EX:NO:1
DATE:

TOWERS OF HANOI USING RECURSION

Aim:

To write a C program to implement Towers of Hanoi Using recursion

Algorithm:

1. move top n-1 disks from A to B
2. move bottom disks from A to C
3. move n-1 disks from B to C using A

Program:

```
void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)
{
   if (n == 1)
   {
      printf("\n Move disk 1 from rod %c to rod %c", from_rod, to_rod);
      return;
   }
   towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
   printf("\n Move disk %d from rod %c to rod %c", n, from_rod, to_rod);
   towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
}

int main()
{
   int n = 4; // Number of disks
   towerOfHanoi(n, \'A\', \'C\', \'B\');  // A, B and C are names of rods
   return 0;
}
```

EX:NO:2
DATE:

ARRAY IMPLEMENTATION OF LIST ADT

Aim:

To write a C program to implement List ADT using an array

Algorithm for main program:

Step 1: Include all the header files required for the program.
Step 2: Declare an array to store the elements
Step 3: Display the Menu
1.Create
2.Insert
3.Delete
4.Display
5.Exit
Step 4: Read the choice
Step 5: If the choice is 1, read the value of n and read the elements of the list
Step 6: If the choice is 2, read the element to be inserted and position where the element to be inserted and call the function insert()
Step 7: If the choice is 3, Read the element to be deleted and call the function delet().
Step 8: If the choice is 4, Call the function Display() to print all the elements
Step 9: If **the** choice is 5, Quit the program.

Algorithm for Insert()

Step 1: Shift all the elements one position to the right until the position where the element to be inserted
Step 2: Insert the element ie List[pos]=Value.
Step 3: Increment the value of n.

Algorithm for Delet()

Step 1: Shift all the elements one position to the left from the position of the element to be deleted
Step 2: Decrement the value of n

Program:

```
#include<stdio.h>
#include<conio.h>
#define SIZE 10

void insert(int,int);
void delet(int);
void display();

int List[SIZE],n,i;
```

```
void main()
{
  int value,p,choice;
  clrscr();
  while(1){
    printf("\n\n***** MENU *****\n");
    printf("1.Create\n 2. Insert\n3. Delete\n4. Display\n5. Exit");
    printf("\nEnter your choice: ");
    scanf("%d",&choice);
    switch(choice){
        case 1:
              printf("Enter the to total number of elements(n)\n");
              scanf("%d",&n);
              printf("enter the elements!!!!\n");
              for(i=0;i<n;i++)
                scanf("%d",&List[i]);
        case 2: printf("Enter the value to be inserted: ");
              scanf("%d",&value);
              printf("Enter the position to be inserted");
              scanf("%d",&p);
              insert(value,p);
              n=n+1;
              break;
        case 3:
              printf("Enter the element to be deleted");
              scanf("%d",&value);
              delet(value);
              n=n-1;
               break;
        case 4: display();
               break;
        case 5: exit(0);
        default: printf("\nWrong selection!!! Try again!!!");
    }
  }
}
void insert(int value,int pos){

  if(n== SIZE-1)
    printf("\nList is Full!!! Insertion is not possible!!!");
  else{
    for(i=n-1;i>=pos;i--)
     List[i+1]=List[i];
    List[i+1]=value;
    printf("\nInsertion success!!!");
  }
}
void delet(int val)
{
  int pos;
```

```
  if(n==0)
    printf("\nList is Empty!!! Deletion is not possible!!!");
  else{
          for(i=0;i<n;i++)
            {
             if(List[i]==val)
             {
              pos=i;
              break;
             }
            }
          for(i=pos;i<n;i++)
           List[i]=List[i+1];
  }
}
void display(){
  if(n==0)
    printf("\nList is Empty!!!");
  else{
    int i;
    printf("\nList elements are:\n");
    for(i=0;i<n;i++)
          printf("%d\n",List[i]);
  }
}
```

EX:NO:3
DATE:

LINKED LIST IMPLMENTATION OF LIST

Aim:

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

Algorithm:

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 4: Implement the **main** method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Inserting At Beginning of the 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

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)

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**'

Deleting from Beginning of the 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

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

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

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<conio.h>
#include<stdlib.h>

void insertAtBeginning(int);
void insertAtEnd(int);
void insertBetween(int,int,int);
void display();
void removeBeginning();
void removeEnd();
void removeSpecific(int);

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

void main()
{
  int choice,value,choice1,loc1,loc2;
  clrscr();
  while(1){
  mainMenu: 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(value);
                            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);
  }
}
```

EX:NO:4
DATE:

IMPLMENTATION OF DOUBLY LINKED LIST

Aim :

Write a program to implement doubly linked list

Algorithm :

Inserting At Beginning of the 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

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)

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).

Deleting from Beginning of the 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

- **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

- **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<conio.h>
void insertAtBeginning(int);
void insertAtEnd(int);
void insertAtAfter(int,int);
void deleteBeginning();
void deleteEnd();
void deleteSpecific(int);
void display();

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

void main()
{
  int choice1, choice2, value, location;
  clrscr();
  while(1)
  {
    printf("\n********** MENU **********\n");
    printf("1. Insert\n2. Delete\n3. Display\n4. Exit\nEnter your choice: ");
    scanf("%d",&choice1);
    switch()
    {
      case 1: printf("Enter the value to be inserted: ");
              scanf("%d",&value);
          while(1)
          {
              printf("\nSelect from the following Inserting options\n");
```

```
                    printf("1. At Beginning\n2. At End\n3. After a Node\n4. Cancel\nEnter your
choice: ");
                scanf("%d",&choice2);
                switch(choice2)
                {
                  case 1:  insertAtBeginning(value);
                           break;
                  case 2:  insertAtEnd(value);
                           break;
                  case 3:  printf("Enter the location after which you want to insert: ");
                           scanf("%d",&location);
                           insertAfter(value,location);
                           break;
                  case 4:  goto EndSwitch;
                  default: printf("\nPlease select correct Inserting option!!!\n");
                }
              }
       case 2: while(1)
              {
                   printf("\nSelect from the following Deleting options\n");
                   printf("1. At Beginning\n2. At End\n3. Specific Node\n4. Cancel\nEnter your
choice: ");
                scanf("%d",&choice2);
                switch(choice2)
                {
                  case 1:  deleteBeginning();
                           break;
                  case 2:  deleteEnd();
                           break;
                  case 3:  printf("Enter the Node value to be deleted: ");
                           scanf("%d",&location);
                           deleteSpecic(location);
                           break;
                  case 4:  goto EndSwitch;
                  default: printf("\nPlease select correct Deleting option!!!\n");
                }
              }
              EndSwitch: break;
       case 3: display();
          break;
       case 4: exit(0);
       default: printf("\nPlease select correct option!!!");
     }
   }
}

void insertAtBeginning(int value)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
```

```
    newNode -> data = value;
    newNode -> previous = NULL;
    if(head == NULL)
    {
      newNode -> next = NULL;
      head = newNode;
    }
    else
    {
      newNode -> next = head;
      head = newNode;
    }
    printf("\nInsertion success!!!");
}
void insertAtEnd(int value)
{
  struct Node *newNode;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode -> data = value;
  newNode -> next = NULL;
  if(head == NULL)
  {
    newNode -> previous = NULL;
    head = newNode;
  }
  else
  {
    struct Node *temp = head;
    while(temp -> next != NULL)
      temp = temp -> next;
    temp -> next = newNode;
    newNode -> previous = temp;
  }
  printf("\nInsertion success!!!");
}
void insertAfter(int value, int location)
{
  struct Node *newNode;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode -> data = value;
  if(head == NULL)
  {
    newNode -> previous = newNode -> next = NULL;
    head = newNode;
  }
  else
  {
    struct Node *temp1 = head, temp2;
    while(temp1 -> data != location)
    {
```

```
        if(temp1 -> next == NULL)
        {
          printf("Given node is not found in the list!!!");
          goto EndFunction;
        }
        else
        {
          temp1 = temp1 -> next;
        }
      }
      temp2 = temp1 -> next;
      temp1 -> next = newNode;
      newNode -> previous = temp1;
      newNode -> next = temp2;
      temp2 -> previous = newNode;
      printf("\nInsertion success!!!");
    }
    EndFunction:
}
void deleteBeginning()
{
    if(head == NULL)
      printf("List is Empty!!! Deletion not possible!!!");
    else
    {
      struct Node *temp = head;
      if(temp -> previous == temp -> next)
      {
        head = NULL;
        free(temp);
      }
      else{
        head = temp -> next;
        head -> previous = NULL;
        free(temp);
      }
      printf("\nDeletion success!!!");
    }
}
void deleteEnd()
{
    if(head == NULL)
      printf("List is Empty!!! Deletion not possible!!!");
    else
    {
      struct Node *temp = head;
      if(temp -> previous == temp -> next)
      {
        head = NULL;
        free(temp);
```

```
    }
    else{
      while(temp -> next != NULL)
        temp = temp -> next;
      temp -> previous -> next = NULL;
      free(temp);
    }
    printf("\nDeletion success!!!");
  }
}
void deleteSpecific(int delValue)
{
  if(head == NULL)
    printf("List is Empty!!! Deletion not possible!!!");
  else
  {
    struct Node *temp = head;
    while(temp -> data != delValue)
    {
      if(temp -> next == NULL)
      {
        printf("\nGiven node is not found in the list!!!");
        goto FuctionEnd;
      }
      else
      {
        temp = temp -> next;
      }
    }
    if(temp == head)
    {
      head = NULL;
      free(temp);
    }
    else
    {
      temp -> previous -> next = temp -> next;
      free(temp);
    }
    printf("\nDeletion success!!!");
  }
  FuctionEnd:
}
void display()
{
  if(head == NULL)
    printf("\nList is Empty!!!");
  else
  {
    struct Node *temp = head;
```

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

EX:NO:5
DATE:

ARRAY IMPLEMENTATION OF STACK

Aim:

To write a C program to implement stack using an array

Algorithm:

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.

Algorithm for push(value) - Inserting value into 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**).

Algorithm for pop() - Delete a value 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--**).

Algorithm for display() - Displays 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]);
  }
}
```

EX:NO:6
DATE:

INFIX TO POSTFIX CONVERSION

Aim:

To write a C program to convert infix expression to postfix expression using stack

Algorithm:

1. Scan X from left to right and repeat Step 3 to 6 for each element of X until the Stack is empty.
2. If an operand is encountered, add it to Y.
3. If a left parenthesis is encountered, push it onto Stack.
4. If an operator is encountered , then:
    1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) which has the same precedence as or higher precedence than operator.
    2. Add operator to Stack.
    [End of If]
5. If a right parenthesis is encountered ,then:
    1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) until a left parenthesis is encountered.
    2. Remove the left Parenthesis.

Program:

```
#define SIZE 50
#include <ctype.h>
char s[SIZE];
int top=-1;       /* Global declarations */

push(char elem)
{                    /* Function for PUSH operation */
   s[++top]=elem;
}

char pop()
{                    /* Function for POP operation */
   return(s[top--]);
}

int pr(char elem)
{               /* Function for precedence */
   switch(elem)
   {
   case '#': return 0;
   case '(': return 1;
   case '+':
   case '-': return 2;
   case '*':
   case '/': return 3;
   }
```

```
}

main()
{                       /* Main Program */
   char infix[50],postfix[50],ch,elem;
   int i=0,k=0;
   printf("\n\nEnter Infix Expression : ");
   scanf("%s",infix);
   push('#');
   while( (ch=infix[i++]) != '\0')
   {
      if( ch == '(') push(ch);
      else
         if(isalnum(ch)) postfix[k++]=ch;
         else
            if( ch == ')')
            {
               while( s[top] != '('
                  postfix[k++]=pop();
               elem=pop(); /* Remove ( */
            }
            else
            {       /* Operator */
               while( pr(s[top]) >= pr(ch) )
                  postfix[k++]=pop();
               push(ch);
            }
   }
   while( s[top] != '#')     /* Pop from stack till empty */
      postfix[k++]=pop();
   postfix[k]='\0';          /* Make postfix as valid string */
   printf("\nPostfix Expression =  %s\n",postfix);
}
```

EX:NO:7
DATE:

POSTFIX EXPRESSION EVALUATION

Aim:

To write a C program to evaluate postfix expression using stack

Algorithm:

1) Create a stack to store operands (or values).
2) Scan the given expression and do following for every scanned element.
   a) If the element is a number, push it into the stack
   b) If the element is a operator, pop operands for the operator from stack. Evaluate the operator and push the result back to the stack
3) When the expression is ended, the number in the stack is the final answer

Program:

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

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

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

void main()
{
	char exp[20];
	char *e;
	int n1,n2,n3,num;
	printf("Enter the expression :: ");
	scanf("%s",exp);
	e = exp;
	while(*e != '\0')
	{
		if(isdigit(*e))
		{
			num = *e - 48;
			push(num);
		}
		else
		{
			n1 = pop();
```

```
                n2 = pop();
                switch(*e)
                {
                        case '+':
                        {
                                n3 = n1 + n2;
                        break;
                        }
                        case '-':
                        {
                                n3 = n2 - n1;
                                break;
                        }
                        case '*':
                        {
                                n3 = n1 * n2;
                                break;
                        }
                        case '/':
                        {
                                n3 = n2 / n1;
                                break;
                        }
                }
                push(n3);
        }
        e++;
    }
    printf("\nThe result of expression %s  =  %d\n\n",exp,pop());
    getch();

}
```

EX:NO:8
DATE:

BINARY TREE TRAVERALS

Aim:

To write a C program to implement binary tree traversals

Algorithm:

Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Algorithm Preorder(tree)

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

Algorithm Postorder(tree)

1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

Program:

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

struct node
{
   int key;
   struct node *left;
   struct node *right;
};

//return a new node with the given value
struct node *getNode(int val)
{
   struct node *newNode;

   newNode = malloc(sizeof(struct node));

   newNode->key   = val;
   newNode->left  = NULL;
```

```
    newNode->right = NULL;

    return newNode;
}

//inserts nodes in the binary search tree
struct node *insertNode(struct node *root, int val)
{
     if(root == NULL)
         return getNode(val);

     if(root->key < val)
         root->right = insertNode(root->right,val);

     if(root->key > val)
         root->left = insertNode(root->left,val);

     return root;
}

//inorder traversal of the binary search tree
void inorder(struct node *root)
{
    if(root == NULL)
        return;

    //traverse the left subtree
    inorder(root->left);

    //visit the root
    printf("%d ",root->key);

    //traverse the right subtree
    inorder(root->right);
}

int main()
{
   struct node *root = NULL;


    int data;
    char ch;
        /*  Do while loop to display various options to select from to decide the input  */
        do
        {
            printf("\nSelect one of the operations::");
            printf("\n1. To insert a new node in the Binary Tree");
            printf("\n2. To display the nodes of the Binary Tree(via Inorder Traversal).\n");
```

```
        int choice;
        scanf("%d",&choice);
        switch (choice)
        {
        case 1 :
           printf("\nEnter the value to be inserted\n");
           scanf("%d",&data);
           root = insertNode(root,data);
           break;
        case 2 :
           printf("\nInorder Traversal of the Binary Tree::\n");
           inorder(root);
           break;
        default :
           printf("Wrong Entry\n");
           break;
        }

        printf("\nDo you want to continue (Type y or n)\n");
        scanf(" %c",&ch);
     } while (ch == 'Y'|| ch == 'y');

  return 0;
}
```

EX:NO:9
DATE:

IMPLEMENTATION OF BINARY SEARCH TREE

Aim :

To write a C program to implement binary search tree and its operations

Algorithm:

Search Operation in BST

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 matching, then display "Given node found!!!" and terminate the function
Step 4: If both are not matching, 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 found exact element or we completed with a leaf node
Step 8: If we reach to the node with search value, then display "Element is found" and terminate the function.
Step 9: If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.

Insertion Operation in BST

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

Deletion Operation in BST

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

Case 3: Deleting a node with two children

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

Program:

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

struct treeNode {
    int data;
    struct treeNode *left, *right;
};

struct treeNode *root = NULL;

/* create a new node with the given data */
struct treeNode* createNode(int data) {
    struct treeNode *newNode;
    newNode = (struct treeNode *) malloc(sizeof (struct treeNode));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return(newNode);
}

/* insertion in binary search tree */
void insertion(struct treeNode **node, int data) {
    if (*node == NULL) {
        *node = createNode(data);
    } else if (data < (*node)->data) {
        insertion(&(*node)->left, data);
    } else if (data > (*node)->data) {
        insertion(&(*node)->right, data);
    }
}


/* deletion in binary search tree */
void deletion(struct treeNode **node, struct treeNode **parent, int
data) {
    struct treeNode *tmpNode, *tmpParent;
    if (*node == NULL)
        return;
```

```
if ((*node)->data == data) {
        /* deleting the leaf node */
        if (!(*node)->left && !(*node)->right) {
                if (parent) {
                        /* delete leaf node */
                        if ((*parent)->left == *node)
                                (*parent)->left = NULL;
                        else
                                (*parent)->right = NULL;
                        free(*node);
                } else {
                        /* delete root node with no children */
                        free(*node);
                }
        /* deleting node with one child */
        } else if (!(*node)->right && (*node)->left) {
                /* deleting node with left child alone */
                tmpNode = *node;
                (*parent)->right = (*node)->left;
                free(tmpNode);
                *node = (*parent)->right;
        } else if ((*node)->right && !(*node)->left) {
                /* deleting node with right child alone */
                tmpNode = *node;
                (*parent)->left = (*node)->right;
                free(tmpNode);
                (*node) = (*parent)->left;
        } else if (!(*node)->right->left) {
                        tmpNode = *node;
                (*node)->right->left = (*node)->left;
                (*parent)->left = (*node)->right;
                free(tmpNode);
                *node = (*parent)->left;
        } else {
                tmpNode = (*node)->right;
                while (tmpNode->left) {
                        tmpParent = tmpNode;
                        tmpNode = tmpNode->left;
                }
                tmpParent->left = tmpNode->right;
                tmpNode->left = (*node)->left;
                tmpNode->right =(*node)->right;
                free(*node);
                *node = tmpNode;
        }
} else if (data < (*node)->data) {
        /* traverse towards left subtree */
        deletion(&(*node)->left, node, data);
} else if (data > (*node)->data) {
        /* traversing towards right subtree */
```

```
            deletion(&(*node)->right, node, data);
        }
}

/* search the given element in binary search tree */
void findElement(struct treeNode *node, int data) {
        if (!node)
                return;
        else if (data < node->data) {
                findElement(node->left, data);
        } else if (data > node->data) {
                findElement(node->right, data);
        } else
                printf("data found: %d\n", node->data);
        return;

}

void traverse(struct treeNode *node) {
        if (node != NULL) {
                traverse(node->left);
                printf("%3d", node->data);
                traverse(node->right);
        }
        return;
}

int main() {
        int data, ch;
        while (1) {
                printf("1. Insertion in Binary Search Tree\n");
                printf("2. Deletion in Binary Search Tree\n");
                printf("3. Search Element in Binary Search Tree\n");
                printf("4. Inorder traversal\n5. Exit\n");
                printf("Enter your choice:");
                scanf("%d", &ch);
                switch (ch) {
                        case 1:
                                while (1) {
                                printf("Enter your data:");
                                scanf("%d", &data);
                                insertion(&root, data);
                                printf("Continue Insertion(0/1):");
                                scanf("%d", &ch);
                                if (!ch)
                                        break;
                                }
                                break;
                        case 2:
                                printf("Enter your data:");
```

```
                scanf("%d", &data);
                deletion(&root, NULL, data);
                break;
            case 3:
                printf("Enter value for data:");
                scanf("%d", &data);
                findElement(root, data);
                break;
            case 4:
                printf("Inorder Traversal:\n");
                traverse(root);
                printf("\n");
                break;
            case 5:
                exit(0);
            default:
                printf("u've entered wrong option\n");
                break;
        }
    }
    return 0;
}
```

EX:NO:10
DATE: **BUBBLE SORT**

Aim: To write a C program to implement bubble sort algorithm

Algorithm:

Step 1: Starting with the first element(index = 0), compare the current element with the next element of the array.

Step 2: If the current element is greater than the next element of the array, swap them.

Step 3:If the current element is less than the next element, move to the next element. **Repeat Step 1**.

Program:

```
#include <stdio.h>
void bubbleSort(int arr[], int n)
{
   int i, j, temp;
   for(i = 0; i < n; i++)
   {
      for(j = 0; j < n-i-1; j++)
      {
         if( arr[j] > arr[j+1])
         {
            // swap the elements
            temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
         }
      }
   }
      // print the sorted array
   printf("Sorted Array: ");
   for(i = 0; i < n; i++)
   {
      printf("%d  ", arr[i]);
   }
}
int main()
{
   int arr[100], i, n, step, temp;
   // ask user for number of elements to be sorted
   printf("Enter the number of elements to be sorted: ");
   scanf("%d", &n);
   // input elements if the array
   for(i = 0; i < n; i++)
   {
      printf("Enter element no. %d: ", i+1);
      scanf("%d", &arr[i]);
   }
   // call the function bubbleSort
```

```
    bubbleSort(arr, n);
        return 0;
}
```

EX:NO:11
DATE: **QUICK SORT**

Aim: To write a C program to implement quick sort

Algorithm:

Step 1: Pick an element from the array, this element is called as pivot element
Step 2: Divide the unsorted array of elements in two arrays with values less than the pivot come in the first sub array, while all elements with values greater than the pivot come in the second sub-array (equal values can go either way). This step is called the partition operation.
Step 3: Recursively repeat the step 2(until the sub-arrays are sorted) to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

Program:

```
#include<stdio.h>
void quicksort(int number[25],int first,int last){
   int i, j, pivot, temp;

   if(first<last){
      pivot=first;
      i=first;
      j=last;

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

      temp=number[pivot];
      number[pivot]=number[j];
      number[j]=temp;
      quicksort(number,first,j-1);
      quicksort(number,j+1,last);

   }
}

int main(){
   int i, count, number[25];

   printf("How many elements are u going to enter?: ");
   scanf("%d",&count);
```

```
  printf("Enter %d elements: ", count);
  for(i=0;i<count;i++)
    scanf("%d",&number[i]);

  quicksort(number,0,count-1);

  printf("Order of Sorted elements: ");
  for(i=0;i<count;i++)
    printf(" %d",number[i]);

  return 0;
}
```

EX:NO:12 **SHORTEST PATH ALGORITHM(DIJKSTRA'S)**

DATE:

Aim: To write a C program to Dijkstra's shortest path algorithm

Algorithm:

Step 1:Create a set sptSet (shortest path tree set) that keeps track of vertices included in shortest

path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.

Step 2:Assign a distance value to all vertices in the input graph. Initialize all distance values as

INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

Step3:While sptSet doesn't include all vertices

a) Pick a vertex u which is not there in sptSet and has minimum distance value.

.b) Include u to sptSet.

c) Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v

Program:

```
#include<stdio.h>
#include<conio.h>
#define INFINITY 9999
#define MAX 10
void dijkstra(int G[MAX][MAX],int n,int startnode);
int main()
{
 int G[MAX][MAX],i,j,n,u;
 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]);

 printf("\nEnter the starting node:");
 scanf("%d",&u);
 dijkstra(G,n,u);

 return 0;
}
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;

 //pred[] stores the predecessor of each node
 //count gives the number of nodes seen so far
 //create the cost matrix
 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];

 //initialize pred[],distance[] and visited[]
 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;
//nextnode gives the node at minimum distance
for(i=0;i<n;i++)
if(distance[i]<mindistance&&!visited[i])
{
mindistance=distance[i];
nextnode=i;
}
//check if a better path exists through nextnode
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++;
}
//print the path and distance of each node
```

```
for(i=0;i<n;i++)
if(i!=startnode)
{
printf("\nDistance of node%d=%d",i,distance[i]);
printf("\nPath=%d",i);

j=i;
do
{
j=pred[j];
printf("<-%d",j);
}while(j!=startnode);
}
}
```

EX:NO: 13

DATE :

FIND MINIMUM SPANNING TREE USING KRUSKAL'S ALGORITHM

Aim :

Write a C programs to find minimum spanning tree using Kruskal's Algorithm

Algorithm

- Step 1: Sort all edges in increasing order of their edge weights.
- Step 2: Pick the smallest edge.
- Step 3: Check if the new edge creates a cycle or loop in a spanning tree.
- Step 4: If it doesn't form the cycle, then include that edge in MST. Otherwise, discard it.
- Step 5: Repeat from step 2 until it includes |V| - 1 edges in MST.

Program:

```
#include <stdio.h>
   #include <conio.h>
   #include <stdlib.h>
   int i,j,k,a,b,u,v,n,ne=1;
   int min,mincost=0,cost[9][9],parent[9];
   int find(int);
   int uni(int,int);
   void main()
   {
         printf("\n\tImplementation of Kruskal's Algorithm\n");
         printf("\nEnter the no. of vertices:");
         scanf("%d",&n);
         printf("\nEnter the cost adjacency matrix:\n");
         for(i=1;i<=n;i++)
         {
         for(j=1;j<=n;j++)
         {
         scanf("%d",&cost[i][j]);
         if(cost[i][j]==0)
         cost[i][j]=999;
         }
         }
```

```
printf("The edges of Minimum Cost Spanning Tree are\n");
        while(ne < n)
        {
        for(i=1,min=999;i<=n;i++)
        {
        for(j=1;j <= n;j++)
        {
        if(cost[i][j] < min)
        {
        min=cost[i][j];
        a=u=i;
        b=v=j;
        }
        }
        }
        u=find(u);
        v=find(v);
        if(uni(u,v))
        {
        printf("%d edge (%d,%d) =%d\n",ne++,a,b,min);
        mincost +=min;
        }
        cost[a][b]=cost[b][a]=999;
        }
        printf("\n\tMinimum cost = %d\n",mincost);
        getch();
    }
    int find(int i)
    {
        while(parent[i])
        i=parent[i];
        return i;
    }
    int uni(int i,int j)
    {
        if(i!=j)
        {
        parent[j]=i;
        return 1;
        }
        return 0;
    }
```