**PROGRAM NO-1(A)**

**AIM:**

Merge two sorted arrays and store in a third array.

**ALGORITHM:**

ENTER (a[10],n)

1. Repeat step 2 for i = 0 to (n-1)

2. Input a[i]

3. Return

DISPLAY(c[20],p)

1. Repeat step 2 for k = 0 to p-1

2. Print c[k]

3. Return

MAIN( )

1. Start

2. Input no. of elements in 1 st & 2 nd array as „n‟ & „m‟

3. Enter (a.n)

4. Enter (b,m)

5. i = j = k = 0

6. Repeat step 7 to 12 while ((i < n)&&(j < m))

7. If (a[i] >= b[j]),goto step 9

8. c[k+1] = a[i+1]

9. If a[i] = b[j] ,goto step 11

10. c[k++] = b[j++] goto step 7

11. c[k++] = a[i++]

12. j++

13. Repeat step 14 while (i<n)

14. c[k++] = a[j++]

15. Repeat step 16 while m > j

16. c[k++] = b[j++]

17. Display merged arrays as display(c;k)

18. Exit

**PROGRAM CODE:**

#include <stdio.h>

void main()

{

int array1[50], array2[50], array3[100], m, n, i, j, k = 0;

printf("\n Enter size of array Array 1: ");

scanf("%d", &m);

printf("\n Enter sorted elements of array 1: \n");

for (i = 0; i < m; i++)

{

scanf("%d", &array1[i]);

}

printf("\n Enter size of array 2: ");

scanf("%d", &n);

printf("\n Enter sorted elements of array 2: \n");

for (j = 0; j < n; j++)

{

scanf("%d", &array2[j]);

}

i = 0;

j = 0;

while (i < m && j < n)

{

if (array1[i] < array2[j])

{

array3[k] = array1[i];

i++;

}

else

{

array3[k] = array2[j];

j++;

}

k++;

}

if (i >= m)

{

while (j < n)

{

array3[k] = array2[j];

j++;

k++;

}

}

if (j >= n)

{

while (i < m)

{

array3[k] = array1[i];

i++;

k++;

}

}

printf("\n Merged array : \n");

for (i = 0; i < m + n; i++)

{

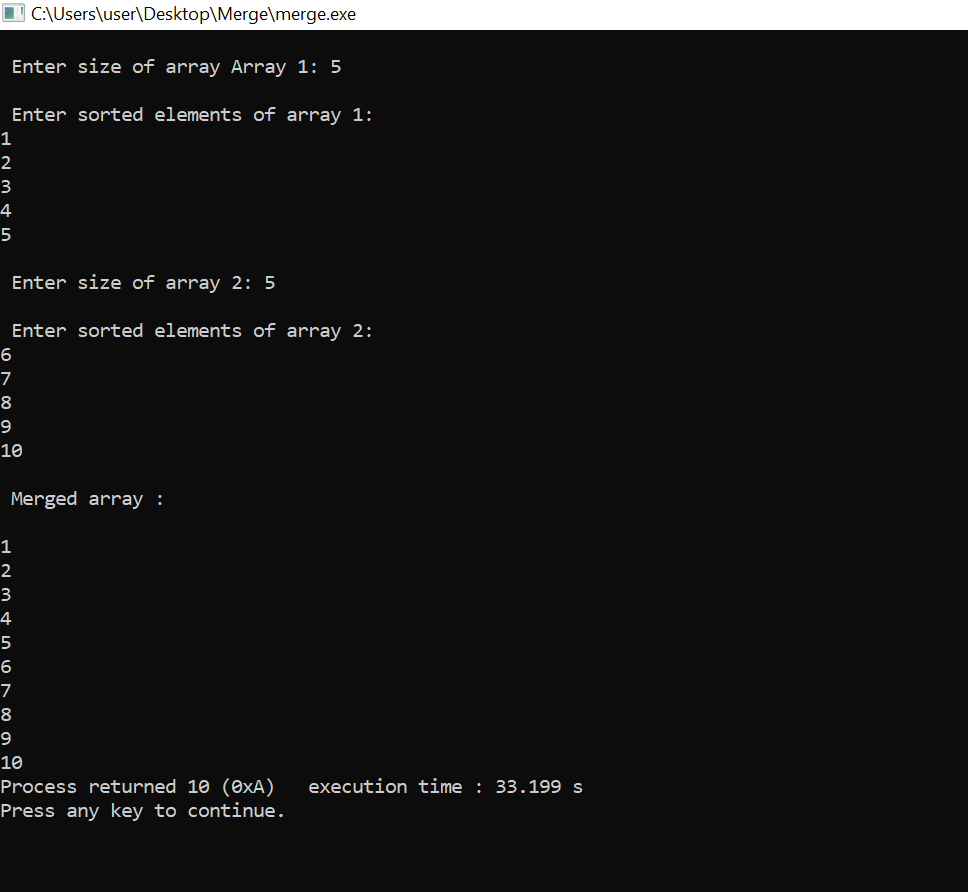
printf("\n%d", array3[i]);

}

}

**RESULT:** The above program is successfully executed and obtained the output.

**OUTPUT**



**PROGRAM N0-1(B)**

**AIM:**

Circular Queue and its operations.

**ALGORITHM:**

**CREATE**

1. t = new node

2. Enter info to be inserted

3. Read n

4. t ->info = n

5. t ->next = front

6. front = t

**INSERTION**

1. r -> next = t

2. t ->next = NULL

3. Return

**DELETION**

1. x = front

2. front = front->next

3. delnode(x)

4. Return

**DISPLAY**

1.If (front = NULL)

Print “ empty queue”

Return

Else

p = start

Repeat until (p< > NULL)

Print p->info

p = p->next

Return

**PROGRAM CODE:**

#include<stdio.h>

#define SIZE 10

void enQueue(int);

void deQueue();

void display();

void exit();

int queue[SIZE],front=-1,rear=-1;

int main()

{

int value,choice;

while(1)

{

printf("\n\n\*\*\*\*\*MENU\*\*\*\*\*\n");

printf("\n1.Insertion\n2.Deletion\n3.Display\n4.Exit");

printf("\n enter your choice:");

scanf("%d",&choice);

switch(choice)

{

case 1: printf("enter value to be inserted:");

scanf("%d",&value);

enQueue(value);

break;

case 2: deQueue();

break;

case 3: display();

break;

case 4: exit(0);

break;

default:

printf("\n wrong selection");

}

}

return 0;

}

void enQueue(int value)

{

if(rear==SIZE-1)

printf("\n Queue is full!!!Insertion not possible!!!");

else

{

if(front==-1)

front=0;

rear++;

queue[rear]=value;

printf("\n Insertion success!!!");

}

}

void deQueue()

{

if(front==-1)

printf("\n Queue is empty!!!Deletion not possible!!!");

else

{

printf("\n Deleted:%d",queue[front]);

front++;

if(front>rear)

front=rear=-1;

}

}

void display()

{

if(rear==-1)

printf("\n Queue is empty!!!");

else

{

int i;

printf("\n Queue elements are:\n");

for(i=front;i<=rear;i++)

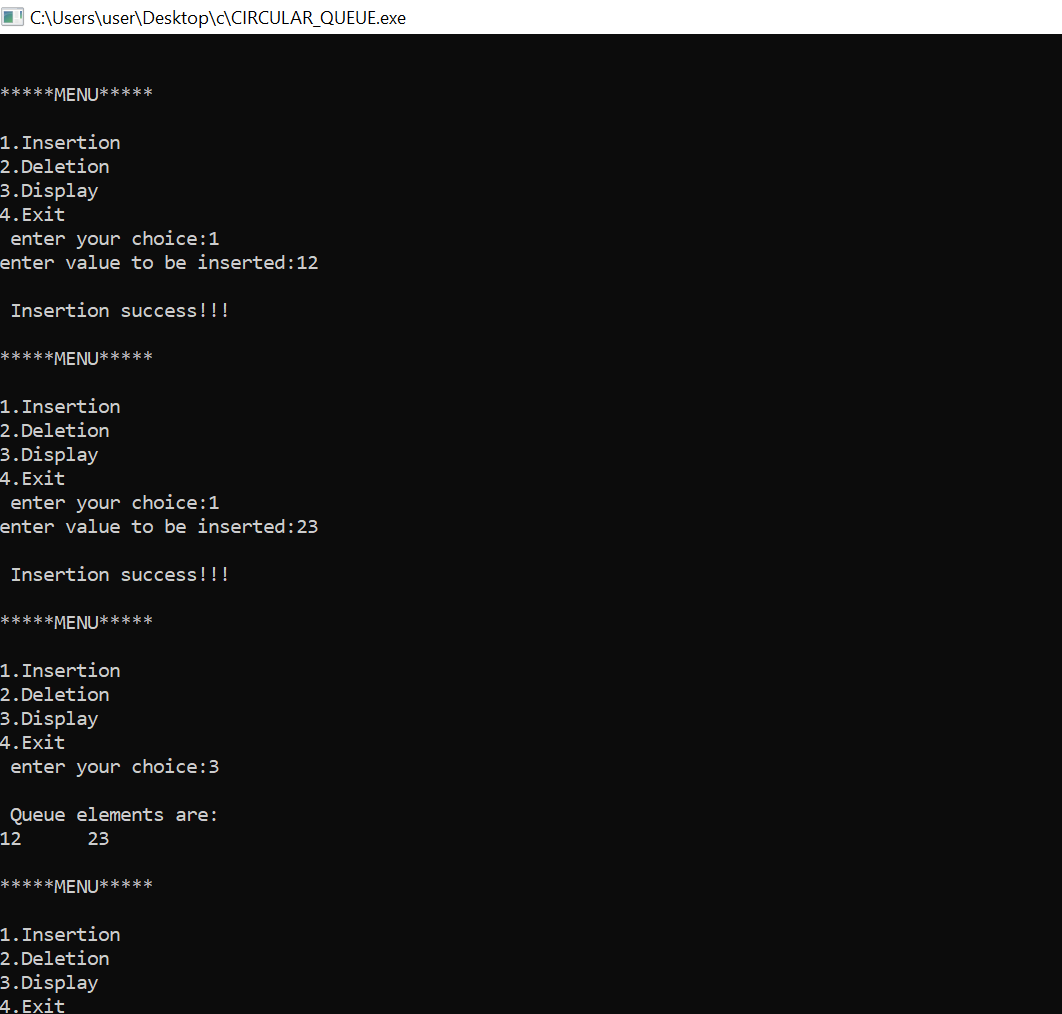
printf("%d\t",queue[i]);

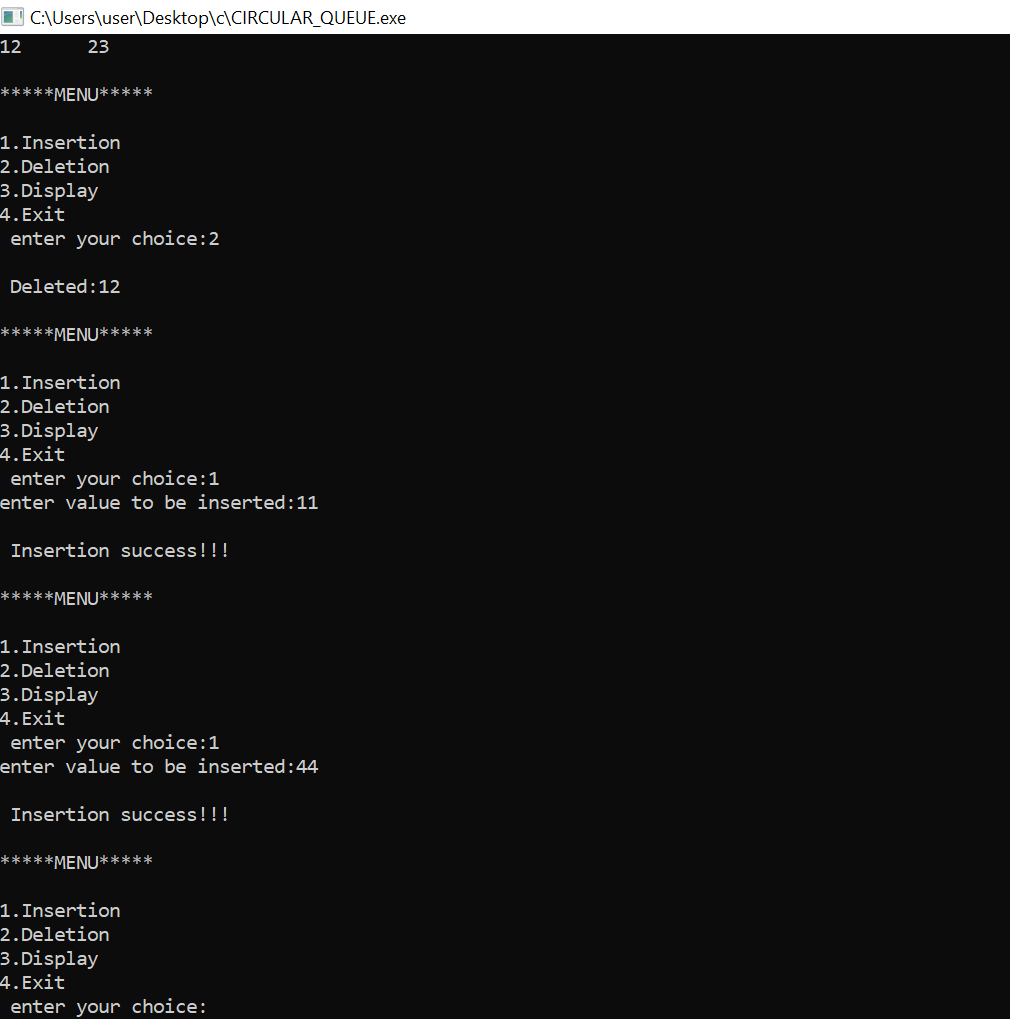
}

}

**RESULT:** The above program is successfully executed and obtained the output.

**OUTPUT**





**PROGRAM NO-1(C)**

**AIM:**

Singly Linked Stack

**ALGORITHM:**

**push(value)**

1. Create a **newNode** with given value.
2. Check whether stack is **Empty** (**top** == **NULL**)
3. If it is **Empty**, then set **newNode → next** = **NULL**.
4. If it is **Not Empty**, then set **newNode → next** = **top**.
5. Finally, set **top** = **newNode**.

**pop()**

1. Check whether **stack** is **Empty** (**top == NULL**).
2. If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function
3. If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.
4. Then set '**top** = **top → next**'.
5. Finally, delete '**temp**'. (**free(temp)**).

**display()**

* 1. Check whether stack is **Empty** (**top** == **NULL**).
  2. If it is **Empty**, then display **'Stack is Empty!!!'** and terminate the function.
  3. If it is **Not Empty**, then define a Node pointer **'temp'** and initialize with **top**.
  4. Display '**temp → data** --->' and move it to the next node. Repeat the same until **temp** reaches to the first node in the stack. (**temp → next** != **NULL**).
  5. Finally! Display '**temp → data** ---> **NULL**'.

**PROGRAM CODE:**

#include<stdio.h>

#include<stdlib.h>

void push();

void pop();

void display();

struct node

{

int value;

struct node\*next;

};

struct node\*head;

int main()

{

int choice=0;

printf("\n\*\*\*\*\*\*\*\*\*Linked Stack\*\*\*\*\*\*\*\*\*\n");

printf("\n--------------------------\n");

while(choice!=4)

{

printf("\n\n Choose one from the below options...\n");

printf("\n1.push\n2.pop\n3.display\n4.exit\n");

printf("\n Enter your choice:");

scanf("%d",&choice);

switch(choice)

{

case 1:

{

push();

break;

}

case 2:

{

pop();

break;

}

case 3:

{

display();

break;

}

case 4:

{

printf("Existing...");

break;

}

default:

{

printf("Please enter valid choice:");

}

};

}

}

void push ()

{

int value;

struct node \*ptr=(struct node\*)malloc(sizeof(struct node));

if(ptr==NULL)

{

printf("not able to push the element");

}

else

{

printf("Enter the value:");

scanf("%d",&value);

if(head==NULL)

{

ptr->value=value;

ptr->next=NULL;

head=ptr;

}

else

{

ptr->value=value;

ptr->next=head;

head=ptr;

}

printf("item pushed");

}

}

void pop()

{

int item;

struct node\*ptr;

if(head==NULL)

{

printf("Underflow");

}

else

{

item=head->value;

ptr=head;

head=head->next;

free(ptr);

printf("item popped");

}

}

void display()

{

int i;

struct node\*ptr;

ptr=head;

if(ptr==NULL)

{

printf("Stack is empty!!!\n");

}

else

{

printf("printing stack elements:\n");

while(ptr!=NULL)

{

printf("%d\n",ptr->value);

ptr=ptr->next;

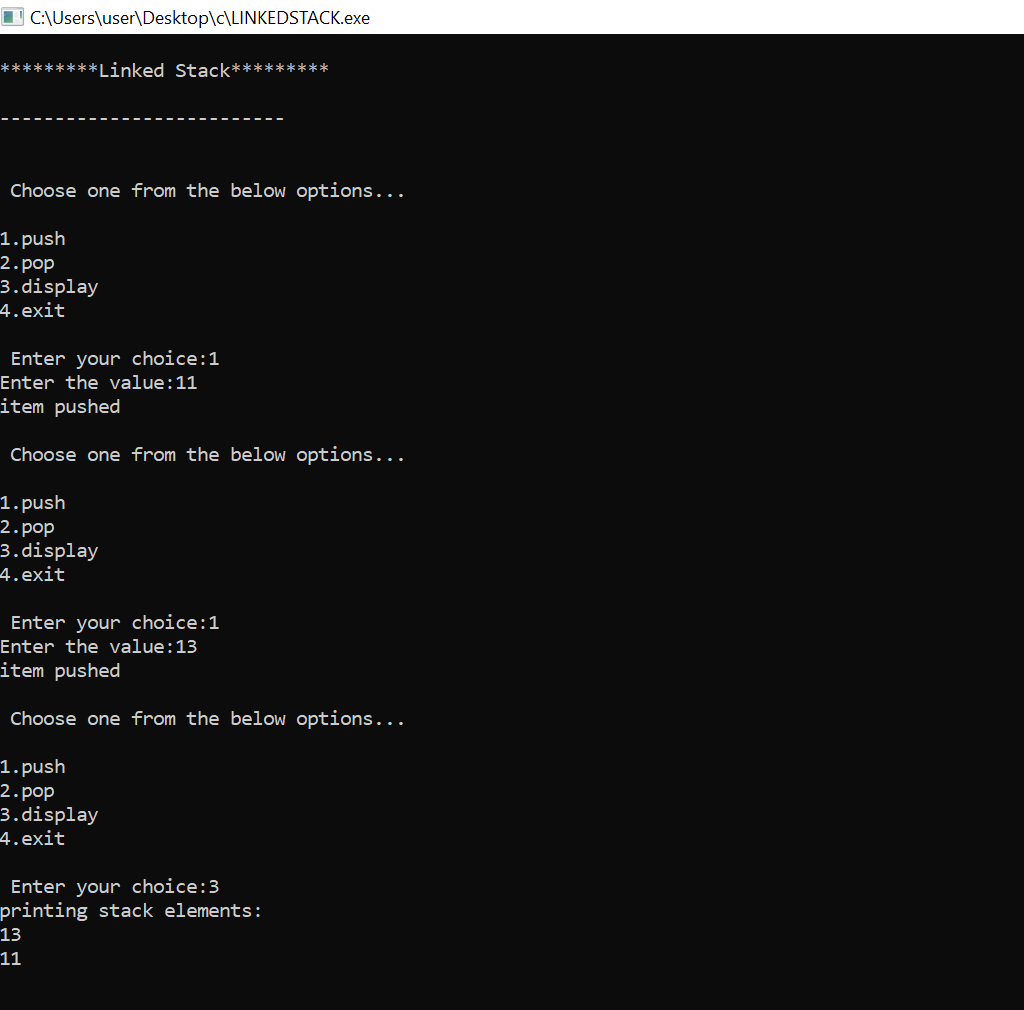
}

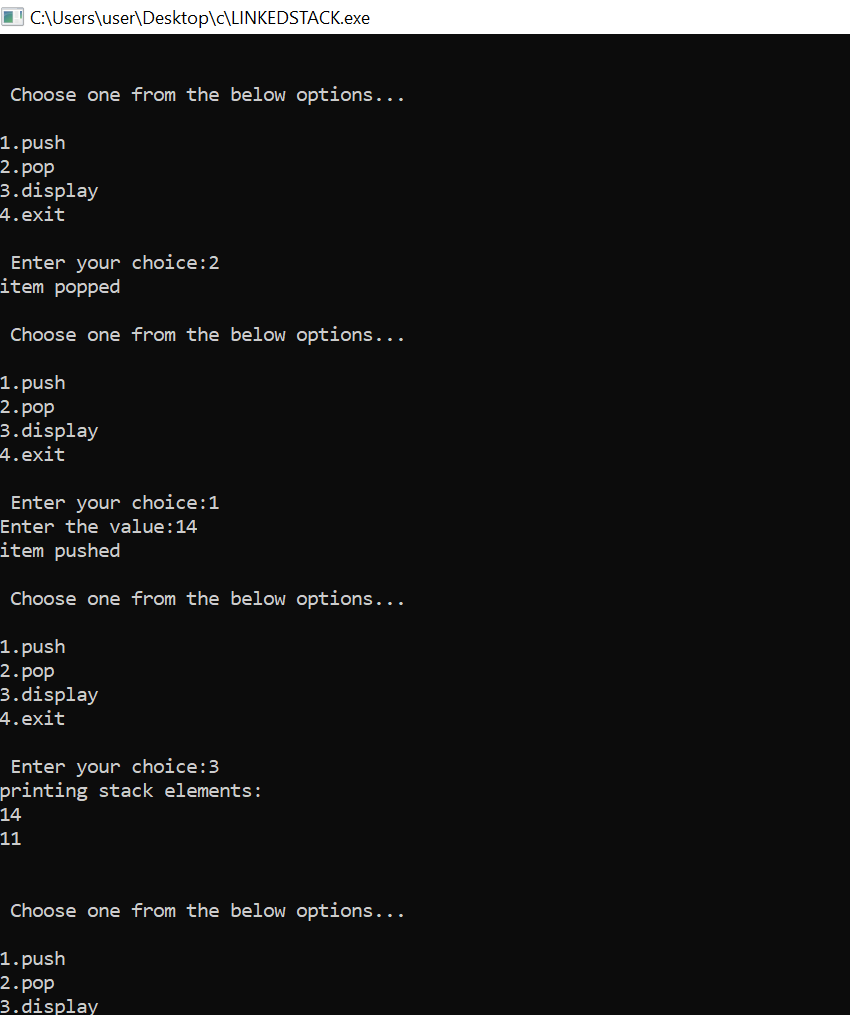
}

}

**RESULT:** The above program is successfully executed and obtained the output.

**OUTPUT**





**PROGRAM NO-1(D)**

**AIM:**

Implement doubly linked list.

**ALGORITHM:**

1. t = new node

2. Enter “the info to be inserted”

3. Read n

4. t -> info = n

5. t -> next = NULL

6. t -> prev = NULL

**INSERTION**

**BEGIN**

1.If start = NULL

start = t

1. else

t->next = NULL

t->next->prev = t

start = t

Return

**MIDDLE**

1. Print “ enter info of the node after which you want to insert”

2. Read x

3. p = start

4. Repeat while p< > NULL

If(p-> info = n)

t-> next = p->next

p-> next = t

t-> prev = p

p->next->prev = t

Return

Else

p = p->next

5. Print x not found

t-> next = NULL

p-> next = t

**DELETION**

**BEGIN**

1. p = start
2. p->next->prev = NULL
3. start = p->next

4. start = p->next

5. delnode(p)

6. Return

**MIDDLE**

1. Enter “info of the node to be deleted”

2. Read x

3. p = start

4. Repeat until p< > NULL

If(p->info = x)

p->prev->next = p->next

p->next->prev = p->prev

delnode(p)

Return

Else

p = p->next

1. Print “x not found”

**LAST**

1. P = start

2. Repeat while p< > NULL

If(p->next = NULL)

Delnode(p)

1. Return

**DISPLAY**

1. p = start

2. Repeat while p < > NULL

Print p->info

p = p->next

**PROGRAM CODE:**

#include<stdio.h>

#include<stdlib.h>

struct node

{

struct node \*prev;

struct node \*next;

int data;

};

struct node \*head;

void insert\_beginning();

void insert\_last();

void insert\_specified();

void delete\_beginning();

void delete\_last();

void delete\_specified();

void display();

void main ()

{

int choice =0;

while(1)

{

printf("\nDoubly linked list operations:\n");

printf("\n 1.Insert at beginning\n 2.Insert at last\n 3.Insert at specific position\n 4.Delete from Beginning\n 5.Delete from last\n 6.Delete the node after the given data\n 7.Display\n 8.Exit\n");

printf("\nEnter your choice:\n");

scanf("\n%d",&choice);

switch(choice)

{

case 1:

insert\_beginning();

break;

case 2:

insert\_last();

break;

case 3:

insert\_specified();

break;

case 4:

delete\_beginning();

break;

case 5:

delete\_last();

break;

case 6:

delete\_specified();

break;

case 7:

display();

break;

case 8:

exit(0);

break;

default:

printf("\n Wrong choice\n");

}

}

}

void insert\_beginning()

{

struct node \*ptr;

int item;

ptr = (struct node \*)malloc(sizeof(struct node));

if(ptr == NULL)

{

printf("\nOVERFLOW");

}

else

{

printf("\nEnter Item value:\n");

scanf("%d",&item);

if(head==NULL)

{

ptr->next = NULL;

ptr->prev=NULL;

ptr->data=item;

head=ptr;

}

else

{

ptr->data=item;

ptr->prev=NULL;

ptr->next = head;

head->prev=ptr;

head=ptr;

}

printf("\nNode inserted\n");

}

}

void insert\_last()

{

struct node \*ptr,\*temp;

int item;

ptr = (struct node \*) malloc(sizeof(struct node));

if(ptr == NULL)

{

printf("\nOVERFLOW");

}

else

{

printf("\nEnter value");

scanf("%d",&item);

ptr->data=item;

if(head == NULL)

{

ptr->next = NULL;

ptr->prev = NULL;

head = ptr;

}

else

{

temp = head;

while(temp->next!=NULL)

{

temp = temp->next;

}

temp->next = ptr;

ptr ->prev=temp;

ptr->next = NULL;

}

}

printf("\n node inserted\n");

}

void insert\_specified()

{

struct node \*ptr,\*temp;

int item,loc,i;

ptr = (struct node \*)malloc(sizeof(struct node));

if(ptr == NULL)

{

printf("\n OVERFLOW");

}

else

{

temp=head;

printf("Enter the location");

scanf("%d",&loc);

for(i=0;i<loc;i++)

{

temp = temp->next;

if(temp == NULL)

{

printf("\n There are less than %d elements", loc);

return;

}

}

printf("\nEnter value\n");

scanf("%d",&item);

ptr->data = item;

ptr->next = temp->next;

ptr -> prev = temp;

temp->next = ptr;

temp->next->prev=ptr;

printf("\n Node inserted\n");

}

}

void delete\_beginning()

{

struct node \*ptr;

if(head == NULL)

{

printf("\n Underflow\n");

}

else if(head->next == NULL)

{

head = NULL;

free(head);

printf("\n Node deleted\n");

}

else

{

ptr = head;

head = head -> next;

head -> prev = NULL;

free(ptr);

printf("\n Node deleted\n");

}

}

void delete\_last()

{

struct node \*ptr;

if(head == NULL)

{

printf("\n Underflow\n");

}

else if(head->next == NULL)

{

head = NULL;

free(head);

printf("\n Node deleted\n");

}

else

{

ptr = head;

if(ptr->next != NULL)

{

ptr = ptr -> next;

}

ptr -> prev -> next = NULL;

free(ptr);

printf("\n Node deleted\n");

}

}

void delete\_specified()

{

struct node \*ptr, \*temp;

int val;

printf("\n Enter the data after which the node is to be deleted : ");

scanf("%d", &val);

ptr = head;

while(ptr -> data != val)

ptr = ptr -> next;

if(ptr -> next == NULL)

{

printf("\nCan't delete\n");

}

else if(ptr -> next -> next == NULL)

{

ptr ->next = NULL;

}

else

{

temp = ptr -> next;

ptr -> next = temp -> next;

temp -> next -> prev = ptr;

free(temp);

printf("\n node deleted\n");

}

}

void display()

{

struct node \*ptr;

printf("\n printing values...\n");

ptr = head;

while(ptr != NULL)

{

printf("%d\n",ptr->data);

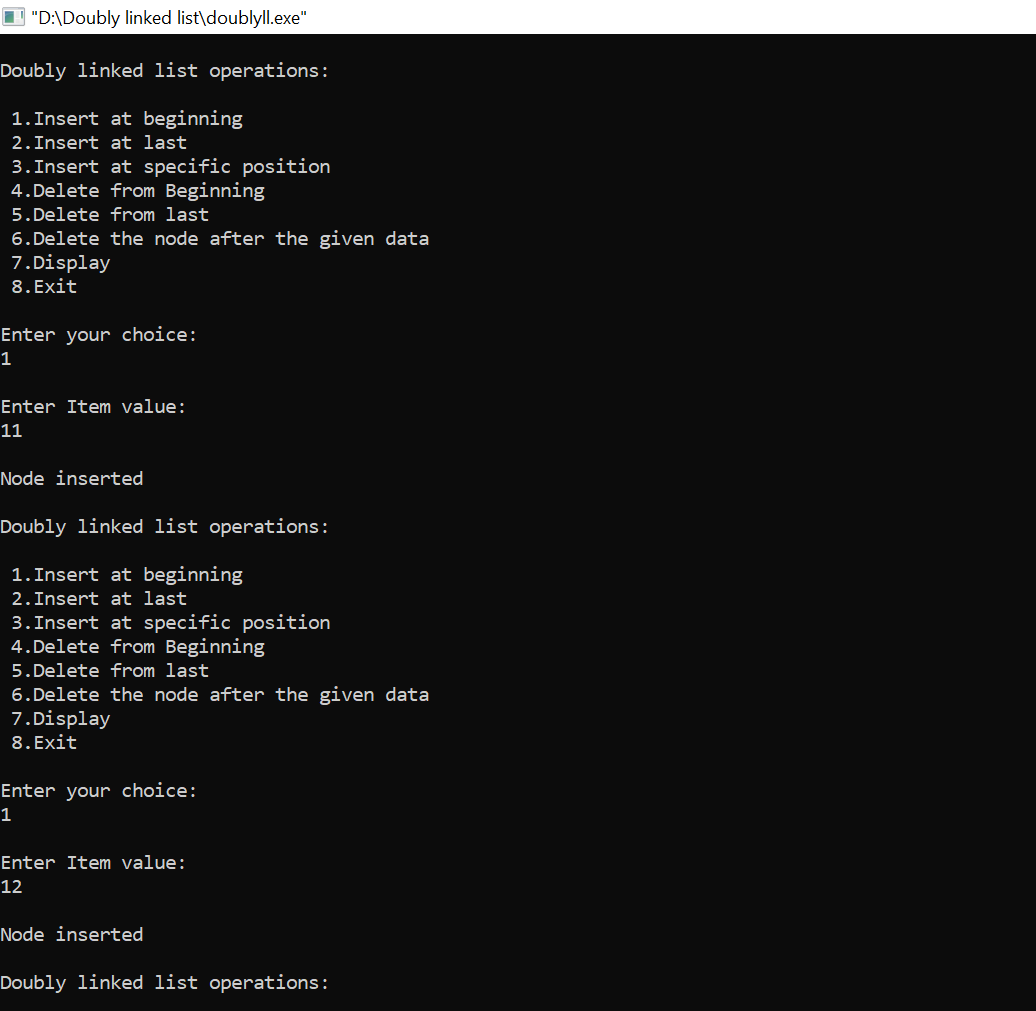
ptr=ptr->next;

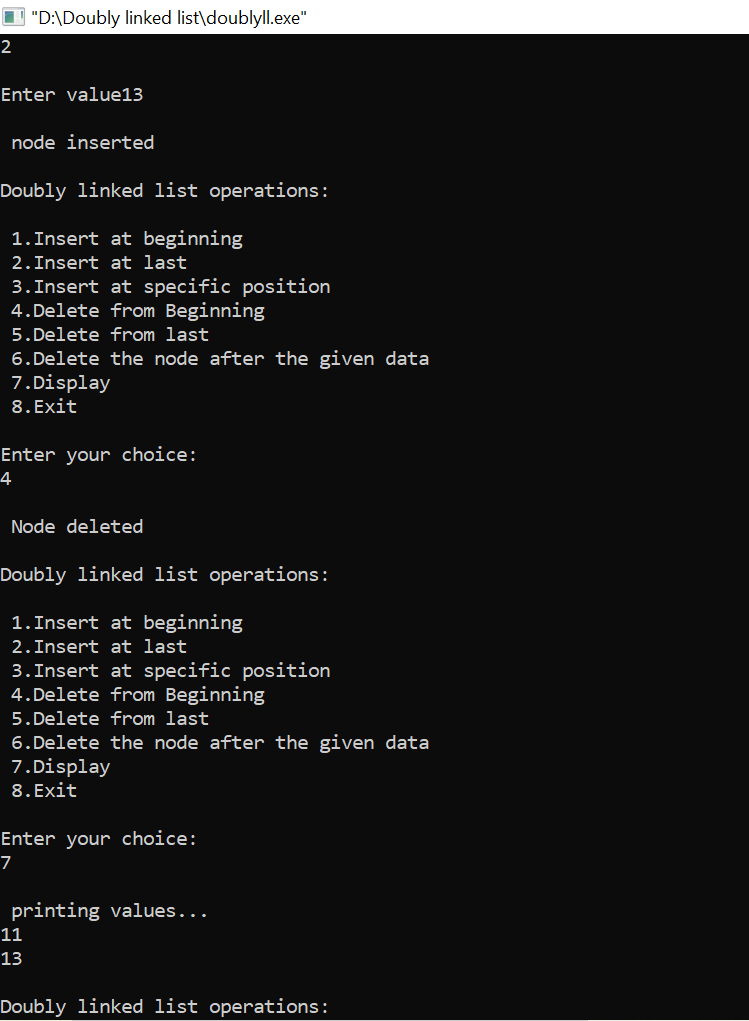
}

}

**RESULT:** The above program is successfully executed and obtained the output.

**OUTPUT**





**PROGRAM NO-1(E)**

**AIM:**

Implement binary search tree

**ALGORITHM:**

**INSERTION**

1. t = newnode info = n

2. t ->info = n

3. t ->left = t->right = NULL

4. If (root = NULL)

root = t

return

5. ptr = root

6. Repeat step 7 until ptr = NULL

7. If (ptr ->info > n)

If (ptr->left = NULL)

Ptr->left = t

Return

Else

Ptr = ptr->left

Else

If(ptr->right = NULL)

Ptr-> right = t

Return

Else

Ptr = ptr->right

**DELETION**

1.If (root = NULL)

Print “Empty tree “

Return

2. ptr = root, par = NULL

3. Repeat step 4 & 5 until (ptr->info = n or ptr = NULL)

4. par = ptr

5. If (ptr->info > n)

ptr = ptr ->left

Else

Ptr = ptr->right

6.If ptr = NULL

print “Number not present”

**PROGRAM CODE**

#include <stdio.h>

#include <stdlib.h>

struct btnode

{

int value;

struct btnode \*l;

struct btnode \*r;

}\*root = NULL, \*temp = NULL, \*t2, \*t1;

void delete1();

void insert();

void delete();

void inorder(struct btnode \*t);

void create();

void search(struct btnode \*t);

void preorder(struct btnode \*t);

void postorder(struct btnode \*t);

void search1(struct btnode \*t,int data);

int smallest(struct btnode \*t);

int largest(struct btnode \*t);

int flag = 1;

void main()

{

int ch;

printf("\nOPERATIONS ---");

printf("\n1 - Insert an element into tree\n");

printf("2 - Delete an element from the tree\n");

printf("3 - Inorder Traversal\n");

printf("4 - Preorder Traversal\n");

printf("5 - Postorder Traversal\n");

printf("6 - Exit\n");

while(1)

{

printf("\nEnter your choice : ");

scanf("%d", &ch);

switch (ch)

{

case 1:

insert();

break;

case 2:

delete();

break;

case 3:

inorder(root);

break;

case 4:

preorder(root);

break

case 5:

postorder(root);

break;

case 6:

exit(0);

default :

printf("Wrong choice, Please enter correct choice ");

break;

}

}

}

void insert()

{

create();

if (root == NULL)

root = temp;

else

search(root);

}

void create()

{

int data;

printf("Enter data of node to be inserted : ");

scanf("%d", &data);

temp = (struct btnode \*)malloc(1\*sizeof(struct btnode));

temp->value = data;

temp->l = temp->r = NULL;

}

void search(struct btnode \*t)

{

if ((temp->value > t->value) && (t->r != NULL)) /\* value more than root node value insert at right \*/

search(t->r);

else if ((temp->value > t->value) && (t->r == NULL))

t->r = temp;

else if ((temp->value < t->value) && (t->l != NULL)) /\* value less than root node value insert at left \*/

search(t->l);

else if ((temp->value < t->value) && (t->l == NULL))

t->l = temp;

}

void inorder(struct btnode \*t)

{

if (root == NULL)

{

printf("No elements in a tree to display");

return;

}

if (t->l != NULL)

inorder(t->l);

printf("%d -> ", t->value);

if (t->r != NULL)

inorder(t->r);

}

void delete()

{

int data;

if (root == NULL)

{

printf("No elements in a tree to delete");

return;

}

printf("Enter the data to be deleted : ");

scanf("%d", &data);

t1 = root;

t2 = root;

search1(root, data);

}

void preorder(struct btnode \*t)

{

if (root == NULL)

{

printf("No elements in a tree to display");

return;

}

printf("%d -> ", t->value);

if (t->l != NULL)

preorder(t->l);

if (t->r != NULL)

preorder(t->r);

}

void postorder(struct btnode \*t)

{

if (root == NULL)

{

printf("No elements in a tree to display ");

return

}

if (t->l != NULL)

postorder(t->l);

if (t->r != NULL)

postorder(t->r);

printf("%d -> ", t->value);

}

void search1(struct btnode \*t, int data)

{

if ((data>t->value))

{

t1 = t;

search1(t->r, data);

}

else if ((data < t->value))

{

t1 = t;

search1(t->l, data);

}

else if ((data==t->value))

{

delete1(t);

}

}

void delete1(struct btnode \*t)

{

int k;

if ((t->l == NULL) && (t->r == NULL))

{

if (t1->l == t)

{

t1->l = NULL;

}

else

{

t1->r = NULL;

}

t = NULL;

free(t);

return;

}

else if ((t->r == NULL))

{

if (t1 == t)

{

root = t->l;

t1 = root;

}

else if (t1->l == t)

{

t1->l = t->l;

}

else

{

t1->r = t->l

}

t = NULL;

free(t);

return;

}

else if (t->l == NULL)

{

if (t1 == t)

{

root = t->r;

t1 = root;

}

else if (t1->r == t)

t1->r = t->r;

else

t1->l = t->r;

t == NULL;

free(t);

return;

}

else if ((t->l != NULL) && (t->r != NULL))

{

t2 = root;

if (t->r != NULL)

{

k = smallest(t->r);

flag = 1;

}

else

{

k =largest(t->l);

flag = 2;

}

search1(root, k);

t->value = k;

}

}

int smallest(struct btnode \*t)

{

t2 = t;

if (t->l != NULL)

{

t2 = t;

return(smallest(t->l));

}

else

return (t->value);

}

int largest(struct btnode \*t)

{

if (t->r != NULL)

{

t2 = t;

return(largest(t->r));

}

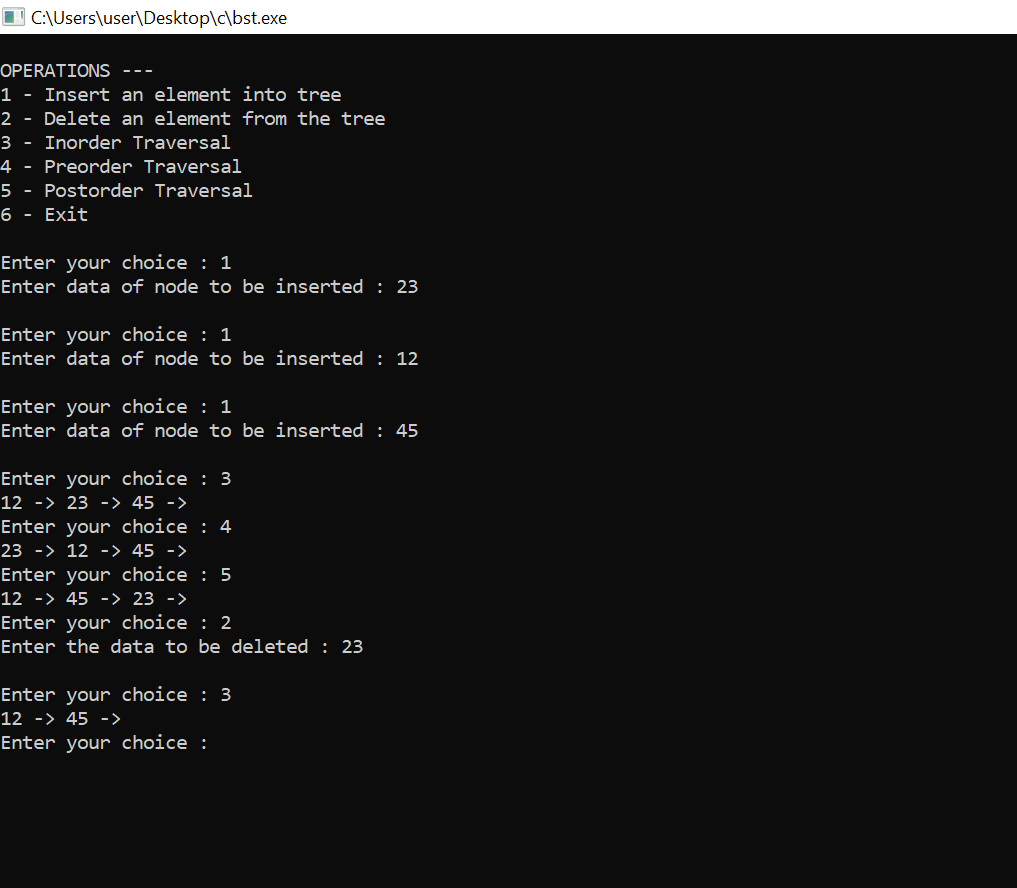
else

return(t->value);

}

**RESULT:** The above program is successfully executed and obtained the output.

**OUTPUT**



**PROGRAM NO-2**

**AIM:**

Set Data Structure and set operations (Union, Intersection and Difference) using Bit String.

**ALGORITHM:**

**Create()**

1. Set[0]=0
2. Read no:of elements as ‘n’
3. Repeat step 4 for i = 1 to n
4. Read set[i]
5. Set[0] = n

**Print()**

1. n = set[0]
2. Repeat step 3 for i = 1 to n
3. Input set[i]

**Union()**

1. Set3[0] = 0
2. n = set1[0]
3. Repeat step 4 for i = 0 to n
4. Set s[i]=set1[i]
5. Repeat step 7 for i = 1 to n
6. If(!member (set3.set3[i]))set3[++set3[0]]=set2[i]

**Member()**

1. n = set[0]
2. Repeat step 3 for i = 1 to n
3. If x = =set[i], return 1

**Intersection()**

1. Set3[0]= 0
2. n = set1[0]
3. Repeat step 4 for i=1 to n
4. If member(set2,set1[i])

set3[++set3[0]]= set1[i]

**Difference()**

1. n = set1[0]
2. set[0]=0
3. Repeat step 4 for i=1 to n
4. If(!member(set2,set1[i]))

Set3[++set3[0]]=set1[i]

**Symmdiff()**

1. n = set1[0]
2. set3[0]=0
3. Repeat step 4 for i=1 to n
4. If(!member(set2,set1[i]))

Set3[++set3[0]]=set1[i]

1. n = set2[0]
2. Repeat step 7 for i=1 to n
3. If(!member(set3,set2[i]))

Set3[++set3[a]]=set2[i]

**PROGRAM CODE:**

#define MAX 30

#include<stdio.h>

#include<conio.h>

void create(int set[]);

void print(int set[]);

void Union(int set1[],int set2[],int set3[]);

void intersection(int set1[],int set2[],int set3[]);

void difference(int set1[],int set2[],int set3[]);

void symmdiff(int set1[],int set2[],int set3[]);

int member(int set[],int x);

void main()

{ int set1[MAX],set2[MAX],set3[MAX];

int x,op;

set1[0]=set2[0]=set3[0]=0;

do

{ printf("\n 1)Create\n 2)Print\n 3)Union\n 4)Intersection\n 5)Difference");

printf("\n 6)Symmetric Difference \n 7)Quit"); printf("\nEnter Your Choice:");

scanf("%d",&op);

switch(op)

{

case 1: printf("\nCreate First Set\*\*\*\*\*\*\*");

create(set1);

printf("\nCreating Second Set\*\*\*\*\*");

create(set2);

break;

case 2: printf("\nFirst Set :\n");

print(set1);

printf("\n\nSecond Set :\n");

print(set2);

printf("\n\nThird Set :\n");

print(set3);

break;

case 3: Union(set1,set2,set3);print(set3);break;

case 4: intersection(set1,set2,set3);print(set3);break;

case 5: difference(set1,set2,set3);print(set3);break;

case 6: symmdiff(set1,set2,set3);print(set3);break;

}

printf("\n press a key............");

getch();

}while(op!=7);

}

/\*creates set[] with initial elements\*/

void create(int set[])

{ int n,i,x;

set[0]=0; /\*make it a null set\*/

printf("\n No. of elements in the set:");

scanf("%d",&n);

printf("\n enter set elements :");

for(i=1;i<=n;i++)

scanf("%d",&set[i]);

set[0]=n; //Number of elements.

}

void print(int set[])

{ int i,n;

n=set[0]; /\* number of elements in the set \*/

printf("\Members of the set :-->");

for(i=1;i<=n;i++)

printf("%d ",set[i]);

}

/\* union of set1[] and set2[] is stored in set3[]\*/

void Union(int set1[],int set2[],int set3[])

{ int i,n;

/\* copy set1[] to set3[]\*/

set3[0]=0; /\*make set3[] a null set \*/

n=set1[0]; /\* number of elements in the set\*/

//Union of set1,set2= set1 + (set2-set1)

for(i=0;i<=n;i++)

set3[i]=set1[i];

n=set2[0];

for(i=1;i<=n;i++)

if(!member(set3,set2[i])) set3[++set3[0]]=set2[i]; // insert and increment no. of elements

}

/\*function returns 1 or 0 depending on whether x belongs

to set[] or not \*/

int member(int set[],int x)

{ int i,n;

n=set[0]; /\* number of elements in the set\*/

for(i=1;i<=n;i++)

if(x==set[i])

return(1);

return(0);

}

/\*intersection of set1[] and set2[] is stored in set3[]\*/

void intersection(int set1[],int set2[],int set3[])

{

int i,n;

set3[0]=0; /\* make a NULL set\*/

n=set1[0]; /\* number of elements in the set\*/

for(i=1;i<=n;i++)

if(member(set2,set1[i])) /\* all common elements are inserted in set3[]\*/

set3[++set3[0]]=set1[i]; // insert and increment no. of elements

}

/\*difference of set1[] and set2[] is stored in set3[]\*/

void difference(int set1[],int set2[],int set3[])

{ int i,n;

n=set1[0]; /\* number of elements in the set\*/

set3[0]=0; /\*make it a null set\*/

for(i=1;i<=n;i++)

if(!member(set2,set1[i]))

set3[++set3[0]]=set1[i]; // insert and increment no. of elements

}

void symmdiff(int set1[],int set2[],int set3[])

{ int i,n;

n=set1[0]; /\* number of elements in the set\*/

set3[0]=0; /\*make it a null set\*/

//Calculate set1-set2

for(i=1;i<=n;i++)

if(!member(set2,set1[i]))

set3[++set3[0]]=set1[i]; // insert and increment no. of elements

//Calculate set2-set1

n=set2[0];

for(i=1;i<=n;i++)

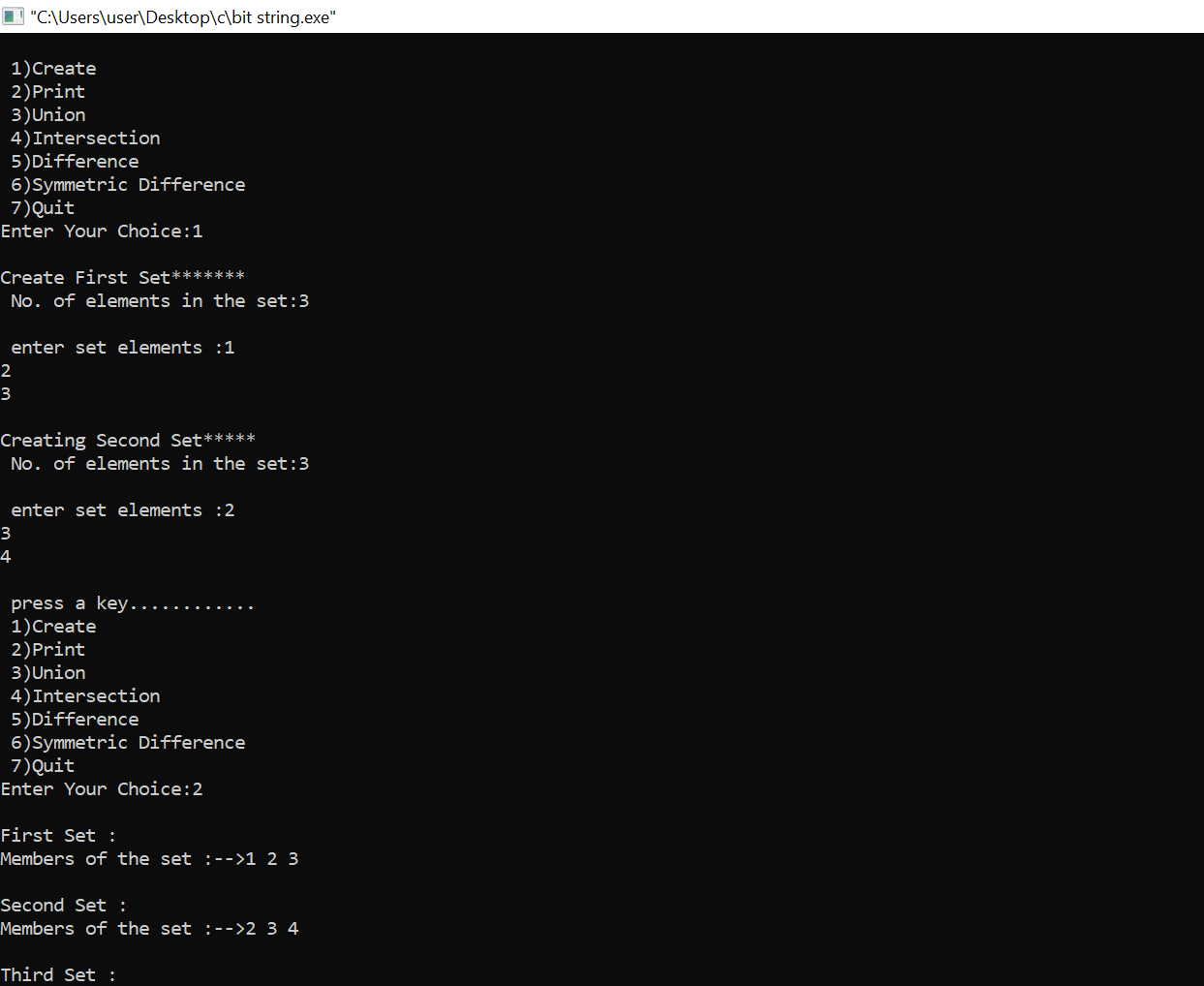
if(!member(set1,set2[i]))

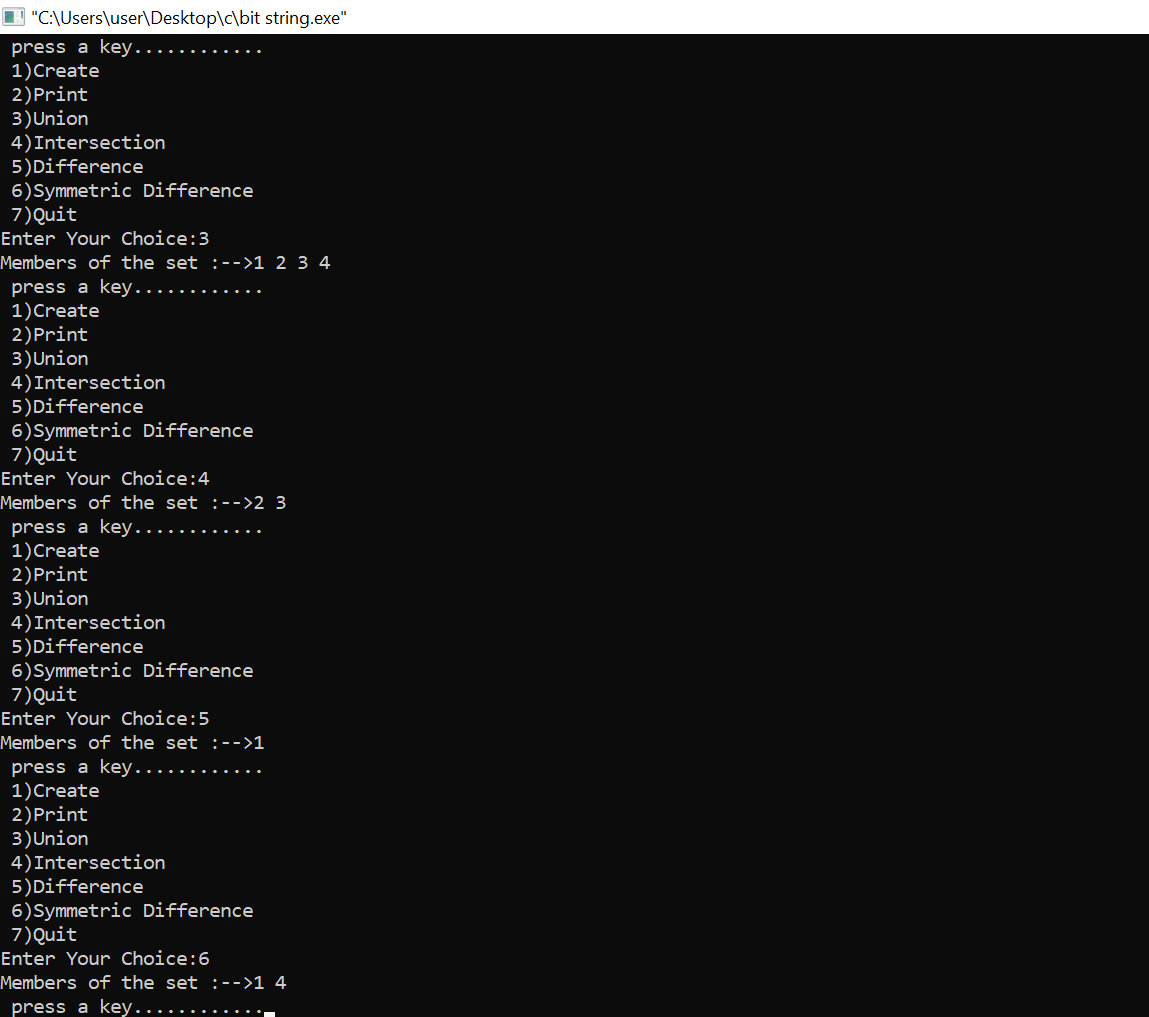
set3[++set3[0]]=set2[i]; // insert and increment no. of elements

}

**RESULT:** The above program is successfully executed and obtained the output.

**OUTPUT**





**PROGRAM NO-3**

**AIM:**

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

**ALGORITHM:**

**Makeset()**

1. Repeat step 2 for i=1 to n
2. Parent[i] = i

**Findparent()**

1. If parent[x]=x

then return x

1. Return findparent(parent[x])

**Merge()**

1. Parent[findparent(a)] = findparent(b)

**Isconnected()**

1. Return findparent(a)==findparent(b)

**PROGRAM CODE:**

#include <stdio.h>

#define MAXN 100

int parent[MAXN];

void makeset(int n)

{

for(int i=1;i<=n;i++) {

parent[i]=i;

}

}

int findparent(int x) {

if(parent[x] == x)

return x;

return findparent(parent[x]);

}

void merge(int a,int b) {

parent[findparent(a)] = findparent(b);

}

int isconnected(int a,int b) {

return findparent(a) == findparent(b);

}

int main() {

makeset(10);

merge(1,2);

merge(2,3);

merge(7,8);

merge(8,9);

printf("%d\n",isconnected(1,3));

printf("%d\n",isconnected(2,8));

printf("%d\n",isconnected(5,8));

merge(5,8);

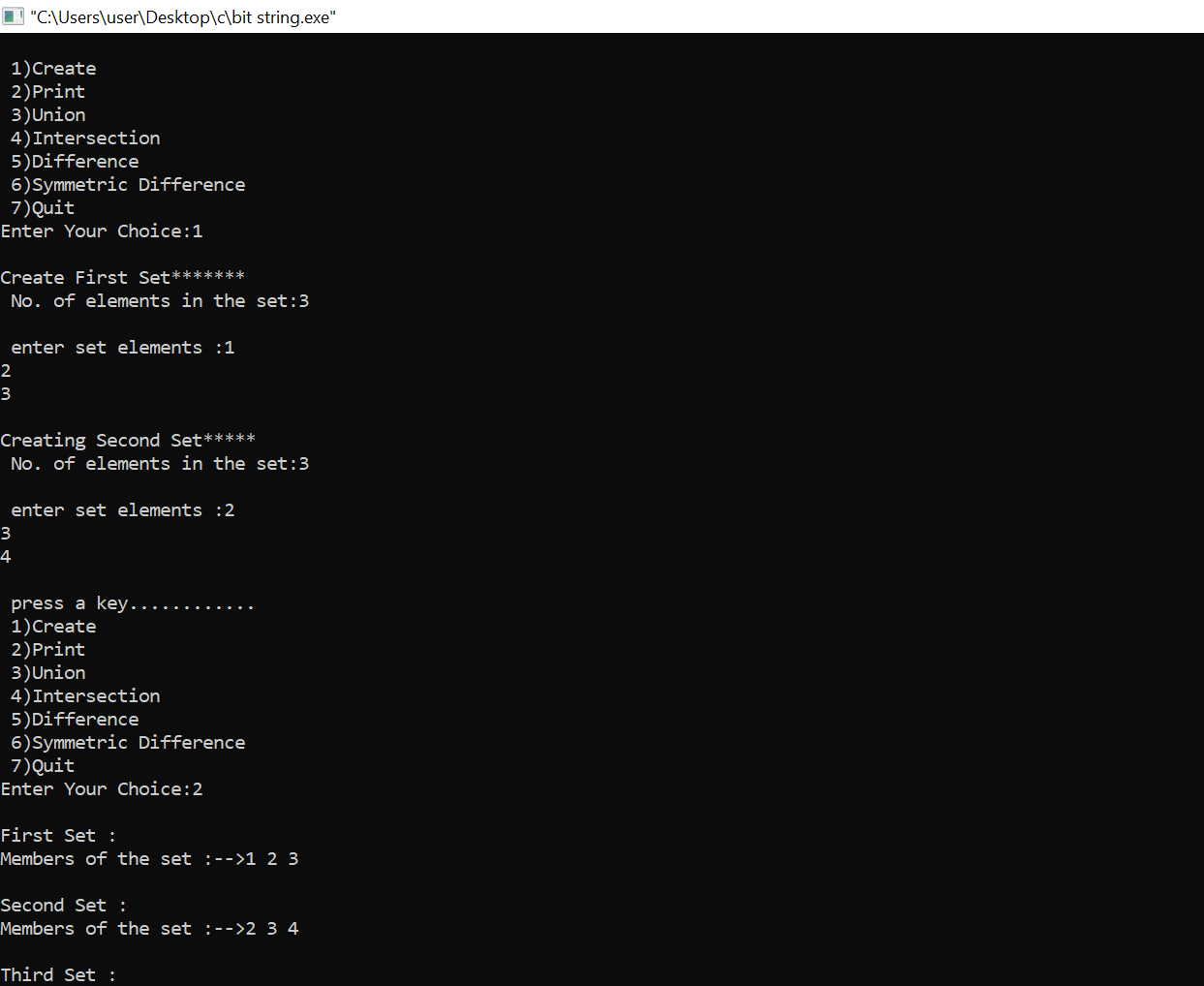
printf("%d\n",isconnected(8,5));

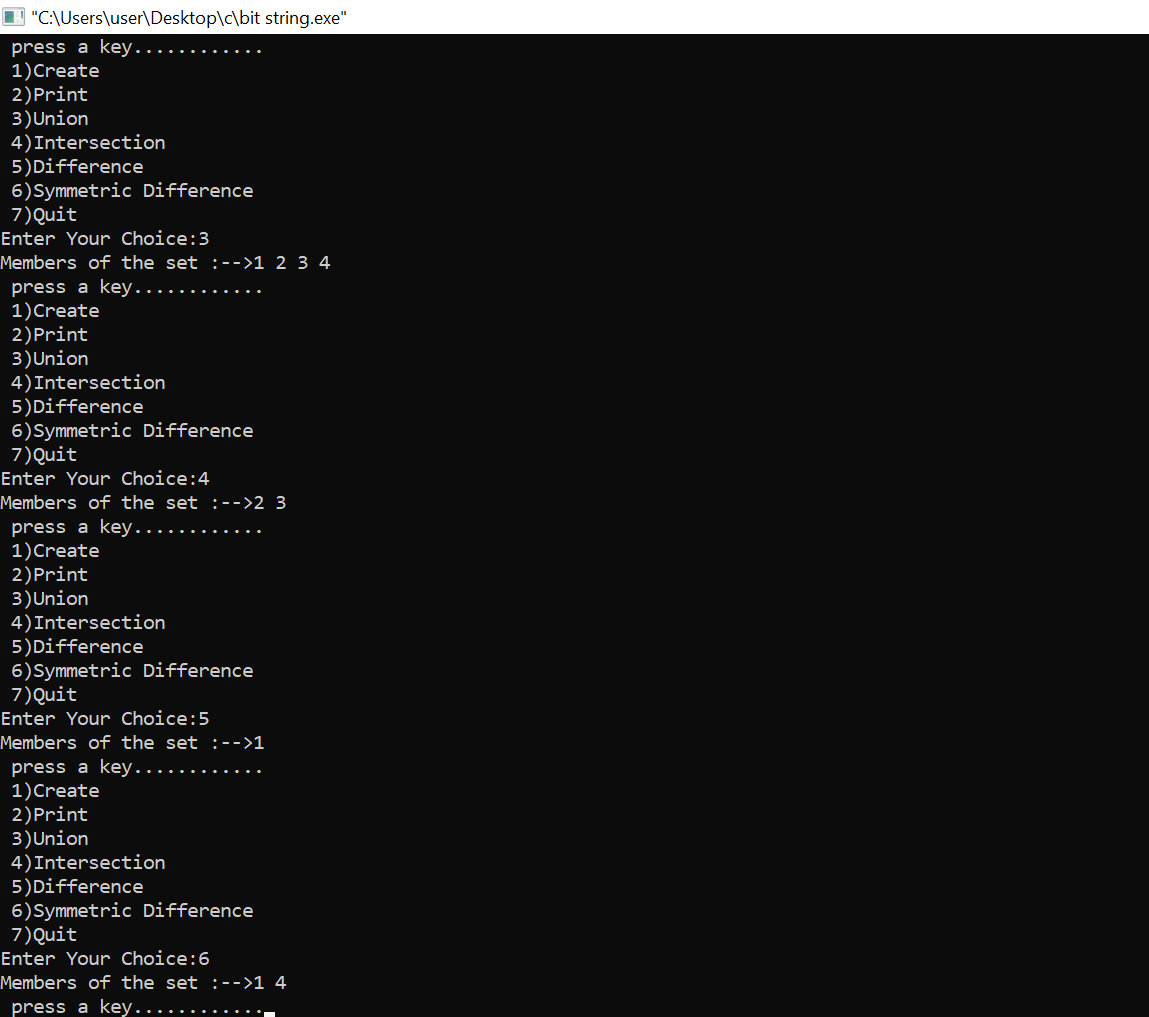
return 0;

}

**RESULT:** The above program is successfully executed and obtained the output.

**OUTPUT**





**PROGRAM NO-4**

**AIM**:

Binomial Heaps and operations (Create, Insert, Delete, Extract-min, Decrease key).

**ALGORITHM:**

BINOMIAL\_HEAP\_INSERT(H,x)

1. H’=Make\_Binomial\_Heap()

2. p(x)<-NIL

3. Child(x)<-NIL

4. Sibling[x]<-NIL

5. Degree[x]<-0

6. Head[H’]<-x

7. H<-Binomial\_heap\_UNION(H,H’)

BINOMIAL\_HEAP\_MINIMUM(H)

1. y<- NIL

2. x<-head[H]

3. min<-∞

4. while x≠NIL

5. do if key[x]<min

6. then min<-key[x]

7. y<-x

8. x<-sibling[x]

9. return y

BINOMIAL\_HEAP\_EXTRACT MIN(H)

1. find the root ‘x’ with the minimum key in the rootlist of H and remove ‘x’ from the rootlist of H. FIND\_MIN\_KEY()

2. H’<-MAKE\_BINOMIAL\_HEAP()

3. Reverse the order of the linked list of x’s children and set head[H’]<-head of resulting list.

4. H<-BINOMIAL\_HEAP\_UNION(H,H’)

Return ‘x’

End

BINOMIAL\_HEAP\_DECREASE\_KEY(H,x,k)

1. Key[x]<-k

2. y<-x

3. z<-p[y]

4. while z ≠ NIL and key[x] < key[y] do exchange key[y]<-key[z]

5. exchange satellite fields of y and z

6. y<-z

7. z<-p[y]

8. end while

9. End

BINOMIAL\_HEAP\_DELETE(H,x)

1. Binomial Heap Decrease Key(H,x,-∞)

2. Binomial Heap Extract MIN(H)

3. End.

**PROGRAM CODE:**

#include<stdio.h>

#include<malloc.h>

struct node {

int n;

int degree;

struct node\* parent;

struct node\* child;

struct node\* sibling;

};

struct node\* MAKE\_bin\_HEAP();

int bin\_LINK(struct node\*, struct node\*);

struct node\* CREATE\_NODE(int);

struct node\* bin\_HEAP\_UNION(struct node\*, struct node\*);

struct node\* bin\_HEAP\_INSERT(struct node\*, struct node\*);

struct node\* bin\_HEAP\_MERGE(struct node\*, struct node\*);

struct node\* bin\_HEAP\_EXTRACT\_MIN(struct node\*);

int REVERT\_LIST(struct node\*);

int DISPLAY(struct node\*);

struct node\* FIND\_NODE(struct node\*, int);

int bin\_HEAP\_DECREASE\_KEY(struct node\*, int, int);

int bin\_HEAP\_DELETE(struct node\*, int);

int count = 1;

struct node\* MAKE\_bin\_HEAP() {

struct node\* np;

np = NULL;

return np;

}

struct node \* H = NULL;

struct node \*Hr = NULL;

int bin\_LINK(struct node\* y, struct node\* z) {

y->parent = z;

y->sibling = z->child;

z->child = y;

z->degree = z->degree + 1;

}

struct node\* CREATE\_NODE(int k) {

struct node\* p;//new node;

p = (struct node\*) malloc(sizeof(struct node));

p->n = k;

return p;

}

struct node\* bin\_HEAP\_UNION(struct node\* H1, struct node\* H2) {

struct node\* prev\_x;

struct node\* next\_x;

struct node\* x;

struct node\* H = MAKE\_bin\_HEAP();

H = bin\_HEAP\_MERGE(H1, H2);

if (H == NULL)

return H;

prev\_x = NULL;

x = H;

next\_x = x->sibling;

while (next\_x != NULL) {

if ((x->degree != next\_x->degree) || ((next\_x->sibling != NULL)

&& (next\_x->sibling)->degree == x->degree)) {

prev\_x = x;

x = next\_x;

} else {

if (x->n <= next\_x->n) {

x->sibling = next\_x->sibling;

bin\_LINK(next\_x, x);

} else {

if (prev\_x == NULL)

H = next\_x;

else

prev\_x->sibling = next\_x;

bin\_LINK(x, next\_x);

x = next\_x;

}

}

next\_x = x->sibling;

}

return H;

}

struct node\* bin\_HEAP\_INSERT(struct node\* H, struct node\* x) {

struct node\* H1 = MAKE\_bin\_HEAP();

x->parent = NULL;

x->child = NULL;

x->sibling = NULL;

x->degree = 0;

H1 = x;

H = bin\_HEAP\_UNION(H, H1);

return H;

}

struct node\* bin\_HEAP\_MERGE(struct node\* H1, struct node\* H2) {

struct node\* H = MAKE\_bin\_HEAP();

struct node\* y;

struct node\* z;

struct node\* a;

struct node\* b;

y = H1;

z = H2;

if (y != NULL) {

if (z != NULL && y->degree <= z->degree)

H = y;

else if (z != NULL && y->degree > z->degree)

/\* need some modifications here;the first and the else conditions can be merged together!!!! \*/

H = z;

else

H = y;

} else

H = z;

while (y != NULL && z != NULL) {

if (y->degree < z->degree) {

y = y->sibling;

} else if (y->degree == z->degree) {

a = y->sibling;

y->sibling = z;

y = a;

} else {

b = z->sibling;

z->sibling = y;

z = b;

}

}

return H;

}

int DISPLAY(struct node\* H) {

struct node\* p;

if (H == NULL) {

printf("\nHEAP EMPTY");

return 0;

}

printf("\nTHE ROOT NODES ARE:-\n");

p = H;

while (p != NULL) {

printf("%d", p->n);

if (p->sibling != NULL)

printf("-->");

p = p->sibling;

}

printf("\n");

}

struct node\* bin\_HEAP\_EXTRACT\_MIN(struct node\* H1) {

int min;

struct node\* t = NULL;

struct node\* x = H1;

struct node \*Hr;

struct node\* p;

Hr = NULL;

if (x == NULL) {

printf("\nNOTHING TO EXTRACT")

return x;

}

// int min=x->n;

p = x;

while (p->sibling != NULL) {

if ((p->sibling)->n < min) {

min = (p->sibling)->n;

t = p;

x = p->sibling;

}

p = p->sibling;

}

if (t == NULL && x->sibling == NULL

H1 = NULL;

else if (t == NULL)

H1 = x->sibling;

else if (t->sibling == NULL)

t = NULL;

else

t->sibling = x->sibling;

if (x->child != NULL) {

REVERT\_LIST(x->child);

(x->child)->sibling = NULL;

}

H = bin\_HEAP\_UNION(H1, Hr)

return x;

}

int REVERT\_LIST(struct node\* y) {

if (y->sibling != NULL) {

REVERT\_LIST(y->sibling);

(y->sibling)->sibling = y;

} else {

Hr = y;

}

}

struct node\* FIND\_NODE(struct node\* H, int k)

struct node\* x = H;

struct node\* p = NULL;

if (x->n == k) {

p = x;

return p;

}

if (x->child != NULL && p == NULL) {

p = FIND\_NODE(x->child, k);

}

if (x->sibling != NULL && p == NULL) {

p = FIND\_NODE(x->sibling, k);

}

return p;

}

int bin\_HEAP\_DECREASE\_KEY(struct node\* H, int i, int k) {

int temp;

struct node\* p;

struct node\* y;

struct node\* z;

p = FIND\_NODE(H, i);

if (p == NULL) {

printf("\nINVALID CHOICE OF KEY TO BE REDUCED");

return 0;

}

if (k > p->n) {

printf("\nSORY!THE NEW KEY IS GREATER THAN CURRENT ONE");

return 0;

}

p->n = k;

y = p;

z = p->parent;

while (z != NULL && y->n < z->n) {

temp = y->n;

y->n = z->n;

z->n = temp;

y = z;

z = z->parent;

}

printf("\nKEY REDUCED SUCCESSFULLY!");

}

int bin\_HEAP\_DELETE(struct node\* H, int k) {

struct node\* np;

if (H == NULL) {

printf("\nHEAP EMPTY");

return 0;

}

bin\_HEAP\_DECREASE\_KEY(H, k, -1000);

np = bin\_HEAP\_EXTRACT\_MIN(H);

if (np != NULL)

printf("\nNODE DELETED SUCCESSFULLY");

}

int main() {

int i, n, m, l;

struct node\* p;

struct node\* np;

char ch;

printf("\nENTER THE NUMBER OF ELEMENTS:");

scanf("%d", &n);

printf("\nENTER THE ELEMENTS:\n");

for (i = 1; i <= n; i++) {

scanf("%d", &m);

np = CREATE\_NODE(m);

H = bin\_HEAP\_INSERT(H, np);

}

DISPLAY(H);

do {

printf("\nMENU:-\n");

printf("\n1)INSERT AN ELEMENT\n2)EXTRACT THE MINIMUM KEY NODE\n3)DECREASE A NODE KEY\n 4)DELETE A NODE\n5)QUIT\n");

scanf("%d", &l);

switch (l) {

case 1:

do {

printf("\nENTER THE ELEMENT TO BE INSERTED:");

scanf("%d", &m);

p = CREATE\_NODE(m);

H = bin\_HEAP\_INSERT(H, p);

printf("\nNOW THE HEAP IS:\n");

DISPLAY(H);

printf("\nINSERT MORE(y/Y)= \n");

fflush(stdin);

scanf("%c", &ch);

} while (ch == 'Y' || ch == 'y');

break;

case 2:

do {

printf("\nEXTRACTING THE MINIMUM KEY NODE");

p = bin\_HEAP\_EXTRACT\_MIN(H);

if (p != NULL)

printf("\nTHE EXTRACTED NODE IS %d", p->n);

printf("\nNOW THE HEAP IS:\n");

DISPLAY(H);

printf("\nEXTRACT MORE(y/Y)\n");

fflush(stdin);

scanf("%c", &ch);

} while (ch == 'Y' || ch == 'y');

break;

case 3:

do {

printf("\nENTER THE KEY OF THE NODE TO BE DECREASED:");

scanf("%d", &m);

printf("\nENTER THE NEW KEY : ");

scanf("%d", &l);

bin\_HEAP\_DECREASE\_KEY(H, m, l);

printf("\nNOW THE HEAP IS:\n");

DISPLAY(H);

printf("\nDECREASE MORE(y/Y)\n");

fflush(stdin);

scanf("%c", &ch);

} while (ch == 'Y' || ch == 'y');

break;

case 4:

do {

printf("\nENTER THE KEY TO BE DELETED: ");

scanf("%d", &m);

bin\_HEAP\_DELETE(H, m);

printf("\nDELETE MORE(y/Y)\n");

fflush(stdin);

scanf("%c", &ch);

} while (ch == 'y' || ch == 'Y');

break;

case 5:

printf("\nTHANK U SIR\n");

break;

default:

printf("\nINVALID ENTRY...TRY AGAIN....\n");

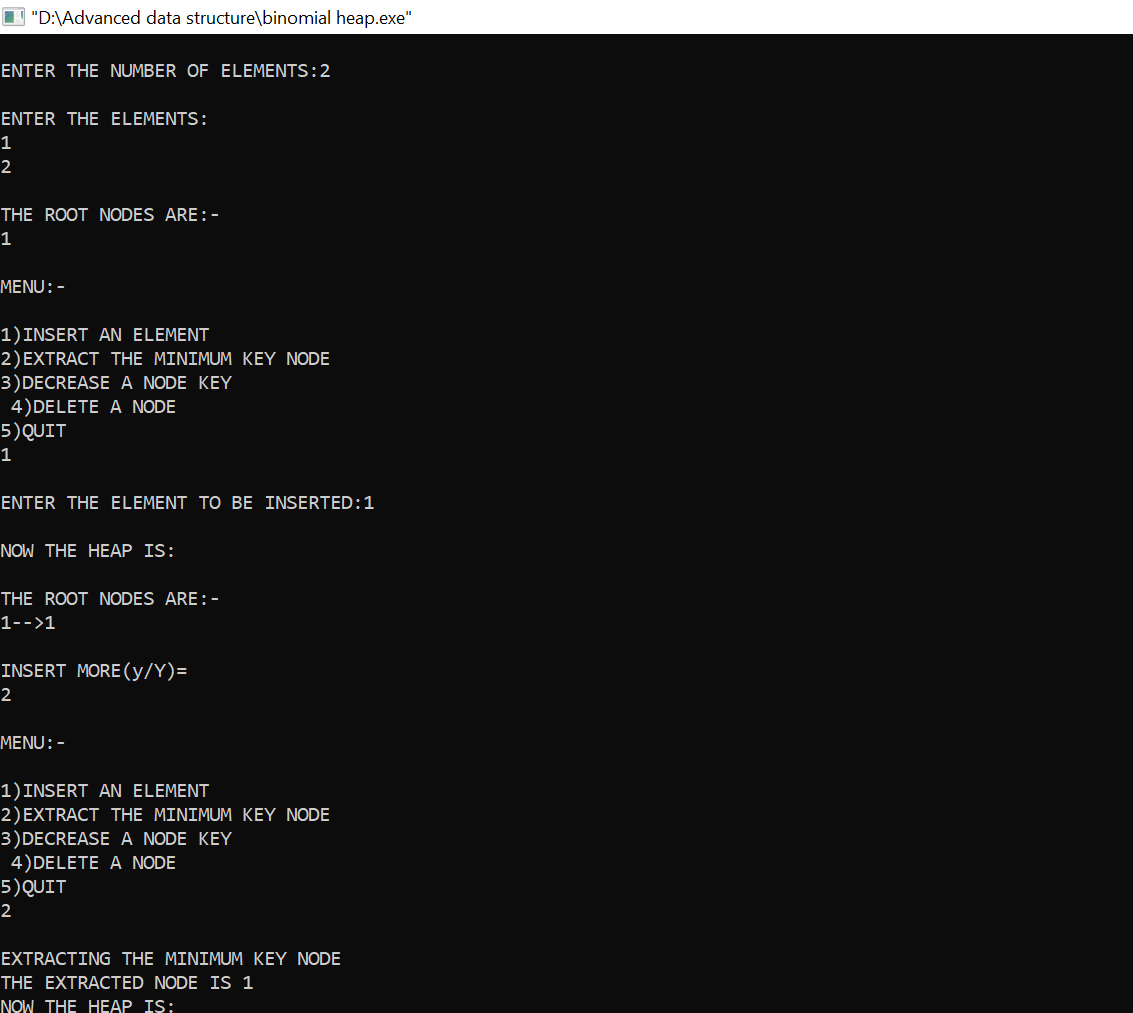
}

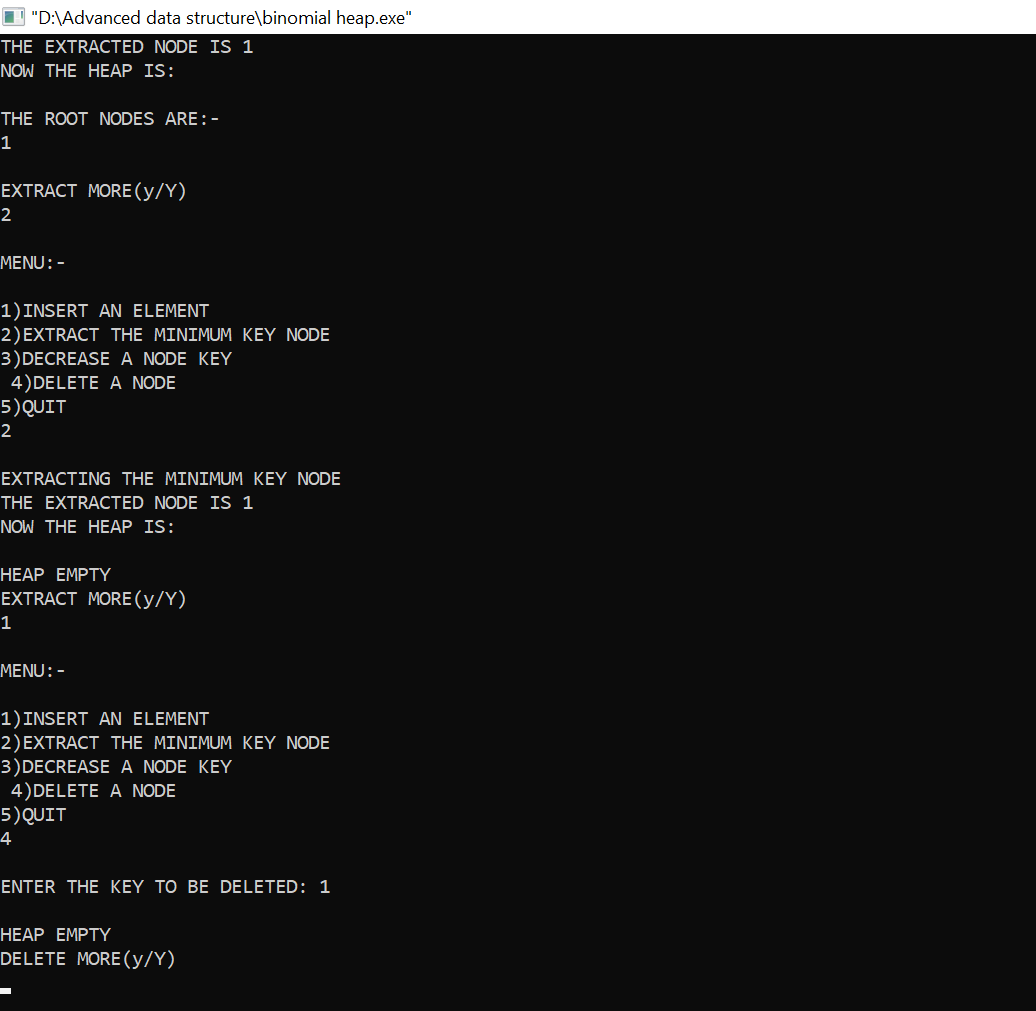
} while (l != 5);

}

**RESULT:** The above program is successfully executed and obtained the output.

**OUTPUT**





**PROGRAM NO-5**

**AIM:**

B Trees and its operations.

**ALGORITHM:**

**(a)INSERTION:**

**BtreeInsertion(T, k)**

r root[T]

if n[r] = 2t – 1

s = AllocateNode()

root[T] = s

leaf[s] = FALSE

n[s] <- 0

c1[s] <- r

BtreeSplitChild(s, 1, r)

BtreeInsertNonFull(s, k)

else BtreeInsertNonFull(r, k)

BtreeInsertNonFull(x, k)

i = n[x]

if leaf[x]

while i ≥ 1 and k < keyi[x]

i=i-1

keyi+1 [x] = k

n[x] = n[x] + 1

else while i ≥ 1 and k < keyi[x]

i = i - 1

i = i + 1

if n[ci[x]] == 2t - 1

BtreeSplitChild(x, i, ci[x])

if k &rt; keyi[x]

i = i + 1

BtreeInsertNonFull(ci[x], k)

BtreeSplitChild(x, i)

BtreeSplitChild(x, i, y)

z = AllocateNode()

leaf[z] = leaf[y]

n[z] = t - 1

for j = 1 to t – 1

keyj[z] = keyj+t[y]

if not leaf [y]

for j = 1 to t

cj[z] = cj + t[y]

n[y] = t - 1

for j = n[x] + 1 to i + 1

cj+1[x] = cj[x]

ci+1[x] = z

for j = n[x] to i

keyj+1[x] = keyj[x]

keyi[x] = keyt[y]

n[x] = n[x] + 1

**(b)DELETION:**

B-Tree-Delete-Key(x, k)

if not leaf[x] then

 y ← Preceding-Child(x)

z ← Successor-Child(x)

 if n[y] > t − 1 then

 k' ← Find-Predecessor-Key(k, x)

 Move-Key(k', y, x)

 Move-Key(k, x, z)

B-Tree-Delete-Key(k, z)

 else if n[z] > t − 1 then

 k' ← Find-Successor-Key(k, x)

Move-Key(k', z, x)

Move-Key(k, x, y)

B-Tree-Delete-Key(k, y)

else

 Move-Key(k, x, y)

Merge-Nodes(y, z)

B-Tree-Delete-Key(k, y)

else (leaf node)

y ← Preceding-Child(x)

 z ← Successor-Child(x)

 w ← root(x)

 v ← RootKey(x)

 if n[x] > t − 1 then Remove-Key(k, x)

 else if n[y] > t − 1 then

 k' ← Find-Predecessor-Key(w, v)

 Move-Key(k', y,w)

 k' ← Find-Successor-Key(w, v)

 Move-Key(k',w, x)

B-Tree-Delete-Key(k, x)

 else if n[w] > t − 1 then

 k' ← Find-Successor-Key(w, v)

 Move-Key(k', z,w)

 k' ← Find-Predecessor-Key(w, v)

 Move-Key(k',w, x)

 B-Tree-Delete-Key(k, x)

 else

 s ← Find-Sibling(w)

 w' ← root(w)

 if n[w'] = t − 1 then

 Merge-Nodes(w',w)

Merge-Nodes(w, s)

 B-Tree-Delete-Key(k, x)

else

Move-Key(v,w, x)

B-Tree-Delete-Key(k, x)

**PROGRAM CODE:**

#include<stdio.h>

#include <stdlib.h>

#define M 5

struct node{

int n; /\* n < M No. of keys in node will always less than order of B

tree \*/

int keys[M-1]; /\*array of keys\*/

struct node \*p[M]; /\* (n+1 pointers will be in use) \*/

}\*root=NULL;

enum KeyStatus { Duplicate,SearchFailure,Success,InsertIt,LessKeys };

void insert(int key);

void display(struct node \*root,int);

void DelNode(int x);

void search(int x);

enum KeyStatus ins(struct node \*r, int x, int\* y, struct node\*\* u);

int searchPos(int x,int \*key\_arr, int n);

enum KeyStatus del(struct node \*r, int x);

int main()

{

int key;

int choice;

printf("Creation of B tree for node %d\n",M);

while(1)

{

printf("1.Insert\n");

printf("2.Delete\n");

printf("3.Search\n");

printf("4.Display\n");

printf("5.Quit\n");

printf("Enter your choice : ");

scanf("%d",&choice);

switch(choice)

{

case 1:

printf("Enter the key : ");

scanf("%d",&key);

insert(key);

break;

case 2:

printf("Enter the key : ");

scanf("%d",&key);

DelNode(key);

break;

case 3:

printf("Enter the key : ");

scanf("%d",&key);

search(key);

break;

case 4:

printf("Btree is :\n");

display(root,0);

break;

case 5:

exit(1);

default:

printf("Wrong choice\n");

break;

}/\*End of switch\*/

}/\*End of while\*/

return 0;

}/\*End of main()\*/

void insert(int key)

{

struct node \*newnode;

int upKey;

enum KeyStatus value;

value = ins(root, key, &upKey, &newnode);

if (value == Duplicate)

printf("Key already available\n");

if (value == InsertIt)

{

struct node \*uproot = root;

root=malloc(sizeof(struct node));

root->n = 1;

root->keys[0] = upKey;

root->p[0] = uproot;

root->p[1] = newnode;

}/\*End of if \*/

}/\*End of insert()\*/

enum KeyStatus ins(struct node \*ptr, int key, int \*upKey,struct node

\*\*newnode)

{

struct node \*newPtr, \*lastPtr;

int pos, i, n,splitPos;

int newKey, lastKey;

enum KeyStatus value;

if (ptr == NULL)

{

\*newnode = NULL;

\*upKey = key;

return InsertIt;

}

n = ptr->n;

pos = searchPos(key, ptr->keys, n);

if (pos < n && key == ptr->keys[pos])

return Duplicate;

value = ins(ptr->p[pos], key, &newKey, &newPtr);

if (value != InsertIt)

return value;

/\*If keys in node is less than M-1 where M is order of B tree\*/

if (n < M - 1)

{

pos = searchPos(newKey, ptr->keys, n);

/\*Shifting the key and pointer right for inserting the new key\*/

for (i=n; i>pos; i--)

{

ptr->keys[i] = ptr->keys[i-1];

ptr->p[i+1] = ptr->p[i];

}

/\*Key is inserted at exact location\*/

ptr->keys[pos] = newKey;

ptr->p[pos+1] = newPtr;

++ptr->n; /\*incrementing the number of keys in node\*/

return Success;

}/\*End of if \*/

/\*If keys in nodes are maximum and position of node to be inserted is

last\*/

if (pos == M - 1)

{

lastKey = newKey;

lastPtr = newPtr;

}

else /\*If keys in node are maximum and position of node to be inserted

is not last\*/

{

lastKey = ptr->keys[M-2];

lastPtr = ptr->p[M-1];

for (i=M-2; i>pos; i--)

{

ptr->keys[i] = ptr->keys[i-1];

ptr->p[i+1] = ptr->p[i];

}

ptr->keys[pos] = newKey;

ptr->p[pos+1] = newPtr;

}

splitPos = (M - 1)/2;

(\*upKey) = ptr->keys[splitPos];

(\*newnode)=malloc(sizeof(struct node));/\*Right node after split\*/

ptr->n = splitPos; /\*No. of keys for left splitted node\*/

(\*newnode)->n = M-1-splitPos;/\*No. of keys for right splitted node\*/

for (i=0; i < (\*newnode)->n; i++)

{

(\*newnode)->p[i] = ptr->p[i + splitPos + 1];

if(i < (\*newnode)->n - 1)

(\*newnode)->keys[i] = ptr->keys[i + splitPos + 1];

else

(\*newnode)->keys[i] = lastKey;

}

(\*newnode)->p[(\*newnode)->n] = lastPtr;

return InsertIt;

}

/\*End of ins()\*/

void display(struct node \*ptr, int blanks)

{

if (ptr)

{

int i;

for(i=1;i<=blanks;i++)

printf(" ");

for (i=0; i < ptr->n; i++)

printf("%d ",ptr->keys[i]);

printf("\n");

for (i=0; i <= ptr->n; i++)

display(ptr->p[i], blanks+10);

}

/\*End of if\*/

}

/\*End of display()\*/

void search(int key)

{

int pos, i, n;

struct node \*ptr = root;

printf("Search path:\n");

while (ptr)

{

n = ptr->n;

for (i=0; i < ptr->n; i++)

printf(" %d",ptr->keys[i]);

printf("\n");

pos = searchPos(key, ptr->keys, n);

if (pos < n && key == ptr->keys[pos])

{

printf("Key %d found in position %d of last dispalyed node\n",key,i);

return;

}

ptr = ptr->p[pos];

}

printf("Key %d is not available\n",key);

}

/\*End of search()\*/

int searchPos(int key, int \*key\_arr, int n)

{

int pos=0;

while (pos < n && key > key\_arr[pos])

pos++;

return pos;

}

/\*End of searchPos()\*/

void DelNode(int key)

{

struct node \*uproot;

enum KeyStatus value;

value = del(root,key);

switch (value)

{

case SearchFailure:

printf("Key %d is not available\n",key);

break;

case LessKeys:

uproot = root;

root = root->p[0];

free(uproot);

break;

}

/\*End of switch\*/

}

/\*End of delnode()\*/

enum KeyStatus del(struct node \*ptr, int key)

{

int pos, i, pivot, n ,min;

int \*key\_arr;

enum KeyStatus value;

struct node \*\*p,\*lptr,\*rptr;

if (ptr == NULL)

return SearchFailure;

/\*Assigns values of node\*/

n=ptr->n;

key\_arr = ptr->keys;

p = ptr->p;

min = (M - 1)/2;

/\*Minimum number of keys\*/

pos = searchPos(key, key\_arr, n);

if (p[0] == NULL)

{

if (pos == n || key < key\_arr[pos])

return SearchFailure;

/\*Shift keys and pointers left\*/

for (i=pos+1; i < n; i++)

{

key\_arr[i-1] = key\_arr[i];

p[i] = p[i+1];

}

return --ptr->n >= (ptr==root ? 1 : min) ? Success : LessKeys;

}

/\*End of if \*/

if (pos < n && key == key\_arr[pos])

{

struct node \*qp = p[pos], \*qp1;

int nkey;

while(1)

{

nkey = qp->n;

qp1 = qp->p[nkey];

if (qp1 == NULL)

break;

qp = qp1;

}

/\*End of while\*/

key\_arr[pos] = qp->keys[nkey-1];

qp->keys[nkey - 1] = key;

}

/\*End of if \*/

value = del(p[pos], key);

if (value != LessKeys)

return value;

if (pos > 0 && p[pos-1]->n > min)

{

pivot = pos - 1;

/\*pivot for left and right node\*/

lptr = p[pivot];

rptr = p[pos];

/\*Assigns values for right node\*/

rptr->p[rptr->n + 1] = rptr->p[rptr->n];

for (i=rptr->n; i>0; i--)

{

rptr->keys[i] = rptr->keys[i-1];

rptr->p[i] = rptr->p[i-1];

}

rptr->n++;

rptr->keys[0] = key\_arr[pivot];

rptr->p[0] = lptr->p[lptr->n];

key\_arr[pivot] = lptr->keys[--lptr->n];

return Success;

}

/\*End of if \*/

if (pos > min)

{

pivot = pos;

/\*pivot for left and right node\*/

lptr = p[pivot];

rptr = p[pivot+1];

/\*Assigns values for left node\*/

lptr->keys[lptr->n] = key\_arr[pivot];

lptr->p[lptr->n + 1] = rptr->p[0];

key\_arr[pivot] = rptr->keys[0];

lptr->n++;

rptr->n--;

for (i=0; i < rptr->n; i++)

{

rptr->keys[i] = rptr->keys[i+1];

rptr->p[i] = rptr->p[i+1];

}

/\*End of for\*/

rptr->p[rptr->n] = rptr->p[rptr->n + 1];

return Success;

}

/\*End of if \*/

if(pos == n)

pivot = pos-1;

else

pivot = pos;

lptr = p[pivot];

rptr = p[pivot+1];

/\*merge right node with left node\*/

lptr->keys[lptr->n] = key\_arr[pivot];

lptr->p[lptr->n + 1] = rptr->p[0];

for (i=0; i < rptr->n; i++)

{

lptr->keys[lptr->n + 1 + i] = rptr->keys[i];

lptr->p[lptr->n + 2 + i] = rptr->p[i+1];

}

lptr->n = lptr->n + rptr->n +1;

free(rptr);

/\*Remove right node\*/

for (i=pos+1; i < n; i++)

{

key\_arr[i-1] = key\_arr[i];

p[i] = p[i+1];

}

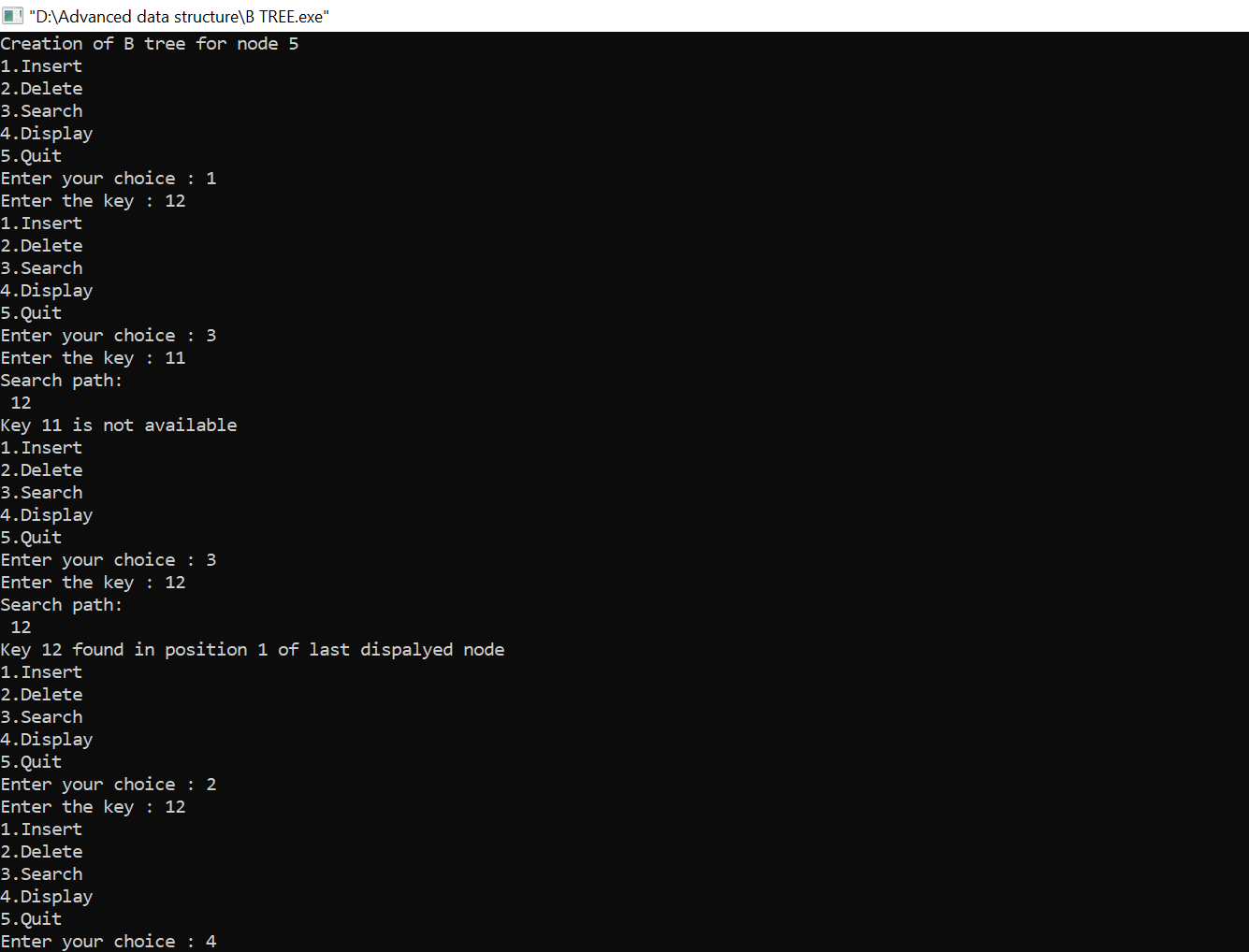
return --ptr->n >= (ptr == root ? 1 : min) ? Success : LessKeys;

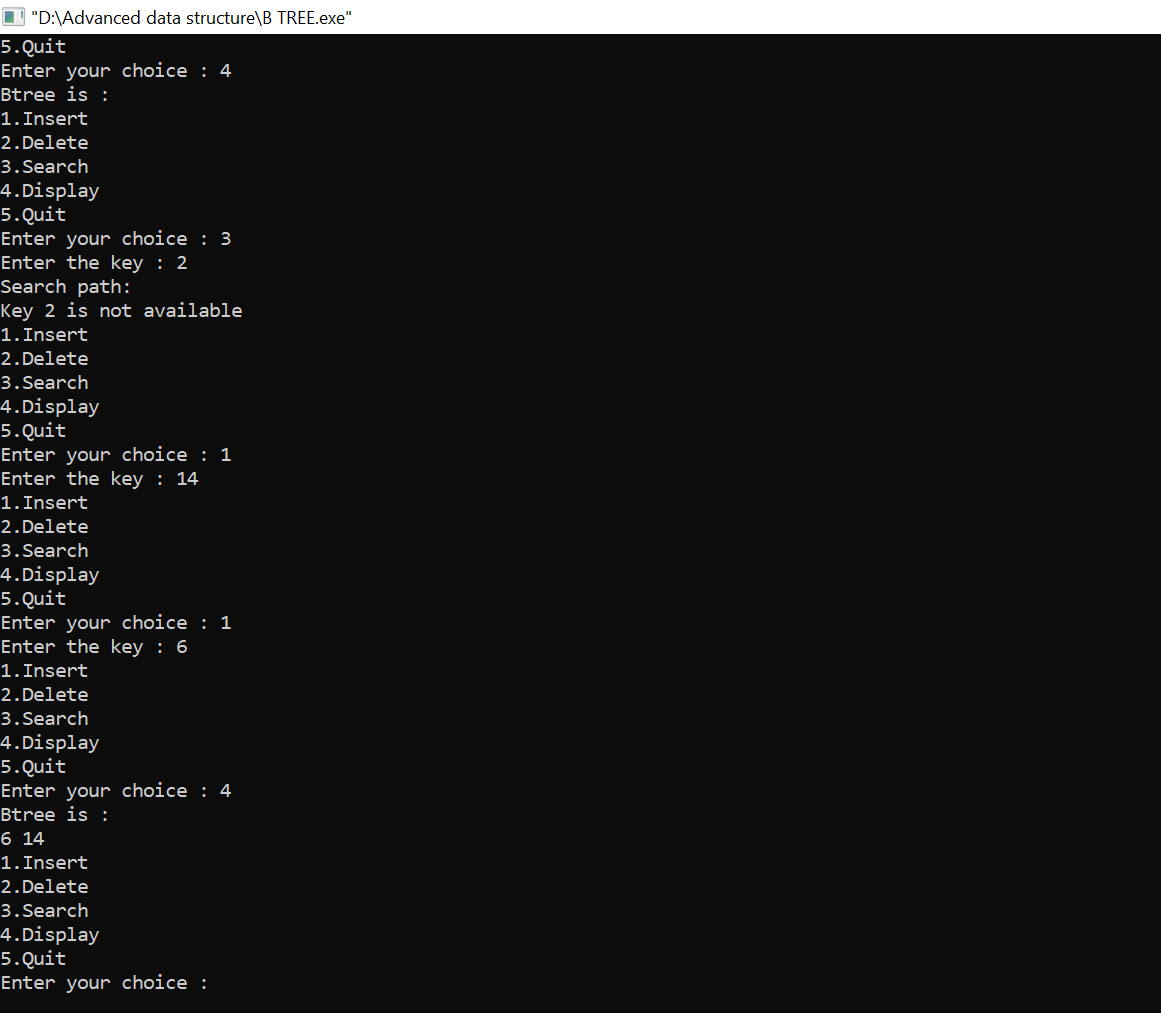
}

/\*End of del()\*/

**RESULT:** The above program is successfully executed and obtained the output.

**OUTPUT**





**PROGRAM NO-6**

**AIM:**

Red Black Trees and its operations.

**ALGORITHM:**

**Algorithm to insert a node**

Following steps are followed for inserting a new element into a red-black tree:

1. Let y be the leaf (ie. NIL) and x be the root of the tree.

2. Check if the tree is empty (ie. whether x is NIL). If yes, insert newNode as a root node and color it black.

3. Else, repeat steps following steps until leaf (NIL) is reached.

a. Compare newKey with rootKey.

b. If newKey is greater than rootKey, traverse through the right subtree.

c. Else traverse through the left subtree.

4. Assign the parent of the leaf as a parent of newNode.

5. If leafKey is greater than newKey, make newNode as rightChild.

6. Else, make newNode as leftChild.

7. Assign NULL to the left and rightChild of newNode.

8. Assign RED color to newNode.

9. Call InsertFix-algorithm to maintain the property of red-black tree if violated.

**Algorithm to maintain red-black property after insertion**

This algorithm is used for maintaining the property of a red-black tree if the insertion of a newNode violates this property.

1. Do the following while the parent of newNode p is RED.

2. If p is the left child of grandParent gP of z, do the following.

**Case-I:**

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

b. Assign gP to newNode.

**Case-II:**

c. Else if newNode is the right child of p then, assign p to newNode.

d. Left-Rotate newNode.

**Case-III:**

e. Set color of p as BLACK and color of gP as RED.

f. Right-Rotate gP.

3. Else, do the following.

a. If the color of the left child of gP of z is RED, set the color of both the children of gP as BLACK and the color of gP as RED.

b. Assign gP to newNode.

c. Else if newNode is the left child of p then, assign p to newNode and RightRotate newNode.

d. Set color of p as BLACK and color of gP as RED.

e. Left-Rotate gP.

4. Set the root of the tree as BLACK.

**Deleting an element from a Red-Black Tree**

**Algorithm to delete a node**

1. Save the color of nodeToBeDeleted in origrinalColor.

2. If the left child of nodeToBeDeleted is NULL

a. Assign the right child of nodeToBeDeleted to x.

b. Transplant nodeToBeDeleted with x.

3. Else if the right child of nodeToBeDeleted is NULL

a. Assign the left child of nodeToBeDeleted into x.

b. Transplant nodeToBeDeleted with x.

4. Else

a. Assign the minimum of right subtree of noteToBeDeleted into y.

b. Save the color of y in originalColor.

c. Assign the rightChild of y into x.

d. If y is a child of nodeToBeDeleted, then set the parent of x as y.

e. Else, transplant y with rightChild of y.

f. Transplant nodeToBeDeleted with y.

g. Set the color of y with originalColor.

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

**Algorithm to maintain Red-Black property after deletion**

1. It reaches the root node.

2. If x points to a red-black node. In this case, x is colored black.

3. Suitable rotations and recoloring are performed.

The following algorithm retains the properties of a red-black tree.

1. Do the following until the x is not the root of the tree and the color of x is BLACK

2. If x is the left child of its parent then,

a. Assign w to the sibling of x.

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

**Case-I:**

a. Set the color of the right child of the parent of x as BLACK.

b. Set the color of the parent of x as RED.

c. Left-Rotate the parent of x.

d. Assign the rightChild of the parent of x to w.

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

**Case-II:**

a. Set the color of w as RED

b. Assign the parent of x to x.

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

**Case-III:**

a. Set the color of the leftChild of w as BLACK

b. Set the color of w as RED

c. Right-Rotate w.

d. Assign the rightChild of the parent of x to w.

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

**Case-IV:**

a. Set the color of w as the color of the parent of x.

b. Set the color of the parent of x as BLACK.

c. Set the color of the right child of w as BLACK.

d. Left-Rotate the parent of x.

e. Set x as the root of the tree.

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

4. Set the color of x as BLACK.

**PROGRAM CODE:**

**(a)INSERTION:**

#include <stdio.h>

#include <stdlib.h>

enum COLOR {Red, Black};

typedef struct tree\_node {

int data;

struct tree\_node \*right;

struct tree\_node \*left;

struct tree\_node \*parent;

enum COLOR color;

}tree\_node;

typedef struct red\_black\_tree {

tree\_node \*root;

tree\_node \*NIL;

}red\_black\_tree;

tree\_node\* new\_tree\_node(int data) {

tree\_node\* n = malloc(sizeof(tree\_node));

n->left = NULL;

n->right = NULL;

n->parent = NULL;

n->data = data;

n->color = Red;

return n;

}

red\_black\_tree\* new\_red\_black\_tree() {

red\_black\_tree \*t = malloc(sizeof(red\_black\_tree));

tree\_node \*nil\_node = malloc(sizeof(tree\_node));

nil\_node->left = NULL;

nil\_node->right = NULL;

nil\_node->parent = NULL;

nil\_node->color = Black;

nil\_node->data = 0;

t->NIL = nil\_node;

t->root = t->NIL;

return t;

}

void left\_rotate(red\_black\_tree \*t, tree\_node \*x) {

tree\_node \*y = x->right;

x->right = y->left;

if(y->left != t->NIL) {

y->left->parent = x;

}

y->parent = x->parent;

if(x->parent == t->NIL) { //x is root

t->root = y;

}

else if(x == x->parent->left) { //x is left child

x->parent->left = y;

}

else { //x is right child

x->parent->right = y;

}

y->left = x;

x->parent = y;

}

void right\_rotate(red\_black\_tree \*t, tree\_node \*x) {

tree\_node \*y = x->left;

x->left = y->right;

if(y->right != t->NIL) {

y->right->parent = x;

}

y->parent = x->parent;

if(x->parent == t->NIL) { //x is root

t->root = y;

}

else if(x == x->parent->right) { //x is left child

x->parent->right = y;

}

else { //x is right child

x->parent->left = y;

}

y->right = x;

x->parent = y;

}

void insertion\_fixup(red\_black\_tree \*t, tree\_node \*z) {

while(z->parent->color == Red) {

if(z->parent == z->parent->parent->left) { //z.parent is the left child

tree\_node \*y = z->parent->parent->right; //uncle of z

if(y->color == Red) { //case 1

z->parent->color = Black;

y->color = Black;

z->parent->parent->color = Red;

z = z->parent->parent;

}

else { //case2 or case3

if(z == z->parent->right) { //case2

z = z->parent; //marked z.parent as new z

left\_rotate(t, z);

}

//case3

z->parent->color = Black; //made parent black

z->parent->parent->color = Red; //made parent red

right\_rotate(t, z->parent->parent);

}

}

else { //z.parent is the right child

tree\_node \*y = z->parent->parent->left; //uncle of z

if(y->color == Red) {

z->parent->color = Black;

y->color = Black;

z->parent->parent->color = Red;

z = z->parent->parent;

}

else {

if(z == z->parent->left) {

z = z->parent; //marked z.parent as new z

right\_rotate(t, z);

}

z->parent->color = Black; //made parent black

z->parent->parent->color = Red; //made parent red

left\_rotate(t, z->parent->parent);

}

}

}

t->root->color = Black;

}

void insert(red\_black\_tree \*t, tree\_node \*z) {

tree\_node\* y = t->NIL; //variable for the parent of the added node

tree\_node\* temp = t->root;

while(temp != t->NIL) {

y = temp;

if(z->data < temp->data)

temp = temp->left;

else

temp = temp->right;

}

z->parent = y;

if(y == t->NIL) { //newly added node is root

t->root = z;

}

else if(z->data < y->data) //data of child is less than its parent, left child

y->left = z;

else

y->right = z;

z->right = t->NIL;

z->left = t->NIL;

insertion\_fixup(t, z);

}

void inorder(red\_black\_tree \*t, tree\_node \*n) {

if(n != t->NIL) {

inorder(t, n->left);

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

inorder(t, n->right);

}

}

int main() {

red\_black\_tree \*t = new\_red\_black\_tree();

tree\_node \*a, \*b, \*c, \*d, \*e, \*f, \*g, \*h, \*i, \*j, \*k, \*l, \*m;

a = new\_tree\_node(10);

b = new\_tree\_node(20);

c = new\_tree\_node(30);

d = new\_tree\_node(100);

e = new\_tree\_node(90);

f = new\_tree\_node(40);

g = new\_tree\_node(50);

h = new\_tree\_node(60);

i = new\_tree\_node(70);

j = new\_tree\_node(80);

k = new\_tree\_node(150);

l = new\_tree\_node(110);

m = new\_tree\_node(120);

insert(t, a);

insert(t, b);

insert(t, c);

insert(t, d);

insert(t, e);

insert(t, f);

insert(t, g);

insert(t, h);

insert(t, i);

insert(t, j);

insert(t, k);

insert(t, l);

insert(t, m);

inorder(t, t->root);

return 0;

}

**(b)DELETION:**

#include <stdio.h>

#include <stdlib.h>

enum COLOR {Red, Black};

typedef struct tree\_node {

int data;

struct tree\_node \*right;

struct tree\_node \*left;

struct tree\_node \*parent;

enum COLOR color;

}tree\_node;

typedef struct red\_black\_tree {

tree\_node \*root;

tree\_node \*NIL;

}red\_black\_tree;

tree\_node\* new\_tree\_node(int data) {

tree\_node\* n = malloc(sizeof(tree\_node));

n->left = NULL;

n->right = NULL;

n->parent = NULL;

n->data = data;

n->color = Red;

return n;

}

red\_black\_tree\* new\_red\_black\_tree() {

red\_black\_tree \*t = malloc(sizeof(red\_black\_tree));

tree\_node \*nil\_node = malloc(sizeof(tree\_node));

nil\_node->left = NULL;

nil\_node->right = NULL;

nil\_node->parent = NULL;

nil\_node->color = Black;

nil\_node->data = 0;

t->NIL = nil\_node;

t->root = t->NIL;

return t;

}

void left\_rotate(red\_black\_tree \*t, tree\_node \*x) {

tree\_node \*y = x->right;

x->right = y->left;

if(y->left != t->NIL) {

y->left->parent = x;

}

y->parent = x->parent;

if(x->parent == t->NIL) { //x is root

t->root = y;

}

else if(x == x->parent->left) { //x is left child

x->parent->left = y;

}

else { //x is right child

x->parent->right = y;

}

y->left = x;

x->parent = y;

}

void right\_rotate(red\_black\_tree \*t, tree\_node \*x) {

tree\_node \*y = x->left;

x->left = y->right;

if(y->right != t->NIL) {

y->right->parent = x;

}

y->parent = x->parent;

if(x->parent == t->NIL) { //x is root

t->root = y;

}

else if(x == x->parent->right) { //x is left child

x->parent->right = y;

}

else { //x is right child

x->parent->left = y;

}

y->right = x;

x->parent = y;

}

void insertion\_fixup(red\_black\_tree \*t, tree\_node \*z) {

while(z->parent->color == Red) {

if(z->parent == z->parent->parent->left) { //z.parent is the left child

tree\_node \*y = z->parent->parent->right; //uncle of z

if(y->color == Red) { //case 1

z->parent->color = Black;

y->color = Black;

z->parent->parent->color = Red;

z = z->parent->parent;

}

else { //case2 or case3

if(z == z->parent->right) { //case2

z = z->parent; //marked z.parent as new z

left\_rotate(t, z);

}

//case3

z->parent->color = Black; //made parent black

z->parent->parent->color = Red; //made parent red

right\_rotate(t, z->parent->parent);

}

}

else { //z.parent is the right child

tree\_node \*y = z->parent->parent->left; //uncle of z

if(y->color == Red) {

z->parent->color = Black;

y->color = Black;

z->parent->parent->color = Red;

z = z->parent->parent;

}

else {

if(z == z->parent->left) {

z = z->parent; //marked z.parent as new z

right\_rotate(t, z);

}

z->parent->color = Black; //made parent black

z->parent->parent->color = Red; //made parent red

left\_rotate(t, z->parent->parent);

}

}

}

t->root->color = Black;

}

void insert(red\_black\_tree \*t, tree\_node \*z) {

tree\_node\* y = t->NIL; //variable for the parent of the added node

tree\_node\* temp = t->root;

while(temp != t->NIL) {

y = temp;

if(z->data < temp->data)

temp = temp->left;

else

temp = temp->right;

}

z->parent = y;

if(y == t->NIL) { //newly added node is root

t->root = z;

}

else if(z->data < y->data) //data of child is less than its parent, left child

y->left = z;

else

y->right = z;

z->right = t->NIL;

z->left = t->NIL;

insertion\_fixup(t, z);

}

void rb\_transplant(red\_black\_tree \*t, tree\_node \*u, tree\_node \*v) {

if(u->parent == t->NIL)

t->root = v;

else if(u == u->parent->left)

u->parent->left = v;

else

u->parent->right = v;

v->parent = u->parent;

}

tree\_node\* minimum(red\_black\_tree \*t, tree\_node \*x) {

while(x->left != t->NIL)

x = x->left;

return x;

}

void rb\_delete\_fixup(red\_black\_tree \*t, tree\_node \*x) {

while(x != t->root && x->color == Black) {

if(x == x->parent->left) {

tree\_node \*w = x->parent->right;

if(w->color == Red) {

w->color = Black;

x->parent->color = Red;

left\_rotate(t, x->parent);

w = x->parent->right;

}

if(w->left->color == Black && w->right->color == Black) {

w->color = Red;

x = x->parent;

}

else {

if(w->right->color == Black) {

w->left->color = Black;

w->color = Red;

right\_rotate(t, w);

w = x->parent->right;

}

w->color = x->parent->color;

x->parent->color = Black;

w->right->color = Black;

left\_rotate(t, x->parent);

x = t->root;

}

}

else {

tree\_node \*w = x->parent->left;

if(w->color == Red) {

w->color = Black;

x->parent->color = Red;

right\_rotate(t, x->parent);

w = x->parent->left;

}

if(w->right->color == Black && w->left->color == Black) {

w->color = Red;

x = x->parent;

}

else {

if(w->left->color == Black) {

w->right->color = Black;

w->color = Red;

left\_rotate(t, w);

w = x->parent->left;

}

w->color = x->parent->color;

x->parent->color = Black;

w->left->color = Black;

right\_rotate(t, x->parent);

x = t->root;

}

}

}

x->color = Black;

}

void rb\_delete(red\_black\_tree \*t, tree\_node \*z) {

tree\_node \*y = z;

tree\_node \*x;

enum COLOR y\_orignal\_color = y->color;

if(z->left == t->NIL) {

x = z->right;

rb\_transplant(t, z, z->right);

}

else if(z->right == t->NIL) {

x = z->left;

rb\_transplant(t, z, z->left);

}

else {

y = minimum(t, z->right);

y\_orignal\_color = y->color;

x = y->right;

if(y->parent == z) {

x->parent = z;

}

else {

rb\_transplant(t, y, y->right);

y->right = z->right;

y->right->parent = y;

}

rb\_transplant(t, z, y);

y->left = z->left;

y->left->parent = y;

y->color = z->color;

}

if(y\_orignal\_color == Black)

rb\_delete\_fixup(t, x);

}

void inorder(red\_black\_tree \*t, tree\_node \*n) {

if(n != t->NIL) {

inorder(t, n->left);

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

inorder(t, n->right);

}

}

int main() {

red\_black\_tree \*t = new\_red\_black\_tree();

tree\_node \*a, \*b, \*c, \*d, \*e, \*f, \*g, \*h, \*i, \*j, \*k, \*l, \*m;

a = new\_tree\_node(10);

b = new\_tree\_node(20);

c = new\_tree\_node(30);

d = new\_tree\_node(100);

e = new\_tree\_node(90);

f = new\_tree\_node(40);

g = new\_tree\_node(50);

h = new\_tree\_node(60);

i = new\_tree\_node(70);

j = new\_tree\_node(80);

k = new\_tree\_node(150);

l = new\_tree\_node(110);

m = new\_tree\_node(120);

insert(t, a);

insert(t, b);

insert(t, c);

insert(t, d);

insert(t, e);

insert(t, f);

insert(t, g);

insert(t, h);

insert(t, i);

insert(t, j);

insert(t, k);

insert(t, l);

insert(t, m);

rb\_delete(t, a);

rb\_delete(t, m);

inorder(t, t->root);

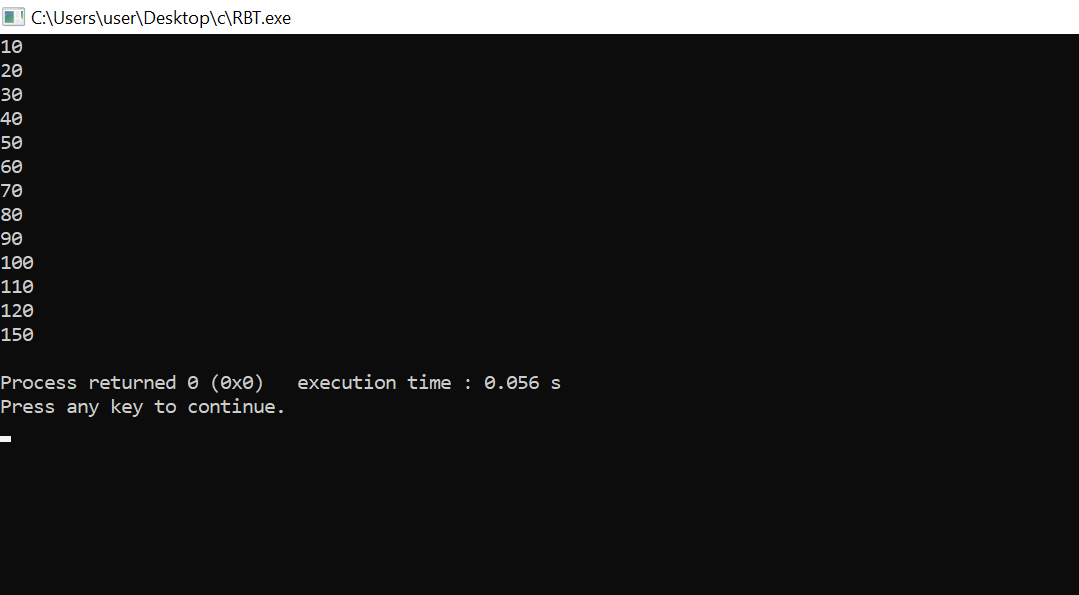
return 0;

}

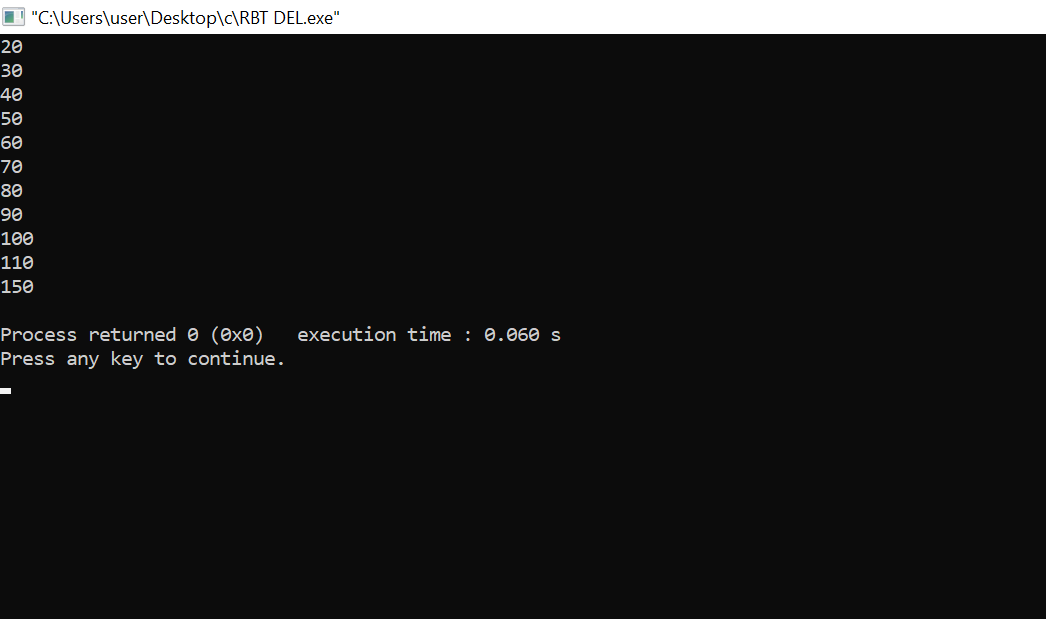
**RESULT:** The above program is successfully executed and obtained the output.

**OUTPUT**

**INSERTION**



**DELETION**



**PROGRAM NO-7**

**AIM:**

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

**ALGORITHM:**

**//DFS**

DFS(G, u)

u.visited = true

for each v ∈ G.Adj[u]

if v.visited == false

DFS(G,v)

init() {

For each u ∈ G

u.visited = false

For each u ∈ G

DFS(G, u)

}

**//BFS**

create a queue Q

mark v as visited and put v into Q

while Q is non-empty

remove the head u of Q

mark and enqueue all (unvisited) neighbours of u

**//Topological sorting**

1. For each vertex UϵV
2. do indegree [U]<-0
3. for each vertex UϵV
4. do for each VϵAdj[U]
5. do indegree[V]<-indegree[V]+1
6. Q<-Ø
7. For each vertex UϵV
8. Do if indegree [U]=0
9. Then EnQueue (Q,U)
10. While Q≠Ø
11. Do U<-DEQUEUE(Q)
12. Output U
13. For each VϵAdj [U]
14. Do indegree [V]<- indegree[V]-1
15. If indegree[V] = 0
16. Then EnQueue(Q,V)
17. Do if indegree[U]≠0
18. Repeat there is a cycle.

**PROGRAM CODE:**

**//DFS**

#include<stdio.h>

int a[20][20],reach[20],n;

int dfs(int v)

{

int i;

reach[v]=1;

for (i=1;i<=n;i++)

if(a[v][i] && !reach[i])

{

printf("\n %d->%d",v,i);

dfs(i);

}

}

int main()

{

int i,j,count=0;

printf("\n Enter number of vertices:");

scanf("%d",&n);

for (i=1;i<=n;i++)

{

reach[i]=0;

for (j=1;j<=n;j++)

a[i][j]=0;

}

printf("\n Enter the adjacency matrix:\n");

for (i=1;i<=n;i++)

for (j=1;j<=n;j++)

scanf("%d",&a[i][j]);

dfs(1);

printf("\n");

for (i=1;i<=n;i++)

{

if(reach[i])

count++;

}

if(count==n)

printf("\n Graph is connected"); else

printf("\n Graph is not connected");

return 0;

}

**//BFS**

#include<stdio.h>

int a[20][20], q[20], visited[20], n, i, j, f = 0, r = -1;

void bfs(int v)

{

for(i = 1; i <= n; i++)

if(a[v][i] && !visited[i])

q[++r] = i;

if(f <= r) {

visited[q[f]] = 1;

bfs(q[f++]);

}

}

int main()

{

int v;

printf("\n Enter the number of vertices:");

scanf("%d", &n);

for(i=1; i <= n; i++)

{

q[i] = 0;

visited[i] = 0;

}

printf("\n Enter graph data in matrix form:\n");

for(i=1; i<=n; i++)

{

for(j=1;j<=n;j++)

{

scanf("%d", &a[i][j]);

}

}

printf("\n Enter the starting vertex:");

scanf("%d", &v);

bfs(v);

printf("\n The node which are reachable are:\n");

for(i=1; i <= n; i++)

{

if(visited[i])

printf("%d\t", i);

else

{

printf("\n Bfs is not possible. Not all nodes are reachable");

break;

}

}

}

**//Topological sorting**

#include <stdio.h>

int main(){

int i,j,k,n,a[10][10],indeg[10],flag[10],count=0;

printf("Enter the no of vertices:\n");

scanf("%d",&n);

printf("Enter the adjacency matrix:\n");

for(i=0;i<n;i++){

printf("Enter row %d\n",i+1);

for(j=0;j<n;j++)

scanf("%d",&a[i][j]);

}

for(i=0;i<n;i++){

indeg[i]=0;

flag[i]=0;

}

for(i=0;i<n;i++)

for(j=0;j<n;j++)

indeg[i]=indeg[i]+a[j][i];

printf("\nThe topological order is:");

while(count<n){

for(k=0;k<n;k++){

if((indeg[k]==0) && (flag[k]==0)){

printf("%d ",(k+1));

flag [k]=1;

}

for(i=0;i<n;i++){

if(a[i][k]==1)

indeg[k]--;

}

}

count++;

}

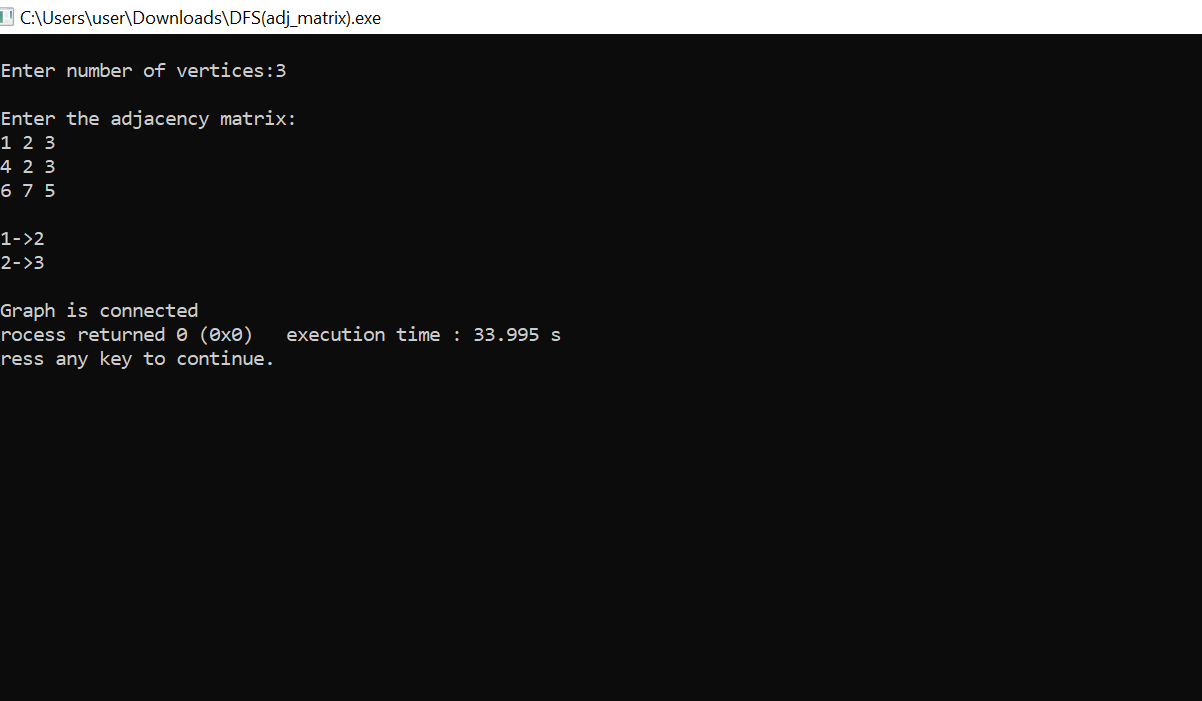
return 0;

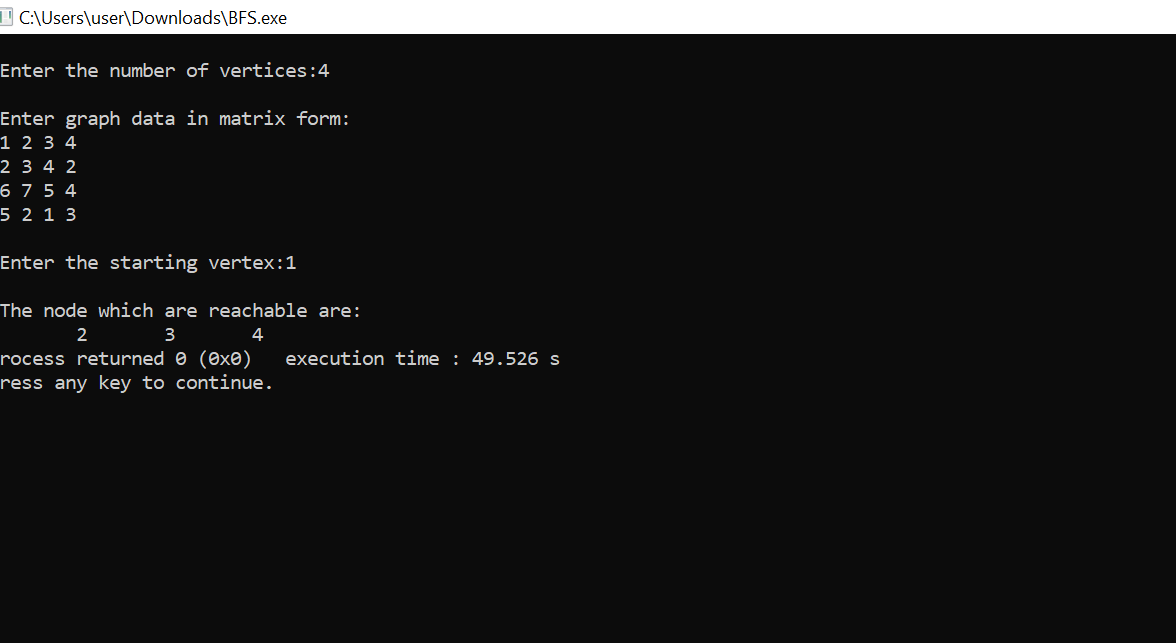
}

**RESULT:** The above program is successfully executed and obtained the output.

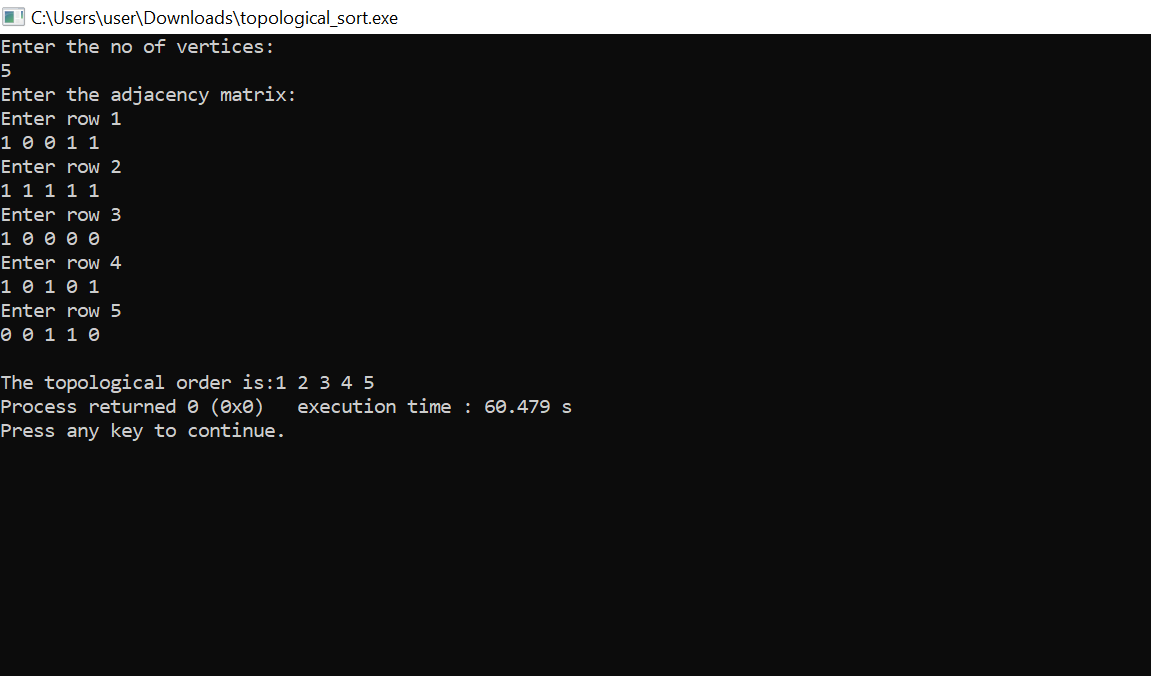
**OUTPUT**

**DFS:**



**//BFS**

**//Topological sorting**



**PROGRAM NO-8**

**AIM:**

Finding the Strongly connected Components in a directed graph.

**ALGORITHM:**

1.Create an empty stack ‘S’

2.Do DFS traversal of a graph. In DFS traversal, after calling recursive DFS for adjacent vertices of a vertex, push the vertex to stack.

3.Reverse directions of all arcs to obtain the transpose graph.

4.One by one pop a vertex from S while S is not empty. Let the popped vertex be ‘V’. Take v as source and do DFS call on ‘V’. The DFS calling from ‘V’ prints strongly connected component of ‘V’.

**PROGRAM CODE:**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_DEGREE 5

#define MAX\_NUM\_VERTICES 20

struct vertices\_s {

int visited;

int deg;

int adj[MAX\_DEGREE]; /\* < 0 if incoming edge \*/

} vertices[] = {

{0, 3, {2, -3, 4}},

{0, 2, {-1, 3}},

{0, 3, {1, -2, 7}},

{0, 3, {-1, -5, 6}},

{0, 2, {4, -7}},

{0, 3, {-4, 7, -8}},

{0, 4, {-3, 5, -6, -12}},

{0, 3, {6, -9, 11}},

{0, 2, {8, -10}},

{0, 3, {9, -11, -12}},

{0, 3, {-8, 10, 12}},

{0, 3, {7, 10, -11}}

};

int num\_vertices = sizeof(vertices) / sizeof(vertices[0]);

struct stack\_s {

int top;

int items[MAX\_NUM\_VERTICES];

} stack = {-1, {}};

void stack\_push(int v) {

stack.top++;

if (stack.top < MAX\_NUM\_VERTICES)

stack.items[stack.top] = v;

else {

printf("Stack is full!\n");

exit(1);

}

}

int stack\_pop() {

return stack.top < 0 ? -1 : stack.items[stack.top--];

}

void dfs(int v, int transpose) {

int i, c, n;

vertices[v].visited = 1;

for (i = 0, c = vertices[v].deg; i < c; ++i) {

n = vertices[v].adj[i] \* transpose;

if (n > 0)

/\* n - 1 because vertex indexing begins at 0 \*/

if (!vertices[n - 1].visited)

dfs(n - 1, transpose);

}

if (transpose < 0)

stack\_push(v);

else

printf("%d ", v + 1);

}

void reset\_visited() {

int i;

for (i = 0; i < num\_vertices; ++i)

vertices[i].visited = 0;

}

void order\_pass() {

int i;

for (i = 0; i < num\_vertices; ++i)

if (!vertices[i].visited)

dfs(i, -1);

}

void scc\_pass() {

int i = 0, v;

while((v = stack\_pop()) != -1) {

if (!vertices[v].visited) {

printf("scc %d: ", ++i);

dfs(v, 1);

printf("\n");

}

}

}

int main(void) {

order\_pass();

reset\_visited();

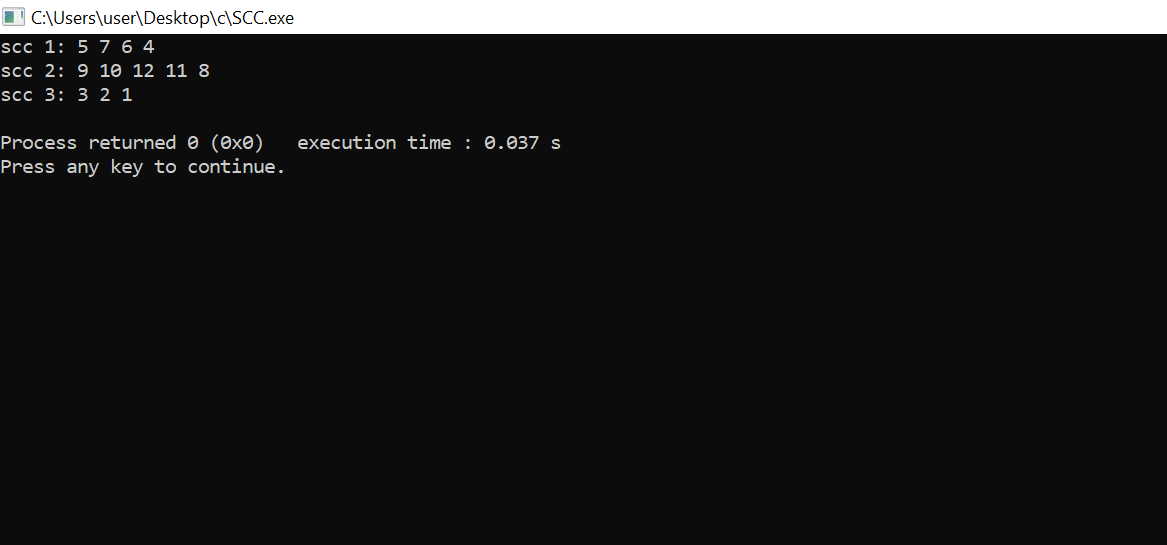
scc\_pass();

return 0;

}

**RESULT:** The program was successfully executed and obtained the result.

**OUTPUT**



**PROGRAM-9**

**AIM:**

Prim’s Algorithm for finding the minimum cost spanning tree

**ALGORITHM:**

**MST PRIMS(G, w,t)**

1. For each uϵV[G]
2. Do key[u]<-∞
3. П[u]<-NIL
4. Key[П]<- 0
5. Q<-V[G]
6. While Q≠Ø

Do u<-extract min(Q)

For each VϵAdj[u]

Do if VϵQ and w[u,v]<key[V]

Then П[V]<-u

Key[V]<-w[u,v]

**PROGRAM CODE:**

#include<stdio.h>

#include<stdbool.h>

#define INF 9999999

// number of vertices in graph

#define V 5

// create a 2d array of size 5x5

//for adjacency matrix to represent graph

int G[V][V] = {

{0, 9, 75, 0, 0},

{9, 0, 95, 19, 42},

{75, 95, 0, 51, 66},

{0, 19, 51, 0, 31},

{0, 42, 66, 31, 0}};

int main() {

int no\_edge; // number of edge

// create a array to track selected vertex

// selected will become true otherwise false

int selected[V];

// set selected false initially

memset(selected, false, sizeof(selected));

// set number of edge to 0

no\_edge = 0;

// the number of egde in minimum spanning tree will be

// always less than (V -1), where V is number of vertices in

//graph

// choose 0th vertex and make it true

selected[0] = true;

int x; // row number

int y; // col number

// print for edge and weight

printf("Edge : Weight\n");

while (no\_edge < V - 1) {

//For every vertex in the set S, find the all adjacent vertices

// , calculate the distance from the vertex selected at step 1.

// if the vertex is already in the set S, discard it otherwise

//choose another vertex nearest to selected vertex at step 1.

int min = INF;

x = 0;

y = 0;

for (int i = 0; i < V; i++) {

if (selected[i]) {

for (int j = 0; j < V; j++) {

if (!selected[j] && G[i][j]) { // not in selected and there is an edge

if (min > G[i][j]) {

min = G[i][j];

x = i;

y = j;

}

}

}

}

}

printf("%d - %d : %d\n", x, y, G[x][y]);

selected[y] = true;

no\_edge++;

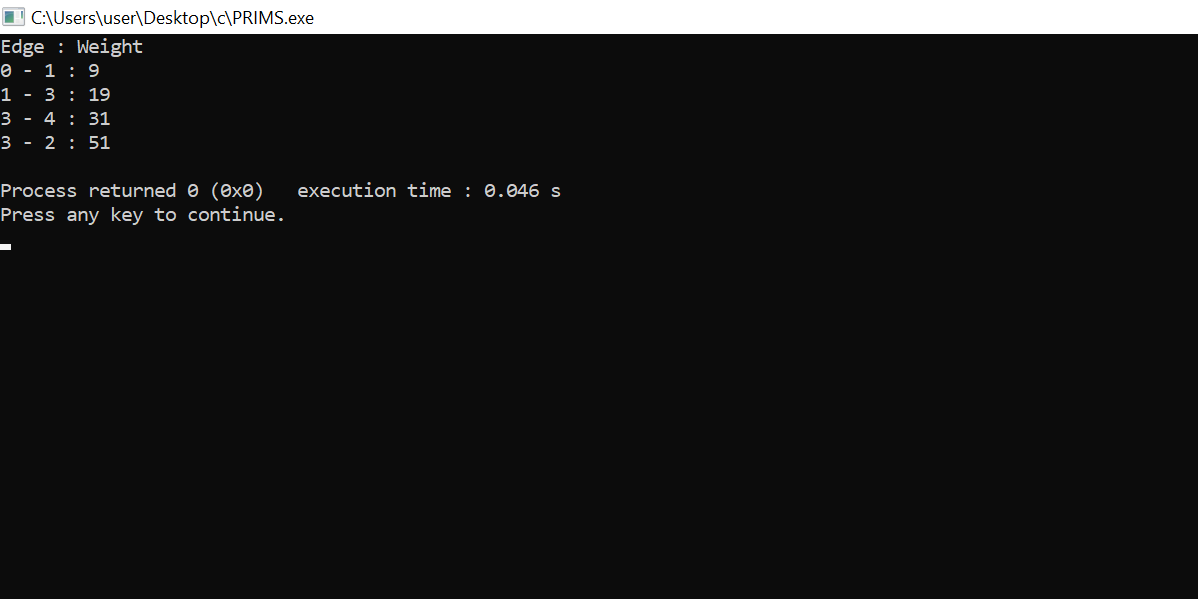
}

return 0;

}

**RESULT:** The program was successfully executed and obtained the result.

**OUTPUT**



**PROGRAM NO-10**

**AIM:**

Kruskal’s Algorithm using the disjoint set data structure.

**ALGORITHM:**

**KRUSKAL(G):**

A = ∅

For each vertex v ∈ G.V:

MAKE-SET(v)

For each edge (u, v) ∈ G.E ordered by increasing order by weight(u, v):

if FIND-SET(u) ≠ FIND-SET(v):

A = A ∪ {(u, v)}

UNION(u, v)

return A

**PROGRAM CODE:**

#include <stdio.h>

#define MAX 30

typedef struct edge {

int u, v, w;

} edge;

typedef struct edge\_list {

edge data[MAX];

int n;

} edge\_list;

edge\_list elist;

int Graph[MAX][MAX], n;

edge\_list spanlist;

void kruskalAlgo();

int find(int belongs[], int vertexno);

void applyUnion(int belongs[], int c1, int c2);

void sort();

void print();

// Applying Krushkal Algo

void kruskalAlgo() {

int belongs[MAX], i, j, cno1, cno2;

elist.n = 0;

for (i = 1; i < n; i++)

for (j = 0; j < i; j++) {

if (Graph[i][j] != 0) {

elist.data[elist.n].u = i;

elist.data[elist.n].v = j;

elist.data[elist.n].w = Graph[i][j];

elist.n++;

}

}

sort();

for (i = 0; i < n; i++)

belongs[i] = i;

spanlist.n = 0;

for (i = 0; i < elist.n; i++) {

cno1 = find(belongs, elist.data[i].u);

cno2 = find(belongs, elist.data[i].v);

if (cno1 != cno2) {

spanlist.data[spanlist.n] = elist.data[i];

spanlist.n = spanlist.n + 1;

applyUnion(belongs, cno1, cno2);

}

}

}

int find(int belongs[], int vertexno) {

return (belongs[vertexno]);

}

void applyUnion(int belongs[], int c1, int c2) {

int i;

for (i = 0; i < n; i++)

if (belongs[i] == c2)

belongs[i] = c1;

}

// Sorting algo

void sort() {

int i, j;

edge temp;

for (i = 1; i < elist.n; i++)

for (j = 0; j < elist.n - 1; j++)

if (elist.data[j].w > elist.data[j + 1].w) {

temp = elist.data[j];

elist.data[j] = elist.data[j + 1];

elist.data[j + 1] = temp;

}

}

// Printing the result

void print() {

int i, cost = 0;

for (i = 0; i < spanlist.n; i++) {

printf("\n%d - %d : %d", spanlist.data[i].u, spanlist.data[i].v, spanlist.data[i].w);

cost = cost + spanlist.data[i].w;

}

printf("\nSpanning tree cost: %d", cost);

}

int main() {

int i, j, total\_cost;

n = 6;

Graph[0][0] = 0;

Graph[0][1] = 4;

Graph[0][2] = 4;

Graph[0][3] = 0;

Graph[0][4] = 0;

Graph[0][5] = 0;

Graph[0][6] = 0;

Graph[1][0] = 4;

Graph[1][1] = 0;

Graph[1][2] = 2;

Graph[1][3] = 0;

Graph[1][4] = 0;

Graph[1][5] = 0;

Graph[1][6] = 0;

Graph[2][0] = 4;

Graph[2][1] = 2;

Graph[2][2] = 0;

Graph[2][3] = 3;

Graph[2][4] = 4;

Graph[2][5] = 0;

Graph[2][6] = 0;

Graph[3][0] = 0;

Graph[3][1] = 0;

Graph[3][2] = 3;

Graph[3][3] = 0;

Graph[3][4] = 3;

Graph[3][5] = 0;

Graph[3][6] = 0;

Graph[4][0] = 0;

Graph[4][1] = 0;

Graph[4][2] = 4;

Graph[4][3] = 3;

Graph[4][4] = 0;

Graph[4][5] = 0;

Graph[4][6] = 0;

Graph[5][0] = 0;

Graph[5][1] = 0;

Graph[5][2] = 2;

Graph[5][3] = 0;

Graph[5][4] = 3;

Graph[5][5] = 0;

Graph[5][6] = 0;

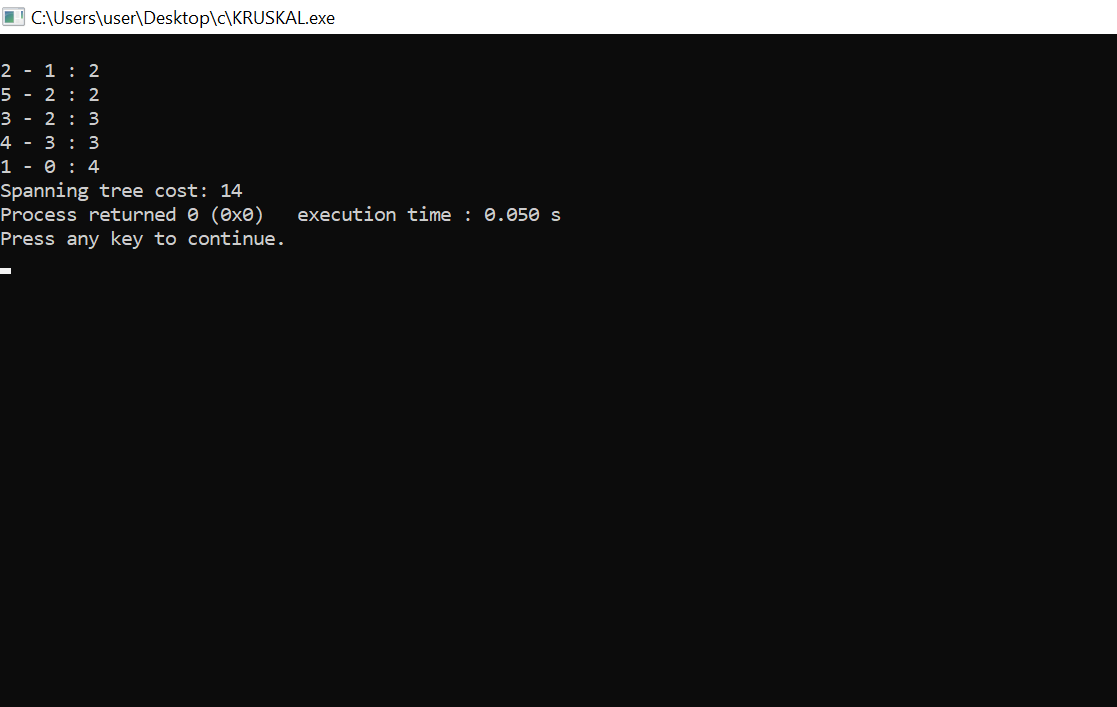
kruskalAlgo();

print();

}

**RESULT:** The above program is successfully executed and obtained the output.

**OUTPUT**



**PROGRAM NO-11**

**AIM:**

Single Source shortest path algorithm using any heap structure that supports mergeable heap operations.

**ALGORITHM:**

1. Initialise single-source(G,s)
2. S<-Ø
3. Q<-V[G]
4. While Q≠Ø

Do u<-extract min(Q)

s<-sՍ{U}

for each vertex VϵAdj[u]

do RELAX(U,V,w)

**Initialise single-source(G,s)**

1. For each vertex VϵV[G]
2. Do d[V]<-∞
3. П[V]<-NIL
4. d[s]<-0

**RELAX(U,V,w)**

1. if d[V]>d[U]+w[u,v]
2. then d[V]<-d[U]+w[u,v]

П[V]<-U

**PROGRAM CODE:**

#include <stdio.h>

#define INFINITY 9999

#define MAX 10

void Dijkstra(int Graph[MAX][MAX], int n, int start);

void Dijkstra(int Graph[MAX][MAX], int n, int start) {

int cost[MAX][MAX], distance[MAX], pred[MAX];

int visited[MAX], count, mindistance, nextnode, i, j;

// Creating cost matrix

for (i = 0; i < n; i++)

for (j = 0; j < n; j++)

if (Graph[i][j] == 0)

cost[i][j] = INFINITY;

else

cost[i][j] = Graph[i][j];

for (i = 0; i < n; i++) {

distance[i] = cost[start][i];

pred[i] = start;

visited[i] = 0;

}

distance[start] = 0;

visited[start] = 1;

count = 1;

while (count < n - 1) {

mindistance = INFINITY;

for (i = 0; i < n; i++)

if (distance[i] < mindistance && !visited[i]) {

mindistance = distance[i];

nextnode = i;

}

visited[nextnode] = 1;

for (i = 0; i < n; i++)

if (!visited[i])

if (mindistance + cost[nextnode][i] < distance[i]) {

distance[i] = mindistance + cost[nextnode][i];

pred[i] = nextnode;

}

count++;

}

// Printing the distance

for (i = 0; i < n; i++)

if (i != start) {

printf("\nDistance from source to %d: %d", i, distance[i]);

}

}

int main() {

int Graph[MAX][MAX], i, j, n, u;

n = 7;

Graph[0][0] = 0;

Graph[0][1] = 0;

Graph[0][2] = 1;

Graph[0][3] = 2;

Graph[0][4] = 0;

Graph[0][5] = 0;

Graph[0][6] = 0;

Graph[1][0] = 0;

Graph[1][1] = 0;

Graph[1][2] = 2;

Graph[1][3] = 0;

Graph[1][4] = 0;

Graph[1][5] = 3;

Graph[1][6] = 0;

Graph[2][0] = 1;

Graph[2][1] = 2;

Graph[2][2] = 0;

Graph[2][3] = 1;

Graph[2][4] = 3;

Graph[2][5] = 0;

Graph[2][6] = 0;

Graph[3][0] = 2;

Graph[3][1] = 0;

Graph[3][2] = 1;

Graph[3][3] = 0;

Graph[3][4] = 0;

Graph[3][5] = 0;

Graph[3][6] = 1;

Graph[4][0] = 0;

Graph[4][1] = 0;

Graph[4][2] = 3;

Graph[4][3] = 0;

Graph[4][4] = 0;

Graph[4][5] = 2;

Graph[4][6] = 0;

Graph[5][0] = 0;

Graph[5][1] = 3;

Graph[5][2] = 0;

Graph[5][3] = 0;

Graph[5][4] = 2;

Graph[5][5] = 0;

Graph[5][6] = 1;

Graph[6][0] = 0;

Graph[6][1] = 0;

Graph[6][2] = 0;

Graph[6][3] = 1;

Graph[6][4] = 0;

Graph[6][5] = 1;

Graph[6][6] = 0;

u = 0;

Dijkstra(Graph, n, u);

return 0;

}

**RESULT:** The above program is successfully executed and obtained the output.

**OUTPUT**

