Sri
SAI RAM ENGINEERING COLLEGE
An Autonomous Institution | Affiliated to Anna University & Approved by AICTE, New Delhi
Accredited by NBA and NAAC "A+" | BIS/EOMS ISO 21001 : 2018 Certified and NIRF ranked institution
Sai Leo Nagar, West Tambaram, Chennai - 600 044. www.sairam.edu.in

# LAB MANUAL

## DATA STRUCTURES AND ALGORITHMS LABORATORY

**24ITPL301**

**(II YEAR / III SEM)**

**Academic Year: 2025-2026**

**BATCH 2024 – 2028**

**DEPARTMENT OF INFORMATION TECHNOLOGY**

## PREFACE

**"A Good Beginning Makes a Good End."**

We have immense pleasure in introducing the lab manual "**DATA STRUCTURES AND ALGORITHMS LABORATORY"**, intended for the curriculum of III semester B. Tech. IT students. With this in mind this manual is prepared as an introductory note for the laboratoryexperiments. Sufficient detail has been included to emphasize self learning.

The main objective of this manual is to make the students to familiarize with good programming design methods and to get exposure in different types of data structures & Algorithms.

This manual makes the students feel easy to do the applications and provide an in-depth knowledge in programming. We hope this manual is very helpful for second year students of B.Tech Information Technology for their **DATA STRUCTURES AND ALGORITHMS LABORATORY**.

Our sincere thanks to all staff members of Information Technology department for their valuable assistance and suggestions during the preparation of this manual.

**Vision of the Department**

To emerge as a "Centre of Excellence in the field of IT" offering Technological Education and Research opportunities of high standards to students, develop high degree of digital knowledge and skill set, making our students technologically superior and ethically strong, who in turn shall contribute to the advancement of society and humankind.

**Mission of the Department**

| | |
|---|---|
| **M1** | Provide quality education in Information Technology and also create technologically new and intellectually inspiringenvironment. |
| **M2** | Focus on research and development of technologies by engaging in value added courses and on evolution of digital environment. |
| **M3** | Design and Develop state-of-the art on learning, creativity, innovation and inculcate in them ethical, social and moralvalues. |
| **M4** | Establish continuous Industry Institute Interaction, participation and collaboration to contribute job oriented skilled IT Engineers by improving their entrepreneurial skills. |

**Program Educational Objectives (PEOs)**

| | |
|---|---|
| **PEO1** | Graduates will embed in applying the mathematical and engineering knowledge insolving problems in digital environment and understanding Industrial requirements |
| **PEO2** | Graduates will excel in Research, designing & developing solution for complex problems in the field of IT by adapting to the rapid technological advancements, thereby increasing industrial collaboration. |
| **PEO3** | Graduates are inculcated with lifelong learning to function as an individual or teamwith ethics & social responsibility, for the advancement of society and service to humankind. |
| **PEO4** | Graduates are trained in technical skills using modern tools to solve real timeproblems and evolve as entrepreneurs in the field of Information Technology. |

## PROGRAMME OUTCOMES (POS)

**PO1: Engineering Knowledge:** Apply knowledge of mathematics, natural science, computing, engineering fundamentals and an engineering specialization as specified in WK1 to WK4 respectively to develop to the solution of complex engineering problems.

**PO2: Problem Analysis:** Identify, formulate, review research literature and analyze complex engineering problems reaching substantiated conclusions with consideration for sustainable development. (WK1 to WK4)

**PO3: Design/Development of Solutions:** Design creative solutions for complex engineering problems and design/develop systems/components/processes to meet identified needs with consideration for the public health and safety, whole-life cost, net zero carbon, culture, society and environment as required. (WK5)

**PO4:Conduct Investigations of Complex Problems:** Conduct investigations of complex engineering problems using research-based knowledge including design of experiments, modelling, analysis & interpretation of data to provide valid conclusions. (WK8).

**PO5: Engineering Tool Usage:** Create, select and apply appropriate techniques, resources and modern engineering & IT tools, including prediction and modelling recognizing their limitations to solve complex engineering problems. (WK2 and WK6)

**PO6: The Engineer and The World:** Analyze and evaluate societal and environmental aspects while solving complex engineering problems for its impact on sustainability with reference to economy, health, safety, legal framework, culture and environment. (WK1, WK5, and WK7).

**PO7: Ethics:** Apply ethical principles and commit to professional ethics, human values, diversity and inclusion; adhere to national & international laws. (WK9)

**PO8: Individual and Collaborative Teamwork:** Function effectively as an individual, and as a member or leader in diverse/multi-disciplinary teams.

**PO9: Communication:** Communicate effectively and inclusively within the engineering community and society at large, such as being able to comprehend and write effective reports and design documentation, make effective presentations considering cultural, language, and learning differences

**PO10: Project Management and Finance:** Apply knowledge and understanding of engineeringmanagementprinciplesandeconomicdecision-makingandapplytheseto one's own work, as a member and leader in a team, and to manage projects and in multidisciplinary environments.

**PO11: Life-LongLearning:** Recognize the need for, and have the preparation and ability for
i) Independent and life-long learning ii) adaptability to new and emerging technologies and iii) critical thinking in the broadest context of technological change. (WK8)

**Program Specific outcomes (PSO):**

**PSO1**

Ability to build network based web application using different secured software designconcepts.

**PSO2**

Ability to design, implement and test information systems architecture to meet specificsoftware requirements following the societal values.

**SYLLABUS**

| | | | | | | |
|---|---|---|---|---|---|---|
| 24ITPL301 - SDG NO. 4 | DATA STRUCTURES AND ALGORITHMS LABORATORY | L | T | P | CP | C |
| | | 0 | 0 | 3 | 3 | 2 |

## OBJECTIVES

- To implement and demonstrate basic data structures
- To understand and apply algorithmic paradigms
- To apply graph algorithms and optimization techniques
- To develop and analyze efficient searching and sorting techniques
- To implement advanced data structures and algorithms

## LIST OF EXPERIMENTS:

1. Implement the basic operations of a stack using arrays (push, pop, peek, and isEmpty).
2. Implement the basic operations of a queue using arrays (enqueue, dequeue, front,rear, and isEmpty).
3. Implement the conversion of an infix expression to a postfix expression using a stack.
4. Implement a singly linked list with basic operations such as insertion, deletion, and traversal.
5. Implement a doubly linked list with basic operations such as insertion, deletion, and traversal.
6. Implement a binary tree and perform basic operations like insertion, deletion, and traversal (inorder, preorder, postorder).
7. Implementation of AVL tree(all operation).
8. Implementation of Priority queue using Heap.
9. Graph representation and Traversal algorithms a) Depth first Search b) Breadth First Search.
10. Implement Dijkstra's algorithm to find the shortest path from a source vertex to all other vertices in a graph.
11. Implement linear search and Binary search.
12. Implement merge sort and quick sort for Divide and Conquer.
13. Implement the 0/1 Knapsack Problem using the Greedy technique
14. Analyze and implement a recursive algorithm (like factorial or Fibonacci) TOTAL: 45 PERIODS

**OUTCOMES:**

Upon completion of the course, the student will be able to:

1. Apply basic linear and non linear data structures (K3)
2. Analyze and compare various trees, searching and sorting algorithms (K4)
3. Apply algorithmic strategies such as greedy and graph-based techniques (K3)

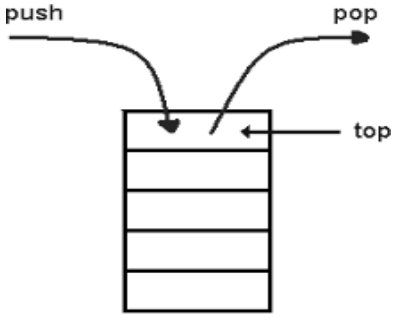
## CO – PO, PSO MAPPING:

| | PO1 | PO2 | PO3 | PO4 | PO5 | PO6 | PO7 | PO8 | PO9 | PO10 | PO11 | PSO1 | PSO2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CO1 | 2 | 3 | 3 | - | - | - | - | - | - | - | 1 | 3 | 3 |
| CO2 | 2 | 3 | 3 | - | - | - | - | - | - | - | 1 | 3 | 3 |
| Co3 | 2 | 2 | 3 | - | - | - | - | - | - | - | 1 | 3 | 3 |

# INDEX

## EX.NO.1 ARRAY IMPLEMENTATION OF STACK

A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle. In the pushdown stacks only two operations are allowed: push the item into the stack, and pop the item out of the stack.




**AIM:**

To write a 'C' program for implementing stacks using array.

**ALGORITHM**

Step1:Define a array which stores stack elements.
Step 2: Increment Top pointer and then PUSH data into the stack at the position pointed by the Top.
Step 3: POP the data out the stack at the position pointed by the Top.
Step 5. The stack represented by array is traversed to display its content.

**PROGRAM**

```
#include<stdio.h>
int stack[100], choice, n, top, x, i;
void push(void);
void pop(void);
void display(void);
int main()
{
   top=-1;
   printf("\n Enter the size of STACK[MAX=100]:");
   scanf("%d",&n);
   printf("\n\t STACK OPERATIONS USING ARRAY");
```

```
    printf("\n\t_____");
    printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");
    do
    {
       printf("\n Enter the Choice:");
       scanf("%d",&choice);
       switch(choice)
       {
          case 1:
          {
             push();
             break;
          }
          case 2:
          {
             pop();
             break;
          }
          case 3:
          {
             display();
             break;
          }
          case 4:
          {
             printf("\n\t EXIT POINT ");
             break;
          }
          default:
          {
             printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
          }     }    }
    while(choice!=4);
    return 0;
}
void push()
{
    if(top>=n-1)
    {
       printf("\n\tSTACK is over flow");
           }
```

```
    else
    {
       printf(" Enter a value to be pushed:");
       scanf("%d",&x);
       top++;
       stack[top]=x;
    }}
void pop()
{
    if(top<=-1)
    {
       printf("\n\t Stack is under flow");
    }
    else
    {
       printf("\n\t The popped elements is %d",stack[top]);
       top--;
    }}
void display()
{
    if(top>=0)
    {
       printf("\n The elements in STACK \n");
       for(i=top; i>=0; i--)
          printf("\n%d",stack[i]);
       printf("\n Press Next Choice");
    }
    else
    {
       printf("\n The STACK is empty");
    }
    }
```

**OUTPUT**

```
Enter the size of STACK[MAX=100]:10

        STACK OPERATIONS USING ARRAY
       ----------------------------------
        1.PUSH
        2.POP
        3.DISPLAY
        4.EXIT
```

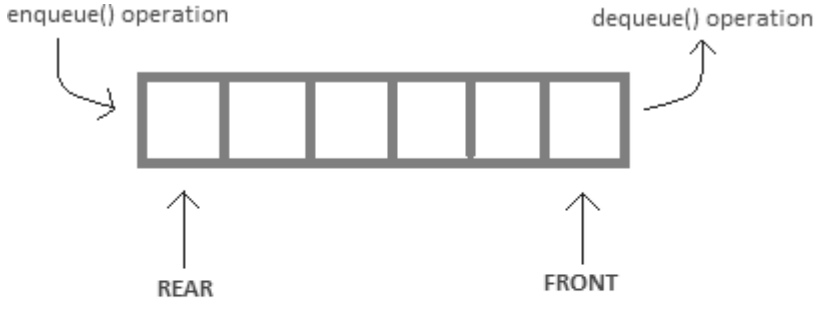
```
Enter the Choice:1
Enter a value to be pushed:12
Enter the Choice:1
Enter a value to be pushed:24
Enter the Choice:1
Enter a value to be pushed:98
Enter the Choice:3

The elements in STACK
98
24
12
Press Next Choice
Enter the Choice:2

      The popped elements is 98
Enter the Choice:3

The elements in STACK
24
12
Press Next Choice
Enter the Choice:4

EXIT POINT
```

**RESULT:**

Thus the program to implement stack using array has been implemented and the output is verified.

**Ex. No. 2** **ARRAY IMPLEMENTATION OF QUEUE**

Queue is a linear structure which follows a particular order in which the operations are performed. The order is First in First out (FIFO). A good example of queue is any queueof consumers for a resource where the consumer that came first is served first. The difference between stacks and queues is in removing. In a stack we remove the item themost recently added; in a queue, we remove the item the least recently added.




**enqueue( )** is the operation for adding an element into Queue.
**dequeue( )** is the operation for removing an element from Queue .

**QUEUE DATA STRUCTURE**

**AIM:**

To write a 'C' program for implementing Queue using array.

**ALGORITHM:**

Step 1 : To insert Check if the queue is full or not.
Step 2 : If the queue is full, then print overflow error and exit the program.
Step 3 : If the queue is not full, then increment the tail and add the element.
Step 4 : Check if the queue is empty or not.
Step 5 : If the queue is empty, then print underflow error and exit the program. Step 6 :If the queue is not empty, then print the element at the head and incrementthe he

**PROGRAM:**

```
#include<stdio.h>
#include<stdlib.h>
#define n 5
void main()
{
   int queue[n],ch=1,front=0,rear=0,i,j=1,x=n;
```

```
//clrscr();
printf("Queue using Array");
printf("\n1.Insertion \n2.Deletion \n3.Display \n4.Exit");
while(ch)
{
   printf("\nEnter the Choice:");
   scanf("%d",&ch);
   switch(ch)
   {
   case 1:
      if(rear==x)
         printf("\n Queue is Full");
      else
      {
         printf("\n Enter no %d:",j++);
         scanf("%d",&queue[rear++]);
      }    break;
   case 2:
      if(front==rear)
      {
         printf("\n Queue is empty");
      }
      else
      {
         printf("\n Deleted Element is %d",queue[front++]);
         x++;
      }
      break;
   case 3:
      printf("\n Queue Elements are:\n ");
      if(front==rear)
         printf("\n Queue is Empty");
      else
      {     for(i=front; i<rear; i++)
         {
            printf("%d",queue[i]);
            printf("\n");
         }
         break;
      case 4:
         exit(0);
```

```
            default:
                printf("Wrong Choice: please see the options");
            }
        }
    }
}
```

**OUTPUT:**

```
Queue using Array
1.Insertion
2.Deletion
3.Display
4.Exit
Enter the Choice:1
Enter no 1:10
Enter the Choice:1
Enter no 2:54
Enter the Choice:1
Enter no 3:98
Enter the Choice:1
Enter no 4:234
Enter the Choice:3
 Queue Elements are:
 10
54
98
234
Enter the Choice:2
Deleted Element is 10
Enter the Choice:3
Queue Elements are:
 54
98
234

Enter the Choice:4
```

**RESULT:**

Thus the ‘C’ program to implement Queue using array has been executed and the output is verified.

## EX.NO.3 CONVERT INFIX TO POSTFIX EXPRESSION USING STACK

**AIM:**

To write a 'C' program to implement stack and use it to convert infix to postfix expression.

**ALGORITHM:**

1. Start the program
2. Scan the Infix string from left to right.
3. Initialize an empty stack.
4. If the scanned character is an operand, add it to the Postfix string. If the scanned character is an operator and if the stack is empty Push the character to stack.
5. If the scanned character is an Operand and the stack is not empty, compare the precedence of the character with the element on top of the stack (topStack). If topStack has higher precedence over the scanned character Pop the stack else Push the scanned character to stack. Repeat this step as long as stack is not empty and topStack has precedence over the character.
6. Repeat this step till all the characters are scanned.
7. (After all characters are scanned, we have to add any character that the stack may have to the Postfix string.) If stack is not empty add topStack to Postfix string and Pop the stack. Repeat this step as long as stack is not empty.
8. Return the Postfix string.
9. Terminate the program.

**PROGRAM:**

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
char stack[100];
int top=0;
char ex[100];
struct table
{
      char s[2];
      int isp;
      int icp;
}pr[7];
int isp(char c)
{
      int i;
      for(i=0;i<=6;i++)
            if(pr[i].s[0]==c)
```

```
            return(pr[i].isp);
return 0;
}
int icp(char c)
{
      int i;
      for(i=0;i<=6;i++)
            if(pr[i].s[0]==c)
                  return(pr[i].icp);
  return 0;
}
void main()
{
      int i;
      strcpy(pr[0].s,"^"); pr[0].isp=3;
      pr[0].icp=4; strcpy(pr[1].s,"*"); pr[1].isp=2;
      pr[1].icp=2; strcpy(pr[2].s,"/"); pr[2].isp=2;
      pr[2].icp=2; strcpy(pr[3].s,"+"); pr[3].isp=1;
      pr[3].icp=1; strcpy(pr[4].s,"-"); pr[4].isp=1;
      pr[4].icp=1; strcpy(pr[5].s,"("); pr[5].isp=0;
      pr[5].icp=4; strcpy(pr[6].s,"="); pr[6].isp=-1;
      pr[6].icp=0;
      stack[top]='=';
      printf("enter the infix expression");
      fgets(ex, 100, stdin);
      i=0;
      printf("the postfix expression is ");
      while(i<strlen(ex))
      {
            if(isalpha(ex[i])==0)
            {
                  if(ex[i]==')')
                  {
                        while(stack[top]!='(')
                        {
                              printf("%c",stack[top]); top--;
                        }
                        top--;
                  }
                  else
                  {
                        while(isp(stack[top])>=icp(ex[i]))
                        {
                              printf("%c",stack[top]); top--;
                        }
```

```
                        top++;
                        stack[top]=ex[i];
                }
        }else
                printf("%c",ex[i]); i++;
    }
    while(top>0)
    {
        printf("%c",stack[top]); top--;
    }
}
```

**OUTPUT:**

enter the infix expression a*(s+d/f)+c
the postfix expression is asdf/+*c+

**RESULT:**

Thus the program to implement Infix to Post fix using stack is executed and verified successfully.

## EX.NO. 4 SINGLY LINKED LIST

**AIM:**

To write a 'C' program to create a singly linked list implementation.

**ALGORITHM:**

Step 1: Start the program.
Step 2: Get the choice from the user.
Step3: If the choice is to add records, get the data from the user and add them to the list.
Step 4: If the choice is to delete records, get the data to be deleted and delete it from the list.
Step 5: If the choice is to display number of records, count the items in the list and display.
Step 6: If the choice is to search for an item, get the item to be searched and respond yes if the item is found, otherwise no.
Step 7: Terminate the program

**PROGRAM:**

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<ctype.h>
struct info
{
int data;
struct info *next;
};
struct info *head,*temp,*disp;
void additem();
void delitem();
void display();
int size();
void search();
void main()
{
int choice;
while(1)
{
printf("\n1.Add records");
```

```
printf("\n2.Delete records");
printf("\n3.Display records");
printf("\n4.Count no. of items in the list");
printf("\n5.Searching an item in the list");
printf("\n6.Exit");
printf("\nEnter your choice:");
scanf("%d",&choice);
fflush(stdin);
switch(choice)
{
case 1:
additem();
break;
case 2:
delitem();
break;
case 3:
display();
break;
case 4:
printf("\nThe size of the list is %d",size());
break;
case 5:
search();
break;
case 6:
exit(0);
}}}
void additem()
{
struct info *add;
char proceed='y';
while(toupper(proceed)=='Y')
{
add=(struct info*)malloc(sizeof(struct info));
printf("Enter data:");
scanf("%d",&add->data); fflush(stdin);
if(head==NULL)
{
head=add;
add->next=NULL; temp=add;
}
else
```

```
{
temp->next=add; add->next=NULL; temp=add;
}
printf("\nWant to proceed y/n"); proceed=getchar(); fflush(stdin);
}}
void delitem()
{
struct info *curr,*prev; int tdata; if(head==NULL)
{
printf("\nNo records to delete"); return;
}
printf("\nEnter the data to delete"); scanf("%d",&tdata);
fflush(stdin); prev=curr=head;
while((curr!=NULL)&&(curr->data!=tdata))
{
prev=curr; curr=curr->next;
}
if(curr==NULL)
{
printf("\nData not found"); return;
}
if(curr==head) head=head->next;
else
{
/*for inbetween element deletion*/
prev->next=curr->next;
/*for the last element deletion*/
if(curr->next==NULL)
temp=prev;
}
free(curr);
}
void display()
{
if(head==NULL)
{
printf("\nNo data to display"); return;
}
for(disp=head;disp!=NULL;disp=disp->next)
{
printf("Data->%d",disp->data);
}}
int size()
```

```
{
int count=0; if(head==NULL)
return count; for(disp=head;disp!=NULL;disp=disp->next)
count++; return count;
}
void search()
{
int titem,found=0; if(head==NULL)
{
printf("\n No data in the list"); return;
}
printf("\n Enter the no. to search:"); scanf("%d",&titem);
for(disp=head;disp!=NULL&&found==0;disp=disp->next)
{
if(disp->data==titem) found=1;
}
if(found==0)
printf("\n Search no. is not present in the list"); else
printf("\nSearch no. is present in the list"); return;
}
```

**OUTPUT:**

```
1.Add records
2.Delete records
3.Display records
4.Count no. of items in the list
5.Searching an item in the list
6.Exit
Enter your choice:1
Enter data:12

 Want to proceed y/n
1.Add records
2.Delete records
3.Display records
4.Count no. of items in the list
5.Searching an item in the list
6.Exit
Enter your choice:1
Enter data:23

 Want to proceed y/n
```

1.Add records
2.Delete records
3.Display records
4.Count no. of items in the list
5.Searching an item in the list
6.Exit
Enter your choice:1
Enter data:34

Want to proceed y/n
1.Add records
2.Delete records
3.Display records
4.Count no. of items in the list
5.Searching an item in the list
6.Exit
Enter your choice:3

Data->12
Data->23
Data->34
1.Add records
2.Delete records
3.Display records
4.Count no. of items in the list
5.Searching an item in the list
6.Exit
Enter your choice:4

The size of the list is 3
1.Add records
2.Delete records
3.Display records
4.Count no. of items in the list
5.Searching an item in the list
6.Exit
Enter your choice:5

Enter the no. to search:34

Search no. is present in the list
1.Add records
2.Delete records

3.Display records
4.Count no. of items in the list
5.Searching an item in the list
6.Exit
Enter your choice:5

Enter the no. to search:67

Search no. is not present in the list
1.Add records
2.Delete records
3.Display records
4.Count no. of items in the list
5.Searching an item in the list
6.Exit
Enter your choice:2

Enter the data to delete34

1.Add records
2.Delete records
3.Display records
4.Count no. of items in the list
5.Searching an item in the list
6.Exit
Enter your choice:3

Data->12
Data->23
1.Add records
2.Delete records
3.Display records
4.Count no. of items in the list
5.Searching an item in the list
6.Exit
Enter your choice:2

Enter the data to delete67

Data not found
1.Add records
2.Delete records
3.Display records

4.Count no. of items in the list
5.Searching an item in the list
6.Exit
Enter your choice:3

Data->12
Data->23
1.Add records
2.Delete records
3.Display records
4.Count no. of items in the list
5.Searching an item in the list
6.Exit
Enter your choice: 6

**RESULT:**

Thus the program to implement Singly Linked List is executed and verified successfully.

## Ex. No. 5 DOUBLY LINKED LIST

**AIM:**

To write a 'C' program to create a doubly linked list implementation.

**ALGORITHM:**

Step 1:Start the program.
Step 2:Get the choice from the user.
Step 3:If the choice is to add records, get the data from the user and add them to the list.
Step 4:If the choice is to delete records, get the data to be deleted and delete it from the list.
Step 5:If the choice is to display number of records, count the items in the list and display.
Step 6:If the choice is to search for an item, get the item to be searched and respond yes if the item is found, otherwise no.
Step 7:Terminate the program

**PROGRAM:**

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
#define NULL 0
struct info
{
int data;
struct info *next;
struct info *prev;
};
struct info *head,*temp,*disp;
void additem();
void delitem();
void display();
int size();
void search();
void main()
{
int choice;
clrscr();
while(1)
```

```
{
printf("\n1.Add records");
printf("\n2.Delete records");
printf("\n3.Display records");
printf("\n4.Count no. of items in the list");
printf("\n5.Searching an item in the list");
printf("\n6.Exit");
printf("\nEnter your choice:");
scanf("%d",&choice);
fflush(stdin);
switch(choice)
{
case 1:
additem();
     break;
case 2:
delitem();
     break;
case 3:
display();
     break;
case 4:
printf("\nThe size of the list is %d",size());
     break;
case 5:
search();
     break;
case 6:
exit(0);
}}}
void additem()
{
struct info *add;
char proceed='y';
while(toupper(proceed)=='Y')
{
add=(struct info*)malloc(sizeof(struct info));
printf("Enter data:");
scanf("%d",&add->data);
fflush(stdin);
if(head==NULL)
```

```
{
head=add;
add->next=NULL;
add->prev=NULL;
temp=add;
}
else
{
temp->next=add;
add->prev=temp;
add->next=NULL;
temp=add;
}
printf("\nWant to proceed y/n");
proceed=getchar();
fflush(stdin);
}
}
void delitem()
{
int x;
struct info *p;;
if(head==NULL)
{
printf("\nNo items in the list");
return;
}
printf("\nEnter the data to delete");
scanf("%d",&x);
//fflush(stdin);
p=(struct info *)malloc(sizeof(struct info));
p=head->next;
if(head->data==x)
{
head=head->next;
return;
}
while(p)
{
if(p->data==x)
{
```

```
p->prev->next=p->next;
if(p->next!=NULL)
    p->next->prev=p->prev;
else
    temp=p->prev;
return;
}
else
{
       p=p->next;
}
}
printf("\nInvalid input");
}
void display()
{
if(head==NULL)
{
printf("\nNo data to display");
return;
}
printf("\nFrom forward direction\n");
for(disp=head;disp!=NULL;disp=disp->next)
{
printf("Data->%d",disp->data);
}
printf("\nFrom backward direction\n");
for(disp=temp;disp!=NULL;disp=disp->prev)
{
printf("Data->%d",disp->data);
}
}
int size()
{
int count=0;
if(head==NULL)
    return count;
for(disp=head;disp!=NULL;disp=disp->next)
   count++;
return count;
}
```

```
void search()
{
int titem,found=0;
if(head==NULL)
{
printf("\nNo data in the list");
return;
}
printf("\Enter the no. to search:");
scanf("%d",&titem);
for(disp=head;disp!=NULL&&found==0;disp=disp->next)
{
if(disp->data==titem)
      found=1;
}
if(found==0)
printf("\nSearch no. is not present in the list");
else
printf("\nSearch no. is present in the list");
return;
}
```

**OUTPUT:**

```
1.Add records
2.Delete records
3.Display records
4.Count no. of items in the list
5.Searching an item in the list
6.Exit
Enter your choice:1
Enter data:12

Want to proceed y/n
1.Add records
2.Delete records
3.Display records
4.Count no. of items in the list
5.Searching an item in the list
6.Exit
Enter your choice:1
```

```
Enter data:23

Want to proceed y/n
1.Add records
2.Delete records
3.Display records
4.Count no. of items in the list
5.Searching an item in the list
6.Exit
Enter your choice:1
Enter data:34

Want to proceed y/n
1.Add records
2.Delete records
3.Display records
4.Count no. of items in the list
5.Searching an item in the list
6.Exit
Enter your choice:3

From forward direction

 Data->12
 Data->23
 Data->34
From backward direction

 Data->34
 Data->23
 Data->12
1.Add records
2.Delete records
3.Display records
4.Count no. of items in the list
5.Searching an item in the list
6.Exit
Enter your choice:4

The size of the list is 3
1.Add records
2.Delete records
3.Display records
```

4.Count no. of items in the list
5.Searching an item in the list
6.Exit
Enter your choice:5
ter the no. to search:23

Search no. is present in the list
1.Add records
2.Delete records
3.Display records
4.Count no. of items in the list
5.Searching an item in the list
6.Exit
Enter your choice:5
ter the no. to search:56

Search no. is not present in the list
1.Add records
2.Delete records
3.Display records
4.Count no. of items in the list
5.Searching an item in the list
6.Exit
Enter your choice:2

Enter the data to delete6

Invalid input
1.Add records
2.Delete records
3.Display records
4.Count no. of items in the list
5.Searching an item in the list
6.Exit
Enter your choice:2

Enter the data to delete12

1.Add records
2.Delete records
3.Display records
4.Count no. of items in the list
5.Searching an item in the list

6.Exit
Enter your choice:3

From forward direction

Data->23
Data->34
From backward direction

Data->34
Data->23
Data->12
1.Add records
2.Delete records
3.Display records
4.Count no. of items in the list
5.Searching an item in the list
6.Exit
Enter your choice: 6

**RESULT:**

Thus the program to implement Doubly Linked List is executed and verified successfully.

## EX.NO.6 BINARY TREE AND OPERATIONS OF BINARY TREE

**AIM:**

To write a 'C' program to implement an expression tree. Produce its pre-order, in-order, and post-order traversals.

**ALGORITHM:**

Step 1: Start the process.
Step 2: Initialize and declare variables.
Step 3: Enter the choice. Inorder / Preorder / Postorder.
Step 4: If choice is Inorder then
- Traverse the left subtree in inorder.
- Process the root node.
- Traverse the right subtree in inorder.

Step 5: If choice is Preorder then
- Process the root node.
- Traverse the left subtree in preorder.
- Traverse the right subtree in preorder.

Step 6: If choice is postorder then
- Traverse the left subtree in postorder.
- Traverse the right subtree in postorder.
- Process the root node.

Step7: Print the Inorder / Preorder / Postorder traversal.
Step 8: Stop the process.

**PROGRAM**

```
#include<stdio.h>
#include<stdlib.h>
typedef struct treenode
{
    int data;
    struct treenode *left; struct treenode *right;
}tnode;

tnode *insertion(int,tnode*);
void preorder(tnode *);
void inorder(tnode *);
```

```
  void postorder(tnode *);
  void main()
  {
        tnode *T=NULL; int ch1,n;
        char ch2;
        do
        {
              printf("\n\t\t****Operation With Tree****"); printf("\n\t1.Insertion");
              printf("\n\t2.Inorder Traversal"); printf("\n\t3.Preorder Traversal");
              printf("\n\t4.Postorder Traversal"); printf("\n\tEnter Your Choice  :");
              scanf("%d",&ch1);
              switch(ch1)
              {
                case 1:
                        printf("\n\nenter the element to be inserted:");
                        scanf("%d",&n);
                        T=insertion(n,T);
                        break;
                case 2:
                        inorder(T);
                        break;
                case 3:
                        preorder(T);
                        break;
                case 4:
                        postorder(T);
                        break;
                default:
                        printf("\n\nInvalid Option"); break;
            }
            printf("\n\nDo you want to continue y/n     : "); \
            scanf("%s",&ch2);
  }while(ch2=='y');
}
tnode *insertion(int x,tnode *T)
{
  if(T==NULL)
  {
     T=(tnode *)malloc(sizeof(tnode));
     if(T==NULL)
        printf("\nout of space");
     else
     {
```

```
            T->data=x;
            T->left=T->right=NULL;
        }
    }
    else
    {
        if(x<(T->data))
            T->left=insertion(x,T->left);
        else
        {
            if(x>T->data)
                T->right=insertion(x,T->right);
        }
    }
    return T;
}
void preorder(tnode *T)
{
    if(T!=NULL)
    {
            printf("\t%d",T->data); preorder(T->left); preorder(T->right);
    }
}
void postorder(tnode *T)
{
    if(T!=NULL)
    {
            postorder(T->left);
            postorder(T->right); printf("\t%d",T->data);
    }
}
void inorder(tnode *T)
    {
            if(T!=NULL)
            {
                    inorder(T->left); printf("\t%d",T->data); inorder(T->right);
            }
    }
```

**OUTPUT:**

```
        ****Operation With Tree****
    1.Insertion
    2.Inorder Traversal
    3.Preorder Traversal
    4.Postorder Traversal
    Enter Your Choice       :1

enter the element to be inserted        :1

Do you want to continue y/n   : y

        ****Operation With Tree****
    1.Insertion
    2.Inorder Traversal
    3.Preorder Traversal
    4.Postorder Traversal
    Enter Your Choice       :1

enter the element to be inserted        :12
Do you want to continue y/n   : y

        ****Operation With Tree****
    1.Insertion
    2.Inorder Traversal
    3.Preorder Traversal
    4.Postorder Traversal
    Enter Your Choice       :1

enter the element to be inserted        :
23

Do you want to continue y/n   : y
```

```
        ****Operation With Tree****
    1.Insertion
    2.Inorder Traversal
    3.Preorder Traversal
    4.Postorder Traversal
    Enter Your Choice       :2
    1    12     23
Do you want to continue y/n  : y

        ****Operation With Tree****
    1.Insertion
    2.Inorder Traversal
    3.Preorder Traversal
    4.Postorder Traversal
    Enter Your Choice       :3
    1    12     23
Do you want to continue y/n  : y

        ****Operation With Tree****
    1.Insertion
    2.Inorder Traversal
    3.Preorder Traversal
    4.Postorder Traversal
    Enter Your Choice       : 4
    23   12     1
Do you want to continue y/n  : n
```

**RESULT:**

Thus the program to implement Binary Trees and its operations is executed and verified successfully.

**Ex. No. 7 AVL TREES**

**AIM:**
To write a C program to implement AVL trees.

**ALGORITHM:**
Step 1: Create an AVL tree with defined number of nodes.
Step 1: Perform Insertion, Deletion operations
Step 1: Perform tree traversal
Step 1:Perform AVL tree rotations

**PROGRAM:**

```
#include<stdio.h>
#include<stdlib.h>
typedef struct node
{
int data;
struct node *left,*right; int ht;
}node;

node *insert(node *,int);
node *Delete(node *,int);
void preorder(node *);
void inorder(node *);
void postorder(node *);

int height( node *);
node *rotateright(node *);
node *rotateleft(node *);
node *RR(node *);
node *LL(node *);
node *LR(node *);
node *RL(node *);
int BF(node *);

int main()
{
node *root=NULL; int x,n,i,op;

do
{
printf("\n1)Create:");
```

```
printf("\n2)Insert:");
printf("\n3)Delete:");
printf("\n4)Print:");
printf("\n5)Quit:"); printf("\n\nEnter Your Choice:"); scanf("%d",&op);

switch(op)
{
case 1: printf("\nEnter no. of elements:"); scanf("%d",&n);
printf("\nEnter tree data:"); root=NULL;
for(i=0;i<n;i++)
{
scanf("%d",&x);
root=insert(root,x);
}
break;

case 2: printf("\nEnter a data:");
scanf("%d",&x);
root=insert(root,x);
break;

case 3: printf("\nEnter a data:");
scanf("%d",&x);
root=Delete(root,x);
break;

case 4: printf("\nPreorder sequence:\n");
preorder(root);
printf("\n\nInorder sequence:\n");
inorder(root);
printf("\n Post order Sequence: \n");
postorder(root);
printf("\n");
break;
}
}while(op!=5);

return 0;
}
```

```
node * insert(node *T,int x)
{
   if(T==NULL)
   {
      T=(node*)malloc(sizeof(node));
      T->data=x;
      T->left=NULL;
      T->right=NULL;
   }
   else if(x > T->data)// insert in right subtree
   {
      T->right=insert(T->right,x);
      if(BF(T)==-2)
         if(x>T->right->data) T=RR(T);
         else
            T=RL(T);
   }
   else if(x<T->data)
   {
      T->left=insert(T->left,x);
      if(BF(T)==2)
         if(x < T->left->data)
            T=LL(T);
         else
            T=LR(T);
   }
   T->ht=height(T);
   return(T);
}

node * Delete(node *T,int x)
{
   node *p;
   if(T==NULL)
   {
     return NULL;
   }
   else if(x > T->data)// insert in right subtree
   {
      T->right=Delete(T->right,x);
      if(BF(T)==2)
         if(BF(T->left)>=0) T=LL(T);
      else
         T=LR(T);
```

```
    }
    else if(x<T->data)
    {
       T->left=Delete(T->left,x);
       if(BF(T)==-2)     //Rebalance during windup if(BF(T->right)<=0)
          if(BF(T->right)<=0)
             T=RR(T);
       else
          T=RL(T);
    }
    else
    {
       //data to be deleted is found if(T->right!=NULL)
       if(T->right!=NULL)
{            //delete its inordersuccesor p=T->right;
while(p->left!= NULL) p=p->left;
T->data=p->data;
T->right=Delete(T->right,p->data); if(BF(T)==2)//Rebalance during windup
if(BF(T->left)>=0) T=LL(T);
else
T=LR(T);
}
else
return(T->left);
}



T->ht=height(T); return(T);
}

int height(node *T)
{
int lh,rh; if(T==NULL)
return(0);

if(T->left==NULL) lh=0;
else
lh=1+T->left->ht;

if(T->right==NULL) rh=0;
else
rh=1+T->right->ht; if(lh>rh)
return(lh); return(rh);
```

```
}
node * rotateright(node *x)
{
node *y; y=x->left;
x->left=y->right; y->right=x;
x->ht=height(x); y->ht=height(y);
return(y);
}
node * rotateleft(node *x)
{
node *y; y=x->right;
x->right=y->left; y->left=x;
x->ht=height(x); y->ht=height(y);

return(y);
}

node * RR(node *T)
{
T=rotateleft(T); return(T);
}

node * LL(node *T)
{
T=rotateright(T); return(T);
}

node * LR(node *T)
{
T->left=rotateleft(T->left);
T=rotateright(T);
return(T);
}

node * RL(node *T)
{
T->right=rotateright(T->right); T=rotateleft(T);
return(T);
}
int BF(node *T)
{
int lh,rh; if(T==NULL)
return(0);
if(T->left==NULL) lh=0;
```

```
else
lh=1+T->left->ht; if(T->right==NULL)
rh=0; else
rh=1+T->right->ht; return(lh-rh);
}
void preorder(node *T)
{
 if(T!=NULL)
{
printf("%d(Bf=%d)",T->data,BF(T));
preorder(T->left);
preorder(T->right);
}
}
void inorder(node *T)
{
if(T!=NULL)
{
inorder(T->left);
printf("%d(Bf=%d)",T->data,BF(T));
inorder(T->right);
}
}
void postorder(node *T) {
    if (T != NULL) {
        postorder(T->left);
        postorder(T->right);
        printf("%d(Bf=%d) ", T->data, BF(T));
    }
}
```

**Output:**

1)Create:
2)Insert:
3)Delete:
4)Print:
5)Quit:

Enter Your Choice:1

Enter no. of elements:5

Enter tree data:12
23
34
45
56

1)Create:
2)Insert:
3)Delete:
4)Print:
5)Quit:

Enter Your Choice:2

Enter a data:67

1)Create:
2)Insert:
3)Delete:
4)Print:
5)Quit:

Enter Your Choice:4

Preorder sequence:
45(Bf=0)23(Bf=0)12(Bf=0)34(Bf=0)56(Bf=-1)67(Bf=0)

Inorder sequence:
12(Bf=0)23(Bf=0)34(Bf=0)45(Bf=0)56(Bf=-1)67(Bf=0)

Post order Sequence:
12(Bf=0) 34(Bf=0) 23(Bf=0) 67(Bf=0) 56(Bf=-1) 45(Bf=0)

1)Create:
2)Insert:
3)Delete:
4)Print:
5)Quit:

Enter Your Choice:3

Enter a data:34

1)Create:
2)Insert:
3)Delete:
4)Print:
5)Quit:

Enter Your Choice:4

Preorder sequence:
45(Bf=0)23(Bf=1)12(Bf=0)56(Bf=-1)67(Bf=0)

Inorder sequence:
12(Bf=0)23(Bf=1)45(Bf=0)56(Bf=-1)67(Bf=0)
Post order Sequence:
12(Bf=0) 23(Bf=1) 67(Bf=0) 56(Bf=-1) 45(Bf=0)

1)Create:
2)Insert:
3)Delete:
4)Print:
5)Quit:

Enter Your Choice:5

**RESULT:**

Thus the program to implement AVL Trees is executed and verified successfully.

**Ex.No.** **PRIORITY QUEUE USING HEAPS**

**AIM:**

To implement priority queue using heaps.

**ALGORITHM:**

Step 1: Start the Program
Step 2: heap is a binary tree with two important properties:

- For any node n other than the root, n.key>= n.parent.key. In other words, the parent always has more priority than its children.
- If the heap has height h, the first h−1 levels are full, and on the last level the nodes are all packed to the left.

Step 3: implement the queue as a linked list, the element with most priority will be the first element of the list, so retrieving the content as well as removing this element are both O(1) operations. However, inserting a new object in its right position requires traversing the list element by element, which is an O(n) operation.
Step 4: Insert Element in Queue
void insert (Object o, int priority) - inserts in the queue the specified object with the specified priority Algorithm
insert (Object o, int priority)
Input: An object and the corresponding priority
Output: The object is inserted in the heap with the corresponding priority
lastNode<-getLast() //get the position at which to insert
lastNode.setKey(priority)
lastnode.setContent(o)
n lastNode
while n.getParent()! = null and n.getParent().getKey() > priority
swap(n,n.getParent())
Step 5: Object DeleteMin() - removes from the queue the object with most priority
Algorithm removeMin()
lastNode<- getLast()
value lastNode.getContent()
swap(lastNode, root)
update lastNode
return value

**PROGRAM:**

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

typedef struct {
    int capacity;
    int size;
    int *elements;
} Heap;

Heap* heap_init(void) {
    int maxelements;
    printf("Enter capacity for min-heap (>= 5): ");
    if (scanf("%d", &maxelements) != 1 || maxelements < 5) {
        fprintf(stderr, "Invalid size—exiting.\n");
        exit(EXIT_FAILURE);
    }

    Heap *h = malloc(sizeof *h);
    if (!h) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    h->elements = malloc((maxelements + 1) * sizeof *(h->elements));
    if (!h->elements) {
        perror("malloc");
        free(h);
        exit(EXIT_FAILURE);
    }

    h->capacity = maxelements;
    h->size = 0;
    h->elements[0] = INT_MIN;  // sentinel for 1-based heap. Must be ≤ any insert value.
    return h;
}

int is_full(const Heap *h)  { return h->size >= h->capacity; }
int is_empty(const Heap *h) { return h->size == 0; }
```

```
void insert(int x, Heap *h) {
    if (is_full(h)) {
        fprintf(stderr, "Heap is full\n");
        return;
    }
    int i = ++h->size;
    // Percolate up until correct spot found
    while (h->elements[i / 2] > x) {
        h->elements[i] = h->elements[i / 2];
        i /= 2;
    }
    h->elements[i] = x;
}

int delete_min(Heap *h) {
    if (is_empty(h)) {
        fprintf(stderr, "Heap is empty\n");
        return INT_MIN;  /* Or handle via exit */
    }
    int ret = h->elements[1];
    int last = h->elements[h->size--];

    int i, child;
    for (i = 1; i * 2 <= h->size; i = child) {
        child = i * 2;
        if (child != h->size && h->elements[child+1] < h->elements[child])
            child++;
        if (h->elements[child] < last)
            h->elements[i] = h->elements[child];
        else
            break;
    }
    h->elements[i] = last;
    return ret;
}

void display(const Heap *h) {
    printf("Heap contents:");
    if (is_empty(h)) {
        printf(" <empty>\n");
        return;
    }
```

```
    for (int i = 1; i <= h->size; i++)
        printf(" %d", h->elements[i]);
    printf("\n");
}

void heap_free(Heap *h) {
    if (h) {
        free(h->elements);
        free(h);
    }
}

int main(void) {
    Heap *h = heap_init();
    while (1) {
        int ch;
        printf("\n1.Insert  2.DeleteMin  3.Display  4.Exit\nChoice: ");
        if (scanf("%d", &ch) != 1) break;
        switch (ch) {
            case 1: {
                int x;
                printf("Enter number: ");
                if (scanf("%d", &x) == 1) insert(x, h);
                break;
            }
            case 2: {
                int m = delete_min(h);
                if (m != INT_MIN)
                    printf("Deleted: %d\n", m);
                break;
            }
            case 3: display(h); break;
            case 4: heap_free(h); return 0;
            default: printf("Invalid choice\n");
        }
    }
    heap_free(h);
    return 0;
}
```

**OUTPUT:**

```
Enter capacity for min-heap (>= 5): 5
```

1.Insert 2.DeleteMin 3.Display 4.Exit
Choice: 1
Enter number: 23

1.Insert 2.DeleteMin 3.Display 4.Exit
Choice: 1
Enter number: 23

1.Insert 2.DeleteMin 3.Display 4.Exit
Choice: 1
Enter number: 34

1.Insert 2.DeleteMin 3.Display 4.Exit
Choice: 1
Enter number: 56

1.Insert 2.DeleteMin 3.Display 4.Exit
Choice: 3
Heap contents: 23 23 34 56

1.Insert 2.DeleteMin 3.Display 4.Exit
Choice: 2
Deleted: 23

1.Insert 2.DeleteMin 3.Display 4.Exit
Choice: 3
Heap contents: 23 56 34

1.Insert 2.DeleteMin 3.Display 4.Exit
Choice: 4

**Result:**

Thus the program to implement Priority Queue using Heaps is executed and verified successfully.

---

**Ex.No.9.a)** **DEPTH FIRST SEARCH**

**Aim:**

To write a C program for representing the graph and traversing it using DFS.

**Algorithm:**

Step 1: Start from any vertex, say $V_i$ .
Step 2: $V_i$ is visited and then all vertices adjacent to $V_i$ are traversed recursivelyusing DFS.
Step 3: Since, a graph can have cycles. We must avoid revisiting a node. To dothis, when we visit a vertex V, we mark it visited.
Step 4: A node that has already been marked as visited should not be selected for traversal.
Step 5: Marking of visited vertices can be done with the help of a global array visited[ ].
Step 6: Array visited[ ] is initialized to false (0).

**PROGRAM:**

```
#include<stdio.h>
#include<stdlib.h>

typedef struct node
{
   struct node *next;
   int vertex;
}node;

node *G[20];
//heads of linked list
int visited[20];
int n;
void read_graph();
//create adjacency list
void insert(int,int);
//insert an edge (vi,vj) in te adjacency list
void DFS(int);

void main()
{
```

```
    int i;
    read_graph();
    //initialised visited to 0
   for(i=0;i<n;i++)
       visited[i]=0;
     DFS(0);
}
 void DFS(int i)
{
    node *p;
   printf("\n%d",i);
   p=G[i];
    visited[i]=1;
    while(p!=NULL)
    {
      i=p->vertex;
      if(!visited[i])
          DFS(i);
       p=p->next;
    }
}

void read_graph()
{
    int i,vi,vj,no_of_edges;
   printf("Enter number of vertices:");
   scanf("%d",&n);
     //initialiseG[] with a null
    for(i=0;i<n;i++)
    {
       G[i]=NULL;
       //read edges and insert them in G[]
        printf("Enter number of edges:");
        scanf("%d",&no_of_edges);
           for(i=0;i<no_of_edges;i++)
       {
           printf("Enter an edge(u,v):");
           scanf("%d%d",&vi,&vj);
           insert(vi,vj);
       }
    }
}
```

```
void insert(int vi,intvj)
{
   node *p,*q;
   //acquire memory for the new node
   q=(node*)malloc(sizeof(node));
   q->vertex=vj;
   q->next=NULL;

   //insert the node in the linked list number vi
   if(G[vi]==NULL)
      G[vi]=q;
   else
   {
      //go to end of the linked list
      p=G[vi];

      while(p->next!=NULL)
         p=p->next;
      p->next=q;
   }
}
```

**Output:**

```
Enter Number of Vertices and Edges in the Graph: 6 12
Add 12 Edges of the Graph
0 0
1 1
2 2
3 3
4 4
5 5
0 1
0 2
1 3
2 4
3 2
4 5
Enter Starting Vertex for DFS Traversal: 1
DFS Traversal: 1 3 2 4 5 0
Process returned 0 (0x0)   execution time : 47.768 s
Press any key to continue.
```

**RESULT:** Thus the C program for representing the graph and traversing it using DFS has been executed and the output is verified

**Ex.No.9.b)** **BREADTH FIRST SEARCH**

**AIM:**

To write a C program for representing the graph and traversing it using BFS.

**ALGORITHM:**

Step 1: Start from any vertex, say Vi .

Step 2: Vi is visited and then all vertices adjacent to Vi are traversed recursivelyusing BFS.

Step 3: avoid revisiting a node. To do this, when we visit a vertex V, we mark it visited.

Step 4: A node that has already been marked as visited should not be selected fortraversal.

Step 5: Marking of visited vertices can be done with the help of a global arrayvisited [ ].

Step 6: Array visited [ ] is initialized to false (0).

**PROGRAM:**

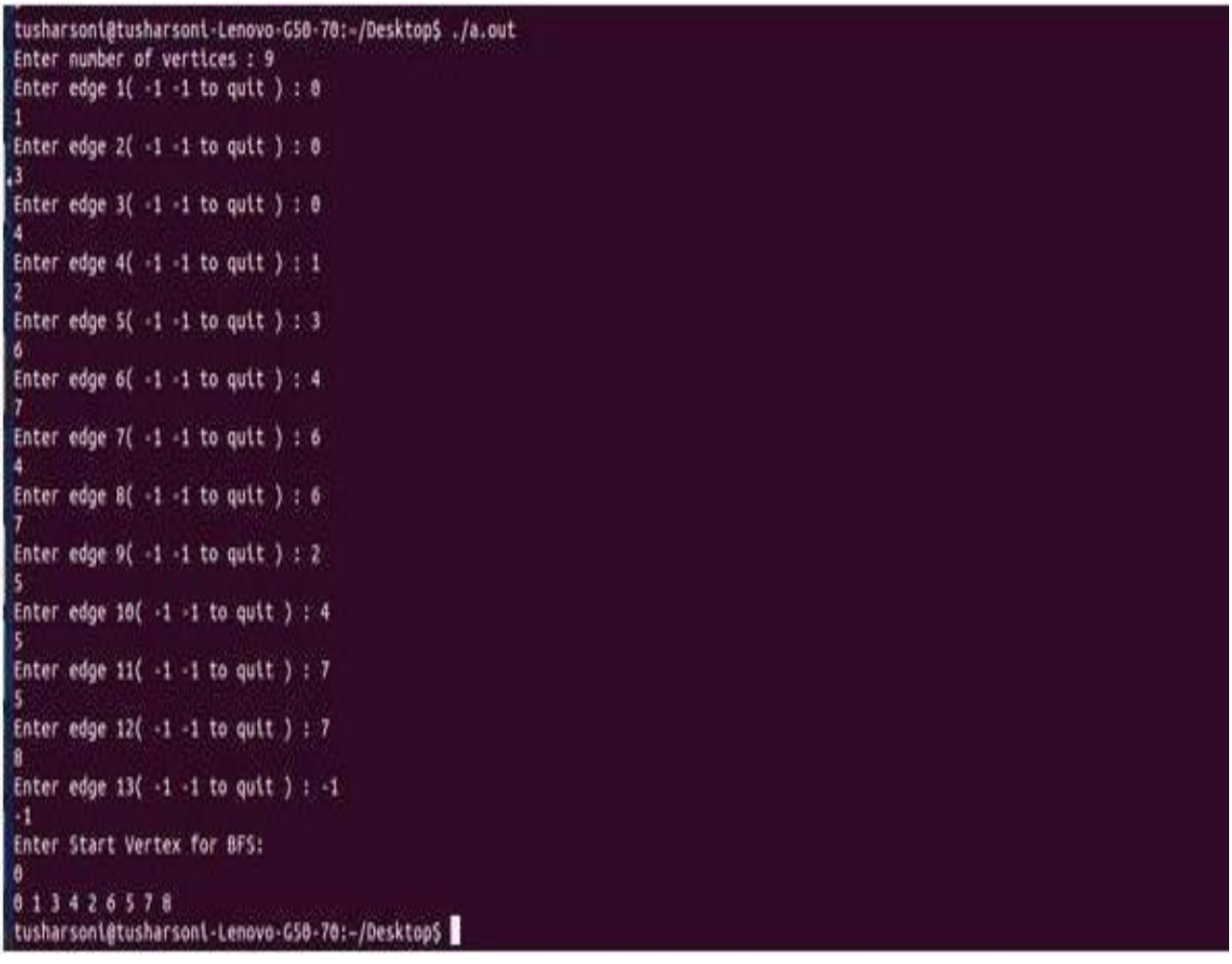
```
#include<stdio.h>
#include<stdlib.h>

#define MAX 100

#define initial 1
#define waiting 2
#define visited 3

int n;
int adj[MAX][MAX];
int state[MAX];
void create_graph();
void BF_Traversal();
void BFS(int v);

int queue[MAX], front = -1,rear = -1;
void insert_queue(int vertex);
int delete_queue();
int isEmpty_queue();
```

```
int main()
{
   create_graph();
   BF_Traversal();
   return 0;
}

void BF_Traversal()
{
   int v;

   for(v=0; v<n; v++)
      state[v] = initial;

   printf("Enter Start Vertex for BFS: \n");
   scanf("%d", &v);
   BFS(v);
}

void BFS(int v)
{
   int i;

   insert_queue(v);
   state[v] = waiting;

   while(!isEmpty_queue())
   {
      v = delete_queue( );
      printf("%d ",v);
      state[v] = visited;

      for(i=0; i<n; i++)
      {
         if(adj[v][i] == 1 && state[i] == initial)
         {
            insert_queue(i);
            state[i] = waiting;
         }
      }
   }
```

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

void insert_queue(int vertex)
{
    if(rear == MAX-1)
        printf("Queue Overflow\n");
    else
    {
        if(front == -1)
            front = 0;
        rear = rear+1;
        queue[rear] = vertex ;
    }
}

int isEmpty_queue()
{
    if(front == -1 || front > rear)
        return 1;
    else
        return 0;
}

int delete_queue()
{
    int delete_item;
    if(front == -1 || front > rear)
    {
        printf("Queue Underflow\n");
        exit(1);
    }

    delete_item = queue[front];
    front = front+1;
    return delete_item;
}

void create_graph()
{
    int count,max_edge,origin,destin;
```

```
    printf("Enter number of vertices : ");
    scanf("%d",&n);
    max_edge = n*(n-1);

    for(count=1; count<=max_edge; count++)
    {
       printf("Enter edge %d( -1 -1 to quit ) : ",count);
       scanf("%d %d",&origin,&destin);

       if((origin == -1) && (destin == -1))
          break;

       if(origin>=n || destin>=n || origin<0 || destin<0)
       {
          printf("Invalid edge!\n");
          count--;
       }
       else
       {
          adj[origin][destin] = 1;
       }
    }
}
```

OUTPUT:




**RESULT:**

Thus the C program for representing the graph and traversing it using BFS has been executed and the output is verified.

**Ex.No. 10 DIJKSTRA'S ALGORITHM**

## AIM:

To write a C program for implementing the Dijkstra's algorithm using priority heaps.

## ALGORITHM:

1. **Initialize:**
   - Set the distance of all vertices to **infinity** (∞)
   - Set the distance of the source vertex `S` to **0**
   - Create a **Min Heap (priority queue)** and insert `(S, 0)` into it
2. **While the priority queue is not empty:**
   - Extract the vertex `u` with the **minimum distance**
   - For each neighbor `v` of `u`:
     - If there is an edge from `u` to `v`
     - Calculate new distance: `newDist = distance[u] + weight(u, v)`
     - If `newDist < distance[v]`:
       - Update `distance[v] = newDist`
       - Insert or update `(v, newDist)` in the priority queue
3. **Repeat until all vertices are visited or the queue is empty**
4. **Result:**
   - The array `distance[]` contains the shortest distance from `S` to every other vertex

## PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define MAXN 1000  // adjust as needed
// Edge in adjacency list
typedef struct Edge {
    int to;
    int weight;
    struct Edge* next;
} Edge;

// Min-heap node
typedef struct {
    int vertex;
    int dist;
} HeapNode;
```

```
// Min-heap structure
typedef struct {
    HeapNode *arr;
    int size;
    int capacity;
    int *pos; // to track position of vertex in heap for decrease-key
} MinHeap;

// Utility to create a new edge
Edge* new_edge(int to, int weight) {
    Edge* e = (Edge*)malloc(sizeof(Edge));
    e->to = to;
    e->weight = weight;
    e->next = NULL;
    return e;
}

// Swap two heap nodes and update positions
void swap_heap_node(HeapNode *a, HeapNode *b) {
    HeapNode tmp = *a;
    *a = *b;
    *b = tmp;
}

// Heapify at idx
void min_heapify(MinHeap* heap, int idx) {
    int smallest = idx;
    int left = 2*idx + 1;
    int right = 2*idx + 2;

    if (left < heap->size &&
        heap->arr[left].dist < heap->arr[smallest].dist)
        smallest = left;
```

```
    if (right < heap->size &&
        heap->arr[right].dist < heap->arr[smallest].dist)
        smallest = right;

    if (smallest != idx) {
        // update positions
        heap->pos[ heap->arr[smallest].vertex ] = idx;
        heap->pos[ heap->arr[idx].vertex ] = smallest;

        swap_heap_node(&heap->arr[smallest], &heap->arr[idx]);
        min_heapify(heap, smallest);
    }
}

// Create min-heap of given capacity
MinHeap* create_min_heap(int capacity) {
    MinHeap* heap = (MinHeap*)malloc(sizeof(MinHeap));
    heap->pos = (int*)malloc(capacity * sizeof(int));
    heap->size = 0;
    heap->capacity = capacity;
    heap->arr = (HeapNode*)malloc(capacity * sizeof(HeapNode));
    return heap;
}

// Check if heap is empty
int is_empty(MinHeap* heap) {
    return heap->size == 0;
}

// Extract minimum node
HeapNode extract_min(MinHeap* heap) {
    HeapNode root = heap->arr[0];
    HeapNode last = heap->arr[heap->size - 1];
```

```
    // Move last to root
    heap->arr[0] = last;

    // Update positions
    heap->pos[root.vertex] = heap->size - 1;
    heap->pos[last.vertex] = 0;

    heap->size--;
    min_heapify(heap, 0);
    return root;
}

// Decrease distance value of a vertex
void decrease_key(MinHeap* heap, int vertex, int dist) {
    int i = heap->pos[vertex];
    heap->arr[i].dist = dist;

    // Bubble up
    while (i && heap->arr[i].dist < heap->arr[(i - 1) / 2].dist) {
        // swap positions
        heap->pos[ heap->arr[i].vertex ] = (i - 1) / 2;
        heap->pos[ heap->arr[(i - 1) / 2].vertex ] = i;
        swap_heap_node(&heap->arr[i], &heap->arr[(i - 1) / 2]);
        i = (i - 1) / 2;
    }
}

// Check if a vertex is in heap
int in_heap(MinHeap* heap, int vertex) {
    if (heap->pos[vertex] < heap->size) return 1;
    return 0;
}

// Dijkstra's algorithm
```

```
void dijkstra(Edge* graph[], int n, int src) {
    int dist[MAXN];
    MinHeap* heap = create_min_heap(n);

    for (int v = 0; v < n; ++v) {
        dist[v] = INT_MAX;
        heap->arr[v].vertex = v;
        heap->arr[v].dist = INT_MAX;
        heap->pos[v] = v;
    }

    // Initialize source
    dist[src] = 0;
    decrease_key(heap, src, 0);
    heap->size = n;

    while (!is_empty(heap)) {
        HeapNode minNode = extract_min(heap);
        int u = minNode.vertex;

        // Traverse adjacency list of u
        for (Edge* e = graph[u]; e != NULL; e = e->next) {
            int v = e->to;
            if (in_heap(heap, v) && dist[u] != INT_MAX &&
                e->weight + dist[u] < dist[v]) {
                dist[v] = dist[u] + e->weight;
                decrease_key(heap, v, dist[v]);
            }
        }
    }

    // Print results
    printf("Vertex\tDistance from Source %d\n", src);
    for (int i = 0; i < n; ++i) {
```

```
        if (dist[i] == INT_MAX)
            printf("%d\t\tINF\n", i);
        else
            printf("%d\t\t%d\n", i, dist[i]);
    }

    // Cleanup
    free(heap->arr);
    free(heap->pos);
    free(heap);
}

int main() {
    int n, m;
    printf("Enter number of vertices and edges: ");
    if (scanf("%d %d", &n, &m) != 2) return 1;
    // Allocate graph
    Edge* graph[MAXN] = {0};
    printf("Enter edges in format: u v w  (0-based vertices)\n");
    for (int i = 0; i < m; ++i) {
        int u, v, w;
        scanf("%d %d %d", &u, &v, &w);
        // add edge u -> v
        Edge* e1 = new_edge(v, w);
        e1->next = graph[u];
        graph[u] = e1;
        // If undirected, also add v -> u:
        // Edge* e2 = new_edge(u, w);
        // e2->next = graph[v];
        // graph[v] = e2;
    }
    int src;
    printf("Enter source vertex: ");
    scanf("%d", &src);
```

```
    dijkstra(graph, n, src);
    // Free edges
    for (int i = 0; i < n; ++i) {
        Edge* cur = graph[i];
        while (cur) {
            Edge* tmp = cur;
            cur = cur->next;
            free(tmp);
        }
    }
    return 0;
}
```

**OUTPUT:**

```
Enter number of vertices and edges: 5 6
Enter edges in format: u v w  (0-based vertices)
0 1 2
0 2 4
1 2 1
1 3 7
2 4 3
3 4 1
Enter source vertex: 0
Vertex                Distance from Source 0
0                      0
1                      2
2                      3
3                      9
4                      6
```

**RESULT:**

Thus the program for implementing C program for Dijkstra's algorithm has been executed and successfully verified.

## Ex.No. 11.a LINEAR SEARCH ALGORITHM

**AIM:**

To write a C program for implementation of Linear Search algorithm.

**ALGORITHM:**

1. **Start**
2. Set `i = 0`
3. Repeat steps while `i < n`:
   If `A[i] == K`, then:Return `i` (element found at index `i`), Else, set `i = i + 1`
4. If loop ends, return `-1` (element not found)
5. **Stop**

**PROGRAM:**

```
    #include <stdio.h>
int linear_search(int arr[], int n, int key) {
   for (int i = 0; i < n; ++i) {
      if (arr[i] == key)
         return i; // return index if found
   }
   return -1; // not found
}

int main() {
   int arr[] = {5, 3, 8, 4, 2, 9, 1};
   int n = sizeof(arr) / sizeof(arr[0]);
   int key = 4;

   printf("Array: ");
   for (int i = 0; i < n; ++i) {
      printf("%d ", arr[i]);
   }
   printf("\n");
   printf("Searching for: %d\n", key);

   int idx = linear_search(arr, n, key);
```

```
    if (idx != -1) {
        printf("Result: Element %d found at index %d.\n", key, idx);
    } else {
        printf("Result: Element %d not found in the array.\n", key);
    }

    return 0;
}
```

**OUTPUT:**

Array: 5 3 8 4 2 9 1
Searching for: 4
Result: Element 4 found at index 3.

**RESULT:**

Thus the C program to implement linear search algorithm has been executed and the output is verified.

## Ex.No. 11.b BINARY SEARCH ALGORITHM

**AIM:**

To write a C program for implementation of binary Search algorithm.

**ALGORITHM:**

Step 1:Input a sorted LIST of size N and Target Value T
OUTPUT: Position of T in the LIST = I
Step 2: If MAX = N and MIN = 1set FOUND = false
Step 3: WHILE (FOUND is false) and (MAX > = MIN) Set
MID = (MAX + MIN)DIV 2
If T = LIST [MID]
I=MID
FOUND = true
Step 4: Else If T < LIST[MID] Set MAX = MID-1
Step 5:Else MIN = MD+1
Step 6: END

**PROGRAM:**

```
#include <stdio.h>
 int main()
{
   int c, first, last, middle, n, search, array[100];

printf("Enter number of elements\n");
scanf("%d",&n);

printf("Enter %d integers\n", n);

   for ( c = 0 ; c < n ; c++ )
scanf("%d",&array[c]);

printf("Enter value to find\n");
scanf("%d",&search);

   first = 0;
   last = n - 1;
   middle = (first+last)/2;

while( first<= last )
```

```
    {
      if ( array[middle] < search )
        first = middle + 1;
      else if ( array[middle] == search )
      {
printf("%d found at location %d.\n", search, middle+1);
        break;
      }
      else
        last = middle - 1;

      middle = (first + last)/2;
    }
    if ( first> last )
printf("Not found! %d is not present in the list.\n", search);

    return 0;
}
```

**OUTPUT:**

Enter number of elements
5
Enter 5 integers
12
3
34
45
56
Enter value to find
34
**34 found at location 3.**

Enter number of elements
5
Enter 5 integers
23
34
78
89

12
Enter value to find
13
**Not found! 13 is not present in the list.**

**RESULT:**

Thus the C program to implement binary search algorithm has been executed and the output is verified.

## Ex. No. 12.a) MERGE SORT USING DIVIDE AND CONQUER METHOD

**AIM:**

To write a C program to implement Merge Sort.

**ALGORITHM:**

1. **Start** with an unsorted array of n elements.
2. **Divide** the array:
    - If the array has more than one element:
        - Find the middle index: mid = (low + high) / 2.
        - Split the array into two halves:
            - Left half: low to mid.
            - Right half: mid + 1 to high.
3. **Recursively sort** both halves:
    - Apply Merge Sort to the left half.
    - Apply Merge Sort to the right half.
4. **Merge the two sorted halves**:
    - Create temporary arrays for left and right halves.
    - Compare elements of both halves.
    - Copy the smaller element into the main array.
    - Continue until all elements from both halves are copied back in sorted order.
5. **End.**

**PROGRAM:**

```
#include <stdio.h>
// Function to merge two subarrays
void merge(int arr[], int left, int mid, int right) {
   int n1 = mid - left + 1;  // Size of left subarray
   int n2 = right - mid;     // Size of right subarray
   int L[n1], R[n2]; // Temporary arrays
   // Copy data to temp arrays L[] and R[]
   for (int i = 0; i < n1; i++)
      L[i] = arr[left + i];
   for (int j = 0; j < n2; j++)
      R[j] = arr[mid + 1 + j];
   // Merge the temp arrays back into arr[left..right]
   int i = 0, j = 0, k = left;
   while (i < n1 && j < n2) {
      if (L[i] <= R[j]) {
         arr[k] = L[i];
         i++;
```

```
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    // Copy remaining elements of L[], if any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    // Copy remaining elements of R[], if any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}


// Recursive Merge Sort function
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        // Recursively sort first and second halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}
// Function to print the array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
// Main function
int main() {
    int arr[100], n;
    printf("Enter number of elements: ");
    scanf("%d", &n);
```

```
    printf("Enter %d elements: ", n);
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);
    printf("Original array: ");
    printArray(arr, n);
    mergeSort(arr, 0, n - 1);
    printf("Sorted array: ");
    printArray(arr, n);
    return 0;
}
```

**Output**:

```
Enter number of elements: 6
Enter 6 elements: 38 27 43 3 9 82
Original array: 38 27 43 3 9 82
Sorted array: 3 9 27 38 43 82
```

**RESULT:**

Thus the C Program to implement Merge sort is executed and output is verified

## Ex. No. 12.b) QUICK SORT USING DIVIDE AND CONQUER METHOD

**AIM:**

To write a C program to implement Quick Sort.

**ALGORITHM:**

Step 1: Input Array
Step 2: Call QuickSort Function

Step 3: QuickSort Function Logic

Function: quickSort(arr, low, high)

If low < high:

- Call partition(arr, low, high) → returns pivotIndex.
- Recursively call quickSort(arr, low, pivotIndex - 1) to sort left part.
- Recursively call quickSort(arr, pivotIndex + 1, high) to sort right part.

Else:

- Return (base case – subarray has 0 or 1 element).

Step 4: Output the Sorted Array

**PROGRAM:**

```
#include <stdio.h>
// Function to swap two elements
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
// Partition function
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // pivot element
    int i = low - 1;        // index of smaller element
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    // place pivot at correct position
```

```
    swap(&arr[i + 1], &arr[high]);
    return (i + 1); // return pivot index
}
// QuickSort function (recursive)
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // pi is partitioning index
        int pi = partition(arr, low, high);
 // Recursively sort elements before and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
// Function to print array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
// Main function
int main() {
    int arr[100], n;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter %d elements: ", n);
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);
    printf("Original array: ");
    printArray(arr, n);
    quickSort(arr, 0, n - 1);
    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

**OUTPUT:**

```
Enter number of elements: 6
Enter 6 elements: 34 7 23 32 5 62
Original array: 34 7 23 32 5 62
Sorted array: 5 7 23 32 34 62
```

**RESULT:**

Thus the C Program to implement Quick sort is executed and output is verified

## Ex.No.13 0/1KNAP SACK PROBLEM USING GREEDY TECHNIQUE

**AIM:**

To write a C program to implement Fractional Knap sack Problem using Greedy Technique.

**ALGORITHM:**

Step 1: Input the number of items and their values and weights

Step 2: Sort items by decreasing value-to-weight ratio

Step 3: Input the capacity of the knapsack

Step 4: Initialize total value and remaining capacity

Step 5: Fill the knapsack using the sorted items

Step 6: Output the total maximum value

**PROGRAM:**

```
#include <stdio.h>
typedef struct {
   int value;
   int weight;
   double ratio;
} Item;
void sortByRatio(Item items[], int n) {
   for (int i = 0; i < n - 1; i++) {
      for (int j = i + 1; j < n; j++) {
         if (items[j].ratio > items[i].ratio) {
            Item temp = items[i];
            items[i] = items[j];
            items[j] = temp;
         }
      }
   }
```

```
}
double fractionalKnapsack(Item items[], int n, int capacity) {
    sortByRatio(items, n);
    double totalValue = 0.0;
    int remainingCapacity = capacity;
    printf("\nSorting items by value/weight ratio:\n");
    for (int i = 0; i < n; i++) {
        printf("Item %d: value = %d, weight = %d, ratio = %.2lf\n", i + 1, items[i].value, items[i].weight,
items[i].ratio);
    }
 printf("\nFilling knapsack capacity %d:\n", capacity);
    for (int i = 0; i < n; i++) {
        if (items[i].weight <= remainingCapacity) {
            // Take the whole item
            totalValue += items[i].value;
            remainingCapacity -= items[i].weight;
            printf("Taking full item %d (value: %d, weight: %d), remaining capacity: %d, total value: %.2lf\n",
                i + 1, items[i].value, items[i].weight, remainingCapacity, totalValue);
        } else {
            // Take fraction of the item
            double fraction = (double)remainingCapacity / items[i].weight;
            totalValue += items[i].value * fraction;
            printf("Taking %.2lf fraction of item %d (value: %d, weight: %d), added value: %.2lf\n",
                fraction, i + 1, items[i].value, items[i].weight, items[i].value * fraction);
            remainingCapacity = 0;
            break;
        }
    }
    return totalValue;
}
int main() {
    int n, capacity;
    printf("Enter number of items: ");
    scanf("%d", &n);
```

```
    Item items[n];
    for (int i = 0; i < n; i++) {
        printf("Enter value and weight of item %d: ", i + 1);
        scanf("%d %d", &items[i].value, &items[i].weight);
        items[i].ratio = (double)items[i].value / items[i].weight;
    }
    printf("Enter knapsack capacity: ");
    scanf("%d", &capacity);
    double maxValue = fractionalKnapsack(items, n, capacity);
    printf("\nMaximum value in the knapsack = %.2lf\n", maxValue);
    return 0;
}
```

**OUTPUT :**

Enter number of items: 3

Enter value and weight of item 1: 60 10

Enter value and weight of item 2: 100 20

Enter value and weight of item 3: 120 30

Enter knapsack capacity: 50

Sorting items by value/weight ratio:

Item 1: value = 60, weight = 10, ratio = 6.00

Item 2: value = 100, weight = 20, ratio = 5.00

Item 3: value = 120, weight = 30, ratio = 4.00

Filling knapsack capacity 50:

Taking full item 1 (value: 60, weight: 10), remaining capacity: 40, total value: 60.00

Taking full item 2 (value: 100, weight: 20), remaining capacity: 20, total value: 160.00

Taking 0.67 fraction of item 3 (value: 120, weight: 30), added value: 80.00

Maximum value in the knapsack = 240.00

**RESULTS:**

Thus the C program to execute the Fractional Knap sack Problem using Greedy Technique is implemented and output is verified

## Ex.No.14.a) FACTORIAL USING RECURION AND ANALYSIS

**AIM:**

To write a C program to implement Factorial using Recursion and perform the analysis of the program

**ALGORITHM:**

Step 1: Start

Step 2: Input the number

Step 3: Check for invalid input :**If n < 0, display a message: "Factorial is not defined for negative numbers" and** stop**.**

Step 4: Call the recursive function **factorial(n).**

Step 5: Recursive Function – **factorial(n)**

- **If** n == 0 or n == 1,
  → Return 1 (base case).
- **Else**,
  → Return n * factorial(n - 1).

**Step 6: Display the result**

- Print the value returned by factorial(n) as the factorial of the number.

Step 7: End

**PROGRAM:**

```
#include <stdio.h>
// Recursive function to calculate factorial
int factorial(int n) {
    if (n == 0 || n == 1)
        return 1; // base case
    else
        return n * factorial(n - 1); // recursive call
}
int main() {
    int num;
    printf("Enter a positive integer: ");
    scanf("%d", &num);
```

```
    if (num < 0) {
        printf("Factorial is not defined for negative numbers.\n");
    } else {
        int result = factorial(num);
        printf("Factorial of %d is %d\n", num, result);
    }
    return 0;
}
```

**OUTPUT:**

```
Enter a positive integer: 5
Factorial of 5 is 120
```

**ANALYSIS OF THE PROGRAM:**

**1. Time Complexity**

- Each function call reduces n by 1 until n == 1.
- So, for input n, the function is called n times.
- Each call does **constant work** (multiplication and recursive call).

**Time Complexity = O(n)**

2. Space Complexity

- The function uses the **call stack** for each recursive call.
- For input n, the recursion depth is n.

**Space Complexity = O(n)** (due to function call stack)

**RESULT:**

Thus the C program to implement Factorial using Recursion is executed and the analysis of the program is performed.

## Ex.No.14.b) FIBONACCI USING RECURION AND ANALYSIS

**AIM:**

To write a C program to implement Fibonacci series using Recursion and perform the analysis of the program.

**ALGORITHM:**

Step 1: Start

Step 2 Read an integer n

Step3: Define a recursive function **fibonacci(k)**

If k == 0, return 0 (base case).

*E*lse if k == 1, return 1 (base case).

Else, return fibonacci(k - 1) + fibonacci(k - 2).

Step 4: Generate and print the series

For i from 0 to n - 1, do: Call fibonacci(i) and print the result.

Step 6: End

**PROGRAM:**

```
#include <stdio.h>
// Recursive function to calculate Fibonacci number
int fibonacci(int n) {
    if (n == 0)
        return 0;       // base case
    else if (n == 1)
        return 1;       // base case
    else
        return fibonacci(n - 1) + fibonacci(n - 2); // recursive call
}
int main() {
    int n;
    printf("Enter the number of terms: ");
    scanf("%d", &n);
    if (n <= 0) {
        printf("Please enter a positive number.\n");
```

```
        return 0;
    }
    printf("Fibonacci Series up to %d terms:\n", n);
    for (int i = 0; i < n; i++) {
        printf("%d ", fibonacci(i));
    }
    printf("\n");
    return 0;
}
```

**OUTPUT:**

```
Enter the number of terms: 7
Fibonacci Series up to 7 terms:
0 1 1 2 3 5 8
```

**ANALYSIS:**

**Time Complexity**

The time complexity of the naive recursive Fibonacci function is:

$T(n) = T(n - 1) + T(n - 2) + O(1)$

This forms a binary tree of calls, resulting in exponential time complexity:

Time Complexity = $O(2^n)$

**Space Complexity**

- Each recursive call uses stack space until the base case is reached.
- The **maximum depth** of recursion is n (in the worst case).

**Space Complexity = O(n)** (due to call stack in the deepest path)

**RESULT:**

Thus the C program to implement Fibonacci series using Recursion is executed and the analysis of the program is performed