METHODIST

COLLEGE OF ENGINEERING & TECHNOLOGY

[Autonomous Institution]

Estd : 2008

Accredited by NAAC with A+ and NBA

Affiliated to Osmania University & Approved by AICTE

LABORATORY MANUAL

DATA STRUCTURES LABORATORY

BE II Semester AY 2022-23

NAME: _____

ROLL NO:_____

BRANCH:_____ **SEM:**_____

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERNG

Empower youth- Architects of Future World

VISION

To produce ethical, socially conscious and innovative professionals who would contribute to sustainable technological development of the society.

MISSION

To impart quality engineering education with latest technological developments and interdisciplinary skills to make students succeed in professional practice.

To encourage research culture among faculty and students by establishing state of art laboratories and exposing them to modern industrial and organizational practices.

To inculcate humane qualities like environmental consciousness, leadership, social values, professional ethics and engage in independent and lifelong learning for sustainable contribution to the society.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

VISION & MISSION

VISION

To become a leader in providing Computer Science & Engineering education with emphasis on knowledge and innovation.

MISSION

- To offer flexible programs of study with collaborations to suit industry needs.
- To provide quality education and training through novel pedagogical practices.
- To expedite high performance of excellence in teaching, research and innovations.
- To impart moral, ethical values and education with social responsibility.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PROGRAM EDUCATIONAL OBJECTIVES

After 3-5 years of graduation, the graduates will be able to

PEO1: Apply technical concepts, Analyze, Synthesize data to Design and create novel products and solutions for the real life problems.

PEO2: Apply the knowledge of Computer Science Engineering to pursue higher education with due consideration to environment and society.

PEO3: Promote collaborative learning and spirit of team work through multidisciplinary projects

PEO4: Engage in life-long learning and develop entrepreneurial skills.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PROGRAM OUTCOMES

Engineering graduates will be able to:

PO1: Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2: 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.

PO3: 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.

PO4: 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.

PO5: Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.

PO6: 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.

PO7: 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.

PO8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9: Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10: 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.

PO11: 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.

PO12: 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.

PROGRAM SPECIFIC OUTCOMES

At the end of 4 years, Computer Science and Engineering graduates at MCET will be able to:

PSO1: Apply the knowledge of Computer Science and Engineering in various domains like networking and data mining to manage projects in multidisciplinary environments.

PSO2: Develop software applications with open-ended programming environments**.**

PSO3: Design and develop solutions by following standard software engineering principles and implement by using suitable programming languages and platforms

Data Structures Lab Syllabus

Course code	Course Title	Core/ Elective					
ES252CS	Data Structures lab	Core					
		L	T	P/D	Credits	CIE	SEE
		0	0	2	1	40	60

Prerequisite: Programming and Problem Solving

Course Objectives: The objective of this course is to make the student

- Design and construct simple programs by using the concepts of structures as abstract data type.
- To have a broad idea about how to use pointers to implement of data structures.
- To enhance programming skills while improving their practical knowledge in data structures.
- To strengthen the practical ability to apply suitable data structure for real time application

Course Outcomes : After completion of the course, the student will be able to

- Implement linear data structures such as single Linked list, double linked list, stacks, queues using array
- Understand and implements non-linear data structures such as trees, graphs.
- Understanding and implementing hashing techniques.
- Implement various kinds of searching, sorting and traversal techniques and know when to choose which technique

Programming Exercise:

1. Implementation of Stacks, Queues ADT using arrays
2. Implementation of Stacks, Queues ADT using linked lists.
3. Implementation of Singly Linked List, Doubly Linked List and Circular List ADT.
4. Implementation of stack and use it to convert infix to postfix expression and postfix evaluation
5. Implementation of Binary search tree and its operations (creation, traversal, min & max, search)
6. Implementation of operations on AVL trees ADT
7. Implementation of Linear search and Binary Search
8. Implementation of Hashing collision resolution techniques.
9. Implementation of Insertion Sort, Selection Sort
10. Implementation of Merge Sort, Quick Sort
11. Implementation of Heap Sort.
12. Implementation of DFS and BFS

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Course Outcomes (CO's):

SUBJECT NAME: DATA STRUCTURES LAB

CODE: 3ES252CS

SEMESTER: II

CO No.	Course Outcome	Taxonomy Level
ES252CS.1	Understand and implement the abstract data type and reusability of a particular data structure.	Remembering
ES252CS.2	Implement linear data structures such as stacks, queues using array and linked list.	Understanding
ES252CS.3	Understand and implements non-linear data structures such as trees, graphs.	Evaluating
ES252CS.4	Implement various kinds of searching, sorting and traversal techniques and know when to choose which technique.	Creating
ES252CS.5	Understanding and implementing hashing techniques.	Analysing

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

GENERAL LABORATORY INSTRUCTIONS

1. Students are advised to come to the laboratory at least 5 minutes before (to 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 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.
    c. Formal 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, misbehaviours 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 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**

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CODE OF CONDUCT FOR THE LABORATORY

- All students must observe the dress code while in the laboratory
- Footwear is NOT allowed
- Foods, drinks and smoking are NOT allowed
- All bags must be left at the indicated place
- The lab timetable must be strictly followed
- Be PUNCTUAL for your laboratory session
- All programs must be completed within the given time
- Noise must be kept to a minimum
- Workspace must be kept clean and tidy at all time
- All students are liable for any damage to system due to their own negligence
- Students are strictly PROHIBITED from taking out any items from the laboratory
- Report immediately to the lab programmer if any damages to equipment

BEFORE LEAVING LAB:

- Arrange all the equipment and chairs properly.
- Turn off / shut down the systems before leaving.
- Please check the laboratory notice board regularly for updates.

Lab In – charge

METHODIST
COLLEGE OF ENGINEERING & TECHNOLOGY
Estd : 2008 Accredited by NAAC with A+ and NBA Affiliated to Osmania University & Approved by AICTE
AUTONOMOUS INSTITUTION
CSE Department

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

LIST OF EXPERIMENTS

S. No.	Name of Experiment	Date of Experiment	Date of Submission	Page No.	Faculty Signature
1	Implementation of Stacks, Queues ADT using arrays				
2	Implementation of Stacks, Queues ADT using linked lists.				
3	Implementation of Singly Linked List, Doubly Linked List and Circular List ADT.				
4	Implementation of stack and use it to convert infix to postfix expression and postfix evaluation				
5	Implementation of Binary search tree and its operations (creation, traversal, min & max, search)				
6	Implementation of operations on AVL trees ADT				
7	Implementation of Linear search and Binary Search				
8	Implementation of Hashing collision resolution techniques.				
9	Implementation of Insertion Sort, Selection Sort				
10	Implementation of Merge Sort, Quick Sort				
11	Implementation of Heap Sort.				
12	Implementation of DFS and BFS				

ADDITIONAL EXPERIMENTS

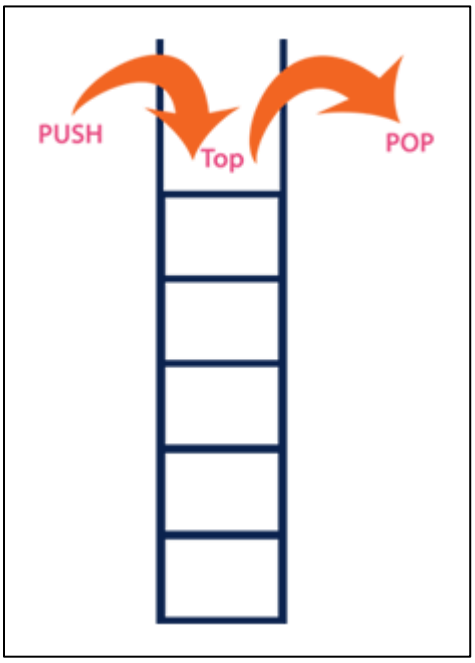
S. No.	Name of Experiment	Date of Experiment	Date of Submission	Page No.	Faculty Signature
13	Program for Tower of Hanoi using Recursion				
14	Implementation Circular DLL				

Program No. 1: Implementation of Stacks, Queues ADT using Arrays.

a) Program to implement stack using arrays

Description:

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.



Algorithm for Stack Operations using Array

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.

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**).

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--**).

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

Program:

```
#include<stdio.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){
    printf("\n\n***** MENU *****\n");
    printf("1. Push\n2. Pop\n3. Display\n4. Exit");
    printf("\nEnter your choice: ");
    scanf("%d",&choice);
    switch(choice){
        case 1: printf("Enter the value to be insert: ");
                scanf("%d",&value);
                push(value);
                break;
        case 2: pop();
                break;
        case 3: display();
                break;
        case 4: exit(0);
        default: printf("\nWrong selection!!! Try again!!!");
    }
  }
}
void push(int value){
  if(top == SIZE-1)
    printf("\nStack is Full!!! Insertion is not possible!!!");
  else{
    top++;
    stack[top] = value;
    printf("\nInsertion success!!!");
  }
}
```

```
void pop(){
  if(top == -1)
    printf("\nStack is Empty!!! Deletion is not possible!!!");
  else{
    printf("\nDeleted : %d", stack[top]);
    top--;
  }
}
void display(){
  if(top == -1)
    printf("\nStack is Empty!!!");
  else{
    int i;
    printf("\nStack elements are:\n");
    for(i=top; i>=0; i--)
        printf("%d\n",stack[i]);
  }
}
```

Output:

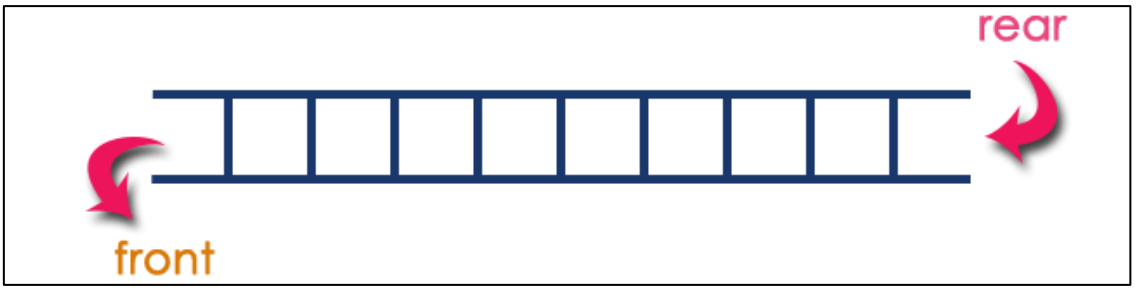
```
***** MENU *****
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 10
Insertion success!!!

***** MENU *****
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
Stack elements are:
10


***** MENU *****
1. Push
2. Pop
3. Display
4. Exit
```

```
Enter your choice: 2
Deleted : 10

***** MENU *****
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
Stack is Empty!!!
```

b) Program to implement Queue using arrays

Description:

Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends. In a queue data structure, adding and removing elements are performed at two different positions. The insertion is performed at one end and deletion is performed at another end. In a queue data structure, the insertion operation is performed at a position which is known as 'rear' and the deletion operation is performed at a position which is known as 'front'. In queue data structure, the insertion and deletion operations are performed based on FIFO (First In First Out) principle.



Algorithm for Queue Operations using Array

Queue data structure using array can be implemented as follows...

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.

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**.

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**).

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**)

Program:

```
#include<stdio.h>
#define n 5
```

```
int main()
{
 int queue[n],ch=1,front=0,rear=0,i,x=n;
 do
 {
 printf("1.enqueue\n2.dequeue\n3.display\n4.Exit\n");
 printf("Enter Your Choice:");
 scanf("%d",&ch);
 switch(ch)
 {
 case 1:if(rear==x)
 {
 printf("Queue is full\n");
 }
 else{
 printf("Enter Element:");
 scanf("%d",&queue[rear++]);
 }
 printf("Element is inserted\n");
 break;
 case 2:if(front==rear)
 {
 printf("Queue is empty\n");
 }
 else{
 printf("%d is deleted\n",queue[front++]);
 x++;
 }
 break;
 case 3:printf("Queue elements are:\n");
 if(front==rear)
 {
 printf("Queue is Empty\n");
 }
 else{
 for(i=front;i<rear;i++)
 {
 printf("%d |",queue[i]);
 }
 }
 printf("\n");
 break;
 case 4:printf("Exit");
 break;
 default:printf("Enter valid Choice>>\n");
```

```
 }
 }while(ch!=4);
 return 0;
}
```

Output:

```
/tmp/wkM3sxjsVE.o
1.enqueue
2.dequeue
3.display
4.Exit
Enter Your Choice:1
Enter Element:10
Element is inserted
1.enqueue
2.dequeue
3.display
4.Exit
Enter Your Choice:1
Enter Element:20
Element is inserted
1.enqueue
2.dequeue
3.display
4.Exit
Enter Your Choice:3
Queue elements are:
10 |20 |
1.enqueue
2.dequeue
3.display
4.Exit
Enter Your Choice:2
10 is deleted
1.enqueue
2.dequeue
3.display
4.Exit
Enter Your Choice:3
Queue elements are:
20 |
1.enqueue
2.dequeue
3.display
```

```
4.Exit
Enter Your Choice:
```

Program No. 2: Implementation of Stacks, Queues ADT using linked lists.

a) Stacks using linked lists.

Description:

The major problem with the stack implemented using an array 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.

Algorithm:

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.

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**.

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)**).

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**'.

Program:

```
#include<stdio.h>
struct Node
{
  int data;
  struct Node *next;
}*top = NULL;

void push(int);
void pop();
void display();

void main()
{
  int choice, value;
  printf("\n:: Stack using Linked List ::\n");
  while(1){
    printf("\n****** MENU ******\n");
    printf("1. Push\n2. Pop\n3. Display\n4. Exit\n");
    printf("Enter your choice: ");
    scanf("%d",&choice);
    switch(choice){
        case 1: printf("Enter the value to be insert: ");
                scanf("%d", &value);
```

```
                    push(value);
                    break;
            case 2: pop(); break;
            case 3: display(); break;
            case 4: exit(0);
            default: printf("\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;
    printf("\nInsertion is Success!!!\n");
}
void pop()
{
    if(top == NULL)
        printf("\nStack is Empty!!!\n");
    else{
        struct Node *temp = top;
        printf("\nDeleted element: %d", temp->data);
        top = temp->next;
        free(temp);
    }
}
void display()
{
    if(top == NULL)
        printf("\nStack is Empty!!!\n");
    else{
        struct Node *temp = top;
        while(temp->next != NULL){
            printf("%d--->",temp->data);
            temp = temp -> next;
        }
        printf("%d--->NULL",temp->data);
    }
}
```

Output:

```
/tmp/wkM3sxjsVE.o
:: Stack using Linked List ::

****** MENU ******
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 10
Insertion is Success!!!

****** MENU ******
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 20
Insertion is Success!!!

****** MENU ******
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 2
Deleted element: 20
****** MENU ******
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
10--->NULL
****** MENU ******
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 2
Deleted element: 10
```

```
****** MENU ******
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
Stack is Empty!!!

****** MENU ******
1. Push
2. Pop
3. Display
4. Exit
Enter your choice:
```

b) Queue using linked lists.

Description:

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'.

Algorithm:

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.

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**.

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)**).

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**'.

Program:

```
#include<stdio.h>
struct Node
{
  int data;
  struct Node *next;
}*front = NULL,*rear = NULL;

void insert(int);
void delete();
void display();

void main()
{
  int choice, value;
  printf("\n:: Queue Implementation using Linked List ::\n");
  while(1){
    printf("\n****** MENU ******\n");
    printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
    printf("Enter your choice: ");
```

```
    scanf("%d",&choice);
    switch(choice){
        case 1: printf("Enter the value to be insert: ");
                scanf("%d", &value);
                insert(value);
                break;
        case 2: delete(); break;
        case 3: display(); break;
        case 4: exit(0);
        default: printf("\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;
  }
  printf("\nInsertion is Success!!!\n");
}
void delete()
{
  if(front == NULL)
    printf("\nQueue is Empty!!!\n");
  else{
    struct Node *temp = front;
    front = front -> next;
    printf("\nDeleted element: %d\n", temp->data);
    free(temp);
  }
}
void display()
{
  if(front == NULL)
    printf("\nQueue is Empty!!!\n");
  else{
    struct Node *temp = front;
    while(temp->next != NULL){
```

```
            printf("%d--->",temp->data);
            temp = temp -> next;
        }
        printf("%d--->NULL\n",temp->data);
    }
}
```

Output:

```
:: Queue Implementation using Linked List ::

****** MENU ******
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 20
Insertion is Success!!!

****** MENU ******
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3
20--->NULL

****** MENU ******
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 2
Deleted element: 20

****** MENU ******
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3
Queue is Empty!!!

****** MENU ******
```

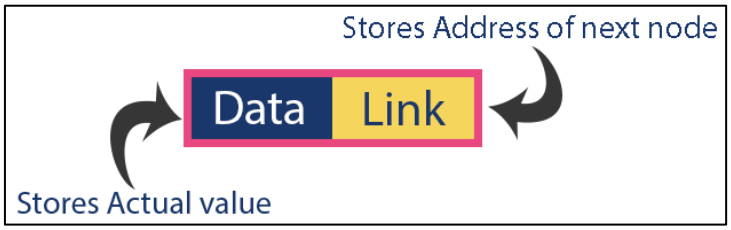
```
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice:
```

Program No. 3: Implementation of Singly Linked List, Doubly Linked List and Circular List ADT.

a) Single Linked List

Description:

Single linked list is a sequence of elements in which every element has link to its next element in the sequence. 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.



Algorithm:

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

- **Step 1 -** Include all the **header files** which are used in the program.
- **Step 2 -** Declare all the **user defined functions**.
- **Step 3 -** Define a **Node** structure with two members **data** and **next**
- **Step 4 -** Define a Node pointer **'head'** and set it to **NULL**.
- **Step 5 -** Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Insertion

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

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

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**.

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**.

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**'

Deletion

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

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

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**.

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**.

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)**).

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**).

Program:

```
#include<stdio.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;
  while(1){
  mainMenu: printf("\n\n****** MENU ******\n1. Insert\n2. Display\n3. Delete\n4. Exit\nEnter your choice: ");
  scanf("%d",&choice);
  switch(choice)
  {
    case 1:     printf("Enter the value to be insert: ");
                scanf("%d",&value);
                while(1){
                printf("Where you want to insert: \n1. At Beginning\n2. At End\n3. Between\nEnter your choice: ");
                scanf("%d",&choice1);
                switch(choice1)
                {
                  case 1:       insertAtBeginning(value);
                                break;
                  case 2:       insertAtEnd(value);
                                break;
                  case 3:     printf("Enter the two values where you wanto insert: ");
                                scanf("%d%d",&loc1,&loc2);
                                insertBetween(value,loc1,loc2);
                                break;
                  default:      printf("\nWrong Input!! Try again!!!\n\n");
                                goto mainMenu;
                }
                goto subMenuEnd;
                }
                subMenuEnd:
                break;
    case 2:     display();
                break;
    case 3:     printf("How do you want to Delete: \n1. From Beginning\n2. From End\n3. Spesific\nEnter your choice: ");
                scanf("%d",&choice1);
                switch(choice1)
                {
                  case 1:       removeBeginning();
                                break;
                  case 2:       removeEnd();
                                break;
                  case 3:     printf("Enter the value which you wanto delete: ");
                                scanf("%d",&loc2);
                                removeSpecific(loc2);
```

```
                                    break;
                        default:    printf("\nWrong Input!! Try again!!!\n\n");
                                    goto mainMenu;
                    }
                    break;
        case 4:     exit(0);
        default: printf("\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;
    }
    printf("\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;
    }
    printf("\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;
  }
  printf("\nOne node inserted!!!\n");
}

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

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

```
    struct Node *temp = head;
    printf("\n\nList elements are - \n");
    while(temp->next != NULL)
    {
        printf("%d --->",temp->data);
        temp = temp->next;
    }
    printf("%d --->NULL",temp->data);
  }
}
```

Output:

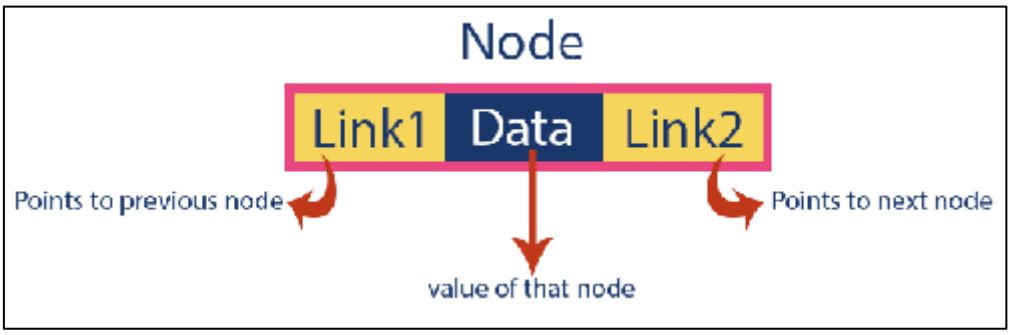
```
/tmp/wkM3sxjsVE.o
****** MENU ******
1. Insert
2. Display
3. Delete
4. Exit
Enter your choice: 1
Enter the value to be insert: 10
Where you want to insert:
1. At Beginning
2. At End
3. Between
Enter your choice: 1
One node inserted!!!

****** MENU ******
1. Insert
2. Display
3. Delete
4. Exit
Enter your choice: 1
Enter the value to be insert: 20
Where you want to insert:
1. At Beginning
2. At End
3. Between
Enter your choice: 1
One node inserted!!!

****** MENU ******
1. Insert
2. Display
```

```
3. Delete
4. Exit
Enter your choice: 2
List elements are -
20 --->10 --->NULL

****** MENU ******
1. Insert
2. Display
3. Delete
4. Exit
Enter your choice: 3
How do you want to Delete:
1. From Beginning
2. From End
3. Spesific
Enter your choice: 2
One node deleted!!!

****** MENU ******
1. Insert
2. Display
3. Delete
4. Exit
Enter your choice: 2
List elements are -
20 --->NULL

****** MENU ******
1. Insert
2. Display
3. Delete
4. Exit
Enter your choice:
```

b) Double Linked List

Description:

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.



Algorithm:

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

1. Insertion
2. Deletion
3. Display

Insertion

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

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

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**.

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**.

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.
- **Step 7 -** Assign **temp1 → next** to **temp2**, **newNode** to **temp1 → next**, **temp1** to **newNode → previous**, **temp2** to **newNode → next** and **newNode** to **temp2 → previous**.

Deletion

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

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

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**.

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**.

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)**).

Displaying a Double Linked List

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**).

Program:

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

// node creation
struct Node {
  int data;
  struct Node* next;
  struct Node* prev;
};

// insert node at the front
void insertFront(struct Node** head, int data) {
  // allocate memory for newNode
  struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

  // assign data to newNode
  newNode->data = data;

  // make newNode as a head
  newNode->next = (*head);

  // assign null to prev
  newNode->prev = NULL;

  // previous of head (now head is the second node) is newNode
  if ((*head) != NULL)
    (*head)->prev = newNode;

  // head points to newNode
  (*head) = newNode;
```

```
}

// insert a node after a specific node
void insertAfter(struct Node* prev_node, int data) {
  // check if previous node is null
  if (prev_node == NULL) {
    printf("previous node cannot be null");
    return;
  }

  // allocate memory for newNode
  struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

  // assign data to newNode
  newNode->data = data;

  // set next of newNode to next of prev node
  newNode->next = prev_node->next;

  // set next of prev node to newNode
  prev_node->next = newNode;

  // set prev of newNode to the previous node
  newNode->prev = prev_node;

  // set prev of newNode's next to newNode
  if (newNode->next != NULL)
    newNode->next->prev = newNode;
}

// insert a newNode at the end of the list
void insertEnd(struct Node** head, int data) {
  // allocate memory for node
  struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

  // assign data to newNode
  newNode->data = data;

  // assign null to next of newNode
  newNode->next = NULL;

  // store the head node temporarily (for later use)
  struct Node* temp = *head;

  // if the linked list is empty, make the newNode as head node
```

```
  if (*head == NULL) {
    newNode->prev = NULL;
    *head = newNode;
    return;
  }

  // if the linked list is not empty, traverse to the end of the linked list
  while (temp->next != NULL)
    temp = temp->next;

  // now, the last node of the linked list is temp

  // assign next of the last node (temp) to newNode
  temp->next = newNode;

  // assign prev of newNode to temp
  newNode->prev = temp;
}

// delete a node from the doubly linked list
void deleteNode(struct Node** head, struct Node* del_node) {
  // if head or del is null, deletion is not possible
  if (*head == NULL || del_node == NULL)
    return;

  // if del_node is the head node, point the head pointer to the next of del_node
  if (*head == del_node)
    *head = del_node->next;

  // if del_node is not at the last node, point the prev of node next to del_node to the previous
of del_node
  if (del_node->next != NULL)
    del_node->next->prev = del_node->prev;

  // if del_node is not the first node, point the next of the previous node to the next node of
del_node
  if (del_node->prev != NULL)
    del_node->prev->next = del_node->next;

  // free the memory of del_node
  free(del_node);
}

// print the doubly linked list
void displayList(struct Node* node) {
```

```
  struct Node* last;

  while (node != NULL) {
    printf("%d->", node->data);
    last = node;
    node = node->next;
  }
  if (node == NULL)
    printf("NULL\n");
}

int main() {
  // initialize an empty node
  struct Node* head = NULL;

  insertEnd(&head, 5);
  insertFront(&head, 1);
  insertFront(&head, 6);
  insertEnd(&head, 9);

  // insert 11 after head
  insertAfter(head, 11);

  // insert 15 after the seond node
  insertAfter(head->next, 15);

  displayList(head);

  // delete the last node
  deleteNode(&head, head->next->next->next->next->next);

  displayList(head);
}
```

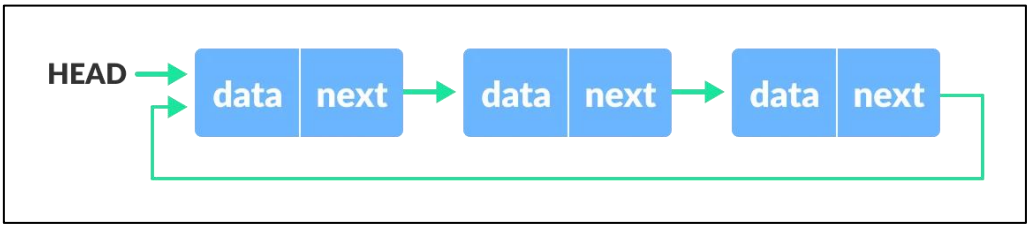
Output:

```
6->11->15->1->5->9->NULL
6->11->15->1->5->NULL
```

c) Circular Linked List

Description:

A circular linked list is a variation of a linked list in which the last element is linked to the first element. This forms a circular loop.



Algorithm:

We can do operations on the circular linked list similar to the singly linked list which are:

Insertion

A node can be added in three ways:

1. Insertion at the beginning of the list
2. Insertion at the end of the list
3. Insertion in between the nodes

Deletion

In a circular linked list, the deletion operation can be performed in three ways:

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Program:

```
#include<stdio.h>
#include<stdlib.h>
struct cll
{
int x;
struct cll *next;
};

int main()
{
int n,i;
struct cll *head,*temp,*new;
head=malloc(sizeof (struct cll));
printf("enter no.of nodes");
scanf("%d",&n);
```

```
for(i=1;i<n;i++)
{
if(head->x==NULL)
{
    printf("enter data:");
    scanf("%d",&head->x);
    head->next=NULL;
    temp=head;
}
new=malloc(sizeof (struct cll));
printf("enter data: ");
scanf("%d",&new->x);
temp->next=new;
temp=new;
new->next=head;
}
temp=head;
printf("Nodes Inserted are:\n");
do
{
printf("%d\n",temp->x);
temp=temp->next;
}while(temp!=head);
printf("%d",temp->x);

return 0;
}
```

Output

```
/tmp/srmrA9Vr7N.o
enter no.of nodes5
enter data: 10
enter data: 20
enter data: 30
enter data: 40
enter data: 50

Nodes Inserted are:

10
20
30
40
50
```

10

Program No. 4: Implementation of stack and use it to convert infix to postfix expression and postfix evaluation

a)Infix to Postfix

Description:

An expression is a collection of operators and operands that represents a specific value. Based on the operator position, expressions are divided into THREE types. They are as follows...

1. Infix Expression: In infix expression, operator is used in between the operands.
2. Postfix Expression: In postfix expression, operator is used after operands.
3. Prefix Expression: In prefix expression, operator is used before operands.

Algorithm:

Step 0. Tokenize the infix expression. i.e Store each element i.e (operator / operand / parentheses) of an infix expression into a list / queue.
Step 1. Push "(" onto a stack and append ")" to the tokenized infix expression list / queue.
Step 2. For each element (operator / operand / parentheses) of the tokenized infix expression stored in the list/queue repeat steps 3 up to 6.
Step 3. If the token equals "(", push it onto the top of the stack.
Step 4. If the token equals ")", pop out all the operators from the stack and append them to the postfix expression till an opening bracket i.e "(" is found.
Step 5. If the token equals "*" or "/" or "+" or "-" or "^", pop out operators with higher precedence at the top of the stack and append them to the postfix expression. Push current token onto the stack.
Step 6. If the token is an operand, append it to the postfix expression. (Positions of the operands do not change in the postfix expression so append an operand as it is.)

Program:

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

char stack[100];
int top = -1;

void push(char x)
{
   stack[++top] = x;
}
```

```
char pop()
{
   if(top == -1)
      return -1;
   else
      return stack[top--];
}

int priority(char x)
{
   if(x == '(')
      return 0;
   if(x == '+' || x == '-')
      return 1;
   if(x == '*' || x == '/')
      return 2;
   return 0;
}

void main()
{
   char exp[100];
   char x;
   printf("Enter the expression : ");
   scanf("%s",exp);
   printf("\n");
   int l=0;
   while(exp[l] != '\0')
   {
      if(isalnum(exp[l]))
         printf("%c ",exp[l]);
      else if(exp[l] == '(')
         push(exp[l]);
      else if(exp[l] == ')')
      {
         while((x = pop()) != '(')
            printf("%c ", x);
      }
      else
      {
         while(priority(stack[top]) >= priority(exp[l]))
            printf("%c ",pop());
         push(exp[l]);
      }
```

```
        l++;
    }

    while(top != -1)
    {
        printf("%c ",pop());
    }
}
```

Output:

```
Enter the expression : (a+b)/(c-d)
a b + c d - /
```

b) Postfix Evaluation

Algorithm:

A postfix expression can be evaluated using the Stack data structure. To evaluate a postfix expression using Stack data structure we can use the following steps...

1. Read all the symbols one by one from left to right in the given Postfix Expression
2. If the reading symbol is operand, then push it on to the Stack.
3. If the reading symbol is operator (+ , - , * , / etc.,), then perform TWO pop operations and store the two popped oparands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.
4. Finally! perform a pop operation and display the popped value as final result.

Program:

```
#include<stdio.h>
#include<ctype.h>
char postfix[20];
int stack[20];
int top = -1;

void push(float ch)
{
    top++;
    stack[top]=ch;
}
int pop()
{
    int x;
```

```
    x=stack[top];
    top--;
    return x;
}
void main()
{
     int op1,op2;
    int i;
    printf("Enter the expression :");
    scanf("%s",postfix);
    for(i=0;postfix[i]!='\0';i++)
    {
        if(isdigit(postfix[i]))
        {
            push(postfix[i]-48);
        }
        else
        {
            op2= pop();
            op1= pop();
            switch(postfix[i])
            {
            case '+':
            {
                push(op1+op2);
                break;
            }
            case '-':
            {
                push(op1-op2);
                break;
            }
            case '*':
            {
                push(op1*op2);
                break;
            }
            case '/':
            {
              if(op2==0)
              printf("Divide by Zero.");
              else
                  push(op1/op2);
               break;
            case '%':
```

```
            {
               push(op1%op2);
               break;
            }
            default: printf("wrong operator");
            }
            }

         }
      }
      printf("\nThe result of expression %s  =  %d\n\n",postfix,stack[top]);
}
```

Output:

```
Enter the expression :26+76-/
The result of expression 26+76-/  =  8
```

Program No. 5: Implementation of Binary search tree and its operations (creation, traversal, min & max, search)

Description:

Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.

Algorithm:

The following operations are performed on a binary search tree...

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

Search Operation in BST

In a binary search tree, the search operation is performed with **O(log n)** time complexity. The search operation is performed as follows...

- **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.

Insertion Operation in BST

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

- **Step 1 -** Create a newNode with given value and set its **left** and **right** to **NULL**.
- **Step 2 -** Check whether tree is Empty.
- **Step 3 -** If the tree is **Empty**, then set **root** to **newNode**.
- **Step 4 -** If the tree is **Not Empty**, then check whether the value of newNode is **smaller** or **larger** than the node (here it is root node).
- **Step 5 -** If newNode is **smaller** than **or equal** to the node then move to its **left** child. If newNode is **larger** than the node then move to its **right** child.
- **Step 6-** Repeat the above steps until we reach to the **leaf** node (i.e., reaches to NULL).
- **Step 7 -** After reaching the leaf node, insert the newNode as **left child** if the newNode is **smaller or equal** to that leaf node or else insert it as **right child**.

Deletion Operation in BST

In a binary search tree, the deletion operation is performed with **O(log n)** time complexity. Deleting a node from Binary search tree includes following three cases...

- **Case 1: Deleting a Leaf node (A node with no children)**
- **Case 2: Deleting a node with one child**
- **Case 3: Deleting a node with two children**

Case 1: Deleting a leaf node

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

- **Step 1 - Find** the node to be deleted using **search operation**
- **Step 2 -** Delete the node using **free** function (If it is a leaf) and terminate the function.

Case 2: Deleting a node with one child

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

- **Step 1 - Find** the node to be deleted using **search operation**
- **Step 2 -** If it has only one child then create a link between its parent node and child node.
- **Step 3 -** Delete the node using **free** function and terminate the function.

Case 3: Deleting a node with two children

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

- **Step 1 - Find** the node to be deleted using **search operation**
- **Step 2 -** If it has two children, then find the **largest** node in its **left subtree** (OR) the **smallest** node in its **right subtree**.
- **Step 3 - Swap** both **deleting node** and node which is found in the above step.
- **Step 4 -** Then check whether deleting node came to **case 1** or **case 2** or else goto step 2
- **Step 5 -** If it comes to **case 1**, then delete using case 1 logic.
- **Step 6-** If it comes to **case 2**, then delete using case 2 logic.
- **Step 7 -** Repeat the same process until the node is deleted from the tree.

Program:

```
#include <stdio.h>

// storing the maximum number of nodes
int max_node = 15;

// array to store the tree
char tree[] = {'\0', 'D', 'A', 'F', 'E', 'B', 'R', 'T', 'G', 'Q', '\0', '\0', 'V', '\0', 'J', 'L'};

int get_right_child(int index) {
  // node is not null
  // and the result must lie within the number of nodes for a complete binary tree
  if (tree[index] != '\0' && ((2 * index) + 1) <= max_node)
    return (2 * index) + 1;
  // right child doesn't exist
  return -1;
}

int get_left_child(int index) {
  // node is not null
  // and the result must lie within the number of nodes for a complete binary tree
  if (tree[index] != '\0' && (2 * index) <= max_node)
    return 2 * index;
  // left child doesn't exist
```

```
  return -1;
}

void preorder(int index) {
  // checking for valid index and null node
  if (index > 0 && tree[index] != '\0') {
    printf(" %c ", tree[index]); // visiting root
    preorder(get_left_child(index)); //visiting left subtree
    preorder(get_right_child(index)); //visiting right subtree
  }
}

void postorder(int index) {
  // checking for valid index and null node
  if (index > 0 && tree[index] != '\0') {
    postorder(get_left_child(index)); //visiting left subtree
    postorder(get_right_child(index)); //visiting right subtree
    printf(" %c ", tree[index]); //visiting root
  }
}

void inorder(int index) {
  // checking for valid index and null node
  if (index > 0 && tree[index] != '\0') {
    inorder(get_left_child(index)); //visiting left subtree
    printf(" %c ", tree[index]); //visiting root
    inorder(get_right_child(index)); // visiting right subtree
  }
}

int main() {
  printf("Preorder:\n");
  preorder(1);
  printf("\nPostorder:\n");
  postorder(1);
  printf("\nInorder:\n");
  inorder(1);
  printf("\n");
  return 0;
}
```

Output:

Preorder:
D A E G Q B F R V T J L

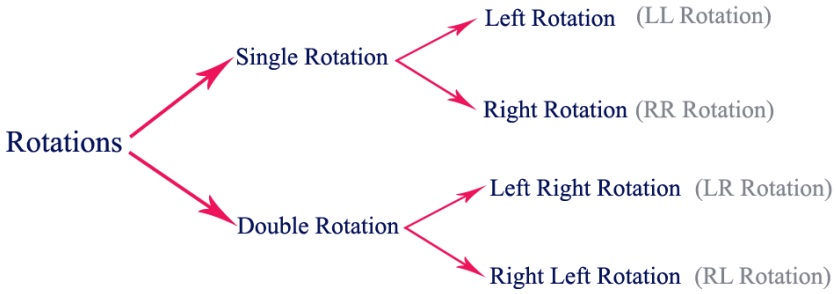
Postorder:
G Q E B A V R J L T F D
Inorder:
G E Q A B D V R F J T L

Program No. 6: Implementation of operations on AVL trees ADT

Description:

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 = heightOfLeftSubtree - heightOfRightSubtree

In AVL tree, after performing operations like insertion and deletion we need to check the balance factor of every node in the tree. If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced. Whenever the tree becomes imbalanced due to any operation we use rotation operations to make the tree balanced.



Algorithm:

The following operations are performed on AVL tree...

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

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.

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.

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.

Program:

```
#include <stdio.h>
#include <stdlib.h>
// Create Node
struct Node {
 int key;
 struct Node *left;
 struct Node *right;
 int height;
};
int max(int a, int b);
// Calculate height
```

```
int height(struct Node *N) {
 if (N == NULL)
 return 0;
 return N->height;
}
int max(int a, int b) {
 return (a > b) ? a : b;
}
// Create a node
struct Node *newNode(int key) {
 struct Node *node = (struct Node *)
 malloc(sizeof(struct Node));
 node->key = key;
 node->left = NULL;
 node->right = NULL;
 node->height = 1;
 return (node);
}
// Right rotate
struct Node *rightRotate(struct Node *y) {
 struct Node *x = y->left;
 struct Node *T2 = x->right;
 x->right = y;
 y->left = T2;
 y->height = max(height(y->left), height(y->right)) + 1;
 x->height = max(height(x->left), height(x->right)) + 1;
 return x;
}
// Left rotate
struct Node *leftRotate(struct Node *x) {
 struct Node *y = x->right;
 struct Node *T2 = y->left;
 y->left = x;
 x->right = T2;
 x->height = max(height(x->left), height(x->right)) + 1;
 y->height = max(height(y->left), height(y->right)) + 1;
 return y;
}
// Get the balance factor
int getBalance(struct Node *N) {
 if (N == NULL)
 return 0;
 return height(N->left) - height(N->right);
}
// Insert node
```

```
struct Node *insertNode(struct Node *node, int key) {
 // Find the correct position to insertNode the node and insertNode it
 if (node == NULL)
 return (newNode(key));
 if (key < node->key)
 node->left = insertNode(node->left, key);
 else if (key > node->key)
 node->right = insertNode(node->right, key);
 else
 return node;
 // Update the balance factor of each node and
 // Balance the tree
 node->height = 1 + max(height(node->left),
 height(node->right));
 int balance = getBalance(node);
 if (balance > 1 && key < node->left->key)
 return rightRotate(node);
 if (balance < -1 && key > node->right->key)
 return leftRotate(node);
 if (balance > 1 && key > node->left->key) {
 node->left = leftRotate(node->left);
 return rightRotate(node);
 }
 if (balance < -1 && key < node->right->key) {
 node->right = rightRotate(node->right);
 return leftRotate(node);
 }
 return node;
}
struct Node *minValueNode(struct Node *node) {
 struct Node *current = node;
 while (current->left != NULL)
 current = current->left;
 return current;
}
// Delete a nodes
struct Node *deleteNode(struct Node *root, int key) {
 // Find the node and delete it
 if (root == NULL)
 return root;
 if (key < root->key)
 root->left = deleteNode(root->left, key);
 else if (key > root->key)
 root->right = deleteNode(root->right, key);
 else {
```

```
 if ((root->left == NULL) || (root->right == NULL)) {
 struct Node *temp = root->left ? root->left : root->right;
 if (temp == NULL) {
 temp = root;
 root = NULL;
 } else
 *root = *temp;
 free(temp);
 } else {
 struct Node *temp = minValueNode(root->right);
 root->key = temp->key;
 root->right = deleteNode(root->right, temp->key);
 }
 }
 if (root == NULL)
 return root;
 // Update the balance factor of each node and
 // balance the tree
 root->height = 1 + max(height(root->left),
 height(root->right));
 int balance = getBalance(root);
 if (balance > 1 && getBalance(root->left) >= 0)
 return rightRotate(root);
 if (balance > 1 && getBalance(root->left) < 0) {
 root->left = leftRotate(root->left);
 return rightRotate(root);
 }
 if (balance < -1 && getBalance(root->right) <= 0)
 return leftRotate(root);
 if (balance < -1 && getBalance(root->right) > 0) {
 root->right = rightRotate(root->right);
 return leftRotate(root);
 }
 return root;
}
// Print the tree
void printPreOrder(struct Node *root) {
 if (root != NULL) {
 printf("%d ", root->key);
 printPreOrder(root->left);
 printPreOrder(root->right);
 }
}
int main() {
 struct Node *root = NULL;
```

```
 root = insertNode(root, 2);
 root = insertNode(root, 1);
 root = insertNode(root, 7);
 root = insertNode(root, 4);
 root = insertNode(root, 5);
 root = insertNode(root, 3);
 root = insertNode(root, 8);
 printPreOrder(root);
 root = deleteNode(root, 3);
 printf("\nAfter deletion: ");
 printPreOrder(root);
 return 0;
}
```

Output:

```
4 2 1 3 7 5 8
After deletion: 4 2 1 7 5 8
```

Program No. 7: Implementation of Linear search and Binary Search

a) Linear Search

Description:

Linear search is a sequential searching algorithm where we start from one end and check every element of the list until the desired element is found. It is the simplest searching algorithm.

Algorithm:

Step 1 - Read the search element from the user.
Step 2 - Compare the search element with the first element in the list.
Step 3 - If both are matched, then display "Given element is found!!!" and terminate the function
Step 4 - If both are not matched, then compare search element with the next element in the list.
Step 5 - Repeat steps 3 and 4 until search element is compared with last element in the list.
Step 6 - If last element in the list also doesn't match, then display "Element is not found!!!" and terminate the function.

Program:

```
#include <stdio.h>
int main()
{
```

```
 int array[100],c,n,search;
printf("enter number of elements in array\n");
scanf("%d",&n);
printf("enter %d elements\n",n);
for(c=0; c<n; c++)
{
 scanf("%d",&array[c]);
}
printf("enter the number you want to search\n");
scanf("%d",&search);
for(c=0; c<n; c++)
{
 if(array[c]==search)
 {
 printf("%d is present at location %d\n",search,c+1);
 break;
 }
}
if(array[c]!=search)
printf("%d is not present in the array\n",search);
return 0;
}
```

Output:

```
enter number of elements in array
5
enter 5 elements
12
10
15
5
4
enter the number you want to search
15
15 is present at location 3
```

b) Binary Search

Description:

The binary search algorithm can be used with only a sorted list of elements. That means the binary search is used only with a list of elements that are already arranged in an order. The binary search can not be used for a list of elements arranged in random order. This search

process starts comparing the search element with the middle element in the list. If both are matched, then the result is "element found". Otherwise, we check whether the search element is smaller or larger than the middle element in the list. If the search element is smaller, then we repeat the same process for the left sublist of the middle element. If the search element is larger, then we repeat the same process for the right sublist of the middle element. We repeat this process until we find the search element in the list or until we left with a sublist of only one element. And if that element also doesn't match with the search element, then the result is "Element not found in the list".

Algorithm:

Step 1 - Read the search element from the user.
Step 2 - Find the middle element in the sorted list.
Step 3 - Compare the search element with the middle element in the sorted list.
Step 4 - If both are matched, then display "Given element is found!!!" and terminate the function.
Step 5 - If both are not matched, then check whether the search element is smaller or larger than the middle element.
Step 6 - If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.
Step 7 - If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.
Step 8 - Repeat the same process until we find the search element in the list or until sublist contains only one element.
Step 9 - If that element also doesn't match with the search element, then display "Element is not found in the list!!!" and terminate the function.

Program:

```
#include <stdio.h>
int main()
{
 int a[30],i,n,x,low,high,mid,xloc=-1;
printf("enter number of elements in array\n");
scanf("%d",&n);
printf("enter %d elements\n",n);
for(i=0; i<n; i++)
{ scanf("%d",&a[i]);
}
printf("enter the number you want to search\n");
scanf("%d",&x);
low=0;high=n-1;
while(low<=high)
{
 mid=(low+high)/2;
```

```
 if(a[mid]==x)
 {
 xloc=mid;
 break;
 }
 else if(x<a[mid])
 high=mid-1;
 else low=mid+1;
}
if(xloc==-1)
printf("%d is not present in the array\n",x);
else
printf("%d is present at location %d\n",x,xloc);
return 0;
}
```

Output:

```
nter number of elements in array
5
enter 5 elements
10
15
20
25
30
enter the number you want to search
25
25 is present at location 3
```

Program No. 8: Implementation of Hashing collision resolution techniques.

Description:

Hash table is just an array which maps a key (data) into the data structure with the help of hash function such that insertion, deletion and search operations are performed with constant time complexity (i.e. O(1)).

When the hash function generates the same index for multiple keys, there will be a conflict (what value to be stored in that index). This is called a hash collision.

We can resolve the hash collision using one of the following techniques.

- Collision resolution by chaining
- Open Addressing: Linear/Quadratic Probing and Double Hashing

Algorithm:

Step 1: The hash value, which serves as the index of the data structure where the item will be stored, is calculated using hash functions, which are mathematical formulas.
Step 2: Next, let's give all alphabetical characters the values:
"a" = 1,
"b" = 2, etc.
Step 3: As a result, the string's numerical value is calculated by adding up all of the characters:
"ab" = 1 + 2 = 3,
"cd" = 3 + 4 = 7 ,
"efg" = 5 + 6 + 7 = 18
Step 4: Assume that we have a 7-column table to hold these strings. The sum of the characters in key mod Table size is utilized as the hash function in this case. By using sum(string) mod 7, we can determine where a string is located in an array.
Step 5: So we will then store
"ab" in 3 mod 7 = 3,
"cd" in 7 mod 7 = 0, and
"efg" in 18 mod 7 = 4.

Program:

```
#include <stdio.h>
#include <stdlib.h>
struct set
{
 int key;
 int data;
};
struct set *array;
int capacity = 10;
int size = 0;

int hashFunction(int key)
{
 return (key % capacity);
}
int checkPrime(int n)
{
 int i;
 if (n == 1 || n == 0)
 {
 return 0;
 }
 for (i = 2; i < n / 2; i++)
 {
```

```
 if (n % i == 0)
 {
 return 0;
 }
 }
 return 1;
}
int getPrime(int n)
{
 if (n % 2 == 0)
 {
 n++;
 }
 while (!checkPrime(n))
 {
 n += 2;
 }
 return n;
}
void init_array()
{
 capacity = getPrime(capacity);
 array = (struct set *)malloc(capacity * sizeof(struct set));
 for (int i = 0; i < capacity; i++)
 {
 array[i].key = 0;
 array[i].data = 0;
 }
}

void insert(int key, int data)
{
 int index = hashFunction(key);
 if (array[index].data == 0)
 {
 array[index].key = key;
 array[index].data = data;
 size++;
 printf("\n Key (%d) has been inserted \n", key);
 }
 else if (array[index].key == key)
 {
 array[index].data = data;
 }
 else
```

```
 {
 printf("\n Collision occured \n");
 }
}

void remove_element(int key)
{
 int index = hashFunction(key);
 if (array[index].data == 0)
 {
 printf("\n This key does not exist \n");
 }
 else
 {
 array[index].key = 0;
 array[index].data = 0;
 size--;
 printf("\n Key (%d) has been removed \n", key);
 }
}
void display()
{
 int i;
 for (i = 0; i < capacity; i++)
 {
 if (array[i].data == 0)
 {
 printf("\n array[%d]: / ", i);
 }
 else
 {
 printf("\n key: %d array[%d]: %d \t", array[i].key, i, array[i].data);
 }
 }
}

int size_of_hashtable()
{
 return size;
}

int main()
{
 int choice, key, data, n;
 int c = 0;
```

```
init_array();

do
{
printf("1.Insert item in the Hash Table"
"\n2.Remove item from the Hash Table"
"\n3.Check the size of Hash Table"
"\n4.Display a Hash Table"
"\n\n Please enter your choice: ");
scanf("%d", &choice);
switch (choice)
{
case 1:
printf("Enter key -:\t");
scanf("%d", &key);
printf("Enter data -:\t");
scanf("%d", &data);
insert(key, data);
break;

case 2:
printf("Enter the key to delete-:");
scanf("%d", &key);
remove_element(key);
break;

case 3:
n = size_of_hashtable();
printf("Size of Hash Table is-:%d\n", n);
break;

case 4:
display();
break;

default:
printf("Invalid Input\n");
}
printf("\nDo you want to continue (press 1 for yes): ");
scanf("%d", &c);
} while (c == 1);
}
```

Output:

```
1.Insert item in the Hash Table
2.Remove item from the Hash Table
3.Check the size of Hash Table
4.Display a Hash Table

 Please enter your choice: 1
 Enter key -:    10
 Enter data -:   15
 Key (10) has been inserted

Do you want to continue (press 1 for yes): 1
1.Insert item in the Hash Table
2.Remove item from the Hash Table
3.Check the size of Hash Table
4.Display a Hash Table

 Please enter your choice: 3
 Size of Hash Table is-:1
```

Program No. 8: Implementation of Insertion Sort, Selection Sort

a) Insertion Sort

Description:

Insertion sort algorithm arranges a list of elements in a particular order. In insertion sort algorithm, every iteration moves an element from unsorted portion to sorted portion until all the elements are sorted in the list.

Algorithm:

The insertion sort algorithm is performed using the following steps...

- **Step 1 -** Assume that first element in the list is in sorted portion and all the remaining elements are in unsorted portion.
- **Step 2:** Take first element from the unsorted portion and insert that element into the sorted portion in the order specified.
- **Step 3:** Repeat the above process until all the elements from the unsorted portion are moved into the sorted portion.

Program:

```
#include<stdio.h>

void main(){
```

```
  int size, i, j, temp, list[100];

  printf("Enter the size of the list: ");
  scanf("%d", &size);

  printf("Enter %d integer values: ", size);
  for (i = 0; i < size; i++)
    scanf("%d", &list[i]);

  //Insertion sort logic
  for (i = 1; i < size; i++) {
    temp = list[i];
    j = i - 1;
    while ((temp < list[j]) && (j >= 0)) {
      list[j + 1] = list[j];
      j = j - 1;
    }
    list[j + 1] = temp;
  }

  printf("List after Sorting is: ");
  for (i = 0; i < size; i++)
    printf(" %d", list[i]);
}
```

Output:

```
Enter the size of the list: 5
Enter 5 integer values: 10
9
4
8
3
List after Sorting is:  3 4 8 9 10
```

a) Selection Sort

Description:

Selection Sort algorithm is used to arrange a list of elements in a particular order (Ascending or Descending). In selection sort, the first element in the list is selected and it is compared repeatedly with all the remaining elements in the list. If any element is smaller than the selected element (for Ascending order), then both are swapped so that first position is filled with the smallest element in the sorted order. Next, we select the element at a second position in the list

and it is compared with all the remaining elements in the list. If any element is smaller than the selected element, then both are swapped. This procedure is repeated until the entire list is sorted.

Algorithm:

The selection sort algorithm is performed using the following steps...

- **Step 1 -** Select the first element of the list (i.e., Element at first position in the list).
- **Step 2:** Compare the selected element with all the other elements in the list.
- **Step 3:** In every comparision, if any element is found smaller than the selected element (for Ascending order), then both are swapped.
- **Step 4:** Repeat the same procedure with element in the next position in the list till the entire list is sorted.

Programs:

```
#include<stdio.h>
void main(){

  int size,i,j,temp,list[100];
  printf("Enter the size of the List: ");
  scanf("%d",&size);

  printf("Enter %d integer values: ",size);
  for(i=0; i<size; i++)
     scanf("%d",&list[i]);

  //Selection sort logic

  for(i=0; i<size; i++){
     for(j=i+1; j<size; j++){
          if(list[i] > list[j])
              {
              temp=list[i];
              list[i]=list[j];
              list[j]=temp;
          }
     }
  }

  printf("List after sorting is: ");
  for(i=0; i<size; i++)
     printf(" %d",list[i]);
}
```

Output:

```
Enter the size of the List: 5
Enter 5 integer values: 15
5
7
12
11
List after sorting is:  5 7 11 12 15
```

Program No. 10: Implementation of Merge Sort, Quick Sort

a) Merge Sort

Description:

Algorithm:

Program:

```
#include <stdio.h>

/* Function to merge the subarrays of a[] */
void merge(int a[], int beg, int mid, int end)
{
    int i, j, k;
    int n1 = mid - beg + 1;
    int n2 = end - mid;

    int LeftArray[n1], RightArray[n2]; //temporary arrays

    /* copy data to temp arrays */
    for (int i = 0; i < n1; i++)
    LeftArray[i] = a[beg + i];
    for (int j = 0; j < n2; j++)
    RightArray[j] = a[mid + 1 + j];

    i = 0; /* initial index of first sub-array */
    j = 0; /* initial index of second sub-array */
    k = beg;  /* initial index of merged sub-array */

    while (i < n1 && j < n2)
```

```
    {
        if(LeftArray[i] <= RightArray[j])
        {
            a[k] = LeftArray[i];
            i++;
        }
        else
        {
            a[k] = RightArray[j];
            j++;
        }
        k++;
    }
    while (i<n1)
    {
        a[k] = LeftArray[i];
        i++;
        k++;
    }

    while (j<n2)
    {
        a[k] = RightArray[j];
        j++;
        k++;
    }
}

void mergeSort(int a[], int beg, int end)
{
    if (beg < end)
    {
        int mid = (beg + end) / 2;
        mergeSort(a, beg, mid);
        mergeSort(a, mid + 1, end);
        merge(a, beg, mid, end);
    }
}

/* Function to print the array */
void printArray(int a[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", a[i]);
```

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

int main()
{
    int a[] = { 12, 31, 25, 8, 32, 17, 40, 42 };
    int n = sizeof(a) / sizeof(a[0]);
    printf("Before sorting array elements are - \n");
    printArray(a, n);
    mergeSort(a, 0, n - 1);
    printf("After sorting array elements are - \n");
    printArray(a, n);
    return 0;
}
```

Output:

```
Before sorting array elements are -
12 31 25 8 32 17 40 42
After sorting array elements are -
8 12 17 25 31 32 40 42
```

b) Quick Sort

Description:

Quick sort is a fast sorting algorithm used to sort a list of elements. Quick sort algorithm is invented by C. A. R. Hoare.
The quick sort algorithm attempts to separate the list of elements into two parts and then sort each part recursively. That means it use divide and conquer strategy. In quick sort, the partition of the list is performed based on the element called pivot. Here pivot element is one of the elements in the list.
The list is divided into two partitions such that "all elements to the left of pivot are smaller than the pivot and all elements to the right of pivot are greater than or equal to the pivot".

Algorithm:

- **Step 1 -** Consider the first element of the list as **pivot** (i.e., Element at first position in the list).
- **Step 2 -** Define two variables i and j. Set i and j to first and last elements of the list respectively.
- **Step 3 -** Increment i until list[i] > pivot then stop.
- **Step 4 -** Decrement j until list[j] < pivot then stop.
- **Step 5 -** If i < j then exchange list[i] and list[j].
- **Step 6 -** Repeat steps 3,4 & 5 until i > j.

- **Step 7 -** Exchange the pivot element with list[j] element.

Program:

```
#include<stdio.h>
void quickSort(int [10],int,int);

void main(){
  int list[20],size,i;
  printf("Enter size of the list: ");
  scanf("%d",&size);
  printf("Enter %d integer values: ",size);
  for(i = 0; i < size; i++)
    scanf("%d",&list[i]);
  quickSort(list,0,size-1);
  printf("List after sorting is: ");
  for(i = 0; i < size; i++)
    printf(" %d",list[i]);
  getch();
}

void quickSort(int list[10],int first,int last){
   int pivot,i,j,temp;
    if(first < last){
        pivot = first;
        i = first;
        j = last;

        while(i < j){
           while(list[i] <= list[pivot] && i < last)
              i++;
           while(list[j] > list[pivot])
              j--;
           if(i <j){
               temp = list[i];
               list[i] = list[j];
               list[j] = temp;
           }
        }

        temp = list[pivot];
        list[pivot] = list[j];
        list[j] = temp;
        quickSort(list,first,j-1);
        quickSort(list,j+1,last);
```

```
    }
}
```

Output:

```
Enter size of the list: 5
Enter 5 integer values: 12
15
9
7
2
List after sorting is:  2 7 9 12 15
```

Program No. 11: Implementation of Heap Sort.

Description:

A heap is a complete binary tree, and the binary tree is a tree in which the node can have the utmost two children. A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node, should be completely filled, and all the nodes should be left-justified.

Heap sort basically recursively performs two main operations -

- Build a heap H, using the elements of array.
- Repeatedly delete the root element of the heap formed in 1st phase.

Algorithm:

The Heap sort algorithm to arrange a list of elements in ascending order is performed using following steps...

- **Step 1 -** Construct a **Binary Tree** with given list of Elements.
- **Step 2 -** Transform the Binary Tree into **Min Heap.**
- **Step 3 -** Delete the root element from Min Heap using **Heapify** method.
- **Step 4 -** Put the deleted element into the Sorted list.
- **Step 5 -** Repeat the same until Min Heap becomes empty.
- **Step 6 -** Display the sorted list.

Program:

```
#include <stdio.h>
/* function to heapify a subtree. Here 'i' is the
index of root node in array a[], and 'n' is the size of heap. */
void heapify(int a[], int n, int i)
{
```

```
    int largest = i; // Initialize largest as root
    int left = 2 * i + 1; // left child
    int right = 2 * i + 2; // right child
    // If left child is larger than root
    if (left < n && a[left] > a[largest])
        largest = left;
    // If right child is larger than root
    if (right < n && a[right] > a[largest])
        largest = right;
    // If root is not largest
    if (largest != i) {
        // swap a[i] with a[largest]
        int temp = a[i];
        a[i] = a[largest];
        a[largest] = temp;

        heapify(a, n, largest);
    }
}
/*Function to implement the heap sort*/
void heapSort(int a[], int n)
{
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(a, n, i);
    // One by one extract an element from heap
    for (int i = n - 1; i >= 0; i--) {
        /* Move current root element to end*/
        // swap a[0] with a[i]
        int temp = a[0];
        a[0] = a[i];
        a[i] = temp;

        heapify(a, i, 0);
    }
}
/* function to print the array elements */
void printArr(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
    {
        printf("%d", arr[i]);
        printf(" ");
    }

}
```

```
int main()
{
   int a[] = {48, 10, 23, 43, 28, 26, 1};
   int n = sizeof(a) / sizeof(a[0]);
   printf("Before sorting array elements are - \n");
   printArr(a, n);
   heapSort(a, n);
   printf("\nAfter sorting array elements are - \n");
   printArr(a, n);
   return 0;
}
```

Output:

```
Before sorting array elements are -
48 10 23 43 28 26 1
After sorting array elements are -
1 10 23 26 28 43 48
```

Program No. 12: Implementation of DFS and BFS

Description:

Graph traversal is a technique used for a searching vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process. A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.

There are two graph traversal techniques and they are as follows...

DFS (Depth First Search)
BFS (Breadth First Search)

DFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph without loops. We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal.

BFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph without loops. We use Queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal.

Algorithm:

We use the following steps to implement DFS traversal...

- **Step 1 -** Define a Stack of size total number of vertices in the graph.
- **Step 2 -** Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.
- **Step 3 -** Visit any one of the non-visited **adjacent** vertices of a vertex which is at the top of stack and push it on to the stack.
- **Step 4 -** Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.
- **Step 5 -** When there is no new vertex to visit then use **back tracking** and pop one vertex from the stack.
- **Step 6 -** Repeat steps 3, 4 and 5 until stack becomes Empty.
- **Step 7 -** When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

We use the following steps to implement BFS traversal...

- **Step 1 -** Define a Queue of size total number of vertices in the graph.
- **Step 2 -** Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.
- **Step 3 -** Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.
- **Step 4 -** When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.
- **Step 5 -** Repeat steps 3 and 4 until queue becomes empty.
- **Step 6 -** When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

Program:

```
#include<stdio.h>

int q[20],top=-1,front=-1,rear=-1,a[20][20],vis[20],stack[20];
int delete();
void add(int item);
void bfs(int s,int n);
void dfs(int s,int n);
void push(int item);
int pop();

void main()
{
int n,i,s,ch,j;
char c,dummy;
printf("ENTER THE NUMBER VERTICES ");
scanf("%d",&n);
for(i=1;i<=n;i++)
```

```
{
for(j=1;j<=n;j++)
{
printf("ENTER 1 IF %d HAS A NODE WITH %d ELSE 0 ",i,j);
scanf("%d",&a[i][j]);
}
}
printf("THE ADJACENCY MATRIX IS\n");
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
printf(" %d",a[i][j]);
}
printf("\n");
}

do
{
for(i=1;i<=n;i++)
vis[i]=0;
printf("\nMENU");
printf("\n1.B.F.S");
printf("\n2.D.F.S");
printf("\nENTER YOUR CHOICE");
scanf("%d",&ch);
printf("ENTER THE SOURCE VERTEX :");
scanf("%d",&s);

switch(ch)
{
case 1:bfs(s,n);
break;
case 2:
dfs(s,n);
break;
}
printf("DO U WANT TO CONTINUE(Y/N) ? ");
scanf("%c",&dummy);
scanf("%c",&c);
}while((c=='y')||(c=='Y'));
}


//**************BFS(breadth-first search) code**************//
```

```
void bfs(int s,int n)
{
int p,i;
add(s);
vis[s]=1;
p=delete();
if(p!=0)
printf(" %d",p);
while(p!=0)
{
for(i=1;i<=n;i++)
if((a[p][i]!=0)&&(vis[i]==0))
{
add(i);
vis[i]=1;
}
p=delete();
if(p!=0)
printf(" %d ",p);
}
for(i=1;i<=n;i++)
if(vis[i]==0)
bfs(i,n);
}


void add(int item)
{
if(rear==19)
printf("QUEUE FULL");
else
{
if(rear==-1)
{
q[++rear]=item;
front++;
}
else
q[++rear]=item;
}
}
int delete()
{
int k;
if((front>rear)||(front==-1))
```

```
return(0);
else
{
k=q[front++];
return(k);
}
}


//***************DFS(depth-first search) code*****************//
void dfs(int s,int n)
{
int i,k;
push(s);
vis[s]=1;
k=pop();
if(k!=0)
printf(" %d ",k);
while(k!=0)
{
for(i=1;i<=n;i++)
if((a[k][i]!=0)&&(vis[i]==0))
{
push(i);
vis[i]=1;
}
k=pop();
if(k!=0)
printf(" %d ",k);
}
for(i=1;i<=n;i++)
if(vis[i]==0)
dfs(i,n);
}
void push(int item)
{
if(top==19)
printf("Stack overflow ");
else
stack[++top]=item;
}
int pop()
{
int k;
if(top==-1)
```

```
return(0);
else
{
k=stack[top--];
return(k);
}
}
```

Output:

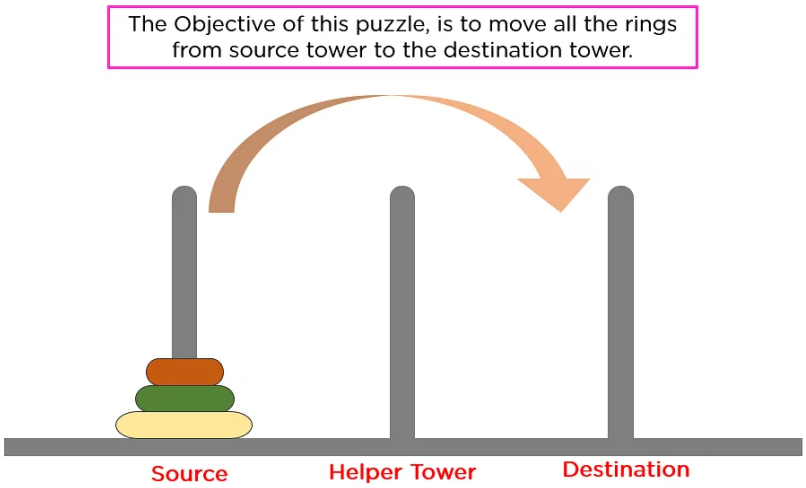
```
ENTER THE NUMBER VERTICES 3
ENTER 1 IF 1 HAS A NODE WITH 1 ELSE 0 1
ENTER 1 IF 1 HAS A NODE WITH 2 ELSE 0 1
ENTER 1 IF 1 HAS A NODE WITH 3 ELSE 0 1
ENTER 1 IF 2 HAS A NODE WITH 1 ELSE 0 1
ENTER 1 IF 2 HAS A NODE WITH 2 ELSE 0 1
ENTER 1 IF 2 HAS A NODE WITH 3 ELSE 0 1
ENTER 1 IF 3 HAS A NODE WITH 1 ELSE 0 1
ENTER 1 IF 3 HAS A NODE WITH 2 ELSE 0 1
ENTER 1 IF 3 HAS A NODE WITH 3 ELSE 0 1
THE ADJACENCY MATRIX IS
 1 1 1
 1 1 1
 1 1 1

MENU
1.B.F.S
2.D.F.S
ENTER YOUR CHOICE
```

ADDITIONAL; PROGRAMS

Program No 13. Implement Tower of Hanoi using Recursion

Description:

The Tower of Hanoi is a mathematical problem composed of three towers and numerous rings arranged in increasing order of their diameters. The number of towers is constant for this problem, whereas the player can vary the number of rings he wants to use. The image given below depicts the setup of the TOH puzzle.



Rules of Tower of Hanoi Puzzle
The Tower of Hanoi problem is solved using the set of rules given below:

- Only one disc can be moved at a time.
- Only the top disc of one stack can be transferred to the top of another stack or an empty rod.
- Larger discs cannot be stacked over smaller ones.

The complexity of this problem can be mapped by evaluating the number of possible moves. The least movements needed to solve the Tower of Hanoi problem with n discs are 2n-1.

Algorithm:

A, B and C are rods or pegs and n is the total number of discs, 1 is the largest disk and 5 is the smallest one.

1. Move n-1 discs from rod A to B which causes disc n alone in on the rod A
2. Transfer the disc n from A to C
3. Transfer n-1 discs from rod B to C so that they sit over disc n

Program:

```
/* C program for Tower of Hanoi*/
/*Application of Recursive function*/
#include <stdio.h>
void hanoifun(int n, char fr, char tr, char ar)//fr=from rod,tr =to rod, ar=aux rod
{
   if (n == 1)
   {
      printf("\n Move disk 1 from rod %c to rod %c", fr, tr);
      return;
```

```
    }
    hanoifun(n-1, fr, ar, tr);
    printf("\n Move disk %d from rod %c to rod %c", n, fr, tr);
    hanoifun(n-1, ar, tr, fr);
}

int main()
{
    int n; // n immplies the number of discs
    printf("Enter no of disks: ");
    scanf("%d",&n);
    hanoifun(n, 'A', 'C', 'B');  // A, B and C are the name of rod
    return 0;
}
```

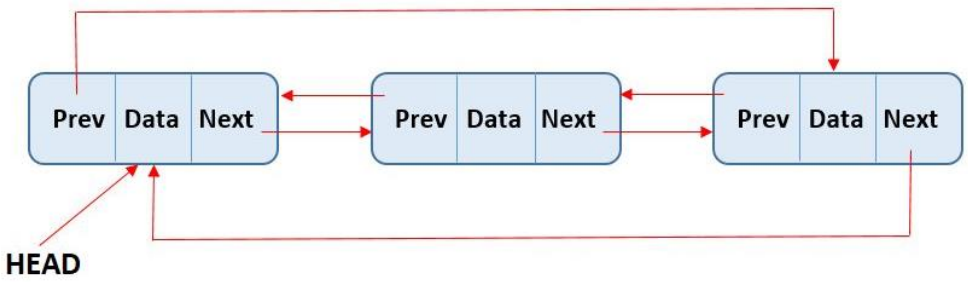
Output:

```
Enter no of disks: 4
Move disk 1 from rod A to rod B
 Move disk 2 from rod A to rod C
 Move disk 1 from rod B to rod C
 Move disk 3 from rod A to rod B
 Move disk 1 from rod C to rod A
 Move disk 2 from rod C to rod B
 Move disk 1 from rod A to rod B
 Move disk 4 from rod A to rod C
 Move disk 1 from rod B to rod C
 Move disk 2 from rod B to rod A
 Move disk 1 from rod C to rod A
 Move disk 3 from rod B to rod C
 Move disk 1 from rod A to rod B
 Move disk 2 from rod A to rod C
 Move disk 1 from rod B to rod C
```

Program No.14: Implementation Circular DLL

Description:

Circular doubly linked list is a more complexed type of data structure in which a node contain pointers to its previous node as well as the next node. Circular doubly linked list doesn't contain NULL in any of the node. The last node of the list contains the address of the first node of the list. The first node of the list also contain address of the last node in its previous pointer.



Algorithm:

We can do some operations on the circular linked list like the double linked list which are:

Insertion

A node can be added in three ways:

1. Insertion at the beginning of the list
2. Insertion at the end of the list
3. Insertion in between the nodes

Deletion

In a circular linked list, the deletion operation can be performed in three ways:

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Program:

```
#include<stdio.h>
#include<stdlib.h>
struct cdl
{
int x;
struct cdl *next;
struct cdl *prev;
};

int main()
{
int n,i;
struct cdl *head,*temp,*new;
head=malloc(sizeof (struct cdl));
printf("enter no.of nodes:");
```

```
scanf("%d",&n);
for(i=1;i<n;i++)
{

   if(head->x==NULL)
   {
      printf("Enter data: ");
      scanf("%d",&head->x);
      head->prev=NULL;
      head->next=NULL;
      temp=head;
   }
new=malloc(sizeof (struct cdl));
printf("Enter data: ");
scanf("%d",&new->x);
new->prev=temp;
new->next=head;
head->prev=new;
temp->next=new;
temp=new;
}

printf("forward direction\n ");
temp=head;
do
{
printf("%d\n",temp->x);
temp=temp->next;
}while(temp!=head);
printf("%d\n",temp->x);
printf("reverse direction \n");
do
{
printf("%d\n",temp->x);
temp=temp->prev;
}while(temp!=head);
printf("%d",temp->x);

return 0;
}
```

Output:

```
enter no.of nodes:4
Enter data: 10
```

```
Enter data: 20
Enter data: 30
Enter data: 40
forward direction
 10
20
30
40
10
reverse direction
10
40
30
20
10
```