Experiment No:1 **Date:23/10/2024**

SINGLY LINKED STACK

AIM

To implement singly linked stack - push,pop,linear search.

ALGORITHM

Initialize the Stack (Global)

1. Create a pointer top and initialize it to NULL, which represents an empty stack

isEmpty()

1. If top == NULL, return 1 (True) (stack is empty).
2. Else, return 0 (False) (stack is not empty).

push(data)

1. Allocate memory for a new node (newnode).
2. If memory allocation fails, print an error message and exit.
3. Set the new node's data field to the provided element (data).
4. Set the new node's next field to the current top (linking the new node to the previous top).
5. Update top to point to the new node (making the new node the top of the stack).
6. Call print() to display the current stack.

pop()

1. If the stack is empty (check isEmpty()), print "Stack Underflow" and exit.
2. Store the current top node in a temporary pointer (temp).
3. Retrieve the data from the top node and store it in val.
4. Update top to point to the next node (top = temp->next).
5. Free the memory occupied by the popped node (free(temp)).
6. Return the value stored in val.

print()

1. If the stack is empty (check isEmpty()), print "Stack is empty" and exit.
2. Set a pointer temp to point to the top node.
3. Traverse the stack by following the next pointers of each node, printing the data value of each node.
4. Stop when temp is NULL (end of the stack).
5. Print a newline after all elements are displayed.

search()

1. Prompt the user to input the element to search for (s).
2. Set a pointer temp to point to top.
3. Traverse the stack:
    1. For each node, check if the data matches s.
    2. If a match is found, print "Element s found" and exit the function.
4. If the entire stack is traversed and no match is found, print "Element s not found".

main()

1. Display a menu with options:
    1. Push
    2. Pop
    3. Display
    4. Search
    5. Exit
2. Continuously prompt the user to input a choice until they choose to exit:
    1. For choice 1 (Push): Prompt the user for the element and call push(data).
    2. For choice 2 (Pop): Call pop() and display the popped element.
    3. For choice 3 (Display): Call print() to show the stack.
    4. For choice 4 (Search): Call search() to find an element in the stack.
    5. For choice 5 (Exit): Print "Exiting..." and terminate the program.
    6. If the user enters an invalid choice, print "Invalid choice".

SOURCE CODE

```
#include<stdio.h>
#include<stdlib.h>
struct node {
   int data;
   struct node *next;
} *top = NULL;
int isEmpty() {
   return top == NULL;
}
void push(int data) {
   struct node *newnode = (struct node*) malloc(sizeof(struct node));
   if (newnode == NULL) {
      printf("Stack Overflow\n");
      exit(1);
   }
   newnode->data = data;
   newnode->next = top;
   top = newnode;
   print();
}
int pop() {
   if (isEmpty()) {
      printf("Stack Underflow\n");
      exit(1);
   }
   struct node *temp = top;
   int val = temp->data;
   top = temp->next;
   free(temp);
```

```
    return val;
}
void print() {
    if (isEmpty()) {
        printf("Stack is empty.\n");
        return;
    }
    struct node *temp = top;
    printf("Stack elements are: ");
    while (temp != NULL) {
        printf("%d\t", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
void search() {
    struct node *temp = top;
    int s, f = 0;
    printf("Enter an element to be searched: ");
    scanf("%d", &s);
    while (temp != NULL) {
        if (temp->data == s) {
            f = 1;
            printf("Element %d found\n", s);
            break;
        }
        temp = temp->next;
    }
    if (f == 0) {
        printf("Element %d not found\n", s);
    }
}
int main() {
    int ch, data;
    while (1) {
        printf("\n1. Push\n2. Pop\n3. Display\n4. Search\n5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &ch);
        switch (ch) {
            case 1:
                printf("Enter the element to be pushed: ");
                scanf("%d", &data);
                push(data);
                break;
            case 2:
```

```
            data = pop();
            printf("Deleted element is %d\n", data);
            print();
            break;
        case 3:
            print();
            break;
        case 4:
            search();
            break;
        case 5:
            printf("Exiting...\n");
            exit(0);
        default:
            printf("Invalid choice\n");
      }
   }
   return 0;
}
```

OUTPUT

```
 1.Push
 2.Pop
 3.Display
 4.Search
 5.Exit
 Enter your choice:1

Enter the element to be pushed:1
Stack elements are:1

 1.Push
 2.Pop
 3.Display
 4.Search
 5.Exit
 Enter your choice:1

Enter the element to be pushed:2
Stack elements are:2    1

 1.Push
 2.Pop
 3.Display
 4.Search
 5.Exit
```

```
 Enter your choice:1

Enter the element to be pushed:3
Stack elements are:3    2   1

 1.Push
 2.Pop
 3.Display
 4.Search
 5.Exit
 Enter your choice:2

 Deleted element is 3
Stack elements are:2    1

 1.Push
 2.Pop
 3.Display
 4.Search
 5.Exit
 Enter your choice:3
Stack elements are:2    1
```

```
 1.Push
 2.Pop
 3.Display
 4.Search
 5.Exit
 Enter your choice:1

Enter the element to be pushed:6
Stack elements are:6    2   1

 1.Push
 2.Pop
 3.Display
 4.Search
 5.Exit
 Enter your choice:4

 Enter an element to be search:8

 element 8 is not found

 1.Push
 2.Pop
 3.Display
 4.Search
```

```
4.Search
5.Exit
Enter your choice:4

Enter an element to be search:6

element 6 found

1.Push
2.Pop
3.Display
4.Search
5.Exit
Enter your choice:5

Exit
```

RESULT

The program to implement singly linked stack is successfully executed and the output is verified.

Experiment No:2 **Date:23/10/2024**

SINGLY LINKED LIST

AIM

To implement singly linked list – insertion,deletion.

ALGORITHM

insert_begin()

1. Create a new node and allocate memory.
2. Read the input value (data).
3. Set the data of the new node to the input value.
4. Point the next of the new node to the current head of the list.
5. Update the head to point to the new node.

insert_anywhere()

1. Read the input value (data) and the position (p).
2. If head is NULL, the list is empty. If position is 1, call insert_begin() to insert at the beginning.
3. Traverse the list until the desired position p is found.
4. Create a new node and set its data to data.
5. Update the next of the new node to point to the current next of the node at position p-1.
6. Update the next of the node at position p-1 to point to the new node.

insert_end()

1. Create a new node and allocate memory.
2. Read the input value (data).
3. If the list is empty (i.e., head is NULL), set head to the new node.
4. Otherwise, traverse the list to find the last node (where next is NULL).
5. Set the next of the last node to point to the new node.
6. Set the next of the new node to NULL (since it's the last node).

search()

1. Read the input element (s) to be searched.
2. Start from head and traverse the list.
3. For each node, compare the data with s.
4. If the element is found, print "Element found".
5. If the list is exhausted and the element isn't found, print "Element not found".

delete_begin()

1. If head is NULL (list is empty), print "List is empty".
2. Otherwise, set the head to the second node (head = head->next).
3. Free the memory of the first node.

delete_end()

1. If head is NULL (list is empty), print "List is empty".

2. If the list has only one node, set head to NULL and free the node.
3. Otherwise, traverse the list to find the second-to-last node (where next->next is NULL).
4. Set next of the second-to-last node to NULL to remove the last node.
5. Free the memory of the last node.

delete_anywhere()

1. If head is NULL (list is empty), print "List is empty".
2. If the position p is 1, call delete_begin() to delete the first node.
3. Otherwise, traverse the list to the node at position p-1.
4. Set next of the node at position p-1 to next of the node at position p (bypassing the node at position p).
5. Free the memory of the node at position p.

display()

1. Start from head and traverse the list.
2. For each node, print the data of the node.
3. If the list is empty (head is NULL), print "Linked list is empty".

SOURCE CODE

```
#include<stdio.h>
#include<stdlib.h>
struct node{
	int data;
	struct node *next;
};
struct node *head=NULL;
void display(){
	struct node *temp=head;
	if(temp==NULL){
		printf("\n Linked list is empty");
	}
	else{
		printf("\n Elements are:");
		while(temp!=NULL)
		{
			printf("%d \t",temp->data);
			temp=temp->next;
		}
	}
}
void insert_begin(){
	int data;
	struct node *newnode=(struct node*)malloc(sizeof(struct node));
```

```
        printf("\n Enter the value to be inserted:");
        scanf("%d",&data);
        newnode->data=data;
        newnode->next=head;
        head=newnode;
        display();
}
void insert_anywhere(){
        int data,p,count=2;
        struct node *newnode=(struct node*)malloc(sizeof(struct node));
        struct node *temp=head;
        printf("\n Enter the data to be inserted:");
        scanf("%d",&data);
        printf("\n Enter the position to be inserted:");
        scanf("%d",&p);
        if(head==NULL)
                printf("List is empty");
        else if(p==1)
                insert_begin();
        else{
                temp=head;
                while(temp!=NULL){
                   if(p==count){
                      newnode->data=data;
                      newnode->next=temp->next;
                      temp->next=newnode;
                      break;
                   }
                   else{
                      temp=temp->next;
                      count++;
                   }
                }
                display();
        }
}
void insert_end(){
        int data;
        struct node *newnode=(struct node*)malloc(sizeof(struct node));
        printf("\n Enter the value to be inserted:");
        scanf("%d",&data);

        if(head==NULL)
head=newnode;
        else{
```

```
            struct node *temp=head;
            while(temp->next!=NULL){
                  temp=temp->next;
            }
            newnode->data=data;
            temp->next=newnode;
            newnode->next=NULL;
      }
      display();
}
void search(){
      int s,f=0;
      printf("\n Enter value to be search:");
      scanf("%d",&s);
      struct node *temp=head;
      if(head==NULL)
            printf("\n List is empty");
      else{
            while(temp!=NULL){
                  if(temp->data==s){
                        f=1;
                        printf("\n Element found");
                  }
                  temp=temp->next;
            }
            if(f!=1)
                  printf("\n Element not found");
}
}
void delete_begin(){
      struct node *temp=head;
      if(temp==NULL)
            printf("\n List is empty");
      else{
            head=temp->next;
            free(temp);
            display();
      }
}
void delete_end(){
      struct node *ptr=head;
      struct node *temp=ptr;
      if(ptr==NULL)
            printf("\n List is empty");
      else{
```

```
            ptr=head;
            while(ptr->next!=NULL){
                  temp=ptr;
                  ptr=ptr->next;
            }
            temp->next=NULL;
            free(ptr);
            display();
      }
}
void delete_anywhere(){
      int p;
      if(head==NULL)
            printf("\n List is empty");
      else{
            printf("\n Enter the position to be deleted:");
            scanf("%d",&p);
            struct node *temp=head;
            struct node *prev=NULL;
            if(p<1)
                  printf("\n Invalid");
            else if(p==1)
                  delete_begin();
            else{
                  for(int i=1;i<p&&temp!=NULL;i++){
                        prev=temp;
                        temp=temp->next;
                        if(temp==NULL)
                              printf("\n Position not found");
                        prev->next=temp->next;
                        free(temp);
                  }
                  display();
            }
      }
}
void main(){
      int ch;
      do{
            printf("\n Menu \n 1.Insert at beginning \n 2.Insert at any position \n
                3.Insert at end \n 4.Search \n 5.Delete from beginning \n 6.delete end \n
               7.delete anyehere \n 8.display \n 9.exit");
            printf("\n Enter your choice:");
            scanf("%d",&ch);
            switch(ch){
```

```
                case 1: insert_begin();
                        break;
                case 2: insert_anywhere();
                        break;
                case 3: insert_end();
                        break;
                case 4: search();
                        break;
                case 5: delete_begin();
                        break;
                case 6: delete_end();
                        break;
                case 7: delete_anywhere();
                        break;
                case 8: display();
                        break;
                case 9: printf("\n Exit...");
                        exit(0);
                default: printf("\n Invalid Entry..");

            }
        }while(ch!=9);
}
```

OUTPUT

```
Menu
1.Insert at beginning
2.Insert at any position
3.Insert at end
4.Search
5.Delete from beginning
6.delete end
7.delete anyehere
8.display
9.exit
Enter your choice:1

Enter the value to be inserted:1

Elements are:1
Menu
1.Insert at beginning
2.Insert at any position
3.Insert at end
4.Search
5.Delete from beginning
6.delete end
7.delete anyehere
8.display
9.exit
```

```
Enter your choice:1

Enter the value to be inserted:2

Elements are:2     1
Menu
1.Insert at beginning
2.Insert at any position
3.Insert at end
4.Search
5.Delete from beginning
6.delete end
7.delete anyehere
8.display
9.exit
Enter your choice:2

Enter the data to be inserted:5

Enter the position to be inserted:2

Elements are:2     5   1
Menu
1.Insert at beginning
2.Insert at any position
3.Insert at end
4.Search
5.Delete from beginning
6.delete end
7.delete anyehere
8.display
9.exit
Enter your choice:3

Enter the value to be inserted:6

Elements are:2     5   1   6
Menu
1.Insert at beginning
2.Insert at any position
3.Insert at end
4.Search
5.Delete from beginning
6.delete end
7.delete anyehere
8.display
9.exit
Enter your choice:4

Enter value to be search:7
```

```
Element not found
Menu
1.Insert at beginning
2.Insert at any position
3.Insert at end
4.Search
5.Delete from beginning
6.delete end
7.delete anyehere
8.display
9.exit
Enter your choice:4

Enter value to be search:5

Element found
Menu
1.Insert at beginning
2.Insert at any position
3.Insert at end
4.Search
5.Delete from beginning
6.delete end
7.delete anyehere
8.display
9.exit
Enter your choice:8

Elements are:2     5   1   6
Menu
1.Insert at beginning
2.Insert at any position
3.Insert at end
4.Search
5.Delete from beginning
6.delete end
7.delete anyehere
8.display
9.exit
Enter your choice:5

Elements are:5     1   6
Menu
1.Insert at beginning
2.Insert at any position
3.Insert at end
4.Search
5.Delete from beginning
6.delete end
7.delete anyehere
8.display
```

```
9.exit
Enter your choice:6

Elements are:5     1
Menu
1.Insert at beginning
2.Insert at any position
3.Insert at end
4.Search
5.Delete from beginning
6.delete end
7.delete anyehere
8.display
9.exit
Enter your choice:7

Enter the position to be deleted:2

Elements are:5
Menu
1.Insert at beginning
2.Insert at any position
3.Insert at end
4.Search
5.Delete from beginning
6.delete end
7.delete anyehere
8.display
9.exit
Enter your choice:9

Exit...
```

RESULT

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

Experiment No:3 **Date:30/10/2024**

DOUBLY LINKED LIST

AIM

To implement doubly linked list - insertion,deletion,search.

ALGORITHM

insert_begin()

1. Create a new node and allocate memory.
2. Set the data of the new node to the input value.
3. Set the next pointer of the new node to point to the current head.
4. Set the prev pointer of the new node to NULL (since it's the first node).
5. If the list is not empty (i.e., head is not NULL), set the prev pointer of the old head to the new node.
6. Update head to point to the new node.

insert_anywhere()

1. Read the position (p) where the new node needs to be inserted and check if the list is empty.
2. If the position is 1, call insert_begin() to insert at the beginning.
3. Otherwise, traverse the list until the desired position (p) is reached.
4. Create a new node and set its data to the input value.
5. Set the next pointer of the new node to point to the node at position p, and update the prev pointer of the node at position p to point to the new node.
6. Set the next pointer of the previous node to the new node, and set the prev pointer of the node after the new node to the new node.

insert_end()

1. Create a new node and allocate memory.
2. Set the data of the new node to the input value.
3. Set the next pointer of the new node to NULL (since it will be the last node).
4. If the list is empty (i.e., head is NULL), set head to point to the new node.
5. Otherwise, traverse the list to the last node (where next is NULL).
6. Set the next pointer of the last node to point to the new node, and set the prev pointer of the new node to point to the last node.

delete_begin()

1. Check if the list is empty (i.e., head is NULL).
2. If the list is not empty, set head to the second node (head = head->next).
3. Set the prev pointer of the new head node to NULL (since it's now the first node).
4. Free the memory of the old first node.

delete_anywhere()

1. Read the position (p) where the node needs to be deleted.
2. If the list is empty (i.e., head is NULL), print an error message.
3. If the position is 1, call delete_begin() to delete the first node.

4. Otherwise, traverse the list to find the node at position p.
5. Set the next pointer of the previous node to point to the next node of the node to be deleted.
6. Set the prev pointer of the next node to point to the previous node.
7. Free the memory of the node to be deleted.

delete_end()

1. If the list is empty (i.e., head is NULL), print an error message.
2. If the list contains only one node, set head to NULL and free the node.
3. Otherwise, traverse the list to the last node (where next is NULL).
4. Set the next pointer of the second-to-last node to NULL.
5. Free the memory of the last node.

search()

1. Read the input element (s) to be searched.
2. Traverse the list from head and compare each node's data with the search element.
3. If the element is found, print "Element found".
4. If the element is not found by the end of the list, print "Element not found".

display()

1. Traverse the list starting from head.
2. Print the data of each node.
3. If the list is empty (i.e., head is NULL), print "Linked list is empty".

SOURCE CODE

```
#include<stdio.h>
#include<stdlib.h>
struct node{
	int data;
	struct node *next;
struct node *prev;
struct node *lock;
};
struct node *head=NULL;
void display(){
	struct node *temp=head;
	if(temp==NULL){
		printf("\n Linked list is empty");
	}
	else{
		printf("\n Elements are:");
		while(temp!=NULL)
		{
			printf("%d \t",temp->data);
			temp=temp->next;
```

```
            }
        }
}
void search(){
        int s,f=0;
        printf("\n Enter value to be search:");
        scanf("%d",&s);
        struct node *temp=head;
        if(head==NULL)
                printf("\n List is empty");
        else{
                while(temp!=NULL){
                        if(temp->data==s){
                                f=1;
                                printf("\n Element found");
                        }
                        temp=temp->next;
                }
                if(f!=1)
                        printf("\n Element not found");
        }
}
void insert_begin() {
int data;
struct node *newnode=(struct node*)malloc(sizeof(struct node));
printf("\n Enter the value to be inserted:");
        scanf("%d",&data);
newnode->data = data;
newnode->next = head;
        newnode->prev = NULL;
if (head!= NULL)
        head->prev = newnode;
        head = newnode;
display();
}
void insert_anywhere() {
struct node *prev=head;
int data,p,count=2;
struct node *temp;
struct node *newnode=(struct node*)malloc(sizeof(struct node));
        printf("\n Enter the position to be inserted:");
        scanf("%d",&p);
        if(head==NULL)
                printf("List is empty");
else if(p==1)
```

```
            insert_begin();
        else{
        printf("\n Enter the data to be inserted:");
        scanf("%d",&data);
            temp=head;
            while(temp!=NULL){
               if(p==count){
                  newnode->data=data;
                  newnode->next=temp->next;
                  temp->next=newnode;
                  break;
               }
               else{
                  temp=temp->next;
                  count++;
               }
            }
        display();
    }
}
void insert_end() {
int data;
        struct node *newnode=(struct node*)malloc(sizeof(struct node));
        printf("\n Enter the value to be inserted:");
        scanf("%d",&data);
        newnode->data = data;
        newnode->next = NULL;
        struct node* temp = head;
        if (head == NULL) {
                newnode->prev = NULL;
                head = newnode;
                return;
        }
        while (temp->next != NULL){
                temp = temp->next;
                temp->next = newnode;
                newnode->prev = temp;
        }
        display();
}
void delete_begin(){
struct node *temp=head;
if (head == NULL)
        printf("\n list is empty");
else{
```

```
            head=temp->next;
            free(temp);
            display();
        }
}
void delete_anywhere(){
int p;
        if(head==NULL)
            printf("\n List is empty");
        else{
            printf("\n Enter the position to be deleted:");
            scanf("%d",&p);
            struct node *temp=head;
            struct node *lock=NULL;
            if(p<1)
                printf("\n Invalid");
            else if(p==1)
                delete_begin();
            else{
               for(int i=1;i<p&&temp!=NULL;i++){
                        lock=temp;
                        temp=temp->next;
                        if(temp==NULL)
                            printf("\n Position not found");
                        lock->next=temp->next;
                        temp->lock=lock->next;
                        free(temp);
                    }
                    display();
            }
        }
}
void delete_end(){
struct node *ptr=head;
        struct node *temp=ptr;
        if(ptr==NULL)
            printf("\n List is empty");
        else if(ptr->next==NULL){
                head=NULL;
                free(ptr);
                printf("\n List is empty");
        }
        else{
            ptr=head;
            while(ptr->next!=NULL){
```

```
                temp=ptr;
                ptr=ptr->next;
            }
            temp->next=NULL;
            free(ptr);
            display();
        }
}
void main()
{
int ch;
        do{
            printf("\n Menu \n 1.Insert at beginning \n 2.Insert at any position \n
               3.Insert at end \n 4.Search \n 5.Delete from beginning \n
               6.delete end \n 7.delete anyehere \n 8.display \n 9.exit");
            printf("\n Enter your choice:");
            scanf("%d",&ch);
            switch(ch){
                case 1: insert_begin();
                        break;
                case 2: insert_anywhere();
                        break;
                case 3: insert_end();
                        break;
                case 4: search();
                        break;
                case 5: delete_begin();
                        break;
                case 6: delete_end();
                        break;
                case 7: delete_anywhere();
                        break;
                case 8: display();
                        break;
                case 9: printf("\n Exit...");
                        exit(0);
                default: printf("\n Invalid Entry..");

            }
        }while(ch!=9);
}
```

OUTPUT

```
Menu
1.Insert at beginning
2.Insert at any position
3.Insert at end
4.Search
5.Delete from beginning
6.delete end
7.delete anyehere
8.display
9.exit
Enter your choice:1

Enter the value to be inserted:1

Elements are:1
Menu
1.Insert at beginning
2.Insert at any position
3.Insert at end
4.Search
5.Delete from beginning
6.delete end
7.delete anyehere
8.display
9.exit
```

```
Enter your choice:1

Enter the value to be inserted:2

Elements are:2     1
Menu
1.Insert at beginning
2.Insert at any position
3.Insert at end
4.Search
5.Delete from beginning
6.delete end
7.delete anyehere
8.display
9.exit
Enter your choice:2

Enter the position to be inserted:1

Enter the value to be inserted:0

Elements are:0     2   1
Menu
1.Insert at beginning
2.Insert at any position
```

```
3.Insert at end
4.Search
5.Delete from beginning
6.delete end
7.delete anyehere
8.display
9.exit
Enter your choice:2

Enter the position to be inserted:2

Enter the data to be inserted:9

Elements are:0     9   2   1
Menu
1.Insert at beginning
2.Insert at any position
3.Insert at end
4.Search
5.Delete from beginning
6.delete end
7.delete anyehere
8.display
9.exit
Enter your choice:3
```

```
Enter the value to be inserted:7

Elements are:0     9   2   1   7
Menu
1.Insert at beginning
2.Insert at any position
3.Insert at end
4.Search
5.Delete from beginning
6.delete end
7.delete anyehere
8.display
9.exit
Enter your choice:4

Enter value to be search:7

Element found
Menu
1.Insert at beginning
2.Insert at any position
3.Insert at end
4.Search
5.Delete from beginning
6.delete end
```

```
7.delete anyehere
8.display
9.exit
Enter your choice:4

Enter value to be search:6

Element not found
Menu
1.Insert at beginning
2.Insert at any position
3.Insert at end
4.Search
5.Delete from beginning
6.delete end
7.delete anyehere
8.display
9.exit
Enter your choice:5

Elements are:9     2   1   7
Menu
1.Insert at beginning
2.Insert at any position
3.Insert at end
4.Search
5.Delete from beginning
6.delete end
7.delete anyehere
8.display
9.exit
Enter your choice:6

Elements are:9     2   1
Menu
1.Insert at beginning
2.Insert at any position
3.Insert at end
4.Search
5.Delete from beginning
6.delete end
7.delete anyehere
8.display
9.exit
Enter your choice:7

Enter the position to be deleted:2

Elements are:9     1
Menu
1.Insert at beginning
```

```
2.Insert at any position
3.Insert at end
4.Search
5.Delete from beginning
6.delete end
7.delete anyehere
8.display
9.exit
Enter your choice:8

Elements are:9     1
Menu
1.Insert at beginning
2.Insert at any position
3.Insert at end
4.Search
5.Delete from beginning
6.delete end
7.delete anyehere
8.display
9.exit
Enter your choice:6

Elements are:9
Menu
1.Insert at beginning
2.Insert at any position
3.Insert at end
4.Search
5.Delete from beginning
6.delete end
7.delete anyehere
8.display
9.exit
Enter your choice:5

Linked list is empty
Menu
1.Insert at beginning
2.Insert at any position
3.Insert at end
4.Search
5.Delete from beginning
6.delete end
7.delete anyehere
8.display
9.exit
Enter your choice:9

Exit...
```

RESULT

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

Experiment No:4 **Date:30/10/2024**

BINARY SEARCH TREE

AIM

To implement binary search trees – insertion,deletion,search.

ALGORITHM

insetNode()

1. Input value (new node to insert).
2. If root is NULL, create a new node with the given value and return it.
   Else:
   - If value < root->data, recursively insert the value into the left subtree.
   - If value > root->data, recursively insert the value into the right subtree.
   - Update the root after insertion.

Delete Operation (deleteNode)

1. Input data (node to delete).
2. If root is NULL, return NULL (node not found).
3. If data < root->data, recursively delete the node from the left subtree.
4. If data > root->data, recursively delete the node from the right subtree.
5. If data = root->data (node to delete is found):
   - If the node has no children (leaf node), free the node and return NULL.
   - If the node has one child, replace the node with its child.
   - If the node has two children:
     - Find the minimum node in the right subtree.
     - Copy the data from the minimum node to the current node.
     - Recursively delete the minimum node from the right subtree.
6. Update the root after deletion.

Search Operation (search)

1. Input s (value to search).
2. While root is not NULL and root->data != s:
   - If s < root->data, move to the left subtree.
   - Else, move to the right subtree.
   - If root is NULL or root->data == s, return the node.

In-order Traversal (inOrderTraversal)

1. If root is not NULL:
   - Recursively traverse the left subtree.
   - Print the data of the current node.
   - Recursively traverse the right subtree.

Pre-order Traversal (preOrder)

1. If root is not NULL:
   - Print the data of the current node.
   - Recursively traverse the left subtree.

Recursively traverse the right subtree.

Post-order Traversal (postOrder)

1. If root is not NULL:
   Recursively traverse the left subtree.
   Recursively traverse the right subtree.
   Print the data of the current node.

SOURCE CODE

```
#include <stdio.h>
#include <stdlib.h>
struct node {
	struct node *left;
	struct node *right;
	int data;
};
struct node *root;
struct node *newnode(int value) {
	struct node *newNode = (struct node*)malloc(sizeof(struct node));
	newNode->data = value;
	newNode->left = NULL;
	newNode->right = NULL;
	return newNode;
}
struct node *insert(struct node *root, int value) {
	if (root == NULL) {
		return newnode(value);
	}
	else if (value == root->data) {
		printf("Same data can't be stored\n");
		return root;
	}
	else if (value > root->data) {
		root->right = insert(root->right, value);
	}
	else {
		root->left = insert(root->left, value);
	}
	return root;
}
void inorderTraversal(struct node *root) {
	if (root == NULL) {
		return;
	}
	else{
```

```
        inorderTraversal(root->left);
        printf("%d\t", root->data);
        inorderTraversal(root->right);}
}
void preorderTraversal(struct node *root) {
        if (root == NULL) {
        return;
        }
        printf("%d->", root->data);
        preorderTraversal(root->left);
        preorderTraversal(root->right);
}
void postorderTraversal(struct node *root) {
        if (root == NULL) {
                return;
        }
        postorderTraversal(root->left);
        postorderTraversal(root->right);
        printf("%d->", root->data);
}
struct node *searchNode(struct node *root, int key) {
        if (root == NULL) {
                printf("\nNOT FOUND\n");
                return NULL;
        }
        else if (root->data == key) {
                printf("\nFOUND\n");
                return root;
        }
        else if (root->data < key) {
                return searchNode(root->right, key);
        }
        else {
                return searchNode(root->left, key);
        }
}
struct node *minValueNode(struct node *root) {
        struct node *current = root;
        while (current && current->left != NULL) {
                current = current->left;
        }
        return current;
}
struct node *deleteNode(struct node *root, int key) {
        if (root == NULL) {
```

```
            return root;
        }
        if (key < root->data) {
            root->left = deleteNode(root->left, key);
        }
        else if (key > root->data) {
            root->right = deleteNode(root->right, key);
        }
        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;
            }
        struct node *temp = minValueNode(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
        }
return root;
}
int main() {
        int choice;
        int value, searchv, key;
        do{
            printf("\n 1) insert Node \n 2) search\n 3) inorderTraversal \n 4)
               preorderTraversal \n 5) postorderTraversal \n 6)delete \n 7) exit \n ");
            printf("choose an option : ");
            scanf("%d", &choice);
            switch (choice)
            {
            case 1:
                printf("\n enter a number:");
                scanf("%d", &value);
                root = insert(root, value);
                break;
            case 2:
                printf("\n enter a number:");
                scanf("%d", &searchv);
                searchNode(root, searchv);
                break;
```

```
            case 3:
                inorderTraversal(root);
                break;
            case 4:
                preorderTraversal(root);
                break;
            case 5:
                postorderTraversal(root);
                break;
            case 6:
                printf("\n enter a number to delete:");
                scanf("%d", &key);
                deleteNode(root, key);
                break;
            case 7:
                exit(0);
                break;
            default:
                printf("invalid option!\n");
                break;

            }
      }while(choice != 8);
return 0;
}
```

OUTPUT

```
1) insert Node
2) search
3) inorderTraversal
4) preorderTraversal
5) postorderTraversal
6)delete
7) exit
choose an option : 1

enter a number:50

1) insert Node
2) search
3) inorderTraversal
4) preorderTraversal
5) postorderTraversal
6)delete
7) exit
choose an option : 1

enter a number:45

1) insert Node
2) search
3) inorderTraversal
```

```
 4) preorderTraversal
 5) postorderTraversal
 6)delete
 7) exit
 choose an option : 1

 enter a number:67

 1) insert Node
 2) search
 3) inorderTraversal
 4) preorderTraversal
 5) postorderTraversal
 6)delete
 7) exit
 choose an option : 1

 enter a number:47

 1) insert Node
 2) search
 3) inorderTraversal
 4) preorderTraversal
 5) postorderTraversal
 6)delete
 7) exit
 choose an option : 2

 enter a number:78

NOT FOUND

 1) insert Node
 2) search
 3) inorderTraversal
 4) preorderTraversal
 5) postorderTraversal
 6)delete
 7) exit
 choose an option : 2

 enter a number:47

FOUND

 1) insert Node
 2) search
 3) inorderTraversal
 4) preorderTraversal
 5) postorderTraversal
 6)delete
```

```
 7) exit
 choose an option : 3
45  47  50  67
 1) insert Node
 2) search
 3) inorderTraversal
 4) preorderTraversal
 5) postorderTraversal
 6)delete
 7) exit
 choose an option : 4
50->45->47->67->
 1) insert Node
 2) search
 3) inorderTraversal
 4) preorderTraversal
 5) postorderTraversal
 6)delete
 7) exit
 choose an option : 5
47->45->67->50->
 1) insert Node
 2) search
 3) inorderTraversal
 4) preorderTraversal
```

```
 5) postorderTraversal
 6)delete
 7) exit
 choose an option : 6

 enter a number to delete:50

 1) insert Node
 2) search
 3) inorderTraversal
 4) preorderTraversal
 5) postorderTraversal
 6)delete
 7) exit
 choose an option : 3
45  47  67
 1) insert Node
 2) search
 3) inorderTraversal
 4) preorderTraversal
 5) postorderTraversal
 6)delete
 7) exit
 choose an option : 7
```

RESULT

The program to implement binary search trees is successfully executed and the output is verified.

Experiment No:5 **Date:30/10/2024**

CIRCULAR QUEUE

AIM

To implement circular queue - add,delete,search.

ALGORITHM

Circular Queue Creation (Initialization)

1. Input the Size of the queue.
2. Allocate memory for the queue array.
3. Initialize front and rear pointers to -1, indicating an empty queue.

Enqueue Operation (insert)

1. Input data (value to insert).
2. Check if the queue is full by evaluating (rear + 1) % size == front.
3. If full, print "Queue overflow" and return.
4. If empty (front == -1), set front to 0.
5. Update the rear pointer using modulo arithmetic: (rear + 1) % size.
6. Insert the element at queue[rear].
7. Print "Element inserted" message and the updated queue state.

Dequeue Operation (delete)

1. Check if the queue is empty (front == -1). If true, print "Queue underflow" and return.
2. Save the element at queue[front].
3. If the queue becomes empty after removal (i.e., front == rear), reset front and rear to -1.
4. Otherwise, update the front pointer using modulo arithmetic: (front + 1) % size.
5. Print the dequeued element and the updated queue state.

Display Operation

1. If the queue is empty, print "Queue is empty".
2. Otherwise, iterate through the queue from front to rear using modulo arithmetic to print all elements.

Search Operation

1. Input s (the element to search).
2. Start from front and iterate through the queue until you reach the rear.
3. Compare each element with s.
4. If the element is found, set a flag f to 1.
5. If the element is found, print "Element found"; otherwise, print "Element not found".

SOURCE CODE

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

```
int front = -1, rear = -1,*queue,size,i;
int isFull()
{
   return (rear + 1) % size == front;
}
int isEmpty()
{
   return front == -1;
}
void enqueue(int data)
{
   if (isFull()) {
      printf("Queue overflow\n");
      return;
   }
   if(front==-1){
      front=0;
      }
   rear = (rear + 1) % size;
   queue[rear] = data;
   printf("Element %d inserted\n", data);
}
int dequeue()
{
   if (isEmpty()) {
      printf("Queue underflow\n");
      return -1;
   }
   int data = queue[front];
   if (front == rear) {
      front = rear = -1;
   }
   else {

      front = (front + 1) % size;
   }
return data;
}
void display()
{
   if (isEmpty()) {
      printf("Queue is empty\n");
      return;
   }
   printf("Queue elements: ");
```

```
    int i = front;
    while (i != rear) {
        printf("%d ", queue[i]);
        i = (i + 1) % size;
    }
    printf("%d\n", queue[rear]);
}
void search()
{
    int s,f=0;
    printf("\n enter element to be search:");
    scanf("%d",&s);
    for(i=front;i!=rear;i=(i+1)%size){
        if(s==queue[i]){
            f=1;
        break;
}
    }
    if(queue[i]==s){
        f=1;
    }
    if(!f){
        printf("\n element not found");
    }
    else{
        printf("\n element found");
    }
}
int main()
{
    int data,ch;
    printf("\n Enter maxsize:");
    scanf("%d",&size);
    queue=(int*)malloc(size* sizeof(int));
    if(queue==NULL){
        printf("Memory  allocate failed");
        exit(1);
    }
    do{
          printf("\n MENU \n 1.insert \n 2.delete \n 3.display \n 4.search \n 5.exit");
          printf("\n enter your choice:");
          scanf("%d",&ch);
          switch(ch){
             case 1: printf("\n enter element to be inserted:");
                   scanf("%d",&data);
```

```
                enqueue(data);
                display();
                break;
            case 2: printf("Dequeued element: %d\n", dequeue());
                display();
                break;
            case 3: display();
                break;
            case 4: search();
                break;
            case 5: printf("\n EXIT");
                exit(0);
                break;
            default:printf("\n invalid entry...try again");
          }
       }while(ch!=5);
   return 0;
}
```

OUTPUT

```
 Enter maxsize:5

 MENU
 1.insert
 2.delete
 3.display
 4.search
 5.exit
 enter your choice:1

 enter element to be inserted:1
Element 1 inserted
Queue elements: 1

 MENU
 1.insert
 2.delete
 3.display
 4.search
 5.exit
 enter your choice:1

 enter element to be inserted:2
Element 2 inserted
Queue elements: 1 2
```

```
 MENU
 1.insert
 2.delete
 3.display
 4.search
 5.exit
 enter your choice:2
Dequeued element: 1
Queue elements: 2

 MENU
 1.insert
 2.delete
 3.display
 4.search
 5.exit
 enter your choice:1

 enter element to be inserted:3
Element 3 inserted
Queue elements: 2 3
```

```
 MENU
 1.insert
 2.delete
 3.display
 4.search
 5.exit
 enter your choice:1

 enter element to be inserted:4
Element 4 inserted
Queue elements: 2 3 4

 MENU
 1.insert
 2.delete
 3.display
 4.search
 5.exit
 enter your choice:1

 enter element to be inserted:5
Element 5 inserted
Queue elements: 2 3 4 5
```

```
 MENU
 1.insert
 2.delete
 3.display
 4.search
 5.exit
 enter your choice:1

 enter element to be inserted:6
Element 6 inserted
Queue elements: 2 3 4 5 6

 MENU
 1.insert
 2.delete
 3.display
 4.search
 5.exit
 enter your choice:1

 enter element to be inserted:7
Queue overflow
Queue elements: 2 3 4 5 6
```

```
 MENU
 1.insert
 2.delete
 3.display
 4.search
 5.exit
 enter your choice:4

 enter element to be search:6

 element found
 MENU
 1.insert
 2.delete
 3.display
 4.search
 5.exit
 enter your choice:4

 enter element to be search:9

 element not found
 MENU
 1.insert
 2.delete
```

```
 3.display
 4.search
 5.exit
 enter your choice:3
Queue elements: 2 3 4 5 6

 MENU
 1.insert
 2.delete
 3.display
 4.search
 5.exit
 enter your choice:2
Dequeued element: 2
Queue elements: 3 4 5 6

 MENU
 1.insert
 2.delete
 3.display
 4.search
 5.exit
 enter your choice:2
Dequeued element: 3
Queue elements: 4 5 6
```

```
 MENU
 1.insert
 2.delete
 3.display
 4.search
 5.exit
 enter your choice:2
Dequeued element: 4
Queue elements: 5 6

 MENU
 1.insert
 2.delete
 3.display
 4.search
 5.exit
 enter your choice:2
Dequeued element: 5
Queue elements: 6

 MENU
 1.insert
 2.delete
 3.display
 4.search
```

```
 5.exit
 enter your choice:2
Dequeued element: 1
Queue is empty

 MENU
 1.insert
 2.delete
 3.display
 4.search
 5.exit
 enter your choice:5

 EXIT
```

RESULT

The program to implement circular queue is successfully executed and the output is verified.

Experiment No:6 **Date:4/12/2024**

SET DATA STRUCTURE

AIM

To implement set data structure and set set operations using bit string.

ALGORITHM

Get Universal Set (getUniversalSet)

1. Input the size of the universal set (superSetSize).
2. If the size exceeds the maximum allowed (MAX_SIZE), print an error and terminate.
3. Otherwise prompt the user to enter the elements of the universal set (superSet[]).

Get Set (getSet)

1. Input the set size (setSize) and its elements.
2. Prompt the user to input elements of a set (arr[]).

Check if Set is in Universal Set (checkSetInUniversal)

1. Input set (arr[]) and its size (size).
2. For each element of the set, check if it exists in the universal set (superSet[]).
3. If any element is not found in the universal set, print an error and return 0.

Generate Bit Strings for Sets (generateBitStrings)

1. Input the sets setA[] and setB[].
2. Initialize bit strings bitStringA[] and bitStringB[] of size superSetSize to 0.
3. For each element in setA[], set the corresponding bit in bitStringA[] to 1.
4. For each element in setB[], set the corresponding bit in bitStringB[] to 1.

Set Union (setUnion)

1. Perform a bitwise OR operation between the bit strings bitStringA[] and bitStringB[] to get the union.
2. Print the union bit string and the resulting set.

Set Intersection (setIntersection)

1. Perform a bitwise AND operation between the bit strings bitStringA[] and bitStringB[] to get the intersection.
2. Print the intersection bit string and the resulting set.

Set Difference A - B (setDifferenceAminusB)

1. Perform a bitwise AND operation between bitStringA[] and the complement of bitStringB[] to get the difference A - B.
2. Print the difference bit string and the resulting set.

Set Difference B - A (setDifferenceBminusA)

1. Perform a bitwise AND operation between bitStringB[] and the complement of bitStringA[] to get the difference B - A.

2. Print the difference bit string and the resulting set.

Print Bit String (printBitString)

1. Print the bit string as {0, 1, 0, 1, ...}

Print Set from Bit String (printSetFromBitString)

1. Iterate over the bit string, and for each 1, print the corresponding element from the universal set.

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];
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: ");
    printBitString(bitStringUnion, superSetSize);
    printf("Union Result (Values): ");
    printSetFromBitString(bitStringUnion, superSetSize);
}
void setIntersection() {
    int bitStringIntersection[MAX_SIZE];
    for (int i = 0; i < superSetSize; i++) {
        bitStringIntersection[i] = bitStringA[i] & bitStringB[i];
    }
    printf("Intersection: ");
    printBitString(bitStringIntersection, superSetSize);
    printf("Intersection Result (Values): ");
    printSetFromBitString(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): ");
    printBitString(bitStringDifferenceAminusB, superSetSize);
    printf("Difference Result (A - B, Values): ");
    printSetFromBitString(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): ");
    printBitString(bitStringDifferenceBminusA, superSetSize);
    printf("Difference Result (B - A, Values): ");
    printSetFromBitString(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;
}
```

OUTPUT

```
Enter Universal Set Size (max 20): 10
Enter 10 elements for the Universal Set:
Element 1: 1
Element 2: 2
Element 3: 3
Element 4: 4
Element 5: 5
Element 6: 6
Element 7: 7
Element 8: 8
Element 9: 9
Element 10: 10
Enter Set A Size (max 10): 5
Enter 5 elements (must be in the Universal Set):
Element 1: 3
Element 2: 6
Element 3: 2
Element 4: 8
Element 5: 9
Enter Set B Size (max 10): 4
Enter 4 elements (must be in the Universal Set):
Element 1: 1
Element 2: 0
Element 3: 4
Element 4: 5
ERROR!
```

```
Error: Element 0 is not in the Universal Set. Please enter the set again.
Enter 4 elements (must be in the Universal Set):
Element 1: 1
Element 2: 4
Element 3: 5
Element 4: 7
Set A Bit String: {0, 1, 1, 0, 0, 1, 0, 1, 1, 0}
Set B Bit String: {1, 0, 0, 1, 1, 0, 1, 0, 0, 0}

Choose an operation:
1. Union of A and B
2. Intersection of A and B
3. Difference (A - B)
4. Difference (B - A)
5. Exit
Enter your choice: 1
Union: {1, 1, 1, 1, 1, 1, 1, 1, 1, 0}
Union Result (Values): {1, 2, 3, 4, 5, 6, 7, 8, 9}

Choose an operation:
1. Union of A and B
2. Intersection of A and B
3. Difference (A - B)
4. Difference (B - A)
5. Exit
```

```
Enter your choice: 2
Intersection: {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
Intersection Result (Values): {}

Choose an operation:
1. Union of A and B
2. Intersection of A and B
3. Difference (A - B)
4. Difference (B - A)
5. Exit
Enter your choice: 3
Difference (A - B): {0, 1, 1, 0, 0, 1, 0, 1, 1, 0}
Difference Result (A - B, Values): {2, 3, 6, 8, 9}

Choose an operation:
1. Union of A and B
2. Intersection of A and B
3. Difference (A - B)
4. Difference (B - A)
5. Exit
Enter your choice: 4
Difference (B - A): {1, 0, 0, 1, 1, 0, 1, 0, 0, 0}
Difference Result (B - A, Values): {1, 4, 5, 7}
```

```
Choose an operation:
1. Union of A and B
2. Intersection of A and B
3. Difference (A - B)
4. Difference (B - A)
5. Exit
Enter your choice: 5
Exiting program.
```

RESULT

The program to implement set data structure and set set operations using bit string is successfully executed and the output is verified.

Experiment No:7 **Date:4/12/2024**

DISJOINT SET

AIM

To implement disjoint sets and the associated operations.

ALGORITHM

Make Set (makeSet)

1. Input an element x to be inserted in a new set.
2. Create a new node with data = x.
3. Set rep (representative) of the node to itself (i.e., rep = node).
4. Set next pointer to NULL (since it's the only element in the set).
5. Add this node to the list of disjoint sets (heads array) and set its tail.
6. Increment the countRoot (number of disjoint sets).

Find (find)

1. Input an element a whose representative is to be found.
2. Traverse through all sets to find the set containing a.
3. For each set, check if a is present.
4. Return the representative (rep) of the set containing a.
5. If a is not found, return NULL.

Union (unionSets)

1. Input two elements a and b to merge their sets.
2. Find the representatives (rep1 and rep2) of the sets containing a and b respectively.
3. If the representatives are different, merge the sets:
    1. Remove the set containing b from the list of sets.
    2. Merge the set containing b into the set containing a by linking the tail of the set of a to the head of the set of b.
    3. Update the rep pointers of all elements in the set of b to point to rep1.
4. If the sets already contain the same representative, no action is needed.

Search (search)

1. Input an element x to check if it exists in any of the disjoint sets.
2. Traverse all sets to check if x is present.
3. Return 1 if x is found, otherwise return 0.

Display Representatives (displayRepresentatives)

1. Traverse through all sets and display the data of the representative (rep) of each set.

Display Sets (displaySets)

1. Traverse through each disjoint set and print the elements in the set.

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;
    printf("Enter the size of the set (1 to 50): ");
    scanf("%d", &setSize);

    while (setSize <= 0 || setSize > 50) {
        printf("Invalid set size. Please enter a size between 1 and 50: ");
        scanf("%d", &setSize);
    }

    printf("\nEnter %d unique elements for the set:\n", setSize);
    for (int i = 0; i < setSize; ) {
        printf("Enter element %d: ", i + 1);
        scanf("%d", &x);
              if (search(x)) {
            printf("Element %d already exists in the set. Please enter a unique element.\n",
x);
        } else {
            makeSet(x);
            i++;
        }
    }
    do {
        printf("\n1. Display set representatives");
        printf("\n2. Union");
        printf("\n3. Find Set");
        printf("\n4. Display all sets");
        printf("\n5. Exit");
        printf("\nEnter your choice: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1:
                displayRepresentatives();
```

```
            break;
        case 2:
            printf("\nEnter first element: ");
            scanf("%d", &x);
            printf("Enter second element: ");
            scanf("%d", &y);
            unionSets(x, y);
            break;
        case 3:
            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 4:
            displaySets();
            break;
        case 5:
            printf("\nExiting program...\n");
            exit(0);
        default:
            printf("\nInvalid choice! Please try again.\n");
            break;
      }
   } while (1);
   return 0;
}
```

OUTPUT

```
Enter the size of the set (1 to 50): 10

Enter 10 unique elements for the set:
Enter element 1: 1
Enter element 2: 2
Enter element 3: 3
Enter element 4: 4
Enter element 5: 5
Enter element 6: 6
Enter element 7: 7
Enter element 8: 8
Enter element 9: 9
Enter element 10: 10

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

Set Representatives: 1 2 3 4 5 6 7 8 9 10
```

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

Enter first element: 3
Enter second element: 6

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

Set Representatives: 1 2 3 4 5 7 8 9 10

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

```
Enter the element to find: 6

The representative of 6 is 3

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

Disjoint Sets:
{ 1 }
{ 2 }
{ 3, 6 }
{ 4 }
{ 5 }
{ 7 }
{ 8 }
{ 9 }
{ 10 }
```

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

Exiting program...
```

RESULT

The program to implement disjoint sets and the associated operations is successfully executed and the output is verified.

Experiment No:8 **Date:7/12/2024**

GRAPH TRAVERSAL TECHNIQUES AND TOPOLOGICAL SORTING

AIM

To implement graph traversal techniques and topological sorting.

ALGORITHM

Graph Initialization

1. Create an array of linked lists (adjacency list) to represent the graph.
2. For each vertex, initialize its adjacency list as NULL (no edges initially).

Add Edge (Directed)

1. Create a new node for vertex `dest`.
2. Add this node to the adjacency list of vertex `src`.

Depth-First Search (DFS)

1. Mark the current vertex as visited.
2. Print the current vertex.
3. For each adjacent vertex of the current vertex:
   a. If the adjacent vertex is not visited, call DFS recursively on it.

Breadth-First Search (BFS)

1. Initialize a queue and mark the start vertex as visited.
2. Enqueue the start vertex.
3. While the queue is not empty:
   a. Dequeue a vertex.
   b. Print the dequeued vertex.
   c. For each adjacent vertex of the dequeued vertex:
      i. If the adjacent vertex is not visited, mark it as visited and enqueue it.

Topological Sort

1. Mark each vertex as not visited.
2. Initialize a stack to store the sorted vertices.
3. For each vertex:
   a. If the vertex is not visited, perform DFS on it.
4. During DFS, push vertices to the stack after visiting all their adjacent vertices.
5. If any vertex is revisited during DFS (cycle detected), print an error (graph is not a DAG).
6. Finally, print the vertices from the stack (topologically sorted).

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 Traversal 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[], bool
recursionStack[], int stack[], int* stackIndex, bool* hasCycle) {
    if (*hasCycle) return;
    visited[vertex] = true;
    recursionStack[vertex] = true;
    struct Node* adjList = graph->adjList[vertex];
    while (adjList != NULL) {
        int adjVertex = adjList->vertex;
        if (recursionStack[adjVertex]) {
            *hasCycle = true;
            return;
        }
        if (!visited[adjVertex]) {
            topologicalSortUtil(graph, adjVertex, visited, recursionStack, stack,
                    stackIndex, hasCycle);
        }
        adjList = adjList->next;
    }
    recursionStack[vertex] = false;
```

```
    stack[(*stackIndex)++] = vertex;
}
void topologicalSort(struct Graph* graph) {
    bool visited[MAX_VERTICES] = { false };
    bool recursionStack[MAX_VERTICES] = { false };
    int stack[MAX_VERTICES];
    int stackIndex = 0;
    bool hasCycle = false;
    for (int i = 0; i < graph->numVertices; i++) {
        if (!visited[i]) {
            topologicalSortUtil(graph, i, visited, recursionStack, stack,
                    &stackIndex, &hasCycle);
            if (hasCycle) {
                printf("The graph is not acyclic (contains a cycle).\n");
                return;
            }
        }
    }
    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("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);
        if (src >= 0 && src < vertices && dest >= 0 && dest < vertices) {
            addEdge(&graph, src, dest);
        } else {
            printf("Invalid edge! Please enter vertices within the range of 0 to %d.\n",
                    vertices - 1);
            i--;
        }
    }
    do {
        printf("\nMenu:\n");
        printf("1. Display Graph\n");
        printf("2. DFS Traversal\n");
        printf("3. BFS Traversal\n");
        printf("4. Topological Sort\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                displayGraph(&graph);
                break;
            case 2:
                printf("\nEnter the start vertex for DFS: ");
                scanf("%d", &startVertex);
                if (startVertex >= 0 && startVertex < vertices) {
                    DFS(&graph, startVertex);
                } else {
                    printf("Invalid start vertex!\n");
                }
                break;
            case 3:
                printf("\nEnter the start vertex for BFS: ");
                scanf("%d", &startVertex);
                if (startVertex >= 0 && startVertex < vertices) {
                    BFS(&graph, startVertex);
                } else {
                    printf("Invalid start vertex!\n");
                }
                break;
            case 4:
                topologicalSort(&graph);
                break;
            case 5:
```

```
            printf("Exiting the program.\n");
            break;
        default:
            printf("Invalid choice! Please try again.\n");
    }
  } while (choice != 5);

  return 0;
}
```

OUTPUT

```
Enter the number of vertices: 5
Enter the number of edges: 7
Enter edge 1 (source destination): 0 1
Enter edge 2 (source destination): 0 2
Enter edge 3 (source destination): 1 2
Enter edge 4 (source destination): 2 3
Enter edge 5 (source destination): 3 4
Enter edge 6 (source destination): 4 1
Enter edge 7 (source destination): 1 3

Menu:
1. Display Graph
2. DFS Traversal
3. BFS Traversal
4. Topological Sort
5. Exit
Enter your choice: 1

Graph Representation (Adjacency List):
Vertex 0: 2 -> 1 -> NULL
Vertex 1: 3 -> 2 -> NULL
Vertex 2: 3 -> NULL
Vertex 3: 4 -> NULL
Vertex 4: 1 -> NULL
```

```
Menu:
1. Display Graph
2. DFS Traversal
3. BFS Traversal
4. Topological Sort
5. Exit
Enter your choice: 2

Enter the start vertex for DFS:
0
DFS Traversal starting from vertex 0: 0 2 3 4 1

Menu:
1. Display Graph
2. DFS Traversal
3. BFS Traversal
4. Topological Sort
5. Exit
Enter your choice: 3

Enter the start vertex for BFS: 4
BFS starting from vertex 4: 4 1 3 2
```

```
Menu:
1. Display Graph
2. DFS Traversal
3. BFS Traversal
4. Topological Sort
5. Exit
Enter your choice: 4
The graph is not acyclic (contains a cycle).

Menu:
1. Display Graph
2. DFS Traversal
3. BFS Traversal
4. Topological Sort
5. Exit
Enter your choice: 5
Exiting the program.
```

RESULT

The program to implement graph traversal techniques and topological sorting is successfully executed and the output is verified.

Experiment No:9 **Date:11/12/2024**

STRONGLY CONNECTED COMPONENTS

AIM

To implement finding the strongly connected components in a directed graph.

ALGORITHM

Initialization

1. Initialize an empty stack stack to store the vertices in the order of their finishing times.
2. Initialize a boolean array visited[] to keep track of visited vertices.
3. Initialize the graph's adjacency list and its reverse (transposed) adjacency list.

DFS on Original Graph

1. For each unvisited vertex vvv in the original graph GGG:
    a. Perform a DFS starting from vertex vvv.
    b. During the DFS, mark the vertex as visited.
    c. Once the DFS is complete for a vertex vvv, push vvv to the stack.

Reverse the Graph

1. Reverse the direction of all edges in the graph (this is already handled during edge insertion in the adjacency list).
2. The reverse of the graph is used to identify SCCs.

DFS on Reversed Graph

1. Re-initialize the visited[] array as false for all vertices.
2. While the stack is not empty:
    a. Pop a vertex vvv from the stack.
    b. If vvv has not been visited yet:
        i. Perform DFS starting from vvv on the reversed graph GTG^TGT and mark all reachable vertices in this DFS traversal as part of the same SCC.
        ii. Each DFS traversal on the reversed graph represents one SCC.

Output the SCCs

1. Output the list of strongly connected components (SCCs) found in the previous step.
2. Each SCC is a set of vertices that are mutually reachable from each other in both directions.

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* reverseAdjList[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;
        graph->reverseAdjList[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;
    newNode = createNode(src);
    newNode->next = graph->reverseAdjList[dest];
    graph->reverseAdjList[dest] = newNode;
}
void DFS(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]) {
            DFS(graph, adjVertex, visited, stack, stackIndex);
        }
        adjList = adjList->next;
    }
    stack[*stackIndex] = vertex;
    (*stackIndex)++;
}
void DFSReverse(struct Graph* graph, int vertex, bool visited[], int* component) {
    visited[vertex] = true;
    component[vertex] = 1;
```

```
    struct Node* adjList = graph->reverseAdjList[vertex];
    while (adjList != NULL) {
        int adjVertex = adjList->vertex;
        if (!visited[adjVertex]) {
            DFSReverse(graph, adjVertex, visited, component);
        }
        adjList = adjList->next;
    }
}
void kosarajuSCC(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]) {
            DFS(graph, i, visited, stack, &stackIndex);
        }
    }
    for (int i = 0; i < graph->numVertices; i++) {
        visited[i] = false;
    }
    printf("Strongly Connected Components (SCCs):\n");
    while (stackIndex > 0) {
        int vertex = stack[--stackIndex];
        if (!visited[vertex]) {
            int component[MAX_VERTICES] = { 0 };
            DFSReverse(graph, vertex, visited, component);
            printf("{ ");
            for (int i = 0; i < graph->numVertices; i++) {
                if (component[i]) {
                    printf("%d ", 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, choice;
    printf("Enter the number of vertices: ");
    scanf("%d", &vertices);
    if(vertices>MAX_VERTICES){
        printf("\n Size exceeded");
        exit(0);
    }
    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);
        if (src >= 0 && src < vertices && dest >= 0 && dest < vertices) {
            addEdge(&graph, src, dest);
        }
        else {
            printf("Invalid edge! Please enter vertices within the range of 0 to %d.\n",
                    vertices - 1);
            i--;
        }
    }
    do {
        printf("\nMenu:\n");
        printf("1. Display Graph\n");
        printf("2. Find Strongly Connected Components (SCCs)\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                displayGraph(&graph);
                break;
            case 2:
                kosarajuSCC(&graph);
                break;
            case 3:
                printf("Exiting the program.\n");
                break;
```

```
            default:
                printf("Invalid choice! Please try again.\n");
        }
    } while (choice != 3);
    return 0;
}
```

OUTPUT

```
Enter the number of vertices: 7
Enter the number of edges: 10
Enter edge 1 (source destination): 0 1
Enter edge 2 (source destination): 1 2
Enter edge 3 (source destination): 1 3
Enter edge 4 (source destination): 2 3
Enter edge 5 (source destination): 2 5
Enter edge 6 (source destination): 3 0
Enter edge 7 (source destination): 3 4
Enter edge 8 (source destination): 4 5
Enter edge 9 (source destination): 5 6
Enter edge 10 (source destination): 6 4

Menu:
1. Display Graph
2. Find Strongly Connected Components (SCCs)
3. Exit
Enter your choice: 1

Graph Representation (Adjacency List):
Vertex 0: 1 -> NULL
Vertex 1: 3 -> 2 -> NULL
Vertex 2: 5 -> 3 -> NULL
Vertex 3: 4 -> 0 -> NULL
Vertex 4: 5 -> NULL
Vertex 5: 6 -> NULL
```

```
Menu:
1. Display Graph
2. Find Strongly Connected Components (SCCs)
3. Exit
Enter your choice: 2
Strongly Connected Components (SCCs):
{ 0 1 2 3 }
{ 4 5 6 }

Menu:
1. Display Graph
2. Find Strongly Connected Components (SCCs)
3. Exit
Enter your choice: 3
Exiting the program.
```

RESULT

The program to implement finding the strongly connected components in a directed graph is successfully executed and the output is verified.

Experiment No:10 **Date:18/12/2024**

PRIM'S ALGORITHM

AIM

To implement prim's algorithm for finding the minimum cost spanning tree.

ALGORITHM

1. Associate with each vertex V for the graph no C[V] and edge E[V].

   To initialize these values, set all value of C[V] to + infinity and set each E[V] to a special flag value indicating that there is no edge connecting V to earlier vertices.

2. Indicating an empty forest F and a set n of vertices that have not yet been included in F.

   Repeat the following steps until queue is empty.

   1. Find and remove G vertex V from 0 having the minimum possible value of E[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 for each v such edge, if w still belongs to G and vw has smallest weight when C[w], perform the following steps.

   a) Set C[w] to the cost of edge vw.

   b) Set E[w] to the point to edge vw.

   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];
int main()
{
	printf("\nEnter the number of nodes:");
	scanf("%d",&n);
	printf("\nEnter 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;
			}
		visited[0]=1;
		printf("\n");
		while(ne<n)
```

```
        {
        for(i=0,min=999;i<n;i++)
                for(j=0;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);
    return 0;
}
```

OUTPUT

```
Enter the number of nodes:9

Enter the adjacency matrix:
0 4 0 0 0 0 0 8 0
4 0 8 0 0 0 0 11 0
0 8 0 7 0 4 0 0 2
0 0 7 0 9 14 0 0 0
0 0 0 9 0 10 0 0 0
0 0 4 14 10 0 2 0 0
0 0 0 0 0 2 0 1 6
8 11 0 0 0 0 1 0 7
0 0 2 0 0 0 6 7 0


Edge 1:(0 1) cost:4
Edge 2:(0 7) cost:8
Edge 3:(7 6) cost:1
Edge 4:(6 5) cost:2
Edge 5:(5 2) cost:4
Edge 6:(2 8) cost:2
Edge 7:(2 3) cost:7
Edge 8:(3 4) cost:9
Minimum cost:37
```

RESULT

The program to implement prim's algorithm to find minimum cost spanning tree is successfully executed and the output is verified.

Experiment No:11 **Date:18/12/2024**

KRUSKAL'S ALGORITHM

AIM

To implement kruskal's algorithm using disjoint set data structure.

ALGORITHM

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

SOURCE CODE

```
#include <stdio.h>
#include <stdlib.h>
typedef struct {
	int src, dest, weight;
} Edge;
typedef struct {
	int parent, rank;
} Subset;
int find(Subset subsets[], int i) {
	if (subsets[i].parent != i) {
		subsets[i].parent = find(subsets, subsets[i].parent);
	}
	return subsets[i].parent;
}
void unionSets(Subset subsets[], int x, int y) {
	int xroot = find(subsets, x);
	int yroot = find(subsets, y);
	if (subsets[xroot].rank < subsets[yroot].rank) {
		subsets[xroot].parent = yroot;
	}
	else if (subsets[xroot].rank > subsets[yroot].rank) {
		subsets[yroot].parent = xroot;
	}
	else {
		subsets[yroot].parent = xroot;
		subsets[xroot].rank++;
	}
}
int compareEdges(const void *a, const void *b) {
```

```
    return ((Edge *)a)->weight - ((Edge *)b)->weight;
}
void kruskalMST(Edge *edges, int V, int E) {
    qsort(edges, E, sizeof(Edge), compareEdges);
    Subset *subsets = (Subset *)malloc(V * sizeof(Subset));
    for (int v = 0; v < V; v++) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }
    Edge *result = (Edge *)malloc((V - 1) * sizeof(Edge));
    int i = 0;
    int total = 0;
    for (int j = 0; j < E && i < V - 1; j++) {
        Edge next_edge = edges[j];
        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);
        if (x != y) {
            result[i++] = next_edge;
            total += next_edge.weight;
            unionSets(subsets, x, y);
        }
    }
    printf("Minimum Spanning Tree (Kruskal's Algorithm):\n");
    for (int j = 0; j < i; j++) {
        printf("%d -- %d (Weight: %d)\n", result[j].src, result[j].dest,
result[j].weight);
    }
    printf("Total cost is: %d\n", total);
    free(subsets);
    free(result);
}
int main() {
    int v, e;
    printf("Enter the number of vertices: ");
    scanf("%d", &v);
    printf("Enter the number of edges: ");
    scanf("%d", &e);
    Edge *edges = (Edge *)malloc(e * sizeof(Edge));
    printf("Enter the edges (src, dest, weight):\n");
    for (int i = 0; i < e; i++) {
        scanf("%d %d %d", &edges[i].src, &edges[i].dest, &edges[i].weight);
    }
    kruskalMST(edges, v, e);
    free(edges);
    return 0;
```

}

OUTPUT

```
Enter the number of vertices: 9
Enter the number of edges: 14
Enter the edges (src, dest, weight):
0 1 4
0 7 8
1 2 8
1 7 11
7 8 7
2 8 2
7 6 1
8 6 6
2 3 7
2 5 4
6 5 2
5 4 10
3 5 14
3 4 9
Minimum Spanning Tree (Kruskal's Algorithm):
7 -- 6 (Weight: 1)
2 -- 8 (Weight: 2)
6 -- 5 (Weight: 2)
0 -- 1 (Weight: 4)
2 -- 5 (Weight: 4)
2 -- 3 (Weight: 7)
0 -- 7 (Weight: 8)
3 -- 4 (Weight: 9)
```

RESULT

The program to implement kruskal's algorithm using disjoint data structure is successfully executed and the output is verified.