DATE : 03-10-2024

PROGRAM NO:4.1

SIMPLE ARRAY OPERATIONS -SUM OF ELEMENTS IN AN ARRAY

AIM : Write a program to find sum of elements in an array.

SOURCE CODE:

```
#include<stdio.h>

void main()
{

      int a[15] , i=0 , n , sum=0;

      printf("Enter the limit:");

      scanf("%d",&n);

      printf("Enter the element:");

      for(i=0;i<n;i++)
      {
         scanf("%d",&a[i]);
      }

      for(i=0;i<n;i++)
      {
          sum=sum+a[i];
      }

      printf("SUM=%d",sum);

}
```

ALGORITHM:

1:Start

2: Read the limit n

3: Set variable i=0 ,sum=0

4: check whether i<n if true go to step 5 otherwise goto step 7

5: Read the elements in array a[i]

6: Increment i by 1 go to step 4

7: Set i=0

8: Check whether i<n if true go to step 9,otherwise go to step 11

9: Set sum=sum+a[i]

10: Increment i by 1 go to step 8

11: Display Sum

12: Stop

OUTPUT:

Enter the limit: 5

Enter the element: 1 2 3 4 5

SUM= 15

DATE: 03-10-2024

PROGRAM NO:4.2

SIMPLE ARRAY OPERATIONS –LINEAR SEARCH

AIM : Write a program to perform linear search .

SOURCE CODE:

```
#include<stdio.h>
void main()
{
      int a[50],i,n,key,flag=0;

      printf("Enter the limit of array :");
      scanf("%d",&n);


      printf("Enter the array elements :");

      for(i=0;i<n;i++)
      {
           scanf("%d",&a[i]);
      }

      printf("Enter the key to be searched :");
      scanf("%d",&key);

      for(i=0;i<n;i++)
      {
         if(a[i] == key)
         {
              flag=1;
              break;
         }

      }

      if(flag == 1 )
      {

           printf(" Search successfull, Element found  ");

      }
```

ALGORITHM:

1: Start

2: Read the limit n from the user

3: Initialize i to 0 and f to 0

4: Loop from i = 0 to i < n:

- Read elements into array a[i]
- Increment i

5: Read the key element to search (key)

6: Loop from i = 0 to i < n:

Check whether key == a[i], set f = 1 and break the loop

7: Check If f == 1, display "Search successful"

8: Otherwise, display "Search unsuccessful"

9: Stop

OUTPUT:

Enter the limit of array : 5

Enter the array elements : 7 8 9 2 4

Enter the key to be searched :2

Search successfull, Element found

Enter the key to be searched: 1

Search Unsuccessfull, Element Not Found

```
while(i < m && j < n)
{
    if(a[i]< b[j])
    {
        c[k]=a[i];
        k++;
        i++;
    }
    else
    {
        c[k]=b[j];
        k++;
        j++;
    }
}

while(i<m)
{
    c[k]=a[i];
    i++;
    k++;
}
while(j<n)
{
    c[k]=b[j];
    k++;
    j++;
}


printf("\n Merged array = ");
```

ALGORITHM:

1. Start
2. Initialize three arrays: a[50], b[50], c[100].
3. Read the size of first array to variable m.
4. Read the size of second array to variable n.
5. Set i = 0, j = 0, k = 0.
6. While i < m and j < n:
    a. Check whether a[i] < b[j], set c[k] = a[i], increment i.
    b. Otherwise, set c[k] = b[j], increment j.
    c. Increment k by 1.
7. While j < n, set c[k] = b[j], increment jand k by 1.
8. While i < m, set c[k] = a[i], increment iand k by 1.
9. Display "Merged Array:" and print c elements.
10. Stop.

OUTPUT:

Enter the size of first array: 4

Enter the elements of first array (sorted order): 2 4 6 9

Enter the size of second array:3

Enter the elements of second array (sorted order): 1 3 8

After Merging: 1 2 3 4 6 8 9

DATE: 17-10-2024

PROGRAM NO: 5.2

SORTING ALGORITHMS -BUBBLE SORT

AIM: Write a program to implement bubble sort.

SOURCE CODE:

```
#include<stdio.h>
void main()
{
  int n,i,j,array[50],temp;

  printf("Enter the limit of array: ");
  scanf("%d",&n);

  printf("\n Enter the elements of the array:");
   for(i=0;i<n;i++)
    {
      scanf("%d",&array[i]);
    }

   for(i=0;i<n-1;i++)
    {
        for(j=0;j<(n-i-1);j++)
        {
            if(array[j]>array[j+1])
            {
                temp=array[j];
                array[j]=array[j+1];
                array[j+1]=temp;
            }
        }
}
```

ALGORITHM:

1. Start

2. Initialize an integer array arr[10] to store the elements.

3. Initialize integers n, i, j, and temp.

4. Display "Enter the size of the array:"

5. Read the limit n .

6. Display "Enter the array elements:"

7. Read the array elements from the user and store them in arr.

8. Set i = 0.

9. Repeat the following steps for i from 0 to n-1:

 a. Initialize j = 0.

 b. Repeat the following steps for j from 0 to n-1-i:

  1. Check whether arr[j] > arr[j+1], then

   a. Swap arr[j] and arr[j+1] using a temporary variable temp.

 c. Increment j by 1.

10. Display "Sorted Array = ".

11. Repeat the following steps for i from 0 to n:

  a. Display the value of arr[i].

12. Stop

OUTPUT:

Enter the limit of array: 5

Enter the array elements: 4 3 6 2 10

Sorted Array = 2 3 4 6 10

DATE : 17 -10-2024

PROGRAM NO: 6

STACK IMPLEMENTATION USING ARRAY

AIM: Write a program to implement stack operations.

.

SOURCE CODE:

```
#include <stdio.h>
#define MAX 10

int stack[MAX];
int top = -1;
void push(int item) {
    if (top == MAX - 1) {
        printf("\n Stack Overflow!");
        return;
    }
    stack[++top] = item;
    printf("\n %d pushed to stack.", item);
}
void pop() {
    if (top == -1) {
        printf("\n Stack Underflow!");
        return;
    }
    printf("\n %d popped from stack.", stack[top--]);
}
void peek() {
    if (top == -1) {
        printf("\n Stack is Empty!");
```

ALGORITHM:

push():

1. Start.
2. Check if the stack is full (top == MAX - 1).
3. If true, display "Stack Overflow" and exit.
4. Otherwise, increment top by 1.
5. Insert the new element at stack[top].
6. Stop.

pop():

1. Start.
2. Check if the stack is empty (top == -1).
3. If true, display "Stack Underflow" and exit.
4. Otherwise, retrieve and display the value at stack[top].
5. Decrement top by 1.
6. Stop.

peek():

1. Start.
2. Check if the stack is empty (top == -1).
3. If true, display "Stack is Empty" and exit.
4. Otherwise, display the value at stack[top].
5. Stop.

display():

1. Start.
2. Check if the stack is empty (top == -1).
3. If true, display "Stack is Empty" and exit.
4. Otherwise, print all elements from top to 0.
5. Stop.

main()

1. Start
2. Initialize variable choice, value
3. Display Stack operations
4. Read choice from user
5. 1: read element from user to insert
   Call push()
   2: Call pop()
   3: Call peek()
   4: Call display()
   5: Exit
6. Repeat until choice is 5
7. Stop

```
        return;
    }
    printf("\n Top element is %d.", stack[top]);
}
void display() {
    if (top == -1) {
        printf("\n Stack is Empty!");
        return;
    }
    printf("\nStack elements are:");
    for (int i = top; i >= 0; i--) {
        printf(" %d", stack[i]);
    }
}
int main() {
    int choice, value;
    while (1) {
        printf("\n\n Stack Operations Menu:");
        printf("\n1. Push");
        printf("\n2. Pop");
        printf("\n3. Peek");
        printf("\n4. Display");
        printf("\n5. Exit");
        printf("\n Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1: printf("\n Enter value to push: ");
                    scanf("%d", &value);
                    push(value);
                    break;
```

OUTPUT:

Stack Operations Menu:

1. Push
2. Pop
4. Display
5. Exit

Enter your choice: 1

Enter value to push: 10

10 pushed to stack.

Enter your choice: 1

Enter value to push: 20

20 pushed to stack.

Enter your choice: 4

Stack elements are: 20 10

Enter your choice: 3

Top element is 20.

Enter your choice: 2

20 popped from the stack.

Enter your choice: 4

Stack elements are: 10

Enter your choice: 5

Exiting program.

DATE: 24-10-2024

PROGRAM NO: 7

CIRCULAR QUEUE USING ARRAY

AIM: Write a program to implement circular queue.

SOURCE CODE:

```
#include<stdio.h>
#define MAX 5
int queue[MAX];
int front = -1,rear = -1;

int isFull()
{
    if((rear+1) % MAX == front)
    {
        return 1;
    }

    return 0;
}

int isEmpty()
{
    if(front == -1 && rear == -1)
    {
        return 1;
    }

    return 0;
}

void display()
{
    int i;

    if(isEmpty())
    {
        printf("\n QUEUE IS EMPTY \n");
        return;
    }

    printf("\n QUEUE ELEMENTS: ");

    i = front;

    do {
        printf("%d ", queue[i]);
        i = (i + 1) % MAX;
    } while (i != (rear + 1) % MAX);

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

ALGORITHM:

enqueue ()

1: Start.
2: Check whether the queue is full using isFull()
    •If true, print "QUEUE IS FULL" and return.
3: If the queue is empty , set front = 0.
4: Update rear = (rear + 1) % MAX.
5: Read x (element to be inserted).
6: Insert x into queue[rear] and print "ELEMENT INSERTED".
7: End.

dequeue ()

1: Start.
2: Check whether the queue is empty using isEmpty()
    •If true, print "QUEUE IS EMPTY" and return.
3: Print queue[front] as deleted.
4: Check whether front == rear: Set front = rear = -1.
5: Otherwise, update front = (front + 1) % MAX.
6: End.

display ()

1: Start.
2: Check whether the queue is empty using isEmpty()
    •If true, print "QUEUE IS EMPTY" and return.
3: Initialize i = front.
4: Repeat until i == (rear + 1) % MAX:
    •Print queue[i].
    •Update i = (i + 1) % MAX.
5: End.

search ()

1: Start.
2: Check whether the queue is empty using isEmpty()
    •If true, print "QUEUE IS EMPTY" and return.
3: Read key (element to search).
4: Initialize i = front and repeat:
    •If queue[i] == key: Print "ELEMENT FOUND AT POSITION i" and return.
    •Update i = (i + 1) % MAX.
5: If loop completes and key is not found, print "ELEMENT NOT FOUND".
6: End.

is Full ()

1: Start.
2: Return 1 if (rear + 1) % MAX == front.
3: Otherwise, return 0.
4: End.

is Empty ()

1: Start.
2: Return 1 if front == -1.
3: Otherwise, return 0.
4: End.

```
void dequeue()
{
    if(isEmpty())
    {
        printf("\n QUEUE IS EMPTY \n");
        return;
    }

    printf("\n %d is DELETED \n",queue[front]);

    if(front == rear)
    {
         front = rear = -1;
    }
    else
    {
         front = (front + 1)%MAX;
    }
}


void enqueue()
{
    int x;

    if(isFull())
    {
        printf("\n QUEUE IS FULL \n");
        return;
    }

    printf("Enter the element to insert ");
    scanf("%d",&x);

    if(isEmpty())
    {
        front = rear = 0;
    }
    else
    {
         rear = (rear+1) % MAX;
    }

    queue[rear]=x;
    printf("\n ELEMENT %d  INSERTED SUCESSFULLY \n ",queue[rear]);

}


void search()
{
    int key, i,found = 0;

    if (isEmpty())
    {
        printf("\n QUEUE IS EMPTY \n");
        return;
    }
     printf("\nEnter the element to search: ");
     scanf("%d", &key);
```

main ()

1: Start.
2: Display menu:
3: Based on the user’s choice, call the corresponding function:
    1: Call enqueue().
    2: Call dequeue().
    3: Call display().
    4: Call search().
    5: Print "EXITING..." and terminate.
4: Repeat until choice is 5.
5: End

OUTPUT :

CIRCULAR QUEUE USING ARRAYS

1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. SEARCH
5. EXIT

Enter your choice: 1
Enter the element to insert: 10
ELEMENT 10 INSERTED SUCCESSFULLY

Enter your choice: 1
Enter the element to insert: 20
ELEMENT 20 INSERTED SUCCESSFULLY

Enter your choice: 3
QUEUE ELEMENTS: 10 20

Enter your choice: 4
Enter the element to search: 20
ELEMENT FOUND AT POSITION 1

Enter your choice: 2
10 IS DELETED

Enter your choice: 3
QUEUE ELEMENTS: 20

Enter your choice: 5
Exiting....

DATE: 24-10-2024

PROGRAMNO: 8

SINGLY LINKED LIST

AIM: Write a C program to implement singly linked list and it's operations.

SOURCE CODE:

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


struct node
{
  int data;
  struct node *link;
};


struct node *head=NULL;


void insertFirst()
{
  struct node *newnode;
  newnode=(struct node*)malloc(sizeof(struct node));

  if(newnode == NULL) {
      printf("\n NO space avilable \n");
      return;
  }

   newnode->link=NULL;
   printf("\n Enter the value to insert to Front \n");
```

ALGORITHM:

insertFirst()

1. Start

2. Create newnode and allocate memory for newnode using malloc

3.Read the element to be inserted and set newnode->link=NULL

4.Check whether head==NULL if true go to step5 otherwise go to step6

5.Set head=newnode

6.Set newnode->link=head and head=newnode

7.Display the inserted element

8.Stop

insertLast()

1.Start

2.Create newnode and allocate memory for newnode using malloc

3.Set newnode->link=NULL and temp=head

4.Check whether head==NULL if true go to step5 otherwise go to step6

5.Set head=newnode

6.While temp->link!=NULL is true go to step7 otherwise go to step8

7.set temp=temp->link

8.Read the element to be inserted

9.Set temp->link=newnode

10.Display newly inserted element

11.Stop

insertLocation()

1.Start

2.Create newnode and allocate memory for newnode using malloc

3.Set newnode->link=NULL

4.Read the value of the node after which the newnode should be inserted and store in variable ‘key’

5.Set temp=head

6.While temp!=NULL and temp->data!=key is true go to step7 otherwise go to step8

```
    scanf("%d",&newnode->data);
    if(head==NULL) {
       head=newnode;
    }
    else {
       newnode->link=head;
        head=newnode;
    }
    printf("\n Element inserted %d",newnode->data);

  }




void insertLast()
{
  struct node *temp=head,*newnode;
  newnode=(struct node*)malloc(sizeof(struct node));

  if(newnode==NULL)  {
       printf("\n No space Available\n");
       return;
  }
    newnode->link=NULL;

    printf("\n Enter the element to insert Last \n ");
    scanf("%d",&newnode->data);

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

7.Set temp=temp->link

8.Check whether temp==NULL is true go to step9 otherwise go to step10

9.Display the key value does not exist and return

10.Read the element to be inserted

11.Set newnode->link=temp->link and temp->link=newnode

12.Display the inserted element

13.Stop

deleteFirst()

1.Start

2.Check whether head==NULL is true go to step3 otherwise go to step4

3.Display list is empty and return

4.Set head=temp

5.Set head=temp->link

6.Display deleted element

7. Free temp

8.Stop

deleteLast()

1.Start

2.Check whether head==NULL if true go to step3 otherwise goto step4

3.Display list is empty and return

4.Set temp=head and prev=NULL

5.Check whether temp->link==NULL if true go to step6 otherwise go to step7

6.Free temp and set head=NULL and return

7.While temp->link!=NULL is true go to step8 otherwise go to step9

8.Set prev=temp and temp=temp->link

9.Display the deleted element

10.Free temp and set prev->link=NULL

11.Stop

```
    else {
       while(temp->link!=NULL) {
                 temp=temp->link;
        }
        temp->link=newnode;

    }
    printf("element inserted successfully  %d",newnode->data);
  }



void insertLocation()
{
  int key;
  struct node *temp=head,*newnode;
  newnode=(struct node*)malloc(sizeof(struct node));
  if(newnode ==NULL) {
      printf("\n No space available \n");
      return;
  }

  newnode->link=NULL;
  if(head==NULL) {
      printf("\n LIST empty \n");
  }
  else {
      printf("\n Enter the key were after you want to add Element \n");
      scanf("%d",&key);
```

deleteLocation()

1.Start

2.Sest temp=head and prev=NULL

3.Read the value of the node to be deleted inn variable ‘key’

4.Check whether head==NULL if true, display list is empty and return

5.Check whether temp->data==key is true go to step6 otherwise go to step8

6.Set head=temp->link

7.Free temp and return

8.While temp!=NULL and temp->data!=key is true go to step9 otherwise go to step10

9.Set prev=temp and temp=temp->link

10.Check whether temp==NULL is true go to step11 otherwise go to step12

11.Display key does not exist and return

12.Set prev->link=temp->link

13.Display deleted element

14.Free temp

15.Stop

search()

1.Start

2.Set temp=head, pos=0 and found=0

3.Check whether head==NULL if true, display list is empty and return

4.Read the element to be searched in ‘val’

5.While temp!=NULL is true go to step6 otherwise go to step10

6.Check whether temp->data==val is true go to step7 otherwise go to step8

7.Display the position of val and set found=1

8.Increment value of pos by 1

9.Set temp=temp->link

10.If !found is true display value does not exist

11.Stop

```
    while(temp != NULL && temp->data != key) {
            temp=temp->link;
    }
    if(temp == NULL) {
      printf("\n Value Not Exist\n");
    }
    else {
      printf("\n Enter the Element to inserted\n");
      scanf("%d",&newnode->data);
      newnode->link=temp->link;
      temp->link=newnode;
      printf("value inserted successfully %d",newnode->data);
    }
  }
}


void deleteFirst()
{
     if(head==NULL) {
       printf("\n List Empty\n");
      return;
     }
     struct node *temp =head;
     head=temp->link;
     printf("\n Value deleted %d \n",temp->data);
     free(temp);

}
```

display()

1.Start

2.Set temp=head

3.Check whether temp==NULL is true display list is empty and return

4.While temp!=NULL is true go to step5 otherwise go to step7

5.Display temp->data

6.Set temp=temp->link

7.Stop

main()

1.Start

2.Perform the given operations until choice!=9 using do while loop

3.Read choice

4. Check the value of choice

5. When choice==1

   Call insertFirst()

   Break

6. When choice==2

   Call insertLast()

   Break

7. When choice==3

   Call insertLocation()

   Break

8. When choice==4

   Call deleteFirst()

   Break

9.When choice==5

   Call deleteLast()

   Break

10.When choice==6

```
void deleteLast()
{
    if(head==NULL)
    {
        printf("\n Empty list \n");
        return;
    }
    struct node *temp=head,*prev=NULL;
    if(temp->link==NULL)
    {
        printf("\n Value %d deleted",temp->data);
        free(temp);
        head=NULL;
        return;
    }
    while(temp->link!=NULL)
    {
        prev=temp;
        temp=temp->link;
    }
    printf("\nvalue %d deleted\n",temp->data);
    prev->link=NULL;
    free(temp);
}
```

Call deleteLocation()

Break

11. Whenchoice==7

Call search()

Break

12. When choice==8

Call display()

Break

13. When choice==9

Display exit

Exit(0)

Break

14.default:

Display enter valid choice

15.stop

OUTPUT:

SINGLY LINKED LIST

1-> InsertFirst

2-> InsertLast

3-> Insert Location

4-> Delete first

5-> Delete last

6-> Delete location

7-> Display

8-> Search

9-> Exit

Enter Choice:

1

```
void deleteLocation()
{
  int key;
  struct node *temp=head,*prev=NULL;

  if(head==NULL) {
     printf("\n Empty list \n");
     return;
  }

  printf("\n Enter the element that you want to delete\n");
  scanf("%d",&key);

  if(temp->data ==key) {
     head=temp->link;
     printf("\n value %d is deleted \n",temp->data);
     free(temp);
     return;
  }

  while(temp!=NULL && temp->data!=key) {
       prev=temp;
       temp=temp->link;
  }

  if(temp==NULL) {
     printf("\n Value Not exsist \n");
     return;
  }
  prev->link=temp->link;
  printf("value %d is deleted",temp->data);
```

Enter the value to insert to Front

1

Element inserted 1

1-> InsertFirst

2-> InsertLast

3-> Insert Location

4-> Delete first

5-> Delete last

6-> Delete location

7-> Search

8-> Display

9-> Exit

Enter Choice:

1

Enter the value to insert to Front

2

Element inserted 2

1-> InsertFirst

2-> InsertLast

3-> Insert Location

4-> Delete first

5-> Delete last

6-> Delete location

7-> Search

8-> Display

9-> Exit

Enter Choice:

2

Enter the element to insert Last

3

```
    free(temp);
}


void search()
{
    struct node *temp=head;
    int pos=0,found=0,val;
    if(head==NULL) {
        printf("\n Empty List \n");
        return;
    }
    printf("\n Enter the value to search");
    scanf("%d",&val);

   while(temp != NULL) {
        if(temp->data == val) {
            printf("%d value found at %d location \n",temp->data,pos);
            found=1;
        }
        pos++;
        temp=temp->link;
    }
    if(!found) {
        printf("Value %d not exsist",val);
    }

}
```

element inserted successfully 3

1-> InsertFirst

2-> InsertLast

3-> Insert Location

4-> Delete first

5-> Delete last

6-> Delete location

7-> Search

8-> Display

9-> Exit

Enter Choice:

2

Enter the element to insert Last

4

element inserted successfully 4

1-> InsertFirst

2-> InsertLast

3-> Insert Location

4-> Delete first

5-> Delete last

6-> Delete location

7-> Search

8-> Display

9-> Exit

Enter Choice:

3

Enter the key were after you want to add Element

2

Enter the Element to inserted

5

```
void display()
{
   struct node *temp=head;

     if(temp==NULL)
    {
        printf("\n List Empty");
        return;
    }

   printf("\n Elements in the List \n");

   while(temp!=NULL)
   {
      printf("%d ",temp->data);
       temp=temp->link;

   }

}




void main()

{

  int choice;

  printf("\n SINGLY LINKED LIST \n");


   do{

      printf("\n 1-> InsertFirst \n 2-> InsertLast \n 3-> Insert Location \n 4-> Delete first \n
               5-> Delete last \n 6-> Delete location \n 7-> Search\n 8-> Display\n 9-> Exit");

      printf("\n Enter Choice: \n");

       scanf("%d",&choice);

       switch(choice)

      {

         case 1: insertFirst();
```

value inserted successfully 5

1-> InsertFirst

2-> InsertLast

3-> Insert Location

4-> Delete first

5-> Delete last

6-> Delete location

7-> Search

8-> Display

9-> Exit

Enter Choice:

8

Elements in the List

2 5 1 3 4

1-> InsertFirst

2-> InsertLast

3-> Insert Location

4-> Delete first

5-> Delete last

6-> Delete location

7-> Search

8-> Display

9-> Exit

Enter Choice:

7

Enter the value to search5

5 value found at 1 location

DATE: 01-11-2024

PROGRAM NO: 9.1

SINGLY LINKED STACK

AIM: Write a C program to implement stack operations using singly linked list.

SOURCE CODE:

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

struct node {
   int data;
   struct node* link;
};


struct node *top=NULL;



void push()
{
  struct node *newnode;
  newnode=(struct node*)malloc(sizeof(struct node));

  if (newnode==NULL){
      printf("\n No space avialble\n");
      return;
  }
  newnode->link=NULL;

  printf("\n Enter the element to insert\n");
```

ALGORITHM:

push()

1. Start

2. Create newnode and allocate memory for newnode using malloc

3.Read the element to be inserted and set newnode->link=NULL

4.Check whether top==NULL if true go to step5 otherwise go to step6

5.Set top=newnode

6.Set newnode->link=top and top=newnode

7.Display the inserted element

8.Stop

pop()

1.Start

2.Check whether top==NULL is true go to step3 otherwise go to step4

3.Display stack is empty and return

4.Set top=temp

5.Set top=temp->link

6.Display deleted element

7.Free temp

8.Stop

peek()

1.Start

2.Set temp=top

3.Check whether temp==NULL is true display stack underflow and return

4.Display top as temp->data

```
  scanf("%d",&newnode->data);

  if(top==NULL){
     top=newnode;
  }
  else{
     newnode->link=top;
      top=newnode;
  }
   printf("\n %d INSERTED SUCCESFULLY",newnode->data);
}

void pop()
{
  struct node *temp=top;

  if(top==NULL){
     printf("\n STACK UNDER FLOW");
     return;
  }

  printf("\n %d is popped ",temp->data);
  top=temp->link;
  free(temp);

}
```

display()

1.Start

2.Set temp=top

3.Check whether temp==NULL is true display stack is empty and return

4.While temp!=NULL is true go to step5 otherwise go to step7

5.Display temp->data

6.Set temp=temp->link

7.Stop

search()

1.Start

2.Set temp=top, pos=0 and found=0

3.Check whether temp==NULL if true, display stack is empty and return

4.Read the element to be searched in ‘val’

5.While temp!=NULL is true go to step6 otherwise go to step10

6.Check whether temp->data==val is true go to step7 otherwise go to step8

7.Display the position of val and set found=1

8.Increment value of pos by 1

9.Set temp=temp->link

10.If !found is true display value does not exist

11.Stop

main()

1.Start

2.Perform the given operations until choice!=6 using do while loop

3.Read choice

4. Check the value of choice

5. When choice==1

   Call push()

```
void display()
{

  struct node *temp=top;

  if(top == NULL){
      printf("\n NO ELEMENTS");
      return;
  }

  printf("\n ELEMENTS IN STACK ARE: \n");
  while(temp != NULL){
         printf("%d ",temp->data);
         temp=temp->link;
  }
}


void peek()
{
  struct node*temp=top;

  if(top==NULL){
     printf("\n STACK UNEDR FLOW");
     return;
  }
  printf("Top Element is %d",temp->data);

}
```

Break

6. When choice==2

Call pop()

Break

7. When choice==3

Call peek()

Break

8. When choice==4

Call display()

Break

9. When choice==5

Call search()

Break

10. When choice==6

Display exit

Exit(0)

Break

11.default:

Display enter valid choice

12.stop

OUTPUT:

*****STACK*****

1->Push()

2->Pop()

3->Peek()

4->Display()

5->Search()

```
void search()
{
   struct node *temp=top;

   int key,found=0;

   if(top == NULL){
       printf("\n STACK UNDERFLOW \n");
       return;
   }

   printf("\n ENTER THE ELEMENT TO SEARCH \n");
   scanf("%d",&key);

   while(temp!=NULL) {
       if(temp->data ==key) {
           printf("\n %d ELEMENT FOUNDED \n",temp->data);
           found=1;
       }
       temp=temp->link;
 }

   if(!found) {
      printf("\n ELEMENT NOT FOUND");
   }

}

void main()
{
  int choice;
```

6->EXIT

ENTER THE CHOICE

1

Enter the element to insert

1

1 INSERTED SUCCESFULLY

*****STACK*****

1->Push()

2->Pop()

3->Peek()

4->Display()

5->Search()

6->EXIT

ENTER THE CHOICE

1

Enter the element to insert

2

2 INSERTED SUCCESFULLY

*****STACK*****

1->Push()

2->Pop()

3->Peek()

4->Display()

5->Search()

6->EXIT

ENTER THE CHOICE

1

Enter the element to insert

```
do{
   printf("\n *****STACK*****\n");
   printf("\n 1->Push() \n 2->Pop() \n 3->Peek() \n 4->Display() \n 5->Search() \n 6->EXIT");

   printf("\n ENTER THE CHOICE \n");
  scanf("%d",&choice);

  switch(choice)  {
    case 1:  push();
             break;
    case 2:  pop();
             break;
    case  3: peek();
             break;
    case 4:  display();
             break;
    case 5:  search();
             break;
    case 6:  printf("\n EXIT \n");
             break;

   default:  printf("Enter a valid Choice");
             break;
  }

 }while(choice!=6);

}
```

3

3 INSERTED SUCCESFULLY

*****STACK*****

1->Push()

2->Pop()

3->Peek()

4->Display()

5->Search()

6->EXIT

ENTER THE CHOICE

1

Enter the element to insert

4

4 INSERTED SUCCESFULLY

*****STACK*****

1->Push()

2->Pop()

3->Peek()

4->Display()

5->Search()

6->EXIT

ENTER THE CHOICE

3

Top Element is 4

DATE: 01-11-2024

PROGRAMNO: 9.2

SINGLY LINKED QUEUE

AIM: Write a C program to implement queue operations using singly linked list.

SOURCE CODE:

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

struct node {
   int data;
   struct node* link;
};
struct node *head=NULL;



void enqueue()
{
  struct node *temp=head,*newnode;

  newnode=(struct node*)malloc(sizeof(struct node));

  if(newnode ==NULL){
     printf("\n NO SPACE AVAILABLE \n");
     return;
  }
  newnode->link=NULL;

  printf("\n ENTER THE ELEMENT TO INSERT \n");
```

ALGORITHM:

enqueue()

1. Start

2.Create newnode and allocate memory for newnode using malloc

3.Set newnode->link=NULL and temp=head

4.Check whether head==NULL if true go to step5 otherwise go to step6

5.Set head=newnode

6.While temp->link!=NULL is true go to step7 otherwise go to step8

7.set temp=temp->link

8.Read the element to be inserted

9.Set temp->link=newnode

10.Display newly inserted element

11.Stop

dequeue()

1.Start

2.Check whether head==NULL is true go to step3 otherwise go to step4

3.Display queue is empty and return

4.Set head=temp

5.Set head=temp->link

6.Display deleted element

7. Free temp

8.Stop

```
  scanf("%d",&newnode->data);

  if(head==NULL){
      head=newnode;
  }
  else{
      while(temp->link!=NULL) {
            temp=temp->link;
       }
      temp->link=newnode;
  }
  printf("\n  %d ELEMENT INSERTED SUCCESSFULLY \n",newnode->data);

}


void dequeue()
{
  struct node *temp=head;

  if(head==NULL) {
      printf("\n NO ELEMENTS \n");
       return;
  }
  printf("\n %d IS DELETED \n",temp->data);
  head=temp->link;
  free(temp);
}
```

display()

1.Start

2.Set temp=head

3.Check whether temp==NULL is true display queue is empty and return

4.While temp!=NULL is true go to step5 otherwise go to step7

5.Display temp->data

6.Set temp=temp->link

7.Stop

search()

1.Start

2.Set temp=head, pos=0 and found=0

3.Check whether head==NULL if true, display queue is empty and return

4.Read the element to be searched in ‘val’

5.While temp!=NULL is true go to step6 otherwise go to step10

6.Check whether temp->data==val is true go to step7 otherwise go to step8

7.Display the position of val and set found=1

8.Increment value of pos by 1

9.Set temp=temp->link

10.If !found is true display value does not exist

11.Stop

main()

1.Start

2.Perform the given operations until choice!=5 using do while loop

3.Read choice

4.Read the value of choice

5. When choice==1

```
void display()
{
  struct node *temp=head;

  if(head == NULL) {
      printf("\n NO ELEMENTS");
      return;
  }

  printf("\n ELEMENTS IN QUEUE ARE \n");
  while(temp != NULL) {
         printf("%d ",temp->data);
         temp=temp->link;
  }
}

void search()
{
  struct node *temp=head;
  int key,pos=0,found=0;

  if(head==NULL){
      printf("\n QUEUE EMPTY\n");
      return;
  }

  printf("\n ENTER THE ELEMENT TO SEARCH \n");
  scanf("%d",&key);

  while(temp!=NULL){
```

Call enqueue()

Break

6. when choice==2

Call dequeue()

Break

7. When choice==3

Call display()

Break

8. when choice==4

Call search()

Break

10.When choice==5

Display exit

Exit(0)

Break

11.default:

Display enter valid choice

12.stop

OUTPUT:

*****QUEUE*****

1->Enqueue()

2->Dequeue()

3->Display()

4->Search()

5->EXIT

ENTER THE CHOICE

1

```
    if(temp->data == key){
        printf("\n %d ELEMENT FOUND AT %d \n",temp->data,pos);
        found=1;
    }
    temp=temp->link;
    pos++;
  }

  if(!found){
      printf("\n ELEMENT NOT FOUND");
  }

}



void main()
{
  int choice;

  do{
      printf("\n *****QUEUE*****\n");
      printf("\n 1->Enqueue() \n 2->Dequeue() \n 3->Display() \n 4->Search() \n 5->EXIT");

      printf("\n ENTER THE CHOICE \n");
      scanf("%d",&choice);

      switch(choice) {
            case 1: enqueue();
                        break;
```

ENTER THE ELEMENT TO INSERT

1

1 ELEMENT INSERTED SUCCESSFULLY

*****QUEUE*****

1->Enqueue()

2->Dequeue()

3->Display()

4->Search()

5->EXIT

ENTER THE CHOICE

1

ENTER THE ELEMENT TO INSERT

2

2 ELEMENT INSERTED SUCCESSFULLY

*****QUEUE*****

1->Enqueue()

2->Dequeue()

3->Display()

4->Search()

5->EXIT

ENTER THE CHOICE

1

ENTER THE ELEMENT TO INSERT

3

3 ELEMENT INSERTED SUCCESSFULLY

```
void insertLast()
{
	struct node *newnode,*temp = head;
	newnode=(struct node*)malloc(sizeof(struct node));

	if(newnode == NULL)
	{
		printf("\n Error: No space available for a new node.\n");
		return;
	}

	newnode->Llink = NULL;
	newnode->Rlink = NULL;

	printf("\n Enter the element to insert");
	scanf("%d",&newnode->data);

	if(head == NULL)
	{
		head = newnode;
	}
	else
	{
		while(temp->Rlink != NULL)
		{
			temp=temp->Rlink;
		}

		newnode->Llink = temp;
		temp->Rlink = newnode;

	}

	printf("%d inserted succesfully",newnode->data);

}


void display()
{
	struct node *temp = head;

	if(head == NULL)
	{
		printf("\n NO ELEMENTS IN LIST ");
		return;
	}
	else
	{
		printf("\n ** ELEMENTS IN LIST **  \n");
		while(temp != NULL)
		{
			printf("%d ",temp->data);
			temp = temp->Rlink;
		}
	}

}
```

ALGORITHM:

insert First()

1: Start

2: Create a new node and allocate memory for it.

3: Check if the list is empty (head == NULL), if true set head to the new node.

4: Otherwise:

- Set the Rlink of the new node to point to the current head.
- Set the Llink of the current head to point to the new node.

5: Update head to point to the new node.

6: Print a success message indicating that the element was inserted successfully.

7: Stop

insert Last ()

1: Start

2: Create a new node and allocate memory for it.

3: Check if the list is empty (head == NULL), if true set head to the new node.

4: Otherwise:

- Traverse the list until reaching the last node (where Rlink == NULL).

5: Set the Llink of the new node to the last node and the Rlink of the last node to the new node.

6: Print a success message indicating that the element was inserted successfully.

7: Stop

display ()

1: Start

2: Check if the list is empty (head == NULL):

- If the list is empty, print the message: "No elements in list" and return.

3: Otherwise :

- Print the message: "Elements in List" to indicate the start of the list.

4: Traverse the list starting from the head node:

- Initialize a temporary pointer (temp) to the head node.
- Traverse through the list using a while loop:
- Print the data of the current node (temp->data).
- Move the temporary pointer to the next node (temp = temp->Rlink).

5: Continue this process until the entire list has been displayed (when temp == NULL).

6: Stop

```
void insertLocation()
{

        struct node *newnode,*temp = head,*nxt;
        int key;
        newnode=(struct node*)malloc(sizeof(struct node));

        if(newnode == NULL)
        {
            printf("\n Error: No space available for a new node.\n");
            return;
        }

        if(head == NULL)
        {
            printf("\n List is empty \n");
        }
        else
        {
            printf("\n Enter the key were after you want to insert the element \n");
            scanf("%d",&key);

            while(temp != NULL  &&  temp->data != key)
            {
                temp = temp->Rlink;
            }

            if(temp == NULL)
            {
                printf("\n NO ELEMENT FOUND \n");
                return;
            }

            printf("\n enter the element to insert \n");
            scanf("%d",&newnode->data);

            if(temp->Rlink == NULL)
            {
                 newnode->Llink = temp;
                 newnode->Rlink = NULL;
                 temp->Rlink = newnode;
            }
            else
            {
                 nxt = temp->Rlink;
                 newnode->Llink = temp;
                 newnode->Rlink = nxt;
                 temp->Rlink = newnode;
                 nxt->Llink = newnode;
            }

           printf("%d inserted succesfully",newnode->data);

        }

}
```

insert Location ()

1: Start

2: Create a new node and allocate memory for it. set newnode->Rlink,Llink = NULL

3: Check if the list is empty (head == NULL):
- If the list is empty, print an error message: "List is empty" and exit.

4: Traverse the list to find the node containing the specified key:
- Start from the head and traverse through the list node by node until the key is found.

5: If the key is not found:
- Print an error message: "Key not found" and exit.

6: If the key is found:

- If the key is in the first node :
  - Set the new node's Rlink to the current head.
  - Set the new node's Llink to NULL .
  - Update the head to point to the new node.

- If the key is in the middle node:
  - Set the new node's Llink to the current node.
  - Set the new node's Rlink to the next node.
  - Update the current node's Rlink to point to the new node.
  - Update the next node's Llink to point to the new node.

- If the key is in the last node:
  - Set the new node's Llink to the current node.
  - Set the new node's Rlink to NULL .
  - Update the current node's Rlink to point to the new node.

7: Print a success message: "Node inserted successfully.".

8: stop

delete First ()

1: Start

2: Check if the list is empty (head == NULL).
- If the list is empty, print "LIST EMPTY" and return.

2: Otherwise :
- Set a temporary pointer temp to the head node.
- Print the message showing the data of the deleted node (temp->data).

4: Check if the node is the only node in the list:
- If the Rlink of the head node is NULL (single node), set head = NULL (list becomes empty).

5: If the node is not the only node:
- Set the next node (nxt = temp->Rlink).
- Update head = nxt.
- Set nxt->Llink = NULL.

6: Free the memory occupied by the deleted node (free(temp)).

7: Stop

```
void deleteFirst()
{

	struct node *temp=head,*nxt;

	if(head==NULL)
	{
		printf("\n LIST EMPTY \n");
		return;
	}

	 printf("\n %d is deleted",temp->data);

	if(temp->Rlink==NULL)
	{
		head=NULL;
	}
	else
	{
		nxt=temp->Rlink;
		head=nxt;
		nxt->Llink=NULL;
	}

	free(temp);

}


void deleteLast()
{
	 struct node *temp = head,*nxt;

	if(head == NULL)
	{
		printf("\n LIST IS EMPTY \n");
		return;
	}

	if(temp->Rlink == NULL)
	{
		 printf("\n %d is deleted",temp->data);
		 head=NULL;
	}
	else
	{
		while(temp->Rlink != NULL)
		{
			temp = temp->Rlink;
		}

		printf("\n %d is deleted",temp->data);
		nxt = temp->Llink;
		nxt->Rlink = NULL;
	}

	free(temp);

}
```

delete Last ()

1: Start

2: Check if the list is empty (head == NULL).
- If the list is empty, print "LIST IS EMPTY" and return.

3: If the list has only one node:
- Set a temporary pointer temp to the head node.
- Print the message showing the data of the deleted node (temp->data).
- Set the head to NULL .

4: If the list has more than one node:
- Traverse the list until the last node (temp->Rlink == NULL).
- Print the message showing the data of the deleted node (temp->data).
- Set the previous node (nxt = temp->Llink).
- Update the Rlink of the previous node to NULL (nxt->Rlink = NULL).

5: Free the memory occupied by the deleted node (free(temp)).

6: Stop

delete Location ()

1: Start

2: Check if the list is empty (head == NULL).
- If empty, print "LIST IS EMPTY" and return.

3: Traverse the List
- Prompt the user to enter the key of the node to delete.
- Traverse the list to find the node containing the specified key.
- If the key is not found, print "NO ELEMENT FOUND" and return.

4: Handle Deletion Based on Node Type
- If the node is the only node
  - Set head = temp->Rlink.
  - If head != NULL, set head->Llink = NULL.
- If the node is the last node:
  - Set temp->Llink->Rlink = NULL.
- If the node is a middle node:
  - Set prev = temp->Llink and next = temp->Rlink.
  - Update prev->Rlink = next and next->Llink = prev.

5: Print Success Message
- Print the message: "{key} Deleted successfully".

6: Free Memory
- Free the memory occupied by the node: free(temp).

7: Stop

```
void deleteLocation()
{
        struct node *temp = head, *prev, *next;
        int key;

        if (head == NULL)
        {
                printf("\n LIST IS EMPTY \n");
                return;
        }

         printf("\n Enter the key which you want to delete: \n");
         scanf("%d", &key);

        while (temp != NULL && temp->data != key)
        {
                temp = temp->Rlink;
        }

        if (temp == NULL)
        {
                printf("\n NO ELEMENT FOUND \n");
                return;
        }

        if (temp->Llink == NULL)
        {
                head = temp->Rlink;

                if (head != NULL)
                {
                     head->Llink = NULL;
                }
        }
        else if (temp->Rlink == NULL)
        {
                 temp->Llink->Rlink = NULL;
        }
        else
        {
                prev = temp->Llink;
                next = temp->Rlink;
                prev->Rlink = next;
                next->Llink = prev;
        }

    printf("%d Deleted successfully\n", temp->data);

    free(temp);

}
```

search ()

1: Start

2. Check if the list is empty (head == NULL).

- If empty, print "LIST EMPTY" and return.

3: Search the List

- Read the key from the user to search .
- Traverse the list, starting from the head:
- For each node, compare its data with the given key.
- If the key is found, print the message: "{key} is found at location {position}" and set the found flag to 1.
- Keep track of the position as you traverse the list.

3: Handle Key Not Found

- If the found flag is still 0 after traversing the list, print: "ELEMENT NOT FOUND".

4: Stop

main ()

1: Start

2: Declare a variable choice to store the user's input.

3: Display Menu

- Display the options for the user:

4: Input User Choice

- Read the user choice .

5: Execute Based on Choice

- Use a switch statement to perform the appropriate action based on the value of choice:
    - Case 1: Call insertFirst() .
    - Case 2: Call insertLast() .
    - Case 3: Call insertLocation() .
    - Case 4: Call deleteFirst() .
    - Case 5: Call deleteLast() .
    - Case 6: Call deleteLocation() .
    - Case 7: Call search() .
    - Case 8: Call display() .
    - Case 9: Print "EXIT" and terminate the program.
    - Default Case: If the user enters an invalid choice, print an error message.

6: Repeat Until Exit

- Repeat Steps 2 to 4 until the user selects option 9 to exit the program.

7: Stop

```
void search()
{
        int key, pos = 0, found = 0;
        struct node *temp = head;

        if(head == NULL)
        {
              printf("\n LIST EMPTY \n");
              return;
        }

        printf("\n Enter the key to search");
        scanf("%d",&key);

        while(temp != NULL)
        {
              if(temp->data == key)
              {
                    printf("%d is found at location %d",temp->data,pos);
                    found = 1;
              }
              temp = temp->Rlink;
              pos++;
        }

        if(!found)
        {
              printf("\n ELEMENT NOT FOUND \n");
        }

}




void main()
{
        int choice;

        do
        {


              printf("\n*****DOUBLY LINKEDLIST ****\n");
              printf("\n 1-> insert First  \n  2->insert Last  \n  3->insert Location  \n  4->delete First
              \n  5->delete Last  \n  6->delete Location  \n 7-> Search  \n  8->Display  \n  9->Exit  \n");

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

               switch(choice)
              {
                     case 1: insertFirst();
                             break;

                     case 2: insertLast();
                             break;
```

OUTPUT :

***** DOUBLY LINKED LIST MENU *****
1 -> Insert First
2 -> Insert Last
3 -> Insert Location
4 -> Delete First
5 -> Delete Last
6 -> Delete Location
7 -> Search
8 -> Display
9 -> Exit

Enter your choice: 1
Enter the element to insert : 10
10 inserted successfully .

Enter your choice: 2
Enter the element to insert : 20
20 inserted successfully .

Enter your choice: 3
Enter the key after which you want to insert: 10
Enter the element to insert: 15
15 inserted successfully .

Enter your choice: 4
10 is deleted .

Enter your choice: 5
20 is deleted .

Enter your choice: 6
Enter the key to delete: 15
15 deleted successfully.

Enter your choice: 7
Enter the key to search: 10
10 found at position 0.

Enter your choice: 8
** ELEMENTS IN THE LIST **
10 15

Enter your choice: 9
EXIT

```
 printf("\n INTERSECTION {");
 for(i=0;i<5;i++) {
       if(ints[i]==1) {
          printf("%d ",U[i]);
       }
 }
 printf("} \n");

//Complement of A
printf("Complement of A is bit representation is=");
for(i=0;i<5;i++){
      compA[i]=1-A[i];
      printf("%d",compA[i]);
 }
 printf("\n A COMPLIMENT {");
 for(i=0;i<5;i++){
       if(compA[i]==1){
          printf("%d ",U[i]);
       }
 }
 printf("} \n");

//Complement of B
 printf("Complement of B is bit representation is=");
 for(i=0;i<5;i++){
       compB[i]=1-B[i];
        printf("%d",compA[i]);
 }
 printf("\n B COMPLIMENT {");
 for(i=0;i<5;i++){
```

ALGORITHM :

1. Start.

2. Declare sets U[5],A[5],B[5],uni[5],ints[5],diffA[5],diffB[5],compA[5],compB[5].

3. Check if i<5, if true goto step 4 otherwise goto step 5.
4. If A[i]==1, display U[i].

5. Check if i<5, if true goto step 6 otherwise goto step 7.
6. If B[i]==1, true display U[i].

7. Check if i<5, if true goto step 8 otherwise goto step 10.
8. Perform uni[i]=A[i]|B[i].
9. Display uni[i].

10. Check if i<5, if true goto step 11, otherwise goto step 12.
11. If uni[i]==1 true,display U[i].

12. Check if I<5, if true goto step 13, otherwise goto step 15.
13. Perform ints[i]=A[i]&B[i].
14. Display ints[i].

15. Check if i<5, if true goto step 16 otherwise goto step 17.
16. If ints[i]==1,is true display[i].

17. Check whether i<5, if true goto step 18 otherwise goto step 20.
18. Perform compA[i]=1-A[i].
19. Display compA[i].

20. Check whether i<5, if true goto step 21 otherwise goto step 22.
21. If compA[i]==1, true display U[i].

22. Check whether i<5, if true goto step 23 otherwise goto step 25.
23. Perform compB[i]=1-B[i].
24. Display compB[i].

25. Check whether i<5, if true goto step 26 otherwise goto step 27.
26. If compB[i]==1, true display U[i].

27. Check whether i<5, if true goto step 28 otherwise goto step 30.
28. Perform diffA[i]=A[i]&compB[i].
29. Display diffA[i].

30. Check whether i<5, if true goto step 31 otherwise goto step 32.
31. Check whether diffA[i]==1,display U[i].

32. Check whether i<5, if true goto step 33 otherwise goto step 35.
33. Perform diffB[i]=B[i]&compA[i].
34. Display diffB[i].

35. Check whether i<5, if true goto step 36 otherwise goto step 37.
36. Check whether diffB[i]==1,true ,display U[i].

37. Stop.

```
  if(compB[i]==1){
      printf("%d ",U[i]);
  }
 }
 printf("} \n");

//Difference of A-B
 printf("Difference of A-B in bit representation is=");
 for(i=0;i<5;i++){
      diffA[i]=A[i]&compB[i];
       printf("%d",compA[i]);
 }
 printf("\n A-B {");
 for(i=0;i<5;i++){
       if(diffA[i]==1){
            printf("%d ",U[i]);
        }
 }
 printf("} \n");

//Difference of B-A
 printf("Difference of B-A in bit representation is=");
 for(i=0;i<5;i++){
       diffB[i]=B[i]&compA[i];
       printf("%d",compB[i]);
 }
 printf("\n B-A {");
 for(i=0;i<5;i++){
       if(diffB[i]==1){
        printf("%d ",U[i]);
```

OUTPUT:

UNIVERSAL SET IS {1,2,3,4,5}

SET A {1,4,5}

SET B {2,3,4}

Union of A and B in bit representation is=11111

UNION {1,2,3,4,5}

Intersection of A and B in bit representation is:00010

INTERSECTION {4}

Complement of A is bit representation is=01100

A COMPLIMENT {2,3}

Complement of B is bit representation is=01100

B COMPLIMENT {1,5}

Difference of A-B in bit representation is=01100

A-B {1 ,5}

Difference of B-A in bit representation is=10001

B-A {2,3}

DATE: 21-11-2024

PROGRAMNO: 12

DISJOINT SET OPERATIONS

AIM: Write a program to implement disjoint set and its operations.

SOURCE CODE:

```
#include <stdio.h>
#define MAX 1000
int Parent[MAX];

void initialize(int n)
{
    for(int i=0;i<n;i++)
    {
        Parent[i]=i;
    }
}

int find(int x)
{
    while(x!=Parent[x])
    {
        x=Parent[x];
    }
    return x;
}

void union_set(int x,int y)
{
    int rootX=find(x);
    int rootY=find(y);

    if(rootX!=rootY)
    {
```

ALGORITHM:

Disjoint set

1.Start
2.Define MAX=1000
3.Declare parent [MAX] array
4.Stop.

initialize(int n)

1.Start
2.Check whether i<n if true, go to step 3 otherwise go to step 4
3.Set parent[i]= i
4.Stop.

int find(int x)

1.Start
2.While x!=parent[x] is true, go to step 3 otherwise go to step 4
3.Set x=parent[x]
4.Return x
5.Stop

union_sets(int x, int y)

1.Start
2.Set int root x=find(x) and int root y=find(y)
3.If root x!=root y true, go to step 4 otherwise go to step 5
4.Set parent[root x]=root y
5.stop.

connected(int x, int y)

1.Start
2.Declare n=10
3.Call the function initialize(n)
4.Call the function union_sets(1,2), union_sets(3,4) and union_sets(2,3)
5.Display connected(1,4) and connected(1,5)
6.Stop

OUTPUT:

Are 1 and 5 connected ? No

Are 1 and 4 connected ? Yes

Date : 28-11-2024

PROGRAM NO: 13

BREADTH-FIRST SEARCH (BFS)

AIM: Write a program to implement Breadth-First Search (BFS) for graph traversal using an adjacency matrix

SOURCE CODE:

```
#include <stdio.h>
#define MAX 100
void BFS(int graph[MAX][MAX], int start, int n);
int main()
{
   int graph[MAX][MAX];
   int n, start;
   printf("Enter the number of vertices: ");
   scanf("%d", &n);
   printf("Enter the adjacency matrix:\n");
   for (int i = 0; i < n; i++) {
          for (int j = 0; j < n; j++) {
                 scanf("%d", &graph[i][j]);
          }
   }
   printf("Enter the starting vertex (0 to %d): ", n - 1);
   scanf("%d", &start);
   BFS(graph, start, n);
   return 0;
}
void BFS(int graph[MAX][MAX], int start, int n)
{
   int queue[MAX], front = 0, rear = 0;
   int visited[MAX] = {0};
   visited[start] = 1;
   queue[rear++] = start;
   printf("BFS Traversal: ");
```

ALGORITHM:

main()

1.Start
2.Declare the variables n, start, graph[MAX][MAX]
3.Read the number of vertices to variable 'n'
4.Read the adjacency matrix
5.Check whether i<n, if true go to step 6
6.Check whether j<n, if true go to step 7
7.Read the matrix to graph[i][j]
8.Read the starting vertex to variable 'start'
9.Call the function BFS(graph, start, n)
10.Stop.

BFS(int graph[MAX][MAX],int start, int n)

1.Start
2.Decalre variables front=0, rear=0, queue [MAX]
3.Set visited [MAX]={0}
4.Set visited [Start]=1
5.Set queue[rear++]=start
6.To do BFS traversal, check whether front<rear, if true, go to step7 otherwise go to step 9
7.Set current=queue[front++]
8.Display current
9.Check whether i<n if true, go to step 10 otherwise go to step 13
10.Check if graph[current][i]==1 && !visited[i] is true, go to step 11 otherwise go to step 12
11.Set visited[i]=1 and queue[rear+1]=i
12.Increment value of i by 1
13.Stop.

OUTPUT:

Enter the number of vertices: 5
Enter the adjacency matrix:

0 1 1 0 0

1 0 1 1 0

1 1 0 0 1

0 1 0 0 1

0 0 1 1 0

Enter the starting vertex (0 to 4): 0

BFS Traversal: 0 1 2 3 4

DATE: 28-11-2024

PROGRAM NO: 14

DEPTH-FIRST SEARCH(DFS)

AIM: Write a C program to implement Depth-First Search (DFS) for graph traversal using an adjacency matrix

SOURCE CODE:

```
#include<stdio.h>

void DFS(int x);

int G[10][10],visited[10],n;


void main()
{
   int i,j;
   printf("enter the number of vertices ");
   scanf("%d",&n);

   printf("enter the adjacency matrix of graph");
   for (int i=0;i<n;i++) {
      for(int j=0;j<n;j++) {
         scanf("%d",&G[i][j]);
      }
   }

   printf("DFS traversal");
   for(i=0;i<n;i++) {
      visited[i] = 0;
   }
    DFS(0);
 }
```

ALGORITHM:

main():

1.Start
2.Readthe number of vertices to variable ‘n’
3.Set i=0 and check whether i<n is True, goto Step4
4.Set j=0 and check whether j<n is True, go to Step 5
5.Read the value G[i][j]
6.Set i=0 and and check whether i<n is True, go to Step 7
7.Set visited[i]=0
8.Call DFS(0)
9.Stop

DFS(int i):

1.Start
2.Display value of i
3.Set visited[i]=1
4.Set j=0 and check whether j<n is true, go to Step 5
5.Check whether!visited[j] and G[i][j]==1 is True,call DFS[j]
6.Stop

OUTPUT :

enter the number of vertices 4

enter the adjacency matrix of graph

0 1 1 0
1 0 1 1
1 1 0 1
0 1 1 0

DFS traversal
0 1 2 3

DATE: 05-12-2024

PROGRAMNO: 15

TOPOLOGICAL SORT

AIM: Write a C program for performing topological sort .

SOURCE CODE:

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

#define MAX 100

int adj[MAX][MAX];
int visited[MAX];
int stack[MAX];
int top = -1;

void addEdge(int u, int v)
{
    adj[u][v] = 1;
}

void dfs(int v, int n)
{
    visited[v] = 1;
    for (int i = 0; i < n; i++) {
        if (adj[v][i] && !visited[i]) {
            dfs(i, n);
        }
    }

    stack[++top] = v;
}

void topologicalSort(int n)
{
    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            dfs(i, n);
        }
    }


    printf("Topological Order: ");

    while (top >= 0) {
        printf("%d ", stack[top--]);
    }
    printf("\n");
}
```

ALGORITHM:

addEdge(int u, int v)

1.Start
2.Set adj[u][v]=1
3.Stop

dfs(int v, int n)

1.Start
2.Set visited[v]=1
3.Set i=0
4.Check whether i<n, if True, go to Step 5
5.Check if adj[v][i] && !visited[i] is true, call dfs(i,n)
6.Increment i by 1, go to Step 4
7.Set stack[++top]=v
8.Stop

topologicalSort(int n):

1.Start
2.Set i=0
3.Check whether i<n is True, go to Step 4, otherwise go to Step 6
4.Check if !visited[i] is True, call dfs(i,n)
5.Increment i by 1, go to Step 3
6.Display the topological order
7.While top>=0 is True, display stack[top--]
8.Stop

main()

1.Start
2.Declare the number of vertices to variable n
3.Read the number of vertices to variable 'n'
4.Check whether i<n is True, go to Step 5
5.Check whether j<n is True, go to Step 6, otherwise go to Step 7
6.Set adj[i][j]=0
7.Set visited[i]=0
8.Read the number of edges in variable 'edges'
9.Read the edges using for loop
10.Check whether i<edges is True, go to Step11, otherwise go to Step 13
11.Read the edges to variables 'u' and 'v'
12.Call addEdges(u,v)
13.Call topologicalSort(n)
14.Stop

OUTPUT :

Enter the number of vertices: 6
Enter the number of edges: 6
Enter the edges (u v) (0-based index):
5 2
5 0
4 0
4 1
2 3
3 1

Topological Order: 5 4 2 3 1 0

DATE: 12-12-2024

PROGRAM NO: 16

KRUSKAL’S ALGORITHM

AIM: Write a program to implementing Kruskal’s algorithm

SOURCE CODE:

```
#include <stdio.h>
#include <stdlib.h>
int i, j, k, a, b, u, v, n, ne = 1;
int min, mincost = 0, cost[9][9], parent[9];
int find(int);
int uni(int, int);

void main()
{
  printf("\n\t Implementation of Kruskal's Algorithm\n");
  printf("\n Enter the no. of vertices:");
  scanf("%d", &n);
  printf("\n Enter the cost adjacency matrix:\n");
  for (i = 1; i<= n; i++) {
      for (j = 1; j <= n; j++) {
          scanf("%d", &cost[i][j]);
          if (cost[i][j] == 0) {
              cost[i][j] = 999;
          }
      }
  }

  printf("The edges of Minimum Cost Spanning Tree are\n");
  while (ne < n) {
      for (i = 1, min = 999; i<= n; i++) {
          for (j = 1; j <= n; j++) {
              if (cost[i][j] < min) {
                  min = cost[i][j];
                  a = u = i;
```

ALGORITHM:

Find(i)

1: Start

2: While parent[i] is not 0, update i to parent[i]

3: Return i

4: Stop

main()

1: Start

2: Read the number of vertices (n) from the user

3: Initialize variables: ne = 1, mincost = 0

4: Read the cost adjacency matrix and replace 0s with 999

5: Dispaly "The edges of Minimum Cost Spanning Tree are"

6: Execute the loop while ne < n

a. Find the minimum cost edge (a, b)

b. Find the root of vertex u and v using the find() function

c. If u and v are not in the same set, perform union using the uni() function

d. Print the selected edge and update the minimum cost

e. Set the cost[a][b] and cost[b][a] to 999 to mark the edge as used

7: Display "Minimum cost = mincost"

8: Stop

uni(i,j)

1: Start

2: If i is not equal to j

a. Set parent[j] to i

b. Return 1

3: Return 0

4: Stop

```
                    b = v = j;
                }
            }
        }
        u = find(u);
        v = find(v);

        if (uni(u, v)) {
          printf("%d edge (%d,%d) = %d\n", ne++, a, b, min);
          mincost += min;
        }
        cost[a][b] = cost[b][a] = 999;

    }

    printf("\n\tMinimum cost = %d\n", mincost);

}



int find(int i)
{
   while (parent[i])
        i = parent[i];
    return i;
}

int uni(int i, int j)
{
    if (i != j) {
        parent[j] = i;
        return 1;
    }

    return 0;

}
```

<u>OUTPUT</u>:

Implementation of Kruskal's Algorithm

Enter the no. of vertices:4

Enter the cost adjacency matrix:

1 2 3 4

1 0 6 999 5

2 6 0 999 5

3 10 999 0 8

The edges of Minimum Cost Spanning Tree are

1 edge (2,1) = 1

2 edge (3,2) = 2

3 edge (4,3) = 3

Minimum cost = 6

DATE: 12-12-2024

PROGRAMNO: 17

PRIM’S ALGORITHM

AIM: Write a C program to implement prim’s algorithm.

SOURCE CODE:

```
#include <stdio.h>
#include <limits.h>
#define MAX_VERTICES 10

int minkey(int key[], int mstset[], int V)
{
    int min = INT_MAX, minIndex;
    for (int v = 0; v < V; v++)
    {
        if ( !mstset[v] &&  key[v] < min)
        {
            min = key[v];
            minIndex = v;
        }
    }
    return minIndex;
}



void primsMST(int graph[MAX_VERTICES][MAX_VERTICES], int V)
{
    int parent[V];
    int key[V];
    int mstSet[V];
    for (int i = 0; i < V; i++) {
        key[i] = INT_MAX;
        mstSet[i] = 0;
```

ALGORITHM:

minkey()

1 : Initialize min = INT_MAXand minIndex = -1.

2 : Loop through all vertices v(from 0 to V−1V-1V−1):

- If mstSet[v] == 0 (vertex is not yet included in the MST) and key[v] < min:
- Set min = key[v].
- Set minIndex = v.

3 : :Return minIndex.

primMST()

1 : Initialize arrays key[], parent[], and mstSet[]:

- Set all key[] values to INT_MAX.
- Set all mstSet[] values to 0.
- Set key[0] = 0 (start with the first vertex).
- Set parent[0] = -1 (root of the MST).

2 : Repeat for V−1V-1V−1iterations:

- Call minkey() to find the vertex u with the smallest key value not yet included in mstSet.
- Mark mstSet[u] = 1 (include vertex u in the MST).
- For each vertex v (from 0 to V−1V-1V−1):
- If there is an edge between u and v (graph[u][v] > 0), and v is not in mstSet, and graph[u][v] < key[v]:
- Update key[v] = graph[u][v].
- Set parent[v] = u.

3 : Print the MST:

- For each vertex i (from 1 to V−1V-1V−1):
- Print the edge (parent[i], i) and its weight graph[i][parent[i]].

main()

1 : Read the number of vertices Vand edges E.

2 : Initialize a 2D array graph[MAX_VERTICES][MAX_VERTICES]with all values as 0.

3 : Loop through Eedges:

- Read vertices u, v, and weight w.
- Update graph[u][v] = wand graph[v][u] = w.

4 : Call primsMST(graph, V)to compute and print the MST.

5 : Stop

```
    }
    key[0] = 0;
    parent[0] = -1;

    for (int count = 0; count < V -1; count++)
    {
        int u = minkey(key, mstSet, V);
        mstSet[u] = 1;

        for (int v = 0; v < V; v++)
        {
            if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v])
            {
                key[v] = graph[u][v];
                parent[v] = u;
            }
        }
    }

    printf("Edge\tWeight\n");

    for (int i = 1; i < V; i++)
    {
        printf("%d -%d\t%d\n", parent[i], i, graph[i][parent[i]]);
    }
}



int main()
{
  int V, E;
  printf("Enter the number of vertices: ");
  scanf("%d", &V);

  int graph[MAX_VERTICES][MAX_VERTICES] = {0};

  printf("Enter the number of edges: ");
  scanf("%d", &E);

  printf("Enter the edges (u, v, w) where u and v are vertices and w is the weight:\n");
```

OUTPUT:

Enter the number of vertices: 4

Enter the number of edges: 5

Enter the edges (u, v, w) where u and v are vertices and w is the weight:

0 1 10

0 2 6

0 3 5

1 3 15

2 3 4

Edge	Weight
0 - 1	10
3 - 2	4
0 - 3	5