KNOWLEDGE INSTITUTE OF TECHNOLOGY

(AN AUTONOMOUS INSTITUTION)

Approved by AICTE, New Delhi and Affiliated to Anna University, Chennai

Kakapalayam (Po), Salem – 637 504

RECORD NOTE BOOK

REG. NO.

Certified that this is the bonafide record of work done by Selvan/Selvi……………………………………………………………of the …………………… Semester **B.TECH INFORMATION TECHNOLOGY** Branch during the year 2025-2026 in the **BE23CS404-DATA STRUCTURES AND ALGORITHMS** Laboratory.

Staff In-charge **Head of the Department**

Submitted for the Model Practical Examination on ___________________.

Internal Examiner

CONTENTS

Ex. No. 1a **Array implementation of List ADT**
Date:

Aim

To write a C program to implement List ADT by using arrays.

Algorithm

1. Insert at Position

- If list is full → display error.
- Shift elements right from the position.
- Insert new element at the position.

2. Delete at Position

- If list is empty or position invalid → display error.
- Shift elements left from the position.

3. Search Element

- Linearly check each element for a match.

4. Display List

- Traverse and print elements.

Program

```
   /* List operation using Arrays */
#include <stdio.h>
#define MAX 100

int list[MAX];
int size = 0;

void insertAtPosition(int pos, int value) {
    if (size == MAX) {
        printf("List is full!\n");
        return;
    }
    if (pos < 0 || pos > size) {
        printf("Invalid position!\n");
        return;
    }
    for (int i = size; i > pos; i--)
        list[i] = list[i - 1];
    list[pos] = value;
    size++;
    printf("Inserted %d at position %d\n", value, pos);
}

void deleteAtPosition(int pos) {
    if (size == 0) {
        printf("List is empty!\n");
        return;
    }
    if (pos < 0 || pos >= size) {
        printf("Invalid position!\n");
        return;
    }
    int deleted = list[pos];
    for (int i = pos; i < size - 1; i++)
        list[i] = list[i + 1];
    size--;
    printf("Deleted %d from position %d\n", deleted, pos);
}

void search(int value) {
    for (int i = 0; i < size; i++) {
        if (list[i] == value) {
            printf("Element %d found at position %d\n", value, i);
            return;
        }
    }
    printf("Element %d not found in the list\n", value);
}

void display() {
    if (size == 0) {
        printf("List is empty!\n");
        return;
    }
    printf("List elements: ");
    for (int i = 0; i < size; i++)
        printf("%d ", list[i]);
    printf("\n");
}

int main() {
    int choice, value, pos;
    while (1) {
        printf("\nMenu:\n");
        printf("1. Insert at position\n");
        printf("2. Delete at position\n");
        printf("3. Search element\n");
        printf("4. Display list\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value and position: ");
                scanf("%d %d", &value, &pos);
                insertAtPosition(pos, value);
                break;
            case 2:
                printf("Enter position to delete: ");
                scanf("%d", &pos);
                deleteAtPosition(pos);
                break;
            case 3:
                printf("Enter value to search: ");
                scanf("%d", &value);
                search(value);
                break;
            case 4:
                display();
                break;
            case 5:
                return 0;
            default:
                printf("Invalid choice!\n");
        }
    }
}
```

Output

```
Menu:
1. Insert at position
2. Delete at position
3. Search element
4. Display list
5. Exit
```

```
Enter your choice: 1
Enter value and position: 10 0
Inserted 10 at position 0

Enter your choice: 1
Enter value and position: 20 1
Inserted 20 at position 1

Enter your choice: 4
List elements: 10 20

Enter your choice: 3
Enter value to search: 20
Element 20 found at position 1

Enter your choice: 2
Enter position to delete: 0
Deleted 10 from position 0

Enter your choice: 4
List elements: 20
```

DEPARTMENT OF IT		
AIM & ALGORITHM	**02**	
PROGRAM	**03**	
EXECUTION & OUTPUT	**03**	
VIVA VOCE	**02**	
TOTAL	**10**	

Result

Thus the C program to implement List ADT by using array is executed successfully and the output is verified.

Ex. No. 1b **Pointer Based / Linked List Implementation of List ADT**
Date:

Aim

To write a C program to implement List ADT by using linked list.

Algorithm

1. Insertion:

- **At beginning**: Create node → Point new node's next to head → Update head.
- **At end**: Traverse to end → Point last node's next to new node.
- **At position**: Traverse to (pos−1)th node → Insert new node after it.

2. Deletion:

- **From position**: Traverse to (pos−1)th node → Remove next node and free memory.

3. Search:

- Traverse each node → Compare data with key.

4. Display:

- Traverse from head to NULL → Print each node’s data.

Program

```
  /* Single Linked List */
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* next;
};

struct Node* head = NULL;

// Create new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}

void insertAtBeginning(int value) {
    struct Node* newNode = createNode(value);
    newNode->next = head;
    head = newNode;
    printf("Inserted %d at beginning.\n", value);
}

void insertAtEnd(int value) {
    struct Node* newNode = createNode(value);
    if (head == NULL) {
        head = newNode;
    } else {
        struct Node* temp = head;
        while (temp->next != NULL)
            temp = temp->next;
        temp->next = newNode;
    }
    printf("Inserted %d at end.\n", value);
}

void insertAtPosition(int pos, int value) {
    if (pos == 0) {
        insertAtBeginning(value);
        return;
    }

    struct Node* newNode = createNode(value);
    struct Node* temp = head;

    for (int i = 0; i < pos - 1; i++) {
        if (temp == NULL) {
            printf("Position out of bounds!\n");
            return;
        }
        temp = temp->next;
    }

    if (temp == NULL) {
        printf("Position out of bounds!\n");
        return;
    }

    newNode->next = temp->next;
    temp->next = newNode;
    printf("Inserted %d at position %d.\n", value, pos);
}

void deleteAtPosition(int pos) {
    if (head == NULL) {
        printf("List is empty!\n");
        return;
    }

    struct Node* temp = head;

    if (pos == 0) {
        head = head->next;
        printf("Deleted %d from position 0\n", temp->data);
        free(temp);
        return;
    }

    for (int i = 0; i < pos - 1; i++) {
        if (temp == NULL || temp->next == NULL) {
            printf("Position out of bounds!\n");
            return;
        }
        temp = temp->next;
    }

    struct Node* delNode = temp->next;
    if (delNode == NULL) {
        printf("Position out of bounds!\n");
        return;
    }

    temp->next = delNode->next;
    printf("Deleted %d from position %d\n", delNode->data, pos);
    free(delNode);
}

void search(int value) {
    struct Node* temp = head;
    int pos = 0;

    while (temp != NULL) {
        if (temp->data == value) {
            printf("Element %d found at position %d\n", value, pos);
            return;
        }
        temp = temp->next;
        pos++;
    }

    printf("Element %d not found in the list\n", value);
}

void display() {
    struct Node* temp = head;
    if (temp == NULL) {
        printf("List is empty!\n");
        return;
    }
    printf("List elements: ");
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
int main() {
    int choice, value, pos;
    while (1) {
        printf("\nMenu:\n");
        printf("1. Insert at beginning\n");
        printf("2. Insert at end\n");
        printf("3. Insert at position\n");
        printf("4. Delete at position\n");
        printf("5. Search element\n");
        printf("6. Display list\n");
        printf("7. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value: ");
                scanf("%d", &value);
                insertAtBeginning(value);
                break;
            case 2:
                printf("Enter value: ");
                scanf("%d", &value);
                insertAtEnd(value);
                break;
            case 3:
                printf("Enter position and value: ");
                scanf("%d %d", &pos, &value);
                insertAtPosition(pos, value);
                break;
            case 4:
                printf("Enter position to delete: ");
                scanf("%d", &pos);
                deleteAtPosition(pos);
                break;
            case 5:
                printf("Enter value to search: ");
                scanf("%d", &value);
                search(value);
                break;
            case 6:
                display();
                break;
            case 7:
                return 0;
            default:
                printf("Invalid choice!\n");
        }
    }
}
```

Output

```
Menu:
1. Insert at beginning
2. Insert at end
3. Insert at position
4. Delete at position
5. Search element
6. Display list
7. Exit
Enter your choice: 1
Enter value: 10
Inserted 10 at beginning.

Enter your choice: 2
Enter value: 20
Inserted 20 at end.

Enter your choice: 3
Enter position and value: 1 15
Inserted 15 at position 1.
Enter your choice: 6
List elements: 10 -> 15 -> 20 -> NULL
Enter your choice: 4
Enter position to delete: 1
Deleted 15 from position 1
Enter your choice: 6
List elements: 10 -> 20 -> NULL
```

DEPARTMENT OF IT		
AIM & ALGORITHM	**02**	
PROGRAM	**03**	
EXECUTION & OUTPUT	**03**	
VIVA VOCE	**02**	
TOTAL	**10**	

Result

Thus the C program to implement List ADT by using linked list is executed successfully and the output is verified.

Ex. No. 2a **Array implementation of Stack ADT**
Date:

Aim

To write a C program to implement Stack ADT by using arrays.

Algorithm

1. Create a Stack[ ] with MAX size as your wish.
2. Write function for all the basic operations of stack - PUSH(), POP() and DISPLAY().
3. By using Switch case, select push() operation to insert element in 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).
4. Similarly, By using Switch case, select pop() operation to delete 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--).
5. Similarly, By using Switch case, select display() operation to display all element from the 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'.
6. Close the program.

Program

```
/* Stack Operation using Arrays */

#include <stdio.h>
#include <conio.h>

#define max 5

static int stack[max];
int top = -1;

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

int pop()
{
   return (stack[top--]);
}

void view()
{
   int i;
   if (top < 0)
      printf("\n Stack Empty \n");
   else
   {
      printf("\n Top-->");
      for(i=top; i>=0; i--)
      {
         printf("%4d", stack[i]);
      }
      printf("\n");
   }
}

main()
{
   int ch=0, val;
   clrscr();

   while(ch != 4)
   {
      printf("\n STACKOPERATION \n");
      printf("1.PUSH  ");
      printf("2.POP  ");
      printf("3.VIEW  ");
      printf("4.QUIT \n");
      printf("Enter Choice : ");
      scanf("%d", &ch);
      switch(ch)
      {
         case 1:
            if(top < max-1)
            {
               printf("\nEnter Stack element : ");
               scanf("%d", &val);
               push(val);
            }
            else
               printf("\n Stack Overflow \n");
            break;
         case 2:
            if(top < 0)
               printf("\n Stack Underflow \n");
            else
            {
               val = pop();
               printf("\n Popped element is %d\n", val);
            }
            break;
         case 3:
            view();
            break;
         case 4:
            exit(0);
         default:
            printf("\n Invalid Choice \n");
      }
   }
}
```

Output

```
STACK   OPERATION
1.PUSH  2.POP  3.VIEW  4.QUIT
Enter Choice : 1

Enter Stack element : 12

STACK   OPERATION
1.PUSH  2.POP  3.VIEW  4.QUIT
Enter Choice : 1

Enter Stack element : 23
```

```
STACK   OPERATION
1.PUSH  2.POP  3.VIEW  4.QUIT
Enter Choice : 1

Enter Stack element : 34

STACK   OPERATION
1.PUSH  2.POP  3.VIEW  4.QUIT
Enter Choice : 1

Enter Stack element : 45

STACK   OPERATION
1.PUSH  2.POP  3.VIEW  4.QUIT
Enter Choice : 3

Top-->  45  34  23  12

STACK   OPERATION
1.PUSH  2.POP  3.VIEW  4.QUIT
Enter Choice : 2

Popped element is 45

STACK   OPERATION
1.PUSH  2.POP  3.VIEW  4.QUIT
Enter Choice : 3

Top-->  34  23  12

STACK   OPERATION
1.PUSH  2.POP  3.VIEW  4.QUIT
Enter Choice : 4
```

DEPARTMENT OF IT		
AIM & ALGORITHM	**02**	
PROGRAM	**03**	
EXECUTION & OUTPUT	**03**	
VIVA VOCE	**02**	
TOTAL	**10**	

Result

Thus the C program to implement Stack ADT by using array is executed successfullyand the output is verified.

Ex. No. 2b **Pointer Based / Linked List Implementation of Stack ADT**

Date:

Aim

To write a C program to implement Stack ADT by using linked list.

Algorithm

1 Include all the header files which are used in the program. And declare all theuser defined functions.
2 Define a 'Node' structure with two members data and next.
3 Define a Node pointer 'top' and set it to NULL.
4 Implement the main method by displaying Menu with list of operations andmake suitable function calls in the main method.
5 push(value) - Inserting an element into the Stack
   a. Create a newNode with given value.
   b. Check whether stack is Empty (top == NULL)
   c. If it is Empty, then set newNode → next = NULL.
   d. If it is Not Empty, then set newNode → next = top.
   e. Finally, set top = newNode.
6 pop() - Deleting an Element from a Stack
   a. Check whether stack is Empty (top == NULL).
   b. If it is Empty, then display "Stack is Empty!!! Deletion is not possible!!!" and terminate the function
   c. If it is Not Empty, then define a Node pointer 'temp' and set it to 'top'.
   d. Then set 'top = top → next'.
   e. Finally, delete 'temp' (free(temp)).
7 display() - Displaying stack of elements
   1. Check whether stack is Empty (top == NULL).
   2. If it is Empty, then display 'Stack is Empty!!!' and terminate the function.
   3. If it is Not Empty, then define a Node pointer 'temp' and initialize with top.
   4. Display 'temp → data --->' and move it to the next node. Repeat the sameuntil temp reaches to the first node in the stack (temp → next != NULL).
8 Finally! Display 'temp → data ---> NULL'.

Program

```
/* Stack using Single Linked List */

#include <stdio.h>
#include <conio.h>
#include <process.h>
#include <alloc.h>

struct node
{
   int label;
   struct node *next;
};

main()
{
   int ch = 0;
   int k;
   struct node *h, *temp, *head;

   /* Head node construction */
   head = (struct node*) malloc(sizeof(struct node));
   head->next = NULL;

   while(1)
   {
      printf("\n Stack using Linked List \n");
      printf("1->Push ");
      printf("2->Pop  ");
      printf("3->View ");
      printf("4->Exit \n");
      printf("Enter your choice : ");
      scanf("%d", &ch);

      switch(ch)
      {
         case 1:
            /* Create a new node */
            temp=(struct node *)(malloc(sizeof(struct node)));
            printf("Enter label for new node : ");
            scanf("%d", &temp->label);
            h = head;
            temp->next = h->next;
            h->next = temp;
            break;

         case 2:
            /* Delink the first node */
            h = head->next;
            head->next = h->next;
```

```
               printf("Node %s deleted\n", h->label);
               free(h);
               break;

            case 3:
               printf("\n HEAD -> ");
               h = head;
               /* Loop till last node */
               while(h->next != NULL)
               {
                  h = h->next;
                  printf("%d -> ",h->label);
               }
               printf("NULL \n");
               break;

            case 4:
               exit(0);
         }
      }
   }
```

Output

```
 Stack using Linked List
1->Push  2->Pop  3->View  4->Exit
Enter your choice : 1
Enter label for new node : 23
New node added

 Stack using Linked List
1->Push  2->Pop  3->View  4->Exit
Enter your choice : 1
Enter label for new node : 34

 Stack using Linked List
1->Push 2->Pop 3->View 4->Exit
Enter your choice : 3
HEAD -> 34 -> 23 -> NULL
```

DEPARTMENT OF IT		
AIM & ALGORITHM	02	
PROGRAM	03	
EXECUTION & OUTPUT	03	
VIVA VOCE	02	
TOTAL	**10**	

Result

Thus the C program to implement Stack ADT by using linked list is executed successfully and the output is verified.

Ex. No. 3a **Array implementation of Queue ADT**

Date:

Aim

To write a C program to implement Queue ADT by using arrays.

Algorithm

1. Create a Queue[ ] with MAX size as your wish.
2. Write function for all the basic operations of stack - Enqueue(), Dequeue() and Display().
3. By using Switch case, select Enqueue() operation to insert element in to the rear/back end of the queue.
   - Check whether queue is FULL. (rear == SIZE-1)
   - If it is FULL, then display "Queue is FULL!!! Insertion is not possible!!!"and terminate the function.
   - If it is NOT FULL, then increment rear value by one (rear++) and set queue[rear] = value
4. Similarly, by using Switch case, select Dequeue() function is used to removethe element from the front end of the queue.
   - Check whether queue is EMPTY. (front == rear)
   - 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 frontand rear to '-1' (front = rear =-1).
5. Similarly, by using Switch case, select display() operation to display all elementof the 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 3: Display 'queue[i]' value and increment 'i' value by one (i++). Repeatthe same until 'i' value is equal to rear (i <= rear)
6. Close the program

Program

```
/* Queue Operation using Arrays */

#include <stdio.h>
#include <conio.h>

#define max 5

static int queue[max];
int front = -1;
int rear  = -1;

void insert(int x)
{
   queue[++rear] = x;
   if (front == -1)
      front = 0;
}

int remove()
{
   int val;
   val = queue[front];
   if (front==rear && rear==max-1)
      front = rear = -1;
   else
      front ++;
   return (val);
}

void view()
{
   int i;

   if (front == -1)
      printf("\n Queue Empty \n");
   else
   {
      printf("\n Front-->");
      for(i=front; i<=rear; i++)
         printf("%4d", queue[i]);
      printf("  <--Rear\n");
   }
}

main()
{
   int ch= 0,val;
   clrscr();
```

```
    while(ch != 4)
    {
       printf("\n QUEUE OPERATION \n");
       printf("1.INSERT ");
       printf("2.DELETE ");
       printf("3.VIEW ");
       printf("4.QUIT\n");
       printf("Enter Choice : ");
       scanf("%d", &ch);

       switch(ch)
       {
          case 1:
             if(rear < max-1)
             {
                printf("\n Enter element to be inserted : ");
                scanf("%d", &val);
                insert(val);
             }
             else
                printf("\n Queue Full \n");
             break;
          case 2:
             if(front == -1)
                printf("\n Queue Empty \n");
             else
             {
                val = remove();
                printf("\n Element deleted : %d \n", val);
             }
             break;
          case 3:
             view();
             break;
          case 4:
             exit(0);
          default:
             printf("\n Invalid Choice \n");
       }
    }
}
```

Output

```
 QUEUE OPERATION
1.INSERT  2.DELETE  3.VIEW  4.QUIT
Enter Choice : 1

Enter element to be inserted : 12

 QUEUE OPERATION
1.INSERT  2.DELETE  3.VIEW  4.QUIT
Enter Choice : 1

Enter element to be inserted : 23

 QUEUE OPERATION
1.INSERT  2.DELETE  3.VIEW  4.QUIT
Enter Choice : 1

Enter element to be inserted : 34

 QUEUE OPERATION
1.INSERT  2.DELETE  3.VIEW  4.QUIT
Enter Choice : 1

Enter element to be inserted : 45

 QUEUE OPERATION
1.INSERT  2.DELETE  3.VIEW  4.QUIT
Enter Choice : 1

Enter element to be inserted : 56

 QUEUE OPERATION
1.INSERT  2.DELETE  3.VIEW  4.QUIT
Enter Choice : 1

Queue Full

 QUEUE OPERATION
1.INSERT  2.DELETE  3.VIEW  4.QUIT
Enter Choice : 3

Front-->  12  23  34  45  56  <--Rear
```

DEPARTMENT OF IT		
AIM & ALGORITHM	**02**	
PROGRAM	**03**	
EXECUTION & OUTPUT	**03**	
VIVA VOCE	**02**	
TOTAL	**10**	

Result:

Thus the C program to implement Queue ADT by using array is executed successfully and the output is verified.

Ex. No. 3b **Pointer Based / Linked List Implementation of Queue ADT**

Date:

Aim

To write a C program to implement Queue ADT by using linked list

Algorithm

1. Include all the header files which are used in the program. And declare all the user defined functions.
2. Define a 'Node' structure with two members data and next.
3. Define two Node pointers 'front' and 'rear' and set both to NULL.
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.
5. enqueue(value) - Inserting an element into the queue
    - Create a newNode with given value and set 'newNode → next' to NULL.
    - Check whether queue is Empty (rear == NULL)
    - If it is Empty then, set front = newNode and rear = newNode.
    - If it is Not Empty then, set rear → next = newNode and rear = newNode.
6. dequeue() - Deleting an Element from queue
    - Check whether queue is Empty (front == NULL).
    - If it is Empty, then display "Queue is Empty!!! Deletion is not possible!!!" and terminate from the function
    - If it is Not Empty then, define a Node pointer 'temp' and set it to 'front'.
    - Then set 'front = front → next' and delete 'temp' (free(temp)).
7. display() - Displaying queue of elements
    - Check whether queue is Empty (front == NULL).
    - If it is Empty then, display 'Queue is Empty!!!' and terminate the function.
    - If it is Not Empty then, define a Node pointer 'temp' and initialize with front.
    - Display 'temp → data --->' and move it to the next node. Repeat the same until 'temp' reaches to 'rear' (temp → next != NULL).
    - Finally! Display 'temp → data ---> NULL'.

Program

```
/* Queue using Single Linked List */

#include <stdio.h>
#include <conio.h>
#include <process.h>
#include <alloc.h>

struct node
{
   int label;
   struct node *next;
};

main()
{
   int ch=0;
   int k;
   struct node *h, *temp, *head;

   /* Head node construction */
   head = (struct node*) malloc(sizeof(struct node));
   head->next = NULL;

   while(1)
   {
      printf("\n Queue using Linked List \n");
      printf("1->Insert ");
      printf("2->Delete ");
      printf("3->View ");
      printf("4->Exit \n");
      printf("Enter your choice : ");
      scanf("%d", &ch);

      switch(ch)
      {
         case 1:
            /* Create a new node */
            temp=(struct node *)(malloc(sizeof(struct node)));
            printf("Enter label for new node : ");
            scanf("%d", &temp->label);

            /* Reorganize the links */
            h = head;
            while (h->next != NULL)
               h = h->next;
            h->next = temp;
            temp->next = NULL;
            break;
         case 2:
            /* Delink the first node */
            h = head->next;
            head->next = h->next;
            printf("Node deleted \n");
            free(h);
            break;

         case 3:
            printf("\n\nHEAD -> ");
            h=head;
            while (h->next!=NULL)
            {
               h = h->next;
               printf("%d -> ",h->label);
            }
            printf("NULL \n");
            break;

         case 4:
            exit(0);
      }}}
```

Output

```
 Queue using Linked List
1->Insert 2->Delete 3->View 4->Exit
Enter your choice : 1
Enter label for new node : 12

 Queue using Linked List
1->Insert 2->Delete 3->View 4->Exit
Enter your choice : 1
Enter label for new node : 23

 Queue using Linked List
1->Insert 2->Delete 3->View 4->Exit
Enter your choice : 3
HEAD -> 12 -> 23 -> NULL
```

DEPARTMENT OF IT		
AIM & ALGORITHM	**02**	
PROGRAM	**03**	
EXECUTION & OUTPUT	**03**	
VIVA VOCE	**02**	
TOTAL	**10**	

Result

Thus the C program to implement Queue ADT by using linked list is executed successfully and the output is verified.

Ex. No. 4a **Implementation of Searching Algorithms (Linear Search)**

Date:

Aim

To perform linear search of an element on the given array.

Algorithm

```
1. Read number of array elements $n$
2. Read array elements $A_i$, $i = 0,1,2,\ldots n-1$
3. Read search value
4. Assign 0 to found
5. Check each array element against search
      If $A_i$ = search then
            found = 1
            Print "Element found"
            Print position i
            Stop
6. If found = 0 then
         print "Element not found"
```

Program

```
/* Linear search on a sorted array */

#include <stdio.h>
#include <conio.h>

main()
{
   int a[50],i, n, val, found;
   clrscr();

   printf("Enter number of elements : ");
   scanf("%d", &n);
   printf("Enter Array Elements : \n");
   for(i=0; i<n; i++)
      scanf("%d", &a[i]);

   printf("Enter element to locate : ");
   scanf("%d", &val);
   found = 0;
   for(i=0; i<n; i++)
   {
      if (a[i] == val)
      {
         printf("Element found at position %d", i);
         found = 1;
         break;
      }
   }
   if (found == 0)
      printf("\n Element not found");
   getch();
}
```

Output

```
Enter number of elements : 7
Enter Array Elements :
23  6  12  5  0  32  10
Enter element to locate : 5
Element found at position 3
```

DEPARTMENT OF IT		
AIM & ALGORITHM	**02**	
PROGRAM	**03**	
EXECUTION & OUTPUT	**03**	
VIVA VOCE	**02**	
TOTAL	**10**	

Result

Thus an array was linearly searched for an element's existence.

Ex. No. 4b **Implementation of Searching Algorithms (Binary Search)**

Date:

Aim

To locate an element in a sorted array using Binary search method

Algorithm

```
1. Read number of array elements, say n
2. Create an array arr consisting n sorted elements
3. Get element, say key to be located
4. Assign 0 to lower and n to upper
5. While (lower < upper)
        Determine middle element mid = (upper+lower)/2 If
        key = arr[mid] then
            Print mid
            Stop
        Else if key > arr[mid] then
            lower = mid + 1
        else
            upper = mid – 1

6. Print "Element not found"
```

Program

```
/* Binary Search on a sorted array */

#include <stdio.h>
#include <conio.h>

main()
{
   int a[50],i, n, upper, lower, mid, val, found;
   clrscr();

   printf("Enter array size  : ");
   scanf("%d", &n);
   for(i=0; i<n; i++)
      a[i] = 2 * i;

   printf("\n Elements in Sorted Order \n");
   for(i=0; i<n; i++)
      printf("%4d", a[i]);

   printf("\n Enter element to locate : ");
   scanf("%d", &val);
   upper = n;
   lower = 0;
   found = -1;
   while (lower <= upper)
   {
      mid = (upper + lower)/2;
      if (a[mid] == val)
      {
         printf("Located at position %d", mid);
         found = 1;
         break;
      }
      else if(a[mid] > val)
         upper = mid - 1;
      else
         lower = mid + 1;
   }

   if (found == -1)
      printf("Element  not found");
   getch();
}
```

Output

```
Enter array size : 9
 Elements in Sorted Order
   0   2   4   6   8  10  12  14  16
 Enter element to locate : 12
 Located at position 6

Enter array size : 10
 Elements in Sorted Order
   0   2   4   6   8  10  12  14  16  18
 Enter element to locate : 13
 Element not found
```

DEPARTMENT OF IT		
AIM & ALGORITHM	**02**	
PROGRAM	**03**	
EXECUTION & OUTPUT	**03**	
VIVA VOCE	**02**	
TOTAL	**10**	

Result

Thus an element is located quickly using binary search method.

Ex. No. 4c **Implementation of Sorting Algorithms**

(Insertion Sort)

Date:

Aim

To sort an array of N numbers using Insertion sort.

Algorithm

1. Read number of array elements *n*
2. Read array elements A_i
3. Sort the elements using insertion sort

   In pass p, move the element in position p left until its correct place is found among the first p + 1 elements.

   Element at position p is saved in temp, and all larger elements (prior to position p) are moved one spot to the right. Then temp is placed in the correct spot.

Program

```
/* Insertion Sort */

#include <stdio.h>
main()
{
   int i, j, k, n, temp, a[20], p=0;
   printf("Enter total elements: ");
   scanf("%d",&n);
   printf("Enter array elements: ");
   for(i=0; i<n; i++)
      scanf("%d", &a[i]);
   for(i=1; i<n; i++)
   {
      temp = a[i];
      j = i - 1;
      while((temp<a[j]) && (j>=0))
      {
         a[j+1] = a[j];
         j = j - 1;
      }
      a[j+1] = temp;

      p++;
      printf("\n After Pass %d: ", p);
      for(k=0; k<n; k++)
         printf("  %d", a[k]);
   }
   printf("\n Sorted List : ");
   for(i=0; i<n; i++)
      printf(" %d", a[i]);
}
```

Output

```
Enter total elements: 6
Enter array elements: 34 8  64  51  32  21
 After Pass 1:   8  34  64  51  32  21
 After Pass 2:   8  34  64  51  32  21
 After Pass 3:   8  34  51  64  32  21
 After Pass 4:   8  32  34  51  64  21
 After Pass 5:   8  21  32  34  51  64
 Sorted List :  8 21 32 34 51 64
```

DEPARTMENT OF IT		
AIM & ALGORITHM	**02**	
PROGRAM	**03**	
EXECUTION & OUTPUT	**03**	
VIVA VOCE	**02**	
TOTAL	**10**	

Result

Thus array elements were sorted using insertion sort.

Ex. No. 4d **Implementation of Sorting Algorithms**

(Selection Sort)

Date:

Aim

To sort an array of N numbers using Selection sort.

Algorithm

1. Set min to the first location
2. Search the minimum element in the array
3. Swap the first location with the minimum value in the array
4. Assign the second element as min.
5. Repeat the process until we get a sorted array.

.

Program

```
/* Selection Sort */

#include <stdio.h>

void selectionSort(int arr[], int n) {
    int i, j, minIndex, temp;

    for (i = 0; i < n - 1; i++) {
        minIndex = i;

        // Find the minimum element in unsorted array
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }

        // Swap the found minimum element with the first element
        if (minIndex != i) {
            temp = arr[i];
            arr[i] = arr[minIndex];
            arr[minIndex] = temp;
        }
    }
}

void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[100], n;

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

    printf("Enter %d integers:\n", n);
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    printf("Original array:\n");
    printArray(arr, n);

    selectionSort(arr, n);

    printf("Sorted array (Ascending Order):\n");
    printArray(arr, n);

    return 0;
}
```

Output:

```
Enter number of elements: 5
Enter 5 integers:
34 12 5 66 1
Original array:
34 12 5 66 1
Sorted array (Ascending Order):
1 5 12 34 66
```

DEPARTMENT OF IT		
AIM & ALGORITHM	**02**	
PROGRAM	**03**	
EXECUTION & OUTPUT	**03**	
VIVA VOCE	**02**	
TOTAL	**10**	

Result

Thus array elements were sorted using Selection sort.

Ex. No. 4e **Implementation of Sorting Algorithms (Merge Sort)**

Date:

Aim

To sort an array of N numbers using Merge sort.

Algorithm

1. Read number of array elements *n*
2. Read array elements A_i
3. Divide the array into sub-arrays with a set of elements
4. Recursively sort the sub-arrays
5. Merge the sorted sub-arrays onto a single sorted array**.**

Program

```
/* Merge sort */

#include <stdio.h>
#include <conio.h>

void merge(int [],int ,int ,int );
void part(int [],int ,int );
int size;

main()
{
   int i, arr[30];
   printf("Enter total no. of elements : ");
   scanf("%d", &size);
   printf("Enter array elements : ");
   for(i=0; i<size; i++)
      scanf("%d", &arr[i]);
   part(arr, 0, size-1);
   printf("\n Merge sorted list : ");
   for(i=0; i<size; i++)
      printf("%d ",arr[i]);
   getch();
}

void part(int arr[], int min, int max)
{
   int i, mid;
   if(min < max)
   {
      mid = (min + max) / 2;
      part(arr, min, mid);
      part(arr, mid+1, max);
      merge(arr, min, mid, max);
   }
   if  (max-min  == (size/2)-1)
   {
      printf("\n Half sorted list : ");
      for(i=min; i<=max; i++)
         printf("%d ", arr[i]);
   }
}

void merge(int arr[],int min,int mid,int max)
{
   int tmp[30];
   int i, j, k, m;
   j = min;
   m = mid + 1;
   for(i=min; j<=mid && m<=max; i++)
   {
      if(arr[j] <= arr[m])
      {
         tmp[i] = arr[j];
         j++;
      }
      else
      {
         tmp[i] = arr[m];
         m++;
      }
   }
   if(j > mid)
   {
      for(k=m; k<=max; k++)
      {
         tmp[i] = arr[k];
         i++;
      }
   }
   else
   {
      for(k=j; k<=mid; k++)
      {
         tmp[i] = arr[k];
         i++;
      }
   }
   for(k=min; k<=max; k++)
      arr[k] = tmp[k];
}
```

Output

```
Enter total no. of elements : 8
Enter array elements : 24 13 26 1 2 27 38 15

 Half sorted list : 1 13 24 26
 Half sorted list : 2 15 27 38
 Merge sorted list : 1 2 13 15 24 26 27 38
```

DEPARTMENT OF IT		
AIM & ALGORITHM	**02**	
PROGRAM	**03**	
EXECUTION & OUTPUT	**03**	
VIVA VOCE	**02**	
TOTAL	**10**	

Result

Thus array elements were sorted using merge sort's divide and conquer method.

Ex. No. 5 **Implementation of Binary Tree Traversal**

Date:

Aim

To implement Binary tree Traversal (Preorder, Inorder, and Postorder traversals) using C.

Algorithm

1. Define Node Structure

- Create a structure Node with members:
- int data for storing data
- struct Node* left for left child
- struct Node* right for right child

2. Create Node

- Write a function createNode(data) that:
- Allocates memory
- Initializes data and sets left and right to NULL
- Returns
- pointer to new node

3
. Build Binary Tree

- Manually create and link nodes to form a sample binary tree structure.

4. Preorder Traversal

- Define preorder(node):
- If node is not NULL:
- Print node->data
- Call preorder(node->left)
- Call preorder(node->right)

5. Inorder Traversal

- Define inorder(node):
- If node is not NULL:
- Call inorder(node->left)
- Print node->data
- Call inorder(node->right)

6. Postorder Traversal

- Define postorder(node):
- If node is not NULL:
- Call postorder(node->left)
- Call postorder(node->right)
- Print node->data

7. Main Function

- Create root and child nodes
- Call and display the results of all three traversal methods

Program

```
/* Binary Tree Traversal */

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

// Define the structure for a binary tree node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*) malloc(sizeof(struct
Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Preorder Traversal (Root, Left, Right)
void preorder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}

// Inorder Traversal (Left, Root, Right)
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

// Postorder Traversal (Left, Right, Root)
void postorder(struct Node* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->data);
    }
}

// Main function
int main() {
            Construct the following binary tree
    struct Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);
    root->right->left = createNode(6);
    root->right->right = createNode(7);

    printf("Preorder traversal: ");
    preorder(root);
    printf("\n");

    printf("Inorder traversal: ");
    inorder(root);
    printf("\n");

    printf("Postorder traversal: ");
    postorder(root);
    printf("\n");

    return 0;
}
```

Output

```
Preorder traversal: 1 2 4 5 3 6 7
Inorder traversal: 4 2 5 1 6 3 7
Postorder traversal: 4 5 2 6 7 3 1
```

DEPARTMENT OF IT		
AIM & ALGORITHM	**02**	
PROGRAM	**03**	
EXECUTION & OUTPUT	**03**	
VIVA VOCE	**02**	
TOTAL	**10**	

Result

Thus the program was successfully executed. The Preorder, Inorder, and Postorder traversals of the binary tree were displayed.

Ex. No. 6 **Implementation of Priority Queues using Heap**
Date:

Aim

To build a binary heap from an array of input elements.

Algorithm

1. In a heap, for every node x with parent p, the key in p is smaller than or equal to the key in x.
2. For insertion operation
   - **a.** Add the element to the bottom level of the heap.
   - **b.** Compare the added element with its parent; if they are in the correct order, stop.
   - **c.** If not, swap the element with its parent and return to the previous step.
3. For deleteMin operation
   - **a.** Replace the root of the heap with the last element on the last level.
   - **b.** Compare the new root with its children; if they are in the correct order, stop.
   - **c.** If not, Swap with its smaller child in a min-heap

Program

```
/* Binary Heap */

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

int heap[1000000], heapSize;

void Init()
{
   heapSize = 0;
   heap[0] = -INT_MAX;
}

void Insert(int element)
{
   heapSize++;
   heap[heapSize] = element;
   int now = heapSize;
   while (heap[now / 2] > element)
   {
      heap[now] = heap[now / 2];
      now /= 2;
   }
   heap[now] = element;
}

int DeleteMin()
{
   int minElement, lastElement, child, now;
   minElement = heap[1];
   lastElement = heap[heapSize--];
   for (now = 1; now * 2 <= heapSize; now = child)
   {
      child = now * 2;
      if (child != heapSize && heap[child + 1] < heap[child])
         child++;
      if (lastElement > heap[child])
         heap[now] = heap[child];
      else
         break;
   }
   heap[now] = lastElement;
   return minElement;
}

main()
{
   int number_of_elements;
   printf("Program to demonstrate Heap:\nEnter the number of
elements: ");
        scanf("%d", &number_of_elements);
        int iter, element;
        Init();
        printf("Enter the elements: ");
        for (iter = 0; iter < number_of_elements; iter++)
        {
            scanf("%d", &element);
            Insert(element);
        }
        for (iter = 0; iter < number_of_elements; iter++)
            printf("%d ", DeleteMin());
        printf("\n");
    }
```

Output

```
Program to demonstrate Heap:
Enter the number of elements: 6
Enter the elements: 3 2 15 5 4 45

2 3 4 5 15 45
```

DEPARTMENT OF IT		
AIM & ALGORITHM	**02**	
PROGRAM	**03**	
EXECUTION & OUTPUT	**03**	
VIVA VOCE	**02**	
TOTAL	**10**	

Result

Thus a binary heap is constructed for the given elements.

Ex. No. 7a **Implementation of Shortest Path Algorithm (Dijkstra's Algorithm)**

Date:

Aim

To find the shortest path for the given graph from a specified source to all other vertices using Dijkstra's algorithm.

Algorithm

1. Obtain no. of vertices and adjacency matrix for the given graph
2. Create cost matrix from adjacency matrix. C[i][j] is the cost of going from vertex i to vertex j. If there is no edge between vertices i and j then C[i][j] is infinity
3. Initialize visited[] to zero
4. Read source vertex and mark it as visited
5. Create the distance matrix, by storing the cost of vertices from vertex no. 0 to n-1 from the source vertex
   ```
   distance[i]=cost[0][i];
   ```
6. Choose a vertex w, such that distance[w] is minimum and visited[w] is 0. Mark visited[w] as 1.
7. Recalculate the shortest distance of remaining vertices from the source.
8. Only, the vertices not marked as 1 in array visited[ ] should be considered for recalculation of distance. i.e. for each vertex v
   ```
   if(visited[v]==0)
       distance[v]=min(distance[v]
       distance[w]+cost[w][v])
   ```

Program

```
/* Dijkstra's Shortest Path */

#include <stdio.h>
#include <conio.h>

#define INFINITY 9999
#define MAX 10

void dijkstra(int G[MAX][MAX], int n, int startnode);

main()
{
    int G[MAX][MAX], i, j, n, u;
    printf("Enter no. of vertices: ");
    scanf("%d", &n);
    printf("Enter the adjacency matrix:\n");
    for(i=0; i<n; i++)
        for(j=0; j<n; j++)
            scanf("%d", &G[i][j]);
    printf("Enter the starting node: ");
    scanf("%d", &u);
    dijkstra(G, n, u);
}

void dijkstra(int G[MAX][MAX], int n,int startnode)
{
    int cost[MAX][MAX], distance[MAX], pred[MAX];
    int visited[MAX],count, mindistance, nextnode, i, j;
    for(i=0; i<n; i++)
        for(j=0; j<n; j++)
            if(G[i][j] == 0)
                cost[i][j] = INFINITY;
            else
                cost[i][j] = G[i][j];
    for(i=0; i<n; i++)
    {
        distance[i] = cost[startnode][i];
        pred[i] = startnode;
        visited[i] = 0;
    }
    distance[startnode] = 0;
    visited[startnode] = 1;
    count = 1;
    while(count < n-1)
    {
        mindistance = INFINITY;
        for(i=0; i<n; i++)
            if(distance[i] < mindistance && !visited[i])
            {
                mindistance = distance[i];
                nextnode=i;
            }
            visited[nextnode] = 1;
            for(i=0; i<n; i++)
                if(!visited[i])
                    if(mindistance + cost[nextnode][i] <
                                                distance[i])
                    {
                        distance[i] = mindistance +
                                          cost[nextnode][i];
                        pred[i] = nextnode;
                    }
        count++;
    }
    for(i=0; i<n; i++)
        if(i != startnode)
        {
            printf("\nDistance to node%d = %d", i,
                                              distance[i]);
            printf("\nPath = %d", i);
            j = i;
            do
            {
                j = pred[j];
                printf("<-%d", j);
            } while(j != startnode);
    }
}
```

Output

```
Enter no. of vertices: 5
Enter  the  adjacency matrix:
0    10   0   30   100
10   0    50  0    0
0    50   0   20   10
30   0    20  0    60
100  0    0   60   0
Enter the starting node: 0
Distance to node1 = 10
Path = 1<-0
Distance to node2 = 50
Path = 2<-3<-0

Distance to node3 = 30
Path = 3<-0
Distance to node4 = 60
Path = 4<-2<-3<-0
```

DEPARTMENT OF IT		
AIM & ALGORITHM	**02**	
PROGRAM	**03**	
EXECUTION & OUTPUT	**03**	
VIVA VOCE	**02**	
TOTAL	**10**	

Result

Thus Dijkstra's algorithm is used to find shortest path from a given vertex.

Ex. No. 7b **Implementation of Shortest Path Algorithm (Floyd’s Algorithm)**

Date:

Aim:

To implement Floyd’s Algorithm in C to find the shortest path between all pairs of vertices in a weighted graph.

Algorithm:

1. Initialize the distance matrix dist[][] same as the input graph's adjacency matrix.

2. For each intermediate vertex k from 0 to V–1, repeat:

   - For each source vertex i from 0 to V–1:
   - For each destination vertex j from 0 to V–1:
   - If dist[i][k] + dist[k][j] < dist[i][j], then update dist[i][j] = dist[i][k] + dist[k][j].

3. After all iterations, dist[i][j] contains the shortest distance from vertex i to j.

4. Display the final distance matrix.

Program

```
/*Implementation of Shortest Path Algorithm(Floyd's Algorithm)*/

#include <stdio.h>
#define INF 99999  // A value to represent infinity
#define V 4         // Number of vertices in the graph

// Function to print the solution matrix
void printSolution(int dist[V][V]) {
    printf("Shortest distances between every pair of vertices:\n");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (dist[i][j] == INF)
                printf("%7s", "INF");
            else
                printf("%7d", dist[i][j]);
        }
        printf("\n");
    }
}

// Function to implement Floyd-Warshall algorithm
void floydWarshall(int graph[V][V]) {
    int dist[V][V];

    // Initialize the solution matrix same as input graph matrix
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            dist[i][j] = graph[i][j];

    // Add all vertices one by one to the set of intermediate
vertices
    for (int k = 0; k < V; k++) {
        // Pick all vertices as source
        for (int i = 0; i < V; i++) {
            // Pick all vertices as destination
            for (int j = 0; j < V; j++) {
                // Update dist[i][j] if k is on the shortest path
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }

    // Print the shortest distance matrix
    printSolution(dist);
}

int main() {
    // Adjacency matrix representation of the graph
    int graph[V][V] = {
        {0,   5,  INF, 10},
        {INF, 0,   3,  INF},
        {INF, INF, 0,   1},
        {INF, INF, INF, 0}
    };

    // Run Floyd-Warshall Algorithm
    floydWarshall(graph);
    return 0;

}
```

Output:

```
Shortest distances between every pair of vertices:
     0      5      8      9
    INF     0      3      4
    INF    INF     0      1
    INF    INF    INF     0
```

DEPARTMENT OF IT		
AIM & ALGORITHM	**02**	
PROGRAM	**03**	
EXECUTION & OUTPUT	**03**	
VIVA VOCE	**02**	
TOTAL	**10**	

Result

Thus Floyd algorithm is used to find Shortest Path from a given vertex.

Ex. No. 8a **Implementation of Minimum Spanning Tree (Prim’s Algorithm)**

Date:

Aim

To find the Minimum Spanning Tree for the given graph from a specified source to all other vertices using Prim’s algorithm.

Algorithm

1. Create edge list of given graph, with their weights.
2. Draw all nodes to create skeleton for spanning tree.
3. Select an edge with lowest weight and add it to skeleton and delete edge from edge list.
4. Add other edges. While adding an edge take care that the one end of the edge should always be in the skeleton tree and its cost should be minimum.
5. Repeat step 5 until n-1 edges are added.

Program

```
/* Prim's Minimum Spanning Tree */

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

#define infinity 9999
#define MAX 20

int G[MAX][MAX],spanning[MAX][MAX],n;

int prims();

int main()
{
int i,j,total_cost;
printf("Enter no. of vertices:");
scanf("%d",&n);
printf("\nEnter the adjacency matrix:\n");
for(i=0;i<n;i++)
for(j=0;j<n;j++)
scanf("%d",&G[i][j]);
total_cost=prims();
printf("\nspanning tree matrix:\n");
for(i=0;i<n;i++)
{
printf("\n");
for(j=0;j<n;j++)
printf("%d\t",spanning[i][j]);
}
printf("\n\nTotal cost of spanning tree=%d",total_cost);
return 0;
}

int prims()
{
int cost[MAX][MAX];
int u,v,min_distance,distance[MAX],from[MAX];
int visited[MAX],no_of_edges,i,min_cost,j;
//create cost[][] matrix,spanning[][]
for(i=0;i<n;i++)
for(j=0;j<n;j++)
{
if(G[i][j]==0)
cost[i][j]=infinity;
else
cost[i][j]=G[i][j];
spanning[i][j]=0;
}
//initialise visited[],distance[] and from[]
distance[0]=0;
visited[0]=1;
for(i=1;i<n;i++)
{
distance[i]=cost[0][i];
from[i]=0;
visited[i]=0;
}
min_cost=0; //cost of spanning tree
no_of_edges=n-1; //no. of edges to be added
while(no_of_edges>0)
{
//find the vertex at minimum distance from the tree
min_distance=infinity;
for(i=1;i<n;i++)
if(visited[i]==0&&distance[i]<min_distance)
{
v=i;
min_distance=distance[i];
}
u=from[v];
//insert the edge in spanning tree
spanning[u][v]=distance[v];
spanning[v][u]=distance[v];
no_of_edges--;
visited[v]=1;
//updated the distance[] array
for(i=1;i<n;i++)
if(visited[i]==0&&cost[i][v]<distance[i])
{
distance[i]=cost[i][v];
from[i]=v;
}
min_cost=min_cost+cost[u][v];
}
return(min_cost);
}
```

Output:

```
Enter no. of vertices:6
Enter the adjacency matrix:
0 3 1 6 0 0
3 0 5 0 3 0
1 5 0 5 6 4
6 0 5 0 0 2
0 3 6 0 0 6
0 0 4 2 6 0

spanning tree matrix:
0 3 1 0 0 0
3 0 0 0 3 0
1 0 0 0 0 4
0 0 0 0 0 2
0 3 0 0 0 0
0 0 4 2 0 0
Total cost of spanning tree=13
```

DEPARTMENT OF IT		
AIM & ALGORITHM	02	
PROGRAM	03	
EXECUTION & OUTPUT	03	
VIVA VOCE	02	
TOTAL	10	

Result

Thus Prim's algorithm is used to find Minimum Spanning Tree from a given vertex.

Ex. No. 8b **Implementation of Minimum Spanning Tree (Kruskal’s Algorithm)**

Date:

Aim:

To implement Kruskal’s Algorithm in C for finding the Minimum Spanning Tree (MST) of a given connected weighted graph.

Algorithm:

1. Create a list of all edges in the graph with their weights.

2. Sort all edges in **non-decreasing order** of their weights.

3. Initialize a set for each vertex (using union-find).

4. For each edge in sorted order:

    - If including the edge doesn’t form a cycle (i.e., vertices are in different sets),
        - Include the edge in the MST.
        - Union the sets of the two vertices.

5. Repeat step 5 until MST contains **(V – 1)** edges.

6. Display the MST and its total cost.

Program

```
/* Kruskal’s Minimum Spanning Tree */

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

#define MAX 100

// Structure to represent an edge
struct Edge {
    int src, dest, weight;
};

// Structure to represent a subset for union-find
struct Subset {
    int parent;
    int rank;
};

// Function to find the parent (with path compression)
int find(struct Subset subsets[], int i) {
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);
    return subsets[i].parent;
}

// Function to do union of two sets (by rank)
void Union(struct Subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;
    else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Compare function for qsort
int compareEdges(const void* a, const void* b) {
    struct Edge* e1 = (struct Edge*)a;
    struct Edge* e2 = (struct Edge*)b;
    return e1->weight - e2->weight;
}

// Kruskal's Algorithm
void kruskalMST(struct Edge edges[], int V, int E) {
    struct Edge result[MAX];
    int e = 0;
    int i = 0;

    qsort(edges, E, sizeof(edges[0]), compareEdges);
    struct Subset *subsets = (struct Subset*) malloc(V *
sizeof(struct Subset));
    for (int v = 0; v < V; ++v) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    while (e < V - 1 && i < E) {
        struct Edge next = edges[i++];

        int x = find(subsets, next.src);
        int y = find(subsets, next.dest);

        if (x != y) {
            result[e++] = next;
            Union(subsets, x, y);
        }
    }

    printf("Edges in the Minimum Spanning Tree:\n");
    int totalCost = 0;
    for (i = 0; i < e; ++i) {
        printf("%d -- %d == %d\n", result[i].src, result[i].dest, 
result[i].weight);
        totalCost += result[i].weight;
    }

    printf("Total cost of MST = %d\n", totalCost);
}

int main() {
    int V = 4;  // Number of vertices
    int E = 5;  // Number of edges
    struct Edge edges[] = {
        {0, 1, 10},
        {0, 2, 6},
        {0, 3, 5},
        {1, 3, 15},
        {2, 3, 4}
    };

    kruskalMST(edges, V, E);

    return 0;

}
```

Output:

```
Edges in the Minimum Spanning Tree:
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Total cost of MST = 19
```

DEPARTMENT OF IT		
AIM & ALGORITHM	**02**	
PROGRAM	**03**	
EXECUTION & OUTPUT	**03**	
VIVA VOCE	**02**	
TOTAL	**10**	

Result: The program successfully implements Kruskal's Algorithm and generates a Minimum Spanning Tree for the given weighted graph. The MST includes the edges with minimum total weight without forming any cycles.