

Step 6 - Every time check whether temp is reached to last node or not. If it is reached to last node then display 'Given node is not found in the list!!! Insertion not possible!!!' and terminate the function. Otherwise move the temp to next node.

Step 7 - Finally, Set 'newNode → next = temp → next' and 'temp → next = newNode'

3. Deletion

In a single linked list, the deletion operation can be performed in three ways. They are as follows...

3.1 Deleting from Beginning of the list

3.2 Deleting from End of the list

3.3 Deleting a Specific Node

3.1 Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the single linked list...

Step 1 - Check whether list is Empty (head == NULL)

Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3 - If it is Not Empty then, define a Node pointer 'temp' and initialize with head.

Step 4 - Check whether list is having only one node (temp → next == NULL)

Step 5 - If it is TRUE then set head = NULL and delete temp (Setting Empty list conditions)

Step 6 - If it is FALSE then set head = temp → next, and delete temp.

3.2 Deleting from End of the list

We can use the following steps to delete a node from end of the single linked list...

Step 1 - Check whether list is Empty (head == NULL)

Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3 - If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.

Step 4 - Check whether list has only one Node (temp1 → next == NULL)

Step 5 - If it is TRUE. Then, set head = NULL and delete temp1. And terminate the function. (Setting Empty list condition)

Step 6 - If it is FALSE. Then, set 'temp2 = temp1 ' and move temp1 to its next node. Repeat the same until it reaches to the last node in the list. (until temp1 →next == NULL)

Step 7 - Finally, Set temp2 → next = NULL and delete temp1.

3.3 Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the single linked list...

Step 1 - Check whether list is Empty (head == NULL)

Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3 - If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.

Step 4 - Keep moving the temp1 until it reaches to the exact node to be deleted or to the last node. And every time set 'temp2 = temp1' before moving the 'temp1' to its next node.

Step 5 - If it is reached to the last node then display 'Given node not found in the list! Deletion not possible!!!'. And terminate the function.

Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node or not

Step 7 - If list has only one node and that is the node to be deleted, then set head = NULL and delete temp1 (free(temp1)).

Step 8 - If list contains multiple nodes, then check whether temp1 is the first node in the list (temp1 == head).

Step 9 - If temp1 is the first node then move the head to the next node (head = head → next) and delete temp1.

Step 10 - If temp1 is not first node then check whether it is last node in the list (temp1 → next == NULL).

Step 11 - If temp1 is last node then set temp2 → next = NULL and delete temp1 (free(temp1)).

Step 12 - If temp1 is not first node and not last node then set temp2 → next = temp1 → next and delete temp1 (free(temp1)).

4. Displaying a Single Linked List

We can use the following steps to display the elements of a single linked list...

Step 1 - Check whether list is Empty (head == NULL)

Step 2 - If it is Empty then, display 'List is Empty!!!' and terminate the function.

Step 3 - If it is Not Empty then, define a Node pointer 'temp' and initialize with head.

Step 4 - Keep displaying temp → data with an arrow (--->) until temp reaches to the last node

Step 5 - Finally display temp → data with arrow pointing to NULL (temp → data ---> NULL).

Source Code: To implement Singly Linked List

```
 #include<iostream.h>
#include<conio.h>
#include<stdlib.h>
void insertAtBeginning(int);
void insertAtEnd(int);
void insertBetween(int,int,int);
void display();
void removeBeginning();
void removeEnd();
void removeSpecific(int);
struct Node
{
  int data;
  struct Node *next;
}*head = NULL;
void main()
{
  int choice,value,choice1,loc1,loc2;
  clrscr();
  while(1){
  mainMenu:
      cout<<"\n\n****** MENU ******\n1. Insert\n2. Display\n3. Delete\n4.  Exit\nEnter your
              choice: ";
     cin>>choice;
  switch(choice)
```

```
{
  case 1:          cout<<"Enter the value to be insert: ";
                   cin>>value;
                   while(1)
                  {
                       cout<<"Where you want to insert: \n1. At Beginning\n2. At End\n3.
                                   Between\nEnter your choice: ";
                        cin>>choice1;
                   switch(choice1)
                   {
                       case 1:          insertAtBeginning(value);
                                        break;
                       case 2:          insertAtEnd(value);
                                        break;
                       case 3:     cout<<"Enter the two values where you wanto insert: ";
                                   cin>>loc1>>loc2;
                                        insertBetween(value,loc1,loc2);
                                        break;
                      default:          cout<<"\nWrong Input!! Try again!!!\n\n";
                                        goto mainMenu;
                   }
                   goto subMenuEnd;
                   }
                   subMenuEnd:
                   break;
    case 2:        display();
                   break;
    case 3:        cout<<"Ho do you want to Delete: \n1. From Beginning\n2. From End\n3.
                      Spesific\nEnter your choice: ";
                   cin>>choice1;
                   switch(choice1)
                   {
                     case 1: removeBeginning();

                             break;
```

```
                    case 2: removeEnd();
                            break;
                    case 3:   cout<<"Enter the value which you wanto delete: ";
                              cin>>loc2;
                              removeSpecific(loc2);
                              break;
                    default:  cout<<"\nWrong Input!! Try again!!!\n\n";
                              goto mainMenu;
                  }
                       break;
                  case 4:  exit(0);
                  default: cout<<"\nWrong input!!! Try again!!\n\n";
  }
   }
}
void insertAtBeginning(int value)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   if(head == NULL)
   {
      newNode->next = NULL;
      head = newNode;
   }
   else
   {
      newNode->next = head;
      head = newNode;
   }
   cout<<"\nOne node inserted!!!\n";
}
void insertAtEnd(int value)
{
```

```
  struct Node *newNode;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = value;
  newNode->next = NULL;
  if(head == NULL)
        head = newNode;
  else
  {
    struct Node *temp = head;
    while(temp->next != NULL)
        temp = temp->next;
    temp->next = newNode;
  }
cout<<"\nOne node inserted!!!\n";
}
void insertBetween(int value, int loc1, int loc2)
{
  struct Node *newNode;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = value;
  if(head == NULL)
  {
    newNode->next = NULL;
    head = newNode;
  }
  else
  {
    struct Node *temp = head;
    while(temp->data != loc1 && temp->data != loc2)
        temp = temp->next;
    newNode->next = temp->next;
    temp->next = newNode;
  }
```

```
 cout<<"\nOne node inserted!!!\n";
}
void removeBeginning()
{
  if(head == NULL)
            cout<<"\n\nList is Empty!!!";
  else
  {
    struct Node *temp = head;
    if(head->next == NULL)
    {
             head = NULL;
             free(temp);
    }
    else
    {
            head = temp->next;


            free(temp);
            cout<<"\nOne node deleted!!!\n\n";
    }
  }
}
void removeEnd()
{
  if(head == NULL)
  {
    cout<<"\nList is Empty!!!\n";
  }
  else
  {
    struct Node *temp1 = head,*temp2;
    if(head->next == NULL)
```

```
      head = NULL;
    else
    {
        while(temp1->next != NULL)
        {
          temp2 = temp1;
          temp1 = temp1->next;
        }
        temp2->next = NULL;
    }
    free(temp1);
    cout<<"\nOne node deleted!!!\n\n";
  }
}
void removeSpecific(int delValue)
{
  struct Node *temp1 = head, *temp2;
  while(temp1->data != delValue)
  {
   if(temp1 -> next == NULL){
        cout<<"\nGiven node not found in the list!!!";
        goto functionEnd;
   }
   temp2 = temp1;
   temp1 = temp1 -> next;
  }
  temp2 -> next = temp1 -> next;
  free(temp1);
  cout<<"\nOne node deleted!!!\n\n";
  functionEnd:
}
```

```
void display()
{
  if(head == NULL)
  {
    cout<<"\nList is Empty\n";

 }
  else
  {
    struct Node *temp = head;
    cout<<"\n\nList elements are - \n";
    while(temp->next != NULL)
    {
            cout<<temp->data<<”\t”;
            temp = temp->next;
    }
    cout<<temp->data;
  }
}
```

Output:

Signature of the Faculty

WEEK-2 **DATE:**

Aim: Write a program that uses functions to perform the following operations on doubly linked List (i)Creation (ii) Insertion (iii) Deletion (iv) Traversal.

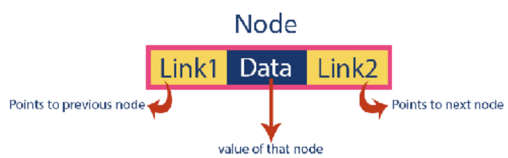
Description:

Double Linked List

In a single linked list, every node has a link to its next node in the sequence. So, we can traverse from one node to another node only in one direction and we can not traverse back. We can solve this kind of problem by using a double linked list. A double linked list can be defined as follows...

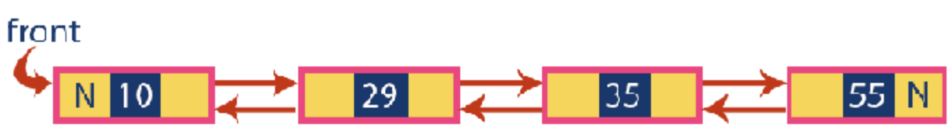
Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.

In a double linked list, every node has a link to its previous node and next node. So, we can traverse forward by using the next field and can traverse backward by using the previous field. Every node in a double linked list contains three fields and they are shown in the following figure...



Here, **'link1'** field is used to store the address of the previous node in the sequence, **'link2'** field is used to store the address of the next node in the sequence and **'data'** field is used to store the actual value of that node.

Example



Operations on Double Linked List

In a double linked list, we perform the following operations...

1. Creation
2. Insertion
3. Deletion
4. Display

1.Creation

Step 1 - Define a Node structure with two members data and next

Step 2 - Define a Node pointer 'head' and set it to NULL.

2.Insertion

In a double linked list, the insertion operation can be performed in three ways as follows...

2.1 Inserting At Beginning of the list

2.2 Inserting At End of the list

2.3 Inserting At Specific location in the list

2.1 Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the double linked list...

Step 1 - Create a **newNode** with given value and **newNode → previous** as **NULL**.

Step 2 - Check whether list is **Empty** (**head** == **NULL**)

Step 3 - If it is **Empty** then, assign **NULL** to **newNode → next** and **newNode** to **head**.

Step 4 - If it is **not Empty** then, assign **head** to **newNode → next** and **newNode** to **head**.

2.2 Inserting At End of the list

We can use the following steps to insert a new node at end of the double linked list...

Step 1 - Create a **newNode** with given value and **newNode → next** as **NULL**.

Step 2 - Check whether list is **Empty** (**head** == **NULL**)

Step 3 - If it is **Empty**, then assign **NULL** to **newNode → previous** and **newNode** to **head**.

Step 4 - If it is **not Empty**, then, define a node pointer **temp** and initialize with **head**.

Step 5 - Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** is equal to **NULL**).

Step 6 - Assign **newNode** to **temp → next** and **temp** to **newNode → previous**.

2.3 Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the double linked list...

Step 1 - Create a **newNode** with given value.

Step 2 - Check whether list is **Empty** (**head** == **NULL**)

Step 3 - If it is **Empty** then, assign **NULL** to both **newNode → previous** & **newNode → next** and set **newNode** to **head**.

Step 4 - If it is **not Empty** then, define two node pointers **temp1** & **temp2** and initialize **temp1** with **head**.

Step 5 - Keep moving the **temp1** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).

Step 6 - Every time check whether **temp1** is reached to the last node. If it is reached to the last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp1** to next node.

Step7- Assign **temp1→next** to **temp2**, **newNode** to **temp1 → next**, **temp1** to **newNode → previous**, **temp2** to **newNode → next**and **newNode** to **temp2 → previous**.

3.Deletion

In a double linked list, the deletion operation can be performed in three ways as follows...

3.1 Deleting from Beginning of the list

3.2 Deleting from End of the list

3.3 Deleting a Specific Node

3.1 Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the double linked list...

Step 1 - Check whether list is **Empty** (**head** == **NULL**)

Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

Step 3 - If it is not Empty then, define a Node pointer **'temp'** and initialize with **head**.

Step 4 - Check whether list is having only one node (**temp → previous** is equal to **temp → next**)

Step 5 - If it is **TRUE**, then set **head** to **NULL** and delete **temp** (Setting **Empty** list conditions)

Step 6 - If it is **FALSE**, then assign **temp → next** to **head**, **NULL** to **head → previous** and delete **temp**.

3.2 Deleting from End of the list

We can use the following steps to delete a node from end of the double linked list...

Step 1 - Check whether list is **Empty** (**head** == **NULL**)

Step 2 - If it is **Empty**, then display **'List is Empty!!! Deletion is not possible'** and terminate the function.

Step 3 - If it is not Empty then, define a Node pointer **'temp'** and initialize with **head**.

Step 4 - Check whether list has only one Node (**temp → previous** and **temp → next** both are **NULL**)

Step 5 - If it is **TRUE**, then assign **NULL** to **head** and delete **temp**. And terminate from the function. (Setting **Empty** list condition)

Step 6 - If it is **FALSE**, then keep moving **temp** until it reaches to the last node in the list. (until **temp → next** is equal to **NULL**)

Step 7 - Assign **NULL** to **temp → previous → next** and delete **temp**.

3.3 Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the double linked list...

Step 1 - Check whether list is **Empty** (**head** == **NULL**)

Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

Step 3 - If it is not Empty, then define a Node pointer **'temp'** and initialize with **head**.

Step 4 - Keep moving the **temp** until it reaches to the exact node to be deleted or to the last node.

Step 5 - If it is reached to the last node, then display **'Given node not found in the list! Deletion not possible!!!'** and terminate the fuction.

Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node or not

Step 7 - If list has only one node and that is the node which is to be deleted then set **head** to **NULL** and delete **temp** (**free(temp)**).

Step 8 - If list contains multiple nodes, then check whether **temp** is the first node in the list (**temp == head**).

Step 9 - If **temp** is the first node, then move the **head** to the next node (**head = head → next**), set **head** of **previous** to **NULL** (**head → previous = NULL**) and delete **temp**.

Step 10 - If **temp** is not the first node, then check whether it is the last node in the list (**temp → next == NULL**).

Step 11 - If **temp** is the last node then set **temp** of **previous** of **next** to **NULL**

(**temp → previous → next = NULL**) and delete **temp**(**free(temp)**).

Step 12 - If temp is not the first node and not the last node, then set temp of previous of next to temp of next (temp → previous → next = temp→next),temp of next of previous to temp of previous (**temp → next → previous = temp → previous**) and delete **temp**(**free(temp)**).

4.Displaying

We can use the following steps to display the elements of a double linked list...

Step 1 - Check whether list is **Empty** (**head** == **NULL**)

Step 2 - If it is **Empty**, then display **'List is Empty!!!'** and terminate the function.

Step 3 - If it is not Empty, then define a Node pointer **'temp'** and initialize with **head**.

Step 4 - Display **'NULL <--- '**.

Step 5 - Keep displaying **temp** → **data** with an arrow (**<===>**) until **temp** reaches to the last node

Step 6 - Finally, display **temp** → **data** with arrow pointing to **NULL**

(**temp** → **data ---> NULL**).

Source Code: To implement Doubly Linked List

```
#include<iostream.h>
#include<conio.h>
void insertAtBeginning(int);
void insertAtEnd(int);
void insertAtAfter(int,int);
void deleteBeginning();
void deleteEnd();
void deleteSpecific(int);
void display();

struct Node
{
  int data;
  struct Node *previous, *next;
}*head = NULL;
```

```
void main()
{
  int choice1, choice2, value, location;
  clrscr();

  while(1)
  {
      cout<<"\n*********** MENU *************\n");
      cout<<"1. Insert\n2. Delete\n3. Display\n4. Exit\nEnter your choice: ";
      cin>>choice1;
    switch(choice1)
    {
      case 1: cout<<"Enter the value to be inserted: ";
              cin>>value;
              while(1)
              {
                 cout<<"\nSelect from the following Inserting options\n";
                 cout<<"1. At Beginning\n2. At End\n3. After a Node\n4. Cancel\nEnter
                           your choice: ";
                 cin>>choice2;
                 switch(choice2)
                 {
                      case 1:  insertAtBeginning(value);
                               break;
                      case 2:  insertAtEnd(value);
                               break;
                      case 3:  cout<<"Enter the location after which you want to insert:";
                               cin>>location;
                               insertAfter(value,location);
                               break;
                      case 4:  goto EndSwitch;
                      default: cout<<"\nPlease select correct Inserting option!!!\n";
                 }
               }
      case 2:


                while(1)
                 {
                    cout<<"\nSelect from the following Deleting options\n";

                    cout<<"1. At Beginning\n2. At End\n3. Specific Node\n4.
                            Cancel\nEnter  your choice: ";
                    cin>>choice2;
                  switch(choice2)
```

```
                    {
                        case 1:  deleteBeginning();
                                 break;
                        case 2:  deleteEnd();
                                 break;

                        case 3:  cout<<"Enter the Node value to be deleted: ";
                                 cin>>location;
                                 deleteSpecic(location);
                                 break;
                        case 4:  goto EndSwitch;
                       default: cout<<"\nPlease select correct Deleting option!!!\n";
                    }
                  }
                    EndSwitch: break;
           case 3: display();
                   break;
           case 4: exit(0);
           default: cout<<"\nPlease select correct option!!!";

        }
      }
   }
      void insertAtBeginning(int value)
      {
        struct Node *newNode;
        newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode -> data = value;
        newNode -> previous = NULL;
        if(head == NULL)
        {
          newNode -> next = NULL;
          head = newNode;
        }
        else
        {
          newNode -> next = head;
          head = newNode;
        }
        cout<<"\nInsertion success!!!";
      }
      void insertAtEnd(int value)
      {
        struct Node *newNode;
        newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode -> data = value;
```

```
newNode -> next = NULL;
  if(head == NULL)
  {
    newNode -> previous = NULL;
    head = newNode;

 }
  else
  {
    struct Node *temp = head;
    while(temp -> next != NULL)
      temp = temp -> next;
    temp -> next = newNode;
    newNode -> previous = temp;
  }
  cout<<"\nInsertion success!!!";
}
void insertAfter(int value, int location)
{
     struct Node *newNode;
     newNode = (struct Node*)malloc(sizeof(struct Node));
     newNode -> data = value;
     if(head == NULL)
     {
        newNode -> previous = newNode -> next = NULL;
        head = newNode;
     }
   else
     {
         struct Node *temp1 = head, temp2;
         while(temp1 -> data != location)
        {
             if(temp1 -> next == NULL)
             {
                  cout<<"Given node is not found in the list!!!";
                  goto EndFunction;
              }
           else
             {
                temp1 = temp1 -> next;
             }
        }
         temp2 = temp1 -> next;
         temp1 -> next = newNode;
         newNode -> previous = temp1;
```

```
            newNode -> next = temp2;
            temp2 -> previous = newNode;
             cout<<"\nInsertion success!!!";

    }
     EndFunction:

  }
  void deleteBeginning()
  {
        if(head == NULL)
         cout<<"List is Empty!!! Deletion not possible!!!";
      else
        {
                struct Node *temp = head;
                 if(temp -> previous == temp -> next)
                {
                   head = NULL;
                   free(temp);
                 }
                else
                {
                      head = temp -> next;
                     head -> previous = NULL;
                     free(temp);
                }
               cout<<"\nDeletion success!!!";
          }
   }
  void deleteEnd()
  {
                if(head == NULL)
                cout<<"List is Empty!!! Deletion not possible!!!";
            else
               {
                        struct Node *temp = head;
                        if(temp -> previous == temp -> next)
                        {
                          head = NULL;
                          free(temp);
                        }
                        else
                        {
                             while(temp -> next != NULL)
                              temp = temp -> next;
                             temp -> previous -> next = NULL;
```

```
                    free(temp);
                }
           cout<<"\nDeletion success!!!";
    }
}
void deleteSpecific(int delValue)

{
        if(head == NULL)
          cout<<"List is Empty!!! Deletion not possible!!!";
        else
         {
              struct Node *temp = head;
              while(temp -> data != delValue)
              {
                  if(temp -> next == NULL)
                  {
                         cout<<"\nGiven node is not found in the list!!!";
                         goto FuctionEnd;
                  }
                  else
                  {
                     temp = temp -> next;

                   }
              }
               if(temp == head)
              {
                        head = NULL;
                        free(temp);
              }
              else
              {
                       temp -> previous -> next = temp -> next;
                       free(temp);
              }
              cout<<"\nDeletion success!!!";
        }
       FuctionEnd:
}
void display()
{
              if(head == NULL)
              cout<<"\nList is Empty!!!";
              else
              {
```

```
                struct Node *temp = head;
                cout<<"\nList elements are: \n";
                cout<<"NULL <--- ";
                while(temp -> next != NULL)
                {
                        cout<<temp -> data<<"\t";

        }
                cout<<temp -> data;
        }
}
```

Output:

Signature of the Faculty

WEEK-3 **DATE:**

Aim: Write a program that uses functions to perform the following operations on circular linked List
(i)Creation (ii) Insertion (iii) Deletion (iv) Traversal.

Description:

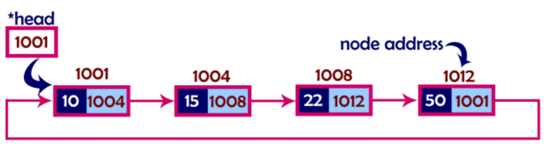
Circular Linked List

In single linked list, every node points to its next node in the sequence and the last node points NULL. But in circular linked list, every node points to its next node in the sequence but the last node points to the first node in the list.

A circular linked list is a sequence of elements in which every element has a link to its next element in the sequence and the last element has a link to the first element.

That means circular linked list is similar to the single linked list except that the last node points to the first node in the list

Example:



Operations

In a circular linked list, we perform the following operations...

1. Creation
2. Insertion
3. Deletion
4. Display

1.Creation

Step 1 - Define a Node structure with two members data and next

Step 2 - Define a Node pointer 'head' and set it to NULL.

2.Insertion

In a circular linked list, the insertion operation can be performed in three ways. They are as follows...

2.1 Inserting At Beginning of the list
2.2 Inserting At End of the list
2.3 Inserting At Specific location in the list

2.1 Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the circular linked list...

Step 1 - Create a **newNode** with given value.

Step 2 - Check whether list is **Empty** (**head** == **NULL**)

Step 3 - If it is **Empty** then, set **head** = **newNode** and **newNode→next** = **head** .

Step 4 - If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with '**head**'.

Step 5 - Keep moving the '**temp**' to its next node until it reaches to the last node (until '**temp** → **next** == **head**').

Step 6 - Set '**newNode** → **next** =**head**', '**head** = **newNode**' and '**temp** → **next** = **head**'.

2.2 Inserting At End of the list

We can use the following steps to insert a new node at end of the circular linked list...

Step 1 - Create a **newNode** with given value.

Step 2 - Check whether list is **Empty** (**head** == **NULL**).

Step 3 - If it is **Empty** then, set **head** = **newNode** and **newNode** → **next** = **head**.

Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.

Step 5 - Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp** → **next** == **head**).

Step 6 - Set **temp** → **next** = **newNode** and **newNode** → **next** = **head**.

2.3 Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the circular linked list...

Step 1 - Create a **newNode** with given value.

Step 2 - Check whether list is **Empty** (**head** == **NULL**)

Step 3 - If it is **Empty** then, set **head** = **newNode** and **newNode** → **next** = **head**.

Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.

Step 5 - Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).

Step 6 - Every time check whether **temp** is reached to the last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.

Step 7 - If **temp** is reached to the exact node after which we want to insert the newNode then check whether it is last node (temp → next == head).

Step 8 - If **temp** is last node then set **temp → next** = **newNode** and **newNode → next** = **head**.

Step 9 - If **temp** is not last node then set **newNode → next** = **temp → next** and **temp → next** = **newNode**.

3. Deletion

In a circular linked list, the deletion operation can be performed in three ways those are as follows...

3.1 Deleting from Beginning of the list
3.2 Deleting from End of the list
3.3 Deleting a Specific Node

3.1 Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the circular linked list...

Step 1 - Check whether list is **Empty** (**head** == **NULL**)

Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize both '**temp1**' and '**temp2**' with **head**.

Step 4 - Check whether list is having only one node (**temp1 → next == head**)

Step 5 - If it is **TRUE** then set **head** = **NULL** and delete **temp1** (Setting **Empty** list conditions)

Step 6 - If it is **FALSE** move the **temp1** until it reaches to the last node.

(until **temp1 → next** == **head**)

Step 7 - Then set **head** = **temp2 → next**, **temp1 → next** = **head** and delete **temp2**.

3.2 Deleting from End of the list

We can use the following steps to delete a node from end of the circular linked list...

Step 1 - Check whether list is **Empty** (**head** == **NULL**)

Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2'** and initialize '**temp1**' with **head**.

Step 4 - Check whether list has only one Node (**temp1 → next** == **head**)

Step 5 - If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate from the function. (Setting **Empty** list condition)

Step 6 - If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until **temp1** reaches to the last node in the list. (until **temp1 → next** == **head**)

Step 7 - Set **temp2 → next** = **head** and delete **temp1**.

3.3 Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the circular linked list...

Step 1 - Check whether list is **Empty** (**head** == **NULL**)

Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.

Step 4 - Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.

Step 5 - If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.

Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node (**temp1 → next**== **head**)

Step 7 - If list has only one node and that is the node to be deleted then set **head** = **NULL** and delete **temp1** (**free(temp1)**).

Step 8 - If list contains multiple nodes then check whether **temp1** is the first node in the list (**temp1 == head**).

Step 9 - If **temp1** is the first node then set **temp2** = **head** and keep moving **temp2** to its next node until **temp2** reaches to the last node. Then set **head = head → next**, **temp2 → next** = **head** and delete **temp1**.

Step 10 - If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == head**).

Step 1 1- If **temp1** is last node then set **temp2 → next** = **head** and delete **temp1** (**free(temp1)**).

Step 12 - If **temp1** is not first node and not last node then set **temp2 → next** = **temp1 → next** and delete **temp1** (**free(temp1)**).

4.Displaying a Circular Linked List

We can use the following steps to display the elements of a circular linked list...

Step 1 - Check whether list is **Empty** (**head** == **NULL**)

Step 2 - If it is **Empty**, then display **'List is Empty!!!'** and terminate the function.

Step 3 - If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.

Step 4 - Keep displaying **temp** → **data** with an arrow (**--->**) until **temp** reaches to the last node

Step 5 - Finally display **temp** → **data** with arrow pointing to **head** → **data**.

Source Code: To implement Circular Linked List.

```
#include<iostream.h>
#include<conio.h>
void insertAtBeginning(int);
void insertAtEnd(int);
void insertAtAfter(int,int);
void deleteBeginning();
void deleteEnd();
void deleteSpecific(int);
void display();

struct Node
{
  int data;
  struct Node *next;
}*head = NULL;

void main()
{
    int choice1, choice2, value, location;
    clrscr();
    while(1)
    {
        cout<<"\n*********** MENU *************\n";
        cout<<"1. Insert\n2. Delete\n3. Display\n4. Exit\nEnter your choice: ";
        cin>>choice1;
        switch(choice1)
        {
```

```
case 1: cout<<"Enter the value to be inserted: ";
        cin>>value;
    while(1)
    {
      cout<<"\nSelect from the following Inserting options\n";
      cout<<"1. At Beginning\n2. At End\n3. After a Node\n4. Cancel\nEnter
      your choice: ";
      cin>>choice2;
     switch(choice2)
     {
         case 1:   insertAtBeginning(value);
                   break;
         case 2:   insertAtEnd(value);
                   break;
         case 3:   cout<<"Enter the location after which you want to insert:";
                   cin>>location;
                   insertAfter(value,location);
                   break;

          case 4:  goto EndSwitch;
         default: cout<<"\nPlease select correct Inserting option!!!\n";
      }
     }
case 2: while(1)
        {
          cout<<"\nSelect from the following Deleting options\n";
          cout<<"1. At Beginning\n2. At End\n3. Specific Node\n4.
                 Cancel\nEnter your choice: ";
            cin>>choice2;
          switch(choice2)
          {
             case 1:  deleteBeginning();
                      break;
             case 2:  deleteEnd();
                      break;
             case 3:  cout<<"Enter the Node value to be deleted: ";
                      cin>>location;
                      deleteSpecic(location);
                      break;
             case 4:  goto EndSwitch;
             default: cout<<"\nPlease select correct Deleting option!!!\n";


           }
    }
```

```
                      EndSwitch: break;
          case 3: display();
                      break;
          case 4:  exit(0);
          default: cout<<"\nPlease select correct option!!!";
        }
      }

  }
  void insertAtBeginning(int value)
  {
     struct Node *newNode;
     newNode = (struct Node*)malloc(sizeof(struct Node));
     newNode -> data = value;
     if(head == NULL)
     {
       head = newNode;
       newNode -> next = head;
     }
     else
     {
       struct Node *temp = head;
       while(temp -> next != head)
        temp = temp -> next;
       newNode -> next = head;
       head = newNode;
      temp -> next = head;
     }
     cout<<"\nInsertion success!!!";
  }
  void insertAtEnd(int value)
  {
    struct Node *newNode;
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode -> data = value;
    if(head == NULL)
    {
      head = newNode;
      newNode -> next = head;
    }

   else
    {
        struct Node *temp = head;
         while(temp -> next != head)
```

```
        temp = temp -> next;
        temp -> next = newNode;
        newNode -> next = head;
    }
    cout<<"\nInsertion success!!!";
}
void insertAfter(int value, int location)
{

  struct Node *newNode;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode -> data = value;
  if(head == NULL)
  {
    head = newNode;
    newNode -> next = head;
  }
  else
  {
    struct Node *temp = head;
    while(temp -> data != location)
    {
      if(temp -> next == head)
      {
        cout<<"Given node is not found in the list!!!";
        goto EndFunction;
      }
      else
      {
        temp = temp -> next;
      }
}
    newNode -> next = temp -> next;
    temp -> next = newNode;
    cout<<"\nInsertion success!!!";
  }
  EndFunction:
}
void deleteBeginning()
{
          if(head == NULL)
          cout<<"List is Empty!!! Deletion not possible!!!";
     else
         {
               struct Node *temp = head;
```

```
                if(temp -> next == head)
                {
                    head = NULL;
                    free(temp);
                }
                else
                {
                    head = head -> next;
                    free(temp);

                }
                cout<<"\nDeletion success!!!";
            }
    }
void deleteEnd()
{
        if(head == NULL)
        cout<<"List is Empty!!! Deletion not possible!!!";
    else
    {
            struct Node *temp1 = head, temp2;
            if(temp1 -> next == head)
            {
                    head = NULL;
                    free(temp1);
            }
            else
            {
                while(temp1 -> next != head){
                temp2 = temp1;
                temp1 = temp1 -> next;
            }
            temp2 -> next = head;
            free(temp1);
    }
    cout<<"\nDeletion success!!!";
  }
}
void deleteSpecific(int delValue)
{
        if(head == NULL)
        cout<<"List is Empty!!! Deletion not possible!!!";
    else
        {
            struct Node *temp1 = head, temp2;
            while(temp1 -> data != delValue)
```

```
                    {
                        if(temp1 -> next == head)
                        {
                                    cout<<"\nGiven node is not found in the list!!!";
                                    goto FuctionEnd;
                         }
                        else
                         {
                               temp2 = temp1;

                               temp1 = temp1 -> next;
                          }
                      }
                      if(temp1 -> next == head)
                      {
                            head = NULL;
                            free(temp1);
                      }
                    else
                    {
                      if(temp1 == head)
                       {
                          temp2 = head;
                          while(temp2 -> next != head)
                          temp2 = temp2 -> next;
                          head = head -> next;
                         temp2 -> next = head;
                         free(temp1);
                       }
                    else
                    {
                          if(temp1 -> next == head)
                         {
                              temp2 -> next = head;
                         }
                        else
                         {
                                   temp2 -> next = temp1 -> next;
                         }
                         free(temp1);

          }
               }
               cout<<"\nDeletion success!!!";
            }
            FuctionEnd:
```

```
}
void display()
{
    if(head == NULL)
    cout<<"\nList is Empty!!!";
  else
  {
    struct Node *temp = head;
    cout<<"\nList elements are: \n";

  while(temp -> next != head)
    {
      cout<<temp -> data;
    }
    cout<< temp -> data, head -> data;
  }
}
```

Output:

Signature of the Faculty

WEEk-4 **DATE:**

Aim: Write a program that implement stack (its operations) using
(i)Arrays (ii)Linked list(Pointers).

Description:

Stack

Stack is a linear data structure in which the insertion and deletion operations are performed at only one end. In a stack, adding and removing of elements are performed at a single position which is known as "**top**".

That means, a new element is added at top of the stack and an element is removed from the top of the stack. In stack, the insertion and deletion operations are performed based on **LIFO**(Last In First Out) principle.

In a stack, the insertion operation is performed using a function called **"push"** and deletion operation is performed using a function called **"pop"**.

In the figure, PUSH and POP operations are performed at a top position in the stack. That means, both the insertion and deletion operations are performed at one end (i.e., at Top).
A stack data structure can be defined as follows...

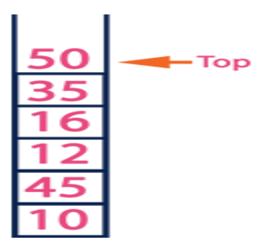
Stack is a linear data structure in which the operations are performed based on LIFO principle.

Stack can also be defined as

"A Collection of similar data items in which both insertion and deletion operations are performed based on LIFO principle".

Example

If we want to create a stack by inserting 10,45,12,16,35 and 50. Then 10 becomes the bottom-most element and 50 is the topmost element. The last inserted element 50 is at Top of the stack as shown in the image below...



Operations on a Stack

The following operations are performed on the stack...

1. Push (To insert an element on to the stack)
2. Pop (To delete an element from the stack)
3. Display (To display elements of the stack)

Stack data structure can be implemented in two ways. They are as follows...

1. Using Arrays
2. Using Linked List

Stack Using Arrays

A stack data structure can be implemented using a one-dimensional array. But stack implemented using array stores only a fixed number of data values. This implementation is very simple. Just define a one dimensional array of specific size and insert or delete the values into that array by using **LIFO principle** with the help of a variable called **'top'**.

Initially, the top is set to -1. Whenever we want to insert a value into the stack, increment the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

Stack Operations

We can Perform the following Operations on Stack

1.Push()
2.Pop()
3.Display()

A stack can be implemented using array as follows...

Before implementing actual operations, first follow the below steps to create an empty stack.

Step 1 - Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.

Step 2 - Declare all the **functions** used in stack implementation.

Step 3 - Create a one dimensional array with fixed size (**int stack[SIZE]**)

Step 4 - Define a integer variable **'top'** and initialize with **'-1'**. (**int top = -1**)

Step 5 - In main method, display menu with list of operations and make suitable function calls to perform operation selected by the user on the stack.

1.Push(value) - Inserting value into the stack

In a stack, push() is a function used to insert an element into the stack. In a stack, the new element is always inserted at **top** position. Push function takes one integer value as parameter and inserts that value into the stack.

We can use the following steps to push an element on to the stack...

Step 1 - Check whether **stack** is **FULL**. (**top == SIZE-1**)

Step 2 - If it is **FULL**, then display **"Stack is FULL!!! Insertion is not possible!!!"** and terminate the function.

Step 3 - If it is **NOT FULL**, then increment **top** value by one (**top++**) and set stack[top] to value (**stack[top] = value**).

2.Pop() - Delete a value from the Stack

In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from **top** position. Pop function does not take any value as parameter. We can use the following steps to pop an element from the stack...

Step 1 - Check whether **stack** is **EMPTY**. (**top == -1**)

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 delete **stack[top]** and decrement **top** value by one (**top--**).

3.Display() - Displays the Elements of a Stack

We can use the following steps to display the elements of a stack...

Step 1 - Check whether **stack** is **EMPTY**. (**top == -1**)

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 variable '**i**' and initialize with top.

Display **stack[i]** value and decrement **i** value by one (**i--**).

Step 4 - Repeat above step until **i** value becomes '0'.

Source Code: To implement Stack Using Arrays

```
#include<iostream.h>
```

```
#include<conio.h>
#define SIZE 10
void push(int);
void pop();
void display();
int stack[SIZE], top = -1;
void main()
{
  int value, choice;
  clrscr();
  while(1){
    cout<<"\n\n***** MENU *****\n";
    cout<<"1. Push\n2. Pop\n3. Display\n4. Exit";
    cout<<"\nEnter your choice: ";
    cin>>choice;
    switch(choice)
   {
          case 1: cout<<"Enter the value to be insert: ";
                  cin>>value;
                  push(value);
                  break;
         case 2: pop();
                  break;
```

```
            case 3: display();
                        break;
            case 4: exit(0);
            default: cout<<"\nWrong selection!!! Try again!!!";
      }
   }
}
void push(int value)
{
         if(top == SIZE-1)
        cout<<"\nStack is Full!!! Insertion is not possible!!!";
      else
       {
            top++;
            stack[top] = value;
             cout<<"\nInsertion success!!!";
       }
}
void pop()
{
            if(top == -1)
            cout<<"\nStack is Empty!!! Deletion is not possible!!!";
```

```
        else
         {
                 cout<<"\nDeleted :"<<stack[top]);
                 top--;
          }
  }
void display()
{
        if(top == -1)
       cout<<"\nStack is Empty!!!";
   else
       {
                 int i;
                cout<<"\nStack elements are:\n";
                for(i=top; i>=0; i--)
                cout<<stack[i];
            }
 }
```

Output:

Signature of the Faculty

(ii) Stack Using Linked List

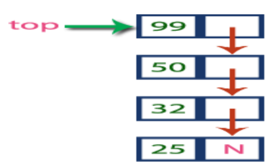
The major problem with the stack implemented using an arrays is, it works only for a fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself.

Stack implemented using an array is not suitable, when we don't know the size of data which we are going to use. A stack data structure can be implemented by using a linked list data structure.

The stack implemented using linked list can work for an unlimited number of values. That means, stack implemented using linked list works for the variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as '**top**' element. That means every newly inserted element is pointed by '**top**'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its previous node in the list. The **next** field of the first element must be always **NULL**.

Example



In the 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.

Stack Operations using Linked List

We can Perform the Following Operations on Stack Using Linked List (i.e)

1. Push()
2. Pop()
3. Display()

To implement a stack using a linked list, we need to set the following things before implementing actual operations.

Step 1 - Include all the **header files** which are used in the program. And declare all the **user defined functions**.

Step 2 - Define a '**Node**' structure with two members **data** and **next**.

Step 3 - Define a **Node** pointer '**top**' and set it to **NULL**.

Step 4 - Implement the **main** method by displaying Menu with list of operations and make suitable function calls in the **main**method.

1.Push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node 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**.

2.Pop() - Deleting an Element from a Stack

We can use the following steps to delete a node from the 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 5 - Finally, delete '**temp**'. (**free(temp)**).

3.Display() - Displaying stack of elements

We can use the following steps to display the elements (nodes) of a stack...

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 5 - Finally! Display '**temp** → **data** ---> **NULL**'.

```
#include<iostream.h>
#include<conio.h>
struct Node
{
   int data;
   struct Node *next;
}*top = NULL;
void push(int);
void pop();
void display();
void main()
{
     int choice, value;
    clrscr();
    cout<<"\n:: Stack using Linked List ::\n";
    while(1)
    {
          cout<<"\n****** MENU ******\n";
          cout<<"1. Push\n2. Pop\n3. Display\n4. Exit\n";
           cout<<"Enter your choice: ";
           cin>>choice;
           switch(choice)
           {
              case 1:    cout<<"Enter the value to be insert: ";
                         cin>>value;
                         push(value);
                         break;
              case 2:   pop();
                        break;
              case 3:  display(); break;
              case 4: exit(0);
             default: cout<<"\nWrong selection!!! Please try again!!!\n";
         }
```

```
}
void push(int value)
{
      struct Node *newNode;
      newNode = (struct Node*)malloc(sizeof(struct Node));
      newNode->data = value;
      if(top == NULL)
      newNode->next = NULL;
     else
      newNode->next = top;
      top = newNode;
      cout<<"\nInsertion is Success!!!\n";
}
void pop()
{
        if(top == NULL)
        cout<<"\nStack is Empty!!!\n";
         else
         {
             struct Node *temp = top;
            cout<<"\nDeleted element:", temp->data;
            top = temp->next;
            free(temp);
        }
}
void display()
{
             if(top == NULL)
              cout<<"\nStack is Empty!!!\n";
         else
            {
                struct Node *temp = top;
                while(temp->next != NULL)
```

```
                    cout<<temp->data;
                     temp = temp -> next;
                 }
                cout<<temp->data;
            }
        }
```

Output:

Signature of the Faculty

WEEK-5 **DATE:**

Aim: Write a program that implement Queue (its operations) using

(i)Arrays (ii)Linked list(Pointers).

Description:

Queue Using Arrays

A queue data structure can be implemented using one dimensional array. The queue implemented using array stores only fixed number of data values.

The implementation of queue data structure using array is very simple. Just define a one dimensional array of specific size and insert or delete the values into that array by using **FIFO (First In First Out) principle** with the help of variables **'front'** and '**rear**'. Initially both '**front**' and '**rear**' are set to -1.

Whenever, we want to insert a new value into the queue, increment '**rear**' value by one and then insert at that position. Whenever we want to delete a value from the queue, then delete the element which is at 'front' position and increment 'front' value by one.

Queue Operations using Array

We can Perform the following operations on Queue
1.enQueue()
2.deQueue()
3.Display()

Before we implement actual operations, first follow the below steps to create an empty queue.

Step 1 - Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.

Step 2 - Declare all the **user defined functions** which are used in queue implementation.

Step 3 - Create a one dimensional array with above defined SIZE (**int queue[SIZE]**).

Step 4 - Define two integer variables **'front'** and '**rear**' and initialize both with **'-1'**. (**int front = -1, rear = -1**).

Step 5 - Then implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on queue.

1.enQueue(value) - Inserting value into the queue

In a queue data structure, enQueue() is a function used to insert a new element into the queue. In a queue, the new element is always inserted at **rear** position. The enQueue() function takes one integer value as a parameter and inserts that value into the queue. We can use the following steps to insert an element into the queue...

Step 1 - Check whether **queue** is **FULL**. (**rear == SIZE-1**)

Step 2 - If it is **FULL**, then display **"Queue is FULL!!! Insertion is not possible!!!"** and terminate the function.

Step 3 - If it is **NOT FULL**, then increment **rear** value by one (**rear++**) and set **queue[rear]** = **value**.

2.deQueue() - Deleting a value from the Queue

In a queue data structure, deQueue() is a function used to delete an element from the queue. In a queue, the element is always deleted from **front** position. The deQueue() function does not take any value as parameter. We can use the following steps to delete an element from the queue...

Step 1 - Check whether **queue** is **EMPTY**. (**front == rear**)

Step 2 - If it is **EMPTY**, then display **"Queue is EMPTY!!! Deletion is not possible!!!"** and terminate the function.

Step 3 - If it is **NOT EMPTY**, then increment the **front** value by one (**front ++**). Then display **queue[front]** as deleted element. Then check whether both **front** and **rear** are equal (**front** == **rear**), if it **TRUE**, then set both **front** and **rear** to '**-1**' (**front** = **rear** = **-1**).

3.Display() - Displays the elements of a Queue

We can use the following steps to display the elements of a queue...

Step 1 - Check whether **queue** is **EMPTY**. (**front == rear**)

Step 2 - If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.

Step 3 - If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front+1**'.

Step 4 - Display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i**' value reaches to **rear** (**i** <= **rear**).

Source Code: To implement Queue using Arrays

```
#include<iostream.h>
#include<conio.h>
#define SIZE 10
void enQueue(int);
void deQueue();
void display();
int queue[SIZE], front = -1, rear = -1;
void main()
{
  int value, choice;
  clrscr();
  while(1){
    cout<<"\n\n***** MENU *****\n";
    cout<<"1. Insertion\n2. Deletion\n3. Display\n4. Exit";
    cout<<"\nEnter your choice: ";
    cin>>choice;
    switch(choice){
          case 1: cout<<"Enter the value to be insert: ";
                  cin>>value;
                  enQueue(value);
                  break;
          case 2: deQueue();
                  break;
          case 3: display();
                  break;
```

```
        default: cout<<"\nWrong selection!!! Try again!!!";
      }
   }
}
void enQueue(int value){
   if(rear == SIZE-1)
      cout<<"\nQueue is Full!!! Insertion is not possible!!!";
   else{
      if(front == -1)
             front = 0;
             rear++;
             queue[rear] = value;
             cout<<"\nInsertion success!!!";
   }
}
void deQueue()
{
            if(front == rear)
             cout<<"\nQueue is Empty!!! Deletion is not possible!!!";
        else
       {
              cout<<"\nDeleted : %d", queue[front];
               front++;
                if(front == rear)
                front = rear = -1;
        }
}
void display()
 {
                if(rear == -1)
                cout<<"\nQueue is Empty!!!";
         else
          {
                int i;
```

```
        cout<<"\nQueue elements are:\n";
         for(i=front; i<=rear; i++)
         cout<<queue[i];
    }
}
```

Output:

Signature of the Faculty

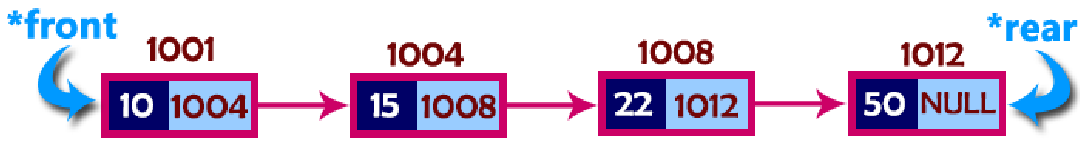
II. Queue Using Linked List

The major problem with the queue implemented using an array is, It will work for an only fixed number of data values. That means, the amount of data must be specified at the beginning itself. Queue using an array is not suitable when we don't know the size of data which we are going to use.

A queue data structure can be implemented using a linked list data structure. The queue which is implemented using a linked list can work for an unlimited number of values. That means, queue using linked list can work for the variable size of data (No need to fix the size at the beginning of the implementation).

The Queue implemented using linked list can organize as many data values as we want. In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.

Example



In above example, the last inserted node is 50 and it is pointed by '**rear**' and the first inserted node is 10 and it is pointed by '**front**'. The order of elements inserted is 10, 15, 22 and 50.

Queue Operations using Array

We can Perform the following operations on Queue
1.enQueue()
2.deQueue()
3.Display()

To implement queue using linked list, we need to set the following things before implementing actual operations.

Step 1 - Include all the **header files** which are used in the program. And declare all the **user defined functions**.

Step 2 - Define a '**Node**' structure with two members **data** and **next**.

Step 3 - Define two **Node** pointers '**front**' and '**rear**' and set both to **NULL**.

Step 4 - Implement the **main** method by displaying Menu of list of operations and make suitable function calls in the **main** method to perform user selected operation.

1.enQueue(value) - Inserting an element into the Queue

We can use the following steps to insert a new node 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**.

2.deQueue() - Deleting an Element from Queue

We can use the following steps to delete a node from the 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)**).

3.Display() - Displaying the elements of Queue

We can use the following steps to display the elements (nodes) of a 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 5 - Finally! Display '**temp** → **data** ---> **NULL**'.

Source Code:To implement Queue using Linked List

```
#include<iostream.h>
#include<conio.h>
```

```
struct Node
{
  int data;
  struct Node *next;
}*front = NULL,*rear = NULL;
void insert(int);
void delete();
void display();
void main()
{
  int choice, value;
  clrscr();
  cout<<"\n:: Queue Implementation using Linked List ::\n";
  while(1){
    cout<<"\n****** MENU ******\n";
    cout<<"1. Insert\n2. Delete\n3. Display\n4. Exit\n";
    cout<<"Enter your choice: ";
    cin>>choice);
    switch(choice){
        case 1: cout<<"Enter the value to be insert: ";
                cin>>value;
                insert(value);
                break;
        case 2: delete(); break;
        case 3: display(); break;
        case 4: exit(0);
        default: cout<<"\nWrong selection!!! Please try again!!!\n";
    }
  }
}
void insert(int value)
{
  struct Node *newNode;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = value;
```

```
  newNode -> next = NULL;
  if(front == NULL)
    front = rear = newNode;
  else{
    rear -> next = newNode;
    rear = newNode;
  }
  cout<<"\nInsertion is Success!!!\n";
}
void delete()
{
  if(front == NULL)
    cout<<"\nQueue is Empty!!!\n";

 else{
    struct Node *temp = front;
    front = front -> next;
    cout<<"\nDeleted element: %d\n", temp->data;
    free(temp);
  }
}
void display()
{
  if(front == NULL)
    cout<<"\nQueue is Empty!!!\n";
  else{
    struct Node *temp = front;
    while(temp->next != NULL){
            cout<<temp->data;
            temp = temp -> next;
    }
    cout<<temp->data;
  }
}
```

Output:

Signature of the Faculty

WEEK-6: **DATE:**

(i)Write a program that implement Circular Queue (its operations) using Arrays .
(ii)Write a program that use both recursive and non recursive functions to perform the following searching operations for a Key value in a given list of integers:

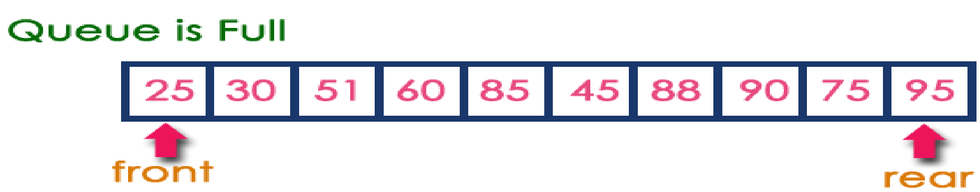
a) Linear search b) Binary search.

Aim: Write a program that implement Circular Queue (its operations) using Arrays .

Description:

Circular Queue

In a normal Queue Data Structure, we can insert elements until queue becomes full. But once the queue becomes full, we can not insert the next element until all the elements are deleted from the
The queue after inserting all the elements into it is as follows...



Now consider the following situation after deleting three elements from the queue...

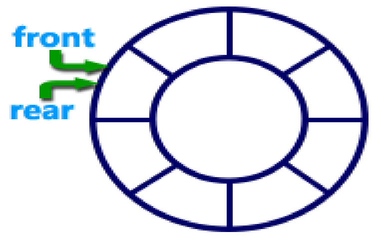
This situation also says that Queue is Full and we cannot insert the new element because '**rear**' is still at last position. In the above situation, even though we have empty positions in the queue we can not make use of them to insert the new element. This is the major problem in a normal queue data structure. To overcome this problem we use a circular queue data structure.

Circular Queue

A Circular Queue can be defined as follows...

A 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.

Graphical representation of a circular queue is as follows...



Implementation of Circular Queue

To implement a circular queue data structure using an array, we first perform the following steps before we implement actual operations.

Step 1 - Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.

Step 2 - Declare all **user defined functions** used in circular queue implementation.

Step 3 - Create a one dimensional array with above defined SIZE (**int cQueue[SIZE]**)

Step 4 - Define two integer variables **'front'** and '**rear**' and initialize both with **'-1'**.

(**int front = -1, rear = -1**)

Step 5 - Implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on circular queue.

1.enQueue(value) - Inserting value into the Circular Queue

In a circular queue, enQueue() is a function which is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at **rear** position. The enQueue() function takes one integer value as parameter and inserts that value into the circular queue. We can use the following steps to insert an element into the circular queue...

Step 1 - Check whether **queue** is **FULL**.

(**(rear == SIZE-1 && front == 0) || (front == rear+1)**)

Step 2 - If it is **FULL**, then display **"Queue is FULL!!! Insertion is not possible!!!"** and terminate the function.

Step 3 - If it is **NOT FULL**, then check **rear == SIZE - 1 && front != 0** if it is **TRUE**, then set **rear = -1**.

Step 4 - Increment **rear** value by one (**rear++**), set **queue[rear]** = **value** and check '**front == -1**' if it is **TRUE**, then set **front = 0**.

2.deQueue() - Deleting a value from the Circular Queue

In a circular queue, deQueue() is a function used to delete an element from the circular queue. In a circular queue, the element is always deleted from **front** position. The deQueue() function doesn't take any value as a parameter. We can use the following steps to delete an element from the circular queue...

Step 1 - Check whether **queue** is **EMPTY**. (**front == -1 && rear == -1**)

Step 2 - If it is **EMPTY**, then display **"Queue is EMPTY!!! Deletion is not possible!!!"** and terminate the function.

Step 3 - If it is **NOT EMPTY**, then display **queue[front]** as deleted element and increment the **front** value by one (**front ++**). Then check whether **front == SIZE**, if it is **TRUE**, then set **front = 0**. Then check whether both **front – 1** and **rear** are equal (**front -1** == **rear**), if it **TRUE**, then set both **front** and **rear** to '**-1**' (**front** = **rear** = **-1**).

3.Display() - Displays the elements of a Circular Queue

We can use the following steps to display the elements of a circular queue...

Step 1 - Check whether **queue** is **EMPTY**. (**front == -1**)

Step 2 - If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.

Step 3 - If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front**'.

Step 4 - Check whether '**front <= rear**', if it is **TRUE**, then display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i <= rear**' becomes **FALSE**.

Step 5 - If '**front <= rear**' is **FALSE**, then display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until'**i <= SIZE - 1**' becomes **FALSE**.

Step 6 - Set **i** to **0**.

Step 7 - Again display '**cQueue[i]**' value and increment **i** value by one (**i++**). Repeat the same until '**i <= rear**' becomes **FALSE**.

Source Code: To implement Circular Queue using Arrays.

```
#include<iostream.h>
#include<conio.h>
#define SIZE 5
void enQueue(int);
void deQueue();
void display();
int cQueue[SIZE], front = -1, rear = -1;
void main()
{
  int choice, value;
  clrscr();
  while(1){
    cout<<"\n****** MENU ******\n";
    cout<<"1. Insert\n2. Delete\n3. Display\n4. Exit\n";
    cout<<"Enter your choice: ";
    cin>>choice;
    switch(choice){
           case 1: cout<<"\nEnter the value to be insert:  ";
                      cin>>value;
                      enQueue(value);
                      break;
           case 2: deQueue();
```

```
                        break;
                case 3: display();
                        break;
                case 4: exit(0);
                default: cout<<"\nPlease select the correct choice!!!\n";
        }
    }
}
void enQueue(int value)
{
  if((front == 0 && rear == SIZE - 1) || (front == rear+1))
    cout<<"\nCircular Queue is Full! Insertion not possible!!!\n";
  else{
    if(rear == SIZE-1 && front != 0)
        rear = -1;
    cQueue[++rear] = value;
    cout<<"\nInsertion Success!!!\n";
    if(front == -1)
        front = 0;
  }
}
void deQueue()
{
  if(front == -1 && rear == -1)
    cout<<"\nCircular Queue is Empty! Deletion is not possible!!!\n";
  else{
    cout<<"\nDeleted element :"<<cQueue[front++];
    if(front == SIZE)
        front = 0;
    if(front-1 == rear)
        front = rear = -1;
  }
}
void display()
```

```
{
  if(front == -1)
        cout<<"\nCircular Queue is Empty!!!\n";
        else{
             int i = front;
            cout<<"\nCircular Queue Elements are : \n";
            if(front <= rear){
                        while(i <= rear)
                          cout<<"\t"<<cQueue[i++];
             }
            else{
            while(i <= SIZE - 1)
              cout<<"\t"<< cQueue[i++];
                        i = 0;
            while(i <= rear)
              cout<<"\t"<<cQueue[i++];
     }
   }
}
```

Output:

Signature of the Faculty

Aim: write a C++ programs to implement recursive and non recursive

i) Linear search ii) Binary Search

Description:

i) **LINEAR SEARCH (SEQUENTIAL SEARCH):**

Search begins by comparing the first element of the list with the target element. If it matches, the search ends. Otherwise, move to next element and compare. In this way, the target element is compared with all the elements until a match occurs. If the match do not occur and there are no more elements to be compared, conclude that target element is absent in the list.

Algorithm for Linear search

Linear_Search (A[ ], N, val , pos)

Step 1 : Set pos = -1 and k = 0

Step 2 : Repeat while k < N Begin

Step 3 : if A[k] = val

Set pos = k

print pos

Goto step 5

End while

Step 4 : print “Value is not present”

Step 5 : Exit

Source code: Non recursive C++ program for Linear search

```
#include<iostream>

using namespace std;


int Lsearch(int list[ ],int n,int key);
```

```
int main()

{
        int n,i,key,list[25],pos;
        cout<<"enter no of elements\n";

        cin>>n;

        cout<<"enter "<<n<<" elements ";

        for(i=0;i<n;i++)

         cin>>list[i];

        cout<<"enter key to search";

        cin>>key;

        pos= Lsearch (list,n,key);

        if(pos==-1)

                cout<<"\nelement not found";
          else
                cout<<"\n element found at index "<<pos;
}
/*function for linear search*/
int Lsearch(int list[],int n,int key)
{
    int i,pos=-1;
        for(i=0;i<n;i++)
        if(key==list[i])
        {
                pos=i;
                break;
        }
       return pos;
}
```

Output:

Signature of the Faculty

Source code: Recursive C++ program for Linear search

```
#include<iostream>

using namespace std;

int Rec_Lsearch(int list[ ],int n,int key);

int main()

{

        int n,i,key,list[25],pos;

        cout<<"enter no of elements\n";

        cin>>n;

        cout<<"enter "<<n<<" elements ";

        for(i=0;i<n;i++)

         cin>>list[i];

        cout<<"enter key to search";

        cin>>key;

        pos=Rec_Lsearch(list,n,key);

        if(pos==-1)

                cout<<"\nelement not found";

        else

        cout<<"\n element found at index "<<pos;

}

/*recursive function for linear search*/

int Rec_Lsearch(int list[],int n,int key)

{
```

```
if(n<=0)
   return -1;

if(list[n]==key)

   return n;

   else

   return Rec_Lsearch(list,n-1,key);

}
```

Output:

Signature of the Faculty

(ii)Binary Search:

Before searching, the list of items should be sorted in ascending order. First compare the key value with the item in the mid position of the array.

If there is a match, we can return immediately the position. if the value is less than the element in middle location of the array, the required value is lie in the lower half of the array.

If the value is greater than the element in middle location of the array, the required value is lie in the upper half of the array. We repeat the above procedure on the lower half or upper half of the array.

Algorithm:

Binary_Search (A [ ], U_bound, VAL)

Step 1 : set BEG = 0 , END = U_bound , POS = -1

Step 2 : Repeat while (BEG <= END)

Step 3 :set MID = (BEG + END) / 2

POS = MID

print VAL " is available at ", POS

GoTo Step 6

End if

if A [MID] > VAL then

set END = MID – 1

Else

set BEG = MID + 1

End if

End while

Step 5 : if POS = -1 then

print VAL " is not present "

End if

Step 6 : EXIT

Source code: Non recursive C++ program for binary search

```
#include<iostream>

using namespace std;

int binary_search(int list[],int key,int low,int high);

int main()

{

int n,i,key,list[25],pos;

        cout<<"enter no of elements\n" ;
```

```
        cout<<"enter "<<n<<" elements in ascending order ";

        for(i=0;i<n;i++)

        cin>>list[i];

        cout<<"enter key to search" ;

        cin>>key;

        pos=binary_search(list,key,0,n-1);

        if(pos==-1)

                cout<<"element not found" ;
        else
                cout<<"element found at index "<<pos;

}

/* function for binary search*/

 int binary_search(int list[],int key,int low,int high)

{

        int mid,pos=-1;

        while(low<=high)

        {
                mid=(low+high)/2;

                if(key==list[mid])

                {
                        pos=mid;

                        break;
                }

                else if(key<list[mid])
```

```
                low=mid+1;
    }
    return pos;
}
```

Output:

Signature of the Faculty

Source code: Recursive C++ program for binary search

```
#include<iostream>
using namespace std;
int rbinary_search(int list[],int key,int low,int high);
int main()
{
int n,i,key,list[25],pos;
    cout<<"enter no of elements\n" ;
```

```
        cin>>n;
        cout<<"enter "<<n<<" elements in ascending order ";
        for(i=0;i<n;i++)
        cin>>list[i];
        cout<<"enter key to search" ;
        cin>>key;
        pos=rbinary_search(list,key,0,n-1);
        if(pos==-1)
                cout<<"element not found" ;
        else
                cout<<"element found at index "<<pos;
}
/*recursive function for binary search*/
int rbinary_search(int list[ ],int key,int low,int high)
{
int mid,pos=-1;

        if(low<=high)
        {
                mid=(low+high)/2;
                if(key==list[mid])
                {
                        pos=mid;
```

```
                }
                else if(key<list[mid])
                    return rbinary_search(list,key,low,mid-1);
                else
                        return rbinary_search(list,key,mid+1,high);
        }
        return pos;
}
```

Output:

Signature of the Faculty

WEEK-7: **DATE:**

Aim: write a C++ programs to implement

i) Bubble sort ii) Selection sort iii) quick sort

Description:

i)Bubble sort

The bubble sort is an example of exchange sort.In this method,repetitive comparisonis performed among elements and essential swapping of elements is done. Bubble sort is commonly used in sorting algorithms.

It is easy to understand but time consuming i.e. takes more number of comparisons to sort a list. In this type, two successive elements are comparedand swapping is done.Thus,step-by-step entire array elements are checked. It is different from the selection sort.

Instead of searching the minimum element and then applying swapping, two records are swapped instantly upon noticing that they are not in order.

Algorithm:

Bubble_Sort (A [ ] , N)

Step 1: Start

Step 2: Take an array of n elements

Step 3: for i=0,………….n-2

Step 4: for j=i+1,…….n-1

Step 5: if arr[j]>arr[j+1] then

Interchange arr[j] and arr[j+1]

End of if

Step 6: Print the sorted array arr

Step 7:Stop

Source code: Program to sort a list of numbers using bubble sort

```
#include<iostream>

using namespace std;

void bubble_sort(int list[30],int n);

int main()

{

int n,i;

int list[30];

cout<<"enter no of elements\n";

cin>>n;

cout<<"enter "<<n<<" numbers ";

for(i=0;i<n;i++)

cin>>list[i];

bubble_sort (list,n);

cout<<" after sorting\n";

for(i=0;i<n;i++)

cout<<list[i]<<endl;

return 0;

}

void bubble_sort (int list[30],int n)

{

        int temp ;
```

```
        int i,j;
        for(i=0;i<n;i++)
        for(j=0;j<n-1;j++)
        if(list[j]>list[j+1])
        {
                temp=list[j];
                list[j]=list[j+1];
                list[j+1]=temp;
        }
}
```

Output:

Signature of Faculty

ii) **Selection sort** (Select the smallest and Exchange):

The first item is compared with the emaining n-1 items, and whichever of all is lowest, is put in the first position. Then the second item from the list is taken and compared with the remaining (n-2) items, if an item with a value less than that of the second item is found on the (n-2) items, it is swapped (Interchanged) with the second item of the list and so on.

Algorithm:

Selection_Sort (A[ ] , N)

Step 1 : start Begin

Step 2 : Set POS = K

Step 3 : Repeat for J = K + 1 to N –1

Begin

If A[J] < A [POS]

Set POS = J

Step 4 : End For Swap A [K] End For with A [POS]

Step 5 : Stop

Source code: Program to implement selection sort

```
#include<iostream>
using namespace std;
void selection_sort (int list[],int n);
int main()
{
    int n,i;
    int list[30];
    cout<<"enter no of elements\n";
    cin>>n;
    cout<<"enter "<<n<<" numbers ";
    for(i=0;i<n;i++)
     cin>>list[i];
     selection_sort (list,n);
     cout<<" after sorting\n";
      for(i=0;i<n;i++)
      cout<<list[i]<<endl;
       return 0;
```

```
}
void selection_sort (int list[],int n)
{
    int min,temp,i,j;
    for(i=0;i<n;i++)
    {
        min=i;
        for(j=i+1;j<n;j++)
        {
            if(list[j]<list[min])
            min=j;
        }
        temp=list[i];
        list[i]=list[min];
        list[min]=temp;
    }
}
```

Output:

Signature of Faculty

iii) Quick sort:

It is a divide and conquer algorithm. Quick sort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quick sort can then recursively sort the sub-arrays.

ALGORITHM:

Step 1: Pick an element, called a pivot, from the array.

Step 2: Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.

Step 3: Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

Source code: To implement Quick sort

```
#include<iostream.h>

using namespace std;

void quicksort(int x[],int Lb,int Ub)

{

    int down,up,pivot,t;

        if(Lb<Ub)

        {
                down=Lb;
                up=Ub;
                pivot=down;
                while(down<up)
                {

                        while((x[down]<=x[pivot])&&(down<Ub))down++;

                        while(x[up]>x[pivot])
                        up--;
```

```
                    if(down<up)
                    {
                     t=x[down];
                     x[down]=x[up];
                     x[up]=t;
                    }/*endif*/
            }
            t=x[pivot];
            x[pivot]=x[up];
            x[up]=t;
        quicksort( x,Lb,up-1);
        quicksort( x,up+1,Ub);
      }
}
int main()
{
        int n,i;
        int list[30];
       cout<<"enter no of elements\n";
       cin>>n;
       cout<<"enter "<<n<<" numbers ";
       for(i=0;i<n;i++)
       cin>>list[i];
       quicksort(list,0,n-1);
```

```
        cout<<" after sorting\n";

        for(i=0;i<n;i++)

        cout<<list[i]<<endl;

return 0;

  }
```

Output:

Signature of Faculty

WEEK-8 **DATE:**

Write a program that implements the following

i) Insertion sort
ii) Merge sort
iii) Heap sort.

(i)Insertion Sort:

It iterates, consuming one input element each repetition, and growing a sorted output list. Each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

Algorithm:

Step 1: start

Step 2: for i ← 1 to length(A)

Step 3: j ← i

Step 4: while j > 0 and A[j-1] > A[j]

Step 5: swap A[j] and A[j-1]

Step 6: j ← j - 1

Step 7: end while

Step 8: end for

Step 9: stop

Source code: Program to implement Insertion Sort

```
#include<iostream>

using namespace std;

void insertion_sort(int a[],int n)

{

int i,t,pos;

        for(i=0;i<n;i++)
```

```
        {
                t=a[i];

                pos=i;

            while(pos>0&&a[pos-1]>t)
                {

                        a[pos]=a[pos-1];

                        pos--;
                }
                a[pos]=t;
        }
}

int main()
{
        int n,i;

         int list[30];

        cout<<"enter no of elements\n";

        cin>>n;

        cout<<"enter "<<n<<" numbers ";

        for(i=0;i<n;i++)

        cin>>list[i];

        insertion_sort(list,n);

        cout<<" after sorting\n";

        for(i=0;i<n;i++)

        cout<<list[i]<<endl;

         return 0;

}
```

Output:

Signature of Faculty

ii)Merge sort:

Merge sort is an O(*n* log *n*) comparison-based sorting algorithm. It is stable, meaning that it preserves the input order of equal elements in the sorted output. It is an example of the divide and conquer algorithmic paradigm.

Merge sort is so inherently sequential that it's practical to run it using slow tape drives as input and output devices. It requires very little memory, and the memory required does not change with the number of data elements.

If you have four tape drives, it works as follows:

1. Divide the data to be sorted in half and put half on each of two tapes.
2. Merge individual pairs of records from the two tapes; write two-record chunks alternately to each of the two output tapes.
3. Merge the two-record chunks from the two output tapes into four-record chunks; write these alternately to the original two input tapes.
4. Merge the four-record chunks into eight-record chunks; write these alternately to the original two output tapes.
5. Repeat until you have one chunk containing all the data, sorted --- that is, for log *n* passes, where *n* is the number of records.

Conceptually, merge sort works as follows:

1. Divide the unsorted list into two sublists of about half the size.
2. Divide each of the two sublists recursively until we have list sizes of length 1, in which case the list itself is returned.
3. Merge the two sublists back into one sorted list.

Source code: To implement Merge Sort

```
#include<iostream.h>
 using namespace std;
```

```
#define max 15

 template<class T>

void merge(T a[],int l,int m,int u)

 {
            T  b[max];
             int i,j,k;
             i=l; j=m+1; k=l;
           while((i<=m)&&(j<=u))
             {
                       if(a[i]<=a[j])
                          {
                                   b[k]=a[i];
                                    ++i;
                         }

                       else

                        {
                                    b[k]=a[j];
                                    ++j;
                          }
                        ++k;
            }
            if(i>m)
            {
                       while(j<=u)
                        {

                                   b[k]=a[j];

                                    ++j;

                                  ++k;

                        }

            }

            else


             {
```

```
                while(i<=m)
                {
                        b[k]=a[i];
                        ++i;
                        ++k;
                }
        }
        for(int r=l;r<=u;r++)
        a[r]=b[r];
}
template <class T>
void mergesort(T a[],int p,int q)
 {
 int mid;
 if(p<q)
  {
        mid=(p+q)/2;
        mergesort(a,p,mid);
        mergesort(a,mid+1,q);
        merge(a,p,mid,q);
        }

}
```

```
int main()
{
int n,i;
int list[30];
        cout<<"enter no of elements\n";
        cin>>n;
        cout<<"enter "<<n<<" numbers ";
        for(i=0;i<n;i++)
        cin>>list[i];
        mergesort (list,0,n-1);
        cout<<" after sorting\n";
        for(i=0;i<n;i++)
        cout<<list[i]<<endl;
return 0;
}
```

Output:

Signature of Faculty

(iii)HEAP SORT

Heap sort is a method in which a binary tree is used. In this method first the heap is created using binary tree and then heap is sorted using priority queue.

Source code:C++ program for implementation of Heap Sort

```
#include <iostream>

using namespace std;

        //      To heapify a subtree rooted with node i which is

        //      an index in arr[]. n is size of heap

void heapify(int arr[], int n, int i)

{

    int largest = i;  // Initialize largest as root

    int l = 2*i + 1;  // left = 2*i + 1

    int r = 2*i + 2;  // right = 2*i + 2

//     If left child is larger than root
      if (l < n && arr[l] > arr[largest])
        largest = l;

//     If right child is larger than largest so far
        if (r < n && arr[r] > arr[largest])

        largest = r;
     //If largest is not root

    if (largest != i)

    {
         swap(arr[i], arr[largest]);
//       Recursively heapify the affected sub-  tree
        heapify(arr, n, largest);

    }

}
```

```
//  main function to do heap sort
      void heapSort(int arr[], int n)
      {
        //Build heap (rearrange array)
        for (int i = n / 2 - 1; i >= 0; i--)
         heapify(arr, n, i);
//      One by one extract an element from heap
        for (int i=n-1; i>=0; i--)
        {
             //Move current root to end
              swap(arr[0], arr[i]);
             //call max heapify on the reduced heap
              heapify(arr, i, 0);
        }


  }

 /* A utility function to print array of size n */
 void printArray(int arr[], int n)
 {

    for (int i=0; i<n; ++i)

        cout << arr[i] << " ";

    cout << "\n";

 }

 int main()

 {

      int n,i;

      int list[30];
```

```
    cout<<"enter no of elements\n";

    cin>>n;

    cout<<"enter "<<n<<" numbers ";

    for(i=0;i<n;i++)

    cin>>list[i];

    heapSort(list, n);

    cout << "Sorted array is \n";

     printArray(list, n);

    return 0;

}
```

Output:

Signature of the Faculty

WEEK-9 **DATE:**

Aim: Write a program to implement all the functions of a dictionary (ADT)using Linked List.

Description:

Dictionary

The most common objective of computer programs is to store and retrieve data. Much of this book is about efficient ways to organize collections of data records so that they can be stored and retrieved quickly. In this section we describe a simple interface for such a collection, called a ***dictionary***.

The dictionary ADT provides operations for storing records, finding records, and removing records from the collection. This ADT gives us a standard basis for comparing various data structures. Loosly speaking, we can say that any data structure that supports insert, search, and deletion is a "dictionary".

Dictionaries depend on the concepts of a ***search key*** and ***comparable*** objects. To implement the dictionary's search function, we will require that keys be ***totally ordered***. Ordering fields that are naturally multi-dimensional, such as a point in two or three dimensions, present special opportunities if we wish to take advantage of their multidimensional nature. This problem is addressed by***spatial data structures***.

Source code: To to implement all the functions of a dictionary (ADT)

```
#include<stdlib.h>
#include<iostream.h>
class node
{
        public: int key;
        int value;
        node*next;
```

```
 };
class dictionary:public node

        {
                int k,data;
                node *head;

        public: dictionary();

        void insert_d( );

        void delete_d( );

        void display_d( );

};

dictionary::dictionary( )

{head=NULL;

}

//code to push an val into dictionary;
 void dictionary::insert_d( )
 {

        node *p,*curr,*prev;

        cout<<"Enter an key and value to be inserted:";

        cin>>k;

        cin>>data;

        p=new node;

        p->key=k;

        p->value=data;

        p->next=NULL;

        if(head==NULL)

            head=p;

        else

        {

                curr=head;
```

```
while((curr->key<p->key)&&(curr->next!=NULL))
 {
        prev=curr;
        curr=curr->next;
 }
 if(curr->next==NULL)

 {
        if(curr->key<p->key)
        {
                curr->next=p;
                prev=curr;
        }
        else
        {
                p->next=prev->next;
                prev->next=p;
        }
 }
 else
 {
        p->next=prev->next;
        prev->next=p;
 }
 cout<<"\nInserted into dictionary Sucesfully....\n";
```

```
        }
}
void dictionary::delete_d( )
{
        node*curr,*prev;
        cout<<"Enter key value that you want to delete...";
        cin>>k;
        if(head==NULL)
                cout<<"\ndictionary is Underflow";
        else
                {
                  curr=head;
                  while(curr!=NULL)
                  {
                     if(curr->key==k)
                       break;
                      prev=curr;
                     curr=curr->next;
                  }
         }
        if(curr==NULL)
                cout<<"Node not found...";
        else
         {
               if(curr==head)
               head=curr->next;
```

```
                    else
                            prev->next=curr->next;

                             delete curr;

                             cout<<"Item deleted from dictionary...";

          }

 }

void dictionary::display_d( )

{

           node*t;

          if(head==NULL)
                  cout<<"\ndictionary Under Flow";
         else
         {
                  cout<<"\nElements in the dictionary are....\n";

                  t=head;

              while(t!=NULL)
               {
                    cout<<t->key<<","<<t->value;

                    t=t->next;

              }

         }

}

int main( )

{

       int choice;

       dictionary d1;
         while(1)
```

```
        {
            cout<<"\n\n***Menu for Dictrionay operations***\n\n";
            cout<<"1.Insert\n2.Delete\n3.DISPLAY\n4.EXIT\n";
            cout<<"Enter Choice:";
            cin>>choice;
            switch(choice)
            {
                case 1:  d1.insert_d();
                    break;
                case 2:  d1.delete_d( );
                    break;
                case 3:  d1.display_d( );
                 break;
        case 4:  exit(0);
        default:cout<<"Invalid choice...Try again...\n";
        }
    }
    }
```

Output:

Signature Of Faculty

WEEK-10 **DATE:**

Aim: Write a C++ program to perform the following operations:

a) Insert an element into a binary search tree.

b) Delete an element from a binary search tree.

c) Search for a key element in a binary search tree.

Description:

Binary Search Tree:

So to make the searching algorithm faster in a binary tree we will go for building the binary search tree. The binary search tree is based on the binary search algorithm. While creating the binary search tree the data is systematically arranged.

That means values at **left sub-tree < root node value < right sub-tree values**.

Source code: To implement Binary Search tree

```
#include<stdlib.h>

#include<iostream.h>
class node

{

public:

        int data;

        node*lchild;

        node*rchild;

};
class bst:public node
{
      int item;

      node *root;

      public: bst();
```

```
        void insert_node();
        void delete_node();
        void display_bst();
        void inorder(node*);
    };
bst::bst()
{
      root=NULL;
 }
 void bst:: insert_node()
{
        node *new_node,*curr,*prev;
        new_node=new node;
        cout<<"Enter data into new node";
        cin>>item;
        new_node->data=item;
        new_node->lchild=NULL;
        new_node->rchild=NULL;
        if(root==NULL)
                root=new_node;
        else
                {
                 curr=prev=root;
                  while(curr!=NULL)
                  {
                        if(new_node->data>curr->data)
                          {
```

```
                    prev=curr;

                        curr=curr->rchild;


                }

                else

                        {
                                prev=curr;
                                curr=curr->lchild;
                         }

        }

        cout<<"Prev:"<<prev->data<<endl;

        if(prev->data>new_node->data)

                prev->lchild=new_node;

        else

                prev->rchild=new_node;

        }

}

//code to delete a node

void bst::delete_node()
{

    if(root==NULL)

     cout<<"Tree is Empty";
    else
    {

        int key;
        cout<<"Enter the key value to be deleted";
        cin>>key;
        node* temp,*parent,*succ_parent;
```

```
temp=root;

while(temp!=NULL)

{
     if(temp->data==key)

    {    //deleting node with two children
          if(temp->lchild!=NULL&&temp->rchild!=NULL)
          {//search for inorder sucessor
              node*temp_succ;
              temp_succ=temp->rchild;
                  while(temp_succ->lchild!=NULL)
                  {
                     succ_parent=temp_succ;
                     temp_succ=temp_succ->lchild;
                   }
                  temp->data=temp_succ->data;
                  succ_parent->lchild=NULL;
                 cout<<"Deleted sucess fully";
          }
          //deleting a node having one left child

          if(temp->lchild!=NULL&temp->rchild==NULL)

          {
                if(parent->lchild==temp)
                   parent->lchild=temp->lchild;
          else
                   parent->rchild=temp->lchild;

          temp=NULL;

          delete(temp);

          cout<<"Deleted sucess fully";

           return;

          }

 //deleting a node having one right child
     if(temp->lchild==NULL&temp->rchild!=NULL)

          {
```

```
                if(parent->lchild==temp)

                        parent->lchild=temp->rchild;

                else
                        parent->rchild=temp->rchild;
                temp=NULL;
                delete(temp);
                cout<<"Deleted sucess fully";

                return;

                }
                //deleting a node having no child
                if(temp->lchild==NULL&temp->rchild==NULL)
                {

                       if(parent->lchild==temp)
                        parent->lchild=NULL;
                    else
                        parent->rchild=NULL;

                        temp=NULL;

                       delete(temp);

                       cout<<"Deleted sucess fully";

                        return;

                }

        }

        else if(temp->data<key)

             {
                     parent=temp;
                     temp=temp->rchild;
              }

        else if(temp->data>key)
             {
                parent=temp;
                temp=temp->lchild;
              }
```

```
        }//end while

  }//end if

 }//end delnode func

  void bst::display_bst()
  {
        if(root==NULL)
            cout<<"\nBST Under Flow";
          else
            inorder(root);
  }

void bst::inorder(node*t)

{
        if(t!=NULL)

        {
                inorder(t->lchild);

                cout<<" "<<t->data;

                inorder(t->rchild);

        }

}

int main()
{

     bst bt;

    int i;

        while(1)

        {

                cout<<"****BST Operations****";

                cout<<"\n1.Insert\n2.Display\n3.del\n4.exit\n";
```

```
                cout<<"Enter Choice:";
                cin>>i;
                switch(i)
                {
                    case 1:bt.insert_node();
                           break;
                   case 2:bt.display_bst();
                          break;
                  case 3:bt.delete_node();
                          break;
                 case 4:exit(0);
                default: cout<<"Enter correct choice";
                }
        }
}
```

Output:

Signature of Faculty

WEEK-11 : **DATE:**

Write C++ programs that use recursive functions to traverse the given

binary tree in a)Preorder b) Inorder and c) Postorder

Aim: To implement Binary tree traversals(PreOrder,InOrder,PostOrder)

Description:

Binary tree traversals

It is often convenient to a single list containing all the nodes in a tree. This list may correspond to an order in which the nodes should be visited when the tree is being searched. We define three such lists here, the **preorder**, **postorder** and **inorder** traversals of the tree. The definitions themselves are recursive:

- if *T* is the empty tree, then the empty list is the preorder, the inorder and the postorder traversal associated with *T*;
- if *T* = [*N*] consists of a single node, the list [*N*] is the preorder, the inorder and the postorder traversal associated with *T*;
- otherwise, *T* contains a root node *n*, and subtrees *T*1,..., *T*n: and
- the *preorder* traversal of the nodes of *T* is the list containing *N*, followed, in order by the preorder traversals of *T*1..., *T*n;
- the *inorder* traversal of the nodes of *T* is the list containing the inorder traversal of *T*1 followed by *N* followed in order by the inorder traversal of each of *T*2,..., *T*n.
- the *postorder* traversal of the nodes of *T* is the list containing in order the postorder traversal of each of *T*1,..., *T*n, followed by *N*.

Source code:

```
#include<stdlib.h>

#include<iostream.h>
```

```
class node

{

public:

        int data;

        node*Lchild;

        node*Rchild;

};

class bst
{

        int item;

        node *root;

        public: bst();

        void insert_node();

        void delete_node();

        void display_bst();

        void preeorder(node*);

        void inorder(node*);

        void postorder(node*);

};
bst::bst()

{

root=NULL;

}

void bst:: insert_node()
```

```
{
        node *new_node,*curr,*prev;
        new_node=new node;
        cout<<"Enter data into new node";
        cin>>item;
        new_node->data=item;
        new_node->Lchild=NULL;
        new_node->Rchild=NULL;
        if(root==NULL)
                root=new_node;
        else
        {
                curr=prev=root;
                while(curr!=NULL)
                {
                        if(new_node->data>curr->data)
                        {
                                prev=curr;
                                curr=curr->Rchild;
                        }
                        else
                        {
                                prev=curr;
```

```
                    curr=curr->Lchild;
                }
            }
            cout<<"Prev:"<<prev->data<<endl;
            if(prev->data>new_node->data)
                prev->Lchild=new_node;
            else
                prev->Rchild=new_node;
            }
}
//code to delete a node
void bst::delete_node()
{
    if(root==NULL)
     cout<<"Tree is Empty";
   else
    {
        int key;
        cout<<"Enter the key value to be deleted";
        cin>>key;
        node* temp,*parent,*succ_parent;
        temp=root;
        while(temp!=NULL)
        {
```

```
if(temp->data==key)
{ //deleting node with two children
  if(temp->Lchild!=NULL&&temp->Rchild!=NULL)
  {//search for sucessor
        node*temp_succ;
        temp_succ=temp->Rchild;
          while(temp_succ->Lchild!=NULL)
            {
                    succ_parent=temp_succ;

                    temp_succ=temp_succ->Lchild;

             }

          temp->data=temp_succ->data;
          succ_parent->Lchild=NULL;
          cout<<"Deleted sucess fully";

          return;

      }

      //deleting a node having one left child

      if(temp->Lchild!=NULL&temp->Rchild==NULL)

      {

      if(parent->Lchild==temp)

            parent->Lchild=temp->Lchild;

      else

            parent->Rchild=temp->Lchild;

      temp=NULL;

      delete(temp);
```

```
                        cout<<"Deleted sucess fully";

                        return;

                        }

        //deleting a node having one right child


      if(temp->Lchild==NULL&temp->Rchild!=NULL)
      {
            if(parent->Lchild==temp)
                        parent->Lchild=temp->Rchild;
                        else
                        parent->Rchild=temp->Rchild;
                        temp=NULL;
                        delete(temp);
                        cout<<"Deleted sucess fully";
                        return;
       }
        //deleting a node having no child
             if(temp->Lchild==NULL&temp->Rchild==NULL)
            {
                        if(parent->Lchild==temp)
                              parent->Lchild=NULL;
                        else
                              parent->Rchild=NULL;
                             temp=NULL;
                             delete(temp);
                            cout<<"Deleted sucess fully";
                          return;


            }
    }
```

```
                else if(temp->data<key)
                    {
                        parent=temp;
                        temp=temp->Rchild;
                    }
                else if(temp->data>key)
                    {
                        parent=temp;
                        temp=temp->Lchild;
                    }
        }//end while
  }//end if
 }//end delnode func
void bst::display_bst()
{
        if(root==NULL)
                cout<<"\nBinary Search Tree is Under Flow";
        else
        {
                 int ch;
                 cout<<"\t\t**Binart Tree Traversals**\n";
                 cout<<"\t\t1.Preeorder\n\t\t2.Inorder\n\t\t3:PostOrder\n";
                 cout<<"\t\tEnter Your Chice:";
                 cin>>ch;
                switch(ch)
```

```
                    {
                        case 1: cout<<"Pree order Tree Traversal\n ";
                                preeorder(root);
                                break;
                        case 2: cout<<"Inorder Tree Traversal is\n ";
                                inorder(root);
                                break;
                        case 3: cout<<"Inorder Tree Traversal is\n";
                                postorder(root);
                                break;
                    }
            }
}
void bst::inorder(node*t)
{
        if(t!=NULL)
        {
                inorder(t->Lchild);
                cout<<" "<<t->data;
                inorder(t->Rchild);
        }
}
void bst::preeorder(node*t)
```

```
        {
                if(t!=NULL)
                {
                        cout<<" "<<t->data;
                        preeorder(t->Lchild);
                        preeorder(t->Rchild);
                }
}
void bst::postorder(node*t)
{
         if(t!=NULL)
         {
                 postorder(t->Lchild);
                 postorder(t->Rchild);
                 cout<<" "<<t->data;
         }
}
int main()
{
        bst bt;
        int i;
         while(1)
         {
                cout<<"\n\n***Operations Binary Search Tree***\n";
                cout<<"1.Insert\n2.Display\n3.del\n4.exit\n"; cout<<"Enter Choice:";
                cin>>i;
                switch(i)
```

```
                {
                        case 1:bt.insert_node();
                                break;
                        case 2:bt.display_bst();
                                break;
                        case 3:bt.delete_node();
                                break;
                        case 4:exit(0);
                         default:cout<<"Enter correct choice";
                 }
        }
}
```

Output :

Signature of the Faculty

WEEK -12 **DATE:**

Aim: Write a C++ program to perform the following operations
a)Insertion into an AVL-tree b) Deletion from an AVL-tree
c) Search for a key element in a AVL tree.

Description:

AVL Tree

AVL tree is a height-balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced if, the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1. In other words, a binary tree is said to be balanced if the height of left and right children of every node differ by either -1, 0 or +1. In an AVL tree, every node maintains an extra information known as "**Balance factor**".

The AVL tree was introduced in the year 1962 by G.M. Adelson-Velsky and E.M. Landis. An AVL tree is defined as follows...

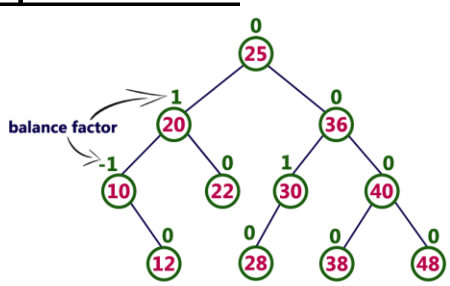
An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.

Balance factor of a node is the difference between the heights of the left and right subtrees of that node. The balance factor of a node is calculated either **height of left subtree - height of right subtree** (OR) **height of right subtree - height of left subtree**.

In the following explanation, we calculate as follows...

Balance factor = heightOfLeftSubtree - heightOfRightSubtree

Example of AVL Tree



The above tree is a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an AVL tree.

Operations on an AVL Tree

The following operations are performed on AVL tree...

1. **Search**
2. **Insertion**
3. **Deletion**

1.Search Operation in AVL Tree

In an AVL tree, the search operation is performed with **O(log n)** time complexity. The search operation in the AVL tree is similar to the search operation in a Binary search tree. We use the following steps to search an element in AVL tree...

Step 1 - Read the search element from the user.

Step 2 - Compare the search element with the value of root node in the tree.

Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function

Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.

Step 5 - If search element is smaller, then continue the search process in left subtree.

Step 6 - If search element is larger, then continue the search process in right subtree.

Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node.

Step 8 - If we reach to the node having the value equal to the search value, then display "Element is found" and terminate the function.

Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

2.Insertion Operation in AVL Tree

In an AVL tree, the insertion operation is performed with **O(log n)** time complexity. In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows...

Step 1 - Insert the new element into the tree using Binary Search Tree insertion logic.

Step 2 - After insertion, check the **Balance Factor** of every node.

Step 3 - If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.

Step 4 - If the **Balance Factor** of any node is other than **0 or 1 or -1** then that tree is said to be imbalanced. In this case, perform suitable **Rotation** to make it balanced and go for next operation.

3.Deletion Operation in AVL Tree

The deletion operation in AVL Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Balance Factor condition. If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.

Aim: To implement AVL tree

Source code:

```
#include <iostream.h>

#include <stdlib.h>

#include conio.h>
struct node
{

    int element;
    node *left;
    node *right;
    int height;
};
```

```
typedef struct
 node *np;
class bstree
{
    public:
       void insert(int,np&);
       void del(int, np &);
       int deletemin(np &);
       void find(int,np &);
       np findmin(np);
       np findmax(np);
       void copy(np &,np &);
       void makeempty(np&);
       np nodecopy(np &);
       void preorder(np);
       void inorder(np);
     void postorder(np);
     int bsheight(np);
     np srl(np &);
     np drl(np &);
     np srr(np &);
     np drr(np &);
     int max(int,int);
     int nonodes(np);
};

//Inserting a node

void bstree::insert(int x,np &p)

{

          if (p == NULL)
          {
```

```
        p = new node;

        p->element = x;

        p->left=NULL;

        p->right = NULL;

        p->height=0;

        if (p==NULL)

                cout<<"Out of Space";

}

else

{

        if (x<p->element)

        {

          insert(x,p->left);


          if ((bsheight(p->left) - bsheight(p->right))==2)

          {

            if (x < p->left->element)

                p=srl(p);

            else

                p = drl(p);

          }

        }
        else if (x>p->element)

        {
            insert(x,p->right);
```

```
        if ((bsheight(p->right) - bsheight(p->left))==2)
        {
                if (x > p->right->element)
                p=srr(p);
                else
                  p = drr(p);
        }
    }
    else
            cout<<"Element Exists";
    }
    int m,n,d;
    m=bsheight(p->left);
    n=bsheight(p->right);
    d=max(m,n);
    p->height = d + 1;
}
//Finding the Smallest
np bstree::findmin(np p)
{
    if (p==NULL)
    {
      cout<<"Empty Tree ";
      return p;
```

```
        }
        else
        {
          while(p->left !=NULL)
                p=p->left;
          return p;
        }
}
//Finding the Largest
np bstree::findmax(np p)
{
        if (p==NULL)
        {
          cout<<"Empty Tree ";
          return p;
        }
        else
        {
          while(p->right !=NULL)
            p=p->right;
          return p;
        }
}
```

```
//Finding an element

void bstree::find(int x,np &p)

{

        if (p==NULL)

          cout<<" Element not found ";

        else

        if (x < p->element)

          find(x,p->left);

        else

           if (x>p->element)

           find(x,p->right);

             else

             cout<<"   Element found !";

}
//Copy a tree

void bstree::copy(np &p,np &p1)

{

        makeempty(p1);

        p1 = nodecopy(p);

}

//Make a tree empty

void bstree::makeempty(np &p)

{
```

```
        np d;
        if (p != NULL)
        {
          makeempty(p->left);
          makeempty(p->right);
          d=p;
          free(d);
          p=NULL;
        }
}
//Copy the nodes
np bstree::nodecopy(np &p)
{

        np temp;
        if (p==NULL)
          return p;
        else
        {
          temp = new node;
          temp->element = p->element;
          temp->left = nodecopy(p->left);
          temp->right = nodecopy(p->right);
          return temp;
```

```
        }
}
//Deleting a node
void bstree::del(int x,np &p)
{
        np d;
        if (p==NULL)
          cout<<"Element not found ";
        else if ( x < p->element)
          del(x,p->left);
        else if (x > p->element)
        del(x,p->right);
        else if ((p->left == NULL) && (p->right == NULL))
        {
          d=p;
          free(d);
          p=NULL;
          cout<<" Element deleted !";
        }
        else if (p->left == NULL)
        {
         d=p;
         free(d);
```

```
  p=p->right;
  cout<<" Element deleted !";
 }
 else if (p->right == NULL)
 {
  d=p;
  p=p->left;
  free(d);
  cout<<" Element deleted !";
 }
 else
  p->element = deletemin(p->right);

}
int bstree::deletemin(np &p)
{
 int c;
 cout<<"inside deltemin";
 if (p->left == NULL)
 {
  c=p->element;
  p=p->right;
  return c;
```

```
        }
        else
        {
         c=deletemin(p->left);
         return c;
        }
}
void bstree::preorder(np p)
{
        if (p!=NULL)
{
         cout<<p->element<<"-->";
         preorder(p->left);
         preorder(p->right);
        }
}
//Inorder Printing
void bstree::inorder(np p)
{
        if (p!=NULL)
        {
          inorder(p->left);
          cout<<p->element<<"-->";
          inorder(p->right);
```

```
    }
}
//PostOrder Printing
void bstree::postorder(np p)
{
    if (p!=NULL)
    {
      postorder(p->left);
      postorder(p->right);
      cout<<p->element<<"-->";
    }
}

int bstree::max(int value1, int value2)
{
    return ((value1 > value2) ? value1 : value2);
}
int bstree::bsheight(np p)
{
    int t;
    if (p == NULL)
        return -1;
    else
    {
        t= p->height;
```

```
            return t;
        }
}

np bstree:: srl(np &p1)

{

        np p2;

        p2 = p1->left;

        p1->left = p2->right;

        p2->right = p1;

        p1->height = max(bsheight(p1->left),bsheight(p1->right)) + 1;

        p2->height = max(bsheight(p2->left),p1->height) + 1; return p2;


}

np bstree:: srr(np &p1)
{

        np p2;

        p2 = p1->right;

        p1->right = p2->left;

        p2->left = p1;

        p1->height = max(bsheight(p1->left),bsheight(p1->right)) + 1;

        p2->height = max(p1->height,bsheight(p2->right)) + 1;

         return p2;

}

np bstree:: drl(np &p1)

{

        p1->left=srr(p1->left);

        return srl(p1);
```

```
}
np bstree::drr(np &p1)
{
        p1->right = srl(p1->right);
        return srr(p1);
}
int bstree::nonodes(np p)
{
        int count=0;
        if (p!=NULL)

        {
                nonodes(p->left);
                nonodes(p->right);
                count++;
        }
        return count;
}
int main()
{
        //clrscr();
        np root,root1,min,max;//,flag;
        int a,choice,findele,delele,leftele,rightele,flag;
```

```
char ch='y';

bstree bst;
//system("clear");

root = NULL;

root1=NULL;

while(1)

{

        cout<<"         \nAVL Tree\n";

        cout<<"         ========\n";

        cout<<"1.Insertion\n2.FindMin\n";

        cout<<"3.FindMax\n4.Find\n5.Copy\n";

        cout<<"6.Delete\n7.Preorder\n8.Inorder\n";

        cout<<"9.Postorder\n10.height\n11.EXIT\n";

        cout<<"Enter the choice:";

        cin>>choice;

        switch(choice)

        {

        case 1:

                cout<<"New node's value ?";

                cin>>a;

                bst.insert(a,root);

                break;

        case 2:
```

```
            if (root !=NULL)
            {
                min=bst.findmin(root);
                cout<<"Min element : "<<min->element;
            }
            break;
    case 3:
            if (root !=NULL)
            {
                max=bst.findmax(root);
                cout<<"Max element :"<<max->element;
            }
            break;
    case 4:
            cout<<"Search node : ";
            cin>>findele;
            if (root != NULL)
                bst.find(findele,root);
            break;
    case 5:
            bst.copy(root,root1);
              bst.inorder(root1);
            break;
    case 6:
```

```
            cout<<"Delete Node ?";

            cin>>delele;

            bst.del(delele,root);

            bst.inorder(root);

            break;

    case 7:

            cout<<" Preorder Printing... :";

            bst.preorder(root);

            break;

    case 8:
             cout<<" Inorder Printing.... :";

             bst.inorder(root);
             break;

    case 9:

              cout<<" Postorder Printing... :";
              bst.postorder(root);
              break;

    case 10:

            cout<<" Height and Depth is ";

            cout<<bst.bsheight(root);

            //cout<<"No. of nodes:"<<bst.nonodes(root);

            break;

    case 11:exit(0);

    }

    }
```

```
    return 0;
}
```

Output :

Signature of the Faculty

Bangladesh Army University of Engineering & Technology (BAUET)

Qadirabad, Natore 6431, Bangladesh

Department of Information and Communication Engineering (ICE)

Data Structures and Algorithms Lab Manual

Course Code	ICE-2112
Course Title	Data Structures and Algorithms Sessional
Credit Hours	1.5

Prepared by	Verified By
Nasirul Mumenin Lecturer, Dept. of ICE	Dr. Md. Rubel Bashar Head, Dept. of ICE

Table of Content

Content Beyond Syllabi		
1	*Write a C Program to check whether two given lists are containing the same data.	52
2	*Write a C program to find the largest element in a given doubly linked list.	54
3	*Write a C program to reverse the elements in the stack using recursion.	56
4	*Write a C program to implement stack using linked list.	58
5	*Write a C program to count the number of nodes in the binary search tree.	62
6	*Write a C program to sort an array of integers in ascending order using radix sort.	64
7	*Write a C program to sort a given list of strings.	66

***Content beyond the university prescribed syllabi**

ANNEXURE		
	Assessment Rubric	67
	Program Outcomes	70
	Knowledge Profile	72
	Complex Engineering Problem	72
	Complex Engineering Activities	73

General Guideline and Safety Instructions

1. Strictly follow the written and verbal instructions given by the teacher /Lab Instructor. If you do not understand the instructions, the handouts and the procedures, ask the instructor or teacher.
2. Students are required to attend all labs with official dress code and wearing ID card.
3. Mobile phones should be switched off in the lab. Keep bags in the bag shelf.
4. Keep the labs clean at all times, no food and drinks allowed inside the lab.
5. Students should work individually/team in the hardware and software task.
6. Students have to bring the lab manual cum lab report file along with them whenever they come for lab work.
7. Should take only the lab manual, calculator (if needed) and a pen or pencil to the work area.
8. Should utilize 3 hours’ time properly to perform the experiment and to record the readings. Do the calculations, draw the graphs and take signature from the instructor.
9. If the experiment is not completed in the stipulated time, the pending work has to be carried out in the leisure hours or extended hours.
10. Intentional misconduct will lead to expulsion from the lab.
11. Do not handle any equipment without reading the safety instructions. Read the handout and procedures in the Lab Manual before starting the experiments.
12. Do your wiring, setup, and a careful circuit checkout before applying power. Do not make circuit changes or perform any wiring when power is on.
13. Avoid contact with energized electrical circuits.
14. Do not insert connectors forcefully into the sockets.
15. **NEVER** try to experiment with the power from the wall plug.
16. Immediately report dangerous or exceptional conditions to the Lab instructor / teacher: Equipment that is not working as expected, wires or connectors are broken, the equipment that smells or “smokes”. If you are not sure what the problem is or what's going on, switch off the Emergency shutdown.
17. Never use damaged instruments, wires or connectors. Hand over these parts to the Lab instructor/Teacher.
18. After completion of Experiment, return the bread board, trainer kits, wires, oscilloscope probes and other components to lab staff. Do not take any item from the lab without permission.
19. Handling of Semiconductor Components: Sensitive electronic circuits and electronic components have to be handled with great care. The inappropriate handling of electronic component can damage or destroy the devices. The devices can be destroyed by driving to high currents through the device, by overheating the device, by mixing up the polarity, or by electrostatic discharge (ESD). Therefore, always handle the electronic devices as indicated by the handout, the specifications in the data sheet or other documentation.
20. Special Precautions during soldering practice
    a. Hold the soldering iron away from your body. Don't point the iron towards you.
    b. Don't use a spread solder on the board as it may cause short circuit.
    c. Do not overheat the components as excess heat may damage the components/board.
    d. In case of burn or injury seek first aid available in the lab or at the college dispensary.

Course Description

Introduction: Data types & data structures, data structure operations, Introduction to algorithms, performance analysis.

Arrays, Records and Pointer: Linear arrays, Relationships of arrays, Operation on arrays, Multidimensional arrays, pointer arrays, Record structures, representation of records, Sparse matrices.
Stacks, Queues and Recursion: Fundamentals, Different types of stacks and queues: circular, de-queues, etc., Evaluation of expressions, recursion, direct and indirect recursion, depth of recursion, Implementation of recursive procedures by stacks.
Linked List: Linked lists, Representation of linked list, Traversing & searching a linked list, Doubly linked list & dynamic storage management, Generalized list, Garbage collection & compaction.
Trees and Graphs: Basic terminology, Binary trees, Binary tree representation, Tree traversal, Extended binary tree, Huffman codes/algorithm, Graphs, Graph representation, Shortest path and transitive closure, Traversing a graph.
Sorting & Searching algorithms: Basic strategies of Algorithm design, Time and space analysis of algorithms, Average, best and worst case analysis, different notations. Sorting, Insertion sort, Shell sort, Heap sort, Radix sort, the general method of divide & conquer method, Merge sort, Quick sort, Selection sort, binary search, spanning trees,single source shortest paths.
Symbol Tables: Static tree tables, Dynamic tree tables, Hash tables overflow handling, Theoretical evaluation of overflow techniques.
Dynamic programming: The general method, multistage graphs, all pairs shortest paths, single source shortest paths problems.

Course Objective:

- **i)** **Develop knowledge** of basic data structures for storage and retrieval of ordered or unordered data. **Apply** information and concepts in data structure and algorithms fundamentals with the familiarity of issues.
- **ii)** **Design** algorithms for the creation, insertion, deletion, searching, and sorting of each data structure.
- iii) **Formulate** new solutions for programming problems or improve existing code using learned algorithms and data structures.

Statement of Course Outcomes (CO):

1. Develop knowledge of basic data structures for storage and retrieval of ordered or unordered data. **Apply** information and concepts in data structure and algorithms fundamentals with the familiarity of issues.

2. Formulate new solutions for programming problems or improve existing code using learned algorithms and data structures.

3. Design or develop algorithms for creation, insertion, deletion, searching, and sorting of each data structure in any complex engineering problem.

Statement of Course Outcome (CO) and their mapping

<table>
<tr><th colspan="7">Knowledge Profile(s): WK3, WK4, WK5</th></tr>
<tr><th>Course Outcomes (CO)</th><th>Statements</th><th>PO/ WA</th><th>POI Code</th><th>Bloom’s Taxonomy Level</th><th>Delivery methods and activities</th><th>Assessment Tools</th></tr>
<tr><td>CO1</td><td>Develop knowledge of basic data structures for storage and retrieval of ordered or unordered data. Apply information and concepts in data structure and algorithms fundamentals with the familiarity of issues.</td><td>1</td><td>1.3.1</td><td>C2,C1,C4</td><td rowspan="3">Lectures, Handouts, White Board Writing, Textbook,</td><td rowspan="3">Class Observation/ Homework, Assignments, Class Tests, Semester Final Examination (Written)</td></tr>
<tr><td>CO2</td><td>Formulate new solutions for programming problems or improve existing code using learned algorithms and data structures.</td><td>2</td><td>2.2.4</td><td>C4, C5</td></tr>
<tr><td>CO3</td><td>Design or develop algorithms for creation, insertion, deletion, searching, and sorting of each data structure in any complex engineering problem.</td><td>3</td><td>3.2.6</td><td>C6</td></tr>
</table>

Assessment Criteria and Marks Distribution

Si. No.	Particulars	Marks
1	Lab Performance	10
2	Lab Reports	20
3	Lab Test	40
4	Quiz Test	20
5	Lab Viva	10
Total		100

Summary of Lab Report

Ex. No.	Experiment Title	Date of Exp.	Date of Subm	Marks	Teacher's Signature	Remarks
01	Write a C program to analyze the structure of array. a) Create an array b) Display the array in reverse order c) Find the size of the array using sizeof function d) Display the addition operation in array e) Traverse the array using pointer.					
02	Write a C program that uses functions to perform the following: a) Create a singly linked list of integers. b) Delete a given integer from the above linked list. c) Display the contents of the above list after deletion.					
03	Write a C program that uses functions to perform the following: a) Create a doubly linked list of integers. b) Delete a given integer from the above doubly linked list. c) Display the contents of the above list after deletion.					
04	Write a C program that uses stack operations to convert a given infix expression into its postfix Equivalent, Implement the stack using an array.					
05	Write C programs to implement a double ended queue ADT using i) array and ii) doubly linked list respectively.					
06	Write a C program that uses functions to perform the following: a) Create a binary search tree of characters. b) Traverse the above Binary search tree recursively in Postorder.					
07	Write a C program that uses functions to perform the following: a) Create a binary search tree of integers. b) Traverse the above Binary search tree non recursively in inorder.					
08	Write C programs for implementing the following sorting methods to arrange a list of integers in ascending order: a) Insertion sort b) Merge sort					
09	Write C programs for implementing the following sorting methods to arrange a list of integers in ascending order:					

	a) Quick sort b) Selection sort					
10	Write a C program to perform the following operation: i) Insertion into a B-tree ii) Write a C program for implementing Heap sort algorithm for sorting a given list of integers in ascending order.					
11	Write a C program to implement all the functions of a dictionary (ADT) using hashing.					
12	Write a C program for implementing Knuth-Morris- Pratt pattern matching algorithm.					
13	Write C programs for implementing the following graph traversal algorithms: a) Depth first traversal b) Breadth first traversal					
	Average marks in lab Report (out of 20)					

Format of Lab Report

All lab reports should have to follow the common format as below

- Experiment No
- Experiment Title
- Objectives
- Theory Overview (with formula/equations and/or figure if required)
- Diagram (with adequate labeling and figure caption)
- Results
- Discussion
- Conclusion

Instruction for Lab Report Writing

- Lab report must be hand written without copying from other works.
- Writing should be neat and clean with proper caption and labeling in figure and table.
- The title page of report should contain all the basic information such as experiment no & title, course code & title, student's information, teacher's information, experiment date, submission date.
- Result should include calculated and/or simulated and/or measured data with proper unit.
- Table and/or graph of result should be neat and clear with axis label and units where applicable.
- The discussion should present your findings from the experiment. Evaluate the outcome objectively, taking a candid and unbiased point of view. Suppose that the outcome is not close to what you expected. Even then, after checking your results, give reasons why you believe that outcome is not consistent with the expected.
- In discussion, state the discrepancies between the experimental results and the model (theory), and discuss the sources of the differences in terms of the errors by offering logical inferences and suggest improvements.
- Conclusion should present, a brief summary of what was done, how it was done, show the results and conclusions of the experiment.
- Report should be submitted timely, late submission will cause reduction of marking.

- All lab reports have to be maintained in a single file which has to bring in every laboratory class.

EXPERIMENT 1

1.1 OBJECTIVE

1. To create a singly linked list of integers.
2. Delete a given integer from the above linked list.
3. Display the contents of the above list after deletion.

1.2 RESOURCE:

Turbo C

1.3 PROGRAM LOGIC

1. Create a node using structure
2. Dynamically allocate memory to node
3. Create and add nodes to linked list

1.4 PROCEDURE

Go to debug -> run or press CTRL + F9 to run the program.

1.5 SOURCE CODE

<u>program to create a single linked list, delete the contents and display the contents</u>

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
/*declaring a structure to create a node*/
struct node
{
    int data;
    struct node *next;
};
struct node *start;

/* inserting nodes into the list*/
/*function to insert values from beginning of the single linked list*/
void insertbeg(void)
{
    struct node *nn;
    int a;
    /*allocating implicit memory to the node*/
    nn=(struct node *)malloc(sizeof(struct node));
    printf("enter data:");
    scanf("%d",&nn->data);
    a=nn->data;
    if(start==NULL)          /*checking if List is empty*/
    {
        nn->next=NULL;
        start=nn;
    }
    else
    {
        nn->next=start;
        start=nn;
    }
    printf("%d succ. inserted\n",a);
    return;
}
 /*function to insert values from the end of the linked list*/
```

```
void insertend(void)
{
      struct node *nn,*lp;int b;
      nn=(struct node *)malloc(sizeof(struct node));
      printf("enter data:");
      scanf("%d",&nn->data);
      b=nn->data;
      if(start==NULL)
      {
            nn->next=NULL;
            start=nn;
      }
      else
      {
            lp=start;
            while(lp->next!=NULL)
            {
                  lp=lp->next;
            }
            lp->next=nn;
            nn->next=NULL;
      }
      printf("%d is succ. inserted\n",b);
      return;
}
```

/*function to insert values from the middle of the linked list*/

```
void insertmid(void)
{
      struct node *nn,*temp,*ptemp;int x,v;
      nn=(struct node *)malloc(sizeof(struct node));
      if(start==NULL)
      {
            printf("sll is empty\n"); return;
      }
      printf("enter data before which no. is to be inserted:\n");
      scanf("%d",&x);
      if(x==start->data)
      {
            insertbeg();
            return;
      }
      ptemp=start;
      temp=start->next;
      while(temp!=NULL&&temp->data!=x)
      {
            ptemp=temp;
            temp=temp->next;
      }
      if(temp==NULL)
      {
            printf("%d data does not exist\n",x);
      }
      else
      {
                  printf("enter data:");
                  scanf("%d",&nn->data);
                  v=nn->data;
                  ptemp->next=nn;
```

```
            nn->next=temp;
            printf("%d succ. inserted\n",v);
    }
    return;
}
 /*deletion operation*/
void deletion(void)
{
    struct node *pt,*t;
    int x;
    if(start==NULL)
    {
        printf("sll is empty\n");
        return;
    }
    printf("enter data to be deleted:");
    scanf("%d",&x);
    if(x==start->data)
    {
        t=start;
        /* assigning first node pointer to next nod pointer to delete a data
        from the starting of the node*/
        start=start->next;
        free(t);
        printf("%d is succ. deleted\n",x);
        return;
    }
    pt=start;
    t=start->next;
    while(t!=NULL&&t->data!=x)
    {
        pt=t;t=t->next;
    }
    if(t==NULL)
    {
        printf("%d does not exist\n",x);return;
     }
    else
    {
        pt->next=t->next;
    }
    printf("%d is succ. deleted\n",x);
    free(t);
    return;
}
void display(void)
{
    struct node *temp;
    if(start==NULL)
    {
        printf("sll is empty\n");
        return;
    }
    printf("elements are:\n");
    temp=start;
    while(temp!=NULL)
    {
        printf("%d\n",temp->data);
```

```
            temp=temp->next;
        }
        return;
    }
    /* main program*/
    int main()
    {
        int c,a;    start=NULL;
        do
        {
            printf("1:insert\n2:delete\n3:display\n4:exit\nenter choice:");
            scanf("%d",&c);
            switch(c)
            {
                case 1:
                printf("1:insertbeg\n2:insert end\n3:insert mid\nenter choice:");
                scanf("%d",&a);
                switch(a)
                {
                    case 1:insertbeg(); break;
                    case 2:insertend(); break;
                    case 3:insertmid(); break;
                }
                break;
                case 2:deletion(); break;
                case 3:display(); break;
                case 4:printf("program ends\n");break;
                default:printf("wrong choice\n");
                break;
            }
        }while(c!=4);return 0;
    }
```

1.6 PRE LAB QUESTIONS

1. What is data structure
2. How the memory is allocated dynamically
3. What is linked list
4. What is node
5. What are the types of linked list

1.7 LAB ASSIGNMENT

1. Write a program to insert a node at first , last and at specified position of linked list
2. Write a program to delete a node from first, last and at specified position of linked list

1.8 POST LAB QUESTIONS

1. How to represent linked list
2. How will you traverse linked list in reverse order
3. List the advantages and disadvantages of linked list?

1.9 INPUT AND OUTPUT

```
geetha@iare:~
[geetha@iare ~]$ gcc week1.c
[geetha@iare ~]$ ./a.out
1:insert
 2:delete
 3:display
 4:exit
 enter choice:1
1:insertbeg
 2:insertend
3:insertmid
enter choice:1
enter data:30
30 succ inserted
1:insert
 2:delete
 3:display
 4:exit
 enter choice:1
1:insertbeg
 2:insertend
3:insertmid
enter choice:1
enter data:20
20 succ inserted
```

```
geetha@iare:~
 4:exit
 enter choice:3
elements are:
20
30
1:insert
 2:delete
 3:display
 4:exit
 enter choice:2
enter data to be deleted20
 20s succ deletion
1:insert
 2:delete
 3:display
 4:exit
 enter choice:3
elements are:
30
1:insert
 2:delete
 3:display
 4:exit
 enter choice:
```

EXPERIMENT 2

2.1 OBJECTIVE

1. Create a doubly linked list of integers.
2. Delete a given integer from the above doubly linked list.
3. Display the contents of the above list after deletion.

2.2 RESOURCE

Turbo C

2.3 PROGRAM LOGIC

1. Create a node using structure
2. Dynamically allocate memory to node
3. Create and add nodes to linked list

2.4 PROCEDURE

Go to debug -> run or press CTRL + F9 to run the program

2.5 SOURCE CODE

<u>Program to create a double linked list to inserting, deleting and displaying the contents</u>

```
#include<stdio.h>
#include<stdlib.h>
/*declaring a structure to create a node*/
struct node
{
      struct node *prev;
      int data;
      struct node *next;
};
struct node *start,*nt;
/* inserting nodes into the list*/
/*function to insert values from beginning of the the double linked list*/

void insertbeg(void)
{
     int a;
     struct node *nn,*temp;
/*allocating implicit memory to the node*/
     nn=(struct node *)malloc(sizeof(struct node));
     printf("enter data:");
     scanf("%d",&nn->data);
     a=nn->data;
    if(start==NULL)     /*checking if List is empty*/
   {
       nn->prev=nn->next=NULL;
       start=nn;
   }
   else
  {
      nn->next=start;
      nn->prev=NULL;
      start->prev=nn;
      start=nn;
   }
   printf("%d succ inserted \n",a);
```

```
}
/*function to insert values from the end of the linked list*/

void insertend(void)
{
     int b;
     struct node *nn,*lp;
     nn=(struct node *)malloc(sizeof(struct node));
     printf("enter data:");
     scanf("%d",&nn->data);
     b=nn->data;
     if(start==NULL)
   {
/* assigning first node pointer to next nod pointer to delete a data from the starting
    of the node*/

        nn->prev=nn->next=NULL;
        start=nn;
   }
   else
  {
      lp=start;
     while(lp->next!=NULL)
    {
        lp=lp->next;
     }
     nn->prev=lp;
     lp->next=nn;
     nn->next=NULL;
  }
 printf("%d succ inserted\n",b);
}
/*function to insert values from the middle of the linked list*/

void insertmid(void)
{
     struct node *nn,*temp,*ptemp;
     int x,c;
    if(start==NULL)
    {
       printf("dll is empty\n");
       return;
    }
     printf("enter data before which nn is to be inserted\n");
     scanf("%d",&x);
     if(x==start->data)
     {
        insertbeg();
      }
      ptemp=start;
      temp=start->next;
      while(temp->next!=NULL&&temp->data!=x)
     {
        ptemp=temp;
        temp=temp->next;
     }
     if(temp==NULL)
    {
```

```
      printf("%d does not exit\n",x);
    }
    else
    {
/*allocating implicit memory to the node*/

        nn=(struct node *)malloc(sizeof(struct node));
        printf("enter data");
        scanf("%d",&nn->data);
       c=nn->data;
       nn->data;
      nn->prev=ptemp;
      nn->next=temp;
      ptemp->next=nn;
      temp->prev=nn;
      printf("%d succ inserted \n",c);
   }
}
/*end of insertion operation*/
/*deletion operation*/
void deletion()
{
    struct node *pt,*t;
    int x;
    t=pt=start;
    if(start==NULL)
    {
       printf("dll is empty\n");
    }
    printf("enter data to be deleted:");
   scanf("%d",&x);
   if(x==start->data)
   {
     t=start;
     t=t->next;
     free(start);
     start=t;
     start=pt;
   }
   else
  {
    while(t->next!=NULL&&t->data!=x)
    {
      pt=t;    /*logic for traversing*/
      t=t->next;
    }
   if(t->next==NULL&&t->data==x)
   {
     free(t);
     pt->next=NULL;
   }
   else
   {
     if(t->next==NULL&&t->data!=x)
         printf("data not found");
    else
     {
        pt->next=t->next;
```

```
        free(t);
      }
   }
   printf("%d is succ deleted\n",x);
  }
}
```
/*end of deletion operation*/

/*display operation*/
```
void display()
{
     struct node *temp;
     if(start==NULL)
     printf("stack is empty ");
     temp=start;
     while(temp->next!=NULL)
    {
        printf("%d",temp->data);
        temp=temp->next;
    }
    printf("%d",temp->data);
 }
```
/*end of display operation*/

/*main program*/
```
int main()
{
     int c,a;
    start=NULL;
    do
   {
        printf("1.insert\n2.delete\n3.display\n4.exit\nenter choice:");
        scanf("%d",&c);
        switch(c)
       {
           case 1:printf("1.insertbeg\n2.insertend\n3.insertmid\nenter choice:");
                      scanf("%d",&a);
                      switch(a)
                     {
                            case 1:insertbeg();
                                       break;
                            case 2:insertend();
                                       break;
                            case 3:insertmid();
                                      break;
                     }
                    break;
          case 2:deletion();
                     break;
         case 3:display();
                     break;
        case 4:printf("program ends\n");
                    break;
        default:printf("wrong choice\n");
                    break;
      }
  }
   while(c!=4);
   return 0;
 }
```

2.6 PRE LAB QUESTIONS

1. What is double linked list
2. How to represent a node in double linked list
3. Differentiate between single and double linked list

2.7 LAB ASSIGNMENT

1. Write a program to insert a node at first , last and at specified position of double linked list
2. Write a program to eliminate duplicates from double linked list
3. Write a program to delete a node from first, last and at specified position of double linked list

2.8 POST LAB QUESTIONS

1. How to represent double linked list
2. How will you traverse double linked list
3. List the advantages of double linked list over single list

2.9 INPUT AND OUTPUT

```
geetha@iare:~
[geetha@iare ~]$ clear
[geetha@iare ~]$ gcc w-2.c
[geetha@iare ~]$ ./a.out
1.insert
2.delete
3.display
4.exit
enter choice:1
1.insertbeg
2.insertend
3.insertmid
enter choice:1
enter data:30
30 succ inserted
1.insert
2.delete
3.display
4.exit
enter choice:1
1.insertbeg
2.insertend
3.insertmid
enter choice:1
enter data:20
```

```
geetha@iare:~
20 succ inserted
1.insert
2.delete
3.display
4.exit
enter choice:
3
20301.insert
2.delete
3.display
4.exit
enter choice:2
enter data to be deleted:30
30 is succ deleted
1.insert
2.delete
3.display
4.exit
enter choice:3
201.insert
2.delete
3.display
4.exit
enter choice:
```

EXPERIMENT 3

3.1 OBJECTIVE
To convert a given infix expression into its postfix Equivalent, Implement the stack using an array**.**

3.2 RESOURCE:
Turbo C

3.3 PROGRAM LOGIC
1. Create a stack
2. Read an infix expression
3. convert infix exression into postfix expression

3.4 PROCEDURE:
Go to debug -> run or press CTRL + F9 to run the program

3.5 SOURCE CODE:

Program to convert a given infix expression into its postfix Equivalent, Implement the stack using an array.

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#define MAX 20
char stack[MAX];
int top=1;
char pop(); /*declaration of pop function*/
void push(char item); /*declaration of push function*/
int prcd(char symbol) /*checking the precedence*/
{
  switch(symbol) /*assigning values for symbols*/
  {
    case '+':
    case '-': return 2;
    break;
    case '*':
    case '/': return 4;
    break;
    case '^':return 6;
    break;
    case '(':
    case ')':
    case '#':return 1;
   break;
 }
}
int(isoperator(char symbol)) /*assigning operators*/
{
  switch(symbol)
  {
     case '+':
     case '*':
```

```
    case '-':
    case '/':
    case '^':
    case '(':
    case ')':return 1;
    break;
    default:return 0;
  }
 }
/*converting infix to postfix*/
void convertip(char infix[],char postfix[])
{
 int i,symbol,j=0;
 stack[++top]='#';
 for(i=0;i<strlen(infix);i++)
 {
  symbol=infix[i];
  if(isoperator(symbol)==0)
  {
    postfix[j]=symbol;
     j++;
  }
  else
  {
    if(symbol=='(')
    push(symbol); /*function call for pushing elements into the stack*/
    else if(symbol==')')
    {
       while(stack[top]!='(')
      {
         postfix[j]=pop();
         j++;
      }
      pop();     /*function call for popping elements into the stack*/
   }
   else
   {
      if(prcd(symbol)>prcd(stack[top]))
      push(symbol);
      else
      {
         while(prcd(symbol)<=prcd(stack[top]))
         {
           postfix[j]=pop();
            j++;
         }
        push(symbol);
      }/*end of else loop*/
    }/*end of else loop*/
   } /*end of else loop*/
  }/*end of for loop*/
  While (stack[top]!='#')
  {
    postfix[j]=pop();
     j++;
  }
  postfix[j]='\0'; /*null terminate string*/
 }
```

/*main program*/
```
void main()
 {
     char infix[20],postfix[20];
     printf("enter the valid infix string \n");
     gets(infix);
     convertip(infix,postfix);    /*function call for converting infix to postfix */
     printf("the corresponding postfix string is:\n");
    puts(postfix);
  }
```
/*push operation*/
```
void push(char item)
{
    top++;
    stack[top]=item;
 }
```
/*pop operation*/
```
char pop()
{
    char a;
    a=stack[top];
    top--;
    return a;
 }
```

3.6 PRE LAB QUESTIONS

1. what is an expression
2. what are infix, prefix and postfix notations
3. what are polish and reverse polish notations
4. which data structure is used for infix to postfix conversion

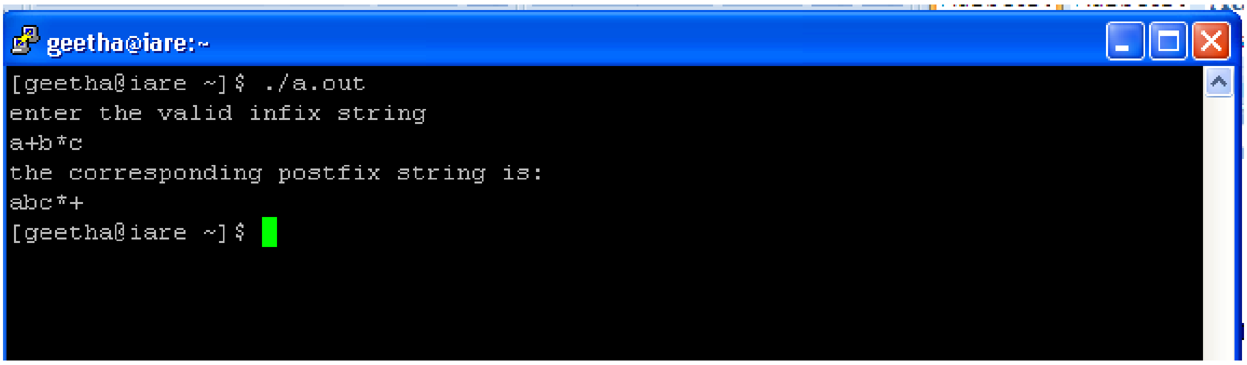
3.7 LAB ASSIGNMENT

1. Convert the infix expression (a+b)-(c*d) into post fix form?
2. Convert the following expression A + (B * C) - ((D * E + F) / G) into post form.

3.8 POST LAB QUESTIONS

1. how to represent stack
2. why reverse polish notation is required
3. can we evaluate polish notation

3.9 INPUT AND OUTPUT



EXPERIMENT 4

4.1 OBJECTIVE

1. To implement a double ended queue ADT using arrays.
2. To implement a double ended queue ADT using doubly linked list.

4.2 RESOURCE:

Turbo C

4.3 PROGRAM LOGIC

Double ended queue ADT using arrays

1. Create a linked list
2. Perform all the operation of double ended queue using arrays
3. Display the content of queue at last.

Double ended queue ADT using doubly linked list

1. Create a linked list
2. Perform all the operation of double ended queue using linked list
3. Display the content of queue at last.

4.4 PROCEDURE:

Go to debug -> run or press CTRL + F9 to run the program

4.5 SOURCE CODE:

Programs to implement a double ended queue ADT using arrays

```
#include<stdio.h>
#define SIZE 30
int dequeue[SIZE];
int front=-1,rear=-1; /* initializing front and rear*/
void insertrear(int);
void deletefront();
void insertfront(int);
void deleterear();
void traverse();
/*main program*/
void main()
{
       int choice,item;
       char ch;
       do
      {
           printf("\n options are");
           printf("\n press 1 to insert at rear");
           printf("\n press 2 to delete at front");
           printf("\n press 3 to insert at front");
           printf("\n press 4 to delete at rear");
           scanf("%d",&choice);
           switch(choice)    /*switch case*/
          {
                case 1: printf("\n enter the element:");
                        scanf("%d",&item);
                        insertrear(item); /*function call for inserting element at rear*/
                        break;
                case 2: deletefront(); /*function call for deleting element at front*/
```

```
                    break;
            case 3: printf("enter the element:");
                    scanf("%d",&item);
                    insertfront(item); /*function call for inserting element at front*/
                    break;
            case 4: deleterear();/*function call for deleting element at rear*/
                    break;
            case 5: traverse(); /*traversing the list*/
                    break;
            default : printf("wrong choice");
          } /*end of switch case*/
          printf("\n do you want to perform more operations?(Y/N):");
          fflush(stdin);
          scanf(" %c",&ch);
    } while(ch=='Y'||ch=='y');
  }

/*insertion at rear*/
void insertrear(int value) /*function definition*/
{
   if(rear==(SIZE-1))
   {
       printf("overflow");
        return;
   }
   else
   {
       if(front==-1)
       {
            printf("underflow so front will be modified");
           front=front+1;
       }
       rear=rear+1;
       dequeue[rear]=value;
   }
}

/*deletion at front*/
void deletefront() /*function definition*/
{
    int value;
    if(front==-1)
   {
        printf("queue is already empty");
        value=-1;
   }
   else
   {
      value=dequeue[front];
      if(front==rear)
      {
          printf("queue contains only one item");
          rear=-1;
          front=-1;
      }
      else
      front=front+1;
   }
```

```
  printf("removed element from front is %d",value);
}

/*insertion at front*/
void insertfront(int value) /*function definition*/
{
    if(front==0)
   {
      printf("front is at the beginning");
     printf("here insertion is not possible");
     return;
   }
   else
  {
    if(front==-1)
   {
      printf("queue is empty so both pointers will modified");
     front=front+1;
     rear=rear+1;
    }
    else
   {
     front=front-1;
   }
   dequeue[front]=value;
  }
}

/*deletion at rear*/
void deleterear() /*function definition*/
{
    int value;
   if(front==-1)
   {
          printf("queue is already empty");
          return;
   }
  else
 {
     value=dequeue[rear];
     if(rear==front)
     {
       printf("queue contains only one item");
       printf("rear and front will be modified");
       rear=-1;
       front=-1;
     }
    else
    {
      rear=rear-1;
     }
  }
  printf("\n the removed element from rear is:%d",value);
}
/*traverse operation*/
void traverse() /*function definition*/
{
     int i;
```

```
        if(front==-1)
        {
            printf("queue empty");
            return;
        }
        else
        {
            printf("\n value in the queue are as follow:");
            for(i=front;i<=rear;i++)
            printf("\n%d",dequeue[i]);
        }
    }
```

program to implement double ended queue adt using doubly linked list

```
#include <stdio.h>
 #include <stdlib.h>
/*declaring a structure to create a node*/

  struct node
{
      int data;
      struct node *prev, *next;
  };

  struct node *head = NULL, *tail = NULL;

  struct node * createNode(int data)
{
/*allocating implicit memory to the node*/

      struct node *newnode = (struct node *)malloc(sizeof (struct node));
      newnode->data = data;
      newnode->next = newnode->prev = NULL;
      return (newnode);
  }

   /* create sentinel(dummy head & tail) that helps us to do insertion and deletion
    operation at front and rear so easily. And these dummy head and tail wont get deleted
    till the end of execution of this program  */

  void createSentinels() /*creating a head and tail*/
 {
      head = createNode(0);
      tail = createNode(0);
      head->next = tail;
      tail->prev = head;
  }

  /* insertion at the front of the queue */
  void enqueueAtFront(int data)
 {
      struct node *newnode, *temp;
      newnode = createNode(data);
      temp = head->next;
      head->next = newnode;
      newnode->prev = head;
      newnode->next = temp;
      temp->prev = newnode;
```

```
}

/*insertion at the rear of the queue */
void enqueueAtRear(int data)
{
      struct node *newnode, *temp;
      newnode = createNode(data);
      temp = tail->prev;
      tail->prev = newnode;
      newnode->next = tail;
      newnode->prev = temp;
      temp->next = newnode;
}

/* deletion at the front of the queue */
void dequeueAtFront()
{
      struct node *temp;
      if (head->next == tail)
      {
            printf("Queue is empty\n");
      }
      Else
      {
            temp = head->next;
            head->next = temp->next;
            temp->next->prev = head;
            free(temp);
       }
       return;
}


/* deletion at the rear of the queue */

void dequeueAtRear()
{
      struct node *temp;
      if (tail->prev == head)
      {
            printf("Queue is empty\n");
      }
      Else
       {
            temp = tail->prev;
            tail->prev = temp->prev;
            temp->prev->next = tail;
            free(temp);
      }
      return;
}

/* display elements present in the queue */
void display()
{
      struct node *temp;
      if (head->next == tail)
       {
```

```
            printf("Queue is empty\n");
            return;
        }

        temp = head->next;
        while (temp != tail)
        {
            printf("%-3d", temp->data);
            temp = temp->next;
        }
        printf("\n");
  }

/*main program*/
  int main()
 {
        int data, ch;
        createSentinels();
        while (1)
        {
            printf("1. Enqueue at front\n2. Enqueue at rear\n");
            printf("3. Dequeue at front\n4. Dequeue at rear\n");
            printf("5. Display\n6. Exit\n");
            printf("Enter your choice:");
            scanf("%d", &ch);
            switch (ch) /*switch case*/
            {
                case 1:
                    printf("Enter the data to insert:");
                    scanf("%d", &data);
                    enqueueAtFront(data);
                    break;

                case 2:
                    printf("Enter ur data to insert:");
                    scanf("%d", &data);
                    enqueueAtRear(data);
                    break;

                case 3:
                    dequeueAtFront();
                    break;

                case 4:
                    dequeueAtRear();
                    break;

                case 5:
                    display();
                    break;

                case 6:
                    exit(0);

                default:
                    printf("Pls. enter correct option\n");
                    break;
            } /*end of switch case*/
```

```
    }
    return 0;
}
```

4.6 PRE LAB QUESTIONS

1. what is queue and its operations
2. what is double ended queue
3. differentiate queue and double ended queue

4.7 LAB ASSIGNMENT

1. Write a program to insert an element when rear is at last position
2. Write a program to delete an element when front is at last position

4.8 POST LAB QUESTIONS

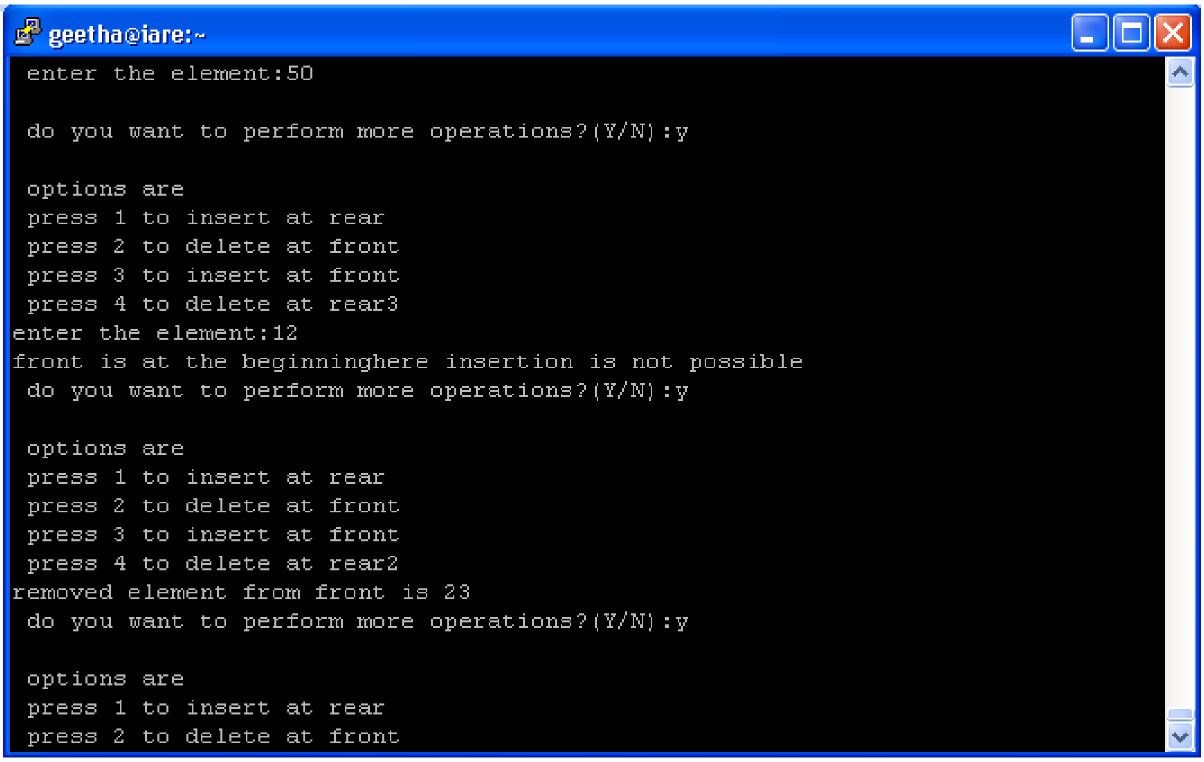
1. Write the condition for queue full
2. Write the condition for queue empty
3. List the advantages of double ended queue over queue

4.9 INPUT AND OUTPUT

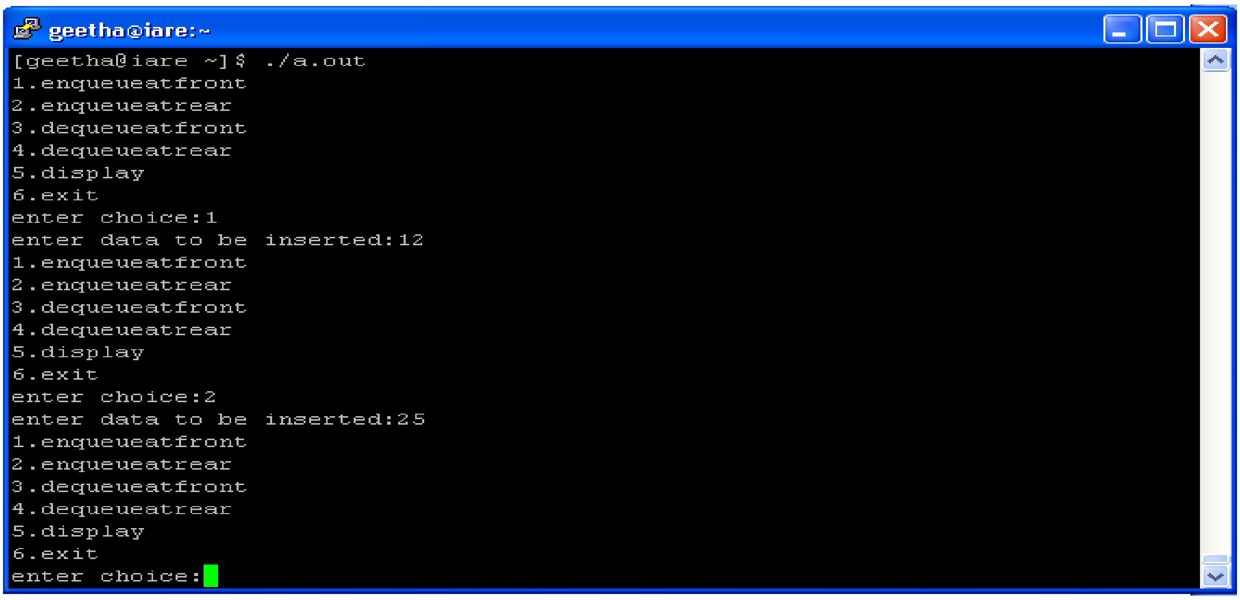
<u>A double ended queue ADT using arrays</u>

```
geetha@iare:~
 options are
 press 1 to insert at rear
 press 2 to delete at front
 press 3 to insert at front
 press 4 to delete at rear1

 enter the element:23
underflow so front will be modified
 do you want to perform more operations?(Y/N):y

 options are
 press 1 to insert at rear
 press 2 to delete at front
 press 3 to insert at front
 press 4 to delete at rear3
enter the element:24
front is at the beginninghere insertion is not possible
 do you want to perform more operations?(Y/N):y

 options are
 press 1 to insert at rear
 press 2 to delete at front
 press 3 to insert at front
 press 4 to delete at rear1
```

```
 press 3 to insert at front
 press 4 to delete at rear4
queue contains only one itemrear and front will be modified
 the removed element from rear is:50
 do you want to perform more operations?(Y/N):n
[geetha@iare ~]$
```



double ended queue adt using doubly linked list



geetha@iare:~

```
enter choice:1
enter data to be inserted:26
1.enqueueatfront
2.enqueueatrear
3.dequeueatfront
4.dequeueatrear
5.display
6.exit
enter choice:2
enter data to be inserted:65
1.enqueueatfront
2.enqueueatrear
3.dequeueatfront
4.dequeueatrear
5.display
6.exit
enter choice:3
1.enqueueatfront
2.enqueueatrear
3.dequeueatfront
4.dequeueatrear
5.display
6.exit
enter choice:
```

```
enter choice:4
1.enqueueatfront
2.enqueueatrear
3.dequeueatfront
4.dequeueatrear
5.display
6.exit
enter choice:5
12 25
1.enqueueatfront
2.enqueueatrear
3.dequeueatfront
4.dequeueatrear
5.display
6.exit
enter choice:6
[geetha@iare ~]$
```

EXPERIMENT 5

5.1 OBJECTIVE

1. To create a binary search tree of characters.
2. Traverse the above Binary search tree recursively in Post order.

5.2 RESOURCE:

Turbo C

5.3 PROGRAM LOGIC

1. Create binary tree with the property binary search tree
2. Visit the tree in post order
3. Visit in the order left, right, root
4. Display the visited nodes

5.4 PROCEDURE:

Go to debug -> run or press CTRL + F9 to run the program

5.5 SOURCE CODE:

/*program for creating and traversing the binary search tree*/

```
#include<stdio.h>
#include<stdlib.h>
typedef struct BST
{
     char d;
/*declaring a structure to create a node*/

    struct BST *lc,*rc;
}node;
/*main program*/
void main()
{
    int choice;
    char ans='N';
    int key;
     node *nn,*root,*parent;
     root=NULL;
     printf("\n program for binary search tree");
     do
    {
      printf("\n 1.create");
      printf("\n 2.resurcive traverse");
      printf("\n 3.exit");
      printf("\n enter your choice");
      scanf("%d",&choice);
      switch(choice) /*switch case*/
     {
       case 1:
                          do
                          {
                                  nn=(node *)malloc(sizeof(node));
                                  printf("\n enter the elements");
                                  nn->lc=NULL;
```

```
                nn->rc=NULL;
                scanf(" %c",&nn->d);
                if(root==NULL)
                root=nn;
               else
                   insert(root,nn);
                printf("\n want to enter more elements?(Y/N)");
                scanf(" %c",&ans);
         }while(ans=='y');
         break;
    case 2:
         if(root==NULL)
         printf("tree is not created");
         else
         {
                printf("\n the inorder display:");
                inorder(root);
              printf("\n the preorder display:");
              preorder(root);
             printf("\n the postorder display:");
             postorder(root);
         }
         break;
   } /*end of switch case*/
  }while(choice!=3);
 }
/*insertion operation*/
void insert(node *root,node *nn)
{
   int c,d;
   c=nn->d;
   d=root->d;
   if(c<d)
  {
    if(root->lc==NULL)
    root->lc=nn;
   else
     insert(root->lc,nn);
  }
}
/*inorder traversal*/
void inorder(node *temp)
{
   if(temp!=NULL)
   {
    inorder(temp->lc);
    printf(" %c",temp->d);
    inorder(temp->rc);
   }
}
/*preorder traversal*/
void preorder(node *temp)
{
  if(temp!=NULL)
  {
    printf(" %c",temp->d);
    preorder(temp->lc);
    preorder(temp->rc);
```

```
    }
  }
/*postorder traversal*/
void postorder(node *temp)
{
   if(temp!=NULL)
   {
     postorder(temp->lc);
     postorder(temp->rc);
     printf(" %c",temp->d);
   }
}
```

5.6 PRE LAB QUESTIONS

1. Differentiate between BST and complete BST
2. What are the properties of BST
3. How many nodes will be there in given nth level.

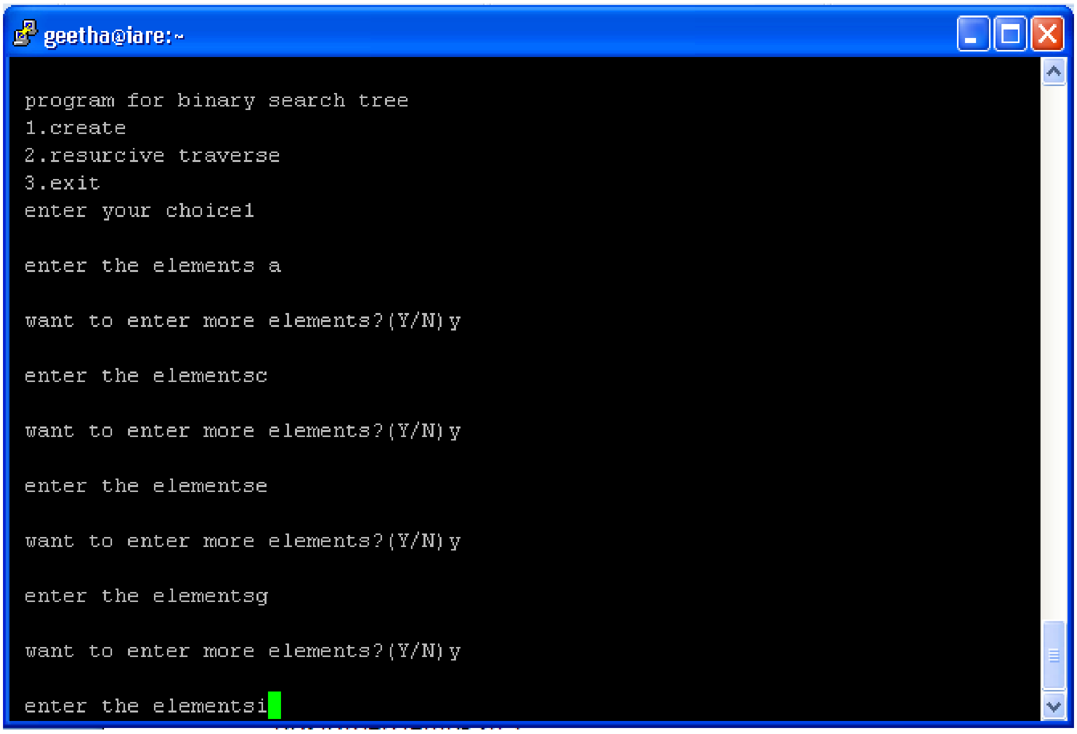
5.7 LAB ASSIGNMENT

1. Construct a binary search tree for the following 80, 40, 75, 30, 20, 90, 50

5.8 POST LAB QUESTIONS

1. List the various tree traversal techniques are there
2. Write the necessary condition for inserting element into BST
3. List the applications of BST

5.9 INPUT AND OUTPUT



EXPERIMENT 6

6.1 OBJECTIVE
1. Create a binary search tree of integers.
2. Traverse the above Binary search tree non recursively in inorder.

6.2 RESOURCE:
Turbo C

6.3 PROGRAM LOGIC
1. Read integers
2. Create binary tree with the property binary search tree
3. Visit the tree in inorder
4. Visit in the order left, root, right,
5. Display the visited nodes

6.4 PROCEDURE
Go to debug -> run or press CTRL + F9 to run the program

6.5 SOURCE CODE:

```
/*creating binary search tree of integer*/

#include<stdio.h>
#include<conio.h>
typedef struct bint
{
/*declaring a structure to create a node*/

   int data,flag;
   struct bint *left,*right;
}node;
node * create(node *r,int d)
{
    if(r == NULL)
   {
/*allocating implicit memory to the node*/

       r = (node *)malloc(sizeof(node));
       r->data = d;
       r->left = r->right = NULL;
   }
   else
  {
    if(r->data <= d)
        r->right = create(r->right,d);
    else
        r->left = create(r->left,d);
  }
  return r;
}
void non_in(node *r)
{
    int top=0;
    node *s[20],*pt=r;
    s[0]=NULL;
```

```
 while(pt != NULL)
{
  s[++top] = pt;
  pt = pt->left;
}
 pt = s[top--];
 while(pt != NULL)
 {
    printf("%d\t",pt->data);
    if(pt->right != NULL)
   {
      pt = pt->right;
      while(pt != NULL)
     {
        s[++top] = pt;
        pt = pt->left;
     }
 }
 pt = s[top--];
}
}
/*main program*/
void main()
{
  int d;
 char ch = 'Y';
  node *root = NULL;
 clrscr();
 while(toupper(ch) == 'Y')
{
  printf("\n Enter the item to insert");
  scanf("%d",&d);
  head = create(root,d);
  printf("\n Do you want to continue(y/n)");
  fflush(stdin);
  ch = getchar();
}

 printf("\n inorder non recursive\n");
 non_in(head);
}
```

6.6 PRE LAB QUESTIONS

1. Differentiate between BST and complete BST
2. What are the properties of BST
3. How many nodes will be there in given nth level.

6.7 LAB ASSIGNMENT

1 Construct a binary search tree for the following 80, 40, 75, 30, 20, 90, 50

6.8 POST LAB QUESTIONS

1. List the various tree traversal techniques are there
2. Write the necessary condition for inserting element into BST
3. List the applications of BST

6.9 INPUT AND OUTPUT

```
geetha@iare:~
[geetha@iare ~]$ ./a.out
enter the item to insert12

 do you want to continue (Y/N)y
enter the item to insert34

 do you want to continue (Y/N)y
enter the item to insert56

 do you want to continue (Y/N)y
enter the item to insert78

 do you want to continue (Y/N)n

 inorder non recursive
12      34      56      78      [geetha@iare ~]$
```

EXPERIMENT 7

7.1 OBJECTIVE

1. Arrange a list of integers in ascending order using Insertion sort
2. Arrange a list of integers in ascending order using Merge sort

7.2 RESOURCE

Turbo C

7.3 PROGRAM LOGIC

INSERTION SORT

1. Start with an empty left hand [sorted array] and the cards face down on the table [unsorted array].
2. Then remove one card [key] at a time from the table [unsorted array], and insert it into the correct position in the left hand [sorted array].
3. To find the correct position for the card, we compare it with each of the cards already in the hand, from right to left.

MERGE SORT

1. **Divide Step**
   If a given array *A* has zero or one element, simply return; it is already sorted. Otherwise, split *A*[*p* .. *r*] into two subarrays *A*[*p* .. *q*] and *A*[*q* + 1 .. *r*], each containing about half of the elements of *A*[*p* .. *r*]. That is, *q* is the halfway point of *A*[*p* .. *r*].
2. **Conquer Step**
   Conquer by recursively sorting the two subarrays *A*[*p* .. *q*] and *A*[*q* + 1 .. *r*].
3. **Combine Step**
   Combine the elements back in *A*[*p* .. *r*] by merging the two sorted subarrays *A*[*p* .. *q*] and *A*[*q* + 1 .. *r*] into a sorted sequence. To accomplish this step, we will define a procedure MERGE (*A*, *p*, *q*, *r*).

7.4 PROCEDURE:

Go to debug -> run or press CTRL + F9 to run the program

7.5 SOURCE CODE:

Insertion sort

```
#include<stdio.h>
void inst_sort(int[]);
void main() {
        int num[5],count;
        printf("\n enter the five elements to sort:\n");
        for(count=0;count<5;count++)
        scanf("%d",&num[count]);
        inst_sort(num); /*function call for insertion sort*/
        printf("\n\n elements after sorting:\n");
        for(count=0;count<5;count++)
        printf("%d\n",num[count]);
}
void inst_sort(int num[]) { /* function definition for insertion sort*/
      int i,j,k;
      for(j=1;j<5;j++)      {
        k=num[j];
        for(i=j-1;i>=0&&k<num[i];i--)
```

```
num[i+1]=num[i];
num[i+1]=k;
}}
```

Merge sort

```
#include<stdio.h>
void mergesort(int[],int,int);
void mergearray(int[],int,int,int);
main() {
      int a[50],n,i;
     printf("\n enter size of an array:");
     scanf("%d",&n);
      printf("\n enter elements of an array:\n");
      for(i=0;i<n;i++)
        scanf("%d",&a[i]);
      mergesort(a,0,n-1);
      printf("\n\nafter sorting:\n");
      for(i=0;i<n;i++)
      printf("\n%d",a[i]);
}
/*merge operation*/
void mergesort(int a[],int beg,int end) {
    int mid;
    if(beg<end)     {
      mid=(beg+end)/2;
      mergesort(a,beg,mid);
      mergesort(a,mid+1,end);
      mergearray(a,beg,mid,end);
  } }
void mergearray(int a[],int beg,int mid,int end) {
     int i,leftend,num,temp,j,k,b[50];
     for(i=beg;i<=end;i++)
     b[i]=a[i];
     i=beg;
     j=mid+1;
     k=beg;
     while((i<=mid)&&(j<=end))      {
    if(b[i]<+b[j])
    {
       a[k]=b[i];
       i++;
       k++;
    }
   else     {
       a[k]=b[j];
       j++;
       k++;
    } }

 if(i<=mid) {
     while(i<=mid)      {
        a[k]=b[i];
        i++;
        k++;
    } }
  else {
     while(j<=end)  {
```

```
        a[k]=b[j];
        j++;
        k++;
      } } }
```

7.6 PRE LAB QUESTIONS

1. what is sorting
2. List the various sorting algorithms
3. What is the advantage of insertion sort
4. How merge sort works
5. Is insertion sort is stable algorithm

7.7 LAB ASSIGNMENT

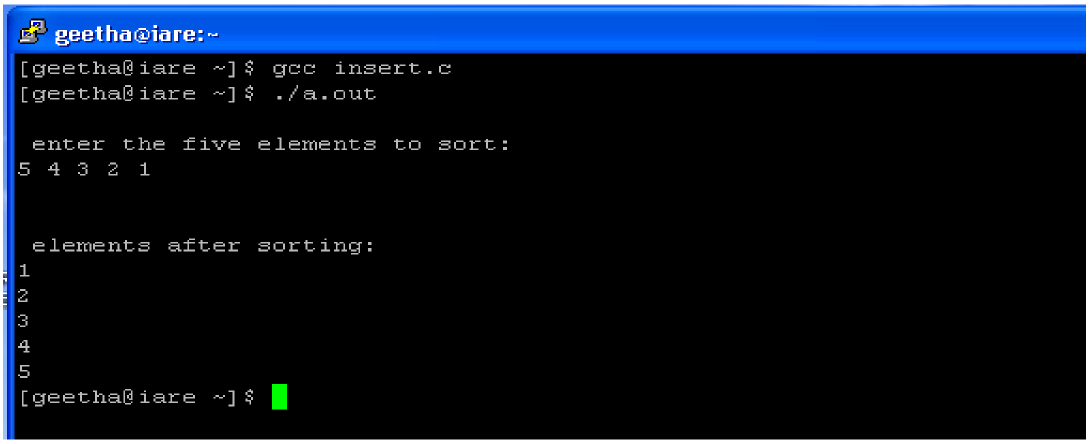
1. Apply the selection sort on the following elements21,11,5,78,49, 54,72,88
2. Apply the merge sort on the following elements21,11,5,78,49, 54,72,88 and 56,28,10

7.8 POST LAB QUESTIONS

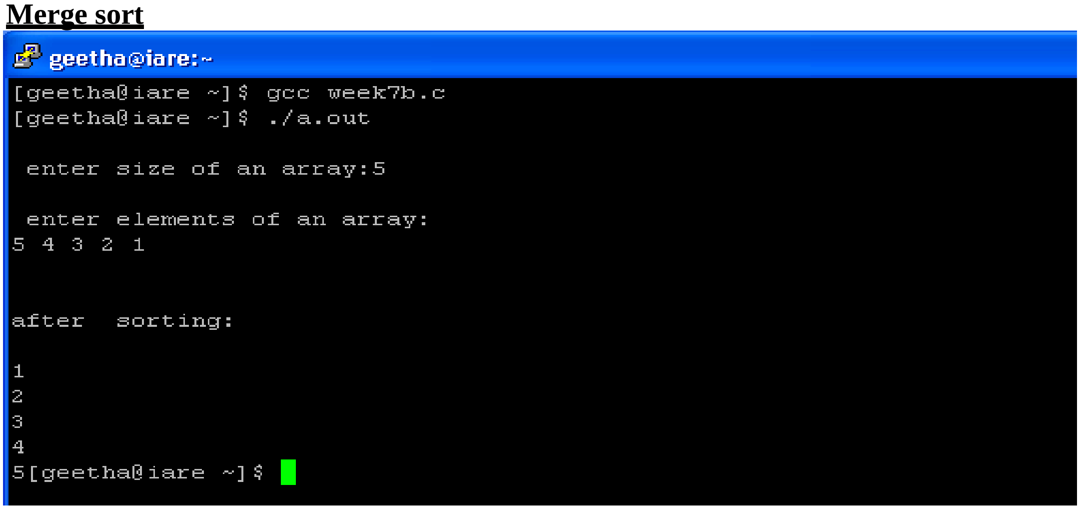
1. What is the time complexity of insertion sort
2. What is the time complexity of merge sort
3. Why sorting is required
4. Is merge sort is in place
5. List the application of merge sort

7.9 INPUT AND OUTPUT

<u>Insertion sort</u>



<u>Merge sort</u>



EXPERIMENT 8

8.1. OBJECTIVE

1. Sorting the list of integers in ascending order using Quick sort
2. Sorting the list of integers in ascending order using Selection sort

8.2 **RESOURCE:**

Turbo C

8.3 PROGRAM LOGIC

Quick sort

1. Read the elements to be sort
2. Find the proper pivot element
3. Apply quick sort method to sort the remaining elements

Selection sort

1. Read the elements to be sort
2. Select the minimum element
3. Apply the selection sort to sort the remaining elements

8.4 PROCEDURE:

Go to debug -> run or press CTRL + F9 to run the program

8.5 SOURCE CODE:

QUICK SORT

```
#include<stdio.h>
main()
{
     int x[10],i,n;
    printf("enter number of elements:");
    scanf("%d",&n);
    printf("enter %d elements:\n");
    for(i=0;i<n;i++)
    scanf("%d",&x[i]);
    quicksort(x,0,n-1);/*function call*/
    printf("sorted elements are:");
    for(i=0;i<n;i++)
      printf("%3d",x[i]);
}
/*called function*/
quicksort(int x[10],int first,int last)
{
     int pivot,i,j,t;
     if(first<last)
    {
       pivot=first;
       i=first;
       j=last;
       while(i<j)
      {
         while(x[i]<=x[pivot]&&i<last)
         i++;
        while(x[j]>x[pivot])
         j--;
       if(i<j)
```

```
            {
              t=x[i];
              x[i]=x[j];
              x[j]=t;
            }
          }
          t=x[pivot];
          x[pivot]=x[j];
          x[j]=t;
          quicksort(x,first,j-1);
          quicksort(x,j+1,last);
        }
      }
```

SELECTION SORT

```
#include<stdio.h>
void sel_sort(int[]);
void main()
{
     int num[5],count;
    printf("enter the five elements to sort:\n");
    for(count=0;count<5;count++)
       scanf("%d",&num[count]);
    sel_sort(num); /*function call*/
    printf("\n\n elements after sorting:\n");
    for(count=0;count<5;count++)
       printf("%d\n",num[count]);
}
/*called function*/
void sel_sort(int num[])
{
    int i,j,min,temp;
    for(j=0;j<5;j++)
   {
      min=j;
      for(i=j;i<5;i++)
      if(num[min]>num[i])
      min=i;
     if(min<5)
    {
         temp=num[j];
         num[j]=num[min];
         num[min]=temp;
    }
     printf("%d\t",num[j]);
  }
}
```

8.6 PRE LAB QUESTIONS

1. what is sorting
2. List the various sorting algorithms
3. What is the advantage of selection sort
4. How to find pivot element in quick sort
5. Is quick sort is stable algorithm

8.7 LAB ASSIGNMENT

1 . Rearrange the following numbers using Quick sort procedure. 42, 12, 18, 98, 67, 83, 8, 10, 71

2.Apply the selection sort on the following elements21,11,5,78,49, 54,72,88

8.8 POST LAB QUESTIONS

6. What is the time complexity of selection sort
7. What is the time complexity of quick sort
8. Why sorting is required
9. Is selection sort is stable
10. What is the worst case for quick sort

8. 9 INPUT AND OUTPUT

QUICK SORT

```
geetha@iare:~
[geetha@iare ~]$ gcc quick.c
[geetha@iare ~]$ ./a.out
enter number of elements:5
enter 1 elements:
5 4 3 2 1
sorted elements are:  1  2  3  4  5[geetha@iare ~]$
```

SELECTION SORT

```
geetha@iare:~
[geetha@iare ~]$ gcc sel.c
[geetha@iare ~]$ ./a.out
enter the five elements to sort:
5
4
3
2
1
1       2       3       4       5

 elements after sorting:
1
2
3
4
5
[geetha@iare ~]$
```

EXPERIMENT 9

9.1 OBJECTIVE

1. To perform the Insertion into a B-tree
2. Implement Heap sort algorithm

9.2 RESOURCE:

Turbo C

9.3 PROGRAM LOGIC

B-tree

1. Read the elements
2. Use the properties B-Tree and construct B-Tree
3. Select the required operation

Heap sort

4. Read the elements to be sort
5. Construct the heap
6. Sort the above heap using deletion

9.4 PROCEDURE:

Go to debug -> run or press CTRL + F9 to run the program

9.5 SOURCE CODE:

B-TREE

```
#include<stdio.h>
#include <stdlib.h>
#define M 5
struct node
{
int n; /* n < M No. of keys in node will always less than order of B
                    tree */
int keys[M-1];      /*array of keys*/
struct node *p[M]; /* (n+1 pointers will be in use) */
}*root=NULL;

enum KeyStatus { Duplicate,SearchFailure,Success,InsertIt,LessKeys };
void insert(int key);
void display(struct node *root,int);
void search(int x);
enum KeyStatus ins(struct node *r, int x, int* y, struct node** u);
int searchPos(int x,int *key_arr, int n);
int main()
{
int key;
int choice;
printf("Creation of B tree for node %d\n",M);
while(1)
{
printf("1.Insert\n");
printf("3.Search\n");
printf("4.Display\n");
printf("5.Quit\n");
printf("Enter your choice : ");
scanf("%d",&choice);
```

```
switch(choice)
{
	case 1:
		printf("Enter the key : ");
		scanf("%d",&key);
		insert(key);
		break;


	case 3:
		printf("Enter the key : ");
		scanf("%d",&key);
		search(key);
		break;
	case 4:
		printf("Btree is :\n");
		display(root,0);
		break;
	case 5:
		exit(1);
	default:
		printf("Wrong choice\n");
		break;
	}/*End of switch*/
	}/*End of while*/
return 0;
}/*End of main()*/
void insert(int key)
{
struct node *newnode;
int upKey;
enum KeyStatus value;
value = ins(root, key, &upKey, &newnode);
if (value == Duplicate)
printf("Key already available\n");
if (value == InsertIt)
{
struct node *uproot = root;
root=malloc(sizeof(struct node));
root->n = 1;
root->keys[0] = upKey;
root->p[0] = uproot;
root->p[1] = newnode;
}/*End of if */
}/*End of insert()*/

enum KeyStatus ins(struct node *ptr, int key, int *upKey,struct node**newnode)
{
struct node *newPtr, *lastPtr;
int pos, i, n,splitPos;
int newKey, lastKey;
enum KeyStatus value;
if (ptr == NULL)
{
*newnode = NULL;
*upKey = key;
return InsertIt;
```

```
}
n = ptr->n;
pos = searchPos(key, ptr->keys, n);
if (pos < n && key == ptr->keys[pos])
return Duplicate;
value = ins(ptr->p[pos], key, &newKey, &newPtr);
if (value != InsertIt)
return value;
/*If keys in node is less than M-1 where M is order of B tree*/
if (n < M - 1)
{
pos = searchPos(newKey, ptr->keys, n);
/*Shifting the key and pointer right for inserting the new key*/
for (i=n; i>pos; i--)
{
ptr->keys[i] = ptr->keys[i-1];
ptr->p[i+1] = ptr->p[i];
}
/*Key is inserted at exact location*/
ptr->keys[pos] = newKey;
ptr->p[pos+1] = newPtr;
++ptr->n; /*incrementing the number of keys in node*/
        return Success;
}/*End of if */
/*If keys in nodes are maximum and position of node to be inserted is
last*/
if (pos == M - 1)
{
lastKey = newKey;
lastPtr = newPtr;
}
else /*If keys in node are maximum and position of node to be inserted
is not last*/
{
lastKey = ptr->keys[M-2];
lastPtr = ptr->p[M-1];
for (i=M-2; i>pos; i--)
{
ptr->keys[i] = ptr->keys[i-1];
ptr->p[i+1] = ptr->p[i];
}
ptr->keys[pos] = newKey;
ptr->p[pos+1] = newPtr;
}
splitPos = (M - 1)/2;
(*upKey) = ptr->keys[splitPos];

(*newnode)=malloc(sizeof(struct node));/*Right node after split*/
ptr->n = splitPos; /*No. of keys for left splitted node*/
(*newnode)->n = M-1-splitPos;/*No. of keys for right splitted node*/
for (i=0; i < (*newnode)->n; i++)
{
(*newnode)->p[i] = ptr->p[i + splitPos + 1];
if(i < (*newnode)->n - 1)
(*newnode)->keys[i] = ptr->keys[i + splitPos + 1];
else
(*newnode)->keys[i] = lastKey;
}
```

```
(*newnode)->p[(*newnode)->n] = lastPtr;
return InsertIt;
}/*End of ins()*/

void display(struct node *ptr, int blanks)
{
if (ptr)
{
int i;
for(i=1;i<=blanks;i++)
printf(" ");
for (i=0; i < ptr->n; i++)
printf("%d ",ptr->keys[i]);
printf("\n");
for (i=0; i <= ptr->n; i++)
display(ptr->p[i], blanks+10);
}/*End of if*/
}/*End of display()*/

void search(int key)
{
int pos, i, n;
struct node *ptr = root;
printf("Search path:\n");
while (ptr)
{
n = ptr->n;
for (i=0; i < ptr->n; i++)
printf(" %d",ptr->keys[i]);
printf("\n");
pos = searchPos(key, ptr->keys, n);
if (pos < n && key == ptr->keys[pos])
{
printf("Key %d found in position %d of last dispalyed node\n",key,i);
return;
}
ptr = ptr->p[pos];
}
printf("Key %d is not available\n",key);
}/*End of search()*/

int searchPos(int key, int *key_arr, int n)
{
int pos=0;
while (pos < n && key > key_arr[pos])
pos++;
return pos;
}/*End of searchPos()*/
```

HEAP SORT

```
#include<stdio.h>
int p(int);
int left(int);
int right(int);
void heapify(int[],int,int);
void buildheap(int[],int);
```

```
void heapsort(int[],int);
void main()
  {
      int x[20],n,i;
      printf("enter the no. of elements to b sorted");
      scanf("%d",&n);
      printf("enter the elements ");
      for(i=0;i<n;i++)
      scanf("%d",&x[i]);
      heapsort(x,n);
      printf("sorted array is");
      for(i=0;i<n;i++)
      printf("%d",x[i]);
   }
   int p(int i)
  {
      return i/2;
  }
  int left(int i)
 {
    return 2*i+1;
  }
  int right(int i)
  {
     return 2*i+2;
  }
  void heapify(int a[],int i,int n)
  {
      int l,r,large,t;
      l=left(i);
      r=right(i);
        if((l<=n-1)&&(a[l]>a[i]))
          large=l;
       else
          large=i;
      if((r<=n-1)&&(a[r]>a[large]))
         large=r;
      if(large!=i)
     {
        t=a[i];
       a[i]=a[large];
       a[large]=t;
       heapify(a,large,n);
     }
 }
void buildheap(int a[],int n)
{
     int i;
    for(i=(n-1)/2;i>=0;i--)
      heapify(a,i,n);
}
void heapsort(int a[],int n)
{
    int i,m,t;
    buildheap(a,n);
    m=n ;
    for(i=n-1;i>=1;i--)
   {
```

```
 t=a[0];
a[0]=a[i];
a[i]=t;
m=m-1;
 heapify(a,0,m);
  }
}
```

9.6 PRE LAB QUESTIONS

1. what is B –tree
2. What is heap
3. What are the properties of B-tree
4. What is sorting

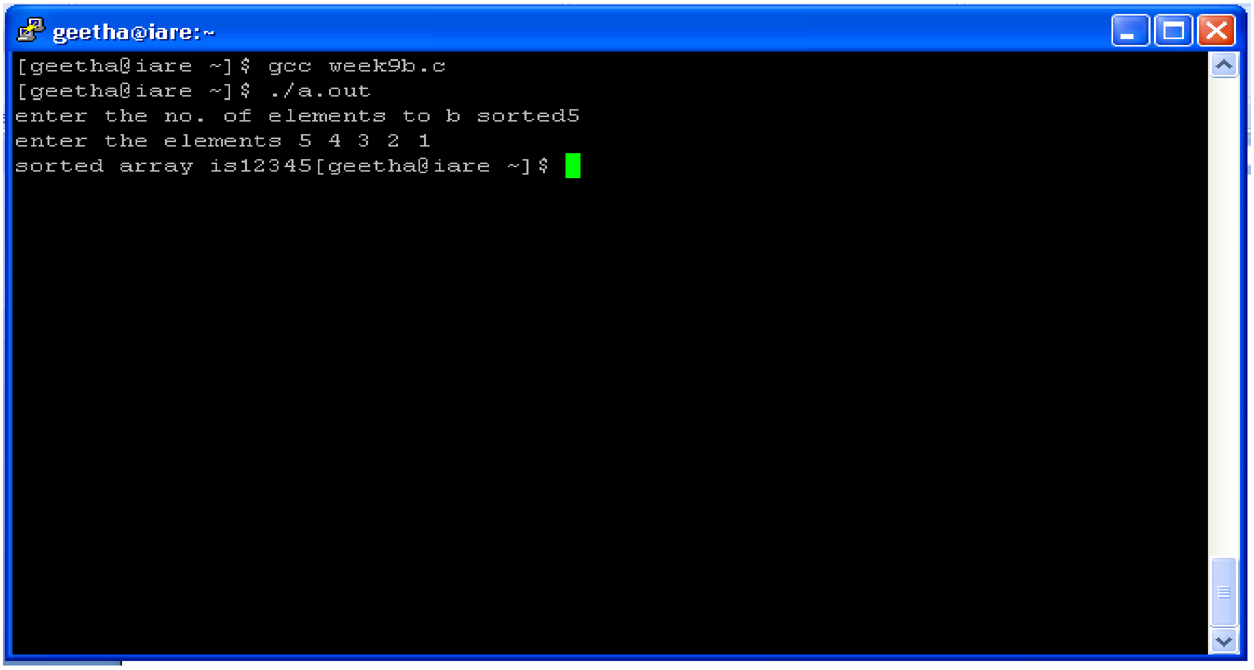
9.7 LAB ASSIGNMENT

1. Apply heap sort on list of elements 14,12,9,8,7,10,18,20,30
2. Construct a B-tree of order 3 with the following elements 10,20,15,3,2,16,21,25,30,40

9.8 POST LAB QUESTIONS

1. What is time complexity of heap sort
2. Write the condition to insert and delete an element into B-tree
3. Is heap sort is stable or not

9. 9 INPUT AND OUTPUT



EXPERIMENT 10

10.1 OBJECTIVE

To implement all the functions of a dictionary (ADT) using hashing

10.2 RESOURCE:

Turbo C

10.3 PROGRAM LOGIC

1. Read the key elements from the dictionary
2. Use the hash function to implement dictionary
3. Apply required operation on the dictionary

10.4 PROCEDURE:

Go to debug -> run or press CTRL + F9 to run the program

10.5 SOURCE CODE:

DICTIONARY USING HASHING

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int b;

int hsearch(int key,int d,int *ht,int *empty)
{
        int i=key%(d);
        int j=i;
        int c=0;
        do
        {
            if(empty[j]||(*(ht+j)==key))
            return j;
            c++;
            j=(i+c)%(d);
        }while(j!=i);
        return 0;
}
int search(int key,int d,int *ht,int *empty)
{
      b=hsearch(key,d,ht,empty);
       printf("%d",b);
       if(empty[b]==1)
         return -1;
      else if(b==0)
         return 1;
      else
       return b;
}
/*insertion operation*/
void insert(int key,int d,int *ht,int *empty)
{
      b=hsearch(key,d,ht,empty);
     if(empty[b])
     {
```

```
            empty[b]=0;
             *(ht+b)=key;
        printf("elements is inserted\n");
    }
}
```

/*deletion operation*/

```
void delete(int key,int d,int *ht,int *empty)
{
        int b=hsearch(key,d,ht,empty);
       *(ht+b)=0;
         empty[b]=1;
        printf("element is deleted\n");
}
void display(int d,int *ht,int *empty)
{
     int i;
     printf("hash table elements are\n");
     for(i=0;i<d;i++)
    {
       if(empty[i])
      printf(" 0");
     else
       printf("%5d",*(ht+i));
    }
    printf("\n");
 }
```

/*main program*/

```
void main()
{
       int choice=1;
       int key;
       int d,i,s;
       int *empty,*ht;
       printf("enter the hash table size:");
       scanf("%d",&d);
       ht=(int *)malloc(d *sizeof(int));
     empty=(int *)malloc(d *sizeof(int));
     for(i=0;i<d;i++)
     empty[i]=1;
     while(1)
    {
            printf("\n");
             printf("\n LINEAR PROBING");
             printf("\n 1:insert hash table:");
             printf("\n 2:delete hash table");
             printf("\n 3:search hash table");
             printf("\n 4:display hash table");
             printf("\n 5:exit");
             printf("enter your choice");
             scanf("%d",&choice);
             switch(choice)
             {
                  case 1:printf("enter the elemants:");
                            scanf("%d",&key);
                            insert(key,d,ht,empty);
                            break;
                  case 2:printf("enter to remove from hash table:");
                            scanf("%d",&key);
```

```
                delete(key,d,ht,empty);
                break;
        case 3:printf("enter the search elements:");
               scanf("%d",&key);
               s=search(key,d,ht,empty);
                 if(s==-1||s==0)
                  printf("not found\n");
             else
                 printf("element found at index %d",hsearch(key,d,ht,empty));
             break;
        case 4:display(d,ht,empty);
                  break;
        case 5:exit(0);
    }
  }return;
}
```

10.6 PRE LAB QUESTIONS

1. what is dictionary
2. What is hashing
3. List types of hash functions
4. Define linear probing
5. What is quadratic probing

10.7 LAB ASSIGNMENT

1. Use quadratic probing to fill the Hash table of size 11. Data elements are 23,0,52,61,78,33,100,8,90,10,14.
2. Analyze input (371, 323, 173, 199, 344, 679, 989) and hash function h(x)=x mod 10, Show the result Separate Chaining, linear probing

10.8 POST LAB QUESTIONS

1. List the application of hashing
2. Explain the condition to insert element into hash table
3. What is double hashing

10.9 INPUT AND OUTPUT

```
geetha@iare:~
[geetha@iare ~]$ gcc week9.c
[geetha@iare ~]$ ./a.out
enter the hash table size:3


 LINEAR PROBING
 1:insert hash table:
 2:delete hash table
 3:search hash table
 4:display hash table
 5:exitenter your choice1
enter the elemants:12
elements is inserted


 LINEAR PROBING
 1:insert hash table:
 2:delete hash table
 3:search hash table
 4:display hash table
 5:exitenter your choice1
enter the elemants:22
elements is inserted
```

```
geetha@iare:~
 2:delete hash table
 3:search hash table
 4:display hash table
 5:exitenter your choice1
enter the elemants:22
elements is inserted


 LINEAR PROBING
 1:insert hash table:
 2:delete hash table
 3:search hash table
 4:display hash table
 5:exitenter your choice1
enter the elemants:31
elements is inserted


 LINEAR PROBING
 1:insert hash table:
 2:delete hash table
 3:search hash table
 4:display hash table
 5:exitenter your choice
```

```
geetha@iare:~
 LINEAR PROBING
 1:insert hash table:
 2:delete hash table
 3:search hash table
 4:display hash table
 5:exitenter your choice4
hash table elements are
   12   22   31


 LINEAR PROBING
 1:insert hash table:
 2:delete hash table
 3:search hash table
 4:display hash table
 5:exitenter your choice2
enter to remove from hash table:22
element is deleted


 LINEAR PROBING
 1:insert hash table:
 2:delete hash table
 3:search hash table
```

```
geetha@iare:~
 2:delete hash table
 3:search hash table
 4:display hash table
 5:exitenter your choice3
enter the search elements:22
1not found


 LINEAR PROBING
 1:insert hash table:
 2:delete hash table
 3:search hash table
 4:display hash table
 5:exitenter your choice4
hash table elements are
   12  0   31


 LINEAR PROBING
 1:insert hash table:
 2:delete hash table
 3:search hash table
 4:display hash table
 5:exitenter your choice
```

EXPERIMENT 11

11.1 OBJECTIVE
To implementing Knuth-Morris- Pratt pattern matching algorithm

11.2 RESOURCE:
Turbo C

11.3 PROGRAM LOGIC
1. Read the pattern and text
2. Compute failure function
3. Apply the KMP algorithm

11.4 PROCEDURE
Go to debug -> run or press CTRL + F9 to run the program

11.5 SOURCE CODE
Knuth-Morris- Pratt pattern matching algorithm

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

void computefailure(char *pat, int M, int *f);

void KMPSearch(char *pat, char *txt) //to implement kmp search
{
    int M = strlen(pat);//length of pattern
    int N = strlen(txt);//length of text

   int *f = (int *)malloc(sizeof(int)*M); //dynamic allocation
   int j = 0; // index for pat[]

   computefailure(pat, M, f);

    int i = 0; // index for txt[]
   while(i < N)
  {
     if(pat[j] == txt[i])
    {
      j++;
      i++;
   }
     if (j == M)
   {
     printf("Found pattern at index %d \n", i-j);
     j = f[j-1];
  }
       else if(pat[j] != txt[i])
 {
          if(j != 0)
    j = f[j-1];
   else
    i = i+1;
 }
```

```
    }
    free(f); }

void computefailure (char *pat, int M, int *f) //compute the failure function
{
        int len = 0;
        int i;

        f[0] = 0; // f[0] is always 0
        i = 1;

        while(i < M)
        {
                if(pat[i] == pat[len])
                {
                  len++;
                  f[i] = len;
                  i++;
                }
                else // (pat[i] != pat[len])
                {
                  if( len != 0 )
                  {
                        len = f[len-1];
                  }
                  else // if (len == 0)
                  {
                    f[i] = 0;
                    i++;
                  }
                }
        }
     }

int main()
{
    int match;
      printf("\nEnter the Text: ");
      gets(text); //reading the text
      printf("\nEnter the Pattern: ");
      gets(pattern); //reading the pattern
      m=strlen(text);
      n=strlen(pattern);
      match=KMPSearch(pat, txt);

      if(match>=0)
      {
              printf("\nMatch found at position %d\n\n",match);
      }
      else
      {
         printf("\nNo Match found!!!\n\n");
      }
      return 0;
}
```

11.6 PRE LAB QUESTIONS

1. what is pattern matching
2. List the various pattern matching algorithms
3. What is failure function

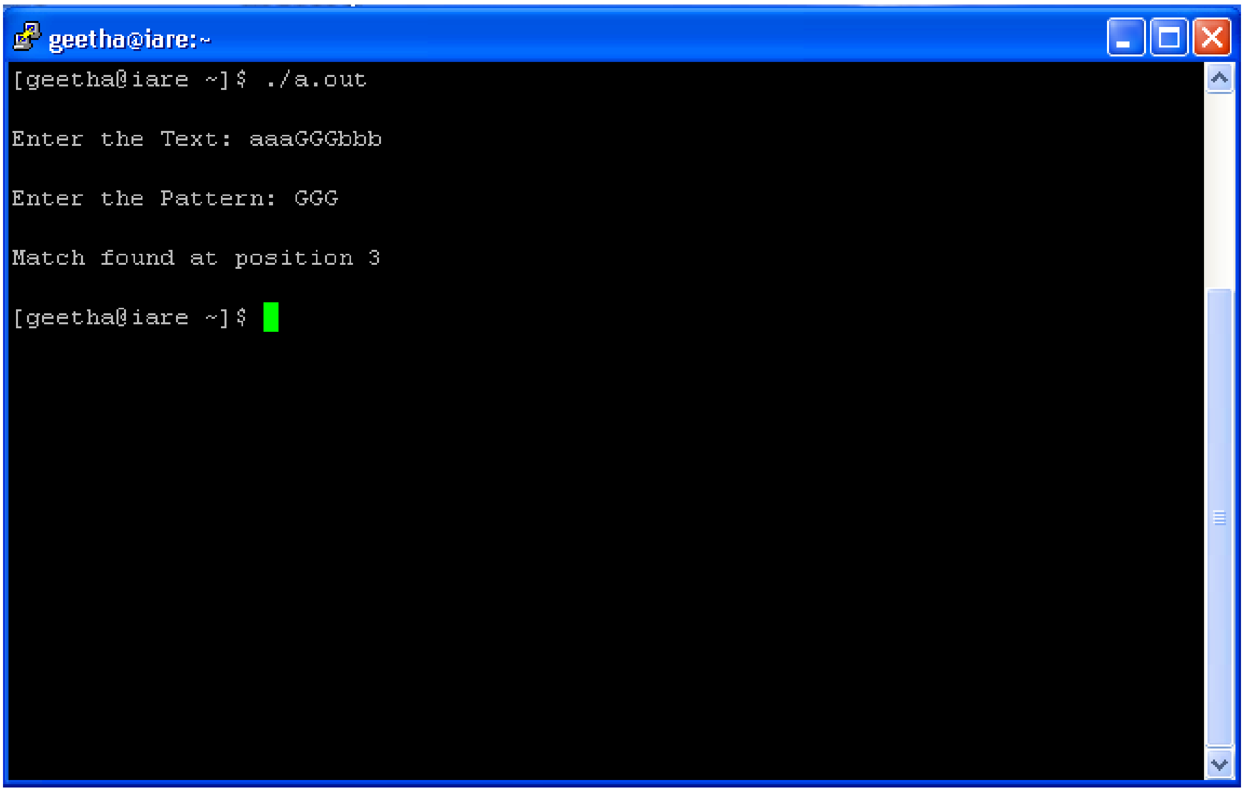
11.7 LAB ASSIGNMENT

1. Apply KMP algorithm on pattern “abacab” and text “abacaabaccabacabaabb”
2. Apply KMP algorithm on pattern “abaa” and text “abbbaababaab”

11.8 POST LAB QUESTIONS

1. Which pattern matching algorithm is better
2. Write the significance of Knuth-Morris- Pratt pattern matching algorithm
3. List the applications of pattern matching algorithms

11. 9 INPUT AND OUTPUT



EXPERIMENT 12

A: Write C programs for implementing the following graph traversal algorithm for Depth first traversal

12.1 OBJECTIVE

To traverse graph

12.2 RESOURCE:

Turbo C

12.3 PROGRAM LOGIC

1. Take the graph as a input
2. Start at some vertex and traverse it using DFS
3. Apply the above procedure for all nodes

12.4 PROCEDURE:

Go to debug -> run or press CTRL + F9 to run the program

12.5 SOURCE CODE:

DFS

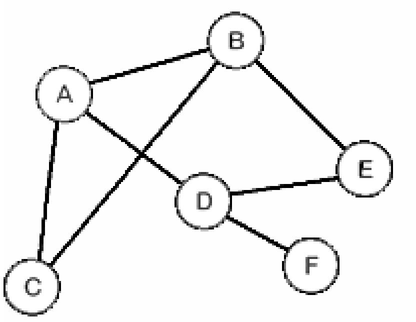
```
#include <stdio.h>
void dfs(int);
int g[10][10],visited[10],n,vertex[10];
void main()
{
    int i,j;
    printf("enter number of vertices:");
    scanf("%d",&n);
    printf("enter the val of vertex:");
    for(i=0;i<n;i++)
            scanf("%d",&vertex[i]);
   printf("\n enter adjecency matrix of the graph:");
   for(i=0;i<n;i++)
             for(j=0;j<n;j++)
            scanf("%d",&g[i][j]);
  for(i=0;i<n;i++)
   visited[i]=0;
   dfs(0);
}
void dfs(int i)
{
   int j;
    printf("%d",vertex[i]);
 visited[i]=1;       for(j=0;j<n;j++)
 if(!visited[j]&&g[i][j]==1)
   dfs(j);
}
```

12.6 PRE LAB QUESTIONS

1. What is graph
2. List various way of representations of graph
3. How many graph traversal algorithms are there

12.7 LAB ASSIGNMENT

1. Find DFS traversal of the following graph

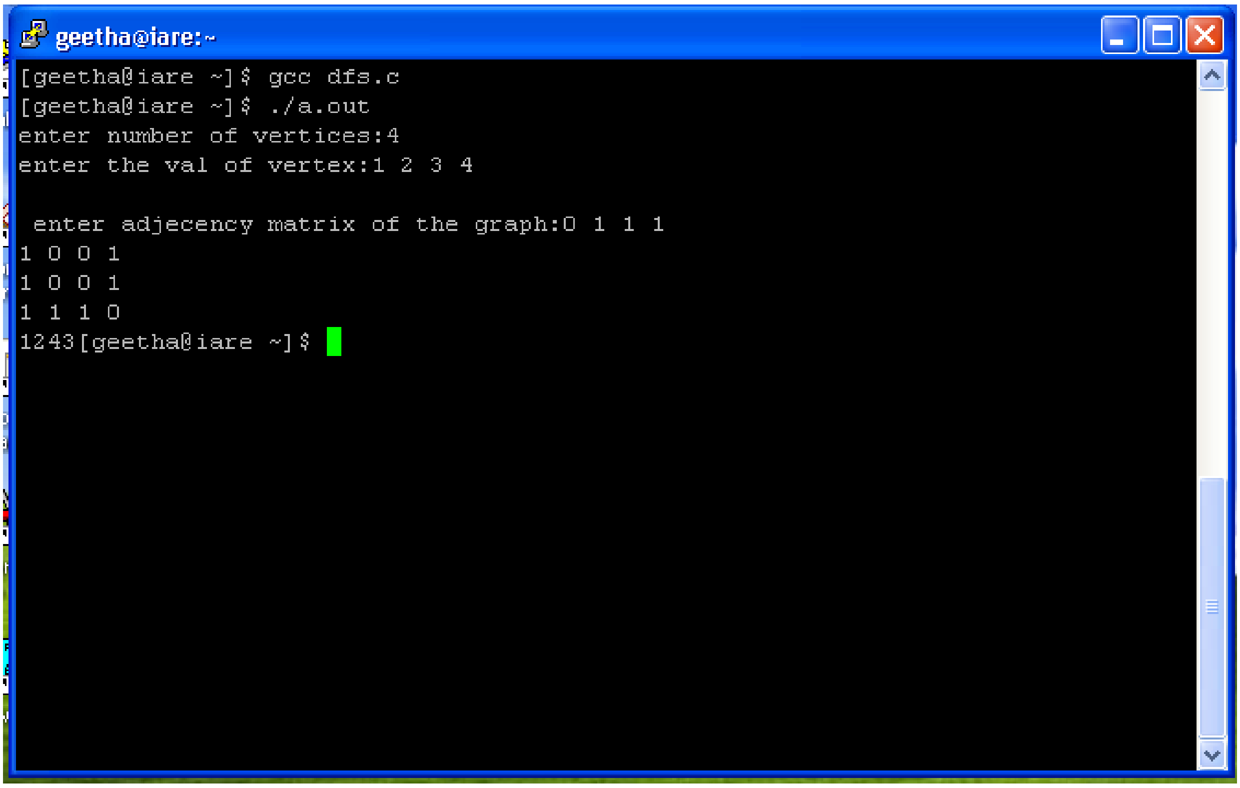


2. Deduce the time complexity of DFS algorithm

12.8 POST LAB QUESTIONS

1. Applications of graph traversals
2. Define minimum spanning tree
3. What is the time complexity of DFS

12. 9 INPUT AND OUTPUT



CONTENT BEYOND SYLLABI

EXPERIMENT 1

1.1 OBJECTIVE

*Write a C Program to check whether two given lists are containing the same data.

1.2 SOURCE CODE

```
#include  <stdio.h>
#include <stdlib.h>
struct node
{
    int num;
    struct node *next;
};

void feedmember(struct node **);
int compare (struct node *, struct node *);
void release(struct node **);

int main()  {
    struct node *p = NULL;
    struct node *q = NULL;
    int result;

    printf("Enter data into first list\n");
    feedmember(&p);
    printf("Enter data into second list\n");
    feedmember(&q);
    result = compare(p, q);
    if (result == 1)       {
        printf("The 2 list are equal.\n");
    }
    else     {
        printf("The 2 lists are unequal.\n");
    }
    release (&p);
    release (&q);

    return 0;
}

int compare (struct node *p, struct node *q) {
    while (p != NULL && q != NULL)
    {
        if (p->num != q-> num)          {
            return 0;
        }
        else
        {
            p = p->next;
            q = q->next;
        }
    }
    if (p != NULL || q != NULL)      {
        return 0;
    }
```

```
    else    {
        return 1;
    }
}

void feedmember (struct node **head)
{
    int c, ch;
    struct node *temp;

    do      {
        printf("Enter number: ");
        scanf("%d", &c);
        temp = (struct node *)malloc(sizeof(struct node));
        temp->num = c;
        temp->next = *head;
        *head = temp;
        printf("Do you wish to continue [1/0]: ");
        scanf("%d", &ch);
    }while (ch != 0);
    printf("\n");
}

void release (struct node **head) {
    struct node *temp = *head;

    while ((*head) != NULL)
    {
        (*head) = (*head)->next;
        free(temp);
        temp = *head;
    }
}
```

INPUT/ OUTPUT

Enter data into first list
Enter number: 12
Do you wish to continue [1/0]: 1
Enter number: 3
Do you wish to continue [1/0]: 1
Enter number: 28
Do you wish to continue [1/0]: 1
Enter number: 9
Do you wish to continue [1/0]: 0

Enter data into second list
Enter number: 12
Do you wish to continue [1/0]: 1
Enter number: 3
Do you wish to continue [1/0]: 1
Enter number: 28
Do you wish to continue [1/0]: 1
Enter number: 9
Do you wish to continue [1/0]: 0

The 2 list are equal.

EXPERIMENT 2

2.1 OBJECTIVE

*Write a C program to find the largest element in a given doubly linked list.

2.2 SOURCE CODE

```
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int num;
    struct node *next;
    struct node *prev;
};

void create(struct node **);
int max(struct node *);
void release(struct node **);

int main()
{
    struct node *p = NULL;
    int n;

    printf("Enter data into the list\n");
    create(&p);
    n = max(p);
    printf("The maximum number entered in the list is %d.\n", n);
    release (&p);

    return 0;
}

int max(struct node *head)
{
    struct node *max, *q;

    q = max = head;
    while (q != NULL)
    {
        if (q->num > max->num)
        {
            max = q;
        }
        q = q->next;
    }

    return (max->num);
}

void create(struct node **head)
{
    int c, ch;
    struct node *temp, *rear;
```

```
    do
    {
        printf("Enter number: ");
        scanf("%d", &c);
        temp = (struct node *)malloc(sizeof(struct node));
        temp->num = c;
        temp->next = NULL;
        temp->prev = NULL;
        if (*head == NULL)
        {
            *head = temp;
        }
        else
        {
            rear->next = temp;
            temp->prev = rear;
        }
        rear = temp;
        printf("Do you wish to continue [1/0]: ");
        scanf("%d", &ch);
    } while (ch != 0);
    printf("\n");
}

void release(struct node **head)
{
    struct node *temp = *head;
    *head = (*head)->next;
    while ((*head) != NULL)
    {
        free(temp);
        temp = *head;
        (*head) = (*head)->next;
    }
}
```

INPUT/ OUTPUT:

```
Enter data into the list
Enter number: 12
Do you wish to continue [1/0]: 1
Enter number: 7
Do you wish to continue [1/0]: 1
Enter number: 23
Do you wish to continue [1/0]: 1
Enter number: 4
Do you wish to continue [1/0]: 1
Enter number: 1
Do you wish to continue [1/0]: 1
Enter number: 16
Do you wish to continue [1/0]: 0

The maximum number entered in the list is 23.
```

EXPERIMENT 3

3.1 OBJECTIVE

*Write a C program to reverse the elements in the stack using recursion.

3.2 SOURCE CODE

```
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int a;
    struct node *next;
};

void generate(struct node **);
void display(struct node *);
void stack_reverse(struct node **, struct node **);
void delete(struct node **);

int main()
{
    struct node *head = NULL;

    generate(&head);
    printf("\nThe sequence of contents in stack\n");
    display(head);
    printf("\nInversing the contents of the stack\n");
    if (head != NULL)
    {
        stack_reverse(&head, &(head->next));
    }
    printf("\nThe contents in stack after reversal\n");
    display(head);
    delete(&head);

    return 0;
}

void stack_reverse(struct node **head, struct node **head_next)
{
    struct node *temp;

    if (*head_next != NULL)
    {
         temp = (*head_next)->next;
        (*head_next)->next = (*head);
        *head = *head_next;
        *head_next = temp;
        stack_reverse(head, head_next);
    }
}

void display(struct node *head)
{
```

```
    if (head != NULL)
    {
        printf("%d ", head->a);
        display(head->next);
    }
}

void generate(struct node **head)
{
    int num, i;
    struct node *temp;

    printf("Enter length of list: ");
    scanf("%d", &num);
    for (i = num; i > 0; i--)
    {
        temp = (struct node *)malloc(sizeof(struct node));
        temp->a = i;
        if (*head == NULL)
        {
            *head = temp;
            (*head)->next = NULL;
        }
        else
        {
            temp->next = *head;
            *head = temp;
        }
    }
}

void delete(struct node **head)
{
    struct node *temp;
    while (*head != NULL)
    {
        temp = *head;
        *head = (*head)->next;
        free(temp);
    }
}
```

INPUT/OUTPUT:

Enter length of list: 10

The sequence of contents in stack
1 2 3 4 5 6 7 8 9 10

Reversing the contents of the stack

The contents in stack after reversal
10 9 8 7 6 5 4 3 2 1

EXPERIMENT 4

4.1 OBJECTIVE

*Write a C program to implement stack using linked list.

4.2 SOURCE CODE

```
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int info;
    struct node *ptr;
}*top,*top1,*temp;

int topelement();
void push(int data);
void pop();
void empty();
void display();
void destroy();
void stack_count();
void create();

int count = 0;

void main()
{
    int no, ch, e;

    printf("\n 1 - Push");
    printf("\n 2 - Pop");
    printf("\n 3 - Top");
    printf("\n 4 - Empty");
    printf("\n 5 - Exit");
    printf("\n 6 - Dipslay");
    printf("\n 7 - Stack Count");
    printf("\n 8 - Destroy stack");

    create();

    while (1)
    {
        printf("\n Enter choice : ");
        scanf("%d", &ch);

        switch (ch)
        {
        case 1:
            printf("Enter data : ");
            scanf("%d", &no);
            push(no);
            break;
        case 2:
            pop();
            break;
```

```
        case 3:
            if (top == NULL)
                printf("No elements in stack");
            else
            {
                e = topelement();
                printf("\n Top element : %d", e);
            }
            break;
        case 4:
            empty();
            break;
        case 5:
            exit(0);
        case 6:
            display();
            break;
        case 7:
            stack_count();
            break;
        case 8:
            destroy();
            break;
        default :
            printf(" Wrong choice, Please enter correct choice ");
            break;
        }
    }
}

/* Create empty stack */
void create()
{
    top = NULL;
}

/* Count stack elements */
void stack_count()
{
    printf("\n No. of elements in stack : %d", count);
}

/* Push data into stack */
void push(int data)
{
    if (top == NULL)
    {
        top =(struct node *)malloc(1*sizeof(struct node));
        top->ptr = NULL;
        top->info = data;
    }
    else
    {
        temp =(struct node *)malloc(1*sizeof(struct node));
        temp->ptr = top;
        temp->info = data;
        top = temp;
    }
```

```
    count++;
}

/* Display stack elements */
void display()
{
    top1 = top;

    if (top1 == NULL)
    {
        printf("Stack is empty");
        return;
    }

    while (top1 != NULL)
    {
        printf("%d ", top1->info);
        top1 = top1->ptr;
    }
 }

/* Pop Operation on stack */
void pop()
{
    top1 = top;

    if (top1 == NULL)
    {
        printf("\n Error : Trying to pop from empty stack");
        return;
    }
    else
        top1 = top1->ptr;
    printf("\n Popped value : %d", top->info);
    free(top);
    top = top1;
    count--;
}

/* Return top element */
int topelement()
{
    return(top->info);
}

/* Check if stack is empty or not */
void empty()
{
    if (top == NULL)
        printf("\n Stack is empty");
    else
        printf("\n Stack is not empty with %d elements", count);
}

/* Destroy entire stack */
void destroy()
{
    top1 = top;
```

```
    while (top1 != NULL)
    {
        top1 = top->ptr;
        free(top);
        top = top1;
        top1 = top1->ptr;
    }
    free(top1);
    top = NULL;

    printf("\n All stack elements destroyed");
    count = 0;
}
```

INPUT/OUTPUT:

```
1 - Push
2 - Pop
3 - Top
4 - Empty
5 - Exit
6 - Dipslay
7 - Stack Count
8 - Destroy stack
Enter choice : 1
Enter data : 56

Enter choice : 1
Enter data : 80

Enter choice : 2

Popped value : 80
Enter choice : 3

Top element : 56
Enter choice : 1
Enter data : 78

Enter choice : 1
Enter data : 90

Enter choice : 6
90 78 56
Enter choice : 7

No. of elements in stack : 3
Enter choice : 8

All stack elements destroyed
Enter choice : 4

Stack is empty
Enter choice : 5
```

EXPERIMENT 5

5.1 OBJECTIVE

*Write a C program to count the number of nodes in the binary search tree.

5.2 SOURCE CODE

```
#include < stdio.h >
#include < conio.h >
#include < alloc.h >
#define new_node (struct node*)malloc(sizeof (struct node))

struct node
{
        int data;
        struct node *lchild;
        struct node *rchild;
};

struct node *create_bin_rec();
void print_bin_pre_rec(struct node *t);
void cnt_nodes(struct node *t, int *l, int *nl);

void main()
{
        struct node *root;
        int leaf,nonleaf;
        clrscr();
        printf("\nCreate a binary tree \n");
        root = create_bin_rec();
        printf("\n Binary tree is ");
        print_bin_pre_rec(root);
        leaf = nonleaf = 0;
        cnt_nodes(root,&leaf,&nonleaf);
        printf("\n Total no. of leaf nodes are : %d ",leaf);
        printf("\n Total no. of nonleaf nodes are : %d ",nonleaf);
        printf("\n Total no. of nodes are : %d ", (leaf+nonleaf));

} // main

struct node *create_bin_rec()
{
        int  data;
        struct node *t;
        printf("\nData ( -1 to exit ) : ");
        scanf("%d",&data);
        if( data == -1)
                return(NULL);
        else
        {
                t = new_node;
                t->data = data;
                t->lchild =create_bin_rec();
                t->rchild =create_bin_rec();
                return(t);
        } //else
```

```
} // create

void print_bin_pre_rec(struct node *t)
{
        if( t != NULL)
        {
                printf("%4d",t->data);
                print_bin_pre_rec(t->lchild);
                print_bin_pre_rec(t->rchild);
        } // if
} // print bin pre rec

void cnt_nodes(struct node *t, int *l, int *nl)
{
        if( t != NULL)
        {
                if( t->lchild == NULL && t->rchild == NULL)
                        (*l)++;
                else
                        (*nl)++;
                        cnt_nodes(t->lchild,l,nl);
                        cnt_nodes(t->rchild,l,nl);
        } // if
} // cnt nodes
```

INPUT OUTPUT

```
Create binary tree
Data (-1 to exit) 10
Data (-1 to exit) 20
Data (-1 to exit)  -1

Binary tree is 10 20
Total no. of leaf nodes are 1
Total no. of non leaf nodes are 1
Total no. of nodes are 2
```

EXPERIMENT 6

6.1 OBJECTIVE

*Write a C program to sort an array of integers in ascending order using radix sort.

6.2 SOURCE CODE

```
#include <stdio.h>
int min = 0, count = 0, array[100] = {0}, array1[100] = {0};

void main()
{
   int k, i, j, temp, t, n;

   printf("Enter size of array :");
   scanf("%d", &count);
   printf("Enter elements into array :");
   for (i = 0; i < count; i++)
   {
      scanf("%d", &array[i]);
      array1[i] = array[i];
   }
   for (k = 0; k < 3; k++)
   {
      for (i = 0; i < count; i++)
      {
         min = array[i] % 10;        /* To find minimum lsd */
         t = i;
         for (j = i + 1; j < count; j++)
         {
            if (min > (array[j] % 10))
            {
               min = array[j] % 10;
               t = j;
            }
         }
         temp = array1[t];
         array1[t] = array1[i];
         array1[i] = temp;
         temp = array[t];
         array[t] = array[i];
         array[i] = temp;

      }
      for (j = 0; j < count; j++)        /*to find MSB */
         array[j] = array[j] / 10;
   }
   printf("Sorted Array (lSdradix sort) : ");
   for (i = 0; i < count; i++)
      printf("%d ", array1[i]);
}
```

INPUT/ OUTPUT:

/* Average Case */

Enter size of array :7

Enter elements into array :170
45
90
75
802
24
2
Sorted Array (ladradix sort) : 2 24 45 75 90 170 802

/*Best case */
Enter size of array :7
Enter elements into array :22
64
121
78
159
206
348
Sorted Array (ladradix sort) : 22 64 78 159 121 206 348

/* Worst case */
Enter size of array :7
Enter elements into array :985
27
64
129
345
325
091
Sorted Array (ladradix sort) : 27 64 91 129 325 345 985

EXPERIMENT 7

7.1 OBJECTIVE

*Write a C program to sort a given list of strings.

7.2 SOURCE CODE

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
int main() {
char *str[5], *temp;
 int i, j, n;
 printf("\nHow many names do you want to have?");
 scanf("%d", &n);
 for (i = 0; i < n; i++) {
      printf("\nEnter the name %d: ", i);
      flushall();
      gets(str[i]);
}
for (i = 0; i < n; i++) {
     for (j = 0; j < n - 1; j++) {
        if (strcmp(str[j], str[j + 1]) > 0) {
            strcpy(temp, str[j]);
            strcpy(str[j], str[j + 1]);
            strcpy(str[j + 1], temp);
} } }
printf("\nSorted List : ");
for (i = 0; i < n; i++)
      puts(str[i]);
   return (0);
}
```

INPUT /OUTPUT:

Enter any strings :

pri

pra

pru

pry

prn

Strings in order are :

pra

pri

prn

pru

pry

ANNEXURE-I

Assessment Rubric

1. **Laboratory Report**

Category	Outstanding (Up to 100%)	Accomplished (Up to 75%)	Developing (Up to 50%)	Beginner (Up to 25%)
Write up format	Aim, Apparatus, material requirement, theoretical basis, procedure of experiment, sketch of the experimental setup etc. is demarcated and presented in clearly labeled and neatly organized sections.	The write up follows the specified format but a couple of the specified parameters are missing.	The report follows the specified format but a few of the formats are missing and the experimental sketch is not included in the report	The write up does not follow the specified format and the presentation is shabby.
Observations and Calculations	The experimental observations and calculations are recorded in neatly prepared table with correct units and significant figures. One sample calculation is explained by substitution of values	The experimental observations and calculations are recorded in neatly prepared table with correct units and significant figures but sample calculation is not shown	The experimental observations and calculations are recorded neatly but correct units and significant figures are not used. Sample calculation is also not shown	The experimental observations and results are recorded carelessly. Correct units significant figures are not followed and sample calculations not shown

Results and Graphs	Results obtained are correct within reasonable limits. Graphs are drawn neatly with labeling of the axes. Relevant calculations are performed from the graphs. Equations are obtained by regression analysis or curve fitting if relevant	Results obtained are correct within reasonable limits. Graphs are drawn neatly with labeling of the axes. Relevant calculations from the graphs are incomplete and equations are not obtained by regression analysis or curve fitting	Results obtained are correct within reasonable limits. Graphs are not drawn neatly and or labeling is not proper. No calculations are done from the graphs and equations are not obtained by regression analysis or curve fitting	Results obtained are not correct within reasonable limits. Graphs are not drawn neatly and or labeling is not proper. No calculations are done from the graphs and equations are not obtained by regression analysis or curve fitting
Discussion of results	All relevant points of the result are discussed and justified in light of theoretical expectations. Reasons for divergent results are identified and corrective measures discussed.	Results are discussed but no theoretical reference is mentioned. Divergent results are identified but no satisfactory reasoning is given for the same.	Discussion of results is incomplete and divergent results are not identified.	Neither relevant points of the results are discussed nor divergent results identified

2. Individual Presentation

Category	Outstanding (Up to 100%)	Accomplished (Up to 75%)	Developing (Up to 50%)	Beginner (Up to 25%)
Content	A concise summary of the topic; Convincing justification for choice of topic; Comprehensive and complete coverage of information	A good summary of the topic; Acceptable justification for choice of topic; Most important information covered; Little irrelevant information	Informative but much of the information irrelevant; Confused justification for choice of topic; Coverage of some of the major points	A brief look at the topic; Little justification for choice of topic; Majority of information irrelevant and significant points left out
Organization	Clear purpose and subject; Pertinent examples, facts, and/or statistics; Supports conclusions/ideas with evidence	Somewhat clear purpose and subject; Some examples, facts, and/or statistics that support the subject; Some data or evidence that supports conclusions	Attempts to define purpose and subject; Weak examples, facts, and/or statistics not adequately supporting the subject; Very thin data or evidence to support conclusion	Subject and purpose not clearly defined; Weak or no support of subject; Insufficient support for ideas or conclusions
Visual Aids	Information is clear and concise with proper key information in points or phrases; Visually appealing/ engaging	Too much information in complete sentences on slides along with proper key information in phrases; Significant visual appeal	Too much information in complete sentences on many slides; Some proper key information; Minimal effort made to make slides appealing	Too much information in complete sentences on slides; No or few proper key information; Repetition of the same information on multiple slides; No visual appeal
Delivery Style	Regular eye contact; Appropriate speaking volume & body language; Proper pace and diction; Fluent avoidance of repetitions, hesitations, gap fillers	Steady eye contact; Adequate volume and energy; Generally good pace and diction; Few or no distracting Gestures; Few repetitions, hesitations, gap fillers	More volume or energy needed at times; Pace too slow or fast; Some distracting gestures or posture; Some repetitions, hesitations, gap fillers	Low volume and energy; Pace too slow or fast; Poor diction; Lots of distracting gestures or posture; Frequent repetitions, hesitations, gap fillers

Question -answer Session	Demonstrates knowledge by answering all types of questions with explanations and elaboration in professional manner	Is at ease with expected answers to all questions without elaboration in somewhat professional manner	Is uncomfortable with information and can answer only rudimentary questions	Does not have grasp of information and cannot answer questions about subject

ANNEXURE-II

Program Outcomes

Program Outcomes (POs) represent the knowledge, skills and attitudes the students should have at the end of a four year engineering program. CSE program of BAUET has 12 Program Outcomes. They are briefly described in the following table.

Sl. No	PO	Category	Description
1	**PO 1**	**Engineering Knowledge**	Apply the **knowledge** of **mathematics**, **science**, **engineering fundamentals**, and an **engineering specialization** to the solution of complex engineering problems.
2	**PO 2**	**Problem Analysis**	**Identify**, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions **using first principles** of mathematics, natural sciences, and engineering sciences.
3	**PO 3**	**Design/Development of Solutions**	**Design solutions** for complex engineering problems and **design system** components or processes that meet the specified needs with appropriate consideration for public health and safety as well as cultural, social and environmental concerns.
4	**PO 4**	**Investigation**	**Conduct investigations** of complex problems, considering design of experiments, analysis and interpretation of data and synthesis of information to provide valid conclusions.
5	**PO 5**	**Modern Tool Usage**	**Create, select, and apply** appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

6	**PO 6**	**The Engineer and Society**	**Apply reasoning** informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7	**PO 7**	**Environment and Sustainability**	Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of need for sustainable development.
8	**PO 8**	**Ethics**	Apply ethical principles and commit to professional Ethics and responsibilities and norms of the engineering practice.
9	**PO 9**	**Individual and Team Work**	Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10	**PO 10**	**Communication**	Communicate effectively on complex engineering activities with the engineering community and with society at large. Some of them are, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11	**PO 11**	**Project Management and Finance**	Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12	**PO 12**	**Life Long Learning**	Recognize the need for, and have the preparation and ability to engage in independent and lifelong learning in the broadest context of technological change.

ANNEXURE-III

Knowledge Profile, Complex Engineering Problem and Complex Engineering Activities

Knowledge Profile (WK/K)- CHARACTERISTIC

WK1	**Natural Sciences**	A systematic, theory-based understanding of the natural sciences applicable to the discipline
WK2	**Mathematics**	Conceptually-based mathematics, numerical analysis, statistics and formal aspects of computer and information science to support analysis and modelling applicable to the discipline
WK3	**Engineering fundamentals**	A systematic, theory-based formulation of engineering fundamentals required in the engineering discipline
WK4	**Specialist knowledge**	Engineering specialist knowledge that provides theoretical frameworks and bodies of knowledge for the accepted practice areas in the engineering discipline; much is at the forefront of the discipline.
WK5	**Engineering design**	Knowledge that supports engineering design in a practice area.
WK6	**Engineering practice**	Knowledge of engineering practice (technology) in the practice areas in the engineering discipline
WK7	**Comprehension**	Comprehension of the role of engineering in society and identified issues in engineering practice in the discipline: ethics and the professional responsibility of an engineer to public safety; the impacts of engineering activity: economic, social, cultural, environmental and sustainability
WK8	**Research literature**	Engagement with selected knowledge in the research literature of the discipline

Complex Engineering Problem (WP/P)

WP	**Preamble**	**COMPLEX PROBLEMS have characteristic of WP1 and some or all of WP2 to WP7**
WP1	**Depth of Knowledge**	**In-depth engineering knowledge** at the level of one or more of WK3, WK4, WK5, WK6 or WK8 which allows a fundamental based, first principles analytical approach
WP2	**Conflicting requirement**	Wide-ranging or **conflicting** technical, engineering and other issues

WP3	**Depth of analysis**	**no obvious solution** and require abstract thinking, originality in analysis to formulate suitable models
WP4	**Familiarity of issues**	**infrequently** encountered issues
WP5	**Extent of applicable codes**	outside problems encompassed by standards and codes of practice for professional engineering
WP6	**Extent of stakeholder**	**diverse groups** of stakeholders with widely varying needs
WP7	**Interdependence**	high level problems including **many component parts** or sub-problems

Complex Engineering Activities (EA)

Activities	Preamble	**Complex activities** means (engineering) activities or projects that have **some or all** of the following characteristics listed below
EA1	**Range of resources**	**Diverse resources** (people, money, equipment, materials, information and technologies).
EA2	**Level of interaction**	Require resolution of significant problems arising from interactions between **wide ranging** or **conflicting** technical, engineering or other issues.
EA3	**Innovation**	Involve creative use of engineering principles and **research-based** knowledge in **novel** ways.
EA4	**Consequences to society and the environment**	Have **significant consequences** in a **range of contexts**, characterised by **difficulty** of prediction and mitigation.
EA5	**Familiarity**	Can extend **beyond previous** experiences by applying **principles-based** approaches.

The End