Experiment No: 1 **Date: 14/10/24**

SINGLY-LINKED STACK

AIM

Implementation of Singly Linked Stack - Push, Pop, Linear Search.

ALGORITHM

1. **Define node with data and link.**
   - Initialize top = NULL.
2. **isEmpty():**
   - Return 1 if top == NULL, else 0.
3. **push(data):**
   - Create a new node with data.
   - Set newNode->link = top and update top = newNode.
4. **pop():**
   - If empty, print "Underflow".
   - Store top->data, update top = top->link, free old node, and return value.
5. **peek():**
   - Return top->data or print "Underflow" if empty.
6. **search(item):**
   - Traverse from top, check for item, and print "Found" or "Not found".
7. **print():**
   - Traverse and display all elements from top.

Main Menu

- Loop through options: Push, Pop, Peek, Print, Search, Exit.
- Call respective functions based on user input.

SOURCE CODE

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node*link;
};
struct node *top = NULL;
int isEmpty()
{
  if(top==NULL)
  {
    return 1;
  }
  else{
    return 0;
   }
}
void push(int data)
{
   struct node *newNode;
   newNode = malloc(sizeof(newNode));
   if(newNode==NULL)
   {
      printf("Stack underflow\n");
   }
```

```
    newNode->data=data;
    newNode->link=NULL;
    newNode->link=top;
    top=newNode;
}
int pop()
{
    struct node*temp;
    int val;
    if(isEmpty())
    {
      printf("Stack Underflow");
    }
    temp=top;
    val=temp->data;
    top=temp->link;
    free(temp);
    return val;
}
int peek()
{

        if(isEmpty())
        {
                printf("Stack underflow");
        }
        return top->data;
}
```

```
void search(int item)
{
	struct node *temp=top;
	int flag=0;
	while(temp!=NULL)
	{
		if(item==temp->data)
		{
			printf("%d is Included in the stack\n",temp->data);
			flag=1;
		}
		temp=temp->link;
	}
	if(flag!=1){
		printf("\nElement not found\n");
	}
}
void print()
{
	struct node*temp=top;
	if(isEmpty())
	{
		printf("Stack Underflow\n");
	}
	printf("The Stack Elements are\n\n");
	while(temp)
	{			printf("%d\t",temp->data);
		temp=temp->link;
```

```
        }
        printf("\n");
}
int main()
{
        int choice,data,sitem;
        while(1)
        {
                printf("\n");
                printf("1.Push\n2.Pop\n3.Print the Top element\n4.Print all the Stack
Element\n5.Search Element\n6.Exit");
                printf("\n\nPlease Enter your choice\n");
                scanf("%d",&choice);
                switch(choice)
                {
                        case 1:
                         printf("Enter the element to be Pushed\n");
                         scanf("%d",&data);
                         push(data);
                         break;
                        case 2:
                         data=pop();
                         printf("%d Element Removed\n",data);
                         break;
                        case 3:
                         printf("The Top most Element of the Stack is %d\n",peek());
                         break;
                        case 4:
                         print();
```

```
                break;
               case 5:
                printf("Enter the item to be searched\n");
                scanf("%d",&sitem);
                search(sitem);
                break;
               case 6:
                exit(1);
               default:
                printf("!!!Wrong Choice!!!\n");
          }
     }
     return 0;
}
```

RESULT

The stack using linked list program is executed successfully and the output is verified

Experiment No: 2 **Date: 14/10/24**

SINGLY-LINKED LIST

AIM

Implementation of singly linked list-Insertion, Deletion.

ALGORITHM

Global Declarations

1. Define node with data and link.
2. Initialize head = NULL.

Insertion

1. **in_beg(data):** Create new node, set head = newnode.
2. **in_end(data):** Create new node, traverse to last, and add.
3. **in_pos(pos, data):** Validate position, insert at pos.

Deletion

1. **del_beg():** Update head if not empty.
2. **del_end():** Traverse and remove last node.
3. **del_pos(pos):** Validate position, delete node at pos.

Utility

1. **size_of_list():** Count nodes.
2. **display():** Print list or "Empty".

Main Function

1. Menu: Insert, Delete, Display, Exit.
2. Switch for operations.

SOURCE CODE

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
	int data;
	struct node* link;
};
struct node *head=NULL;
void in_beg(int data)
{
	struct node *newnode=(struct node*)malloc(sizeof(struct node));
	newnode->data=data;
	newnode->link=NULL;
	if(head==NULL)
	{
		head=newnode;
	}
	else
	{
		newnode->link=head;
		head=newnode;
	}
	printf("%d inserted at the beginning.\n",newnode->data);
}
void in_end(int data)
{
```

```
    struct node *newnode=(struct node*)malloc(sizeof(struct node));
    struct node *temp=head;
    newnode->data=data;
    newnode->link=NULL;
    if(head==NULL)
    {
        head=newnode;
    }
    else
    {
        while(temp->link!=NULL)
        {
            temp=temp->link;
        }
        temp->link=newnode;
    }
    printf("%d inserted at the end.\n",newnode->data);
}
int size_of_list()
{
    struct node* temp = head;
    int count = 0;

    while(temp != NULL)
    {
      count ++;
      temp = temp->link;
    }
```

```
        return count;
}
void in_pos(int pos,int data)
{
        struct node *newnode=(struct node*)malloc(sizeof(struct node));
        struct node *temp=head;
        newnode->data=data;
        newnode->link=NULL;
        int count=0;
        count=size_of_list();
        if(head==NULL&&(pos<=0||pos>1))
        {
           printf("\nInvalid position to insert a node\n");
           return;
        }
        if(head!=NULL&&(pos<=0||pos>count+1))
        {
           printf("\nInvalid position to insert a node\n");
           return;
        }
        if(pos==1)
        {
               in_beg(data);
               return;
        }
        else

        {
```

```
            while(pos!=2)
            {
                  temp=temp->link;
                  pos--;
            }
      newnode->link=temp->link;
      temp->link=newnode;
      }
      printf("%d inserted.\n",newnode->data);
}
void del_beg()
{
      if(head==NULL)
      {
            printf("List is already empty.\n");
            return;
      }
            struct node *temp=head;
            head=head->link;
            printf("Deleted %d from the beginning.\n", temp->data);
            free(temp);
            temp=NULL;
}
void del_end()
{
   struct node *temp = head;

   struct node *temp2 = NULL;
```

```
    if (head == NULL)
    {
        printf("List is already empty.\n");
        return;
    }
    if (head->link == NULL)
    {
        free(head);
        head = NULL;
        printf("List is now empty.\n");
        return;
    }
    while (temp->link != NULL)
    {
        temp2 = temp;
        temp = temp->link;
    }
    temp2->link = NULL;
    printf("Deleted %d from the end.\n", temp->data);
    free(temp);
    temp=NULL;
}
void del_pos(int pos)
{
    int count = 0;

    count=size_of_list();
    struct node *prev,*curr=head;
```

```
if(head == NULL)
{
   printf("List is empty.\n");
   return;
}
else if(pos<=0||pos>count)
{
   printf("\nInvalid position to delete a node\n");
   return;
}
else if(pos==1)
{
   del_beg();
}
else if(count==pos)
{
   del_end();
}
else
{
   while(pos!=1)
   {
      prev=curr;
      curr=curr->link;
      pos--;
   }

   prev->link=curr->link;
```

```
        free(curr);
        curr=NULL;
        printf("Node deleted.\n");
    }
}
void display()
{
        struct node *ptr=head;
        if(head==NULL)
        {
                printf("List is empty\n");
                return;
        }
        while(ptr!=NULL)
        {
                printf("%d \n",ptr->data);
                ptr=ptr->link;
        }
}
int main()
{
int choice,data,pos;
while(1)
{
        printf("\n1.Insert at the beginning\n2.Insert at the end\n3.Insert at any
position\n4.Display\n5.Delete from the beginning\n6.Delete from the end.\n7.Delete
from any position\n8.Exit");
```

```
 printf("\nEnter your choice:");
scanf("%d",&choice);
switch(choice)
{
      case 1: printf("Enter the data to be inserted at the beginning:\n");
            scanf("%d",&data);
            in_beg(data);
            break;
      case 2: printf("Enter the data to be inserted at the end:\n");
            scanf("%d",&data);
            in_end(data);
            break;
      case 3: printf("Enter the data to be inserted :\n");
            scanf("%d",&data);
            printf("Enter the position to be inserted :\n");
            scanf("%d",&pos);
            in_pos(pos,data);
            break;
      case 4: display();
            break;
      case 5: del_beg();
            break;
      case 6: del_end();
            break;
      case 7: printf("Enter the position to be deleted:\n");
            scanf("%d",&pos);
            del_pos(pos);
            break;
```

```
        case 8: exit(0);
            break;
        default:printf("\nWrong Input!");
    }
}
return 0;
}
```

RESULT

The singly linked list program is executed successfully and the output is verified

Experiment No: 3 **Date: 21/10/24**

DOUBLY LINKED LIST

AIM

Implementation of Doubly linked list - Insertion, Deletion, Search.

ALGORITHM

Global Declarations

1. Define node with prev, data, and next.
2. Initialize head = NULL.

Insertion

1. **in_beg(data):** Create new node, adjust prev and next pointers, update head.
2. **in_end(data):** Create new node, traverse to last, and add.
3. **in_pos(pos, data):** Validate position, insert at pos.

Deletion

1. **del_beg():** Update head if not empty, adjust prev pointer.
2. **del_end():** Traverse and remove last node.
3. **del_pos(pos):** Validate position, delete node at pos.

Utility

1. **size_of_list():** Count nodes.
2. **display():** Print list or "Empty".

Main Function

1. Menu: Insert, Delete, Display, Exit.
2. Switch for operations.

SOURCE CODE

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
	struct node* prev;
	int data;
	struct node* next;
};
struct node *head=NULL;
void in_beg(int data)
{
	struct node *newnode=(struct node*)malloc(sizeof(struct node));
	newnode->prev=NULL;
	newnode->data=data;
	newnode->next=NULL;
	if(head==NULL)
	{
		head=newnode;
	}
	else
	{
		newnode->next=head;
		head->prev=newnode;
		head=newnode;
	}
	printf("%d inserted at the beginning.\n",newnode->data);
```

```
}
void in_end(int data)
{
    struct node *newnode=(struct node*)malloc(sizeof(struct node));
    newnode->prev=NULL;
    struct node *temp=head;
    newnode->data=data;
    newnode->next=NULL;
    if(head==NULL)
    {
        head=newnode;
    }
    else
    {
        while(temp->next!=NULL)
        {
            temp=temp->next;
        }
        temp->next=newnode;
        newnode->prev=temp;
    }
    printf("%d inserted at the end.\n",newnode->data);
}
int size_of_list()
{
    struct node* temp = head;
    int count = 0;
    while(temp != NULL)
```

```
        {
           count ++;
           temp = temp->next;
        }
        return count;
}
void in_pos(int pos,int data)
{
        struct node *newnode=(struct node*)malloc(sizeof(struct node));
        struct node *temp=head;
        newnode->prev=NULL;
        newnode->data=data;
        newnode->next=NULL;
        int count=0;
        count=size_of_list();
        if(head==NULL&&(pos<=0||pos>1))
        {
           printf("\nInvalid position to insert a node\n");
           return;
        }
        if(head!=NULL&&(pos<=0||pos>count+1))
        {
           printf("\nInvalid position to insert a node\n");
           return;
        }
        if(pos==1)
        {
              in_beg(data);
```

```
        return;
    }
    else if(count+1==pos)
    {
        in_end(data);
        return;
    }
    else
    {
        while(pos!=2)
        {
            temp=temp->next;
            pos--;
        }
    temp->next->prev=newnode;
    newnode->next=temp->next;

    temp->next=newnode;
    newnode->prev=temp;
    }
    printf("%d inserted.\n",newnode->data);
}
void del_beg()
{

    if(head==NULL)
    {
        printf("List is already empty.\n");
```

```
        return;
    }
        struct node *temp=head;
        head=head->next;
        printf("Deleted %d from the beginning.\n", temp->data);
        free(temp);
        temp=NULL;
}
void del_end()
{
  struct node *temp = head;
  struct node *temp2 = NULL;

  if (head == NULL)
  {
    printf("List is already empty.\n");
    return;
  }
  if (head->next == NULL)
  {
    free(head);
    head = NULL;
    printf("List is now empty.\n");
    return;
  }
  while (temp->next != NULL)
  {
    temp2 = temp;
```

```
        temp = temp->next;
    }
    temp2->next = NULL;
    printf("Deleted %d from the end.\n", temp->data);
    free(temp);
    temp=NULL;
}
void del_pos(int pos)
{
    int count = 0;
    count=size_of_list();
    struct node *prev,*curr=head;
    if(pos<=0||pos>count)
    {
        printf("\nInvalid position to delete a node\n");
        return;
    }
    if(head == NULL) {
        printf("List is empty.\n");
        return;
    }
    else if(pos==1)
    {
        del_beg();
    }
    else if(count==pos)
    {
        del_end();
```

```
    }
    else
    {
        while(pos!=1)
        {
            prev=curr;
            curr=curr->next;
            pos--;
        }
        prev->next=curr->next;
        curr->next->prev=prev;
        free(curr);
        curr=NULL;
        printf("Node deleted.\n");
    }
}
void display(){
        struct node *ptr=head;
        if(head==NULL)
        {
                printf("List is empty\n");
                return;
        }
        while(ptr!=NULL)
        {
                printf("%d \n",ptr->data);
                ptr=ptr->next;
        }
```

```
}
int main()
{
int choice,data,pos;
while(1)
{
	printf("\n1.Insert at the beginning\n2.Insert at the end\n3.Insert at any
position\n4.Display\n5.Delete from the beginning\n6.Delete from the end.\n7.Delete
from any position\n8.Exit");
	printf("\nEnter your choice:");
	scanf("%d",&choice);
	switch(choice)
	{
		case 1: printf("Enter the data to be inserted at the beginning:\n");
			scanf("%d",&data);
			in_beg(data);
			break;
		case 2: printf("Enter the data to be inserted at the end:\n");
			scanf("%d",&data);
			in_end(data);
			break;
		case 3: printf("Enter the data to be inserted :\n");
			scanf("%d",&data);
			printf("Enter the position to be inserted :\n");
			scanf("%d",&pos);
			in_pos(pos,data);
			break;
		case 4: display();
			break;
```

```
        case 5: del_beg();
            break;
        case 6: del_end();
            break;
        case 7: printf("Enter the position to be deleted:\n");
            scanf("%d",&pos);
            del_pos(pos);
            break;
        case 8: exit(0);
            break;
        default:printf("\nWrong Input!");
    }
}
return 0;
}
```

RESULT

The doubly linked list program is executed successfully and the output is verified

Experiment No: 4 **Date: 04/11/24**

BINARY SEARCH TREE

AIM

Implementation of Binary Search Trees- Insertion, Deletion, Search.

ALGORITHM

Global Declarations

1. Define Node structure with data, left, and right.
2. Initialize root = NULL.

Functions

1. **createnode(data):** Create a node with data, set left and right to NULL.
2. **insert(root, data):** Insert a node in the correct position (left or right) based on value.
3. **findMin(root):** Find the leftmost node (minimum value).
4. **deleteNode(root, data):** Delete a node, handle 3 cases (no child, one child, two children).
5. **search(root, data):** Search for a node with the specified value.
6. **preorder(root):** Print root, then left and right children recursively.
7. **inorder(root):** Print left, then root, then right children recursively.
8. **postorder(root):** Print left, then right, then root recursively.

Main Function

1. Menu with options: Insert, Delete, Search, Preorder, Inorder, Postorder, Exit.
2. Use a switch statement for user input, calling the respective function.
3. Exit on option 7.

SOURCE CODE

```
#include<stdio.h>
#include<stdlib.h>
struct Node{
int data;
struct Node* left;
struct Node* right;
};
//struct Node* root=NULL;
//functn to create a new node
struct Node* createnode(int data)
{
	struct Node* newNode=(struct Node*)malloc(sizeof(struct Node));
	newNode->data=data;
	newNode->left=NULL;
	newNode->right=NULL;
	return newNode;
}

struct Node* insert(struct Node* root,int data)
{
	if(root==NULL)
	{
		root=createnode(data);
	}
	else if(data<root->data)
```

```
        {
            root->left=insert(root->left,data);
        }
        else if(data>root->data)
        {
            root->right=insert(root->right,data);
        }
        return root;
}
struct Node* findMin(struct Node *root)
{
        while(root && root->left!=NULL)
        {
        root=root->left;
        }
        return root;
}
//dlt a node from bst
struct Node* deleteNode(struct Node* root,int data){
if(root==NULL){
        printf("The value to be deleted is not present in the tree\n");
        return root;
        }
if(data<root->data){
        root->left=deleteNode(root->left,data);
}else if(data>root->data){
        root->right=deleteNode(root->right,data);
}
```

```
else{
//node with one child or no child
        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;
        }
//node with 2 children
        struct Node* temp=findMin(root->right);
        root->data=temp->data;
        root->right=deleteNode(root->right,temp->data);
        }
        return root;
}
//search a node in bst
struct Node* search(struct Node* root,int data){
if(root==NULL||root->data==data){
        return root;
        }

if(data<root->data){
        return search(root->left,data);
        }
else{
```

```
        return search(root->right,data);
        }
}
//preorder traversal
void preorder(struct Node* root){
if(root!=NULL){
        printf("%d\t",root->data);
        preorder(root->left);
        preorder(root->right);
        }
}
//inorder traversal
void inorder(struct Node* root){
if(root!=NULL){
        inorder(root->left);
        printf("%d\t",root->data);
        inorder(root->right);
        }
}
//postorder traversal
void postorder(struct Node* root){
if(root!=NULL){
        postorder(root->left);
        postorder(root->right);

        printf("%d\t",root->data);
        }
}
```

```
int main(){
        struct Node* root=NULL;
        int choice,value;
        struct Node* foundNode;
        while(1){
                printf("1.INSERT NODE\n2.DELETE NODE\n3.SEARCH 
NODE\n4.PREORDER TRAVERSAL\n5.INORDER TRAVERSAL\n6.POSTORDER 
TRAVERSAL\n7.EXIT\n");
                printf("Enter your choice:");
                scanf("%d",&choice);
                switch(choice){
                        case 1:
                                printf("enter the value to be inserted :");
                                scanf("%d",&value);
                                root=insert(root,value);
                                break;
                        case 2:
                                if(root==NULL){
                                        printf("tree is empty \n");
                                }
                                else{
                                        printf("enter the value to delete:");
                                        scanf("%d",&value);
                                        root=deleteNode(root,value);
                                        }
                                break;

                        case 3:
                                if(root==NULL){
```

```
                printf("tree is empty");
        }
        else{
                printf("enter value to search:");
                scanf("%d",&value);
                foundNode=search(root,value);
                if(foundNode!=NULL){
                                printf("value %d found in the tree ",value);
                        }else{
                                printf("value %d not found in the tree",value);
                        }
                }
        break;
case 4:
if(root==NULL){
                printf("tree is empty");
        }
        else{
                printf("preorder traversal:\n");
                preorder(root);
                printf("\n");
                }
        break;
case 5:
        if(root==NULL){
                printf("tree is empty");


        }
```

```
            else{
                printf("inorder traversal:\n");
                inorder(root);
                printf("\n");
                }
            break;
        case 6:
            if(root==NULL){
                printf("tree is empty");
            }
            else{
                printf("postorder traversal:\n");
                postorder(root);
                printf("\n");
                }
            break;
        case 7:
            exit(0);
        default:
            printf("invalid choice!please try again\n");
        }
    }
return 0;
}
```

RESULT

The Implementation of Binary Search Trees program is executed successfully and the output is verified

Experiment No: 5 **Date: 17/11/24**

CIRCULAR QUEUE

AIM

Implementation of Circular Queue - Add, Delete, Search.

ALGORITHM

Global Declarations

1. Define integer pointer queue, size, and initialize front, rear to -1.

initializeQueue()

1. Allocate memory for the queue.

enqueue(element)

1. If full, print "QUEUE IS FULL".
2. If empty, set front = rear = 0.
3. Otherwise, increment rear and insert element.

dequeue()

1. If empty, print "QUEUE IS EMPTY".
2. Remove element at queue[front] and increment front.

searchElement(element)

1. If empty, print "QUEUE IS EMPTY".
2. Traverse and return position or -1.

displayQueue()

1. If empty, print "QUEUE IS EMPTY".
2. Print elements from front to rear.

Main Function

1. Initialize queue, display menu, and call appropriate functions.

SOURCE CODE

```
#include <stdio.h>
#include <stdlib.h>
int *queue;
int size;
int front = -1, rear = -1;
void initializeQueue() {
   queue = (int *)malloc(size * sizeof(int));
}
void enqueue(int element) {
   if (front == (rear + 1) % size) {
      printf("\nQUEUE IS FULL\n");
      return;
   }
   if (front == -1 && rear == -1) {
      front = rear = 0;
   } else {
      rear = (rear + 1) % size;
   }
   queue[rear] = element;
   printf("\n%d is Inserted\n",element);
}

int dequeue() {
   int element;
   if (front == -1 && rear == -1) {
      printf("\nQUEUE IS EMPTY\n");
```

```
        return -1;
    }
    element = queue[front];
    if (front == rear) {
        front = rear = -1;
    } else {
        front = (front + 1) % size;
    }
    printf("\n%d ELEMENT IS DELETED FROM THE QUEUE\n", element);
    return element;
}
int searchElement(int element) {
    if (front == -1 && rear == -1) {
        printf("\nQUEUE IS EMPTY\n");
        return -1;
    }
    int current = front;
    int position = 1;
    do {
        if (queue[current] == element) {
            return position;
        }
        current = (current + 1) % size;
        position++;
    } while (current != (rear + 1) % size);
    return -1;
}
void displayQueue() {
```

```
    if (front == -1 && rear == -1) {
        printf("\nQUEUE IS EMPTY\n");
        return;
    }
    printf("QUEUE ELEMENTS ARE: ");
    int current = front;
    do {
        printf("%d ", queue[current]);
        current = (current + 1) % size;
    } while (current != (rear + 1) % size);
    printf("\n");
}
int main() {
    int choice, searchResult, element;
    printf("ENTER THE SIZR OF THE QUEUE: ");
    scanf("%d", &size);
    initializeQueue();
    do {
        printf("\nCIRCULAR QUEUE MENU\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Search Element\n");
        printf("4. Display\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
```

```
            printf("Enter the element to enqueue: ");
            scanf("%d", &element);
            enqueue(element);
            break;
        case 2:
            dequeue();
            break;
        case 3:
            printf("Enter the element to search: ");
            scanf("%d", &element);
            searchResult = searchElement(element);
            if (searchResult != -1) {
                printf("%d found at position %d\n", element, searchResult);
            } else {
                printf("%d not found in the queue\n", element);
            }
            break;
        case 4:
            displayQueue();
            break;
        case 5:
            printf("Exiting!!!.\n");
            break;
        default:
            printf("Invalid choice. Please enter a valid option.\n");
            break;
    }
} while (choice != 5);
```

```
    free(queue);
    return 0;
}
```

RESULT

The Implementation of Circular Queue program is executed successfully and the output is verified.

Experiment No: 6 **Date: 11/11/24**

SET DATA STRUCTURE AND SET OPERATIONS

AIM

Implementation of Set Data Structure and set operations (Union, Intersection and Difference) using BitString.

ALGORITHM

Global Declarations

1. Define arrays for superSet, setA, setB, bitStringA, bitStringB.
2. Initialize size variables.

getUniversalSet()

1. Input size and elements for the Universal Set.

getSet(arr[], size)

1. Input elements for Set A or Set B.
2. Ensure elements are in the Universal Set.

checkSetInUniversal()

1. Check if each element exists in the Universal Set.

generateBitStrings()

1. Set corresponding bits in bitStringA and bitStringB for Set A and Set B.

setUnion()

1. Perform bitwise OR and print the union.

setIntersection()

1. Perform bitwise AND and print the intersection.

setDifferenceAminusB()

1. Perform bitwise AND with complement and print A - B.

setDifferenceBminusA()

1. Perform bitwise AND with complement and print B - A.

printBitString()

1. Display the bit string.

printSetFromBitString()

1. Display elements corresponding to the bit string.

main()

1. Input and validate sets.
2. Generate bit strings.
3. Display menu for operations.

SOURCE CODE

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 20
int superSet[MAX_SIZE], superSetSize = 0;
int setA[MAX_SIZE], setASize = 0;
int setB[MAX_SIZE], setBSize = 0;
int bitStringA[MAX_SIZE], bitStringB[MAX_SIZE];
// Function prototypes
void getUniversalSet();
void getSet(int arr[], int *size);
int checkSetInUniversal(int arr[], int size);
void generateBitStrings();

void setUnion();
void setIntersection();
```

```
void setDifferenceAminusB();
void setDifferenceBminusA();
void printBitString(int arr[], int size);
void printSetFromBitString(int arr[], int size);
void getUniversalSet() {
    printf("Enter Universal Set Size (max %d): ", MAX_SIZE);
    scanf("%d", &superSetSize);
    if (superSetSize > MAX_SIZE) {
        printf("Error: Size exceeds maximum limit.\n");
        exit(1);
    }
    printf("Enter %d elements for the Universal Set:\n", superSetSize);
    for (int i = 0; i < superSetSize; i++) {
        printf("Element %d: ", i + 1);
        scanf("%d", &superSet[i]);
    }
}
void getSet(int arr[], int *size) {
    printf("Enter %d elements (must be in the Universal Set):\n", *size);
    for (int i = 0; i < *size; i++) {
        printf("Element %d: ", i + 1);
        scanf("%d", &arr[i]);


    }
}
int checkSetInUniversal(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        int found = 0;
```

```
        for (int j = 0; j < superSetSize; j++) {
            if (arr[i] == superSet[j]) {
                found = 1;
                break;
            }
        }

        if (!found) {
                printf("Error: Element %d is not in the Universal Set. Please enter the set
                again.\n", arr[i]);
            return 0;
        }
    }
    return 1;
}
void generateBitStrings() {
    for (int i = 0; i < superSetSize; i++) {
        bitStringA[i] = 0;
        bitStringB[i] = 0;
    }
    for (int i = 0; i < setASize; i++) {

        for (int j = 0; j < superSetSize; j++) {
            if (setA[i] == superSet[j]) {
                bitStringA[j] = 1;
                break;
            }
        }
    }
```

```
    for (int i = 0; i < setBSize; i++) {
        for (int j = 0; j < superSetSize; j++) {
            if (setB[i] == superSet[j]) {
                bitStringB[j] = 1;
                break;
            }
        }
    }
    printf("Set A Bit String: ");
    printBitString(bitStringA, superSetSize);
    printf("Set B Bit String: ");
    printBitString(bitStringB, superSetSize);
}
void setUnion() {
    int bitStringUnion[MAX_SIZE];
    for (int i = 0; i < superSetSize; i++) {
        bitStringUnion[i] = bitStringA[i] | bitStringB[i];
    }
    printf("Union: ");
    printSetFromBitString(bitStringUnion, superSetSize);
    printf("Union Bit String");
    printBitString(bitStringUnion,superSetSize);
}
void setIntersection() {
    int bitStringIntersection[MAX_SIZE];
    for (int i = 0; i < superSetSize; i++) {
        bitStringIntersection[i] = bitStringA[i] & bitStringB[i];
    }
```

```
    printf("Intersection: ");
    printSetFromBitString(bitStringIntersection, superSetSize);
    printBitString(bitStringIntersection,superSetSize);
}
void setDifferenceAminusB() {
    int bitStringDifferenceAminusB[MAX_SIZE];
    for (int i = 0; i < superSetSize; i++) {
        bitStringDifferenceAminusB[i] = bitStringA[i] & (1 - bitStringB[i]);
    }
    printf("Difference (A - B): ");
    printSetFromBitString(bitStringDifferenceAminusB, superSetSize);
    printBitString(bitStringDifferenceAminusB,superSetSize);
}
void setDifferenceBminusA() {
    int bitStringDifferenceBminusA[MAX_SIZE];
    for (int i = 0; i < superSetSize; i++) {
        bitStringDifferenceBminusA[i] = bitStringB[i] & (1 - bitStringA[i]);
    }
    printf("Difference (B - A): ");
    printSetFromBitString(bitStringDifferenceBminusA, superSetSize);
    printBitString(bitStringDifferenceBminusA,superSetSize);
}
void printBitString(int arr[], int size) {
    printf("{");
    for (int i = 0; i < size; i++) {
        printf("%d", arr[i]);
        if (i < size - 1) {
            printf(", ");
```

```
        }
    }
    printf("}\n");
}
void printSetFromBitString(int arr[], int size) {
    int first = 1;
    printf("{");
    for (int i = 0; i < size; i++) {
        if (arr[i] == 1) {
            if (!first) {
                printf(", ");
            }
            printf("%d", superSet[i]);
            first = 0;
        }
    }
    printf("}\n");
}
int main() {
    int choice;
    getUniversalSet();
    do {
        printf("Enter Set A Size (max %d): ", superSetSize);
        scanf("%d", &setASize);
        if (setASize > superSetSize) {
            printf("Error: Set A size cannot exceed Universal Set size.\n");
        }
    } while (setASize > superSetSize);
```

```
do {
   getSet(setA, &setASize);
} while (checkSetInUniversal(setA, setASize) == 0);
do {
   printf("Enter Set B Size (max %d): ", superSetSize);
   scanf("%d", &setBSize);
   if (setBSize > superSetSize) {
      printf("Error: Set B size cannot exceed Universal Set size.\n");
   }
} while (setBSize > superSetSize);
do {
   getSet(setB, &setBSize);

} while (checkSetInUniversal(setB, setBSize) == 0);
generateBitStrings();
do {
   printf("\nChoose an operation:\n");
   printf("1. Union of A and B\n");
   printf("2. Intersection of A and B\n");
   printf("3. Difference (A - B)\n");
   printf("4. Difference (B - A)\n");
   printf("5. Exit\n");
   printf("Enter your choice: ");
   scanf("%d", &choice);
   switch (choice) {
      case 1:
         setUnion();
         break;
```

```
            case 2:
                setIntersection();
                break;
            case 3:
                setDifferenceAminusB();
                break;
            case 4:
                setDifferenceBminusA();
                break;
            case 5:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 5);
    return 0;
}
```

RESULT

The Implementation of Set Data Structure and set operations program is executed successfully and the output is verified.

Experiment No: 7 **Date: 18/11/24**

DISJOINT SETS

AIM

Implementation of Disjoint Sets and the associated operations (create, union, find).

ALGORITHM

Global Declarations

1. Define a node structure with rep, next, and data.
2. Declare heads[50], tails[50], and countRoot.

makeSet(x)

1. Create a node with data x, set rep to itself, store in heads and tails, and increment countRoot.

find(a)

1. Search for a's representative.

unionSets(a, b)

1. Find representatives of a and b.
2. Merge sets if different.

search(x)

1. Search for element x in all sets.

displayRepresentatives()

1. Print all representatives.

displaySets()

1. Print elements of all sets.

main()

1. Input set size and show menu with options:
    - Make set.
    - Display representatives.
    - Perform union.
    - Find representative.
    - Display sets.
    - Exit.

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;
    countRoot++;
}
```

```
struct node* find(int a) {
   int i;
   struct node *tmp;
   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, j, pos, flag = 0;
   struct node *tail2;
   struct node *rep1 = find(a);
   struct node *rep2 = find(b);
   if (rep1 == NULL || rep2 == NULL) {
      printf("\nElement(s) 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];
                }
                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;
    for (i = 0; i < countRoot; i++) {
        tmp = heads[i];
        while (tmp != NULL) {
            if (tmp->data == x)
```

```
            return 1;
          tmp = tmp->next;
        }
    }
    return 0;
}
void displayRepresentatives() {
    printf("\nSet Representatives: ");
    for (int i = 0; i < countRoot; i++) {
        printf("%d ", heads[i]->data);
    }
    printf("\n");
}
void displaySets() {
    int i, j;
    struct node *temp;
    printf("\nDisjoint Sets:\n");
    for (i = 0; i < countRoot; i++) {
        temp = heads[i];
        printf("{ ");
        int first = 1;
        while (temp != NULL) {
            if (!first) printf(", ");
            printf("%d", temp->data);
            first = 0;
            temp = temp->next;
        }
        printf(" }\n");
```

```
    }
}
int main() {
   int choice, x, y, setSize,temp=0;
do {
       printf("\n1. Make Set");
       printf("\n2. Display set representatives");
       printf("\n3. Union");
       printf("\n4. Find Set");
       printf("\n5. Display all sets");
       printf("\n6. Exit");
       printf("\nEnter your choice: ");
       scanf("%d", &choice);
       switch(choice) {
          case 1:
                printf("Enter the Element to Make a Set: ");
                scanf("%d",&x);
                if(search(x)){
                        printf("\nElement %d is already Exist In the Set, Enter the Unique Element.\n",x);
                }
                else{
                        makeSet(x);
                }
                break;
          case 2:
             displayRepresentatives();
             break;
          case 3:
```

```
            printf("\nEnter first element: ");
            scanf("%d", &x);
            printf("Enter second element: ");
            scanf("%d", &y);
            unionSets(x, y);
            break;
        case 4:
            printf("\nEnter the element to find: ");
            scanf("%d", &x);
            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:
            displaySets();
            break;
        case 6:
            printf("\nExiting program...\n");
            exit(0);
        default:
            printf("\nInvalid choice! Please try again.\n");
            break;
    }
} while (1);
return 0;
```

```
}
```

RESULT

The Implementation of Disjoint Sets and the associated operations program is executed successfully and the output is verified.

Experiment No: 8 **Date: 09/12/24**

DFS AND BFS

AIM

Implementation of Graph Traversal techniques (DFS and BFS) and Topological Sorting.

ALGORITHM

Global Declarations

1. Define Node with int vertex and Node* next.
2. Define Graph with int numVertices and Node* adjList[MAX_VERTICES].

createNode(vertex)

1. Allocate and return node with vertex.

initGraph(graph, vertices)

1. Set numVertices and initialize adjacency list.

addEdge(graph, src, dest)

1. Insert node for dest in src's list.

DFS(graph, startVertex)

1. Initialize visited array, print "DFS", and call DFSUtil.

DFSUtil(graph, vertex, visited)

1. Mark vertex visited, print it, and visit neighbors.

BFS(graph, startVertex)

1. Initialize visited and queue, print "BFS", and traverse.

topologicalSort(graph)

1. Initialize visited and stack, recursively sort, and print stack.

topologicalSortUtil(graph, vertex, visited, stack)

1. Mark vertex visited, visit neighbors, push vertex to stack.

displayGraph(graph)

1. Print adjacency list.

main()

1. Initialize graph, input vertices/edges, and show menu.

Output

1. Display graph, DFS/BFS, and topological order (if DAG).

SOURCE CODE

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX_VERTICES 10
struct Node {
   int vertex;
   struct Node* next;
};
struct Graph {
   int numVertices;
   struct Node* adjList[MAX_VERTICES];
};
struct Node* createNode(int vertex) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->vertex = vertex;
   newNode->next = NULL;
   return newNode;
```

```
}
void initGraph(struct Graph* graph, int vertices) {
    graph->numVertices = vertices;
    for (int i = 0; i < vertices; i++) {
        graph->adjList[i] = NULL;
    }
}
void addEdge(struct Graph* graph, int src, int dest) {
    struct Node* newNode = createNode(dest);
    newNode->next = graph->adjList[src];
    graph->adjList[src] = newNode;
}
void DFSUtil(struct Graph* graph, int vertex, bool visited[]) {
    visited[vertex] = true;
    printf("%d ", vertex);
    struct Node* adjList = graph->adjList[vertex];
    while (adjList != NULL) {
        int adjVertex = adjList->vertex;
        if (!visited[adjVertex]) {
            DFSUtil(graph, adjVertex, visited);
        }
        adjList = adjList->next;
    }
}
void DFS(struct Graph* graph, int startVertex) {
    bool visited[MAX_VERTICES] = {false};
    printf("DFS starting from vertex %d: ", startVertex);
    DFSUtil(graph, startVertex, visited);
```

```
    for (int i = 0; i < graph->numVertices; i++) {
        if (!visited[i]) {
            DFSUtil(graph, i, visited);
        }
    }
    printf("\n");
}
void BFS(struct Graph* graph, int startVertex) {
    bool visited[MAX_VERTICES] = {false};
    int queue[MAX_VERTICES];
    int front = 0, rear = 0;
    visited[startVertex] = true;
    queue[rear++] = startVertex;
    printf("BFS starting from vertex %d: ", startVertex);
    while (front < rear) {
        int currentVertex = queue[front++];
        printf("%d ", currentVertex);
        struct Node* adjList = graph->adjList[currentVertex];
        while (adjList != NULL) {
            int adjVertex = adjList->vertex;
            if (!visited[adjVertex]) {
                visited[adjVertex] = true;
                queue[rear++] = adjVertex;
            }
            adjList = adjList->next;
        }
    }
    printf("\n");
```

```
}
void topologicalSortUtil(struct Graph* graph, int vertex, bool visited[], int stack[], int* stackIndex) {
    visited[vertex] = true;
    struct Node* adjList = graph->adjList[vertex];
    while (adjList != NULL) {
        int adjVertex = adjList->vertex;
        if (!visited[adjVertex]) {
            topologicalSortUtil(graph, adjVertex, visited, stack, stackIndex);
        }
        adjList = adjList->next;
    }
    stack[(*stackIndex)++] = vertex;
}
void topologicalSort(struct Graph* graph) {
    bool visited[MAX_VERTICES] = {false};
    int stack[MAX_VERTICES];
    int stackIndex = 0;
    for (int i = 0; i < graph->numVertices; i++) {
        if (!visited[i]) {
            topologicalSortUtil(graph, i, visited, stack, &stackIndex);
        }
    }
    printf("Topological Sort: ");
    for (int i = stackIndex - 1; i >= 0; i--) {
        printf("%d ", stack[i]);
    }
    printf("\n");
}
```

```
void displayGraph(struct Graph* graph) {
    printf("\nGraph Representation (Adjacency List):\n");
    for (int i = 0; i < graph->numVertices; i++) {
        struct Node* adjList = graph->adjList[i];
        printf("Vertex %d: ", i);
        while (adjList != NULL) {
            printf("%d -> ", adjList->vertex);
            adjList = adjList->next;
        }
        printf("NULL\n");
    }
}
int main() {
    struct Graph graph;
    int vertices, edges, src, dest, startVertex, choice;
    printf("*** BFS, DFS, and Topological Sort Implementation ***\n");
    printf("Enter the number of vertices: ");
    scanf("%d", &vertices);
    initGraph(&graph, vertices);
    printf("Enter the number of edges: ");
    scanf("%d", &edges);
    for (int i = 0; i < edges; i++) {
        printf("Enter edge %d (source destination): ", i + 1);
        scanf("%d %d", &src, &dest);
        addEdge(&graph, src, dest);
    }
    do {
        printf("\nMenu:\n");
```

```
printf("1. Display Graph\n");
printf("2. Perform DFS Traversal\n");
printf("3. Perform BFS Traversal\n");
printf("4. Perform Topological Sort\n");
printf("5. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);

switch (choice) {
  case 1:
    displayGraph(&graph);
    break;
  case 2:
    printf("Enter start vertex for DFS: ");
    scanf("%d", &startVertex);
    DFS(&graph, startVertex);
    break;
  case 3:
    printf("Enter start vertex for BFS: ");
    scanf("%d", &startVertex);
    BFS(&graph, startVertex);
    break;
  case 4:
    topologicalSort(&graph);
    break;
  case 5:
    printf("Exiting program.\n");
    break;
```

```
        default:
            printf("Invalid choice! Try again.\n");

        }
    } while (choice != 5);

    return 0;
}
```

RESULT

The mplementation of Graph Traversal techniques (DFS and BFS) and Topological Sorting program is executed successfully and the output is verified.

Experiment No: 9 **Date: 19/12/24**

STRONGLY CONNECTED COMPONENTS

AIM

Implementation of Finding the Strongly connected Components in a directed graph.

ALGORITHM

Global Declarations

1. Define stack[MAX_SIZE], top, and Graph structure with V, visited[], and adjacency list arrays.

new_adj_list_node(dest)

1. Create and return a new adjacency node with dest as the destination.

create_graph(V)

1. Allocate and return a graph with V vertices and initialized adjacency lists.

get_transpose(gr, src, dest)

1. Create reverse edge (dest -> src) in transposed graph.

add_edge(graph, gr, src, dest)

1. Add edge (src -> dest) to original and (dest -> src) to transposed graph.

print_graph(graph)

1. Print each vertex and its adjacency list.

push(x)

1. Push x to the stack if not full.

pop()

1. Pop from stack if not empty.

set_fill_order(graph, v, visited[], stack)

1. Perform DFS and push vertex v to stack after visiting neighbors.

dfs_recursive(gr, v, visited[])

1. Perform DFS, mark visited vertices, and print.

strongly_connected_components(graph, gr, V)

1. Call set_fill_order() for all vertices.
2. For each unvisited vertex in stack, perform DFS on transposed graph and print SCC.

main()

1. Input the number of vertices and edges.
2. Create graphs and add edges.
3. Find and print SCCs.

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;
struct adj_list_node {
   int dest;
   struct adj_list_node *next;
};
struct adj_list {
   struct adj_list_node *head;
};
struct Graph {
   int V;
```

```
    int *visited;
    struct adj_list *array;
};
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->next = NULL;
    return newNode;
}
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;
}
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;
}
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);
```

```
}
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 overflow");
    } else {
        top++;
        stack[top] = x;
    }
}
void pop() {
    if (top <= -1) {
        printf("\n\t Stack is underflow");
    } else {
        top--;
    }
}
void set_fill_order(struct Graph *graph, int v, bool visited[], int *stack) {
    visited[v] = true;
```

```
    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);
}
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;
}
```

RESULT

The Implementation of Finding the Strongly connected Components in a directed graph program is executed successfully and the output is verified.

Experiment No: 10 **Date: 26/12/24**

PRIM’S ALGORITHM

AIM

Implementation of Prim’s Algorithm for finding the minimum cost spanning tree.

ALGORITHM

Global Declarations

1. Define integers n, i, j, u, v, a, b, cost[10][10], visited[10], min, mincost, and ne.

main()

1. GET the number of nodes n and the adjacency matrix.
2. Set cost[i][j] = 999 if cost[i][j] is 0 (no edge).
3. Mark the first node as visited (visited[1] = 1).
4. While ne < n:
    - Initialize min = 999.
    - Loop through the matrix to find the minimum edge (min).
    - If either u or v is not visited, print the edge (a, b), add min to mincost, and mark b as visited.
    - Set cost[a][b] = cost[b][a] = 999 to avoid re-selection.
5. Output the total MST cost (mincost).

SOURCE CODE

```
include <stdio.h>
int n, i, j, u, v, a, b;
int cost[10][10], visited[10]= {0}, min, mincost= 0, ne= 1;
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 && 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("\n\nMinimum cost: %d\n", mincost);
}
```

RESULT

The Implementation of Prim's Algorithm for finding the minimum cost spanning tree program is executed successfully and the output is verified.

Experiment No: 11 **Date: 26/12/24**

KRUSKAL'S ALGORITHM

AIM

Implementation of Kruskal's algorithm using the Disjoint set data structure.

ALGORITHM

Global Declarations

1. Define parent[10], n, and cost[10][10].

find(i)

1. Loop to find the root of node i:
   - While parent[i] != i, set i = parent[i].
   - Return i.

union_set(i, j)

1. Find the root parents of i and j using find(i) and find(j).
2. Set parent[a] = b to merge the sets.

main()

1. GET number of nodes n and the adjacency matrix, replacing 0 with 999 (no edge).
2. Initialize parent[i] = i.
3. Print "Edges in the Minimum Spanning Tree".
4. Loop until ne == n - 1:
   - Initialize min = 999.
   - Find the smallest edge (a, b) with cost min.
   - If find(u) != find(v), print the edge, add min to mincost,union_set(u, v).
   - Mark the edge as processed by setting cost[a][b] = 999.
5. Output the MST total cost (mincost).

SOURCE CODE

```
#include <stdio.h>
int parent[10], n, cost[10][10];
// Function to find the parent of a node
int find(int i) {
   while (parent[i] != i)
      i = parent[i];
   return i;
}
// Function to perform union of two sets
int union_set(int i, int j) {
   int a = find(i);
   int b = find(j);
   parent[a] = b;
   return 0;
}
void main() {
   int i, j, a, b, u, v, ne = 1, min, mincost = 0;
   printf("Enter the number of nodes: ");
   scanf("%d", &n);
   printf("Enter the adjacency matrix:\n");
   for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) {
         scanf("%d", &cost[i][j]);
         if (cost[i][j] == 0)
            cost[i][j] = 999; // Replace 0 with infinity (999) if no edge exists
      }
```

```
    }
    for (i = 0; i < n; i++)
        parent[i] = i;
    printf("\nEdges in the Minimum Spanning Tree:\n");
    while (ne < n) {
        for (i = 0, min = 999; i < n; i++) {
            for (j = 0; 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 adding this edge does not form a cycle
        if (u != v) {
            printf("Edge %d: (%d, %d) cost: %d\n", ne++, a, b, min);
            mincost += min;
            union_set(u, v);
        }
        // Mark the edge as processed
        cost[a][b] = cost[b][a] = 999;
    }
    printf("\nMinimum cost: %d\n", mincost);
}
```

RESULT

The Implementation of Kruskal’s algorithm using the Disjoint set data structure. program is executed successfully and the output is verified.