GOVERNMENT OF KERALA

DEPARTMENT OF TECHNICAL EDUCATION

RAJIV GANDHI INSTITUTE OF TECHNOLOGY

(GOVT. ENGINEERING COLLEGE)

KOTTAYAM -686501

RECORD BOOK

GOVERNMENT OF KERALA

DEPARTMENT OF TECHNICAL EDUCATION

RAJIV GANDHI INSTITUTE OF TECHNOLOGY

(GOVT. ENGINEERING COLLEGE)

KOTTAYAM -686501

20MCA135 ADVANCED DATA STRUCTURE LAB RECORD

Name :.....

Branch :.....

Semester :.....Roll No. :.....

CERTIFIED BONAFIDE RECORD WORK DONE BY

Reg No.....

KOTTAYAM STAFF IN CHARGE

INDEX

LAB CYCLE: 1
EXPERIMENT NO:1
DATE:18/11/2021

AIM:
Program to Merge two sorted arrays and store in a third array.

ALGORITHM:

Step1:read the size of 2 arrays
Step2:read the elements of both array as per size
Step3:for all elements of array
    If array[i] > array[i+1]
        swap(array[i], array[i+1])
    end if
    end for
Step4:Create an array arr3[] of size n1 + n2.
    Simultaneously traverse arr1[] and arr2[].
    Pick smaller of the current elements in arr1[] and arr2[], copy this smaller element to the next position in arr3[] and move ahead in arr3[] and the array whose element is picked.
    If there are remaining elements in arr1[] or arr2[], copy them also in arr3[]
step5:print arr3

SOURCE CODE:

```
#include<stdio.h>
void main(){
    int m,n,i,j,k,temp;
    printf("\n Enter the size of the first array: ");
    scanf("%d",&m);
    printf("\n Enter the size of the second array: ");
    scanf("%d",&n);
    int num1[m],num2[n],num3[m+n];
    printf("\n Enter the elements of the first array: ");
    for(i=0;i<m;i++)
        scanf("%d",&num1[i]);
    for(i=0;i<m-1;i++){
        for(j=0;j<m-i-1;j++){
            if(num1[j]>num1[j+1]){
                temp=num1[j];
                num1[j]=num1[j+1];
                num1[j+1]=temp;
            }
        }
    }
    printf("\n Sorted first array: ");
```

```
	for(i=0;i<m;i++)
		printf("%d\t",num1[i]);

	printf("\n Enter the elements of the second array: ");
	for(i=0;i<n;i++)
		scanf("%d",&num2[i]);
	for(i=0;i<n-1;i++){
		for(j=0;j<n-i-1;j++){
			if(num2[j]>num2[j+1]){
				temp=num2[j];
				num2[j]=num2[j+1];
				num2[j+1]=temp;
			}
		}
	}
	printf("\n Sorted second array: ");
	for(i=0;i<n;i++)
		printf("%d\t",num2[i]);
	i=j=k=0;
	while(i<m&&j<n){
		if(num1[i]<num2[j]){
			num3[k]=num1[i];
			i++;
		}
		else{
			num3[k]=num2[j];
			j++;
		}
		k++;
	}
	while(i<m){
		num3[k]=num1[i];
		i++;
		k++;
	}
	while(j<n){
		num3[k]=num2[j];
		j++;
		k++;
	}
	printf("\n Merged array is: ");
	for(i=0;i<(m+n);i++)
		printf("%d\t",num3[i]);
}
```

OUTPUT:

```
Enter the size of the first array: 5

Enter the size of the second array: 3

Enter the elements of the first array: 90
-2
34
55
12

Sorted first array: -2 12      34      55      90
Enter the elements of the second array: 45
900
-3

Sorted second array: -3        45      900
Merged array is: -3    -2      12      34      45      55      90      900
```

RESULT:

Program is successfully executed and output is obtained.

LAB CYCLE: 1
EXPERIMENT NO:2
DATE:18/11/2021

AIM:
Program to perform circular queue operations
a.Insertion
b.Deletion
c.Search

ALGORITHM:
Insert
```
step1:if (front=(rear+1)%SZ)
                print("Queue is full");
                goto step4
        endif
 step2:. if(front=-1)
                front=0
        endif
step3:  rear=(rear+1)%SZ
        q[rear]=x
step4:stop
```

Delete
```
step1:if (front=-1)
          Print "Queue is empty";
          goto step4
    endif
 step2:print “Deleted Element”, q[front];
  step3:if (front=rear)
                front=rear=-1;
        else
                front=(front+1)%SZ;
        endif
step4.stop
```

Search:
```
step1:if (front==-1)
          print("Queue is empty")
                goto step4
    endif
step2: f=front;     pos=1;
step3: while(True)
           if (q[f]=x)
                print "Element found at location ",pos
                goto step4
          endif
           if (f=rear)
                print("Element not found");
                goto step4
           endif
```

```
            f=(f+1)%SZ;

            pos=pos+1;
          end while
step4:stop
```

SOURCE CODE:

```
#include<stdio.h>
#include<stdlib.h>
# define SIZE 5
int queue_arr[SIZE];
int rear=-1,front=-1;
int isfull(){
        if((front==0&&rear==SIZE-1)||(front==rear+1)){
                return 1;
        }
        return 0;

}
int isempty(){
        if(front==-1){
                return 1;
        }
        return 0;

}
int enqueue(int item){
        if(isfull()){
                printf("\n Queue overflow");
        }
        else{
                if(front==-1)
                        front=rear=0;
                else{
                        if(rear==SIZE-1)
                                rear=0;
                        else
                                rear=(rear+1)%SIZE;
                }
                queue_arr[rear]=item;
                printf("\n Inserted successfully");
        }
}
int dequeue(){
        if(isempty()){
                printf("\n Queue underflow");
        }
        else{
                printf("\nDeleted element is : %d",queue_arr[front]);
                if(front==rear)
                        front=rear=-1;
```

```
                else{
                        if(front==SIZE-1)
                                front=0;

                        else
                                front++;
                }
        }
}
int display(){
        int pos_1,pos_2;
        pos_1=front;pos_2=rear;
        if(front==-1)
                printf("\nQueue is empty");
        else{
                printf("\nQueue elements are: \t");
                if(pos_1<=pos_2){
                        while(pos_1<=pos_2){
                                printf("%d\t",queue_arr[pos_1]);
                                pos_1++;
                        }
                }
                else{
                        while(pos_1<=SIZE-1){
                                printf("%d\t",queue_arr[pos_1]);
                                pos_1++;
                        }
                        pos_1=0;
                        while(pos_1<=pos_2){
                                printf("%d\t",queue_arr[pos_1]);
                                pos_1++;
                        }

                }
        }
}

int search_elem(int element){
        int i,flag=0;
        for(i=0;i<SIZE;i++){
                if(queue_arr[i]==element){
                        flag=1;
                        break;
                }
        }
        if(flag==1)
                printf("\nElement found at position %d",i+1);
        else
                printf("\nNot found");
}
int main(){
        int choice,item;
```

```
do{
        printf("\n1.Insert \n 2.Delete \n 3.Search \n 4.Display \n 5.Exit");
        printf("\nEnter your choice: ");
        scanf("%d",&choice);
        switch(choice){
            case 1:printf("\nEnter the element to be inserted: ");
                scanf("%d",&item);
                enqueue(item);
                break;
            case 2:dequeue();
                break;
            case 3:printf("\nEnter the search element: ");
                scanf("%d",&item);
                search_elem(item);
                break;
            case 4:display();break;
            case 5:exit(0);break;
            default:printf("\nWrong choice"); }
    }while(choice!=0);
    return 0;  }
```

OUTPUT:

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

Enter the element to be inserted: 10

 Inserted successfully
1.Insert
 2.Delete
 3.Search
 4.Display
 5.Exit
Enter your choice: 1

Enter the element to be inserted: 5

 Inserted successfully
1.Insert
 2.Delete
 3.Search
 4.Display
 5.Exit
Enter your choice: 1

Enter the element to be inserted: 34

 Inserted successfully
1.Insert
 2.Delete
 3.Search
 4.Display
 5.Exit
Enter your choice: 1

Enter the element to be inserted: 78

 Inserted successfully
1.Insert
 2.Delete
 3.Search
 4.Display
 5.Exit
Enter your choice: 4

Queue elements are:      10       5        34       78
```

RESULT:

Program is successfully executed and output is obtained.

LAB CYCLE: 1
EXPERIMENT NO:3
DATE:2/12/2021

AIM:
Program to perform stack operations using linked list
a.push
b.pop
c.search

ALGORITHM:
Push:
Step 1 : Create a newNode with given value.
Step 2 : Check whether stack is Empty (top == NULL)

Step 3 : If it is Empty, then set newNode → next=NULL.

Step 4 : If it is Not Empty, then set newNode → next =top.

Step 5 : Finally, set top=newNode.

Pop:
Step 1 : Check whether stack is Empty(top == NULL).
Step 2 : If it is Empty, then display "Stack is Empty!!! Deletion is not possible!!!" and terminate the function

Step 3 : If it is Not Empty, then define a Node pointer 'temp' and set it to 'top'.

Step 4 : Then set 'top=top → next'.

Step 5 : Finally, delete 'temp'. (free(temp)).

Search:
Step 1 : Check whether TOP == NULL of Stack.
Step 2 : If TOP == NULL then print “Stack is Empty” and terminate the function
Step 3 : If TOP != NULL, then define a Node pointer ‘temp’ and initialize with top.Also read search element.
Step 4 : Compare temp → data with search element until temp → next != NULL.
Step 5 : If search element is found then print the location.
Step 6 : Else, terminate the function

SOURCE CODE:

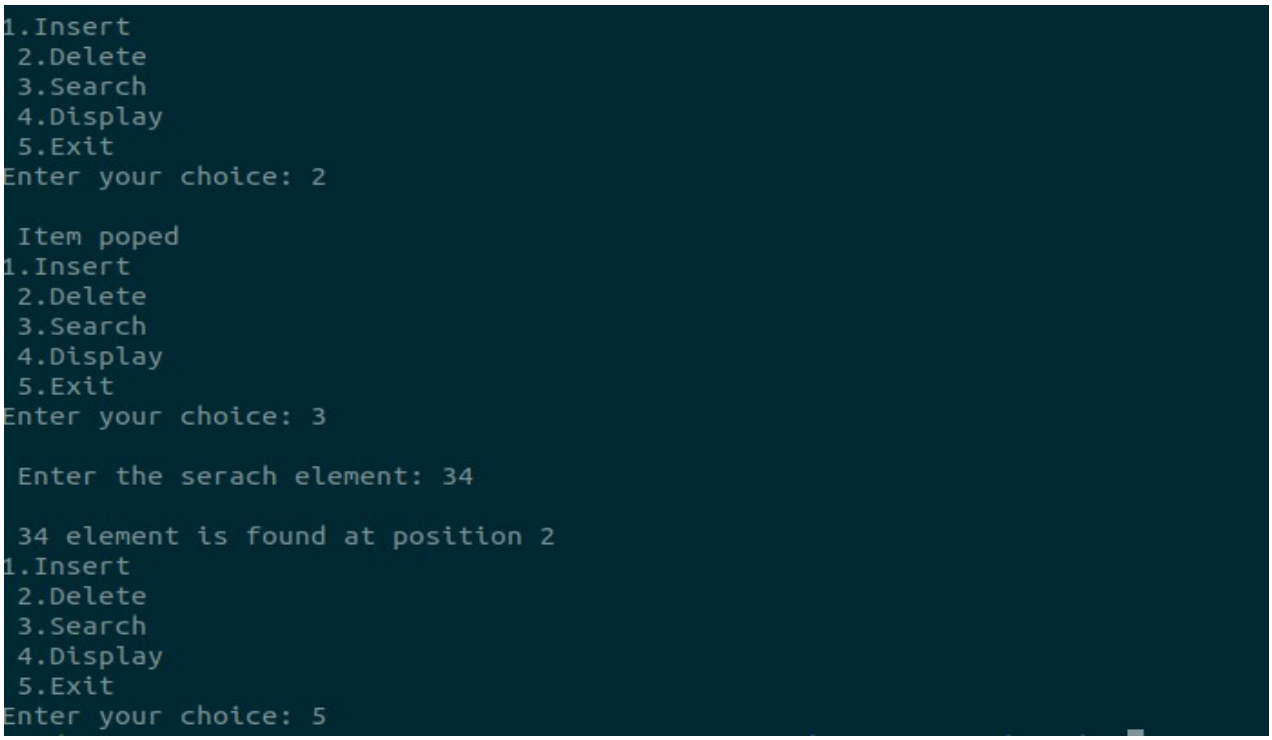
```
#include<stdio.h>
#include<stdlib.h>
struct node{
        int data;
        struct node *next;
}*head=NULL;

void push(){
        int item;
        struct node *ptr=(struct node*)malloc(sizeof(struct node));
```

```
        if(ptr==NULL)

                printf("\n Memory is not allocated");
        else{
                printf("\n Enter the value to be inserted: ");
                scanf("%d",&item);
                ptr->data=item;
                if(head==NULL)
                        ptr->next=NULL;
                else
                        ptr->next=head;
                head=ptr;
                printf("\n Item pushed");
        }
}
void pop(){
        int item;
        struct node*ptr;
        if(head==NULL)
                printf("\n Stack underflow");
        else{
                item=head->data;
                ptr=head;
                head=head->next;
                free(ptr);
                printf("\n Item poped");
        }}
void search_elem(){
        int item,i=0,flag=0;
        struct node*ptr;
        ptr=head;
        if(ptr==NULL)
                printf("\n Stack is empty");
        else{
                printf("\n Enter the serach element: ");
                scanf("%d",&item);
                while(ptr!=NULL){
                        if(ptr->data==item){
                                printf("\n %d element is found at position %d",item,i+1);
                                flag=1;
                                break;
                        }
                        i++;
                        ptr=ptr->next;
                }
                if(flag==0)
                        printf("\n Not found");          } }

void display(){
        int i;
        struct node*ptr;
        ptr=head;
```

```
        if(ptr==NULL)
                printf("\n Stack is empty ");
        else{
                printf("\n Elements are: ");
                while(ptr!=NULL){
                        printf("%d\t",ptr->data);
                        ptr=ptr->next;
                }}}
int main(){
        int choice;
        do{
                printf("\n1.Insert \n 2.Delete \n 3.Search \n 4.Display \n 5.Exit ");
                printf("\nEnter your choice: ");
                scanf("%d",&choice);
                switch(choice){
                        case 1:push();break;
                        case 2:pop();break;
                        case 3:search_elem();break;
                        case 4:display();break;
                        case 5:exit(0);break;
                        default:printf("\nWrong choice");}
        }while(choice!=0);}
```

OUTPUT:

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

 Enter the value to be inserted: 12

 Item pushed
1.Insert
 2.Delete
 3.Search
 4.Display
 5.Exit
Enter your choice: 1

 Enter the value to be inserted: 34

 Item pushed
1.Insert
 2.Delete
 3.Search
 4.Display
 5.Exit
Enter your choice: 1

 Enter the value to be inserted: 67

 Item pushed
1.Insert
 2.Delete
 3.Search
 4.Display
 5.Exit
Enter your choice: 1

 Enter the value to be inserted: 88

 Item pushed
1.Insert
 2.Delete
 3.Search
 4.Display
 5.Exit
Enter your choice: 4

 Elements are: 88        67       34       12
```



RESULT:

Program is successfully executed and output is obtained.

LAB CYCLE: 1
EXPERIMENT NO:4
DATE:2/12/2021

AIM:
Program to perform doubly linked list operations.
a.Insertion
b.Deletion
c.Search

ALGORITHM:
Insert at Beginning:
Step1 : Start
Step2 : Input the DATA to be inserted
Step3 : Create a new node.
Step4 : NewNode → Data = DATA, NewNode →Lpoint =NULL
Step5 : If START is NULL , NewNode→ Rpoint = NULL
Step6 : Else NewNode → Rpoint = HEAD , HEAD→Lpoint = NewNode
Step7 : HEAD =NewNode
Step8 : Stop

Insert at End:

Step1: Start

Step2 : Input DATA to be inserted
Step3 : Create a NewNode
Step4 : NewNode → DATA = DATA
Step5 : NewNode → RPoint = NULL
Step6 : If (HEAD equal to NULL)
a. HEAD = NewNode
b. NewNode → LPoint=NULL
Step7 : Else
a. TEMP = START
b. While (TEMP → Next not equal to NULL)
TEMP = TEMP → Next
c. TEMP → RPoint = NewNode
d. NewNode → LPoint = TEMP
Step8 : Stop

Insert at Specified Location:

Step1 : Start

Step2 : Input the DATA and POS

Step3 : Initialize TEMP = START; i = 0
Step4 : Repeat the step 4 if (i less than POS) and (TEMP is not equal to NULL)
TEMP = TEMP → RPoint; i = i +1
Step5 : If (TEMP not equal to NULL) and (i equal to POS)
(a) Create a New Node
(b) NewNode → DATA = DATA
(c) NewNode → RPoint = TEMP → RPoint

(d) NewNode → LPoint = TEMP

(e) (TEMP → RPoint) → LPoint = NewNode
(f) TEMP → RPoint = New Node

Step6 : Else
(a) Display “Position NOT found”
Step7 : Stop

Deletion at Beginning:

Step1 : Copy the head node in some temporary node.

Step2 : Make the second node as the head node.

Step3 : The prev pointer of the head node is referenced to NULL.

Step4 : Delete the temporary node.

Deletion at End:

Step1 : Copy the last node to a temporary node.

Step2 : Shift the last node to the second last position.

Step3 : Make the last node's next pointer as NULL.

Step4 : Delete the temporary node.

Deletion at specified position:

```
Step1 : If (HEAD==NULL)
                Print "list is empty”
                 goto step7
Step2 : If (HEAD->DATA==x)
                ptr=HEAD;
          HEAD=ptr->right;
          If (HEAD!=NULL)
                HEAD->left=NULL;
         end if
         free(ptr);
         goto step7
        end if
Step3 : ptr= HEAD;
Step 4. while(ptr->data!=x && ptr->right!=NULL)
                ptr=ptr->right;
        end while
  Step5 : if (ptr->data==x)
                next=ptr->right;
                prev=ptr->left;
                prev->right=ptr->right;
                if (next!=NULL)
                        next->left=ptr->left;
                end if
                free(ptr);
                goto step7
```

end if

Step6 : print(“Element not present in the list")

Step7 : stop

Search:

Step1 : If HEAD== NULL, print "List is empty"
goto step8
[END OF IF]

Step2 : Set PTR = HEAD

Step3 : Set i = 0

Step4 : Repeat step 5 to 7 while PTR != NULL

Step5 : If PTR → data = item , return i
[END OF IF]

Step6 : i = i + 1

Step7 : PTR = PTR → next

Step8 : Exit

Display:

Step1 : If (HEAD=NULL)
Print ‘List Empty’
Goto step4
Endif
Step2 : ptr= HEAD
Step3 : while(ptr!=NULL)
Print ptr->data
Ptr=ptr->right
End while
Step4 : stop

SOURCE CODE:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   struct node *prev;
   struct node *next;
   int data;
};
struct node *head;
void insertion_beginning();
void insertion_last();
void insertion_specified();
void deletion_beginning();
void deletion_last();
```

```
void deletion_specified();
void display();
void search();
void main ()

{
int c ;
   while(1)
   {
    printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random location\n4.Delete from
Beginning\n5.Delete from last\n6.Delete from random location\n7.Search\n8.display\n9.Exit");
       printf("\nEnter your choice?");
       scanf("\n%d",&c);
       switch(c)
       {
          case 1:
          insertion_beginning();
          break;
          case 2:
                insertion_last();
          break;
          case 3:
          insertion_specified();
          break;
         case 4:
          deletion_beginning();
          break;
          case 5:
          deletion_last();
          break;
          case 6:
          deletion_specified();
          break;
          case 7:
          search();
          break;
          case 8:
          display();
          break;
          case 9:
          exit(0);
          break;
          default:
          printf("Please enter valid choice..");
       }
   }
}
void insertion_beginning()
{
  struct node *new;
  int item;
 new = (struct node *)malloc(sizeof(struct node));
```

```
  if(new == NULL)
  {
     printf("memory out of space");
  }
  else
  {

   printf("\nEnter Item value");
   scanf("%d",&item);

  if(head==NULL)
  {
     new->next = NULL;
    new->prev=NULL;
     new->data=item;
     head=new;
  }
  else
  {
     new->data=item;
    new->prev=NULL;
     new->next = head;
     head->next->prev=new;
     head=new;
  }
  printf("\nNode inserted\n");
}

}
void insertion_last()
{
  struct node *new,*temp;
  int item;
  new = (struct node *) malloc(sizeof(struct node));
  if(new == NULL)
  {
     printf("memory out of space");
  }
  else
  {
     printf("\nEnter value");
     scanf("%d",&item);
      new->data=item;
     if(head == NULL)
     {
        new->next = NULL;
       new->prev = NULL;
        head = new;
     }
     else
     {
       temp = head;
```

```
          while(temp->next!=NULL)
          {
             temp = temp->next;
          }
          temp->next = new;
          new ->prev=temp;
         new->next = NULL;
          }

       }
     printf("\nnode inserted\n");
    }
void insertion_specified()
{
  struct node *new,*temp;
  int item,loc,i;
 new= (struct node *)malloc(sizeof(struct node));
  if(new == NULL)
  {
     printf("memory out of space");
  }
  else
  {
     temp=head;
     printf("Enter the location");
     scanf("%d",&loc);
     for(i=1;i< loc-1;i++)
     {
        temp = temp->next;
        if(temp == NULL)
        {
           printf("\n There are less than %d elements", loc);
           return;
        }
     }
   printf("Enter value");
     scanf("%d",&item);
     new->data = item;
     new->next = temp->next;//points new node to node after temp
     new-> prev = temp;//new node points to next
     temp->next = new;//temp points to new node
     temp->next->prev=new;//node after temp points to new node
     printf("\nnode inserted\n");


  }
}
void deletion_beginning()
{
   struct node *new;
   if(head == NULL)
   {
```

```
        printf("\n UNDERFLOW");
    }
    else if(head->next == NULL)
    {
        head = NULL;
        free(head);
        printf("\nnode deleted\n");
    }
    else
    {

        new = head;
        head = head -> next;
        head -> prev = NULL;
        free(new);
        printf("\nnode deleted\n");
    }

}
void deletion_last()
{
    struct node *new;
    if(head == NULL)
    {
        printf("\n UNDERFLOW");
    }
    else if(head->next == NULL)
    {
        head = NULL;
        free(head);
        printf("\nnode deleted\n");
    }
    else
    {
        new = head;
        while(new->next != NULL)
        {           new =new -> next;
        }
        new-> prev -> next = NULL;//secondlast node next null
        free(new);
        printf("\nnode deleted\n");
    }
}
void deletion_specified()
{
    struct node *del, *temp;
    int pos,i;
   printf("enter the position to be deleted");
    scanf("%d", &pos);
i=1;
    del = head;
    while(del ->next != NULL){
```

```
if(i==pos)
  break;
i++;
temp=del;
del=del->next;
}
  temp->next=del->next;
free(del);
   printf("\nnode deleted\n");
}
  void display()
{

    struct node *ptr;
    ptr = head;
    if(ptr==NULL)
    printf("list is empty");
    else{
    printf("\n printing values...\n");

    while(ptr != NULL)
    {
       printf("%d\n",ptr->data);
       ptr=ptr->next;
    }
    }
}
void search()
{
    struct node *ptr;
    int item,i=0,flag;
    ptr = head;
    if(ptr == NULL)
    {
       printf("\nEmpty List\n");
    }
    else
    {
       printf("\nEnter item which you want to search?\n");
       scanf("%d",&item);
       while (ptr!=NULL)
       {
          if(ptr->data == item)
          {
             printf("\nitem found at location %d ",i+1);
             flag=0;
             break;
          }
          else
          {
             flag=1;
          }
```

```
            i++;
            ptr = ptr -> next;
        }
        if(flag==1)
        {
            printf("\nItem not found\n");
        }
    }

}
```

OUTPUT:

```
1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete from random location
7.Search
8.display
9.Exit
Enter your choice?1

Enter Item value12

Node inserted

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete from random location
7.Search
8.display
9.Exit
Enter your choice?2

Enter value30

node inserted

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete from random location
7.Search
8.display
9.Exit
Enter your choice?3
Enter the location2
Enter value25

node inserted
```

```
1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete from random location
7.Search
8.display
9.Exit
Enter your choice?8

 printing values...
12      25      30
1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete from random location
7.Search
8.display
9.Exit
Enter your choice?4

node deleted

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete from random location
7.Search
8.display
9.Exit
Enter your choice?7

Enter item which you want to search?
12

Item not found
```

```
1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete from random location
7.Search
8.display
9.Exit
Enter your choice?2

Enter value50

node inserted

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete from random location
7.Search
8.display
9.Exit
Enter your choice?8

 printing values...
25      30      50
1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete from random location
7.Search
8.display
9.Exit
Enter your choice?6
enter the position to be deleted2

node deleted
```

RESULT:

Program is successfully executed and output is obtained.

LAB CYCLE: 1
EXPERIMENT NO:5
DATE:13/12/2021

AIM:
Program to implement Binary Search Tree (BST) and its operations.
a.Insertion
b.Deletion
c.Search

ALGORITHM:
Insertion:
Step1 : Create a new BST node and assign values to it.
Step2 : insert(node, key)
i) If root == NULL,

return the new node to the calling function.

ii) if root=>data < key

call the insert function with root=>right and assign the return value in root=>right.

root->right = insert(root=>right,key)

iii) if root=>data > key

call the insert function with root->left and assign the return value in root=>left.

root=>left = insert(root=>left,key)

Step3 : Finally, return the original root pointer to the calling function.

Deletion:
Step1: If the node is leaf (both left and right will be NULL), remove the node directly and free its memory.
Step2 : If the node has only right child (left will be NULL), make the node points to the right node and free the node.
Step3: If the node has both left and right child,
i) find the smallest node in the right subtree. say min

ii) make node->data = min

iii) again delete the min node.

Search:
Step1 : Compare the element to be searched with the root element of the tree.

Step2 : If root is matched with the target element, then return the node's location.

Step3 : If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the left subtree.

Step4 : If it is larger than the root element, then move to the right subtree.

Step5 : Repeat the above procedure recursively until the match is found.

Step6 : If the element is not found or not present in the tree, then return NULL.

SOURCE CODE:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int key;
   struct node *left;
   struct node *right;
};
struct node *root = NULL;

struct node *getNewNode(int val)
{
   struct node *newNode = malloc(sizeof(struct node));
   newNode->key   = val;
   newNode->left  = NULL;
   newNode->right = NULL;
   return newNode;
}

struct node *insert(struct node *root, int val)
{
   if(root == NULL)
         return getNewNode(val);
   if(root->key <= val)
      root->right = insert(root->right,val);
   else if(root->key > val)
      root->left = insert(root->left,val);
   return root;
}
void display(struct node *temp,int level)
{
         int i;
         if ( temp!=NULL )
         {
                  display(temp->right, level+1);
                  printf("\n");
                  for (i = 0; i < level; i++)
                           printf("    ");
                  printf("%d", temp->key);
                  display(temp->left, level+1);
         }
}
int getRightMin(struct node *root)
{
   struct node *temp = root;
   while(temp->left != NULL){
```

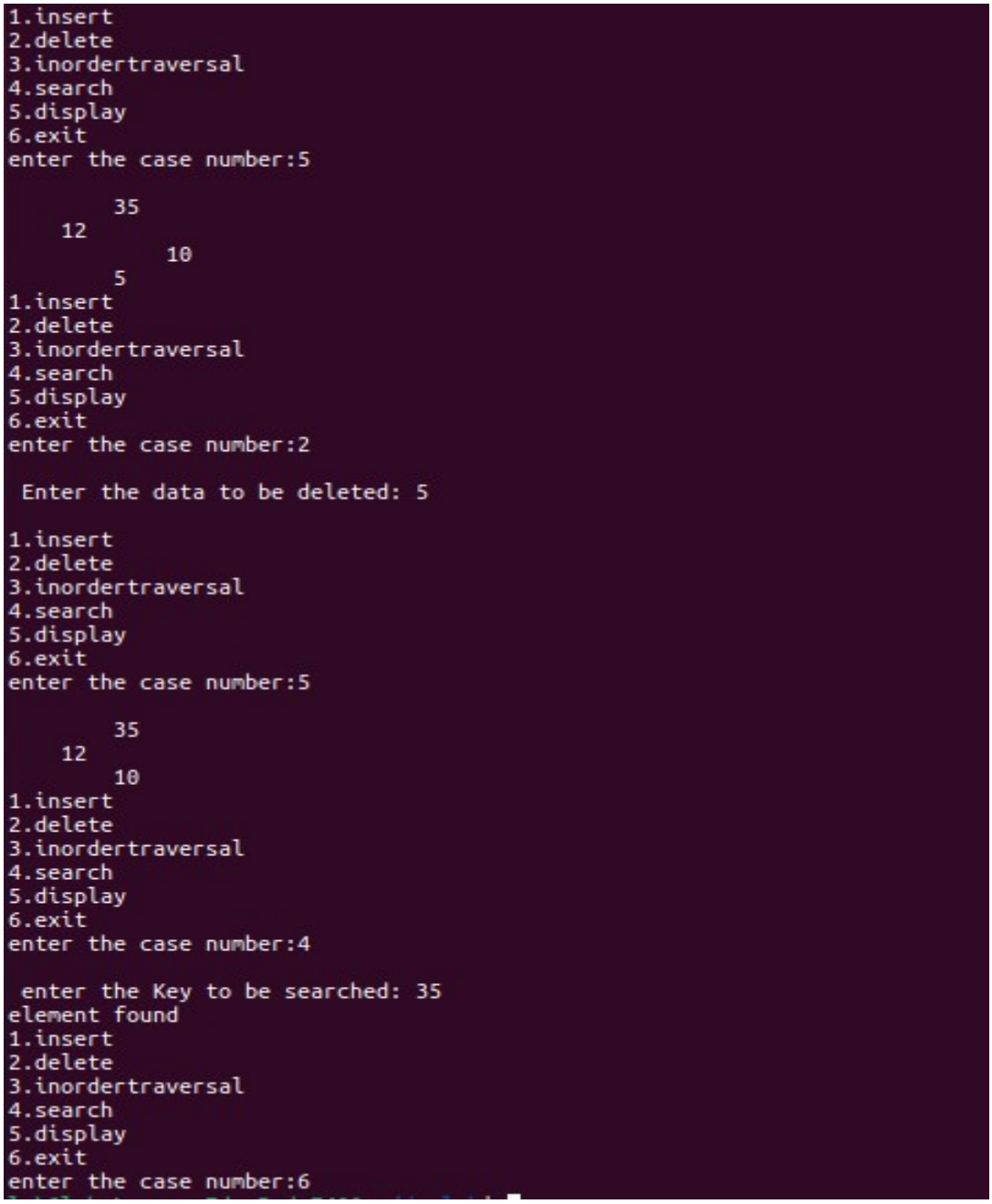
```
            temp = temp->left;}
    return temp->key;
}
void search (struct node *temp,int item)
{

if(item==temp->key)
{
printf("element found");
return;
}
else if((item<temp->key)&&(temp->left==NULL))
{
printf("element not found");
return;
}
else if((item<temp->key)&&(temp->left!=NULL))
{
search(temp->left,item);
}
else if((item>temp->key)&&(temp->right==NULL))
{
printf("element not found");
return;
}
else if((item>=temp->key)&&(temp->right!=NULL))
{
search(temp->right,item);
}
}
struct node *removeNode(struct node *root, int val)
{

   if(root == NULL){
      return NULL;
   }
   if(root->key < val)
      root->right = removeNode(root->right,val);
   else if(root->key > val)
      root->left = removeNode(root->left,val);
   else
   {

      if(root->left == NULL && root->right == NULL)
      {
         free(root);
         return NULL;
      }
      else if(root->left == NULL)
      {
         struct node *temp = root->right;
         free(root);
```

```
            return temp;
        }
        else if(root->right == NULL)
        {
            struct node *temp = root->left;
            free(root);
            return temp;

    }
        else
        {
            int rightMin = getRightMin(root->right);
            root->key = rightMin;
            root->right = removeNode(root->right,rightMin);
        }

    }
    return root;
}
void inorder(struct node *root)
{
    if(root == NULL){
        return;
      }
    inorder(root->left);
    printf("%d ",root->key);
    inorder(root->right);
}
int main()
{
     int item;
     int c;
while(1)
{
printf("\n1.insert\n2.delete\n3.inordertraversal\n4.search\n5.display\n6.exit\n");
printf("enter the case number:");
scanf("%d",&c);
switch(c)
{
case 1:printf("\n Enter the data to be inserted: ");
            scanf("%d",&item);
            root = insert(root,item);
            break;
case 2: printf("\n Enter the data to be deleted: ");
             scanf("%d",&item);
            root = removeNode(root,item);
            break;
case 3:inorder(root);
            break;
case 4:printf("\n enter the Key to be searched: ");
            scanf("%d",&item);
            search(root,item);
```

```
            break;
  case 5:display(root,1);
            break;
  case 6:exit(0);
            break;
  }

  }
     printf("\n");
     return 0;
  }
```

OUTPUT:

```
1.insert
2.delete
3.inordertraversal
4.search
5.display
6.exit
enter the case number:1

 Enter the data to be inserted: 12

1.insert
2.delete
3.inordertraversal
4.search
5.display
6.exit
enter the case number:1

 Enter the data to be inserted: 5

1.insert
2.delete
3.inordertraversal
4.search
5.display
6.exit
enter the case number:1

 Enter the data to be inserted: 35

1.insert
2.delete
3.inordertraversal
4.search
5.display
6.exit
enter the case number:1

 Enter the data to be inserted: 10

1.insert
2.delete
3.inordertraversal
4.search
5.display
6.exit
enter the case number:3
5 10 12 35
```



RESULT:

Program is successfully executed and output is obtained.

LAB CYCLE: 2
EXPERIMENT NO:6
DATE:3/01/2022

AIM:
Program to implement Set Data Structure and set operations (Union, Intersection and Difference) using Bit String.

ALGORITHM:

Step1 : Read the Universal Set
Step2 : Read the set A
Step3 : Read the set B
Step4 : Generate bitstrings of Set A and B
Step5 : Do set union of bitstrings by or operator
Step6 : Do set intersection of bitstrings by and operator
Step7 : Do set difference of bitstrings by (AnB)' operation

SOURCE CODE:

```
#include<stdio.h>
#include<stdlib.h>
int superSet[20],superSetSize=0,
setA[20],setASize=0,
setB[20],setBSize=0,
bitStringA[20],bitStringB[20],bitStringUnion[20],bitStringIntersection[20],bitStringDifference[20
];
int isBitStringReady=0;
void generateBitString(int arr[],int size,int bitStringArray[]);
void printSet(int arr[],int size);
int search(int arr[],int arrSize,int elem);

void setUnion(int arr1[],int arr2[]){
        int i;
   for(i=0;i<superSetSize;i++){
      bitStringUnion[i]=arr1[i]|arr2[i];
   }
}
void setIntersection(int arr1[],int arr2[]){
        int i;
   for(i=0;i<superSetSize;i++){
      bitStringIntersection[i]=arr1[i]&arr2[i];
   }
}
void setDifference(int arr1[],int arr2[]){
        int i;
   for(i=0;i<superSetSize;i++){

      bitStringDifference[i]=arr1[i]&(!arr2[i]);
   }
}
```

```
int checkBitStringStatus(){

if(isBitStringReady==0){
        printf("\n Generate Bit String first!");
        return 0;
    }
    return 1;
}
int getSet(int arr[],int size){
            int i,j,k;
    printf("\nEnter set\n");
    for(i=0;i<size;i++){
        scanf("%d",&arr[i]);
    }
    for(i=0;i<size;i++)
{
    for(j=i+1;j<size;j++)
    {
        if(arr[i]==arr[j])
        {
            for(k=j;k<size-1;k++)
            {
               arr[k]=arr[k+1];
            }
            size--;

            }

        }
        }
    return size;
}

void printSet(int arr[],int size){
            int i;
    printf("{");
    for(i=0;i<size;i++){
        printf("%d",arr[i]);
        if(i!=size-1){
            printf(",");
        }
    }
    printf("}\n");
}

void generateAndPrintBitStrings(){
            int i;
    for(i=0;i<superSetSize;i++){
        bitStringA[i]=0;
        bitStringB[i]=0;
        bitStringUnion[i]=0;
        bitStringIntersection[i]=0;
```

```
        bitStringDifference[i]=0;
    }
    generateBitString(setA,setASize,bitStringA);

   generateBitString(setB,setBSize,bitStringB);
    printf("\nSet A Bit String representation : ");
    printSet(bitStringA,superSetSize);
    printf("\nSet B Bit String representation : ");
    printSet(bitStringB,superSetSize);
    isBitStringReady=1;
}

void generateBitString(int arr[],int size,int bitStringArray[]){
        int i;
    for(i=0;i<size;i++){
        int pos=search(superSet,superSetSize,arr[i]);
        if(pos>=0){
            bitStringArray[pos]=1;
        }
    }
}
int search(int arr[],int arrSize,int elem){
        int i;
    for(i=0;i<arrSize;i++){
        if(arr[i]==elem)
            return i;
    }
    return -1;
    }

void main(){
int choice;

        printf("1.Enter Universal Set\n");
        printf("2.Enter Set A\n");
        printf("3.Enter Set B\n");
        printf("4.Generate Bit Strings\n");
        printf("5.Union\n");
        printf("6.Intersection\n");
        printf("7.Difference\n");
      while(1)
{

        printf("Enter Choice:");
        scanf("%d",&choice);

    switch(choice){
        case 1:
            printf("Enter Super Set Size:");
            scanf("%d",&superSetSize);
            superSetSize=getSet(superSet,superSetSize);
            break;
```

```
            case 2:
                printf("Enter Set A Size:");
                scanf("%d",&setASize);
                getSet(setA,setASize);
                break;

            case 3:
                printf("Enter Set B Size:");
                scanf("%d",&setBSize);
                getSet(setB,setBSize);
                 break;
            case 4:
                printf("Generating bit strings\n");
                generateAndPrintBitStrings();
                 break;
            case 5:
                printf("Set union\n");
                if(checkBitStringStatus()==1)
                {
                   setUnion(bitStringA,bitStringB);
                   printSet(bitStringUnion, superSetSize);
                }
                 break;
            case 6:
                printf("Set Intersection\n");
                if(checkBitStringStatus()==1)
                {
                   setIntersection(bitStringA,bitStringB);
                  printSet(bitStringIntersection, superSetSize);
                }
                 break;
            case 7:
                printf("Set Difference\n");
                if(checkBitStringStatus()==1)
                {
                   setDifference(bitStringA,bitStringB);
                   printSet(bitStringDifference, superSetSize);
                }
                 break;

        }
    }
    }
```

OUTPUT:

```
1.Enter Universal Set
2.Enter Set A
3.Enter Set B
4.Generate Bit Strings
5.Union
6.Intersection
7.Difference
9.Exit
Enter Choice:1
Enter Super Set Size:5

Enter set
1
2
3
4
5
Enter Choice:2
Enter Set A Size:3

Enter set
1
3
5
Enter Choice:3
Enter Set B Size:3

Enter set
2
4
1
Enter Choice:4
Generating bit strings

Set A Bit String representation : {1,0,1,0,1}

Set B Bit String representation : {1,1,0,1,0}
Enter Choice:5
Set union
{1,1,1,1,1}
Enter Choice:6
Set Intersection
{1,0,0,0,0}
Enter Choice:7
Set Difference
{0,0,1,0,1}
```

RESULT:

Program is successfully executed and output is obtained.

LAB CYCLE: 2
EXPERIMENT NO:7
DATE:06/01/2022

AIM:
Program to implement Disjoint Sets and the associated operations (create, union, find).

ALGORITHM:

Algorithm of makeSet(*x*):
Step1 : if *x* is not already in the forest then
a) *x*.parent := *x*
b) x.size := 1
c)*x*.rank := 0

end of if

Algorithm of find(x):
Step1: If x.parent == x
return x
Step2 : else
return f*ind*(x.parent)

Algorithm of unionSets():
Step1 : xRoot =f*ind*(x)
Step2 : yRoot =f*ind*(y)
Step3 : xRoot.parent = yRoot

SOURCE CODE:

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

struct node{
  struct node *rep;
  struct node *next;
  int data;
}*heads[50],*tails[50];

static int countRoot=0;

void makeSet(int x){
        struct node *new=(struct node *)malloc(sizeof(struct node));
        new->rep=new;
        new->next=NULL;
        new->data=x;
        heads[countRoot]=new;
        tails[countRoot++]=new;
}
```

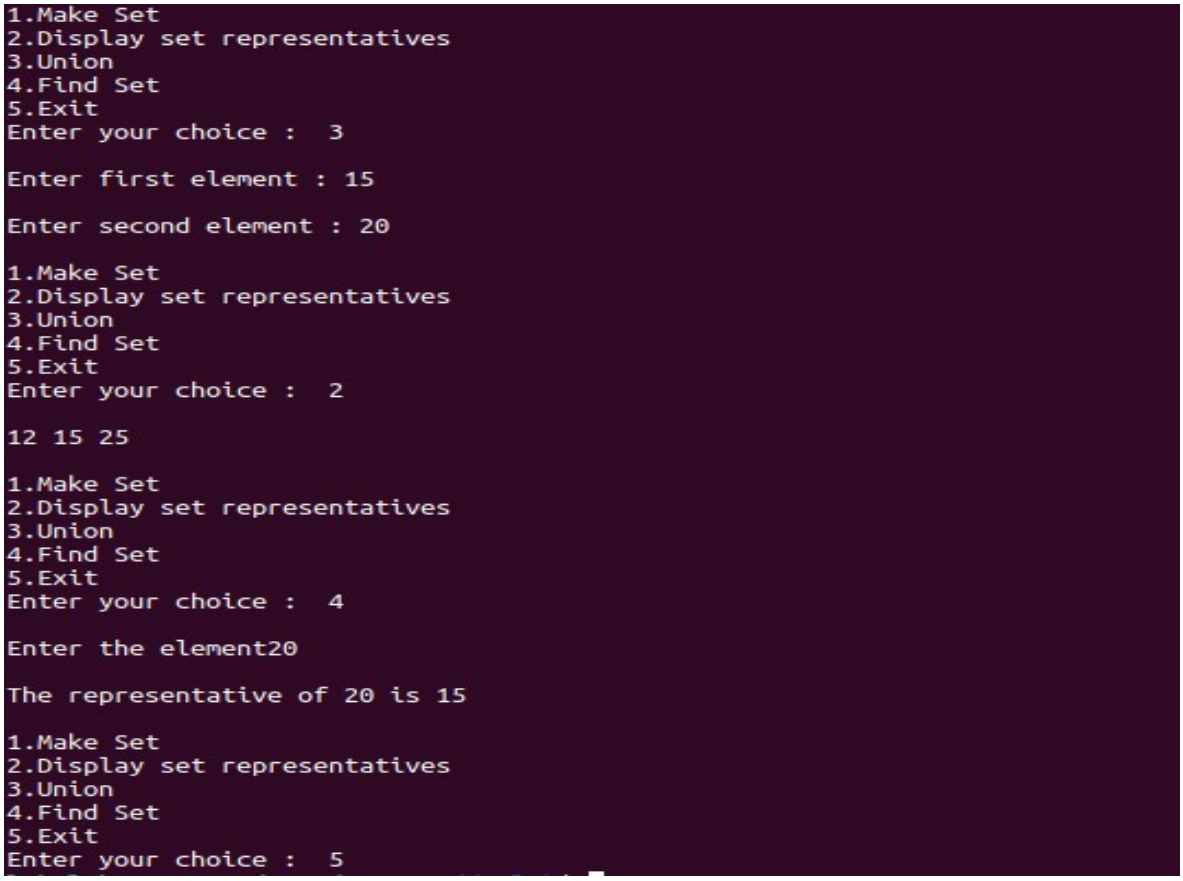
```
struct node* find(int a){
	int i;
	struct node *tmp=(struct node *)malloc(sizeof(struct node));
	for(i=0;i<countRoot;i++){
		tmp=heads[i];
		while(tmp!=NULL){
		if(tmp->data==a)
		return tmp->rep;
		tmp=tmp->next;
		}
	}
	return NULL;
}

void unionSets(int a,int b){
	int i,pos,flag=0,j;
	struct node *tail2=(struct node *)malloc(sizeof(struct node));
	struct node *rep1=find(a);
	struct node *rep2=find(b);
	if(rep1==NULL||rep2==NULL){
		printf("\nElement not present in the DS\n");
		return;
	}
	if(rep1!=rep2){
		for(j=0;j<countRoot;j++){
			if(heads[j]==rep2){
				pos=j;
				flag=1;
				countRoot-=1;
				tail2=tails[j];
				for(i=pos;i<countRoot;i++){
					heads[i]=heads[i+1];
					tails[i]=tails[i+1];
				}
			}
			if(flag==1)
				break;
		}
		for(j=0;j<countRoot;j++){
			if(heads[j]==rep1){
				tails[j]->next=rep2;
				tails[j]=tail2;
				break;
			}
		}
		while(rep2!=NULL){
		rep2->rep=rep1;
		rep2=rep2->next;
		}
	}
}
```

```
int search(int x){
	int i;
	struct node *tmp=(struct node *)malloc(sizeof(struct node));
	for(i=0;i<countRoot;i++){
		tmp=heads[i];
		if(heads[i]->data==x)
			return 1;
		while(tmp!=NULL){
			if(tmp->data==x)
				return 1;
			tmp=tmp->next;
		}
	}
	return 0;
}


void main(){
int choice,x,i,j,y,flag=0;

	do{
		printf("\n1.Make Set\n2.Display set representatives\n3.Union\n4.Find Set\n5.Exit\
n");
		printf("Enter your choice :  ");
		scanf("%d",&choice);
		switch(choice){
		case 1:
			printf("\nEnter new element : ");
			scanf("%d",&x);
			if(search(x)==1)
				printf("\nElement already present in the disjoint set DS\n");
			else
				makeSet(x);
			break;
		case 2:
			printf("\n");
			for(i=0;i<countRoot;i++)
				printf("%d ",heads[i]->data);
			printf("\n");
			break;
		case 3:
			printf("\nEnter first element : ");
			scanf("%d",&x);
			printf("\nEnter second element : ");
			scanf("%d",&y);
			unionSets(x,y);
			break;
		case 4:
			printf("\nEnter the element");
			scanf("%d",&x);
			struct node *rep=(struct node *)malloc(sizeof(struct node));
```

```
                        rep=find(x);
                        if(rep==NULL)
                        printf("\nElement not present in the DS\n");
                        else
                        printf("\nThe representative of %d is %d\n",x,rep->data);
                        break;
                case 5:
                        exit(0);
                default:
                        printf("\nWrong choice\n");
                        break;
                }
        }
        while(1);

};
```

OUTPUT:

```
1.Make Set
2.Display set representatives
3.Union
4.Find Set
5.Exit
Enter your choice :  1

Enter new element : 12

1.Make Set
2.Display set representatives
3.Union
4.Find Set
5.Exit
Enter your choice :  1

Enter new element : 15

1.Make Set
2.Display set representatives
3.Union
4.Find Set
5.Exit
Enter your choice :  1

Enter new element : 20

1.Make Set
2.Display set representatives
3.Union
4.Find Set
5.Exit
Enter your choice :  1

Enter new element : 25

1.Make Set
2.Display set representatives
3.Union
4.Find Set
5.Exit
Enter your choice :  2

12 15 20 25
```



RESULT:

Program is successfully executed and output is obtained.

LAB CYCLE: 2
EXPERIMENT NO:8
DATE:13/01/2022

AIM:
Program to implement Red Black Tree and its operations.

ALGORITHM:
Left-Rotate:
Step1 : If y has a left subtree, assign x as the parent of the left subtree of y.
Step2 : If the parent of x s NULL, make y as the root of the tree.
Step3 : Else if x is the left child of p, make y as the left child of p.
Step4 : Else assign y as the right child of p
Step5 : Make y as the parent of x

Right-Rotate:
Step1 : If x has a right subtree, assign y as the parent of the right subtree of x
Step2 : If the parent ofy is NULL, make x as the root of the tree.
Step3 : Else if y is the right child of its parent p, make x as the right child of p
Step4 : Else assign x as the left child of p

Step5 : Make x as the parent of y

Algorithm to maintain red-black property after insertion:
Step1 : Do the following while the parent of newNode p is RED.
Step2 : If p is the left child of grandParent gP of z, do the following.

Case-I:

a. If the color of the right child of gP of z is RED, set the color of both the children of gP as BLACK and the color of gP as RED.
b. Assign gP to newNode.

Case-II:

c. Else if newNode is the right child of p then, assign p to newNode.
d. Left-Rotate newNode.

Case-III:

e. Set color of p as BLACK and color of gP as RED.
f. Right-Rotate gP.

Step3 : Else, do the following.

g. If the color of the left child of gP of z is RED, set the color of both the children of gP as BLACK and the color of gP as RED.Assign gP to newNode.

h. Else if newNode is the left child of p then, assign p to newNode and Right-Rotate newNode.
i. Set color of p as BLACK and color of gP as RED.
j. Left-Rotate gP.

Step4 : Set the root of the tree as BLACK.

Insertion:

Step1 : Let y be the leaf (ie.NIL) and x be the root of the tree.
Step2 : Check if the tree is empty (ie. whether(x is NIL)). If yes, insert newNode as a root node and color it black.
Step3 : Else, repeat steps following steps until leaf NIL is reached.
a. Compare newKey with rootKey.
b. If newKey is greater than rootKey, traverse through the right subtree.Else, traverse through the left subtree.
Step4 : Assign the parent of the leaf as a parent of newNode.
Step5 : If leafKey is greater than newKey, make newNode as rightChild.
Step6 : Else, make newNode as leftChild.
Step7 : Assign NULL to the left and rightChild of newNode.
Step8 : Assign RED color to newNode.
Step9 : Call InsertFix-algorithm to maintain the property of red-black tree if violated.

Algorithm to maintain Red-Black property after deletion:

Step1 : Do the following until the x is not the root of the tree and the color of x is BLACK

Step2 : If x is the left child of its parent then,

a. Assign w to the sibling of x.

b. If the right child of parent of x is RED

Case-I:

a. Set the color of the right child of the parent of x as BLACK.
b. Set the color of the parent of x as RED.
c. Left-Rotate the parent of x.
d. Assign the rightChild of the parent of x to w.

c.If the color of both the right and the leftChild of w is BLACK,

Case-II:

e. Set the color of w as RED
f. Assign the parent of x to x.

d.Else if the color of the rightChild of w is BLACK

Case-III:

g. Set the color of the leftChild of w as BLACK
h. Set the color of w as RED
i. Right-Rotate w.
j. Assign the rightChild of the parent of x to w.

e. If any of the above cases do not occur, then do the following.

Case-IV:

k. Set the color of w as the color of the parent of x.
l. Set the color of the parent of x as BLACK.
m. Set the color of the right child of w as BLACK.
n. Left-Rotate the parent of x.

o. Set x as the root of the tree.

Step3 : Else the same as above with right changed to left and vice versa.

Step4 : Set the color of x as BLACK.

Deletion:

Step1 : Save the color of nodeToBeDeleted in origrinalColor.

Step2 : If the left child of nodeToBeDeleted is NULL

a. Assign the right child of nodeToBeDeleted to x.
b. Transplant nodeToBeDeleted with x.

Step3 : Else if the right child of nodeToBeDeleted is NULL

c. Assign the left child of nodeToBeDeleted into x.
d. Transplant nodeToBeDeleted with x.

Step4 : Else

a. Assign the minimum of right subtree of noteToBeDeleted into y.
b. Save the color of y in originalColor.
c. Assign the rightChild of y into x.
d. If y is a child of nodeToBeDeleted, then set the parent of x as y.
e. Else, transplant y with rightChild of y.
f. Transplant nodeToBeDeleted with y.
g. Set the color of y with originalColor.

Step5 : If the originalColor is BLACK, call DeleteFix(x).

SOURCE CODE:

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

enum nodeColor {
RED,
BLACK
};
struct rbNode {
int data, color;
struct rbNode *link[2];
};
struct rbNode *root = NULL;

struct rbNode *createNode(int data) {
struct rbNode *newnode;
newnode = (struct rbNode *)malloc(sizeof(struct rbNode));
newnode->data = data;
newnode->color = RED;
newnode->link[0] = newnode->link[1] = NULL;
return newnode;
}
```

```
void insertion(int data) {
struct rbNode *stack[98], *ptr, *newnode, *xPtr, *yPtr;
int dir[98], ht = 0, index;
ptr = root;
if (!root) {
root = createNode(data);
return;
}

stack[ht] = root;
dir[ht++] = 0;
while (ptr != NULL) {
if (ptr->data == data) {
printf("Duplicates Not Allowed!!\n");
return;
}
index = (data - ptr->data) > 0 ? 1 : 0;
stack[ht] = ptr;
ptr = ptr->link[index];
dir[ht++] = index;
}
stack[ht - 1]->link[index] = newnode = createNode(data);
while ((ht >= 3) && (stack[ht - 1]->color == RED)) {
if (dir[ht - 2] == 0) {
yPtr = stack[ht - 2]->link[1];
if (yPtr != NULL && yPtr->color == RED) {
stack[ht - 2]->color = RED;
stack[ht - 1]->color = yPtr->color = BLACK;
ht = ht - 2;
} else {
if (dir[ht - 1] == 0) {
yPtr = stack[ht - 1];
} else {
xPtr = stack[ht - 1];
yPtr = xPtr->link[1];
xPtr->link[1] = yPtr->link[0];
yPtr->link[0] = xPtr;
stack[ht - 2]->link[0] = yPtr;
}
xPtr = stack[ht - 2];
xPtr->color = RED;
yPtr->color = BLACK;
xPtr->link[0] = yPtr->link[1];
yPtr->link[1] = xPtr;
if (xPtr == root) {
root = yPtr;
} else {
stack[ht - 3]->link[dir[ht - 3]] = yPtr;
}
break;
}
```

```
} else {
yPtr = stack[ht - 2]->link[0];
if ((yPtr != NULL) && (yPtr->color == RED)) {
stack[ht - 2]->color = RED;
stack[ht - 1]->color = yPtr->color = BLACK;
ht = ht - 2;
} else {
if (dir[ht - 1] == 1) {
yPtr = stack[ht - 1];
} else {
xPtr = stack[ht - 1];
yPtr = xPtr->link[0];
xPtr->link[0] = yPtr->link[1];
yPtr->link[1] = xPtr;
stack[ht - 2]->link[1] = yPtr;
}
xPtr = stack[ht - 2];
yPtr->color = BLACK;
xPtr->color = RED;
xPtr->link[1] = yPtr->link[0];
yPtr->link[0] = xPtr;
if (xPtr == root) {
root = yPtr;
} else {
stack[ht - 3]->link[dir[ht - 3]] = yPtr;
}
break;
}
}
}
root->color = BLACK;
}

void deletion(int data) {
struct rbNode *stack[98], *ptr, *xPtr, *yPtr;
struct rbNode *pPtr, *qPtr, *rPtr;
int dir[98], ht = 0, diff, i;
enum nodeColor color;

if (!root) {
printf("Tree not available\n");
return;
}

ptr = root;
while (ptr != NULL) {
if ((data - ptr->data) == 0)
break;
diff = (data - ptr->data) > 0 ? 1 : 0;
stack[ht] = ptr;
dir[ht++] = diff;
ptr = ptr->link[diff];
```

```
}

if (ptr->link[1] == NULL) {
if ((ptr == root) && (ptr->link[0] == NULL)) {
free(ptr);
root = NULL;
} else if (ptr == root) {
root = ptr->link[0];
free(ptr);
} else {
stack[ht - 1]->link[dir[ht - 1]] = ptr->link[0];
}
} else {
xPtr = ptr->link[1];
if (xPtr->link[0] == NULL) {
xPtr->link[0] = ptr->link[0];
color = xPtr->color;
xPtr->color = ptr->color;
ptr->color = color;

if (ptr == root) {
root = xPtr;
} else {
stack[ht - 1]->link[dir[ht - 1]] = xPtr;
}

dir[ht] = 1;
stack[ht++] = xPtr;
} else {
i = ht++;
while (1) {
dir[ht] = 0;
stack[ht++] = xPtr;
yPtr = xPtr->link[0];
if (!yPtr->link[0])
break;
xPtr = yPtr;
}

dir[i] = 1;
stack[i] = yPtr;
if (i > 0)
stack[i - 1]->link[dir[i - 1]] = yPtr;

yPtr->link[0] = ptr->link[0];

xPtr->link[0] = yPtr->link[1];
yPtr->link[1] = ptr->link[1];

if (ptr == root) {
root = yPtr;
}
```

```
color = yPtr->color;
yPtr->color = ptr->color;
ptr->color = color;
}
}

if (ht < 1)
return;

if (ptr->color == BLACK) {
while (1) {
pPtr = stack[ht - 1]->link[dir[ht - 1]];
if (pPtr && pPtr->color == RED) {
pPtr->color = BLACK;
break;
}

if (ht < 2)
break;

if (dir[ht - 2] == 0) {
rPtr = stack[ht - 1]->link[1];

if (!rPtr)
break;

if (rPtr->color == RED) {
stack[ht - 1]->color = RED;
rPtr->color = BLACK;
stack[ht - 1]->link[1] = rPtr->link[0];
rPtr->link[0] = stack[ht - 1];

if (stack[ht - 1] == root) {
root = rPtr;
} else {
stack[ht - 2]->link[dir[ht - 2]] = rPtr;
}
dir[ht] = 0;
stack[ht] = stack[ht - 1];
stack[ht - 1] = rPtr;
ht++;

rPtr = stack[ht - 1]->link[1];
}

if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&
(!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {
rPtr->color = RED;
} else {
if (!rPtr->link[1] || rPtr->link[1]->color == BLACK) {
qPtr = rPtr->link[0];
```

```
rPtr->color = RED;
qPtr->color = BLACK;
rPtr->link[0] = qPtr->link[1];
qPtr->link[1] = rPtr;
rPtr = stack[ht - 1]->link[1] = qPtr;
}
rPtr->color = stack[ht - 1]->color;
stack[ht - 1]->color = BLACK;
rPtr->link[1]->color = BLACK;
stack[ht - 1]->link[1] = rPtr->link[0];
rPtr->link[0] = stack[ht - 1];
if (stack[ht - 1] == root) {
root = rPtr;
} else {
stack[ht - 2]->link[dir[ht - 2]] = rPtr;
}
break;
}
} else {
rPtr = stack[ht - 1]->link[0];
if (!rPtr)
break;

if (rPtr->color == RED) {
stack[ht - 1]->color = RED;
rPtr->color = BLACK;
stack[ht - 1]->link[0] = rPtr->link[1];
rPtr->link[1] = stack[ht - 1];

if (stack[ht - 1] == root) {
root = rPtr;
} else {
stack[ht - 2]->link[dir[ht - 2]] = rPtr;
}
dir[ht] = 1;
stack[ht] = stack[ht - 1];
stack[ht - 1] = rPtr;
ht++;

rPtr = stack[ht - 1]->link[0];
}
if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&
(!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {
rPtr->color = RED;
} else {
if (!rPtr->link[0] || rPtr->link[0]->color == BLACK) {
qPtr = rPtr->link[1];
rPtr->color = RED;
qPtr->color = BLACK;
rPtr->link[1] = qPtr->link[0];
qPtr->link[0] = rPtr;
rPtr = stack[ht - 1]->link[0] = qPtr;
```

```
}
rPtr->color = stack[ht - 1]->color;
stack[ht - 1]->color = BLACK;
rPtr->link[0]->color = BLACK;
stack[ht - 1]->link[0] = rPtr->link[1];
rPtr->link[1] = stack[ht - 1];
if (stack[ht - 1] == root) {
root = rPtr;
} else {
stack[ht - 2]->link[dir[ht - 2]] = rPtr;
}
break;
}
}
ht--;
}
}
}

// Print the inorder traversal of the tree
void inorderTraversal(struct rbNode *node) {
if (node) {
inorderTraversal(node->link[0]);
printf("%d ", node->data);
inorderTraversal(node->link[1]);
}
return;
}

// Driver code
int main() {
int ch, data;
while (1) {
printf("1. Insertion\t2. Deletion\n");
printf("3. Traverse\t4. Exit");
printf("\nEnter your choice:");
scanf("%d", &ch);
switch (ch) {
case 1:
printf("Enter the element to insert:");
scanf("%d", &data);
insertion(data);
break;
case 2:
printf("Enter the element to delete:");
scanf("%d", &data);
deletion(data);
break;
case 3:
inorderTraversal(root);
printf("\n");
break;
```

```
case 4:
exit(0);
default:
printf("Not available\n");
break;
}
printf("\n");
}
return 0;
}
```

OUTPUT:

```
1. Insertion    2. Deletion
3. Traverse     4. Exit
Enter your choice:1
Enter the element to insert:11

1. Insertion    2. Deletion
3. Traverse     4. Exit
Enter your choice:1
Enter the element to insert:2

1. Insertion    2. Deletion
3. Traverse     4. Exit
Enter your choice:1
Enter the element to insert:1

1. Insertion    2. Deletion
3. Traverse     4. Exit
Enter your choice:1
Enter the element to insert:7

1. Insertion    2. Deletion
3. Traverse     4. Exit
Enter your choice:1
Enter the element to insert:5

1. Insertion    2. Deletion
3. Traverse     4. Exit
Enter your choice:1
Enter the element to insert:8

1. Insertion    2. Deletion
3. Traverse     4. Exit
Enter your choice:1
Enter the element to insert:14
```

```
1. Insertion    2. Deletion
3. Traverse     4. Exit
Enter your choice:1
Enter the element to insert:15

1. Insertion    2. Deletion
3. Traverse     4. Exit
Enter your choice:3
1  2  5  7  8  11  14  15

1. Insertion    2. Deletion
3. Traverse     4. Exit
Enter your choice:2
Enter the element to delete:7

1. Insertion    2. Deletion
3. Traverse     4. Exit
Enter your choice:3
1  2  5  8  11  14  15

1. Insertion    2. Deletion
3. Traverse     4. Exit
Enter your choice:4
```

RESULT:

Program is successfully executed and output is obtained.

LAB CYCLE: 2
EXPERIMENT NO:9
DATE:18/01/2022

AIM:
Program to implement B Tree and its operations.

ALGORITHM:
Insertion:
Step1 : If the tree is empty, allocate a root node and insert the key.
Step2 : Update the allowed number of keys in the node.
Step3 : Search the appropriate node for insertion.
Step4 : If the node is full, follow the steps below.
Step5 : Insert the elements in increasing order.
Step6 : Now, there are elements greater than its limit. So, split at the median.
Step7 : Push the median key upwards and make the left keys as a left child and the right keys as a right child.
Step8 : If the node is not full, follow the steps below.
Step9 : Insert the node in increasing order.

Search:
Step1 : Starting from the root node, compare k with the first key of the node.
If k = the first key of the node, return the node and the index.
Step2 : If k.leaf = true, return NULL (i.e. not found).
Step3 : If k < the first key of the root node, search the left child of this key recursively.
Step4 : If there is more than one key in the current node and k > the first key, compare k with the next key in the node.
If k < next key, search the left child of this key (ie. k lies in between the first and the second keys).
Else, search the right child of the key.
Step5 : Repeat steps 1 to 4 until the leaf is reached.

Deletion:

Step1: If the key to be deleted lies in the leaf then,

Step2 : If the deletion of the key does not violate the property of the minimum number of keys a node should hold then, remove the key from the tree.

Step3 : Else,

i) visit the immediate left sibling.

ii) If the left sibling node has more than a minimum number of keys, then borrow a key from this node.

iii) Else, check to borrow from the immediate right sibling node.

iv) goto step 9

Step4 : If the key to be deleted lies in the internal node then,

Step5 : If the left child has more than the minimum number of keys then replace it by an inorder predecessor.

Step6 : Else if the right child has more than the minimum number of keys then replace it by inorder successor.

Step7 : Else, merge the left and the right children.

Step8 :goto step1

Step9 : stop

SOURCE CODE:

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

  #define MAX 4
  #define MIN 2

  struct btreeNode {
       int val[MAX + 1], count;
       struct btreeNode *link[MAX + 1];
  };

  struct btreeNode *root;

  /* creating new node */
  struct btreeNode * createNode(int val, struct btreeNode *child) {
       struct btreeNode *newNode;
       newNode = (struct btreeNode *)malloc(sizeof(struct btreeNode));
       newNode->val[1] = val;
       newNode->count = 1;
       newNode->link[0] = root;
       newNode->link[1] = child;
       return newNode;
  }

  /* Places the value in appropriate position */
  void addValToNode(int val, int pos, struct btreeNode *node,
                    struct btreeNode *child) {
       int j = node->count;
       while (j > pos) {
             node->val[j + 1] = node->val[j];
             node->link[j + 1] = node->link[j];
             j--;
       }
       node->val[j + 1] = val;
       node->link[j + 1] = child;
       node->count++;
  }

  /* split the node */
  void splitNode (int val, int *pval, int pos, struct btreeNode *node,
    struct btreeNode *child, struct btreeNode **newNode) {
       int median, j;
```

```
        if (pos > MIN)
                median = MIN + 1;
        else
                median = MIN;

        *newNode = (struct btreeNode *)malloc(sizeof(struct btreeNode));
        j = median + 1;
        while (j <= MAX) {
                (*newNode)->val[j - median] = node->val[j];
                (*newNode)->link[j - median] = node->link[j];
                j++;
        }
        node->count = median;
        (*newNode)->count = MAX - median;

        if (pos <= MIN) {
                addValToNode(val, pos, node, child);
        } else {
                addValToNode(val, pos - median, *newNode, child);
        }
        *pval = node->val[node->count];
        (*newNode)->link[0] = node->link[node->count];
        node->count--;
}

/* sets the value val in the node */
int setValueInNode(int val, int *pval,
  struct btreeNode *node, struct btreeNode **child) {

        int pos;
        if (!node) {
                *pval = val;
                *child = NULL;
                return 1;
        }

        if (val < node->val[1]) {
                pos = 0;
        } else {
                for (pos = node->count;
                        (val < node->val[pos] && pos > 1); pos--);
                if (val == node->val[pos]) {
                        printf("Duplicates not allowed\n");
                        return 0;
                }
        }
        if (setValueInNode(val, pval, node->link[pos], child)) {
                if (node->count < MAX) {
                        addValToNode(*pval, pos, node, *child);
                } else {
                        splitNode(*pval, pval, pos, node, *child, child);
                        return 1;
```

```
            }
        }
        return 0;
}

/* insert val in B-Tree */
void insertion(int val) {
        int flag, i;
        struct btreeNode *child;

        flag = setValueInNode(val, &i, root, &child);
        if (flag)
                root = createNode(i, child);
}

/* copy successor for the value to be deleted */
void copySuccessor(struct btreeNode *myNode, int pos) {
        struct btreeNode *dummy;
        dummy = myNode->link[pos];

        for (;dummy->link[0] != NULL;)
                dummy = dummy->link[0];
        myNode->val[pos] = dummy->val[1];

}

/* removes the value from the given node and rearrange values */
void removeVal(struct btreeNode *myNode, int pos) {
        int i = pos + 1;
        while (i <= myNode->count) {
                myNode->val[i - 1] = myNode->val[i];
                myNode->link[i - 1] = myNode->link[i];
                i++;
        }
        myNode->count--;
}

/* shifts value from parent to right child */
void doRightShift(struct btreeNode *myNode, int pos) {
        struct btreeNode *x = myNode->link[pos];
        int j = x->count;

        while (j > 0) {
                x->val[j + 1] = x->val[j];
                x->link[j + 1] = x->link[j];
        }
        x->val[1] = myNode->val[pos];
        x->link[1] = x->link[0];
        x->count++;

        x = myNode->link[pos - 1];
        myNode->val[pos] = x->val[x->count];
```

```
        myNode->link[pos] = x->link[x->count];
        x->count--;
        return;
    }

    /* shifts value from parent to left child */
    void doLeftShift(struct btreeNode *myNode, int pos) {
        int j = 1;
        struct btreeNode *x = myNode->link[pos - 1];

        x->count++;
        x->val[x->count] = myNode->val[pos];
        x->link[x->count] = myNode->link[pos]->link[0];

        x = myNode->link[pos];
        myNode->val[pos] = x->val[1];
        x->link[0] = x->link[1];
        x->count--;

        while (j <= x->count) {
            x->val[j] = x->val[j + 1];
            x->link[j] = x->link[j + 1];
            j++;
        }
        return;
    }

    /* merge nodes */
    void mergeNodes(struct btreeNode *myNode, int pos) {
        int j = 1;
        struct btreeNode *x1 = myNode->link[pos], *x2 = myNode->link[pos - 1];

        x2->count++;
        x2->val[x2->count] = myNode->val[pos];
        x2->link[x2->count] = myNode->link[0];

        while (j <= x1->count) {
            x2->count++;
            x2->val[x2->count] = x1->val[j];
            x2->link[x2->count] = x1->link[j];
            j++;
        }

        j = pos;
        while (j < myNode->count) {
            myNode->val[j] = myNode->val[j + 1];
            myNode->link[j] = myNode->link[j + 1];
            j++;
        }
        myNode->count--;
        free(x1);
    }
```

```
/* adjusts the given node */
void adjustNode(struct btreeNode *myNode, int pos) {
    if (!pos) {
        if (myNode->link[1]->count > MIN) {
            doLeftShift(myNode, 1);
        } else {
            mergeNodes(myNode, 1);
        }
    } else {
        if (myNode->count != pos) {
            if(myNode->link[pos - 1]->count > MIN) {
                doRightShift(myNode, pos);
            } else {
                if (myNode->link[pos + 1]->count > MIN) {
                    doLeftShift(myNode, pos + 1);
                } else {
                    mergeNodes(myNode, pos);
                }
            }
        } else {
            if (myNode->link[pos - 1]->count > MIN)
                doRightShift(myNode, pos);
            else
                mergeNodes(myNode, pos);
        }
    }
}

/* delete val from the node */
int delValFromNode(int val, struct btreeNode *myNode) {
    int pos, flag = 0;
    if (myNode) {
        if (val < myNode->val[1]) {
            pos = 0;
            flag = 0;
        } else {
            for (pos = myNode->count;
                (val < myNode->val[pos] && pos > 1); pos--);
             if (val == myNode->val[pos]) {
                flag = 1;
            } else {
                flag = 0;
            }
        }
        if (flag) {
            if (myNode->link[pos - 1]) {
                copySuccessor(myNode, pos);
                flag = delValFromNode(myNode->val[pos], myNode->link[pos]);
                if (flag == 0) {
                    printf("Given data is not present in B-Tree\n");
                }
```

```
                } else {
                    removeVal(myNode, pos);
                }
            } else {
                flag = delValFromNode(val, myNode->link[pos]);
            }
            if (myNode->link[pos]) {
                if (myNode->link[pos]->count < MIN)
                    adjustNode(myNode, pos);
            }
        }
        return flag;
    }

    /* delete val from B-tree */
    void deletion(int val, struct btreeNode *myNode) {
        struct btreeNode *tmp;
        if (!delValFromNode(val, myNode)) {
            printf("Given value is not present in B-Tree\n");
            return;
        } else {
            if (myNode->count == 0) {
                tmp = myNode;
                myNode = myNode->link[0];
                free(tmp);
            }
        }
        root = myNode;
        return;
    }

    /* search val in B-Tree */
    void searching(int val, int *pos, struct btreeNode *myNode) {
        if (!myNode) {
            return;
        }

        if (val < myNode->val[1]) {
            *pos = 0;
        } else {
            for (*pos = myNode->count;
                (val < myNode->val[*pos] && *pos > 1); (*pos)--);
            if (val == myNode->val[*pos]) {
                printf("Given data %d is present in B-Tree", val);
                return;
            }
        }
        searching(val, pos, myNode->link[*pos]);
        return;
    }

    /* B-Tree Traversal */
```

```
void traversal(struct btreeNode *myNode) {
    int i;
    if (myNode) {
        for (i = 0; i < myNode->count; i++) {
            traversal(myNode->link[i]);
            printf("%d ", myNode->val[i + 1]);
        }
        traversal(myNode->link[i]);
    }
}

int main() {
    int val, ch;
    while (1) {
        printf("1. Insertion\t2. Deletion\n");
        printf("3. Searching\t4. Traversal\n");
        printf("5. Exit\nEnter your choice:");
        scanf("%d", &ch);
        switch (ch) {
            case 1:
                printf("Enter your input:");
                scanf("%d", &val);
                insertion(val);
                break;
            case 2:
                printf("Enter the element to delete:");
                scanf("%d", &val);
                deletion(val, root);
                break;
            case 3:
                printf("Enter the element to search:");
                scanf("%d", &val);
                searching(val, &ch, root);
                break;
            case 4:
                traversal(root);
                break;
            case 5:
                exit(0);
            default:
                printf("U have entered wrong option!!\n");
                break;
        }
        printf("\n");
    }

}
```

OUTPUT:

```
1. Insertion    2. Deletion
3. Searching    4. Traversal
5. Exit
Enter your choice:1
Enter your input:8

1. Insertion    2. Deletion
3. Searching    4. Traversal
5. Exit
Enter your choice:1
Enter your input:9

1. Insertion    2. Deletion
3. Searching    4. Traversal
5. Exit
Enter your choice:1
Enter your input:10

1. Insertion    2. Deletion
3. Searching    4. Traversal
5. Exit
Enter your choice:1
Enter your input:11

1. Insertion    2. Deletion
3. Searching    4. Traversal
5. Exit
Enter your choice:1
Enter your input:15

1. Insertion    2. Deletion
3. Searching    4. Traversal
5. Exit
Enter your choice:1
Enter your input:20

1. Insertion    2. Deletion
3. Searching    4. Traversal
5. Exit
Enter your choice:1
Enter your input:17
```

```
1. Insertion    2. Deletion
3. Searching    4. Traversal
5. Exit
Enter your choice:4
8 9 10 11 15 17 20
1. Insertion    2. Deletion
3. Searching    4. Traversal
5. Exit
Enter your choice:2
Enter the element to delete:11

1. Insertion    2. Deletion
3. Searching    4. Traversal
5. Exit
Enter your choice:4
8 9 10 15 17 20
1. Insertion    2. Deletion
3. Searching    4. Traversal
5. Exit
Enter your choice:3
Enter the element to search:15
Given data 15 is present in B-Tree
1. Insertion    2. Deletion
3. Searching    4. Traversal
5. Exit
Enter your choice:5
```

RESULT:

Program is successfully executed and output is obtained.

LAB CYCLE: 2
EXPERIMENT NO:10
DATE:20/01/2022

AIM:
Program to implement Binomial Heaps and its operations (Create, Insert, Delete, Extract-min, Decrease key)

ALGORITHM:
Creating a new binomial heap:

To make an empty binomial heap, the MAKE_BIN_HEAP procedure simply allocates and returns an object H, where $head[H]$ = NIL. The running time is $\Theta(1)$.

Inserting a node:
BIN_HEAP-_NSERT(H,x)

Step1 : $H' \leftarrow$ MAKE_BIN_HEAP()

Step2 : $p[x] \leftarrow$ NIL

Step3 : $child[x] \leftarrow$ NIL

Step4 : $sibling[x] \leftarrow$ NIL

Step5 : $degree[x] \leftarrow 0$

Step6 : $head[H'] \leftarrow$ x

Step7 : $H \leftarrow$ BIN-HEAP_UNION(H,H')

Uniting two binomial heaps:

The BIN_HEAP_UNION procedure repeatedly links binomial trees whose roots have the same degree. The following procedure links the B_k-1 tree rooted at node y to the B_k-1 tree rooted at node z; that is, it makes z the parent of y. Node z thus becomes the root of a B_k tree.

BIN_LINK(y,z)

Step1 : p[y] $\leftarrow$ z

Step2 : sibling[y] $\leftarrow$ child[z]

Step3 : child[z] $\leftarrow$ y

Step4 : degree[z] $\leftarrow$ degree[z]+1

BIN_HEAP_UNION(H1, H2)

Step1 : H $\leftarrow$ MAKE_BIN_HEAP()

Step2 : head[H] $\leftarrow$ BIN_HEAP_MERGE(H1, H2)

Step3 : free the objects H1 and H2 but not the lists they point to

Step4 : if head[H] = NIL

Step5 : then return H

Step6 : prev-x ← NIL

Step7 : x ← head[H]

Step8 : next-x ← sibling[x]

Step9 : while next-x = NIL

Step10 : do if (degree[x] = degree[next-x]) or (sibling[next-x] = NIL and degree[sibling[next-x]] = degree[x])

Step11 : then prev-x ← x

Step12 : x ← next-x

Step13 : else if key[x] ≤ key[next-x]

Step14 : then sibling[x] ← sibling[next-x]

Step15 : BIN_LINK(next-x, x)

Step16 : else if prev-x = NIL

Step17 : then head[H] ← next-x

Step18 : else sibling[prev-x] ← next-x

Step19 : BIN_LINK(x, next-x)

Step20 : x ← next-x

Step21 : next-x ← sibling[x]

Step22 : return H

Extracting the node with minimum key:

BIN_HEAP_EXTRACT_MIN(*H*)

Step1 : find the root *x* with the minimum key in the root list of *H a*nd remove *x* from the root list of *H*

Step2 : *H*' ← MAKE_BIN_HEAP()

Step3 : reverse the order of the linked list of *x*'s children and set *head*[*H*'] to point to the head of the resulting list

Step4 : $H \leftarrow$ BIN_HEAP_UNION(H,H')

Step5 : return x

Decreasing a key:

BIN_HEAP_DECREASE_KEY(H, x, k)

Step1 : if k > key[x]

Step2 : then error “new key is greater than current key”

Step3 : key[x] ← k

Step4 : y ← x

Step5 : z ← p[y]

Step6 : while z = NIL and key[y] < key[z]

Step7 : do exchange key[y] ↔ key[z]

Step8 : If y and z have satellite fields, exchange them, too.

Step9 : y ← z

Step10 : z ← p[y]

Deleting a key:

BIN-HEAP-DELETE(H,x)

Step1 : BINOMIAL_HEAP_DECREASE_KEY($H,x,-\infty$)

Step2 : BINOMIAL_HEAP_EXTRACT_MIN(H)

SOURCE CODE:

```
#include<stdio.h>
#include<malloc.h>
#include<stdlib.h>
struct node {
   int n;
   int degree;
   struct node* parent;
   struct node* child;
   struct node* sibling;
};

struct node* MAKE_BIN_HEAP();
int BIN_LINK(struct node*, struct node*);
struct node* create_node(int);
```

```
struct node* BIN_HEAP_UNION(struct node*, struct node*);
struct node* BIN_HEAP_INSERT(struct node*, struct node*);
struct node* BIN_HEAP_MERGE(struct node*, struct node*);
struct node* BIN_HEAP_EXTRACT_MIN(struct node*);
int REVERT_LIST(struct node*);
int display(struct node*);
struct node* FIND_NODE(struct node*, int);
int BIN_HEAP_DECREASE_KEY(struct node*, int, int);
int BIN_HEAP_DELETE(struct node*, int);

int count = 1;
struct node * H = NULL;
struct node *Hr = NULL;

struct node* MAKE_BIN_HEAP() {
   struct node* np;
   np = NULL;
   return np;
}

int BIN_LINK(struct node* y, struct node* z) {
   y->parent = z;
   y->sibling = z->child;
   z->child = y;
   z->degree = z->degree + 1;
}

struct node* create_node(int k) {
   struct node* p;
   p = (struct node*) malloc(sizeof(struct node));
   p->n = k;
   return p;
}

struct node* BIN_HEAP_UNION(struct node* H1, struct node* H2) {
   struct node* prev_x;
   struct node* next_x;
   struct node* x;
   struct node* H = MAKE_BIN_HEAP();
   H = BIN_HEAP_MERGE(H1, H2);
   if (H == NULL)
      return H;
   prev_x = NULL;
   x = H;
   next_x = x->sibling;
   while (next_x != NULL) {
      if ((x->degree != next_x->degree) || ((next_x->sibling != NULL)
            && (next_x->sibling)->degree == x->degree)) {
         prev_x = x;
         x = next_x;
      } else {
         if (x->n <= next_x->n) {
```

```
            x->sibling = next_x->sibling;
            BIN_LINK(next_x, x);
          } else {
            if (prev_x == NULL)
              H = next_x;
            else
              prev_x->sibling = next_x;
            BIN_LINK(x, next_x);
            x = next_x;
          }
        }
        next_x = x->sibling;
    }
    return H;
}

struct node* BIN_HEAP_INSERT(struct node* H, struct node* x) {
    struct node* H1 = MAKE_BIN_HEAP();
    x->parent = NULL;
    x->child = NULL;
    x->sibling = NULL;
    x->degree = 0;
    H1 = x;
    H = BIN_HEAP_UNION(H, H1);
    return H;
}

struct node* BIN_HEAP_MERGE(struct node* H1, struct node* H2) {
    struct node* H = MAKE_BIN_HEAP();
    struct node* y;
    struct node* z;
    struct node* a;
    struct node* b;
    y = H1;
    z = H2;
    if (y != NULL) {
        if (z != NULL && y->degree <= z->degree)
            H = y;
        else if (z != NULL && y->degree > z->degree)
            H = z;
        else
            H = y;
    } else
        H = z;
    while (y != NULL && z != NULL) {
        if (y->degree < z->degree) {
            y = y->sibling;
        } else if (y->degree == z->degree) {
            a = y->sibling;
            y->sibling = z;
            y = a;
        } else {
```

```
            b = z->sibling;
            z->sibling = y;
            z = b;
        }
    }
    return H;
}

int display(struct node* H) {
    struct node* p;
    if (H == NULL) {
        printf("\n Heap is empty!!!");
        return 0;
    }
    printf("\n Root nodes are : ");
    p = H;
    while (p != NULL) {
        printf("%d", p->n);
        if (p->sibling != NULL)
            printf("-->");
        p = p->sibling;
    }
    printf("\n");
}

struct node* BIN_HEAP_EXTRACT_MIN(struct node* H1) {
    int min;
    struct node* t = NULL;
    struct node* x = H1;
    struct node* p;
    Hr=NULL;
    if (x == NULL) {
        printf("\n Nothing to extract!!!");
        return x;
    }
    min=x->n;
    p = x;
    while (p->sibling != NULL) {
        if ((p->sibling)->n < min) {
            min = (p->sibling)->n;
            t = p;
            x = p->sibling;
        }
        p = p->sibling;
    }
    if (t == NULL && x->sibling == NULL)
        H1 = NULL;
    else if (t == NULL)
        H1 = x->sibling;
    else if (t->sibling == NULL)
        t = NULL;
    else
```

```
        t->sibling = x->sibling;
    if (x->child != NULL) {
        REVERT_LIST(x->child);
        (x->child)->sibling = NULL;
    }
    H = BIN_HEAP_UNION(H1, Hr);
    return x;
}
int REVERT_LIST(struct node* y) {
    if (y->sibling != NULL) {
        REVERT_LIST(y->sibling);
        (y->sibling)->sibling = y;
    } else {
        Hr = y;
    }
}
struct node* FIND_NODE(struct node* H, int k) {
    struct node* x = H;
    struct node* p = NULL;
    if (x->n == k) {
        p = x;
        return p;
    }
    if (x->child != NULL && p == NULL) {
        p = FIND_NODE(x->child, k);
    }

    if (x->sibling != NULL && p == NULL) {
        p = FIND_NODE(x->sibling, k);
    }
    return p;
}
int BIN_HEAP_DECREASE_KEY(struct node* H, int i, int k) {
    int temp;
    struct node* p;
    struct node* y;
    struct node* z;
    p = FIND_NODE(H, i);
    if (p == NULL) {
        printf("\n Invalid choice of key to be reduced!!! ");
        return 0;
    }
    if (k > p->n) {
        printf("\n New key value is greater than the current key value!!!");
        return 0;
    }
    p->n = k;
    y = p;
    z = p->parent;
    while (z != NULL && y->n < z->n) {
        temp = y->n;
        y->n = z->n;
```

```
        z->n = temp;
        y = z;
        z = z->parent;
    }
    printf("\n Key reduced successfully!!!");
}
int BIN_HEAP_DELETE(struct node* H, int k) {
    struct node* np;
    if (H == NULL) {
        printf("\n Heap is empty!!!");
        return 0;
    }
    BIN_HEAP_DECREASE_KEY(H, k, -1000);
    np = BIN_HEAP_EXTRACT_MIN(H);
    if (np != NULL)
        printf("\n Node deleted successfully!!!");
}
int main() {
    int i, n, m, choice,l;
    struct node* p;
    struct node* np;
    char ch;
    printf("\n Enter the no. of elements: ");
    scanf("%d", &n);
    printf("\n Enter the elements: ");
    for (i = 1; i <= n; i++) {
        scanf("%d", &m);
        np = create_node(m);
        H = BIN_HEAP_INSERT(H, np);
    }
    display(H);
    printf("\n 1)Insert an element \n 2)Extract min node \n 3)Decrease a key value \n 4)Delete a
node \n 5)Exit");
    do {
        printf("\n Enter the choice: ");
        scanf("%d", &choice);
        switch (choice) {
        case 1:
                printf("\n Enter the element to be inserted: ");
                scanf("%d", &m);
                p = create_node(m);
                H = BIN_HEAP_INSERT(H, p);
                printf("\n Heap is: ");
                display(H);
                 break;
        case 2:
                 p = BIN_HEAP_EXTRACT_MIN(H);
                 if (p != NULL)
                    printf("\n Extracted node is  %d", p->n);
                 printf("\n Heap will be: ");
                 display(H);
                 break;
```

```
        case 3:
                printf("\n Enter the node to be decreased: ");
                scanf("%d", &m);
                printf("\n Enter the new value: ");
                scanf("%d", &l);
                BIN_HEAP_DECREASE_KEY(H, m, l);
                printf("\n Heap will be: ");
                display(H);
                break;
        case 4:
                printf("\n Enter the key value of the node to be deleted: ");
                scanf("%d", &m);
                BIN_HEAP_DELETE(H, m);
                      printf("\n Heap will be: ");
                display(H);
                break;
        case 5:exit(0);
              break;
        default:printf("\n Invalid entry!!!");
        }
    } while (choice!= 5);}
```

OUTPUT:

```
 Enter the no. of elements: 5

 Enter the elements: 1
5
12
67
33

 Root nodes are : 33-->1

 1)Insert an element
 2)Extract min node
 3)Decrease a key value
 4)Delete a node
 5)Exit
 Enter the choice: 1

 Enter the element to be inserted: 60

 Heap is:
 Root nodes are : 33-->1

 Enter the choice: 2

 Extracted node is  1
 Heap will be:
 Root nodes are : 5-->12

 Enter the choice: 3

 Enter the node to be decreased: 33

 Enter the new value: 2

 Key reduced successfully!!!
 Heap will be:
 Root nodes are : 5-->2

 Enter the choice: 4

 Enter the key value of the node to be deleted: 12

 Key reduced successfully!!!
 Node deleted successfully!!!
 Heap will be:
 Root nodes are : 2
```

RESULT:

Program is successfully executed and output is obtained.

LAB CYCLE: 3
EXPERIMENT NO:11
DATE:25/01/2022

AIM:
Program to implement Graph Traversal techniques (DFS and BFS) and Topological Sorting

ALGORITHM:
DFS
A standard DFS implementation puts each vertex of the graph into one of two categories:

1.Visited

2.Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

Step1: Start by putting any one of the graph's vertices on top of a stack.

Step2: Take the top item of the stack and add it to the visited list.

Step3:Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.

Step4:Keep repeating steps 2 and 3 until the stack is empty.

BFS

A standard BFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

Step1:Start by putting any one of the graph's vertices at the back of a queue.

Step2:Take the front item of the queue and add it to the visited list.

Step3:Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.

Step4:Keep repeating steps 2 and 3 until the queue is empty.

Topological Sort:

Step1: Compute the in-degree for each of the vertex present in the DAG and initialize the count of visited nodes as ‘0’.
Step2 : Pick the vertices with in-degree as ‘0’ and insert them into a QUEUE.
Step3 : Remove a vertex from the QUEUE and do,
a)Increment count of visited nodes by ‘1’.
b)Decrease the in-degree by one for all its neighbouring nodes.
c)If the in-degree of neighbouring nodes is to ‘0’ then, insert it to the QUEUE.
Step4 : Repeat step3 until the QUEUE is empty.

SOURCE CODE:

```
#include<stdio.h>
int a[20][20],q[20],visited[20],n,i,j,f=0,r=-1;
typedef enum boolean{false,true} bool;
bool visited_dfs[10];
void bfs(int v)
{
 for (i=1;i<=n;i++)
 if(a[v][i] && !visited[i])
 q[++r]=i;
 if(f<=r)
 { visited[q[f]]=1;
 bfs(q[f++]);
 }
}
void bfs_sort(){
           int v;
 printf("\n enter the number of vertices:");
 scanf("%d",&n);
 for (i=1;i<=n;i++)
 {
 q[i]=0;
 visited[i]=0;
 }
printf("\n enter graph data in matrix form:\n");
for (i=1;i<=n;i++)
 for (j=1;j<=n;j++)
 scanf("%d",&a[i][j]);
 printf("\n Enter the starting vertex:");
 scanf("%d",&v);
bfs(v);
printf("\n The node which are reachable are:\n");
for (i=1;i<=n;i++)
 if(visited[i])
 printf("%d\t",i);
 else
 printf("\n Bfs is not possible");
}
void dfs(int v)
{
 int i,stack[10],top=-1,pop;
 top++;
 stack[top]=v;
 while(top>=0)
 {
 pop=stack[top];
 top--;
 if(visited_dfs[pop]==false)
 {
 printf("%d",pop);
```

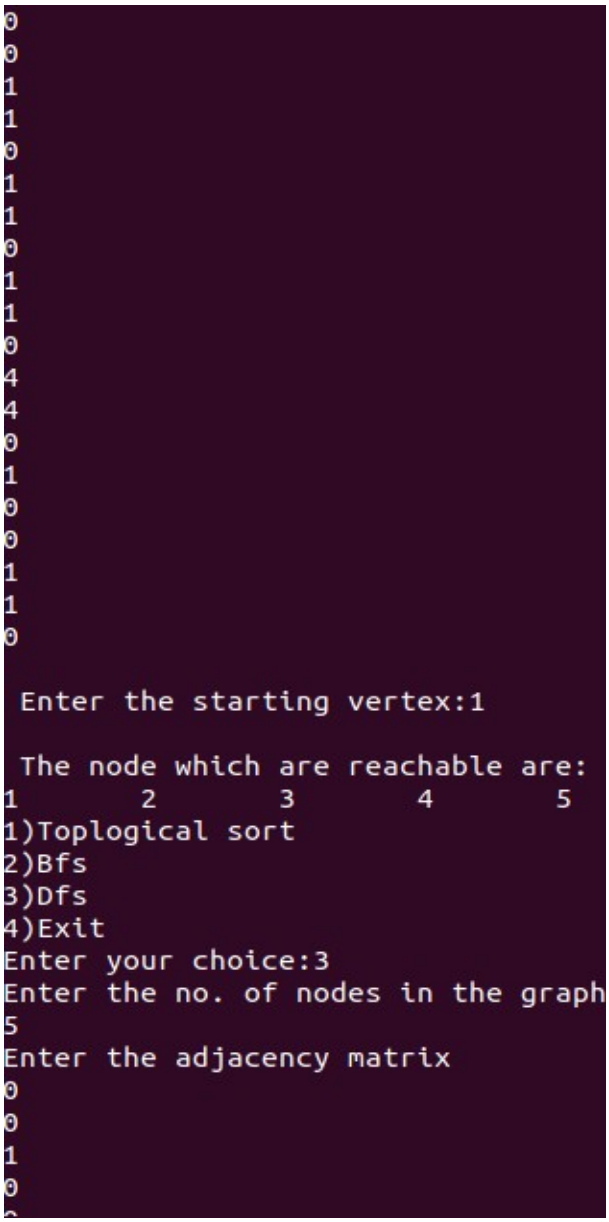
```
 visited_dfs[pop]=true;
 }
 else
 continue;
 for(i=n;i>=1;i--)
 {
 if(a[pop][i]==1 && visited_dfs[i]==false)
 {
 top++;
 stack[top]=i;
 }
 }
 }
}
void dfs_sort(){
        int i,j,v;
printf("Enter the no. of nodes in the graph\n");scanf("%d",&n);
printf("Enter the adjacency matrix \n");
for(i=1;i<=n;i++)
{
 for(j=1;j<=n;j++)
 {
 scanf("%d",&a[i][j]);
 }
}
printf("The adjacency matrix shown as\n");
for(i=1;i<=n;i++)
{
 for(j=1;j<=n;j++)
 {
 printf("%d ",a[i][j]);
 }
 printf("\n");
}
printf("Enter the starting node for Depth First search\n");
scanf("%d",&v);
for(i=1;i<=n;i++)
 visited_dfs[i]=false;
dfs(v);
 }

void top_sort(){
int i,j,k,n,a[10][10],indeg[10],flag[10],count=0;
printf("Enter the no of vertices:\n");
scanf("%d",&n);
printf("Enter the adjacency matrix:\n");
for(i=0;i<n;i++){
printf("Enter row %d\n",i+1);
for(j=0;j<n;j++)
scanf("%d",&a[i][j]);
}
for(i=0;i<n;i++){
```

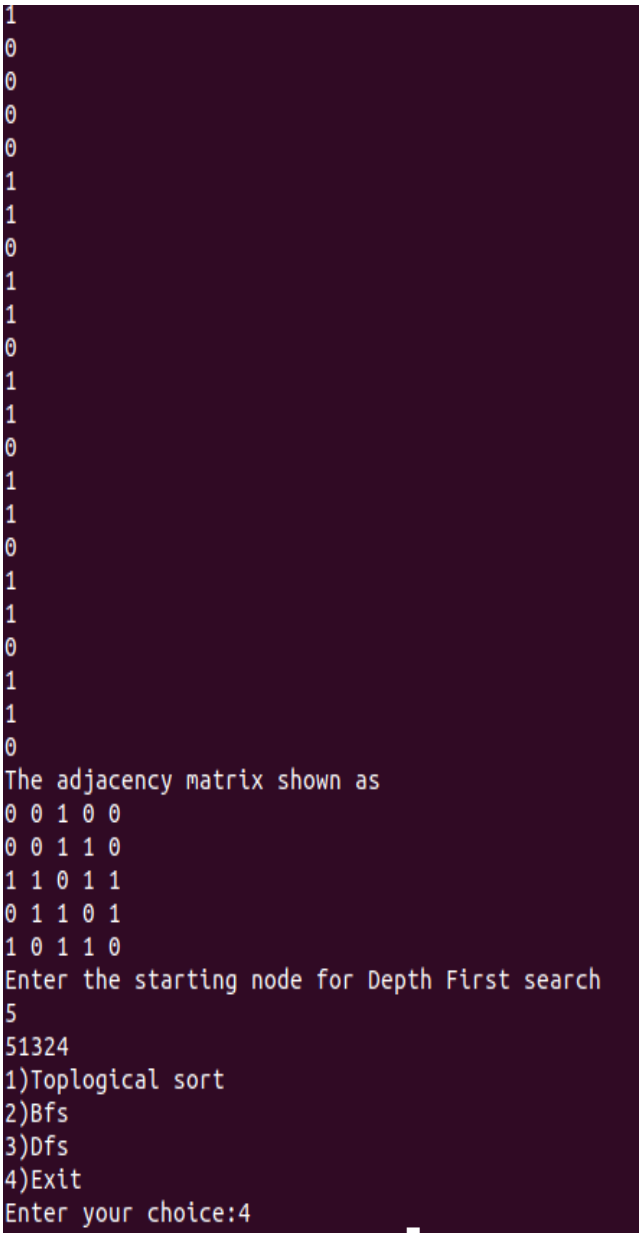
```
 indeg[i]=0;
 flag[i]=0;
 }
 for(i=0;i<n;i++)
 for(j=0;j<n;j++)
 indeg[i]=indeg[i]+a[j][i];
 printf("\nThe topological order is:");
 while(count<n){
 for(k=0;k<n;k++){
 if((indeg[k]==0) && (flag[k]==0)){
 printf("%d ",(k+1));
 flag [k]=1;
 }
 for(i=0;i<n;i++){
 if(a[i][k]==1)
 indeg[k]--;
 }
 }
 count++;
 }
}
void main(){
        int ch;
        do{
        printf("\n1)Toplogical sort\n2)Bfs\n3)Dfs\n4)Exit\n");
        printf("Enter your choice:");
        scanf("%d",&ch);
        switch(ch)
        {
         case 1:
                        top_sort();
                        break;
         case 2:
                        bfs_sort();
                        break;
         case 3:
                        dfs_sort();
                        break;
         case 4:
                        printf("Exit");
                        break;
         default:
                        printf("Enter a valid choice:");
        }
        }while(ch!=4);
}
```

OUTPUT:

```
1)Toplogical sort
2)Bfs
3)Dfs
4)Exit
Enter your choice:1
Enter the no of vertices:
6
Enter the adjacency matrix:
Enter row 1
0
0
0
0
0
0
Enter row 2
0
0
0
0
0
0
Enter row 3
0
0
0
1
0
0
Enter row 4
0
1
0
0
0
0
```

```
Enter row 5
1
1
0
0
0
0
Enter row 6
1
0
1
0
0
0

The topological order is:5 6 1 2 3 4
1)Toplogical sort
2)Bfs
3)Dfs
4)Exit
Enter your choice:2

 enter the number of vertices:5

 enter graph data in matrix form:
0
0
1
0
0
0
0
1
1
0
1
1
```





RESULT:

Program is successfully executed and output is obtained.

LAB CYCLE: 3
EXPERIMENT NO:12
DATE:27/01/2022

AIM:
Program for finding the Strongly connected Components in a directed graph.

ALGORITHM:

Step1 : Perform a depth first search on the whole graph.
Step2 : Reverse the original graph.
Step3 : Perform depth-first search on the reversed graph.

SOURCE CODE:

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

#define MAX_SIZE 5

struct Graph *graph;
struct Graph *gr;
int stack[MAX_SIZE], top;
// A structure to represent an adjacency list node
struct adj_list_node{
        int dest;
        //int weight;
        struct adj_list_node *next;
};
// A structure to represent an adjacency list
struct adj_list{
        struct adj_list_node *head;
};
// A structure to represent a graph
struct Graph{
        int V;
        int *visited;
        struct adj_list *array;
};

// Function to create a new adjacency list node
struct adj_list_node *new_adj_list_node(int dest){
        struct adj_list_node *newNode = (struct adj_list_node *)malloc(sizeof(struct
adj_list_node));
        newNode->dest = dest;
        //newNode->weight = weight;
        newNode->next = NULL;
        return newNode;
}
// Function to creates a graph with V vertices
```

```
struct Graph *create_graph(int V){
	struct Graph *graph = (struct Graph *)malloc(sizeof(struct Graph));
	graph->V = V;
	graph->array = (struct adj_list *)malloc(V * sizeof(struct adj_list));
	int i;
	for (i = 0; i < V; ++i)
		graph->array[i].head = NULL;
	return graph;
}

// Fuction to add edges to transpose graph
void get_transpose(struct Graph *gr, int src, int dest){
	struct adj_list_node *newNode = new_adj_list_node(src);
	newNode->next = gr->array[dest].head;
	gr->array[dest].head = newNode;
}

// Fuction to add edges to graph
void add_edge(struct Graph *graph, struct Graph *gr, int src, int dest){
	struct adj_list_node *newNode = new_adj_list_node(dest);
	newNode->next = graph->array[src].head;
	graph->array[src].head = newNode;
	get_transpose(gr, src, dest);
}
// Function to print the graph
void print_graph(struct Graph *graph1){
	int v;
	for (v = 0; v < graph1->V; ++v){
		struct adj_list_node *temp = graph1->array[v].head;
		while (temp){
			printf("(%d -> %d)\t", v, temp->dest);
			temp = temp->next;
		}
	}
}

void push(int x){
	if (top >= MAX_SIZE-1){
		printf("\n\tSTACK is over flow");
	}
	else{
		top++;
		stack[top] = x;
	}
}
// Function to pop item to stack
void pop(){
	if (top <= -1){
		printf("\n\t Stack is under flow");
	}
	else{
		top-- ;
```

```
        }
}
// Fuction to fill the stack
void set_fill_order(struct Graph *graph, int v, bool visited[], int *stack){
        visited[v] = true;
        int i = 0;
        struct adj_list_node *temp = graph->array[v].head;
        while (temp){
                if (!visited[temp->dest]){
                        set_fill_order(graph, temp->dest, visited, stack);
                }
                temp = temp->next;
        }
        push(v);
}
// A recursive function to print DFS starting from v
void dfs_recursive(struct Graph *gr, int v, bool visited[]){
        visited[v] = true;
        printf("%d ", v);
        struct adj_list_node *temp = gr->array[v].head;
        while (temp){
                if (!visited[temp->dest])
                dfs_recursive(gr, temp->dest, visited);
        temp = temp->next;
        }
}

void strongly_connected_components(struct Graph *graph, struct Graph *gr, int V){
        bool visited[V];
        for (int i = 0; i < V; i++)
                visited[i] = false;
        for (int i = 0; i < V; i++){
                if (visited[i] == false){
                        set_fill_order(graph, i, visited, stack);
                }
        }
        int count = 1;
        for (int i = 0; i < V; i++)
                visited[i] = false;
        while (top != -1){
                int v = stack[top];
                pop();
                if (visited[v] == false){
                        printf("Strongly connected component %d: \n", count++);
                        dfs_recursive(gr, v, visited);
                        printf("\n");
                }
        }
}

int main(){
        int v,max_edges,i,origin,destin;
```

```
        top = -1;
        printf("\n Enter the number of vertices: ");
        scanf("%d",&v);
        struct Graph *graph = create_graph(v);
        struct Graph *gr = create_graph(v);
        max_edges = v * (v - 1);
      for (i = 0; i <=max_edges; i++) {
        printf("Enter edge %d( 0 0 ) to quit : ", i);
            scanf("%d %d", &origin, &destin)    ;
            if ((origin == 0) && (destin == 0))
                  break;
            if (origin > v || destin > v || origin < 0 || destin < 0) {
                  printf("Invalid edge!\n");
                  i--;
            }
            else
                add_edge(graph, gr, origin, destin);

      }
        strongly_connected_components(graph, gr, v);
        return 0;
}
```

OUTPUT:

```
 Enter the number of vertices: 5
Enter edge 0( 0 0 ) to quit : 0
3
Enter edge 1( 0 0 ) to quit : 0
2
Enter edge 2( 0 0 ) to quit : 2
1
Enter edge 3( 0 0 ) to quit : 1
0
Enter edge 4( 0 0 ) to quit : 3
4
Enter edge 5( 0 0 ) to quit : 0
0
Strongly connected component 1:
0 1 2
Strongly connected component 2:
3
Strongly connected component 3:
4
```

RESULT:

Program is successfully executed and output is obtained.

LAB CYCLE: 3
EXPERIMENT NO:13
DATE:03/02/2022

AIM:
Program to implement Prim's Algorithm for finding the minimum cost spanning tree.

ALGORITHM:

Step1 : Associate with each vertex *v* of the graph a number *C*[*v*] (the cheapest cost of a connection to *v*) and an edge *E*[*v*] (the edge providing that cheapest connection). To initialize these values, set all values of *C*[*v*] to +∞ (or to any number larger than the maximum edge weight) and set each *E*[*v*] to a special flag value indicating that there is no edge connecting *v* to earlier vertices.

Step2 : Initialize an empty forest *F* and a set *Q* of vertices that have not yet been included in *F* (initially, all vertices).

Step3 : Repeat the following steps until *Q* is empty:

1. Find and remove a vertex *v* from *Q* having the minimum possible value of *C*[*v*]
2. Add *v* to *F* and, if *E*[*v*] is not the special flag value, also add *E*[*v*] to *F*
3. Loop over the edges *vw* connecting *v* to other vertices *w*. For each such edge, if *w* still belongs to *Q* and *vw* has smaller weight than *C*[*w*], perform the following steps:
    a) Set *C*[*w*] to the cost of edge *vw*
    b) Set *E*[*w*] to point to edge *vw*.

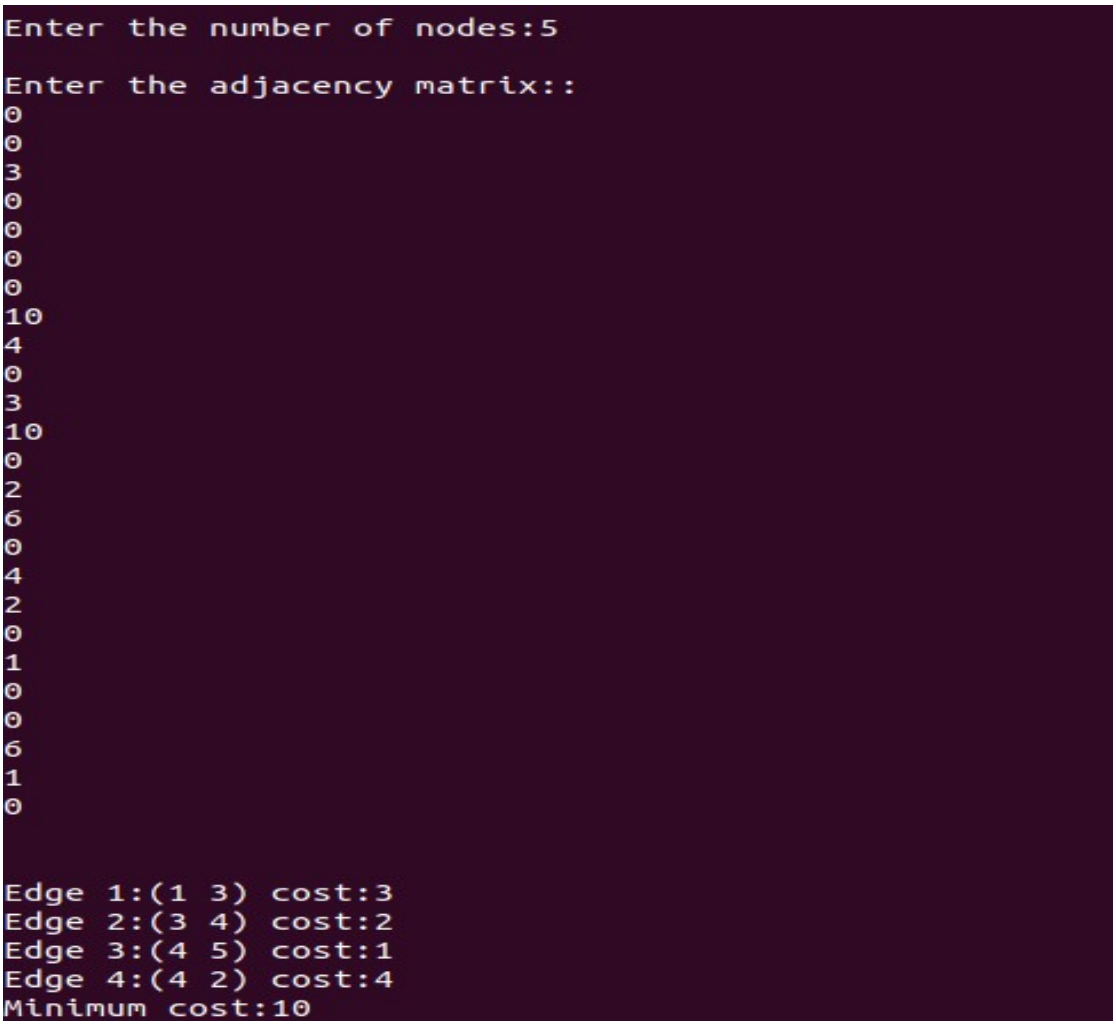
Step4 : Return *F*

SOURCE CODE:

```
#include <stdio.h>
int a,b,u,v,n,i,j,ne=1;
int visited[10]={0},min,mincost=0,cost[10][10];
void main()
{
printf("\nEnter the number of nodes:");
scanf("%d",&n);
printf("\nEnter the 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;
}
visited[1]=1;
printf("\n");
while(ne<n)
{
for(i=1,min=999;i<=n;i++)
for(j=1;j<=n;j++)
```

```
if(cost[i][j]<min)
if(visited[i]!=0)
{
min=cost[i][j];
a=u=i;
b=v=j;
}
if(visited[u]==0||visited[v]==0)
{
printf("\nEdge %d:(%d %d) cost:%d",ne++,a,b,min);
mincost+=min;
visited[b]=1;
}
cost[a][b]=cost[b][a]=999;
}
printf("\nMinimum cost:%d\n",mincost);
}
```

OUTPUT:



RESULT:

Program is successfully executed and output is obtained.

LAB CYCLE: 3
EXPERIMENT NO:14
DATE:03/02/2022

AIM:
Program to implement Kruskal's Algorithm using the Disjoint set data structure.

ALGORITHM:

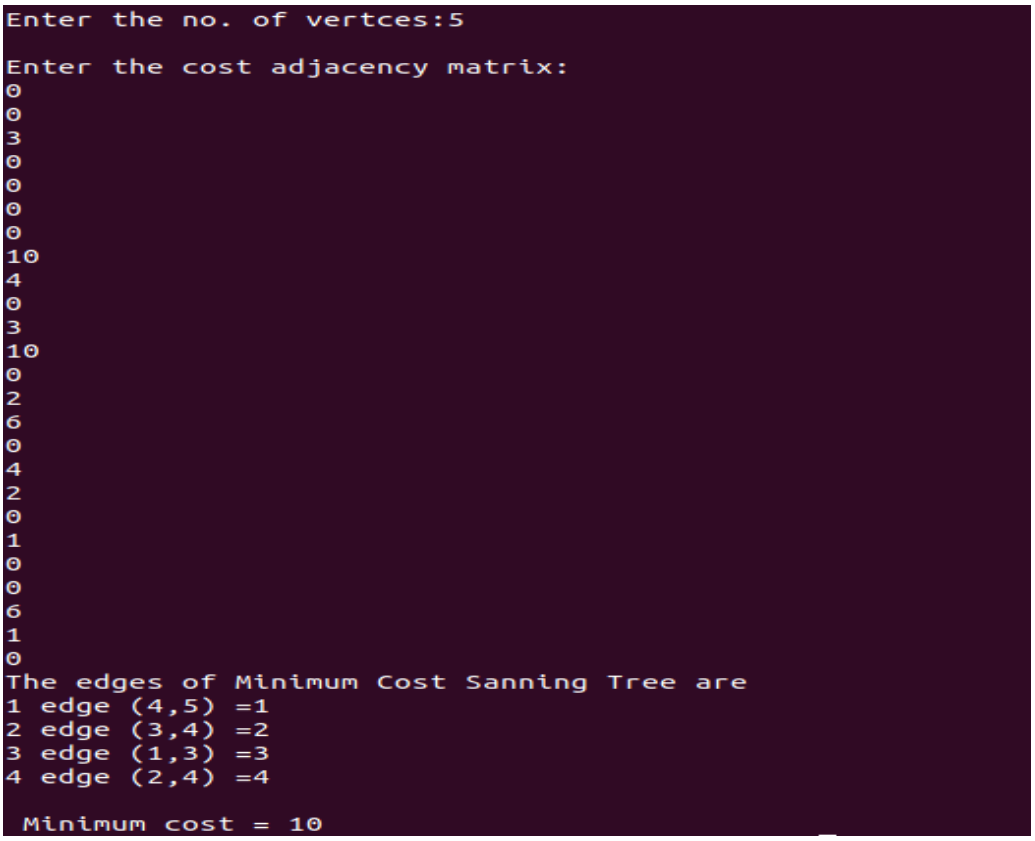
Step1 : Create a forest *F* (a set of trees), where each vertex in the graph is a separate tree.
Step2 : Create a set *E* containing all the edges in the graph
Step3 : Repeat Steps 4 and 5 while E is NOT EMPTY and F is not spanning
Step4 : Remove an edge from E with minimum weight
Step5 : If the edge obtained in Step4 connects two different trees, then add it to the forest (F)
Else, discard the edge
Step6 : End

SOURCE CODE:

```
#include <stdio.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("\nEnter the no. of vertces:");
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 Sanning 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 Minimum cost = %d\n",mincost);
}
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;}
```

OUTPUT:



RESULT:

Program is successfully executed and output is obtained.

LAB CYCLE: 3
EXPERIMENT NO:15
DATE:04/02/2022

AIM:
Program to implement Single Source shortest path algorithm

ALGORITHM:

Step1 : Let's create an array d[] where for each vertex v we store the current length of the shortest path from s to v in d[v]. Initially, d[s]=0 and for all other vertices this length equals infinity.
Step2 : Maintain a Boolean array u[] which stores for each vertex v whether it's marked. Initially all vertices are unmarked.
Step3 : Repeat steps 4 to 6 till the vertices ends
Step4 : Choose a starting vertex.
Step5 : Mark the selected vertex v.
Step6 : Perform relaxation from vertex v:
all edges of the form (v,t) are considered, and for each vertex t the algorithm tries to improve the value d[t].
If the length of the current edge equals len, the code for relaxation is:
d[t]=min(d[t],d[v]+len)
Step7 : Stop

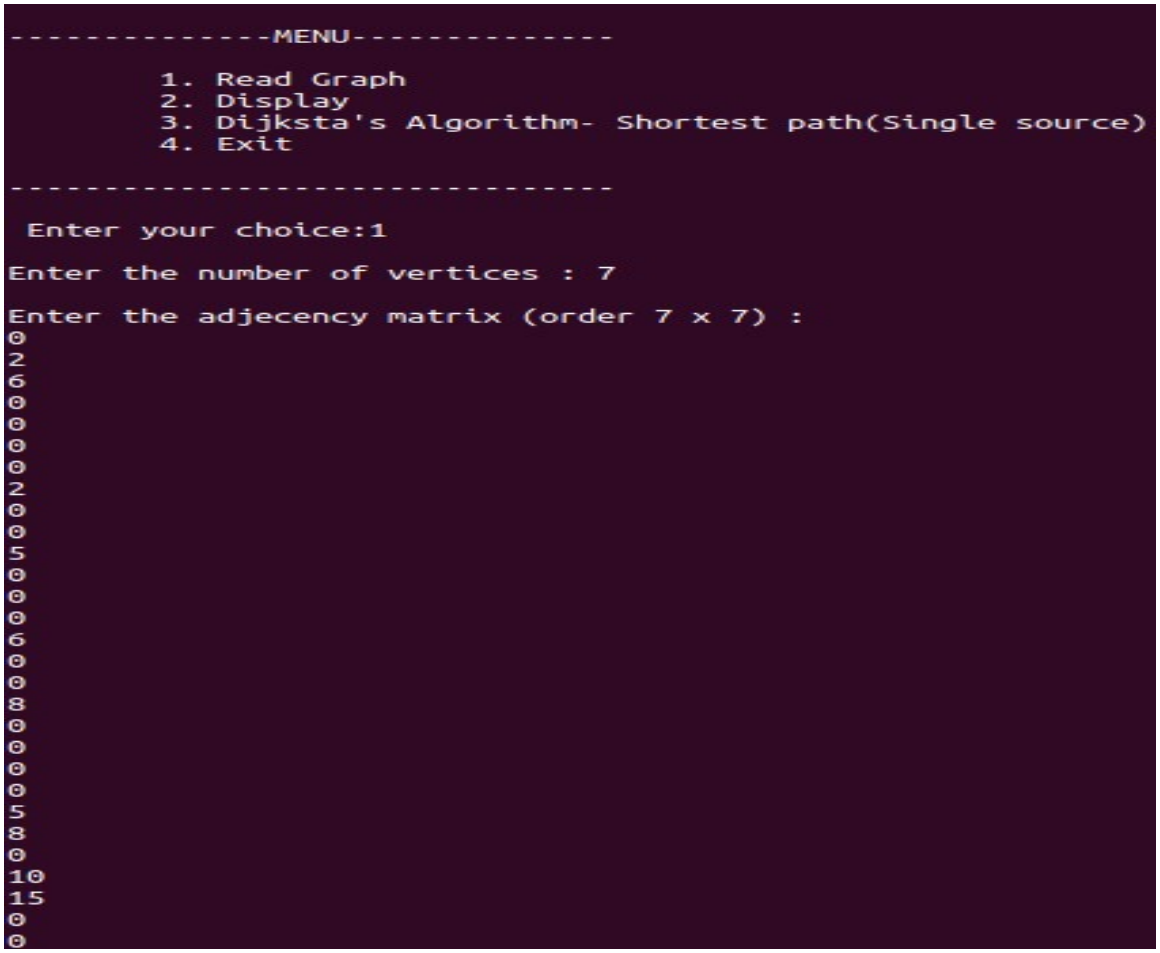
SOURCE CODE:

```
#include <stdio.h>
#define SIZE 10
#define INFINITY 999
void read_graph(int *nv, int adj[][SIZE])
{
   int i, j;
   printf("\nEnter the number of vertices : ");
   scanf("%d", nv);
   printf("\nEnter the adjecency matrix (order %d x %d) :\n", *nv, *nv);
   for (i = 0; i < *nv; i++)
      for (j = 0; j < *nv; j++)
         scanf("%d", &adj[i][j]);
}
void Dijkstra(int adj[][SIZE], int *nv, int start, int distance[])
{
   int cost[SIZE][SIZE], pred[SIZE];
   int visited[SIZE], count, mindistance, nextnode, i, j;
   if (!*nv)
   {
      printf("\nPlease read a graph...\n");
      return;
   }
   for (i = 0; i < *nv; i++)
```

```
for (j = 0; j < *nv; j++)
        if (adj[i][j] == 0)
           cost[i][j] = INFINITY;
        else
           cost[i][j] = adj[i][j];
   for (i = 0; i < *nv; i++)
   {
      distance[i] = cost[start][i];
      pred[i] = start;
      visited[i] = 0;
   }
   distance[start] = 0;
   visited[start] = 1;
   count = 1;
   while (count < *nv - 1)
   {
      mindistance = INFINITY;
      for (i = 0; i < *nv; i++)
         if (distance[i] < mindistance && !visited[i])
         {
            mindistance = distance[i];
            nextnode = i;
         }
      visited[nextnode] = 1;
      for (i = 0; i < *nv; i++)
         if (!visited[i])
            if (mindistance + cost[nextnode][i] < distance[i])
            {
               distance[i] = mindistance + cost[nextnode][i];
               pred[i] = nextnode;
            }
      count++;
   }
printf("\nSuccessfully created shortest path vector beased on the given start vertex %d \n",
start);
for(i = 0; i < *nv; i++)
if(i != start)
{
      printf("\nDistance from source to %d: %d", i, distance[i]);
}}
void display(int adj[][SIZE], int *nv, int flag, int distance[], int start)
{
   int i, j;
   if (!*nv)
   {
      printf("\nPlease read a graph...\n");
      return;
   }
   printf("\nThe given graph (adjacency matrix) is:\n");
   for (i = 0; i < *nv; i++)
   {
```

```
        for (j = 0; j < *nv; j++)
            printf("%d ", adj[i][j]);
        printf("\n");
    }
    if (flag)
    {
        for (i = 0; i < *nv; i++)
            if (i != start)
            {
                printf("\nDistance from source to %d: %d", i, distance[i]);
            }
    }
}
int main()
{
    int adj[SIZE][SIZE],distance[SIZE];
   int nv;
    int start = 0;
    int flag = 0;
    int e = 1, ch;
    while (e)
    {
        printf("\n----------MENU----------\n");
printf( "\n\t1. Read Graph\n\t2. Display\n\t3. Dijksta's Algorithm- Shortest path(Single
source)\n\t4. Exit\n" );
printf( "\n---------------------------------\n" );
printf( "\n Enter your choice:" );
scanf( "%d", &ch );
switch( ch )
{
        case 1:
            read_graph(&nv, adj);
            break;
        case 2:
            display(adj, &nv, flag, distance, start);
            break;
        case 3:
            flag = 1;
            Dijkstra(adj, &nv, start, distance);
            break;
        case 4:
            e = 0;
            break;
        default:
            printf("\n Invalid choice \n");
             return 0;
```

OUTPUT:


--------------MENU--------------
1. Read Graph
2. Display
3. Dijksta's Algorithm- Shortest path(Single source)
4. Exit

Enter your choice:1
Enter the number of vertices : 7
Enter the adjecency matrix (order 7 x 7) :
0
2
6
0
0
0
0
2
0
0
5
0
0
0
6
0
0
8
0
0
0
0
5
8
0
10
15
0
0

```
15
6
0
6
0
0
0
0
2
6
0

--------------MENU--------------

        1. Read Graph
        2. Display
        3. Dijksta's Algorithm- Shortest path(Single source)
        4. Exit

---------------------------------

 Enter your choice:2

The given graph (adjacency matrix) is:
0 2 6 0 0 0 0
2 0 0 5 0 0 0
6 0 0 8 0 0 0
0 5 8 0 10 15 0
0 0 0 10 0 6 2
0 0 0 15 6 0 6
0 0 0 0 2 6 0
```

```
-------------MENU-------------

       1. Read Graph
       2. Display
       3. Dijksta's Algorithm- Shortest path(Single source)
       4. Exit

------------------------------

 Enter your choice:3

Successfully created shortest path vector beased on the given start vertex 0

Distance from source to 1: 2
Distance from source to 2: 6
Distance from source to 3: 7
Distance from source to 4: 17
Distance from source to 5: 22
Distance from source to 6: 19
```

RESULT:

Program is executed successfully and output is obtained.